Oracle® Database

SQL Tuning Guide

21c

F31828-11

August 2023

Oracle Database SQL Tuning Guide, 21c

F31828-11

Primary Author: Lance Ashdown

Contributing Authors: Nigel Bayliss, Maria Colgan, Tom Kyte, Frederick Kush

Contributors: Hermann Baer, Bjorn Bolltoft, Ali Cakmak, Sunil Chakkappen, Immanuel Chan, Deba Chatterjee, Chris Chiappa, Dinesh Das, Kurt Engeleiter, Leonidas Galanis, William Endress, Marcus Fallen, Bruce Golbus, Katsumi Inoue, Mark Jefferys, Shantanu Joshi, Adam Kociubes, Praveen Kumar Tupati Jaganath, Keith Laker, Allison Lee, Sue Lee, Cheng Li, Jiakun Li, David McDermid, Colin McGregor, Ajit Mylavarapu, Kantikiran Pasupuleti, Ted Persky, Palash Sharma, Lei Sheng, Ekrem Soylemez, Hong Su, Murali Thiyagarajah, Randy Urbano, Satya Valluri, Sahil Vazirani, Bharath Venkatakrishnan, Hailing Yu, Yuying Zhang, John Zimmerman

Contents

Preface

This manual explains how to tune Oracle SQL.

Audience

This document is intended for database administrators and application developers who perform the following tasks:

- Generating and interpreting SQL execution plans
- Managing optimizer statistics
- Influencing the optimizer through initialization parameters or SQL hints
- Controlling cursor sharing for SQL statements
- Monitoring SQL execution
- Performing application tracing
- Managing SQL tuning sets
- Using SQL Tuning Advisor or SQL Access Advisor
- Managing SQL profiles
- Managing SQL baselines

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs if you are hearing impaired.

Related Documents

This manual assumes that you are familiar with *Oracle Database Concepts*. The following books are frequently referenced:

- *Oracle Database Data Warehousing Guide*
- *Oracle Database VLDB and Partitioning Guide*

- *Oracle Database SQL Language Reference*
- *Oracle Database Reference*

Many examples in this book use the sample schemas, which are installed by default when you select the Basic Installation option with an Oracle Database. See *Oracle Database Sample Schemas* for information on how these schemas were created and how you can use them.

Conventions

The following text conventions are used in this document:

Convention	Meaning
boldface	Boldface type indicates graphical user interface elements associated with an action, or terms defined in text or the glossary.
italic	Italic type indicates book titles, emphasis, or placeholder variables for which you supply particular values.
`monospace`	Monospace type indicates commands within a paragraph, URLs, code in examples, text that appears on the screen, or text that you enter.

Part I

SQL Performance Fundamentals

SQL tuning is improving SQL statement performance to meet specific, measurable, and achievable goals.

1

Introduction to SQL Tuning

SQL tuning is the attempt to diagnose and repair SQL statements that fail to meet a performance standard.

1.1 Changes in Oracle Database Release 21c for SQL Tuning Guide

The following features are new in this release:

- Session-level controls for automatic indexing

  By setting the `OPTIMIZER_SESSION_TYPE` initialization parameter to `ADHOC` in a session, you can suspend automatic indexing for queries in this session. The automatic indexing process does not identify index candidates, or create and verify indexes. This control may be useful for ad hoc queries or testing new functionality.

  See "Automatic Indexing".
- Controls for enabling or disabling the gathering of real-time statistics.

  When the `OPTIMIZER_REAL_TIME_STATISTICS` initialization parameter is set to `true`, Oracle Database automatically gathers real-time statistics during conventional DML operations. The default setting is `false`, which means real-time statistics are disabled.
- Additional controls for SQL Quarantine

  To enable SQL Quarantine to create configurations automatically after the Resource Manager terminates a query, set the `OPTIMIZER_CAPTURE_SQL_QUARANTINE` initialization parameter to `TRUE` (the default is `FALSE`). To disable the use of existing SQL Quarantine configurations, set `OPTIMIZER_USE_SQL_QUARANTINE` to `FALSE` (the default is `TRUE`).

1.2 About SQL Tuning

SQL tuning is the iterative process of improving SQL statement performance to meet specific, measurable, and achievable goals.

SQL tuning implies fixing problems in deployed applications. In contrast, application design sets the security and performance goals *before* deploying an application.

See Also:

- SQL Performance Methodology
- "Guidelines for Designing Your Application" to learn how to design for SQL performance

1.3 Purpose of SQL Tuning

A SQL statement becomes a problem when it fails to perform according to a predetermined and measurable standard.

After you have identified the problem, a typical tuning session has one of the following goals:

- Reduce user response time, which means decreasing the time between when a user issues a statement and receives a response
- Improve throughput, which means using the least amount of resources necessary to process all rows accessed by a statement

For a response time problem, consider an online book seller application that hangs for three minutes after a customer updates the shopping cart. Contrast with a three-minute parallel query in a data warehouse that consumes all of the database host CPU, preventing other queries from running. In each case, the user response time is three minutes, but the cause of the problem is different, and so is the tuning goal.

1.4 Prerequisites for SQL Tuning

SQL performance tuning requires a foundation of database knowledge.

If you are tuning SQL performance, then this manual assumes that you have the knowledge and skills shown in the following table.

Table 1-1 Required Knowledge

Required Knowledge	Description	To Learn More
Database architecture	Database architecture is not the domain of administrators alone. As a developer, you want to develop applications in the least amount of time against an Oracle database, which requires exploiting the database architecture and features. For example, not understanding Oracle Database concurrency controls and multiversioning read consistency may make an application corrupt the integrity of the data, run slowly, and decrease scalability.	*Oracle Database Concepts* explains the basic relational data structures, transaction management, storage structures, and instance architecture of Oracle Database.
SQL and PL/SQL	Because of the existence of GUI-based tools, it is possible to create applications and administer a database without knowing SQL. However, it is impossible to tune applications or a database without knowing SQL.	*Oracle Database Concepts* includes an introduction to Oracle SQL and PL/SQL. You must also have a working knowledge of *Oracle Database SQL Language Reference*, *Oracle Database PL/SQL Packages and Types Reference*, and *Oracle Database PL/SQL Packages and Types Reference*.

Table 1-1 (Cont.) Required Knowledge

Required Knowledge	Description	To Learn More
SQL tuning tools	The database generates performance statistics, and provides SQL tuning tools that interpret these statistics.	*Oracle Database 2 Day + Performance Tuning Guide* provides an introduction to the principal SQL tuning tools.

1.5 Tasks and Tools for SQL Tuning

After you have identified the goal for a tuning session, for example, reducing user response time from three minutes to less than a second, the problem becomes how to accomplish this goal.

1.5.1 SQL Tuning Tasks

The specifics of a tuning session depend on many factors, including whether you tune proactively or reactively.

In **proactive SQL tuning**, you regularly use SQL Tuning Advisor to determine whether you can make SQL statements perform better. In **reactive SQL tuning**, you correct a SQL-related problem that a user has experienced.

Whether you tune proactively or reactively, a typical SQL tuning session involves all or most of the following tasks:

1. Identifying high-load SQL statements

   Review past execution history to find the statements responsible for a large share of the application workload and system resources.

2. Gathering performance-related data

   The optimizer statistics are crucial to SQL tuning. If these statistics do not exist or are no longer accurate, then the optimizer cannot generate the best plan. Other data relevant to SQL performance include the structure of tables and views that the statement accessed, and definitions of any indexes available to the statement.

3. Determining the causes of the problem

   Typically, causes of SQL performance problems include:

   - Inefficiently designed SQL statements

     If a SQL statement is written so that it performs unnecessary work, then the optimizer cannot do much to improve its performance. Examples of inefficient design include

     – Neglecting to add a join condition, which leads to a Cartesian join

     – Using hints to specify a large table as the driving table in a join

     – Specifying `UNION` instead of `UNION ALL`

     – Making a subquery execute for every row in an outer query

   - Suboptimal execution plans

     The query optimizer (also called the optimizer) is internal software that determines which execution plan is most efficient. Sometimes the optimizer chooses a plan with a suboptimal access path, which is the means by which the database retrieves data from the database. For example, the plan for a query predicate with low selectivity may use a full table scan on a large table instead of an index.

You can compare the execution plan of an optimally performing SQL statement to the plan of the statement when it performs suboptimally. This comparison, along with information such as changes in data volumes, can help identify causes of performance degradation.

- Missing SQL access structures

  Absence of SQL access structures, such as indexes and materialized views, is a typical reason for suboptimal SQL performance. The optimal set of access structures can improve SQL performance by orders of magnitude.

- Stale optimizer statistics

  Statistics gathered by `DBMS_STATS` can become stale when the statistics maintenance operations, either automatic or manual, cannot keep up with the changes to the table data caused by DML. Because stale statistics on a table do not accurately reflect the table data, the optimizer can make decisions based on faulty information and generate suboptimal execution plans.

- Hardware problems

  Suboptimal performance might be connected with memory, I/O, and CPU problems.

4. Defining the scope of the problem

   The scope of the solution must match the scope of the problem. Consider a problem at the database level and a problem at the statement level. For example, the shared pool is too small, which causes cursors to age out quickly, which in turn causes many hard parses. Using an initialization parameter to increase the shared pool size fixes the problem at the database level and improves performance for all sessions. However, if a single SQL statement is not using a helpful index, then changing the optimizer initialization parameters for the entire database could harm overall performance. If a single SQL statement has a problem, then an appropriately scoped solution addresses just this problem with this statement.

5. Implementing corrective actions for suboptimally performing SQL statements

   These actions vary depending on circumstances. For example, you might rewrite a SQL statement to be more efficient, avoiding unnecessary hard parsing by rewriting the statement to use bind variables. You might also use equijoins, remove functions from `WHERE` clauses, and break a complex SQL statement into multiple simple statements.

   In some cases, you improve SQL performance not by rewriting the statement, but by restructuring schema objects. For example, you might index a new access path, or reorder columns in a concatenated index. You might also partition a table, introduce derived values, or even change the database design.

6. Preventing SQL performance regressions

   To ensure optimal SQL performance, verify that execution plans continue to provide optimal performance, and choose better plans if they come available. You can achieve these goals using optimizer statistics, SQL profiles, and SQL plan baselines.

See Also:

- "Shared Pool Check"
- *Oracle Database Concepts* to learn more about the shared pool

1.5.2 SQL Tuning Tools

SQL tuning tools are either automated or manual.

In this context, a tool is automated if the database itself can provide diagnosis, advice, or corrective actions. A manual tool requires you to perform all of these operations.

All tuning tools depend on the basic tools of the dynamic performance views, statistics, and metrics that the database instance collects. The database itself contains the data and metadata required to tune SQL statements.

1.5.2.1 Automated SQL Tuning Tools

Oracle Database provides several advisors relevant for SQL tuning.

Additionally, SQL plan management is a mechanism that can prevent performance regressions and also help you to improve SQL performance.

All of the automated SQL tuning tools can use SQL tuning sets as input. A SQL tuning set (STS) is a database object that includes one or more SQL statements along with their execution statistics and execution context.

See Also:

- "About SQL Tuning Sets"
- *Oracle Database 2 Day + Performance Tuning Guide* to learn more about managing SQL tuning sets

1.5.2.1.1 Automatic Database Diagnostic Monitor (ADDM)

ADDM is self-diagnostic software built into Oracle Database.

ADDM can automatically locate the root causes of performance problems, provide recommendations for correction, and quantify the expected benefits. ADDM also identifies areas where no action is necessary.

ADDM and other advisors use Automatic Workload Repository (AWR), which is an infrastructure that provides services to database components to collect, maintain, and use statistics. ADDM examines and analyzes statistics in AWR to determine possible performance problems, including high-load SQL.

For example, you can configure ADDM to run nightly. In the morning, you can examine the latest ADDM report to see what might have caused a problem and if there is a recommended fix. The report might show that a particular `SELECT` statement consumed a huge amount of CPU, and recommend that you run SQL Tuning Advisor.

See Also:

- *Oracle Database 2 Day + Performance Tuning Guide*
- *Oracle Database Performance Tuning Guide*

1.5.2.1.2 SQL Tuning Advisor

SQL Tuning Advisor is internal diagnostic software that identifies problematic SQL statements and recommends how to improve statement performance.

When run during database maintenance windows as an automated maintenance task, SQL Tuning Advisor is known as Automatic SQL Tuning Advisor.

SQL Tuning Advisor takes one or more SQL statements as an input and invokes the Automatic Tuning Optimizer to perform SQL tuning on the statements. The advisor performs the following types of analysis:

- Checks for missing or stale statistics
- Builds SQL profiles

  A SQL profile is a set of auxiliary information specific to a SQL statement. A SQL profile contains corrections for suboptimal optimizer estimates discovered during Automatic SQL Tuning. This information can improve optimizer estimates for cardinality, which is the number of rows that is estimated to be or actually is returned by an operation in an execution plan, and selectivity. These improved estimates lead the optimizer to select better plans.
- Explores whether a different access path can significantly improve performance
- Identifies SQL statements that lend themselves to suboptimal plans

The output is in the form of advice or recommendations, along with a rationale for each recommendation and its expected benefit. The recommendation relates to a collection of statistics on objects, creation of new indexes, restructuring of the SQL statement, or creation of a SQL profile. You can choose to accept the recommendations to complete the tuning of the SQL statements.

See Also:

- "Analyzing SQL with SQL Tuning Advisor"
- *Oracle Database 2 Day + Performance Tuning Guide*

1.5.2.1.3 SQL Access Advisor

SQL Access Advisor is internal diagnostic software that recommends which materialized views, indexes, and materialized view logs to create, drop, or retain.

SQL Access Advisor takes an actual workload as input, or the advisor can derive a hypothetical workload from the schema. SQL Access Advisor considers the trade-offs between space usage and query performance, and recommends the most cost-effective configuration of new and existing materialized views and indexes. The advisor also makes recommendations about partitioning.

See Also:

- "About SQL Access Advisor"
- *Oracle Database 2 Day + Performance Tuning Guide*
- *Oracle Database Administrator's Guide* to learn more about automated indexing
- *Oracle Database Licensing Information User Manual* for details on whether automated indexing is supported for different editions and services

1.5.2.1.4 Automatic Indexing

Oracle Database can constantly monitor the application workload, creating and managing indexes automatically.

Note:

See *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services.

Creating indexes manually requires deep knowledge of the data model, application, and data distribution. Often DBAs make choices about which indexes to create, and then never revise their choices. As a result, opportunities for improvement are lost, and unnecessary indexes can become a performance liability. Automatic index management solves this problem.

1.5.2.1.4.1 How Automatic Indexing Works

The automatic indexing process runs in the background every 15 minutes and performs the following operations:

1. Automatic index candidates are identified based on the usage of table columns in SQL statements. Ensure that table statistics are up to date. Tables without statistics are not considered for automatic indexing. Tables with stale statistics are not considered for automatic indexing.
2. Index candidates are initially created invisible and unusable. They are not visible to the application workload. Invisible automatic indexes cannot be used by SQL statements in the application workload.

   Automatic indexes can be single-column or multi-column. They are considered for the following:

   - Table columns (including virtual columns)
   - Partitioned and non-partitioned tables
   - Selected expressions (for example, JSON expressions)
3. A sample of workload SQL statements is tested against the candidate indexes. During this verification phase, some or all candidate indexes will be built and made valid so that

the performance effect on SQL statements can be measured. All candidate indexes remain invisible during the verification step.

If the performance of SQL statements is not improved by using the candidate indexes, they remain invisible.

4. Candidate valid indexes found to improve SQL performance will be made visible and available to the application workload. Candidate indexes that do not improve SQL performance will revert to invisible and be unusable after a short delay.

   During the verification stage, if an index is found to be beneficial, but an individual SQL statement suffers a performance regression, a SQL plan baseline is created to prevent the regression when the index is made visible.

5. Unusable and unused valid indexes are deleted by the automatic indexing process.

The automatic indexing process runs in the background every 15 minutes and performs the following operations:

a. Automatic index candidates are identified based on the usage of table columns in SQL statements. Ensure that table statistics are up to date. Tables without statistics are not considered for automatic indexing. Tables with stale statistics are not considered for automatic indexing.

b. Index candidates are initially created invisible and unusable. They are not visible to the application workload. Invisible automatic indexes cannot be used by SQL statements in the application workload.

   Automatic indexes can be single-column or multi-column. They are considered for the following:

   - Table columns (including virtual columns)
   - Partitioned and non-partitioned tables
   - Selected expressions (for example, JSON expressions)

c. A sample of workload SQL statements is tested against the candidate indexes. During this verification phase, some or all candidate indexes will be built and made valid so that the performance effect on SQL statements can be measured. All candidate indexes remain invisible during the verification step.

   If the performance of SQL statements is not improved by using the candidate indexes, they remain invisible.

d. Candidate valid indexes found to improve SQL performance will be made visible and available to the application workload. Candidate indexes that do not improve SQL performance will revert to invisible and be unusable after a short delay.

   During the verification stage, if an index is found to be beneficial, but an individual SQL statement suffers a performance regression, a SQL plan baseline is created to prevent the regression when the index is made visible.

e. Unusable and unused valid indexes are deleted by the automatic indexing process.

> **Note:**
>
> By default, the unused automatic indexes are deleted after 373 days. The period for retaining the unused automatic indexes in a database can be configured using the `DBMS_AUTO_INDEX.CONFIGURE` procedure.

> **See Also:**
>
> *Configuring Automatic Indexing in Oracle Database*

1.5.2.1.4.2 Enabling and Managing Automatic Indexing

The DBMS_AUTO_INDEX package provides options for configuring, dropping, monitoring, and reporting on automatic indexing.

You can use the DBMS_AUTO_INDEX package to do the following:

- Enable automatic indexing.
  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_MODE','IMPLEMENT')`
- Configure additional settings, such as how long to retain unused auto indexes
  `EXEC DBMS_AUTO_INDEX.CONFIGURE('AUTO_INDEX_RETENTION_FOR_AUTO','180')`
- Drop an automatic index. Carefully note the use of single and double quotation marks in the first example.
  Drop a single index owned by a schema and allow recreate.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH','"SYS_AI_612UD3J5NGF0C"',TRUE)
  ```

  Drop all indexes owned by a schema and allow recreate.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('SH',NULL,TRUE)
  ```

  Drop all indexes owned by a schema and disallow recreate. Then, change the recreation status back to `allow`.

  ```
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL)
  EXEC DBMS_AUTO_INDEX.DROP_AUTO_INDEXES('HR',NULL,TRUE)
  ```

- Report on the automatic indexing task and configuration settings.

Additional Controls

By setting the `OPTIMIZER_SESSION_TYPE` initialization parameter to `ADHOC` in a session, you can suspend automatic indexing for queries in this session. The automatic indexing process does not identify index candidates, or create and verify indexes. This control may be useful for ad hoc queries or testing new functionality.

See Also:

- *Oracle Database Administrator’s Guide* to learn more about automatic indexing
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the procedures and functions available in the `DBMS_AUTO_INDEX` package
- *Oracle Database Reference* to learn more about `OPTIMIZER_SESSION_TYPE`.

1.5.2.1.5 SQL Plan Management

SQL plan management is a preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans.

This mechanism can build a SQL plan baseline, which contains one or more accepted plans for each SQL statement. By using baselines, SQL plan management can prevent plan regressions from environmental changes, while permitting the optimizer to discover and use better plans.

See Also:

- "Overview of SQL Plan Management"
- *Oracle Database PL/SQL Packages and Types Reference*

  to learn about the `DBMS_SPM` package

1.5.2.1.6 SQL Performance Analyzer

SQL Performance Analyzer determines the effect of a change on a SQL workload by identifying performance divergence for each SQL statement.

System changes such as upgrading a database or adding an index may cause changes to execution plans, affecting SQL performance. By using SQL Performance Analyzer, you can accurately forecast the effect of system changes on SQL performance. Using this information, you can tune the database when SQL performance regresses, or validate and measure the gain when SQL performance improves.

See Also:

Oracle Database Testing Guide

1.5.2.2 Manual SQL Tuning Tools

In some situations, you may want to run manual tools in addition to the automated tools. Alternatively, you may not have access to the automated tools.

1.5.2.2.1 Execution Plans

Execution plans are the principal diagnostic tool in manual SQL tuning. For example, you can view plans to determine whether the optimizer selects the plan you expect, or identify the effect of creating an index on a table.

You can display execution plans in multiple ways. The following tools are the most commonly used:

- `DBMS_XPLAN`

  You can use the `DBMS_XPLAN` package methods to display the execution plan generated by the `EXPLAIN PLAN` command and query of `V$SQL_PLAN`.

- `EXPLAIN PLAN`

  This SQL statement enables you to view the execution plan that the optimizer would use to execute a SQL statement without actually executing the statement. See *Oracle Database SQL Language Reference*.

- `V$SQL_PLAN` and related views

  These views contain information about executed SQL statements, and their execution plans, that are still in the shared pool. See *Oracle Database Reference*.

- `AUTOTRACE`

  The `AUTOTRACE` command in SQL*Plus generates the execution plan and statistics about the performance of a query. This command provides statistics such as disk reads and memory reads. See *SQL*Plus User's Guide and Reference*.

1.5.2.2.2 Real-Time SQL Monitoring and Real-Time Database Operations

The Real-Time SQL Monitoring feature of Oracle Database enables you to monitor the performance of SQL statements while they are executing. By default, SQL monitoring starts automatically when a statement runs in parallel, or when it has consumed at least 5 seconds of CPU or I/O time in a single execution.

A database operation is a set of database tasks defined by end users or application code, for example, a batch job or Extraction, Transformation, and Loading (ETL) processing. You can define, monitor, and report on database operations. Real-Time Database Operations provides the ability to monitor composite operations automatically. The database automatically monitors parallel queries, DML, and DDL statements as soon as execution begins.

Oracle Enterprise Manager Cloud Control (Cloud Control) provides easy-to-use SQL monitoring pages. Alternatively, you can monitor SQL-related statistics using the `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views. You can use these views with the following views to get more information about executions that you are monitoring:

- `V$ACTIVE_SESSION_HISTORY`
- `V$SESSION`
- `V$SESSION_LONGOPS`

- `V$SQL`
- `V$SQL_PLAN`

> **See Also:**
>
> - "About Monitoring Database Operations"
> - *Oracle Database Reference* to learn about the `V$` views

1.5.2.2.3 Application Tracing

A **SQL trace file** provides performance information on individual SQL statements: parse counts, physical and logical reads, misses on the library cache, and so on.

Trace files are sometimes useful for diagnosing SQL performance problems. You can enable and disable SQL tracing for a specific session using the `DBMS_MONITOR` or `DBMS_SESSION` packages. Oracle Database implements tracing by generating a trace file for each server process when you enable the tracing mechanism.

Oracle Database provides the following command-line tools for analyzing trace files:

- `TKPROF`

  This utility accepts as input a trace file produced by the SQL Trace facility, and then produces a formatted output file.
- `trcsess`

  This utility consolidates trace output from multiple trace files based on criteria such as session ID, client ID, and service ID. After `trcsess` merges the trace information into a single output file, you can format the output file with `TKPROF`. `trcsess` is useful for consolidating the tracing of a particular session for performance or debugging purposes.

End-to-End Application Tracing simplifies the process of diagnosing performance problems in multitier environments. In these environments, the middle tier routes a request from an end client to different database sessions, making it difficult to track a client across database sessions. End-to-End application tracing uses a client ID to uniquely trace a specific end-client through all tiers to the database.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_MONITOR` and `DBMS_SESSION`

1.5.2.2.4 Optimizer Hints

A **hint** is an instruction passed to the optimizer through comments in a SQL statement.

Hints enable you to make decisions normally made automatically by the optimizer. In a test or development environment, hints are useful for testing the performance of a specific access path. For example, you may know that a specific index is more selective for certain queries. In this case, you may use hints to instruct the optimizer to use a better execution plan, as in the following example:

```
SELECT /*+ INDEX (employees emp_department_ix) */
       employee_id, department_id
FROM   employees
WHERE  department_id > 50;
```

Sometimes the database may not use a hint because of typos, invalid arguments, conflicting hints, and hints that are made invalid by transformations. Starting in Oracle Database 19c, you can generate a report about which hints were used or not used during plan generation.

See Also:

- "Influencing the Optimizer with Hints"
- *Oracle Database SQL Language Reference* to learn more about hints

1.5.3 User Interfaces to SQL Tuning Tools

Cloud Control is a system management tool that provides centralized management of a database environment. Cloud Control provides access to most tuning tools.

By combining a graphical console, Oracle Management Servers, Oracle Intelligent Agents, common services, and administrative tools, Cloud Control provides a comprehensive system management platform.

You can access all SQL tuning tools using a command-line interface. For example, the `DBMS_SQLTUNE` package is the command-line interface for SQL Tuning Advisor.

Oracle recommends Cloud Control as the best interface for database administration and tuning. In cases where the command-line interface better illustrates a particular concept or task, this manual uses command-line examples. However, in these cases the tuning tasks include a reference to the principal Cloud Control page associated with the task.

2

SQL Performance Methodology

This chapter describes the recommended methodology for SQL tuning.

Note:

This book assumes that you have learned the Oracle Database performance methodology described in *Oracle Database 2 Day + Performance Tuning Guide*.

2.1 Guidelines for Designing Your Application

The key to obtaining good SQL performance is to design your application with performance in mind.

2.1.1 Guideline for Data Modeling

Data modeling is important to successful application design.

You must perform data modeling in a way that represents the business practices. Heated debates may occur about the correct data model. The important thing is to apply greatest modeling efforts to those entities affected by the most frequent business transactions.

In the modeling phase, there is a great temptation to spend too much time modeling the non-core data elements, which results in increased development lead times. Use of modeling tools can then rapidly generate schema definitions and can be useful when a fast prototype is required.

2.1.2 Guideline for Writing Efficient Applications

During the design and architecture phase of system development, ensure that the application developers understand SQL execution efficiency.

To achieve this goal, the development environment must support the following characteristics:

- Good database connection management

  Connecting to the database is an expensive operation that is not scalable. Therefore, a best practice is to minimize the number of concurrent connections to the database. A simple system, where a user connects at application initialization, is ideal. However, in a web-based or multitiered application in which application servers multiplex database connections to users, this approach can be difficult. With these types of applications, design them to pool database connections, and not reestablish connections for each user request.

- Good cursor usage and management

Maintaining user connections is equally important to minimizing the parsing activity on the system. Parsing is the process of interpreting a SQL statement and creating an execution plan for it. This process has many phases, including syntax checking, security checking, execution plan generation, and loading shared structures into the shared pool. There are two types of parse operations:

– Hard parsing

  A SQL statement is submitted for the first time, and no match is found in the shared pool. Hard parses are the most resource-intensive and unscalable, because they perform all the operations involved in a parse.

– Soft parsing

  A SQL statement is submitted for the first time, and a match is found in the shared pool. The match can be the result of previous execution by another user. The SQL statement is shared, which is optimal for performance. However, soft parses are not ideal, because they still require syntax and security checking, which consume system resources.

Because parsing should be minimized as much as possible, application developers should design their applications to parse SQL statements once and execute them many times. This is done through cursors. Experienced SQL programmers should be familiar with the concept of opening and re-executing cursors.

- Effective use of bind variables

  Application developers must also ensure that SQL statements are shared within the shared pool. To achieve this goal, use bind variables to represent the parts of the query that change from execution to execution. If this is not done, then the SQL statement is likely to be parsed once and never re-used by other users. To ensure that SQL is shared, use bind variables and do not use string literals with SQL statements. For example:

  Statement with string literals:

```
SELECT *
FROM   employees
WHERE  last_name LIKE 'KING';
```

  Statement with bind variables:

```
SELECT *
FROM   employees
WHERE  last_name LIKE :1;
```

  The following example shows the results of some tests on a simple OLTP application:

```
Test                             #Users Supported
No Parsing all statements               270
Soft Parsing all statements             150
Hard Parsing all statements              60
Re-Connecting for each Transaction       30
```

These tests were performed on a four-CPU computer. The differences increase as the number of CPUs on the system increase.

2.2 Guidelines for Deploying Your Application

To achieve optimal performance, deploy your application with the same care that you put into designing it.

2.2.1 Guideline for Deploying in a Test Environment

The testing process mainly consists of functional and stability testing. At some point in the process, you must perform performance testing.

The following list describes simple rules for performance testing an application. If correctly documented, then this list provides important information for the production application and the capacity planning process after the application has gone live.

- Use the Automatic Database Diagnostic Monitor (ADDM) and SQL Tuning Advisor for design validation.
- Test with realistic data volumes and distributions.

  All testing must be done with fully populated tables. The test database should contain data representative of the production system in terms of data volume and cardinality between tables. All the production indexes should be built and the schema statistics should be populated correctly.
- Use the correct optimizer mode.

  Perform all testing with the optimizer mode that you plan to use in production.
- Test a single user performance.

  Test a single user on an idle or lightly-used database for acceptable performance. If a single user cannot achieve acceptable performance under ideal conditions, then multiple users cannot achieve acceptable performance under real conditions.
- Obtain and document plans for all SQL statements.

  Obtain an execution plan for each SQL statement. Use this process to verify that the optimizer is obtaining an optimal execution plan, and that the relative cost of the SQL statement is understood in terms of CPU time and physical I/Os. This process assists in identifying the heavy use transactions that require the most tuning and performance work in the future.
- Attempt multiuser testing.

  This process is difficult to perform accurately, because user workload and profiles might not be fully quantified. However, transactions performing DML statements should be tested to ensure that there are no locking conflicts or serialization problems.
- Test with the correct hardware configuration.

  Test with a configuration as close to the production system as possible. Using a realistic system is particularly important for network latencies, I/O subsystem bandwidth, and processor type and speed. Failing to use this approach may result in an incorrect analysis of potential performance problems.
- Measure steady state performance.

When benchmarking, it is important to measure the performance under steady state conditions. Each benchmark run should have a ramp-up phase, where users are connected to the application and gradually start performing work on the application. This process allows for frequently cached data to be initialized into the cache and single execution operations—such as parsing—to be completed before the steady state condition. Likewise, after a benchmark run, a ramp-down period is useful so that the system frees resources, and users cease work and disconnect.

2.2.2 Guidelines for Application Rollout

When new applications are rolled out, two strategies are commonly adopted: the Big Bang approach, in which all users migrate to the new system at once, and the trickle approach, in which users slowly migrate from existing systems to the new one.

Both approaches have merits and disadvantages. The Big Bang approach relies on reliable testing of the application at the required scale, but has the advantage of minimal data conversion and synchronization with the old system, because it is simply switched off. The Trickle approach allows debugging of scalability issues as the workload increases, but might mean that data must be migrated to and from legacy systems as the transition takes place.

It is difficult to recommend one approach over the other, because each technique has associated risks that could lead to system outages as the transition takes place. Certainly, the Trickle approach allows profiling of real users as they are introduced to the new application, and allows the system to be reconfigured while only affecting the migrated users. This approach affects the work of the early adopters, but limits the load on support services. Thus, unscheduled outages only affect a small percentage of the user population.

The decision on how to roll out a new application is specific to each business. Any adopted approach has its own unique pressures and stresses. The more testing and knowledge that you derive from the testing process, the more you realize what is best for the rollout.

Part II
Query Optimizer Fundamentals

To tune Oracle SQL, you must understand the query optimizer. The optimizer is built-in software that determines the most efficient method for a statement to access data.

3

SQL Processing

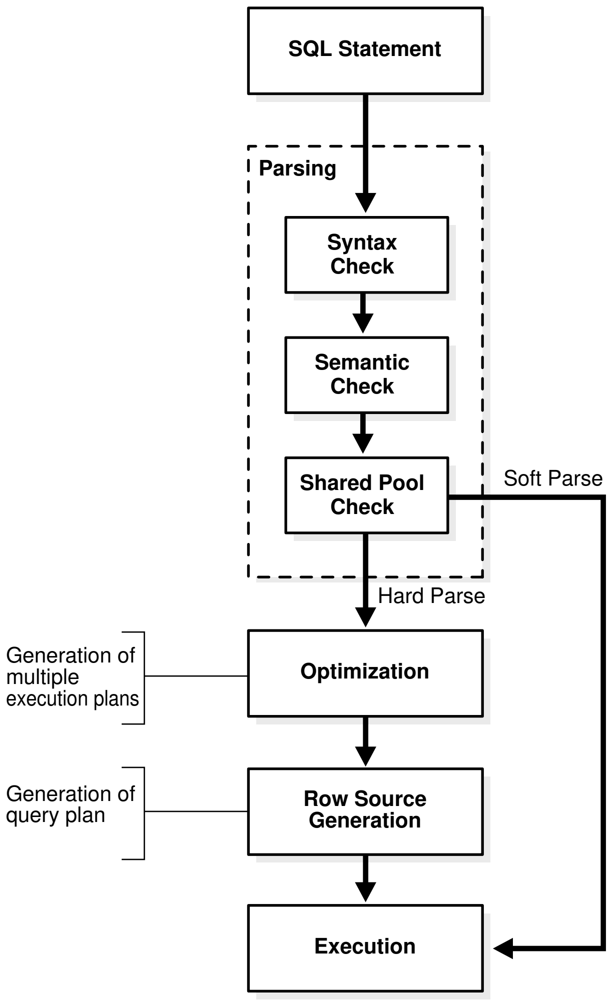
This chapter explains how database processes DDL statements to create objects, DML to modify data, and queries to retrieve data.

3.1 About SQL Processing

SQL processing is the parsing, optimization, row source generation, and execution of a SQL statement.

The following figure depicts the general stages of SQL processing. Depending on the statement, the database may omit some of these stages.

Figure 3-1 Stages of SQL Processing



3.1.1 SQL Parsing

The first stage of SQL processing is **parsing**.

The parsing stage involves separating the pieces of a SQL statement into a data structure that other routines can process. The database parses a statement when instructed by the application, which means that only the application, and not the database itself, can reduce the number of parses.

When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution. The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information. The cursor and private SQL area are in the program global area (PGA).

During the parse call, the database performs checks that identify the errors that can be found *before statement execution*. Some errors cannot be caught by parsing. For example, the database can encounter deadlocks or errors in data conversion only during statement execution.

See Also:

Oracle Database Concepts to learn about deadlocks

3.1.1.1 Syntax Check

Oracle Database must check each SQL statement for syntactic validity.

A statement that breaks a rule for well-formed SQL syntax fails the check. For example, the following statement fails because the keyword `FROM` is misspelled as `FORM`:

```
SQL> SELECT * FORM employees;
SELECT * FORM employees
         *
ERROR at line 1:
ORA-00923: FROM keyword not found where expected
```

3.1.1.2 Semantic Check

The semantics of a statement are its meaning. A semantic check determines whether a statement is meaningful, for example, whether the objects and columns in the statement exist.

A syntactically correct statement can fail a semantic check, as shown in the following example of a query of a nonexistent table:

```
SQL> SELECT * FROM nonexistent_table;
SELECT * FROM nonexistent_table
              *
```

```
ERROR at line 1:
ORA-00942: table or view does not exist
```

3.1.1.3 Shared Pool Check

During the parse, the database performs a shared pool check to determine whether it can skip resource-intensive steps of statement processing.

To this end, the database uses a hashing algorithm to generate a hash value for every SQL statement. The statement hash value is the SQL ID shown in `V$SQL.SQL_ID`. This hash value is deterministic within a version of Oracle Database, so the same statement in a single instance or in different instances has the same SQL ID.

When a user submits a SQL statement, the database searches the shared SQL area to see if an existing parsed statement has the same hash value. The hash value of a SQL statement is distinct from the following values:

- Memory address for the statement

  Oracle Database uses the SQL ID to perform a keyed read in a lookup table. In this way, the database obtains possible memory addresses of the statement.
- Hash value of an execution plan for the statement

  A SQL statement can have multiple plans in the shared pool. Typically, each plan has a different hash value. If the same SQL ID has multiple plan hash values, then the database knows that multiple plans exist for this SQL ID.

Parse operations fall into the following categories, depending on the type of statement submitted and the result of the hash check:

- Hard parse

  If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code. This operation is known as a hard parse, or a library cache miss.

  > **Note:**
  >
  > The database always performs a hard parse of DDL.

  During the hard parse, the database accesses the library cache and data dictionary cache numerous times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change. Latch contention increases statement execution time and decreases concurrency.
- Soft parse

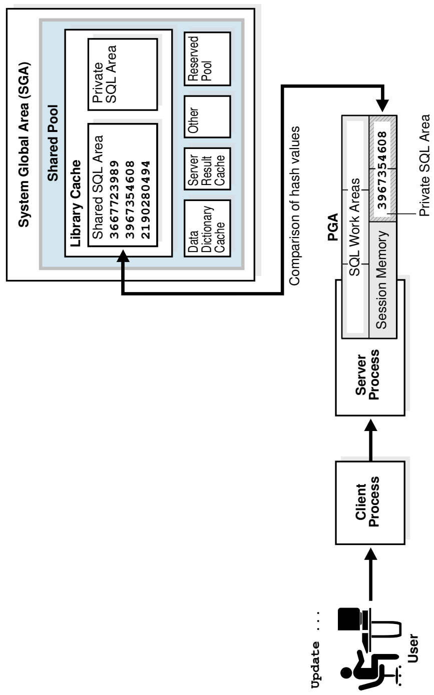
  A soft parse is any parse that is not a hard parse. If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a library cache hit.

  Soft parses can vary in how much work they perform. For example, configuring the session shared SQL area can sometimes reduce the amount of latching in the soft parses, making them "softer."

In general, a soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution.

The following graphic is a simplified representation of a shared pool check of an `UPDATE` statement in a dedicated server architecture.

Figure 3-2   Shared Pool Check



If a check determines that a statement in the shared pool has the same hash value, then the database performs semantic and environment checks to determine whether the statements have the same meaning. Identical syntax is not sufficient. For example, suppose two different users log in to the database and issue the following SQL statements:

```
CREATE TABLE my_table ( some_col INTEGER );
SELECT * FROM my_table;
```

The `SELECT` statements for the two users are syntactically identical, but two separate schema objects are named `my_table`. This semantic difference means that the second statement cannot reuse the code for the first statement.

Even if two statements are semantically identical, an environmental difference can force a hard parse. In this context, the optimizer environment is the totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode). Consider the following series of SQL statements executed by a single user:

```
ALTER SESSION SET OPTIMIZER_MODE=ALL_ROWS;
ALTER SYSTEM FLUSH SHARED_POOL;               # optimizer environment 1
SELECT * FROM sh.sales;
```

```
ALTER SESSION SET OPTIMIZER_MODE=FIRST_ROWS;  # optimizer environment 2
SELECT * FROM sh.sales;

ALTER SESSION SET SQL_TRACE=true;             # optimizer environment 3
SELECT * FROM sh.sales;
```

In the preceding example, the same `SELECT` statement is executed in three different optimizer environments. Consequently, the database creates three separate shared SQL areas for these statements and forces a hard parse of each statement.

See Also:

- *Oracle Database Concepts* to learn about private SQL areas and shared SQL areas
- *Oracle Database Performance Tuning Guide* to learn how to configure the shared pool
- *Oracle Database Concepts* to learn about latches

3.1.2 SQL Optimization

During optimization, Oracle Database must perform a hard parse at least once for every unique DML statement and performs the optimization during this parse.

The database does not optimize DDL. The only exception is when the DDL includes a DML component such as a subquery that requires optimization.

3.1.3 SQL Row Source Generation

The **row source generator** is software that receives the optimal execution plan from the optimizer and produces an iterative execution plan that is usable by the rest of the database.

The iterative plan is a binary program that, when executed by the SQL engine, produces the result set. The plan takes the form of a combination of steps. Each step returns a row set. The next step either uses the rows in this set, or the last step returns the rows to the application issuing the SQL statement.

A row source is a row set returned by a step in the execution plan along with a control structure that can iteratively process the rows. The row source can be a table, view, or result of a join or grouping operation.

The row source generator produces a row source tree, which is a collection of row sources. The row source tree shows the following information:

- An ordering of the tables referenced by the statement
- An access method for each table mentioned in the statement
- A join method for tables affected by join operations in the statement
- Data operations such as filter, sort, or aggregation

Example 3-1   Execution Plan

This example shows the execution plan of a `SELECT` statement when `AUTOTRACE` is enabled. The statement selects the last name, job title, and department name for all employees whose last names begin with the letter `A`. The execution plan for this statement is the output of the row source generator.

```
SELECT e.last_name, j.job_title, d.department_name
FROM   hr.employees e, hr.departments d, hr.jobs j
WHERE  e.department_id = d.department_id
AND    e.job_id = j.job_id
AND    e.last_name LIKE 'A%';

Execution Plan
----------------------------------------------------------
Plan hash value: 975837011

-----------------------------------------------------------------------
----
| Id| Operation                    | Name        |Rows|Bytes|Cost(%CPU)|
Time|
-----------------------------------------------------------------------
----
| 0| SELECT STATEMENT              |              | 3 |189 |7(15)|
00:00:01 |
|*1|  HASH JOIN                    |              | 3 |189 |7(15)|
00:00:01 |
|*2|   HASH JOIN                   |              | 3 |141 |5(20)|
00:00:01 |
| 3|    TABLE ACCESS BY INDEX ROWID| EMPLOYEES    | 3 | 60 |2 (0)|
00:00:01 |
|*4|     INDEX RANGE SCAN          | EMP_NAME_IX | 3 |    |1 (0)|
00:00:01 |
| 5|    TABLE ACCESS FULL          | JOBS        |19 |513 |2 (0)|
00:00:01 |
| 6|   TABLE ACCESS FULL           | DEPARTMENTS |27 |432 |2 (0)|
00:00:01 |
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   2 - access("E"."JOB_ID"="J"."JOB_ID")
   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
```

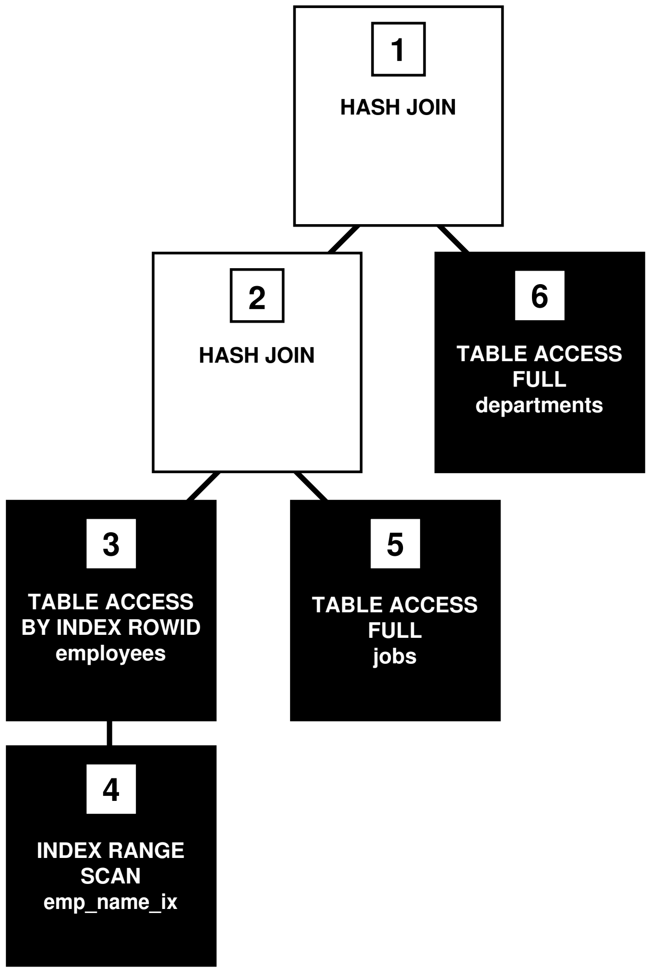
3.1.4 SQL Execution

During execution, the SQL engine executes each row source in the tree produced by the row source generator. This step is the only mandatory step in DML processing.

Figure 3-3 is an execution tree, also called a *parse tree*, that shows the flow of row sources from one step to another in the plan in Example 3-1. In general, the order of the steps in execution is the *reverse* of the order in the plan, so you read the plan from the bottom up.

Each step in an execution plan has an ID number. The numbers in Figure 3-3 correspond to the `Id` column in the plan shown in Example 3-1. Initial spaces in the `Operation` column of the plan indicate hierarchical relationships. For example, if the name of an operation is preceded by two spaces, then this operation is a child of an operation preceded by one space. Operations preceded by one space are children of the `SELECT` statement itself.

Figure 3-3 Row Source Tree


1
HASH JOIN
2
HASH JOIN
6
TABLE ACCESS FULL departments
3
TABLE ACCESS BY INDEX ROWID employees
5
TABLE ACCESS FULL jobs
4
INDEX RANGE SCAN emp_name_ix

In Figure 3-3, each node of the tree acts as a row source, which means that each step of the execution plan in Example 3-1 either retrieves rows from the database or accepts rows from one or more row sources as input. The SQL engine executes each row source as follows:

- Steps indicated by the black boxes physically retrieve data from an object in the database. These steps are the access paths, or techniques for retrieving data from the database.
  - Step 6 uses a full table scan to retrieve all rows from the `departments` table.

  - Step 5 uses a full table scan to retrieve all rows from the `jobs` table.
  - Step 4 scans the `emp_name_ix` index in order, looking for each key that begins with the letter `A` and retrieving the corresponding rowid. For example, the rowid corresponding to `Atkinson` is `AAAPzRAAFAAAABSAAe`.
  - Step 3 retrieves from the `employees` table the rows whose rowids were returned by Step 4. For example, the database uses rowid `AAAPzRAAFAAAABSAAe` to retrieve the row for `Atkinson`.
- Steps indicated by the clear boxes operate on row sources.
  - Step 2 performs a hash join, accepting row sources from Steps 3 and 5, joining each row from the Step 5 row source to its corresponding row in Step 3, and returning the resulting rows to Step 1.

    For example, the row for employee `Atkinson` is associated with the job name `Stock Clerk`.
  - Step 1 performs another hash join, accepting row sources from Steps 2 and 6, joining each row from the Step 6 source to its corresponding row in Step 2, and returning the result to the client.

    For example, the row for employee `Atkinson` is associated with the department named `Shipping`.

In some execution plans the steps are iterative and in others sequential. The hash join shown in Example 3-1 is sequential. The database completes the steps in their entirety based on the join order. The database starts with the index range scan of `emp_name_ix`. Using the rowids that it retrieves from the index, the database reads the matching rows in the `employees` table, and then scans the `jobs` table. After it retrieves the rows from the `jobs` table, the database performs the hash join.

During execution, the database reads the data from disk into memory if the data is not in memory. The database also takes out any locks and latches necessary to ensure data integrity and logs any changes made during the SQL execution. The final stage of processing a SQL statement is closing the cursor.

3.2 How Oracle Database Processes DML

Most DML statements have a query component. In a query, execution of a cursor places the results of the query into a set of rows called the **result set**.

3.2.1 How Row Sets Are Fetched

Result set rows can be fetched either a row at a time or in groups.

In the fetch stage, the database selects rows and, if requested by the query, orders the rows. Each successive fetch retrieves another row of the result until the last row has been fetched.

In general, the database cannot determine for certain the number of rows to be retrieved by a query until the last row is fetched. Oracle Database retrieves the data in response to fetch calls, so that the more rows the database reads, the more work it performs. For some queries the database returns the first row as quickly as possible, whereas for others it creates the entire result set before returning the first row.

3.2.2 Read Consistency

In general, a query retrieves data by using the Oracle Database read consistency mechanism, which guarantees that all data blocks read by a query are consistent to a single point in time.

Read consistency uses undo data to show past versions of data. For an example, suppose a query must read 100 data blocks in a full table scan. The query processes the first 10 blocks while DML in a different session modifies block 75. When the first session reaches block 75, it realizes the change and uses undo data to retrieve the old, unmodified version of the data and construct a noncurrent version of block 75 in memory.

See Also:

Oracle Database Concepts to learn about multiversion read consistency

3.2.3 Data Changes

DML statements that must change data use read consistency to retrieve only the data that matched the search criteria when the modification began.

Afterward, these statements retrieve the data blocks as they exist in their current state and make the required modifications. The database must perform other actions related to the modification of the data such as generating redo and undo data.

3.3 How Oracle Database Processes DDL

Oracle Database processes DDL differently from DML.

For example, when you create a table, the database does not optimize the `CREATE TABLE` statement. Instead, Oracle Database parses the DDL statement and carries out the command.

The database processes DDL differently because it is a means of defining an object in the data dictionary. Typically, Oracle Database must parse and execute many recursive SQL statements to execute a DDL statement. Suppose you create a table as follows:

```
CREATE TABLE mytable (mycolumn INTEGER);
```

Typically, the database would run dozens of recursive statements to execute the preceding statement. The recursive SQL would perform actions such as the following:

- Issue a `COMMIT` before executing the `CREATE TABLE` statement
- Verify that user privileges are sufficient to create the table
- Determine which tablespace the table should reside in
- Ensure that the tablespace quota has not been exceeded
- Ensure that no object in the schema has the same name
- Insert rows that define the table into the data dictionary

- Issue a `COMMIT` if the DDL statement succeeded or a `ROLLBACK` if it did not

See Also:

Oracle Database Development Guide to learn about processing DDL, transaction control, and other types of statements

4

Query Optimizer Concepts

This chapter describes the most important concepts relating to the query optimizer, including its principal components.

4.1 Introduction to the Query Optimizer

The **query optimizer** (called simply the **optimizer**) is built-in database software that determines the most efficient method for a SQL statement to access requested data.

4.1.1 Purpose of the Query Optimizer

The optimizer attempts to generate the most optimal execution plan for a SQL statement.

The optimizer choose the plan with the lowest cost among all considered candidate plans. The optimizer uses available statistics to calculate cost. For a specific query in a given environment, the cost computation accounts for factors of query execution such as I/O, CPU, and communication.

For example, a query might request information about employees who are managers. If the optimizer statistics indicate that 80% of employees are managers, then the optimizer may decide that a full table scan is most efficient. However, if statistics indicate that very few employees are managers, then reading an index followed by a table access by rowid may be more efficient than a full table scan.

Because the database has many internal statistics and tools at its disposal, the optimizer is usually in a better position than the user to determine the optimal method of statement execution. For this reason, all SQL statements use the optimizer.

4.1.2 Cost-Based Optimization

Query optimization is the process of choosing the most efficient means of executing a SQL statement.

SQL is a nonprocedural language, so the optimizer is free to merge, reorganize, and process in any order. The database optimizes each SQL statement based on statistics collected about the accessed data. The optimizer determines the optimal plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, different join methods such as nested loops and hash joins, different join orders, and possible transformations.

For a given query and environment, the optimizer assigns a relative numerical cost to each step of a possible plan, and then factors these values together to generate an overall cost estimate for the plan. After calculating the costs of alternative plans, the optimizer chooses the plan with the lowest cost estimate. For this reason, the optimizer is sometimes called the cost-based optimizer (CBO) to contrast it with the legacy rule-based optimizer (RBO).

Note:

The optimizer may not make the same decisions from one version of Oracle Database to the next. In recent versions, the optimizer might make different decision because better information is available and more optimizer transformations are possible.

4.1.3 Execution Plans

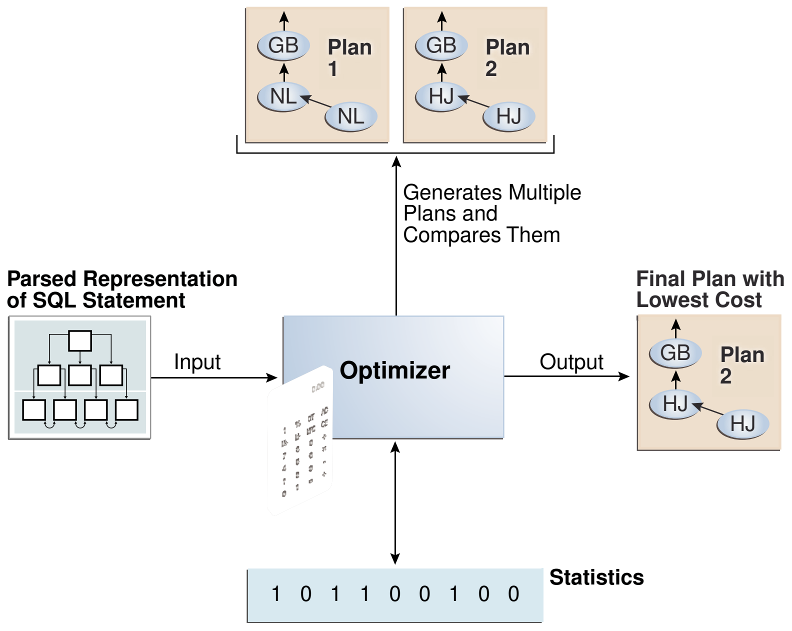
An **execution plan** describes a recommended method of execution for a SQL statement.

The plan shows the combination of the steps Oracle Database uses to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement.

An execution plan displays the cost of the entire plan, indicated on line 0, and each separate operation. The cost is an internal unit that the execution plan only displays to allow for plan comparisons. Thus, you cannot tune or change the cost value.

In the following graphic, the optimizer generates two possible execution plans for an input SQL statement, uses statistics to estimate their costs, compares their costs, and then chooses the plan with the lowest cost.

Figure 4-1 Execution Plans


GB
Plan 1
NL
NL
GB
Plan 2
HJ
HJ
Generates Multiple Plans and Compares Them
Parsed Representation of SQL Statement
Input
Optimizer
Output
Final Plan with Lowest Cost
GB
Plan 2
HJ
HJ
1 0 1 1 0 0 1 0 0
Statistics

4.1.3.1 Query Blocks

The input to the optimizer is a parsed representation of a SQL statement.

Each `SELECT` block in the original SQL statement is represented internally by a query block. A query block can be a top-level statement, subquery, or unmerged view.

Example 4-1 Query Blocks

The following SQL statement consists of two query blocks. The subquery in parentheses is the inner query block. The outer query block, which is the rest of the SQL statement, retrieves names of employees in the departments whose IDs were supplied by the subquery. The query form determines how query blocks are interrelated.

```
SELECT first_name, last_name
FROM   hr.employees
WHERE  department_id
IN     (SELECT department_id
        FROM   hr.departments
        WHERE  location_id = 1800);
```

See Also:

- "View Merging"
- *Oracle Database Concepts* for an overview of SQL processing

4.1.3.2 Query Subplans

For each query block, the optimizer generates a query subplan.

The database optimizes query blocks separately from the bottom up. Thus, the database optimizes the innermost query block first and generates a subplan for it, and then generates the outer query block representing the entire query.

The number of possible plans for a query block is proportional to the number of objects in the `FROM` clause. This number rises exponentially with the number of objects. For example, the possible plans for a join of five tables are significantly higher than the possible plans for a join of two tables.

4.1.3.3 Analogy for the Optimizer

One analogy for the optimizer is an online trip advisor.

A cyclist wants to know the most efficient bicycle route from point A to point B. A query is like the directive "I need the most efficient route from point A to point B" or "I need the most efficient route from point A to point B by way of point C." The trip advisor uses an internal algorithm, which relies on factors such as speed and difficulty, to determine the most efficient route. The cyclist can influence the trip advisor's decision by using directives such as "I want to arrive as fast as possible" or "I want the easiest ride possible."

In this analogy, an execution plan is a possible route generated by the trip advisor. Internally, the advisor may divide the overall route into several subroutes (subplans), and calculate the efficiency for each subroute separately. For example, the trip advisor may estimate one subroute at 15 minutes with medium difficulty, an alternative subroute at 22 minutes with minimal difficulty, and so on.

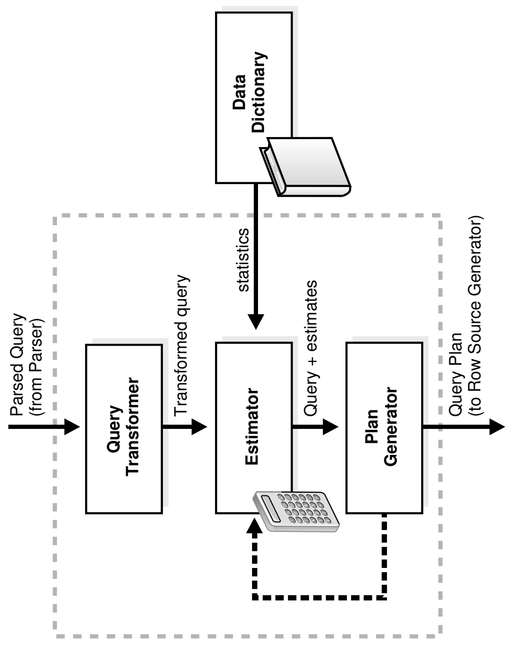
The advisor picks the most efficient (lowest cost) overall route based on user-specified goals and the available statistics about roads and traffic conditions. The more accurate the statistics, the better the advice. For example, if the advisor is not frequently notified of traffic jams, road closures, and poor road conditions, then the recommended route may turn out to be inefficient (high cost).

4.2 About Optimizer Components

The optimizer contains three components: the transformer, estimator, and plan generator.

The following graphic illustrates the components.

Figure 4-2 Optimizer Components



A set of query blocks represents a parsed query, which is the input to the optimizer. The following table describes the optimizer operations.

Table 4-1 Optimizer Operations

Phase	Operation	Description	To Learn More
1	Query Transformer	The optimizer determines whether it is helpful to change the form of the query so that the optimizer can generate a better execution plan.	"Query Transformer"
2	Estimator	The optimizer estimates the cost of each plan based on statistics in the data dictionary.	"Estimator"

Table 4-1 (Cont.) Optimizer Operations

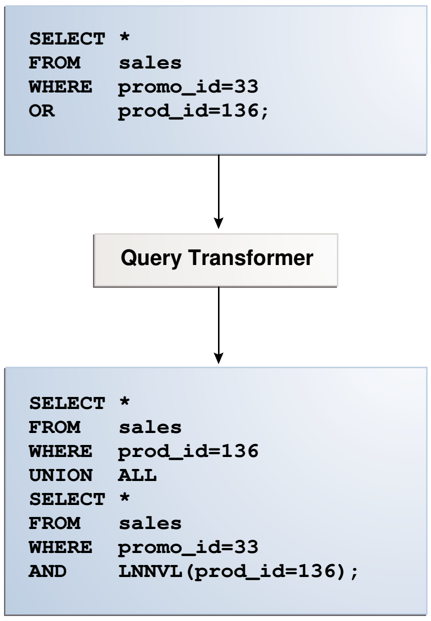
Phase	Operation	Description	To Learn More
3	Plan Generator	The optimizer compares the costs of plans and chooses the lowest-cost plan, known as the execution plan, to pass to the row source generator.	"Plan Generator"

4.2.1 Query Transformer

For some statements, the query transformer determines whether it is advantageous to rewrite the original SQL statement into a semantically equivalent SQL statement with a lower cost.

When a viable alternative exists, the database calculates the cost of the alternatives separately and chooses the lowest-cost alternative. The following graphic shows the query transformer rewriting an input query that uses `OR` into an output query that uses `UNION ALL`.

Figure 4-3 Query Transformer



```
SELECT *
FROM   sales
WHERE  promo_id=33
OR     prod_id=136;
```

```
SELECT *
FROM   sales
WHERE  prod_id=136
UNION  ALL
SELECT *
FROM   sales
WHERE  promo_id=33
AND    LNNVL(prod_id=136);
```

4.2.2 Estimator

The **estimator** is the component of the optimizer that determines the overall cost of a given execution plan.

The estimator uses three different measures to determine cost:

- Selectivity

  The percentage of rows in the row set that the query selects, with `0` meaning no rows and `1` meaning all rows. Selectivity is tied to a query predicate, such as `WHERE last_name LIKE 'A%'`, or a combination of predicates. A predicate becomes more selective as the selectivity value approaches `0` and less selective (or more unselective) as the value approaches `1`.

> **Note:**
>
> Selectivity is an internal calculation that is not visible in the execution plans.

- Cardinality

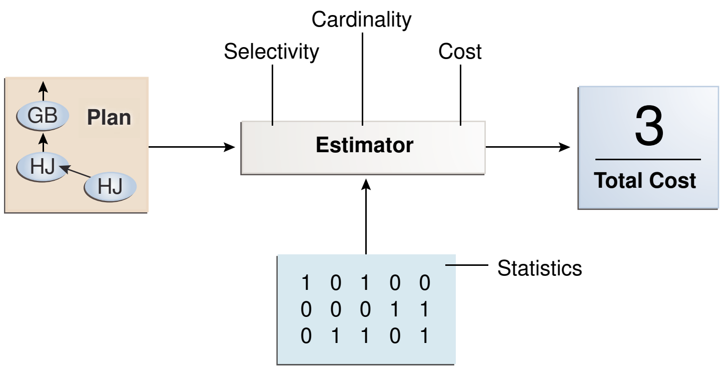
  The cardinality is the number of rows returned by each operation in an execution plan. This input, which is crucial to obtaining an optimal plan, is common to all cost functions. The estimator can derive cardinality from the table statistics collected by `DBMS_STATS`, or derive it after accounting for effects from predicates (filter, join, and so on), `DISTINCT` or `GROUP BY` operations, and so on. The `Rows` column in an execution plan shows the estimated cardinality.

- Cost

  This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.

As shown in the following graphic, if statistics are available, then the estimator uses them to compute the measures. The statistics improve the degree of accuracy of the measures.

Figure 4-4 Estimator



For the query shown in Example 4-1, the estimator uses selectivity, estimated cardinality (a total return of 10 rows), and cost measures to produce its total cost estimate of 3:

```
------------------------------------------------------------------------
----
|Id| Operation                   |Name          |Rows|Bytes|Cost %CPU|
Time|
------------------------------------------------------------------------
----
| 0| SELECT STATEMENT            |                  |10|250|3  (0)|
00:00:01|
| 1|  NESTED LOOPS               |                  |  |   |
|        |
| 2|   NESTED LOOPS                 |                 |10|250|3  (0)|00:00:01|
|*3|    TABLE ACCESS FULL           |DEPARTMENTS      | 1|  7|2  (0)|00:00:01|
|*4|    INDEX RANGE SCAN            |EMP_DEPARTMENT_IX|10|   |0  (0)|00:00:01|
| 5|   TABLE ACCESS BY INDEX ROWID|EMPLOYEES        |10|180|1  (0)|00:00:01|
----------------------------------------------------------------------
```

4.2.2.1 Selectivity

The **selectivity** represents a fraction of rows from a row set.

The row set can be a base table, a view, or the result of a join. The selectivity is tied to a query predicate, such as `last_name = 'Smith'`, or a combination of predicates, such as `last_name = 'Smith' AND job_id = 'SH_CLERK'`.

> **Note:**
>
> Selectivity is an internal calculation that is not visible in execution plans.

A predicate filters a specific number of rows from a row set. Thus, the selectivity of a predicate indicates how many rows pass the predicate test. Selectivity ranges from 0.0 to 1.0. A selectivity of 0.0 means that no rows are selected from a row set, whereas a selectivity of 1.0 means that all rows are selected. A predicate becomes more selective as the value approaches 0.0 and less selective (or more unselective) as the value approaches 1.0.

The optimizer estimates selectivity depending on whether statistics are available:

- Statistics not available

  Depending on the value of the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter, the optimizer either uses dynamic statistics or an internal default value. The database uses different internal defaults depending on the predicate type. For example, the internal default for an equality predicate (`last_name = 'Smith'`) is lower than for a range predicate (`last_name > 'Smith'`) because an equality predicate is expected to return a smaller fraction of rows.

- Statistics available

  When statistics are available, the estimator uses them to estimate selectivity. Assume there are 150 distinct employee last names. For an equality predicate `last_name = 'Smith'`, selectivity is the reciprocal of the number *n* of distinct values of `last_name`, which in this example is .006 because the query selects rows that contain 1 out of 150 distinct values.

  If a histogram exists on the `last_name` column, then the estimator uses the histogram instead of the number of distinct values. The histogram captures the distribution of different values in a column, so it yields better selectivity estimates, especially for columns that have data skew.

See Also:

- "Histograms "
- *Oracle Database Reference* to learn more about `OPTIMIZER_DYNAMIC_SAMPLING`

4.2.2.2 Cardinality

The **cardinality** is the number of rows returned by each operation in an execution plan.

For example, if the optimizer estimate for the number of rows returned by a full table scan is 100, then the cardinality estimate for this operation is 100. The cardinality estimate appears in the `Rows` column of the execution plan.

The optimizer determines the cardinality for each operation based on a complex set of formulas that use both table and column level statistics, or dynamic statistics, as input. The optimizer uses one of the simplest formulas when a single equality predicate appears in a single-table query, with no histogram. In this case, the optimizer assumes a uniform distribution and calculates the cardinality for the query by dividing the total number of rows in the table by the number of distinct values in the column used in the `WHERE` clause predicate.

For example, user `hr` queries the `employees` table as follows:

```
SELECT first_name, last_name
FROM   employees
WHERE  salary='10200';
```

The `employees` table contains 107 rows. The current database statistics indicate that the number of distinct values in the `salary` column is `58`. Therefore, the optimizer estimates the cardinality of the result set as `2`, using the formula `107/58=1.84`.

Cardinality estimates must be as accurate as possible because they influence all aspects of the execution plan. Cardinality is important when the optimizer determines the cost of a join. For example, in a nested loops join of the `employees` and `departments` tables, the number of rows in `employees` determines how often the database must probe the `departments` table. Cardinality is also important for determining the cost of sorts.

4.2.2.3 Cost

The **optimizer cost model** accounts for the machine resources that a query is predicted to use.

The cost is an internal numeric measure that represents the estimated resource usage for a plan. The cost is *specific* to a query in an optimizer environment. To estimate cost, the optimizer considers factors such as the following:

- System resources, which includes estimated I/O, CPU, and memory
- Estimated number of rows returned (cardinality)

- Size of the initial data sets
- Distribution of the data
- Access structures

> **Note:**
>
> The cost is an *internal* measure that the optimizer uses to compare different plans for the same query. You cannot tune or change cost.

The execution time is a function of the cost, but cost does not equate directly to time. For example, if the plan for query *A* has a lower cost than the plan for query *B*, then the following outcomes are possible:

- *A* executes faster than *B*.
- *A* executes slower than *B*.
- *A* executes in the same amount of time as *B*.

Therefore, you cannot compare the costs of different queries with one another. Also, you cannot compare the costs of semantically equivalent queries that use different optimizer modes.

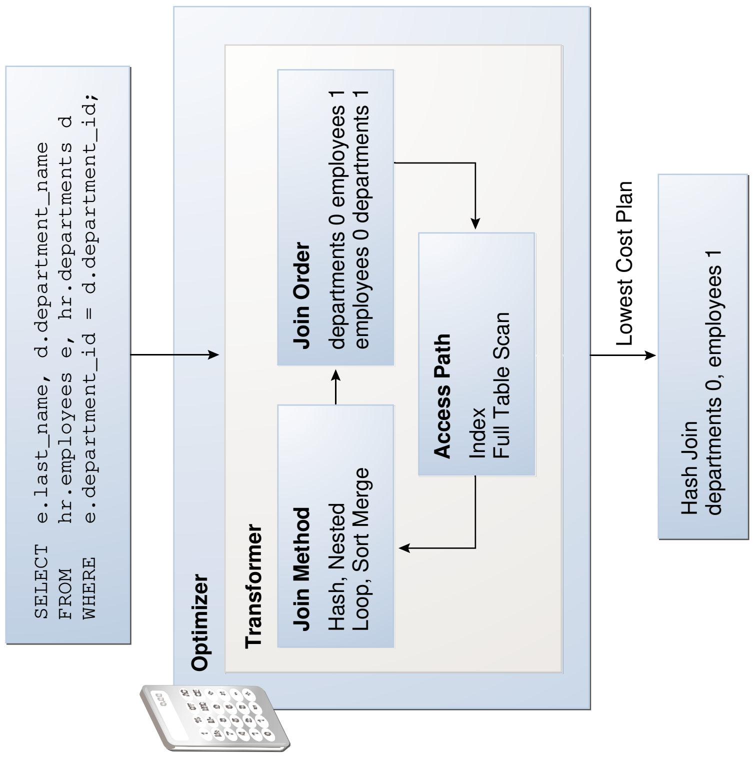
4.2.3 Plan Generator

The **plan generator** explores various plans for a query block by trying out different access paths, join methods, and join orders.

Many plans are possible because of the various combinations that the database can use to produce the same result. The optimizer picks the plan with the lowest cost.

The following graphic shows the optimizer testing different plans for an input query.

Figure 4-5 Plan Generator


SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d
WHERE e.department_id = d.department_id;
Optimizer
Transformer
Join Order
departments 0 employees 1
employees 0 departments 1
Join Method
Hash, Nested Loop, Sort Merge
Access Path
Index
Full Table Scan
Lowest Cost Plan
Hash Join
departments 0, employees 1

The following snippet from an optimizer trace file shows some computations that the optimizer performs:

```
GENERAL PLANS
***************************************
Considering cardinality-based initial join order.
Permutations for Starting Table :0
Join order[1]:  DEPARTMENTS[D]#0  EMPLOYEES[E]#1

***************
Now joining: EMPLOYEES[E]#1
***************
NL Join
  Outer table: Card: 27.00  Cost: 2.01  Resp: 2.01  Degree: 1  Bytes:
16
Access path analysis for EMPLOYEES
. . .
  Best NL cost: 13.17
. . .
SM Join
  SM cost: 6.08
     resc: 6.08 resc_io: 4.00 resc_cpu: 2501688
     resp: 6.08 resp_io: 4.00 resp_cpu: 2501688
. . .
SM Join (with index on outer)
  Access Path: index (FullScan)
. . .
```

```
HA Join
  HA cost: 4.57
     resc: 4.57 resc_io: 4.00 resc_cpu: 678154
     resp: 4.57 resp_io: 4.00 resp_cpu: 678154
Best:: JoinMethod: Hash
       Cost: 4.57  Degree: 1  Resp: 4.57  Card: 106.00 Bytes: 27
. . .

***********************
Join order[2]:  EMPLOYEES[E]#1  DEPARTMENTS[D]#0
. . .

***************
Now joining: DEPARTMENTS[D]#0
***************
. . .
HA Join
  HA cost: 4.58
     resc: 4.58 resc_io: 4.00 resc_cpu: 690054
     resp: 4.58 resp_io: 4.00 resp_cpu: 690054
Join order aborted: cost > best plan cost
***********************
```

The trace file shows the optimizer first trying the `departments` table as the outer table in the join. The optimizer calculates the cost for three different join methods: nested loops join (NL), sort merge (SM), and hash join (HA). The optimizer picks the hash join as the most efficient method:

```
Best:: JoinMethod: Hash
       Cost: 4.57  Degree: 1  Resp: 4.57  Card: 106.00 Bytes: 27
```

The optimizer then tries a different join order, using `employees` as the outer table. This join order costs more than the previous join order, so it is abandoned.

The optimizer uses an internal cutoff to reduce the number of plans it tries when finding the lowest-cost plan. The cutoff is based on the cost of the current best plan. If the current best cost is large, then the optimizer explores alternative plans to find a lower cost plan. If the current best cost is small, then the optimizer ends the search swiftly because further cost improvement is not significant.

4.3 About Automatic Tuning Optimizer

The optimizer performs different operations depending on how it is invoked.

The database provides the following types of optimization:

- Normal optimization

  The optimizer compiles the SQL and generates an execution plan. The normal mode generates a reasonable plan for most SQL statements. Under normal mode, the optimizer operates with strict time constraints, usually a fraction of a second, during which it must find an optimal plan.

- SQL Tuning Advisor optimization

When SQL Tuning Advisor invokes the optimizer, the optimizer is known as Automatic Tuning Optimizer. In this case, the optimizer performs additional analysis to further improve the plan produced in normal mode. The optimizer output is not an execution plan, but a series of actions, along with their rationale and expected benefit for producing a significantly better plan.

See Also:

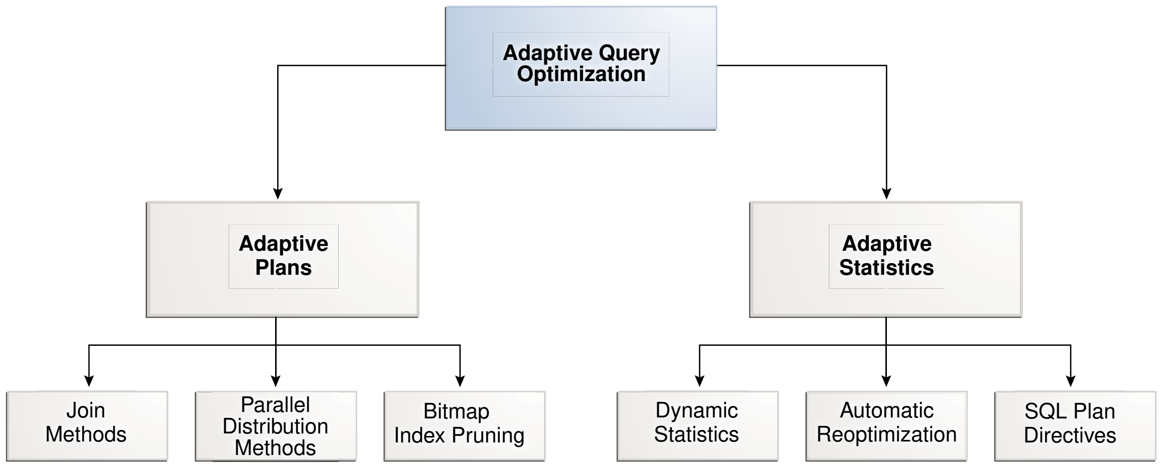
- "Analyzing SQL with SQL Tuning Advisor"
- *Oracle Database 2 Day + Performance Tuning Guide* to learn more about SQL Tuning Advisor

4.4 About Adaptive Query Optimization

In Oracle Database, **adaptive query optimization** enables the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.

Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan. The following graphic shows the feature set for adaptive query optimization.

Figure 4-6 Adaptive Query Optimization



4.4.1 Adaptive Query Plans

An **adaptive query plan** enables the optimizer to make a plan decision for a statement during execution.

Adaptive query plans enable the optimizer to fix some classes of problems at run time. Adaptive plans are enabled by default.

4.4.1.1 About Adaptive Query Plans

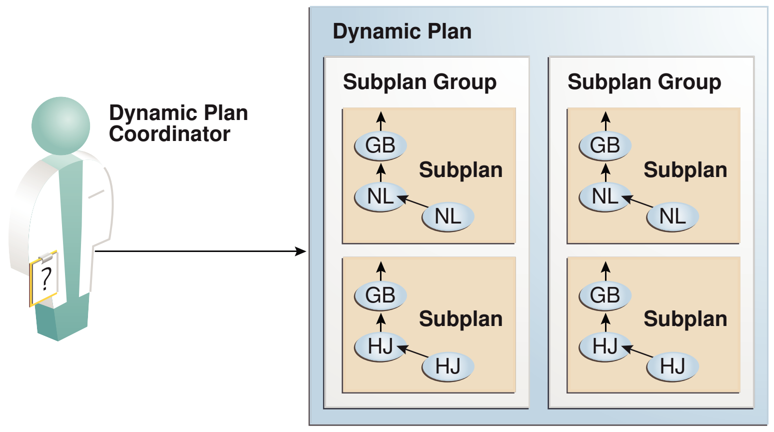
An adaptive query plan contains multiple predetermined subplans, and an optimizer statistics collector. Based on the statistics collected during execution, the dynamic plan coordinator chooses the best plan at run time.

Dynamic Plans

To change plans at runtime, adaptive query plans use a dynamic plan, which is represented as a set of subplan groups. A subplan group is a set of subplans. A subplan is a portion of a plan that the optimizer can switch to as an alternative at run time. For example, a nested loops join could switch to a hash join during execution.

The optimizer decides which subplan to use at run time. When notified of a new statistic value relevant to a subplan group, the coordinator dispatches it to the handler function for this subgroup.

Figure 4-7 Dynamic Plan Coordinator



Optimizer Statistics Collector

An optimizer statistics collector is a row source inserted into a plan at key points to collect run-time statistics relating to cardinality and histograms. These statistics help the optimizer make a final decision between multiple subplans. The collector also supports optional buffering up to an internal threshold.

For parallel buffering statistics collectors, each parallel execution server collects the statistics, which the parallel query coordinator aggregates and then sends to the clients. In this context, a *client* is a consumer of the collected statistics, such as a dynamic plan. Each client specifies a callback function to be executed on each parallel server or on the query coordinator.

4.4.1.2 Purpose of Adaptive Query Plans

The ability of the optimizer to adapt a plan, based on statistics obtained during execution, can greatly improve query performance.

Adaptive query plans are useful because the optimizer occasionally picks a suboptimal default plan because of a cardinality misestimate. The ability of the optimizer to pick the best plan at run time based on actual execution statistics results in a more optimal final plan. After choosing the final plan, the optimizer uses it for subsequent executions, thus ensuring that the suboptimal plan is not reused.

4.4.1.3 How Adaptive Query Plans Work

For the first execution of a statement, the optimizer uses the default plan, and then stores an adaptive plan. The database uses the adaptive plan for subsequent executions unless specific conditions are met.

During the *first* execution of a statement, the database performs the following steps:

1. The database begins executing the statement using the default plan.
2. The statistics collector gathers information about the in-progress execution, and buffers some rows received by the subplan.

   For parallel buffering statistics collectors, each child process collects the statistics, which the query coordinator aggregates before sending to the clients.
3. Based on the statistics gathered by the collector, the optimizer chooses a subplan.

   The dynamic plan coordinator decides which subplan to use at runtime for all such subplan groups. When notified of a new statistic value relevant to a subplan group, the coordinator dispatches it to the handler function for this subgroup.
4. The collector stops collecting statistics and buffering rows, permitting rows to pass through instead.
5. The database stores the adaptive plan in the child cursor, so that the *next* execution of the statement can use it.

On *subsequent* executions of the child cursor, the optimizer continues to use the same adaptive plan unless one of the following conditions is true, in which case it picks a new plan for the current execution:

- The current plan ages out of the shared pool.
- A different optimizer feature (for example, adaptive cursor sharing or statistics feedback) invalidates the current plan.

4.4.1.3.1 Adaptive Query Plans: Join Method Example

This example shows how the optimizer can choose a different plan based on information collected at runtime.

The following query shows a join of the `order_items` and `prod_info` tables.

```
SELECT product_name
FROM   order_items o, prod_info p
WHERE  o.unit_price = 15
AND    quantity > 1
AND    p.product_id = o.product_id
```

An adaptive query plan for this statement shows two possible plans, one with a nested loops join and the other with a hash join:

```
SELECT * FROM TABLE(DBMS_XPLAN.display_cursor(FORMAT => 'ADAPTIVE'));
```

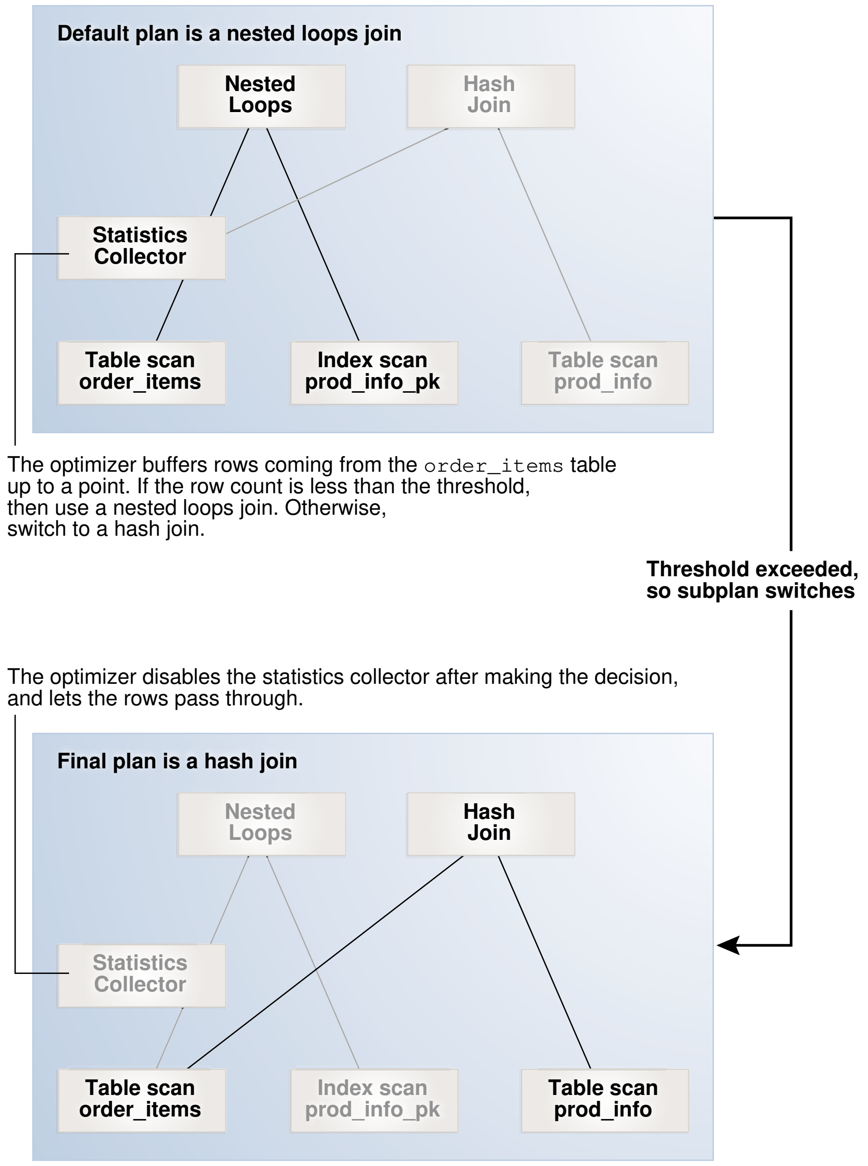
```
SQL_ID    7hj8dwwy6gm7p, child number 0
-------------------------------------
SELECT product_name FROM   order_items o, prod_info p WHERE
o.unit_price = 15 AND     quantity > 1 AND    p.product_id = o.product_id

Plan hash value: 1553478007

-----------------------------------------------------------------------------
| Id | Operation                     | Name       |Rows|Bytes|Cost (%CPU)|Time|
-----------------------------------------------------------------------------
|   0| SELECT STATEMENT              |              | |      |7(100)|        |
| * 1|  HASH JOIN                    |              |4| 128  | 7  (0)|00:00:01|
|-  2|   NESTED LOOPS                |              |4| 128  | 7  (0)|00:00:01|
|-  3|    NESTED LOOPS               |              |4| 128  | 7  (0)|00:00:01|
|-  4|     STATISTICS COLLECTOR      |              | |      |       |        |
| * 5|      TABLE ACCESS FULL        | ORDER_ITEMS  |4|  48  | 3  (0)|00:00:01|
|-* 6|     INDEX UNIQUE SCAN         | PROD_INFO_PK |1|      | 0  (0)|        |
|-  7|    TABLE ACCESS BY INDEX ROWID| PROD_INFO    |1|  20  | 1  (0)|00:00:01|
|   8|   TABLE ACCESS FULL           | PROD_INFO    |1|  20  | 1  (0)|00:00:01|
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
   5 - filter(("O"."UNIT_PRICE"=15 AND "QUANTITY">1))
   6 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

A nested loops join is preferable if the database can avoid scanning a significant portion of `prod_info` because its rows are filtered by the join predicate. If few rows are filtered, however, then scanning the right table in a hash join is preferable.

The following graphic shows the adaptive process. For the query in the preceding example, the adaptive portion of the default plan contains two subplans, each of which uses a different join method. The optimizer automatically determines when each join method is optimal, depending on the cardinality of the left side of the join.

The statistics collector buffers enough rows coming from the `order_items` table to determine which join method to use. If the row count is below the threshold determined by the optimizer, then the optimizer chooses the nested loops join; otherwise, the optimizer chooses the hash join. In this case, the row count coming from the `order_items` table is above the threshold, so the optimizer chooses a hash join for the final plan, and disables buffering.

Figure 4-8 Adaptive Join Methods


Default plan is a nested loops join
Nested Loops
Hash Join
Statistics Collector
Table scan order_items
Index scan prod_info_pk
Table scan prod_info
The optimizer buffers rows coming from the order_items table up to a point. If the row count is less than the threshold, then use a nested loops join. Otherwise, switch to a hash join.
Threshold exceeded, so subplan switches
The optimizer disables the statistics collector after making the decision, and lets the rows pass through.
Final plan is a hash join
Nested Loops
Hash Join
Statistics Collector
Table scan order_items
Index scan prod_info_pk
Table scan prod_info

The `Note` section of the execution plan indicates whether the plan is adaptive, and which rows in the plan are inactive.

See Also:

- "Controlling Adaptive Optimization"
- "Displaying Adaptive Query Plans: Tutorial" for an extended example showing an adaptive query plan

4.4.1.3.2 Adaptive Query Plans: Parallel Distribution Methods

Typically, parallel execution requires data redistribution to perform operations such as parallel sorts, aggregations, and joins.

Oracle Database can use many different data distributions methods. The database chooses the method based on the number of rows to be distributed and the number of parallel server processes in the operation.

For example, consider the following alternative cases:

- Many parallel server processes distribute few rows.

  The database may choose the broadcast distribution method. In this case, each parallel server process receives each row in the result set.
- Few parallel server processes distribute many rows.

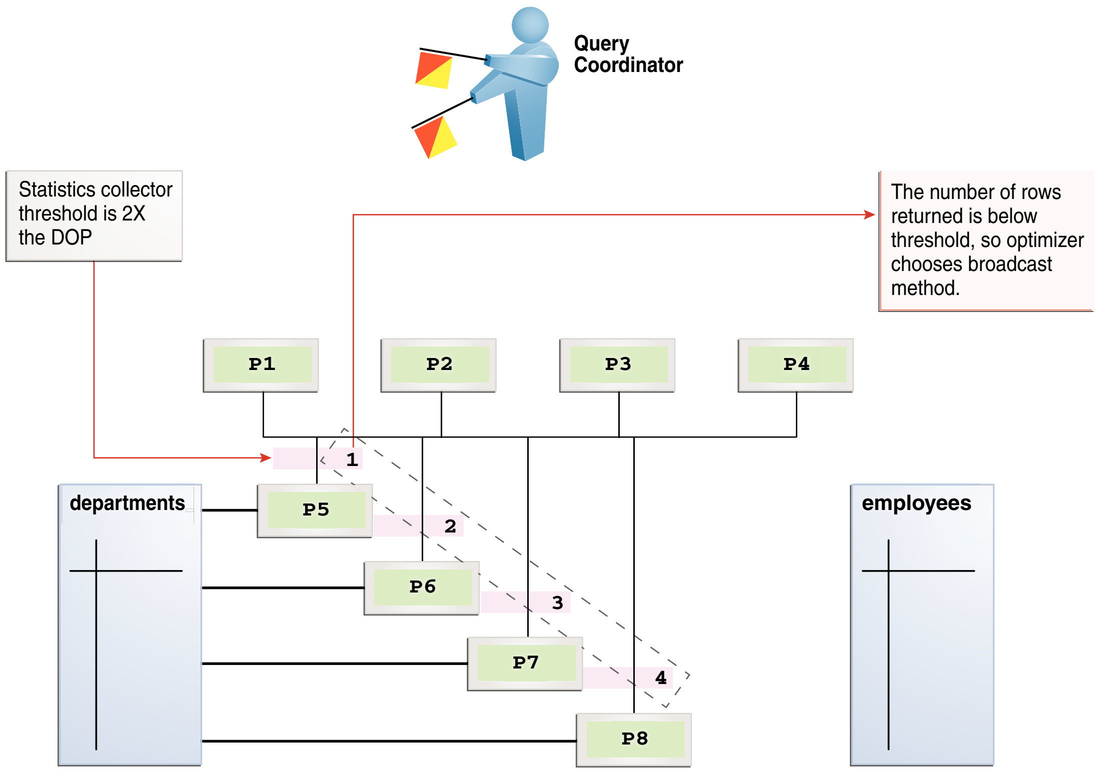
  If a data skew is encountered during the data redistribution, then it could adversely affect the performance of the statement. The database is more likely to pick a hash distribution to ensure that each parallel server process receives an equal number of rows.

The hybrid hash distribution technique is an adaptive parallel data distribution that does not decide the final data distribution method until execution time. The optimizer inserts statistic collectors in front of the parallel server processes on the producer side of the operation. If the number of rows is less than a threshold, defined as twice the degree of parallelism (DOP), then the data distribution method switches from hash to broadcast. Otherwise, the distribution method is a hash.

Broadcast Distribution

The following graphic depicts a hybrid hash join between the `departments` and `employees` tables, with a query coordinator directing 8 parallel server processes: P5-P8 are producers, whereas P1-P4 are consumers. Each producer has its own consumer.

Figure 4-9 Adaptive Query with DOP of 4


Query Coordinator
Statistics collector threshold is 2X the DOP
The number of rows returned is below threshold, so optimizer chooses broadcast method.
P1
P2
P3
P4
1
departments
P5
2
P6
3
P7
4
P8
employees

The database inserts a statistics collector in front of each producer process scanning the `departments` table. The query coordinator aggregates the collected statistics. The distribution method is based on the run-time statistics. In Figure 4-9, the number of rows is *below* the threshold (8), which is twice the DOP (4), so the optimizer chooses a broadcast technique for the `departments` table.

Hybrid Hash Distribution

Consider an example that returns a greater number of rows. In the following plan, the threshold is 8, or twice the specified DOP of 4. However, because the statistics collector (Step 10) discovers that the number of rows (27) is greater than the threshold (8), the optimizer chooses a hybrid hash distribution rather than a broadcast distribution.

Note:

The values for `Name` and `Time` are truncated in the following plan so that the lines can fit on the page.

```
EXPLAIN PLAN FOR
  SELECT /*+ parallel(4) full(e) full(d) */ department_name, sum(salary)
  FROM   employees e, departments d
  WHERE  d.department_id=e.department_id
  GROUP BY department_name;

-------------------------------------------------------------------------------------------
Plan hash value: 2940813933
-------------------------------------------------------------------------------------------
|Id|Operation                        |Name    |Rows|Bytes|Cost|Time| TQ  |IN-OUT|PQ Distrib|
-------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT                 |DEPARTME| 27|621  |6(34)|:01|     |    |           |
| 1| PX COORDINATOR                  |        |   |     |    |   |     |    |           |
| 2|  PX SEND QC (RANDOM)            |:TQ10003| 27|621  |6(34)|:01|Q1,03|P->S| QC (RAND) |
| 3|   HASH GROUP BY                 |        | 27|621  |6(34)|:01|Q1,03|PCWP|           |
| 4|    PX RECEIVE                   |        | 27|621  |6(34)|:01|Q1,03|PCWP|           |
| 5|     PX SEND HASH                |:TQ10002| 27|621  |6(34)|:01|Q1,02|P->P| HASH      |
| 6|      HASH GROUP BY              |        | 27|621  |6(34)|:01|Q1,02|PCWP|           |
|*7|       HASH JOIN                 |        |106|2438|5(20)|:01|Q1,02|PCWP|           |
| 8|        PX RECEIVE               |        | 27|432  |2  (0)|:01|Q1,02|PCWP|           |
| 9|         PX SEND HYBRID HASH     |:TQ10000| 27|432  |2  (0)|:01|Q1,00|P->P|HYBRID HASH|
|10|          STATISTICS COLLECTOR   |        |   |     |    |   |Q1,00|PCWC|           |
|11|           PX BLOCK ITERATOR     |        | 27|432  |2  (0)|:01|Q1,00|PCWC|           |
|12|            TABLE ACCESS FULL    |DEPARTME| 27|432  |2  (0)|:01|Q1,00|PCWP|           |
|13|        PX RECEIVE               |        |107|749  |2  (0)|:01|Q1,02|PCWP|           |
|14|         PX SEND HYBRID HASH (SKEW)|:TQ10001|107|749  |2  (0)|:01|Q1,01|P->P|HYBRID HASH|
|15|          PX BLOCK ITERATOR      |        |107|749  |2  (0)|:01|Q1,01|PCWC|           |
|16|           TABLE ACCESS FULL     |EMPLOYEE|107|749  |2  (0)|:01|Q1,01|PCWP|           |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   7 - access("D"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")

Note
-----
   - Degree of Parallelism is 4 because of hint

32 rows selected.
```

See Also:

Oracle Database VLDB and Partitioning Guide to learn more about parallel data redistribution techniques

4.4.1.3.3 Adaptive Query Plans: Bitmap Index Pruning

Adaptive plans prune indexes that do not significantly reduce the number of matched rows.

When the optimizer generates a star transformation plan, it must choose the right combination of bitmap indexes to reduce the relevant set of rowids as efficiently as possible. If many indexes exist, some indexes might not reduce the rowid set substantially, but nevertheless introduce significant processing cost during query execution. Adaptive plans can solve this problem by not using indexes that degrade performance.

Example 4-2 Bitmap Index Pruning

In this example, you issue the following star query, which joins the `cars` fact table with multiple dimension tables (sample output included):

```
SELECT /*+ star_transformation(r) */ l.color_name, k.make_name,
       h.filter_col, count(*)
FROM   cars r, colors l, makes k, models d, hcc_tab h
WHERE  r.make_id = k.make_id
AND    r.color_id = l.color_id
AND    r.model_id = d.model_id
AND    r.high_card_col = h.high_card_col
AND    d.model_name = 'RAV4'
AND    k.make_name = 'Toyota'
AND    l.color_name = 'Burgundy'
AND    h.filter_col = 100
GROUP BY l.color_name, k.make_name, h.filter_col;


COLOR_NA MAKE_N FILTER_COL   COUNT(*)
-------- ------ ---------- ----------
Burgundy Toyota        100      15000
```

The following sample execution plan shows that the query generated no rows for the bitmap node in Step 12 and Step 17. The adaptive optimizer determined that filtering rows by using the `CAR_MODEL_IDX` and `CAR_MAKE_IDX` indexes was inefficient. The query did not use the steps in the plan that begin with a dash (-).

```
-----------------------------------------------------------
| Id  | Operation                         | Name          |
-----------------------------------------------------------
|   0 | SELECT STATEMENT                  |               |
|   1 |  SORT GROUP BY NOSORT             |               |
|   2 |   HASH JOIN                       |               |
|   3 |    VIEW                           | VW_ST_5497B905 |
|   4 |     NESTED LOOPS                  |               |
|   5 |      BITMAP CONVERSION TO ROWIDS  |               |
|   6 |       BITMAP AND                  |               |
|   7 |        BITMAP MERGE               |               |
|   8 |         BITMAP KEY ITERATION      |               |
|   9 |          TABLE ACCESS FULL        | COLORS        |
|  10 |          BITMAP INDEX RANGE SCAN  | CAR_COLOR_IDX |
|- 11 |        STATISTICS COLLECTOR       |               |
|- 12 |         BITMAP MERGE              |               |
|- 13 |          BITMAP KEY ITERATION     |               |
|- 14 |           TABLE ACCESS FULL       | MODELS        |
|- 15 |           BITMAP INDEX RANGE SCAN | CAR_MODEL_IDX |
|- 16 |        STATISTICS COLLECTOR       |               |
|- 17 |         BITMAP MERGE              |               |
|- 18 |          BITMAP KEY ITERATION     |               |
|- 19 |           TABLE ACCESS FULL       | MAKES         |
|- 20 |           BITMAP INDEX RANGE SCAN | CAR_MAKE_IDX  |
|  21 |      TABLE ACCESS BY USER ROWID   | CARS          |
|  22 |    MERGE JOIN CARTESIAN           |               |
|  23 |     MERGE JOIN CARTESIAN          |               |
|  24 |      MERGE JOIN CARTESIAN         |               |
|  25 |       TABLE ACCESS FULL           | MAKES         |
|  26 |       BUFFER SORT                 |               |
|  27 |        TABLE ACCESS FULL          | MODELS        |
|  28 |      BUFFER SORT                  |               |
|  29 |       TABLE ACCESS FULL           | COLORS        |
|  30 |     BUFFER SORT                   |               |
|  31 |      TABLE ACCESS FULL            | HCC_TAB       |
-----------------------------------------------------------

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - star transformation used for this statement
   - this is an adaptive plan (rows marked '-' are inactive)
```

4.4.1.4 When Adaptive Query Plans Are Enabled

Adaptive query plans are enabled by default.

Adaptive plans are enabled when the following initialization parameters are set:

- `OPTIMIZER_ADAPTIVE_PLANS` is `TRUE` (default)
- `OPTIMIZER_FEATURES_ENABLE` is `12.1.0.1` or later
- `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` is `FALSE` (default)

Adaptive plans control the following optimizations:

- Nested loops and hash join selection
- Star transformation bitmap pruning
- Adaptive parallel distribution method

See Also:

- "Controlling Adaptive Optimization"
- *Oracle Database Reference* to learn more about `OPTIMIZER_ADAPTIVE_PLANS`

4.4.2 Adaptive Statistics

The optimizer can use **adaptive statistics** when query predicates are too complex to rely on base table statistics alone. By default, adaptive statistics are disabled (`OPTIMIZER_ADAPTIVE_STATISTICS` is `false`).

4.4.2.1 Dynamic Statistics

Dynamic statistics are an optimization technique in which the database executes a recursive SQL statement to scan a small random sample of a table's blocks to estimate predicate cardinalities.

During SQL compilation, the optimizer decides whether to use dynamic statistics by considering whether available statistics are sufficient to generate an optimal plan. If the available statistics are insufficient, then the optimizer uses dynamic statistics to augment the statistics. To improve the quality of optimizer decisions, the optimizer can use dynamic statistics for table scans, index access, joins, and `GROUP BY` operations.

4.4.2.2 Automatic Reoptimization

In **automatic reoptimization**, the optimizer changes a plan on subsequent executions *after* the initial execution.

Adaptive query plans are not feasible for all kinds of plan changes. For example, a query with an inefficient join order might perform suboptimally, but adaptive query plans do not support adapting the join order *during* execution. At the end of the first execution of a SQL statement, the optimizer uses the information gathered during execution to determine whether automatic reoptimization has a cost benefit. If execution information differs significantly from optimizer estimates, then the optimizer looks for a replacement plan on the next execution.

The optimizer uses the information gathered during the previous execution to help determine an alternative plan. The optimizer can reoptimize a query several times, each time gathering additional data and further improving the plan.

4.4.2.2.1 Reoptimization: Statistics Feedback

A form of reoptimization known as **statistics feedback** (formerly known as **cardinality feedback**) automatically improves plans for repeated queries that have cardinality misestimates.

The optimizer can estimate cardinalities incorrectly for many reasons, such as missing statistics, inaccurate statistics, or complex predicates. The basic process of reoptimization using statistics feedback is as follows:

1. During the first execution of a SQL statement, the optimizer generates an execution plan.

   The optimizer may enable monitoring for statistics feedback for the shared SQL area in the following cases:

   - Tables with no statistics
   - Multiple conjunctive or disjunctive filter predicates on a table
   - Predicates containing complex operators for which the optimizer cannot accurately compute selectivity estimates
2. At the end of the first execution, the optimizer compares its initial cardinality estimates to the actual number of rows returned by each operation in the plan during execution.

   If estimates differ significantly from actual cardinalities, then the optimizer stores the correct estimates for subsequent use. The optimizer also creates a SQL plan directive so that other SQL statements can benefit from the information obtained during this initial execution.
3. If the query executes again, then the optimizer uses the corrected cardinality estimates instead of its usual estimates.

The `OPTIMIZER_ADAPTIVE_STATISTICS` initialization parameter does not control all features of automatic reoptimization. Specifically, this parameter controls statistics feedback for join cardinality only in the context of automatic reoptimization. For example, setting `OPTIMIZER_ADAPTIVE_STATISTICS` to `FALSE` disables statistics feedback for join cardinality misestimates, but it does not disable statistics feedback for single-table cardinality misestimates.

Example 4-3 Statistics Feedback

This example shows how the database uses statistics feedback to adjust incorrect estimates.

1. The user `oe` runs the following query of the `orders`, `order_items`, and `product_information` tables:

```
SELECT o.order_id, v.product_name
FROM   orders o,
       ( SELECT order_id, product_name
         FROM   order_items o, product_information p
         WHERE  p.product_id = o.product_id
         AND    list_price < 50
         AND    min_price < 40 ) v
WHERE  o.order_id = v.order_id
```

2. Querying the plan in the cursor shows that the estimated rows (`E-Rows`) is far fewer than the actual rows (`A-Rows`).

```
-------------------------------------------------------------------------------------------------
| Id | Operation             | Name          |Starts|E-Rows|A-Rows|A-Time|Buffers|OMem|1Mem|O/1/M|
-------------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT      |                    |    1|      | 269 |00:00:00.14|1338|    |    |     |
| 1|  NESTED LOOPS         |                    |    1|    1 | 269 |00:00:00.14|1338|    |    |     |
| 2|   MERGE JOIN CARTESIAN|                    |    1|    4 |9135 |00:00:00.05|  33|    |    |     |
|*3|    TABLE ACCESS FULL  |PRODUCT_INFORMATION|    1|    1 |  87 |00:00:00.01|  32|    |    |     |
| 4|    BUFFER SORT        |                    |   87| 105 |9135 |00:00:00.02|   1|4096|4096|1/0/0|
| 5|     INDEX FULL SCAN   |ORDER_PK            |    1| 105 | 105 |00:00:00.01|   1|    |    |     |
|*6|   INDEX UNIQUE SCAN   |ORDER_ITEMS_UK      |9135|    1 | 269 |00:00:00.04|1305|    |    |     |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("MIN_PRICE"<40 AND "LIST_PRICE"<50))
   6 - access("O"."ORDER_ID"="ORDER_ID" AND "P"."PRODUCT_ID"="O"."PRODUCT_ID")
```

3. The user `oe` reruns the query in Step 1.

4. Querying the plan in the cursor shows that the optimizer used statistics feedback (shown in the `Note`) for the second execution, and also chose a different plan.

```
-------------------------------------------------------------------------------------------------
|Id | Operation              | Name    | Starts |E-Rows|A-Rows|A-Time|Buffers|Reads|OMem|1Mem|O/1/M|
-------------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT       |                    |  1|   | 269 |00:00:00.05|60|1|     |     |     |
| 1|  NESTED LOOPS          |                    |  1|269| 269 |00:00:00.05|60|1|     |     |     |
|*2|   HASH JOIN            |                    |  1|313| 269 |00:00:00.05|39|1|1398K|1398K|1/0/0|
|*3|    TABLE ACCESS FULL   |PRODUCT_INFORMATION|  1| 87|  87 |00:00:00.01|15|0|     |     |     |
| 4|    INDEX FAST FULL SCAN|ORDER_ITEMS_UK      |  1|665| 665 |00:00:00.01|24|1|     |     |     |
|*5|   INDEX UNIQUE SCAN    |ORDER_PK            |269|  1| 269 |00:00:00.01|21|0|     |     |     |
-------------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
   3 - filter(("MIN_PRICE"<40 AND "LIST_PRICE"<50))
   5 - access("O"."ORDER_ID"="ORDER_ID")

Note
-----
   - statistics feedback used for this statement
```

In the preceding output, the estimated number of rows (`269`) in Step 1 matches the actual number of rows.

4.4.2.2.2 Reoptimization: Performance Feedback

Another form of reoptimization is performance feedback. This reoptimization helps improve the degree of parallelism automatically chosen for repeated SQL statements when `PARALLEL_DEGREE_POLICY` is set to `ADAPTIVE`.

The basic process of reoptimization using performance feedback is as follows:

1. During the first execution of a SQL statement, when `PARALLEL_DEGREE_POLICY` is set to `ADAPTIVE`, the optimizer determines whether to execute the statement in parallel, and if so, which degree of parallelism to use.

   The optimizer chooses the degree of parallelism based on the estimated performance of the statement. Additional performance monitoring is enabled for all statements.

2. At the end of the initial execution, the optimizer compares the following:

   - The degree of parallelism chosen by the optimizer
   - The degree of parallelism computed based on the performance statistics (for example, the CPU time) gathered during the actual execution of the statement

   If the two values vary significantly, then the database marks the statement for reparsing, and stores the initial execution statistics as feedback. This feedback helps better compute the degree of parallelism for subsequent executions.

3. If the query executes again, then the optimizer uses the performance statistics gathered during the initial execution to better determine a degree of parallelism for the statement.

Note:

Even if `PARALLEL_DEGREE_POLICY` is not set to `ADAPTIVE`, statistics feedback may influence the degree of parallelism chosen for a statement.

4.4.2.3 SQL Plan Directives

A **SQL plan directive** is additional information that the optimizer uses to generate a more optimal plan.

The directive is a “note to self” by the optimizer that it is misestimating cardinalities of certain types of predicates, and also a reminder to `DBMS_STATS` to gather statistics needed to correct the misestimates in the future.

For example, during query optimization, when deciding whether the table is a candidate for dynamic statistics, the database queries the statistics repository for directives on a table. If the query joins two tables that have a data skew in their join columns, then a SQL plan directive can direct the optimizer to use dynamic statistics to obtain an accurate cardinality estimate.

The optimizer collects SQL plan directives on query expressions rather than at the statement level so that it can apply directives to multiple SQL statements. The optimizer not only corrects itself, but also records information about the mistake, so that the database can continue to correct its estimates even after a query—and any similar query—is flushed from the shared pool.

The database automatically creates directives, and stores them in the `SYSAUX` tablespace. You can alter, save to disk, and transport directives using the PL/SQL package `DBMS_SPD`.

See Also:

- "SQL Plan Directives"
- "Managing SQL Plan Directives"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPD` package

4.4.2.4 When Adaptive Statistics Are Enabled

Adaptive statistics are disabled by default.

Adaptive statistics are enabled when the following initialization parameters are set:

- `OPTIMIZER_ADAPTIVE_STATISTICS` is `TRUE` (the default is `FALSE`)
- `OPTIMIZER_FEATURES_ENABLE` is `12.1.0.1` or later

Setting `OPTIMIZER_ADAPTIVE_STATISTICS` to `TRUE` enables the following features:

- SQL plan directives
- Statistics feedback for join cardinality
- Adaptive dynamic sampling

Note:

Setting `OPTIMIZER_ADAPTIVE_STATISTICS` to `FALSE` preserves statistics feedback for single-table cardinality misestimates.

See Also:

- "Controlling Adaptive Optimization"
- *Oracle Database Reference* to learn more about `OPTIMIZER_ADAPTIVE_STATISTICS`

4.5 About Approximate Query Processing

Approximate query processing is a set of optimization techniques that speed analytic queries by calculating results within an acceptable range of error.

Business intelligence (BI) queries heavily rely on sorts that involve aggregate functions such as `COUNT DISTINCT`, `SUM`, `RANK`, and `MEDIAN`. For example, an application generates reports showing how many distinct customers are logged on, or which products were most popular last week. It is not uncommon for BI applications to have the following requirements:

- Queries must be able to process data sets that are orders of magnitude larger than in traditional data warehouses.

  For example, the daily volumes of web logs of a popular website can reach tens or hundreds of terabytes a day.
- Queries must provide near real-time response.

  For example, a company requires quick detection and response to credit card fraud.
- Explorative queries of large data sets must be fast.

  For example, a user might want to find out a list of departments whose sales have approximately reached a specific threshold. A user would form targeted queries on these departments to find more detailed information, such as the exact sales number, the locations of these departments, and so on.

For large data sets, exact aggregation queries consume extensive memory, often spilling to temp space, and can be unacceptably slow. Applications are often more interested in a *general* pattern than *exact* results, so customers are willing to sacrifice exactitude for speed. For example, if the goal is to show a bar chart depicting the most popular products, then whether a product sold 1 million units or .999 million units is statistically insignificant.

Oracle Database implements its solution through approximate query processing. Typically, the accuracy of the approximate aggregation is over 97% (with 95% confidence), but the processing time is orders of magnitude faster. The database uses less CPU, and avoids the I/O cost of writing to temp files.

See Also:

"NDV Algorithms: Adaptive Sampling and HyperLogLog"

4.5.1 Approximate Query Initialization Parameters

You can implement approximate query processing without changing existing code by using the `APPROX_FOR_*` initialization parameters.

Set these parameters at the database or session level. The following table describes initialization parameters and SQL functions relevant to approximation techniques.

Table 4-2 Approximate Query Initialization Parameters

Initialization Parameter	Default	Description	See Also
`APPROX_FOR_AGGREGATION`	`FALSE`	Enables (`TRUE`) or disables (`FALSE`) approximate query processing. This parameter acts as an umbrella parameter for enabling the use of functions that return approximate results.	*Oracle Database Reference*
`APPROX_FOR_COUNT_DISTINCT`	`FALSE`	Converts `COUNT(DISTINCT)` to `APPROX_COUNT_DISTINCT`.	*Oracle Database Reference*
`APPROX_FOR_PERCENTILE`	`none`	Converts eligible exact percentile functions to their `APPROX_PERCENTILE_*` counterparts.	*Oracle Database Reference*

See Also:

- "About Optimizer Initialization Parameters"
- *Oracle Database Data Warehousing Guide* to learn more about approximate query processing

4.5.2 Approximate Query SQL Functions

Approximate query processing uses SQL functions to provide real-time responses to explorative queries where approximations are acceptable.

The following table describes SQL functions that return approximate results.

Table 4-3 Approximate Query User Interface

SQL Function	Description	See Also

`APPROX_COUNT`	Calculates the approximate top *n* most common values when used with the `APPROX_RANK` function.
	Returns the approximate count of an expression. If you supply `MAX_ERROR` as the second argument, then the function returns the maximum error between the actual and approximate count.
	You must use this function with a corresponding `APPROX_RANK` function in the `HAVING` clause. If a query uses `APPROX_COUNT`, `APPROX_SUM`, or `APPROX_RANK`, then the query must not use any other non-approximate aggregation functions.
	The following query returns the 10 most common jobs within every department:
	`SELECT department_id, job_id,`
	`       APPROX_COUNT(*)`
	`FROM   employees`
	`GROUP BY department_id, job_id`
	`HAVING`
	`  APPROX_RANK (`
	`  PARTITION BY department_id`
	`  ORDER BY APPROX_COUNT(*)`
	`  DESC ) <= 10;`
		Oracle Database SQL Language Reference

`APPROX_COUNT_DISTINCT`	Returns the approximate number of rows that contain distinct values of an expression.	*Oracle Database SQL Language Reference*
`APPROX_COUNT_DISTINCT_AGG`	Aggregates the precomputed approximate count distinct synopses to a higher level.	*Oracle Database SQL Language Reference*
`APPROX_COUNT_DISTINCT_DETAIL`	Returns the synopses of the `APPROX_COUNT_DISTINCT` function as a BLOB. The database can persist the returned result to disk for further aggregation.	*Oracle Database SQL Language Reference*
`APPROX_MEDIAN`	Accepts a numeric or date-time value, and returns an approximate middle or approximate interpolated value that would be the middle value when the values are sorted. This function provides an alternative to the `MEDIAN` function.	*Oracle Database SQL Language Reference*
`APPROX_PERCENTILE`	Accepts a percentile value and a sort specification, and returns an approximate interpolated value that falls into that percentile value with respect to the sort specification. This function provides an alternative to the `PERCENTILE_CONT` function.	*Oracle Database SQL Language Reference*

Table 4-3 (Cont.) Approximate Query User Interface

SQL Function	Description	See Also
`APPROX_RANK`	Returns the approximate value in a group of values. This function takes an optional `PARTITION BY` clause followed by a mandatory `ORDER BY ... DESC` clause. The `PARTITION BY` key must be a subset of the `GROUP BY` key. The `ORDER BY` clause must include either `APPROX_COUNT` or `APPROX_SUM`.	*Oracle Database SQL Language Reference*

`APPROX_SUM`	Calculates the approximate top *n* accumulated values when used with the `APPROX_RANK` function.
	If you supply `MAX_ERROR` as the second argument, then the function returns the maximum error between the actual and approximate sum.
	You must use this function with a corresponding `APPROX_RANK` function in the `HAVING` clause. If a query uses `APPROX_COUNT`, `APPROX_SUM`, or `APPROX_RANK`, then the query must not use any other non-approximate aggregation functions.
	The following query returns the 10 job types within every department that have the highest aggregate salary:
	`SELECT department_id, job_id,`
	`       APPROX_SUM(salary)`
	`FROM   employees`
	`GROUP BY department_id, job_id`
	`HAVING`
	`  APPROX_RANK (`
	`  PARTITION BY department_id`
	`  ORDER BY APPROX_SUM(salary)`
	`  DESC ) <= 10;`
	Note that `APPROX_SUM` returns an error when the input is a negative number.
		Oracle Database SQL Language Reference

See Also:

Oracle Database Data Warehousing Guide to learn more about approximate query processing

4.6 About SQL Plan Management

SQL plan management enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans.

SQL plan management can build a SQL plan baseline, which contains one or more accepted plans for each SQL statement. The optimizer can access and manage the plan history and SQL plan baselines of SQL statements. The main objectives are as follows:

- Identify repeatable SQL statements
- Maintain plan history, and possibly SQL plan baselines, for a set of SQL statements
- Detect plans that are not in the plan history
- Detect potentially better plans that are not in the SQL plan baseline

The optimizer uses the normal cost-based search method.

See Also:

- "Managing SQL Plan Baselines"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package

4.7 About Quarantined SQL Plans

You can configure Oracle Database to automatically quarantine the plans for SQL statements terminated by Oracle Database Resource Manager (the Resource Manager) for exceeding resource limits.

How SQL Quarantine Works

The Resource Manager can set a maximum estimated execution time for a SQL statement, for example, 20 minutes. If a statement execution exceeds this limit, then the Resource Manager terminates the statement. However, the statement may run repeatedly before being terminated, wasting 20 minutes of resources each time it is executed.

The SQL Quarantine infrastructure (SQL Quarantine) solves the problem of repeatedly wasting resources. If a statement exceeds the specified resource limit, then the Resource Manager terminates the execution and “quarantines” the plan. To quarantine the plan means to put it on a blocklist of plans that the database will not execute for this statement. Note that the *plan* for a terminated statement is quarantined, not the statement itself.

The query in our example runs for 20 minutes only once, and then never again—unless the resource limit increases or the plan changes. If the limit is increased to 25 minutes, then the Resource Manager permits the statement to run again with the quarantined plan. If the statement runs for 23 minutes, which is below the new threshold, then the Resource Manager removes the plan from quarantine. If the statement runs for 26 minutes, which is above the new threshold, then the plan remains quarantined unless the limit is increased.

SQL Quarantine User Interface

The `DBMS_SQLQ` PL/SQL package enables you to manually create quarantine configurations for execution plans by specifying thresholds for consuming system resources. For example, you can enable a quarantine threshold of 10 seconds for CPU time or drop the threshold for I/O requests. You can also immediately save the quarantine information to disk or drop configurations.

To enable SQL Quarantine to create configurations automatically after the Resource Manager terminates a query, set the `OPTIMIZER_CAPTURE_SQL_QUARANTINE` initialization parameter to `true` (the default is `false`). To disable the use of existing SQL Quarantine configurations, set `OPTIMIZER_USE_SQL_QUARANTINE` to `false` (the default is `true`).

The `V$SQL.SQL_QUARANTINE` column indicates whether a plan was quarantined for a statement after the Resource Manager canceled execution. The `AVOIDED_EXECUTIONS` column indicates how often Oracle Database prevented the statement from running with the quarantined plan.

See Also:

- *Oracle Database Administrator's Guide* to learn how to configure SQL Quarantine and use `DBMS_SQLQ`
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_SQLQ`
- *Oracle Database Reference* to learn about `OPTIMIZER_CAPTURE_SQL_QUARANTINE`
- *Oracle Database Reference* to learn about `OPTIMIZER_USE_SQL_QUARANTINE`
- *Oracle Database Reference* to learn about `V$SQL`
- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

4.8 About the Expression Statistics Store (ESS)

The **Expression Statistics Store (ESS)** is a repository maintained by the optimizer to store statistics about expression evaluation.

When an IM column store is enabled, the database leverages the ESS for its In-Memory Expressions (IM expressions) feature. However, the ESS is independent of the IM column store. The ESS is a permanent component of the database and cannot be disabled.

The database uses the ESS to determine whether an expression is "hot" (frequently accessed), and thus a candidate for an IM expression. During a hard parse of a query, the ESS looks for active expressions in the `SELECT` list, `WHERE` clause, `GROUP BY` clause, and so on.

For each segment, the ESS maintains expression statistics such as the following:

- Frequency of execution
- Cost of evaluation
- Timestamp evaluation

The optimizer assigns each expression a weighted score based on cost and the number of times it was evaluated. The values are approximate rather than exact. More active expressions have higher scores. The ESS maintains an internal list of the most frequently accessed expressions.

The ESS resides in the SGA and also persists on disk. The database saves the statistics to disk every 15 minutes, or immediately using the

`DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` procedure. The ESS statistics are visible in the `DBA_EXPRESSION_STATISTICS` view.

See Also:

- *Oracle Database In-Memory Guide* to learn more about the ESS
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO`

5

Query Transformations

This chapter describes the most important optimizer techniques for transforming queries.

The optimizer decides whether to use an available transformation based on cost. Transformations may not be available to the optimizer for a variety of reasons, including hints or lack of constraints. For example, transformations such as subquery unnesting are not available for hybrid partitioned tables, which contain external partitions that do not support constraints.

5.1 OR Expansion

In `OR` expansion, the optimizer transforms a query block containing top-level disjunctions into the form of a `UNION ALL` query that contains two or more branches.

The optimizer achieves this goal by splitting the disjunction into its components, and then associating each component with a branch of a `UNION ALL` query. The optimizer can choose `OR` expansion for various reasons. For example, it may enable more efficient access paths or alternative join methods that avoid Cartesian products. As always, the optimizer performs the expansion only if the cost of the transformed statement is lower than the cost of the original statement.

In previous releases, the optimizer used the `CONCATENATION` operator to perform the `OR` expansion. Starting in Oracle Database 12c Release 2 (12.2), the optimizer uses the `UNION-ALL` operator instead. The framework provides the following enhancements:

- Enables interaction among various transformations
- Avoids sharing query structures
- Enables the exploration of various search strategies
- Provides the reuse of cost annotation
- Supports the standard SQL syntax

Example 5-1 Transformed Query: UNION ALL Condition

To prepare for this example, log in to the database as an administrator, execute the following statements to add a unique constraint to the `hr.departments.department_name` column, and then add 100,000 rows to the `hr.employees` table:

```
ALTER TABLE hr.departments ADD CONSTRAINT department_name_uk UNIQUE
(department_name);
DELETE FROM hr.employees WHERE employee_id > 999;
DECLARE
v_counter NUMBER(7) := 1000;
BEGIN
 FOR i IN 1..100000 LOOP
 INSERT INTO hr.employees
    VALUES (v_counter,null,'Doe','Doe' || v_counter ||
'@example.com',null,'07-JUN-02','AC_ACCOUNT',null,null,null,50);
```

```
 v_counter := v_counter + 1;
 END LOOP;
END;
/
COMMIT;
EXEC DBMS_STATS.GATHER_TABLE_STATS ( ownname => 'hr', tabname => 
'employees');
```

You then connect as the user `hr`, and execute the following query, which joins the `employees` and `departments` tables:

```
SELECT *
FROM   employees e, departments d
WHERE  (e.email='SSTILES' OR d.department_name='Treasury')
AND    e.department_id = d.department_id;
```

Without `OR` expansion, the optimizer treats `e.email='SSTILES' OR d.department_name='Treasury'` as a single unit. Consequently, the optimizer cannot use the index on either the `e.email` or `d.department_name` column, and so performs a full table scan of `employees` and `departments`.

With `OR` expansion, the optimizer breaks the disjunctive predicate into two independent predicates, as shown in the following example:

```
SELECT *
FROM   employees e, departments d
WHERE  e.email = 'SSTILES'
AND    e.department_id = d.department_id
UNION ALL
SELECT *
FROM   employees e, departments d
WHERE  d.department_name = 'Treasury'
AND    e.department_id = d.department_id;
```

This transformation enables the `e.email` and `d.department_name` columns to serve as index keys. Performance improves because the database filters data using two unique indexes instead of two full table scans, as shown in the following execution plan:

```
Plan hash value: 2512933241

-------------------------------------------------------------------------------------
-----
| Id| Operation                          | Name            |Rows|Bytes|Cost(%CPU)|
Time    |
-------------------------------------------------------------------------------------
-----
| 0 |SELECT STATEMENT                    |                  |    |     |      
122(100)|        |
| 1 | VIEW                               |VW_ORE_19FF4E3E   |9102|1679K|122  (5)  |
00:00:01|
| 2 |  UNION-ALL                         |                  |    |     |
|        |
| 3 |   NESTED LOOPS                     |                  |  1 |  78 |   4 (0)  |
00:00:01|
| 4 |     TABLE ACCESS BY INDEX ROWID       | EMPLOYEES         |  1 |  57 |  3 (0) |00:00:01|
|*5 |      INDEX UNIQUE SCAN                | EMP_EMAIL_UK      |  1 |     |  2 (0) |00:00:01|
| 6 |     TABLE ACCESS BY INDEX ROWID       | DEPARTMENTS       |  1 |  21 |  1 (0) |00:00:01|
|*7 |      INDEX UNIQUE SCAN                | DEPT_ID_PK        |  1 |     |  0 (0) |        |
| 8 |    NESTED LOOPS                       |                   |9101| 693K|118 (5) |00:00:01|
| 9 |     TABLE ACCESS BY INDEX ROWID       | DEPARTMENTS       |  1 |  21 |  1 (0) |00:00:01|
|*10|      INDEX UNIQUE SCAN                |DEPARTMENT_NAME_UK|  1 |     |  0 (0) |        |
|*11|     TABLE ACCESS BY INDEX ROWID BATCH| EMPLOYEES         |9101| 506K|117 (5) |00:00:01|
|*12|      INDEX RANGE SCAN                 |EMP_DEPARTMENT_IX |9101|     | 35 (6) |00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("E"."EMAIL"='SSTILES')
   7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
  10 - access("D"."DEPARTMENT_NAME"='Treasury')
  11 - filter(LNNVL("E"."EMAIL"='SSTILES'))
  12 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

35 rows selected.
```

5.2 View Merging

In **view merging**, the optimizer merges the **query block** representing a view into the query block that contains it.

View merging can improve plans by enabling the optimizer to consider additional join orders, access methods, and other transformations. For example, after a view has been merged and several tables reside in one query block, a table inside a view may permit the optimizer to use join elimination to remove a table outside the view.

For certain simple views in which merging always leads to a better plan, the optimizer automatically merges the view without considering cost. Otherwise, the optimizer uses cost to make the determination. The optimizer may choose not to merge a view for many reasons, including cost or validity restrictions.

If `OPTIMIZER_SECURE_VIEW_MERGING` is `true` (default), then Oracle Database performs checks to ensure that view merging and predicate pushing do not violate the security intentions of the view creator. To disable these additional security checks for a specific view, you can grant the `MERGE VIEW` privilege to a user for this view. To disable additional security checks for all views for a specific user, you can grant the `MERGE ANY VIEW` privilege to that user.

Note:

You can use hints to override view merging rejected because of cost or heuristics, but not validity.

See Also:

- *Oracle Database SQL Language Reference* for more information about the `MERGE ANY VIEW` and `MERGE VIEW` privileges
- *Oracle Database Reference* for more information about the `OPTIMIZER_SECURE_VIEW_MERGING` initialization parameter

5.2.1 Query Blocks in View Merging

The optimizer represents each nested **subquery** or unmerged view by a separate query block.

The database optimizes query blocks separately from the bottom up. Thus, the database optimizes the innermost query block first, generates the part of the plan for it, and then generates the plan for the outer query block, representing the entire query.

The parser expands each view referenced in a query into a separate query block. The block essentially represents the view definition, and thus the result of a view. One option for the optimizer is to analyze the view query block separately, generate a view subplan, and then process the rest of the query by using the view subplan to generate an overall execution plan. However, this technique may lead to a suboptimal execution plan because the view is optimized separately.

View merging can sometimes improve performance. As shown in "Example 5-2", view merging merges the tables from the view into the outer query block, removing the inner query block. Thus, separate optimization of the view is not necessary.

5.2.2 Simple View Merging

In **simple view merging**, the optimizer merges select-project-join views.

For example, a query of the `employees` table contains a subquery that joins the `departments` and `locations` tables.

Simple view merging frequently results in a more optimal plan because of the additional join orders and access paths available after the merge. A view may not be valid for simple view merging because:

- The view contains constructs not included in select-project-join views, including:
  - `GROUP BY`
  - `DISTINCT`
  - Outer join
  - `MODEL`
  - `CONNECT BY`
  - Set operators
  - Aggregation
- The view appears on the right side of a semijoin or antijoin.
- The view contains subqueries in the `SELECT` list.

- The outer query block contains PL/SQL functions.
- The view participates in an outer join, and does not meet one of the several additional validity requirements that determine whether the view can be merged.

Example 5-2 Simple View Merging

The following query joins the `hr.employees` table with the `dept_locs_v` view, which returns the street address for each department. `dept_locs_v` is a join of the `departments` and `locations` tables.

```
SELECT e.first_name, e.last_name, dept_locs_v.street_address,
       dept_locs_v.postal_code
FROM   employees e,
      ( SELECT d.department_id, d.department_name,
               l.street_address, l.postal_code
        FROM   departments d, locations l
        WHERE  d.location_id = l.location_id ) dept_locs_v
WHERE  dept_locs_v.department_id = e.department_id
AND    e.last_name = 'Smith';
```

The database can execute the preceding query by joining `departments` and `locations` to generate the rows of the view, and then joining this result to `employees`. Because the query contains the view `dept_locs_v`, and this view contains two tables, the optimizer must use one of the following join orders:

- `employees`, `dept_locs_v` (`departments`, `locations`)
- `employees`, `dept_locs_v` (`locations`, `departments`)
- `dept_locs_v` (`departments`, `locations`), `employees`
- `dept_locs_v` (`locations`, `departments`), `employees`

Join methods are also constrained. The index-based nested loops join is not feasible for join orders that begin with `employees` because no index exists on the column from this view. Without view merging, the optimizer generates the following execution plan:

```
-----------------------------------------------------------------
| Id  | Operation                    | Name        | Cost (%CPU)|
-----------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |     7  (15)|
|*  1 |  HASH JOIN                   |             |     7  (15)|
|   2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |     2   (0)|
|*  3 |    INDEX RANGE SCAN          | EMP_NAME_IX |     1   (0)|
|   4 |   VIEW                       |             |     5  (20)|
|*  5 |    HASH JOIN                 |             |     5  (20)|
|   6 |     TABLE ACCESS FULL        | LOCATIONS   |     2   (0)|
|   7 |     TABLE ACCESS FULL        | DEPARTMENTS |     2   (0)|
-----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
1 - access("DEPT_LOCS_V"."DEPARTMENT_ID"="E"."DEPARTMENT_ID")
3 - access("E"."LAST_NAME"='Smith')
5 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
```

View merging merges the tables from the view into the outer query block, removing the inner query block. After view merging, the query is as follows:

```
SELECT e.first_name, e.last_name, l.street_address, l.postal_code
FROM   employees e, departments d, locations l
WHERE  d.location_id = l.location_id
AND    d.department_id = e.department_id
AND    e.last_name = 'Smith';
```

Because all three tables appear in one query block, the optimizer can choose from the following six join orders:

- `employees`, `departments`, `locations`
- `employees`, `locations`, `departments`
- `departments`, `employees`, `locations`
- `departments`, `locations`, `employees`
- `locations`, `employees`, `departments`
- `locations`, `departments`, `employees`

The joins to `employees` and `departments` can now be index-based. After view merging, the optimizer chooses the following more efficient plan, which uses nested loops:

```
-------------------------------------------------------------------
| Id  | Operation                       | Name          | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT                |               |     4   (0)|
|   1 |  NESTED LOOPS                   |               |            |
|   2 |   NESTED LOOPS                  |               |     4   (0)|
|   3 |    NESTED LOOPS                 |               |     3   (0)|
|   4 |     TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |     2   (0)|
|*  5 |      INDEX RANGE SCAN          | EMP_NAME_IX   |     1   (0)|
|   6 |     TABLE ACCESS BY INDEX ROWID| DEPARTMENTS   |     1   (0)|
|*  7 |      INDEX UNIQUE SCAN         | DEPT_ID_PK    |     0   (0)|
|*  8 |    INDEX UNIQUE SCAN           | LOC_ID_PK     |     0   (0)|
|   9 |   TABLE ACCESS BY INDEX ROWID  | LOCATIONS     |     1   (0)|
-------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
 5 - access("E"."LAST_NAME"='Smith')
 7 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
 8 - access("D"."LOCATION_ID"="L"."LOCATION_ID")
```

See Also:

The Oracle Optimizer blog at `https://blogs.oracle.com/optimizer/` to learn about outer join view merging, which is a special case of simple view merging

5.2.3 Complex View Merging

In **view merging**, the optimizer merges views containing `GROUP BY` and `DISTINCT` views. Like simple view merging, complex merging enables the optimizer to consider additional join orders and access paths.

The optimizer can delay evaluation of `GROUP BY` or `DISTINCT` operations until after it has evaluated the joins. Delaying these operations can improve or worsen performance depending on the data characteristics. If the joins use filters, then delaying the operation until after joins can reduce the data set on which the operation is to be performed. Evaluating the operation early can reduce the amount of data to be processed by subsequent joins, or the joins could increase the amount of data to be processed by the operation. The optimizer uses cost to evaluate view merging and merges the view only when it is the lower cost option.

Aside from cost, the optimizer may be unable to perform complex view merging for the following reasons:

- The outer query tables do not have a rowid or unique column.
- The view appears in a `CONNECT BY` query block.
- The view contains `GROUPING SETS`, `ROLLUP`, or `PIVOT` clauses.
- The view or outer query block contains the `MODEL` clause.

Example 5-3 Complex View Joins with GROUP BY

The following view uses a `GROUP BY` clause:

```
CREATE VIEW cust_prod_totals_v AS
SELECT SUM(s.quantity_sold) total, s.cust_id, s.prod_id
FROM   sales s
GROUP BY s.cust_id, s.prod_id;
```

The following query finds all of the customers from the United States who have bought at least 100 fur-trimmed sweaters:

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name, c.cust_email
FROM   customers c, products p, cust_prod_totals_v
WHERE  c.country_id = 52790
AND    c.cust_id = cust_prod_totals_v.cust_id
AND    cust_prod_totals_v.total > 100
AND    cust_prod_totals_v.prod_id = p.prod_id
AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater';
```

The `cust_prod_totals_v` view is eligible for complex view merging. After merging, the query is as follows:

```
SELECT c.cust_id, cust_first_name, cust_last_name, cust_email
FROM   customers c, products p, sales s
WHERE  c.country_id = 52790
AND    c.cust_id = s.cust_id
AND    s.prod_id = p.prod_id
AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater'
GROUP BY s.cust_id, s.prod_id, p.rowid, c.rowid, c.cust_email,
c.cust_last_name,
         c.cust_first_name, c.cust_id
HAVING SUM(s.quantity_sold) > 100;
```

The transformed query is cheaper than the untransformed query, so the optimizer chooses to merge the view. In the untransformed query, the `GROUP BY` operator applies to the entire `sales` table in the view. In the transformed query, the joins to `products` and `customers` filter out a large portion of the rows from the `sales` table, so the `GROUP BY` operation is lower cost. The join is more expensive because the `sales` table has not been reduced, but it is not much more expensive because the `GROUP BY` operation does not reduce the size of the row set very much in the original query. If any of the preceding characteristics were to change, merging the view might no longer be lower cost. The final plan, which does not include a view, is as follows:

```
--------------------------------------------------------
| Id  | Operation             | Name      | Cost (%CPU)|
--------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  2101  (18)|
|*  1 |  FILTER               |           |            |
|   2 |   HASH GROUP BY       |           |  2101  (18)|
|*  3 |    HASH JOIN          |           |  2099  (18)|
|*  4 |     HASH JOIN         |           |  1801  (19)|
|*  5 |      TABLE ACCESS FULL| PRODUCTS  |    96   (5)|
|   6 |      TABLE ACCESS FULL| SALES     |  1620  (15)|
|*  7 |     TABLE ACCESS FULL | CUSTOMERS |   296  (11)|
--------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------
1 - filter(SUM("QUANTITY_SOLD")>100)
3 - access("C"."CUST_ID"="CUST_ID")
4 - access("PROD_ID"="P"."PROD_ID")
5 - filter("P"."PROD_NAME"='T3 Faux Fur-Trimmed Sweater')
7 - filter("C"."COUNTRY_ID"='US')
```

Example 5-4 Complex View Joins with DISTINCT

The following query of the `cust_prod_v` view uses a `DISTINCT` operator:

```
SELECT c.cust_id, c.cust_first_name, c.cust_last_name, c.cust_email
FROM   customers c, products p,
       ( SELECT DISTINCT s.cust_id, s.prod_id
         FROM   sales s) cust_prod_v
WHERE  c.country_id = 52790
AND    c.cust_id = cust_prod_v.cust_id
AND    cust_prod_v.prod_id = p.prod_id
AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater';
```

After determining that view merging produces a lower-cost plan, the optimizer rewrites the query into this equivalent query:

```
SELECT nwvw.cust_id, nwvw.cust_first_name, nwvw.cust_last_name,
nwvw.cust_email
FROM   ( SELECT DISTINCT(c.rowid), p.rowid, s.prod_id, s.cust_id,
                c.cust_first_name, c.cust_last_name, c.cust_email
         FROM   customers c, products p, sales s
         WHERE  c.country_id = 52790
         AND    c.cust_id = s.cust_id
         AND    s.prod_id = p.prod_id
         AND    p.prod_name = 'T3 Faux Fur-Trimmed Sweater' ) nwvw;
```

The plan for the preceding query is as follows:

```
-------------------------------------------
| Id  | Operation             | Name      |
-------------------------------------------
|   0 | SELECT STATEMENT      |           |
|   1 |  VIEW                 | VM_NWVW_1 |
|   2 |   HASH UNIQUE         |           |
|*  3 |    HASH JOIN          |           |
|*  4 |     HASH JOIN         |           |
|*  5 |      TABLE ACCESS FULL| PRODUCTS  |
|   6 |      TABLE ACCESS FULL| SALES     |
|*  7 |     TABLE ACCESS FULL | CUSTOMERS |
-------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
  3 - access("C"."CUST_ID"="S"."CUST_ID")
  4 - access("S"."PROD_ID"="P"."PROD_ID")
  5 - filter("P"."PROD_NAME"='T3 Faux Fur-Trimmed Sweater')
  7 - filter("C"."COUNTRY_ID"='US')
```

The preceding plan contains a view named `vm_nwvw_1`, known as a projection view, even after view merging has occurred. Projection views appear in queries in which a `DISTINCT` view has been merged, or a `GROUP BY` view is merged into an outer query block that also contains `GROUP BY`, `HAVING`, or aggregates. In the latter case, the projection view contains the `GROUP BY`, `HAVING`, and aggregates from the original outer query block.

In the preceding example of a projection view, when the optimizer merges the view, it moves the `DISTINCT` operator to the outer query block, and then adds several additional columns to maintain semantic equivalence with the original query. Afterward, the query can select only the desired columns in the `SELECT` list of the outer query block. The optimization retains all of the benefits of view merging: all tables are in one query block, the optimizer can permute them as needed in the final join order, and the `DISTINCT` operation has been delayed until after all of the joins complete.

5.3 Predicate Pushing

In **predicate pushing**, the optimizer "pushes" the relevant predicates from the containing query block into the view query block.

For views that are not merged, this technique improves the subplan of the unmerged view. The database can use the pushed-in predicates to access indexes or to use as filters.

For example, suppose you create a table `hr.contract_workers` as follows:

```
DROP TABLE contract_workers;
CREATE TABLE contract_workers AS (SELECT * FROM employees where 1=2);
INSERT INTO contract_workers VALUES (306, 'Bill', 'Jones', 'BJONES',
  '555.555.2000', '07-JUN-02', 'AC_ACCOUNT', 8300, 0,205, 110);
INSERT INTO contract_workers VALUES (406, 'Jill', 'Ashworth',
'JASHWORTH',
  '555.999.8181', '09-JUN-05', 'AC_ACCOUNT', 8300, 0,205, 50);
INSERT INTO contract_workers VALUES (506, 'Marcie', 'Lunsford',
  'MLUNSFORD', '555.888.2233', '22-JUL-01', 'AC_ACCOUNT', 8300,
  0, 205, 110);
COMMIT;
CREATE INDEX contract_workers_index ON contract_workers(department_id);
```

You create a view that references `employees` and `contract_workers`. The view is defined with a query that uses the `UNION` set operator, as follows:

```
CREATE VIEW all_employees_vw AS
  ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
    FROM   employees )
  UNION
  ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
    FROM   contract_workers );
```

You then query the view as follows:

```
SELECT last_name
FROM   all_employees_vw
WHERE  department_id = 50;
```

Because the view is a `UNION` set query, the optimizer cannot merge the view's query into the accessing query block. Instead, the optimizer can transform the accessing statement by pushing its predicate, the `WHERE` clause condition `department_id=50`, into the view's `UNION` set query. The equivalent transformed query is as follows:

```
SELECT last_name
FROM   ( SELECT employee_id, last_name, job_id, commission_pct,
department_id
         FROM   employees
         WHERE  department_id=50
         UNION
         SELECT employee_id, last_name, job_id, commission_pct,
department_id
         FROM   contract_workers
         WHERE  department_id=50 );
```

The transformed query can now consider index access in each of the query blocks.

5.4 Subquery Unnesting

In **subquery unnesting**, the optimizer transforms a nested query into an equivalent join statement, and then optimizes the join.

This transformation enables the optimizer to consider the subquery tables during access path, join method, and join order selection. The optimizer can perform this transformation only if the resulting join statement is guaranteed to return the same rows as the original statement, and if subqueries do not contain aggregate functions such as `AVG`.

For example, suppose you connect as user `sh` and execute the following query:

```
SELECT *
FROM   sales
WHERE  cust_id IN ( SELECT cust_id
                    FROM   customers );
```

Because the `customers.cust_id` column is a primary key, the optimizer can transform the complex query into the following join statement that is guaranteed to return the same data:

```
SELECT sales.*
FROM   sales, customers
WHERE  sales.cust_id = customers.cust_id;
```

If the optimizer cannot transform a complex statement into a join statement, it selects execution plans for the parent statement and the subquery as though they were separate statements. The optimizer then executes the subquery and uses the rows returned to execute the parent query. To improve execution speed of the overall execution plan, the optimizer orders the subplans efficiently.

5.5 Query Rewrite with Materialized Views

A **materialized view** is a query result stored in a table.

When the optimizer finds a user query compatible with the query associated with a materialized view, the database can rewrite the query in terms of the materialized view. This technique improves query execution because the database has precomputed most of the query result.

The optimizer looks for materialized views that are compatible with the user query, and then uses a cost-based algorithm to select materialized views to rewrite the query. The optimizer does not rewrite the query when the plan generated unless the materialized views has a lower cost than the plan generated with the materialized views.

See Also:

Oracle Database Data Warehousing Guide to learn more about query rewrite

5.5.1 About Query Rewrite and the Optimizer

A query undergoes several checks to determine whether it is a candidate for query rewrite.

If the query fails any check, then the query is applied to the detail tables rather than the materialized view. The inability to rewrite can be costly in terms of response time and processing power.

The optimizer uses two different methods to determine when to rewrite a query in terms of a materialized view. The first method matches the SQL text of the query to the SQL text of the materialized view definition. If the first method fails, then the optimizer uses the more general method in which it compares joins, selections, data columns, grouping columns, and aggregate functions between the query and materialized views.

Query rewrite operates on queries and subqueries in the following types of SQL statements:

- `SELECT`
- `CREATE TABLE … AS SELECT`
- `INSERT INTO … SELECT`

It also operates on subqueries in the set operators `UNION`, `UNION ALL`, `INTERSECT`, `INTERSECT ALL`, `EXCEPT`, `EXCEPT ALL`, `MINUS`, and `MINUS ALL`, and subqueries in DML statements such as `INSERT`, `DELETE`, and `UPDATE`.

Dimensions, constraints, and rewrite integrity levels affect whether a query is rewritten to use materialized views. Additionally, query rewrite can be enabled or disabled by `REWRITE` and `NOREWRITE` hints and the `QUERY_REWRITE_ENABLED` session parameter.

The `DBMS_MVIEW.EXPLAIN_REWRITE` procedure advises whether query rewrite is possible on a query and, if so, which materialized views are used. It also explains why a query cannot be rewritten.

5.5.2 About Initialization Parameters for Query Rewrite

Query rewrite behavior is controlled by certain database initialization parameters.

Table 5-1 Initialization Parameters that Control Query Rewrite Behavior

Initialization Parameter Name	Initialization Parameter Value	Behavior of Query Rewrite
`OPTIMIZER_MODE`	`ALL_ROWS` (default), `FIRST_ROWS`, or `FIRST_ROWS_`*n*	With `OPTIMIZER_MODE` set to `FIRST_ROWS`, the optimizer uses a mix of costs and heuristics to find a best plan for fast delivery of the first few rows. When set to `FIRST_ROWS_`*n*, the optimizer uses a cost-based approach and optimizes with a goal of best response time to return the first *n* rows (where n = 1, 10, 100, 1000).

Table 5-1 (Cont.) Initialization Parameters that Control Query Rewrite Behavior

Initialization Parameter Name	Initialization Parameter Value	Behavior of Query Rewrite
`QUERY_REWRITE_ENABLED`	`TRUE` (default), `FALSE`, or `FORCE`	This option enables the query rewrite feature of the optimizer, enabling the optimizer to utilize materialized views to enhance performance. If set to `FALSE`, this option disables the query rewrite feature of the optimizer and directs the optimizer not to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower. If set to `FORCE`, this option enables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unrewritten query is lower.
`QUERY_REWRITE_INTEGRITY`	`STALE_TOLERATED`, `TRUSTED`, or `ENFORCED` (the default)	This parameter is optional. However, if it is set, the value must be one of these specified in the Initialization Parameter Value column. By default, the integrity level is set to `ENFORCED`. In this mode, all constraints must be validated. Therefore, if you use `ENABLE NOVALIDATE RELY` , certain types of query rewrite might not work. To enable query rewrite in this environment (where constraints have not been validated), you should set the integrity level to a lower level of granularity such as `TRUSTED` or `STALE_TOLERATED`.

Related Topics

- About the Accuracy of Query Rewrite
  Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter `QUERY_REWRITE_INTEGRITY`.

5.5.3 About the Accuracy of Query Rewrite

Query rewrite offers three levels of rewrite integrity that are controlled by the initialization parameter `QUERY_REWRITE_INTEGRITY`.

The values that you can set for the `QUERY_REWRITE_INTEGRITY` parameter are as follows:

- `ENFORCED`

  This is the default mode. The optimizer only uses fresh data from the materialized views and only use those relationships that are based on `ENABLED VALIDATED` primary, unique, or foreign key constraints.

- `TRUSTED`

  In `TRUSTED` mode, the optimizer trusts that the relationships declared in dimensions and `RELY` constraints are correct. In this mode, the optimizer also uses prebuilt materialized views or materialized views based on views, and it uses relationships that are not enforced as well as those that are enforced. It also trusts declared but not `ENABLED VALIDATED` primary or unique key constraints and data relationships specified using dimensions. This mode offers greater query rewrite capabilities but also creates the risk of incorrect results if any of the trusted relationships you have declared are incorrect.

- `STALE_TOLERATED`

  In `STALE_TOLERATED` mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating inaccurate results.

If rewrite integrity is set to the safest level, `ENFORCED`, the optimizer uses only enforced primary key constraints and referential integrity constraints to ensure that the results of the query are the same as the results when accessing the detail tables directly.

If the rewrite integrity is set to levels other than `ENFORCED`, there are several situations where the output with rewrite can be different from that without it:

- A materialized view can be out of synchronization with the primary copy of the data. This generally happens because the materialized view refresh procedure is pending following bulk load or DML operations to one or more detail tables of a materialized view. At some data warehouse sites, this situation is desirable because it is not uncommon for some materialized views to be refreshed at certain time intervals.
- The relationships implied by the dimension objects are invalid. For example, values at a certain level in a hierarchy do not roll up to exactly one parent value.
- The values stored in a prebuilt materialized view table might be incorrect.
- A wrong answer can occur because of bad data relationships defined by unenforced table or view constraints.

You can set `QUERY_REWRITE_INTEGRITY` either in your initialization parameter file or using an `ALTER SYSTEM` or `ALTER SESSION` statement.

5.5.4 Example of Query Rewrite

This example illustrates the power of query rewrite with materialized views.

Consider the following materialized view, `cal_month_sales_mv`, which provides an aggregation of the dollar amount sold in every month:

```
CREATE MATERIALIZED VIEW cal_month_sales_mv
ENABLE QUERY REWRITE AS
SELECT t.calendar_month_desc, SUM(s.amount_sold) AS dollars
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

Let us assume that, in a typical month, the number of sales in the store is around one million. So this materialized aggregate view has the precomputed aggregates for the dollar amount sold for each month.

Consider the following query, which asks for the sum of the amount sold at the store for each calendar month:

```
SELECT t.calendar_month_desc, SUM(s.amount_sold)
FROM sales s, times t WHERE s.time_id = t.time_id
GROUP BY t.calendar_month_desc;
```

In the absence of the previous materialized view and query rewrite feature, Oracle Database must access the `sales` table directly and compute the sum of the amount sold to return the results. This involves reading many million rows from the `sales` table, which will invariably increase the query response time due to the disk access. The join in the query will also further slow down the query response as the join needs to be computed on many million rows.

In the presence of the materialized view `cal_month_sales_mv`, query rewrite will transparently rewrite the previous query into the following query:

```
SELECT calendar_month, dollars
FROM cal_month_sales_mv;
```

Because there are only a few dozen rows in the materialized view `cal_month_sales_mv` and no joins, Oracle Database returns the results instantly.

5.6 Star Transformation

Star transformation is an optimizer transformation that avoids full table scans of fact tables in a star schema.

5.6.1 About Star Schemas

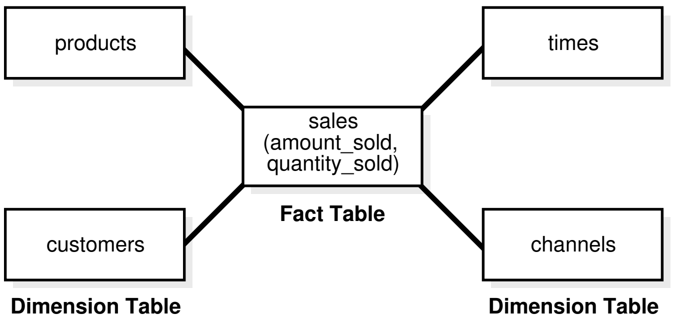
A **star schema** divides data into facts and dimensions.

Facts are the measurements of an event such as a sale and are typically numbers. Dimensions are the categories that identify facts, such as date, location, and product.

A fact table has a composite key made up of the primary keys of the dimension tables of the schema. Dimension tables act as lookup or reference tables that enable you to choose values that constrain your queries.

Diagrams typically show a central fact table with lines joining it to the dimension tables, giving the appearance of a star. The following graphic shows `sales` as the fact table and `products`, `times`, `customers`, and `channels` as the dimension tables.

Figure 5-1 Star Schema



A snowflake schema is a star schema in which the dimension tables reference other tables. A snowstorm schema is a combination of snowflake schemas.

See Also:

Oracle Database Data Warehousing Guide to learn more about star schemas

5.6.2 Purpose of Star Transformations

In joins of fact and dimension tables, a star transformation can avoid a full scan of a fact table.

The star transformation improves performance by fetching only relevant fact rows that join to the constraint dimension rows. In some cases, queries have restrictive filters on other columns of the dimension tables. The combination of filters can dramatically reduce the data set that the database processes from the fact table.

5.6.3 How Star Transformation Works

Star transformation adds subquery predicates, called **bitmap semijoin predicates**, corresponding to the constraint dimensions.

The optimizer performs the transformation when indexes exist on the fact join columns. By driving bitmap `AND` and `OR` operations of key values supplied by the subqueries, the database only needs to retrieve relevant rows from the fact table. If the predicates on the dimension tables filter out significant data, then the transformation can be more efficient than a full scan on the fact table.

After the database has retrieved the relevant rows from the fact table, the database may need to join these rows back to the dimension tables using the original predicates. The database can eliminate the join back of the dimension table when the following conditions are met:

- All the predicates on dimension tables are part of the semijoin subquery predicate.
- The columns selected from the subquery are unique.
- The dimension columns are not in the `SELECT` list, `GROUP BY` clause, and so on.

5.6.4 Controls for Star Transformation

The `STAR_TRANSFORMATION_ENABLED` initialization parameter controls star transformations.

This parameter takes the following values:

- `true`

  The optimizer performs the star transformation by identifying the fact and constraint dimension tables automatically. The optimizer performs the star transformation only if the cost of the transformed plan is lower than the alternatives. Also, the optimizer attempts temporary table transformation automatically whenever materialization improves performance (see "Temporary Table Transformation: Scenario").
- `false` (default)

The optimizer does not perform star transformations.

- `TEMP_DISABLE`

  This value is identical to `true` except that the optimizer does not attempt temporary table transformation.

See Also:

Oracle Database Reference to learn about the `STAR_TRANSFORMATION_ENABLED` initialization parameter

5.6.5 Star Transformation: Scenario

This scenario demonstrates a star transformation of a star query.

Example 5-5 Star Query

The following query finds the total Internet sales amount in all cities in California for quarters Q1 and Q2 of year 1999:

```
SELECT c.cust_city,
       t.calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM   sales s,
       times t,
       customers c,
       channels ch
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    s.channel_id = ch.channel_id
AND    c.cust_state_province = 'CA'
AND    ch.channel_desc = 'Internet'
AND    t.calendar_quarter_desc IN ('1999-01','1999-02')
GROUP BY c.cust_city, t.calendar_quarter_desc;
```

Sample output is as follows:

```
CUST_CITY                      CALENDA SALES_AMOUNT
------------------------------ ------- ------------
Montara                        1999-02      1618.01
Pala                           1999-01      3263.93
Cloverdale                     1999-01        52.64
Cloverdale                     1999-02       266.28
. . .
```

In this example, `sales` is the fact table, and the other tables are dimension tables. The `sales` table contains one row for every sale of a product, so it could conceivably contain billions of sales records. However, only a few products are sold to customers in California through the Internet for the specified quarters.

Example 5-6 Star Transformation

This example shows a star transformation of the query in Example 5-5. The transformation avoids a full table scan of `sales`.

```
SELECT c.cust_city, t.calendar_quarter_desc, SUM(s.amount_sold)
sales_amount
FROM   sales s, times t, customers c
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    c.cust_state_province = 'CA'
AND    t.calendar_quarter_desc IN ('1999-01','1999-02')
AND    s.time_id IN ( SELECT time_id
                      FROM   times
                      WHERE  calendar_quarter_desc
IN('1999-01','1999-02') )
AND    s.cust_id IN ( SELECT cust_id
                      FROM   customers
                      WHERE  cust_state_province='CA' )
AND    s.channel_id IN ( SELECT channel_id
                         FROM   channels
                         WHERE  channel_desc = 'Internet' )
GROUP BY c.cust_city, t.calendar_quarter_desc;
```

Example 5-7 Partial Execution Plan for Star Transformation

This example shows an edited version of the execution plan for the star transformation in Example 5-6.

Line 26 shows that the `sales` table has an index access path instead of a full table scan. For each key value that results from the subqueries of `channels` (line 14), `times` (line 19), and `customers` (line 24), the database retrieves a bitmap from the indexes on the `sales` fact table (lines 15, 20, 25).

Each bit in the bitmap corresponds to a row in the fact table. The bit is set when the key value from the subquery is same as the value in the row of the fact table. For example, in the bitmap `101000...` (the ellipses indicates that the values for the remaining rows are `0`), rows 1 and 3 of the fact table have matching key values from the subquery.

The operations in lines 12, 17, and 22 iterate over the keys from the subqueries and retrieve the corresponding bitmaps. In Example 5-6, the `customers` subquery seeks the IDs of customers whose state or province is `CA`. Assume that the bitmap `101000...` corresponds to the customer ID key value `103515` from the `customers` table subquery. Also assume that the `customers` subquery produces the key value `103516` with the bitmap `010000...`, which means that only row 2 in `sales` has a matching key value from the subquery.

The database merges (using the `OR` operator) the bitmaps for each subquery (lines 11, 16, 21). In our `customers` example, the database produces a single bitmap `111000...` for the `customers` subquery after merging the two bitmaps:

```
101000...   # bitmap corresponding to key 103515
010000...   # bitmap corresponding to key 103516
```

```
---------
111000...   # result of OR operation
```

In line 10, the database applies the `AND` operator to the merged bitmaps. Assume that after the database has performed all `OR` operations, the resulting bitmap for `channels` is `100000...` If the database performs an `AND` operation on this bitmap and the bitmap from `customers` subquery, then the result is as follows:

```
100000...   # channels bitmap after all OR operations performed
111000...   # customers bitmap after all OR operations performed
---------
100000...   # bitmap result of AND operation for channels and customers
```

In line 9, the database generates the corresponding rowids of the final bitmap. The database retrieves rows from the `sales` fact table using the rowids (line 26). In our example, the database generate only one rowid, which corresponds to the first row, and thus fetches only a single row instead of scanning the entire `sales` table.

```
---------------------------------------------------------------------------
| Id  | Operation                          | Name
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                   |
|   1 |  HASH GROUP BY                     |
|*  2 |   HASH JOIN                        |
|*  3 |    TABLE ACCESS FULL               | CUSTOMERS
|*  4 |    HASH JOIN                       |
|*  5 |     TABLE ACCESS FULL              | TIMES
|   6 |     VIEW                           | VW_ST_B1772830
|   7 |      NESTED LOOPS                  |
|   8 |       PARTITION RANGE SUBQUERY     |
|   9 |        BITMAP CONVERSION TO ROWIDS|
|  10 |         BITMAP AND                 |
|  11 |          BITMAP MERGE              |
|  12 |           BITMAP KEY ITERATION     |
|  13 |            BUFFER SORT             |
|* 14 |             TABLE ACCESS FULL      | CHANNELS
|* 15 |            BITMAP INDEX RANGE SCAN| SALES_CHANNEL_BIX
|  16 |          BITMAP MERGE              |
|  17 |           BITMAP KEY ITERATION     |
|  18 |            BUFFER SORT             |
|* 19 |             TABLE ACCESS FULL      | TIMES
|* 20 |            BITMAP INDEX RANGE SCAN| SALES_TIME_BIX
|  21 |          BITMAP MERGE              |
|  22 |           BITMAP KEY ITERATION     |
|  23 |            BUFFER SORT             |
|* 24 |             TABLE ACCESS FULL      | CUSTOMERS
|* 25 |            BITMAP INDEX RANGE SCAN| SALES_CUST_BIX
|  26 |       TABLE ACCESS BY USER ROWID  | SALES
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   2 - access("ITEM_1"="C"."CUST_ID")
   3 - filter("C"."CUST_STATE_PROVINCE"='CA')
   4 - access("ITEM_2"="T"."TIME_ID")
   5 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-01'
               OR "T"."CALENDAR_QUARTER_DESC"='1999-02'))
  14 - filter("CH"."CHANNEL_DESC"='Internet')
  15 - access("S"."CHANNEL_ID"="CH"."CHANNEL_ID")
  19 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-01'
               OR "T"."CALENDAR_QUARTER_DESC"='1999-02'))
  20 - access("S"."TIME_ID"="T"."TIME_ID")
  24 - filter("C"."CUST_STATE_PROVINCE"='CA')
  25 - access("S"."CUST_ID"="C"."CUST_ID")

Note
-----
   - star transformation used for this statement
```

5.6.6 Temporary Table Transformation: Scenario

In the preceding scenario, the optimizer does not join back the table `channels` to the `sales` table because it is not referenced outside and the `channel_id` is unique.

If the optimizer cannot eliminate the join back, however, then the database stores the subquery results in a temporary table to avoid rescanning the dimension table for bitmap key generation and join back. Also, if the query runs in parallel, then the database materializes the results so that each parallel execution server can select the results from the temporary table instead of executing the subquery again.

Example 5-8 Star Transformation Using Temporary Table

In this example, the database materializes the results of the subquery on `customers` into a temporary table:

```
SELECT t1.c1 cust_city, t.calendar_quarter_desc calendar_quarter_desc,
       SUM(s.amount_sold) sales_amount
FROM   sales s, sh.times t, sys_temp_0fd9d6621_e7e24 t1
WHERE  s.time_id=t.time_id
AND    s.cust_id=t1.c0
AND    (t.calendar_quarter_desc='1999-q1' OR
t.calendar_quarter_desc='1999-q2')
AND    s.cust_id IN    ( SELECT t1.c0
                         FROM   sys_temp_0fd9d6621_e7e24 t1 )
AND    s.channel_id IN ( SELECT ch.channel_id
                         FROM   channels ch
                         WHERE  ch.channel_desc='internet' )
AND    s.time_id IN    ( SELECT t.time_id
                         FROM   times t
                         WHERE  t.calendar_quarter_desc='1999-q1'
                         OR     t.calendar_quarter_desc='1999-q2' )
GROUP BY t1.c1, t.calendar_quarter_desc
```

The optimizer replaces `customers` with the temporary table `sys_temp_0fd9d6621_e7e24`, and replaces references to columns `cust_id` and `cust_city` with the corresponding columns of the temporary table. The database creates the temporary table with two columns: `(c0 NUMBER, c1 VARCHAR2(30))`. These columns correspond to `cust_id` and `cust_city` of the `customers` table. The database populates the temporary table by executing the following query at the beginning of the execution of the previous query:

```
SELECT c.cust_id, c.cust_city FROM customers WHERE c.cust_state_province =
'CA'
```

Example 5-9   Partial Execution Plan for Star Transformation Using Temporary Table

The following example shows an edited version of the execution plan for the query in Example 5-8:

```
---------------------------------------------------------------------------
| Id  | Operation                            | Name
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |
|   1 |  TEMP TABLE TRANSFORMATION           |
|   2 |   LOAD AS SELECT                     |
|*  3 |    TABLE ACCESS FULL                 | CUSTOMERS
|   4 |   HASH GROUP BY                      |
|*  5 |    HASH JOIN                         |
|   6 |     TABLE ACCESS FULL                | SYS_TEMP_0FD9D6613_C716F
|*  7 |     HASH JOIN                        |
|*  8 |      TABLE ACCESS FULL               | TIMES
|   9 |      VIEW                            | VW_ST_A3F94988
|  10 |       NESTED LOOPS                   |
|  11 |        PARTITION RANGE SUBQUERY      |
|  12 |         BITMAP CONVERSION TO ROWIDS|
|  13 |          BITMAP AND                  |
|  14 |           BITMAP MERGE               |
|  15 |            BITMAP KEY ITERATION      |
|  16 |             BUFFER SORT              |
|* 17 |              TABLE ACCESS FULL       | CHANNELS
|* 18 |             BITMAP INDEX RANGE SCAN| SALES_CHANNEL_BIX
|  19 |           BITMAP MERGE               |
|  20 |            BITMAP KEY ITERATION      |
|  21 |             BUFFER SORT              |
|* 22 |              TABLE ACCESS FULL       | TIMES
|* 23 |             BITMAP INDEX RANGE SCAN| SALES_TIME_BIX
|  24 |           BITMAP MERGE               |
|  25 |            BITMAP KEY ITERATION      |
|  26 |             BUFFER SORT              |
|  27 |              TABLE ACCESS FULL       | SYS_TEMP_0FD9D6613_C716F
|* 28 |             BITMAP INDEX RANGE SCAN| SALES_CUST_BIX
|  29 |        TABLE ACCESS BY USER ROWID    | SALES
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("C"."CUST_STATE_PROVINCE"='CA')
   5 - access("ITEM_1"="C0")
   7 - access("ITEM_2"="T"."TIME_ID")
   8 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-01' OR
               "T"."CALENDAR_QUARTER_DESC"='1999-02'))
  17 - filter("CH"."CHANNEL_DESC"='Internet')
  18 - access("S"."CHANNEL_ID"="CH"."CHANNEL_ID")
  22 - filter(("T"."CALENDAR_QUARTER_DESC"='1999-01' OR
               "T"."CALENDAR_QUARTER_DESC"='1999-02'))
  23 - access("S"."TIME_ID"="T"."TIME_ID")
  28 - access("S"."CUST_ID"="C0")
```

Lines 1, 2, and 3 of the plan materialize the `customers` subquery into the temporary table. In line 6, the database scans the temporary table (instead of the subquery) to build the bitmap from the fact table. Line 27 scans the temporary table for joining back instead of scanning `customers`. The database does not need to apply the filter on `customers` on the temporary table because the filter is applied while materializing the temporary table.

5.7 In-Memory Aggregation (VECTOR GROUP BY)

The key optimization of in-memory aggregation is to aggregate while scanning.

To optimize query blocks involving aggregation and joins from a single large table to multiple small tables, such as in a typical star query, the transformation uses `KEY VECTOR` and `VECTOR GROUP BY` operations. These operations use efficient in-memory arrays for joins and aggregation, and are especially effective when the underlying tables are in-memory columnar tables.

See Also:

Oracle Database In-Memory Guide to learn more about in-memory aggregation

5.8 Cursor-Duration Temporary Tables

To materialize the intermediate results of a query, Oracle Database may implicitly create a **cursor-duration temporary table** in memory during query compilation.

5.8.1 Purpose of Cursor-Duration Temporary Tables

Complex queries sometimes process the same query block multiple times, which creates unnecessary performance overhead.

To avoid this scenario, Oracle Database can automatically create temporary tables for the query results and store them in memory for the duration of the cursor. For complex operations such as `WITH` clause queries, star transformations, and grouping sets, this optimization enhances the materialization of intermediate results from repetitively used subqueries. In this way, cursor-duration temporary tables improve performance and optimize I/O.

5.8.2 How Cursor-Duration Temporary Tables Work

The definition of the cursor-definition temporary table resides in memory. The table definition is associated with the cursor, and is only visible to the session executing the cursor.

When using cursor-duration temporary tables, the database performs the following steps:

1. Chooses a plan that uses a cursor-duration temporary table
2. Creates the temporary table using a unique name
3. Rewrites the query to refer to the temporary table
4. Loads data into memory until no memory remains, in which case it creates temporary segments on disk
5. Executes the query, returning data from the temporary table
6. Truncates the table, releasing memory and any on-disk temporary segments

Note:

The metadata for the cursor-duration temporary table stays in memory as long as the cursor is in memory. The metadata is not stored in the data dictionary, which means it is not visible through data dictionary views. You cannot drop the metadata explicitly.

The preceding scenario depends on the availability of memory. For serial queries, the temporary tables use PGA memory.

The implementation of cursor-duration temporary tables is similar to sorts. If no more memory is available, then the database writes data to temporary segments. For cursor-duration temporary tables, the differences are as follows:

- The database releases memory and temporary segments at the end of the query rather than when the row source is no longer active.
- Data in memory stays in memory, unlike in sorts where data can move between memory and temporary segments.

When the database uses cursor-duration temporary tables, the keyword `CURSOR DURATION MEMORY` appears in the execution plan.

5.8.3 Cursor-Duration Temporary Tables: Example

A `WITH` query that repeats the same subquery can sometimes benefit from a cursor-duration temporary table.

The following query uses a `WITH` clause to create three subquery blocks:

```
WITH
  q1 AS (SELECT department_id, SUM(salary) sum_sal FROM hr.employees GROUP
BY department_id),
  q2 AS (SELECT * FROM q1),
  q3 AS (SELECT department_id, sum_sal FROM q1)
```

```
SELECT * FROM q1
UNION ALL
SELECT * FROM q2
UNION ALL
SELECT * FROM q3;
```

The following sample plan shows the transformation:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'BASIC +ROWS +COST'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------------
-----
| Id | Operation                                | Name                    |Rows |Cost
(%CPU)|
---------------------------------------------------------------------------------
-----
|  0 | SELECT STATEMENT                         |                          |    |6
(100)|
|  1 |  TEMP TABLE TRANSFORMATION               |                          |
|       |
|  2 |   LOAD AS SELECT (CURSOR DURATION MEMORY) | SYS_TEMP_0FD9D6606_1AE004 |
|       |
|  3 |    HASH GROUP BY                         |                          | 11 | 3
(34)|
|  4 |     TABLE ACCESS FULL                    | EMPLOYEES                |107 | 2
(0)  |
|  5 |   UNION-ALL                              |                          |
|       |
|  6 |    VIEW                                  |                          | 11 | 2
(0)  |
|  7 |     TABLE ACCESS FULL                    | SYS_TEMP_0FD9D6606_1AE004 | 11 | 2
(0)  |
|  8 |    VIEW                                  |                          | 11 | 2
(0)  |
|  9 |     TABLE ACCESS FULL                    | SYS_TEMP_0FD9D6606_1AE004 | 11 | 2
(0)  |
| 10 |    VIEW                                  |                          | 11 | 2
(0)  |
| 11 |     TABLE ACCESS FULL                    | SYS_TEMP_0FD9D6606_1AE004 | 11 | 2
(0)  |
---------------------------------------------------------------------------------
-----
```

In the preceding plan, `TEMP TABLE TRANSFORMATION` in Step 1 indicates that the database used cursor-duration temporary tables to execute the query. The `CURSOR DURATION MEMORY` keyword in Step 2 indicates that the database used memory, if available, to store the results of `SYS_TEMP_0FD9D6606_1AE004`. If memory was unavailable, then the database wrote the temporary data to disk.

5.9 Table Expansion

In **table expansion**, the optimizer generates a plan that uses indexes on the read-mostly portion of a partitioned table, but not on the active portion of the table.

5.9.1 Purpose of Table Expansion

Index-based plans can improve performance, but index maintenance creates overhead. In many databases, DML affects only a small portion of the data.

Table expansion uses index-based plans for high-update tables. You can create an index only on the read-mostly data, eliminating index overhead on the active data. In this way, table expansion improves performance while avoiding index maintenance.

5.9.2 How Table Expansion Works

Table partitioning makes table expansion possible.

If a local index exists on a partitioned table, then the optimizer can mark the index as unusable for specific partitions. In effect, some partitions are not indexed.

In table expansion, the optimizer transforms the query into a `UNION ALL` statement, with some subqueries accessing indexed partitions and other subqueries accessing unindexed partitions. The optimizer can choose the most efficient access method available for a partition, regardless of whether it exists for all of the partitions accessed in the query.

The optimizer does not always choose table expansion:

- Table expansion is cost-based.

  While the database accesses each partition of the expanded table only once across all branches of the `UNION ALL`, any tables that the database joins to it are accessed in each branch.
- Semantic issues may render expansion invalid.

  For example, a table appearing on the right side of an outer join is not valid for table expansion.

You can control table expansion with the hint `EXPAND_TABLE` hint. The hint overrides the cost-based decision, but not the semantic checks.

See Also:

- "Influencing the Optimizer with Hints"
- *Oracle Database SQL Language Reference* to learn more about SQL hints

5.9.3 Table Expansion: Scenario

The optimizer keeps track of which partitions must be accessed from each table, based on predicates that appear in the query. Partition pruning enables the optimizer to use table expansion to generate more optimal plans.

Assumptions

This scenario assumes the following:

- You want to run a star query against the `sh.sales` table, which is range-partitioned on the `time_id` column.
- You want to disable indexes on specific partitions to see the benefits of table expansion.

To use table expansion:

1. Log in to the database as the `sh` user.
2. Run the following query:

```
SELECT *
FROM   sales
WHERE  time_id >= TO_DATE('2000-01-01 00:00:00', 'SYYYY-MM-DD
HH24:MI:SS')
AND    prod_id = 38;
```

3. Explain the plan by querying `DBMS_XPLAN`:

```
SET LINESIZE 150
SET PAGESIZE 0
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format =>
'BASIC,PARTITION'));
```

As shown in the `Pstart` and `Pstop` columns in the following plan, the optimizer determines from the filter that only 16 of the 28 partitions in the table must be accessed:

```
Plan hash value: 3087065703

---------------------------------------------------------------------
----
|Id| Operation                                  | Name    |Pstart|
Pstop|
---------------------------------------------------------------------
----
| 0| SELECT STATEMENT                           |              |
|    |
| 1|  PARTITION RANGE ITERATOR                  |              |13|
28 |
| 2|   TABLE ACCESS BY LOCAL INDEX ROWID BATCHED| SALES        |13|
28 |
| 3|    BITMAP CONVERSION TO ROWIDS             |              |
|    |
|*4|     BITMAP INDEX SINGLE VALUE              |SALES_PROD_BIX|13| 28 |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("PROD_ID"=38)
```

After the optimizer has determined the partitions to be accessed, it considers any index that is usable on all of those partitions. In the preceding plan, the optimizer chose to use the `sales_prod_bix` bitmap index.

4. Disable the index on the `SALES_1995` partition of the `sales` table:

```
ALTER INDEX sales_prod_bix MODIFY PARTITION sales_1995 UNUSABLE;
```

The preceding DDL disables the index on partition 1, which contains all sales from before 1996.

Note:

You can obtain the partition information by querying the `USER_IND_PARTITIONS` view.

5. Execute the query of sales again, and then query `DBMS_XPLAN` to obtain the plan.

The output shows that the plan did not change:

```
Plan hash value: 3087065703

-----------------------------------------------------------------------
|Id| Operation                                  | Name     |Pstart|Pstop
-----------------------------------------------------------------------
| 0| SELECT STATEMENT                           |               |  |    |
| 1|  PARTITION RANGE ITERATOR                  |               |13|28  |
| 2|   TABLE ACCESS BY LOCAL INDEX ROWID BATCHED| SALES         |13|28  |
| 3|    BITMAP CONVERSION TO ROWIDS             |               |  |    |
|*4|     BITMAP INDEX SINGLE VALUE              | SALES_PROD_BIX|13|28  |
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("PROD_ID"=38)
```

The plan is the same because the disabled index partition is not relevant to the query. If all partitions that the query accesses are indexed, then the database can answer the query using the index. Because the query only accesses partitions 16 through 28, disabling the index on partition 1 does not affect the plan.

6. Disable the indexes for partition 28 (`SALES_Q4_2003`), which is a partition that the query needs to access:

```
ALTER INDEX sales_prod_bix MODIFY PARTITION sales_q4_2003 UNUSABLE;
ALTER INDEX sales_time_bix MODIFY PARTITION sales_q4_2003 UNUSABLE;
```

   By disabling the indexes on a partition that the query does need to access, the query can no longer use this index (without table expansion).

7. Query the plan using `DBMS_XPLAN`.

   As shown in the following plan, the optimizer does not use the index:

```
Plan hash value: 3087065703

--------------------------------------------------------------------
----
| Id| Operation                              | Name       |Pstart|
Pstop
--------------------------------------------------------------------
----
| 0 | SELECT STATEMENT                       |                |
|    |
| 1 |  PARTITION RANGE ITERATOR              |                |13 |
28 |
|*2 |   TABLE ACCESS FULL                    | SALES          |13 |
28 |
--------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("PROD_ID"=38)
```

   In the preceding example, the query accesses 16 partitions. On 15 of these partitions, an index is available, but no index is available for the final partition. Because the optimizer has to choose one access path or the other, the optimizer cannot use the index on any of the partitions.

8. With table expansion, the optimizer rewrites the original query as follows:

```
SELECT *
FROM   sales
WHERE  time_id >= TO_DATE('2000-01-01 00:00:00', 'SYYYY-MM-DD
HH24:MI:SS')
AND    time_id <  TO_DATE('2003-10-01 00:00:00', 'SYYYY-MM-DD
HH24:MI:SS')
AND    prod_id = 38
UNION ALL
SELECT *
FROM   sales
WHERE  time_id >= TO_DATE('2003-10-01 00:00:00', 'SYYYY-MM-DD
HH24:MI:SS')
AND    time_id < TO_DATE('2004-01-01 00:00:00', 'SYYYY-MM-DD
HH24:MI:SS')
AND    prod_id = 38;
```

In the preceding query, the first query block in the `UNION ALL` accesses the partitions that are indexed, while the second query block accesses the partition that is not. The two subqueries enable the optimizer to choose to use the index in the first query block, if it is more optimal than using a table scan of all of the partitions that are accessed.

9. Query the plan using `DBMS_XPLAN`.

   The plan appears as follows:

```
Plan hash value: 2120767686

-----------------------------------------------------------------------
|Id| Operation                                  |Name        |Pstart|Pstop|
-----------------------------------------------------------------------
| 0|SELECT STATEMENT                            |             |  |   |
| 1| VIEW                                       |VW_TE_2      |  |   |
| 2|  UNION-ALL                                 |             |  |   |
| 3|   PARTITION RANGE ITERATOR                 |             |13| 27|
| 4|    TABLE ACCESS BY LOCAL INDEX ROWID BATCHED|SALES        |13| 27|
| 5|     BITMAP CONVERSION TO ROWIDS            |             |  |   |
|*6|      BITMAP INDEX SINGLE VALUE             |SALES_PROD_BIX|13| 27|
| 7|   PARTITION RANGE SINGLE                   |             |28| 28|
|*8|    TABLE ACCESS FULL                       |SALES        |28| 28|
-----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   6 - access("PROD_ID"=38)
   8 - filter("PROD_ID"=38)
```

   As shown in the preceding plan, the optimizer uses a `UNION ALL` for two query blocks (Step 2). The optimizer chooses an index to access partitions 13 to 27 in the first query block (Step 6). Because no index is available for partition 28, the optimizer chooses a full table scan in the second query block (Step 8).

5.9.4 Table Expansion and Star Transformation: Scenario

Star transformation enables specific types of queries to avoid accessing large portions of big fact tables.

Star transformation requires defining several indexes, which in an actively updated table can have overhead. With table expansion, you can define indexes on only the inactive partitions so that the optimizer can consider star transformation on only the indexed portions of the table.

Assumptions

This scenario assumes the following:

- You query the same schema used in "Star Transformation: Scenario".

- The last partition of `sales` is actively being updated, as is often the case with time-partitioned tables.
- You want the optimizer to take advantage of table expansion.

To take advantage of table expansion in a star query:

1. Disable the indexes on the last partition as follows:

```
ALTER INDEX sales_channel_bix MODIFY PARTITION sales_q4_2003
UNUSABLE;
ALTER INDEX sales_cust_bix MODIFY PARTITION sales_q4_2003 UNUSABLE;
```

2. Execute the following star query:

```
SELECT t.calendar_quarter_desc, SUM(s.amount_sold) sales_amount
FROM   sales s, times t, customers c, channels ch
WHERE  s.time_id = t.time_id
AND    s.cust_id = c.cust_id
AND    s.channel_id = ch.channel_id
AND    c.cust_state_province = 'CA'
AND    ch.channel_desc = 'Internet'
AND    t.calendar_quarter_desc IN ('1999-01','1999-02')
GROUP BY t.calendar_quarter_desc;
```

3. Query the cursor using `DBMS_XPLAN`, which shows the following plan:

```
---------------------------------------------------------------------------
|Id| Operation                            | Name             | Pstart| Pstop |
---------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |                  |       |       |
| 1|  HASH GROUP BY                       |                  |       |       |
| 2|   VIEW                               |VW_TE_14          |       |       |
| 3|    UNION-ALL                         |                  |       |       |
| 4|     HASH JOIN                        |                  |       |       |
| 5|      TABLE ACCESS FULL               |TIMES             |       |       |
| 6|      VIEW                            |VW_ST_1319B6D8    |       |       |
| 7|       NESTED LOOPS                   |                  |       |       |
| 8|        PARTITION RANGE SUBQUERY      |                  |KEY(SQ)|KEY(SQ)|
| 9|         BITMAP CONVERSION TO ROWIDS|                  |       |       |
|10|          BITMAP AND                  |                  |       |       |
|11|           BITMAP MERGE               |                  |       |       |
|12|            BITMAP KEY ITERATION      |                  |       |       |
|13|             BUFFER SORT              |                  |       |       |
|14|              TABLE ACCESS FULL       |CHANNELS          |       |       |
|15|             BITMAP INDEX RANGE SCAN|SALES_CHANNEL_BIX|KEY(SQ)|KEY(SQ)|
|16|           BITMAP MERGE               |                  |       |       |
|17|            BITMAP KEY ITERATION      |                  |       |       |
|18|             BUFFER SORT              |                  |       |       |
|19|              TABLE ACCESS FULL       |TIMES             |       |       |
|20|             BITMAP INDEX RANGE SCAN|SALES_TIME_BIX    |KEY(SQ)|KEY(SQ)|
|21|           BITMAP MERGE               |                  |       |       |
|22|            BITMAP KEY ITERATION      |                  |       |       |
|23|             BUFFER SORT              |                  |       |       |
|24|              TABLE ACCESS FULL       |CUSTOMERS         |       |       |
|25|            BITMAP INDEX RANGE SCAN|SALES_CUST_BIX   |KEY(SQ)|KEY(SQ)|
|26|         TABLE ACCESS BY USER ROWID |SALES            | ROWID | ROWID |
|27|      NESTED LOOPS                  |                 |       |       |
|28|       NESTED LOOPS                 |                 |       |       |
|29|        NESTED LOOPS                |                 |       |       |
|30|         NESTED LOOPS               |                 |       |       |
|31|          PARTITION RANGE SINGLE    |                 |   28  |   28  |
|32|           TABLE ACCESS FULL        |SALES            |   28  |   28  |
|33|          TABLE ACCESS BY INDEX ROWID|CHANNELS        |       |       |
|34|           INDEX UNIQUE SCAN        |CHANNELS_PK      |       |       |
|35|         TABLE ACCESS BY INDEX ROWID |CUSTOMERS       |       |       |
|36|          INDEX UNIQUE SCAN         |CUSTOMERS_PK     |       |       |
|37|        INDEX UNIQUE SCAN           |TIMES_PK         |       |       |
|38|       TABLE ACCESS BY INDEX ROWID  |TIMES            |       |       |
---------------------------------------------------------------------------
```

The preceding plan uses table expansion. The `UNION ALL` branch that is accessing every partition except the last partition uses star transformation. Because the indexes on partition 28 are disabled, the database accesses the final partition using a full table scan.

5.10 Join Factorization

In the cost-based transformation known as **join factorization**, the optimizer can factorize common computations from branches of a `UNION ALL` query.

5.10.1 Purpose of Join Factorization

`UNION ALL` queries are common in database applications, especially in data integration applications.

Often, branches in a `UNION ALL` query refer to the same base tables. Without join factorization, the optimizer evaluates each branch of a `UNION ALL` query independently, which leads to repetitive processing, including data access and joins. Join factorization transformation can share common computations across the `UNION ALL` branches. Avoiding an extra scan of a large base table can lead to a huge performance improvement.

5.10.2 How Join Factorization Works

Join factorization can factorize multiple tables and from more than two `UNION ALL` branches.

Join factorization is best explained through examples.

Example 5-10 UNION ALL Query

The following query shows a query of four tables (`t1`, `t2`, `t3`, and `t4`) and two `UNION ALL` branches:

```
SELECT t1.c1, t2.c2
FROM   t1, t2, t3
WHERE  t1.c1 = t2.c1
AND    t1.c1 > 1
AND    t2.c2 = 2
AND    t2.c2 = t3.c2
UNION ALL
```

```
SELECT t1.c1, t2.c2
FROM   t1, t2, t4
WHERE  t1.c1 = t2.c1
AND    t1.c1 > 1
AND    t2.c3 = t4.c3
```

In the preceding query, table `t1` appears in both `UNION ALL` branches, as does the filter predicate `t1.c1 > 1` and the join predicate `t1.c1 = t2.c1`. Without any transformation, the database must perform the scan and the filtering on table `t1` twice, one time for each branch.

Example 5-11 Factorized Query

Example 5-10

```
SELECT t1.c1, VW_JF_1.item_2
FROM   t1, (SELECT t2.c1 item_1, t2.c2 item_2
            FROM   t2, t3
            WHERE  t2.c2 = t3.c2
            AND    t2.c2 = 2
            UNION ALL
            SELECT t2.c1 item_1, t2.c2 item_2
            FROM   t2, t4
            WHERE  t2.c3 = t4.c3) VW_JF_1
WHERE  t1.c1 = VW_JF_1.item_1
AND    t1.c1 > 1
```

In this case, because table `t1` is factorized, the database performs the table scan and the filtering on `t1` only one time. If `t1` is large, then this factorization avoids the huge performance cost of scanning and filtering `t1` twice.

Note:

If the branches in a `UNION ALL` query have clauses that use the `DISTINCT` function, then join factorization is not valid.

5.10.3 Factorization and Join Orders: Scenario

Join factorization can create more possibilities for join orders

Example 5-12 Query Involving Five Tables

In the following query, view `V` is same as the query as in Example 5-10:

```
SELECT *
FROM   t5, (SELECT t1.c1, t2.c2
            FROM   t1, t2, t3
            WHERE  t1.c1 = t2.c1
            AND    t1.c1 > 1
            AND    t2.c2 = 2
            AND    t2.c2 = t3.c2
            UNION ALL
            SELECT t1.c1, t2.c2
            FROM   t1, t2, t4
            WHERE  t1.c1 = t2.c1
            AND    t1.c1 > 1
            AND    t2.c3 = t4.c3) V
WHERE  t5.c1 = V.c1
```

t1t2t3t5

Example 5-13   Factorization of t1 from View V

If join factorization factorizes `t1` from view `V`, as shown in the following query, then the database can join `t1` with `t5`.:

```
SELECT *
FROM   t5, ( SELECT t1.c1, VW_JF_1.item_2
             FROM   t1, (SELECT t2.c1 item_1, t2.c2 item_2
                         FROM   t2, t3
                         WHERE  t2.c2 = t3.c2
                         AND    t2.c2 = 2
                         UNION ALL
                         SELECT t2.c1 item_1, t2.c2 item_2
                         FROM   t2, t4
                         WHERE  t2.c3 = t4.c3) VW_JF_1
             WHERE  t1.c1 = VW_JF_1.item_1
             AND    t1.c1 > 1 )
WHERE  t5.c1 = V.c1
```

The preceding query transformation opens up new join orders. However, join factorization imposes specific join orders. For example, in the preceding query, tables `t2` and `t3` appear in the first branch of the `UNION ALL` query in view `VW_JF_1`. The database must join `t2` with `t3` before it can join with `t1`, which is not defined within the `VW_JF_1` view. The imposed join order may not necessarily be the best join order. For this reason, the optimizer performs join factorization using the cost-based transformation framework. The optimizer calculates the cost of the plans with and without join factorization, and then chooses the cheapest plan.

Example 5-14   Factorization of t1 from View V with View Definition Removed

The following query is the same query in Example 5-13, but with the view definition removed so that the factorization is easier to see:

```
SELECT *
FROM   t5, (SELECT t1.c1, VW_JF_1.item_2
            FROM   t1, VW_JF_1
            WHERE  t1.c1 = VW_JF_1.item_1
            AND    t1.c1 > 1)
WHERE  t5.c1 = V.c1
```

5.10.4 Factorization of Outer Joins: Scenario

The database supports join factorization of outer joins, antijoins, and semijoins, but only for the right tables in such joins.

For example, join factorization can transform the following `UNION ALL` query by factorizing `t2`:

```
SELECT t1.c2, t2.c2
FROM   t1, t2
WHERE  t1.c1 = t2.c1(+)
AND    t1.c1 = 1
UNION ALL
SELECT t1.c2, t2.c2
FROM   t1, t2
WHERE  t1.c1 = t2.c1(+)
AND    t1.c1 = 2
```

The following example shows the transformation. Table `t2` now no longer appears in the `UNION ALL` branches of the subquery.

```
SELECT VW_JF_1.item_2, t2.c2
FROM   t2, (SELECT t1.c1 item_1, t1.c2 item_2
            FROM   t1
            WHERE  t1.c1 = 1
            UNION ALL
            SELECT t1.c1 item_1, t1.c2 item_2
            FROM   t1
            WHERE  t1.c1 = 2) VW_JF_1
WHERE  VW_JF_1.item_1 = t2.c1(+)
```

Part III
Query Execution Plans

If a query has suboptimal performance, the execution plan is the key tool for understanding the problem and supplying a solution.

6

Explaining and Displaying Execution Plans

Knowledge of how to explain a statement and display its plan is essential to SQL tuning.

6.1 Introduction to Execution Plans

An **execution plan** is the sequence of operations that the database performs to run a SQL statement.

6.1.1 Contents of an Execution Plan

The execution plan operation alone cannot differentiate between well-tuned statements and those that perform suboptimally.

The plan consists of a series of steps. Every step either retrieves rows of data physically from the database or prepares them for the user issuing the statement. The following plan shows a join of the `employees` and `departments` tables:

```
SQL_ID  g9xaqjktdhbcd, child number 0
-------------------------------------
SELECT employee_id, last_name, first_name, department_name from
employees e, departments d WHERE e.department_id = d.department_id and
last_name like 'T%' ORDER BY last_name

Plan hash value: 1219589317

----------------------------------------------------------------------------------------
| Id | Operation                    | Name        |Rows | Bytes |Cost (%CPU)| Time     |
----------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |             |     |       |    5 (100)|          |
|  1 |  NESTED LOOPS                |             |   5 |   190 |    5   (0)| 00:00:01 |
|  2 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES   |   5 |   110 |    2   (0)| 00:00:01 |
|* 3 |    INDEX RANGE SCAN          | EMP_NAME_IX |   5 |       |    1   (0)| 00:00:01 |
|* 4 |   TABLE ACCESS FULL          | DEPARTMENTS |   1 |    16 |    1   (0)| 00:00:01 |
----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("LAST_NAME" LIKE 'T%')
       filter("LAST_NAME" LIKE 'T%')
   4 - filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

The row source tree is the core of the execution plan. The tree shows the following information:

- The join order of the tables referenced by the statement

In the preceding plan, `employees` is the outer row source and `departments` is the inner row source.

- An access path for each table mentioned in the statement

  In the preceding plan, the optimizer chooses to access `employees` using an index scan and `departments` using a full scan.
- A join method for tables affected by join operations in the statement

  In the preceding plan, the optimizer chooses a nested loops join.
- Data operations like filter, sort, or aggregation

  In the preceding plan, the optimizer filters on last names that begin with `T` and matches on `department_id`.

In addition to the row source tree, the plan table contains information about the following:

- Optimization, such as the cost and cardinality of each operation
- Partitioning, such as the set of accessed partitions
- Parallel execution, such as the distribution method of join inputs

6.1.2 Why Execution Plans Change

Execution plans can and do change as the underlying optimizer inputs change.

Note:

To avoid possible SQL performance regression that may result from execution plan changes, consider using SQL plan management.

See Also:

- "Overview of SQL Plan Management"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package

6.1.2.1 Different Schemas

Schemas can differ for various reasons.

Principal reasons include the following:

- The execution and explain plan occur on different databases.
- The user explaining the statement is different from the user running the statement. Two users might be pointing to different objects in the same database, resulting in different execution plans.
- Schema changes (often changes in indexes) between the two operations.

6.1.2.2 Different Costs

Even if the schemas are the same, the optimizer can choose different execution plans when the costs are different.

Some factors that affect the costs include the following:

- Data volume and statistics
- Bind variable types and values
- Initialization parameters set globally or at session level

6.2 Generating Plan Output Using the EXPLAIN PLAN Statement

The `EXPLAIN PLAN` statement enables you to examine the execution plan that the optimizer chose for a SQL statement.

6.2.1 About the EXPLAIN PLAN Statement

The `EXPLAIN PLAN` statement displays execution plans that the optimizer chooses for `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements.

`EXPLAIN PLAN` output shows how the database would have run the SQL statement when the statement was explained. Because of differences in the execution environment and explain plan environment, the explained plan can differ from the actual plan used during statement execution.

When the `EXPLAIN PLAN` statement is issued, the optimizer chooses an execution plan and then inserts a row describing each step of the execution plan into a specified plan table. You can also issue the `EXPLAIN PLAN` statement as part of the SQL trace facility.

The `EXPLAIN PLAN` statement is a DML statement rather than a DDL statement. Therefore, Oracle Database does not implicitly commit the changes made by an `EXPLAIN PLAN` statement.

See Also:

- "SQL Row Source Generation"
- *Oracle Database SQL Language Reference* to learn about the `EXPLAIN PLAN` statement

6.2.1.1 About PLAN_TABLE

`PLAN_TABLE` is the default sample output table into which the `EXPLAIN PLAN` statement inserts rows describing execution plans.

Oracle Database automatically creates a global temporary table `PLAN_TABLE$` in the `SYS` schema, and creates `PLAN_TABLE` as a synonym. All necessary privileges to `PLAN_TABLE` are granted to `PUBLIC`. Consequently, every session gets its own private copy of `PLAN_TABLE` in its temporary tablespace.

You can use the SQL script `catplan.sql` to manually create the global temporary table and the `PLAN_TABLE` synonym. The name and location of this script depends on your operating system. On UNIX and Linux, the script is located in the `$ORACLE_HOME/rdbms/admin` directory. For example, start a SQL*Plus session, connect with `SYSDBA` privileges, and run the script as follows:

```
@$ORACLE_HOME/rdbms/admin/catplan.sql
```

The definition of a sample output table `PLAN_TABLE` is available in a SQL script on your distribution media. Your output table must have the same column names and data types as this table. The common name of this script is `utlxplan.sql`. The exact name and location depend on your operating system.

See Also:

Oracle Database SQL Language Reference for a complete description of `EXPLAIN PLAN` syntax.

6.2.1.2 EXPLAIN PLAN Restrictions

Oracle Database does not support `EXPLAIN PLAN` for statements performing implicit type conversion of date bind variables.

With bind variables in general, the `EXPLAIN PLAN` output might not represent the real execution plan.

From the text of a SQL statement, `TKPROF` cannot determine the types of the bind variables. It assumes that the type is `VARCHAR`, and gives an error message otherwise. You can avoid this limitation by putting appropriate type conversions in the SQL statement.

See Also:

- "Performing Application Tracing "
- "Guideline for Avoiding the Argument Trap"
- *Oracle Database SQL Language Reference* to learn more about SQL data types

6.2.2 Explaining a SQL Statement: Basic Steps

Use `EXPLAIN PLAN` to store the plan for a SQL statement in `PLAN_TABLE`.

Prerequisites

This task assumes that a sample output table named `PLAN_TABLE` exists in your schema. If this table does not exist, then run the SQL script `catplan.sql`.

To execute `EXPLAIN PLAN`, you must have the following privileges:

- You must have the privileges necessary to insert rows into an existing output table that you specify to hold the execution plan
- You must also have the privileges necessary to execute the SQL statement for which you are determining the execution plan. If the SQL statement accesses a view, then you must have privileges to access any tables and views on which the view is based. If the view is based on another view that is based on a table, then you must have privileges to access both the other view and its underlying table.

To examine the execution plan produced by an `EXPLAIN PLAN` statement, you must have the privileges necessary to query the output table.

To explain a statement:

1. Start SQL*Plus or SQL Developer, and log in to the database as a user with the requisite permissions.
2. Include the `EXPLAIN PLAN FOR` clause immediately before the SQL statement.

   The following example explains the plan for a query of the `employees` table:

   ```
   EXPLAIN PLAN FOR
     SELECT e.last_name, d.department_name, e.salary
     FROM   employees e, departments d
     WHERE  salary < 3000
     AND    e.department_id = d.department_id
     ORDER BY salary DESC;
   ```

3. After issuing the `EXPLAIN PLAN` statement, use a script or package provided by Oracle Database to display the most recent plan table output.

   The following example uses the `DBMS_XPLAN.DISPLAY` function:

   ```
   SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));
   ```

4. Review the plan output.

   For example, the following plan shows a hash join:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));
Plan hash value: 3556827125

------------------------------------------------------------------------------
| Id | Operation           | Name          |Rows | Bytes |Cost (%CPU)| Time    |
------------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |               |   4 |   124 |   5  (20)| 00:00:01 |
```

```
|  1 |  SORT ORDER BY      |             |  4 |  124 |   5  (20)| 00:00:01 |
|* 2 |   HASH JOIN         |             |  4 |  124 |   4   (0)| 00:00:01 |
|* 3 |    TABLE ACCESS FULL| EMPLOYEES   |  4 |   60 |   2   (0)| 00:00:01 |
|  4 |    TABLE ACCESS FULL| DEPARTMENTS | 27 |  432 |   2   (0)| 00:00:01 |
------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / E@SEL$1
   4 - SEL$1 / D@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
   3 - filter("SALARY"<3000)

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=1) INTERNAL_FUNCTION("E"."SALARY")[22],
       "E"."LAST_NAME"[VARCHAR2,25], "D"."DEPARTMENT_NAME"[VARCHAR2,30]
   2 - (#keys=1) "E"."LAST_NAME"[VARCHAR2,25], "SALARY"[NUMBER,22],
       "D"."DEPARTMENT_NAME"[VARCHAR2,30], "D"."DEPARTMENT_NAME"[VARCHAR2,30]
   3 - "E"."LAST_NAME"[VARCHAR2,25], "SALARY"[NUMBER,22],
       "E"."DEPARTMENT_ID"[NUMBER,22]
   4 - "D"."DEPARTMENT_ID"[NUMBER,22], "D"."DEPARTMENT_NAME"[VARCHAR2,30]

Note
-----
   - this is an adaptive plan
```

The execution order in `EXPLAIN PLAN` output begins with the line that is the furthest indented to the right. The next step is the parent of that line. If two lines are indented equally, then the top line is normally executed first.

Note:

The steps in the `EXPLAIN PLAN` output in this chapter may be different on your database. The optimizer may choose different execution plans, depending on database configurations.

See Also:

- "About PLAN_TABLE"
- "About the Display of PLAN_TABLE Output"
- *Oracle Database SQL Language Reference* for the syntax and semantics of `EXPLAIN PLAN`

6.2.3 Specifying a Statement ID in EXPLAIN PLAN: Example

With multiple statements, you can specify a statement identifier and use that to identify your specific execution plan.

Before using `SET STATEMENT ID`, remove any existing rows for that statement ID. In the following example, `st1` is specified as the statement identifier.

Example 6-1   Using EXPLAIN PLAN with the STATEMENT ID Clause

```
EXPLAIN PLAN
  SET STATEMENT_ID = 'st1' FOR
  SELECT last_name FROM employees;
```

6.2.4 Specifying a Different Location for EXPLAIN PLAN Output: Example

The `INTO` clause of `EXPLAIN PLAN` specifies a different table in which to store the output.

If you do not want to use the name `PLAN_TABLE`, create a new synonym after running the `catplan.sql` script. For example:

```
CREATE OR REPLACE PUBLIC SYNONYM my_plan_table for plan_table$
```

The following statement directs output to `my_plan_table`:

```
EXPLAIN PLAN
  INTO my_plan_table FOR
  SELECT last_name FROM employees;
```

You can specify a statement ID when using the `INTO` clause, as in the following statement:

```
EXPLAIN PLAN
   SET STATEMENT_ID = 'st1'
   INTO my_plan_table FOR
   SELECT last_name FROM employees;
```

See Also:

- "PLAN_TABLE Columns" for a description of the columns in `PLAN_TABLE`
- *Oracle Database SQL Language Reference* to learn about `CREATE SYNONYM`

6.2.5 EXPLAIN PLAN Output for a CONTAINERS Query: Example

The `CONTAINERS` clause can be used to query both user-created and Oracle-supplied tables and views. It enables you to query these tables and views across all containers.

The following example illustrates the output of an `EXPLAIN PLAN` for a query using the `CONTAINERS` clause.

```
SQL> explain plan for select con_id, count(*) from
containers(sys.dba_tables) where con_id < 10 group by con_id order by
con_id;

Explained.

SQL> @?/rdbms/admin/utlxpls

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
---------------------------------------
Plan hash value: 891225627

-----------------------------------------------------------------------
---------------------------------------
| Id  | Operation                        | Name       | Rows   | Bytes
| Cost (%CPU)| Time       | Pstart| Pstop |
-----------------------------------------------------------------------
---------------------------------------
|   0 | SELECT STATEMENT                 |            |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   1 |  PX COORDINATOR                  |            |       |
|             |            |       |       |
|   2 |   PX SEND QC (ORDER)             | :TQ10001   |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   3 |    SORT GROUP BY                 |            |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   4 |     PX RECEIVE                   |            |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   5 |      PX SEND RANGE               | :TQ10000   |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   6 |       HASH GROUP BY              |            |   234K|
2970K|   145 (100)| 00:00:01 |       |       |
|   7 |        PX PARTITION LIST ITERATOR|            |   234K|
2970K|   139 (100)| 00:00:01 |     1 |     9 |
|   8 |         CONTAINERS FULL          | DBA_TABLES |   234K|
2970K|   139 (100)| 00:00:01 |       |       |
-------------------------------------------------------------------------------
----------------------------------

15 rows selected.
```

At Row 8 of this plan, `CONTAINERS` is shown in the `Operation` column as the value `CONTAINERS FULL`. The Name column in the same row shows the argument to `CONTAINERS`.

Default Partitioning

A query using the `CONTAINERS` clause is partitioned by default. At Row 7 in the plan, the `PX PARTITION LIST ITERATOR` in the `Operation` column indicates that the query is partitioned. Iteration over containers is implemented in this partition iterator. On the same row, the `Pstart` and `Pstop` values 1 and 9 are derived from the `con_id < 10` predicate in the query.

Default Parallelism

A query using the `CONTAINERS` clause uses parallel execution servers by default. In Row 1 of the plan above, `PX COORDINATOR` in the `Operation` column indicates that parallel execution servers will be used. Each container is assigned to a parallel execution process (`P00*`). When the parallel execution process executes the part of the query `EXECUTION PLAN` that corresponds to `CONTAINERS FULL`, then the process switches into the container it has been assigned to work on. It retrieves rows from the base object by executing a recursive SQL statement.

6.3 Displaying Execution Plans

The easiest way to display execution plans is to use `DBMS_XPLAN` display functions or `V$` views.

6.3.1 About the Display of PLAN_TABLE Output

To display the plan table output, you can use either SQL scripts or the `DBMS_XPLAN` package.

After you have explained the plan, use the following SQL scripts or PL/SQL package provided by Oracle Database to display the most recent plan table output:

- `DBMS_XPLAN.DISPLAY` table function

  This function accepts options for displaying the plan table output. You can specify:

  - A plan table name if you are using a table different than `PLAN_TABLE`
  - A statement ID if you have set a statement ID with the `EXPLAIN PLAN`
  - A format option that determines the level of detail: `BASIC`, `SERIAL`, `TYPICAL`, and `ALL`

  Examples of using `DBMS_XPLAN` to display `PLAN_TABLE` output are:

  ```
  SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

  SELECT PLAN_TABLE_OUTPUT
    FROM TABLE(DBMS_XPLAN.DISPLAY('MY_PLAN_TABLE', 'st1','TYPICAL'));
  ```

- `utlxpls.sql`

This script displays the plan table output for serial processing

- `utlxplp.sql`

  This script displays the plan table output including parallel execution columns.

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about the `DBMS_XPLAN` package

6.3.1.1 DBMS_XPLAN Display Functions

You can use the `DBMS_XPLAN` display functions to show plans.

The display functions accept options for displaying the plan table output. You can specify:

- A plan table name if you are using a table different from `PLAN_TABLE`
- A statement ID if you have set a statement ID with the `EXPLAIN PLAN`
- A format option that determines the level of detail: `BASIC`, `SERIAL`, `TYPICAL`, `ALL`, and in some cases `ADAPTIVE`

Table 6-1 DBMS_XPLAN Display Functions

Display Functions	Notes
`DISPLAY`	This table function displays the contents of the plan table. In addition, you can use this table function to display any plan (with or without statistics) stored in a table as long as the columns of this table are named the same as columns of the plan table (or `V$SQL_PLAN_STATISTICS_ALL` if statistics are included). You can apply a predicate on the specified table to select rows of the plan to display. The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, and `ALL`.
`DISPLAY_AWR`	This table function displays the contents of an execution plan stored in AWR. The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, and `ALL`.

Table 6-1 (Cont.) DBMS_XPLAN Display Functions

Display Functions	Notes
`DISPLAY_CURSOR`	This table function displays the explain plan of any cursor loaded in the cursor cache. In addition to the explain plan, various plan statistics (such as. I/O, memory and timing) can be reported (based on the `V$SQL_PLAN_STATISTICS_ALL VIEWS`). The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, `ALL`, and `ADAPTIVE`. When you specify `ADAPTIVE`, the output includes: • The final plan. If the execution has not completed, then the output shows the current plan. This section also includes notes about run-time optimizations that affect the plan. • Recommended plan. In reporting mode, the output includes the plan that would be chosen based on execution statistics. • Dynamic plan. The output summarizes the portions of the plan that differ from the default plan chosen by the optimizer. • Reoptimization. The output displays the plan that would be chosen on a subsequent execution because of reoptimization.
`DISPLAY_PLAN`	This table function displays the contents of the plan table in a variety of formats with `CLOB` output type. The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, `ALL`, and `ADAPTIVE`. When you specify `ADAPTIVE`, the output includes the default plan. For each dynamic subplan, the plan shows a list of the row sources from the original that may be replaced, and the row sources that would replace them. If the `format` argument specifies the outline display, then the function displays the hints for each option in the dynamic subplan. If the plan is not an adaptive query plan, then the function displays the default plan. When you do not specify `ADAPTIVE`, the plan is shown as-is, but with additional comments in the Note section that show any row sources that are dynamic.
`DISPLAY_SQL_PLAN_BASELINE`	This table function displays one or more execution plans for the specified SQL handle of a SQL plan baseline. This function uses plan information stored in the plan baseline to explain and display the plans. The `plan_id` stored in the SQL management base may not match the `plan_id` of the generated plan. A mismatch between the stored `plan_id` and generated `plan_id` means that it is a non-reproducible plan. Such a plan is deemed invalid and is bypassed by the optimizer during SQL compilation.
`DISPLAY_SQLSET`	This table function displays the execution plan of a given statement stored in a SQL tuning set. The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, and `ALL`.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_XPLAN` display functions

6.3.1.2 Plan-Related Views

You can obtain information about execution plans by querying dynamic performance and data dictionary views.

Table 6-2 Execution Plan Views

View	Description
`V$SQL`	Lists statistics for cursors and contains one row for each child of the original SQL text entered. Starting in Oracle Database 19c, `V$SQL.QUARANTINED` indicates whether a statement has been terminated by the Resource Manager because the statement consumed too many resources. Oracle Database records and marks the quarantined plans and prevents the execution of statements using these plans from executing. The `AVOIDED_EXECUTIONS` column indicates the number of executions attempted but prevented because of the quarantined statement.
`V$SQL_SHARED_CURSOR`	Explains why a particular child cursor is not shared with existing child cursors. Each column identifies a specific reason why the cursor cannot be shared. The `USE_FEEDBACK_STATS` column shows whether a child cursor fails to match because of reoptimization.
`V$SQL_PLAN`	Contains the plan for every statement stored in the shared SQL area. The view definition is similar to `PLAN_TABLE`. The view includes a superset of all rows appearing in all final plans. `PLAN_LINE_ID` is consecutively numbered, but for a single final plan, the IDs may not be consecutive. As an alternative to `EXPLAIN PLAN`, you can display the plan by querying `V$SQL_PLAN`. The advantage of `V$SQL_PLAN` over `EXPLAIN PLAN` is that you do not need to know the compilation environment that was used to execute a particular statement. For `EXPLAIN PLAN`, you would need to set up an identical environment to get the same plan when executing the statement.
`V$SQL_PLAN_STATISTICS`	Provides the actual execution statistics for every operation in the plan, such as the number of output rows and elapsed time. All statistics, except the number of output rows, are cumulative. For example, the statistics for a join operation also includes the statistics for its two inputs. The statistics in `V$SQL_PLAN_STATISTICS` are available for cursors that have been compiled with the `STATISTICS_LEVEL` initialization parameter set to `ALL`.

Table 6-2 (Cont.) Execution Plan Views

View	Description
`V$SQL_PLAN_STATISTICS_ALL`	Contains memory usage statistics for row sources that use SQL memory (sort or hash join). This view concatenates information in `V$SQL_PLAN` with execution statistics from `V$SQL_PLAN_STATISTICS` and `V$SQL_WORKAREA`. `V$SQL_PLAN_STATISTICS_ALL` enables side-by-side comparisons of the estimates that the optimizer provides for the number of rows and elapsed time. This view combines both `V$SQL_PLAN` and `V$SQL_PLAN_STATISTICS` information for every cursor.

See Also:

- "PLAN_TABLE Columns"
- "Monitoring Database Operations " for information about the `V$SQL_PLAN_MONITOR` view
- *Oracle Database Reference* for more information about `V$SQL_PLAN` views
- *Oracle Database Reference* for information about the `STATISTICS_LEVEL` initialization parameter

6.3.2 Displaying Execution Plans: Basic Steps

The `DBMS_XPLAN.DISPLAY` function is a simple way to display an explained plan.

By default, the `DISPLAY` function uses the format setting of `TYPICAL`. In this case, the plan the most relevant information in the plan: operation id, name and option, rows, bytes and optimizer cost. Pruning, parallel and predicate information are only displayed when applicable.

To display an execution plan:

1. Start SQL*Plus or SQL Developer and log in to the session in which you explained the plan.
2. Explain a plan.
3. Query `PLAN_TABLE` using `DBMS_XPLAN.DISPLAY`.

   Specify the query as follows:

   ```
   SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY);
   ```

Alternatively, specify the statement ID using the `statement_id` parameter:

```
SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY(statement_id
=> 'statement_id));
```

Example 6-2   EXPLAIN PLAN for Statement ID ex_plan1

This example explains a query of `employees` that uses the statement ID `ex_plan1`, and then queries `PLAN_TABLE`:

```
EXPLAIN PLAN
  SET statement_id = 'ex_plan1' FOR
  SELECT phone_number
  FROM   employees
  WHERE  phone_number LIKE '650%';

SELECT PLAN_TABLE_OUTPUT
  FROM TABLE(DBMS_XPLAN.DISPLAY(statement_id => 'ex_plan1'));
```

Sample output appears below:

```
Plan hash value: 1445457117

-----------------------------------------------------------------------
----
|Id | Operation         | Name       |Rows | Bytes | Cost (%CPU)|
Time     |
-----------------------------------------------------------------------
----
|  0| SELECT STATEMENT  |            |   1 |    15 |      2   (0)|
00:00:01 |
|* 1|  TABLE ACCESS FULL| EMPLOYEES |   1 |    15 |      2   (0)|
00:00:01 |
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("PHONE_NUMBER" LIKE '650%')
```

Example 6-3   EXPLAIN PLAN for Statement ID ex_plan2

This example explains a query of `employees` that uses the statement ID `ex_plan2`, and then displays the plan using the `BASIC` format:

```
EXPLAIN PLAN
  SET statement_id = 'ex_plan2' FOR
  SELECT last_name
  FROM   employees
  WHERE  last_name LIKE 'Pe%';
```

```
SELECT PLAN_TABLE_OUTPUT
  FROM TABLE(DBMS_XPLAN.DISPLAY(NULL, 'ex_plan2','BASIC'));
```

Sample output appears below:

```
----------------------------------------
| Id  | Operation         | Name         |
----------------------------------------
|   0 | SELECT STATEMENT |              |
|   1 |  INDEX RANGE SCAN| EMP_NAME_IX |
----------------------------------------
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about the DBMS_XPLAN package

6.3.3 Displaying Adaptive Query Plans: Tutorial

The **adaptive optimizer** is a feature of the optimizer that enables it to adapt plans based on run-time statistics. All adaptive mechanisms can execute a final plan for a statement that differs from the default plan.

An adaptive query plan chooses among subplans *during* the current statement execution. In contrast, automatic reoptimization changes a plan only on executions that occur *after* the current statement execution.

You can determine whether the database used adaptive query optimization for a SQL statement based on the comments in the Notes section of plan. The comments indicate whether row sources are dynamic, or whether automatic reoptimization adapted a plan.

Assumptions

This tutorial assumes the following:

- The STATISTICS_LEVEL initialization parameter is set to ALL.
- The database uses the default settings for adaptive execution.
- As user oe, you want to issue the following separate queries:

```
SELECT o.order_id, v.product_name
FROM   orders o,
       (  SELECT order_id, product_name
          FROM   order_items o, product_information p
          WHERE  p.product_id = o.product_id
          AND    list_price < 50
          AND    min_price < 40  ) v
WHERE  o.order_id = v.order_id

SELECT product_name
FROM   order_items o, product_information p
WHERE  o.unit_price = 15
```

```
AND    quantity > 1
AND    p.product_id = o.product_id
```

- Before executing each query, you want to query `DBMS_XPLAN.DISPLAY_PLAN` to see the default plan, that is, the plan that the optimizer chose before applying its adaptive mechanism.
- After executing each query, you want to query `DBMS_XPLAN.DISPLAY_CURSOR` to see the final plan and adaptive query plan.
- `SYS` has granted `oe` the following privileges:
  - `GRANT SELECT ON V_$SESSION TO oe`
  - `GRANT SELECT ON V_$SQL TO oe`
  - `GRANT SELECT ON V_$SQL_PLAN TO oe`
  - `GRANT SELECT ON V_$SQL_PLAN_STATISTICS_ALL TO oe`

To see the results of adaptive optimization:

1. Start SQL*Plus, and then connect to the database as user `oe`.
2. Query `orders`.

   For example, use the following statement:

   ```
   SELECT o.order_id, v.product_name
   FROM   orders o,
          (  SELECT order_id, product_name
             FROM   order_items o, product_information p
             WHERE  p.product_id = o.product_id
             AND    list_price < 50
             AND    min_price < 40  ) v
   WHERE  o.order_id = v.order_id;
   ```

3. View the plan in the cursor.

   For example, run the following commands:

   ```
   SET LINESIZE 165
   SET PAGESIZE 0
   SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'+ALLSTATS'));
   ```

   The following sample output has been reformatted to fit on the page. In this plan, the optimizer chooses a nested loops join. The original optimizer estimates are shown in the `E-Rows` column, whereas the actual statistics gathered during execution are shown in the `A-Rows` column. In the `MERGE JOIN` operation, the difference between the estimated and actual number of rows is significant.

```
---------------------------------------------------------------------------------------------
|Id| Operation              | Name           |Start|E-Rows|A-Rows|A-Time|Buff|OMem|1Mem|O/1/M|
---------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT       |                |    1|   | 269|00:00:00.09|1338|    |    |     |
| 1|  NESTED LOOPS          |                |    1|  1| 269|00:00:00.09|1338|    |    |     |
| 2|   MERGE JOIN CARTESIAN |                |    1|  4|9135|00:00:00.03|  33|    |    |     |
|*3|    TABLE ACCESS FULL   |PRODUCT_INFORMAT|    1|  1|  87|00:00:00.01|  32|    |    |     |
| 4|    BUFFER SORT         |                |   87|105|9135|00:00:00.01|   1|4096|4096|1/0/0|
| 5|     INDEX FULL SCAN    | ORDER_PK       |    1|105| 105|00:00:00.01|   1|    |    |     |
```

```
|*6|   INDEX UNIQUE SCAN   | ORDER_ITEMS_UK |9135|  1| 269|00:00:00.03|1305|    |     |     |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter(("MIN_PRICE"<40 AND "LIST_PRICE"<50))
   6 - access("O"."ORDER_ID"="ORDER_ID" AND "P"."PRODUCT_ID"="O"."PRODUCT_ID")
```

4. Run the same query of `orders` that you ran in Step 2.
5. View the execution plan in the cursor by using the same `SELECT` statement that you ran in Step 3.

   The following example shows that the optimizer has chosen a different plan, using a hash join. The Note section shows that the optimizer used statistics feedback to adjust its cost estimates for the second execution of the query, thus illustrating automatic reoptimization.

```
---------------------------------------------------------------------------------------------
|Id| Operation              |Name    |Start|E-Rows|A-Rows|A-Time|Buff|Reads|OMem|1Mem|O/1/M|
---------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT       |                | 1 |   |269|00:00:00.02|60|1|     |     |     |
| 1|  NESTED LOOPS          |                | 1 |269|269|00:00:00.02|60|1|     |     |     |
|*2|   HASH JOIN            |                | 1 |313|269|00:00:00.02|39|1|1000K|1000K|1/0/0|
|*3|    TABLE ACCESS FULL   |PRODUCT_INFORMA| 1 | 87| 87|00:00:00.01|15|0|     |     |     |
| 4|    INDEX FAST FULL SCAN|ORDER_ITEMS_UK | 1 |665|665|00:00:00.01|24|1|     |     |     |
|*5|   INDEX UNIQUE SCAN    |ORDER_PK       |269|  1|269|00:00:00.01|21|0|     |     |     |
---------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
   3 - filter(("MIN_PRICE"<40 AND "LIST_PRICE"<50))
   5 - access("O"."ORDER_ID"="ORDER_ID")

Note
-----
   - statistics feedback used for this statement
```

6. Query `V$SQL` to verify the performance improvement.

   The following query shows the performance of the two statements (sample output included).

   ```
   SELECT CHILD_NUMBER, CPU_TIME, ELAPSED_TIME, BUFFER_GETS
   FROM   V$SQL
   WHERE  SQL_ID = 'gm2npz344xqn8';

   CHILD_NUMBER   CPU_TIME ELAPSED_TIME BUFFER_GETS
   ------------ ---------- ------------ -----------
              0      92006       131485        1831
              1      12000        24156          60
   ```

   The second statement executed, which is child number `1`, used statistics feedback. CPU time, elapsed time, and buffer gets are all significantly lower.
7. Explain the plan for the query of `order_items`.

For example, use the following statement:

```
EXPLAIN PLAN FOR
  SELECT product_name
  FROM   order_items o, product_information p
  WHERE  o.unit_price = 15
  AND    quantity > 1
  AND    p.product_id = o.product_id
```

8. View the plan in the plan table.

   For example, run the following statement:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

   Sample output appears below:

```
-------------------------------------------------------------------------------
|Id| Operation                  | Name                |Rows|Bytes|Cost (%CPU)|Time|
-------------------------------------------------------------------------------
| 0| SELECT STATEMENT             |                      |4|128|7 (0)|00:00:01|
| 1|  NESTED LOOPS                |                      | |   |     |        |
| 2|   NESTED LOOPS               |                      |4|128|7 (0)|00:00:01|
|*3|    TABLE ACCESS FULL         |ORDER_ITEMS           |4|48 |3 (0)|00:00:01|
|*4|    INDEX UNIQUE SCAN         |PRODUCT_INFORMATION_PK|1|   |0 (0)|00:00:01|
| 5|   TABLE ACCESS BY INDEX ROWID|PRODUCT_INFORMATION   |1|20 |1 (0)|00:00:01|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("O"."UNIT_PRICE"=15 AND "QUANTITY">1)
   4 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
```

   In this plan, the optimizer chooses a nested loops join.

9. Run the query that you previously explained.

   For example, use the following statement:

```
SELECT product_name
FROM   order_items o, product_information p
WHERE  o.unit_price = 15
AND    quantity > 1
AND    p.product_id = o.product_id
```

10. View the plan in the cursor.

    For example, run the following commands:

```
SET LINESIZE 165
SET PAGESIZE 0
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(FORMAT=>'+ADAPTIVE'));
```

Sample output appears below. Based on statistics collected at run time (Step 4), the optimizer chose a hash join rather than the nested loops join. The dashes (-) indicate the steps in the nested loops plan that the optimizer considered but do not ultimately choose. The switch illustrates the adaptive query plan feature.

```
-------------------------------------------------------------------------------
|Id | Operation                          | Name         |Rows|Bytes|Cost(%CPU)|Time     |
-------------------------------------------------------------------------------
|  0| SELECT STATEMENT                   |                      |4|128|7(0)|00:00:01|
| *1|  HASH JOIN                         |                      |4|128|7(0)|00:00:01|
|- 2|   NESTED LOOPS                     |                      | |   |    |        |
|- 3|    NESTED LOOPS                    |                      | |128|7(0)|00:00:01|
|- 4|     STATISTICS COLLECTOR           |                      | |   |    |        |
| *5|      TABLE ACCESS FULL             | ORDER_ITEMS          |4| 48|3(0)|00:00:01|
|-*6|     INDEX UNIQUE SCAN              | PRODUCT_INFORMATI_PK|1|   |0(0)|00:00:01|
|- 7|    TABLE ACCESS BY INDEX ROWID| PRODUCT_INFORMATION |1| 20|1(0)|00:00:01|
|  8|   TABLE ACCESS FULL                | PRODUCT_INFORMATION |1| 20|1(0)|00:00:01|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")
   5 - filter("O"."UNIT_PRICE"=15 AND "QUANTITY">1)
   6 - access("P"."PRODUCT_ID"="O"."PRODUCT_ID")

Note
-----
   - this is an adaptive plan (rows marked '-' are inactive)
```

See Also:

- "Adaptive Query Plans"
- "Table 6-1"
- "Controlling Adaptive Optimization"
- *Oracle Database Reference* to learn about the `STATISTICS_LEVEL` initialization parameter
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_XPLAN`

6.3.4 Display Execution Plans: Examples

These examples show different ways of displaying execution plans.

6.3.4.1 Customizing PLAN_TABLE Output

If you have specified a statement identifier, then you can write your own script to query the `PLAN_TABLE`.

For example:

- Start with ID = 0 and given `STATEMENT_ID`.
- Use the `CONNECT BY` clause to walk the tree from parent to child, the join keys being `STATEMENT_ID = PRIOR STATMENT_ID` and `PARENT_ID = PRIOR ID`.
- Use the pseudo-column `LEVEL` (associated with `CONNECT BY`) to indent the children.

```
SELECT  cardinality "Rows", lpad(' ',level-1) || operation
        ||' '||options||' '||object_name "Plan"
FROM    PLAN_TABLE
CONNECT BY prior id = parent_id
        AND prior statement_id = statement_id
  START WITH id = 0
        AND statement_id = 'st1'
  ORDER BY id;

   Rows Plan
------- ----------------------------------------
        SELECT STATEMENT
         TABLE ACCESS FULL EMPLOYEES
```

The `NULL` in the `Rows` column indicates that the optimizer does not have any statistics on the table. Analyzing the table shows the following:

```
   Rows Plan
------- ----------------------------------------
  16957 SELECT STATEMENT
  16957  TABLE ACCESS FULL EMPLOYEES
```

You can also select the `COST`. This is useful for comparing execution plans or for understanding why the optimizer chooses one execution plan over another.

Note:

These simplified examples are not valid for recursive SQL.

6.3.4.2 Displaying Parallel Execution Plans: Example

Plans for parallel queries differ in important ways from plans for serial queries.

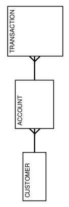
6.3.4.2.1 About EXPLAIN PLAN and Parallel Queries

Tuning a parallel query begins much like a non-parallel query tuning exercise by choosing the driving table. However, the rules governing the choice are different.

In the serial case, the best driving table produces the fewest numbers of rows after applying limiting conditions. The database joins a small number of rows to larger tables using non-unique indexes.

For example, consider a table hierarchy consisting of `customer`, `account`, and `transaction`.

Figure 6-1   A Table Hierarchy



In this example, `customer` is the smallest table, whereas `transaction` is the largest table. A typical OLTP query retrieves transaction information about a specific customer account. The query drives from the `customer` table. The goal is to minimize logical I/O, which typically minimizes other critical resources including physical I/O and CPU time.

For parallel queries, the driving table is usually the *largest* table. It would not be efficient to use parallel query in this case because only a few rows from each table are accessed. However, what if it were necessary to identify all customers who had transactions of a certain type last month? It would be more efficient to drive from the `transaction` table because no limiting conditions exist on the `customer` table. The database would join rows from the `transaction` table to the `account` table, and then finally join the result set to the `customer` table. In this case, the used on the `account` and `customer` table are probably highly selective primary key or unique indexes rather than the non-unique indexes used in the first query. Because the `transaction` table is large and the column is not selective, it would be beneficial to use parallel query driving from the `transaction` table.

Parallel operations include the following:

- `PARALLEL_TO_PARALLEL`
- `PARALLEL_TO_SERIAL`

  A `PARALLEL_TO_SERIAL` operation is always the step that occurs when the query coordinator consumes rows from a parallel operation. Another type of operation that does not occur in this query is a `SERIAL` operation. If these types of operations occur, then consider making them parallel operations to improve performance because they too are potential bottlenecks.
- `PARALLEL_FROM_SERIAL`
- `PARALLEL_TO_PARALLEL`

  If the workloads in each step are relatively equivalent, then the `PARALLEL_TO_PARALLEL` operations generally produce the best performance.
- `PARALLEL_COMBINED_WITH_CHILD`
- `PARALLEL_COMBINED_WITH_PARENT`

  A `PARALLEL_COMBINED_WITH_PARENT` operation occurs when the database performs the step simultaneously with the parent step.

If a parallel step produces many rows, then the QC may not be able to consume the rows as fast as they are produced. Little can be done to improve this situation.

See Also:

The OTHER_TAG column in "PLAN_TABLE Columns"

6.3.4.2.2 Viewing Parallel Queries with EXPLAIN PLAN: Example

When using EXPLAIN PLAN with parallel queries, the database compiles and executes one parallel plan. This plan is derived from the serial plan by allocating row sources specific to the parallel support in the QC plan.

The table queue row sources (PX Send and PX Receive), the granule iterator, and buffer sorts, required by the two parallel execution server set PQ model, are directly inserted into the parallel plan. This plan is the same plan for all parallel execution servers when executed in parallel or for the QC when executed serially.

Example 6-4 Parallel Query Explain Plan

The following simple example illustrates an EXPLAIN PLAN for a parallel query:

```
CREATE TABLE emp2 AS SELECT * FROM employees;

ALTER TABLE emp2 PARALLEL 2;

EXPLAIN PLAN FOR
  SELECT SUM(salary)
  FROM   emp2
  GROUP BY department_id;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
-------------------------------------------------------------------------------------
|Id | Operation              | Name    |Rows| Bytes |Cost %CPU| TQ  |IN-OUT|PQ Distrib|
-------------------------------------------------------------------------------------
|0| SELECT STATEMENT         |        |107| 2782 | 3 (34) |       |      |          |
|1|  PX COORDINATOR          |        |   |      |        |       |      |          |
|2|   PX SEND QC (RANDOM)    |:TQ10001|107| 2782 | 3 (34) | Q1,01 | P->S |QC (RAND) |
|3|    HASH GROUP BY         |        |107| 2782 | 3 (34) | Q1,01 | PCWP |          |
|4|     PX RECEIVE           |        |107| 2782 | 3 (34) | Q1,01 | PCWP |          |
|5|      PX SEND HASH        |:TQ10000|107| 2782 | 3 (34) | Q1,00 | P->P |HASH      |
|6|       HASH GROUP BY      |        |107| 2782 | 3 (34) | Q1,00 | PCWP |          |
|7|        PX BLOCK ITERATOR |        |107| 2782 | 2 (0)  | Q1,00 | PCWP |          |
|8|         TABLE ACCESS FULL|EMP2    |107| 2782 | 2 (0)  | Q1,00 | PCWP |          |
-------------------------------------------------------------------------------------
```

One set of parallel execution servers scans EMP2 in parallel, while the second set performs the aggregation for the GROUP BY operation. The PX BLOCK ITERATOR row source represents the splitting up of the table EMP2 into pieces to divide the scan workload between the parallel execution servers. The PX SEND and PX RECEIVE row sources represent the pipe that connects the two sets of parallel execution servers as rows flow up from the parallel scan, get repartitioned through the HASH table queue, and then read by and aggregated on the top set. The PX SEND QC row source represents the aggregated values being sent to the QC in random (RAND) order. The

PX COORDINATOR row source represents the QC or Query Coordinator which controls and schedules the parallel plan appearing below it in the plan tree.

6.3.4.3 Displaying Bitmap Index Plans: Example

Index row sources using bitmap indexes appear in the EXPLAIN PLAN output with the word BITMAP indicating the type of the index.

Example 6-5 EXPLAIN PLAN with Bitmap Indexes

In this example, the predicate c1=2 yields a bitmap from which a subtraction can take place. From this bitmap, the bits in the bitmap for c2=6 are subtracted. Also, the bits in the bitmap for c2 IS NULL are subtracted, explaining why there are two MINUS row sources in the plan. The NULL subtraction is necessary for semantic correctness unless the column has a NOT NULL constraint. The TO ROWIDS option generates the rowids necessary for the table access.

Note:

Queries using bitmap join index indicate the bitmap join index access path. The operation for bitmap join index is the same as bitmap index.

```
EXPLAIN PLAN FOR  SELECT *
  FROM   t
  WHERE  c1 = 2
  AND    c2 <> 6
  OR     c3 BETWEEN 10 AND 20;

SELECT STATEMENT
   TABLE ACCESS T BY INDEX ROWID
      BITMAP CONVERSION TO ROWID
         BITMAP OR
            BITMAP MINUS
               BITMAP MINUS
                  BITMAP INDEX C1_IND SINGLE VALUE
                  BITMAP INDEX C2_IND SINGLE VALUE
               BITMAP INDEX C2_IND SINGLE VALUE
            BITMAP MERGE
               BITMAP INDEX C3_IND RANGE SCAN
```

6.3.4.4 Displaying Result Cache Plans: Example

When your query contains the result_cache hint, the ResultCache operator is inserted into the execution plan.

Starting in Oracle Database 21c, the result_cache hint accepts a new option: result_cache(TEMP={TRUE|FALSE}). A value of TRUE enables the query to spill to disk, whereas FALSE prevents a Temp object from being formed. Instead, the Result object will enter the ’Bypass’ status.

For example, you might explain a query as follows:

```
EXPLAIN PLAN FOR
SELECT /*+ result_cache(TEMP=TRUE) */ department_id, AVG(salary)
FROM   employees
GROUP BY department_id;
SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY(format =>
'ALL'));
```

The `EXPLAIN PLAN` output for this query includes a `Result Cache Information` section, and should look similar to the following:

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
-----
Plan hash value: 1192169904

-------------------------------------------------------------------------------------
----
| Id  | Operation             | Name                       | Rows  |Bytes|Cost (%CPU)|
Time|
-------------------------------------------------------------------------------------
----
|   0 | SELECT STATEMENT     |                            |  11 |   77 | 4   (25)|
00:00:01 |
|   1 |  RESULT CACHE        | ch5r45jxt05rk0xc1brct197fp |  11 |   77 | 4   (25)|
00:00:01 |
|   2 |   HASH GROUP BY      |                            |  11 |   77 | 4   (25)|
00:00:01 |
|   3 |    TABLE ACCESS FULL| EMPLOYEES                  | 107 | 749 | 3    (0)|
00:00:01 |
-------------------------------------------------------------------------------------
----

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1
   3 - SEL$1 / "EMPLOYEES"@"SEL$1"

Column Projection Information (identified by operation id):
-----------------------------------------------------------
   1 - "DEPARTMENT_ID"[NUMBER,22], SUM("SALARY")/COUNT("SALARY")[22]
   2 - (#keys=1) "DEPARTMENT_ID"[NUMBER,22], COUNT("SALARY")[22], SUM("SALARY")[22]
   3 - (rowset=256) "SALARY"[NUMBER,22], "DEPARTMENT_ID"[NUMBER,22]

Result Cache Information (identified by operation id):
------------------------------------------------------
   1 - column-count=2; dependencies=(HR.EMPLOYEES);
name="SELECT /*+ result_cache(TEMP=TRUE) */ department_id, AVG(salary)
FROM   employees
GROUP BY department_id"
```

In this plan, the `RESULT CACHE` operations is identified by its cache ID, which is `ch5r45jxt05rk0xc1brct197fp`. You can query the `V$RESULT_CACHE_OBJECTS` view by using this `CACHE_ID`, as shown in the following example (sample output included):

```
SELECT SUBCACHE_ID, TYPE, STATUS, BLOCK_COUNT,
       ROW_COUNT, INVALIDATIONS
FROM   V$RESULT_CACHE_OBJECTS
WHERE  CACHE_ID = 'ch5r45jxt05rk0xc1brct197fp';

SUBCACHE_ID TYPE       STATUS    BLOCK_COUNT  ROW_COUNT INVALIDATIONS
----------- ---------- --------- ----------- ---------- -------------
          0 Result     Published           1         12             0
```

6.3.4.5 Displaying Plans for Partitioned Objects: Example

Use `EXPLAIN PLAN` to determine how Oracle Database accesses partitioned objects for specific queries.

Partitions accessed after pruning are shown in the `PARTITION START` and `PARTITION STOP` columns. The row source name for the range partition is `PARTITION RANGE`. For hash partitions, the row source name is `PARTITION HASH`.

A join is implemented using partial partition-wise join if the `DISTRIBUTION` column of the plan table of one of the joined tables contains `PARTITION(KEY)`. Partial partition-wise join is possible if one of the joined tables is partitioned on its join column and the table is parallelized.

A join is implemented using full partition-wise join if the partition row source appears before the join row source in the `EXPLAIN PLAN` output. Full partition-wise joins are possible only if both joined tables are equipartitioned on their respective join columns. Examples of execution plans for several types of partitioning follow.

6.3.4.5.1 Displaying Range and Hash Partitioning with EXPLAIN PLAN: Examples

This example illustrates pruning by using the `emp_range` table, which partitioned by range on `hire_date`.

Assume that the tables `employees` and `departments` from the Oracle Database sample schema exist.

```
CREATE TABLE emp_range
PARTITION BY RANGE(hire_date)
(
  PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1992','DD-MON-YYYY')),
  PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1994','DD-MON-YYYY')),
  PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1996','DD-MON-YYYY')),
  PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1998','DD-MON-YYYY')),
  PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-2001','DD-MON-YYYY'))
)
AS SELECT * FROM employees;
```

For the first example, consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT * FROM emp_range;
```

Oracle Database displays something similar to the following:

```
--------------------------------------------------------------------
|Id| Operation           | Name      |Rows| Bytes|Cost|Pstart|Pstop|
--------------------------------------------------------------------
| 0| SELECT STATEMENT    |           |  105| 13965 | 2 |    |       |
| 1|  PARTITION RANGE ALL|           |  105| 13965 | 2 | 1 |      5 |
| 2|   TABLE ACCESS FULL | EMP_RANGE |  105| 13965 | 2 | 1 |      5 |
--------------------------------------------------------------------
```

The database creates a partition row source on top of the table access row source. It iterates over the set of partitions to be accessed. In this example, the partition iterator covers all partitions (option `ALL`), because a predicate was not used for pruning. The `PARTITION_START` and `PARTITION_STOP` columns of the `PLAN_TABLE` show access to all partitions from 1 to 5.

For the next example, consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   emp_range
  WHERE  hire_date >= TO_DATE('1-JAN-1996','DD-MON-YYYY');
```

```
-----------------------------------------------------------------------
| Id | Operation                | Name    |Rows|Bytes|Cost|Pstart|Pstop|
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT         |         | 3 | 399 |    2 |     |     |
|  1 |  PARTITION RANGE ITERATOR|         | 3 | 399 |    2 |    4 |    5 |
| *2 |   TABLE ACCESS FULL      |EMP_RANGE| 3 | 399 |    2 |    4 |    5 |
-----------------------------------------------------------------------
```

In the previous example, the partition row source iterates from partition 4 to 5 because the database prunes the other partitions using a predicate on `hire_date`.

Finally, consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   emp_range
  WHERE  hire_date < TO_DATE('1-JAN-1992','DD-MON-YYYY');
```

```
-----------------------------------------------------------------------
| Id  | Operation             | Name      |Rows|Bytes|Cost|Pstart|Pstop|
-----------------------------------------------------------------------
|   0 | SELECT STATEMENT       |           |    1 |   133 | 2 |   |   |
|   1 |  PARTITION RANGE SINGLE|           |    1 |   133 | 2 | 1 | 1 |
|*  2 |   TABLE ACCESS FULL    | EMP_RANGE |    1 |   133 | 2 | 1 | 1 |
-----------------------------------------------------------------------
```

In the previous example, only partition 1 is accessed and known at compile time; thus, there is no need for a partition row source.

Note:

Oracle Database displays the same information for hash partitioned objects, except the partition row source name is `PARTITION HASH` instead of `PARTITION RANGE`. Also, with hash partitioning, pruning is only possible using equality or `IN`-list predicates.

6.3.4.5.2 Pruning Information with Composite Partitioned Objects: Examples

To illustrate how Oracle Database displays pruning information for composite partitioned objects, consider the table `emp_comp`. It is range-partitioned on `hiredate` and subpartitioned by hash on `deptno`.

```
CREATE TABLE emp_comp PARTITION BY RANGE(hire_date)
      SUBPARTITION BY HASH(department_id) SUBPARTITIONS 3
(
PARTITION emp_p1 VALUES LESS THAN (TO_DATE('1-JAN-1992','DD-MON-YYYY')),
PARTITION emp_p2 VALUES LESS THAN (TO_DATE('1-JAN-1994','DD-MON-YYYY')),
PARTITION emp_p3 VALUES LESS THAN (TO_DATE('1-JAN-1996','DD-MON-YYYY')),
PARTITION emp_p4 VALUES LESS THAN (TO_DATE('1-JAN-1998','DD-MON-YYYY')),
PARTITION emp_p5 VALUES LESS THAN (TO_DATE('1-JAN-2001','DD-MON-YYYY'))
)
AS SELECT * FROM employees;
```

For the first example, consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT * FROM emp_comp;
```

```
-----------------------------------------------------------------------
|Id| Operation             | Name      | Rows   | Bytes |Cost|Pstart|Pstop|
-----------------------------------------------------------------------
| 0| SELECT STATEMENT      |           | 10120 |  1314K| 78 |     |        |
| 1|  PARTITION RANGE ALL|           | 10120 |  1314K| 78 |   1 |      5 |
| 2|   PARTITION HASH ALL|           | 10120 |  1314K| 78 |   1 |      3 |
| 3|    TABLE ACCESS FULL| EMP_COMP | 10120 |  1314K| 78 |   1 |     15 |
-----------------------------------------------------------------------
```

This example shows the plan when Oracle Database accesses all subpartitions of all partitions of a composite object. The database uses two partition row sources for this purpose: a range partition row source to iterate over the partitions, and a hash partition row source to iterate over the subpartitions of each accessed partition.

In the following example, the range partition row source iterates from partition 1 to 5, because the database performs no pruning. Within each partition, the hash partition row source iterates over subpartitions 1 to 3 of the current partition. As a result, the table access row source accesses subpartitions 1 to 15. In other words, the database accesses all subpartitions of the composite object.

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   emp_comp
  WHERE  hire_date = TO_DATE('15-FEB-1998', 'DD-MON-YYYY');

-----------------------------------------------------------------------
| Id | Operation              | Name     |Rows|Bytes |Cost|Pstart|Pstop|
-----------------------------------------------------------------------
|  0 | SELECT STATEMENT       |          | 20 | 2660 | 17 |      |     |
|  1 |  PARTITION RANGE SINGLE|          | 20 | 2660 | 17 |    5 |   5 |
|  2 |   PARTITION HASH ALL   |          | 20 | 2660 | 17 |    1 |   3 |
|* 3 |    TABLE ACCESS FULL   | EMP_COMP | 20 | 2660 | 17 |   13 |  15 |
-----------------------------------------------------------------------
```

In the previous example, only the last partition, partition 5, is accessed. This partition is known at compile time, so the database does not need to show it in the plan. The hash partition row source shows accessing of all subpartitions within that partition; that is, subpartitions 1 to 3, which translates into subpartitions 13 to 15 of the `emp_comp` table.

Now consider the following statement:

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   emp_comp
  WHERE  department_id = 20;

-----------------------------------------------------------------------
-
| Id | Operation              |Name     |Rows | Bytes |Cost|Pstart|
Pstop|
-----------------------------------------------------------------------
-
|  0 | SELECT STATEMENT       |          | 101 | 13433 | 78 |    |
|
|  1 |  PARTITION RANGE ALL   |          | 101 | 13433 | 78 |  1 |   5
|
|  2 |   PARTITION HASH SINGLE|          | 101 | 13433 | 78 |  3 |   3
|
|* 3 |    TABLE ACCESS FULL   | EMP_COMP | 101 | 13433 | 78 |    |
|
-----------------------------------------------------------------------
-
```

In the previous example, the predicate `deptno=20` enables pruning on the hash dimension within each partition. Therefore, Oracle Database only needs to access a single subpartition. The number of this subpartition is known at compile time, so the hash partition row source is not needed.

Finally, consider the following statement:

```
VARIABLE dno NUMBER;
EXPLAIN PLAN FOR
  SELECT *
  FROM   emp_comp
  WHERE  department_id = :dno;

-----------------------------------------------------------------------
| Id| Operation                | Name     |Rows| Bytes |Cost|Pstart|Pstop|
-----------------------------------------------------------------------
| 0 | SELECT STATEMENT         |          | 101| 13433 | 78 |      |     |
| 1 |  PARTITION RANGE ALL     |          | 101| 13433 | 78 |    1 |   5 |
| 2 |   PARTITION HASH SINGLE|          | 101| 13433 | 78 | KEY  | KEY |
|*3 |    TABLE ACCESS FULL     | EMP_COMP | 101| 13433 | 78 |      |     |
-----------------------------------------------------------------------
```

The last two examples are the same, except that `department_id` = :dno replaces `deptno=20`. In this last case, the subpartition number is unknown at compile time, and a hash partition row source is allocated. The option is `SINGLE` for this row source because Oracle Database accesses only one subpartition within each partition. In Step 2, both `PARTITION_START` and `PARTITION_STOP` are set to `KEY`. This value means that Oracle Database determines the number of subpartitions at run time.

6.3.4.5.3 Examples of Partial Partition-Wise Joins

In these examples, the `PQ_DISTRIBUTE` hint explicitly forces a partial partition-wise join because the query optimizer could have chosen a different plan based on cost in this query.

Example 6-6 Partial Partition-Wise Join with Range Partition

In the following example, the database joins `emp_range_did` on the partitioning column `department_id` and parallelizes it. The database can use a partial partition-wise join because the `dept2` table is not partitioned. Oracle Database dynamically partitions the `dept2` table before the join.

```
CREATE TABLE dept2 AS SELECT * FROM departments;
ALTER TABLE dept2 PARALLEL 2;

CREATE TABLE emp_range_did PARTITION BY RANGE(department_id)
   (PARTITION emp_p1 VALUES LESS THAN (150),
    PARTITION emp_p5 VALUES LESS THAN (MAXVALUE) )
  AS SELECT * FROM employees;

ALTER TABLE emp_range_did PARALLEL 2;

EXPLAIN PLAN FOR
  SELECT /*+ PQ_DISTRIBUTE(d NONE PARTITION) ORDERED */ e.last_name,
         d.department_name
  FROM   emp_range_did e, dept2 d
  WHERE  e.department_id = d.department_id;

-------------------------------------------------------------------------------------------
|Id| Operation                    |Name    |Row|Byte|Cost|Pstart|Pstop|TQ|IN-OUT|PQ Distrib|
--------------------------------------------------------------------------------------
-----
| 0| SELECT STATEMENT            |              |284 |16188|6|  | |        |
|          |
| 1|  PX COORDINATOR             |              |    |     | |  | |        |
|          |
| 2|   PX SEND QC (RANDOM)       |:TQ10001      |284 |16188|6|  | | Q1,01 |P->S|QC
(RAND)  |
|*3|    HASH JOIN                |              |284 |16188|6|  | | Q1,01 |
PCWP|          |
| 4|     PX PARTITION RANGE ALL  |              |284 |7668 |2|1 |2| Q1,01 |
PCWC|          |
| 5|      TABLE ACCESS FULL      |EMP_RANGE_DID|284 |7668 |2|1 |2| Q1,01 |
PCWP|          |
| 6|     BUFFER SORT             |              |    |     | |  | | Q1,01 |
PCWC|          |
| 7|      PX RECEIVE             |              | 21 | 630 |2|  | | Q1,01 |
PCWP|          |
| 8|       PX SEND PARTITION (KEY)|:TQ10000     | 21 | 630 |2|  | |        |S->P|PART
(KEY)|
| 9|        TABLE ACCESS FULL    |DEPT2         | 21 | 630 |2|  | |        |
|          |
--------------------------------------------------------------------------------------
-----
```

The execution plan shows that the table `dept2` is scanned serially and all rows with the same partitioning column value of `emp_range_did (department_id)` are sent through a `PART (KEY)`, or partition key, table queue to the same parallel execution server doing the partial partition-wise join.

Example 6-7 Partial Partition-Wise Join with Composite Partition

In the following example, `emp_comp` is joined on the partitioning column and is parallelized, enabling use of a partial partition-wise join because `dept2` is not partitioned. The database dynamically partitions `dept2` before the join.

```
ALTER TABLE emp_comp PARALLEL 2;

EXPLAIN PLAN FOR
  SELECT /*+ PQ_DISTRIBUTE(d NONE PARTITION) ORDERED */ e.last_name,
         d.department_name
  FROM   emp_comp e, dept2 d
  WHERE  e.department_id = d.department_id;

SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());
```

```
--------------------------------------------------------------------------------------
-----
| Id| Operation              | Name  |Rows |Bytes |Cost|Pstart|Pstop|TQ |IN-OUT|PQ
Distrib|
--------------------------------------------------------------------------------------
-----
| 0 | SELECT STATEMENT             |        | 445 |17800| 5 |   |   |        |
|          |
| 1 |  PX COORDINATOR              |        |     |     |   |   |   |        |
|          |
| 2 |   PX SEND QC (RANDOM)       |:TQ10001| 445 |17800| 5 |  |   | Q1,01 |P->S| QC (RAND)|
|*3 |    HASH JOIN                |        | 445 |17800| 5 |  |   | Q1,01 |PCWP|          |
| 4 |     PX PARTITION RANGE ALL  |        | 107 | 1070| 3 |1 | 5 | Q1,01 |PCWC|          |
| 5 |      PX PARTITION HASH ALL  |        | 107 | 1070| 3 |1 | 3 | Q1,01 |PCWC|          |
| 6 |       TABLE ACCESS FULL     |EMP_COMP| 107 | 1070| 3 |1 | 15| Q1,01 |PCWP|          |
| 7 |     PX RECEIVE              |        |  21 |  630| 1 |  |   | Q1,01 |PCWP|          |
| 8 |      PX SEND PARTITION (KEY)|:TQ10000|  21 |  630| 1 |  |   | Q1,00 |P->P|PART (KEY)|
| 9 |       PX BLOCK ITERATOR     |        |  21 |  630| 1 |  |   | Q1,00 |PCWC|          |
|10 |        TABLE ACCESS FULL    |DEPT2   |  21 |  630| 1 |  |   | Q1,00 |PCWP|          |
-------------------------------------------------------------------------------------------
```

The plan shows that the optimizer selects partial partition-wise join from one of two columns. The `PX SEND` node type is `PARTITION (KEY)` and the `PQ Distrib` column contains the text `PART (KEY)`, or partition key. This implies that the table `dept2` is re-partitioned based on the join column `department_id` to be sent to the parallel execution servers executing the scan of `EMP_COMP` and the join.

6.3.4.5.4 Example of Full Partition-Wise Join

In this example, `emp_comp` and `dept_hash` are joined on their hash partitioning columns, enabling use of a full partition-wise join.

The `PARTITION HASH` row source appears on top of the join row source in the plan table output.

```
CREATE TABLE dept_hash
   PARTITION BY HASH(department_id)
   PARTITIONS 3
   PARALLEL 2
   AS SELECT * FROM departments;

EXPLAIN PLAN FOR
  SELECT /*+ PQ_DISTRIBUTE(e NONE NONE) ORDERED */ e.last_name,
         d.department_name
  FROM   emp_comp e, dept_hash d
  WHERE  e.department_id = d.department_id;

-------------------------------------------------------------------------------------------
|Id| Operation                  | Name |Rows|Bytes|Cost|Pstart|Pstop|TQ |IN-OUT|PQ Distrib|
-------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT           |          | 106 | 2544 |8|  |   |       |      |         |
| 1|  PX COORDINATOR            |          |     |      | |  |   |       |      |         |
| 2|   PX SEND QC (RANDOM)      |:TQ10000  | 106 | 2544 |8|  |   | Q1,00 | P->S |QC (RAND)|
| 3|    PX PARTITION HASH ALL   |          | 106 | 2544 |8|1 | 3 | Q1,00 | PCWC |         |
|*4|     HASH JOIN              |          | 106 | 2544 |8|  |   | Q1,00 | PCWP |         |
| 5|      PX PARTITION RANGE ALL|          | 107 | 1070 |3|1 | 5 | Q1,00 | PCWC |         |
| 6|       TABLE ACCESS FULL    |EMP_COMP  | 107 | 1070 |3|1 |15 | Q1,00 | PCWP |         |
| 7|      TABLE ACCESS FULL     |DEPT_HASH |  27 |  378 |4|1 | 3 | Q1,00 | PCWP |         |
-------------------------------------------------------------------------------------------
```

The `PX PARTITION HASH` row source appears on top of the join row source in the plan table output while the `PX PARTITION RANGE` row source appears over the scan of `emp_comp`. Each parallel execution server performs the join of an entire hash partition of `emp_comp` with an entire partition of `dept_hash`.

6.3.4.5.5 Examples of INLIST ITERATOR and EXPLAIN PLAN

An `INLIST ITERATOR` operation appears in the `EXPLAIN PLAN` output if an index implements an `IN`-list predicate.

Consider the following statement:

```
SELECT * FROM emp WHERE empno IN (7876, 7900, 7902);
```

The `EXPLAIN PLAN` output appears as follows:

```
OPERATION         OPTIONS          OBJECT_NAME
----------------  ---------------  --------------
SELECT STATEMENT
INLIST ITERATOR
TABLE ACCESS      BY ROWID         EMP
INDEX             RANGE SCAN       EMP_EMPNO
```

The `INLIST ITERATOR` operation iterates over the next operation in the plan for each value in the `IN`-list predicate. The following sections describe the three possible types of `IN`-list columns for partitioned tables and indexes.

6.3.4.5.5.1 When the IN-List Column is an Index Column: Example

If the `IN`-list column `empno` is an index column but not a partition column, then the `IN`-list operator appears before the table operation but after the partition operation in the plan.

```
OPERATION        OPTIONS              OBJECT_NAME PARTIT_START PARTITI_STOP
---------------- ------------         ----------- ------------ ------------
SELECT STATEMENT
PARTITION RANGE  ALL                            KEY(INLIST)    KEY(INLIST)
INLIST ITERATOR
TABLE ACCESS     BY LOCAL INDEX ROWID EMP         KEY(INLIST)  KEY(INLIST)
INDEX            RANGE SCAN           EMP_EMPNO   KEY(INLIST)  KEY(INLIST)
```

The `KEY(INLIST)` designation for the partition start and stop keys specifies that an `IN`-list predicate appears on the index start and stop keys.

6.3.4.5.5.2 When the IN-List Column is an Index and a Partition Column: Example

If `empno` is an indexed and a partition column, then the plan contains an `INLIST ITERATOR` operation before the partition operation.

```
OPERATION        OPTIONS              OBJECT_NAME PARTITION_START PARTITION_STOP
---------------- ------------         ----------- --------------- --------------
SELECT STATEMENT
INLIST ITERATOR
PARTITION RANGE  ITERATOR                         KEY(INLIST)     KEY(INLIST)
```

```
TABLE ACCESS      BY LOCAL INDEX ROWID EMP          KEY(INLIST)      KEY(INLIST)
INDEX             RANGE SCAN           EMP_EMPNO    KEY(INLIST)      KEY(INLIST)
```

6.3.4.5.5.3 When the IN-List Column is a Partition Column: Example

If `empno` is a partition column and no indexes exist, then no `INLIST ITERATOR` operation is allocated.

```
OPERATION          OPTIONS         OBJECT_NAME    PARTITION_START    PARTITION_STOP
----------------   ------------    -----------    ---------------    --------------
SELECT STATEMENT
PARTITION RANGE    INLIST                         KEY(INLIST)        KEY(INLIST)
TABLE ACCESS       FULL            EMP            KEY(INLIST)        KEY(INLIST)
```

If `emp_empno` is a bitmap index, then the plan is as follows:

```
OPERATION          OPTIONS            OBJECT_NAME
----------------   ---------------    --------------
SELECT STATEMENT
INLIST ITERATOR
TABLE ACCESS       BY INDEX ROWID     EMP
BITMAP CONVERSION  TO ROWIDS
BITMAP INDEX       SINGLE VALUE       EMP_EMPNO
```

6.3.4.5.6 Example of Domain Indexes and EXPLAIN PLAN

You can use `EXPLAIN PLAN` to derive user-defined CPU and I/O costs for domain indexes.

`EXPLAIN PLAN` displays domain index statistics in the `OTHER` column of `PLAN_TABLE`. For example, assume table `emp` has user-defined operator `CONTAINS` with a domain index `emp_resume` on the `resume` column, and the index type of `emp_resume` supports the operator `CONTAINS`. You explain the plan for the following query:

```
SELECT * FROM emp WHERE CONTAINS(resume, 'Oracle') = 1
```

The database could display the following plan:

```
OPERATION            OPTIONS      OBJECT_NAME     OTHER
-----------------    -----------  ------------    ----------------
SELECT STATEMENT
TABLE ACCESS         BY ROWID     EMP
DOMAIN INDEX                      EMP_RESUME      CPU: 300, I/O: 4
```

6.4 Comparing Execution Plans

The plan comparison tool takes a reference plan and an arbitrary list of test plans and highlights the differences between them. The plan comparison is logical rather than line by line.

6.4.1 Purpose of Plan Comparison

The plan comparison report identifies the source of differences, which helps users triage plan reproducibility issues.

The plan comparison report is particularly useful in the following scenarios:

- You want to compare the current plan of a query whose performance is regressing with an old plan captured in AWR.
- A SQL plan baseline fails to reproduce the originally intended plan, and you want to determine the difference between the new plan and the intended plan.
- You want to determine how adding a hint, changing a parameter, or creating an index will affect a plan.
- You want to determine how a plan generated based on a SQL profile or by SQL Performance Analyzer differs from the original plan.

6.4.2 User Interface for Plan Comparison

You can use `DBMS_XPLAN.COMPARE_PLANS` to generate a report in text, XML, or HTML format.

Compare Plans Report Format

The report begins with a summary. The `COMPARE PLANS REPORT` section includes information such as the user who ran the report and the number of plans compared, as shown in the following example:

```
COMPARE PLANS REPORT
-----------------------------------------------------------------------
--
  Current user           : SH
  Total number of plans  : 2
  Number of findings     : 1
-----------------------------------------------------------------------
--
```

The `COMPARISON DETAILS` section of the report contains the following information:

- Plan information

  The information includes the plan number, the plan source, plan attributes (which differ depending on the source), parsing schema, and SQL text.
- Plans

  This section displays the plan rows, including the predicates and notes.
- Comparison results

  This section summarizes the comparison findings, highlighting logical differences such as join order, join methods, access paths, and parallel distribution method. The findings start at number `1`. For findings that relate to a particular query block, the text starts with the name of the block. For findings that relate to a particular object alias, the text starts with the name of the query block and the object alias. The following

```
Comparison Results (1):
-----------------------------
 1. Query block SEL$1, Alias PRODUCTS@SEL$1: Some columns (OPERATION,
    OPTIONS, OBJECT_NAME) do not match between the reference
    plan (id: 2) and the current plan (id: 2).
```

DBMS_XPLAN.PLAN_OBJECT_LIST Table Type

The `plan_object_list` type allows for a list of generic objects as input to the `DBMS_XPLAN.COMPARE_PLANS` function. The syntax is as follows:

```
TYPE plan_object_list IS TABLE OF generic_plan_object;
```

The generic object abstracts the common attributes of plans from all plan sources. Every plan source is a subclass of the `plan_object_list` superclass. The following table summarizes the different plan sources. Note that when an optional parameter is null, it can correspond to multiple objects. For example, if you do not specify a child number for `cursor_cache_object`, then it matches all cursor cache statements with the specified SQL ID.

Table 6-3 Plan Sources for PLAN_OBJECT_LIST

Plan Source	Specification	Description
Plan table	`plan_table_object(owner, plan_table_name, statement_id, plan_id)`	The parameters are as follows: • `owner`—The owner of the plan table • `plan_table_name`—The name of the plan table • `statement_id`—The ID of the statement (optional) • `plan_id`—The ID of the plan (optional)
Cursor cache	`cursor_cache_object(sql_id, child_number)`	The parameters are as follows: • `sql_id`—The SQL ID of the plan • `child_number`—The child number of the plan in the cursor cache (optional)
AWR	`awr_object(sql_id, dbid, con_dbid, plan_hash_value)`	The parameters are as follows: • `sql_id`—The SQL ID of the plan • `dbid`—The database ID (optional) • `con_dbid`—The CDB ID (optional) • `plan_hash_value`—The hash value of the plan (optional)

Table 6-3 (Cont.) Plan Sources for PLAN_OBJECT_LIST

Plan Source	Specification	Description
SQL tuning set	`sqlset_object (sqlset_owner, sqlset_name, sql_id, plan_hash_value)`	The parameters are as follows: • `sqlset_owner`—The owner of the SQL tuning set • `sqlset_name`—The name of the SQL tuning set • `sql_id`—The SQL ID of the plan • `plan_hash_value`—The hash value of the plan (optional)
SQL plan management	`spm_object (sql_handle, plan_name)`	The parameters are as follows: • `sql_handle`—The SQL handle of plans protected by SQL plan management • `plan_name`—The name of the SQL plan baseline (optional)
SQL profile	`sql_profile_object (profile_name)`	The `profile_name` parameter specifies the name of the SQL profile.
Advisor	`advisor_object (task_name, execution_name, sql_id, plan_id)`	The parameters are as follows: • `task_name`—The name of the advisor task • `execution_name`—The name of the task execution • `sql_id`—The SQL ID of the plan • `plan_id`—The advisor plan ID (optional)

DBMS_XPLAN.COMPARE_PLANS Function

The interface for the compare plan tools is the following function:

```
DBMS_XPLAN.COMPARE_PLANS(
    reference_plan        IN generic_plan_object,
    compare_plan_list     IN plan_object_list,
    type                  IN VARCHAR2 := 'TEXT',
    level                 IN VARCHAR2 := 'TYPICAL',
    section               IN VARCHAR2 := 'ALL')
RETURN CLOB;
```

The following table describes the parameters that specify that plans to be compared.

Table 6-4 Parameters for the COMPARE_PLANS Function

Parameter	Description
`reference_plan`	Specifies a single plan of type `generic_plan_object`.

Table 6-4 (Cont.) Parameters for the COMPARE_PLANS Function

Parameter	Description
`compare_plan_list`	Specifies a list of plan objects. An object might correspond to one or more plans.

Example 6-8 Comparing Plans from Child Cursors

This example compares the plan of child cursor number 2 for the SQL ID `8mkxm7ur07za0` with the plan for child cursor number 4 for the same SQL ID.

```
VAR v_report CLOB;

BEGIN
  :v_report := DBMS_XPLAN.COMPARE_PLANS(
    reference_plan    => CURSOR_CACHE_OBJECT('8mkxm7ur07za0', 2),
    compare_plan_list =>
PLAN_OBJECT_LIST(CURSOR_CACHE_OBJECT('8mkxm7ur07za0', 4)));
END;
/

PRINT v_report
```

Example 6-9 Comparing Plan from Child Cursor with Plan from SQL Plan Baseline

This example compares the plan of child cursor number 2 for the SQL ID `8mkxm7ur07za0` with the plan from the SQL plan baseline. The baseline query has a SQL handle of `SQL_024d0f7d21351f5d` and a plan name of `SQL_PLAN_sdfjkd`.

```
VAR v_report CLOB;
BEGIN
  :v_report := DBMS_XPLAN.COMPARE_PLANS( -
    reference_plan    => CURSOR_CACHE_OBJECT('8mkxm7ur07za0', 2),
    compare_plan_list => PLAN_OBJECT_LIST(SPM_OBJECT('SQL_024d0f7d21351f5d',
'SQL_PLAN_sdfjkd')));
END;

PRINT v_report
```

Example 6-10 Comparing a Plan with Plans from Multiple Sources

This example prints the summary section only. The program compares the plan of child cursor number 2 for the SQL ID `8mkxm7ur07za0` with every plan in the following list:

- All plans in the shared SQL area that are generated for the SQL ID `8mkxm7ur07za0`
- All plans generated in the SQL tuning set `SH. SQLT_WORKLOAD` for the SQL ID `6vfqvav0rgyad`
- All plans in AWR that are captured for database ID 5 and SQL ID `6vfqvav0rgyad`
- The plan baseline for the query with handle `SQL_024d0f7d21351f5d` with name `SQL_PLAN_sdfjkd`

- The plan stored in `sh.plan_table` identified by `plan_id=38`
- The plan identified by the SQL profile name `pe3r3ejsfd`
- All plans stored in SQL advisor identified by task name `TASK_1228`, execution name `EXEC_1928`, and SQL ID `8mkxm7ur07za0`

```
VAR v_report CLOB
BEGIN
  :v_report := DBMS_XPLAN.COMPARE_PLANS(
    reference_plan    => CURSOR_CACHE_OBJECT('8mkxm7ur07za0', 2),
    compare_plan_list => plan_object_list(
         cursor_cache_object('8mkxm7ur07za0'),
         sqlset_object('SH', 'SQLT_WORKLOAD', '6vfqvav0rgyad'),
         awr_object('6vfqvav0rgyad', 5),
         spm_object('SQL_024d0f7d21351f5d', 'SQL_PLAN_sdfjkd'),
         plan_table_object('SH', 'plan_table', 38),
         sql_profile_object('pe3r3ejsfd'),
         advisor_object('TASK_1228', 'EXEC_1928', '8mkxm7ur07za0')),
    type              => 'XML',
    level             => 'ALL',
    section => 'SUMMARY');
END;
/

PRINT v_report
```

Note:

Oracle Database PL/SQL Packages and Types Reference for more information about the `DBMS_XPLAN` package

6.4.3 Comparing Execution Plans: Tutorial

To compare plans, use the `DBMS_XPLAN.COMPARE_PLANS` function.

In this tutorial, you compare two distinct queries. The compare plans report shows that the optimizer was able to use a join elimination transformation in one query but not the other.

Assumptions

This tutorial assumes that user `sh` issued the following queries:

```
select count(*)
from   products p, sales s
where  p.prod_id = s.prod_id
and    p.prod_min_price > 200;

select count(*)
from   products p, sales s
```

```
where  p.prod_id = s.prod_id
and    s.quantity_sold = 43;
```

To compare execution plans:

1. Start SQL*Plus, and log in to the database with administrative privileges.
2. Query `V$SQL` to determine the SQL IDs of the two queries.

   The following query queries `V$SQL` for queries that contain the string `products`:

   ```
   SET LINESIZE 120
   COL SQL_ID FORMAT a20
   COL SQL_TEXT FORMAT a60

   SELECT SQL_ID, SQL_TEXT
   FROM   V$SQL
   WHERE  SQL_TEXT LIKE '%products%'
   AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
   ORDER BY SQL_ID;

   SQL_ID            SQL_TEXT
   ----------------- ------------------------------------------------
   0hxmvnfkasg6q     select count(*) from products p, sales s where
                     p.prod_id = s.prod_id and s.quantity_sold = 43

   10dqxjph6bwum     select count(*) from products p, sales s where
                     p.prod_id = s.prod_id and p.prod_min_price > 200
   ```

3. Log in to the database as user `sh`.
4. Execute the `DBMS_XPLAN.COMPARE_PLANS` function, specifying the SQL IDs obtained in the previous step.

   For example, execute the following program:

   ```
   VARIABLE v_rep CLOB

   BEGIN
     :v_rep := DBMS_XPLAN.COMPARE_PLANS(
       reference_plan    => cursor_cache_object('0hxmvnfkasg6q', NULL),
       compare_plan_list =>
   plan_object_list(cursor_cache_object('10dqxjph6bwum', NULL)),
       type              => 'TEXT',
       level             => 'TYPICAL',
       section           => 'ALL');
   END;
   /
   ```

5. Print the report.

   For example, run the following query:

   ```
   SET PAGESIZE 50000
   SET LONG 100000
   SET LINESIZE 210
   ```

```
COLUMN report FORMAT a200
SELECT :v_rep REPORT FROM DUAL;
```

The `Comparison Results` section of the following sample report shows that only the first query used a join elimination transformation:

```
REPORT
---------------------------------------------------------------------------------
----

COMPARE PLANS REPORT
---------------------------------------------------------------------------------
----
  Current user           : SH
  Total number of plans  : 2
  Number of findings     : 1
---------------------------------------------------------------------------------
----

COMPARISON DETAILS
---------------------------------------------------------------------------------
----
 Plan Number            : 1 (Reference Plan)
 Plan Found             : Yes
 Plan Source            : Cursor Cache
 SQL ID                 : 0hxmvnfkasg6q
 Child Number           : 0
 Plan Database Version  : 19.0.0.0
 Parsing Schema         : "SH"
 SQL Text               : select count(*) from products p, sales s where
                        p.prod_id = s.prod_id and s.quantity_sold = 43

Plan
-----------------------------

 Plan Hash Value  : 3519235612

-------------------------------------------------------------------------
| Id  | Operation               | Name  | Rows | Bytes | Cost | Time     |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT        |       |      |       |  469 |          |
|   1 |   SORT AGGREGATE        |       |    1 |     3 |      |          |
|   2 |    PARTITION RANGE ALL |       |    1 |     3 |  469 | 00:00:01 |
| * 3 |     TABLE ACCESS FULL  | SALES |    1 |     3 |  469 | 00:00:01 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 3 - filter("S"."QUANTITY_SOLD"=43)

---------------------------------------------------------------------------------
----
 Plan Number            : 2
 Plan Found             : Yes
```

```
  Plan Source              : Cursor Cache
  SQL ID                   : 10dqxjph6bwum
  Child Number             : 0
  Plan Database Version    : 19.0.0.0
  Parsing Schema           : "SH"
  SQL Text                 : select count(*) from products p, sales s where
                           p.prod_id = s.prod_id and p.prod_min_price > 200

Plan
-----------------------------

 Plan Hash Value  : 3037679890

---------------------------------------------------------------------------------------
|Id| Operation                       | Name            | Rows   | Bytes    |Cost |Time |
---------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                |                 |        |          |34|        |
| 1|   SORT AGGREGATE                |                 |      1 |       13 |  |        |
|*2|    HASH JOIN                    |                 |781685  |10161905  |34|00:00:01|
|*3|     TABLE ACCESS FULL           | PRODUCTS        |     61 |      549 | 2|00:00:01|
| 4|     PARTITION RANGE ALL         |                 |918843  | 3675372  |29|00:00:01|
| 5|      BITMAP CONVERSION TO ROWIDS |                 |918843  | 3675372  |29|00:00:01|
| 6|       BITMAP INDEX FAST FULL SCAN | SALES_PROD_BIX |        |          |  |        |
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("P"."PROD_ID"="S"."PROD_ID")
* 3 - filter("P"."PROD_MIN_PRICE">200)


Notes
-----
- This is an adaptive plan


Comparison Results (1):
-----------------------------
 1. Query block SEL$1: Transformation JOIN REMOVED FROM QUERY BLOCK occurred
    only in the reference plan (result query block: SEL$A43D1678).
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about the `DBMS_XPLAN` package

6.4.4 Comparing Execution Plans: Examples

These examples demonstrate how to generate compare plans reports for queries of tables in the `sh` schema.

Example 6-11   Comparing an Explained Plan with a Plan in a Cursor

This example explains a plan for a query of tables in the `sh` schema, and then executes the query:

```
EXPLAIN PLAN
  SET STATEMENT_ID='TEST' FOR
  SELECT c.cust_city, SUM(s.quantity_sold)
  FROM   customers c, sales s, products p
  WHERE  c.cust_id=s.cust_id
  AND    p.prod_id=s.prod_id
  AND    prod_min_price>100
  GROUP BY c.cust_city;

SELECT c.cust_city, SUM(s.quantity_sold)
FROM   customers c, sales s, products p
WHERE  c.cust_id=s.cust_id
AND    p.prod_id=s.prod_id
AND    prod_min_price>100
GROUP BY c.cust_city;
```

Assume that the SQL ID of the executed query is `9mp7z6qq83k5y`. The following PL/SQL program compares the plan in `PLAN_TABLE` and the plan in the shared SQL area:

```
BEGIN
  :v_rep := DBMS_XPLAN.COMPARE_PLANS(
    reference_plan    => plan_table_object('SH', 'PLAN_TABLE', 'TEST', 
NULL),
    compare_plan_list => 
plan_object_list(cursor_cache_object('9mp7z6qq83k5y')),
    type              => 'TEXT',
    level             => 'TYPICAL',
    section           => 'ALL');
END;
/

PRINT v_rep
```

The following sample report shows that the plans are the same:

```
COMPARE PLANS REPORT
-----------------------------------------------------------------------
--
  Current user           : SH
  Total number of plans  : 2
  Number of findings     : 1
-----------------------------------------------------------------------
--

COMPARISON DETAILS
-----------------------------------------------------------------------
--
 Plan Number            : 1 (Reference Plan)
 Plan Found              : Yes
 Plan Source             : Plan Table
 Plan Table Owner        : SH
 Plan Table Name         : PLAN_TABLE
 Statement ID            : TEST
 Plan ID                 : 52
 Plan Database Version   : 19.0.0.0
 Parsing Schema          : "SH"
 SQL Text                : No SQL Text

Plan
-----------------------------
 Plan Hash Value  : 3473931970

--------------------------------------------------------------------------
| Id| Operation                | Name     | Rows | Bytes  |Cost| Time     |
--------------------------------------------------------------------------
|  0| SELECT STATEMENT         |          |   620|   22320|1213| 00:00:01 |
|  1|   HASH GROUP BY          |          |   620|   22320|1213| 00:00:01 |
|* 2|    HASH JOIN             |          |160348| 5772528|1209| 00:00:01 |
|  3|     TABLE ACCESS FULL    |CUSTOMERS| 55500|  832500| 414| 00:00:01 |
|* 4|     HASH JOIN            |          |160348| 3367308| 472| 00:00:01 |
|* 5|      TABLE ACCESS FULL   |PRODUCTS |    13|     117|   2| 00:00:01 |
|  6|      PARTITION RANGE ALL |          |918843|11026116| 467| 00:00:01 |
|  7|       TABLE ACCESS FULL  |SALES     |918843|11026116| 467| 00:00:01 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("C"."CUST_ID"="S"."CUST_ID")
* 4 - access("P"."PROD_ID"="S"."PROD_ID")
* 5 - filter("PROD_MIN_PRICE">100)

Notes
-----
- This is an adaptive plan

--------------------------------------------------------------------------
 Plan Number             : 2
 Plan Found              : Yes
 Plan Source             : Cursor Cache
 SQL ID                  : 9mp7z6qq83k5y
 Child Number            : 0
 Plan Database Version   : 19.0.0.0
 Parsing Schema          : "SH"
 SQL Text                : select c.cust_city, sum(s.quantity_sold) from
                        customers c, sales s, products p where
                        c.cust_id=s.cust_id and p.prod_id=s.prod_id and
                        prod_min_price>100 group by c.cust_city

Plan
-----------------------------
 Plan Hash Value  : 3473931970

---------------------------------------------------------------------------
```

```
| Id  | Operation              | Name     | Rows | Bytes  | Cost|
Time     |
-----------------------------------------------------------------------
----
|  0| SELECT STATEMENT        |          |      |        |1213
|         |
|  1|   HASH GROUP BY         |          |   620|   22320|1213 |
00:00:01 |
|* 2|    HASH JOIN            |          |160348| 5772528|1209 |
00:00:01 |
|  3|     TABLE ACCESS FULL   |CUSTOMERS | 55500|  832500| 414 |
00:00:01 |
|* 4|     HASH JOIN           |          |160348| 3367308| 472 |
00:00:01 |
|* 5|      TABLE ACCESS FULL  |PRODUCTS  |    13|     117|   2 |
00:00:01 |
|  6|      PARTITION RANGE ALL |          |918843|11026116| 467 |
00:00:01 |
|  7|       TABLE ACCESS FULL |SALES     |918843|11026116| 467 |
00:00:01 |
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("C"."CUST_ID"="S"."CUST_ID")
* 4 - access("P"."PROD_ID"="S"."PROD_ID")
* 5 - filter("PROD_MIN_PRICE">100)

Notes
-----
- This is an adaptive plan

Comparison Results (1):
-----------------------------
 1. The plans are the same.
```

Example 6-12   Comparing Plans in a Baseline and SQL Tuning Set

Assume that you want to compare the plans for the following queries, which differ only in the `NO_MERGE` hint contained in the subquery:

```
SELECT c.cust_city, SUM(s.quantity_sold)
FROM   customers c, sales s,
       (SELECT prod_id FROM products WHERE prod_min_price>100) p
WHERE  c.cust_id=s.cust_id
AND    p.prod_id=s.prod_id
GROUP BY c.cust_city;

SELECT c.cust_city, SUM(s.quantity_sold)
FROM   customers c, sales s,
       (SELECT /*+ NO_MERGE */ prod_id FROM products WHERE
prod_min_price>100)
WHERE  c.cust_id=s.cust_id
```

```
AND    p.prod_id=s.prod_id
GROUP BY c.cust_city;
```

The plan for the first query is captured in a SQL plan management baseline with SQL handle `SQL_c522f5888cc4613e`. The plan for the second query is stored in a SQL tuning set named `MYSTS1` and has a SQL ID of `d07p7qmrm13nc`. You run the following PL/SQL program to compare the plans:

```
VAR v_rep CLOB

BEGIN
  v_rep := DBMS_XPLAN.COMPARE_PLANS(
   reference_plan    => spm_object('SQL_c522f5888cc4613e'),
   compare_plan_list => plan_object_list(sqlset_object('SH', 'MYSTS1',
'd07p7qmrm13nc', null)),
   type              => 'TEXT',
   level             => 'TYPICAL',
   section           => 'ALL');
END;
/

PRINT v_rep
```

The following output shows that the only the reference plan, which corresponds to the query without the hint, used a view merge:

```

---------------------------------------------------------------------------
COMPARE PLANS REPORT
---------------------------------------------------------------------------
Current user       : SH
Total number of plans : 2
Number of findings     : 1
---------------------------------------------------------------------------

COMPARISON DETAILS
---------------------------------------------------------------------------
Plan Number             : 1 (Reference Plan)
Plan Found              : Yes
Plan Source             : SQL Plan Baseline
SQL Handle              : SQL_c522f5888cc4613e
Plan Name               : SQL_PLAN_ca8rpj26c8s9y7c2279c4
Plan Database Version   : 19.0.0.0
Parsing Schema          : "SH"
SQL Text                : select c.cust_city, sum(s.quantity_sold) from
                        customers c, sales s, (select prod_id from
                        products where prod_min_price>100) p where
                        c.cust_id=s.cust_id and p.prod id=s.prod_id
                        group by c.cust_city

Plan
-----------------------------

Plan Hash Value  : 2082634180
```

```
-----------------------------------------------------------------------
----
| Id | Operation                | Name      |Rows |Bytes |Cost |
Time     |
-----------------------------------------------------------------------
----
|  0 | SELECT STATEMENT         |           |     |      |  22
|          |
|  1 |   HASH GROUP BY          |           | 300 |11400 |  22 |
00:00:01 |
|  2 |    HASH JOIN             |           | 718 |27284 |  21 |
00:00:01 |
|  3 |     TABLE ACCESS FULL    | CUSTOMERS | 630 | 9450 |   5 |
00:00:01 |
|  4 |     HASH JOIN            |           | 718 |16514 |  15 |
00:00:01 |
|  5 |      TABLE ACCESS FULL   | PRODUCTS  | 573 | 5730 |   9 |
00:00:01 |
|  6 |      PARTITION RANGE ALL |           | 960 |12480 |   5 |
00:00:01 |
|  7 |       TABLE ACCESS FULL  | SALES     | 960 |12480 |   5 |
00:00:01 |
-----------------------------------------------------------------------
----

-----------------------------------------------------------------------
----
Plan Number            : 2
Plan Found             : Yes
Plan Source            : SQL Tuning Set
SQL Tuning Set Owner   : SH
SQL Tuning Set Name    : MYSTS1
SQL ID                 : d07p7qmrm13nc
Plan Hash Value        : 655891922
Plan Database Version  : 19.0.0.0
Parsing Schema         : "SH"
SQL Text               : select c.cust_city, sum(s.quantity_sold) from
                       customers c, sales s, (select /*+ NO_MERGE */
                       prod_id from products where prod_min_price>100)
                       p where c.cust_id=s.cust_id and
                       p.prod_id=s.prod_id group by c.cust_city

Plan
-----------------------------

Plan Hash Value  : 655891922
-----------------------------------------------------------------------
--
|Id | Operation                | Name      |Rows | Bytes |Cost|
Time    |
-----------------------------------------------------------------------
--
| 0 | SELECT STATEMENT         |           |     |       | 23
|         |
| 1 |   HASH GROUP BY          |           | 300 |  9900 | 23 |
00:00:01 |
| 2 |    HASH JOIN             |           | 718 | 23694 |  21 |00:00:01 |
| 3 |     HASH JOIN            |           | 718 | 12924 |  15 |00:00:01 |
| 4 |      VIEW                |           | 573 |  2865 |   9 |00:00:01 |
| 5 |        TABLE ACCESS FULL  | PRODUCTS  | 573 |  5730 |   9 |00:00:01 |
| 6 |       PARTITION RANGE ALL |           | 960 | 12480 |   5 |00:00:01 |
| 7 |        TABLE ACCESS FULL  | SALES     | 960 | 12480 |   5 |00:00:01 |
| 8 |      TABLE ACCESS FULL    | CUSTOMERS | 630 |  9450 |   5 |00:00:01 |
-------------------------------------------------------------------------

Notes
-----
- This is an adaptive plan

Comparison Results (1):
-----------------------------
1. Query block SEL$1: Transformation VIEW MERGE occurred only in the
reference plan (result query block: SEL$F5BB74E1).
```

Example 6-13 Comparing Plans Before and After Adding an Index

In this example, you test the effect of an index on a query plan:

```
EXPLAIN PLAN
  SET STATEMENT_ID='TST1' FOR
  SELECT COUNT(*) FROM products WHERE prod_min_price>100;

CREATE INDEX newprodidx ON products(prod_min_price);

EXPLAIN PLAN
  SET STATEMENT_ID='TST2' FOR
  SELECT COUNT(*) FROM products WHERE prod_min_price>100;
```

You execute the following PL/SQL program to generate the report:

```
VAR v_rep CLOB

BEGIN
  :v_rep := DBMS_XPLAN.COMPARE_PLANS(
    reference_plan    => plan_table_object('SH','PLAN_TABLE','TST1',NULL),
    compare_plan_list => plan_object_list(plan_table_object('SH','PLAN_TABLE','TST2',NULL)),
    TYPE              => 'TEXT',
    level             => 'TYPICAL',
    section           => 'ALL');
END;
/

PRINT v_rep
```

The following report indicates that the operations in the two plans are different:

```
COMPARE PLANS REPORT
--------------------------------------------------------------------------
```

```
  Current user           : SH
  Total number of plans  : 2
  Number of findings     : 1
-----------------------------------------------------------------------
---

COMPARISON DETAILS
-----------------------------------------------------------------------
---
 Plan Number            : 1 (Reference Plan)
 Plan Found             : Yes
 Plan Source            : Plan Table
 Plan Table Owner       : SH
 Plan Table Name        : PLAN_TABLE
 Statement ID           : TST1
 Plan ID                : 56
 Plan Database Version  : 19.0.0.0
 Parsing Schema         : "SH"
 SQL Text               : No SQL Text

Plan
-----------------------------
 Plan Hash Value  : 3421487369

-----------------------------------------------------------------------
---
| Id  | Operation            | Name     | Rows | Bytes | Cost |
Time     |
-----------------------------------------------------------------------
---
|   0 | SELECT STATEMENT     |          |    1 |     5 |    2 |
00:00:01 |
|   1 |   SORT AGGREGATE     |          |    1 |     5 |
|          |
| * 2 |    TABLE ACCESS FULL | PRODUCTS |   13 |    65 |    2 |
00:00:01 |
-----------------------------------------------------------------------
---

Predicate Information (identified by operation id):
------------------------------------------
* 2 - filter("PROD_MIN_PRICE">100)

-----------------------------------------------------------------------
---
 Plan Number            : 2
 Plan Found             : Yes
 Plan Source            : Plan Table
 Plan Table Owner       : SH
 Plan Table Name        : PLAN_TABLE
 Statement ID           : TST2
 Plan ID                : 57
 Plan Database Version  : 19.0.0.0
 Parsing Schema         : "SH"
 SQL Text               : No SQL Text
```

```
Plan
-----------------------------
 Plan Hash Value  : 2694011010

---------------------------------------------------------------------------
| Id  | Operation            | Name         | Rows | Bytes | Cost | Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |              |    1 |     5 |    1 | 00:00:01 |
|   1 |   SORT AGGREGATE     |              |    1 |     5 |      |          |
| * 2 |    INDEX RANGE SCAN  | NEWPRODIDX   |   13 |    65 |    1 | 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("PROD_MIN_PRICE">100)

Comparison Results (1):
-----------------------------
 1. Query block SEL$1, Alias PRODUCTS@SEL$1: Some columns (OPERATION,
    OPTIONS, OBJECT_NAME) do not match between the reference plan
    (id: 2) and the current plan (id: 2).
```

Example 6-14 Comparing Plans with Visible and Invisible Indexes

In this example, an application executes the following query:

```
select count(*)
  from products p, sales s
 where p.prod_id = s.prod_id
   and p.prod_status = 'obsolete';
```

The plan for this query uses two indexes: `sales_prod_bix` and `products_prod_status_bix`. The database generates four plans, using all combinations of visible and invisible for both indexes. Assume that SQL plan management accepts the following plans in the baseline for the query:

- `sales_prod_bix` visible and `products_prod_status_bix` visible
- `sales_prod_bix` visible and `products_prod_status_bix` invisible
- `sales_prod_bix` invisible and `products_prod_status_bix` visible

You make both indexes invisible, and then execute the query again. The optimizer, unable to use the invisible indexes, generates a new plan. The three baseline plans, all of which rely on at least one index being visible, fail to reproduce. Therefore, the optimizer uses the new plan and adds it to the SQL plan baseline for the query. To compare the plan currently in the shared SQL area, which is the reference plan, with all four plans in the baseline, you execute the following PL/SQL code:

```
VAR v_rep CLOB

BEGIN
  :v_rep := DBMS_XPLAN.COMPARE_PLANS(
    reference_plan    => cursor_cache_object('45ns3tzutg0ds'),
    compare_plan_list =>
```

```
               plan_object_list(spm_object('SQL_aec814b0d452da8a')),
                   TYPE              => 'TEXT',
                   level             => 'TYPICAL',
                   section           => 'ALL');
               END;
               /

               PRINT v_rep
```

The following report compares all five plans:

```
-----------------------------------------------------------------------------
COMPARE PLANS REPORT
-----------------------------------------------------------------------------
Current user       : SH
Total number of plans : 5
Number of findings     : 19
-----------------------------------------------------------------------------

COMPARISON DETAILS
-----------------------------------------------------------------------------
Plan Number             : 1 (Reference Plan)
Plan Found              : Yes
Plan Source             : Cursor Cache
SQL ID                  : 45ns3tzutg0ds
Child Number            : 0
Plan Database Version   : 19.0.0.0
Parsing Schema          : "SH"
SQL Text                : select count(*) from products p, sales s where p.prod_id
                        = s.prod_id and p.prod_status = 'obsolete'

Plan
-----------------------------

Plan Hash Value  : 1136711713
------------------------------------------------------------------------------
| Id  | Operation                | Name     | Rows | Bytes | Cost | Time     |
------------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |      |       |   15 |          |
|   1 |   SORT AGGREGATE         |          |    1 |    30 |      |          |
| * 2 |    HASH JOIN             |          |  320 |  9600 |   15 | 00:00:01 |
|   3 |     JOIN FILTER CREATE   | :BF0000  |  255 |  6375 |    9 | 00:00:01 |
| * 4 |      TABLE ACCESS FULL   | PRODUCTS |  255 |  6375 |    9 | 00:00:01 |
|   5 |     JOIN FILTER USE      | :BF0000  |  960 |  4800 |    5 | 00:00:01 |
|   6 |      PARTITION RANGE ALL |          |  960 |  4800 |    5 | 00:00:01 |
| * 7 |       TABLE ACCESS FULL  | SALES    |  960 |  4800 |    5 | 00:00:01 |
------------------------------------------------------------------------------
Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("P"."PROD_ID"="S"."PROD_ID")
* 4 - filter("P"."PROD_STATUS"='obsolete')
* 7 - filter(SYS_OP_BLOOM_FILTER(:BF0000,"S"."PROD_ID"))

Notes
```

```
-----
- baseline_repro_fail = yes

-----------------------------------------------------------------------------------
Plan Number            : 2
Plan Found             : Yes
Plan Source            : SQL Plan Baseline
SQL Handle             : SQL_aec814b0d452da8a
Plan Name              : SQL_PLAN_axk0nq3a55qna6e039463
Plan Database Version  : 19.0.0.0
Parsing Schema         : "SH"
SQL Text               : select count(*) from products p, sales s where p.prod_id =
                       s.prod_id and p.prod_status = 'obsolete'

Plan
-----------------------------

Plan Hash Value  : 1845728355
-------------------------------------------------------------------------------------------
| Id| Operation                          | Name                      |Rows|Bytes|Cost| Time |
-------------------------------------------------------------------------------------------
|  0| SELECT STATEMENT                   |                           |  1|  30 |11 |00:00:01|
|  1|   SORT AGGREGATE                   |                           |  1|  30 |   |        |
| *2|    HASH JOIN                       |                           |320|9600 |11 |00:00:01|
|  3|     JOIN FILTER CREATE             | :BF0000                   |255|6375 | 5 |00:00:01|
| *4|      VIEW                          | index$_join$_001          |255|6375 | 5 |00:00:01|
| *5|       HASH JOIN                    |                           |   |     |   |        |
|  6|        BITMAP CONVERSION TO ROWIDS |                           |255|6375 | 1 |00:00:01|
| *7|         BITMAP INDEX SINGLE VALUE  | PRODUCTS_PROD_STATUS_BIX|   |     |   |        |
|  8|        INDEX FAST FULL SCAN        | PRODUCTS_PK               |255|6375 | 4 |00:00:01|
|  9|     JOIN FILTER USE                | :BF0000                   |960|4800 | 5 |00:00:01|
| 10|      PARTITION RANGE ALL           |                           |960|4800 | 5 |00:00:01|
|*11|       TABLE ACCESS FULL            | SALES                     |960|4800 | 5 |00:00:01|
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("P"."PROD_ID"="S"."PROD_ID")
* 4 - filter("P"."PROD_STATUS"='obsolete')
* 5 - access(ROWID=ROWID)
* 7 - access("P"."PROD_STATUS"='obsolete')
* 11 - filter(SYS_OP_BLOOM_FILTER(:BF0000,"S"."PROD_ID"))

Comparison Results (4):
-----------------------------
1. Query block SEL$1, Alias P@SEL$1: Some lines (id: 4) in the reference plan are missing
in the current plan.
2. Query block SEL$1, Alias S@SEL$1: Some columns (ID) do not match between the reference
plan (id: 5) and the current plan (id: 9).
3. Query block SEL$1, Alias S@SEL$1: Some columns (ID, PARENT_ID, PARTITION_ID) do not
match between the reference plan (id: 6) and the current plan (id: 10).
4. Query block SEL$1, Alias S@SEL$1: Some columns (ID, PARENT_ID, PARTITION_ID) do not
match between the reference plan (id: 7) and the current plan (id: 11).

-------------------------------------------------------------------------------------------
```

```
Plan Number                : 3
Plan Found                 : Yes
Plan Source                : SQL Plan Baseline
SQL Handle                 : SQL_aec814b0d452da8a
Plan Name                  : SQL_PLAN_axk0nq3a55qna43c0d821
Plan Database Version      : 19.0.0.0
Parsing Schema             : "SH"
SQL Text                   : select count(*) from products p, sales s where p.prod_id =
s.prod_id and
                           p.prod_status = 'obsolete'

Plan
-----------------------------
Plan Hash Value  : 1136711713

-----------------------------------------------------------------------------
| Id  | Operation                | Name     | Rows | Bytes | Cost | Time     |
-----------------------------------------------------------------------------
|   0 | SELECT STATEMENT         |          |    1 |    30 |   15 | 00:00:01 |
|   1 |   SORT AGGREGATE         |          |    1 |    30 |      |          |
| * 2 |    HASH JOIN             |          |  320 |  9600 |   15 | 00:00:01 |
|   3 |     JOIN FILTER CREATE   | :BF0000  |  255 |  6375 |    9 | 00:00:01 |
| * 4 |      TABLE ACCESS FULL   | PRODUCTS |  255 |  6375 |    9 | 00:00:01 |
|   5 |     JOIN FILTER USE      | :BF0000  |  960 |  4800 |    5 | 00:00:01 |
|   6 |      PARTITION RANGE ALL |          |  960 |  4800 |    5 | 00:00:01 |
| * 7 |       TABLE ACCESS FULL  | SALES    |  960 |  4800 |    5 | 00:00:01 |
-----------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("P"."PROD_ID"="S"."PROD_ID")
* 4 - filter("P"."PROD_STATUS"='obsolete')
* 7 - filter(SYS_OP_BLOOM_FILTER(:BF0000,"S"."PROD_ID"))

Comparison Results (1):
-----------------------------
1. The plans are the same.

-----------------------------------------------------------------------------
Plan Number                : 4
Plan Found                 : Yes
Plan Source                : SQL Plan Baseline
SQL Handle                 : SQL_aec814b0d452da8a
Plan Name                  : SQL_PLAN_axk0nq3a55qna1b7aea6c
Plan Database Version      : 19.0.0.0
Parsing Schema             : "SH"
SQL Text                   : select count(*) from products p, sales s where p.prod_id =
s.prod_id and
                           p.prod_status = 'obsolete'

Plan
-----------------------------

Plan Hash Value  : 461040236
------------------------------------------------------------------------------------
```

```
| Id | Operation                     | Name            |Rows|Bytes | Cost | Time     |
----------------------------------------------------------- ---------------------------
|  0 | SELECT STATEMENT              |                 |  1 |   30 | 10 | 00:00:01 |
|  1 |   SORT AGGREGATE              |                 |  1 |   30 |    |          |
|  2 |    NESTED LOOPS               |                 |320 | 9600 | 10 | 00:00:01 |
|* 3 |     TABLE ACCESS FULL         | PRODUCTS        |255 | 6375 |  9 | 00:00:01 |
|  4 |     PARTITION RANGE ALL       |                 |  1 |      5 | 10 | 00:00:01 |
|  5 |      BITMAP CONVERSION COUNT  |                 |  1 |      5 | 10 | 00:00:01 |
|* 6 |       BITMAP INDEX SINGLE VALUE | SALES_PROD_BIX |    |      |    |          |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 3 - filter("P"."PROD_STATUS"='obsolete')
* 6 - access("P"."PROD_ID"="S"."PROD_ID")
```

```
Comparison Results (7):
-----------------------------
1. Query block SEL$1, Alias P@SEL$1: Some lines (id: 3) in the reference plan are missing
in the current plan.
2. Query block SEL$1, Alias S@SEL$1: Some lines (id: 5) in the reference plan are missing
in the current plan.
3. Query block SEL$1, Alias S@SEL$1: Some lines (id: 7) in the reference plan are missing
in the current plan.
4. Query block SEL$1, Alias S@SEL$1: Some lines (id: 5,6) in the current plan are missing
in the reference plan.
5. Query block SEL$1, Alias P@SEL$1: Some columns (OPERATION) do not match between the
reference plan (id: 2) and the current plan (id: 2).
6. Query block SEL$1, Alias P@SEL$1: Some columns (ID, PARENT_ID, DEPTH) do not match
between the reference plan (id: 4) and the current plan (id: 3).
7. Query block SEL$1, Alias S@SEL$1: Some columns (ID, PARENT_ID, DEPTH, POSITION,
PARTITION_ID) do not match between the reference plan (id: 6) and the current plan (id: 4).
```

```
-------------------------------------------------------------------------------------------
Plan Number           : 5
Plan Found            : Yes
Plan Source           : SQL Plan Baseline
SQL Handle            : SQL_aec814b0d452da8a
Plan Name             : SQL_PLAN_axk0nq3a55qna0628afbd
Plan Database Version : 19.0.0.0
Parsing Schema        : "SH"
SQL Text              : select count(*) from products p, sales s where p.prod_id =
s.prod_id and
                       p.prod_status = 'obsolete'

Plan
-----------------------------

Plan Hash Value  : 103329725
-------------------------------------------------------------------------------------------
|Id| Operation                      | Name                | Rows|Bytes|Cost|Time    |
-------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT               |                     |     |      | 5 |          |
| 1|   SORT AGGREGATE               |                     |  1 |   30 |    |          |
| 2|    NESTED LOOPS                |                     |320 |9600 | 5 |00:00:01|
| 3|     VIEW                          | index$_join$_001         |255  |6375  | 5  |
00:00:01|
| 4|      HASH JOIN                    |                          |     |      |
|         |
| 5|       BITMAP CONVERSION TO ROWIDS |                          |255  |6375  | 1  |
00:00:01|
| 6|        BITMAP INDEX SINGLE VALUE  | PRODUCTS_PROD_STATUS_BIX |     |      |
|         |
| 7|       INDEX FAST FULL SCAN        | PRODUCTS_PK              |255  |6375  | 4  |
00:00:01|
| 8|     PARTITION RANGE ALL           |                          |  1  |    5  | 5  |
00:00:01|
| 9|      BITMAP CONVERSION TO ROWIDS  |                          |  1  |    5  | 5  |
00:00:01|
|10|       BITMAP INDEX SINGLE VALUE   | SALES_PROD_BIX           |     |      |
|         |
--------------------------------------------------------------------------------------
-----

Comparison Results (7):
-----------------------------
1. Query block SEL$1, Alias P@SEL$1: Some lines (id: 3) in the reference plan are
missing
in the current plan.
2. Query block SEL$1, Alias P@SEL$1: Some lines (id: 4) in the reference plan are
missing
in the current plan.
3. Query block SEL$1, Alias S@SEL$1: Some lines (id: 5) in the reference plan are
missing
in the current plan.
4. Query block SEL$1, Alias S@SEL$1: Some lines (id: 7) in the reference plan are
missing
in the current plan.
5. Query block SEL$1, Alias S@SEL$1: Some lines (id: 9,10) in the current plan are
missing
in the reference plan.
6. Query block SEL$1, Alias P@SEL$1: Some columns (OPERATION) do not match between
the
reference plan (id: 2) and the current plan (id: 2).
7. Query block SEL$1, Alias S@SEL$1: Some columns (ID, PARENT_ID, DEPTH, POSITION,
PARTITION_ID) do not match between the reference plan (id: 6) and the current plan
(id: 8).
```

The preceding report shows the following:

- Plan 1 is the reference plan from the shared SQL area. The plan does not use the indexes, which are both invisible, and does not reproduce a baseline plan.
- Plan 2 is in the baseline and assumes `sales_prod_bix` is invisible and `products_prod_status_bix` is visible.
- Plan 3 is in the baseline and assumes both indexes are invisible. Plan 1 and Plan 3 are the same.
- Plan 4 is in the baseline and assumes `sales_prod_bix` is visible and `products_prod_status_bix` is invisible.

- Plan 5 is in the baseline and assumes that both indexes are visible.

The comparison report shows that Plan 1 could not reproduce a plan from that baseline. The reason is that the plan in the cursor (Plan 1) was added to the baseline because no baseline plan was available at the time of execution, so the database performed a soft parse of the statement and generated the no-index plan. If the current cursor were to be invalidated, and if the query were to be executed again, then a comparison report would show that the cursor plan did reproduce a baseline plan.

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information about the `DBMS_XPLAN` package

Example 6-15 Comparing a Baseline That Fails to Reproduce

One use case is to compare a cost-based plan with a SQL plan baseline. In this example, you create a unique index. The database captures a plan baseline that uses this index. You then make the index invisible and execute the query again. The baseline plan fails to reproduce because the index is not visible, forcing the optimizer to choose a different plan. A compare plans report between the baseline plan and the cost-based plan shows the difference in the access path between the two plans.

1. Log in to the database as user `hr`, and then create a plan table:

```
CREATE TABLE PLAN_TABLE (
 STATEMENT_ID                VARCHAR2(30),
 PLAN_ID                     NUMBER,
 TIMESTAMP                   DATE,
 REMARKS                     VARCHAR2(4000),
 OPERATION                   VARCHAR2(30),
 OPTIONS                     VARCHAR2(255),
 OBJECT_NODE                 VARCHAR2(128),
 OBJECT_OWNER                VARCHAR2(30),
 OBJECT_NAME                 VARCHAR2(30),
 OBJECT_ALIAS                VARCHAR2(65),
 OBJECT_INSTANCE             NUMBER(38),
 OBJECT_TYPE                 VARCHAR2(30),
 OPTIMIZER                   VARCHAR2(255),
 SEARCH_COLUMNS              NUMBER,
 ID                          NUMBER(38),
 PARENT_ID                   NUMBER(38),
 DEPTH                       NUMBER(38),
 POSITION                    NUMBER(38),
 COST                        NUMBER(38),
 CARDINALITY                 NUMBER(38),
 BYTES                       NUMBER(38),
 OTHER_TAG                   VARCHAR2(255),
 PARTITION_START             VARCHAR2(255),
 PARTITION_STOP              VARCHAR2(255),
 PARTITION_ID                NUMBER(38),
 OTHER                       LONG,
 DISTRIBUTION                VARCHAR2(30),
 CPU_COST                     NUMBER(38),
 IO_COST                      NUMBER(38),
 TEMP_SPACE                   NUMBER(38),
 ACCESS_PREDICATES            VARCHAR2(4000),
 FILTER_PREDICATES            VARCHAR2(4000),
 PROJECTION                   VARCHAR2(4000),
 TIME                         NUMBER(38),
 QBLOCK_NAME                  VARCHAR2(30),
 OTHER_XML                    CLOB);
```

2. Execute the following DDL statements, which create a table named `staff` and an index on the `staff.employee_id` column:

```
CREATE TABLE staff AS (SELECT * FROM employees);
CREATE UNIQUE INDEX staff_employee_id ON staff (employee_id);
```

3. Execute the following statements to place a query of `staff` under the protection of SQL Plan Management, and then make the index invisible:

```
ALTER SESSION SET optimizer_capture_sql_plan_baselines = TRUE;
SELECT COUNT(*) FROM staff WHERE employee_id = 20;
-- execute query a second time to create a baseline
SELECT COUNT(*) FROM staff WHERE employee_id = 20;
ALTER SESSION SET optimizer_capture_sql_plan_baselines = FALSE;
ALTER INDEX staff_employee_id INVISIBLE;
```

4. Explain the plan, and then query the plan table (sample output included):

```
EXPLAIN PLAN SET STATEMENT_ID='STAFF' FOR SELECT COUNT(*) FROM
staff
  WHERE employee_id = 20;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(FORMAT=>'TYPICAL'));

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------
----
Plan hash value: 1778552452

--------------------------------------------------------------------
----
| Id  | Operation           | Name  |Rows  |Bytes |Cost (%CPU)|
Time     |
--------------------------------------------------------------------
----
|   0 | SELECT STATEMENT    |       |    1 |    4 |    2   (0)|
00:00:01 |
|   1 |  SORT AGGREGATE     |       |    1 |    4 |
|         |
|*  2 |   TABLE ACCESS FULL| STAFF |    1 |    4 |    2   (0)|
00:00:01 |
--------------------------------------------------------------------
----

Predicate Information (identified by operation id):
```

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------

   2 - filter("EMPLOYEE_ID"=20)

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
   - Failed to use SQL plan baseline for this statement
```

As the preceding output shows, the optimizer chooses a full table scan because the index is invisible. Because the SQL plan baseline uses an index, the optimizer cannot reproduce the plan.

5. In a separate session, log in as `SYS` and query the handle and plan name of the SQL plan baseline (sample output included):

```
SET LINESIZE 120
COL SQL_HANDLE FORMAT a25
COL PLAN_NAME FORMAT a35

SELECT DISTINCT SQL_HANDLE,PLAN_NAME,ACCEPTED
FROM   DBA_SQL_PLAN_BASELINES
WHERE  PARSING_SCHEMA_NAME = 'HR';

SQL_HANDLE                PLAN_NAME                           ACC
------------------------- ----------------------------------- ---
SQL_3fa3b23c5ba1bf60      SQL_PLAN_3z8xk7jdu3gv0b7aa092a      YES
```

6. Compare the plans, specifying the SQL handle and plan baseline name obtained from the previous step:

```
VAR v_report CLOB

BEGIN
 :v_report := DBMS_XPLAN.COMPARE_PLANS(
   reference_plan    => plan_table_object('HR', 'PLAN_TABLE', 'STAFF'),
   compare_plan_list => plan_object_list
(SPM_OBJECT('SQL_3fa3b23c5ba1bf60','SQL_PLAN_3z8xk7jdu3gv0b7aa092a')),
   type              => 'TEXT',
   level             => 'ALL',
   section           => 'ALL');
END;
/
```

7. Query the compare plans report (sample output included):

```
SET LONG 1000000
SET PAGESIZE 50000
SET LINESIZE 200
SELECT :v_report rep FROM DUAL;

REP
```

```
--------------------------------------------------------------------
----

COMPARE PLANS REPORT
--------------------------------------------------------------------
----
  Current user           : SYS
  Total number of plans  : 2
  Number of findings     : 1
--------------------------------------------------------------------
----

COMPARISON DETAILS
--------------------------------------------------------------------
----
 Plan Number            : 1 (Reference Plan)
 Plan Found             : Yes
 Plan Source            : Plan Table
 Plan Table Owner       : HR
 Plan Table Name        : PLAN_TABLE
 Statement ID           : STAFF
 Plan ID                : 72
 Plan Database Version  : 19.0.0.0
 Parsing Schema         : "HR"
 SQL Text               : No SQL Text

Plan
-----------------------------
 Plan Hash Value  : 1766070819

--------------------------------------------------------------------
| Id | Operation             | Name  |Rows| Bytes | Cost | Time     |
--------------------------------------------------------------------
|   0| SELECT STATEMENT      |       |  1 |    13 |    2 | 00:00:01 |
|   1|   SORT AGGREGATE      |       |  1 |    13 |      |          |
| * 2|    TABLE ACCESS FULL  | STAFF |  1 |    13 |    2 | 00:00:01 |
--------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - filter("EMPLOYEE_ID"=20)

Notes
-----
- Dynamic sampling used for this statement ( level = 2 )
- baseline_repro_fail = yes

--------------------------------------------------------------------
 Plan Number            : 2
 Plan Found             : Yes
 Plan Source            : SQL Plan Baseline
 SQL Handle             : SQL_3fa3b23c5ba1bf60
 Plan Name              : SQL_PLAN_3z8xk7jdu3gv0b7aa092a
 Plan Database Version  : 19.0.0.0
 Parsing Schema         : "HR"
 SQL Text               : SELECT COUNT(*) FROM staff WHERE employee_id =
20

Plan
-----------------------------

 Plan Hash Value  : 3081373994

-------------------------------------------------------------------------
|Id| Operation              | Name             |Rows|Bytes |Cost |Time     |
-------------------------------------------------------------------------
| 0| SELECT STATEMENT       |                    | 1 |  13 |   0 |00:00:01|
| 1|   SORT AGGREGATE       |                    | 1 |  13 |     |        |
|*2|    INDEX UNIQUE SCAN | STAFF_EMPLOYEE_ID | 1 |  13 |   0 |00:00:01|
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("EMPLOYEE_ID"=20)

Comparison Results (1):
-----------------------------
 1. Query block SEL$1, Alias "STAFF"@"SEL$1": Some columns (OPERATION,
OPTIONS, OBJECT_NAME) do not match between the reference plan (id: 2)
and the current plan (id: 2)
-------------------------------------------------------------------------
```

7

PLAN_TABLE Reference

This chapter describes PLAN_TABLE columns.

7.1 PLAN_TABLE Columns

PLAN_TABLE is populated by the EXPLAIN PLAN statement.

The following table describes the columns in PLAN_TABLE.

Table 7-1 PLAN_TABLE Columns

Column	Type	Description
STATEMENT_ID	VARCHAR2(30)	Value of the optional STATEMENT_ID parameter specified in the EXPLAIN PLAN statement.
PLAN_ID	NUMBER	Unique identifier of a plan in the database.
TIMESTAMP	DATE	Date and time when the EXPLAIN PLAN statement was generated.
REMARKS	VARCHAR2(80)	Any comment (of up to 80 bytes) you want to associate with each step of the explained plan. This column indicates whether the database used an outline or SQL profile for the query. If you need to add or change a remark on any row of the PLAN_TABLE, then use the UPDATE statement to modify the rows of the PLAN_TABLE.
OPERATION	VARCHAR2(30)	Name of the internal operation performed in this step. In the first row generated for a statement, the column contains one of the following values: • DELETE STATEMENT • INSERT STATEMENT • SELECT STATEMENT • UPDATE STATEMENT See "OPERATION and OPTION Columns of PLAN_TABLE" for more information about values for this column.
OPTIONS	VARCHAR2(225)	A variation on the operation that the OPERATION column describes. See "OPERATION and OPTION Columns of PLAN_TABLE" for more information about values for this column.
OBJECT_NODE	VARCHAR2(128)	Name of the database link used to reference the object (a table name or view name). For local queries using parallel execution, this column describes the order in which the database consumes output from operations.

Table 7-1 (Cont.) PLAN_TABLE Columns

Column	Type	Description
`OBJECT_OWNER`	`VARCHAR2(30)`	Name of the user who owns the schema containing the table or index.
`OBJECT_NAME`	`VARCHAR2(30)`	Name of the table or index.
`OBJECT_ALIAS`	`VARCHAR2(65)`	Unique alias of a table or view in a SQL statement. For indexes, it is the object alias of the underlying table.
`OBJECT_INSTANCE`	`NUMERIC`	Number corresponding to the ordinal position of the object as it appears in the original statement. The numbering proceeds from left to right, outer to inner for the original statement text. View expansion results in unpredictable numbers.
`OBJECT_TYPE`	`VARCHAR2(30)`	Modifier that provides descriptive information about the object; for example, `NONUNIQUE` for indexes.
`OPTIMIZER`	`VARCHAR2(255)`	Current mode of the optimizer.
`SEARCH_COLUMNS`	`NUMBERIC`	Not currently used.
`ID`	`NUMERIC`	A number assigned to each step in the execution plan.
`PARENT_ID`	`NUMERIC`	The ID of the next execution step that operates on the output of the `ID` step.
`DEPTH`	`NUMERIC`	Depth of the operation in the row source tree that the plan represents. You can use this value to indent the rows in a plan table report.
`POSITION`	`NUMERIC`	For the first row of output, this indicates the estimated cost of executing the statement. For the other rows, it indicates the position relative to the other children of the same parent.
`COST`	`NUMERIC`	Cost of the operation as estimated by the optimizer. Cost is not determined for table access operations. The value of this column does not have any particular unit of measurement; it is a weighted value used to compare costs of execution plans. The value of this column is a function of the `CPU_COST` and `IO_COST` columns.
`CARDINALITY`	`NUMERIC`	Estimate by the query optimization approach of the number of rows that the operation accessed.
`BYTES`	`NUMERIC`	Estimate by the query optimization approach of the number of bytes that the operation accessed.

Table 7-1 (Cont.) PLAN_TABLE Columns

Column	Type	Description
OTHER_TAG	VARCHAR2(255)	Describes the contents of the OTHER column. Values are: • SERIAL (blank): Serial execution. Currently, SQL is not loaded in the OTHER column for this case. • SERIAL_FROM_REMOTE (S -> R): Serial execution at a remote site. • PARALLEL_FROM_SERIAL (S -> P): Serial execution. Output of step is partitioned or broadcast to parallel execution servers. • PARALLEL_TO_SERIAL (P -> S): Parallel execution. Output of step is returned to serial QC process. • PARALLEL_TO_PARALLEL (P -> P): Parallel execution. Output of step is repartitioned to second set of parallel execution servers. • PARALLEL_COMBINED_WITH_PARENT (PWP): Parallel execution; Output of step goes to next step in same parallel process. No interprocess communication to parent. • PARALLEL_COMBINED_WITH_CHILD (PWC): Parallel execution. Input of step comes from prior step in same parallel process. No interprocess communication from child.
PARTITION_START	VARCHAR2(255)	Start partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the start partition has been identified by the SQL compiler, and its partition number is given by *n*. KEY indicates that the start partition is identified at run time from partitioning key values. ROW LOCATION indicates that the database computes the start partition (same as the stop partition) at run time from the location of each retrieved record. The record location is obtained by a user-specified ROWID or from a global index. INVALID indicates that the range of accessed partitions is empty.
PARTITION_STOP	VARCHAR2(255)	Stop partition of a range of accessed partitions. It can take one of the following values: *n* indicates that the stop partition has been identified by the SQL compiler, and its partition number is given by *n*. KEY indicates that the stop partition is identified at run time from partitioning key values. ROW LOCATION indicates that the database computes the stop partition (same as the start partition) at run time from the location of each retrieved record. The record location is obtained by a user or from a global index. INVALID indicates that the range of accessed partitions is empty.
PARTITION_ID	NUMERIC	Step that has computed the pair of values of the PARTITION_START and PARTITION_STOP columns.

Table 7-1 (Cont.) PLAN_TABLE Columns

Column	Type	Description
`OTHER`	`LONG`	Other information that is specific to the execution step that a user might find useful. See the `OTHER_TAG` column.
`DISTRIBUTION`	`VARCHAR2(30)`	Method used to distribute rows from producer query servers to consumer query servers. See "DISTRIBUTION Column of PLAN_TABLE" for more information about the possible values for this column. For more information about consumer and producer query servers, see *Oracle Database VLDB and Partitioning Guide*.
`CPU_COST`	`NUMERIC`	CPU cost of the operation as estimated by the optimizer. The value of this column is proportional to the number of machine cycles required for the operation. For statements that use the rule-based approach, this column is null.
`IO_COST`	`NUMERIC`	I/O cost of the operation as estimated by the optimizer. The value of this column is proportional to the number of data blocks read by the operation. For statements that use the rule-based approach, this column is null.
`TEMP_SPACE`	`NUMERIC`	Temporary space, in bytes, used by the operation as estimated by the optimizer. For statements that use the rule-based approach, or for operations that do not use any temporary space, this column is null.
`ACCESS_PREDICATES`	`VARCHAR2(4000)`	Predicates used to locate rows in an access structure. For example, start or stop predicates for an index range scan.
`FILTER_PREDICATES`	`VARCHAR2(4000)`	Predicates used to filter rows before producing them.
`PROJECTION`	`VARCHAR2(4000)`	Expressions produced by the operation.
`TIME`	`NUMBER(20,2)`	Elapsed time in seconds of the operation as estimated by query optimization. For statements that use the rule-based approach, this column is null.
`QBLOCK_NAME`	`VARCHAR2(30)`	Name of the query block, either system-generated or defined by the user with the `QB_NAME` hint.

"OPERATION and OPTION Columns of PLAN_TABLE" lists each combination of `OPERATION` and `OPTIONS` produced by the `EXPLAIN PLAN` statement and its meaning within an execution plan.

See Also:

Oracle Database Reference for more information about `PLAN_TABLE`

7.2 OPERATION and OPTION Columns of PLAN_TABLE

This table lists each combination of the `OPERATION` and `OPTIONS` columns of `PLAN_TABLE` and their meaning within an execution plan.

Table 7-2 OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
AND-EQUAL		Operation accepting multiple sets of rowids, returning the intersection of the sets, eliminating duplicates. Used for the single-column indexes access path.
BITMAP	CONVERSION	TO ROWIDS converts bitmap representations to actual rowids that you can use to access the table. FROM ROWIDS converts the rowids to a bitmap representation. COUNT returns the number of rowids if the actual values are not needed.
BITMAP	INDEX	SINGLE VALUE looks up the bitmap for a single key value in the index. RANGE SCAN retrieves bitmaps for a key value range. FULL SCAN performs a full scan of a bitmap index if there is no start or stop key.
BITMAP	MERGE	Merges several bitmaps resulting from a range scan into one bitmap.
BITMAP	MINUS	Subtracts bits of one bitmap from another. Row source is used for negated predicates. This option is usable only if there are non-negated predicates yielding a bitmap from which the subtraction can take place.
BITMAP	OR	Computes the bitwise OR of two bitmaps.
BITMAP	AND	Computes the bitwise AND of two bitmaps.
BITMAP	KEY ITERATION	Takes each row from a table row source and finds the corresponding bitmap from a bitmap index. This set of bitmaps are then merged into one bitmap in a following BITMAP MERGE operation.
CONNECT BY		Retrieves rows in hierarchical order for a query containing a CONNECT BY clause.
CONCATENATION		Operation accepting multiple sets of rows returning the union-all of the sets.
COUNT		Operation counting the number of rows selected from a table.
COUNT	STOPKEY	Count operation where the number of rows returned is limited by the ROWNUM expression in the WHERE clause.
CUBE JOIN		Joins a table or view on the left and a cube on the right. See *Oracle Database SQL Language Reference* to learn about the NO_USE_CUBE and USE_CUBE hints.
CUBE JOIN	ANTI	Uses an antijoin for a table or view on the left and a cube on the right.
CUBE JOIN	ANTI SNA	Uses an antijoin (single-sided null aware) for a table or view on the left and a cube on the right. The join column on the right (cube side) is NOT NULL.
CUBE JOIN	OUTER	Uses an outer join for a table or view on the left and a cube on the right.
CUBE JOIN	RIGHT SEMI	Uses a right semijoin for a table or view on the left and a cube on the right.
CUBE SCAN		Uses inner joins for all cube access.

Table 7-2 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
`CUBE SCAN`	`PARTIAL OUTER`	Uses an outer join for at least one dimension, and inner joins for the other dimensions.
`CUBE SCAN`	`OUTER`	Uses outer joins for all cube access.
`DOMAIN INDEX`		Retrieval of one or more rowids from a domain index. The options column contain information supplied by a user-defined domain index cost function, if any.
`FILTER`		Operation accepting a set of rows, eliminates some of them, and returns the rest.
`FIRST ROW`		Retrieval of only the first row selected by a query.
`FOR UPDATE`		Operation retrieving and locking the rows selected by a query containing a `FOR UPDATE` clause.
`HASH`	`GROUP BY`	Operation hashing a set of rows into groups for a query with a `GROUP BY` clause.
`HASH`	`GROUP BY PIVOT`	Operation hashing a set of rows into groups for a query with a `GROUP BY` clause. The `PIVOT` option indicates a pivot-specific optimization for the `HASH GROUP BY` operator.
`HASH JOIN` (These are join operations.)		Operation joining two sets of rows and returning the result. This join method is useful for joining large data sets of data (DSS, Batch). The join condition is an efficient way of accessing the second table. Query optimizer uses the smaller of the two tables/data sources to build a hash table on the join key in memory. Then it scans the larger table, probing the hash table to find the joined rows.
`HASH JOIN`	`ANTI`	Hash (left) antijoin
`HASH JOIN`	`SEMI`	Hash (left) semijoin
`HASH JOIN`	`RIGHT ANTI`	Hash right antijoin
`HASH JOIN`	`RIGHT SEMI`	Hash right semijoin
`HASH JOIN`	`OUTER`	Hash (left) outer join
`HASH JOIN`	`RIGHT OUTER`	Hash right outer join
`INDEX` (These are access methods.)	`UNIQUE SCAN`	Retrieval of a single rowid from an index.
`INDEX`	`RANGE SCAN`	Retrieval of one or more rowids from an index. Indexed values are scanned in ascending order.
`INDEX`	`RANGE SCAN DESCENDING`	Retrieval of one or more rowids from an index. Indexed values are scanned in descending order.
`INDEX`	`FULL SCAN`	Retrieval of all rowids from an index when there is no start or stop key. Indexed values are scanned in ascending order.
`INDEX`	`FULL SCAN DESCENDING`	Retrieval of all rowids from an index when there is no start or stop key. Indexed values are scanned in descending order.
`INDEX`	`FAST FULL SCAN`	Retrieval of all rowids (and column values) using multiblock reads. No sorting order can be defined. Compares to a full table scan on only the indexed columns. Only available with the cost based optimizer.

Table 7-2 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
`INDEX`	`SKIP SCAN`	Retrieval of rowids from a concatenated index without using the leading column(s) in the index. Only available with the cost based optimizer.
`INLIST ITERATOR`		Iterates over the next operation in the plan for each value in the `IN`-list predicate.
`INTERSECTION`		Operation accepting two sets of rows and returning the intersection of the sets, eliminating duplicates.
`MERGE JOIN` (These are join operations.)		Operation accepting two sets of rows, each sorted by a value, combining each row from one set with the matching rows from the other, and returning the result.
`MERGE JOIN`	`OUTER`	Merge join operation to perform an outer join statement.
`MERGE JOIN`	`ANTI`	Merge antijoin.
`MERGE JOIN`	`SEMI`	Merge semijoin.
`MERGE JOIN`	`CARTESIAN`	Can result from 1 or more of the tables not having any join conditions to any other tables in the statement. Can occur even with a join and it may not be flagged as `CARTESIAN` in the plan.
`CONNECT BY`		Retrieval of rows in hierarchical order for a query containing a `CONNECT BY` clause.
`MAT_VIEW REWRITE ACCESS` (These are access methods.)	`FULL`	Retrieval of all rows from a materialized view.
`MAT_VIEW REWRITE ACCESS`	`SAMPLE`	Retrieval of sampled rows from a materialized view.
`MAT_VIEW REWRITE ACCESS`	`CLUSTER`	Retrieval of rows from a materialized view based on a value of an indexed cluster key.
`MAT_VIEW REWRITE ACCESS`	`HASH`	Retrieval of rows from materialized view based on hash cluster key value.
`MAT_VIEW REWRITE ACCESS`	`BY ROWID RANGE`	Retrieval of rows from a materialized view based on a rowid range.
`MAT_VIEW REWRITE ACCESS`	`SAMPLE BY ROWID RANGE`	Retrieval of sampled rows from a materialized view based on a rowid range.
`MAT_VIEW REWRITE ACCESS`	`BY USER ROWID`	If the materialized view rows are located using user-supplied rowids.
`MAT_VIEW REWRITE ACCESS`	`BY INDEX ROWID`	If the materialized view is nonpartitioned and rows are located using indexes.
`MAT_VIEW REWRITE ACCESS`	`BY GLOBAL INDEX ROWID`	If the materialized view is partitioned and rows are located using only global indexes.

Table 7-2 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
MAT_VIEW REWRITE ACCESS	BY LOCAL INDEX ROWID	If the materialized view is partitioned and rows are located using one or more local indexes and possibly some global indexes. Partition Boundaries: The partition boundaries might have been computed by: A previous PARTITION step, in which case the PARTITION_START and PARTITION_STOP column values replicate the values present in the PARTITION step, and the PARTITION_ID contains the ID of the PARTITION step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, and INVALID. The MAT_VIEW REWRITE ACCESS or INDEX step itself, in which case the PARTITION_ID contains the ID of the step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, ROW REMOVE_LOCATION (MAT_VIEW REWRITE ACCESS only), and INVALID.
MINUS		Operation accepting two sets of rows and returning rows appearing in the first set but not in the second, eliminating duplicates.
NESTED LOOPS (These are join operations.)		Operation accepting two sets of rows, an outer set and an inner set. Oracle Database compares each row of the outer set with each row of the inner set, returning rows that satisfy a condition. This join method is useful for joining small subsets of data (OLTP). The join condition is an efficient way of accessing the second table.
NESTED LOOPS	OUTER	Nested loops operation to perform an outer join statement.
PARTITION		Iterates over the next operation in the plan for each partition in the range given by the PARTITION_START and PARTITION_STOP columns. PARTITION describes partition boundaries applicable to a single partitioned object (table or index) or to a set of equipartitioned objects (a partitioned table and its local indexes). The partition boundaries are provided by the values of PARTITION_START and PARTITION_STOP of the PARTITION. Refer to Table 6-2 for valid values of partition start and stop.
PARTITION	SINGLE	Access one partition.
PARTITION	ITERATOR	Access many partitions (a subset).
PARTITION	ALL	Access all partitions.
PARTITION	INLIST	Similar to iterator, but based on an IN-list predicate.
PARTITION	INVALID	Indicates that the partition set to be accessed is empty.
POLYMORPHIC TABLE FUNCTION		Indicates the row source for a polymorphic table function, which is a table function whose return type is determined by its arguments.
PX ITERATOR	BLOCK, CHUNK	Implements the division of an object into block or chunk ranges among a set of parallel execution servers.
PX COORDINATOR		Implements the Query Coordinator which controls, schedules, and executes the parallel plan below it using parallel execution servers. It also represents a serialization point, as the end of the part of the plan executed in parallel and always has a PX SEND QC operation below it.
PX PARTITION		Same semantics as the regular PARTITION operation except that it appears in a parallel plan.

Table 7-2 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
`PX RECEIVE`		Shows the consumer/receiver parallel execution node reading repartitioned data from a send/producer (QC or parallel execution server) executing on a PX SEND node. This information was formerly displayed into the `DISTRIBUTION` column. See Table 7-1.
`PX SEND`	`QC (RANDOM), HASH, RANGE`	Implements the distribution method taking place between two sets of parallel execution servers. Shows the boundary between two sets and how data is repartitioned on the send/producer side (QC or side. This information was formerly displayed into the `DISTRIBUTION` column. See Table 7-1.
`REMOTE`		Retrieval of data from a remote database.
`SEQUENCE`		Operation involving accessing values of a sequence.
`SORT`	`AGGREGATE`	Retrieval of a single row after applying a group function to a set of selected rows. In this case, the database "sorts" a single row.
`SORT`	`UNIQUE`	Operation sorting a set of rows to eliminate duplicates.
`SORT`	`GROUP BY`	Operation sorting a set of rows into groups for a query with a `GROUP BY` clause.
`SORT`	`GROUP BY PIVOT`	Operation sorting a set of rows into groups for a query with a `GROUP BY` clause. The `PIVOT` option indicates a pivot-specific optimization for the `SORT GROUP BY` operator.
`SORT`	`JOIN`	Operation sorting a set of rows before a merge-join.
`SORT`	`ORDER BY`	Operation sorting a set of rows for a query with an `ORDER BY` clause.
`TABLE ACCESS` (These are access methods.)	`FULL`	Retrieval of all rows from a table. `INMEMORY FULL` indicates a scan of the In-Memory column store (IM column store). The absence of `INMEMORY` indicates a scan of the row store. `INMEMORY FULL (HYBRID)` indicates a hybrid In-Memory scan. In this case, the query is divided into two parts, with one part scanning the IM column store to perform filters, and the other part scanning the row store to project the filtered query results.
`TABLE ACCESS`	`SAMPLE`	Retrieval of sampled rows from a table.
`TABLE ACCESS`	`CLUSTER`	Retrieval of rows from a table based on a value of an indexed cluster key.
`TABLE ACCESS`	`HASH`	Retrieval of rows from table based on hash cluster key value.
`TABLE ACCESS`	`BY ROWID RANGE`	Retrieval of rows from a table based on a rowid range.
`TABLE ACCESS`	`SAMPLE BY ROWID RANGE`	Retrieval of sampled rows from a table based on a rowid range.
`TABLE ACCESS`	`BY USER ROWID`	If the table rows are located using user-supplied rowids.
`TABLE ACCESS`	`BY INDEX ROWID`	If the table is nonpartitioned and rows are located using index(es).
`TABLE ACCESS`	`BY GLOBAL INDEX ROWID`	If the table is partitioned and rows are located using only global indexes.

Table 7-2 (Cont.) OPERATION and OPTIONS Values Produced by EXPLAIN PLAN

Operation	Option	Description
TABLE ACCESS	BY LOCAL INDEX ROWID	If the table is partitioned and rows are located using one or more local indexes and possibly some global indexes. Partition Boundaries: The partition boundaries might have been computed by: A previous PARTITION step, in which case the PARTITION_START and PARTITION_STOP column values replicate the values present in the PARTITION step, and the PARTITION_ID contains the ID of the PARTITION step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, and INVALID. The TABLE ACCESS or INDEX step itself, in which case the PARTITION_ID contains the ID of the step. Possible values for PARTITION_START and PARTITION_STOP are NUMBER(n), KEY, ROW REMOVE_LOCATION (TABLE ACCESS only), and INVALID.
TRANSPOSE		Operation evaluating a PIVOT operation by transposing the results of GROUP BY to produce the final pivoted data.
UNION		Operation accepting two sets of rows and returns the union of the sets, eliminating duplicates.
UNPIVOT		Operation that rotates data from columns into rows.
VIEW		Operation performing a view's query and then returning the resulting rows to another operation.

7.3 DISTRIBUTION Column of PLAN_TABLE

The DISTRIBUTION column indicates the method used to distribute rows from producer query servers to consumer query servers.

Table 7-3 Values of DISTRIBUTION Column of the PLAN_TABLE

DISTRIBUTION Text	Description
PARTITION (ROWID)	Maps rows to query servers based on the partitioning of a table or index using the rowid of the row to UPDATE/DELETE.
PARTITION (KEY)	Maps rows to query servers based on the partitioning of a table or index using a set of columns. Used for partial partition-wise join, PARALLEL INSERT, CREATE TABLE AS SELECT of a partitioned table, and CREATE PARTITIONED GLOBAL INDEX.
HASH	Maps rows to query servers using a hash function on the join key. Used for PARALLEL JOIN or PARALLEL GROUP BY.
RANGE	Maps rows to query servers using ranges of the sort key. Used when the statement contains an ORDER BY clause.
ROUND-ROBIN	Randomly maps rows to query servers.
BROADCAST	Broadcasts the rows of the entire table to each query server. Used for a parallel join when one table is very small compared to the other.
QC (ORDER)	The QC consumes the input in order, from the first to the last query server. Used when the statement contains an ORDER BY clause.

Table 7-3 (Cont.) Values of DISTRIBUTION Column of the PLAN_TABLE

DISTRIBUTION Text	Description
`QC (RANDOM)`	The QC consumes the input randomly. Used when the statement does not have an `ORDER BY` clause.

Part IV

SQL Operators: Access Paths and Joins

A **row source** is a set of rows returned by a step in the execution plan. A **SQL operator** acts on a row source.

A unary operator acts on one input, as with access paths. A binary operator acts on two outputs, as with joins.

8
Optimizer Access Paths

An **access path** is a technique used by a query to retrieve rows from a row source.

8.1 Introduction to Access Paths

A **row source** is a set of rows returned by a step in an execution plan. A row source can be a table, view, or result of a join or grouping operation.

A unary operation such as an access path, which is a technique used by a query to retrieve rows from a row source, accepts a single row source as input. For example, a full table scan is the retrieval of rows of a single row source. In contrast, a join is binary and receives inputs from exactly two row sources

The database uses different access paths for different relational data structures. The following table summarizes common access paths for the major data structures.

Table 8-1 Data Structures and Access Paths

Access Path	Heap-Organized Tables	B-Tree Indexes and IOTs	Bitmap Indexes	Table Clusters
Full Table Scans	x			
Table Access by Rowid	x			
Sample Table Scans	x			
Index Unique Scans		x		
Index Range Scans		x		
Index Full Scans		x		
Index Fast Full Scans		x		
Index Skip Scans		x		
Index Join Scans		x		
Bitmap Index Single Value			x	
Bitmap Index Range Scans			x	
Bitmap Merge			x	
Bitmap Index Range Scans			x	
Cluster Scans				x
Hash Scans				x

The optimizer considers different possible execution plans, and then assigns each plan a cost. The optimizer chooses the plan with the lowest cost. In general, index access paths are more efficient for statements that retrieve a small subset of table rows, whereas full table scans are more efficient when accessing a large portion of a table.

See Also:

- "Joins"
- "Cost-Based Optimization"
- *Oracle Database Concepts* for an overview of these structures

8.2 Table Access Paths

A table is the basic unit of data organization in an Oracle database.

Relational tables are the most common table type. Relational tables have with the following organizational characteristics:

- A heap-organized table does not store rows in any particular order.
- An index-organized table orders rows according to the primary key values.
- An external table is a read-only table whose metadata is stored in the database but whose data is stored outside the database.

See Also:

- *Oracle Database Concepts* for an overview of tables
- *Oracle Database Administrator's Guide* to learn how to manage tables

8.2.1 About Heap-Organized Table Access

By default, a table is organized as a heap, which means that the database places rows where they fit best rather than in a user-specified order.

As users add rows, the database places the rows in the first available free space in the data segment. Rows are not guaranteed to be retrieved in the order in which they were inserted.

8.2.1.1 Row Storage in Data Blocks and Segments: A Primer

The database stores rows in data blocks. In tables, the database can write a row anywhere in the bottom part of the block. Oracle Database uses the block overhead, which contains the row directory and table directory, to manage the block itself.

An extent is made up of logically contiguous data blocks. The blocks may not be physically contiguous on disk. A segment is a set of extents that contains all the data for a logical storage structure within a tablespace. For example, Oracle Database allocates one or more extents to form the data segment for a table. The database also allocates one or more extents to form the index segment for a table.

By default, the database uses automatic segment space management (ASSM) for permanent, locally managed tablespaces. When a session first inserts data into a table, the database formats a bitmap block. The bitmap tracks the blocks in the segment. The database uses the bitmap to find free blocks and then formats each block before writing to it. ASSM spread out inserts among blocks to avoid concurrency issues.

The high water mark (HWM) is the point in a segment beyond which data blocks are unformatted and have never been used. Below the HWM, a block may be formatted and written to, formatted and empty, or unformatted. The low high water mark (low HWM) marks the point below which all blocks are known to be formatted because they either contain data or formerly contained data.

During a full table scan, the database reads all blocks up to the low HWM, which are known to be formatted, and then reads the segment bitmap to determine which blocks between the HWM and low HWM are formatted and safe to read. The database knows not to read past the HWM because these blocks are unformatted.

See Also:

Oracle Database Concepts to learn about data block storage

8.2.1.2 Importance of Rowids for Row Access

Every row in a heap-organized table has a rowid unique to this table that corresponds to the physical address of a row piece. A rowid is a 10-byte physical address of a row.

The rowid points to a specific file, block, and row number. For example, in the rowid `AAAPecAAFAAAABSAAA`, the final `AAA` represents the row number. The row number is an index into a row directory entry. The row directory entry contains a pointer to the location of the row on the block.

The database can sometimes move a row in the bottom part of the block. For example, if row movement is enabled, then the row can move because of partition key updates, Flashback Table operations, shrink table operations, and so on. If the database moves a row within a block, then the database updates the row directory entry to modify the pointer. The rowid stays constant.

Oracle Database uses rowids internally for the construction of indexes. For example, each key in a B-tree index is associated with a rowid that points to the address of the associated row. Physical rowids provide the fastest possible access to a table row, enabling the database to retrieve a row in as little as a single I/O.

See Also:

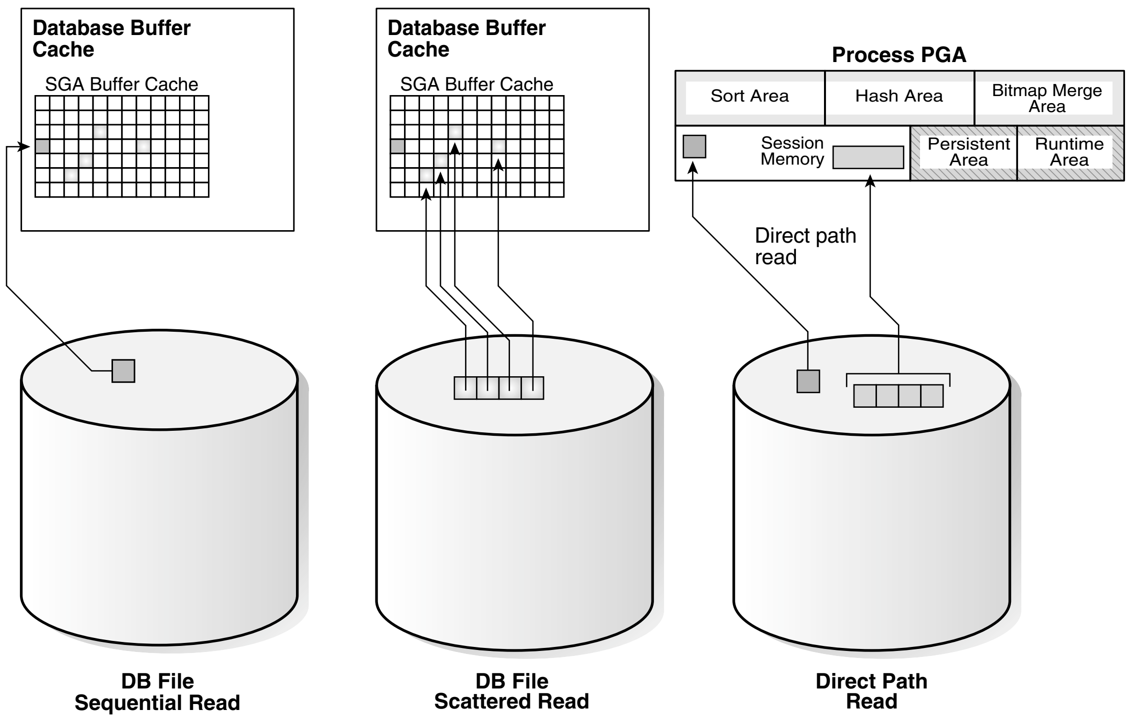
Oracle Database Concepts to learn about rowids

8.2.1.3 Direct Path Reads

In a **direct path read**, the database reads buffers from disk directly into the PGA, bypassing the SGA entirely.

The following figure shows the difference between scattered and sequential reads, which store buffers in the SGA, and direct path reads.

Figure 8-1 Direct Path Reads



Situations in which Oracle Database may perform direct path reads include:

- Execution of a `CREATE TABLE AS SELECT` statement
- Execution of an `ALTER REBUILD` or `ALTER MOVE` statement
- Reads from a temporary tablespace
- Parallel queries
- Reads from a LOB segment

See Also:

Oracle Database Performance Tuning Guide to learn about wait events for direct path reads

8.2.2 Full Table Scans

A **full table scan** reads all rows from a table, and then filters out those rows that do not meet the selection criteria.

8.2.2.1 When the Optimizer Considers a Full Table Scan

In general, the optimizer chooses a full table scan when it cannot use a different access path, or another usable access path is higher cost.

The following table shows typical reasons for choosing a full table scan.

Table 8-2 Typical Reasons for a Full Table Scan

Reason	Explanation	To Learn More
No index exists.	If no index exists, then the optimizer uses a full table scan.	*Oracle Database Concepts*
The query predicate applies a function to the indexed column.	Unless the index is a function-based index, the database indexes the values of the column, not the values of the column with the function applied. A typical application-level mistake is to index a character column, such as `char_col`, and then query the column using syntax such as `WHERE char_col=1`. The database implicitly applies a `TO_NUMBER` function to the constant number `1`, which prevents use of the index.	*Oracle Database Development Guide*
A `SELECT COUNT(*)` query is issued, and an index exists, but the indexed column contains nulls.	The optimizer cannot use the index to count the number of table rows because the index cannot contain null entries.	"B-Tree Indexes and Nulls"
The query predicate does not use the leading edge of a B-tree index.	For example, an index might exist on `employees(first_name,last_name)`. If a user issues a query with the predicate `WHERE last_name='KING'`, then the optimizer may not choose an index because column `first_name` is not in the predicate. However, in this situation the optimizer may choose to use an index skip scan.	"Index Skip Scans"
The query is unselective.	If the optimizer determines that the query requires most of the blocks in the table, then it uses a full table scan, even though indexes are available. Full table scans can use larger I/O calls. Making fewer large I/O calls is cheaper than making many smaller calls.	"Selectivity"
The table statistics are stale.	For example, a table was small, but now has grown large. If the table statistics are stale and do not reflect the current size of the table, then the optimizer does not know that an index is now most efficient than a full table scan.	"Introduction to Optimizer Statistics"
The table is small.	If a table contains fewer than *n* blocks under the high water mark, where *n* equals the setting for the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter, then a full table scan may be cheaper than an index range scan. The scan may be less expensive regardless of the fraction of tables being accessed or indexes present.	*Oracle Database Reference*

Table 8-2 (Cont.) Typical Reasons for a Full Table Scan

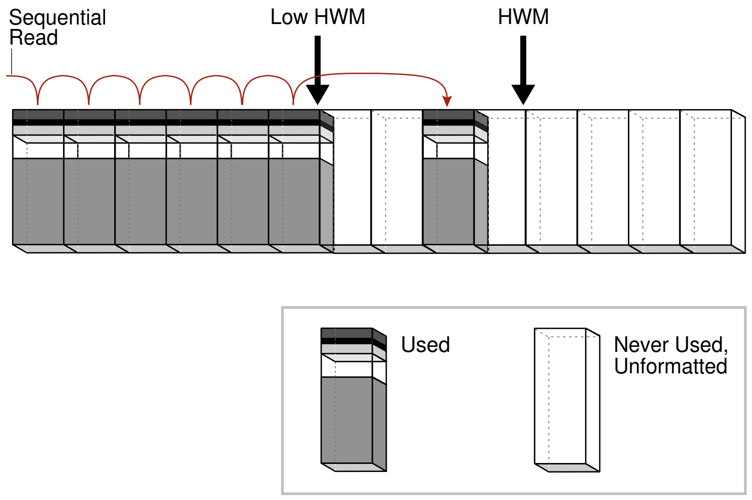
Reason	Explanation	To Learn More
The table has a high degree of parallelism.	A high degree of parallelism for a table skews the optimizer toward full table scans over range scans. Query the value in the `ALL_TABLES.DEGREE` column to determine the degree of parallelism.	*Oracle Database Reference*
The query uses a full table scan hint.	The hint `FULL(`*`table alias`*`)` instructs the optimizer to use a full table scan.	*Oracle Database SQL Language Reference*

8.2.2.2 How a Full Table Scan Works

In a full table scan, the database sequentially reads every formatted block under the high water mark. The database reads each block only once.

The following graphic depicts a scan of a table segment, showing how the scan skips unformatted blocks below the high water mark.

Figure 8-2 High Water Mark



Because the blocks are adjacent, the database can speed up the scan by making I/O calls larger than a single block, known as a multiblock read. The size of a read call ranges from one block to the number of blocks specified by the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter. For example, setting this parameter to `4` instructs the database to read up to 4 blocks in a single call.

The algorithms for caching blocks during full table scans are complex. For example, the database caches blocks differently depending on whether tables are small or large.

See Also:

- "Table 19-2"
- *Oracle Database Concepts* for an overview of the default caching mode
- *Oracle Database Reference* to learn about the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter

8.2.2.3 Full Table Scan: Example

This example scans the `hr.employees` table.

The following statement queries monthly salaries over $4000:

```
SELECT salary
FROM   hr.employees
WHERE  salary > 4000;
```

Example 8-1   Full Table Scan

The following plan was retrieved using the `DBMS_XPLAN.DISPLAY_CURSOR` function. Because no index exists on the `salary` column, the optimizer cannot use an index range scan, and so uses a full table scan.

```
SQL_ID  54c20f3udfnws, child number 0
-------------------------------------
select salary from hr.employees where salary > 4000

Plan hash value: 3476115102

---------------------------------------------------------------------------
| Id| Operation         | Name      | Rows | Bytes |Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|  0| SELECT STATEMENT  |           |      |       |    3 (100)|          |
|* 1|  TABLE ACCESS FULL| EMPLOYEES |   98 |  6762 |    3   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SALARY">4000)
```

8.2.3 Table Access by Rowid

A **rowid** is an internal representation of the storage location of data.

The rowid of a row specifies the data file and data block containing the row and the location of the row in that block. Locating a row by specifying its rowid is the fastest way to retrieve a single row because it specifies the exact location of the row in the database.

Note:

Rowids can change between versions. Accessing data based on position is not recommended because rows can move.

See Also:

Oracle Database Development Guide to learn more about rowids

8.2.3.1 When the Optimizer Chooses Table Access by Rowid

In most cases, the database accesses a table by rowid after a scan of one or more indexes.

However, table access by rowid need not follow every index scan. If the index contains all needed columns, then access by rowid might not occur.

8.2.3.2 How Table Access by Rowid Works

To access a table by rowid, the database performs multiple steps.

The database does the following:

1. Obtains the rowids of the selected rows, either from the statement `WHERE` clause or through an index scan of one or more indexes

   Table access may be needed for columns in the statement not present in the index.
2. Locates each selected row in the table based on its rowid

8.2.3.3 Table Access by Rowid: Example

This example demonstrates rowid access of the `hr.employees` table.

Assume that you run the following query:

```
SELECT *
FROM   employees
WHERE  employee_id > 190;
```

Step 2 of the following plan shows a range scan of the `emp_emp_id_pk` index on the `hr.employees` table. The database uses the rowids obtained from the index to find the corresponding rows from the `employees` table, and then retrieve them. The `BATCHED` access shown in Step 1 means that the database retrieves a few rowids from the index, and then attempts to access rows in block order to improve the clustering and reduce the number of times that the database must access a block.

```
--------------------------------------------------------------------------------
|Id| Operation                                   | Name      |Rows|Bytes|Cost(%CPU)|Time|
--------------------------------------------------------------------------------
| 0| SELECT STATEMENT                    |            |  |    |2(100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|EMPLOYEES    |16|1104|2   (0)|00:00:01|
|*2|   INDEX RANGE SCAN                  |EMP_EMP_ID_PK|16|    |1   (0)|00:00:01|
--------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("EMPLOYEE_ID">190)
```

8.2.4 Sample Table Scans

A **sample table scan** retrieves a random sample of data from a simple table or a complex `SELECT` statement, such as a statement involving joins and views.

8.2.4.1 When the Optimizer Chooses a Sample Table Scan

The database uses a sample table scan when a statement `FROM` clause includes the `SAMPLE` keyword.

The `SAMPLE` clause has the following forms:

- `SAMPLE (`*`sample_percent`*`)`

  The database reads a specified percentage of rows in the table to perform a sample table scan.

- `SAMPLE BLOCK (`*`sample_percent`*`)`

  The database reads a specified percentage of table blocks to perform a sample table scan.

The *sample_percent* specifies the percentage of the total row or block count to include in the sample. The value must be in the range `.000001` up to, but not including, `100`. This percentage indicates the probability of each row, or each cluster of rows in block sampling, being selected for the sample. It does not mean that the database retrieves exactly *sample_percent* of the rows.

Note:

Block sampling is possible only during full table scans or index fast full scans. If a more efficient execution path exists, then the database does not sample blocks. To guarantee block sampling for a specific table or index, use the `FULL` or `INDEX_FFS` hint.

See Also:

- "Influencing the Optimizer with Hints"
- *Oracle Database SQL Language Reference* to learn about the `SAMPLE` clause

8.2.4.2 Sample Table Scans: Example

This example uses a sample table scan to access 1% of the `employees` table, sampling by blocks instead of rows.

Example 8-2 Sample Table Scan

```
SELECT * FROM hr.employees SAMPLE BLOCK (1);
```

The `EXPLAIN PLAN` output for this statement might look as follows:

```
------------------------------------------------------------------------
--
| Id  | Operation             |  Name        | Rows  | Bytes | Cost
(%CPU)|
------------------------------------------------------------------------
--
|   0 | SELECT STATEMENT      |              |     1 |    68 |     3
(34)|
|   1 |  TABLE ACCESS SAMPLE | EMPLOYEES    |     1 |    68 |     3
(34)|
------------------------------------------------------------------------
--
```

8.2.5 In-Memory Table Scans

An **In-Memory scan** retrieves rows from the In-Memory Column Store (IM column store).

The IM column store is an optional SGA area that stores copies of tables and partitions in a special columnar format optimized for rapid scans.

See Also:

Oracle Database In-Memory Guide for an introduction to the IM column store

8.2.5.1 When the Optimizer Chooses an In-Memory Table Scan

The optimizer cost model is aware of the content of the IM column store.

When a user executes a query that references a table in the IM column store, the optimizer calculates the cost of all possible access methods—including the In-Memory table scan—and selects the access method with the lowest cost.

8.2.5.2 In-Memory Query Controls

You can control In-Memory queries using initialization parameters.

The following database initialization parameters affect the In-Memory features:

- `INMEMORY_QUERY`

  This parameter enables or disables In-Memory queries for the database at the session or system level. This parameter is helpful when you want to test workloads with and without the use of the IM column store.

- `OPTIMIZER_INMEMORY_AWARE`

  This parameter enables (`TRUE`) or disables (`FALSE`) all of the In-Memory enhancements made to the optimizer cost model, table expansion, bloom filters, and so on. Setting the parameter to `FALSE` causes the optimizer to ignore the In-Memory property of tables during the optimization of SQL statements.

- `OPTIMIZER_FEATURES_ENABLE`

  When set to values lower than `12.1.0.2`, this parameter has the same effect as setting `OPTIMIZER_INMEMORY_AWARE` to `FALSE`.

To enable or disable In-Memory queries, you can specify the `INMEMORY` or `NO_INMEMORY` hints, which are the per-query equivalent of the `INMEMORY_QUERY` initialization parameter. If a SQL statement uses the `INMEMORY` hint, but the object it references is not already loaded in the IM column store, then the database does not wait for the object to be populated in the IM column store before executing the statement. However, initial access of the object triggers the object population in the IM column store.

See Also:

- *Oracle Database Reference* to learn more about the `INMEMORY_QUERY`, `OPTIMIZER_INMEMORY_AWARE`, and `OPTIMIZER_FEATURES_ENABLE` initialization parameters
- *Oracle Database SQL Language Reference* to learn more about the `INMEMORY` hints

8.2.5.3 In-Memory Table Scans: Example

This example shows an execution plan that includes the `TABLE ACCESS INMEMORY` operation.

The following example shows a query of the `oe.product_information` table, which has been altered with the `INMEMORY HIGH` option.

Example 8-3 In-Memory Table Scan

```
SELECT *
FROM   oe.product_information
WHERE  list_price > 10
ORDER BY product_id
```

The plan for this statement might look as follows, with the `INMEMORY` keyword in Step 2 indicating that some or all of the object was accessed from the IM column store:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

SQL_ID  2mb4h57x8pabw, child number 0
```

```
-------------------------------------
select * from oe.product_information where list_price > 10 order byproduct_id

Plan hash value: 2256295385
--------------------------------------------------------------------------------------
-----
|Id| Operation                   | Name                 |Rows|Bytes|TempSpc|Cost(%CPU)|
Time|
--------------------------------------------------------------------------------------
-----
| 0| SELECT STATEMENT            |                      |    |     |       |21
(100)|        |
| 1|  SORT ORDER BY              |                      |285| 62415|82000|21    (5)|
00:00:01|
|*2|   TABLE ACCESS INMEMORY FULL| PRODUCT_INFORMATION |285| 62415|      | 5    (0)|
00:00:01|
--------------------------------------------------------------------------------------
-----

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - inmemory("LIST_PRICE">10)
       filter("LIST_PRICE">10)
```

8.3 B-Tree Index Access Paths

An **index** is an optional structure, associated with a table or table cluster, that can sometimes speed data access.

By creating an index on one or more columns of a table, you gain the ability in some cases to retrieve a small set of randomly distributed rows from the table. Indexes are one of many means of reducing disk I/O.

See Also:

- *Oracle Database Concepts* for an overview of indexes
- *Oracle Database Administrator's Guide* to learn more about automatic and manual index creation

8.3.1 About B-Tree Index Access

B-trees, short for *balanced trees*, are the most common type of database index.

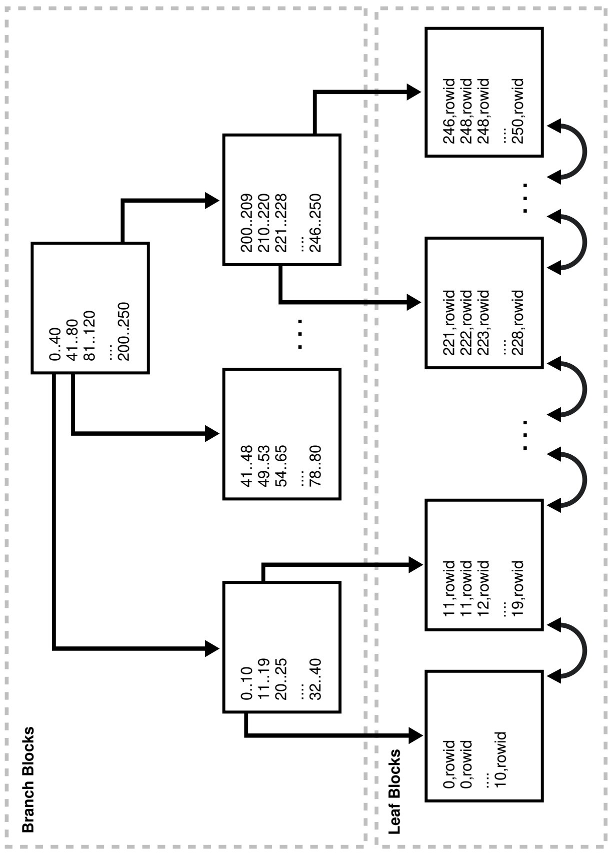
A B-tree index is an ordered list of values divided into ranges. By associating a key with a row or range of rows, B-trees provide excellent retrieval performance for a wide range of queries, including exact match and range searches.

8.3.1.1 B-Tree Index Structure

A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values.

The following graphic illustrates the logical structure of a B-tree index. Branch blocks store the minimum key prefix needed to make a branching decision between two keys. The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. Each index entry is sorted by (key, rowid). The leaf blocks are doubly linked.

Figure 8-3 B-Tree Index Structure



8.3.1.2 How Index Storage Affects Index Scans

Bitmap index blocks can appear anywhere in the index segment.

Figure 8-3 shows the leaf blocks as adjacent to each other. For example, the `1-10` block is next to and before the `11-19` block. This sequencing illustrates the linked lists that connect the index entries. However, index blocks need not be stored in order within an *index segment*. For example, the `246-250` block could appear anywhere in the segment, including directly before the `1-10` block. For this reason, ordered index scans must perform single-block I/O. The database must read an index block to determine which index block it must read next.

The index block body stores the index entries in a heap, just like table rows. For example, if the value `10` is inserted first into a table, then the index entry with key `10` might be inserted at the bottom of the index block. If 0 is inserted next into the table, then the index entry for key 0 might be inserted on top of the entry for 10. Thus, the index entries in the block *body* are not stored in key order. However, within the index block, the row header stores records in key order. For example, the first record in the header points to the index entry with key 0, and so on sequentially up to the record that points to the index entry with key 10. Thus, index scans can read the row header to determine where to begin and end range scans, avoiding the necessity of reading every entry in the block.

See Also:

Oracle Database Concepts to learn about index blocks

8.3.1.3 Unique and Nonunique Indexes

In a nonunique index, the database stores the rowid by appending it to the key as an extra column. The entry adds a length byte to make the key unique.

For example, the first index key in the nonunique index shown in Figure 8-3 is the pair 0,*rowid* and not simply 0. The database sorts the data by index key values and then by rowid ascending. For example, the entries are sorted as follows:

```
0,AAAPvCAAFAAAAFaAAa
0,AAAPvCAAFAAAAFaAAg
0,AAAPvCAAFAAAAFaAAl
2,AAAPvCAAFAAAAFaAAm
```

In a unique index, the index key does not include the rowid. The database sorts the data only by the index key values, such as 0, 1, 2, and so on.

See Also:

Oracle Database Concepts for an overview of unique and nonunique indexes

8.3.1.4 B-Tree Indexes and Nulls

B-tree indexes never store completely null keys, which is important for how the optimizer chooses access paths. A consequence of this rule is that single-column B-tree indexes never store nulls.

An example helps illustrate. The hr.employees table has a primary key index on employee_id, and a unique index on department_id. The department_id column can contain nulls, making it a *nullable column*, but the employee_id column cannot.

```
SQL> SELECT COUNT(*) FROM employees WHERE department_id IS NULL;

  COUNT(*)
----------
```

```
         1

SQL> SELECT COUNT(*) FROM employees WHERE employee_id IS NULL;

  COUNT(*)
----------
         0
```

The following example shows that the optimizer chooses a full table scan for a query of all department IDs in `hr.employees`. The optimizer cannot use the index on `employees.department_id` because the index is not guaranteed to include entries for every row in the table.

```
SQL> EXPLAIN PLAN FOR SELECT department_id FROM employees;

Explained.

SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 3476115102

---------------------------------------------------------------------------
|Id | Operation         | Name       | Rows| Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT  |            | 107 |   321 |     2   (0)| 00:00:01 |
| 1 |  TABLE ACCESS FULL| EMPLOYEES  | 107 |   321 |     2   (0)| 00:00:01 |
---------------------------------------------------------------------------
```

The following example shows the optimizer can use the index on `department_id` for a query of a specific department ID because all non-null rows are indexed.

```
SQL> EXPLAIN PLAN FOR SELECT department_id FROM employees WHERE
department_id=10;

Explained.

SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 67425611

---------------------------------------------------------------------------
|Id| Operation        | Name              |Rows|Bytes|Cost (%CPU)| Time    |
---------------------------------------------------------------------------
| 0| SELECT STATEMENT |                   | 1 |   3 |   1    (0)| 00:0 0:01|
|*1|  INDEX RANGE SCAN| EMP_DEPARTMENT_IX | 1 |   3 |   1    (0)| 00:0 0:01|
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
   1 - access("DEPARTMENT_ID"=10)
```

The following example shows that the optimizer chooses an index scan when the predicate excludes null values:

```
SQL> EXPLAIN PLAN FOR SELECT department_id FROM employees
WHERE department_id IS NOT NULL;

Explained.

SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----
Plan hash value: 1590637672

-----------------------------------------------------------------------
----
| Id| Operation        | Name              |Rows|Bytes| Cost (%CPU)|
Time |
-----------------------------------------------------------------------
----
| 0| SELECT STATEMENT |                   |106| 318 |   1   (0)| 00:0
0:01|
|*1|  INDEX FULL SCAN | EMP_DEPARTMENT_IX |106| 318 |   1   (0)| 00:0
0:01|
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
   1 - filter("DEPARTMENT_ID" IS NOT NULL)
```

8.3.2 Index Unique Scans

An **index unique scan** returns at most 1 rowid.

8.3.2.1 When the Optimizer Considers Index Unique Scans

An index unique scan requires an equality predicate.

Specifically, the database performs a unique scan only when a query predicate references all columns in a unique index key using an equality operator, such as `WHERE prod_id=10`.

A unique or primary key constraint is insufficient by itself to produce an index unique scan because a non-unique index on the column may already exist. Consider the following example, which creates the `t_table` table and then creates a non-unique index on `numcol`:

```
SQL> CREATE TABLE t_table(numcol INT);
SQL> CREATE INDEX t_table_idx ON t_table(numcol);
SQL> SELECT UNIQUENESS FROM USER_INDEXES WHERE INDEX_NAME =
'T_TABLE_IDX';

UNIQUENES
```

```
---------
NONUNIQUE
```

The following code creates a primary key constraint on a column with a non-unique index, resulting in an index range scan rather than an index unique scan:

```
SQL> ALTER TABLE t_table ADD CONSTRAINT t_table_pk PRIMARY KEY(numcol);
SQL> SET AUTOTRACE TRACEONLY EXPLAIN
SQL> SELECT * FROM t_table WHERE numcol = 1;

Execution Plan
----------------------------------------------------------
Plan hash value: 868081059

---------------------------------------------------------------------------
| Id | Operation        | Name         |Rows  |Bytes |Cost  (%CPU)|Time     |
---------------------------------------------------------------------------
|  0 | SELECT STATEMENT |              |    1 |   13 |     1    (0)|00:00:01 |
|* 1 |  INDEX RANGE SCAN| T_TABLE_IDX |    1 |   13 |     1    (0)|00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
    1 - access("NUMCOL"=1)
```

You can use the `INDEX(`*`alias index_name`*`)` hint to specify the index to use, but not a specific type of index access path.

See Also:

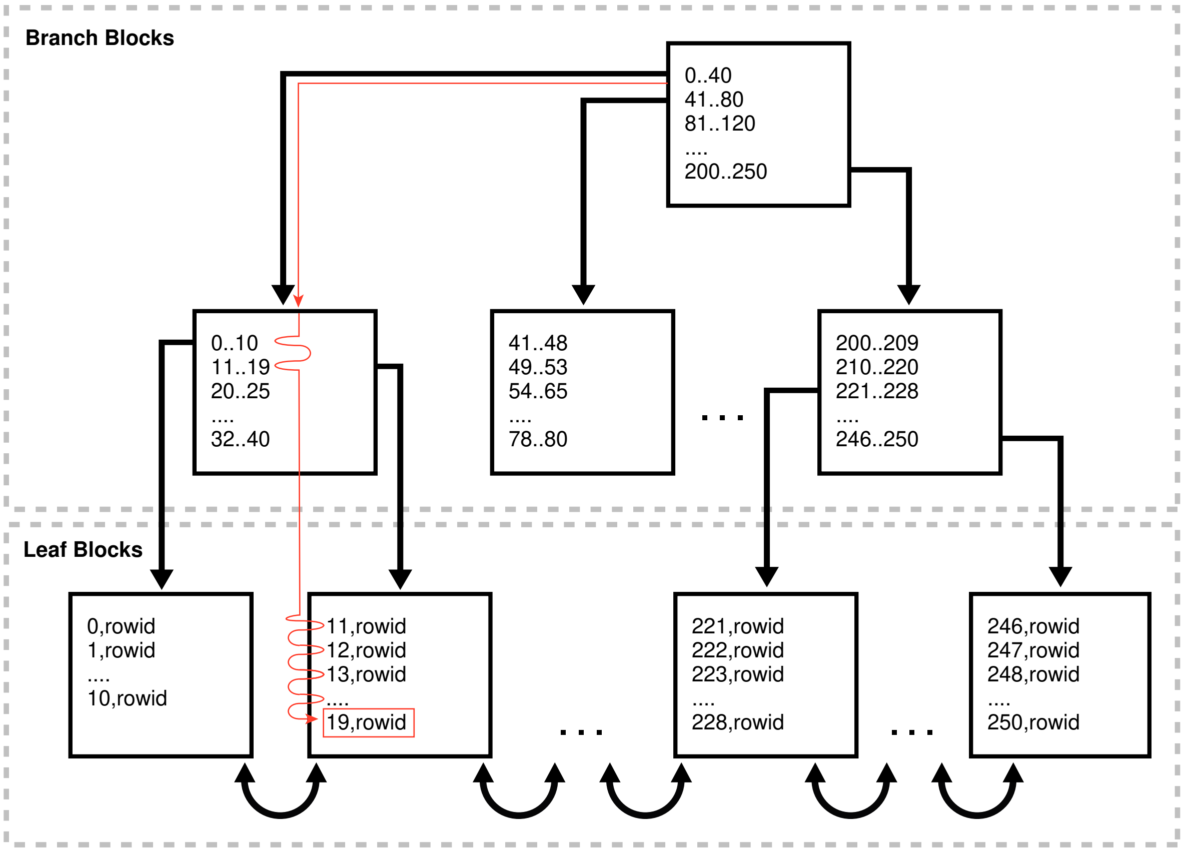
- *Oracle Database Concepts* for more details on index structures and for detailed information on how a B-tree is searched
- *Oracle Database SQL Language Reference* to learn more about the `INDEX` hint

8.3.2.2 How Index Unique Scans Work

The scan searches the index in order for the specified key. An index unique scan stops processing as soon as it finds the first record because no second record is possible. The database obtains the rowid from the index entry, and then retrieves the row specified by the rowid.

The following figure illustrates an index unique scan. The statement requests the record for product ID `19` in the `prod_id` column, which has a primary key index.

Figure 8-4 Index Unique Scan



8.3.2.3 Index Unique Scans: Example

This example uses a unique scan to retrieve a row from the `products` table.

The following statement queries the record for product `19` in the `sh.products` table:

```
SELECT *
FROM   sh.products
WHERE  prod_id = 19;
```

Because a primary key index exists on the `products.prod_id` column, and the `WHERE` clause references all of the columns using an equality operator, the optimizer chooses a unique scan:

```
SQL_ID  3ptq5tsd5vb3d, child number 0
-------------------------------------
select * from sh.products where prod_id = 19

Plan hash value: 4047888317

-----------------------------------------------------------------------
----
| Id| Operation                   | Name    |Rows|Bytes|Cost (%CPU)|
Time   |
---------------------------------------------------------------------------
|  0| SELECT STATEMENT            |              |  |     |1  (100)|        |
|  1|  TABLE ACCESS BY INDEX ROWID| PRODUCTS     |1 | 173 |1    (0)|00:00:01|
|* 2|   INDEX UNIQUE SCAN         | PRODUCTS_PK  |1 |     |0    (0)|        |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("PROD_ID"=19)
```

8.3.3 Index Range Scans

An **index range scan** is an ordered scan of values.

The range in the scan can be bounded on both sides, or unbounded on one or both sides. The optimizer typically chooses a range scan for queries with high selectivity.

By default, the database stores indexes in ascending order, and scans them in the same order. For example, a query with the predicate `department_id >= 20` uses a range scan to return rows sorted by index keys `20`, `30`, `40`, and so on. If multiple index entries have identical keys, then the database returns them in ascending order by rowid, so that `0,AAAPvCAAFAAAAFaAAa` is followed by `0,AAAPvCAAFAAAAFaAAg`, and so on.

An index range scan descending is identical to an index range scan except that the database returns rows in descending order. Usually, the database uses a descending scan when ordering data in a descending order, or when seeking a value less than a specified value.

8.3.3.1 When the Optimizer Considers Index Range Scans

For an index range scan, multiple values must be possible for the index key.

Specifically, the optimizer considers index range scans in the following circumstances:

- One or more leading columns of an index are specified in conditions.

  A condition specifies a combination of one or more expressions and logical (Boolean) operators and returns a value of `TRUE`, `FALSE`, or `UNKNOWN`. Examples of conditions include:

  - `department_id = :id`
  - `department_id < :id`
  - `department_id > :id`
  - `AND` combination of the preceding conditions for leading columns in the index, such as `department_id > :low AND department_id < :hi`.

  Note:

  For the optimizer to consider a range scan, wild-card searches of the form `col1 LIKE '%ASD'` must not be in a leading position.

- 0, 1, or more values are possible for an index key.

Tip:

If you require sorted data, then use the `ORDER BY` clause, and do not rely on an index. If an index can satisfy an `ORDER BY` clause, then the optimizer uses this option and thereby avoids a sort.

The optimizer considers an index range scan descending when an index can satisfy an `ORDER BY DESCENDING` clause.

If the optimizer chooses a full table scan or another index, then a hint may be required to force this access path. The `INDEX(`*`tbl_alias ix_name`*`)` and `INDEX_DESC(`*`tbl_alias ix_name`*`)` hints instruct the optimizer to use a specific index.

See Also:

Oracle Database SQL Language Reference to learn more about the `INDEX` and `INDEX_DESC` hints

8.3.3.2 How Index Range Scans Work

During an index range scan, Oracle Database proceeds from root to branch.

In general, the scan algorithm is as follows:

1. Read the root block.
2. Read the branch block.
3. Alternate the following steps until all data is retrieved:
   a. Read a leaf block to obtain a rowid.
   b. Read a table block to retrieve a row.

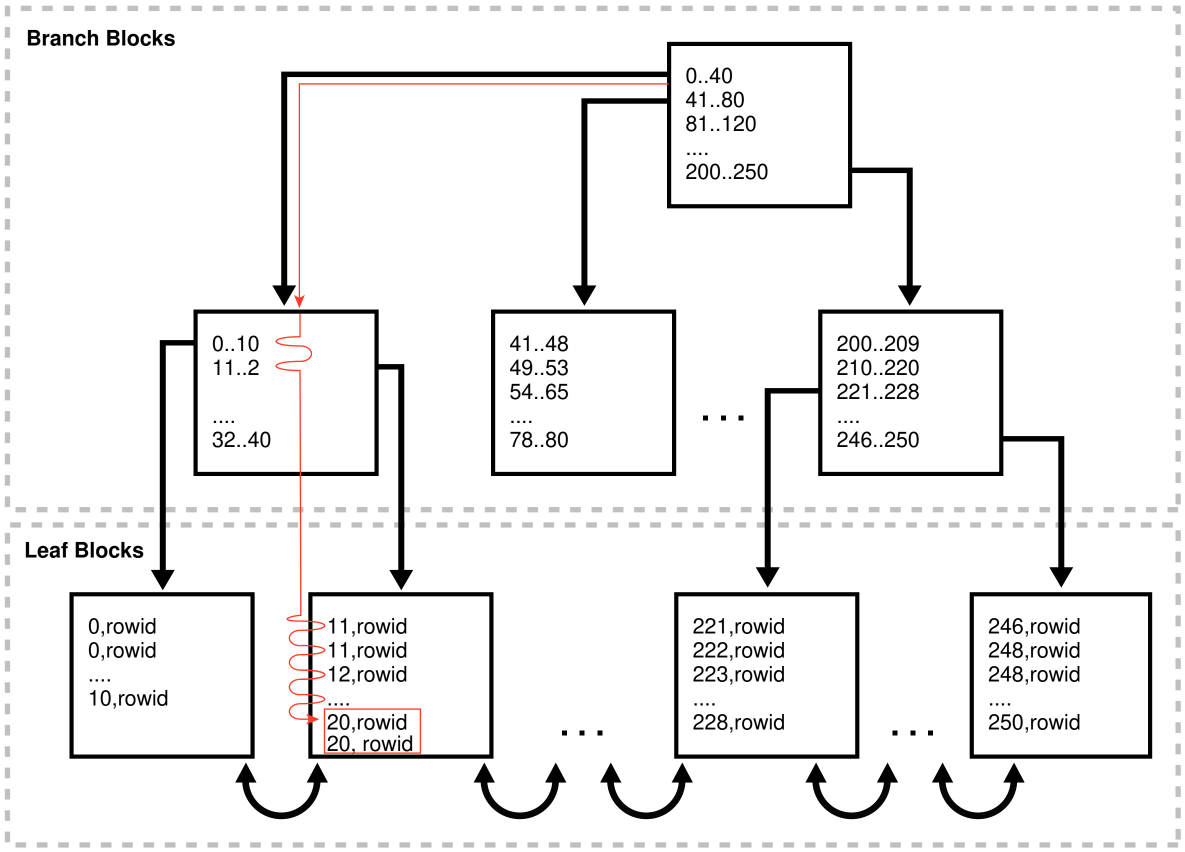
   Note:

   In some cases, an index scan reads a set of index blocks, sorts the rowids, and then reads a set of table blocks.

Thus, to scan the index, the database moves backward or forward through the leaf blocks. For example, a scan for IDs between 20 and 40 locates the first leaf block that has the lowest key value that is 20 or greater. The scan proceeds horizontally through the linked list of leaf nodes until it finds a value greater than 40, and then stops.

The following figure illustrates an index range scan using ascending order. A statement requests the `employees` records with the value `20` in the `department_id` column, which has a nonunique index. In this example, 2 index entries for department `20` exist.

Figure 8-5 Index Range Scan



8.3.3.3 Index Range Scan: Example

This example retrieves a set of values from the `employees` table using an index range scan.

The following statement queries the records for employees in department `20` with salaries greater than `1000`:

```
SELECT *
FROM   employees
WHERE  department_id = 20
AND    salary > 1000;
```

The preceding query has low cardinality (returns few rows), so the query uses the index on the `department_id` column. The database scans the index, fetches the records from the employees table, and then applies the `salary > 1000` filter to these fetched records to generate the result.

```
SQL_ID  brt5abvbxw9tq, child number 0
-------------------------------------
SELECT * FROM   employees WHERE  department_id = 20 AND    salary > 1000

Plan hash value: 2799965532
```

```
--------------------------------------------------------------------------------------
-----
|Id | Operation                            | Name              |Rows|Bytes|Cost(%CPU)|
Time |
--------------------------------------------------------------------------------------
-----
| 0 | SELECT STATEMENT                     |                   |    |     | 2
(100)|         |
|*1 |  TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES         |  2 | 138 | 2   (0)|
00:00:01|
|*2 |   INDEX RANGE SCAN                   | EMP_DEPARTMENT_IX|  2 |     | 1   (0)|
00:00:01|
--------------------------------------------------------------------------------------
-----

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("SALARY">1000)
   2 - access("DEPARTMENT_ID"=20)
```

8.3.3.4 Index Range Scan Descending: Example

This example uses an index to retrieve rows from the `employees` table in sorted order.

The following statement queries the records for employees in department `20` in descending order:

```
SELECT *
FROM   employees
WHERE  department_id < 20
ORDER BY department_id DESC;
```

This preceding query has low cardinality, so the query uses the index on the `department_id` column.

```
SQL_ID  8182ndfj1ttj6, child number 0
-------------------------------------
SELECT * FROM employees WHERE department_id<20 ORDER BY department_id
DESC

Plan hash value: 1681890450
---------------------------------------------------------------------------
----
|Id| Operation                    | Name         |Rows|Bytes|Cost(%CPU)|
Time |
---------------------------------------------------------------------------
----
| 0| SELECT STATEMENT             |                  | |   |
2(100)|         |
| 1|  TABLE ACCESS BY INDEX ROWID |EMPLOYEES        |2|138|2   (0)|
00:00:01|
|*2|   INDEX RANGE SCAN DESCENDING|EMP_DEPARTMENT_IX|2|   |1   (0)|
00:00:01|
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"<20)
```

The database locates the first index leaf block that contains the highest key value that is `20` or less. The scan then proceeds horizontally to the left through the linked list of leaf nodes. The database obtains the rowid from each index entry, and then retrieves the row specified by the rowid.

8.3.4 Index Full Scans

An **index full scan** reads the entire index in order. An index full scan can eliminate a separate sorting operation because the data in the index is ordered by index key.

8.3.4.1 When the Optimizer Considers Index Full Scans

The optimizer considers an index full scan in a variety of situations.

The situations include the following:

- A predicate references a column in the index. This column need not be the leading column.
- No predicate is specified, but all of the following conditions are met:
  - All columns in the table and in the query are in the index.
  - At least one indexed column is not null.
- A query includes an `ORDER BY` on indexed non-nullable columns.

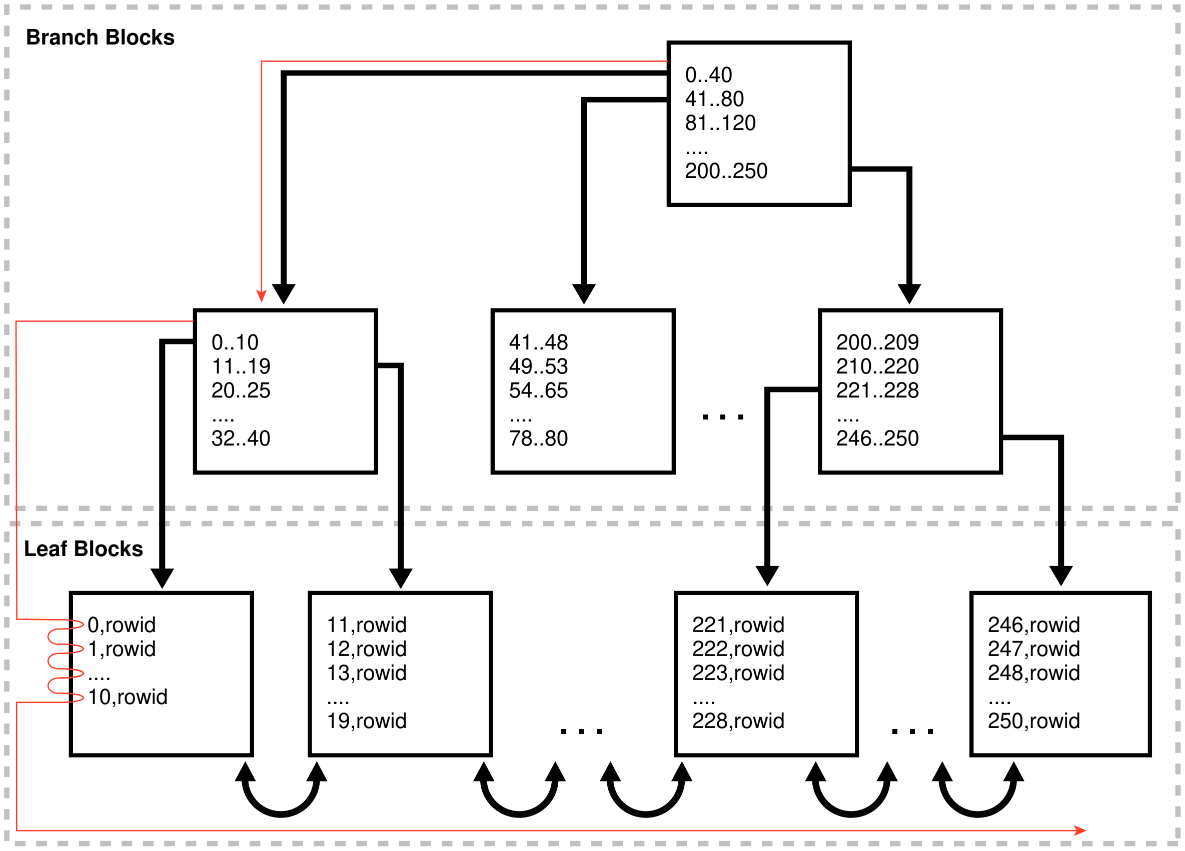
8.3.4.2 How Index Full Scans Work

The database reads the root block, and then navigates down the left hand side of the index (or right if doing a descending full scan) until it reaches a leaf block.

Then the database reaches a leaf block, the scan proceeds across the bottom of the index, one block at a time, in sorted order. The database uses single-block I/O rather than multiblock I/O.

The following graphic illustrates an index full scan. A statement requests the `departments` records ordered by `department_id`.

Figure 8-6 Index Full Scan



8.3.4.3 Index Full Scans: Example

This example uses an index full scan to satisfy a query with an `ORDER BY` clause.

The following statement queries the ID and name for departments in order of department ID:

```
SELECT department_id, department_name
FROM   departments
ORDER BY department_id;
```

The following plan shows that the optimizer chose an index full scan:

```
SQL_ID  94t4a20h8what, child number 0
-------------------------------------
select department_id, department_name from departments order by
department_id

Plan hash value: 4179022242

------------------------------------------------------------------------
-
|Id | Operation                  | Name      |Rows|Bytes|Cost(%CPU)|Time
```

```
|
-----------------------------------------------------------------------
|0| SELECT STATEMENT            |             |   |   |2 (100)|        |
|1|  TABLE ACCESS BY INDEX ROWID|DEPARTMENTS |27 |432|2    (0)|00:00:01 |
|2|   INDEX FULL SCAN           |DEPT_ID_PK  |27 |   |1    (0)|00:00:01 |
-----------------------------------------------------------------------
```

The database locates the first index leaf block, and then proceeds horizontally to the right through the linked list of leaf nodes. For each index entry, the database obtains the rowid from the entry, and then retrieves the table row specified by the rowid. Because the index is sorted on `department_id`, the database avoids a separate operation to sort the retrieved rows.

8.3.5 Index Fast Full Scans

An **index fast full scan** reads the index blocks in unsorted order, as they exist on disk. This scan does not use the index to probe the table, but reads the index instead of the table, essentially using the index itself as a table.

8.3.5.1 When the Optimizer Considers Index Fast Full Scans

The optimizer considers this scan when a query only accesses attributes in the index.

Note:

Unlike a full scan, a fast full scan cannot eliminate a sort operation because it does not read the index in order.

The `INDEX_FFS(`*`table_name index_name`*`)` hint forces a fast full index scan.

See Also:

Oracle Database SQL Language Reference to learn more about the `INDEX` hint

8.3.5.2 How Index Fast Full Scans Work

The database uses multiblock I/O to read the root block and all of the leaf and branch blocks. The databases ignores the branch and root blocks and reads the index entries on the leaf blocks.

8.3.5.3 Index Fast Full Scans: Example

This examples uses a fast full index scan as a result of an optimizer hint.

The following statement queries the ID and name for departments in order of department ID:

```
SELECT /*+ INDEX_FFS(departments dept_id_pk) */ COUNT(*)
FROM   departments;
```

The following plan shows that the optimizer chose a fast full index scan:

```
SQL_ID  fu0k5nvx7sftm, child number 0
-------------------------------------
select /*+ index_ffs(departments dept_id_pk) */ count(*) from
departments

Plan hash value: 3940160378
-----------------------------------------------------------------------
---
| Id | Operation              | Name         | Rows  |Cost (%CPU)|
Time     |
-----------------------------------------------------------------------
---
|  0 | SELECT STATEMENT       |              |       |    2
(100)|          |
|  1 |  SORT AGGREGATE        |              |     1 |
|          |
|  2 |   INDEX FAST FULL SCAN| DEPT_ID_PK |    27 |    2   (0)|
00:00:01 |
-----------------------------------------------------------------------
---
```

8.3.6 Index Skip Scans

An **index skip scan** occurs when the initial column of a composite index is "skipped" or not specified in the query.

See Also:

Oracle Database Concepts

8.3.6.1 When the Optimizer Considers Index Skip Scans

Often, skip scanning index blocks is faster than scanning table blocks, and faster than performing full index scans.

The optimizer considers a skip scan when the following criteria are met:

- The leading column of a composite index is not specified in the query predicate.

  For example, the query predicate does not reference the `cust_gender` column, and the composite index key is `(cust_gender,cust_email)`.
- Many distinct values exist in the nonleading key of the index and relatively few distinct values exist in the leading key.

  For example, if the composite index key is `(cust_gender,cust_email)`, then the `cust_gender` column has only two distinct values, but `cust_email` has thousands.

8.3.6.2 How Index Skip Scans Work

An index skip scan logically splits a composite index into smaller subindexes.

The number of distinct values in the leading columns of the index determines the number of logical subindexes. The lower the number, the fewer logical subindexes the optimizer must create, and the more efficient the scan becomes. The scan reads each logical index separately, and "skips" index blocks that do not meet the filter condition on the non-leading column.

8.3.6.3 Index Skip Scans: Example

This example uses an index skip scan to satisfy a query of the `sh.customers` table.

The `customers` table contains a column `cust_gender` whose values are either `M` or `F`. While logged in to the database as user `sh`, you create a composite index on the columns (`cust_gender, cust_email`) as follows:

```
CREATE INDEX cust_gender_email_ix
  ON sh.customers (cust_gender, cust_email);
```

Conceptually, a portion of the index might look as follows, with the gender value of `F` or `M` as the leading edge of the index.

```
F,Wolf@company.example.com,rowid
F,Wolsey@company.example.com,rowid
F,Wood@company.example.com,rowid
F,Woodman@company.example.com,rowid
F,Yang@company.example.com,rowid
F,Zimmerman@company.example.com,rowid
M,Abbassi@company.example.com,rowid
M,Abbey@company.example.com,rowid
```

You run the following query for a customer in the `sh.customers` table:

```
SELECT *
FROM   sh.customers
WHERE  cust_email = 'Abbey@company.example.com';
```

The database can use a skip scan of the `customers_gender_email` index even though `cust_gender` is not specified in the `WHERE` clause. In the sample index, the leading column `cust_gender` has two possible values: `F` and `M`. The database logically splits the index into two. One subindex has the key `F`, with entries in the following form:

```
F,Wolf@company.example.com,rowid
F,Wolsey@company.example.com,rowid
F,Wood@company.example.com,rowid
F,Woodman@company.example.com,rowid
F,Yang@company.example.com,rowid
F,Zimmerman@company.example.com,rowid
```

The second subindex has the key `M`, with entries in the following form:

```
M,Abbassi@company.example.com,rowid
M,Abbey@company.example.com,rowid
```

When searching for the record for the customer whose email is `Abbey@company.example.com`, the database searches the subindex with the leading value `F` first, and then searches the subindex with the leading value `M`. Conceptually, the database processes the query as follows:

```
( SELECT *
  FROM   sh.customers
  WHERE  cust_gender = 'F'
  AND    cust_email = 'Abbey@company.example.com' )
UNION ALL
( SELECT *
  FROM   sh.customers
  WHERE  cust_gender = 'M'
  AND    cust_email = 'Abbey@company.example.com' )
```

The plan for the query is as follows:

```
SQL_ID  d7a6xurcnx2dj, child number 0
-------------------------------------
SELECT * FROM   sh.customers WHERE  cust_email = 'Abbey@company.example.com'

Plan hash value: 797907791

-----------------------------------------------------------------------------------
---
|Id| Operation                          | Name                 |Rows|Bytes|Cost(%CPU)|
Time|
-----------------------------------------------------------------------------------
---
| 0|SELECT STATEMENT                    |                      |  |  |    |
10(100)|        |
| 1| TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS            |33|6237|  10(0)|
00:00:01|
|*2|  INDEX SKIP SCAN                   | CUST_GENDER_EMAIL_IX |33|    |   4(0)|
00:00:01|
-----------------------------------------------------------------------------------
---

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("CUST_EMAIL"='Abbey@company.example.com')
       filter("CUST_EMAIL"='Abbey@company.example.com')
```

See Also:

Oracle Database Concepts to learn more about skip scans

8.3.7 Index Join Scans

An index join scan is a hash join of multiple indexes that together return all columns requested by a query. The database does not need to access the table because all data is retrieved from the indexes.

8.3.7.1 When the Optimizer Considers Index Join Scans

In some cases, avoiding table access is the most cost efficient option.

The optimizer considers an index join in the following circumstances:

- A hash join of multiple indexes retrieves all data requested by the query, without requiring table access.
- The cost of retrieving rows from the table is higher than reading the indexes without retrieving rows from the table. An index join is often expensive. For example, when scanning two indexes and joining them, it is often less costly to choose the most selective index, and then probe the table.

You can specify an index join with the `INDEX_JOIN(`*`table_name`*`)` hint.

See Also:

Oracle Database SQL Language Reference

8.3.7.2 How Index Join Scans Work

An index join involves scanning multiple indexes, and then using a hash join on the rowids obtained from these scans to return the rows.

In an index join scan, table access is always avoided. For example, the process for joining two indexes on a single table is as follows:

1. Scan the first index to retrieve rowids.
2. Scan the second index to retrieve rowids.
3. Perform a hash join by rowid to obtain the rows.

8.3.7.3 Index Join Scans: Example

This example queries the last name and email for employees whose last name begins with `A`, specifying an index join.

```
SELECT /*+ INDEX_JOIN(employees) */ last_name, email
FROM   employees
WHERE  last_name like 'A%';
```

Separate indexes exist on the `(last_name,first_name)` and `email` columns. Part of the `emp_name_ix` index might look as follows:

```
Banda,Amit,AAAVgdAALAAAABSABD
Bates,Elizabeth,AAAVgdAALAAAABSABI
Bell,Sarah,AAAVgdAALAAAABSABc
Bernstein,David,AAAVgdAALAAAABSAAz
Bissot,Laura,AAAVgdAALAAAABSAAd
Bloom,Harrison,AAAVgdAALAAAABSABF
Bull,Alexis,AAAVgdAALAAAABSABV
```

The first part of the `emp_email_uk` index might look as follows:

```
ABANDA,AAAVgdAALAAAABSABD
ABULL,AAAVgdAALAAAABSABV
ACABRIO,AAAVgdAALAAAABSABX
AERRAZUR,AAAVgdAALAAAABSAAv
AFRIPP,AAAVgdAALAAAABSAAV
AHUNOLD,AAAVgdAALAAAABSAAD
AHUTTON,AAAVgdAALAAAABSABL
```

The following example retrieves the plan using the `DBMS_XPLAN.DISPLAY_CURSOR` function. The database retrieves all rowids in the `emp_email_uk` index, and then retrieves rowids in `emp_name_ix` for last names that begin with `A`. The database uses a hash join to search both sets of rowids for matches. For example, rowid `AAAVgdAALAAAABSABD` occurs in both sets of rowids, so the database probes the `employees` table for the record corresponding to this rowid.

Example 8-4 Index Join Scan

```
SQL_ID  d2djchyc9hmrz, child number 0
-------------------------------------
SELECT /*+ INDEX_JOIN(employees) */ last_name, email FROM   employees
WHERE  last_name like 'A%'

Plan hash value: 3719800892
-------------------------------------------------------------------------------------
-----
| Id  | Operation              | Name             | Rows  | Bytes | Cost (%CPU)|
Time     |
-------------------------------------------------------------------------------------
-----
|   0 | SELECT STATEMENT       |                  |       |       |     3     
(100)|          |
|*  1 |  VIEW                  | index$_join$_001 |     3 |    48 |     3  (34)|
00:00:01 |
|*  2 |   HASH JOIN            |                  |       |       |            
|          |
|*  3 |    INDEX RANGE SCAN    | EMP_NAME_IX      |     3 |    48 |     1   (0)|
00:00:01 |
|   4 |    INDEX FAST FULL SCAN| EMP_EMAIL_UK     |     3 |    48 |     1   (0)|
00:00:01 |
-------------------------------------------------------------------------------------
```

```
-----

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("LAST_NAME" LIKE 'A%')
   2 - access(ROWID=ROWID)
   3 - access("LAST_NAME" LIKE 'A%')
```

8.4 Bitmap Index Access Paths

Bitmap indexes combine the indexed data with a rowid range.

8.4.1 About Bitmap Index Access

In a conventional B-tree index, one index entry points to a single row. In a bitmap index, the key is the combination of the indexed data and the rowid range.

The database stores at least one bitmap for each index key. Each value in the bitmap, which is a series of `1` and `0` values, points to a row within a rowid range. Thus, in a bitmap index, one index entry points to a set of rows rather than a single row.

8.4.1.1 Differences Between Bitmap and B-Tree Indexes

A bitmap index uses a different key from a B-tree index, but is stored in a B-tree structure.

The following table shows the differences among types of index entries.

Table 8-3 Index Entries for B-Trees and Bitmaps

Index Entry	Key	Data	Example
Unique B-tree	Indexed data only	Rowid	In an entry of the index on the `employees.employee_id` column, employee ID `101` is the key, and the rowid `AAAPvCAAFAAAAFaAAa` is the data: `101,AAAPvCAAFAAAAFaAAa`
Nonunique B-tree	Indexed data combined with rowid	None	In an entry of the index on the `employees.last_name` column, the name and rowid combination `Smith,AAAPvCAAFAAAAFaAAa` is the key, and there is no data: `Smith,AAAPvCAAFAAAAFaAAa`
Bitmap	Indexed data combined with rowid range	Bitmap	In an entry of the index on the `customers.cust_gender` column, the `M,`*`low-rowid,high-rowid`* part is the key, and the series of `1` and `0` values is the data: `M,`*`low-rowid,high-rowid`*`,1000101010101010`

The database stores a bitmap index in a B-tree structure. The database can search the B-tree quickly on the first part of the key, which is the set of attributes on which the index is defined, and then obtain the corresponding rowid range and bitmap.

See Also:

- "Bitmap Storage"
- *Oracle Database Concepts* for an overview of bitmap indexes
- *Oracle Database Data Warehousing Guide* for more information about bitmap indexes

8.4.1.2 Purpose of Bitmap Indexes

Bitmap indexes are typically suitable for infrequently modified data with a low or medium number of distinct values (NDV).

In general, B-tree indexes are suitable for columns with high NDV and frequent DML activity. For example, the optimizer might choose a B-tree index for a query of a `sales.amount` column that returns few rows. In contrast, the `customers.state` and `customers.county` columns are candidates for bitmap indexes because they have few distinct values, are infrequently updated, and can benefit from efficient `AND` and `OR` operations.

Bitmap indexes are a useful way to speed ad hoc queries in a data warehouse. They are fundamental to star transformations. Specifically, bitmap indexes are useful in queries that contain the following:

- Multiple conditions in the `WHERE` clause

  Before the table itself is accessed, the database filters out rows that satisfy some, but not all, conditions.

- `AND`, `OR`, and `NOT` operations on columns with low or medium NDV

  Combining bitmap indexes makes these operations more efficient. The database can merge bitmaps from bitmap indexes very quickly. For example, if bitmap indexes exist on the `customers.state` and `customers.county` columns, then these indexes can enormously improve the performance of the following query:

  ```
  SELECT *
  FROM   customers
  WHERE  state = 'CA'
  AND    county = 'San Mateo'
  ```

  The database can convert `1` values in the merged bitmap into rowids efficiently.

- The `COUNT` function

  The database can scan the bitmap index without needing to scan the table.

- Predicates that select for null values

  Unlike B-tree indexes, bitmap indexes can contain nulls. Queries that count the number of nulls in a column can use the bitmap index without scanning the table.

- Columns that do not experience heavy DML

  The reason is that one index key points to many rows. If a session modifies the indexed data, then the database cannot lock a single bit in the bitmap: rather, the database locks the entire index *entry*, which in practice locks the rows pointed to by the bitmap. For example, if the county of residence for a specific customer changes from `San Mateo` to `Alameda`, then the database must get exclusive access to the `San Mateo` index entry and `Alameda` index entry in the bitmap. Rows containing these two values cannot be modified until `COMMIT`.

See Also:

- "Star Transformation"
- *Oracle Database SQL Language Reference* to learn about the `COUNT` function

8.4.1.3 Bitmaps and Rowids

For a particular value in a bitmap, the value is `1` if the row values match the bitmap condition, and `0` if it does not. Based on these values, the database uses an internal algorithm to map bitmaps onto rowids.

The bitmap entry contains the indexed value, the rowid range (start and end rowids), and a bitmap. Each `0` or `1` value in the bitmap is an offset into the rowid range, and maps to a potential row in the table, even if the row does not exist. Because the number of possible rows in a block is predetermined, the database can use the range endpoints to determine the rowid of an arbitrary row in the range.

Note:

The Hakan factor is an optimization used by the bitmap index algorithms to limit the number of rows that Oracle Database assumes can be stored in a single block. By artificially limiting the number of rows, the database reduces the size of the bitmaps.

Table 8-4 shows part of a sample bitmap for the `sh.customers.cust_marital_status` column, which is nullable. The actual index has 12 distinct values. Only 3 are shown in the sample: null, `married`, and `single`.

Table 8-4 Bitmap Index Entries

Column Value for cust_marital_status	Start Rowid in Range	End Rowid in Range	1st Row in Range	2nd Row in Range	3rd Row in Range	4th Row in Range	5th Row in Range	6th Row in Range
(null)	`AAA ...`	`CCC ...`	0	0	0	0	0	1
`married`	`AAA ...`	`CCC ...`	1	0	1	1	1	0
`single`	`AAA ...`	`CCC ...`	0	1	0	0	0	0
`single`	`DDD ...`	`EEE ...`	1	0	1	0	1	1

As shown in Table 8-4, bitmap indexes can include keys that consist entirely of null values, unlike B-tree indexes. In Table 8-4, the null has a value of `1` for the 6th row in the range, which means that the `cust_marital_status` value is null for the 6th row in the range. Indexing nulls can be useful for some SQL statements, such as queries with the aggregate function `COUNT`.

See Also:

Oracle Database Concepts to learn about rowid formats

8.4.1.4 Bitmap Join Indexes

A **bitmap join index** is a bitmap index for the join of two or more tables.

The optimizer can use a bitmap join index to reduce or eliminate the volume of data that must be joined during plan execution. Bitmap join indexes can be much more efficient in storage than materialized join views.

The following example creates a bitmap index on the `sh.sales` and `sh.customers` tables:

```
CREATE BITMAP INDEX cust_sales_bji ON sales(c.cust_city)
  FROM sales s, customers c
  WHERE c.cust_id = s.cust_id LOCAL;
```

The `FROM` and `WHERE` clause in the preceding `CREATE` statement represent the join condition between the tables. The `customers.cust_city` column is the index key.

Each key value in the index represents a possible city in the `customers` table. Conceptually, key values for the index might look as follows, with one bitmap associated with each key value:

```
San Francisco   0 0 0 1 0 1 0 0 0 1 0 0 0 0 0 . . .
San Mateo       0 1 0 0 0 0 0 0 1 0 0 0 0 0 1 . . .
Smithville      1 0 0 0 1 0 0 1 0 0 1 0 1 0 0 . . .
.
.
.
```

Each bit in a bitmap corresponds to one row in the `sales` table. In the `Smithville` key, the value `1` means that the first row in the `sales` table corresponds to a product sold to a Smithville customer, whereas the value `0` means that the second row corresponds to a product not sold to a Smithville customer.

Consider the following query of the number of separate sales to Smithville customers:

```
SELECT COUNT (*)
FROM   sales s, customers c
WHERE  c.cust_id = s.cust_id
AND    c.cust_city = 'Smithville';
```

The following plan shows that the database reads the `Smithville` bitmap to derive the number of Smithville sales (Step 4), thereby avoiding a join of the `customers` and `sales` tables.

```
SQL_ID  57s100mh142wy, child number 0
-------------------------------------
SELECT COUNT (*) FROM sales s, customers c WHERE c.cust_id = s.cust_id
AND c.cust_city = 'Smithville'

Plan hash value: 3663491772

------------------------------------------------------------------------------------
|Id| Operation                    | Name  |Rows|Bytes|Cost (%CPU)| Time|Pstart|Pstop|
------------------------------------------------------------------------------------
| 0| SELECT STATEMENT             |              |     |    |29 (100)|        | |  |
| 1|  SORT AGGREGATE              |              |   1 |   5|        |        | |  |
| 2|   PARTITION RANGE ALL        |              | 1708|8540|29   (0)|00:00:01|1|28|
| 3|    BITMAP CONVERSION COUNT   |              | 1708|8540|29   (0)|00:00:01| |  |
|*4|     BITMAP INDEX SINGLE VALUE|CUST_SALES_BJI|     |    |        |        |1|28|
------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("S"."SYS_NC00008$"='Smithville')
```

See Also:

Oracle Database Concepts to learn about the `CREATE INDEX` statement

8.4.1.5 Bitmap Storage

A bitmap index resides in a B-tree structure, using branch blocks and leaf blocks just as in a B-tree.

For example, if the `customers.cust_marital_status` column has 12 distinct values, then one branch block might point to the keys `married,`*`rowid-range`* and `single,`*`rowid-range`*, another branch block might point to the `widowed,`*`rowid-range`* key, and so on. Alternatively, a single branch block could point to a leaf block containing all 12 distinct keys.

Each indexed column value may have one or more bitmap pieces, each with its own rowid range occupying a contiguous set of rows in one or more extents. The database can use a bitmap piece to break up an index entry that is large relative to the size of a block. For example, the database could break a single index entry into three pieces, with the first two pieces in separate blocks in the same extent, and the last piece in a separate block in a different extent.

To conserve space, Oracle Database can compression consecutive ranges of `0` values.

8.4.2 Bitmap Conversion to Rowid

A bitmap conversion translates between an entry in the bitmap and a row in a table. The conversion can go from entry to row (`TO ROWID`), or from row to entry (`FROM ROWID`).

8.4.2.1 When the Optimizer Chooses Bitmap Conversion to Rowid

The optimizer uses a conversion whenever it retrieves a row from a table using a bitmap index entry.

8.4.2.2 How Bitmap Conversion to Rowid Works

Conceptually, a bitmap can be represented as table.

For example, Table 8-4 represents the bitmap as a table with `customers` row numbers as columns and `cust_marital_status` values as rows. Each field in Table 8-4 has the value `1` or `0`, and represents a column value in a row. Conceptually, the bitmap conversion uses an internal algorithm that says, "Field *F* in the bitmap corresponds to the *N*th row of the *M*th block of the table," or "The *N*th row of the *M*th block in the table corresponds to field *F* in the bitmap."

8.4.2.3 Bitmap Conversion to Rowid: Example

In this example, the optimizer chooses a bitmap conversion operation to satisfy a query using a range predicate.

A query of the `sh.customers` table selects the names of all customers born before 1918:

```
SELECT cust_last_name, cust_first_name
FROM   customers
WHERE  cust_year_of_birth < 1918;
```

The following plan shows that the database uses a range scan to find all key values less than `1918` (Step 3), converts the `1` values in the bitmap to rowids (Step 2), and then uses the rowids to obtain the rows from the `customers` table (Step 1):

```
-----------------------------------------------------------------------------------
-----
|Id| Operation                          | Name             |Rows|Bytes|Cost(%CPU)|
Time  |
-----------------------------------------------------------------------------------
-----
| 0| SELECT STATEMENT                   |                  |    |     |421
(100)|         |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED| CUSTOMERS        |3604|68476|421   (1)|
00:00:01|
| 2|   BITMAP CONVERSION TO ROWIDS      |                  |    |     |
|        |
|*3|    BITMAP INDEX RANGE SCAN         | CUSTOMERS_YOB_BIX|    |     |
|        |
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("CUST_YEAR_OF_BIRTH"<1918)
       filter("CUST_YEAR_OF_BIRTH"<1918)
```

8.4.3 Bitmap Index Single Value

This type of access path uses a bitmap index to look up a single key value.

8.4.3.1 When the Optimizer Considers Bitmap Index Single Value

The optimizer considers this access path when the predicate contains an equality operator.

8.4.3.2 How Bitmap Index Single Value Works

The query scans a single bitmap for positions containing a `1` value. The database converts the `1` values into rowids, and then uses the rowids to find the rows.

The database only needs to process a single bitmap. For example, the following table represents the bitmap index (in two bitmap pieces) for the value `widowed` in the `sh.customers.cust_marital_status` column. To satisfy a query of customers with the status `widowed`, the database can search for each `1` value in the `widowed` bitmap and find the rowid of the corresponding row.

Table 8-5 Bitmap Index Entries

Column Value	Start Rowid in Range	End Rowid in Range	1st Row in Range	2nd Row in Range	3rd Row in Range	4th Row in Range	5th Row in Range	6th Row in Range
`widowed`	`AAA ...`	`CCC ...`	0	1	0	0	0	0
`widowed`	`DDD ...`	`EEE ...`	1	0	1	0	1	1

8.4.3.3 Bitmap Index Single Value: Example

In this example, the optimizer chooses a bitmap index single value operation to satisfy a query that uses an equality predicate.

A query of the `sh.customers` table selects all widowed customers:

```
SELECT *
FROM   customers
WHERE  cust_marital_status = 'Widowed';
```

The following plan shows that the database reads the entry with the `Widowed` key in the `customers` bitmap index (Step 3), converts the `1` values in the bitmap to rowids (Step 2), and then uses the rowids to obtain the rows from the `customers` table (Step 1):

```
SQL_ID  ff5an2xsn086h, child number 0
-------------------------------------
```

```
SELECT * FROM customers WHERE cust_marital_status = 'Widowed'

Plan hash value: 2579015045
-------------------------------------------------------------------------------------
-----
|Id| Operation                           | Name                 |Rows|Bytes|Cost (%CPU)|
Time|
-------------------------------------------------------------------------------------
-----
| 0|SELECT STATEMENT                     |                      |    |     |
412(100)|        |
| 1| TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS             |3461|638K|412  (2)|
00:00:01|
| 2|  BITMAP CONVERSION TO ROWIDS        |                      |    |     |
|        |
|*3|   BITMAP INDEX SINGLE VALUE         |CUSTOMERS_MARITAL_BIX|    |     |
|        |
-------------------------------------------------------------------------------------
-----

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("CUST_MARITAL_STATUS"='Widowed')
```

8.4.4 Bitmap Index Range Scans

This type of access path uses a bitmap index to look up a range of values.

8.4.4.1 When the Optimizer Considers Bitmap Index Range Scans

The optimizer considers this access path when the predicate selects a range of values.

The range in the scan can be bounded on both sides, or unbounded on one or both sides. The optimizer typically chooses a range scan for selective queries.

See Also:

"Index Range Scans"

8.4.4.2 How Bitmap Index Range Scans Work

This scan works similarly to a B-tree range scan.

For example, the following table represents three values in the bitmap index for the `sh.customers.cust_year_of_birth` column. If a query requests all customers born before 1917, then the database can scan this index for values lower than `1917`, and then obtain the rowids for rows that have a `1`.

Table 8-6 Bitmap Index Entries

Column Value	Start Rowid in Range	End Rowid in Range	1st Row in Range	2nd Row in Range	3rd Row in Range	4th Row in Range	5th Row in Range	6th Row in Range
1913	AAA ...	CCC ...	0	0	0	0	0	1
1917	AAA ...	CCC ...	1	0	1	1	1	0
1918	AAA ...	CCC ...	0	1	0	0	0	0
1918	DDD ...	EEE ...	1	0	1	0	1	1

See Also:

"Index Range Scans"

8.4.4.3 Bitmap Index Range Scans: Example

This example uses a range scan to select customers born before a single year.

A query of the `sh.customers` table selects the names of customers born before 1918:

```
SELECT cust_last_name, cust_first_name
FROM   customers
WHERE  cust_year_of_birth < 1918
```

The following plan shows that the database obtains all bitmaps for `cust_year_of_birth` keys lower than `1918` (Step 3), converts the bitmaps to rowids (Step 2), and then fetches the rows (Step 1):

```
SQL_ID  672z2h9rawyjg, child number 0
-------------------------------------
SELECT cust_last_name, cust_first_name FROM   customers WHERE
cust_year_of_birth < 1918

Plan hash value: 4198466611

-------------------------------------------------------------------------------------------
|Id| Operation                          | Name              |Rows|Bytes|Cost(%CPU)|Time    |
-------------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                   |    |     |421 (100)|          |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS          |3604|68476|421   (1)|00:00:01|
| 2|   BITMAP CONVERSION TO ROWIDS      |                   |    |     |          |        |
|*3|    BITMAP INDEX RANGE SCAN         |CUSTOMERS_YOB_BIX |    |     |          |        |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("CUST_YEAR_OF_BIRTH"<1918)
       filter("CUST_YEAR_OF_BIRTH"<1918)
```

8.4.5 Bitmap Merge

This access path merges multiple bitmaps, and returns a single bitmap as a result.

A bitmap merge is indicated by the `BITMAP MERGE` operation in an execution plan.

8.4.5.1 When the Optimizer Considers Bitmap Merge

The optimizer typically uses a bitmap merge to combine bitmaps generated from a bitmap index range scan.

8.4.5.2 How Bitmap Merge Works

A merge uses a Boolean `OR` operation between two bitmaps. The resulting bitmap selects all rows from the first bitmap, plus all rows from every subsequent bitmap.

A query might select all customers born before 1918. The following example shows sample bitmaps for three `customers.cust_year_of_birth` keys: `1917`, `1916`, and `1915`. If any position in any bitmap has a `1`, then the merged bitmap has a `1` in the same position. Otherwise, the merged bitmap has a `0`.

```
1917      1  0  1  0  0  0  0  0  0  0  0  0  0  1
1916      0  1  0  0  0  0  0  0  0  0  0  0  0  0
1915      0  0  0  0  0  0  0  0  1  0  0  0  0  0
------------------------------------
merged:   1  1  1  0  0  0  0  0  1  0  0  0  0  1
```

The `1` values in resulting bitmap correspond to rows that contain the values `1915`, `1916`, or `1917`.

8.4.5.3 Bitmap Merge: Example

This example shows how the database merges bitmaps to optimize a query using a range predicate.

A query of the `sh.customers` table selects the names of female customers born before 1918:

```
SELECT cust_last_name, cust_first_name
FROM   customers
WHERE  cust_gender = 'F'
AND    cust_year_of_birth < 1918
```

The following plan shows that the database obtains all bitmaps for `cust_year_of_birth` keys lower than `1918` (Step 6), and then merges these bitmaps using `OR` logic to create a single bitmap (Step 5). The database obtains a single bitmap for the `cust_gender` key of `F` (Step 4), and then performs an `AND` operation on these two bitmaps. The result is a single bitmap that contains `1` values for the requested rows (Step 3).

```
SQL_ID  1xf59h179zdg2, child number 0
-------------------------------------
```

```
select cust_last_name, cust_first_name from customers where cust_gender
= 'F' and cust_year_of_birth < 1918

Plan hash value: 49820847
-------------------------------------------------------------------------------------------
|Id| Operation                           | Name                |Rows|Bytes|Cost(%CPU)|Time  |
-------------------------------------------------------------------------------------------
| 0|SELECT STATEMENT                     |                     |    |     |288(100)|       |
| 1| TABLE ACCESS BY INDEX ROWID BATCHED|CUSTOMERS            |1802|37842|288  (1)|00:00:01|
| 2|  BITMAP CONVERSION TO ROWIDS        |                     |    |     |        |        |
| 3|   BITMAP AND                        |                     |    |     |        |        |
|*4|    BITMAP INDEX SINGLE VALUE        |CUSTOMERS_GENDER_BIX|    |     |        |        |
| 5|    BITMAP MERGE                     |                     |    |     |        |        |
|*6|     BITMAP INDEX RANGE SCAN         |CUSTOMERS_YOB_BIX   |    |     |        |        |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access("CUST_GENDER"='F')
   6 - access("CUST_YEAR_OF_BIRTH"<1918)
       filter("CUST_YEAR_OF_BIRTH"<1918)
```

8.5 Table Cluster Access Paths

A **table cluster** is a group of tables that share common columns and store related data in the same blocks. When tables are clustered, a single data block can contain rows from multiple tables.

See Also:

Oracle Database Concepts for an overview of table clusters

8.5.1 Cluster Scans

An **index cluster** is a table cluster that uses an index to locate data.

The cluster index is a B-tree index on the cluster key. A cluster scan retrieves all rows that have the same cluster key value from a table stored in an indexed cluster.

8.5.1.1 When the Optimizer Considers Cluster Scans

The database considers a cluster scan when a query accesses a table in an indexed cluster.

8.5.1.2 How a Cluster Scan Works

In an indexed cluster, the database stores all rows with the same cluster key value in the same data block.

For example, if the `hr.employees2` and `hr.departments2` tables are clustered in `emp_dept_cluster`, and if the cluster key is `department_id`, then the database stores all employees in department 10 in the same block, all employees in department 20 in the same block, and so on.

The B-tree cluster index associates the cluster key value with the database block address (DBA) of the block containing the data. For example, the index entry for key 30 shows the address of the block that contains rows for employees in department 30:

```
30,AADAAAA9d
```

When a user requests rows in the cluster, the database scans the index to obtain the DBAs of the blocks containing the rows. Oracle Database then locates the rows based on these DBAs.

8.5.1.3 Cluster Scans: Example

This example clusters the employees and departments tables on the department_id column, and then queries the cluster for a single department.

As user hr, you create a table cluster, cluster index, and tables in the cluster as follows:

```
CREATE CLUSTER employees_departments_cluster
   (department_id NUMBER(4)) SIZE 512;

CREATE INDEX idx_emp_dept_cluster
   ON CLUSTER employees_departments_cluster;

CREATE TABLE employees2
   CLUSTER employees_departments_cluster (department_id)
   AS SELECT * FROM employees;
 CREATE TABLE departments2
   CLUSTER employees_departments_cluster (department_id)
   AS SELECT * FROM departments;
```

You query the employees in department 30 as follows:

```
SELECT *
FROM   employees2
WHERE  department_id = 30;
```

To perform the scan, Oracle Database first obtains the rowid of the row describing department 30 by scanning the cluster index (Step 2). Oracle Database then locates the rows in employees2 using this rowid (Step 1).

```
SQL_ID  b7xk1jzuwdc6t, child number 0
-------------------------------------
SELECT * FROM employees2 WHERE department_id = 30

Plan hash value: 49826199

-----------------------------------------------------------------------
----
|Id| Operation            | Name               |Rows|Bytes|Cost(%CPU)|
Time|
---------------------------------------------------------------------------
| 0| SELECT STATEMENT     |                     |   |     | 2  (100)|        |
| 1|  TABLE ACCESS CLUSTER| EMPLOYEES2          | 6 |798  | 2    (0)|00:00:01|
|*2|   INDEX UNIQUE SCAN  |IDX_EMP_DEPT_CLUSTER| 1 |     | 1    (0)|00:00:01|
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("DEPARTMENT_ID"=30)
```

See Also:

Oracle Database Concepts to learn about indexed clusters

8.5.2 Hash Scans

A **hash cluster** is like an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists.

In a hash cluster, the data *is* the index. The database uses a hash scan to locate rows in a hash cluster based on a hash value.

8.5.2.1 When the Optimizer Considers a Hash Scan

The database considers a hash scan when a query accesses a table in a hash cluster.

8.5.2.2 How a Hash Scan Works

In a hash cluster, all rows with the same hash value are stored in the same data block.

To perform a hash scan of the cluster, Oracle Database first obtains the hash value by applying a hash function to a cluster key value specified by the statement. Oracle Database then scans the data blocks containing rows with this hash value.

8.5.2.3 Hash Scans: Example

This example hashes the `employees` and `departments` tables on the `department_id` column, and then queries the cluster for a single department.

You create a hash cluster and tables in the cluster as follows:

```
CREATE CLUSTER employees_departments_cluster
   (department_id NUMBER(4)) SIZE 8192 HASHKEYS 100;

CREATE TABLE employees2
   CLUSTER employees_departments_cluster (department_id)
   AS SELECT * FROM employees;

CREATE TABLE departments2
```

```
   CLUSTER employees_departments_cluster (department_id)
   AS SELECT * FROM departments;
```

You query the employees in department `30` as follows:

```
SELECT *
FROM   employees2
WHERE  department_id = 30
```

To perform a hash scan, Oracle Database first obtains the hash value by applying a hash function to the key value `30`, and then uses this hash value to scan the data blocks and retrieve the rows (Step 1).

```
SQL_ID  919x7hyyxr6p4, child number 0
-------------------------------------
SELECT * FROM employees2 WHERE department_id = 30

Plan hash value: 2399378016

----------------------------------------------------------------
| Id  | Operation         | Name        | Rows  | Bytes | Cost  |
----------------------------------------------------------------
|   0 | SELECT STATEMENT  |             |       |       |     1 |
|*  1 |  TABLE ACCESS HASH| EMPLOYEES2 |    10 |  1330 |       |
----------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("DEPARTMENT_ID"=30)
```

See Also:

Oracle Database Concepts to learn about hash clusters

9

Joins

Oracle Database provides several optimizations for joining row sets.

9.1 About Joins

A **join** combines the output from exactly two row sources, such as tables or views, and returns one row source. The returned row source is the data set.

A join is characterized by multiple tables in the `WHERE` (non-ANSI) or `FROM ... JOIN` (ANSI) clause of a SQL statement. Whenever multiple tables exist in the `FROM` clause, Oracle Database performs a join.

A join condition compares two row sources using an expression. The join condition defines the relationship between the tables. If the statement does not specify a join condition, then the database performs a Cartesian join, matching every row in one table with every row in the other table.

See Also:

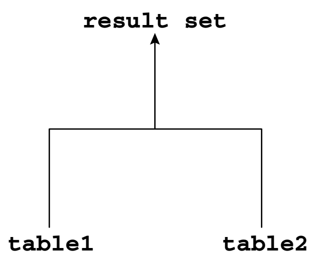
- "Cartesian Joins"
- *Oracle Database SQL Language Reference* for a concise discussion of joins in Oracle SQL

9.1.1 Join Trees

Typically, a join tree is represented as an upside-down tree structure.

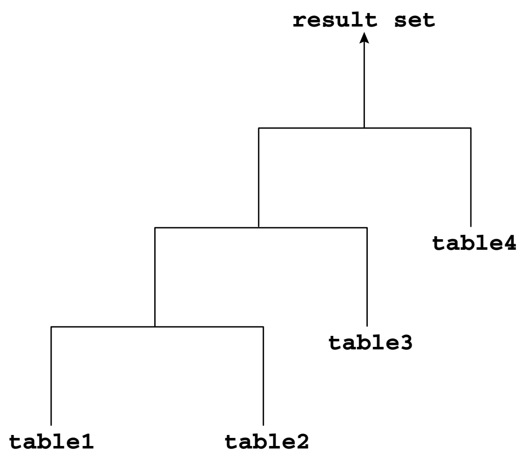
As shown in the following graphic, `table1` is the left table, and `table2` is the right table. The optimizer processes the join from left to right. For example, if this graphic depicted a nested loops join, then `table1` is the outer loop, and `table2` is the inner loop.

Figure 9-1 Join Tree



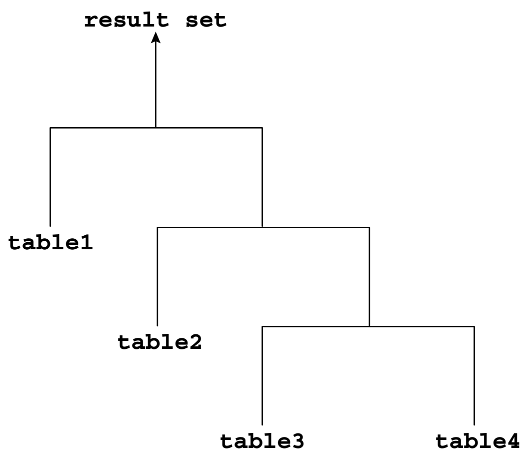
The input of a join can be the result set from a previous join. If the right child of every internal node of a join tree is a table, then the tree is a left deep join tree, as shown in the following example. Most join trees are left deep joins.

Figure 9-2 Left Deep Join Tree



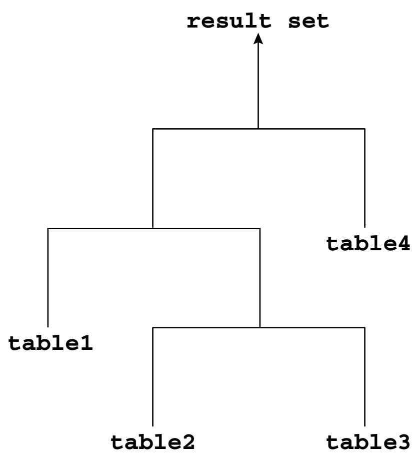
If the left child of every internal node of a join tree is a table, then the tree is called a right deep join tree, as shown in the following diagram.

Figure 9-3 Right Deep Join Tree



If the left or the right child of an internal node of a join tree can be a join node, then the tree is called a bushy join tree. In the following example, `table4` is a right child of a join node, `table1` is the left child of a join node, and `table2` is the left child of a join node.

Figure 9-4 Bushy Join Tree



In yet another variation, both inputs of a join are the results of a previous join.

9.1.2 How the Optimizer Executes Join Statements

The database joins pairs of row sources. When multiple tables exist in the `FROM` clause, the optimizer must determine which join operation is most efficient for each pair.

The optimizer must make the interrelated decisions shown in the following table.

Table 9-1 Join Operations

Operation	Explanation	To Learn More
Access paths	As for simple statements, the optimizer must choose an access path to retrieve data from each table in the join statement. For example, the optimizer might choose between a full table scan or an index scan..	"Optimizer Access Paths"
Join methods	To join each pair of row sources, Oracle Database must decide how to do it. The "how" is the join method. The possible join methods are nested loop, sort merge, and hash joins. A Cartesian join requires one of the preceding join methods. Each join method has specific situations in which it is more suitable than the others.	"Join Methods"
Join types	The join condition determines the join type. For example, an inner join retrieves only rows that match the join condition. An outer join retrieves rows that do not match the join condition.	"Join Types"

Table 9-1 (Cont.) Join Operations

Operation	Explanation	To Learn More
Join order	To execute a statement that joins more than two tables, Oracle Database joins two tables and then joins the resulting row source to the next table. This process continues until all tables are joined into the result. For example, the database joins two tables, and then joins the result to a third table, and then joins this result to a fourth table, and so on.	N/A

9.1.3 How the Optimizer Chooses Execution Plans for Joins

When determining the join order and method, the optimizer goal is to reduce the number of rows early so it performs less work throughout the execution of the SQL statement.

The optimizer generates a set of execution plans, according to possible join orders, join methods, and available access paths. The optimizer then estimates the cost of each plan and chooses the one with the lowest cost. When choosing an execution plan, the optimizer considers the following factors:

- The optimizer first determines whether joining two or more tables results in a row source containing at most one row.

  The optimizer recognizes such situations based on `UNIQUE` and `PRIMARY KEY` constraints on the tables. If such a situation exists, then the optimizer places these tables first in the join order. The optimizer then optimizes the join of the remaining set of tables.
- For join statements with outer join conditions, the table with the outer join operator typically comes after the other table in the condition in the join order.

  In general, the optimizer does not consider join orders that violate this guideline, although the optimizer overrides this ordering condition in certain circumstances. Similarly, when a subquery has been converted into an antijoin or semijoin, the tables from the subquery must come after those tables in the outer query block to which they were connected or correlated. However, hash antijoins and semijoins are able to override this ordering condition in certain circumstances.

The optimizer estimates the cost of a query plan by computing the estimated I/Os and CPU. These I/Os have specific costs associated with them: one cost for a single block I/O, and another cost for multiblock I/Os. Also, different functions and expressions have CPU costs associated with them. The optimizer determines the total cost of a query plan using these metrics. These metrics may be influenced by many initialization parameter and session settings at compile time, such as the `DB_FILE_MULTI_BLOCK_READ_COUNT` setting, system statistics, and so on.

For example, the optimizer estimates costs in the following ways:

- The cost of a nested loops join depends on the cost of reading each selected row of the outer table and each of its matching rows of the inner table into memory. The optimizer estimates these costs using statistics in the data dictionary.
- The cost of a sort merge join depends largely on the cost of reading all the sources into memory and sorting them.

- The cost of a hash join largely depends on the cost of building a hash table on one of the input sides to the join and using the rows from the other side of the join to probe it.

Example 9-1 Estimating Costs for Join Order and Method

Conceptually, the optimizer constructs a matrix of join orders and methods and the cost associated with each. For example, the optimizer must determine how best to join the `date_dim` and `lineorder` tables in a query. The following table shows the possible variations of methods and orders, and the cost for each. In this example, a nested loops join in the order `date_dim`, `lineorder` has the lowest cost.

Table 9-2 Sample Costs for Join of date_dim and lineorder Tables

Join Method	Cost of date_dim, lineorder	Cost of lineorder, date_dim
Nested Loops	39,480	6,187,540
Hash Join	187,528	194,909
Sort Merge	217,129	217,129

See Also:

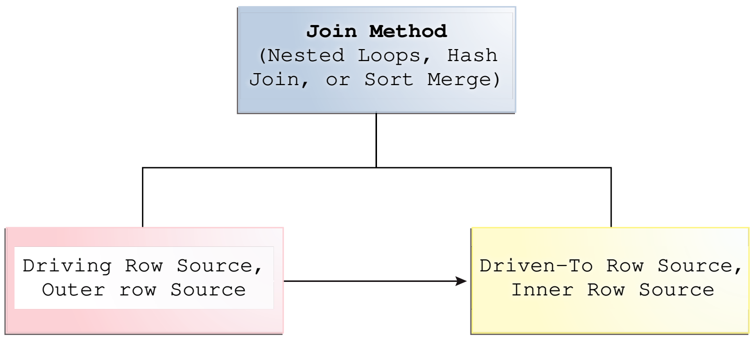
- "Introduction to Optimizer Statistics"
- "Influencing the Optimizer " for more information about optimizer hints
- *Oracle Database Reference* to learn about `DB_FILE_MULTIBLOCK_READ_COUNT`

9.2 Join Methods

A join method is the mechanism for joining two row sources.

Depending on the statistics, the optimizer chooses the method with the lowest estimated cost. As shown in Figure 9-5, each join method has two children: the driving (also called *outer*) row source and the driven-to (also called *inner*) row source.

Figure 9-5 Join Method


Join Method
(Nested Loops, Hash Join, or Sort Merge)
Driving Row Source, Outer row Source
Driven-To Row Source, Inner Row Source

9.2.1 Nested Loops Joins

Nested loops join an outer data set to an inner data set.

For each row in the outer data set that matches the single-table predicates, the database retrieves all rows in the inner data set that satisfy the join predicate. If an index is available, then the database can use it to access the inner data set by rowid.

9.2.1.1 When the Optimizer Considers Nested Loops Joins

Nested loops joins are useful when the database joins small subsets of data, the database joins large sets of data with the optimizer mode set to `FIRST_ROWS`, or the join condition is an efficient method of accessing the inner table.

Note:

The number of rows expected from the join is what drives the optimizer decision, not the size of the underlying tables. For example, a query might join two tables of a billion rows each, but because of the filters the optimizer expects data sets of 5 rows each.

In general, nested loops joins work best on small tables with indexes on the join conditions. If a row source has only one row, as with an equality lookup on a primary key value (for example, `WHERE employee_id=101`), then the join is a simple lookup. The optimizer always tries to put the smallest row source first, making it the driving table.

Various factors enter into the optimizer decision to use nested loops. For example, the database may read several rows from the outer row source in a batch. Based on the number of rows retrieved, the optimizer may choose either a nested loop or a hash join to the inner row source. For example, if a query joins `departments` to driving table `employees`, and if the predicate specifies a value in `employees.last_name`, then the database might read enough entries in the index on `last_name` to determine whether an internal threshold is passed. If the threshold is not passed, then the optimizer picks a nested loop join to `departments`, and if the threshold is passed, then the database performs a hash join, which means reading the rest of `employees`, hashing it into memory, and then joining to `departments`.

If the access path for the inner loop is not dependent on the outer loop, then the result can be a Cartesian product: for every iteration of the outer loop, the inner loop produces the same set of rows. To avoid this problem, use other join methods to join two independent row sources.

See Also:

- "Table 19-2"
- "Adaptive Query Plans"

9.2.1.2 How Nested Loops Joins Work

Conceptually, nested loops are equivalent to two nested `for` loops.

For example, if a query joins `employees` and `departments`, then a nested loop in pseudocode might be:

```
FOR erow IN (select * from employees where X=Y) LOOP
  FOR drow IN (select * from departments where erow is matched) LOOP
    output values from erow and drow
  END LOOP
END LOOP
```

The inner loop is executed for every row of the outer loop. The `employees` table is the "outer" data set because it is in the exterior `for` loop. The outer table is sometimes called a driving table. The `departments` table is the "inner" data set because it is in the interior `for` loop.

A nested loops join involves the following basic steps:

1. The optimizer determines the driving row source and designates it as the outer loop.

   The outer loop produces a set of rows for driving the join condition. The row source can be a table accessed using an index scan, a full table scan, or any other operation that generates rows.

   The number of iterations of the inner loop depends on the number of rows retrieved in the outer loop. For example, if 10 rows are retrieved from the outer table, then the database must perform 10 lookups in the inner table. If 10,000,000 rows are retrieved from the outer table, then the database must perform 10,000,000 lookups in the inner table.
2. The optimizer designates the other row source as the inner loop.

   The outer loop appears before the inner loop in the execution plan, as follows:

   ```
   NESTED LOOPS
     outer_loop
     inner_loop
   ```
3. For every fetch request from the client, the basic process is as follows:
   a. Fetch a row from the outer row source
   b. Probe the inner row source to find rows that match the predicate criteria
   c. Repeat the preceding steps until all rows are obtained by the fetch request

   Sometimes the database sorts rowids to obtain a more efficient buffer access pattern.

9.2.1.3 Nested Nested Loops

The outer loop of a nested loop can itself be a row source generated by a different nested loop.

The database can nest two or more outer loops to join as many tables as needed. Each loop is a data access method. The following template shows how the database iterates through three nested loops:

```
SELECT STATEMENT
  NESTED LOOPS 3
    NESTED LOOPS 2          - Row source becomes OUTER LOOP 3.1
      NESTED LOOPS 1        - Row source becomes OUTER LOOP 2.1
        OUTER LOOP 1.1
        INNER LOOP 1.2
      INNER LOOP 2.2
    INNER LOOP 3.2
```

The database orders the loops as follows:

1. The database iterates through `NESTED LOOPS 1`:

   ```
   NESTED LOOPS 1
     OUTER LOOP 1.1
     INNER LOOP 1.2
   ```

   The output of `NESTED LOOP 1` is a row source.

2. The database iterates through `NESTED LOOPS 2`, using the row source generated by `NESTED LOOPS 1` as its outer loop:

   ```
   NESTED LOOPS 2
     OUTER LOOP 2.1         - Row source generated by NESTED LOOPS 1
     INNER LOOP 2.2
   ```

   The output of `NESTED LOOPS 2` is another row source.

3. The database iterates through `NESTED LOOPS 3`, using the row source generated by `NESTED LOOPS 2` as its outer loop:

   ```
   NESTED LOOPS 3
     OUTER LOOP 3.1         - Row source generated by NESTED LOOPS 2
     INNER LOOP 3.2
   ```

Example 9-2 Nested Nested Loops Join

Suppose you join the `employees` and `departments` tables as follows:

```
SELECT /*+ ORDERED USE_NL(d) */ e.last_name, e.first_name,
d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id
AND    e.last_name like 'A%';
```

The plan reveals that the optimizer chose two nested loops (Step 1 and Step 2) to access the data:

```
SQL_ID  ahuavfcv4tnz4, child number 0
-------------------------------------
SELECT /*+ ORDERED USE_NL(d) */ e.last_name, d.department_name FROM
employees e, departments d WHERE  e.department_id=d.department_id AND
 e.last_name like 'A%'

Plan hash value: 1667998133

----------------------------------------------------------------------------------
|Id| Operation                            |Name        |Rows|Bytes|Cost(%CPU)|Time|
----------------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |             |   |   |5 (100)|        |
| 1|  NESTED LOOPS                        |             |   |   |       |        |
| 2|   NESTED LOOPS                       |             | 3|102|5   (0)|00:00:01|
| 3|    TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   | 3| 54|2   (0)|00:00:01|
|*4|     INDEX RANGE SCAN                 | EMP_NAME_IX | 3|   |1   (0)|00:00:01|
|*5|    INDEX UNIQUE SCAN                 | DEPT_ID_PK  | 1|   |0   (0)|        |
| 6|   TABLE ACCESS BY INDEX ROWID        | DEPARTMENTS | 1| 16|1   (0)|00:00:01|
----------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."LAST_NAME" LIKE 'A%')
       filter("E"."LAST_NAME" LIKE 'A%')
   5 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

In this example, the basic process is as follows:

1. The database begins iterating through the inner nested loop (Step 2) as follows:
   a. The database searches the `emp_name_ix` for the rowids for all last names that begins with `A` (Step 4).

      For example:

      ```
      Abel,employees_rowid
      Ande,employees_rowid
      Atkinson,employees_rowid
      Austin,employees_rowid
      ```

   b. Using the rowids from the previous step, the database retrieves a batch of rows from the `employees` table (Step 3). For example:

      ```
      Abel,Ellen,80
      Abel,John,50
      ```

      These rows become the outer row source for the innermost nested loop.

      The batch step is typically part of adaptive execution plans. To determine whether a nested loop is better than a hash join, the optimizer needs to determine many rows come back from the row source. If too many rows are returned, then the optimizer switches to a different join method.

   c. For each row in the outer row source, the database scans the `dept_id_pk` index to obtain the rowid in `departments` of the matching department ID (Step 5), and joins it to the `employees` rows. For example:

      ```
      Abel,Ellen,80,departments_rowid
      Ande,Sundar,80,departments_rowid
      Atkinson,Mozhe,50,departments_rowid
      Austin,David,60,departments_rowid
      ```

      These rows become the outer row source for the outer nested loop (Step 1).

2. The database iterates through the outer nested loop as follows:

   a. The database reads the first row in outer row source.

      For example:

      ```
      Abel,Ellen,80,departments_rowid
      ```

   b. The database uses the `departments` rowid to retrieve the corresponding row from `departments` (Step 6), and then joins the result to obtain the requested values (Step 1).

      For example:

      ```
      Abel,Ellen,80,Sales
      ```

   c. The database reads the next row in the outer row source, uses the `departments` rowid to retrieve the corresponding row from `departments` (Step 6), and iterates through the loop until all rows are retrieved.

      The result set has the following form:

      ```
      Abel,Ellen,80,Sales
      Ande,Sundar,80,Sales
      Atkinson,Mozhe,50,Shipping
      Austin,David,60,IT
      ```

9.2.1.4 Current Implementation for Nested Loops Joins

Oracle Database 11g introduced a new implementation for nested loops that reduces overall latency for physical I/O.

When an index or a table block is not in the buffer cache and is needed to process the join, a physical I/O is required. The database can batch multiple physical I/O requests and process them using a vector I/O (array) instead of one at a time. The database sends an array of rowids to the operating system, which performs the read.

As part of the new implementation, two `NESTED LOOPS` join row sources might appear in the execution plan where only one would have appeared in prior releases. In such cases, Oracle Database allocates one `NESTED LOOPS` join row source to join the values from the table on the outer side of the join with the index on the inner side. A second row source is allocated to join the result of the first join, which includes the rowids stored in the index, with the table on the inner side of the join.

Consider the query in "Original Implementation for Nested Loops Joins". In the current implementation, the execution plan for this query might be as follows:

```
-------------------------------------------------------------------------------------
| Id | Operation                     | Name              |Rows|Bytes|Cost%CPU| Time   |
-------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT              |                   | 19 | 722 |  3 (0)|00:00:01|
|  1 |  NESTED LOOPS                 |                   |    |     |        |        |
|  2 |   NESTED LOOPS                |                   | 19 | 722 |  3 (0)|00:00:01|
|* 3 |    TABLE ACCESS FULL          | DEPARTMENTS       |  2 |  32 |  2 (0)|00:00:01|
|* 4 |    INDEX RANGE SCAN           | EMP_DEPARTMENT_IX | 10 |     |  0 (0)|00:00:01|
|  5 |   TABLE ACCESS BY INDEX ROWID| EMPLOYEES         | 10 | 220 |  1 (0)|00:00:01|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("D"."DEPARTMENT_NAME"='Marketing' OR "D"."DEPARTMENT_NAME"='Sales')
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

In this case, rows from the `hr.departments` table form the outer row source (Step 3) of the inner nested loop (Step 2). The index `emp_department_ix` is the inner row source (Step 4) of the inner nested loop. The results of the inner nested loop form the outer row source (Row 2) of the outer nested loop (Row 1). The `hr.employees` table is the outer row source (Row 5) of the outer nested loop.

For each fetch request, the basic process is as follows:

1. The database iterates through the inner nested loop (Step 2) to obtain the rows requested in the fetch:

   a. The database reads the first row of `departments` to obtain the department IDs for departments named `Marketing` or `Sales` (Step 3). For example:

      ```
      Marketing,20
      ```

      This row set is the outer loop. The database caches the data in the PGA.

   b. The database scans `emp_department_ix`, which is an index on the `employees` table, to find `employees` rowids that correspond to this department ID (Step 4), and then joins the result (Step 2).

      The result set has the following form:

      ```
      Marketing,20,employees_rowid
      Marketing,20,employees_rowid
      Marketing,20,employees_rowid
      ```

   c. The database reads the next row of `departments`, scans `emp_department_ix` to find `employees` rowids that correspond to this department ID, and then iterates through the loop until the client request is satisfied.

In this example, the database only iterates through the outer loop twice because only two rows from `departments` satisfy the predicate filter. Conceptually, the result set has the following form:

```
Marketing,20,employees_rowid
Marketing,20,employees_rowid
Marketing,20,employees_rowid
.
.
.
Sales,80,employees_rowid
Sales,80,employees_rowid
Sales,80,employees_rowid
.
.
.
```

These rows become the outer row source for the outer nested loop (Step 1). This row set is cached in the PGA.

2. The database organizes the rowids obtained in the previous step so that it can more efficiently access them in the cache.
3. The database begins iterating through the outer nested loop as follows:
   a. The database retrieves the first row from the row set obtained in the previous step, as in the following example:

      ```
      Marketing,20,employees_rowid
      ```

   b. Using the rowid, the database retrieves a row from `employees` to obtain the requested values (Step 1), as in the following example:

      ```
      Michael,Hartstein,13000,Marketing
      ```

   c. The database retrieves the next row from the row set, uses the rowid to probe `employees` for the matching row, and iterates through the loop until all rows are retrieved.

      The result set has the following form:

      ```
      Michael,Hartstein,13000,Marketing
      Pat,Fay,6000,Marketing
      John,Russell,14000,Sales
      Karen,Partners,13500,Sales
      Alberto,Errazuriz,12000,Sales
      .
      .
      .
      ```

In some cases, a second join row source is not allocated, and the execution plan looks the same as it did before Oracle Database 11g. The following list describes such cases:

- All of the columns needed from the inner side of the join are present in the index, and there is no table access required. In this case, Oracle Database allocates only one join row source.
- The order of the rows returned might be different from the order returned in releases earlier than Oracle Database 12c. Thus, when Oracle Database tries to preserve a specific ordering of the rows, for example to eliminate the need for an `ORDER BY` sort, Oracle Database might use the original implementation for nested loops joins.
- The `OPTIMIZER_FEATURES_ENABLE` initialization parameter is set to a release before Oracle Database 11g. In this case, Oracle Database uses the original implementation for nested loops joins.

9.2.1.5 Original Implementation for Nested Loops Joins

In the current release, both the new and original implementation of nested loops are possible.

For an example of the original implementation, consider the following join of the `hr.employees` and `hr.departments` tables:

```
SELECT e.first_name, e.last_name, e.salary, d.department_name
FROM   hr.employees e, hr.departments d
WHERE  d.department_name IN ('Marketing', 'Sales')
AND    e.department_id = d.department_id;
```

In releases before Oracle Database 11*g*, the execution plan for this query might appear as follows:

```
-------------------------------------------------------------------------------------------
| Id  | Operation                     | Name               | Rows | Bytes |Cost (%CPU)|Time  |
-------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |                    |   19 |   722 | 3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES          |   10 |   220 | 1   (0)| 00:00:01 |
|   2 |   NESTED LOOPS                |                    |   19 |   722 | 3   (0)| 00:00:01 |
|*  3 |    TABLE ACCESS FULL          | DEPARTMENTS        |    2 |    32 | 2   (0)| 00:00:01 |
|*  4 |    INDEX RANGE SCAN           | EMP_DEPARTMENT_IX |   10 |       | 0   (0)| 00:00:01 |
-------------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - filter("D"."DEPARTMENT_NAME"='Marketing' OR "D"."DEPARTMENT_NAME"='Sales')
   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

For each fetch request, the basic process is as follows:

1. The database iterates through the loop to obtain the rows requested in the fetch:
   a. The database reads the first row of `departments` to obtain the department IDs for departments named `Marketing` or `Sales` (Step 3). For example:

      ```
      Marketing,20
      ```

      This row set is the outer loop. The database caches the row in the PGA.

b. The database scans emp_department_ix, which is an index on the employees.department_id column, to find employees rowids that correspond to this department ID (Step 4), and then joins the result (Step 2).

   Conceptually, the result set has the following form:

```
Marketing,20,employees_rowid
Marketing,20,employees_rowid
Marketing,20,employees_rowid
```

c. The database reads the next row of departments, scans emp_department_ix to find employees rowids that correspond to this department ID, and iterates through the loop until the client request is satisfied.

   In this example, the database only iterates through the outer loop twice because only two rows from departments satisfy the predicate filter. Conceptually, the result set has the following form:

```
Marketing,20,employees_rowid
Marketing,20,employees_rowid
Marketing,20,employees_rowid
.
.
.
Sales,80,employees_rowid
Sales,80,employees_rowid
Sales,80,employees_rowid
.
.
.
```

2. Depending on the circumstances, the database may organize the cached rowids obtained in the previous step so that it can more efficiently access them.
3. For each employees rowid in the result set generated by the nested loop, the database retrieves a row from employees to obtain the requested values (Step 1).

   Thus, the basic process is to read a rowid and retrieve the matching employees row, read the next rowid and retrieve the matching employees row, and so on. Conceptually, the result set has the following form:

```
Michael,Hartstein,13000,Marketing
Pat,Fay,6000,Marketing
John,Russell,14000,Sales
Karen,Partners,13500,Sales
Alberto,Errazuriz,12000,Sales
.
.
.
```

9.2.1.6 Nested Loops Controls

You can add the USE_NL hint to instruct the optimizer to join each specified table to another row source with a nested loops join, using the specified table as the inner table.

The related hint `USE_NL_WITH_INDEX(`*`table index`*`)` hint instructs the optimizer to join the specified table to another row source with a nested loops join using the specified table as the inner table. The index is optional. If no index is specified, then the nested loops join uses an index with at least one join predicate as the index key.

Example 9-3 Nested Loops Hint

Assume that the optimizer chooses a hash join for the following query:

```
SELECT e.last_name, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id;
```

The plan looks as follows:

```
---------------------------------------------------------------------------
|Id | Operation           | Name          | Rows| Bytes |Cost(%CPU)| Time     |
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT    |               |     |       |  5 (100)|          |
|*1 |  HASH JOIN          |               | 106 |  2862 |  5  (20)| 00:00:01 |
| 2 |   TABLE ACCESS FULL| DEPARTMENTS |  27 |   432 |  2    (0)| 00:00:01 |
| 3 |   TABLE ACCESS FULL| EMPLOYEES   | 107 |  1177 |  2    (0)| 00:00:01 |
---------------------------------------------------------------------------
```

To force a nested loops join using `departments` as the inner table, add the `USE_NL` hint as in the following query:

```
SELECT /*+ ORDERED USE_NL(d) */ e.last_name, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id;
```

The plan looks as follows:

```
---------------------------------------------------------------------------
| Id | Operation           | Name          | Rows |Bytes |Cost (%CPU)|Time     |
---------------------------------------------------------------------------
|  0 | SELECT STATEMENT    |               |      |      | 34 (100)|          |
|  1 |  NESTED LOOPS       |               | 106  | 2862 | 34    (3)| 00:00:01 |
|  2 |   TABLE ACCESS FULL| EMPLOYEES     | 107  | 1177 |  2    (0)| 00:00:01 |
|* 3 |   TABLE ACCESS FULL| DEPARTMENTS   |    1 |   16 |  0    (0)|          |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

The database obtains the result set as follows:

1. In the nested loop, the database reads `employees` to obtain the last name and department ID for an employee (Step 2). For example:

   ```
   De Haan,90
   ```

2. For the row obtained in the previous step, the database scans `departments` to find the department name that matches the `employees` department ID (Step 3), and joins the result (Step 1). For example:

   ```
   De Haan,Executive
   ```

3. The database retrieves the next row in `employees`, retrieves the matching row from `departments`, and then repeats this process until all rows are retrieved.

   The result set has the following form:

   ```
   De Haan,Executive
   Kochnar,Executive
   Baer,Public Relations
   King,Executive
   .
   .
   .
   ```

See Also:

- "Guidelines for Join Order Hints" to learn more about the `USE_NL` hint
- *Oracle Database SQL Language Reference* to learn about the `USE_NL` hint

9.2.2 Hash Joins

The database uses a **hash join** to join larger data sets.

The optimizer uses the smaller of two data sets to build a hash table on the join key in memory, using a deterministic hash function to specify the location in the hash table in which to store each row. The database then scans the larger data set, probing the hash table to find the rows that meet the join condition.

9.2.2.1 When the Optimizer Considers Hash Joins

In general, the optimizer considers a hash join when a relatively large amount of data must be joined (or a large percentage of a small table must be joined), and the join is an equijoin.

A hash join is most cost effective when the smaller data set fits in memory. In this case, the cost is limited to a single read pass over the two data sets.

Because the hash table is in the PGA, Oracle Database can access rows without latching them. This technique reduces logical I/O by avoiding the necessity of repeatedly latching and reading blocks in the database buffer cache.

If the data sets do not fit in memory, then the database partitions the row sources, and the join proceeds partition by partition. This can use a lot of sort area memory, and I/O to the temporary tablespace. This method can still be the most cost effective, especially when the database uses parallel query servers.

9.2.2.2 How Hash Joins Work

A hashing algorithm takes a set of inputs and applies a deterministic hash function to generate a random hash value.

In a hash join, the input values are the join keys. The output values are indexes (slots) in an array, which is the hash table.

9.2.2.2.1 Hash Tables

To illustrate a hash table, assume that the database hashes `hr.departments` in a join of `departments` and `employees`. The join key column is `department_id`.

The first 5 rows of `departments` are as follows:

```
SQL> select * from departments where rownum < 6;

DEPARTMENT_ID DEPARTMENT_NAME                MANAGER_ID LOCATION_ID
------------- ------------------------------ ---------- -----------
           10 Administration                        200        1700
           20 Marketing                             201        1800
           30 Purchasing                            114        1700
           40 Human Resources                       203        2400
           50 Shipping                              121        1500
```

The database applies the hash function to each `department_id` in the table, generating a hash value for each. For this illustration, the hash table has 5 slots (it could have more or less). Because *n* is `5`, the possible hash values range from `1` to `5`. The hash functions might generate the following values for the department IDs:

```
f(10) = 4
f(20) = 1
f(30) = 4
f(40) = 2
f(50) = 5
```

Note that the hash function happens to generate the same hash value of `4` for departments `10` and `30`. This is known as a hash collision. In this case, the database puts the records for departments `10` and `30` in the same slot, using a linked list. Conceptually, the hash table looks as follows:

```
1    20,Marketing,201,1800
2    40,Human Resources,203,2400
3
4    10,Administration,200,1700 -> 30,Purchasing,114,1700
5    50,Shipping,121,1500
```

9.2.2.2.2 Hash Join: Basic Steps

The optimizer uses the smaller data source to build a hash table on the join key in memory, and then scans the larger table to find the joined rows.

The basic steps are as follows:

1. The database performs a full scan of the smaller data set, called the **build table**, and then applies a hash function to the join key in each row to build a hash table in the PGA.

   In pseudocode, the algorithm might look as follows:

   ```
   FOR small_table_row IN (SELECT * FROM small_table)
   LOOP
     slot_number := HASH(small_table_row.join_key);
     INSERT_HASH_TABLE(slot_number,small_table_row);
   END LOOP;
   ```

2. The database probes the second data set, called the **probe table**, using whichever access mechanism has the lowest cost.

   Typically, the database performs a full scan of both the smaller and larger data set. The algorithm in pseudocode might look as follows:

   ```
   FOR large_table_row IN (SELECT * FROM large_table)
   LOOP
      slot_number := HASH(large_table_row.join_key);
      small_table_row =
   LOOKUP_HASH_TABLE(slot_number,large_table_row.join_key);
      IF small_table_row FOUND
      THEN
         output small_table_row + large_table_row;
      END IF;
   END LOOP;
   ```

   For each row retrieved from the larger data set, the database does the following:

   a. Applies the same hash function to the join column or columns to calculate the number of the relevant slot in the hash table.

      For example, to probe the hash table for department ID `30`, the database applies the hash function to `30`, which generates the hash value `4`.

   b. Probes the hash table to determine whether rows exists in the slot.

      If no rows exist, then the database processes the next row in the larger data set. If rows exist, then the database proceeds to the next step.

   c. Checks the join column or columns for a match. If a match occurs, then the database either reports the rows or passes them to the next step in the plan, and then processes the next row in the larger data set.

      If multiple rows exist in the hash table slot, the database walks through the linked list of rows, checking each one. For example, if department `30` hashes to slot `4`, then the database checks each row until it finds `30`.

Example 9-4   Hash Joins

An application queries the `oe.orders` and `oe.order_items` tables, joining on the `order_id` column.

```
SELECT o.customer_id, l.unit_price * l.quantity
FROM   orders o, order_items l
WHERE  l.order_id = o.order_id;
```

The execution plan is as follows:

```
--------------------------------------------------------------------------
| Id  | Operation            |  Name        | Rows  | Bytes | Cost (%CPU)|
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |              |   665 | 13300 |     8  (25)|
|*  1 |  HASH JOIN           |              |   665 | 13300 |     8  (25)|
|   2 |   TABLE ACCESS FULL  | ORDERS       |   105 |   840 |     4  (25)|
|   3 |   TABLE ACCESS FULL  | ORDER_ITEMS  |   665 |  7980 |     4  (25)|
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - access("L"."ORDER_ID"="O"."ORDER_ID")
```

Because the `orders` table is small relative to the `order_items` table, which is 6 times larger, the database hashes `orders`. In a hash join, the data set for the build table always appears first in the list of operations (Step 2). In Step 3, the database performs a full scan of the larger `order_items` later, probing the hash table for each row.

9.2.2.3 How Hash Joins Work When the Hash Table Does Not Fit in the PGA

The database must use a different technique when the hash table does not fit entirely in the PGA. In this case, the database uses a temporary space to hold portions (called partitions) of the hash table, and sometimes portions of the larger table that probes the hash table.

The basic process is as follows:

1. The database performs a full scan of the smaller data set, and then builds an array of hash buckets in both the PGA and on disk.

   When the PGA hash area fills up, the database finds the largest partition within the hash table and writes it to temporary space on disk. The database stores any new row that belongs to this on-disk partition on disk, and all other rows in the PGA. Thus, part of the hash table is in memory and part of it on disk.

2. The database takes a first pass at reading the other data set.

   For each row, the database does the following:

   a. Applies the same hash function to the join column or columns to calculate the number of the relevant hash bucket.

   b. Probes the hash table to determine whether rows exist in the bucket *in memory*.

      If the hashed value points to a row in memory, then the database completes the join and returns the row. If the value points to a hash partition on disk, however, then the database stores this row in the temporary tablespace, using the same partitioning scheme used for the original data set.

3. The database reads each on-disk temporary partition one by one
4. The database joins each partition row to the row in the corresponding on-disk temporary partition.

9.2.2.4 Hash Join Controls

The `USE_HASH` hint instructs the optimizer to use a hash join when joining two tables together.

See Also:

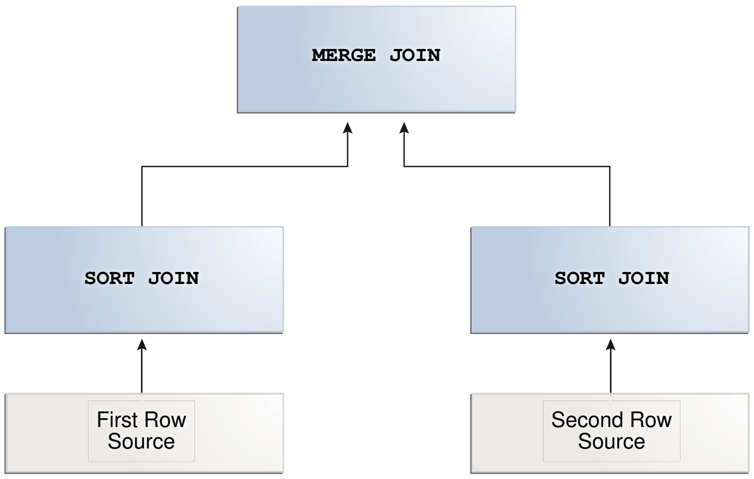
- "Guidelines for Join Order Hints"
- *Oracle Database SQL Language Reference* to learn about `USE_HASH`

9.2.3 Sort Merge Joins

A sort merge join is a variation on a nested loops join.

If the two data sets in the join are not already sorted, then the database sorts them. These are the `SORT JOIN` operations. For each row in the first data set, the database probes the second data set for matching rows and joins them, basing its start position on the match made in the previous iteration. This is the `MERGE JOIN` operation.

Figure 9-6 Sort Merge Join



9.2.3.1 When the Optimizer Considers Sort Merge Joins

A hash join requires one hash table and one probe of this table, whereas a sort merge join requires two sorts.

The optimizer may choose a sort merge join over a hash join for joining large amounts of data when any of the following conditions is true:

- The join condition between two tables is not an equijoin, that is, uses an inequality condition such as <, <=, >, or >=.

  In contrast to sort merges, hash joins require an equality condition.
- Because of sorts required by other operations, the optimizer finds it cheaper to use a sort merge.

  If an index exists, then the database can avoid sorting the first data set. However, the database always sorts the second data set, regardless of indexes.

A sort merge has the same advantage over a nested loops join as the hash join: the database accesses rows in the PGA rather than the SGA, reducing logical I/O by avoiding the necessity of repeatedly latching and reading blocks in the database buffer cache. In general, hash joins perform better than sort merge joins because sorting is expensive. However, sort merge joins offer the following advantages over a hash join:

- After the initial sort, the merge phase is optimized, resulting in faster generation of output rows.
- A sort merge can be more cost-effective than a hash join when the hash table does not fit completely in memory.

  A hash join with insufficient memory requires both the hash table and the other data set to be copied to disk. In this case, the database may have to read from disk multiple times. In a sort merge, if memory cannot hold the two data sets, then the database writes them both to disk, but reads each data set no more than once.

9.2.3.2 How Sort Merge Joins Work

As in a nested loops join, a sort merge join reads two data sets, but sorts them when they are not already sorted.

For each row in the first data set, the database finds a starting row in the second data set, and then reads the second data set until it finds a nonmatching row. In pseudocode, the high-level algorithm for sort merge might look as follows:

```
READ data_set_1 SORT BY JOIN KEY TO temp_ds1
READ data_set_2 SORT BY JOIN KEY TO temp_ds2

READ ds1_row FROM temp_ds1
READ ds2_row FROM temp_ds2

WHILE NOT eof ON temp_ds1,temp_ds2
LOOP
    IF ( temp_ds1.key = temp_ds2.key ) OUTPUT JOIN ds1_row,ds2_row
    ELSIF ( temp_ds1.key <= temp_ds2.key ) READ ds1_row FROM temp_ds1
    ELSIF ( temp_ds1.key => temp_ds2.key ) READ ds2_row FROM temp_ds2
END LOOP
```

For example, the following table shows sorted values in two data sets: `temp_ds1` and `temp_ds2`.

Table 9-3 Sorted Data Sets

temp_ds1	temp_ds2
10	20
20	20
30	40
40	40
50	40
60	40
70	40
.	60
.	70
.	70

As shown in the following table, the database begins by reading `10` in `temp_ds1`, and then reads the first value in `temp_ds2`. Because `20` in `temp_ds2` is higher than `10` in `temp_ds1`, the database stops reading `temp_ds2`.

Table 9-4 Start at 10 in temp_ds1

temp_ds1	temp_ds2	Action
10 [start here]	20 [start here] [stop here]	20 in temp_ds2 is higher than 10 in temp_ds1. Stop. Start again with next row in temp_ds1.
20	20	N/A
30	40	N/A
40	40	N/A
50	40	N/A
60	40	N/A
70	40	N/A
.	60	N/A
.	70	N/A
.	70	N/A

The database proceeds to the next value in `temp_ds1`, which is `20`. The database proceeds through `temp_ds2` as shown in the following table.

Table 9-5 Start at 20 in temp_ds1

temp_ds1	temp_ds2	Action
10	20 [start here]	Match. Proceed to next value in temp_ds2.
20 [start here]	20	Match. Proceed to next value in temp_ds2.
30	40 [stop here]	40 in temp_ds2 is higher than 20 in temp_ds1. Stop. Start again with next row in temp_ds1.
40	40	N/A
50	40	N/A
60	40	N/A
70	40	N/A

Table 9-5 (Cont.) Start at 20 in temp_ds1

temp_ds1	temp_ds2	Action
.	60	N/A
.	70	N/A
.	70	N/A

The database proceeds to the next row in `temp_ds1`, which is `30`. The database starts at the number of its last match, which was `20`, and then proceeds through `temp_ds2` looking for a match, as shown in the following table.

Table 9-6 Start at 30 in temp_ds1

temp_ds1	temp_ds2	Action
10	20	N/A
20	20 [start at last match]	20 in temp_ds2 is lower than 30 in temp_ds1. Proceed to next value in temp_ds2.
30 [start here]	40 [stop here]	40 in temp_ds2 is higher than 30 in temp_ds1. Stop. Start again with next row in temp_ds1.
40	40	N/A
50	40	N/A
60	40	N/A
70	40	N/A
.	60	N/A
.	70	N/A
.	70	N/A

The database proceeds to the next row in `temp_ds1`, which is `40`. As shown in the following table, the database starts at the number of its last match in `temp_ds2`, which was `20`, and then proceeds through `temp_ds2` looking for a match.

Table 9-7 Start at 40 in temp_ds1

temp_ds1	temp_ds2	Action
10	20	N/A
20	20 [start at last match]	20 in temp_ds2 is lower than 40 in temp_ds1. Proceed to next value in temp_ds2.
30	40	Match. Proceed to next value in temp_ds2.
40 [start here]	40	Match. Proceed to next value in temp_ds2.
50	40	Match. Proceed to next value in temp_ds2.
60	40	Match. Proceed to next value in temp_ds2.
70	40	Match. Proceed to next value in temp_ds2.
.	60 [stop here]	60 in temp_ds2 is higher than 40 in temp_ds1. Stop. Start again with next row in temp_ds1.
.	70	N/A
.	70	N/A

The database continues in this way until it has matched the final 70 in temp_ds2. This scenario demonstrates that the database, as it reads through temp_ds1, does not need to read every row in temp_ds2. This is an advantage over a nested loops join.

Example 9-5   Sort Merge Join Using Index

The following query joins the employees and departments tables on the department_id column, ordering the rows on department_id as follows:

```
SELECT e.employee_id, e.last_name, e.first_name, e.department_id,
       d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id
ORDER BY department_id;
```

A query of DBMS_XPLAN.DISPLAY_CURSOR shows that the plan uses a sort merge join:

```
-----------------------------------------------------------------------
----
|Id| Operation                    | Name       |Rows|Bytes|Cost (%CPU)|
Time|
-----------------------------------------------------------------------
----
| 0| SELECT STATEMENT             |            |    |     |
5(100)|        |
| 1|  MERGE JOIN                  |            |106 |4028 |5 (20)|
00:00:01|
| 2|   TABLE ACCESS BY INDEX ROWID|DEPARTMENTS | 27 | 432 |2  (0)|
00:00:01|
| 3|    INDEX FULL SCAN           |DEPT_ID_PK  | 27 |     |1  (0)|
00:00:01|
|*4|   SORT JOIN                  |            |107 |2354 |3 (34)|
00:00:01|
| 5|    TABLE ACCESS FULL         |EMPLOYEES   |107 |2354 |2  (0)|
00:00:01|
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

The two data sets are the departments table and the employees table. Because an index orders the departments table by department_id, the database can read this index and avoid a sort (Step 3). The database only needs to sort the employees table (Step 4), which is the most CPU-intensive operation.

Example 9-6   Sort Merge Join Without an Index

You join the `employees` and `departments` tables on the `department_id` column, ordering the rows on `department_id` as follows. In this example, you specify `NO_INDEX` and `USE_MERGE` to force the optimizer to choose a sort merge:

```
SELECT /*+ USE_MERGE(d e) NO_INDEX(d) */ e.employee_id, e.last_name,
e.first_name,
       e.department_id, d.department_name
FROM   employees e, departments d
WHERE  e.department_id = d.department_id
ORDER BY department_id;
```

A query of `DBMS_XPLAN.DISPLAY_CURSOR` shows that the plan uses a sort merge join:

```
----------------------------------------------------------------------------
| Id| Operation             | Name          | Rows| Bytes|Cost (%CPU)|Time     |
----------------------------------------------------------------------------
| 0 | SELECT STATEMENT      |               |     |      |   6 (100)|          |
| 1 |  MERGE JOIN           |               | 106 | 9540 |   6  (34)| 00:00:01|
| 2 |   SORT JOIN           |               |  27 |  567 |   3  (34)| 00:00:01|
| 3 |    TABLE ACCESS FULL| DEPARTMENTS |  27 |  567 |   2   (0)| 00:00:01|
|*4 |   SORT JOIN           |               | 107 | 7383 |   3  (34)| 00:00:01|
| 5 |    TABLE ACCESS FULL| EMPLOYEES   | 107 | 7383 |   2   (0)| 00:00:01|
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

Because the `departments.department_id` index is ignored, the optimizer performs a sort, which increases the combined cost of Step 2 and Step 3 by 67% (from 3 to 5).

9.2.3.3 Sort Merge Join Controls

The `USE_MERGE` hint instructs the optimizer to use a sort merge join.

In some situations it may make sense to override the optimizer with the `USE_MERGE` hint. For example, the optimizer can choose a full scan on a table and avoid a sort operation in a query. However, there is an increased cost because a large table is accessed through an index and single block reads, as opposed to faster access through a full table scan.

See Also:

Oracle Database SQL Language Reference to learn about the `USE_MERGE` hint

9.3 Join Types

A join type is determined by the type of join condition.

9.3.1 Inner Joins

An **inner join** (sometimes called a *simple join*) is a join that returns only rows that satisfy the join condition. Inner joins are either equijoins or nonequijoins.

9.3.1.1 Equijoins

An **equijoin** is an inner join whose join condition contains an equality operator.

The following example is an equijoin because the join condition contains only an equality operator:

```
SELECT e.employee_id, e.last_name, d.department_name
FROM   employees e, departments d
WHERE  e.department_id=d.department_id;
```

In the preceding query, the join condition is `e.department_id=d.department_id`. If a row in the `employees` table has a department ID that matches the value in a row in the `departments` table, then the database returns the joined result; otherwise, the database does not return a result.

9.3.1.2 Nonequijoins

A **nonequijoin** is an inner join whose join condition contains an operator that is not an equality operator.

The following query lists all employees whose hire date occurred when employee 176 (who is listed in `job_history` because they changed jobs in 2007) was working at the company:

```
SELECT e.employee_id, e.first_name, e.last_name, e.hire_date
FROM   employees e, job_history h
WHERE  h.employee_id = 176
AND    e.hire_date BETWEEN h.start_date AND h.end_date;
```

In the preceding example, the condition joining `employees` and `job_history` does not contain an equality operator, so it is a nonequijoin. Nonequijoins are relatively rare.

Note that a hash join requires at least a partial equijoin. The following SQL script contains an equality join condition (`e1.empno = e2.empno`) and a nonequality condition:

```
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT *
FROM   scott.emp e1 JOIN scott.emp e2
ON     ( e1.empno = e2.empno
AND      e1.hiredate BETWEEN e2.hiredate-1 AND e2.hiredate+1 )
```

The optimizer chooses a hash join for the preceding query, as shown in the following plan:

```
Execution Plan
----------------------------------------------------------
Plan hash value: 3638257876

---------------------------------------------------------------------------
| Id  | Operation          | Name | Rows   | Bytes | Cost (%CPU)| Time     |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |      1 |   174 |      5  (20)| 00:00:01 |
|*  1 |  HASH JOIN         |      |      1 |   174 |      5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMP  |     14 |  1218 |      2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMP  |     14 |  1218 |      2   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("E1"."EMPNO"="E2"."EMPNO")
       filter("E1"."HIREDATE">=INTERNAL_FUNCTION("E2"."HIREDATE")-1 AND
              "E1"."HIREDATE"<=INTERNAL_FUNCTION("E2"."HIREDATE")+1)
```

9.3.1.3 Band Joins

A **band join** is a special type of nonequijoin in which key values in one data set must fall within the specified range (“band”) of the second data set. The same table can serve as both the first and second data sets.

Starting in Oracle Database 12c Release 2 (12.2), the database evaluates band joins more efficiently. The optimization avoids the unnecessary scanning of rows that fall outside the defined bands.

The optimizer uses a cost estimate to choose the join method (hash, nested loops, or sort merge) and the parallel data distribution method. In most cases, optimized performance is comparable to an equijoin.

This following examples query employees whose salaries are between $100 less and $100 more than the salary of each employee. Thus, the band has a width of $200. The examples assume that it is permissible to compare the salary of every employee with itself. The following query includes partial sample output:

```
SELECT  e1.last_name ||
        ' has salary between 100 less and 100 more than ' ||
        e2.last_name AS "SALARY COMPARISON"
FROM    employees e1,
        employees e2
WHERE   e1.salary
BETWEEN e2.salary - 100
AND     e2.salary + 100;

SALARY COMPARISON
-------------------------------------------------------------
King has salary between 100 less and 100 more than King
Kochhar has salary between 100 less and 100 more than Kochhar
Kochhar has salary between 100 less and 100 more than De Haan
```

```
De Haan has salary between 100 less and 100 more than Kochhar
De Haan has salary between 100 less and 100 more than De Haan
Russell has salary between 100 less and 100 more than Russell
Partners has salary between 100 less and 100 more than Partners
...
```

Example 9-7   Query Without Band Join Optimization

Without the band join optimization, the database uses the following query plan:

```
------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------
------------------------------------------
| Id  | Operation              | Name      |
------------------------------------------
|   0 | SELECT STATEMENT       |           |
|   1 |  MERGE JOIN            |           |
|   2 |   SORT JOIN            |           |
|   3 |    TABLE ACCESS FULL   | EMPLOYEES |
|*  4 |   FILTER               |           |
|*  5 |    SORT JOIN           |           |
|   6 |     TABLE ACCESS FULL| EMPLOYEES |
------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - filter("E1"."SAL"<="E2"."SAL"+100)
   5 - access(INTERNAL_FUNCTION("E1"."SAL")>="E2"."SAL"-100)
       filter(INTERNAL_FUNCTION("E1"."SAL")>="E2"."SAL"-100)
```

In this plan, Step 2 sorts the `e1` row source, and Step 5 sorts the `e2` row source. The sorted row sources are illustrated in the following table.

Table 9-8   Sorted row Sources

e1 Sorted (Step 2 of Plan)	e2 Sorted (Step 5 of Plan)
24000 (King)	24000 (King)
17000 (Kochhar)	17000 (Kochhar)
17000 (De Haan)	17000 (De Haan)
14000 (Russell)	14000 (Russell)
13500 (Partners)	13500 (Partners)

The join begins by iterating through the sorted input (`e1`), which is the left branch of the join, corresponding to Step 2 of the plan. The original query contains two predicates:

- `e1.sal >= e2.sal-100`, which is the Step 5 filter
- `e1.sal >= e2.sal+100`, which is the Step 4 filter

For each iteration of the sorted row source `e1`, the database iterates through row source `e2`, checking every row against Step 5 filter `e1.sal >= e2.sal-100`. If the row passes the Step 5 filter, then the database sends it to the Step 4 filter, and then proceeds to test the next row in e2 against the Step 5 filter. However, if a row fails the Step 5 filter, then the scan of e2 stops, and the database proceeds through the next iteration of e1.

The following table shows the first iteration of e1, which begins with `24000 (King)` in data set e1. The database determines that the first row in e2, which is `24000 (King)`, passes the Step 5 filter. The database then sends the row to the Step 4 filter, `e1.sal <= w2.sal+100`, which also passes. The database sends this row to the MERGE row source. Next, the database checks `17000 (Kochhar)` against the Step 5 filter, which also passes. However, the row fails the Step 4 filter, and is discarded. The database proceeds to test `17000 (De Haan)` against the Step 5 filter.

Table 9-9   First Iteration of e1: Separate SORT JOIN and FILTER

Scan e2	Step 5 Filter (e1.sal >= e2.sal–100)	Step 4 Filter (e1.sal <= e2.sal+100)
24000 (King)	Pass because 24000 >= 23900. Send to Step 4 filter.	Pass because 24000 <= 24100. Return row for merging.
17000 (Kochhar)	Pass because 24000 >= 16900. Send to Step 4 filter.	Fail because 24000 <=17100 is false. Discard row. Scan next row in e2.
17000 (De Haan)	Pass because 24000 >= 16900. Send to Step 4 filter.	Fail because 24000 <=17100 is false. Discard row. Scan next row in e2.
14000 (Russell)	Pass because 24000 >= 13900. Send to Step 4 filter.	Fail because 24000 <=14100 is false. Discard row. Scan next row in e2.
13500 (Partners)	Pass because 24000 >= 13400. Send to Step 4 filter.	Fail because 24000 <=13600 is false. Discard row. Scan next row in e2.

As shown in the preceding table, every e2 row necessarily passes the Step 5 filter because the e2 salaries are sorted in descending order. Thus, the Step 5 filter always sends the row to the Step 4 filter. Because the e2 salaries are sorted in descending order, the Step 4 filter necessarily fails every row starting with `17000 (Kochhar)`. The inefficiency occurs because the database tests every subsequent row in e2 against the Step 5 filter, which necessarily passes, and then against the Step 4 filter, which necessarily fails.

Example 9-8   Query With Band Join Optimization

Starting in Oracle Database 12c Release 2 (12.2), the database optimizes the band join by using the following plan, which does not have a separate FILTER operation:

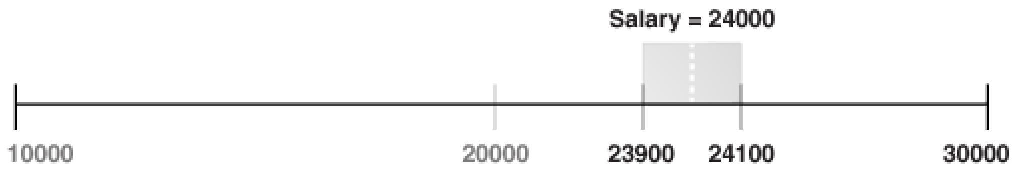
```
------------------------------------------
PLAN_TABLE_OUTPUT
------------------------------------------
| Id  | Operation              | Name      |
------------------------------------------
|   0 | SELECT STATEMENT       |           |
|   1 |  MERGE JOIN            |           |
|   2 |   SORT JOIN            |           |
|   3 |    TABLE ACCESS FULL   | EMPLOYEES |
|*  4 |   SORT JOIN            |           |
|   5 |    TABLE ACCESS FULL   | EMPLOYEES |
------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   4 - access(INTERNAL_FUNCTION("E1"."SALARY")>="E2"."SALARY"-100)
```

```
filter(("E1"."SALARY"<="E2"."SALARY"+100 AND
   INTERNAL_FUNCTION("E1"."SALARY")>="E2"."SALARY"-100))
```

The difference is that Step 4 uses Boolean `AND` logic for the two predicates to create a *single* filter. Instead of checking a row against one filter, and then sending it to a different row source for checking against a second filter, the database performs one check against one filter. If the check fails, then processing stops.

In this example, the query begins the first iteration of `e1`, which begins with `24000 (King)`. The following figure represents the range. `e2` values below 23900 and above 24100 fall outside the range.

Figure 9-7 Band Join



The following table shows that the database tests the first row of `e2`, which is `24000 (King)`, against the Step 4 filter. The row passes the test, so the database sends the row to be merged. The next row in `e2` is `17000 (Kochhar)`. This row falls outside of the range (band) and thus does not satisfy the filter predicate, so the database stops testing `e2` rows in this iteration. The database stops testing because the descending sort of `e2` ensures that all subsequent rows in `e2` fail the filter test. Thus, the database can proceed to the second iteration of `e1`.

Table 9-10 First Iteration of e1: Single SORT JOIN

Scan e2	Filter 4 (e1.sal >= e2.sal – 100) AND (e1.sal <= e2.sal + 100)
24000 (King)	Passes test because it is true that `(24000 >= 23900) AND (24000 <= 24100)`. Send row to `MERGE`. Test next row.
17000 (Kochhar)	Fails test because it is false that `(24000 >= 16900) AND (24000 <= 17100)`. Stop scanning `e2`. Begin next iteration of `e1`.
17000 (De Haan)	n/a
14000 (Russell)	n/a
13500 (Partners)	n/a

In this way, the band join optimization eliminates unnecessary processing. Instead of scanning every row in `e2` as in the unoptimized case, the database scans only the minimum two rows.

9.3.2 Outer Joins

An **outer join** returns all rows that satisfy the join condition and also rows from one table for which no rows from the other table satisfy the condition. Thus, the result set of an outer join is the superset of an inner join.

In ANSI syntax, the `OUTER JOIN` clause specifies an outer join. In the `FROM` clause, the left table appears to the left of the `OUTER JOIN` keywords, and the right table appears to the right of these keywords. The left table is also called the *outer table*, and the right table is also called the *inner table*. For example, in the following statement the `employees` table is the left or outer table:

```
SELECT employee_id, last_name, first_name
FROM   employees LEFT OUTER JOIN departments
ON     (employees.department_id=departments.departments_id);
```

Outer joins require the outer-joined table to be the driving table. In the preceding example, `employees` is the driving table, and `departments` is the driven-to table.

9.3.2.1 Nested Loops Outer Joins

The database uses this operation to loop through an outer join between two tables. The outer join returns the outer (preserved) table rows, even when no corresponding rows are in the inner (optional) table.

In a standard nested loop, the optimizer chooses the order of tables—which is the driving table and which the driven table—based on the cost. However, in a nested loop outer join, the join condition determines the order of tables. The database uses the outer, row-preserved table to drive to the inner table.

The optimizer uses nested loops joins to process an outer join in the following circumstances:

- It is possible to drive from the outer table to the inner table.
- Data volume is low enough to make the nested loop method efficient.

For an example of a nested loop outer join, you can add the `USE_NL` hint to Example 9-9 to instruct the optimizer to use a nested loop. For example:

```
SELECT /*+ USE_NL(c o) */ cust_last_name,
       SUM(NVL2(o.customer_id,0,1)) "Count"
FROM   customers c, orders o
WHERE  c.credit_limit > 1000
AND    c.customer_id = o.customer_id(+)
GROUP BY cust_last_name;
```

9.3.2.2 Hash Join Outer Joins

The optimizer uses hash joins for processing an outer join when either the data volume is large enough to make a hash join efficient, or it is impossible to drive from the outer table to the inner table.

The cost determines the order of tables. The outer table, including preserved rows, may be used to build the hash table, or it may be used to probe the hash table.

Example 9-9   Hash Join Outer Joins

This example shows a typical hash join outer join query, and its execution plan. In this example, all the customers with credit limits greater than 1000 are queried. An outer join is needed so that the query captures customers who have no orders.

- The outer table is `customers`.
- The inner table is `orders`.
- The join preserves the `customers` rows, including those rows without a corresponding row in `orders`.

You could use a `NOT EXISTS` subquery to return the rows. However, because you are querying all the rows in the table, the hash join performs better (unless the `NOT EXISTS` subquery is not nested).

```
SELECT cust_last_name, SUM(NVL2(o.customer_id,0,1)) "Count"
FROM   customers c, orders o
WHERE  c.credit_limit > 1000
AND    c.customer_id = o.customer_id(+)
GROUP BY cust_last_name;

-----------------------------------------------------------------------
----
| Id  | Operation            | Name       |Rows   |Bytes|Cost (%CPU)|
Time      |
-----------------------------------------------------------------------
----
|  0 | SELECT STATEMENT     |            |       |       | 7
(100)|           |
|  1 |  HASH GROUP BY       |            |  168 |  3192 | 7  (29)|
00:00:01 |
|* 2 |   HASH JOIN OUTER    |            |  318 |  6042 | 6  (17)|
00:00:01 |
|* 3 |    TABLE ACCESS FULL| CUSTOMERS |  260 |  3900 | 3   (0)|
00:00:01 |
|* 4 |    TABLE ACCESS FULL| ORDERS     |  105 |   420 | 2   (0)|
00:00:01 |
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("C"."CUSTOMER_ID"="O"."CUSTOMER_ID")

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
---
   3 - filter("C"."CREDIT_LIMIT">1000)
   4 - filter("O"."CUSTOMER_ID">0)
```

The query looks for customers which satisfy various conditions. An outer join returns `NULL` for the inner table columns along with the outer (preserved) table rows when it does not find any corresponding rows in the inner table. This operation finds all the `customers` rows that do not have any `orders` rows.

In this case, the outer join condition is the following:

```
customers.customer_id = orders.customer_id(+)
```

The components of this condition represent the following:

Example 9-10 Outer Join to a Multitable View

In this example, the outer join is to a multitable view. The optimizer cannot drive into the view like in a normal join or push the predicates, so it builds the entire row set of the view.

```
SELECT c.cust_last_name, sum(revenue)
FROM   customers c, v_orders o
WHERE  c.credit_limit > 2000
AND    o.customer_id(+) = c.customer_id
GROUP BY c.cust_last_name;

---------------------------------------------------------------------------
| Id  | Operation                 |  Name         | Rows | Bytes | Cost (%CPU)|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |               |  144 |  4608 |    16  (32)|
|   1 |  HASH GROUP BY            |               |  144 |  4608 |    16  (32)|
|*  2 |   HASH JOIN OUTER         |               |  663 | 21216 |    15  (27)|
|*  3 |    TABLE ACCESS FULL      | CUSTOMERS     |  195 |  2925 |     6  (17)|
|   4 |    VIEW                   | V_ORDERS      |  665 | 11305 |            |
|   5 |     HASH GROUP BY         |               |  665 | 15960 |     9  (34)|
|*  6 |      HASH JOIN            |               |  665 | 15960 |     8  (25)|
|*  7 |       TABLE ACCESS FULL| ORDERS        |  105 |   840 |     4  (25)|
|   8 |       TABLE ACCESS FULL| ORDER_ITEMS   |  665 | 10640 |     4  (25)|
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   2 - access("O"."CUSTOMER_ID"(+)="C"."CUSTOMER_ID")
   3 - filter("C"."CREDIT_LIMIT">2000)
   6 - access("O"."ORDER_ID"="L"."ORDER_ID")
   7 - filter("O"."CUSTOMER_ID">0)
```

The view definition is as follows:

```
CREATE OR REPLACE view v_orders AS
SELECT l.product_id, SUM(l.quantity*unit_price) revenue,
       o.order_id, o.customer_id
FROM   orders o, order_items l
WHERE  o.order_id = l.order_id
GROUP BY l.product_id, o.order_id, o.customer_id;
```

9.3.2.3 Sort Merge Outer Joins

When an outer join cannot drive from the outer (preserved) table to the inner (optional) table, it cannot use a hash join or nested loops joins.

In this case, it uses the sort merge outer join.

The optimizer uses sort merge for an outer join in the following cases:

- A nested loops join is inefficient. A nested loops join can be inefficient because of data volumes.
- The optimizer finds it is cheaper to use a sort merge over a hash join because of sorts required by other operations.

9.3.2.4 Full Outer Joins

A **full outer join** is a combination of the left and right outer joins.

In addition to the inner join, rows from both tables that have not been returned in the result of the inner join are preserved and extended with nulls. In other words, full outer joins join tables together, yet show rows with no corresponding rows in the joined tables.

Example 9-11 Full Outer Join

The following query retrieves all departments and all employees in each department, but also includes:

- Any employees without departments
- Any departments without employees

```
SELECT d.department_id, e.employee_id
FROM   employees e FULL OUTER JOIN departments d
ON     e.department_id = d.department_id
ORDER BY d.department_id;
```

The statement produces the following output:

```
DEPARTMENT_ID EMPLOYEE_ID
------------- -----------
           10         200
           20         201
           20         202
           30         114
           30         115
           30         116
...
          270
          280
                      178
                      207

125 rows selected.
```

Example 9-12 Execution Plan for a Full Outer Join

Starting with Oracle Database 11*g*, Oracle Database automatically uses a native execution method based on a hash join for executing full outer joins whenever possible. When the database uses the new method to execute a full outer join, the execution plan for the query contains `HASH JOIN FULL OUTER`. The query in Example 9-11 uses the following execution plan:

```
----------------------------------------------------------------------------
| Id| Operation                 | Name       |Rows|Bytes |Cost (%CPU)|Time  |
----------------------------------------------------------------------------
| 0 | SELECT STATEMENT          |            |122 | 4758 | 6  (34)|00:0 0:01|
| 1 |  SORT ORDER BY            |            |122 | 4758 | 6  (34)|00:0 0:01|
| 2 |   VIEW                    | VW_FOJ_0   |122 | 4758 | 5  (20)|00:0 0:01|
|*3 |    HASH JOIN FULL OUTER   |            |122 | 1342 | 5  (20)|00:0 0:01|
| 4 |     INDEX FAST FULL SCAN| DEPT_ID_PK | 27 |  108 | 2   (0)|00:0 0:01|
| 5 |     TABLE ACCESS FULL     | EMPLOYEES  |107 |  749 | 2   (0)|00:0 0:01|
----------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   3 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

`HASH JOIN FULL OUTER` is included in the preceding plan (Step 3), indicating that the query uses the hash full outer join execution method. Typically, when the full outer join condition between two tables is an equijoin, the hash full outer join execution method is possible, and Oracle Database uses it automatically.

To instruct the optimizer to consider using the hash full outer join execution method, apply the `NATIVE_FULL_OUTER_JOIN` hint. To instruct the optimizer not to consider using the hash full outer join execution method, apply the `NO_NATIVE_FULL_OUTER_JOIN` hint. The `NO_NATIVE_FULL_OUTER_JOIN` hint instructs the optimizer to exclude the native execution method when joining each specified table. Instead, the full outer join is executed as a union of left outer join and an antijoin.

9.3.2.5 Multiple Tables on the Left of an Outer Join

In Oracle Database 12c, multiple tables may exist on the left side of an outer-joined table.

This enhancement enables Oracle Database to merge a view that contains multiple tables and appears on the left of the outer join. In releases before Oracle Database 12c, a query such as the following was invalid, and would trigger an `ORA-01417` error message:

```
SELECT t1.d, t3.c
FROM   t1, t2, t3
WHERE  t1.z = t2.z
AND    t1.x = t3.x (+)
AND    t2.y = t3.y (+);
```

Starting in Oracle Database 12c, the preceding query is valid.

9.3.3 Semijoins

A **semijoin** is a join between two data sets that returns a row from the first set when a matching row exists in the subquery data set.

The database stops processing the second data set at the first match. Thus, optimization does not duplicate rows from the first data set when multiple rows in the second data set satisfy the subquery criteria.

Note:

Semijoins and antijoins are considered join types even though the SQL constructs that cause them are subqueries. They are internal algorithms that the optimizer uses to flatten subquery constructs so that they can be resolved in a join-like way.

9.3.3.1 When the Optimizer Considers Semijoins

A semijoin avoids returning a huge number of rows when a query only needs to determine whether a match exists.

With large data sets, this optimization can result in significant time savings over a nested loops join that must loop through every record returned by the inner query for every row in the outer query. The optimizer can apply the semijoin optimization to nested loops joins, hash joins, and sort merge joins.

The optimizer may choose a semijoin in the following circumstances:

- The statement uses either an `IN` or `EXISTS` clause.
- The statement contains a subquery in the `IN` or `EXISTS` clause.
- The `IN` or `EXISTS` clause is not contained inside an `OR` branch.

9.3.3.2 How Semijoins Work

The semijoin optimization is implemented differently depending on what type of join is used.

The following pseudocode shows a semijoin for a nested loops join:

```
FOR ds1_row IN ds1 LOOP
  match := false;
  FOR ds2_row IN ds2_subquery LOOP
    IF (ds1_row matches ds2_row) THEN
      match := true;
      EXIT -- stop processing second data set when a match is found
    END IF
  END LOOP
  IF (match = true) THEN
    RETURN ds1_row
  END IF
END LOOP
```

In the preceding pseudocode, `ds1` is the first data set, and `ds2_subquery` is the subquery data set. The code obtains the first row from the first data set, and then loops through the subquery data set looking for a match. The code exits the inner loop as soon as it finds a match, and then begins processing the next row in the first data set.

Example 9-13   Semijoin Using WHERE EXISTS

The following query uses a `WHERE EXISTS` clause to list only the departments that contain employees:

```
SELECT department_id, department_name
FROM   departments
WHERE EXISTS (SELECT 1
              FROM   employees
              WHERE  employees.department_id = departments.department_id)
```

The execution plan reveals a `NESTED LOOPS SEMI` operation in Step 1:

```
---------------------------------------------------------------------------
| Id| Operation           | Name              |Rows|Bytes|Cost (%CPU)|Time |
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT    |                   |    |     | 2 (100)|         |
| 1 |  NESTED LOOPS SEMI  |                   |11  | 209 | 2   (0)|00:00:01 |
| 2 |   TABLE ACCESS FULL| DEPARTMENTS       |27  | 432 | 2   (0)|00:00:01 |
|*3 |   INDEX RANGE SCAN | EMP_DEPARTMENT_IX |44  | 132 | 0   (0)|         |
---------------------------------------------------------------------------
```

For each row in `departments`, which forms the outer loop, the database obtains the department ID, and then probes the `employees.department_id` index for matching entries. Conceptually, the index looks as follows:

```
10,rowid
10,rowid
10,rowid
10,rowid
30,rowid
30,rowid
30,rowid
...
```

If the first entry in the `departments` table is department `30`, then the database performs a range scan of the index until it finds the first `30` entry, at which point it stops reading the index and returns the matching row from `departments`. If the next row in the outer loop is department `20`, then the database scans the index for a `20` entry, and not finding any matches, performs the next iteration of the outer loop. The database proceeds in this way until all matching rows are returned.

Example 9-14   Semijoin Using IN

The following query uses a `IN` clause to list only the departments that contain employees:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id IN
       (SELECT department_id
        FROM   employees);
```

The execution plan reveals a `NESTED LOOPS SEMI` operation in Step 1:

```
-----------------------------------------------------------------------
----
| Id| Operation           | Name              |Rows|Bytes|Cost (%CPU)|
Time |
-----------------------------------------------------------------------
----
| 0 | SELECT STATEMENT    |                   |    |     | 2
(100)|          |
| 1 |  NESTED LOOPS SEMI  |                   |11  | 209 | 2   (0)|
00:00:01 |
| 2 |   TABLE ACCESS FULL| DEPARTMENTS       |27  | 432 | 2   (0)|
00:00:01 |
|*3 |   INDEX RANGE SCAN | EMP_DEPARTMENT_IX |44  | 132 | 0
(0)|          |
-----------------------------------------------------------------------
----
```

The plan is identical to the plan in Example 9-13.

9.3.4 Antijoins

An **antijoin** is a join between two data sets that returns a row from the first set when a matching row does not exist in the subquery data set.

Like a semijoin, an antijoin stops processing the subquery data set when the first match is found. Unlike a semijoin, the antijoin only returns a row when no match is found.

9.3.4.1 When the Optimizer Considers Antijoins

An antijoin avoids unnecessary processing when a query only needs to return a row when a match does not exist.

With large data sets, this optimization can result in significant time savings over a nested loops join. The latter join must loop through every record returned by the inner query for every row in the outer query. The optimizer can apply the antijoin optimization to nested loops joins, hash joins, and sort merge joins.

The optimizer may choose an antijoin in the following circumstances:

- The statement uses either the `NOT IN` or `NOT EXISTS` clause.
- The statement has a subquery in the `NOT IN` or `NOT EXISTS` clause.
- The `NOT IN` or `NOT EXISTS` clause is not contained inside an `OR` branch.
- The statement performs an outer join and applies an `IS NULL` condition to a join column, as in the following example:

  ```
  SET AUTOTRACE TRACEONLY EXPLAIN
  SELECT emp.*
  FROM   emp, dept
  WHERE  emp.deptno = dept.deptno(+)
  AND    dept.deptno IS NULL
  ```

```
Execution Plan
----------------------------------------------------------
Plan hash value: 1543991079

------------------------------------------------------------------------
| Id  | Operation           | Name | Rows   | Bytes |Cost (%CPU)|Time    |
------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |    14 |  1400 |  5  (20)| 00:00:01 |
|*  1 |  HASH JOIN ANTI     |      |    14 |  1400 |  5  (20)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| EMP  |    14 |  1218 |  2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| DEPT |     4 |    52 |  2   (0)| 00:00:01 |
------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("EMP"."DEPTNO"="DEPT"."DEPTNO")

Note
-----
   - dynamic statistics used: dynamic sampling (level=2)
```

9.3.4.2 How Antijoins Work

The antijoin optimization is implemented differently depending on what type of join is used.

The following pseudocode shows an antijoin for a nested loops join:

```
FOR ds1_row IN ds1 LOOP
  match := true;
  FOR ds2_row IN ds2 LOOP
    IF (ds1_row matches ds2_row) THEN
      match := false;
      EXIT -- stop processing second data set when a match is found
    END IF
  END LOOP
  IF (match = true) THEN
    RETURN ds1_row
  END IF
END LOOP
```

In the preceding pseudocode, `ds1` is the first data set, and `ds2` is the second data set. The code obtains the first row from the first data set, and then loops through the second data set looking for a match. The code exits the inner loop as soon as it finds a match, and begins processing the next row in the first data set.

Example 9-15 Semijoin Using WHERE EXISTS

The following query uses a `WHERE EXISTS` clause to list only the departments that contain employees:

```
SELECT department_id, department_name
FROM   departments
```

```
WHERE EXISTS (SELECT 1
              FROM   employees
              WHERE  employees.department_id =
departments.department_id)
```

The execution plan reveals a `NESTED LOOPS SEMI` operation in Step 1:

```
-----------------------------------------------------------------------
----
| Id| Operation            | Name              |Rows|Bytes |Cost(%CPU)|
Time |
-----------------------------------------------------------------------
----
| 0 | SELECT STATEMENT   |                    |    |      | 2
(100)|          |
| 1 |  NESTED LOOPS SEMI |                    |11  | 209  | 2    (0)|
00:00:01 |
| 2 |   TABLE ACCESS FULL| DEPARTMENTS        |27  | 432  | 2    (0)|
00:00:01 |
|*3 |   INDEX RANGE SCAN | EMP_DEPARTMENT_IX |44  | 132  | 0
(0)|          |
-----------------------------------------------------------------------
----
```

For each row in `departments`, which forms the outer loop, the database obtains the department ID, and then probes the `employees.department_id` index for matching entries. Conceptually, the index looks as follows:

```
10,rowid
10,rowid
10,rowid
10,rowid
30,rowid
30,rowid
30,rowid
...
```

If the first record in the `departments` table is department `30`, then the database performs a range scan of the index until it finds the first `30` entry, at which point it stops reading the index and returns the matching row from `departments`. If the next row in the outer loop is department `20`, then the database scans the index for a `20` entry, and not finding any matches, performs the next iteration of the outer loop. The database proceeds in this way until all matching rows are returned.

9.3.4.3 How Antijoins Handle Nulls

For semijoins, `IN` and `EXISTS` are functionally equivalent. However, `NOT IN` and `NOT EXISTS` are not functionally equivalent because of nulls.

If a null value is returned to a NOT IN operator, then the statement returns no records. To see why, consider the following WHERE clause:

```
WHERE department_id NOT IN (null, 10, 20)
```

The database tests the preceding expression as follows:

```
WHERE  (department_id != null)
AND    (department_id != 10)
AND    (department_id != 20)
```

For the entire expression to be true, each individual condition must be true. However, a null value cannot be compared to another value, so the department_id !=null condition cannot be true, and thus the whole expression is always false. The following techniques enable a statement to return records even when nulls are returned to the NOT IN operator:

- Apply an NVL function to the columns returned by the subquery.
- Add an IS NOT NULL predicate to the subquery.
- Implement NOT NULL constraints.

In contrast to NOT IN, the NOT EXISTS clause only considers predicates that return the existence of a match, and ignores any row that does not match or could not be determined because of nulls. If at least one row in the subquery matches the row from the outer query, then NOT EXISTS returns false. If no tuples match, then NOT EXISTS returns true. The presence of nulls in the subquery does not affect the search for matching records.

In releases earlier than Oracle Database 11g, the optimizer could not use an antijoin optimization when nulls could be returned by a subquery. However, starting in Oracle Database 11g, the ANTI NA (and ANTI SNA) optimizations described in the following sections enable the optimizer to use an antijoin even when nulls are possible.

Example 9-16 Antijoin Using NOT IN

Suppose that a user issues the following query with a NOT IN clause to list the departments that contain no employees:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN
       (SELECT department_id
        FROM   employees);
```

The preceding query returns no rows even though several departments contain no employees. This result, which was not intended by the user, occurs because the employees.department_id column is nullable.

The execution plan reveals a NESTED LOOPS ANTI SNA operation in Step 2:

```
---------------------------------------------------------------------------
| Id| Operation              | Name            |Rows|Bytes|Cost (%CPU)|Time|
---------------------------------------------------------------------------
|  0| SELECT STATEMENT       |                    |    |     | 4(100)|         |
|*1|  FILTER                 |                    |    |     |       |         |
| 2|   NESTED LOOPS ANTI SNA|                   |17 |323 | 4 (50)|
00:00:01|
| 3|    TABLE ACCESS FULL   | DEPARTMENTS       |27 |432 | 2  (0)|
00:00:01|
|*4|    INDEX RANGE SCAN    | EMP_DEPARTMENT_IX |41 |123 | 0
(0)|        |
|*5|   TABLE ACCESS FULL    | EMPLOYEES         | 1 |  3 | 2  (0)|
00:00:01|
-----------------------------------------------------------------------
----

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter( IS NULL)
   4 - access("DEPARTMENT_ID"="DEPARTMENT_ID")
   5 - filter("DEPARTMENT_ID" IS NULL)
```

The `ANTI SNA` stands for "single null-aware antijoin." `ANTI NA` stands for "null-aware antijoin." The null-aware operation enables the optimizer to use the antijoin optimization even on a nullable column. In releases earlier than Oracle Database 11g, the database could not perform antijoins on `NOT IN` queries when nulls were possible.

Suppose that the user rewrites the query by applying an `IS NOT NULL` condition to the subquery:

```
SELECT department_id, department_name
FROM   departments
WHERE  department_id NOT IN
       (SELECT department_id
        FROM   employees
        WHERE  department_id IS NOT NULL);
```

The preceding query returns 16 rows, which is the expected result. Step 1 in the plan shows a standard `NESTED LOOPS ANTI` join instead of an `ANTI NA` or `ANTI SNA` join because the subquery cannot returns nulls:

```
-----------------------------------------------------------------------
----
|Id| Operation          | Name              |Rows|Bytes |Cost (%CPU)|
Time |
-----------------------------------------------------------------------
----
| 0| SELECT STATEMENT   |                   |    |      | 2
(100)|         |
| 1|  NESTED LOOPS ANTI |                   | 17 | 323 | 2   (0)|
00:00:01 |
| 2|   TABLE ACCESS FULL| DEPARTMENTS       | 27 | 432 | 2   (0)|
00:00:01 |
|*3|   INDEX RANGE SCAN | EMP_DEPARTMENT_IX | 41 | 123 | 0
(0)|         |
---------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"="DEPARTMENT_ID")
       filter("DEPARTMENT_ID" IS NOT NULL)
```

Example 9-17   Antijoin Using NOT EXISTS

Suppose that a user issues the following query with a `NOT EXISTS` clause to list the departments that contain no employees:

```
SELECT department_id, department_name
FROM   departments d
WHERE  NOT EXISTS
       (SELECT null
        FROM   employees e
        WHERE  e.department_id = d.department_id)
```

The preceding query avoids the null problem for `NOT IN` clauses. Thus, even though `employees.department_id` column is nullable, the statement returns the desired result.

Step 1 of the execution plan reveals a `NESTED LOOPS ANTI` operation, not the `ANTI NA` variant, which was necessary for `NOT IN` when nulls were possible:

```
---------------------------------------------------------------------------
| Id| Operation          | Name              |Rows|Bytes| Cost (%CPU)|Time|
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT   |                   |    |     | 2 (100)|        |
| 1 |  NESTED LOOPS ANTI |                   | 17 | 323 | 2   (0)|00:00:01|
| 2 |   TABLE ACCESS FULL| DEPARTMENTS       | 27 | 432 | 2   (0)|00:00:01|
|*3 |   INDEX RANGE SCAN | EMP_DEPARTMENT_IX | 41 | 123 | 0   (0)|        |
---------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
```

9.3.5 Cartesian Joins

The database uses a **Cartesian join** when one or more of the tables does not have any join conditions to any other tables in the statement.

The optimizer joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets. Therefore, the total number of rows resulting from the join is calculated using the following formula, where `rs1` is the number of rows in first row set and `rs2` is the number of rows in the second row set:

```
rs1 X rs2 = total rows in result set
```

9.3.5.1 When the Optimizer Considers Cartesian Joins

The optimizer uses a Cartesian join for two row sources only in specific circumstances.

Typically, the situation is one of the following:

- No join condition exists.

  In some cases, the optimizer could pick up a common filter condition between the two tables as a possible join condition.

  > **Note:**
  >
  > If a Cartesian join appears in a query plan, it could be caused by an inadvertently omitted join condition. In general, if a query joins *n* tables, then *n*-1 join conditions are required to avoid a Cartesian join.

- A Cartesian join is an efficient method.

  For example, the optimizer may decide to generate a Cartesian product of two very small tables that are both joined to the same large table.

- The `ORDERED` hint specifies a table before its join table is specified.

9.3.5.2 How Cartesian Joins Work

A Cartesian join uses nested `FOR` loops.

At a high level, the algorithm for a Cartesian join looks as follows, where `ds1` is typically the smaller data set, and `ds2` is the larger data set:

```
FOR ds1_row IN ds1 LOOP
  FOR ds2_row IN ds2 LOOP
    output ds1_row and ds2_row
  END LOOP
END LOOP
```

Example 9-18   Cartesian Join

In this example, a user intends to perform an inner join of the `employees` and `departments` tables, but accidentally leaves off the join condition:

```
SELECT e.last_name, d.department_name
FROM   employees e, departments d
```

The execution plan is as follows:

```
---------------------------------------------------------------------------
| Id| Operation             | Name         | Rows  | Bytes |Cost  (%CPU)|Time|
---------------------------------------------------------------------------
| 0| SELECT STATEMENT       |              |       |       |11  (100)|        |
| 1|  MERGE JOIN CARTESIAN  |              | 2889 |57780 |11    (0)|00:00:01|
| 2|   TABLE ACCESS FULL    | DEPARTMENTS |   27 |   324 | 2    (0)|00:00:01|
| 3|   BUFFER SORT          |              |  107 |   856 | 9    (0)|00:00:01|
| 4|    INDEX FAST FULL SCAN| EMP_NAME_IX |  107 |   856 | 0    (0)|        |
---------------------------------------------------------------------------
```

In Step 1 of the preceding plan, the `CARTESIAN` keyword indicates the presence of a Cartesian join. The number of rows (2889) is the product of 27 and 107.

In Step 3, the `BUFFER SORT` operation indicates that the database is copying the data blocks obtained by the scan of `emp_name_ix` from the SGA to the PGA. This strategy avoids multiple scans of the same blocks in the database buffer cache, which would generate many logical reads and permit resource contention.

9.3.5.3 Cartesian Join Controls

The `ORDERED` hint instructs the optimizer to join tables in the order in which they appear in the `FROM` clause. By forcing a join between two row sources that have no direct connection, the optimizer must perform a Cartesian join.

Example 9-19 ORDERED Hint

In the following example, the `ORDERED` hint instructs the optimizer to join `employees` and `locations`, but no join condition connects these two row sources:

```
SELECT /*+ORDERED*/ e.last_name, d.department_name, l.country_id,
l.state_province
FROM   employees e, locations l, departments d
WHERE  e.department_id = d.department_id
AND    d.location_id = l.location_id
```

The following execution plan shows a Cartesian product (Step 3) between `locations` (Step 6) and `employees` (Step 4), which is then joined to the `departments` table (Step 2):

```
---------------------------------------------------------------------------
| Id| Operation             | Name        |Rows | Bytes |Cost (%CPU)|Time |
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT       |             |     |       |37 (100)|         |
|*1 |  HASH JOIN             |             | 106 | 4664 |37    (6)|00:00:01 |
| 2 |   TABLE ACCESS FULL    | DEPARTMENTS |  27 |   513 | 2    (0)|00:00:01 |
| 3 |   MERGE JOIN CARTESIAN|             |2461 |61525 |34    (3)|00:00:01 |
| 4 |    TABLE ACCESS FULL   | EMPLOYEES   | 107 | 1177 | 2    (0)|00:00:01 |
| 5 |    BUFFER SORT         |             |  23 |   322 |32    (4)|00:00:01 |
| 6 |     TABLE ACCESS FULL  | LOCATIONS   |  23 |   322 | 0    (0)|         |
---------------------------------------------------------------------------
```

See Also:

Oracle Database SQL Language Reference to learn about the `ORDERED` hint

9.4 Join Optimizations

Join optimizations enable joins to be more efficient.

9.4.1 Bloom Filters

A **Bloom filter**, named after its creator Burton Bloom, is a low-memory data structure that tests membership in a set.

A Bloom filter correctly indicates when an element is not in a set, but can incorrectly indicate when an element is in a set. Thus, false negatives are impossible but false positives are possible.

9.4.1.1 Purpose of Bloom Filters

A Bloom filter tests one set of values to determine whether they are members another set.

For example, one set is (10,20,30,40) and the second set is (10,30,60,70). A Bloom filter can determine that 60 and 70 are *guaranteed* to be excluded from the first set, and that 10 and 30 are *probably* members. Bloom filters are especially useful when the amount of memory needed to store the filter is small relative to the amount of data in the data set, and when most data is expected to fail the membership test.

Oracle Database uses Bloom filters to various specific goals, including the following:

- Reduce the amount of data transferred to child processes in a parallel query, especially when the database discards most rows because they do not fulfill a join condition
- Eliminate unneeded partitions when building a partition access list in a join, known as *partition pruning*
- Test whether data exists in the server result cache, thereby avoiding a disk read
- Filter members in Exadata cells, especially when joining a large fact table and small dimension tables in a star schema

Bloom filters can occur in both parallel and serial processing.

9.4.1.2 How Bloom Filters Work

A Bloom filter uses an array of bits to indicate inclusion in a set.

For example, 8 elements (an arbitrary number used for this example) in an array are initially set to `0`:

```
e1 e2 e3 e4 e5 e6 e7 e8
 0  0  0  0  0  0  0  0
```

This array represents a set. To represent an input value *i* in this array, three separate hash functions (three is arbitrary) are applied to *i*, each generating a hash value between 1 and 8:

```
f1(i) = h1
f2(i) = h2
f3(i) = h3
```

For example, to store the value 17 in this array, the hash functions set *i* to 17, and then return the following hash values:

```
f1(17) = 5
f2(17) = 3
f3(17) = 5
```

In the preceding example, two of the hash functions happened to return the same value of 5, known as a *hash collision*. Because the distinct hash values are 5 and 3, the 5th and 3rd elements in the array are set to 1:

```
e1 e2 e3 e4 e5 e6 e7 e8
 0  0  1  0  1  0  0  0
```

Testing the membership of 17 in the set reverses the process. To test whether the set *excludes* the value 17, element 3 or element 5 must contain a 0. If a 0 is present in either element, then the set cannot contain 17. No false negatives are possible.

To test whether the set *includes* 17, both element 3 and element 5 must contain 1 values. However, if the test indicates a 1 for both elements, then it is still possible for the set *not* to include 17. False positives are possible. For example, the following array might represent the value 22, which also has a 1 for both element 3 and element 5:

```
e1 e2 e3 e4 e5 e6 e7 e8
 1  0  1  0  1  0  0  0
```

9.4.1.3 Bloom Filter Controls

The optimizer automatically determines whether to use Bloom filters.

To override optimizer decisions, use the hints PX_JOIN_FILTER and NO_PX_JOIN_FILTER.

See Also:

Oracle Database SQL Language Reference to learn more about the bloom filter hints

9.4.1.4 Bloom Filter Metadata

V$ views contain metadata about Bloom filters.

You can query the following views:

- `V$SQL_JOIN_FILTER`

  This view shows the number of rows filtered out (`FILTERED` column) and tested (`PROBED` column) by an active Bloom filter.

- `V$PQ_TQSTAT`

  This view displays the number of rows processed through each parallel execution server at each stage of the execution tree. You can use it to monitor how much Bloom filters have reduced data transfer among parallel processes.

In an execution plan, a Bloom filter is indicated by keywords `JOIN FILTER` in the `Operation` column, and the prefix `:BF` in the `Name` column, as in the 9th step of the following plan snippet:

```
------------------------------------------------------------------------
----
| Id  | Operation                  | Name     |    TQ  |IN-OUT| PQ
Distrib |
------------------------------------------------------------------------
----
...
|   9 |      JOIN FILTER CREATE    | :BF0000  |  Q1,03 | PCWP
|             |
```

In the `Predicate Information` section of the plan, filters that contain functions beginning with the string `SYS_OP_BLOOM_FILTER` indicate use of a Bloom filter.

9.4.1.5 Bloom Filters: Scenario

In this example, a parallel query joins the `sales` fact table to the `products` and `times` dimension tables, and filters on fiscal week `18`.

```
SELECT /*+ parallel(s) */ p.prod_name, s.quantity_sold
FROM   sh.sales s, sh.products p, sh.times t
WHERE  s.prod_id = p.prod_id
AND    s.time_id = t.time_id
AND    t.fiscal_week_number = 18;
```

Querying `DBMS_XPLAN.DISPLAY_CURSOR` provides the following output:

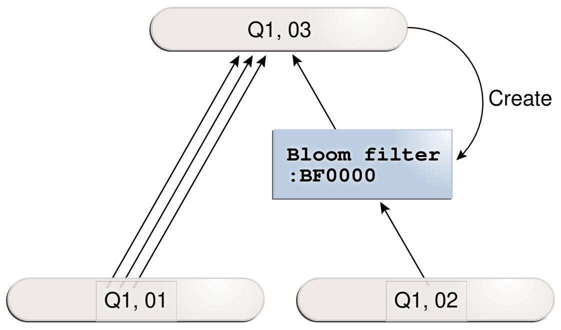
```
SELECT * FROM
  TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format =>
'BASIC,+PARALLEL,+PREDICATE'));

EXPLAINED SQL STATEMENT:
------------------------
SELECT /*+ parallel(s) */ p.prod_name, s.quantity_sold FROM sh.sales s,
sh.products p, sh.times t WHERE s.prod_id = p.prod_id AND s.time_id =
t.time_id AND t.fiscal_week_number = 18

Plan hash value: 1183628457

------------------------------------------------------------------------
----
```

```
| Id | Operation                    | Name     |   TQ   |IN-OUT| PQ Distrib |
---------------------------------------------------------------------------
|  0 | SELECT STATEMENT             |          |        |      |            |
|  1 |  PX COORDINATOR              |          |        |      |            |
|  2 |   PX SEND QC (RANDOM)        | :TQ10003 |  Q1,03 | P->S | QC (RAND)  |
|* 3 |    HASH JOIN BUFFERED        |          |  Q1,03 | PCWP |            |
|  4 |     PX RECEIVE               |          |  Q1,03 | PCWP |            |
|  5 |      PX SEND BROADCAST       | :TQ10001 |  Q1,01 | S->P | BROADCAST  |
|  6 |       PX SELECTOR            |          |  Q1,01 | SCWC |            |
|  7 |        TABLE ACCESS FULL     | PRODUCTS |  Q1,01 | SCWP |            |
|* 8 |     HASH JOIN                |          |  Q1,03 | PCWP |            |
|  9 |      JOIN FILTER CREATE      | :BF0000  |  Q1,03 | PCWP |            |
| 10 |       BUFFER SORT            |          |  Q1,03 | PCWC |            |
| 11 |        PX RECEIVE            |          |  Q1,03 | PCWP |            |
| 12 |         PX SEND HYBRID HASH| :TQ10000 |        | S->P | HYBRID HASH|
|*13 |          TABLE ACCESS FULL | TIMES    |        |      |            |
| 14 |      PX RECEIVE              |          |  Q1,03 | PCWP |            |
| 15 |       PX SEND HYBRID HASH    | :TQ10002 |  Q1,02 | P->P | HYBRID HASH|
| 16 |        JOIN FILTER USE       | :BF0000  |  Q1,02 | PCWP |            |
| 17 |         PX BLOCK ITERATOR    |          |  Q1,02 | PCWC |            |
|*18 |          TABLE ACCESS FULL | SALES    |  Q1,02 | PCWP |            |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("S"."PROD_ID"="P"."PROD_ID")
   8 - access("S"."TIME_ID"="T"."TIME_ID")
  13 - filter("T"."FISCAL_WEEK_NUMBER"=18)
  18 - access(:Z>=:Z AND :Z<=:Z)
       filter(SYS_OP_BLOOM_FILTER(:BF0000,"S"."TIME_ID"))
```

A single server process scans the `times` table (Step 13), and then uses a hybrid hash distribution method to send the rows to the parallel execution servers (Step 12). The processes in set `Q1,03` create a bloom filter (Step 9). The processes in set `Q1,02` scan `sales` in parallel (Step 18), and then use the Bloom filter to discard rows from `sales` (Step 16) before sending them on to set `Q1,03` using hybrid hash distribution (Step 15). The processes in set `Q1,03` hash join the `times` rows to the filtered `sales` rows (Step 8). The processes in set `Q1,01` scan `products` (Step 7), and then send the rows to `Q1,03` (Step 5). Finally, the processes in `Q1,03` join the `products` rows to the rows generated by the previous hash join (Step 3).

The following figure illustrates the basic process.

Figure 9-8 Bloom Filter



9.4.2 Partition-Wise Joins

A **partition-wise join** is an optimization that divides a large join of two tables, one of which must be partitioned on the join key, into several smaller joins.

Partition-wise joins are either of the following:

- Full partition-wise join

  Both tables must be equipartitioned on their join keys, or use reference partitioning (that is, be related by referential constraints). The database divides a large join into smaller joins between two partitions from the two joined tables.

- Partial partition-wise joins

  Only one table is partitioned on the join key. The other table may or may not be partitioned.

See Also:

Oracle Database VLDB and Partitioning Guide explains partition-wise joins in detail

9.4.2.1 Purpose of Partition-Wise Joins

Partition-wise joins reduce query response time by minimizing the amount of data exchanged among parallel execution servers when joins execute in parallel.

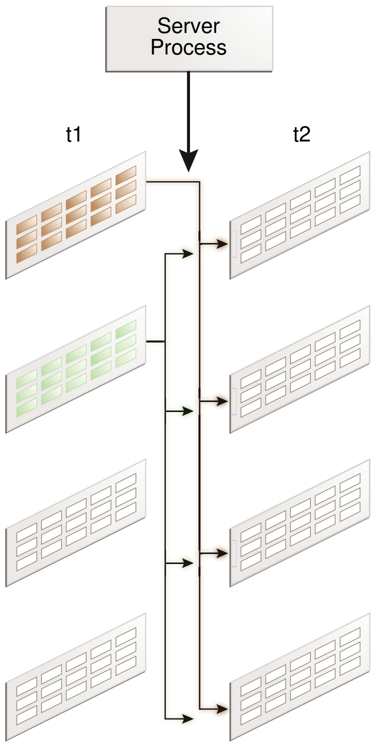
This technique significantly reduces response time and improves the use of CPU and memory. In Oracle Real Application Clusters (Oracle RAC) environments, partition-wise joins also avoid or at least limit the data traffic over the interconnect, which is the key to achieving good scalability for massive join operations.

9.4.2.2 How Partition-Wise Joins Work

When the database serially joins two partitioned tables *without* using a partition-wise join, a single server process performs the join.

In the following illustration, the join is *not* partition-wise because the server process joins every partition of table `t1` to every partition of table `t2`.

Figure 9-9   Join That Is Not Partition-Wise

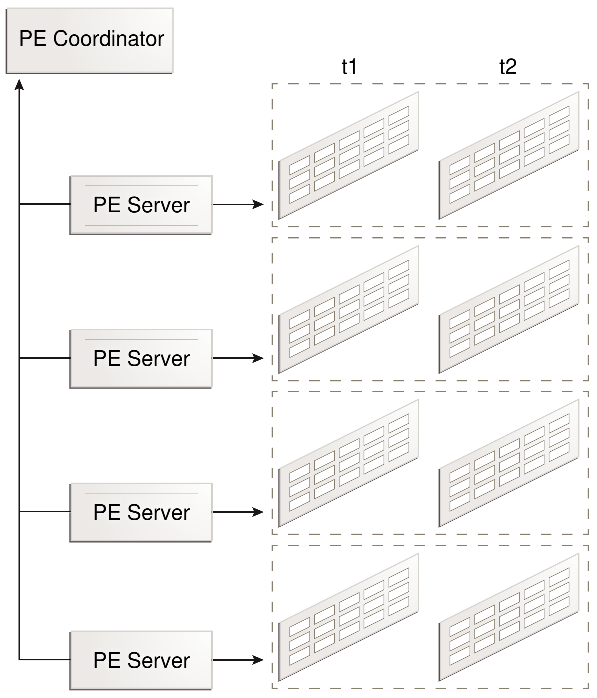


9.4.2.2.1 How a Full Partition-Wise Join Works

The database performs a full partition-wise join either serially or in parallel.

The following graphic shows a full partition-wise join performed in parallel. In this case, the granule of parallelism is a partition. Each parallel execution server joins the partitions in pairs. For example, the first parallel execution server joins the first partition of `t1` to the first partition of `t2`. The parallel execution coordinator then assembles the result.

Figure 9-10 Full Partition-Wise Join in Parallel



A full partition-wise join can also join partitions to subpartitions, which is useful when the tables use different partitioning methods. For example, `customers` is partitioned by hash, but `sales` is partitioned by range. If you subpartition `sales` by hash, then the database can perform a full partition-wise join between the hash partitions of the `customers` and the hash subpartitions of `sales`.

In the execution plan, the presence of a partition operation before the join signals the presence of a full partition-wise join, as in the following snippet:

```
|   8 |         PX PARTITION HASH ALL|
|*  9 |          HASH JOIN           |
```

See Also:

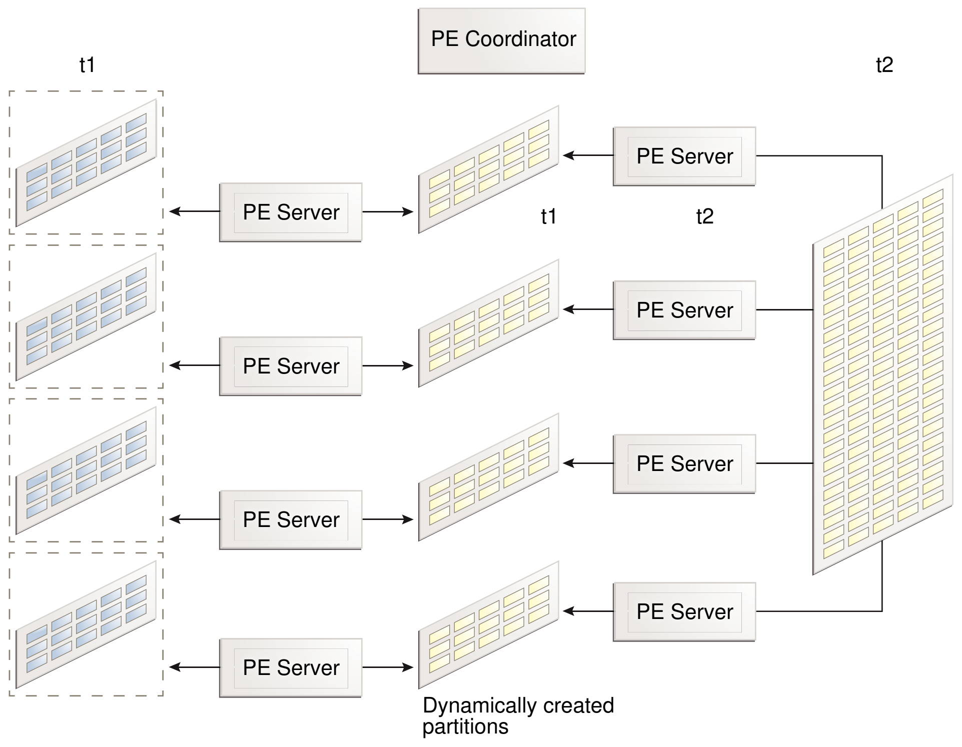
Oracle Database VLDB and Partitioning Guide explains full partition-wise joins in detail, and includes several examples

9.4.2.2.2 How a Partial Partition-Wise Join Works

Partial partition-wise joins, unlike their full partition-wise counterpart, must execute in parallel.

The following graphic shows a partial partition-wise join between `t1`, which is partitioned, and `t2`, which is not partitioned.

Figure 9-11 Partial Partition-Wise Join



Because `t2` is not partitioned, a set of parallel execution servers must generate partitions from `t2` as needed. A different set of parallel execution servers then joins the `t1` partitions to the dynamically generated partitions. The parallel execution coordinator assembles the result.

In the execution plan, the operation `PX SEND PARTITION (KEY)` signals a partial partition-wise join, as in the following snippet:

```
|  11 |             PX SEND PARTITION (KEY)    |
```

See Also:

Oracle Database VLDB and Partitioning Guide explains full partition-wise joins in detail, and includes several examples

9.4.3 In-Memory Join Groups

A **join group** is a user-created object that lists two or more columns that can be meaningfully joined.

In certain queries, join groups eliminate the performance overhead of decompressing and hashing column values. Join groups require an In-Memory Column Store (IM column store).

See Also:

Oracle Database In-Memory Guide to learn how to optimize In-Memory queries with join groups

Part V
Optimizer Statistics

The accuracy of an execution plan depends on the quality of the optimizer statistics.

10 Optimizer Statistics Concepts

Oracle Database **optimizer statistics** describe details about the database and its objects.

10.1 Introduction to Optimizer Statistics

The optimizer **cost model** relies on statistics collected about the objects involved in a query, and the database and host where the query runs.

The optimizer uses statistics to get an estimate of the number of rows (and number of bytes) retrieved from a table, partition, or index. The optimizer estimates the cost for the access, determines the cost for possible plans, and then picks the execution plan with the lowest cost.

Optimizer statistics include the following:

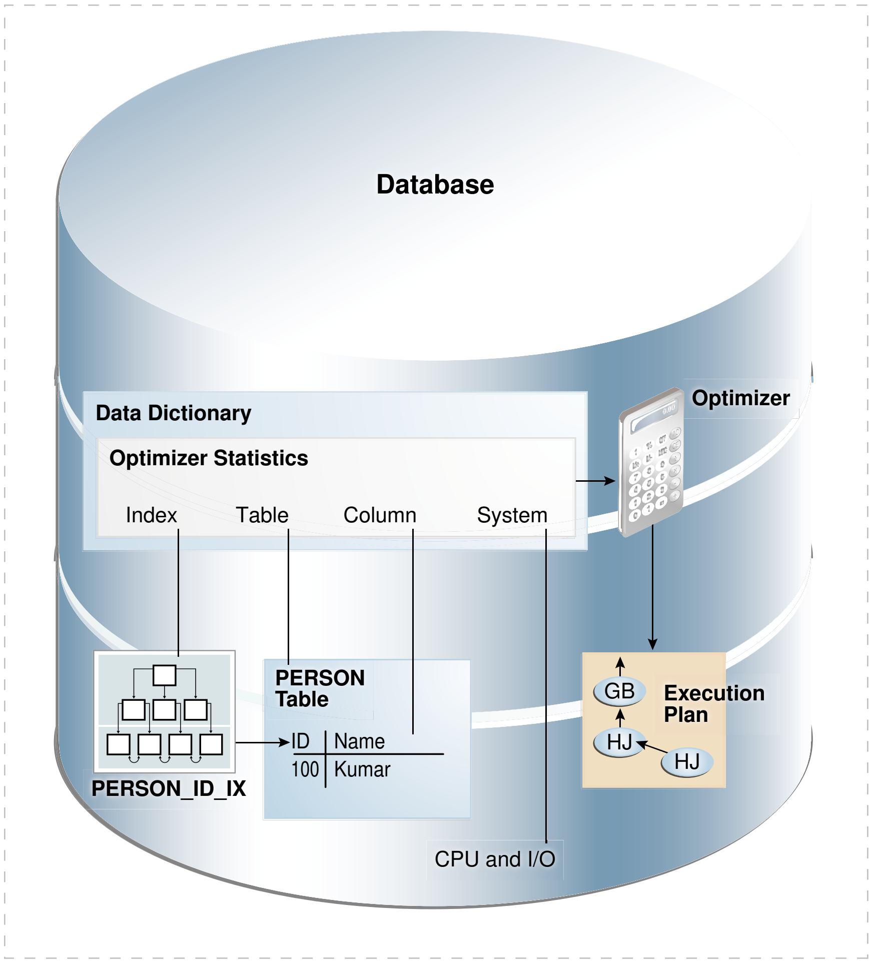
- Table statistics
  - Number of rows
  - Number of blocks
  - Average row length
- Column statistics
  - Number of distinct values (NDV) in a column
  - Number of nulls in a column
  - Data distribution (histogram)
  - Extended statistics
- Index statistics
  - Number of leaf blocks
  - Number of levels
  - Index clustering factor
- System statistics
  - I/O performance and utilization
  - CPU performance and utilization

As shown in Figure 10-1, the database stores optimizer statistics for tables, columns, indexes, and the system in the data dictionary. You can access these statistics using data dictionary views.

Note:

The optimizer statistics are different from the performance statistics visible through `V$` views.

Figure 10-1 Optimizer Statistics



10.2 About Optimizer Statistics Types

The optimizer collects statistics on different types of database objects and characteristics of the database environment.

10.2.1 Table Statistics

Table statistics contain metadata that the optimizer uses when developing an execution plan.

10.2.1.1 Permanent Table Statistics

In Oracle Database, **table statistics** include information about rows and blocks.

The optimizer uses these statistics to determine the cost of table scans and table joins. The database tracks all relevant statistics about permanent tables. For example, table statistics stored in DBA_TAB_STATISTICS track the following:

- Number of rows

  The database uses the row count stored in DBA_TAB_STATISTICS when determining cardinality.
- Average row length
- Number of data blocks

  The optimizer uses the number of data blocks with the DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter to determine the base table access cost.
- Number of empty data blocks

DBMS_STATS.GATHER_TABLE_STATS commits before gathering statistics on permanent tables.

Example 10-1 Table Statistics

This example queries table statistics for the sh.customers table.

```
SELECT NUM_ROWS, AVG_ROW_LEN, BLOCKS,
       EMPTY_BLOCKS, LAST_ANALYZED
FROM   DBA_TAB_STATISTICS
WHERE  OWNER='SH'
AND    TABLE_NAME='CUSTOMERS';
```

Sample output appears as follows:

```
  NUM_ROWS AVG_ROW_LEN     BLOCKS EMPTY_BLOCKS LAST_ANAL
---------- ----------- ---------- ------------ ---------
     55500         189       1517            0 25-MAY-17
```

See Also:

- "About Optimizer Initialization Parameters"
- "Gathering Schema and Table Statistics"
- *Oracle Database Reference* for a description of the DBA_TAB_STATISTICS view and the DB_FILE_MULTIBLOCK_READ_COUNT initialization parameter

10.2.1.2 Temporary Table Statistics

DBMS_STATS can gather statistics for both permanent and global temporary tables, but additional considerations apply to the latter.

10.2.1.2.1 Types of Temporary Tables

Temporary tables are classified as global, private, or cursor-duration.

In all types of temporary table, the data is only visible to the session that inserts it. The tables differ as follows:

- A **global temporary table** is an explicitly created persistent object that stores intermediate session-private data for a specific duration.

  The table is global because the definition is visible to all sessions. The `ON COMMIT` clause of `CREATE GLOBAL TEMPORARY TABLE` indicates whether the table is transaction-specific (`DELETE ROWS`) or session-specific (`PRESERVE ROWS`). Optimizer statistics for global temporary tables can be shared or session-specific.
- A **private temporary table** is an explicitly created object, defined by private memory-only metadata, that stores intermediate session-private data for a specific duration.

  The table is private because the definition is visible only to the session that created the table. The `ON COMMIT` clause of `CREATE PRIVATE TEMPORARY TABLE` indicates whether the table is transaction-specific (`DROP DEFINITION`) or session-specific (`PRESERVE DEFINITION`).
- A **cursor-duration temporary table** is an implicitly created memory-only object that is associated with a cursor.

  Unlike global and private temporary tables, `DBMS_STATS` cannot gather statistics for cursor-duration temporary tables.

The tables differ in where they store data, how they are created and dropped, and in the duration and visibility of metadata. Note that the database allocates storage space when a session first inserts data into a global temporary table, not at table creation.

Table 10-1 Important Characteristics of Temporary Tables

Characteristic	Global Temporary Table	Private Temporary Table	Cursor-Duration Temporary Table
Visibility of Data	Session inserting data	Session inserting data	Session inserting data
Storage of Data	Persistent	Memory or tempfiles, but only for the duration of the session or transaction	Only in memory
Visibility of Metadata	All sessions	Session that created table (in `USER_PRIVATE_TEMP_TABLES` view, which is based on a `V$` view)	Session executing cursor
Duration of Metadata	Until table is explicitly dropped	Until table is explicitly dropped, or end of session (`PRESERVE DEFINITION`) or transaction (`DROP DEFINITION`)	Until cursor ages out of shared pool

Table 10-1 (Cont.) Important Characteristics of Temporary Tables

Characteristic	Global Temporary Table	Private Temporary Table	Cursor-Duration Temporary Table
Creation of Table	`CREATE GLOBAL TEMPORARY TABLE` (supports `AS SELECT`)	`CREATE PRIVATE TEMPORARY TABLE` (supports `AS SELECT`)	Implicitly created when optimizer considers it useful
Effect of Creation on Existing Transactions	No implicit commit	No implicit commit	No implicit commit
Naming Rules	Same as for permanent tables	Must begin with `ORA$PTT_`	Internally generated unique name
Dropping of Table	`DROP GLOBAL TEMPORARY TABLE`	`DROP PRIVATE TEMPORARY TABLE`, or implicitly dropped at end of session (`PRESERVE DEFINITION`) or transaction (`DROP DEFINITION`)	Implicitly dropped at end of session

See Also:

- "Cursor-Duration Temporary Tables"
- *Oracle Database Administrator’s Guide* to learn how to manage temporary tables

10.2.1.2.2 Statistics for Global Temporary Tables

`DBMS_STATS` collects the same types of statistics for global temporary tables as for permanent tables.

Note:

You cannot collect statistics for private temporary tables.

The following table shows how global temporary tables differ in how they gather and store optimizer statistics, depending on whether the tables are scoped to a transaction or session.

Table 10-2 Optimizer Statistics for Global Temporary Tables

Characteristic	Transaction-Specific	Session-Specific
Effect of `DBMS_STATS` collection	Does not commit	Commits
Storage of statistics	Memory only	Dictionary tables
Histogram creation	Not supported	Supported

The following procedures do not commit for transaction-specific temporary tables, so that rows in these tables are not deleted:

- `GATHER_TABLE_STATS`
- `DELETE_`*`obj`*`_STATS`, where *`obj`* is `TABLE`, `COLUMN`, or `INDEX`
- `SET_`*`obj`*`_STATS`, where *`obj`* is `TABLE`, `COLUMN`, or `INDEX`
- `GET_`*`obj`*`_STATS`, where *`obj`* is `TABLE`, `COLUMN`, or `INDEX`

The preceding program units observe the `GLOBAL_TEMP_TABLE_STATS` statistics preference. For example, if the table preference is set to `SESSION`, then `SET_TABLE_STATS` sets the session statistics, and `GATHER_TABLE_STATS` *preserves* all rows in a transaction-specific temporary table. If the table preference is set to `SHARED`, however, then `SET_TABLE_STATS` sets the shared statistics, and `GATHER_TABLE_STATS` *deletes* all rows from a transaction-specific temporary table.

See Also:

- "Gathering Schema and Table Statistics"
- *Oracle Database Concepts* to learn about global temporary tables
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure

10.2.1.2.3 Shared and Session-Specific Statistics for Global Temporary Tables

Starting in Oracle Database 12c, you can set the table-level preference `GLOBAL_TEMP_TABLE_STATS` to make statistics on a global temporary table shared (`SHARED`) or session-specific (`SESSION`).

When `GLOBAL_TEMP_TABLE_STATS` is `SESSION`, you can gather optimizer statistics for a global temporary table in one session, and then use the statistics for this session only. Meanwhile, users can continue to maintain a shared version of the statistics. During optimization, the optimizer first checks whether a global temporary table has session-specific statistics. If yes, then the optimizer uses them. Otherwise, the optimizer uses shared statistics if they exist.

Note:

In releases before Oracle Database 12c, the database did not maintain optimizer statistics for global temporary tables and non-global temporary tables differently. The database maintained one version of the statistics shared by all sessions, even though data in different sessions could differ.

Session-specific optimizer statistics have the following characteristics:

- Dictionary views that track statistics show both the shared statistics and the session-specific statistics in the current session.

The views are `DBA_TAB_STATISTICS`, `DBA_IND_STATISTICS`, `DBA_TAB_HISTOGRAMS`, and `DBA_TAB_COL_STATISTICS` (each view has a corresponding `USER_` and `ALL_` version). The `SCOPE` column shows whether statistics are session-specific or shared. Session-specific statistics must be stored in the data dictionary so that multiple processes can access them in Oracle RAC.

- `CREATE ... AS SELECT` automatically gathers optimizer statistics. When `GLOBAL_TEMP_TABLE_STATS` is set to `SHARED`, however, you must gather statistics manually using `DBMS_STATS`.
- Pending statistics are not supported.
- Other sessions do not share a cursor that uses the session-specific statistics.

  Different sessions can share a cursor that uses *shared* statistics, as in releases earlier than Oracle Database 12c. The same session can share a cursor that uses session-specific statistics.
- By default, `GATHER_TABLE_STATS` for the temporary table immediately invalidates previous cursors compiled in the same session. However, this procedure does not invalidate cursors compiled in other sessions.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `GLOBAL_TEMP_TABLE_STATS` preference
- *Oracle Database Reference* for a description of the `DBA_TAB_STATISTICS` view

10.2.2 Column Statistics

Column statistics track information about column values and data distribution.

The optimizer uses column statistics to generate accurate **cardinality** estimates and make better decisions about index usage, join orders, join methods, and so on. For example, statistics in `DBA_TAB_COL_STATISTICS` track the following:

- Number of distinct values
- Number of nulls
- High and low values
- Histogram-related information

The optimizer can use extended statistics, which are a special type of column statistics. These statistics are useful for informing the optimizer of logical relationships among columns.

See Also:

- "Histograms "
- "About Statistics on Column Groups"
- *Oracle Database Reference* for a description of the `DBA_TAB_COL_STATISTICS` view

10.2.3 Index Statistics

The **index statistics** include information about the number of index levels, the number of index blocks, and the relationship between the index and the data blocks. The optimizer uses these statistics to determine the cost of index scans.

10.2.3.1 Types of Index Statistics

The `DBA_IND_STATISTICS` view tracks index statistics.

Statistics include the following:

- Levels

  The `BLEVEL` column shows the number of blocks required to go from the root block to a leaf block. A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values. See *Oracle Database Concepts* for a conceptual overview of B-tree indexes.
- Distinct keys

  This columns tracks the number of distinct indexed values. If a unique constraint is defined, and if no `NOT NULL` constraint is defined, then this value equals the number of non-null values.
- Average number of leaf blocks for each distinct indexed key
- Average number of data blocks pointed to by each distinct indexed key

See Also:

Oracle Database Reference for a description of the `DBA_IND_STATISTICS` view

Example 10-2 Index Statistics

This example queries some index statistics for the `cust_lname_ix` and `customers_pk` indexes on the `sh.customers` table (sample output included):

```
SELECT INDEX_NAME, BLEVEL, LEAF_BLOCKS AS "LEAFBLK", DISTINCT_KEYS AS
"DIST_KEY",
       AVG_LEAF_BLOCKS_PER_KEY AS "LEAFBLK_PER_KEY",
       AVG_DATA_BLOCKS_PER_KEY AS "DATABLK_PER_KEY"
FROM   DBA_IND_STATISTICS
```

```
WHERE  OWNER = 'SH'
AND    INDEX_NAME IN ('CUST_LNAME_IX','CUSTOMERS_PK');

INDEX_NAME     BLEVEL LEAFBLK DIST_KEY LEAFBLK_PER_KEY DATABLK_PER_KEY
-------------- ------ ------- -------- --------------- ---------------
CUSTOMERS_PK        1     115    55500               1               1
CUST_LNAME_IX       1     141      908               1              10
```

10.2.3.2 Index Clustering Factor

For a B-tree index, the **index clustering factor** measures the physical grouping of rows in relation to an index value, such as last name.

The index clustering factor helps the optimizer decide whether an index scan or full table scan is more efficient for certain queries). A low clustering factor indicates an efficient index scan.

A clustering factor that is close to the number of *blocks* in a table indicates that the rows are physically ordered in the table blocks by the index key. If the database performs a full table scan, then the database tends to retrieve the rows as they are stored on disk sorted by the index key. A clustering factor that is close to the number of *rows* indicates that the rows are scattered randomly across the database blocks in relation to the index key. If the database performs a full table scan, then the database would not retrieve rows in any sorted order by this index key.

The clustering factor is a property of a specific index, not a table. If multiple indexes exist on a table, then the clustering factor for one index might be small while the factor for another index is large. An attempt to reorganize the table to improve the clustering factor for one index may degrade the clustering factor of the other index.

Example 10-3 Index Clustering Factor

This example shows how the optimizer uses the index clustering factor to determine whether using an index is more effective than a full table scan.

1. Start SQL*Plus and connect to a database as `sh`, and then query the number of rows and blocks in the `sh.customers` table (sample output included):

   ```
   SELECT  table_name, num_rows, blocks
   FROM    user_tables
   WHERE   table_name='CUSTOMERS';

   TABLE_NAME                       NUM_ROWS     BLOCKS
   ------------------------------ ---------- ----------
   CUSTOMERS                           55500       1486
   ```

2. Create an index on the `customers.cust_last_name` column.

   For example, execute the following statement:

   ```
   CREATE INDEX CUSTOMERS_LAST_NAME_IDX ON customers(cust_last_name);
   ```

3. Query the index clustering factor of the newly created index.

The following query shows that the `customers_last_name_idx` index has a high clustering factor because the clustering factor is significantly more than the number of blocks in the table:

```
SELECT index_name, blevel, leaf_blocks, clustering_factor
FROM   user_indexes
WHERE  table_name='CUSTOMERS'
AND    index_name= 'CUSTOMERS_LAST_NAME_IDX';

INDEX_NAME                         BLEVEL LEAF_BLOCKS
CLUSTERING_FACTOR
------------------------------ ---------- -----------
-----------------
CUSTOMERS_LAST_NAME_IDX                 1         141
9859
```

4. Create a new copy of the `customers` table, with rows ordered by `cust_last_name`.

   For example, execute the following statements:

```
DROP TABLE customers3 PURGE;
CREATE TABLE customers3 AS
  SELECT *
  FROM   customers
  ORDER BY cust_last_name;
```

5. Gather statistics on the `customers3` table.

   For example, execute the `GATHER_TABLE_STATS` procedure as follows:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(null,'CUSTOMERS3');
```

6. Query the number of rows and blocks in the `customers3` table .

   For example, enter the following query (sample output included):

```
SELECT     TABLE_NAME, NUM_ROWS, BLOCKS
FROM       USER_TABLES
WHERE      TABLE_NAME='CUSTOMERS3';

TABLE_NAME                       NUM_ROWS     BLOCKS
------------------------------ ---------- ----------
CUSTOMERS3                          55500       1485
```

7. Create an index on the `cust_last_name` column of `customers3`.

   For example, execute the following statement:

```
CREATE INDEX CUSTOMERS3_LAST_NAME_IDX ON customers3(cust_last_name);
```

8. Query the index clustering factor of the `customers3_last_name_idx` index.

   The following query shows that the `customers3_last_name_idx` index has a lower clustering factor:

```
SELECT INDEX_NAME, BLEVEL, LEAF_BLOCKS, CLUSTERING_FACTOR
FROM   USER_INDEXES
```

```
WHERE  TABLE_NAME = 'CUSTOMERS3'
AND    INDEX_NAME = 'CUSTOMERS3_LAST_NAME_IDX';

INDEX_NAME                         BLEVEL LEAF_BLOCKS CLUSTERING_FACTOR
------------------------------ ---------- ----------- -----------------
CUSTOMERS3_LAST_NAME_IDX                1         141              1455
```

The table customers3 has the same data as the original customers table, but the index on customers3 has a much lower clustering factor because the data in the table is ordered by the cust_last_name. The clustering factor is now about 10 times the number of blocks instead of 70 times.

9. Query the customers table.

   For example, execute the following query (sample output included):

```
SELECT cust_first_name, cust_last_name
FROM   customers
WHERE  cust_last_name BETWEEN 'Puleo' AND 'Quinn';

CUST_FIRST_NAME      CUST_LAST_NAME
-------------------- ----------------------------------------
Vida                 Puleo
Harriett             Quinlan
Madeleine            Quinn
Caresse              Puleo
```

10. Display the cursor for the query.

    For example, execute the following query (partial sample output included):

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());

-------------------------------------------------------------------------------
| Id | Operation                 | Name      | Rows |Bytes|Cost (%CPU)| Time   |
-------------------------------------------------------------------------------
|   0| SELECT STATEMENT          |           |      |     | 405 (100)|        |
|*  1|  TABLE ACCESS STORAGE FULL| CUSTOMERS |  2335|35025| 405   (1)|00:00:01|
-------------------------------------------------------------------------------
```

The preceding plan shows that the optimizer did not use the index on the original customers tables.

11. Query the customers3 table.

    For example, execute the following query (sample output included):

```
SELECT cust_first_name, cust_last_name
FROM   customers3
WHERE  cust_last_name BETWEEN 'Puleo' AND 'Quinn';

CUST_FIRST_NAME      CUST_LAST_NAME
-------------------- ----------------------------------------
Vida                 Puleo
Harriett             Quinlan
```

```
Madeleine            Quinn
Caresse              Puleo
```

12. Display the cursor for the query.

    For example, execute the following query (partial sample output included):

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());

-------------------------------------------------------------------------------------
----
|Id| Operation                   | Name                    |Rows|Bytes|Cost(%CPU)|
Time|
-------------------------------------------------------------------------------------
----
| 0| SELECT STATEMENT            |                         |    |     |
69(100)|         |
| 1|  TABLE ACCESS BY INDEX ROWID|CUSTOMERS3               |2335|35025|69(0)   |
00:00:01|
|*2|   INDEX RANGE SCAN          |CUSTOMERS3_LAST_NAME_IDX|2335|     |7(0)    |
00:00:01|
-------------------------------------------------------------------------------------
----
```

    The result set is the same, but the optimizer chooses the index. The plan cost is much less than the cost of the plan used on the original `customers` table.

13. Query `customers` with a hint that forces the optimizer to use the index.

    For example, execute the following query (partial sample output included):

```
SELECT /*+ index (Customers CUSTOMERS_LAST_NAME_IDX) */
cust_first_name,
       cust_last_name
FROM   customers
WHERE  cust_last_name BETWEEN 'Puleo' and 'Quinn';

CUST_FIRST_NAME      CUST_LAST_NAME
-------------------- ----------------------------------------
Vida                 Puleo
Caresse              Puleo
Harriett             Quinlan
Madeleine            Quinn
```

14. Display the cursor for the query.

    For example, execute the following query (partial sample output included):

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());

-------------------------------------------------------------------------------------
----
| Id | Operation                 | Name                    |Rows|Bytes|Cost(%CPU)|
Time    |
-------------------------------------------------------------------------------------
----
```

```
| 0| SELECT STATEMENT            |                      |    |     |422(100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID|CUSTOMERS             |335 |35025|422(0)  |00:00:01|
|*2|   INDEX RANGE SCAN          |CUSTOMERS_LAST_NAME_IDX|2335|     |7(0)    |00:00:01|
-------------------------------------------------------------------------------------
```

The preceding plan shows that the cost of using the index on `customers` is higher than the cost of a full table scan. Thus, using an index does not necessarily improve performance. The index clustering factor is a measure of whether an index scan is more effective than a full table scan.

10.2.3.3 Effect of Index Clustering Factor on Cost: Example

This example illustrates how the index clustering factor can influence the cost of table access.

Consider the following scenario:

- A table contains 9 rows that are stored in 3 data blocks.
- The `col1` column currently stores the values `A`, `B`, and `C`.
- A nonunique index named `col1_idx` exists on `col1` for this table.

Example 10-4 Collocated Data

Assume that the rows are stored in the data blocks as follows:

```
Block 1       Block 2        Block 3
-------       -------        -------
A  A  A       B  B  B        C  C  C
```

In this example, the index clustering factor for `col1_idx` is low. The rows that have the same indexed column values for `col1` are in the same data blocks in the table. Thus, the cost of using an index range scan to return all rows with value `A` is low because only one block in the table must be read.

Example 10-5 Scattered Data

Assume that the same rows are scattered across the data blocks as follows:

```
Block 1       Block 2        Block 3
-------       -------        -------
A  B  C       A  C  B        B  A  C
```

In this example, the index clustering factor for `col1_idx` is higher. The database must read all three blocks in the table to retrieve all rows with the value `A` in `col1`.

See Also:

Oracle Database Reference for a description of the `DBA_INDEXES` view

10.2.4 System Statistics

The **system statistics** describe hardware characteristics such as I/O and CPU performance and utilization.

System statistics enable the query optimizer to more accurately estimate I/O and CPU costs when choosing execution plans. The database does not invalidate previously parsed SQL statements when updating system statistics. The database parses all new SQL statements using new statistics.

See Also:

- "Gathering System Statistics Manually"
- *Oracle Database Reference*

10.2.5 User-Defined Optimizer Statistics

The **extensible optimizer** enables authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions.

The optimizer cost model is extended to integrate information supplied by the user to assess CPU and the I/O cost. Statistics types act as interfaces for user-defined functions that influence the choice of execution plan. However, to use a statistics type, the optimizer requires a mechanism to bind the type to a database object such as a column, standalone function, object type, index, indextype, or package. The SQL statement `ASSOCIATE STATISTICS` allows this binding to occur.

Functions for user-defined statistics are relevant for columns that use both standard SQL data types and object types, and for domain indexes. When you associate a statistics type with a column or domain index, the database calls the statistics collection method in the statistics type whenever `DBMS_STATS` gathers statistics.

See Also:

"Gathering Schema and Table Statistics"

10.3 How the Database Gathers Optimizer Statistics

Oracle Database provides several mechanisms to gather statistics.

10.3.1 DBMS_STATS Package

The `DBMS_STATS` PL/SQL package collects and manages optimizer statistics.

This package enables you to control what and how statistics are collected, including the degree of parallelism, sampling methods, and granularity of statistics collection in partitioned tables.

Note:

Do not use the `COMPUTE` and `ESTIMATE` clauses of the `ANALYZE` statement to collect optimizer statistics. These clauses have been deprecated. Instead, use `DBMS_STATS`.

Statistics gathered with the `DBMS_STATS` package are required for the creation of accurate execution plans. For example, table statistics gathered by `DBMS_STATS` include the number of rows, number of blocks, and average row length.

By default, Oracle Database uses automatic optimizer statistics collection. In this case, the database automatically runs `DBMS_STATS` to collect optimizer statistics for all schema objects for which statistics are missing or stale. The process eliminates many manual tasks associated with managing the optimizer, and significantly reduces the risks of generating suboptimal execution plans because of missing or stale statistics. You can also update and manage optimizer statistics by manually executing `DBMS_STATS`.

Oracle Database 19c introduces high-frequency automatic optimizer statistics collection. This lightweight task periodically gathers statistics for stale objects. The default interval is 15 minutes. In contrast to the automated statistics collection job, the high-frequency task does not perform actions such as purging statistics for non-existent objects or invoking Optimizer Statistics Advisor. You can set preferences for the high-frequency task using the `DBMS_STATS.SET_GLOBAL_PREFS` procedure, and view metadata using `DBA_AUTO_STAT_EXECUTIONS`.

See Also:

- "Configuring Automatic Optimizer Statistics Collection"
- "Gathering Optimizer Statistics Manually"
- *Oracle Database Administrator's Guide* to learn more about automated maintenance tasks
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_STATS`

10.3.2 Supplemental Dynamic Statistics

By default, when optimizer statistics are missing, stale, or insufficient, the database automatically gathers **dynamic statistics** during a parse. The database uses **recursive SQL** to scan a small random sample of table blocks.

Note:

Dynamic statistics *augment* statistics rather than providing an alternative to them.

Dynamic statistics supplement optimizer statistics such as table and index block counts, table and join cardinalities (estimated number of rows), join column statistics, and `GROUP BY` statistics. This information helps the optimizer improve plans by making better estimates for predicate cardinality.

Dynamic statistics are beneficial in the following situations:

- An execution plan is suboptimal because of complex predicates.
- The sampling time is a small fraction of total execution time for the query.
- The query executes many times so that the sampling time is amortized.

10.3.3 Online Statistics Gathering

In some circumstances, DDL and DML operations automatically trigger online statistics gathering.

10.3.3.1 Online Statistics Gathering for Bulk Loads

The database can gather table statistics automatically during the following types of bulk loads: `INSERT INTO ... SELECT` using a direct path insert, and `CREATE TABLE AS SELECT`.

By default, a parallel insert uses a direct path insert. You can force a direct path insert by using the `/*+APPEND*/` hint.

See Also:

Oracle Database Data Warehousing Guide to learn more about bulk loads

10.3.3.1.1 Purpose of Online Statistics Gathering for Bulk Loads

Data warehouse applications typically load large amounts of data into the database. For example, a sales data warehouse might load data every day, week, or month.

In releases earlier than Oracle Database 12c, the best practice was to gather statistics manually after a bulk load. However, many applications did not gather statistics after the load because of negligence or because they waited for the maintenance window to initiate collection. Missing statistics are the leading cause of suboptimal execution plans.

Automatic statistics gathering during bulk loads has the following benefits:

- Improved performance

  Gathering statistics during the load avoids an additional table scan to gather table statistics.
- Improved manageability

  No user intervention is required to gather statistics after a bulk load.

10.3.3.1.2 Global Statistics During Inserts into Partitioned Tables

When inserting rows into a partitioned table, the database gathers global statistics during the insert.

For example, if `sales` is a partitioned table, and if you run `INSERT INTO sales SELECT`, then the database gathers global statistics. However, the database does not gather partition-level statistics.

Assume a different case in which you use partition-extended syntax to insert rows into a specific partition or subpartition. The database gathers statistics on the partition during the insert. However, the database does not gather global statistics.

Assume that you run `INSERT INTO sales PARTITION (sales_q4_2000) SELECT`. The database gathers statistics during the insert. If the `INCREMENTAL` preference is enabled for `sales`, then the database also gathers a synopsis for `sales_q4_2000`. Statistics are immediately available after the insert. However, if you roll back the transaction, then the database automatically deletes statistics gathered during the bulk load.

See Also:

- "Considerations for Incremental Statistics Maintenance"
- *Oracle Database SQL Language Reference* for `INSERT` syntax and semantics

10.3.3.1.3 Histogram Creation After Bulk Loads

After gathering online statistics, the database does not automatically create histograms.

If histograms are required, then after the bulk load Oracle recommends running `DBMS_STATS.GATHER_TABLE_STATS` with `options=>GATHER AUTO`. For example, the following program gathers statistics for the `myt` table:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS(user, 'MYT', options=>'GATHER AUTO');
```

The preceding PL/SQL program only gathers missing or stale statistics. The database does not gather table and basic column statistics collected during the bulk load.

Note:

You can set the table preference `options` to `GATHER AUTO` on the tables that you plan to bulk load. In this way, you need not explicitly set the `options` parameter when running `GATHER_TABLE_STATS`.

See Also:

- "Gathering Schema and Table Statistics"
- *Oracle Database Data Warehousing Guide* to learn more about bulk loads

10.3.3.1.4 Restrictions for Online Statistics Gathering for Bulk Loads

In certain cases, bulk loads do not automatically gather optimizer statistics.

Specifically, bulk loads do *not* gather statistics automatically when any of the following conditions applies to the target table, partition, or subpartition:

- The object contains data. Bulk loads *only* gather online statistics automatically when the object is empty.
- It is in an Oracle-owned schema such as `SYS`.
- It is one of the following types of tables: nested table, index-organized table (IOT), external table, or global temporary table defined as `ON COMMIT DELETE ROWS`.

> **Note:**
>
> The database *does* gather online statistics automatically for the *internal* partitions of a hybrid partitioned table.

- It has a `PUBLISH` preference set to `FALSE`.
- Its statistics are locked.
- It is loaded using a multitable `INSERT` statement.

> **See Also:**
>
> - "Gathering Schema and Table Statistics"
> - *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS.SET_TABLE_PREFS`

10.3.3.1.5 User Interface for Online Statistics Gathering for Bulk Loads

By default, the database gathers statistics during bulk loads.

You can enable the feature at the statement level by using the `GATHER_OPTIMIZER_STATISTICS` hint. You can disable the feature at the statement level by using the `NO_GATHER_OPTIMIZER_STATISTICS` hint. For example, the following statement disables online statistics gathering for bulk loads:

```
CREATE TABLE employees2 AS
  SELECT /*+NO_GATHER_OPTIMIZER_STATISTICS*/ * FROM employees
```

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about the `GATHER_OPTIMIZER_STATISTICS` and `NO_GATHER_OPTIMIZER_STATISTICS` hints

10.3.3.2 Online Statistics Gathering for Partition Maintenance Operations

Oracle Database provides analogous support for online statistics during specific partition maintenance operations.

For `MOVE`, `COALESCE`, and `MERGE`, the database maintains global and partition-level statistics as follows:

- If the partition uses either incremental or non-incremental statistics, then the database makes a direct update to the `BLOCKS` value in the global table statistics. Note that this update is not a statistics gathering operation.
- The database generates fresh statistics for the resulting partition. If incremental statistics are enabled, then the database maintains partition synopses.

For `TRUNCATE` or `DROP PARTITION`, the database updates the `BLOCKS` and `NUM_ROWS` values in the global table statistics. The update does not require a gathering statistics operation. The statistics update occurs when either incremental or non-incremental statistics are used.

Note:

The database does not maintain partition-level statistics for maintenance operations that have multiple destination segments.

See Also:

Oracle Database VLDB and Partitioning Guide to learn more about partition maintenance operations

10.3.3.3 Real-Time Statistics

Oracle Database can automatically gather real-time statistics during conventional DML operations.

See Also:

Oracle Database Licensing Information User Manual for details on which features are supported for different editions and services

10.3.3.3.1 Purpose of Real-Time Statistics

Online statistics, whether for bulk loads or conventional DML, aim to reduce the possibility of the optimizer being misled by stale statistics.

Oracle Database 12c introduced online statistics gathering for `CREATE TABLE AS SELECT` statements and direct-path inserts. Oracle Database 19c introduces real-time statistics, which extend online support to conventional DML statements. Because statistics can go stale between `DBMS_STATS` jobs, real-time statistics help the optimizer generate more optimal plans.

Whereas bulk load operations gather all necessary statistics, real-time statistics *augment* rather than replace traditional statistics. For this reason, you must continue to gather statistics regularly using `DBMS_STATS`, preferably using the AutoTask job.

10.3.3.3.2 How Real-Time Statistics Work

Oracle Database dynamically computes values for the most essential statistics during DML operations.

Consider a scenario in which a transaction is currently adding tens of thousands of rows to the `oe.orders` table. Real-time statistics records important changes in statistics, such as maximum column values. This enables the optimizer to obtain more accurate cost estimates.

Existing cursors are not marked invalid when real-time statistics values change.

10.3.3.3.2.1 Regression Models for Real-Time Statistics

As of release 21c, Oracle Database automatically builds regression models to predict the number of distinct values (NDV) for statistics on volatile tables. The use of models enables the optimizer to produce accurate estimates of NDV at low cost.

Note:

The time required to build a regression model may vary. The first step in the process is to model how a column's NDV is seen to change over time. This relies on the information about NDV changes that is derived from the statistics history. If the information immediately available is not sufficient, then the build of the model remains pending until enough historical information has been collected.

Using DBMS_STATS to Delete, Export, and Import Regression Models

Regression models are built automatically by the database as needed and do not require intervention by the DBA. However, you can use `DBMS_STATS` to delete, import, or export regression models. The default `stat_category` includes the default parameter value `MODELS` along with the previously supported values `OBJECT_STATS`, `SYNOPSES` and `REALTIME_STATS`. These are the relevant APIs:

```
DBMS_STATS.DELETE_*_STATS
DBMS_STATS_EXPORT_*_STATS
DBMS_STATS.IMPORT_*_STATS
```

Dictionary Views for Examining Real-Time Statistics Models

As of Oracle Database 21c, these new dictionary views are available for examining saved real-time statistics models.

- `ALL_TAB_COL_STAT_MODELS`

- `DBA_TAB_COL_STAT_MODELS`
- `USER_TAB_COL_STAT_MODELS`

See Also:

- DBMS_STATS in the *Oracle Database PL/SQL Packages and Types Reference*.
- Oracle Database Reference for descriptions of the dictionary views listed above.

10.3.3.3.3 User Interface for Real-Time Statistics

You can use manage and access real-time statistics through PL/SQL packages, data dictionary views, and hints.

OPTIMIZER_REAL_TIME_STATISTICS Initialization Parameter

When the `OPTIMIZER_REAL_TIME_STATISTICS` initialization parameter is set to `TRUE`, Oracle Database automatically gathers real-time statistics during conventional DML operations. The default setting is `FALSE`, which means real-time statistics are disabled.

DBMS_STATS

By default, `DBMS_STATS` subprograms include real-time statistics. You can also specify parameters to include only these statistics.

Table 10-3 Subprograms for Real-Time Statistics

Subprogram	Description
`EXPORT_TABLE_STATS` and `EXPORT_SCHEMA_STATS`	These subprograms enable you to export statistics. By default, the `stat_category` parameter includes real-time statistics. The `REALTIME_STATS` value specifies only real-time statistics.
`IMPORT_TABLE_STATS` and `IMPORT_SCHEMA_STATS`	These subprograms enable you to import statistics. By default, the `stat_category` parameter includes real-time statistics. The `REALTIME_STATS` value specifies only real-time statistics.
`DELETE_TABLE_STATS` and `DELETE_SCHEMA_STATS`	These subprograms enable you to delete statistics. By default, the `stat_category` parameter includes real-time statistics. The `REALTIME_STATS` value specifies only real-time statistics.
`DIFF_TABLE_STATS_IN_STATTAB`	This function compares table statistics from two sources. The statistics always include real-time statistics.
`DIFF_TABLE_STATS_IN_HISTORY`	This function compares statistics for a table as of two specified timestamps. The statistics always include real-time statistics.

Views

Real-time statistics can be viewed in the data dictionary table statistics views such as USER_TAB_STATISTICS and USER_TAB_COL_STATISTICS when the NOTES columns is STATS_ON_CONVENTIONAL_DML, as described in the table below.

The `DBA_*` views have `ALL_*` and `USER_*` versions.

Table 10-4 Views for Real-Time Statistics

View	Description
`DBA_TAB_COL_STATISTICS`	This view displays column statistics and histogram information extracted from `DBA_TAB_COLUMNS`. Real-time statistics are indicated by `STATS_ON_CONVENTIONAL_DML` in the `NOTES` column.
`DBA_TAB_STATISTICS`	This view displays optimizer statistics for the tables accessible to the current user. Real-time statistics are indicated by `STATS_ON_CONVENTIONAL_DML` in the `NOTES` column.

Hints

The `NO_GATHER_OPTIMIZER_STATISTICS` hint prevents the collection of real-time statistics.

See Also:

- "Importing and Exporting Optimizer Statistics"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_STATS` subprograms
- *Oracle Database Reference* to learn about the `ALL_TAB_COL_STATISTICS` view
- *Oracle Database Licensing Information User Manual* for details on whether the real-time statistics feature is supported for different editions and services

10.3.3.3.4 Real-Time Statistics: Example

In this example, a conventional `INSERT` statement triggers the collection of real-time statistics.

This example assumes that the `sh` user has been granted the DBA role, and you have logged in to the database as `sh`. You perform the following steps:

1. Gather statistics for the `sales` table:

```
BEGIN
 DBMS_STATS.GATHER_TABLE_STATS('SH', 'SALES');
```

```
END;
/
```

2. Query the column-level statistics for `sales`:

```
SET PAGESIZE 5000
SET LINESIZE 200
COL COLUMN_NAME FORMAT a13
COL LOW_VALUE FORMAT a14
COL HIGH_VALUE FORMAT a14
COL NOTES FORMAT a5
COL PARTITION_NAME FORMAT a13

SELECT COLUMN_NAME, LOW_VALUE, HIGH_VALUE, SAMPLE_SIZE, NOTES
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME = 'SALES'
ORDER BY 1, 5;
```

   The `Notes` fields are blank, meaning that real-time statistics have not been gathered:

```
COLUMN_NAME   LOW_VALUE      HIGH_VALUE     SAMPLE_SIZE NOTES
------------- -------------- -------------- ----------- -----
AMOUNT_SOLD   C10729         C2125349              5594
CHANNEL_ID    C103           C10A                918843
CUST_ID       C103           C30B0B                5595
PROD_ID       C10E           C20231                5593
PROMO_ID      C122           C20A64              918843
QUANTITY_SOLD C102           C102                  5593
TIME_ID       77C60101010101 78650C1F010101        5593

7 rows selected.
```

3. Query the table-level statistics for `sales`:

```
SELECT NVL(PARTITION_NAME, 'GLOBAL') PARTITION_NAME, NUM_ROWS, BLOCKS,
NOTES
FROM   USER_TAB_STATISTICS
WHERE  TABLE_NAME = 'SALES'
ORDER BY 1, 4;
```

   The `Notes` fields are blank, meaning that real-time statistics have not been gathered:

```
PARTITION_NAM   NUM_ROWS     BLOCKS NOTES
------------- ---------- ---------- -----
GLOBAL            918843       3315
SALES_1995             0          0
SALES_1996             0          0
SALES_H1_1997          0          0
SALES_H2_1997          0          0
SALES_Q1_1998      43687        162
SALES_Q1_1999      64186        227
SALES_Q1_2000      62197        222
SALES_Q1_2001      60608        222
```

```
SALES_Q1_2002          0          0
SALES_Q1_2003          0          0
SALES_Q2_1998      35758        132
SALES_Q2_1999      54233        187
SALES_Q2_2000      55515        197
SALES_Q2_2001      63292        227
SALES_Q2_2002          0          0
SALES_Q2_2003          0          0
SALES_Q3_1998      50515        182
SALES_Q3_1999      67138        232
SALES_Q3_2000      58950        212
SALES_Q3_2001      65769        242
SALES_Q3_2002          0          0
SALES_Q3_2003          0          0
SALES_Q4_1998      48874        192
SALES_Q4_1999      62388        217
SALES_Q4_2000      55984        202
SALES_Q4_2001      69749        260
SALES_Q4_2002          0          0
SALES_Q4_2003          0          0

29 rows selected.
```

4. Load 918,843 rows into `sales` by using a conventional `INSERT` statement:

```
INSERT INTO sales(prod_id, cust_id, time_id, channel_id, promo_id,
                  quantity_sold, amount_sold)
  SELECT prod_id, cust_id, time_id, channel_id, promo_id,
         quantity_sold * 2, amount_sold * 2
  FROM   sales;
COMMIT;
```

5. Obtain the execution plan from the cursor:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format=>'TYPICAL'));
```

The plan shows `LOAD TABLE CONVENTIONAL` in Step 1 and `OPTIMIZER STATISTICS GATHERING` in Step 2, which means that the database gathered real-time statistics during the conventional insert:

```
----------------------------------------------------------------------------------
----
|Id| Operation                        | Name|Rows|Bytes|Cost (%CPU)|Time| Pstart|
Pstop|
----------------------------------------------------------------------------------
----
| 0| INSERT STATEMENT                 |     |    |     |910 (100)|        |
|     |
| 1|  LOAD TABLE CONVENTIONAL         |SALES|    |     |          |        |
|     |
| 2|   OPTIMIZER STATISTICS GATHERING |     |918K|  25M|910   (2)|00:00:01|
|     |
| 3|    PARTITION RANGE ALL           |     |918K|  25M|910   (2)|00:00:01|   1 |
28 |
| 4|     TABLE ACCESS FULL           |SALES|918K|  25M|910   (2)|00:00:01|   1 |  28 |
---------------------------------------------------------------------------------------
```

6. For the purposes of testing, force the database to write optimizer statistics to the data dictionary immediately.

```
EXEC DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO;
```

7. Query the column-level statistics for `sales`.

```
SET PAGESIZE 5000
SET LINESIZE 200
COL COLUMN_NAME FORMAT a13
COL LOW_VALUE FORMAT a14
COL HIGH_VALUE FORMAT a14
COL NOTES FORMAT a25
COL PARTITION_NAME FORMAT a13

SELECT COLUMN_NAME, LOW_VALUE, HIGH_VALUE, SAMPLE_SIZE, NOTES
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME = 'SALES'
ORDER BY 1, 5;
```

Now the `Notes` fields show `STATS_ON_CONVENTIONAL_DML`, meaning that the database gathered real-time statistics during the insert:

```
COLUMN_NAME   LOW_VALUE      HIGH_VALUE     SAMPLE_SIZE NOTES
------------- -------------- -------------- ----------- -------------------------
AMOUNT_SOLD   C10729         C224422D              9073 STATS_ON_CONVENTIONAL_DML
AMOUNT_SOLD   C10729         C2125349              5702
CHANNEL_ID    C103           C10A                  9073 STATS_ON_CONVENTIONAL_DML
CHANNEL_ID    C103           C10A                918843
CUST_ID       C103           C30B0B                9073 STATS_ON_CONVENTIONAL_DML
CUST_ID       C103           C30B0B                5702
PROD_ID       C10E           C20231                9073 STATS_ON_CONVENTIONAL_DML
PROD_ID       C10E           C20231                5701
PROMO_ID      C122           C20A64                9073 STATS_ON_CONVENTIONAL_DML
PROMO_ID      C122           C20A64              918843
QUANTITY_SOLD C102           C103                  9073 STATS_ON_CONVENTIONAL_DML
QUANTITY_SOLD C102           C102                  5701
TIME_ID       77C60101010101 78650C1F010101        9073 STATS_ON_CONVENTIONAL_DML
TIME_ID       77C60101010101 78650C1F010101        5701
```

The sample size is 9073, which is roughly 1% of the 918,843 rows inserted. In `QUANTITY_SOLD` and `AMOUNT_SOLD`, the high and low values combine the manually gathered statistics and the real-time statistics.

8. Query the table-level statistics for `sales`.

```
SELECT NVL(PARTITION_NAME, 'GLOBAL') PARTITION_NAME, NUM_ROWS, BLOCKS,
NOTES
FROM   USER_TAB_STATISTICS
```

```
WHERE  TABLE_NAME = 'SALES'
ORDER BY 1, 4;
```

The `Notes` field shows that real-time statistics have been gathered at the global level, showing the number of rows as 1,837,686:

```
PARTITION_NAM   NUM_ROWS     BLOCKS NOTES
------------- ---------- ---------- -------------------------
GLOBAL           1837686       3315 STATS_ON_CONVENTIONAL_DML
GLOBAL            918843       3315
SALES_1995             0          0
SALES_1996             0          0
SALES_H1_1997          0          0
SALES_H2_1997          0          0
SALES_Q1_1998      43687        162
SALES_Q1_1999      64186        227
SALES_Q1_2000      62197        222
SALES_Q1_2001      60608        222
SALES_Q1_2002          0          0
SALES_Q1_2003          0          0
SALES_Q2_1998      35758        132
SALES_Q2_1999      54233        187
SALES_Q2_2000      55515        197
SALES_Q2_2001      63292        227
SALES_Q2_2002          0          0
SALES_Q2_2003          0          0
SALES_Q3_1998      50515        182
SALES_Q3_1999      67138        232
SALES_Q3_2000      58950        212
SALES_Q3_2001      65769        242
SALES_Q3_2002          0          0
SALES_Q3_2003          0          0
SALES_Q4_1998      48874        192
SALES_Q4_1999      62388        217
SALES_Q4_2000      55984        202
SALES_Q4_2001      69749        260
SALES_Q4_2002          0          0
SALES_Q4_2003          0          0
```

9. Query the `quantity_sold` column:

```
SELECT COUNT(*) FROM sales WHERE quantity_sold > 50;
```

10. Obtain the execution plan from the cursor:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format=>'TYPICAL'));
```

The `Note` field shows that the query used the real-time statistics.

```
Plan hash value: 3519235612

---------------------------------------------------------------------
----
```

```
|Id| Operation           |Name|Rows|Bytes|Cost (%CPU)|Time|Pstart|Pstop|
------------------------------------------------------------------------
| 0| SELECT STATEMENT     |       |    |   |921 (100)|          |   |    |
| 1|  SORT AGGREGATE      |       |   1|  3|          |          |   |    |
| 2|   PARTITION RANGE ALL|       |  11|  3|921   (3)|00:00:01| 1 | 28 |
|*3|    TABLE ACCESS FULL | SALES |  11|  3|921   (3)|00:00:01| 1 | 28 |
------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - filter("QUANTITY_SOLD">50)

Note
-----
   - dynamic statistics used: stats for conventional DML
```

See Also:

Oracle Database Reference to learn about `USER_TAB_COL_STATISTICS`and `USER_TAB_STATISTICS`.

10.4 When the Database Gathers Optimizer Statistics

The database collects optimizer statistics at various times and from various sources.

10.4.1 Sources for Optimizer Statistics

The optimizer uses several different sources for optimizer statistics.

The sources are as follows:

- `DBMS_STATS` execution, automatic or manual

  This PL/SQL package is the primary means of gathering optimizer statistics.

- SQL compilation

  During SQL compilation, the database can augment the statistics previously gathered by `DBMS_STATS`. In this stage, the database runs additional queries to obtain more accurate information on how many rows in the tables satisfy the `WHERE` clause predicates in the SQL statement.

- SQL execution

  During execution, the database can further augment previously gathered statistics. In this stage, Oracle Database collects the number of rows produced by every row source during the execution of a SQL statement. At the end of execution, the optimizer determines whether the estimated number of rows is inaccurate enough to warrant reparsing at the next statement execution. If the cursor is marked for reparsing, then the optimizer uses actual row counts from the previous execution instead of estimates.

- SQL profiles

A SQL profile is a collection of auxiliary statistics on a query. The profile stores these supplemental statistics in the data dictionary. The optimizer uses SQL profiles during optimization to determine the most optimal plan.

The database stores optimizer statistics in the data dictionary and updates or replaces them as needed. You can query statistics in data dictionary views.

See Also:

- "When the Database Samples Data"
- "About SQL Profiles"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure

10.4.2 SQL Plan Directives

A **SQL plan directive** is additional information and instructions that the optimizer can use to generate a more optimal plan.

The directive is a “note to self” by the optimizer that it is misestimating cardinalities of certain types of predicates, and also a reminder to `DBMS_STATS` to gather statistics needed to correct the misestimates in the future. For example, when joining two tables that have a data skew in their join columns, a SQL plan directive can direct the optimizer to use dynamic statistics to obtain a more accurate join cardinality estimate.

10.4.2.1 When the Database Creates SQL Plan Directives

The database creates SQL plan directives automatically based on information learned during automatic reoptimization. If a cardinality misestimate occurs during SQL execution, then the database creates SQL plan directives.

For each new directive, the `DBA_SQL_PLAN_DIRECTIVES.STATE` column shows the value `USABLE`. This value indicates that the database can use the directive to correct misestimates.

The optimizer defines a SQL plan directive on a query expression, for example, filter predicates on two columns being used together. A directive is not tied to a specific SQL statement or SQL ID. For this reason, the optimizer can use directives for statements that are not identical. For example, directives can help the optimizer with queries that use similar patterns, such as queries that are identical except for a select list item.

The Notes section of the execution plan indicates the number of SQL plan directives used for a statement. Obtain more information about the directives by querying the `DBA_SQL_PLAN_DIRECTIVES` and `DBA_SQL_PLAN_DIR_OBJECTS` views.

See Also:

Oracle Database Reference to learn more about `DBA_SQL_PLAN_DIRECTIVES`

10.4.2.2 How the Database Uses SQL Plan Directives

When compiling a SQL statement, if the optimizer sees a directive, then it obeys the directive by gathering additional information.

The optimizer uses directives in the following ways:

- Dynamic statistics

  The optimizer uses dynamic statistics whenever it does not have sufficient statistics corresponding to the directive. For example, the cardinality estimates for a query whose predicate contains a specific pair of columns may be significantly wrong. A SQL plan directive indicates that the whenever a query that contains these columns is parsed, the optimizer needs to use dynamic sampling to avoid a serious cardinality misestimate.

  Dynamic statistics have some performance overhead. Every time the optimizer hard parses a query to which a dynamic statistics directive applies, the database must perform the extra sampling.

  Starting in Oracle Database 12c Release 2 (12.2), the database writes statistics from adaptive dynamic sampling to the SQL plan directives store, making them available to other queries.

- Column groups

  The optimizer examines the query corresponding to the directive. If there is a missing column group, and if the `DBMS_STATS` preference `AUTO_STAT_EXTENSIONS` is set to `ON` (the default is `OFF`) for the affected table, then the optimizer automatically creates this column group the next time `DBMS_STATS` gathers statistics on the table. Otherwise, the optimizer does not automatically create the column group.

  If a column group exists, then the next time this statement executes, the optimizer uses the column group statistics in place of the SQL plan directive when possible (equality predicates, `GROUP BY`, and so on). In subsequent executions, the optimizer may create additional SQL plan directives to address other problems in the plan, such as join or `GROUP BY` cardinality misestimates.

> **Note:**
>
> Currently, the optimizer monitors only column groups. The optimizer does not create an extension on expressions.

When the problem that occasioned a directive is solved, either because a better directive exists or because a histogram or extension exists, the `DBA_SQL_PLAN_DIRECTIVES.STATE` value changes from `USABLE` to `SUPERSEDED`. More information about the directive state is exposed in the `DBA_SQL_PLAN_DIRECTIVES.NOTES` column.

See Also:

- "Managing Extended Statistics"
- "About Statistics on Column Groups"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `AUTO_STAT_EXTENSIONS` preference for `DBMS_STATS.SET_TABLE_STATS`

10.4.2.3 SQL Plan Directive Maintenance

The database automatically creates SQL plan directives. You cannot create them manually.

The database initially creates directives in the shared pool. The database periodically writes the directives to the `SYSAUX` tablespace. The database automatically purges any SQL plan directive that is not used after the specified number of weeks (`SPD_RETENTION_WEEKS`), which is 53 by default.

You can manage directives by using the `DBMS_SPD` package. For example, you can:

- Enable and disable SQL plan directives (`ALTER_SQL_PLAN_DIRECTIVE`)
- Change the retention period for SQL plan directives (`SET_PREFS`)
- Export a directive to a staging table (`PACK_STGTAB_DIRECTIVE`)
- Drop a directive (`DROP_SQL_PLAN_DIRECTIVE`)
- Force the database to write directives to disk (`FLUSH_SQL_PLAN_DIRECTIVE`)

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_SPD` package

10.4.2.4 How the Optimizer Uses SQL Plan Directives: Example

This example shows how the database automatically creates and uses SQL plan directives for SQL statements.

Assumptions

You plan to run queries against the `sh` schema, and you have privileges on this schema and on data dictionary and `V$` views.

To see how the database uses a SQL plan directive:

1. Query the `sh.customers` table.

```
SELECT /*+gather_plan_statistics*/ *
FROM   customers
```

```
WHERE  cust_state_province='CA'
AND    country_id='US';
```

The `gather_plan_statistics` hint shows the actual number of rows returned from each operation in the plan. Thus, you can compare the optimizer estimates with the actual number of rows returned.

2. Query the plan for the preceding query.

   The following example shows the execution plan (sample output included):

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  b74nw722wjvy3, child number 0
-------------------------------------
select /*+gather_plan_statistics*/ * from customers where
CUST_STATE_PROVINCE='CA' and country_id='US'

Plan hash value: 1683234692
---------------------------------------------------------------------------------------
| Id| Operation         | Name      |Starts|E-Rows|A-Rows| Time        | Buffers| Reads |
---------------------------------------------------------------------------------------
| 0 | SELECT STATEMENT  |           |  1 |      |     29 |00:00:00.01 |      17 |    14 |
|*1 |  TABLE ACCESS FULL| CUSTOMERS |  1 |    8 |     29 |00:00:00.01 |      17 |    14 |
---------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_STATE_PROVINCE"='CA' AND "COUNTRY_ID"='US'))
```

   The actual number of rows (`A-Rows`) returned by each operation in the plan varies greatly from the estimates (`E-Rows`). This statement is a candidate for automatic reoptimization.

3. Check whether the `customers` query can be reoptimized.

   The following statement queries the `V$SQL.IS_REOPTIMIZABLE` value (sample output included):

```
SELECT SQL_ID, CHILD_NUMBER, SQL_TEXT, IS_REOPTIMIZABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE 'SELECT /*+gather_plan_statistics*/%';

SQL_ID        CHILD_NUMBER SQL_TEXT    I
------------- ------------ ----------- -
b74nw722wjvy3            0 select /*+g Y
                           ather_plan_
                           statistics*
                           / * from cu
                           stomers whe
                           re CUST_STA
                           TE_PROVINCE
                           ='CA' and c
```

```
                                                ountry_id='
                                                US'
```

The `IS_REOPTIMIZABLE` column is marked `Y`, so the database will perform a hard parse of the `customers` query on the next execution. The optimizer uses the execution statistics from this initial execution to determine the plan. The database persists the information learned from reoptimization as a SQL plan directive.

4. Display the directives for the `sh` schema.

   The following example uses `DBMS_SPD` to write the SQL plan directives to disk, and then shows the directives for the `sh` schema only:

```
EXEC DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE;

SELECT TO_CHAR(d.DIRECTIVE_ID) dir_id, o.OWNER AS "OWN", o.OBJECT_NAME AS 
"OBJECT",
       o.SUBOBJECT_NAME col_name, o.OBJECT_TYPE, d.TYPE, d.STATE, d.REASON
FROM   DBA_SQL_PLAN_DIRECTIVES d, DBA_SQL_PLAN_DIR_OBJECTS o
WHERE  d.DIRECTIVE_ID=o.DIRECTIVE_ID
AND    o.OWNER IN ('SH')
ORDER BY 1,2,3,4,5;

DIR_ID              OW OBJECT    COL_NAME   OBJECT TYPE          STATE  REASON
------------------- -- --------- ---------- ------ ------------- ------ 
------------
1484026771529551585 SH CUSTOMERS COUNTRY_ID COLUMN DYNAMIC_SAMPL USABLE SINGLE 
TABLE 
                                                                        CARDINALITY
                                                                        MISESTIMATE
1484026771529551585 SH CUSTOMERS CUST_STATE COLUMN DYNAMIC_SAMPL USABLE SINGLE 
TABLE 
                                 _PROVINCE                              CARDINALITY
                                                                        
MISESTIMATE
1484026771529551585 SH CUSTOMERS            TABLE  DYNAMIC_SAMPL USABLE SINGLE 
TABLE 
                                                                        CARDINALITY
                                                                        MISESTIMATE
```

Initially, the database stores SQL plan directives in memory, and then writes them to disk every 15 minutes. Thus, the preceding example calls `DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE` to force the database to write the directives to the `SYSAUX` tablespace.

Monitor directives using the views `DBA_SQL_PLAN_DIRECTIVES` and `DBA_SQL_PLAN_DIR_OBJECTS`. Three entries appear in the views, one for the `customers` table itself, and one for each of the correlated columns. Because the `customers` query has the `IS_REOPTIMIZABLE` value of `Y`, if you reexecute the statement, then the database will hard parse it again, and then generate a plan based on the previous execution statistics.

5. Query the `customers` table again.

For example, enter the following statement:

```
SELECT /*+gather_plan_statistics*/ *
FROM   customers
WHERE  cust_state_province='CA'
AND    country_id='US';
```

6. Query the plan in the cursor.

   The following example shows the execution plan (sample output included):

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  b74nw722wjvy3, child number 1
-------------------------------------
select /*+gather_plan_statistics*/ * from customers where
CUST_STATE_PROVINCE='CA' and country_id='US'

Plan hash value: 1683234692
------------------------------------------------------------------------
|Id| Operation         |Name     |Start|E-Rows|A-Rows|   A-Time   |Buffers|
------------------------------------------------------------------------
| 0| SELECT STATEMENT  |         |   1|      |    29|00:00:00.01|     17|
|*1|  TABLE ACCESS FULL|CUSTOMERS|   1|    29|    29|00:00:00.01|     17|
------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_STATE_PROVINCE"='CA' AND "COUNTRY_ID"='US'))

Note
-----
   - cardinality feedback used for this statement
```

   The `Note` section indicates that the database used reoptimization for this statement. The estimated number of rows (`E-Rows`) is now correct. The SQL plan directive has not been used yet.

7. Query the cursors for the `customers` query.

   For example, run the following query (sample output included):

```
SELECT SQL_ID, CHILD_NUMBER, SQL_TEXT, IS_REOPTIMIZABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE 'SELECT /*+gather_plan_statistics*/%';

SQL_ID        CHILD_NUMBER SQL_TEXT    I
------------- ------------ ----------- -
b74nw722wjvy3            0 select /*+g Y
                           ather_plan_
                           statistics*
                           / * from cu
                           stomers whe
                           re CUST_STA
                           TE_PROVINCE
                           ='CA' and c
                           ountry_id='
                           US'

b74nw722wjvy3            1 select /*+g N
                           ather_plan_
                           statistics*
                           / * from cu
                           stomers whe
                           re CUST_STA
                           TE_PROVINCE
                           ='CA' and c
                           ountry_id='
                           US'
```

A new plan exists for the `customers` query, and also a new child cursor.

8. Confirm that a SQL plan directive exists and is usable for other statements.

   For example, run the following query, which is similar but not identical to the original `customers` query (the state is `MA` instead of `CA`):

```
SELECT /*+gather_plan_statistics*/ CUST_EMAIL
FROM   CUSTOMERS
WHERE  CUST_STATE_PROVINCE='MA'
AND    COUNTRY_ID='US';
```

9. Query the plan in the cursor.

   The following statement queries the cursor (sample output included).:

```
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS
LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  3tk6hj3nkcs2u, child number 0
-------------------------------------
Select /*+gather_plan_statistics*/ cust_email From   customers Where
cust_state_province='MA' And    country_id='US'

Plan hash value: 1683234692

--------------------------------------------------------------------
--
|Id | Operation         | Name    |Starts|E-Rows|A-Rows|A-Time|
Buffers|
--------------------------------------------------------------------
--
| 0 | SELECT STATEMENT  |            |  1 |    |  2 |00:00:00.01|
16 |
|*1 |  TABLE ACCESS FULL| CUSTOMERS |  1 | 2 |  2 |00:00:00.01|
16 |
----------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_STATE_PROVINCE"='MA' AND "COUNTRY_ID"='US'))

Note
-----
   - dynamic sampling used for this statement (level=2)
   - 1 Sql Plan Directive used for this statement
```

The `Note` section of the plan shows that the optimizer used the SQL directive for this statement, and also used dynamic statistics.

See Also:

- "Automatic Reoptimization"
- "Managing SQL Plan Directives"
- *Oracle Database Reference* to learn about `DBA_SQL_PLAN_DIRECTIVES`, `V$SQL`, and other database views
- *Oracle Database Reference* to learn about `DBMS_SPD`

10.4.2.5 How the Optimizer Uses Extensions and SQL Plan Directives: Example

The example shows how the database uses a SQL plan directive until the optimizer verifies that an extension exists and the statistics are applicable.

At this point, the directive changes its status to `SUPERSEDED`. Subsequent compilations use the statistics instead of the directive.

Assumptions

This example assumes you have already followed the steps in "How the Optimizer Uses SQL Plan Directives: Example".

To see how the optimizer uses an extension and SQL plan directive:

1. Gather statistics for the `sh.customers` table.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS('SH','CUSTOMERS');
   END;
   /
   ```

2. Check whether an extension exists on the `customers` table.

For example, execute the following query (sample output included):

```
SELECT TABLE_NAME, EXTENSION_NAME, EXTENSION
FROM   DBA_STAT_EXTENSIONS
WHERE  OWNER='SH'
AND    TABLE_NAME='CUSTOMERS';

TABLE_NAM EXTENSION_NAME                 EXTENSION
--------- ------------------------------ -----------------------
CUSTOMERS SYS_STU#S#WF25Z#QAHIHE#MOFFMM_ ("CUST_STATE_PROVINCE",
                                         "COUNTRY_ID")
```

The preceding output indicates that a column group extension exists on the `cust_state_province` and `country_id` columns.

3. Query the state of the SQL plan directive.

   Example 10-6 queries the data dictionary for information about the directive.

   Although column group statistics exist, the directive has a state of `USABLE` because the database has not yet recompiled the statement. During the next compilation, the optimizer verifies that the statistics are applicable. If they are applicable, then the status of the directive changes to `SUPERSEDED`. Subsequent compilations use the statistics instead of the directive.

4. Query the `sh.customers` table.

   ```
   SELECT /*+gather_plan_statistics*/ *
   FROM   customers
   WHERE  cust_state_province='CA'
   AND    country_id='US';
   ```

5. Query the plan in the cursor.

   Example 10-7 shows the execution plan (sample output included).

   The `Note` section shows that the optimizer used the directive and not the extended statistics. During the compilation, the database verified the extended statistics.

6. Query the state of the SQL plan directive.

   Example 10-8 queries the data dictionary for information about the directive.

   The state of the directive, which has changed to `SUPERSEDED`, indicates that the corresponding column or groups have an extension or histogram, or that another SQL plan directive exists that can be used for the directive.

7. Query the `sh.customers` table again, using a slightly different form of the statement.

   For example, run the following query:

   ```
   SELECT /*+gather_plan_statistics*/ /* force reparse */ *
   FROM   customers
   WHERE  cust_state_province='CA'
   AND    country_id='US';
   ```

If the cursor is in the shared SQL area, then the database typically shares the cursor. To force a reparse, this step changes the SQL text slightly by adding a comment.

8. Query the plan in the cursor.

   Example 10-9 shows the execution plan (sample output included).

   The absence of a `Note` shows that the optimizer used the extended statistics instead of the SQL plan directive. If the directive is not used for 53 weeks, then the database automatically purges it.

See Also:

- "Managing SQL Plan Directives"
- *Oracle Database Reference* to learn about `DBA_SQL_PLAN_DIRECTIVES`, `V$SQL`, and other database views
- *Oracle Database Reference* to learn about `DBMS_SPD`

Example 10-6 Display Directives for sh Schema

```
EXEC DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE;

SELECT TO_CHAR(d.DIRECTIVE_ID) dir_id, o.OWNER, o.OBJECT_NAME,
       o.SUBOBJECT_NAME col_name, o.OBJECT_TYPE, d.TYPE, d.STATE, d.REASON
FROM   DBA_SQL_PLAN_DIRECTIVES d, DBA_SQL_PLAN_DIR_OBJECTS o
WHERE  d.DIRECTIVE_ID=o.DIRECTIVE_ID
AND    o.OWNER IN ('SH')
ORDER BY 1,2,3,4,5;

DIR_ID              OWN OBJECT_NA COL_NAME   OBJECT  TYPE             STATE  REASON
------------------- --- --------- ---------- ------- ---------------- ------ ------------
1484026771529551585  SH CUSTOMERS COUNTRY_ID  COLUMN DYNAMIC_SAMPLING USABLE SINGLE TABLE
                                                                             CARDINALITY
                                                                             MISESTIMATE
1484026771529551585  SH CUSTOMERS CUST_STATE_ COLUMN DYNAMIC_SAMPLING USABLE SINGLE TABLE
                                  PROVINCE                                   CARDINALITY
                                                                             MISESTIMATE
1484026771529551585  SH CUSTOMERS              TABLE DYNAMIC_SAMPLING USABLE SINGLE TABLE
                                                                             CARDINALITY
                                                                             MISESTIMATE
```

Example 10-7 Execution Plan

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  b74nw722wjvy3, child number 0
-------------------------------------
select /*+gather_plan_statistics*/ * from customers where
CUST_STATE_PROVINCE='CA' and country_id='US'
```

```
Plan hash value: 1683234692

-------------------------------------------------------------------------------------
---
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time   |
Buffers |
-------------------------------------------------------------------------------------
---
|   0 | SELECT STATEMENT  |           |      1 |        |     29 |00:00:00.01 |
16 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |      1 |     29 |     29 |00:00:00.01 |
16 |
-------------------------------------------------------------------------------------
---

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_STATE_PROVINCE"='CA' AND "COUNTRY_ID"='US'))

Note
-----
   - dynamic sampling used for this statement (level=2)
   - 1 Sql Plan Directive used for this statement
```

Example 10-8 Display Directives for sh Schema

```
EXEC DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE;

SELECT TO_CHAR(d.DIRECTIVE_ID) dir_id, o.OWNER, o.OBJECT_NAME,
       o.SUBOBJECT_NAME col_name, o.OBJECT_TYPE, d.TYPE, d.STATE, d.REASON
FROM   DBA_SQL_PLAN_DIRECTIVES d, DBA_SQL_PLAN_DIR_OBJECTS o
WHERE  d.DIRECTIVE_ID=o.DIRECTIVE_ID
AND    o.OWNER IN ('SH')
ORDER BY 1,2,3,4,5;

DIR_ID              OWN OBJECT_NA  COL_NAME    OBJECT TYPE     STATE      REASON
------------------- --- ---------- ----------- ------ -------- ---------  ------------
1484026771529551585  SH CUSTOMERS  COUNTRY_ID  COLUMN DYNAMIC_ SUPERSEDED SINGLE
TABLE
                                                      SAMPLING            CARDINALITY
                                                                          MISESTIMATE
1484026771529551585  SH CUSTOMERS  CUST_STATE_ COLUMN DYNAMIC_ SUPERSEDED SINGLE
TABLE
                                   PROVINCE           SAMPLING            CARDINALITY
                                                                          MISESTIMATE
1484026771529551585  SH CUSTOMERS               TABLE DYNAMIC_ SUPERSEDED SINGLE TABLE
                                                      SAMPLING            CARDINALITY
                                                                          MISESTIMATE
```

Example 10-9 Execution Plan

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(FORMAT=>'ALLSTATS LAST'));

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID  b74nw722wjvy3, child number 0
-------------------------------------
select /*+gather_plan_statistics*/ * from customers where
CUST_STATE_PROVINCE='CA' and country_id='US'

Plan hash value: 1683234692

-----------------------------------------------------------------------------------------
| Id  | Operation          | Name      | Starts | E-Rows | A-Rows |   A-Time    | Buffers |
-----------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |           |      1 |        |     29 |00:00:00.01 |      17 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |      1 |     29 |     29 |00:00:00.01 |      17 |
-----------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(("CUST_STATE_PROVINCE"='CA' AND "COUNTRY_ID"='US'))

19 rows selected.
```

10.4.3 When the Database Samples Data

Starting in Oracle Database 12c, the optimizer automatically decides whether dynamic statistics are useful and which sample size to use for all SQL statements.

Note:

In earlier releases, dynamic statistics were called *dynamic sampling*.

The primary factor in the decision to use dynamic statistics is whether available statistics are sufficient to generate an optimal plan. If statistics are insufficient, then the optimizer uses dynamic statistics.

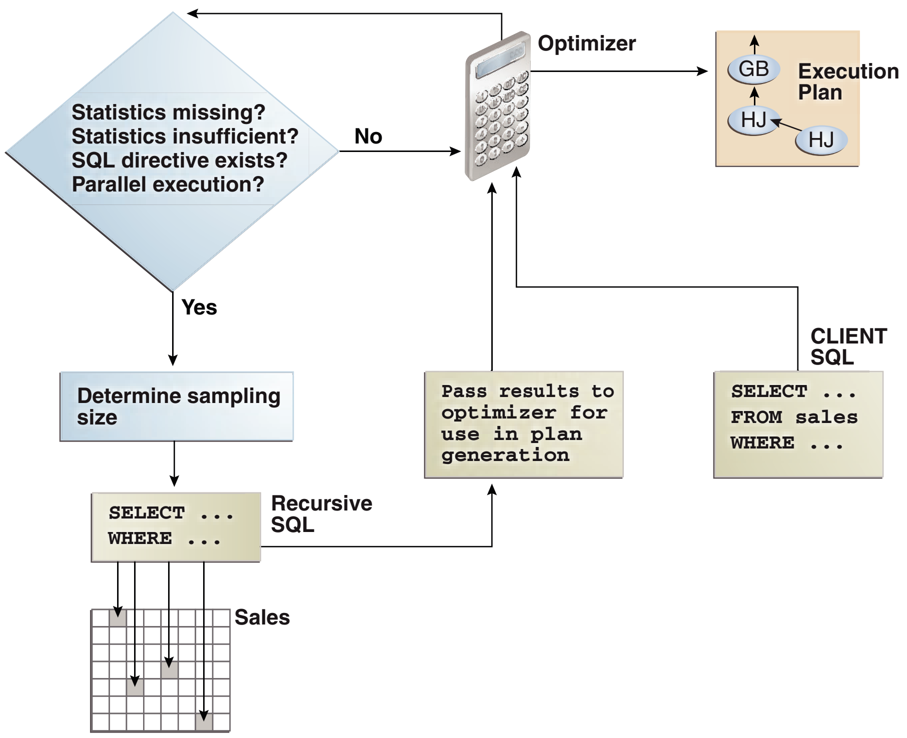
Automatic dynamic statistics are enabled when the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter is not set to `0`. By default, the dynamic statistics level is set to `2`.

In general, the optimizer uses default statistics rather than dynamic statistics to compute statistics needed during optimizations on tables, indexes, and columns. The optimizer decides whether to use dynamic statistics based on several factors, including the following:

- The SQL statement uses parallel execution.
- A SQL plan directive exists.

The following diagram illustrates the process of gathering dynamic statistics.

Figure 10-2 Dynamic Statistics



As shown in Figure 10-2, the optimizer automatically gathers dynamic statistics in the following cases:

- Missing statistics

  When tables in a query have no statistics, the optimizer gathers basic statistics on these tables before optimization. Statistics can be missing because the application creates new objects without a follow-up call to `DBMS_STATS` to gather statistics, or because statistics were locked on an object before statistics were gathered.

  In this case, the statistics are not as high-quality or as complete as the statistics gathered using the `DBMS_STATS` package. This trade-off is made to limit the impact on the compile time of the statement.

- Insufficient statistics

  Statistics can be insufficient whenever the optimizer estimates the selectivity of predicates (filter or join) or the `GROUP BY` clause without taking into account correlation between columns, skew in the column data distribution, statistics on expressions, and so on.

  Extended statistics help the optimizer obtain accurate quality cardinality estimates for complex predicate expressions. The optimizer can use dynamic statistics to compensate for the lack of extended statistics or when it cannot use extended statistics, for example, for non-equality predicates.

Note:

The database does not use dynamic statistics for queries that contain the `AS OF` clause.

See Also:

- "Configuring Options for Dynamic Statistics"
- "About Statistics on Column Groups"
- *Oracle Database Reference* to learn about the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter

10.4.4 How the Database Samples Data

At the beginning of optimization, when deciding whether a table is a candidate for dynamic statistics, the optimizer checks for the existence of persistent SQL plan directives on the table.

For each directive, the optimizer registers a statistics expression that the optimizer computes when determining the cardinality of a predicate involving the table. In Figure 10-2, the database issues a recursive SQL statement to scan a small random sample of the table blocks. The database applies the relevant single-table predicates and joins to estimate predicate cardinalities.

The database persists the results of dynamic statistics as sharable statistics. The database can share the results during the SQL compilation of one query with recompilations of the same query. The database can also reuse the results for queries that have the same patterns.

See Also:

- "Configuring Options for Dynamic Statistics" to learn how to set the dynamic statistics level
- *Oracle Database Reference* for details about the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter

11

Histograms

A **histogram** is a special type of column statistic that provides more detailed information about the data distribution in a table column. A histogram sorts values into "buckets," as you might sort coins into buckets.

Based on the NDV and the distribution of the data, the database chooses the type of histogram to create. (In some cases, when creating a histogram, the database samples an internally predetermined number of rows.) The types of histograms are as follows:

- Frequency histograms and top frequency histograms
- Height-Balanced histograms (legacy)
- Hybrid histograms

11.1 Purpose of Histograms

By default the optimizer assumes a uniform distribution of rows across the distinct values in a column.

For columns that contain data skew (a nonuniform distribution of data within the column), a histogram enables the optimizer to generate accurate cardinality estimates for filter and join predicates that involve these columns.

For example, a California-based book store ships 95% of the books to California, 4% to Oregon, and 1% to Nevada. The book orders table has 300,000 rows. A table column stores the state to which orders are shipped. A user queries the number of books shipped to Oregon. Without a histogram, the optimizer assumes an even distribution of 300000/3 (the NDV is 3), estimating cardinality at 100,000 rows. With this estimate, the optimizer chooses a full table scan. With a histogram, the optimizer calculates that 4% of the books are shipped to Oregon, and chooses an index scan.

11.2 When Oracle Database Creates Histograms

If `DBMS_STATS` gathers statistics for a table, and if queries have referenced the columns in this table, then Oracle Database creates histograms automatically as needed according to the previous query workload.

The basic process is as follows:

1. You run `DBMS_STATS` for a table with the `METHOD_OPT` parameter set to the default `SIZE AUTO`.
2. A user queries the table.
3. The database notes the predicates in the preceding query and updates the data dictionary table `SYS.COL_USAGE$`.
4. You run `DBMS_STATS` again, causing `DBMS_STATS` to query `SYS.COL_USAGE$` to determine which columns require histograms based on the previous query workload.

Consequences of the `AUTO` feature include the following:

- As queries change over time, DBMS_STATS may change which statistics it gathers. For example, even if the data in a table does not change, queries and DBMS_STATS operations can cause the plans for queries that reference these tables to change.
- If you gather statistics for a table and do not query the table, then the database does not create histograms for columns in this table. For the database to create the histograms automatically, you must run one or more queries to populate the column usage information in SYS.COL_USAGE$.

Example 11-1 Automatic Histogram Creation

Assume that sh.sh_ext is an external table that contains the same rows as the sh.sales table. You create new table sales2 and perform a bulk load using sh_ext as a source, which automatically creates statistics for sales2. You also create indexes as follows:

```
SQL> CREATE TABLE sales2 AS SELECT * FROM sh_ext;
SQL> CREATE INDEX sh_12c_idx1 ON sales2(prod_id);
SQL> CREATE INDEX sh_12c_idx2 ON sales2(cust_id,time_id);
```

You query the data dictionary to determine whether histograms exist for the sales2 columns. Because sales2 has not yet been queried, the database has not yet created histograms:

```
SQL> SELECT COLUMN_NAME, NOTES, HISTOGRAM
  2  FROM   USER_TAB_COL_STATISTICS
  3  WHERE  TABLE_NAME = 'SALES2';

COLUMN_NAME   NOTES          HISTOGRAM
------------- -------------- ---------
AMOUNT_SOLD   STATS_ON_LOAD  NONE
QUANTITY_SOLD STATS_ON_LOAD  NONE
PROMO_ID      STATS_ON_LOAD  NONE
CHANNEL_ID    STATS_ON_LOAD  NONE
TIME_ID       STATS_ON_LOAD  NONE
CUST_ID       STATS_ON_LOAD  NONE
PROD_ID       STATS_ON_LOAD  NONE
```

You query sales2 for the number of rows for product 42, and then gather table statistics using the GATHER AUTO option:

```
SQL> SELECT COUNT(*) FROM sales2 WHERE prod_id = 42;

  COUNT(*)
----------
     12116

SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS(USER,'SALES2',OPTIONS=>'GATHER
AUTO');
```

A query of the data dictionary now shows that the database created a histogram on the `prod_id` column based on the information gather during the preceding query:

```
SQL> SELECT COLUMN_NAME, NOTES, HISTOGRAM
  2  FROM   USER_TAB_COL_STATISTICS
  3  WHERE  TABLE_NAME = 'SALES2';

COLUMN_NAME   NOTES          HISTOGRAM
------------- -------------- ---------
AMOUNT_SOLD   STATS_ON_LOAD  NONE
QUANTITY_SOLD STATS_ON_LOAD  NONE
PROMO_ID      STATS_ON_LOAD  NONE
CHANNEL_ID    STATS_ON_LOAD  NONE
TIME_ID       STATS_ON_LOAD  NONE
CUST_ID       STATS_ON_LOAD  NONE
PROD_ID       HISTOGRAM_ONLY FREQUENCY
```

11.3 How Oracle Database Chooses the Histogram Type

Oracle Database uses several criteria to determine which histogram to create: frequency, top frequency, height-balanced, or hybrid.

The histogram formula uses the following variables:

- NDV

  This represents the number of distinct values in a column. For example, if a column only contains the values `100`, `200`, and `300`, then the NDV for this column is 3.

- *n*

  This variable represents the number of histogram buckets. The default is 254.

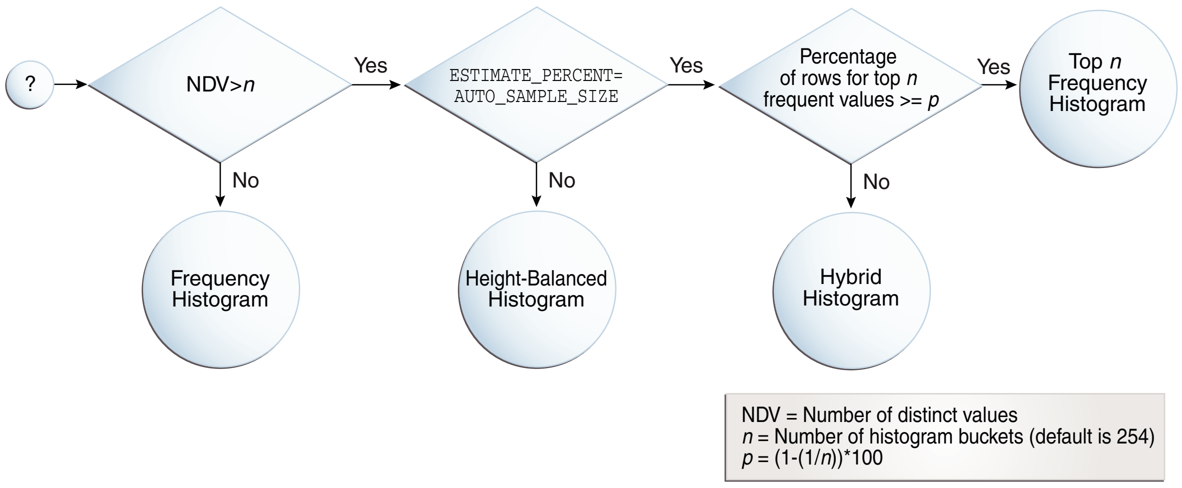
- *p*

  This variable represents an internal percentage threshold that is equal to (1–(1/*n*)) * 100. For example, if *n* = 254, then *p* is 99.6.

An additional criterion is whether the `estimate_percent` parameter in the `DBMS_STATS` statistics gathering procedure is set to `AUTO_SAMPLE_SIZE` (default).

The following diagram shows the decision tree for histogram creation.

Figure 11-1 Decision Tree for Histogram Creation



11.4 Cardinality Algorithms When Using Histograms

For histograms, the algorithm for cardinality depends on factors such as the endpoint numbers and values, and whether column values are popular or nonpopular.

11.4.1 Endpoint Numbers and Values

An **endpoint number** is a number that uniquely identifies a bucket. In frequency and hybrid histograms, the endpoint number is the cumulative frequency of all values included in the current and previous buckets.

For example, a bucket with endpoint number `100` means the total frequency of values in the current and all previous buckets is 100. In height-balanced histograms, the optimizer numbers buckets sequentially, starting at `0` or `1`. In all cases, the endpoint number is the bucket number.

An endpoint value is the highest value in the range of values in a bucket. For example, if a bucket contains only the values `52794` and `52795`, then the endpoint value is `52795`.

11.4.2 Popular and Nonpopular Values

The popularity of a value in a histogram affects the cardinality estimate algorithm.

Specifically, the cardinality estimate is affected as follows:

- Popular values

  A popular value occurs as an endpoint value of multiple buckets. The optimizer determines whether a value is popular by first checking whether it is the endpoint value for a bucket. If so, then for frequency histograms, the optimizer subtracts the endpoint number of the previous bucket from the endpoint number of the current bucket. Hybrid histograms already store this information for each endpoint individually. If this value is greater than 1, then the value is popular.

The optimizer calculates its cardinality estimate for popular values using the following formula:

```
cardinality of popular value =
  (num of rows in table) *
  (num of endpoints spanned by this value / total num of endpoints)
```

- Nonpopular values

  Any value that is not popular is a nonpopular value. The optimizer calculates the cardinality estimates for nonpopular values using the following formula:

  ```
  cardinality of nonpopular value =
    (num of rows in table) * density
  ```

  The optimizer calculates density using an internal algorithm based on factors such as the number of buckets and the NDV. Density is expressed as a decimal number between `0` and `1`. Values close to `1` indicate that the optimizer expects many rows to be returned by a query referencing this column in its predicate list. Values close to `0` indicate that the optimizer expects few rows to be returned.

See Also:

Oracle Database Reference to learn about the `DBA_TAB_COL_STATISTICS.DENSITY` column

11.4.3 Bucket Compression

In some cases, to reduce the total number of buckets, the optimizer compresses multiple buckets into a single bucket.

For example, the following frequency histogram indicates that the first bucket number is `1` and the last bucket number is `23`:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              1          52792
              6          52793
              8          52794
              9          52795
             10          52796
             12          52797
             14          52798
             23          52799
```

Several buckets are "missing." Originally, buckets `2` through `6` each contained a single instance of value `52793`. The optimizer compressed all of these buckets into the bucket with the highest endpoint number (bucket `6`), which now contains 5 instances of value `52793`. This value is popular because the difference between the endpoint number of the current bucket (`6`) and the previous bucket (`1`) is 5. Thus, before compression the value `52793` was the endpoint for 5 buckets.

The following annotations show which buckets are compressed, and which values are popular:

```
ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              1          52792 -> nonpopular
              6          52793 -> buckets 2-6 compressed into 6;
popular
              8          52794 -> buckets 7-8 compressed into 8;
popular
              9          52795 -> nonpopular
             10          52796 -> nonpopular
             12          52797 -> buckets 11-12 compressed into 12;
popular
             14          52798 -> buckets 13-14 compressed into 14;
popular
             23          52799 -> buckets 15-23 compressed into 23;
popular
```

11.5 Frequency Histograms

In a **frequency histogram**, each distinct column value corresponds to a single bucket of the histogram. Because each value has its own dedicated bucket, some buckets may have many values, whereas others have few.

An analogy to a frequency histogram is sorting coins so that each individual coin initially gets its own bucket. For example, the first penny is in bucket 1, the second penny is in bucket 2, the first nickel is in bucket 3, and so on. You then consolidate all the pennies into a single penny bucket, all the nickels into a single nickel bucket, and so on with the remainder of the coins.

11.5.1 Criteria For Frequency Histograms

Frequency histograms depend on the number of requested histogram buckets.

As shown in the logic diagram in "How Oracle Database Chooses the Histogram Type", the database creates a frequency histogram when the following criteria are met:

- NDV is less than or equal to *n*, where *n* is the number of histogram buckets (default `254`).

  For example, the `sh.countries.country_subregion_id` column has 8 distinct values, ranging sequentially from `52792` to `52799`. If *n* is the default of `254`, then the optimizer creates a frequency histogram because `8 <= 254`.

- The `estimate_percent` parameter in the `DBMS_STATS` statistics gathering procedure is set to either a user-specified value or to `AUTO_SAMPLE_SIZE`.

Starting in Oracle Database 12c, if the sampling size is the default of `AUTO_SAMPLE_SIZE`, then the database creates frequency histograms from a full table scan. For all other sampling percentage specifications, the database derives frequency histograms from a sample. In releases earlier than Oracle Database 12c, the database gathered histograms based on a small sample, which meant that low-frequency values often did not appear in the sample. Using density in this case sometimes led the optimizer to overestimate selectivity.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about `AUTO_SAMPLE_SIZE`

11.5.2 Generating a Frequency Histogram

This scenario shows how to generate a frequency histogram using the sample schemas.

Assumptions

This scenario assumes that you want to generate a frequency histogram on the `sh.countries.country_subregion_id` column. This table has 23 rows.

The following query shows that the `country_subregion_id` column contains 8 distinct values (sample output included) that are unevenly distributed:

```
SELECT country_subregion_id, count(*)
FROM   sh.countries
GROUP BY country_subregion_id
ORDER BY 1;

COUNTRY_SUBREGION_ID   COUNT(*)
-------------------- ----------
               52792          1
               52793          5
               52794          2
               52795          1
               52796          1
               52797          2
               52798          2
               52799          9
```

To generate a frequency histogram:

1. Gather statistics for `sh.countries` and the `country_subregion_id` column, letting the number of buckets default to 254.

   For example, execute the following PL/SQL anonymous block:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS (
       ownname    => 'SH'
   ,   tabname    => 'COUNTRIES'
   ,   method_opt => 'FOR COLUMNS COUNTRY_SUBREGION_ID'
   );
   END;
   ```

2. Query the histogram information for the `country_subregion_id` column.

For example, use the following query (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, HISTOGRAM
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

TABLE_NAME COLUMN_NAME          NUM_DISTINCT HISTOGRAM
---------- -------------------- ------------ ---------------
COUNTRIES  COUNTRY_SUBREGION_ID            8 FREQUENCY
```

The optimizer chooses a frequency histogram because *n* or fewer distinct values exist in the column, where *n* defaults to `254`.

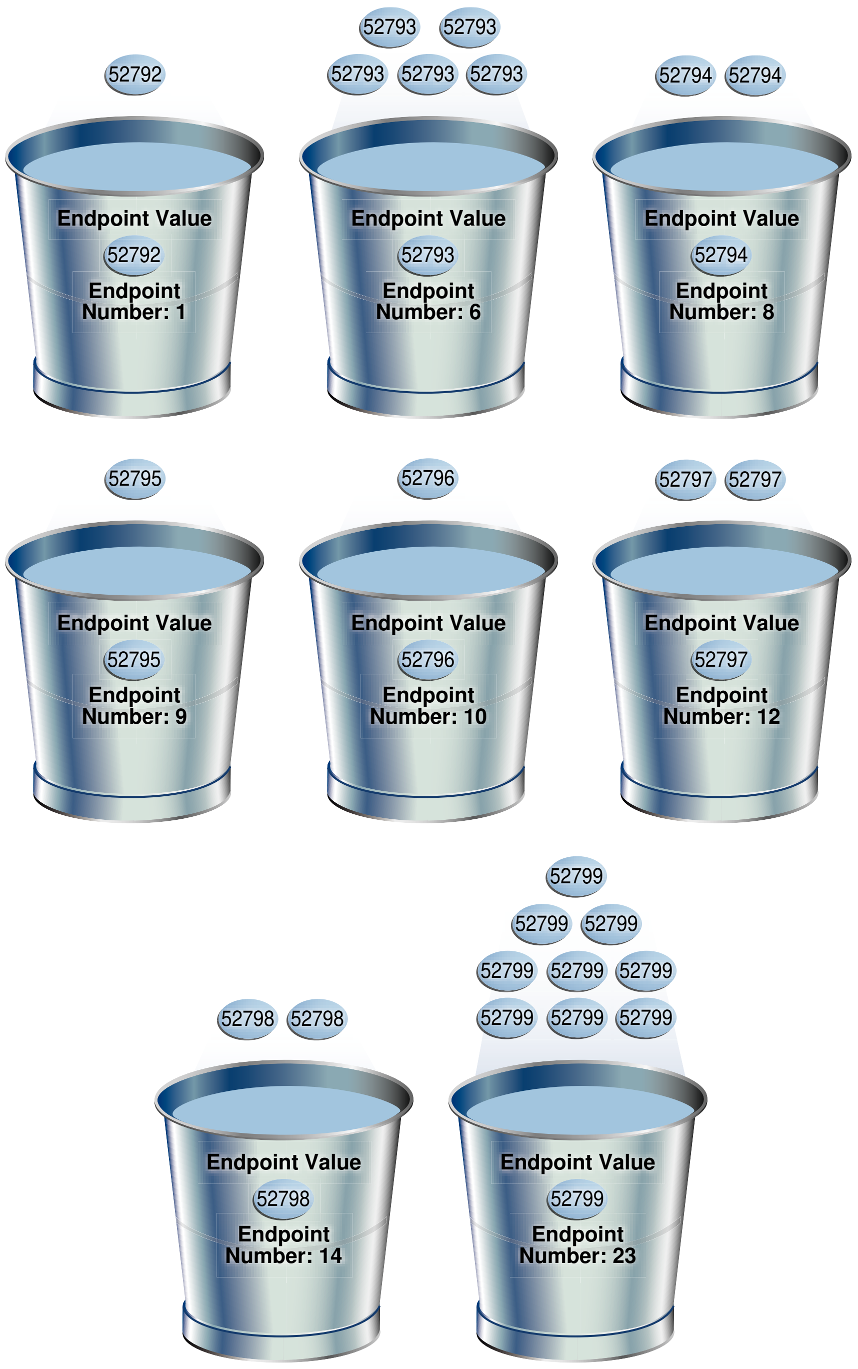
3. Query the endpoint number and endpoint value for the `country_subregion_id` column.

   For example, use the following query (sample output included):

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
FROM   USER_HISTOGRAMS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              1          52792
              6          52793
              8          52794
              9          52795
             10          52796
             12          52797
             14          52798
             23          52799
```

Figure 11-2 is a graphical illustration of the 8 buckets in the histogram. Each value is represented as a coin that is dropped into a bucket.

Figure 11-2 Frequency Histogram



As shown in Figure 11-2, each distinct value has its own bucket. Because this is a frequency histogram, the endpoint number is the cumulative frequency of endpoints. For `52793`, the endpoint number `6` indicates that the value appears 5 times (6 - 1). For `52794`, the endpoint number `8` indicates that the value appears 2 times (8 - 6).

Every bucket whose endpoint is at least 2 greater than the previous endpoint contains a popular value. Thus, buckets `6`, `8`, `12`, `14`, and `23` contain popular values. The optimizer calculates their cardinality based on endpoint numbers. For example, the optimizer calculates the cardinality (`C`) of value `52799` using the following formula, where the number of rows in the table is 23:

```
C = 23 * ( 9 / 23 )
```

Buckets `1`, `9`, and `10` contain nonpopular values. The optimizer estimates their cardinality based on density.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure
- *Oracle Database Reference* to learn about the `USER_TAB_COL_STATISTICS` view
- *Oracle Database Reference* to learn about the `USER_HISTOGRAMS` view

11.6 Top Frequency Histograms

A **top frequency histogram** is a variation on a frequency histogram that ignores nonpopular values that are statistically insignificant.

For example, if a pile of 1000 coins contains only a single penny, then you can ignore the penny when sorting the coins into buckets. A top frequency histogram can produce a better histogram for highly popular values.

11.6.1 Criteria For Top Frequency Histograms

If a small number of values occupies most of the rows, then creating a frequency histogram on this small set of values is useful even when the NDV is greater than the number of requested histogram buckets. To create a better quality histogram for popular values, the optimizer ignores the nonpopular values and creates a top frequency histogram.

As shown in the logic diagram in "How Oracle Database Chooses the Histogram Type", the database creates a top frequency histogram when the following criteria are met:

- NDV is greater than *n*, where *n* is the number of histogram buckets (default `254`).
- The percentage of rows occupied by the top *n* frequent values is equal to or greater than threshold *p*, where *p* is `(1-(1/`*n*`))*100`.
- The `estimate_percent` parameter in the `DBMS_STATS` statistics gathering procedure is set to `AUTO_SAMPLE_SIZE`.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about `AUTO_SAMPLE_SIZE`

11.6.2 Generating a Top Frequency Histogram

This scenario shows how to generate a top frequency histogram using the sample schemas.

Assumptions

This scenario assumes that you want to generate a top frequency histogram on the `sh.countries.country_subregion_id` column. This table has 23 rows.

The following query shows that the `country_subregion_id` column contains 8 distinct values (sample output included) that are unevenly distributed:

```
SELECT country_subregion_id, count(*)
FROM   sh.countries
GROUP BY country_subregion_id
ORDER BY 1;

COUNTRY_SUBREGION_ID   COUNT(*)
-------------------- ----------
               52792          1
               52793          5
               52794          2
               52795          1
               52796          1
               52797          2
               52798          2
               52799          9
```

To generate a top frequency histogram:

1. Gather statistics for `sh.countries` and the `country_subregion_id` column, specifying fewer buckets than distinct values.

   For example, enter the following command to specify 7 buckets:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS (
       ownname    => 'SH'
   ,   tabname    => 'COUNTRIES'
   ,   method_opt => 'FOR COLUMNS COUNTRY_SUBREGION_ID SIZE 7'
   );
   END;
   ```

2. Query the histogram information for the `country_subregion_id` column.

For example, use the following query (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, HISTOGRAM
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

TABLE_NAME COLUMN_NAME          NUM_DISTINCT HISTOGRAM
---------- -------------------- ------------ ---------------
COUNTRIES  COUNTRY_SUBREGION_ID            7 TOP-FREQUENCY
```

The `sh.countries.country_subregion_id` column contains 8 distinct values, but the histogram only contains 7 buckets, making *n*=`7`. In this case, the database can only create a top frequency or hybrid histogram. In the `country_subregion_id` column, the top 7 most frequent values occupy 95.6% of the rows, which exceeds the threshold of 85.7%, generating a top frequency histogram.

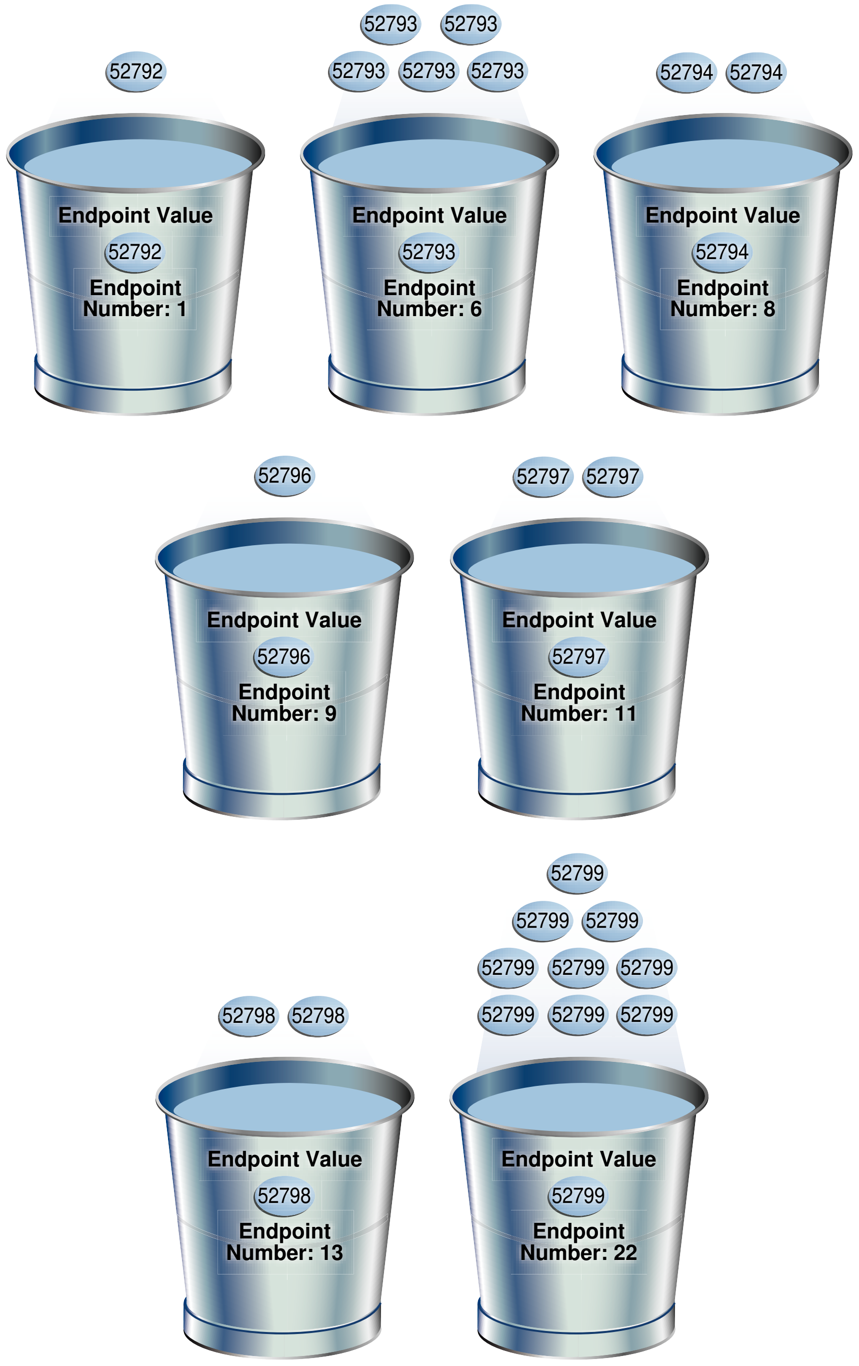
3. Query the endpoint number and endpoint value for the column.

   For example, use the following query (sample output included):

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
FROM   USER_HISTOGRAMS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              1          52792
              6          52793
              8          52794
              9          52796
             11          52797
             13          52798
             22          52799
```

Figure 11-3 is a graphical illustration of the 7 buckets in the top frequency histogram. The values are represented in the diagram as coins.

Figure 11-3 Top Frequency Histogram



As shown in Figure 11-3, each distinct value has its own bucket except for `52795`, which is excluded from the histogram because it is nonpopular and statistically insignificant. As in a standard frequency histogram, the endpoint number represents the cumulative frequency of values.

See Also:

- "Criteria For Frequency Histograms"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure
- *Oracle Database Reference* to learn about the `USER_TAB_COL_STATISTICS` view
- *Oracle Database Reference* to learn about the `USER_HISTOGRAMS` view

11.7 Height-Balanced Histograms (Legacy)

In a legacy height-balanced histogram, column values are divided into buckets so that each bucket contains approximately the same number of rows.

For example, if you have 99 coins to distribute among 4 buckets, each bucket contains about 25 coins. The histogram shows where the endpoints fall in the range of values.

11.7.1 Criteria for Height-Balanced Histograms

Before Oracle Database 12c, the database created a height-balanced histogram when the NDV was greater than *n*. This type of histogram was useful for range predicates, and equality predicates on values that appear as endpoints in at least two buckets.

As shown in the logic diagram in "How Oracle Database Chooses the Histogram Type", the database creates a height-balanced histogram when the following criteria are met:

- NDV is greater than *n*, where *n* is the number of histogram buckets (default `254`).
- The `estimate_percent` parameter in the `DBMS_STATS` statistics gathering procedure is *not* set to `AUTO_SAMPLE_SIZE`.

It follows that if Oracle Database 12c creates new histograms, and if the sampling percentage is `AUTO_SAMPLE_SIZE`, then the histograms are either top frequency or hybrid, but not height-balanced.

If you upgrade Oracle Database 11g to Oracle Database 12c, then any height-based histograms created *before* the upgrade remain in use. However, if you refresh statistics on the table on which the histogram was created, then the database replaces existing height-balanced histograms on this table. The type of replacement histogram depends on both the NDV and the following criteria:

- If the sampling percentage is `AUTO_SAMPLE_SIZE`, then the database creates either hybrid or frequency histograms.
- If the sampling percentage is not `AUTO_SAMPLE_SIZE`, then the database creates either height-balanced or frequency histograms.

11.7.2 Generating a Height-Balanced Histogram

This scenario shows how to generate a height-balanced histogram using the sample schemas.

Assumptions

This scenario assumes that you want to generate a height-balanced histogram on the `sh.countries.country_subregion_id` column. This table has 23 rows.

The following query shows that the `country_subregion_id` column contains 8 distinct values (sample output included) that are unevenly distributed:

```
SELECT country_subregion_id, count(*)
FROM   sh.countries
GROUP BY country_subregion_id
ORDER BY 1;

COUNTRY_SUBREGION_ID   COUNT(*)
-------------------- ----------
               52792          1
               52793          5
               52794          2
               52795          1
               52796          1
               52797          2
               52798          2
               52799          9
```

To generate a height-balanced histogram:

1. Gather statistics for `sh.countries` and the `country_subregion_id` column, specifying fewer buckets than distinct values.

   Note:

   To simulate Oracle Database 11g behavior, which is necessary to create a height-based histogram, set `estimate_percent` to a nondefault value. If you specify a nondefault percentage, then the database creates frequency or height-balanced histograms.

   For example, enter the following command:

```
BEGIN  DBMS_STATS.GATHER_TABLE_STATS (
    ownname          => 'SH'
,   tabname          => 'COUNTRIES'
,   method_opt       => 'FOR COLUMNS COUNTRY_SUBREGION_ID SIZE 7'
,   estimate_percent => 100
);
END;
```

2. Query the histogram information for the `country_subregion_id` column.

   For example, use the following query (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, HISTOGRAM
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

TABLE_NAME COLUMN_NAME          NUM_DISTINCT HISTOGRAM
---------- -------------------- ------------ ---------------
COUNTRIES  COUNTRY_SUBREGION_ID            8 HEIGHT BALANCED
```

   The optimizer chooses a height-balanced histogram because the number of distinct values (8) is greater than the number of buckets (7), and the `estimate_percent` value is nondefault.

3. Query the number of rows occupied by each distinct value.

   For example, use the following query (sample output included):

```
SELECT COUNT(country_subregion_id) AS NUM_OF_ROWS,
country_subregion_id
FROM   countries
GROUP BY country_subregion_id
ORDER BY 2;

NUM_OF_ROWS COUNTRY_SUBREGION_ID
----------- --------------------
          1                52792
          5                52793
          2                52794
          1                52795
          1                52796
          2                52797
          2                52798
          9                52799
```

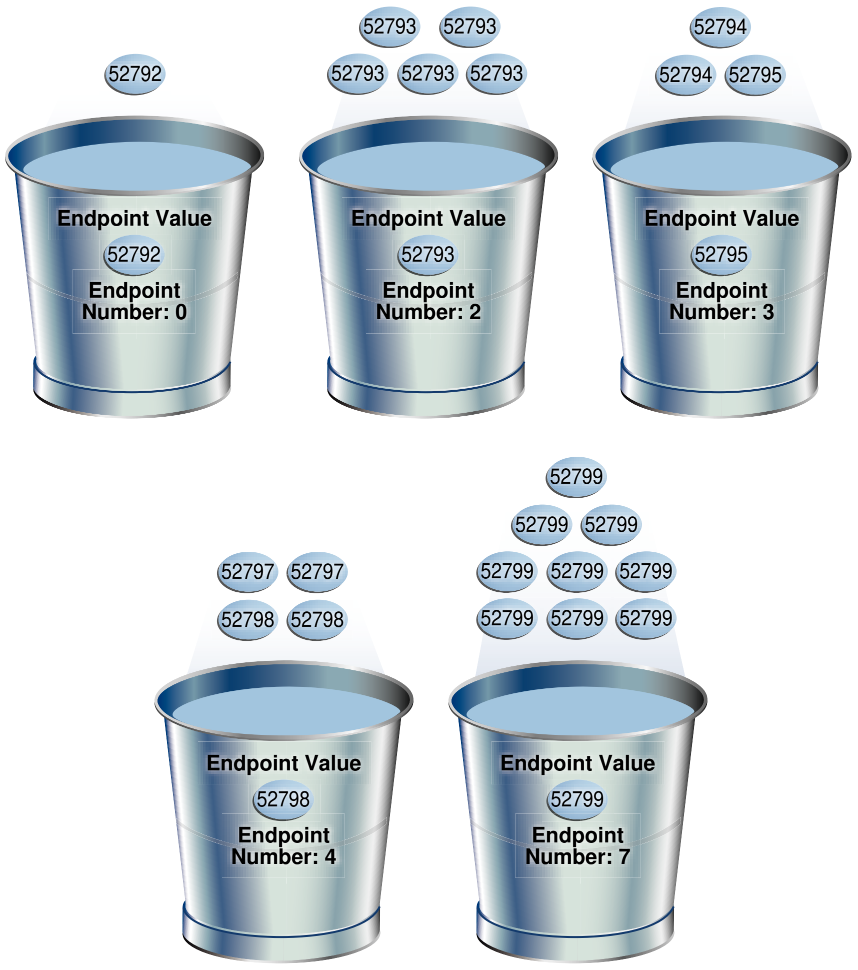
4. Query the endpoint number and endpoint value for the `country_subregion_id` column.

   For example, use the following query (sample output included):

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE
FROM   USER_HISTOGRAMS
WHERE  TABLE_NAME='COUNTRIES'
AND    COLUMN_NAME='COUNTRY_SUBREGION_ID';

ENDPOINT_NUMBER ENDPOINT_VALUE
--------------- --------------
              0          52792
              2          52793
              3          52795
              4          52798
              7          52799
```

The following illustration represents a height-balanced histogram. The values are represented in the diagram as coins.

Figure 11-4   Height-Balanced Histogram



The bucket number is identical to the endpoint number. The optimizer records the value of the last row in each bucket as the endpoint value, and then checks to ensure that the minimum value is the endpoint value of the first bucket, and the maximum value is the endpoint value of the last bucket. In this example, the optimizer adds bucket `0` so that the minimum value `52792` is the endpoint of a bucket.

The optimizer must evenly distribute 23 rows into the 7 specified histogram buckets, so each bucket contains approximately 3 rows. However, the optimizer compresses buckets with the same endpoint. So, instead of bucket `1` containing 2 instances of value `52793`, and bucket `2` containing 3 instances of value `52793`, the optimizer puts all 5 instances of value `52793` into bucket `2`. Similarly, instead of having buckets `5`, `6`, and `7` contain 3 values each, with the endpoint of each bucket as `52799`, the optimizer puts all 9 instances of value `52799` into bucket `7`.

In this example, buckets `3` and `4` contain nonpopular values because the difference between the current endpoint number and previous endpoint number is 1. The optimizer calculates cardinality for these values based on density. The remaining buckets contain popular values. The optimizer calculates cardinality for these values based on endpoint numbers.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure
- *Oracle Database Reference* to learn about the `USER_TAB_COL_STATISTICS` and `USER_HISTOGRAMS` views

11.8 Hybrid Histograms

A **hybrid histogram** combines characteristics of both height-based histograms and frequency histograms. This "best of both worlds" approach enables the optimizer to obtain better selectivity estimates in some situations.

The height-based histogram sometimes produces inaccurate estimates for values that are *almost* popular. For example, a value that occurs as an endpoint value of only one bucket but almost occupies two buckets is not considered popular.

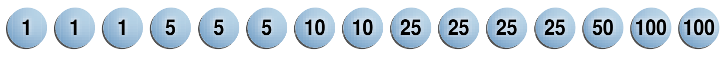
To solve this problem, a hybrid histogram distributes values so that no value occupies more than one bucket, and then stores the endpoint repeat count value, which is the number of times the endpoint value is repeated, for each endpoint (bucket) in the histogram. By using the repeat count, the optimizer can obtain accurate estimates for almost popular values.

11.8.1 How Endpoint Repeat Counts Work

The analogy of coins distributed among buckets illustrate show endpoint repeat counts work.

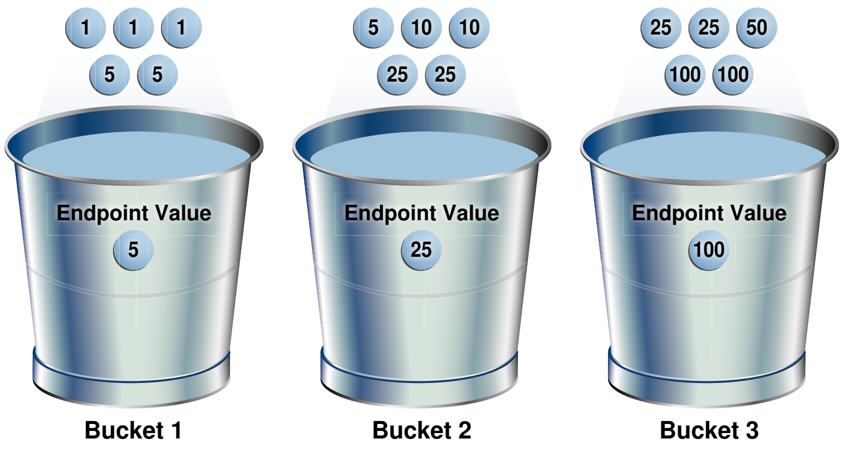
The following figure illustrates a `coins` column that sorts values from low to high.

Figure 11-5 Coins



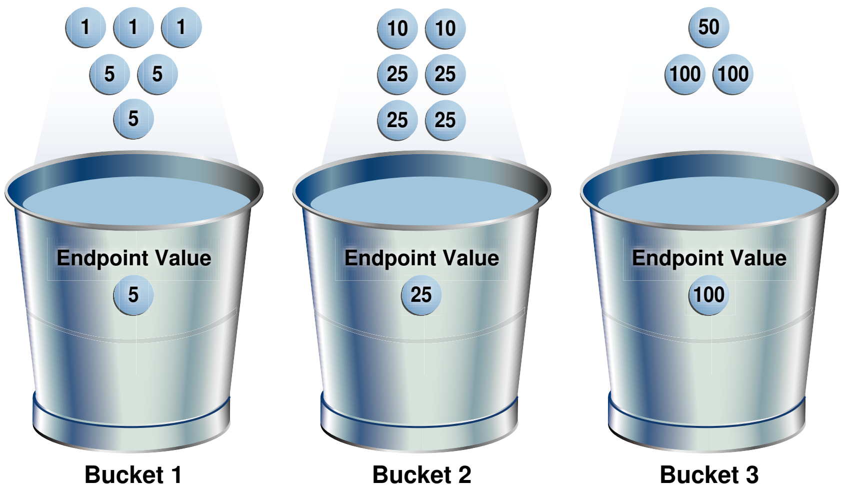
You gather statistics for this table, setting the `method_opt` argument of `DBMS_STATS.GATHER_TABLE_STATS` to `FOR ALL COLUMNS SIZE 3`. In this case, the optimizer initially groups the values in the `coins` column into three buckets, as shown in the following figure.

Figure 11-6 Initial Distribution of Values



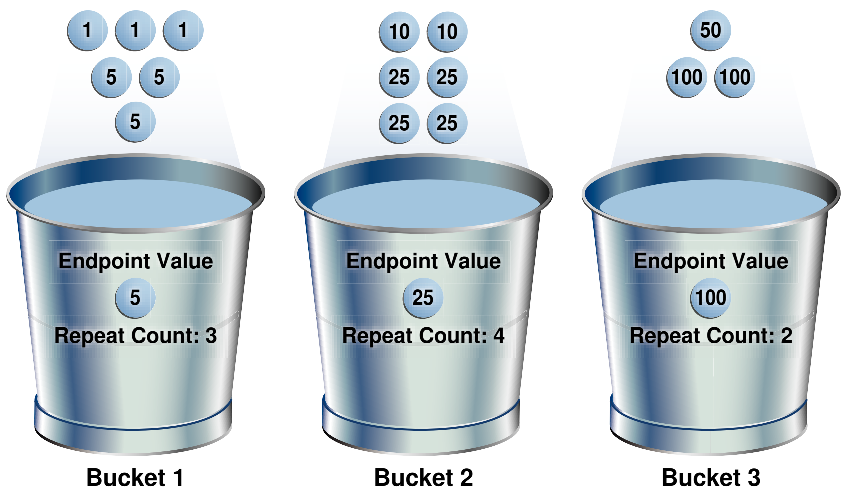
If a bucket border splits a value so that some occurrences of the value are in one bucket and some in another, then the optimizer shifts the bucket border (and all other following bucket borders) forward to include all occurrences of the value. For example, the optimizer shifts value `5` so that it is now wholly in the first bucket, and the value `25` is now wholly in the second bucket.

Figure 11-7 Redistribution of Values



The endpoint repeat count measures the number of times that the corresponding bucket endpoint, which is the value at the right bucket border, repeats itself. For example, in the first bucket, the value `5` is repeated 3 times, so the endpoint repeat count is `3`.

Figure 11-8 Endpoint Repeat Count



Height-balanced histograms do not store as much information as hybrid histograms. By using repeat counts, the optimizer can determine exactly how many occurrences of an endpoint value exist. For example, the optimizer knows that the value `5` appears 3 times, the value `25` appears 4 times, and the value `100` appears 2 times. This frequency information helps the optimizer to generate better cardinality estimates.

11.8.2 Criteria for Hybrid Histograms

The only differentiating criterion for hybrid histograms as compared to top frequency histograms is that the top *n* frequent values is less than internal threshold *p*.

As shown in the logic diagram in "How Oracle Database Chooses the Histogram Type", the database creates a hybrid histogram when the following criteria are met:

- NDV is greater than *n*, where *n* is the number of histogram buckets (default is `254`).
- The criteria for top frequency histograms do not apply.

  This is another way to stating that the percentage of rows occupied by the top *n* frequent values is less than threshold *p*, where *p* is `(1-(1/n))*100`.
- The `estimate_percent` parameter in the `DBMS_STATS` statistics gathering procedure is set to `AUTO_SAMPLE_SIZE`.

  If users specify their own percentage, then the database creates frequency or height-balanced histograms.

See Also:

- "Criteria For Top Frequency Histograms."
- "Height-Balanced Histograms (Legacy)"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `estimate_percent` parameter

11.8.3 Generating a Hybrid Histogram

This scenario shows how to generate a hybrid histogram using the sample schemas.

Assumptions

This scenario assumes that you want to generate a hybrid histogram on the `sh.products.prod_subcategory_id` column. This table has 72 rows. The `prod_subcategory_id` column contains 22 distinct values.

To generate a hybrid histogram:

1. Gather statistics for `sh.products` and the `prod_subcategory_id` column, specifying 10 buckets.

   For example, enter the following command:

```
BEGIN  DBMS_STATS.GATHER_TABLE_STATS (
    ownname     => 'SH'
,   tabname     => 'PRODUCTS'
,   method_opt  => 'FOR COLUMNS PROD_SUBCATEGORY_ID SIZE 10'
);
END;
```

2. Query the number of rows occupied by each distinct value.

   For example, use the following query (sample output included):

```
SELECT COUNT(prod_subcategory_id) AS NUM_OF_ROWS, prod_subcategory_id
FROM   products
GROUP BY prod_subcategory_id
ORDER BY 1 DESC;

NUM_OF_ROWS PROD_SUBCATEGORY_ID
----------- -------------------
          8                2014
          7                2055
          6                2032
          6                2054
          5                2056
          5                2031
          5                2042
          5                2051
          4                2036
          3                2043
```

```
                   2                 2033
                   2                 2034
                   2                 2013
                   2                 2012
                   2                 2053
                   2                 2035
                   1                 2022
                   1                 2041
                   1                 2044
                   1                 2011
                   1                 2021
                   1                 2052

22 rows selected.
```

The column contains 22 distinct values. Because the number of buckets (10) is less than 22, the optimizer cannot create a frequency histogram. The optimizer considers both hybrid and top frequency histograms. To qualify for a top frequency histogram, the percentage of rows occupied by the top 10 most frequent values must be equal to or greater than threshold *p*, where *p* is (1-(1/10))*100, or 90%. However, in this case the top 10 most frequent values occupy 54 rows out of 72, which is only 75% of the total. Therefore, the optimizer chooses a hybrid histogram because the criteria for a top frequency histogram do not apply.

3. Query the histogram information for the `country_subregion_id` column.

   For example, use the following query (sample output included):

```
SELECT TABLE_NAME, COLUMN_NAME, NUM_DISTINCT, HISTOGRAM
FROM   USER_TAB_COL_STATISTICS
WHERE  TABLE_NAME='PRODUCTS'
AND    COLUMN_NAME='PROD_SUBCATEGORY_ID';

TABLE_NAME COLUMN_NAME         NUM_DISTINCT HISTOGRAM
---------- ------------------- ------------ ---------
PRODUCTS   PROD_SUBCATEGORY_ID 22           HYBRID
```

4. Query the endpoint number, endpoint value, and endpoint repeat count for the `country_subregion_id` column.

   For example, use the following query (sample output included):

```
SELECT ENDPOINT_NUMBER, ENDPOINT_VALUE, ENDPOINT_REPEAT_COUNT
FROM   USER_HISTOGRAMS
WHERE  TABLE_NAME='PRODUCTS'
AND    COLUMN_NAME='PROD_SUBCATEGORY_ID'
ORDER BY 1;

ENDPOINT_NUMBER ENDPOINT_VALUE ENDPOINT_REPEAT_COUNT
--------------- -------------- ---------------------
              1           2011                     1
             13           2014                     8
             26           2032                     6
             36           2036                     4
             45           2043                     3
             51           2051                     5
```

```
          52             2052                      1
          54             2053                      2
          60             2054                      6
          72             2056                      5

10 rows selected.
```

In a height-based histogram, the optimizer would evenly distribute 72 rows into the 10 specified histogram buckets, so that each bucket contains approximately 7 rows. Because this is a hybrid histogram, the optimizer distributes the values so that no value occupies more than one bucket. For example, the optimizer does not put some instances of value `2036` into one bucket and some instances of this value into another bucket: all instances are in bucket `36`.

The endpoint repeat count shows the number of times the highest value in the bucket is repeated. By using the endpoint number and repeat count for these values, the optimizer can estimate cardinality. For example, bucket `36` contains instances of values `2033`, `2034`, `2035`, and `2036`. The endpoint value `2036` has an endpoint repeat count of `4`, so the optimizer knows that 4 instances of this value exist. For values such as `2033`, which are not endpoints, the optimizer estimates cardinality using density.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure
- *Oracle Database Reference* to learn about the `USER_TAB_COL_STATISTICS` view
- *Oracle Database Reference* to learn about the `USER_HISTOGRAMS` view

12 Configuring Options for Optimizer Statistics Gathering

This chapter explains what optimizer statistics collection is and how to set statistics preferences.

12.1 About Optimizer Statistics Collection

In Oracle Database, **optimizer statistics collection** is the gathering of optimizer statistics for database objects, including fixed objects.

The database can collect optimizer statistics automatically. You can also collect them manually using the `DBMS_STATS` package.

12.1.1 Purpose of Optimizer Statistics Collection

The contents of tables and associated indexes change frequently, which can lead the optimizer to choose suboptimal execution plan for queries. To avoid potential performance issues, statistics must be kept current.

To minimize DBA involvement, Oracle Database automatically gathers optimizer statistics at various times. Some automatic options are configurable, such enabling AutoTask to run `DBMS_STATS`.

12.1.2 User Interfaces for Optimizer Statistics Management

You can manage optimizer statistics either through Oracle Enterprise Manager Cloud Control (Cloud Control) or using PL/SQL on the command line.

12.1.2.1 Graphical Interface for Optimizer Statistics Management

The Manage Optimizer Statistics page in Cloud Control is a GUI that enables you to manage optimizer statistics.

12.1.2.1.1 Accessing the Database Home Page in Cloud Control

Oracle Enterprise Manager Cloud Control enables you to manage multiple databases within a single GUI-based framework.

To access a database home page using Cloud Control:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.

4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.

> **See Also:**
>
> Cloud Control online help

12.1.2.1.2 Accessing the Optimizer Statistics Console

You can perform most necessary tasks relating to optimizer statistics through pages linked to by the Optimizer Statistics Console page.

To manage optimizer statistics using Cloud Control:

1. In Cloud Control, access the Database Home page.
2. From the **Performance** menu, select **SQL**, then **Optimizer Statistics**.

   The Optimizer Statistics Console appears.

> **See Also:**
>
> Online Help for Oracle Enterprise Manager Cloud Control

12.1.2.2 Command-Line Interface for Optimizer Statistics Management

The `DBMS_STATS` package performs most optimizer statistics tasks.

To enable and disable automatic statistics gathering, use the `DBMS_AUTO_TASK_ADMIN` PL/SQL package.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn how to use `DBMS_STATS` and `DBMS_AUTO_TASK_ADMIN`

12.2 Setting Optimizer Statistics Preferences

This topic explains how to set optimizer statistics defaults using `DBMS_STATS.SET_*_PREFS` procedures.

12.2.1 About Optimizer Statistics Preferences

The **optimizer statistics preferences** set the default values of the parameters used by automatic statistics collection and the `DBMS_STATS` statistics gathering procedures.

12.2.1.1 Purpose of Optimizer Statistics Preferences

Preferences enable you to maintain optimizer statistics automatically when some objects require settings that differ from the default.

Preferences give you more granular control over how Oracle Database gathers statistics. You can set optimizer statistics preferences at the following levels:

- Table
- Schema
- Database (all tables)
- Global (tables with no preferences and any tables created in the future)

The `DBMS_STATS` procedures for setting preferences have names of the form `SET_*_PREFS`.

12.2.1.2 Examples of Statistics Preferences

Set preferences using the `pname` parameter of the `SET_*_PREFS` procedures.

Preferences that you can set include, but are not limited to, the following:

- `ESTIMATE_PERCENT`

  This preference determines the percentage of rows to estimate.
- `CONCURRENT`

  This preference determines whether the database gathers statistics concurrently on multiple objects, or serially, one object at a time.
- `STALE_PERCENT`

  This preference determines the percentage of rows in a table that must change before the database deems the statistics stale and in need of regathering.
- `AUTO_STAT_EXTENSIONS`

  When set to the non-default value of `ON`, this preference enables a SQL plan directive to trigger the creation of column group statistics based on usage of columns in the predicates in the workload.
- `INCREMENTAL`

  This preference determines whether the database maintains the global statistics of a partitioned table without performing a full table scan. Possible values are `TRUE` and `FALSE`.

  For example, by the default setting for `INCREMENTAL` is `FALSE`. You can set `INCREMENTAL` to `TRUE` for a range-partitioned table when the last few partitions are updated. Also, when performing a partition exchange operation on a nonpartitioned table, Oracle recommends that you set `INCREMENTAL` to `TRUE` and `INCREMENTAL_LEVEL` to `TABLE`. With these settings, `DBMS_STATS` gathers table-level synopses on this table.
- `INCREMENTAL_LEVEL`

  This preference controls what synopses to collect when `INCREMENTAL` preference is set to `TRUE`. It takes two values: `TABLE` or `PARTITION`.
- `APPROXIMATE_NDV_ALGORITHM`

This preference controls which algorithm to use when calculating the number of distinct values for partitioned tables using incremental statistics.

- `ROOT_TRIGGER_PDB`

  This preference controls whether to accept or reject the statistics gathering triggered from an application root in a CDB.

  By default, when gathering statistics for a metadata-linked table in the application root, if the statistics the application PDB are stale, the database does *not* trigger statistics gathering on the application PDB. When set to `TRUE`, `ROOT_TRIGGER_PDB` triggers statistics gathering on the application PDB, and then derives the global statistics in the application root.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS` procedures for setting optimizer statistics preferences

12.2.1.3 DBMS_STATS Procedures for Setting Statistics Preferences

The `DBMS_STATS.SET_*_PREFS` procedures change the defaults of parameters used by the `DBMS_STATS.GATHER_*_STATS` procedures. To query the current preferences, use the `DBMS_STATS.GET_PREFS` function.

When setting statistics preferences, the order of precedence is:

1. Table preference (set for a specific table, all tables in a schema, or all tables in the database)
2. Global preference
3. Default preference

The following table summarizes the relevant `DBMS_STATS` procedures.

Table 12-1 DBMS_STATS Procedures for Setting Optimizer Statistics Preferences

Procedure	Scope
`SET_TABLE_PREFS`	Specified table only.
`SET_SCHEMA_PREFS`	All existing tables in the specified schema. This procedure calls `SET_TABLE_PREFS` for each table in the specified schema. Calling `SET_SCHEMA_PREFS` does not affect any new tables created after it has been run. New tables use the `GLOBAL_PREF` values for all parameters.
`SET_DATABASE_PREFS`	All user-defined schemas in the database. You can include system-owned schemas such as `SYS` and `SYSTEM` by setting the `ADD_SYS` parameter to `true`. This procedure calls `SET_TABLE_PREFS` for each table in the specified schema. Calling `SET_DATABASE_PREFS` does not affect any new objects created after it has been run. New objects use the `GLOBAL_PREF` values for all parameters.

Table 12-1 (Cont.) DBMS_STATS Procedures for Setting Optimizer Statistics Preferences

Procedure	Scope
`SET_GLOBAL_PREFS`	Any table that does not have an existing table preference. All parameters default to the global setting unless a table preference is set or the parameter is explicitly set in the `DBMS_STATS.GATHER_*_STATS` statement. Changes made by `SET_GLOBAL_PREFS` affect any new objects created after it runs. New objects use the `SET_GLOBAL_PREFS` values for all parameters. With `SET_GLOBAL_PREFS`, you can set a default value for the parameter `AUTOSTATS_TARGET`. This additional parameter controls which objects the automatic statistic gathering job running in the nightly maintenance window affects. Possible values for `AUTOSTATS_TARGET` are `ALL`, `ORACLE`, and `AUTO` (default). You can only set the `CONCURRENT` preference at the global level. You cannot set the preference `INCREMENTAL_LEVEL` using `SET_GLOBAL_PREFS`.

See Also:

- "About Concurrent Statistics Gathering"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` procedures for setting optimizer statistics preferences

12.2.1.4 Statistics Preference Overrides

The `preference_overrides_parameter` statistics preference determines whether, when gathering optimizer statistics, to override the input value of a parameter with the statistics preference. In this way, you control when the database honors a parameter value passed to the statistics gathering procedures.

When `preference_overrides_parameter` is set to `FALSE` (default), the input values for statistics gathering procedures are honored. When set to `TRUE`, the input values are ignored.

Set the `preference_overrides_parameter` preference using the `SET_TABLE_PREFS`, `SET_SCHEMA_PREFS`, or `SET_GLOBAL_PREFS` procedures in `DBMS_STATS`. Regardless of whether `preference_overrides_parameter` is set, the database uses the same order of precedence for setting statistics:

1. Table preference (set for a specific table, all tables in a schema, or all tables in the database)
2. Global preference
3. Default preference

Example 12-1 Overriding Statistics Preferences at the Table Level

In this example, legacy scripts set `estimate_percent` explicitly rather than using the recommended `AUTO_SAMPLE_SIZE`. Your goal is to prevent users from using these scripts to set preferences on the `sh.costs` table.

Table 12-2 Overriding Statistics Preferences at the Table Level

Action	Description

```
SQL> SELECT DBMS_STATS.GET_PREFS
('estimate_percent', 'sh','costs')
AS "STAT_PREFS" FROM DUAL;

STAT_PREFS
----------
DBMS_STATS.AUTO_SAMPLE_SIZE
```
	No preference for `estimate_percent` is set for `sh.costs` or at the global level, so the preference defaults to `AUTO_SAMPLE_SIZE`.

```
SQL> EXEC DBMS_STATS.SET_TABLE_PREFS
('sh', 'costs',
'preference_overrides_parameter', 'true');

PL/SQL procedure successfully completed.
```
	By default, Oracle Database accepts preferences that are passed to the `GATHER_*_STATS` procedures. To override these parameters, you use `SET_TABLE_PREFS` to set the `preference_overrides_parameter` preference to `true` for the `costs` table only.

```
SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS
('sh', 'costs', estimate_percent=>100);

PL/SQL procedure successfully completed.
```
	You attempt to set `estimate_percent` to `100` when gathering statistics for `sh.costs`. However, because `preference_overrides_parameter` is `true` for this table, Oracle Database does not honor the `estimate_percent=>100`setting. Instead, the database gathers statistics using `AUTO_SAMPLE_SIZE`, which is the default.

Example 12-2 Overriding Statistics Preferences at the Global Level

In this example, you set `estimate_percent` to `5` at the global level, which means that this preference applies to every table in the database that does *not* have a table preference set. You then set an override on the `sh.sales` table, which does not have a table-level preference set, to prevent users from overriding the global setting in their scripts.

Table 12-3 Overriding Statistics Preferences at the Global Level

Action	Description
`SQL> SELECT DBMS_STATS.GET_PREFS` `('estimate_percent', 'sh','sales')` `AS "STAT_PREFS" FROM DUAL;` `STAT_PREFS` `----------` `DBMS_STATS.AUTO_SAMPLE_SIZE`	No preference for `estimate_percent` is set for `sh.sales` or at the global level, so the preference defaults to `AUTO_SAMPLE_SIZE`.
`SQL> EXEC DBMS_STATS.SET_GLOBAL_PREFS` `('estimate_percent', '5');` `PL/SQL procedure successfully completed.`	You use the `SET_GLOBAL_PREFS` procedure to set the `estimate_percent` preference to `5` for every table in the database that does not have a table preference set.
`SQL> SELECT DBMS_STATS.GET_PREFS` `('estimate_percent', 'sh','sales')` `AS "STAT_PREFS" FROM DUAL;` `STAT_PREFS` `----------` `5`	Because `sh.sales` does not have a preference set, the global setting applies to this table. A query of the preferences for `sh.sales` now shows that the `estimate_percent` setting is `5`, which is the global setting.
`SQL> EXEC DBMS_STATS.SET_TABLE_PREFS` `('sh', 'sales',` `'preference_overrides_parameter', 'true');` `PL/SQL procedure successfully completed.`	You use `SET_TABLE_PREFS` to set the `preference_overrides_parameter` preference to `true` for the `sh.sales` table only.
`SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS` `('sh', 'sales', estimate_percent=>10);` `PL/SQL procedure successfully completed.`	You attempt to set `estimate_percent` to `10` when gathering statistics for `sh.sales`. However, because `preference_overrides_parameter` is `true` for the `sales` table, and because a global preference is defined, Oracle Database actually gathers statistics using the global setting of `5`.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS` procedures for setting optimizer statistics

12.2.1.5 Setting Statistics Preferences: Example

This example illustrates the relationship between `SET_TABLE_PREFS`, `SET_SCHEMA_STATS`, and `SET_DATABASE_PREFS`.

Table 12-4 Changing Preferences for Statistics Gathering Procedures

Action	Description
`SQL> SELECT DBMS_STATS.GET_PREFS ('incremental', 'sh','costs') AS "STAT_PREFS" FROM DUAL;` `STAT_PREFS` `----------` `TRUE`	You query the `INCREMENTAL` preference for `costs` and determine that it is set to `true`.
`SQL> EXEC DBMS_STATS.SET_TABLE_PREFS ('sh', 'costs', 'incremental', 'false');` `PL/SQL procedure successfully completed.`	You use `SET_TABLE_PREFS` to set the `INCREMENTAL` preference to `false` for the `costs` table only.
`SQL> SELECT DBMS_STATS.GET_PREFS ('incremental', 'sh', 'costs') AS "STAT_PREFS" FROM DUAL;` `STAT_PREFS` `----------` `FALSE`	You query the `INCREMENTAL` preference for `costs` and confirm that it is set to `false`.
`SQL> EXEC DBMS_STATS.SET_SCHEMA_PREFS ('sh', 'incremental', 'true');` `PL/SQL procedure successfully completed.`	You use `SET_SCHEMA_PREFS` to set the `INCREMENTAL` preference to `true` for every table in the `sh` schema, including `costs`.
`SQL> SELECT DBMS_STATS.GET_PREFS ('incremental', 'sh', 'costs') AS "STAT_PREFS" FROM DUAL;` `STAT_PREFS` `----------` `TRUE`	You query the `INCREMENTAL` preference for `costs` and confirm that it is set to `true`.
`SQL> EXEC DBMS_STATS.SET_DATABASE_PREFS ('incremental', 'false');` `PL/SQL procedure successfully completed.`	You use `SET_DATABASE_PREFS` to set the `INCREMENTAL` preference for all tables in all user-defined schemas to `false`.

Table 12-4 (Cont.) Changing Preferences for Statistics Gathering Procedures

Action	Description

`SQL> SELECT DBMS_STATS.GET_PREFS ('incremental', 'sh', 'costs') AS "STAT_PREFS" FROM DUAL;`
 	
`STAT_PREFS`
`----------`
`FALSE`
	You query the `INCREMENTAL` preference for `costs` and confirm that it is set to `false`.

12.2.2 Setting Global Optimizer Statistics Preferences Using Cloud Control

A global preference applies to any object in the database that does *not* have an existing table preference. You can set optimizer statistics preferences at the global level using Cloud Control.

To set global optimizer statistics preferences using Cloud Control:

1. In Cloud Control, access the Database Home page.
2. From the **Performance** menu, select **SQL**, then **Optimizer Statistics**.

   The Optimizer Statistics Console appears.
3. Click **Global Statistics Gathering Options**.

   The Global Statistics Gathering Options page appears.
4. Make your desired changes, and click **Apply**.

See Also:

Online Help for Oracle Enterprise Manager Cloud Control

12.2.3 Setting Object-Level Optimizer Statistics Preferences Using Cloud Control

You can set optimizer statistics preferences at the database, schema, and table level using Cloud Control.

To set object-level optimizer statistics preferences using Cloud Control:

1. In Cloud Control, access the Database Home page.
2. From the **Performance** menu, select **SQL**, then **Optimizer Statistics**.

   The Optimizer Statistics Console appears.
3. Click **Object Level Statistics Gathering Preferences**.

   The Object Level Statistics Gathering Preferences page appears.

4. To modify table preferences for a table that has preferences set at the table level, do the following (otherwise, skip to the next step):
   a. Enter values in **Schema** and **Table Name**. Leave **Table Name** blank to see all tables in the schema.

      The page refreshes with the table names.
   b. Select the desired tables and click **Edit Preferences**.

      The General subpage of the Edit Preferences page appears.
   c. Change preferences as needed and click **Apply**.
5. To set preferences for a table that does *not* have preferences set at the table level, do the following (otherwise, skip to the next step):
   a. Click **Add Table Preferences**.

      The General subpage of the Add Table Preferences page appears.
   b. In **Table Name**, enter the schema and table name.
   c. Change preferences as needed and click **OK**.
6. To set preferences for a schema, do the following:
   a. Click **Set Schema Tables Preferences**.

      The General subpage of the Edit Schema Preferences page appears.
   b. In **Schema**, enter the schema name.
   c. Change preferences as needed and click **OK**.

See Also:

Online Help for Oracle Enterprise Manager Cloud Control

12.2.4 Setting Optimizer Statistics Preferences from the Command Line

If you do not use Cloud Control to set optimizer statistics preferences, then you can invoke the `DBMS_STATS` procedures from the command line.

Prerequisites

This task has the following prerequisites:

- To set the global or database preferences, you must have `SYSDBA` privileges, or both `ANALYZE ANY DICTIONARY` and `ANALYZE ANY` system privileges.
- To set schema preferences, you must connect as owner, or have `SYSDBA` privileges, or have the `ANALYZE ANY` system privilege.
- To set table preferences, you must connect as owner of the table or have the `ANALYZE ANY` system privilege.

To set optimizer statistics preferences from the command line:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Optionally, call the `DBMS_STATS.GET_PREFS` procedure to see preferences set at the object level, or at the global level if a specific table is not set.

   For example, obtain the `STALE_PERCENT` parameter setting for the `sh.sales` table as follows:

   ```
   SELECT DBMS_STATS.GET_PREFS('STALE_PERCENT', 'SH', 'SALES')
   FROM   DUAL;
   ```

3. Execute the appropriate procedure from Table 12-1, specifying the following parameters:
   - `ownname` - Set schema name (`SET_TAB_PREFS` and `SET_SCHEMA_PREFS` only)
   - `tabname` - Set table name (`SET_TAB_PREFS` only)
   - `pname` - Set parameter name
   - `pvalue` - Set parameter value
   - `add_sys` - Include system tables (optional, `SET_DATABASE_PREFS` only)

   The following example specifies that 13% of rows in `sh.sales` must change before the statistics on that table are considered stale:

   ```
   EXEC DBMS_STATS.SET_TABLE_PREFS('SH', 'SALES', 'STALE_PERCENT', '13');
   ```

4. Optionally, query the `*_TAB_STAT_PREFS` view to confirm the change.

   For example, query `DBA_TAB_STAT_PREFS` as follows:

   ```
   COL OWNER FORMAT a5
   COL TABLE_NAME FORMAT a15
   COL PREFERENCE_NAME FORMAT a20
   COL PREFERENCE_VALUE FORMAT a30
   SELECT * FROM DBA_TAB_STAT_PREFS;
   ```

   Sample output appears as follows:

   ```
   OWNER TABLE_NAME      PREFERENCE_NAME      PREFERENCE_VALUE
   ----- --------------- -------------------- -----------------------------
   OE    CUSTOMERS       NO_INVALIDATE        DBMS_STATS.AUTO_INVALIDATE
   SH    SALES           STALE_PERCENT        13
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference for descriptions of the parameter names and values for program units

12.3 Configuring Options for Dynamic Statistics

Dynamic statistics are an optimization technique in which the database uses recursive SQL to scan a small random sample of the blocks in a table.

The sample scan estimate predicate selectivities. Using these estimates, the database determines better default statistics for unanalyzed segments, and verifies its estimates. By default, when optimizer statistics are missing, stale, or insufficient, dynamic statistics automatically run recursive SQL during parsing to scan a small random sample of table blocks.

12.3.1 About Dynamic Statistics Levels

The **dynamic statistics level** controls both when the database gathers dynamic statistics, and the size of the sample that the optimizer uses to gather the statistics.

Set the dynamic statistics level using either the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter or a statement hint.

Note:

Dynamic statistics were called *dynamic sampling* in releases earlier than Oracle Database 12c Release 1 (12.1).

The following table describes the levels for dynamic statistics. Note the following:

- If dynamic statistics are enabled, then the database may choose to use dynamic statistics when a SQL statement uses parallel execution.
- If `OPTIMIZER_ADAPTIVE_STATISTICS` is `TRUE`, then the optimizer uses dynamic statistics when relevant SQL plan directives exist. The database maintains the resulting statistics in the SQL plan directives store, making them available to other queries.

Table 12-5 Dynamic Statistics Levels

Level	When the Optimizer Uses Dynamic Statistics	Sample Size (Blocks)
0	Do not use dynamic statistics.	n/a
1	Use dynamic statistics for all tables that do not have statistics, but only if the following criteria are met: • At least one nonpartitioned table in the query does not have statistics. • This table has no indexes. • This table has more blocks than the number of blocks that would be used for dynamic statistics of this table.	32
2	Use dynamic statistics if at least one table in the statement has no statistics. This is the default value.	64

Table 12-5 (Cont.) Dynamic Statistics Levels

Level	When the Optimizer Uses Dynamic Statistics	Sample Size (Blocks)
3	Use dynamic statistics if any of the following conditions is true: • At least one table in the statement has no statistics. • The statement has one or more expressions used in the `WHERE` clause predicates, for example, `WHERE SUBSTR(CUSTLASTNAME,1,3).`	64
4	Use dynamic statistics if any of the following conditions is true: • At least one table in the statement has no statistics. • The statement has one or more expressions used in the `WHERE` clause predicates, for example, `WHERE SUBSTR(CUSTLASTNAME,1,3).` • The statement uses complex predicates (an `OR` or `AND` operator between multiple predicates on the same table).	64
5	The criteria are identical to level 4, but the database uses a different sample size.	128
6	The criteria are identical to level 4, but the database uses a different sample size.	256
7	The criteria are identical to level 4, but the database uses a different sample size.	512
8	The criteria are identical to level 4, but the database uses a different sample size.	1024
9	The criteria are identical to level 4, but the database uses a different sample size.	4086
10	The criteria are identical to level 4, but the database uses a different sample size.	All blocks
11	The database uses adaptive dynamic sampling automatically when the optimizer deems it necessary.	Automatically determined

See Also:

- "When the Database Samples Data"
- *Oracle Database Reference* to learn about the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter

12.3.2 Setting Dynamic Statistics Levels Manually

Determining a database-level setting that would be beneficial to all SQL statements can be difficult.

When setting the level for dynamic statistics, Oracle recommends setting the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter at the session level.

Assumptions

This tutorial assumes the following:

- You want correct selectivity estimates for the following query, which has `WHERE` clause predicates on two correlated columns:

  ```
  SELECT *
  FROM   sh.customers
  WHERE  cust_city='Los Angeles'
  AND    cust_state_province='CA';
  ```

- The preceding query uses serial processing.
- The `sh.customers` table contains 932 rows that meet the conditions in the query.
- You have gathered statistics on the `sh.customers` table.
- You created an index on the `cust_city` and `cust_state_province` columns.
- The `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter is set to the default level of `2`.

To set the dynamic statistics level manually:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Explain the execution plan as follows:

   ```
   EXPLAIN PLAN FOR
     SELECT *
     FROM   sh.customers
     WHERE  cust_city='Los Angeles'
     AND    cust_state_province='CA';
   ```

3. Query the plan as follows:

   ```
   SET LINESIZE 130
   SET PAGESIZE 0
   SELECT *
   FROM   TABLE(DBMS_XPLAN.DISPLAY);
   ```

   The output appears below (the example has been reformatted to fit on the page):

```
-------------------------------------------------------------------------------
|Id| Operation                          | Name              |Rows|Bytes|Cost | Time    |
-------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |                   | 53|  9593|53(0)|00:00:01|
| 1|  TABLE ACCESS BY INDEX ROWID|CUSTOMERS          | 53|  9593|53(0)|00:00:01|
|*2|   INDEX RANGE SCAN          |CUST_CITY_STATE_IND| 53|  9593| 3(0)|00:00:01|
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("CUST_CITY"='Los Angeles' AND "CUST_STATE_PROVINCE"='CA')
```

   The columns in the `WHERE` clause have a real-world correlation, but the optimizer is not aware that Los Angeles is in California and assumes both predicates reduce the number of rows returned. Thus, the table contains 932 rows that meet the conditions, but the optimizer estimates 53, as shown in bold.

If the database had used dynamic statistics for this plan, then the `Note` section of the plan output would have indicated this fact. The optimizer did not use dynamic statistics because the statement executed serially, standard statistics exist, and the parameter `OPTIMIZER_DYNAMIC_SAMPLING` is set to the default of `2`.

4. Set the dynamic statistics level to `4` in the session using the following statement:

```
ALTER SESSION SET OPTIMIZER_DYNAMIC_SAMPLING=4;
```

5. Explain the plan again:

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   sh.customers
  WHERE  cust_city='Los Angeles'
  AND    cust_state_province='CA';
```

The new plan shows a more accurate estimate of the number of rows, as shown by the value 932 in bold:

```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------
Plan hash value: 2008213504

-------------------------------------------------------------------------
| Id  | Operation         | Name      |Rows |Bytes |Cost (%CPU)|Time    |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           | 932 | 271K|   406 (1)| 00:00:05 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS | 932 | 271K|   406 (1)| 00:00:05 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter("CUST_CITY"='Los Angeles' AND "CUST_STATE_PROVINCE"='CA')

Note
-----
   - dynamic statistics used for this statement (level=4)
```

The note at the bottom of the plan indicates that the sampling level is `4`. The additional dynamic statistics made the optimizer aware of the real-world relationship between the `cust_city` and `cust_state_province` columns, thereby enabling it to produce a more accurate estimate for the number of rows: 932 rather than 53.

See Also:

- *Oracle Database SQL Language Reference* to learn about setting sampling levels with the `DYNAMIC_SAMPLING` hint
- *Oracle Database Reference* to learn about the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter

12.3.3 Disabling Dynamic Statistics

In general, the best practice is not to incur the cost of dynamic statistics for queries whose compile times must be as fast as possible, for example, unrepeated OLTP queries.

To disable dynamic statistics at the session level:

1. Connect SQL*Plus to the database with the appropriate privileges.
2. Set the dynamic statistics level to `0`.

   For example, run the following statement:

   ```
   ALTER SESSION SET OPTIMIZER_DYNAMIC_SAMPLING=0;
   ```

See Also:

Oracle Database Reference to learn about the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter

12.4 Managing SQL Plan Directives

A **SQL plan directive** is additional information and instructions that the optimizer can use to generate a more optimal plan.

A directive informs the database that the optimizer is misestimate cardinalities of certain types of predicates, and alerts `DBMS_STATS` to gather additional statistics in the future. Thus, directives have an effect on statistics gathering.

The database automatically creates and manages SQL plan directives in the SGA, and then periodically writes them to the data dictionary. If the directives are not used within 53 weeks, then the database automatically purges them.

You can use `DBMS_SPD` procedures and functions to alter, save, drop, and transport directives manually. The following table lists some of the more commonly used procedures and functions.

Table 12-6 DBMS_SPD Procedures

Procedure	Description
`FLUSH_SQL_PLAN_DIRECTIVE`	Forces the database to write directives from memory to persistent storage in the `SYSAUX` tablespace.
`DROP_SQL_PLAN_DIRECTIVE`	Drops a SQL plan directive. If a directive that triggers dynamic sampling is creating unacceptable performance overhead, then you may want to remove it manually. If a SQL plan directive is dropped manually or automatically, then the database can re-create it. To prevent its re-creation, you can use `DBMS_SPM.ALTER_SQL_PLAN_DIRECTIVE` to do the following: • Disable the directive by setting `ENABLED` to `NO` • Prevent the directive from being dropped by setting `AUTO_DROP` to `NO` To disable SQL plan directives, set `OPTIMIZER_ADAPTIVE_STATISTICS` to `FALSE`.

Prerequisites

You must have the Administer SQL Management Object privilege to execute the `DBMS_SPD` APIs.

Assumptions

This tutorial assumes that you want to do the following:

- Write all directives for the `sh` schema to persistent storage.
- Delete all directives for the `sh` schema.

To write and then delete all sh schema plan directives:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Force the database to write the SQL plan directives to disk.

   For example, execute the following `DBMS_SPD` program:

   ```
   BEGIN
     DBMS_SPD.FLUSH_SQL_PLAN_DIRECTIVE;
   END;
   /
   ```

3. Query the data dictionary for information about existing directives in the `sh` schema.

   Example 12-3 queries the data dictionary for information about the directive.
4. Delete the existing SQL plan directive for the `sh` schema.

   The following PL/SQL program unit deletes the SQL plan directive with the ID `1484026771529551585`:

   ```
   BEGIN
     DBMS_SPD.DROP_SQL_PLAN_DIRECTIVE ( directive_id =>
   ```

```
                  14840267715295515855 );
                  END;
                  /
```

Example 12-3   Display Directives for sh Schema

This example shows SQL plan directives, and the results of SQL plan directive dynamic sampling queries.

```
SELECT TO_CHAR(d.DIRECTIVE_ID) dir_id, o.OWNER, o.OBJECT_NAME,
       o.SUBOBJECT_NAME col_name, o.OBJECT_TYPE object, d.TYPE,
       d.STATE, d.REASON
FROM   DBA_SQL_PLAN_DIRECTIVES d, DBA_SQL_PLAN_DIR_OBJECTS o
WHERE  d.DIRECTIVE_ID=o.DIRECTIVE_ID
AND    o.OWNER IN ('SH')
ORDER BY 1,2,3,4,5;

DIR_ID              OWN OBJECT_NA COL_NAME   OBJECT  TYPE     STATE      REASON
------------------- --- --------- ---------- ------- -------- ---------- ------------
1484026771529551585  SH CUSTOMERS COUNTRY_ID  COLUMN DYNAMIC_ SUPERSEDED SINGLE
TABLE
                                                     SAMPLING            CARDINALITY
                                                                         MISESTIMATE
1484026771529551585  SH CUSTOMERS CUST_STATE_ COLUMN DYNAMIC_ SUPERSEDED SINGLE TABLE
                                  PROVINCE           SAMPLING            CARDINALITY
                                                                         MISESTIMATE
1484026771529551585  SH CUSTOMERS              TABLE DYNAMIC_ SUPERSEDED SINGLE TABLE
                                                     SAMPLING            CARDINALITY
                                                                         MISESTIMATE
9781501826140511330  SH dyg4msnst5           SQL STA DYNAMIC_     USABLE VERIFY
                                             TEMENT  SAMPLING            CARDINALITY
                                                     _RESULT             ESTIMATE
9872337207064898539  SH TIMES                  TABLE DYNAMIC_     USABLE VERIFY
                                                     SAMPLING            CARDINALITY
                                                     _RESULT             ESTIMATE
9781501826140511330  SH 2nk1v0fdx0           SQL STA DYNAMIC_     USABLE VERIFY
                                             TEMENT  SAMPLING            CARDINALITY
                                                     _RESULT             ESTIMATE
```

See Also:

- "SQL Plan Directives"
- *Oracle Database PL/SQL Packages and Types Reference* for complete syntax and semantics for the `DBMS_SPD` package.
- *Oracle Database Reference* to learn about `DBA_SQL_PLAN_DIRECTIVES`

13
Gathering Optimizer Statistics

This chapter explains how to use the `DBMS_STATS.GATHER_*_STATS` program units.

See Also:

- "Optimizer Statistics Concepts"
- "Query Optimizer Concepts "
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_STATS.GATHER_TABLE_STATS`

13.1 Configuring Automatic Optimizer Statistics Collection

Oracle Database can gather optimizer statistics automatically.

13.1.1 About Automatic Optimizer Statistics Collection

The automated maintenance tasks infrastructure (known as AutoTask) schedules tasks to run automatically in Oracle Scheduler windows known as maintenance windows.

Difference Between Automatic and Manual Statistics Collection

The principal difference is that automatic collection prioritizes database objects that need statistics. Before the maintenance window closes, automatic collection assesses all objects and prioritizes objects that have no statistics or very old statistics.

When gathering statistics manually, you can reproduce the object prioritization of automatic collection by using the `DBMS_AUTO_TASK_IMMEDIATE` package. This package runs the same statistics gathering job that is executed during the automatic nightly statistics gathering job.

How Automatic Statistics Collection Works

Automatic optimizer statistics collection runs as part of AutoTask. By default, the collection runs in all predefined maintenance windows. One window is scheduled for each day of the week.

To collect the optimizer statistics, the database calls an internal procedure that operates similarly to the `GATHER_DATABASE_STATS` procedure with the `GATHER AUTO` option. Automatic statistics collection honors all preferences set in the database.

When an automatic optimizer statistics collection task gathers data for a PDB, it stores this data in the PDB. This data is included if the PDB is unplugged. A common user whose current container is the CDB root can view optimizer statistics data for PDBs. A user whose current container is a PDB can view optimizer statistics data for the PDB only.

13.1.2 Configuring Automatic Optimizer Statistics Collection Using Cloud Control

You can enable and disable all automatic maintenance tasks, including automatic optimizer statistics collection, using Cloud Control.

The default window timing works well for most situations. However, you may have operations such as bulk loads that occur during the window. In such cases, to avoid potential conflicts that result from operations occurring at the same time as automatic statistics collection, Oracle recommends that you change the window accordingly.

Prerequisites

Access the Database Home page, as described in "Accessing the Database Home Page in Cloud Control."

To control automatic optimizer statistics collection using Cloud Control:

1. From the **Administration** menu, select **Oracle Scheduler**, then **Automated Maintenance Tasks**.

   The Automated Maintenance Tasks page appears.

   This page shows the predefined tasks. To retrieve information about each task, click the corresponding link for the task.

2. Click **Configure**.

   The Automated Maintenance Tasks Configuration page appears.

   By default, automatic optimizer statistics collection executes in all predefined maintenance windows in `MAINTENANCE_WINDOW_GROUP`.

3. Perform the following steps:

   a. In the Task Settings section for Optimizer Statistics Gathering, select either **Enabled** or **Disabled** to enable or disable an automated task.

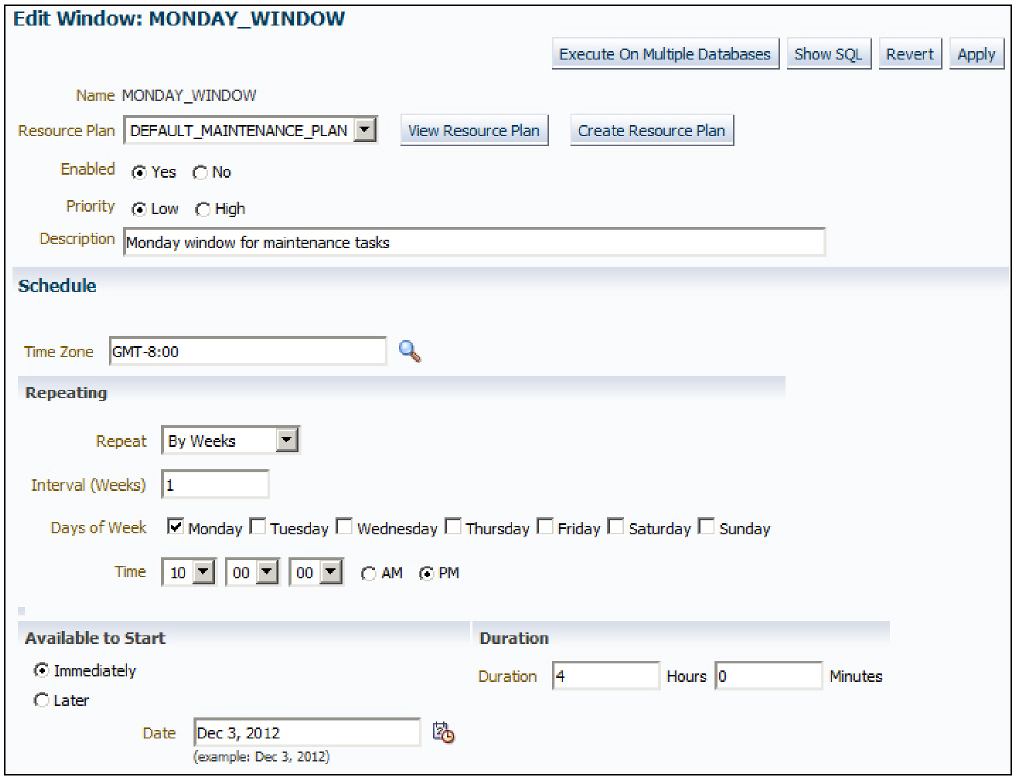
   > **Note:**
   >
   > Oracle strongly recommends that you not disable automatic statistics gathering because it is critical for the optimizer to generate optimal plans for queries against dictionary and user objects. If you disable automatic collection, ensure that you have a good manual statistics collection strategy for dictionary and user schemas.

   b. To disable statistics gathering for specific days in the week, check the appropriate box next to the window name.

   c. To change the characteristics of a window group, click **Edit Window Group**.

   d. To change the times for a window, click the name of the window (for example, **MONDAY_WINDOW**), and then in the Schedule section, click **Edit**.

   The Edit Window page appears.



Name MONDAY_WINDOW
Resource Plan DEFAULT_MAINTENANCE_PLAN
Enabled [✓ Yes] No
Priority [✓ Low] High
Description Monday window for maintenance tasks
Time Zone GMT-8:00
Repeat By Weeks
Interval (Weeks) 1
Days of Week [✓ Monday] Tuesday Wednesday Thursday Friday Saturday Sunday
Time 10 00 00 AM [✓ PM]
[✓ Immediately] Later
Date Dec 3, 2012 (example: Dec 3, 2012)
Duration 4 Hours 0 Minutes

In this page, you can change the parameters such as duration and start time for window execution.

e. Click **Apply**.

See Also:

Online Help for Oracle Enterprise Manager Cloud Control

13.1.3 Configuring Automatic Optimizer Statistics Collection from the Command Line

If you do not use Cloud Control to configure automatic optimizer statistics collection, then you must use the command line.

You have the following options:

- Run the `ENABLE` or `DISABLE` procedure in the `DBMS_AUTO_TASK_ADMIN` PL/SQL package.

  This package is the recommended command-line technique. For both the `ENABLE` and `DISABLE` procedures, you can specify a particular maintenance window with the `window_name` parameter.

- Set the `STATISTICS_LEVEL` initialization level to `BASIC` to disable collection of *all* advisories and statistics, including Automatic SQL Tuning Advisor.

Note:

Because monitoring and many automatic features are disabled, Oracle strongly recommends that you do not set `STATISTICS_LEVEL` to `BASIC`.

To control automatic statistics collection using DBMS_AUTO_TASK_ADMIN:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Do one of the following:
   - To enable the automated task, execute the following PL/SQL block:

     ```
     BEGIN
       DBMS_AUTO_TASK_ADMIN.ENABLE (
         client_name  => 'auto optimizer stats collection'
     ,   operation    => NULL
     ,   window_name  => NULL
     );
     END;
     /
     ```

   - To disable the automated task, execute the following PL/SQL block:

     ```
     BEGIN
       DBMS_AUTO_TASK_ADMIN.DISABLE (
         client_name  => 'auto optimizer stats collection'
     ,   operation    => NULL
     ,   window_name  => NULL
     );
     END;
     /
     ```

3. Query the data dictionary to confirm the change.

   For example, query `DBA_AUTOTASK_CLIENT` as follows:

   ```
   COL CLIENT_NAME FORMAT a31

   SELECT CLIENT_NAME, STATUS
   FROM   DBA_AUTOTASK_CLIENT
   WHERE  CLIENT_NAME = 'auto optimizer stats collection';
   ```

   Sample output appears as follows:

   ```
   CLIENT_NAME                     STATUS
   ------------------------------- --------
   auto optimizer stats collection ENABLED
   ```

To change the window attributes for automatic statistics collection:

1. Connect SQL*Plus to the database with administrator privileges.

2. Change the attributes of the maintenance window as needed.

   For example, to change the Monday maintenance window so that it starts at 5 a.m., execute the following PL/SQL program:

```
BEGIN
  DBMS_SCHEDULER.SET_ATTRIBUTE (
    'MONDAY_WINDOW'
,   'repeat_interval'
,   'freq=daily;byday=MON;byhour=05;byminute=0;bysecond=0'
);
END;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_AUTO_TASK_ADMIN` package
- *Oracle Database Reference* to learn about the `STATISTICS_LEVEL` initialization parameter

13.2 Configuring High-Frequency Automatic Optimizer Statistics Collection

This lightweight task supplements standard automatic statistics collection.

13.2.1 About High-Frequency Automatic Optimizer Statistics Collection

You can configure automatic statistics collection to occur more frequently.

Purpose of High-Frequency Automatic Optimizer Statistics Collection

AutoTask schedules tasks to run automatically in maintenance windows. By default, one window is scheduled for each day of the week. Automatic optimizer statistics collection (`DBMS_STATS`) runs in all predefined maintenance windows.

Statistics can go stale between two consecutive statistics collection tasks. If data changes frequently, the stale statistics could cause performance problems. For example, a brokerage company might receive tremendous data during trading hours, leading the optimizer to use stale statistics for queries executed during this period.

High-frequency automatic optimizer statistics collection complements the standard statistics collection job. By default, the collection occurs every 15 minutes, meaning that statistics have less time in which to be stale.

How High-Frequency Automatic Optimizer Statistics Collection Works

To enable and disable the high-frequency task, set the execution interval, and set the maximum run time, use the `DBMS_STATS.SET_GLOBAL_PREFS` procedure. The high-frequency task is “lightweight” and only gathers stale statistics. It does not perform actions such as purging statistics for non-existent objects or invoking Optimizer Statistics Advisor. The standard automated job performs these additional tasks.

Automatic statistics collection jobs that run in the maintenance window are not affected by the high-frequency jobs. The high-frequency task may execute in maintenance windows, but it will not execute while the maintenance window auto stats gathering job is executing. You can monitor the tasks by querying `DBA_AUTO_STAT_EXECUTIONS`.

13.2.2 Setting Preferences for High-Frequency Automatic Optimizer Statistics Collection

To enable and disable the task, use `DBMS_STATS.SET_GLOBAL_PREFS`.

You can use `DBMS_STATS.SET_GLOBAL_PREFS` to set preferences to any of the following values:

- `AUTO_TASK_STATUS`

  Enables or disables the high-frequency automatic optimizer statistics collection. Values are:

  - `ON` — Enables high-frequency automatic optimizer statistics collection.
  - `OFF` — Disables high-frequency automatic optimizer statistics collection. This is the default.

- `AUTO_TASK_MAX_RUN_TIME`

  Configures the maximum run time in seconds of an execution of high-frequency automatic optimizer statistics collection. The maximum value is `3600` (equal to 1 hour), which is the default.

- `AUTO_TASK_INTERVAL`

  Specifies the interval in seconds between executions of high-frequency automatic optimizer statistics collection. The minimum value is `60`. The default is `900` (equal to 15 minutes).

To configure the high-frequency task, you must have administrator privileges.

To configure the high-frequency task:

1. Log in to the database as a user with administrator privileges.
2. To enable the high-frequency task, set the `AUTO_TASK_STATUS` preference to `ON`.

   The following example enables the automatic task:

   ```
   EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_STATUS','ON');
   ```

3. To set the maximum run time, set the `AUTO_TASK_MAX_RUN_TIME` preference to the desired number of seconds.

   The following example sets the maximum run time to 10 minutes:

   ```
   EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_MAX_RUN_TIME','600');
   ```

4. To set the frequency, set the `AUTO_TASK_INTERVAL` preference to the desired number of seconds.

The following example sets the frequency to 8 minutes:

```
EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_INTERVAL','240');
```

13.2.3 High-Frequency Automatic Optimizer Statistics Collection: Example

In this example, you enable run DML statements, and then enable the high-frequency statistics collection job.

This example assumes the following:

- You are logged in to the database as an administrator.
- The statistics for the `sh` schema are fresh.
- High-frequency automatic optimizer statistics collection

  is not enabled.

1. Query the data dictionary for the statistics for the `sales` and `customers` tables (sample output included):

```
SET LINESIZE 170
SET PAGESIZE 5000
COL TABLE_NAME FORMAT a20
COL PARTITION_NAME FORMAT a20
COL NUM_ROWS FORMAT 9999999
COL STALE_STATS FORMAT a3

SELECT TABLE_NAME, PARTITION_NAME, NUM_ROWS, STALE_STATS
FROM   DBA_TAB_STATISTICS
WHERE  OWNER = 'SH'
AND    TABLE_NAME IN ('CUSTOMERS','SALES')
ORDER BY TABLE_NAME, PARTITION_NAME;

TABLE_NAME           PARTITION_NAME       NUM_ROWS STA
-------------------- -------------------- -------- ---
CUSTOMERS                                    55500 NO
SALES                SALES_1995                  0 NO
SALES                SALES_1996                  0 NO
SALES                SALES_H1_1997               0 NO
SALES                SALES_H2_1997               0 NO
SALES                SALES_Q1_1998           43687 NO
SALES                SALES_Q1_1999           64186 NO
SALES                SALES_Q1_2000           62197 NO
SALES                SALES_Q1_2001           60608 NO
SALES                SALES_Q1_2002               0 NO
SALES                SALES_Q1_2003               0 NO
SALES                SALES_Q2_1998           35758 NO
SALES                SALES_Q2_1999           54233 NO
SALES                SALES_Q2_2000           55515 NO
SALES                SALES_Q2_2001           63292 NO
SALES                SALES_Q2_2002               0 NO
SALES                SALES_Q2_2003               0 NO
SALES                SALES_Q3_1998           50515 NO
SALES                SALES_Q3_1999           67138 NO
SALES                SALES_Q3_2000            58950 NO
SALES                SALES_Q3_2001            65769 NO
SALES                SALES_Q3_2002                0 NO
SALES                SALES_Q3_2003                0 NO
SALES                SALES_Q4_1998            48874 NO
SALES                SALES_Q4_1999            62388 NO
SALES                SALES_Q4_2000            55984 NO
SALES                SALES_Q4_2001            69749 NO
SALES                SALES_Q4_2002                0 NO
SALES                SALES_Q4_2003                0 NO
SALES                                        918843 NO
```

The preceding output shows that none of the statistics are stale.

2. Perform DML on `sales` and `customers`:

```
-- insert 918K rows in sales
INSERT INTO sh.sales SELECT * FROM sh.sales;
-- update around 15% of sales rows
UPDATE sh.sales SET amount_sold = amount_sold + 1 WHERE amount_sold
> 100;
-- insert 1 row into customers
INSERT INTO sh.customers(cust_id, cust_first_name, cust_last_name,
    cust_gender, cust_year_of_birth, cust_main_phone_number,
    cust_street_address, cust_postal_code, cust_city_id,
    cust_city, cust_state_province_id, cust_state_province,
    country_id, cust_total, cust_total_id)
  VALUES(188710, 'Jenny', 'Smith', 'F', '1966', '555-111-2222',
    '400 oracle parkway','94065',51402, 'Redwood Shores',
    52564, 'CA', 52790, 'Customer total', '52772');
COMMIT;
```

The total number of `sales` rows increased by 100%, but only 1 row was added to `customers`.

3. Save the optimizer statistics to disk:

```
EXEC DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO;
```

4. Query the table statistics again (sample output included):

```
SELECT TABLE_NAME, PARTITION_NAME, NUM_ROWS, STALE_STATS
FROM   DBA_TAB_STATISTICS
WHERE  OWNER = 'SH'
AND    TABLE_NAME IN ('CUSTOMERS','SALES')
ORDER BY TABLE_NAME, PARTITION_NAME;

TABLE_NAME           PARTITION_NAME        NUM_ROWS STA
-------------------- -------------------- --------- ---
CUSTOMERS                                     55500 NO
SALES                SALES_1995                   0 NO
SALES                SALES_1996                   0 NO
SALES                SALES_H1_1997                0 NO
SALES                SALES_H2_1997                0 NO
SALES                SALES_Q1_1998          43687 YES
SALES                SALES_Q1_1999          64186 YES
SALES                SALES_Q1_2000          62197 YES
SALES                SALES_Q1_2001          60608 YES
SALES                SALES_Q1_2002              0 NO
SALES                SALES_Q1_2003              0 NO
SALES                SALES_Q2_1998          35758 YES
SALES                SALES_Q2_1999          54233 YES
SALES                SALES_Q2_2000          55515 YES
SALES                SALES_Q2_2001          63292 YES
SALES                SALES_Q2_2002              0 NO
SALES                SALES_Q2_2003              0 NO
SALES                SALES_Q3_1998          50515 YES
SALES                SALES_Q3_1999          67138 YES
SALES                SALES_Q3_2000          58950 YES
SALES                SALES_Q3_2001          65769 YES
SALES                SALES_Q3_2002              0 NO
SALES                SALES_Q3_2003              0 NO
SALES                SALES_Q4_1998          48874 YES
SALES                SALES_Q4_1999          62388 YES
SALES                SALES_Q4_2000          55984 YES
SALES                SALES_Q4_2001          69749 YES
SALES                SALES_Q4_2002              0 NO
SALES                SALES_Q4_2003              0 NO
SALES                                     1837686
SALES                                      918843 YES

31 rows selected.
```

The preceding output shows that the statistics are not stale for `customers` but are stale for `sales`.

5. Configure high-frequency automatic optimizer statistics collection:

```
EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_STATUS','ON');
EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_MAX_RUN_TIME','180');
EXEC DBMS_STATS.SET_GLOBAL_PREFS('AUTO_TASK_INTERVAL','240');
```

The preceding PL/SQL programs enable high-frequency collection, set the maximum run time to 3 minutes, and set the task execution interval to 4 minutes.

6. Wait for a few minutes, and then query the data dictionary:

```
COL OPID FORMAT 9999
COL STATUS FORMAT a11
COL ORIGIN FORMAT a20
COL COMPLETED FORMAT 99999
COL FAILED FORMAT 99999
COL TIMEOUT FORMAT 99999
COL INPROG FORMAT 99999

SELECT OPID, ORIGIN, STATUS, TO_CHAR(START_TIME, 'DD/MM HH24:MI:SS' ) AS 
BEGIN_TIME,
       TO_CHAR(END_TIME, 'DD/MM HH24:MI:SS') AS END_TIME, COMPLETED, 
FAILED,
       TIMED_OUT AS TIMEOUT, IN_PROGRESS AS INPROG
FROM  DBA_AUTO_STAT_EXECUTIONS
ORDER BY OPID;
```

The output shows that the high-frequency job executed twice, and the standard automatic statistics collection job executed once:

```
ID  ORIGIN               STATUS   BEGIN_TIME     END_TIME       COMP FAIL TIMEO
INPRO
--- -------------------- -------- -------------- -------------- ---- ---- -----
-----
790 HIGH_FREQ_AUTO_TASK  COMPLETE 03/10 14:54:02 03/10 14:54:35  338    3
0     0
793 HIGH_FREQ_AUTO_TASK  COMPLETE 03/10 14:58:11 03/10 14:58:45  193    3
0     0
794 AUTO_TASK            COMPLETE 03/10 15:00:02 03/10 15:00:20   52    3
0     0
```

13.3 Gathering Optimizer Statistics Manually

As an alternative or supplement to automatic statistics gathering, you can use the `DBMS_STATS` package to gather optimizer statistics manually.

See Also:

- "Configuring Automatic Optimizer Statistics Collection"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` package

13.3.1 About Manual Statistics Collection with DBMS_STATS

Use the `DBMS_STATS` package to manipulate optimizer statistics. You can gather statistics on objects and columns at various levels of granularity: object, schema, and database. You can also gather statistics for the physical system.

The following table summarizes the `DBMS_STATS` procedures for gathering optimizer statistics. This package does not gather statistics for table clusters. However, you can gather statistics on individual tables in a table cluster.

Table 13-1 DBMS_STATS Procedures for Gathering Optimizer Statistics

Procedure	Purpose
`GATHER_INDEX_STATS`	Collects index statistics
`GATHER_TABLE_STATS`	Collects table, column, and index statistics
`GATHER_SCHEMA_STATS`	Collects statistics for all objects in a schema

Table 13-1 (Cont.) DBMS_STATS Procedures for Gathering Optimizer Statistics

Procedure	Purpose
`GATHER_DICTIONARY_STATS`	Collects statistics for all system schemas, including `SYS` and `SYSTEM`, and other optional schemas, such as `CTXSYS` and `DRSYS`
`GATHER_DATABASE_STATS`	Collects statistics for all objects in a database

When the `OPTIONS` parameter is set to `GATHER STALE` or `GATHER AUTO`, the `GATHER_SCHEMA_STATS` and `GATHER_DATABASE_STATS` procedures gather statistics for any table that has stale statistics and any table that is missing statistics. If a monitored table has been modified more than 10%, then the database considers these statistics stale and gathers them again.

Note:

As explained in "Configuring Automatic Optimizer Statistics Collection", you can configure a nightly job to gather statistics automatically.

See Also:

- "Gathering System Statistics Manually"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_STATS` package

13.3.2 Guidelines for Gathering Optimizer Statistics Manually

In most cases, automatic statistics collection is sufficient for database objects modified at a moderate speed.

Automatic collection may sometimes be inadequate or unavailable, as shown in the following table.

Table 13-2 Reasons for Gathering Statistics Manually

Issue	To Learn More
You perform certain types of bulk load and cannot wait for the maintenance window to collect statistics because queries must be executed immediately.	"Online Statistics Gathering for Bulk Loads"
During a nonrepresentative workload, automatic statistics collection gathers statistics for fixed tables.	"Gathering Statistics for Fixed Objects"
Automatic statistics collection does not gather system statistics.	"Gathering System Statistics Manually"

Table 13-2 (Cont.) Reasons for Gathering Statistics Manually

Issue	To Learn More
Volatile tables are being deleted or truncated, and then rebuilt during the day.	"Gathering Statistics for Volatile Tables Using Dynamic Statistics"

13.3.2.1 Guideline for Setting the Sample Size

In the context of optimizer statistics, **sampling** is the gathering of statistics from a random subset of table rows. By enabling the database to avoid full table scans and sorts of entire tables, sampling minimizes the resources necessary to gather statistics.

The database gathers the most accurate statistics when it processes all rows in the table, which is a 100% sample. However, larger sample sizes increase the time of statistics gathering operations. The challenge is determining a sample size that provides accurate statistics in a reasonable time.

`DBMS_STATS` uses sampling when a user specifies the parameter `ESTIMATE_PERCENT`, which controls the percentage of the rows in the table to sample. To maximize performance gains while achieving necessary statistical accuracy, Oracle recommends that the `ESTIMATE_PERCENT` parameter use the default setting of `DBMS_STATS.AUTO_SAMPLE_SIZE`. In this case, Oracle Database chooses the sample size automatically. This setting enables the use of the following:

- A hash-based algorithm that is much faster than sampling

  This algorithm reads all rows and produces statistics that are nearly as accurate as statistics from a 100% sample. The statistics computed using this technique are deterministic.
- Incremental statistics
- Concurrent statistics
- New histogram types

The `DBA_TABLES.SAMPLE_SIZE` column indicates the actual sample size used to gather statistics.

See Also:

- "Hybrid Histograms"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS.AUTO_SAMPLE_SIZE`

13.3.2.2 Guideline for Gathering Statistics in Parallel

By default, the database gathers statistics with the parallelism degree specified at the table or index level.

You can override this setting with the `degree` argument to the `DBMS_STATS` gathering procedures. Oracle recommends setting `degree` to `DBMS_STATS.AUTO_DEGREE`. This setting enables the database to choose an appropriate degree of parallelism based on the object size and the settings for the parallelism-related initialization parameters.

The database can gather most statistics serially or in parallel. However, the database does not gather some index statistics in parallel, including cluster indexes, domain indexes, and bitmap join indexes. The database can use sampling when gathering parallel statistics.

Note:

Do not confuse gathering statistics in parallel with gathering statistics concurrently.

See Also:

- "About Concurrent Statistics Gathering"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS.AUTO_DEGREE`

13.3.2.3 Guideline for Partitioned Objects

For partitioned tables and indexes, `DBMS_STATS` can gather separate statistics for each partition and global statistics for the entire table or index.

Similarly, for composite partitioning, `DBMS_STATS` can gather separate statistics for subpartitions, partitions, and the entire table or index.

To determine the type of partitioning statistics to be gathered, specify the `granularity` argument to the `DBMS_STATS` procedures. Oracle recommends setting `granularity` to the default value of `AUTO` to gather subpartition, partition, or global statistics, depending on partition type. The `ALL` setting gathers statistics for all types.

See Also:

"Gathering Incremental Statistics on Partitioned Objects"

13.3.2.4 Guideline for Frequently Changing Objects

When tables are frequently modified, gather statistics often enough so that they do not go stale, but not so often that collection overhead degrades performance.

You may only need to gather new statistics every week or month. The best practice is to use a script or job scheduler to regularly run the `DBMS_STATS.GATHER_SCHEMA_STATS` and `DBMS_STATS.GATHER_DATABASE_STATS` procedures.

13.3.2.5 Guideline for External Tables

Because the database does not permit data manipulation against external tables, the database never marks statistics on external tables as stale. If new statistics are required for an external table, for example, because the underlying data files change, then regather the statistics.

For external tables, use the same `DBMS_STATS` procedures that you use for internal tables. Note that the `scanrate` parameter of `DBMS_STATS.SET_TABLE_STATS` and `DBMS_STATS.GET_TABLE_STATS` specifies the rate (in MB/s) at which Oracle Database scans data in tables, and is relevant only for external tables. The `SCAN_RATE` column appears in the `DBA_TAB_STATISTICS` and `DBA_TAB_PENDING_STATS` data dictionary views.

See Also:

- "Creating Artificial Optimizer Statistics for Testing"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `SET_TABLE_STATS` and `GET_TABLE_STATS`
- *Oracle Database Reference* to learn about the `DBA_TAB_STATISTICS` view

13.3.3 Determining When Optimizer Statistics Are Stale

Stale statistics on a table do not accurately reflect its data. To help you determine when a database object needs new statistics, the database provides a table monitoring facility.

Monitoring tracks the approximate number of DML operations on a table and whether the table has been truncated since the most recent statistics collection. To check whether statistics are stale, query the `STALE_STATS` column in `DBA_TAB_STATISTICS` and `DBA_IND_STATISTICS`. This column is based on data in the `DBA_TAB_MODIFICATIONS` view and the `STALE_PERCENT` preference for `DBMS_STATS`.

Note:

Starting in Oracle Database 12c Release 2 (12.2), you no longer need to use `DBMS_STATS.FLUSH_DATABASE_MONITORING_INFO` to ensure that view metadata is current. The statistics shown in the `DBA_TAB_STATISTICS`, `DBA_IND_STATISTICS`, and `DBA_TAB_MODIFICATIONS` views are obtained from both disk and memory.

The `STALE_STATS` column has the following possible values:

- `YES`

  The statistics are stale.
- `NO`

The statistics are not stale.

- null

  The statistics are not collected.

Executing GATHER_SCHEMA_STATS or GATHER_DATABASE_STATS with the GATHER AUTO option collects statistics only for objects with no statistics or stale statistics.

To determine stale statistics:

1. Start SQL*Plus, and then log in to the database as a user with the necessary privileges.
2. Query the data dictionary for stale statistics.

   The following example queries stale statistics for the sh.sales table (partial output included):

   ```
   COL PARTITION_NAME FORMAT a15

   SELECT PARTITION_NAME, STALE_STATS
   FROM   DBA_TAB_STATISTICS
   WHERE  TABLE_NAME = 'SALES'
   AND    OWNER = 'SH'
   ORDER BY PARTITION_NAME;

   PARTITION_NAME  STA
   --------------- ---
   SALES_1995      NO
   SALES_1996      NO
   SALES_H1_1997   NO
   SALES_H2_1997   NO
   SALES_Q1_1998   NO
   SALES_Q1_1999   NO
   .
   .
   .
   ```

See Also:

Oracle Database Reference to learn about the DBA_TAB_MODIFICATIONS view

13.3.4 Gathering Schema and Table Statistics

Use GATHER_TABLE_STATS to collect table statistics, and GATHER_SCHEMA_STATS to collect statistics for all objects in a schema.

To gather schema statistics using DBMS_STATS:

1. Start SQL*Plus, and connect to the database with the appropriate privileges for the procedure that you intend to run.
2. Run the GATHER_TABLE_STATS or GATHER_SCHEMA_STATS procedure, specifying the desired parameters.

Typical parameters include:

- Owner - `ownname`
- Object name - `tabname`, `indname`, `partname`
- Degree of parallelism - `degree`

Example 13-1   Gathering Statistics for a Table

This example uses the `DBMS_STATS` package to gather statistics on the `sh.customers` table with a parallelism setting of `2`.

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
    ownname => 'sh'
,   tabname => 'customers'
,   degree  => 2
);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `GATHER_TABLE_STATS` procedure

13.3.5 Gathering Statistics for Fixed Objects

Fixed objects are dynamic performance tables and their indexes. These objects record current database activity.

Unlike other database tables, the database does not automatically use dynamic statistics for SQL statement referencing `X$` tables when optimizer statistics are missing. Instead, the optimizer uses predefined default values. These defaults may not be representative and could potentially lead to a suboptimal execution plan. Thus, it is important to keep fixed object statistics current.

Oracle Database automatically gathers fixed object statistics as part of automated statistics gathering if they have not been previously collected. You can also manually collect statistics on fixed objects by calling `DBMS_STATS.GATHER_FIXED_OBJECTS_STATS`. Oracle recommends that you gather statistics when the database has representative activity.

Prerequisites

You must have the `SYSDBA` or `ANALYZE ANY DICTIONARY` system privilege to execute this procedure.

To gather schema statistics using GATHER_FIXED_OBJECTS_STATS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.

2. Run the DBMS_STATS.GATHER_FIXED_OBJECTS_STATS procedure, specifying the desired parameters.

   Typical parameters include:

   - Table identifier describing where to save the current statistics - stattab
   - Identifier to associate with these statistics within stattab (optional) - statid
   - Schema containing stattab (if different from current schema) - statown

Example 13-2 Gathering Statistics for a Table

This example uses the DBMS_STATS package to gather fixed object statistics.

```
BEGIN
  DBMS_STATS.GATHER_FIXED_OBJECTS_STATS;
END;
/
```

See Also:

- "Configuring Automatic Optimizer Statistics Collection"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the GATHER_TABLE_STATS procedure

13.3.6 Gathering Statistics for Volatile Tables Using Dynamic Statistics

Statistics for volatile tables, which are tables modified significantly during the day, go stale quickly. For example, a table may be deleted or truncated, and then rebuilt.

When you set the statistics of a volatile object to null, Oracle Database dynamically gathers the necessary statistics during optimization using dynamic statistics. The OPTIMIZER_DYNAMIC_SAMPLING initialization parameter controls this feature.

Assumptions

This tutorial assumes the following:

- The oe.orders table is extremely volatile.
- You want to delete and then lock the statistics on the orders table to prevent the database from gathering statistics on the table. In this way, the database can dynamically gather necessary statistics as part of query optimization.
- The oe user has the necessary privileges to query DBMS_XPLAN.DISPLAY_CURSOR.

To delete and the lock optimizer statistics:

1. Connect to the database as user oe, and then delete the statistics for the oe table.

   For example, execute the following procedure:

```
BEGIN
  DBMS_STATS.DELETE_TABLE_STATS('OE','ORDERS');
```

```
END;
/
```

2. Lock the statistics for the `oe` table.

   For example, execute the following procedure:

```
BEGIN
  DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
END;
/
```

3. You query the `orders` table.

   For example, use the following statement:

```
SELECT COUNT(order_id) FROM orders;
```

4. You query the plan in the cursor.

   You run the following commands (partial output included):

```
SET LINESIZE 150
SET PAGESIZE 0

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

SQL_ID  aut9632fr3358, child number 0
-------------------------------------
SELECT COUNT(order_id) FROM orders

Plan hash value: 425895392

--------------------------------------------------------------------
-
| Id  | Operation          | Name   | Rows  | Cost (%CPU)| Time
|
--------------------------------------------------------------------
-
|   0 | SELECT STATEMENT   |        |       |     2 (100)|
|
|   1 |  SORT AGGREGATE    |        |     1 |            |
|
|   2 |   TABLE ACCESS FULL| ORDERS |   105 |     2   (0)| 00:00:01
|
--------------------------------------------------------------------
-

Note
-----
   - dynamic statistics used for this statement (level=2)
```

The Note in the preceding execution plan shows that the database used dynamic statistics for the `SELECT` statement.

See Also:

- "Configuring Options for Dynamic Statistics"
- "Locking and Unlocking Optimizer Statistics" to learn how to gather representative statistics and then lock them, which is an alternative technique for preventing statistics for volatile tables from going stale

13.3.7 Gathering Optimizer Statistics Concurrently

Oracle Database can gather statistics on multiple tables or partitions concurrently.

13.3.7.1 About Concurrent Statistics Gathering

By default, each partition of a partition table is gathered sequentially.

When **concurrent statistics gathering mode** is enabled, the database can simultaneously gather optimizer statistics for multiple tables in a schema, or multiple partitions or subpartitions in a table. Concurrency can reduce the overall time required to gather statistics by enabling the database to fully use multiple processors.

Note:

Concurrent statistics gathering mode does not rely on parallel query processing, but is usable with it.

13.3.7.1.1 How DBMS_STATS Gathers Statistics Concurrently

Oracle Database employs multiple tools and technologies to create and manage multiple statistics gathering jobs concurrently.

The database uses the following:

- Oracle Scheduler
- Oracle Database Advanced Queuing (AQ)
- Oracle Database Resource Manager (the Resource Manager)

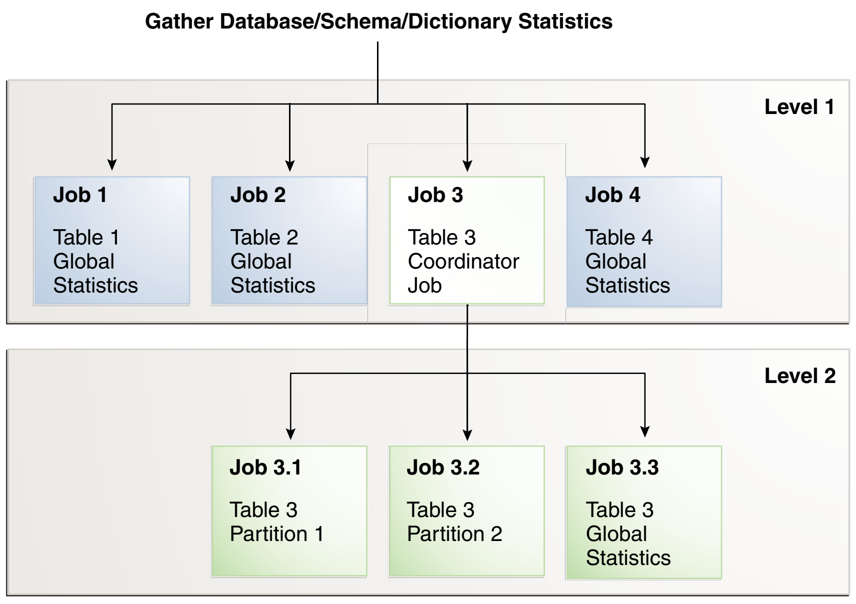
Enable concurrent statistics gathering by setting the `CONCURRENT` preference with `DBMS_STATS.SET_GLOBAL_PREF`.

The database runs as many concurrent jobs as possible. The Job Scheduler decides how many jobs to execute concurrently and how many to queue. As running jobs complete, the scheduler dequeues and runs more jobs until the database has gathered statistics on all tables, partitions, and subpartitions. The maximum number of jobs is bounded by the `JOB_QUEUE_PROCESSES` initialization parameter and available system resources.

In most cases, the `DBMS_STATS` procedures create a separate job for each table partition or subpartition. However, if the partition or subpartition is empty or very small, then the database may automatically batch the object with other small objects into a single job to reduce the overhead of job maintenance.

The following figure illustrates the creation of jobs at different levels, where Table 3 is a partitioned table, and the other tables are nonpartitioned. Job 3 acts as a coordinator job for Table 3, and creates a job for each partition in that table, and a separate job for the global statistics of Table 3. This example assumes that incremental statistics gathering is disabled; if enabled, then the database derives global statistics from partition-level statistics after jobs for partitions complete.

Figure 13-1 Concurrent Statistics Gathering Jobs


Gather Database/Schema/Dictionary Statistics
Level 1
Job 1 Table 1 Global Statistics
Job 2 Table 2 Global Statistics
Job 3 Table 3 Coordinator Job
Job 4 Table 4 Global Statistics
Level 2
Job 3.1 Table 3 Partition 1
Job 3.2 Table 3 Partition 2
Job 3.3 Table 3 Global Statistics

See Also:

- "Enabling Concurrent Statistics Gathering"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` package
- *Oracle Database Reference* to learn about the `JOB_QUEUE_PROCESSES` initialization parameter

13.3.7.1.2 Concurrent Statistics Gathering and Resource Management

The `DBMS_STATS` package does not explicitly manage resources used by concurrent statistics gathering jobs that are part of a user-initiated statistics gathering call.

Thus, the database may use system resources fully during concurrent statistics gathering. To address this situation, use the Resource Manager to cap resources consumed by concurrent statistics gathering jobs. The Resource Manager must be enabled to gather statistics concurrently.

The system-supplied consumer group `ORA$AUTOTASK` registers all statistics gathering jobs. You can create a resource plan with proper resource allocations for `ORA$AUTOTASK` to prevent concurrent statistics gathering from consuming all available resources. If you lack your own resource plan, and if choose not to create one, then consider activating the Resource Manager with the system-supplied `DEFAULT_PLAN`.

Note:

The `ORA$AUTOTASK` consumer group is shared with the maintenance tasks that automatically run during the maintenance windows. Thus, when concurrency is activated for automatic statistics gathering, the database automatically manages resources, with no extra steps required.

See Also:

Oracle Database Administrator's Guide to learn about the Resource Manager

13.3.7.2 Enabling Concurrent Statistics Gathering

To enable concurrent statistics gathering, use the `DBMS_STATS.SET_GLOBAL_PREFS` procedure to set the `CONCURRENT` preference.

Possible values are as follows:

- `MANUAL`

  Concurrency is enabled only for manual statistics gathering.
- `AUTOMATIC`

  Concurrency is enabled only for automatic statistics gathering.
- `ALL`

  Concurrency is enabled for both manual and automatic statistics gathering.
- `OFF`

  Concurrency is disabled for both manual and automatic statistics gathering. This is the default value.

This tutorial in this section explains how to enable concurrent statistics gathering.

Prerequisites

This tutorial has the following prerequisites:

- In addition to the standard privileges for gathering statistics, you must have the following privileges:
  - `CREATE JOB`
  - `MANAGE SCHEDULER`
  - `MANAGE ANY QUEUE`
- The `SYSAUX` tablespace must be online because the scheduler stores its internal tables and views in this tablespace.
- The `JOB_QUEUE_PROCESSES` initialization parameter must be set to at least `4`.

- The Resource Manager must be enabled.

  By default, the Resource Manager is disabled. If you do not have a resource plan, then consider enabling the Resource Manager with the system-supplied `DEFAULT_PLAN`.

Assumptions

This tutorial assumes that you want to do the following:

- Enable concurrent statistics gathering
- Gather statistics for the `sh` schema
- Monitor the gathering of the `sh` statistics

To enable concurrent statistics gathering:

1. Connect SQL*Plus to the database with the appropriate privileges, and then enable the Resource Manager.

   The following example uses the default plan for the Resource Manager:

   ```
   ALTER SYSTEM SET RESOURCE_MANAGER_PLAN = 'DEFAULT_PLAN';
   ```

2. Set the `JOB_QUEUE_PROCESSES` initialization parameter to at least twice the number of CPU cores.

   In Oracle Real Application Clusters, the `JOB_QUEUE_PROCESSES` setting applies to each node.

   Assume that the system has 4 CPU cores. The following example sets the parameter to `8` (twice the number of cores):

   ```
   ALTER SYSTEM SET JOB_QUEUE_PROCESSES=8;
   ```

3. Confirm that the parameter change took effect.

   For example, enter the following command in SQL*Plus (sample output included):

   ```
   SHOW PARAMETER PROCESSES;

   NAME                              TYPE        VALUE
   --------------------------------- ----------- -----
   _high_priority_processes          string      VKTM
   aq_tm_processes                   integer     1
   db_writer_processes               integer     1
   gcs_server_processes              integer     0
   global_txn_processes              integer     1
   job_queue_processes               integer     8
   log_archive_max_processes         integer     4
   processes                         integer     100
   ```

4. Enable concurrent statistics.

   For example, execute the following PL/SQL anonymous block:

   ```
   BEGIN
     DBMS_STATS.SET_GLOBAL_PREFS('CONCURRENT','ALL');
   ```

```
END;
/
```

5. Confirm that the statistics were enabled.

   For example, execute the following query (sample output included):

```
SELECT DBMS_STATS.GET_PREFS('CONCURRENT') FROM DUAL;

DBMS_STATS.GET_PREFS('CONCURRENT')
----------------------------------
ALL
```

6. Gather the statistics for the `SH` schema.

   For example, execute the following procedure:

```
EXEC DBMS_STATS.GATHER_SCHEMA_STATS('SH');
```

7. In a separate session, monitor the job progress by querying `DBA_OPTSTAT_OPERATION_TASKS`.

   For example, execute the following query (sample output included):

```
SET LINESIZE 1000

COLUMN TARGET FORMAT a8
COLUMN TARGET_TYPE FORMAT a25
COLUMN JOB_NAME FORMAT a14
COLUMN START_TIME FORMAT a40

SELECT TARGET, TARGET_TYPE, JOB_NAME,
       TO_CHAR(START_TIME, 'dd-mon-yyyy hh24:mi:ss')
FROM   DBA_OPTSTAT_OPERATION_TASKS
WHERE  STATUS = 'IN PROGRESS'
AND    OPID = (SELECT MAX(ID)
               FROM   DBA_OPTSTAT_OPERATIONS
               WHERE  OPERATION = 'gather_schema_stats');

TARGET    TARGET_TYPE               JOB_NAME       TO_CHAR(START_TIME,'
--------- ------------------------- -------------- --------------------
SH.SALES  TABLE (GLOBAL STATS ONLY) ST$T292_1_B29  30-nov-2012 14:22:47
SH.SALES  TABLE (COORDINATOR JOB)   ST$SD290_1_B10 30-nov-2012 14:22:08
```

8. In the original session, disable concurrent statistics gathering.

   For example, execute the following query:

```
EXEC DBMS_STATS.SET_GLOBAL_PREFS('CONCURRENT','OFF');
```

See Also:

- "Monitoring Statistics Gathering Operations"
- *Oracle Database Administrator's Guide*
- *Oracle Database PL/SQL Packages and Types Reference* to learn how to use the `DBMS_STATS.SET_GLOBAL_PREFS` procedure

13.3.7.3 Monitoring Statistics Gathering Operations

You can monitor statistics gathering jobs using data dictionary views.

The following views are relevant:

- `DBA_OPTSTAT_OPERATION_TASKS`

  This view contains the history of tasks that are performed or currently in progress as part of statistics gathering operations (recorded in `DBA_OPTSTAT_OPERATIONS`). Each task represents a target object to be processed in the corresponding parent operation.
- `DBA_OPTSTAT_OPERATIONS`

  This view contains a history of statistics operations performed or currently in progress at the table, schema, and database level using the `DBMS_STATS` package.

The `TARGET` column in the preceding views shows the target object for that statistics gathering job in the following form:

```
OWNER.TABLE_NAME.PARTITION_OR_SUBPARTITION_NAME
```

All statistics gathering job names start with the string `ST$`.

To display currently running statistics tasks and jobs:

- To list statistics gathering currently running tasks from all user sessions, use the following SQL statement (sample output included):

  ```
  SELECT OPID, TARGET, JOB_NAME,
         (SYSTIMESTAMP - START_TIME) AS elapsed_time
  FROM   DBA_OPTSTAT_OPERATION_TASKS
  WHERE  STATUS = 'IN PROGRESS';

  OPID TARGET                       JOB_NAME      ELAPSED_TIME
  ---- ------------------------- -------------
  --------------------------
   981 SH.SALES.SALES_Q4_1998     ST$T82_1_B29  +000000000
  00:00:00.596321
   981 SH.SALES                   ST$SD80_1_B10 +000000000
  00:00:27.972033
  ```

To display completed statistics tasks and jobs:

- To list only completed tasks and jobs from a particular operation, first identify the operation ID from the `DBA_OPTSTAT_OPERATIONS` view based on the statistics gathering operation name, target, and start time. After you identify the operation ID, you can query the `DBA_OPTSTAT_OPERATION_TASKS` view to find the corresponding tasks in that operation

  For example, to list operations with the ID 981, use the following commands in SQL*Plus (sample output included):

  ```
  VARIABLE id NUMBER
  EXEC :id := 981

  SELECT TARGET, JOB_NAME, (END_TIME - START_TIME) AS ELAPSED_TIME
  FROM   DBA_OPTSTAT_OPERATION_TASKS
  WHERE  STATUS <> 'IN PROGRESS'
  AND    OPID = :id;

  TARGET                    JOB_NAME      ELAPSED_TIME
  ------------------------- ------------- --------------------------
  SH.SALES_TRANSACTIONS_EXT               +000000000 00:00:45.479233
  SH.CAL_MONTH_SALES_MV     ST$SD88_1_B10 +000000000 00:00:45.382764
  SH.CHANNELS               ST$SD88_1_B10 +000000000 00:00:45.307397
  ```

To display statistics gathering tasks and jobs that have failed:

- Use the following SQL statement (partial sample output included):

  ```
  SET LONG 10000

  SELECT TARGET, JOB_NAME AS NM,
         (END_TIME - START_TIME) AS ELAPSED_TIME, NOTES
  FROM   DBA_OPTSTAT_OPERATION_TASKS
  WHERE  STATUS = 'FAILED';

  TARGET             NM ELAPSED_TIME               NOTES
  ------------------ -- -------------------------- -----------------
  SYS.OPATCH_XML_INV    +000000007 02:36:31.130314 <error>ORA-20011:
                                                   Approximate NDV
                                                   failed: ORA-29913:
                                                   error in
  ```

See Also:

Oracle Database Reference to learn about the `DBA_SCHEDULER_JOBS` view

13.3.8 Gathering Incremental Statistics on Partitioned Objects

Incremental statistics scan only changed partitions. When gathering statistics on large partitioned table by deriving global statistics from partition-level statistics, **incremental statistics maintenance** improves performance.

13.3.8.1 Purpose of Incremental Statistics

In a typical case, an application loads data into a new partition of a range-partitioned table. As applications add new partitions and load data, the database must gather statistics on the new partition and keep global statistics up to date.

Typically, data warehouse applications access large partitioned tables. Often these tables are partitioned on date columns, with only the recent partitions subject to frequent DML changes. Without incremental statistics, statistics collection typically uses a two-pass approach:

1. The database scans the table to gather the global statistics.

   The full scan of the table for global statistics collection can be very expensive, depending on the size of the table. As the table adds partitions, the longer the execution time for `GATHER_TABLE_STATS` because of the full table scan required for the global statistics. The database must perform the scan of the entire table even if only a small subset of partitions change.

2. The database scans the changed partitions to gather their partition-level statistics.

Incremental maintenance provides a huge performance benefit for data warehouse applications because of the following:

- The database must scan the table only once to gather partition statistics and to derive the global statistics by aggregating partition-level statistics. Thus, the database avoids the *two* full scans that are required when not using incremental statistics: one scan for the partition-level statistics, and one scan for the global-level statistics.
- In subsequent statistics gathering, the database only needs to scan the stale partitions and update their statistics (including synopses). The database can derive global statistics from the fresh partition statistics, which saves a full table scan.

When using incremental statistics, the database must still gather statistics on any partition that will change the global or table-level statistics. Incremental statistics maintenance yields the same statistics as gathering table statistics from scratch, but performs better.

13.3.8.2 How DBMS_STATS Derives Global Statistics for Partitioned tables

When incremental statistics maintenance is enabled, `DBMS_STATS` gathers statistics and creates synopses for changed partitions only. The database also automatically merges partition-level synopses into a global synopsis, and derives global statistics from the partition-level statistics and global synopses.

The database avoids a full table scan when computing global statistics by deriving some global statistics from the partition-level statistics. For example, the number of rows at the global level is the sum of number of rows of partitions. Even global histograms can be derived from partition histograms.

However, the database cannot derive *all* statistics from partition-level statistics, including the NDV of a column. The following example shows the NDV for two partitions in a table:

Table 13-3 NDV for Two Partitions

Object	Column Values	NDV
Partition 1	1,3,3,4,5	4
Partition 2	2,3,4,5,6	5

Calculating the NDV in the table by adding the NDV of the individual partitions produces an NDV of 9, which is incorrect. Thus, a more accurate technique is required: synopses.

13.3.8.2.1 Partition-Level Synopses

A **synopsis** is special type of statistic that tracks the number of distinct values (NDV) for each column in a partition. You can consider a synopsis as an internal management structure that samples distinct values.

The database can accurately derive the global-level NDV for each column by merging partition-level synopses. In the example shown in Table 13-3, the database can use synopses to calculate the NDV for the column as 6.

Each partition maintains a synopsis in incremental mode. When a new partition is added to the table you only need to gather statistics for the new partition. The database automatically updates the global statistics by aggregating the new partition synopsis with the synopses for existing partitions. Subsequent statistics gathering operations are faster than when synopses are not used.

The database stores synopses in data dictionary tables `WRI$_OPTSTAT_SYNOPSIS_HEAD$` and `WRI$_OPTSTAT_SYNOPSIS$` in the `SYSAUX` tablespace. The `DBA_PART_COL_STATISTICS` dictionary view contains information of the column statistics in partitions. If the `NOTES` column contains the keyword `INCREMENTAL`, then this column has synopses.

See Also:

Oracle Database Reference to learn more about `DBA_PART_COL_STATISTICS`

13.3.8.2.2 NDV Algorithms: Adaptive Sampling and HyperLogLog

Starting in Oracle Database 12c Release 2 (12.2), the HyperLogLog algorithm can improve NDV (number of distinct values) calculation performance, and also reduce the storage space required for synopses.

The legacy algorithm for calculating NDV uses **adaptive sampling**. A synopsis is a sample of the distinct values. When calculating the NDV, the database initially stores every distinct value in a hash table. Each distinct value occupies a distinct hash bucket, so a column with 5000 distinct values has 5000 hash buckets. The database then halves the number of hash buckets, and then continues to halve the result until a small number of buckets remain. The algorithm is “adaptive” because the sampling rate changes based on the number of hash table splits.

To calculate the NDV for the column, the database uses the following formula, where *B* is the number of hash buckets remaining after all the splits have been performed, and *S* is the number of splits:

```
NDV = B * 2^S
```

Adaptive sampling produces accurate NDV statistics, but has the following consequences:

- Synopses occupy significant disk space, especially when tables have many columns and partitions, and the NDV in each column is high.

  For example, a 60-column table might have 300,000 partitions, with an average per-column NDV of 5,000. In this example, each partition has 300,000 entries (60 x 5000). In total, the synopses tables have 90 billion entries (300,000 squared), which occupies at least 720 GB of storage space.
- Bulk processing of synopses can negatively affect performance.

  Before the database regathers statistics on the stale partitions, it must delete the associated synopses. Bulk deletion can be slow because it generates significant amounts of undo and redo data.

In contrast to dynamic sampling, the HyperLogLog algorithm uses a randomization technique. Although the algorithm is complex, the foundational insight is that in a stream of random values, *n* distinct values will be spaced on average 1/*n* apart. Therefore, if you know the smallest value in the stream, then you can roughly estimate the number of distinct values. For example, if the values range from 0 to 1, and if the smallest value observed is .2, then the numbers will on average be evenly spaced .2 apart, so the NDV estimate is 5.

The HyperLogLog algorithm expands on and corrects the original estimate. The database applies a hash function to every column value, resulting in a set of hash values with the same cardinality as the column. For the base estimate, the NDV equals 2^n, where *n* is the maximum number of trailing zeroes observed in the binary representation of the hash values. The database refines its NDV estimate by using part of the output to split values into different hash buckets.

The advantages of the HyperLogLog algorithm over adaptive sampling are:

- The accuracy of the new algorithm is similar to the original algorithm.
- The memory required is *significantly* lower, which typically leads to huge reductions in synopsis size.

  Synopses can become large when many partitions exist, and they have many columns with high NDV. Synopses that use the HyperLogLog algorithm are more compact. Creating and deleting synopses affects batch run times. Any operational procedures that manage partitions reduce run time.

The `DBMS_STATS` preference `APPROXIMATE_NDV_ALGORITHM` determines which algorithm the database uses for NDV calculation.

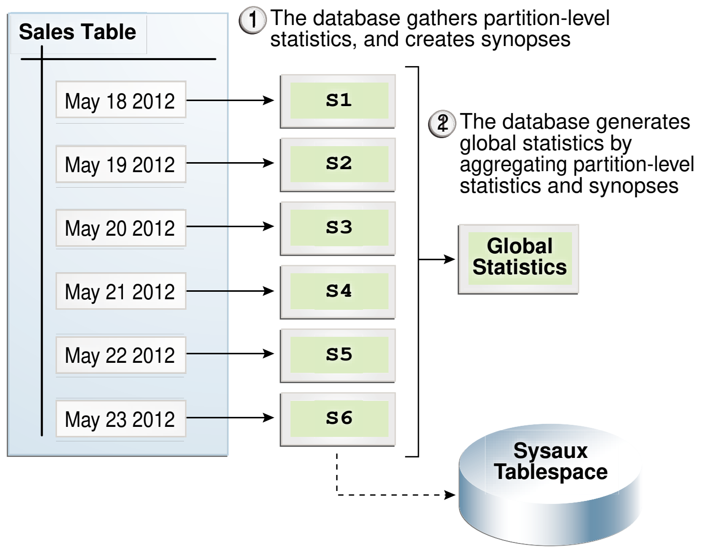
See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `APPROXIMATE_NDV_ALGORITHM` preference

13.3.8.2.3 Aggregation of Global Statistics Using Synopses: Example

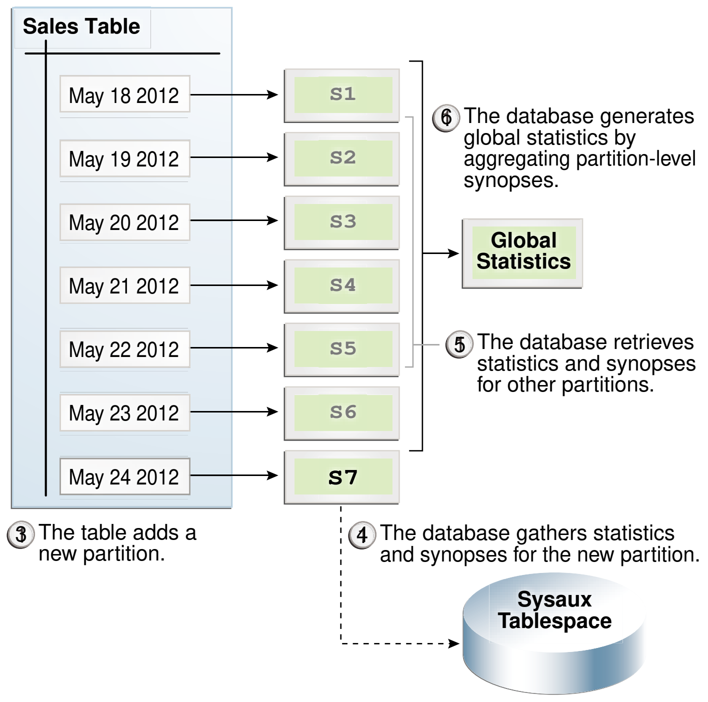
In this example, the database gathers statistics for the initial six partitions of the `sales` table, and then creates synopses for each partition (`S1`, `S2`, and so on). The database creates global statistics by aggregating the partition-level statistics and synopses.

Figure 13-2 Aggregating Statistics


Sales Table
1 The database gathers partition-level statistics, and creates synopses
May 18 2012
May 19 2012
May 20 2012
May 21 2012
May 22 2012
May 23 2012
S1
S2
S3
S4
S5
S6
2 The database generates global statistics by aggregating partition-level statistics and synopses
Global Statistics
Sysaux Tablespace

The following graphic shows a new partition, containing data for May 24, being added to the `sales` table. The database gathers statistics for the newly added partition, retrieves synopses for the other partitions, and then aggregates the synopses to create global statistics.

Figure 13-3 Aggregating Statistics after Adding a Partition


Sales Table
May 18 2012
May 19 2012
May 20 2012
May 21 2012
May 22 2012
May 23 2012
May 24 2012
S1
S2
S3
S4
S5
S6
S7
6 The database generates global statistics by aggregating partition-level synopses.
Global Statistics
5 The database retrieves statistics and synopses for other partitions.
3 The table adds a new partition.
4 The database gathers statistics and synopses for the new partition.
Sysaux Tablespace

13.3.8.3 Gathering Statistics for a Partitioned Table: Basic Steps

This section explains how to gather optimizer statistics for a partitioned table.

13.3.8.3.1 Considerations for Incremental Statistics Maintenance

Enabling incremental statistics maintenance has several consequences.

Specifically, note the following:

- If a table uses composite partitioning, then the database only gathers statistics for modified subpartitions. The database does not gather statistics at the subpartition level for unmodified subpartitions. In this way, the database reduces work by skipping unmodified partitions.
- If a table uses incremental statistics, and if this table has a locally partitioned index, then the database gathers index statistics at the global level and for modified (not unmodified) index partitions. The database does not generate global index statistics from the partition-level index statistics. Rather, the database gathers global index statistics by performing a full index scan.
- A hybrid partitioned table contains both internal and external partitions. For internal partitions only, DDL changes invoke incremental statistic maintenance on individual partitions and on the table itself. For example, if `june18` is an internal partition, then `ALTER TABLE ... MODIFY PARTITION jun18 ...` triggers incremental statistics maintenance during statistics collection; if `june18` is an external partition, however, then incremental maintenance does not occur.
- The `SYSAUX` tablespace consumes additional space to maintain global statistics for partitioned tables.

See Also:

- *Oracle Database VLDB and Partitioning Guide* to learn how to create hybrid partitioned tables
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS`

13.3.8.3.2 Enabling Incremental Statistics Using SET_TABLE_PREFS

To enable incremental statistics maintenance for a partitioned table, use `DBMS_STATS.SET_TABLE_PREFS` to set the `INCREMENTAL` value to `true`. When `INCREMENTAL` is set to `false`, which is the default, the database uses a full table scan to maintain global statistics.

For the database to update global statistics incrementally by scanning *only* the partitions that have changed, the following conditions must be met:

- The `PUBLISH` value for the partitioned table is `true`.
- The `INCREMENTAL` value for the partitioned table is `true`.
- The statistics gathering procedure must specify `AUTO_SAMPLE_SIZE` for `ESTIMATE_PERCENT` and `AUTO` for `GRANULARITY`.

Example 13-3 Enabling Incremental Statistics

Assume that the `PUBLISH` value for the partitioned table `sh.sales` is `true`. The following program enables incremental statistics for this table:

```
EXEC DBMS_STATS.SET_TABLE_PREFS('sh', 'sales', 'INCREMENTAL', 'TRUE');
```

13.3.8.3.3 About the APPROXIMATE_NDV_ALGORITHM Settings

The `DBMS_STATS.APPROXIMATE_NDV_ALGORITHM` preference specifies the synopsis generation algorithm, either HyperLogLog or adaptive sampling. The `INCREMENTAL_STALENESS` preference controls when the database reformats synopses that use the adaptive sampling format.

The `APPROXIMATE_NDV_ALGORITHM` preference has the following possible values:

- `REPEAT OR HYPERLOGLOG`

  This is the default. If `INCREMENTAL` is enabled on the table, then the database preserves the format of any existing synopses that use the adaptive sampling algorithm. However, the database creates any new synopses in HyperLogLog format. This approach is attractive when existing performance is acceptable, and you do not want to incur the performance cost of reformatting legacy content.

- `ADAPTIVE SAMPLING`

  The database uses the adaptive sampling algorithm for all synopses. This is the most conservative option.

- `HYPERLOGLOG`

  The database uses the HyperLogLog algorithm for all new and stale synopses.

The `INCREMENTAL_STALENESS` preference controls when a synopsis is considered stale. When the `APPROXIMATE_NDV_ALGORITHM` preference is set to `HYPERLOGLOG`, then the following `INCREMENTAL_STALENESS` settings apply:

- `ALLOW_MIXED_FORMAT`

  This is the default. If this value is specified, and if the following conditions are met, then the database does *not* consider existing adaptive sampling synopses as stale:

  – The synopses are fresh.

  – You gather statistics manually.

  Thus, synopses in both the legacy and HyperLogLog formats can co-exist. However, over time the automatic statistics gathering job regathers statistics on synopses that use the old format, and replaces them with synopses in HyperLogLog format. In this way, the automatic statistics gather job gradually phases out the old format. Manual statistics gathering jobs do not reformat synopses that use the adaptive sampling format.

- Null

  Any partitions with the synopses in the legacy format are considered stale, which *immediately* triggers the database to regather statistics for stale synopses. The advantage is that the performance cost occurs only once. The disadvantage is that regathering all statistics on large tables can be resource-intensive.

13.3.8.3.4 Configuring Synopsis Generation: Examples

These examples show different approaches, both conservative and aggressive, to switching synopses to the new HyperLogLog format.

Example 13-4 Taking a Conservative Approach to Reformatting Synopses

In this example, you allow synopses in mixed formats to coexist for the `sh.sales` table. Mixed formats yield less accurate statistics. However, you do *not* need to regather statistics for all partitions of the table.

To ensure that all new and stale synopses use the HyperLogLog algorithm, set the `APPROXIMATE_NDV_ALGORITHM` preference to `HYPERLOGLOG`. To ensure that the automatic statistics gathering job reformats stale synopses gradually over time, set the `INCREMENTAL_STALENESS` preference to `ALLOW_MIXED_FORMAT`.

```
BEGIN
  DBMS_STATS.SET_TABLE_PREFS
    (   ownname => 'sh'
    ,   tabname => 'sales'
    ,   pname   => 'approximate_ndv_algorithm'
    ,   pvalue  => 'hyperloglog' );

  DBMS_STATS.SET_TABLE_PREFS
    (  ownname  => 'sh'
    ,  tabname  => 'sales'
    ,  pname    => 'incremental_staleness'
    ,  pvalue   => 'allow_mixed_format' );
END;
```

Example 13-5 Taking an Aggressive Approach to Reformatting Synopses

In this example, you force all synopses to use the HyperLogLog algorithm for the `sh.sales` table. In this case, the database must regather statistics for all partitions of the table.

To ensure that all new and stale synopses use the HyperLogLog algorithm, set the `APPROXIMATE_NDV_ALGORITHM` preference to `HYPERLOGLOG`. To force the database to immediately regather statistics for all partitions in the table and store them in the new format, set the `INCREMENTAL_STALENESS` preference to null.

```
BEGIN
  DBMS_STATS.SET_TABLE_PREFS
    (   ownname => 'sh'
    ,   tabname => 'sales'
    ,   pname   => 'approximate_ndv_algorithm'
    ,   pvalue  => 'hyperloglog' );

  DBMS_STATS.SET_TABLE_PREFS
    (  ownname   => 'sh'
    ,  tabname   => 'sales'
    ,  pname     => 'incremental_staleness'
    ,  pvalue    => 'null' );
END;
```

13.3.8.4 Maintaining Incremental Statistics for Partition Maintenance Operations

A **partition maintenance operation** is a partition-related operation such as adding, exchanging, merging, or splitting table partitions.

Oracle Database provides the following support for incremental statistics maintenance:

- If a partition maintenance operation triggers statistics gathering, then the database can reuse synopses that would previously have been dropped with the old segments.
- `DBMS_STATS` can create a synopsis on a nonpartitioned table. The synopsis enables the database to maintain incremental statistics as part of a partition exchange operation without having to explicitly gather statistics on the partition after the exchange.

When the `DBMS_STATS` preference `INCREMENTAL` is set to `true` on a table, the `INCREMENTAL_LEVEL` preference controls which synopses are collected and when. This preference takes the following values:

- `TABLE`

  `DBMS_STATS` gathers table-level synopses on this table. You can only set `INCREMENTAL_LEVEL` to `TABLE` at the table level, not at the schema, database, or global level.
- `PARTITION` (default)

  `DBMS_STATS` only gathers synopsis at the partition level of partitioned tables.

When performing a partition exchange, to have synopses after the exchange for the partition being exchanged, set `INCREMENTAL` to `true` and `INCREMENTAL_LEVEL` to `TABLE` on the table to be exchanged with the partition.

Assumptions

This tutorial assumes the following:

- You want to load empty partition `p_sales_01_2010` in a `sales` table.
- You create a staging table `t_sales_01_2010`, and then populate the table.
- You want the database to maintain incremental statistics as part of the partition exchange operation without having to explicitly gather statistics on the partition after the exchange.

To maintain incremental statistics as part of a partition exchange operation:

1. Set incremental statistics preferences for staging table `t_sales_01_2010`.

   For example, run the following statement:

   ```
   BEGIN
     DBMS_STATS.SET_TABLE_PREFS (
         ownname  =>  'sh'
   ,     tabname  =>  't_sales_01_2010'
   ,     pname    =>  'INCREMENTAL'
   ,     pvalue   =>  'true'
   );
     DBMS_STATS.SET_TABLE_PREFS (
         ownname  =>  'sh'
   ,     tabname  =>  't_sales_01_2010'
   ,     pname    =>  'INCREMENTAL_LEVEL'
   ,     pvalue   =>  'table'
   );
   END;
   ```

2. Gather statistics on staging table `t_sales_01_2010`.

   For example, run the following PL/SQL code:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS (
         ownname  => 'SH'
   ,     tabname  => 'T_SALES_01_2010'
   );
   END;
   /
   ```

   `DBMS_STATS` gathers table-level synopses on `t_sales_01_2010`.

3. Ensure that the `INCREMENTAL` preference is `true` on the `sh.sales` table.

   For example, run the following PL/SQL code:

   ```
   BEGIN
     DBMS_STATS.SET_TABLE_PREFS (
         ownname  =>  'sh'
   ,     tabname  =>  'sales'
   ,     pname    =>  'INCREMENTAL'
   ,     pvalue   =>  'true'
   ```

```
);
END;
/
```

4. If you have never gathered statistics on `sh.sales` before with `INCREMENTAL` set to `true`, then gather statistics on the partition to be exchanged.

   For example, run the following PL/SQL code:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS (
         ownname  =>  'sh'
   ,     tabname  =>  'sales'
   ,     pname    =>  'p_sales_01_2010'
   ,     pvalue   =>  granularity=>'partition'
   );
   END;
   /
   ```

5. Perform the partition exchange.

   For example, use the following SQL statement:

   ```
   ALTER TABLE sales EXCHANGE PARTITION p_sales_01_2010 WITH TABLE
   t_sales_01_2010;
   ```

   After the exchange, the partitioned table has both statistics and a synopsis for partition `p_sales_01_2010`.

   In releases before Oracle Database 12c, the preceding statement swapped the segment data and statistics of `p_sales_01_2010` with `t_sales_01_2010`. The database did not maintain synopses for nonpartitioned tables such as `t_sales_01_2010`. To gather global statistics on the partitioned table, you needed to rescan the `p_sales_01_2010` partition to obtain its synopses.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.SET_TABLE_PREFS`

13.3.8.5 Maintaining Incremental Statistics for Tables with Stale or Locked Partition Statistics

Starting in Oracle Database 12c, incremental statistics can automatically calculate global statistics for a partitioned table even if the partition or subpartition statistics are stale and locked.

When incremental statistics are enabled in releases before Oracle Database 12c, if any DML occurs on a partition, then the optimizer considers statistics on this partition to be stale. Thus, `DBMS_STATS` must gather the statistics again to accurately aggregate the global statistics. Furthermore, if DML occurs on a partition whose statistics are locked, then `DBMS_STATS` cannot regather the statistics on the partition, so a full table scan is the only means of gathering global statistics. Regathering statistics creates performance overhead.

In Oracle Database 12c, the statistics preference `INCREMENTAL_STALENESS` controls how the database determines whether the statistics on a partition or subpartition are stale. This preference takes the following values:

- `USE_STALE_PERCENT`

  A partition or subpartition is not considered stale if DML changes are less than the `STALE_PERCENT` preference specified for the table. The default value of `STALE_PERCENT` is `10`, which means that if DML causes more than 10% of row changes, then the table is considered stale.

- `USE_LOCKED_STATS`

  Locked partition or subpartition statistics are not considered stale, regardless of DML changes.

- `NULL` (default)

  A partition or subpartition is considered stale if it has any DML changes. This behavior is identical to the Oracle Database 11g behavior. When the default value is used, statistics gathered in incremental mode are guaranteed to be the same as statistics gathered in nonincremental mode. When a nondefault value is used, statistics gathered in incremental mode might be less accurate than those gathered in nonincremental mode.

You can specify `USE_STALE_PERCENT` and `USE_LOCKED_STATS` together. For example, you can write the following anonymous block:

```
BEGIN
  DBMS_STATS.SET_TABLE_PREFS (
    ownname     => null
,   table_name  => 't'
,   pname       => 'incremental_staleness'
,   pvalue      => 'use_stale_percent,use_locked_stats'
);
END;
```

Assumptions

This tutorial assumes the following:

- The `STALE_PERCENT` for a partitioned table is set to `10`.
- The `INCREMENTAL` value is set to `true`.
- The table has had statistics gathered in `INCREMENTAL` mode before.
- You want to discover how statistics gathering changes depending on the setting for `INCREMENTAL_STALENESS`, whether the statistics are locked, and the percentage of DML changes.

To test for tables with stale or locked partition statistics:

1. Set `INCREMENTAL_STALENESS` to `NULL`.

   Afterward, 5% of the rows in one partition change because of DML activity.

2. Use `DBMS_STATS` to gather statistics on the table.

`DBMS_STATS` regathers statistics for the partition that had the 5% DML activity, and incrementally maintains the global statistics.

3. Set `INCREMENTAL_STALENESS` to `USE_STALE_PERCENT`.

   Afterward, 5% of the rows in one partition change because of DML activity.

4. Use `DBMS_STATS` to gather statistics on the table.

   `DBMS_STATS` does *not* regather statistics for the partition that had DML activity (because the changes are under the staleness threshold of 10%), and incrementally maintains the global statistics.

5. Lock the partition statistics.

   Afterward, 20% of the rows in one partition change because of DML activity.

6. Use `DBMS_STATS` to gather statistics on the table.

   `DBMS_STATS` does *not* regather statistics for the partition because the statistics are locked. The database gathers the global statistics with a full table scan.

   Afterward, 5% of the rows in one partition change because of DML activity.

7. Use `DBMS_STATS` to gather statistics on the table.

   When you gather statistics on this table, `DBMS_STATS` does not regather statistics for the partition because they are not considered stale. The database maintains global statistics incrementally using the existing statistics for this partition.

8. Set `INCREMENTAL_STALENESS` to `USE_LOCKED_STATS` and `USE_STALE_PERCENT`.

   Afterward, 20% of the rows in one partition change because of DML activity.

9. Use `DBMS_STATS` to gather statistics on the table.

   Because `USE_LOCKED_STATS` is set, `DBMS_STATS` ignores the fact that the statistics are stale and uses the locked statistics. The database maintains global statistics incrementally using the existing statistics for this partition.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.SET_TABLE_PREFS`

13.4 Gathering System Statistics Manually

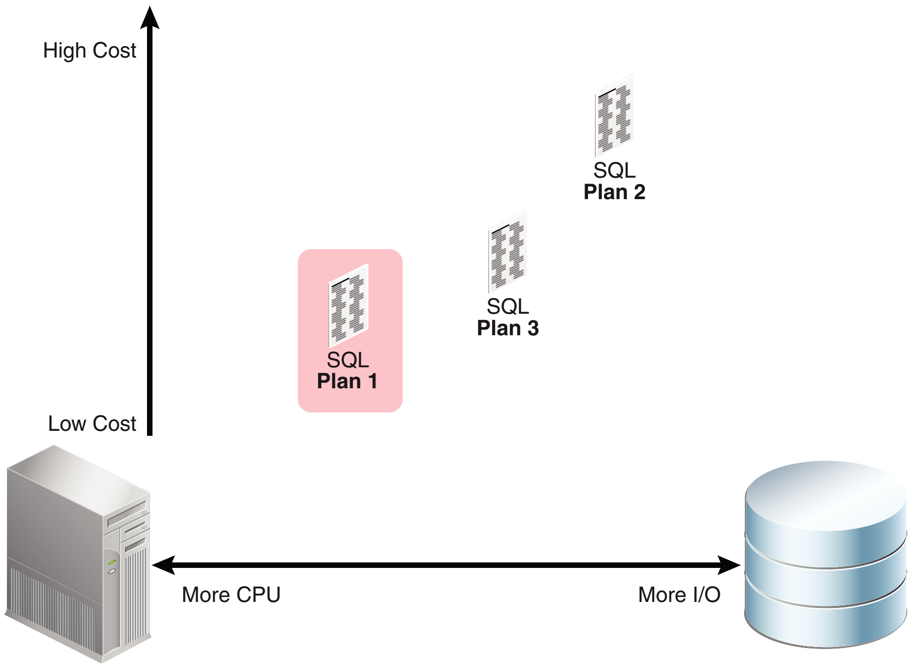
System statistics describe hardware characteristics, such as I/O and CPU performance and utilization, to the optimizer.

13.4.1 About System Statistics

System statistics measure the performance of CPU and storage so that the optimizer can use these inputs when evaluating plans.

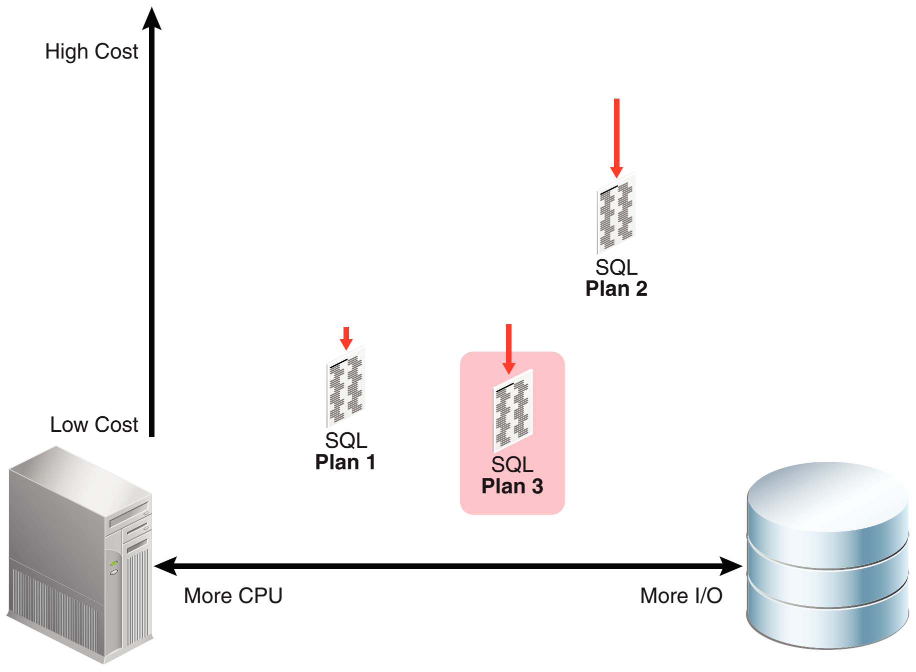
When a query executes, it consumes CPU. In many cases, a query also consumes storage subsystem resources. Each plan in a typical query may consume a different proportion of CPU and I/O. Using the cost metric, the optimizer chooses the plan that it estimates will execute most quickly. If the optimizer knows the speed of CPU and storage, then it can make finer judgments about the cost of each alternative plan.

The following figure shows a query that has three possible plans. Each plan uses different amounts of CPU and I/O. For the sake of this example, the optimizer has assigned Plan 1 the lowest cost.



The database automatically gathers essential system statistics, called noworkload statistics, at the first instance startup. Typically, these characteristics only change when some aspect of the hardware configuration is upgraded.

The following figure shows the same database after adding high-performance storage. Gathering system statistics enables the optimizer to take the storage performance into account. In this example, the high-performance storage lowers the relative cost of Plan 2 and Plan 3 significantly. Plan 1 shows only marginal improvement because it uses less I/O. Plan 3 has now been assigned the lowest cost.



On systems with fast I/O infrastructure, system statistics increase the probability that queries choose table scans over index access methods.

13.4.2 Guidelines for Gathering System Statistics

Unless there is a good reason to gather manually, Oracle recommends using the defaults for system statistics.

System statistics are important for performance because they affect *every* SQL statement executed in the database. Changing system statistics may change SQL execution plans, perhaps in unexpected or unwanted ways. For this reason, Oracle recommends considering the options carefully before changing system statistics.

When to Consider Gathering System Statistics Manually

If you are using Oracle Exadata, and if the database is running a pure data warehouse load, then gathering system statistics using the `EXADATA` option can help performance in some cases because table scans are more strongly favored. However, even on Exadata, the defaults are best for most workloads.

If you are not using Oracle Exadata, and if you choose to gather system statistics manually, then Oracle recommends the following:

- Gather system statistics when a physical change occurs in your environment, for example, the server gets faster CPUs, more memory, or different disk storage. Oracle recommends that you gather noworkload statistics after you create new tablespaces on storage that is not used by any other tablespace.
- Capture statistics when the system has the most common workload. Gathering workload statistics does not generate additional overhead.

When to Consider Using Default Statistics

Oracle recommends using the defaults for system statistics in most cases. To reset system statistics to their default values, execute `DBMS_STATS.DELETE_SYSTEM_STATS`, and then shut down and reopen the database. To ensure that appropriate defaults are used, this step is also recommended on a newly created database.

13.4.3 Gathering System Statistics with DBMS_STATS

To gather system statistics manually, use the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure.

13.4.3.1 About the GATHER_SYSTEM_STATS Procedure

The `DBMS_STATS.GATHER_SYSTEM_STATS` procedure analyzes activity in a specified time period (**workload statistics**) or simulates a workload (**noworkload statistics**).

The input arguments to `DBMS_STATS.GATHER_SYSTEM_STATS` are:

- `NOWORKLOAD`

  The optimizer gathers statistics based on system characteristics only, without regard to the workload.
- `INTERVAL`

  After the specified number of minutes has passed, the optimizer updates system statistics either in the data dictionary, or in an alternative table (specified by `stattab`). Statistics are based on system activity during the specified interval.
- `START` and `STOP`

  `START` initiates gathering statistics. `STOP` calculates statistics for the elapsed period (since `START`) and refreshes the data dictionary or an alternative table (specified by `stattab`). The optimizer ignores `INTERVAL`.
- `EXADATA`

  The system statistics consider the unique capabilities provided by using Exadata, such as large I/O size and high I/O throughput. The optimizer sets the multiblock read count and I/O throughput statistics along with CPU speed.

The following table lists the optimizer system statistics gathered by `DBMS_STATS` and the options for gathering or manually setting specific system statistics.

Table 13-4 Optimizer System Statistics in the DBMS_STATS Package

Parameter Name	Description	Initialization	Options for Gathering or Setting Statistics	Unit
`cpuspeedNW`	Represents noworkload CPU speed. CPU speed is the average number of CPU cycles in each second.	At system startup	Set `gathering_mode` = `NOWORKLOAD` or set statistics manually.	millions/s

Table 13-4 (Cont.) Optimizer System Statistics in the DBMS_STATS Package

Parameter Name	Description	Initialization	Options for Gathering or Setting Statistics	Unit
`ioseektim`	Represents the time it takes to position the disk head to read data. I/O seek time equals seek time + latency time + operating system overhead time.	At system startup 10 (default)	Set `gathering_mode` = `NOWORKLOAD` or set statistics manually.	ms
`iotfrspeed`	Represents the rate at which an Oracle database can read data in the single read request.	At system startup 4096 (default)	Set `gathering_mode` = `NOWORKLOAD` or set statistics manually.	bytes/ms
`cpuspeed`	Represents workload CPU speed. CPU speed is the average number of CPU cycles in each second.	None	Set `gathering_mode` = `NOWORKLOAD`, `INTERVAL`, or `START|STOP`, or set statistics manually.	millions/s
`maxthr`	Maximum I/O throughput is the maximum throughput that the I/O subsystem can deliver.	None	Set `gathering_mode` = `NOWORKLOAD`, `INTERVAL`, or `START|STOP`, or set statistics manually.	bytes/s
`slavethr`	Slave I/O throughput is the average parallel execution server I/O throughput.	None	Set `gathering_mode` = `INTERVAL` or `START|STOP`, or set statistics manually.	bytes/s
`sreadtim`	Single-block read time is the average time to read a single block randomly.	None	Set `gathering_mode` = `INTERVAL` or `START|STOP`, or set statistics manually.	ms
`mreadtim`	Multiblock read is the average time to read a multiblock sequentially.	None	Set `gathering_mode` = `INTERVAL` or `START|STOP`, or set statistics manually.	ms
`mbrc`	Multiblock count is the average multiblock read count sequentially.	None	Set `gathering_mode` = `INTERVAL` or `START|STOP`, or set statistics manually.	blocks

See Also:

Oracle Database PL/SQL Packages and Types Reference for detailed information on the procedures in the `DBMS_STATS` package for gathering and deleting system statistics

13.4.3.2 Gathering Workload Statistics

Oracle recommends that you use `DBMS_STATS.GATHER_SYSTEM_STATS` to capture statistics when the database has the most typical workload.

For example, database applications can process OLTP transactions during the day and generate OLAP reports at night.

13.4.3.2.1 About Workload Statistics

Workload statistics analyze activity in a specified time period.

Workload statistics include the following statistics listed in Table 13-4:

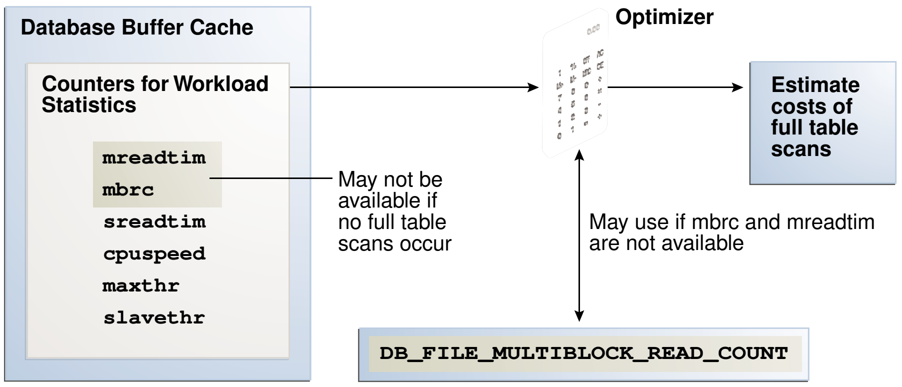
- Single block (`sreadtim`) and multiblock (`mreadtim`) read times
- Multiblock count (`mbrc`)
- CPU speed (`cpuspeed`)
- Maximum system throughput (`maxthr`)
- Average parallel execution throughput (`slavethr`)

The database computes `sreadtim`, `mreadtim`, and `mbrc` by comparing the number of physical sequential and random reads between two points in time from the beginning to the end of a workload. The database implements these values through counters that change when the buffer cache completes synchronous read requests.

Because the counters are in the buffer cache, they include not only I/O delays, but also waits related to latch contention and task switching. Thus, workload statistics depend on system activity during the workload window. If system is I/O bound (both latch contention and I/O throughput), then the statistics promote a less I/O-intensive plan after the database uses the statistics.

As shown in Figure 13-4, if you gather workload statistics, then the optimizer uses the `mbrc` value gathered for workload statistics to estimate the cost of a full table scan.

Figure 13-4   Workload Statistics Counters


Database Buffer Cache
Counters for Workload Statistics
mreadtim
mbrc
sreadtim
cpuspeed
maxthr
slavethr
May not be available if no full table scans occur
Optimizer
Estimate costs of full table scans
May use if mbrc and mreadtim are not available
DB_FILE_MULTIBLOCK_READ_COUNT

When gathering workload statistics, the database may not gather the `mbrc` and `mreadtim` values if no table scans occur during serial workloads, as is typical of OLTP systems. However, full table scans occur frequently on DSS systems. These scans may run parallel and bypass the buffer cache. In such cases, the database still gathers the `sreadtim` because index lookups use the buffer cache.

If the database cannot gather or validate gathered `mbrc` or `mreadtim` values, but has gathered `sreadtim` and `cpuspeed`, then the database uses only `sreadtim` and `cpuspeed` for costing. In this case, the optimizer uses the value of the initialization parameter `DB_FILE_MULTIBLOCK_READ_COUNT` to cost a full table scan. However, if `DB_FILE_MULTIBLOCK_READ_COUNT` is `0` or is not set, then the optimizer uses a value of `8` for calculating cost.

Use the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure to gather workload statistics. The `GATHER_SYSTEM_STATS` procedure refreshes the data dictionary or a staging table with statistics for the elapsed period. To set the duration of the collection, use either of the following techniques:

- Specify `START` the beginning of the workload window, and then `STOP` at the end of the workload window.
- Specify `INTERVAL` and the number of minutes before statistics gathering automatically stops. If needed, you can use `GATHER_SYSTEM_STATS (gathering_mode=>'STOP')` to end gathering earlier than scheduled.

See Also:

Oracle Database Reference to learn about the `DB_FILE_MULTIBLOCK_READ_COUNT` initialization parameter

13.4.3.2.2 Starting and Stopping System Statistics Gathering

This tutorial explains how to set the workload interval with the `START` and `STOP` parameters of `GATHER_SYSTEM_STATS`.

Assumptions

This tutorial assumes the following:

- The hour between 10 a.m. and 11 a.m. is representative of the daily workload.
- You intend to collect system statistics directly in the data dictionary.

To gather workload statistics using START and STOP:

1. Start SQL*Plus and connect to the database with administrator privileges.
2. Start statistics collection.

   For example, at 10 a.m., execute the following procedure to start collection:

   ```
   EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS( gathering_mode => 'START' );
   ```

3. Generate the workload.
4. End statistics collection.

   For example, at 11 a.m., execute the following procedure to end collection:

   ```
   EXECUTE DBMS_STATS.GATHER_SYSTEM_STATS( gathering_mode => 'STOP' );
   ```

   The optimizer can now use the workload statistics to generate execution plans that are effective during the normal daily workload.
5. Optionally, query the system statistics.

   For example, run the following query:

   ```
   COL PNAME FORMAT a15
   SELECT PNAME, PVAL1
   FROM   SYS.AUX_STATS$
   WHERE  SNAME = 'SYSSTATS_MAIN';
   ```

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure

13.4.3.2.3 Gathering System Statistics During a Specified Interval

This tutorial explains how to set the workload interval with the `INTERVAL` parameter of `GATHER_SYSTEM_STATS`.

Assumptions

This tutorial assumes the following:

- The database application processes OLTP transactions during the day and runs OLAP reports at night. To gather representative statistics, you collect them during the day for two hours and then at night for two hours.
- You want to store statistics in a table named `workload_stats`.
- You intend to switch between the statistics gathered.

To gather workload statistics using INTERVAL:

1. Start SQL*Plus and connect to the production database as administrator `dba1`.
2. Create a table to hold the production statistics.

   For example, execute the following PL/SQL program to create user statistics table `workload_stats`:

   ```
   BEGIN
     DBMS_STATS.CREATE_STAT_TABLE (
         ownname  =>  'dba1'
   ,     stattab  =>  'workload_stats'
   );
   END;
   /
   ```

3. Ensure that `JOB_QUEUE_PROCESSES` is not `0` so that `DBMS_JOB` jobs and Oracle Scheduler jobs run.

   ```
   ALTER SYSTEM SET JOB_QUEUE_PROCESSES = 1;
   ```

4. Gather statistics during the day.

   For example, gather statistics for two hours with the following program:

   ```
   BEGIN
     DBMS_STATS.GATHER_SYSTEM_STATS (
         interval  =>  120
   ,     stattab   => 'workload_stats'
   ,     statid    => 'OLTP'
   );
   END;
   /
   ```

5. Gather statistics during the evening.

For example, gather statistics for two hours with the following program:

```
BEGIN
  DBMS_STATS.GATHER_SYSTEM_STATS (
      interval   =>  120
,     stattab    => 'workload_stats'
,     statid     => 'OLAP'
);
END;
/
```

6. In the day or evening, import the appropriate statistics into the data dictionary.

   For example, during the day you can import the OLTP statistics from the staging table into the dictionary with the following program:

```
BEGIN
  DBMS_STATS.IMPORT_SYSTEM_STATS (
      stattab   =>  'workload_stats'
,     statid    =>  'OLTP'
);
END;
/
```

   For example, during the night you can import the OLAP statistics from the staging table into the dictionary with the following program:

```
BEGIN
  DBMS_STATS.IMPORT_SYSTEM_STATS (
      stattab   =>  'workload_stats'
,     statid    =>  'OLAP'
);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure

13.4.3.3 Gathering Noworkload Statistics

Noworkload statistics capture characteristics of the I/O system.

By default, Oracle Database uses noworkload statistics and the CPU cost model. The values of noworkload statistics are initialized to defaults at the first instance startup. You can also use the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure to gather noworkload statistics manually.

Noworkload statistics include the following system statistics listed in Table 13-4:

- I/O transfer speed (`iotfrspeed`)

- I/O seek time (`ioseektim`)
- CPU speed (`cpuspeednw`)

The major difference between workload statistics and noworkload statistics is in the gathering method. Noworkload statistics gather data by submitting random reads against all data files, whereas workload statistics uses counters updated when database activity occurs. If you gather workload statistics, then Oracle Database uses them instead of noworkload statistics.

To gather noworkload statistics, run `DBMS_STATS.GATHER_SYSTEM_STATS` with no arguments or with the gathering mode set to `noworkload`. There is an overhead on the I/O system during the gathering process of noworkload statistics. The gathering process may take from a few seconds to several minutes, depending on I/O performance and database size.

When you gather noworkload statistics, the database analyzes the information and verifies it for consistency. In some cases, the values of noworkload statistics may retain their default values. You can either gather the statistics again, or use `SET_SYSTEM_STATS` to set the values manually to the I/O system specifications.

Assumptions

This tutorial assumes that you want to gather noworkload statistics manually.

To gather noworkload statistics manually:

1. Start SQL*Plus and connect to the database with administrator privileges.
2. Gather the noworkload statistics.

   For example, run the following statement:

   ```
   BEGIN
     DBMS_STATS.GATHER_SYSTEM_STATS (
       gathering_mode => 'NOWORKLOAD'
   );
   END;
   ```

3. Optionally, query the system statistics.

   For example, run the following query:

   ```
   COL PNAME FORMAT a15

   SELECT PNAME, PVAL1
   FROM   SYS.AUX_STATS$
   WHERE  SNAME = 'SYSSTATS_MAIN';
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.GATHER_SYSTEM_STATS` procedure

13.4.4 Deleting System Statistics

The `DBMS_STATS.DELETE_SYSTEM_STATS` procedure deletes system statistics.

This procedure deletes workload statistics collected using the `INTERVAL` or `START` and `STOP` options, and then resets the default to noworkload statistics. However, if the `stattab` parameter specifies a table for storing statistics, then the subprogram deletes all system statistics with the associated `statid` from the statistics table.

If the database is newly created, then Oracle recommends deleting system statistics, shutting down the database, and then reopening the database. This sequence of steps ensures that the database establishes appropriate defaults for system statistics.

Assumptions

This tutorial assumes the following:

- You gathered statistics for a specific intensive workload, but no longer want the optimizer to use these statistics.
- You stored workload statistics in the default location, not in a user-specified table.

To delete system statistics:

1. In SQL*Plus, log in to the database as a user with administrative privileges.
2. Delete the system statistics.

   For example, run the following statement:

   ```
   EXEC DBMS_STATS.DELETE_SYSTEM_STATS;
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.DELETE_SYSTEM_STATS` procedure

13.5 Running Statistics Gathering Functions in Reporting Mode

You can run the `DBMS_STATS` statistics gathering procedures in reporting mode.

When you use the `REPORT_*` procedures, the optimizer does not actually gather statistics. Rather, the package reports objects that *would* be processed if you were to use a specified statistics gathering function.

The following table lists the `DBMS_STATS.REPORT_GATHER_*_STATS` functions. For all functions, the input parameters are the same as for the corresponding `GATHER_*_STATS` procedure, with the following additional parameters: `detail_level` and `format`. Supported formats are `XML`, `HTML`, and `TEXT`.

Table 13-5 DBMS_STATS Reporting Mode Functions

Function	Description
`REPORT_GATHER_TABLE_STATS`	Runs `GATHER_TABLE_STATS` in reporting mode. The procedure does not collect statistics, but reports all objects that would be affected by invoking `GATHER_TABLE_STATS`.
`REPORT_GATHER_SCHEMA_STATS`	Runs `GATHER_SCHEMA_STATS` in reporting mode. The procedure does not actually collect statistics, but reports all objects that would be affected by invoking `GATHER_SCHEMA_STATS`.
`REPORT_GATHER_DICTIONARY_STATS`	Runs `GATHER_DICTIONARY_STATS` in reporting mode. The procedure does not actually collect statistics, but reports all objects that would be affected by invoking `GATHER_DICTIONARY_STATS`.
`REPORT_GATHER_DATABASE_STATS`	Runs `GATHER_DATABASE_STATS` in reporting mode. The procedure does not actually collect statistics, but reports all objects that would be affected by invoking `GATHER_DATABASE_STATS`.
`REPORT_GATHER_FIXED_OBJ_STATS`	Runs `GATHER_FIXED_OBJ_STATS` in reporting mode. The procedure does not actually collect statistics, but reports all objects that would be affected by invoking `GATHER_FIXED_OBJ_STATS`.
`REPORT_GATHER_AUTO_STATS`	Runs the automatic statistics gather job in reporting mode. The procedure does not actually collect statistics, but reports all objects that would be affected by running the job.

Assumptions

This tutorial assumes that you want to generate an HTML report of the objects that would be affected by running `GATHER_SCHEMA_STATS` on the `oe` schema.

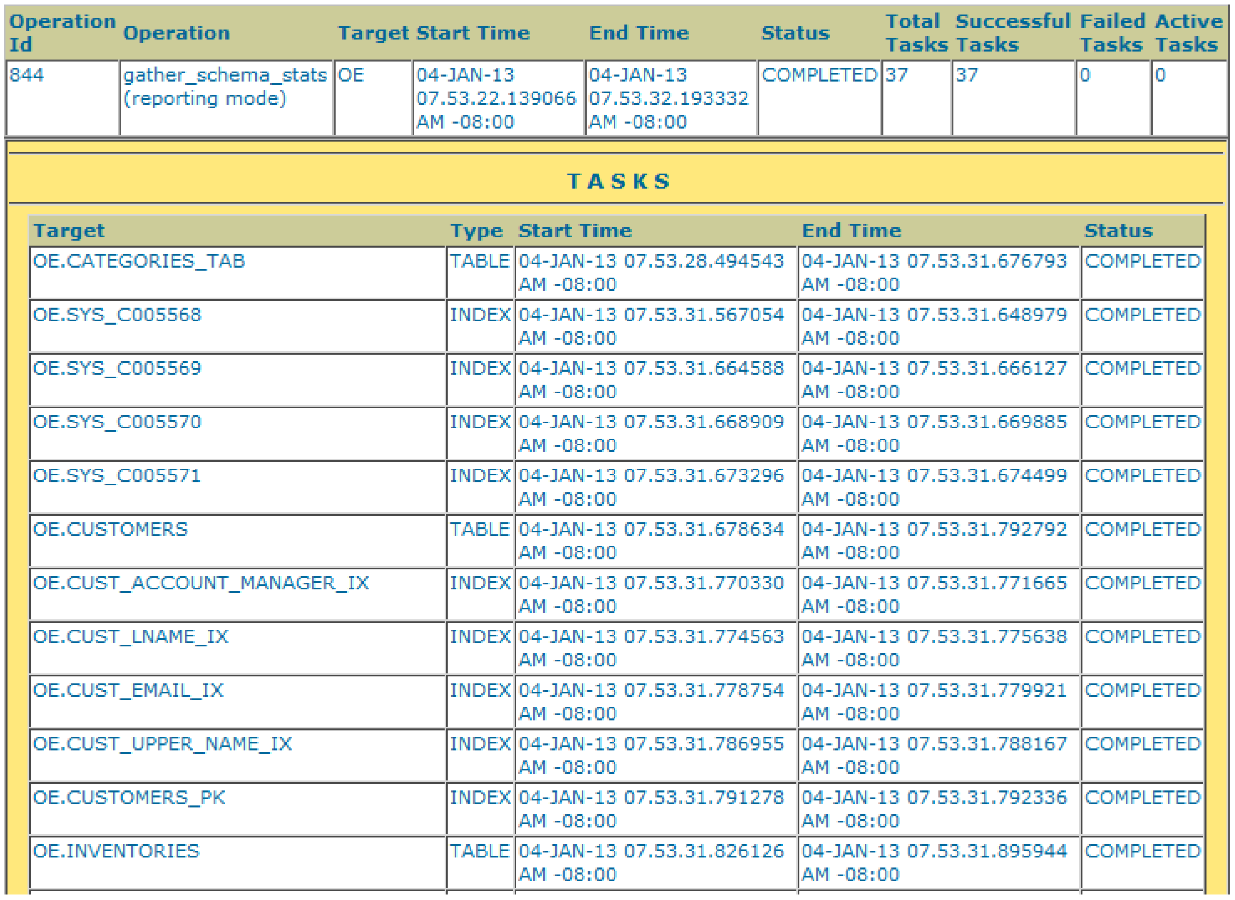
To report on objects affected by running GATHER_SCHEMA_STATS:

1. Start SQL*Plus and connect to the database with administrator privileges.
2. Run the `DBMS_STATS.REPORT_GATHER_SCHEMA_STATS` function.

   For example, run the following commands in SQL*Plus:

```
SET LINES 200 PAGES 0
SET LONG 100000
COLUMN REPORT FORMAT A200

VARIABLE my_report CLOB;
BEGIN
  :my_report :=DBMS_STATS.REPORT_GATHER_SCHEMA_STATS(
    ownname      => 'OE'        ,
    detail_level => 'TYPICAL'   ,
    format       => 'HTML'      );
END;
/
```

The following graphic shows a partial example report:



Operation Id	Operation	Target	Start Time	End Time	Status	Total Tasks	Successful Tasks	Failed Tasks	Active Tasks
844	gather_schema_stats (reporting mode)	OE	04-JAN-13 07.53.22.139066 AM -08:00	04-JAN-13 07.53.32.193332 AM -08:00	COMPLETED	37	37	0	0

TASKS

Target	Type	Start Time	End Time	Status
OE.CATEGORIES_TAB	TABLE	04-JAN-13 07.53.28.494543 AM -08:00	04-JAN-13 07.53.31.676793 AM -08:00	COMPLETED
OE.SYS_C005568	INDEX	04-JAN-13 07.53.31.567054 AM -08:00	04-JAN-13 07.53.31.648979 AM -08:00	COMPLETED
OE.SYS_C005569	INDEX	04-JAN-13 07.53.31.664588 AM -08:00	04-JAN-13 07.53.31.666127 AM -08:00	COMPLETED
OE.SYS_C005570	INDEX	04-JAN-13 07.53.31.668909 AM -08:00	04-JAN-13 07.53.31.669885 AM -08:00	COMPLETED
OE.SYS_C005571	INDEX	04-JAN-13 07.53.31.673296 AM -08:00	04-JAN-13 07.53.31.674499 AM -08:00	COMPLETED
OE.CUSTOMERS	TABLE	04-JAN-13 07.53.31.678634 AM -08:00	04-JAN-13 07.53.31.792792 AM -08:00	COMPLETED
OE.CUST_ACCOUNT_MANAGER_IX	INDEX	04-JAN-13 07.53.31.770330 AM -08:00	04-JAN-13 07.53.31.771665 AM -08:00	COMPLETED
OE.CUST_LNAME_IX	INDEX	04-JAN-13 07.53.31.774563 AM -08:00	04-JAN-13 07.53.31.775638 AM -08:00	COMPLETED
OE.CUST_EMAIL_IX	INDEX	04-JAN-13 07.53.31.778754 AM -08:00	04-JAN-13 07.53.31.779921 AM -08:00	COMPLETED
OE.CUST_UPPER_NAME_IX	INDEX	04-JAN-13 07.53.31.786955 AM -08:00	04-JAN-13 07.53.31.788167 AM -08:00	COMPLETED
OE.CUSTOMERS_PK	INDEX	04-JAN-13 07.53.31.791278 AM -08:00	04-JAN-13 07.53.31.792336 AM -08:00	COMPLETED
OE.INVENTORIES	TABLE	04-JAN-13 07.53.31.826126 AM -08:00	04-JAN-13 07.53.31.895944 AM -08:00	COMPLETED

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS`

14 Managing Extended Statistics

DBMS_STATS enables you to collect **extended statistics**, which are statistics that can improve cardinality estimates when multiple predicates exist on different columns of a table, or when predicates use expressions.

An **extension** is either a column group or an expression. Column group statistics can improve cardinality estimates when multiple columns from the same table occur together in a SQL statement. Expression statistics improves optimizer estimates when predicates use expressions, for example, built-in or user-defined functions.

Note:

You cannot create extended statistics on virtual columns.

See Also:

Oracle Database SQL Language Reference for a list of restrictions on virtual columns

14.1 Managing Column Group Statistics

A **column group** is a set of columns that is treated as a unit.

Essentially, a column group is a virtual column. By gathering statistics on a column group, the optimizer can more accurately determine the cardinality estimate when a query groups these columns together.

The following sections provide an overview of column group statistics, and explain how to manage them manually.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the DBMS_STATS package

14.1.1 About Statistics on Column Groups

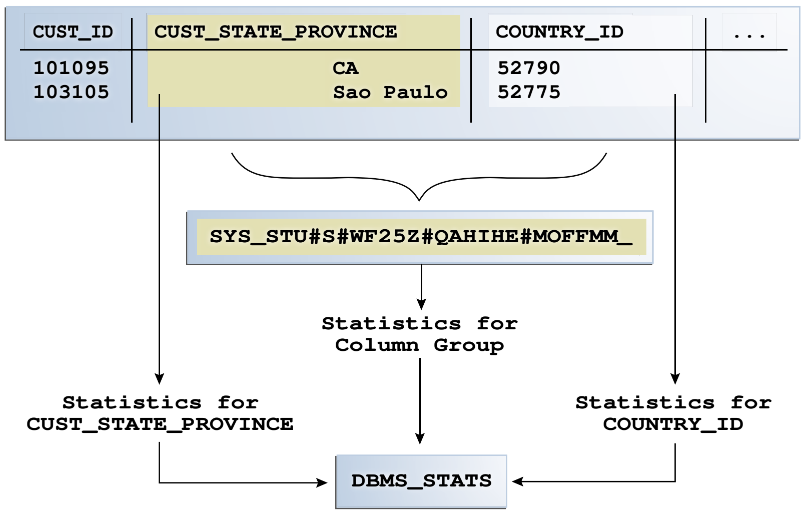
Individual column statistics are useful for determining the selectivity of a single predicate in a WHERE clause.

When the WHERE clause includes multiple predicates on different columns from the same table, individual column statistics do not show the relationship between the columns. This is the problem solved by a column group.

The optimizer calculates the selectivity of the predicates independently, and then combines them. However, if a correlation between the individual columns exists, then the optimizer cannot take it into account when determining a cardinality estimate, which it creates by multiplying the selectivity of each table predicate by the number of rows.

The following graphic contrasts two ways of gathering statistics on the cust_state_province and country_id columns of the sh.customers table. The diagram shows DBMS_STATS collecting statistics on each column individually and on the group. The column group has a system-generated name.

Figure 14-1 Column Group Statistics


CUST_ID
CUST_STATE_PROVINCE
COUNTRY_ID
...
101095
103105
CA
Sao Paulo
52790
52775
SYS_STU#S#WF25Z#QAHIHE#MOFFMM_
Statistics for Column Group
Statistics for CUST_STATE_PROVINCE
Statistics for COUNTRY_ID
DBMS_STATS

Note:

The optimizer uses column group statistics for equality predicates, inlist predicates, and for estimating the GROUP BY cardinality.

14.1.1.1 Why Column Group Statistics Are Needed: Example

This example demonstrates how column group statistics enable the optimizer to give a more accurate cardinality estimate.

The following query of the DBA_TAB_COL_STATISTICS table shows information about statistics that have been gathered on the columns cust_state_province and country_id from the sh.customers table:

```
COL COLUMN_NAME FORMAT a20
COL NDV FORMAT 999
```

```
SELECT COLUMN_NAME, NUM_DISTINCT AS "NDV", HISTOGRAM
FROM   DBA_TAB_COL_STATISTICS
WHERE  OWNER = 'SH'
AND    TABLE_NAME = 'CUSTOMERS'
AND    COLUMN_NAME IN ('CUST_STATE_PROVINCE', 'COUNTRY_ID');
```

Sample output is as follows:

```
COLUMN_NAME                  NDV HISTOGRAM
-------------------- ---------- ---------------
CUST_STATE_PROVINCE          145 FREQUENCY
COUNTRY_ID                    19 FREQUENCY
```

As shown in the following query, 3341 customers reside in California:

```
SELECT COUNT(*)
FROM   sh.customers
WHERE  cust_state_province = 'CA';

 COUNT(*)
----------
    3341
```

Consider an explain plan for a query of customers in the state `CA` and in the country with ID `52790` (USA):

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   sh.customers
  WHERE  cust_state_province = 'CA'
  AND    country_id=52790;

Explained.

sys@PROD> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 1683234692

--------------------------------------------------------------------------
| Id  | Operation         | Name      | Rows | Bytes |Cost (%CPU)| Time   |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |           |  128 | 24192 | 442  (7)| 00:00:06 |
|*  1 |  TABLE ACCESS FULL| CUSTOMERS |  128 | 24192 | 442  (7)| 00:00:06 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
```

```
   1 - filter("CUST_STATE_PROVINCE"='CA' AND "COUNTRY_ID"=52790)

13 rows selected.
```

Based on the single-column statistics for the `country_id` and `cust_state_province` columns, the optimizer estimates that the query of California customers in the USA will return 128 rows. In fact, 3341 customers reside in California, but the optimizer does not know that the state of California is in the country of the USA, and so greatly underestimates cardinality by assuming that both predicates reduce the number of returned rows.

You can make the optimizer aware of the real-world relationship between values in `country_id` and `cust_state_province` by gathering column group statistics. These statistics enable the optimizer to give a more accurate cardinality estimate.

See Also:

- "Detecting Useful Column Groups for a Specific Workload"
- "Creating Column Groups Detected During Workload Monitoring"
- "Creating and Gathering Statistics on Column Groups Manually"

14.1.1.2 Automatic and Manual Column Group Statistics

Oracle Database can create column group statistics either automatically or manually.

The optimizer can use SQL plan directives to generate a more optimal plan. If the `DBMS_STATS` preference `AUTO_STAT_EXTENSIONS` is set to `ON` (by default it is `OFF`), then a SQL plan directive can automatically trigger the creation of column group statistics based on usage of predicates in the workload. You can set `AUTO_STAT_EXTENSIONS` with the `SET_TABLE_PREFS`, `SET_GLOBAL_PREFS`, or `SET_SCHEMA_PREFS` procedures.

When you want to manage column group statistics manually, then use `DBMS_STATS` as follows:

- Detect column groups
- Create previously detected column groups
- Create column groups manually and gather column group statistics

See Also:

- "Detecting Useful Column Groups for a Specific Workload"
- "Creating Column Groups Detected During Workload Monitoring"
- "Creating and Gathering Statistics on Column Groups Manually"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` procedures for setting optimizer statistics

14.1.1.3 User Interface for Column Group Statistics

Several DBMS_STATS program units have preferences that are relevant for column groups.

Table 14-1 DBMS_STATS APIs Relevant for Column Groups

Program Unit or Preference	Description
SEED_COL_USAGE Procedure	Iterates over the SQL statements in the specified workload, compiles them, and then seeds column usage information for the columns that appear in these statements. To determine the appropriate column groups, the database must observe a representative workload. You do not need to run the queries themselves during the monitoring period. Instead, you can run EXPLAIN PLAN for some longer-running queries in your workload to ensure that the database is recording column group information for these queries.
REPORT_COL_USAGE Function	Generates a report that lists the columns that were seen in filter predicates, join predicates, and GROUP BY clauses in the workload. You can use this function to review column usage information recorded for a specific table.
CREATE_EXTENDED_STATS Function	Creates extensions, which are either column groups or expressions. The database gathers statistics for the extension when either a user-generated or automatic statistics gathering job gathers statistics for the table.
AUTO_STAT_EXTENSIONS Preference	Controls the automatic creation of extensions, including column groups, when optimizer statistics are gathered. Set this preference using SET_TABLE_PREFS, SET_SCHEMA_PREFS, or SET_GLOBAL_PREFS. When AUTO_STAT_EXTENSIONS is set to OFF (default), the database does not create column group statistics automatically. To create extensions, you must execute the CREATE_EXTENDED_STATS function or specify extended statistics explicitly in the METHOD_OPT parameter in the DBMS_STATS API. When set to ON, a SQL plan directive can trigger the creation of column group statistics automatically based on usage of columns in the predicates in the workload.

See Also:

- "Setting Artificial Optimizer Statistics for a Table"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the DBMS_STATS package

14.1.2 Detecting Useful Column Groups for a Specific Workload

You can use `DBMS_STATS.SEED_COL_USAGE` and `REPORT_COL_USAGE` to determine which column groups are required for a table based on a specified workload.

This technique is useful when you do not know which extended statistics to create. This technique does not work for expression statistics.

Assumptions

This tutorial assumes the following:

- Cardinality estimates have been incorrect for queries of the `sh.customers_test` table (created from the `customers` table) that use predicates referencing the columns `country_id` and `cust_state_province`.
- You want the database to monitor your workload for 5 minutes (300 seconds).
- You want the database to determine which column groups are needed automatically.

To detect column groups:

1. Start SQL*Plus or SQL Developer, and log in to the database as user `sh`.
2. Create the `customers_test` table and gather statistics for it:

   ```
   DROP TABLE customers_test;
   CREATE TABLE customers_test AS SELECT * FROM customer;
   EXEC DBMS_STATS.GATHER_TABLE_STATS(user, 'customers_test');
   ```

3. Enable workload monitoring.

   In a different SQL*Plus session, connect as `SYS` and run the following PL/SQL program to enable monitoring for 300 seconds:

   ```
   BEGIN
     DBMS_STATS.SEED_COL_USAGE(null,null,300);
   END;
   /
   ```

4. As user `sh`, run explain plans for two queries in the workload.

   The following examples show the explain plans for two queries on the `customers_test` table:

   ```
   EXPLAIN PLAN FOR
     SELECT *
     FROM   customers_test
     WHERE  cust_city = 'Los Angeles'
     AND    cust_state_province = 'CA'
     AND    country_id = 52790;

   SELECT PLAN_TABLE_OUTPUT
   FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table', null,'basic rows'));
   ```

```
EXPLAIN PLAN FOR
  SELECT   country_id, cust_state_province, count(cust_city)
  FROM     customers_test
  GROUP BY country_id, cust_state_province;

SELECT PLAN_TABLE_OUTPUT
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table', null,'basic rows'));
```

Sample output appears below:

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Plan hash value: 4115398853

----------------------------------------------------
| Id  | Operation         | Name           | Rows  |
----------------------------------------------------
|   0 | SELECT STATEMENT  |                |     1 |
|   1 |  TABLE ACCESS FULL| CUSTOMERS_TEST |     1 |
----------------------------------------------------

8 rows selected.

PLAN_TABLE_OUTPUT
------------------------------------------------------------------------
Plan hash value: 3050654408

-----------------------------------------------------
| Id  | Operation          | Name           | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |                |  1949 |
|   1 |  HASH GROUP BY     |                |  1949 |
|   2 |   TABLE ACCESS FULL| CUSTOMERS_TEST | 55500 |
-----------------------------------------------------

9 rows selected.
```

The first plan shows a cardinality of 1 row for a query that returns 932 rows. The second plan shows a cardinality of 1949 rows for a query that returns 145 rows.

5. Optionally, review the column usage information recorded for the table.

   Call the `DBMS_STATS.REPORT_COL_USAGE` function to generate a report:

```
SET LONG 100000
SET LINES 120
SET PAGES 0
SELECT DBMS_STATS.REPORT_COL_USAGE(user, 'customers_test')
FROM   DUAL;
```

The report appears below:

```
LEGEND:
.......
```

```
EQ         : Used in single table EQuality predicate
RANGE      : Used in single table RANGE predicate
LIKE       : Used in single table LIKE predicate
NULL       : Used in single table is (not) NULL predicate
EQ_JOIN    : Used in EQuality JOIN predicate
NONEQ_JOIN : Used in NON EQuality JOIN predicate
FILTER     : Used in single table FILTER predicate
JOIN       : Used in JOIN predicate
GROUP_BY   : Used in GROUP BY expression
....................................................................
....

####################################################################
####

COLUMN USAGE REPORT FOR SH.CUSTOMERS_TEST
.........................................

1. COUNTRY_ID                          : EQ
2. CUST_CITY                           : EQ
3. CUST_STATE_PROVINCE                 : EQ
4. (CUST_CITY, CUST_STATE_PROVINCE,
    COUNTRY_ID)                        : FILTER
5. (CUST_STATE_PROVINCE, COUNTRY_ID)   : GROUP_BY
####################################################################
####
```

In the preceding report, the first three columns were used in equality predicates in the first monitored query:

```
...
WHERE  cust_city = 'Los Angeles'
AND    cust_state_province = 'CA'
AND    country_id = 52790;
```

All three columns appeared in the same `WHERE` clause, so the report shows them as a group filter. In the second query, two columns appeared in the `GROUP BY` clause, so the report labels them as `GROUP_BY`. The sets of columns in the `FILTER` and `GROUP_BY` report are candidates for column groups.

See Also:

- "Capturing Workloads in SQL Tuning Sets"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` package

14.1.3 Creating Column Groups Detected During Workload Monitoring

You can use the `DBMS_STATS.CREATE_EXTENDED_STATS` function to create column groups that were detected previously by executing `DBMS_STATS.SEED_COL_USAGE`.

Assumptions

This tutorial assumes that you have performed the steps in "Detecting Useful Column Groups for a Specific Workload".

To create column groups:

1. Create column groups for the `customers_test` table based on the usage information captured during the monitoring window.

   For example, run the following query:

   ```
   SELECT DBMS_STATS.CREATE_EXTENDED_STATS(user, 'customers_test') FROM DUAL;
   ```

   Sample output appears below:

   ```
   ###########################################################################
   EXTENSIONS FOR SH.CUSTOMERS_TEST
   ................................
   1. (CUST_CITY, CUST_STATE_PROVINCE,
       COUNTRY_ID)                     :SYS_STUMZ$C3AIHLPBROI#SKA58H_N
   created
   2. (CUST_STATE_PROVINCE, COUNTRY_ID):SYS_STU#S#WF25Z#QAHIHE#MOFFMM_
   created
   ###########################################################################
   ```

   The database created two column groups for `customers_test`: one column group for the filter predicate and one group for the `GROUP BY` operation.

2. Regather table statistics.

   Run `GATHER_TABLE_STATS` to regather the statistics for `customers_test`:

   ```
   EXEC DBMS_STATS.GATHER_TABLE_STATS(user,'customers_test');
   ```

3. As user `sh`, run explain plans for two queries in the workload.

   Check the `USER_TAB_COL_STATISTICS` view to determine which additional statistics were created by the database:

   ```
   SELECT COLUMN_NAME, NUM_DISTINCT, HISTOGRAM
   FROM   USER_TAB_COL_STATISTICS
   WHERE  TABLE_NAME = 'CUSTOMERS_TEST'
   ORDER BY 1;
   ```

Partial sample output appears below:

```
CUST_CITY                                620 HEIGHT BALANCED
...
SYS_STU#S#WF25Z#QAHIHE#MOFFMM_           145 NONE
SYS_STUMZ$C3AIHLPBROI#SKA58H_N           620 HEIGHT BALANCED
```

This example shows the two column group names returned from the `DBMS_STATS.CREATE_EXTENDED_STATS` function. The column group created on `CUST_CITY`, `CUST_STATE_PROVINCE`, and `COUNTRY_ID` has a height-balanced histogram.

4. Explain the plans again.

   The following examples show the explain plans for two queries on the `customers_test` table:

```
EXPLAIN PLAN FOR
  SELECT *
  FROM   customers_test
  WHERE  cust_city = 'Los Angeles'
  AND    cust_state_province = 'CA'
  AND    country_id = 52790;

SELECT PLAN_TABLE_OUTPUT
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table', null,'basic rows'));

EXPLAIN PLAN FOR
  SELECT   country_id, cust_state_province, count(cust_city)
  FROM     customers_test
  GROUP BY country_id, cust_state_province;

SELECT PLAN_TABLE_OUTPUT
FROM   TABLE(DBMS_XPLAN.DISPLAY('plan_table', null,'basic rows'));
```

   The new plans show more accurate cardinality estimates:

```
----------------------------------------------------
| Id  | Operation          | Name            | Rows  |
----------------------------------------------------
|   0 | SELECT STATEMENT   |                 |  1093 |
|   1 |  TABLE ACCESS FULL| CUSTOMERS_TEST |  1093 |
----------------------------------------------------

8 rows selected.

Plan hash value: 3050654408

-----------------------------------------------------
| Id  | Operation          | Name            | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT   |                 |   145 |
|   1 |  HASH GROUP BY     |                 |   145 |
|   2 |   TABLE ACCESS FULL| CUSTOMERS_TEST | 55500 |
-----------------------------------------------------
9 rows selected.
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS` package

14.1.4 Creating and Gathering Statistics on Column Groups Manually

In some cases, you may know the column group that you want to create.

The `METHOD_OPT` argument of the `DBMS_STATS.GATHER_TABLE_STATS` function can create and gather statistics on a column group automatically. You can create a new column group by specifying the group of columns using `FOR COLUMNS`.

Assumptions

This tutorial assumes the following:

- You want to create a column group for the `cust_state_province` and `country_id` columns in the `customers` table in `sh` schema.
- You want to gather statistics (including histograms) on the entire table and the new column group.

To create a column group and gather statistics for this group:

1. In SQL*Plus, log in to the database as the `sh` user.
2. Create the column group and gather statistics.

   For example, execute the following PL/SQL program:

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS( 'sh','customers',
  METHOD_OPT => 'FOR ALL COLUMNS SIZE SKEWONLY ' ||
                'FOR COLUMNS SIZE SKEWONLY (cust_state_province,country_id)' );
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure

14.1.5 Displaying Column Group Information

To obtain the name of a column group, use the `DBMS_STATS.SHOW_EXTENDED_STATS_NAME` function or a database view.

You can also use views to obtain information such as the number of distinct values, and whether the column group has a histogram.

Assumptions

This tutorial assumes the following:

- You created a column group for the `cust_state_province` and `country_id` columns in the `customers` table in `sh` schema.
- You want to determine the column group name, the number of distinct values, and whether a histogram has been created for a column group.

To monitor a column group:

1. Start SQL*Plus and connect to the database as the `sh` user.
2. To determine the column group name, do one of the following.
   - Execute the `SHOW_EXTENDED_STATS_NAME` function.

     For example, run the following PL/SQL program:

     ```
     SELECT SYS.DBMS_STATS.SHOW_EXTENDED_STATS_NAME( 'sh','customers',
            '(cust_state_province,country_id)' ) col_group_name
     FROM   DUAL;
     ```

     The output is similar to the following:

     ```
     COL_GROUP_NAME
     ----------------
     SYS_STU#S#WF25Z#QAHIHE#MOFFMM_
     ```

   - Query the `USER_STAT_EXTENSIONS` view.

     For example, run the following query:

     ```
     SELECT EXTENSION_NAME, EXTENSION
     FROM   USER_STAT_EXTENSIONS
     WHERE  TABLE_NAME='CUSTOMERS';

     EXTENSION_NAME                     EXTENSION
     -----------------------------------------------------------------
     ---
     SYS_STU#S#WF25Z#QAHIHE#MOFFMM_
     ("CUST_STATE_PROVINCE","COUNTRY_ID")
     ```

3. Query the number of distinct values and find whether a histogram has been created for a column group.

For example, run the following query:

```
SELECT e.EXTENSION col_group, t.NUM_DISTINCT, t.HISTOGRAM
FROM   USER_STAT_EXTENSIONS e, USER_TAB_COL_STATISTICS t
WHERE  e.EXTENSION_NAME=t.COLUMN_NAME
AND    e.TABLE_NAME=t.TABLE_NAME
AND    t.TABLE_NAME='CUSTOMERS';

COL_GROUP                             NUM_DISTINCT        HISTOGRAM
-------------------------------------------------------------------
("COUNTRY_ID","CUST_STATE_PROVINCE")  145                 FREQUENCY
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.SHOW_EXTENDED_STATS_NAME` function

14.1.6 Dropping a Column Group

Use the `DBMS_STATS.DROP_EXTENDED_STATS` function to delete a column group from a table.

Assumptions

This tutorial assumes the following:

- You created a column group for the `cust_state_province` and `country_id` columns in the `customers` table in `sh` schema.
- You want to drop the column group.

To drop a column group:

1. Start SQL*Plus and connect to the database as the `sh` user.
2. Drop the column group.

   For example, the following PL/SQL program deletes a column group from the `customers` table:

```
BEGIN
  DBMS_STATS.DROP_EXTENDED_STATS( 'sh', 'customers',
                                  '(cust_state_province, country_id)' );
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.DROP_EXTENDED_STATS` function

14.2 Managing Expression Statistics

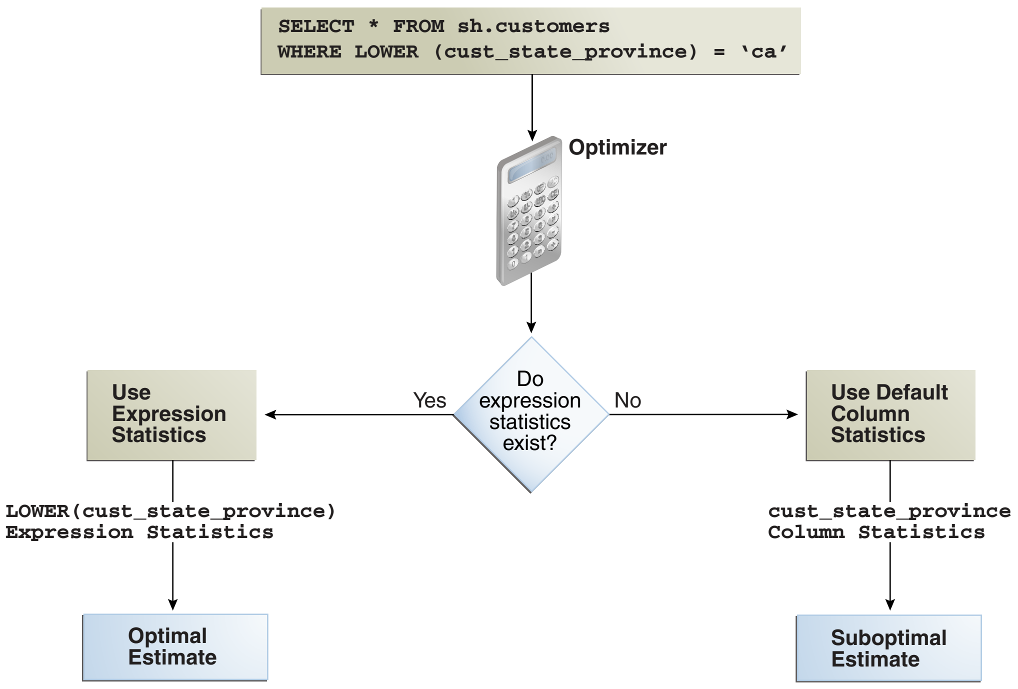
The type of extended statistics known as **expression statistics** improve optimizer estimates when a `WHERE` clause has predicates that use expressions.

14.2.1 About Expression Statistics

For an **expression** in the form `(`*`function`*`(`*`col`*`)=`*`constant`*`)` applied to a `WHERE` clause column, the optimizer does not know how this function affects predicate cardinality unless a function-based index exists. However, you can gather expression statistics on the expression `(`*`function`*`(`*`col`*`)` itself.

The following graphic shows the optimizer using statistics to generate a plan for a query that uses a function. The top shows the optimizer checking statistics for the column. The bottom shows the optimizer checking statistics corresponding to the expression used in the query. The expression statistics yield more accurate estimates.

Figure 14-2   Expression Statistics


SELECT * FROM sh.customers
WHERE LOWER (cust_state_province) = 'ca'
Optimizer
Do expression statistics exist?
Yes
No
Use Expression Statistics
Use Default Column Statistics
LOWER(cust_state_province) Expression Statistics
cust_state_province Column Statistics
Optimal Estimate
Suboptimal Estimate

As shown in Figure 14-2, when expression statistics are not available, the optimizer can produce suboptimal plans.

> **See Also:**
>
> *Oracle Database SQL Language Reference* to learn about SQL functions

14.2.1.1 When Expression Statistics Are Useful: Example

The following query of the `sh.customers` table shows that 3341 customers are in the state of California:

```
sys@PROD> SELECT COUNT(*) FROM sh.customers WHERE cust_state_province='CA';

  COUNT(*)
----------
      3341
```

Consider the plan for the same query with the `LOWER()` function applied:

```
sys@PROD> EXPLAIN PLAN FOR
  2  SELECT * FROM sh.customers WHERE LOWER(cust_state_province)='ca';
Explained.

sys@PROD> select * from table(dbms_xplan.display);

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 2008213504

---------------------------------------------------------------------------
|Id | Operation         | Name      | Rows | Bytes | Cost (%CPU)| Time    |
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT  |           |  555 |  108K |  406   (1)| 00:00:05 |
|*1 |  TABLE ACCESS FULL| CUSTOMERS |  555 |  108K |  406   (1)| 00:00:05 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - filter(LOWER("CUST_STATE_PROVINCE")='ca')
```

Because no expression statistics exist for `LOWER(cust_state_province)='ca'`, the optimizer estimate is significantly off. You can use `DBMS_STATS` procedures to correct these estimates.

14.2.2 Creating Expression Statistics

You can use `DBMS_STATS` to create statistics for a user-specified expression.

You can use either of the following program units:

- `GATHER_TABLE_STATS` procedure
- `CREATE_EXTENDED_STATISTICS` function followed by the `GATHER_TABLE_STATS` procedure

Assumptions

This tutorial assumes the following:

- Selectivity estimates are inaccurate for queries of `sh.customers` that use the `UPPER(cust_state_province)` function.
- You want to gather statistics on the `UPPER(cust_state_province)` expression.

To create expression statistics:

1. Start SQL*Plus and connect to the database as the `sh` user.
2. Gather table statistics.

   For example, run the following command, specifying the function in the `method_opt` argument:

   ```
   BEGIN
     DBMS_STATS.GATHER_TABLE_STATS(
       'sh'
   ,   'customers'
   ,   method_opt => 'FOR ALL COLUMNS SIZE SKEWONLY ' ||
                     'FOR COLUMNS (LOWER(cust_state_province)) SIZE
   SKEWONLY'
   );
   END;
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.GATHER_TABLE_STATS` procedure

14.2.3 Displaying Expression Statistics

To obtain information about expression statistics, use the database view `DBA_STAT_EXTENSIONS` and the `DBMS_STATS.SHOW_EXTENDED_STATS_NAME` function.

You can also use views to obtain information such as the number of distinct values, and whether the column group has a histogram.

Assumptions

This tutorial assumes the following:

- You created extended statistics for the `LOWER(cust_state_province)` expression.
- You want to determine the column group name, the number of distinct values, and whether a histogram has been created for a column group.

To monitor expression statistics:

1. Start SQL*Plus and connect to the database as the `sh` user.
2. Query the name and definition of the statistics extension.

For example, run the following query:

```
COL EXTENSION_NAME FORMAT a30
COL EXTENSION FORMAT a35

SELECT EXTENSION_NAME, EXTENSION
FROM   USER_STAT_EXTENSIONS
WHERE  TABLE_NAME='CUSTOMERS';
```

Sample output appears as follows:

```
EXTENSION_NAME                 EXTENSION
------------------------------ ------------------------------
SYS_STUBPHJSBRKOIK9O2YV3W8HOUE (LOWER("CUST_STATE_PROVINCE"))
```

3. Query the number of distinct values and find whether a histogram has been created for the expression.

   For example, run the following query:

```
SELECT e.EXTENSION expression, t.NUM_DISTINCT, t.HISTOGRAM
FROM   USER_STAT_EXTENSIONS e, USER_TAB_COL_STATISTICS t
WHERE  e.EXTENSION_NAME=t.COLUMN_NAME
AND    e.TABLE_NAME=t.TABLE_NAME
AND    t.TABLE_NAME='CUSTOMERS';

EXPRESSION                                     NUM_DISTINCT        HISTOGRAM
-------------------------------------------------------------------
(LOWER("CUST_STATE_PROVINCE"))        145                 FREQUENCY
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.SHOW_EXTENDED_STATS_NAME` procedure
- *Oracle Database Reference* to learn about the `DBA_STAT_EXTENSIONS` view

14.2.4 Dropping Expression Statistics

To delete a column group from a table, use the `DBMS_STATS.DROP_EXTENDED_STATS` function.

Assumptions

This tutorial assumes the following:

- You created extended statistics for the `LOWER(cust_state_province)` expression.
- You want to drop the expression statistics.

To drop expression statistics:

1. Start SQL*Plus and connect to the database as the `sh` user.

2. Drop the column group.

   For example, the following PL/SQL program deletes a column group from the `customers` table:

   ```
   BEGIN
     DBMS_STATS.DROP_EXTENDED_STATS(
       'sh'
   ,   'customers'
   ,   '(LOWER(cust_state_province))'
   );
   END;
   /
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.DROP_EXTENDED_STATS` procedure

15 Controlling the Use of Optimizer Statistics

Using DBMS_STATS, you can specify when and how the optimizer uses statistics.

15.1 Locking and Unlocking Optimizer Statistics

You can lock statistics to prevent them from changing.

After statistics are locked, you cannot make modifications to the statistics until the statistics have been unlocked. Locking procedures are useful in a static environment when you want to guarantee that the statistics and resulting plan never change. For example, you may want to prevent new statistics from being gathered on a table or schema by the DBMS_STATS_JOB process, such as highly volatile tables.

When you lock statistics on a table, all dependent statistics are locked. The locked statistics include table statistics, column statistics, histograms, and dependent index statistics. To overwrite statistics even when they are locked, you can set the value of the FORCE argument in various DBMS_STATS procedures, for example, DELETE_*_STATS and RESTORE_*_STATS, to true.

15.1.1 Locking Statistics

The DBMS_STATS package provides two procedures for locking statistics: LOCK_SCHEMA_STATS and LOCK_TABLE_STATS.

Assumptions

This tutorial assumes the following:

- You gathered statistics on the oe.orders table and on the hr schema.
- You want to prevent the oe.orders table statistics and hr schema statistics from changing.

To lock statistics:

1. Start SQL*Plus and connect to the database as the oe user.
2. Lock the statistics on oe.orders.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_STATS.LOCK_TABLE_STATS('OE','ORDERS');
   END;
   /
   ```

3. Connect to the database as the hr user.
4. Lock the statistics in the hr schema.

For example, execute the following PL/SQL program:

```
BEGIN
  DBMS_STATS.LOCK_SCHEMA_STATS('HR');
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.LOCK_TABLE_STATS` procedure

15.1.2 Unlocking Statistics

The `DBMS_STATS` package provides two procedures for unlocking statistics: `UNLOCK_SCHEMA_STATS` and `UNLOCK_TABLE_STATS`.

Assumptions

This tutorial assumes the following:

- You locked statistics on the `oe.orders` table and on the `hr` schema.
- You want to unlock these statistics.

To unlock statistics:

1. Start SQL*Plus and connect to the database as the `oe` user.
2. Unlock the statistics on `oe.orders`.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_STATS.UNLOCK_TABLE_STATS('OE','ORDERS');
   END;
   /
   ```

3. Connect to the database as the `hr` user.
4. Unlock the statistics in the `hr` schema.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_STATS.UNLOCK_SCHEMA_STATS('HR');
   END;
   /
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.UNLOCK_TABLE_STATS` procedure

15.2 Publishing Pending Optimizer Statistics

By default, the database automatically publishes statistics when the statistics collection ends.

Alternatively, you can use pending statistics to save the statistics and not publish them immediately after the collection. This technique is useful for testing queries in a session with pending statistics. When the test results are satisfactory, you can publish the statistics to make them available for the entire database.

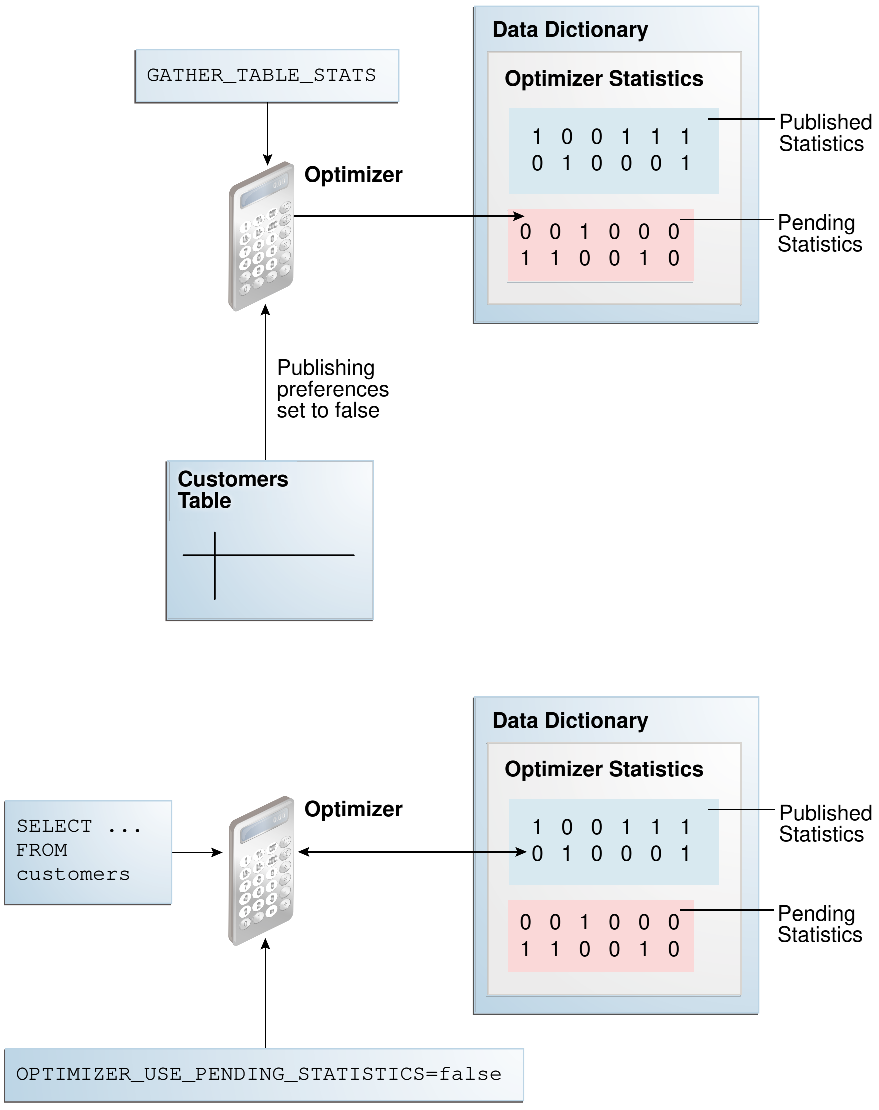
15.2.1 About Pending Optimizer Statistics

The database stores pending statistics in the data dictionary just as for published statistics.

By default, the optimizer uses published statistics. You can change the default behavior by setting the `OPTIMIZER_USE_PENDING_STATISTICS` initialization parameter to `true` (the default is `false`).

The top part of the following graphic shows the optimizer gathering statistics for the `sh.customers` table and storing them in the data dictionary with pending status. The bottom part of the diagram shows the optimizer using only published statistics to process a query of `sh.customers`.

Figure 15-1 Published and Pending Statistics



In some cases, the optimizer can use a combination of published and pending statistics. For example, the database stores both published and pending statistics for the `customers` table. For the `orders` table, the database stores only published statistics. If `OPTIMIZER_USE_PENDING_STATS = true`, then the optimizer uses pending statistics for `customers` and published statistics for `orders`. If `OPTIMIZER_USE_PENDING_STATS = false`, then the optimizer uses published statistics for `customers` and `orders`.

See Also:

Oracle Database Reference to learn about the `OPTIMIZER_USE_PENDING_STATISTICS` initialization parameter

15.2.2 User Interfaces for Publishing Optimizer Statistics

You can use the `DBMS_STATS` package to perform operations relating to publishing statistics.

The following table lists the relevant program units.

Table 15-1 DBMS_STATS Program Units Relevant for Publishing Optimizer Statistics

Program Unit	Description
`GET_PREFS`	Check whether the statistics are automatically published as soon as `DBMS_STATS` gathers them. For the parameter `PUBLISH`, `true` indicates that the statistics must be published when the database gathers them, whereas `false` indicates that the database must keep the statistics pending.
`SET_TABLE_PREFS`	Set the `PUBLISH` setting to `true` or `false` at the table level.
`SET_SCHEMA_PREFS`	Set the `PUBLISH` setting to `true` or `false` at the schema level.
`PUBLISH_PENDING_STATS`	Publish valid pending statistics for all objects or only specified objects.
`DELETE_PENDING_STATS`	Delete pending statistics.
`EXPORT_PENDING_STATS`	Export pending statistics.

The initialization parameter `OPTIMIZER_USE_PENDING_STATISTICS` determines whether the database uses pending statistics when they are available. The default value is `false`, which means that the optimizer uses only published statistics. Set to `true` to specify that the optimizer uses any existing pending statistics instead. The best practice is to set this parameter at the session level rather than at the database level.

You can use access information about published statistics from data dictionary views. Table 15-2 lists relevant views.

Table 15-2 Views Relevant for Publishing Optimizer Statistics

View	Description
`USER_TAB_STATISTICS`	Displays optimizer statistics for the tables accessible to the current user.
`USER_TAB_COL_STATISTICS`	Displays column statistics and histogram information extracted from `ALL_TAB_COLUMNS`.
`USER_PART_COL_STATISTICS`	Displays column statistics and histogram information for the table partitions owned by the current user.
`USER_SUBPART_COL_STATISTICS`	Describes column statistics and histogram information for subpartitions of partitioned objects owned by the current user.
`USER_IND_STATISTICS`	Displays optimizer statistics for the indexes accessible to the current user.
`USER_TAB_PENDING_STATS`	Describes pending statistics for tables, partitions, and subpartitions accessible to the current user.
`USER_COL_PENDING_STATS`	Describes the pending statistics of the columns accessible to the current user.

Table 15-2 (Cont.) Views Relevant for Publishing Optimizer Statistics

View	Description
`USER_IND_PENDING_STATS`	Describes the pending statistics for tables, partitions, and subpartitions accessible to the current user collected using the `DBMS_STATS` package.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` package
- *Oracle Database Reference* to learn about `USER_TAB_PENDING_STATS` and related views

15.2.3 Managing Published and Pending Statistics

This section explains how to use `DBMS_STATS` program units to change the publishing behavior of optimizer statistics, and also to export and delete these statistics.

Assumptions

This tutorial assumes the following:

- You want to change the preferences for the `sh.customers` and `sh.sales` tables so that newly collected statistics have pending status.
- You want the current session to use pending statistics.
- You want to gather and publish pending statistics on the `sh.customers` table.
- You gather the pending statistics on the `sh.sales` table, but decide to delete them without publishing them.
- You want to change the preferences for the `sh.customers` and `sh.sales` tables so that newly collected statistics are published.

To manage published and pending statistics:

1. Start SQL*Plus and connect to the database as user `sh`.
2. Query the global optimizer statistics publishing setting.

   Run the following query (sample output included):

   ```
   sh@PROD> SELECT DBMS_STATS.GET_PREFS('PUBLISH') PUBLISH FROM DUAL;

   PUBLISH
   -------
   TRUE
   ```

   The value `true` indicates that the database publishes statistics as it gathers them. Every table uses this value unless a specific table preference has been set.

When using GET_PREFS, you can also specify a schema and table name. The function returns a table preference if it is set. Otherwise, the function returns the global preference.

3. Query the pending statistics.

   For example, run the following query (sample output included):

```
sh@PROD> SELECT * FROM USER_TAB_PENDING_STATS;

no rows selected
```

   This example shows that the database currently stores no pending statistics for the sh schema.

4. Change the publishing preferences for the sh.customers table.

   For example, execute the following procedure so that statistics are marked as pending:

```
BEGIN
  DBMS_STATS.SET_TABLE_PREFS('sh', 'customers', 'publish', 'false');
END;
/
```

   Subsequently, when you gather statistics on the customers table, the database does not automatically publish statistics when the gather job completes. Instead, the database stores the newly gathered statistics in the USER_TAB_PENDING_STATS table.

5. Gather statistics for sh.customers.

   For example, run the following program:

```
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS('sh','customers');
END;
/
```

6. Query the pending statistics.

   For example, run the following query (sample output included):

```
sh@PROD> SELECT TABLE_NAME, NUM_ROWS FROM USER_TAB_PENDING_STATS;

TABLE_NAME                       NUM_ROWS
------------------------------ ----------
CUSTOMERS                           55500
```

   This example shows that the database now stores pending statistics for the sh.customers table.

7. Instruct the optimizer to use the pending statistics in this session.

   Set the initialization parameter OPTIMIZER_USE_PENDING_STATISTICS to true as shown:

```
ALTER SESSION SET OPTIMIZER_USE_PENDING_STATISTICS = true;
```

8. Run a workload.

The following example changes the email addresses of all customers named Bruce Chalmers:

```
UPDATE  sh.customers
  SET   cust_email='ChalmersB@company.example.icom'
  WHERE cust_first_name = 'Bruce'
  AND   cust_last_name = 'Chalmers';
COMMIT;
```

The optimizer uses the pending statistics instead of the published statistics when compiling all SQL statements in this session.

9. Publish the pending statistics for `sh.customers`.

   For example, execute the following program:

   ```
   BEGIN
     DBMS_STATS.PUBLISH_PENDING_STATS('SH','CUSTOMERS');
   END;
   /
   ```

10. Change the publishing preferences for the `sh.sales` table.

    For example, execute the following program:

    ```
    BEGIN
      DBMS_STATS.SET_TABLE_PREFS('sh', 'sales', 'publish', 'false');
    END;
    /
    ```

    Subsequently, when you gather statistics on the `sh.sales` table, the database does not automatically publish statistics when the gather job completes. Instead, the database stores the statistics in the `USER_TAB_PENDING_STATS` table.

11. Gather statistics for `sh.sales`.

    For example, run the following program:

    ```
    BEGIN
      DBMS_STATS.GATHER_TABLE_STATS('sh','sales');
    END;
    /
    ```

12. Delete the pending statistics for `sh.sales`.

    Assume you change your mind and now want to delete pending statistics for `sh.sales`. Run the following program:

    ```
    BEGIN
      DBMS_STATS.DELETE_PENDING_STATS('sh','sales');
    END;
    /
    ```

13. Change the publishing preferences for the `sh.customers` and `sh.sales` tables back to their default setting.

For example, execute the following program:

```
BEGIN
  DBMS_STATS.SET_TABLE_PREFS('sh', 'customers', 'publish', null);
  DBMS_STATS.SET_TABLE_PREFS('sh', 'sales', 'publish', null);
END;
/
```

15.3 Creating Artificial Optimizer Statistics for Testing

To provide the optimizer with user-created statistics for testing purposes, you can use the `DBMS_STATS.SET_*_STATS` procedures. These procedures provide the optimizer with artificial values for the specified statistics.

15.3.1 About Artificial Optimizer Statistics

For testing purposes, you can manually create artificial statistics for a table, index, or the system using the `DBMS_STATS.SET_*_STATS` procedures.

When `stattab` is null, the `DBMS_STATS.SET_*_STATS` procedures insert the artificial statistics into the data dictionary directly. Alternatively, you can specify a user-created table.

Caution:

The `DBMS_STATS.SET_*_STATS` procedures are intended for development testing only. Do not use them in a production database. If you set statistics in the data dictionary, then Oracle Database considers the set statistics as the “real” statistics, which means that statistics gathering jobs may not re-gather artificial statistics when they do not meet the criteria for staleness.

Typical use cases for the `DBMS_STATS.SET_*_STATS` procedures are:

- Showing how execution plans change as the numbers of rows or blocks in a table change

  For example, `SET_TABLE_STATS` can set number of rows and blocks in a small or empty table to a large number. When you execute a query using the altered statistics, the optimizer may change the execution plan. For example, the increased row count may lead the optimizer to choose an index scan rather than a full table scan. By experimenting with different values, you can see how the optimizer will change its execution plan over time.
- Creating realistic statistics for temporary tables

  You may want to see what the optimizer does when a large temporary table is referenced in multiple SQL statements. You can create a regular table, load representative data, and then use `GET_TABLE_STATS` to retrieve the statistics. After you create the temporary table, you can “deceive” the optimizer into using these statistics by invoking `SET_TABLE_STATS`.

Optionally, you can specify a unique ID for statistics in a user-created table. The `SET_*_STATS` procedures have corresponding `GET_*_STATS` procedures.

Table 15-3 DBMS_STATS Procedures for Setting Optimizer Statistics

DBMS_STATS Procedure	Description
`SET_TABLE_STATS`	Sets table or partition statistics using parameters such as `numrows`, `numblks`, and `avgrlen`. If the database uses the In-Memory Column store, you can set `im_imcu_count` to the number of IMCUs in the table or partition, and `im_block_count` to the number of blocks in the table or partition. For an external table, `scanrate` specifies the rate at which data is scanned in MB/second. The optimizer uses the cached data to estimate the number of cached blocks for index or statistics table access. The total cost is the I/O cost of reading data blocks from disk, the CPU cost of reading cached blocks from the buffer cache, and the CPU cost of processing the data.
`SET_COLUMN_STATS`	Sets column statistics using parameters such as `distcnt`, `density`, `nullcnt`, and so on. In the version of this procedure that deals with user-defined statistics, use `stattypname` to specify the type of statistics to store in the data dictionary.
`SET_SYSTEM_STATS`	Sets system statistics using parameters such as `iotfrspeed`, `sreadtim`, and `cpuspeed`.
`SET_INDEX_STATS`	Sets index statistics using parameters such as `numrows`, `numlblks`, `avglblk`, `clstfct`, and `indlevel`. In the version of this procedure that deals with user-defined statistics, use `stattypname` to specify the type of statistics to store in the data dictionary.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.SET_TABLE_STATS` and the other procedures for setting optimizer statistics

15.3.2 Setting Artificial Optimizer Statistics for a Table

This topic explains how to set artificial statistics for a table using `DBMS_STATS.SET_TABLE_STATS`. The basic steps are the same for `SET_INDEX_STATS` and `SET_SYSTEM_STATS`.

Note the following task prerequisites:

- For an object not owned by `SYS`, you must be the owner of the object, or have the `ANALYZE ANY` privilege.
- For an object owned by `SYS`, you must have the `ANALYZE ANY DICTIONARY` privilege or the `SYSDBA` privilege.
- When invoking `GET_*_STATS` for a table, column, or index, the referenced object must exist.

This task assumes the following:

- You have the required privileges to use `DBMS_STATS.SET_TABLE_STATS` for the specified table.
- You intend to store the statistics in the data dictionary.

1. In SQL*Plus, log in to the database as a user with the required privileges.
2. Run the `DBMS_STATS.SET_TABLE_STATS` procedure, specifying the appropriate parameters for the statistics.

   Typical parameters include the following:

   - `ownname` (not null)

     This parameter specifies the name of the schema containing the table.
   - `tabname` (not null)

     This parameter specifies the name of the table whose statistics you intend to set.
   - `partname`

     This parameter specifies the name of a partition of the table.
   - `numrows`

     This parameter specifies the number of rows in the table.
   - `numblks`

     This parameter specifies the number of blocks in the table.
3. Query the table.
4. Optionally, to determine how the statistics affected the optimizer, query the execution plan.
5. Optionally, to perform further testing, return to Step 2 and reset the optimizer statistics.

15.3.3 Setting Optimizer Statistics: Example

This example shows how to gather optimizer statistics for a table, set artificial statistics, and then compare the plans that the optimizer chooses based on the differing statistics.

This example assumes:

- You are logged in to the database as a user with DBA privileges.
- You want to test when the optimizer chooses an index scan.

1. Create a table called `contractors`, and index the `salary` column.

```
CREATE TABLE contractors (
  con_id    NUMBER,
  last_name VARCHAR2(50),
  salary    NUMBER,
  CONSTRAINT cond_id_pk PRIMARY KEY(con_id) );

CREATE INDEX salary_ix ON contractors(salary);
```

2. Insert a single row into this table.

```
INSERT INTO contractors VALUES (8, 'JONES',1000);
COMMIT;
```

3. Gather statistics for the table.

```
EXECUTE DBMS_STATS.GATHER_TABLE_STATS( user, tabname =>
'CONTRACTORS' );
```

4. Query the number of rows for the table and index (sample output included):

```
SQL> SELECT NUM_ROWS FROM USER_TABLES WHERE TABLE_NAME =
'CONTRACTORS';

  NUM_ROWS
----------
         1

SQL> SELECT NUM_ROWS FROM USER_INDEXES WHERE INDEX_NAME =
'SALARY_IX';

  NUM_ROWS
----------
         1
```

5. Query contractors whose salary is 1000, using the `dynamic_sampling` hint to disable dynamic sampling:

```
SELECT /*+ dynamic_sampling(contractors 0) */ *
FROM   contractors
WHERE  salary = 1000;
```

6. Query the execution plan chosen by the optimizer (sample output included):

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

SQL_ID    cy0wzytc16g9n, child number 0
-------------------------------------
SELECT /*+ dynamic_sampling(contractors 0) */ * FROM contractors
WHERE
salary = 1000

Plan hash value: 5038823

--------------------------------------------------------------------
--
| Id  | Operation         | Name        |Rows|Bytes|Cost (%CPU)|
Time|
--------------------------------------------------------------------
--
|   0 | SELECT STATEMENT  |             |    |     | 2
(100)|          |
|*  1 |  TABLE ACCESS FULL| CONTRACTORS | 1  | 12 | 2   (0)|
00:00:01 |
--------------------------------------------------------------------
--

Predicate Information (identified by operation id):
```

```
---------------------------------------------------

   1 - filter("SALARY"=1000)

19 rows selected.
```

Because only 1 row exists in the table, the optimizer chooses a full table scan over an index range scan.

7. Use `SET_TABLE_STATS` and `SET_INDEX_STATS` to simulate statistics for a table with 2000 rows stored in 10 data blocks:

```
BEGIN
  DBMS_STATS.SET_TABLE_STATS(
    ownname => user
  , tabname => 'CONTRACTORS'
  , numrows => 2000
  , numblks => 10 );
END;
/

BEGIN
  DBMS_STATS.SET_INDEX_STATS(
    ownname => user
  , indname => 'SALARY_IX'
  , numrows => 2000 );
END;
/
```

8. Query the number of rows for the table and index (sample output included):

```
SQL> SELECT NUM_ROWS FROM USER_TABLES WHERE TABLE_NAME = 'CONTRACTORS';

  NUM_ROWS
----------
      2000

SQL> SELECT NUM_ROWS FROM USER_INDEXES WHERE INDEX_NAME =  'SALARY_IX';

  NUM_ROWS
----------
      2000
```

Now the optimizer believes that the table contains 2000 rows in 10 blocks, even though only 1 row actually exists in one block.

9. Flush the shared pool to eliminate possibility of plan reuse, and then execute the same query of `contractors`:

```
ALTER SYSTEM FLUSH SHARED_POOL;

SELECT /*+ dynamic_sampling(contractors 0) */ *
FROM   contractors
WHERE  salary = 1000;
```

10. Query the execution plan chosen by the optimizer based on the artificial statistics (sample output included):

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

SQL_ID    cy0wzytc16g9n, child number 0
-------------------------------------
SELECT /*+ dynamic_sampling(contractors 0) */ * FROM contractors WHERE
salary = 1000

Plan hash value: 996794789

---------------------------------------------------------------------------------
|Id| Operation                          | Name      |Rows|Bytes|Cost(%CPU)|Time|
---------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |           |    |     |3(100)|        |
| 1|  TABLE ACCESS BY INDEX ROWID BATCHED|CONTRACTORS|2000|24000|3 (34)|00:00:01|
|*2|   INDEX RANGE SCAN                 |SALARY_IX  |2000|     |1   (0)|00:00:01|
---------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SALARY"=1000)

20 rows selected.
```

Based on the artificially generated statistics for the number of rows and block distribution, the optimizer considers an index range scan more cost-effective.

16 Managing Historical Optimizer Statistics

This chapter how to retain, report on, and restore non-current statistics.

16.1 Restoring Optimizer Statistics

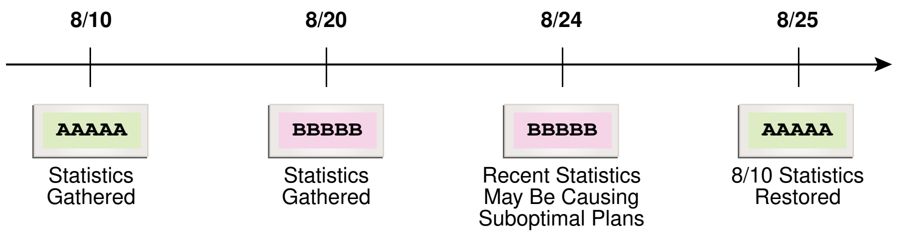
You can use DBMS_STATS to restore old versions of statistics that are stored in the data dictionary.

16.1.1 About Restore Operations for Optimizer Statistics

Whenever statistics in the data dictionary are modified, the database automatically saves old versions of statistics. If newly collected statistics lead to suboptimal execution plans, then you may want to revert to the previous statistics.

Restoring optimizer statistics can aid in troubleshooting suboptimal plans. The following graphic illustrates a timeline for restoring statistics. In the graphic, statistics collection occurs on August 10 and August 20. On August 24, the DBA determines that the current statistics may be causing the optimizer to generate suboptimal plans. On August 25, the administrator restores the statistics collected on August 10.

Figure 16-1 Restoring Optimizer Statistics



16.1.2 Guidelines for Restoring Optimizer Statistics

Restoring statistics is similar to importing and exporting statistics.

In general, restore statistics instead of exporting them in the following situations:

- You want to recover older versions of the statistics. For example, you want to restore the optimizer behavior to an earlier date.
- You want the database to manage the retention and purging of statistics histories.

Export statistics rather than restoring them in the following situations:

- You want to experiment with multiple sets of statistics and change the values back and forth.
- You want to move the statistics from one database to another database. For example, moving statistics from a production system to a test system.

- You want to preserve a known set of statistics for a longer period than the desired retention date for restoring statistics.

> **See Also:**
>
> *Oracle Database PL/SQL Packages and Types Reference* for an overview of the procedures for restoring and importing statistics

16.1.3 Restrictions for Restoring Optimizer Statistics

When restoring previous versions of statistics, various limitations apply.

Restrictions include the following:

- `DBMS_STATS.RESTORE_*_STATS` procedures cannot restore user-defined statistics.
- Old versions of statistics are not stored when the `ANALYZE` command has been used for collecting statistics.
- Dropping a table removes the workload data used by the automatic histogram feature and the statistics history used by `DBMS_STATS.RESTORE_*_STATS`. Without this data, these features do not work properly. Therefore, to remove all rows from a table and repopulate it, Oracle recommends using `TRUNCATE` instead of dropping and re-creating the table.

  > **Note:**
  >
  > If a table resides in the recycle bin, then flashing back the table also retrieves the statistics.

16.1.4 Restoring Optimizer Statistics Using DBMS_STATS

You can restore statistics using the `DBMS_STATS.RESTORE_*_STATS` procedures.

The procedures listed in the following table accept a timestamp as an argument and restore statistics as of the specified time (`as_of_timestamp`).

Table 16-1 DBMS_STATS Restore Procedures

Procedure	Description
`RESTORE_DICTIONARY_STATS`	Restores statistics of all dictionary tables (tables of `SYS`, `SYSTEM`, and `RDBMS` component schemas) as of a specified timestamp.
`RESTORE_FIXED_OBJECTS_STATS`	Restores statistics of all fixed tables as of a specified timestamp.
`RESTORE_SCHEMA_STATS`	Restores statistics of all tables of a schema as of a specified timestamp.
`RESTORE_SYSTEM_STATS`	Restores system statistics as of a specified timestamp.

Table 16-1 (Cont.) DBMS_STATS Restore Procedures

Procedure	Description
`RESTORE_TABLE_STATS`	Restores statistics of a table as of a specified timestamp. The procedure also restores statistics of associated indexes and columns. If the table statistics were locked at the specified timestamp, then the procedure locks the statistics.

Dictionary views display the time of statistics modifications. You can use the following views to determine the time stamp to be use for the restore operation:

- The `DBA_OPTSTAT_OPERATIONS` view contain history of statistics operations performed at schema and database level using `DBMS_STATS`.
- The `DBA_TAB_STATS_HISTORY` views contains a history of table statistics modifications.

Assumptions

This tutorial assumes the following:

- After the most recent statistics collection for the `oe.orders` table, the optimizer began choosing suboptimal plans for queries of this table.
- You want to restore the statistics from before the most recent statistics collection to see if the plans improve.

To restore optimizer statistics:

1. Start SQL*Plus and connect to the database with administrator privileges.
2. Query the statistics history for `oe.orders`.

   For example, run the following query:

   ```
   COL TABLE_NAME FORMAT a10
   SELECT TABLE_NAME,
          TO_CHAR(STATS_UPDATE_TIME,'YYYY-MM-DD:HH24:MI:SS') AS
   STATS_MOD_TIME
   FROM   DBA_TAB_STATS_HISTORY
   WHERE  TABLE_NAME='ORDERS'
   AND    OWNER='OE'
   ORDER BY STATS_UPDATE_TIME DESC;
   ```

   Sample output is as follows:

   ```
   TABLE_NAME STATS_MOD_TIME
   ---------- -------------------
   ORDERS     2012-08-20:11:36:38
   ORDERS     2012-08-10:11:06:20
   ```

3. Restore the optimizer statistics to the previous modification time.

For example, restore the `oe.orders` table statistics to August 10, 2012:

```
BEGIN
  DBMS_STATS.RESTORE_TABLE_STATS( 'OE','ORDERS',
               TO_TIMESTAMP('2012-08-10:11:06:20','YYYY-MM-
DD:HH24:MI:SS') );
END;
/
```

You can specify any date between 8/10 and 8/20 because `DBMS_STATS` restores statistics as of the specified time.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DBMS_STATS.RESTORE_TABLE_STATS` procedure

16.2 Managing Optimizer Statistics Retention

By default, the database retains optimizer statistics for 31 days, after which time the statistics are scheduled for purging.

You can use the `DBMS_STATS` package to determine the retention period, change the period, and manually purge old statistics.

16.2.1 Obtaining Optimizer Statistics History

You can use `DBMS_STATS` procedures to obtain historical information for optimizer statistics.

Historical information is useful when you want to determine how long the database retains optimizer statistics, and how far back these statistics can be restored. You can use the following procedure to obtain information about the optimizer statistics history:

- `GET_STATS_HISTORY_RETENTION`

  This function can retrieve the current statistics history retention value.

- `GET_STATS_HISTORY_AVAILABILITY`

  This function retrieves the oldest time stamp when statistics history is available. Users cannot restore statistics to a time stamp older than the oldest time stamp.

To obtain optimizer statistics history information:

1. Start SQL*Plus and connect to the database with the necessary privileges.
2. Execute the following PL/SQL program:

```
DECLARE
  v_stats_retn  NUMBER;
  v_stats_date  DATE;
BEGIN
```

```
  v_stats_retn := DBMS_STATS.GET_STATS_HISTORY_RETENTION;
  DBMS_OUTPUT.PUT_LINE('The retention setting is ' ||
    v_stats_retn || '.');
  v_stats_date := DBMS_STATS.GET_STATS_HISTORY_AVAILABILITY;
  DBMS_OUTPUT.PUT_LINE('Earliest restore date is ' ||
    v_stats_date || '.');
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.GET_STATS_HISTORY_RETENTION` procedure

16.2.2 Changing the Optimizer Statistics Retention Period

You can configure the retention period using the `DBMS_STATS.ALTER_STATS_HISTORY_RETENTION` procedure. The default is 31 days.

Prerequisites

To run this procedure, you must have either the `SYSDBA` privilege, or both the `ANALYZE ANY DICTIONARY` and `ANALYZE ANY` system privileges.

Assumptions

This tutorial assumes the following:

- The current retention period for optimizer statistics is 31 days.
- You run queries annually as part of an annual report. To keep the statistics history for more than 365 days so that you have access to last year's plan (in case a suboptimal plan occurs now), you set the retention period to 366 days.
- You want to create a PL/SQL procedure `set_opt_stats_retention` that you can use to change the optimizer statistics retention period.

To change the optimizer statistics retention period:

1. Start SQL*Plus and connect to the database with the necessary privileges.
2. Create a procedure that changes the retention period.

   For example, create the following procedure:

```
CREATE OR REPLACE PROCEDURE set_opt_stats_retention
  ( p_stats_retn   IN NUMBER )
IS
  v_stats_retn NUMBER;
BEGIN
  v_stats_retn := DBMS_STATS.GET_STATS_HISTORY_RETENTION;
  DBMS_OUTPUT.PUT_LINE('Old retention setting is ' ||
    v_stats_retn || '.');
  DBMS_STATS.ALTER_STATS_HISTORY_RETENTION(p_stats_retn);
  v_stats_retn := DBMS_STATS.GET_STATS_HISTORY_RETENTION;
  DBMS_OUTPUT.PUT_LINE('New retention setting is ' ||
    v_stats_retn || '.');
END;
/
```

3. Change the retention period to 366 days.

   For example, execute the procedure that you created in the previous step (sample output included):

   ```
   SQL> EXECUTE set_opt_stats_retention(366)

   The old retention setting is 31.
   The new retention setting is 366.

   PL/SQL procedure successfully completed.
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.ALTER_STATS_HISTORY_RETENTION` procedure

16.2.3 Purging Optimizer Statistics

Automatic purging is enabled when the `STATISTICS_LEVEL` initialization parameter is set to `TYPICAL` or `ALL`.

The database purges all history older than the older of (current time - the `ALTER_STATS_HISTORY_RETENTION` setting) and (time of the most recent statistics gathering - 1).

You can purge old statistics manually using the `PURGE_STATS` procedure. If you do not specify an argument, then this procedure uses the automatic purging policy. If you specify the `before_timestamp` parameter, then the database purges statistics saved before the specified timestamp.

Prerequisites

To run this procedure, you must have either the `SYSDBA` privilege, or both the `ANALYZE ANY DICTIONARY` and `ANALYZE ANY` system privileges.

Assumptions

This tutorial assumes that you want to purge statistics more than one week old.

To purge optimizer statistics:

1. In SQL*Plus, log in to the database with the necessary privileges.
2. Execute the `DBMS_STATS.PURGE_STATS` procedure.

For example, execute the procedure as follows:

```
EXEC DBMS_STATS.PURGE_STATS( SYSDATE-7 );
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS.PURGE_STATS` procedure

16.3 Reporting on Past Statistics Gathering Operations

You can use `DBMS_STATS` functions to report on a specific statistics gathering operation or on operations that occurred during a specified time.

Different operations from different PDBs may have the same operation ID. If a PDB ID is not provided, then the report may contain multiple operations.

Table 16-2 lists the functions.

Table 16-2 DBMS_STATS Reporting Functions

Function	Description
`REPORT_STATS_OPERATIONS`	Generates a report of all statistics operations that occurred between two points in time. You can narrow the scope of the report to include only automatic statistics gathering runs. You can also use `container_ids` to provide a set of container IDs so that the database reports only statistics operations from the specified PDBs.
`REPORT_SINGLE_STATS_OPERATION`	Generates a report of the specified operation. You can use `container_id` to specify a particular PDB.

Assumptions

This tutorial assumes that you want to generate HTML reports of the following:

- All statistics gathering operations within the last day
- The most recent statistics gathering operation

To report on all operations in the past day:

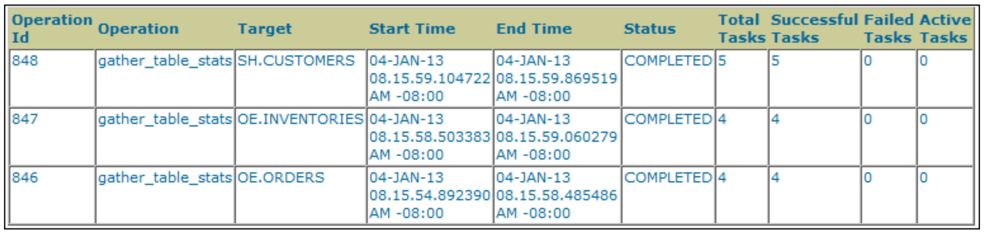
1. Start SQL*Plus and connect to the database with administrator privileges.
2. Run the `DBMS_STATS.REPORT_STATS_OPERATIONS` function.

   For example, run the following commands:

   ```
   SET LINES 200 PAGES 0
   SET LONG 100000
   COLUMN REPORT FORMAT A200

   VARIABLE my_report CLOB;
   ```

```
BEGIN
  :my_report := DBMS_STATS.REPORT_STATS_OPERATIONS (
     since         => SYSDATE-1
,    until         => SYSDATE
,    detail_level => 'TYPICAL'
,    format        => 'HTML'
);
END;
/
```

The following graphic shows a sample report:



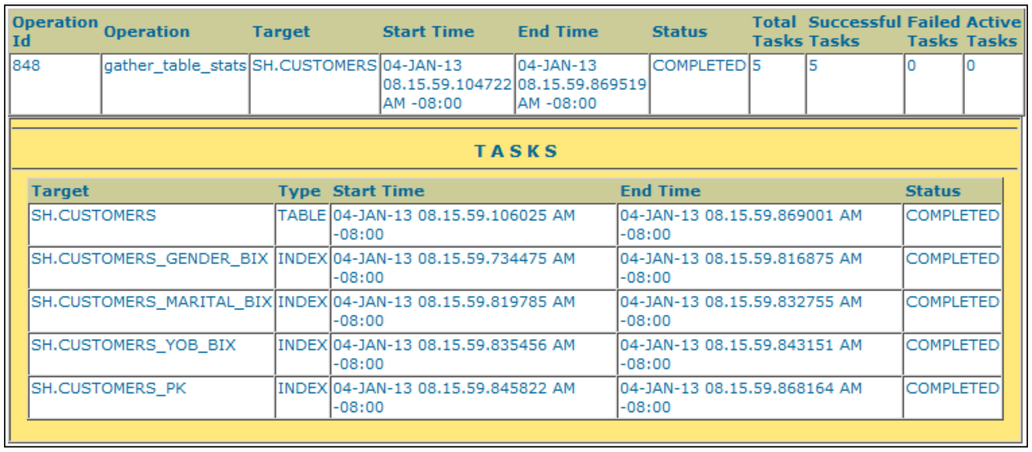
Operation Id	Operation	Target	Start Time	End Time	Status	Total Tasks	Successful Tasks	Failed Tasks	Active Tasks
848	gather_table_stats	SH.CUSTOMERS	04-JAN-13 08.15.59.104722 AM -08:00	04-JAN-13 08.15.59.869519 AM -08:00	COMPLETED	5	5	0	0
847	gather_table_stats	OE.INVENTORIES	04-JAN-13 08.15.58.503383 AM -08:00	04-JAN-13 08.15.59.060279 AM -08:00	COMPLETED	4	4	0	0
846	gather_table_stats	OE.ORDERS	04-JAN-13 08.15.54.892390 AM -08:00	04-JAN-13 08.15.58.485486 AM -08:00	COMPLETED	4	4	0	0

3. Run the `DBMS_STATS.REPORT_SINGLE_STATS_OPERATION` function for an individual operation.

   For example, run the following program to generate a report of operation `848`:

```
BEGIN
  :my_report :=DBMS_STATS.REPORT_SINGLE_STATS_OPERATION (
     OPID    => 848
,    FORMAT  => 'HTML'
);
END;
```

The following graphic shows a sample report:



Operation Id	Operation	Target	Start Time	End Time	Status	Total Tasks	Successful Tasks	Failed Tasks	Active Tasks
848	gather_table_stats	SH.CUSTOMERS	04-JAN-13 08.15.59.104722 AM -08:00	04-JAN-13 08.15.59.869519 AM -08:00	COMPLETED	5	5	0	0

TASKS

Target	Type	Start Time	End Time	Status
SH.CUSTOMERS	TABLE	04-JAN-13 08.15.59.106025 AM -08:00	04-JAN-13 08.15.59.869001 AM -08:00	COMPLETED
SH.CUSTOMERS_GENDER_BIX	INDEX	04-JAN-13 08.15.59.734475 AM -08:00	04-JAN-13 08.15.59.816875 AM -08:00	COMPLETED
SH.CUSTOMERS_MARITAL_BIX	INDEX	04-JAN-13 08.15.59.819785 AM -08:00	04-JAN-13 08.15.59.832755 AM -08:00	COMPLETED
SH.CUSTOMERS_YOB_BIX	INDEX	04-JAN-13 08.15.59.835456 AM -08:00	04-JAN-13 08.15.59.843151 AM -08:00	COMPLETED
SH.CUSTOMERS_PK	INDEX	04-JAN-13 08.15.59.845822 AM -08:00	04-JAN-13 08.15.59.868164 AM -08:00	COMPLETED

See Also:

- "Graphical Interface for Optimizer Statistics Management" to learn about the Cloud Control GUI for statistics management
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS`

17

Importing and Exporting Optimizer Statistics

You can export and import optimizer statistics from the data dictionary to user-defined statistics tables. You can also copy statistics from one database to another database.

17.1 About Transporting Optimizer Statistics

When you transport optimizer statistics between databases, you must use `DBMS_STATS` to copy the statistics to and from a staging table, and tools to make the table contents accessible to the destination database.

17.1.1 Purpose of Transporting Optimizer Statistics

Importing and exporting are especially useful for testing an application using production statistics.

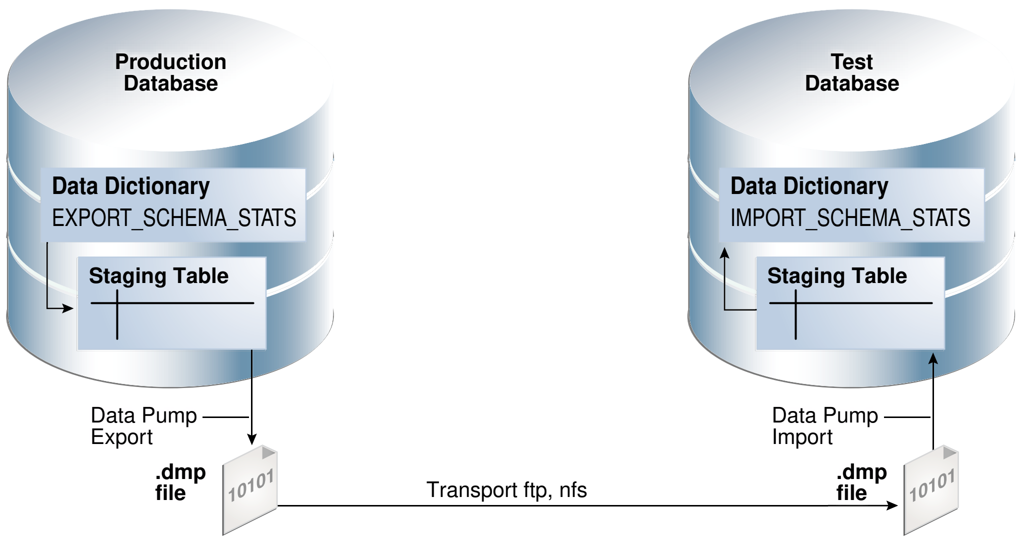
Developers often want to tune query plans in a realistic environment before deploying applications. A typical scenario would be to use `DBMS_STATS.EXPORT_SCHEMA_STATS` to export schema statistics from a production database to a test database.

17.1.2 How Transporting Optimizer Statistics Works

The typical transport operation uses a combination of `DBMS_STATS` and file transfer utilities.

The following figure illustrates the process using Oracle Data Pump and `ftp`.

Figure 17-1 Transporting Optimizer Statistics



The basic steps are as follows:

1. In the production database, copy the statistics from the data dictionary to a staging table using `DBMS_STATS.EXPORT_SCHEMA_STATS`.
2. Export the statistics from the staging table to a `.dmp` file using Oracle Data Pump.
3. Transfer the `.dmp` file from the production host to the test host using a transfer tool such as `ftp`.
4. In the test database, import the statistics from the `.dmp` file to a staging table using Oracle Data Pump.
5. Copy the statistics from the staging table to the data dictionary using `DBMS_STATS.IMPORT_SCHEMA_STATS`.

17.1.3 User Interface for Importing and Exporting Optimizer Statistics

`DBMS_STATS` provides the interface for importing and exporting statistics for schemas and tables.

The following subprograms in `DBMS_STATS` enable you to export schemas and different types of tables.

Table 17-1 Subprograms for Exporting Schema and Table Statistics

Subprogram	Description
`EXPORT_DATABASE_STATS`	This procedure exports statistics for all objects in the database and stores them in the user statistics tables identified by `statown.stattab`.
`EXPORT_DICTIONARY_STATS`	This procedure exports statistics for all data dictionary schemas (`SYS`, `SYSTEM`, and RDBMS component schemas) and stores them in the user statistics table identified by `stattab`.
`EXPORT_FIXED_OBJECT_STATS`	This procedure exports statistics for fixed tables and stores them in the user statistics table identified by `stattab`.
`EXPORT_SCHEMA_STATS`	This procedure exports statistics for all objects in the schema identified by `ownname` and stores them in the user statistics tables identified by `stattab`. By default, the `stat_category` parameter includes statistics collected during real-time statistics. The `REALTIME_STATS` value specifies only online statistics.
`EXPORT_TABLE_STATS`	This procedure exports statistics for a specified table (including associated index statistics) and stores them in the user statistics table identified by `stattab`. By default, the `stat_category` parameter includes statistics collected during real-time statistics. The `REALTIME_STATS` value specifies only online statistics.

The following subprograms in `DBMS_STATS` enable you to import schemas and different types of tables.

Table 17-2 Subprograms for Importing Optimizer Statistics

Subprogram	Description
`IMPORT_DATABASE_STATS`	This procedure imports statistics for all objects in the database from the user statistics table and stores them in the data dictionary.
`IMPORT_DICTIONARY_STATS`	This procedure imports statistics for all data dictionary schemas (`SYS`, `SYSTEM`, and RDBMS component schemas) from the user statistics table and stores them in the dictionary.
`IMPORT_FIXED_OBJECT_STATS`	This procedure imports statistics for fixed tables from the user statistics table and stores them in the data dictionary.
`IMPORT_SCHEMA_STATS`	This procedure imports statistics for all objects in the schema identified by `ownname` from the user statistics table and stores them in the data dictionary. By default, the `stat_category` parameter includes statistics collected during real-time statistics. The `REALTIME_STATS` value specifies only online statistics.
`IMPORT_TABLE_STATS`	This procedure import statistics for a specified table from the user statistics table identified by `stattab` and stores them in the data dictionary. By default, the `stat_category` parameter includes statistics collected during real-time statistics. The `REALTIME_STATS` value specifies only online statistics.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about `DBMS_STATS`

17.2 Transporting Optimizer Statistics to a Test Database: Tutorial

You can transport schema statistics from a production database to a test database using Oracle Data Pump.

Prerequisites and Restrictions

When preparing to export optimizer statistics, note the following:

- Before exporting statistics, you must create a table to hold the statistics. The procedure `DBMS_STATS.CREATE_STAT_TABLE` creates the statistics table.
- The optimizer does not use statistics stored in a user-owned table. The only statistics used by the optimizer are the statistics stored in the data dictionary. To make the

optimizer use statistics in user-defined tables, import these statistics into the data dictionary using the `DBMS_STATS` import procedure.

- The Data Pump Export and Import utilities export and import optimizer statistics from the database along with the table. When a column has system-generated names, Original Export (`exp`) does not export statistics with the data, but this restriction does not apply to Data Pump Export.

> **Note:**
>
> Exporting and importing statistics using `DBMS_STATS` is a distinct operation from using Data Pump Export and Import.

Assumptions

This tutorial assumes the following:

- You want to generate representative `sh` schema statistics on a production database and use `DBMS_STATS` to import them into a test database.
- Administrative user `dba1` exists on both production and test databases.
- You intend to create table `opt_stats` to store the schema statistics.
- You intend to use Oracle Data Pump to export and import table `opt_stats`.

To generate schema statistics and import them into a separate database:

1. On the production host, start SQL*Plus and connect to the production database as administrator `dba1`.
2. Create a table to hold the production statistics.

   For example, execute the following PL/SQL program to create user statistics table `opt_stats`:

   ```
   BEGIN
     DBMS_STATS.CREATE_STAT_TABLE (
       ownname => 'dba1'
   ,   stattab => 'opt_stats'
   );
   END;
   /
   ```

3. Gather schema statistics.

   For example, manually gather schema statistics as follows:

   ```
   -- generate representative workload
   EXEC DBMS_STATS.GATHER_SCHEMA_STATS('SH');
   ```

4. Use `DBMS_STATS` to export the statistics.

For example, retrieve schema statistics and store them in the `opt_stats` table created previously:

```
BEGIN
  DBMS_STATS.EXPORT_SCHEMA_STATS (
    ownname => 'dba1'
,   stattab => 'opt_stats'
);
END;
/
```

5. Use Oracle Data Pump to export the contents of the statistics table.

   For example, run the `expdp` command at the operating schema prompt:

   ```
   expdp dba1 DIRECTORY=dpump_dir1 DUMPFILE=stat.dmp TABLES=opt_stats
   ```

6. Transfer the dump file to the test database host.
7. Log in to the test host, and then use Oracle Data Pump to import the contents of the statistics table.

   For example, run the `impdp` command at the operating schema prompt:

   ```
   impdp dba1 DIRECTORY=dpump_dir1 DUMPFILE=stat.dmp TABLES=opt_stats
   ```

8. On the test host, start SQL*Plus and connect to the test database as administrator `dba1`.
9. Use `DBMS_STATS` to import statistics from the user statistics table and store them in the data dictionary.

   The following PL/SQL program imports schema statistics from table `opt_stats` into the data dictionary:

   ```
   BEGIN
     DBMS_STATS.IMPORT_SCHEMA_STATS(
       ownname => 'dba1'
   ,   stattab => 'opt_stats'
   );
   END;
   /
   ```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS.CREATE_STAT_TABLE` function
- *Oracle Database PL/SQL Packages and Types Reference* for an overview of the statistics transfer functions
- *Oracle Database Utilities* to learn about Oracle Data Pump

18
Analyzing Statistics Using Optimizer Statistics Advisor

Optimizer Statistics Advisor analyzes how optimizer statistics are gathered, and then makes recommendations.

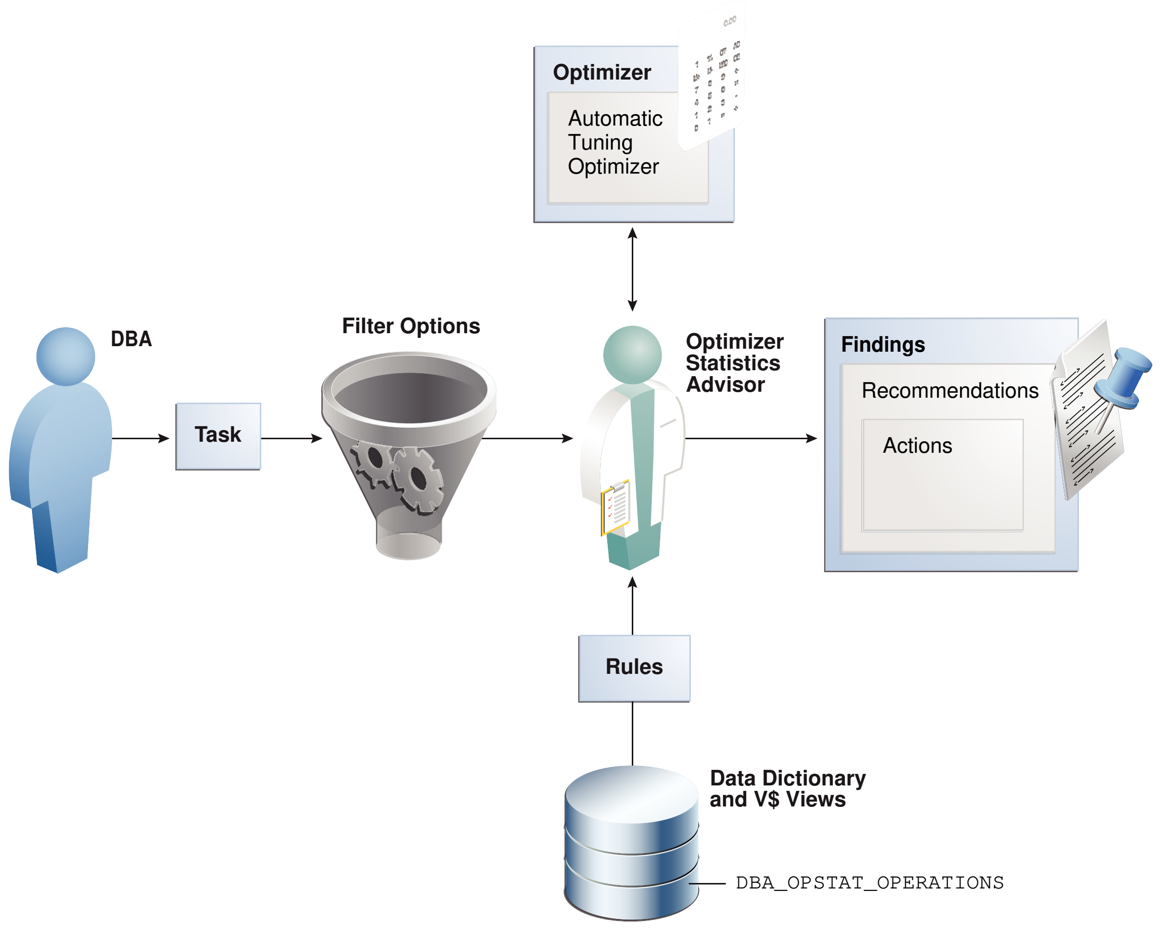
18.1 About Optimizer Statistics Advisor

Optimizer Statistics Advisor is built-in diagnostic software that analyzes the quality of statistics and statistics-related tasks.

The advisor task runs automatically in the maintenance window, but you can also run it on demand. You can then view the advisor report. If the advisor makes recommendations, then in some cases you can run system-generated scripts to implement them.

The following figure provides a conceptual overview of Optimizer Statistics Advisor.

Figure 18-1 Optimizer Statistics Advisor



18.1.1 Purpose of Optimizer Statistics Advisor

Optimizer Statistics Advisor inspects how optimizer statistics are gathered.

The advisor automatically diagnoses problems in the existing practices for gathering statistics. The advisor does *not* gather a new or alternative set of optimizer statistics. The output of the advisor is a report of findings and recommendations, which helps you follow best practices for gathering statistics.

Optimizer statistics play a significant part in determining the execution plan for queries. Therefore, it is critical for the optimizer to gather and maintain accurate and up-to-date statistics. The optimizer provides the `DBMS_STATS` package, which evolves from release to release, for this purpose. Typically, users develop their own strategies for gathering statistics based on specific workloads, and then use homegrown scripts to implement these strategies.

18.1.1.1 Problems with a Traditional Script-Based Approach

The advantage of the scripted approach is that the scripts are typically tested and reviewed. However, the owner of suboptimal legacy scripts may not change them for fear of causing plan changes.

The traditional approach has the following problems:

- Legacy scripts may not keep pace with new best practices, which can change from release to release.

  Frequently, successive releases add enhancements to histograms, sampling, workload monitoring, concurrency, and other optimizer-related features. For example, starting in Oracle Database 12c, Oracle recommends setting `AUTO_SAMPLE_SIZE` instead of a percentage. However, legacy scripts typically specify a sampling percentage, which may lead to suboptimal execution plans.
- Resources are wasted on unnecessary statistics gathering.

  A script may gather statistics multiple times each day on the same table.
- Automatic statistics gathering jobs do not guarantee accurate and up-to-date statistics.

  For example, sometimes the automatic statistics gathering job is not running because an initialization parameter combination disables it, or the job is terminated. Moreover, sometimes the automatic job maintenance window is insufficient because of resource constraints, or because too many objects require statistics collection. Jobs that stop running before gathering all statistics cause either no statistics or stale statistics for some objects, which can in turn cause suboptimal plans.
- Statistics can sometimes be missing, stale, or incorrect.

  For example, statistics may be inconsistent between a table and its index, or between tables with a primary key-foreign key relationship. Alternatively, a statistics gathering job may have been disabled by accident, or you may be unaware that a script has failed.
- Lack of knowledge of the problem can be time-consuming and resource-intensive.

  For example, a service request might seek a resolution to a problem, unaware that the problem is caused by suboptimal statistics. The diagnosis might require a great deal of time emailing scripts of the problematic queries, enabling traces, and investigating traces.

- Recommended fixes may not be feasible.

  Performance engineers may recommend changing the application code that maintains statistics. In some organizations, this requirement may be difficult or impossible to satisfy.

18.1.1.2 Advantages of Optimizer Statistics Advisor

An advisor-based approach offers better scalability and maintainability than the traditional approach.

If best practices change in a new release, then Optimizer Statistics Advisor encodes these practices in its rules. In this way, the advisor always provides the most up-to-date recommendations.

The advisor analyzes how you are currently gathering statistics (using manual scripts, explicitly setting parameters, and so on), the effectiveness of existing statistics gathering jobs, and the quality of the gathered statistics. Optimizer Statistics Advisor does *not* gather a new or alternative set of optimizer statistics, and so does not affect the workload. Rather, Optimizer Statistics Advisor analyzes information stored in the data dictionary, and then stores the findings and recommendations in the database.

Optimizer Statistics Advisor provides the following advantages over the traditional approach:

- Provides easy-to-understand reports

  The advisor applies rules to generate findings, recommendations, and actions.

- Supplies scripts to implement necessary fixes *without* requiring changes to application code

  When you implement a recommended action, benefit accrues to every execution of the improved statements. For example, if you set a global preference so that the sample size is `AUTO_SAMPLE_SIZE` rather than a suboptimal percentage, then every plan based on the improved statistics can benefit from this change.

- Runs a predefined task named `AUTO_STATS_ADVISOR_TASK` once every day in the maintenance window

  For the automated job to run, the `STATISTICS_LEVEL` initialization parameter must be set to `TYPICAL` or `ALL`.

- Supplies an API in the `DBMS_STATS` package that enables you to create and run tasks manually, store findings and recommendations in data dictionary views, generate reports for the tasks, and implement corrections when necessary

- Integrates with existing tools

  The advisor integrates with SQL Tuning Advisor and AWR, which summarize the Optimizer Statistics Advisor results.

18.1.2 Optimizer Statistics Advisor Concepts

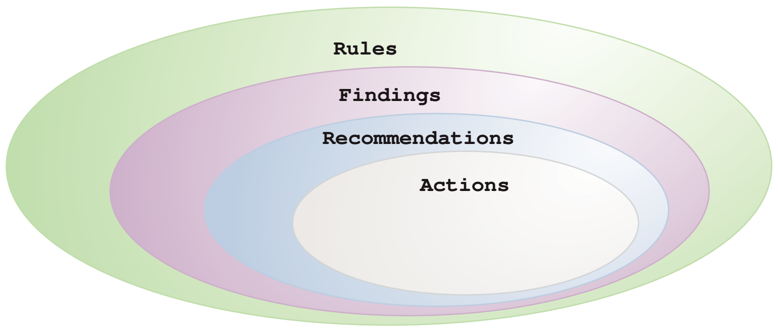
Optimizer Statistics Advisor uses the same advisor framework as Automatic Database Diagnostic Monitor (ADDM), SQL Performance Analyzer, and other advisors.

18.1.2.1 Components of Optimizer Statistics Advisor

The Optimizer Statistics Optimizer framework stores its metadata in data dictionary and dynamic performance views.

The following Venn diagram shows the relationships among rules, findings, recommendations, and actions for Optimizer Statistics Advisor. For example, all findings are derived from rules, but not all rules generate findings.

Figure 18-2 Optimizer Statistics Advisor Components



18.1.2.1.1 Rules for Optimizer Statistics Advisor

An **Optimizer Statistics Advisor rule** is an Oracle-supplied standard by which Optimizer Statistics Advisor performs its checks.

The rules embody Oracle best practices based on the current feature set. If the best practices change from release to release, then the Optimizer Statistics Advisor rules also change.

The advisor organizes rules into the following classes:

- System

  This class checks the preferences for statistics collection, status of the automated statistics gathering job, use of SQL plan directives, and so on. Rules in this class have the value `SYSTEM` in `V$STATS_ADVISOR_RULES.RULE_TYPE`.
- Operation

  This class checks whether statistics collection uses the defaults, test statistics are created using the `SET_*_STATS` procedures, and so on. Rules in this class have the value `OPERATION` in `V$STATS_ADVISOR_RULES.RULE_TYPE`.
- Object

  This class checks for the quality of the statistics, staleness of statistics, unnecessary collection of statistics, and so on. Rules in this class have the value `OBJECT` in `V$STATS_ADVISOR_RULES.RULE_TYPE`.

The rules check for the following problems:

- How to gather statistics

For example, one rule might specify the recommended setting for an initialization parameter. Another rule might specify that statistics should be gathered at the schema level.

- When to gather statistics

  For example, the advisor may recommend that the maintenance window for the automatic statistics gathering job should be enabled, or that the window should be extended.

- How to improve the efficiency of statistics gathering

  For example, a rule might specify that default parameters should be used in `DBMS_STATS`, or that statistics should not be set manually.

In `V$STATS_ADVISOR_RULES`, each rule has a unique string ID that is usable in the `DBMS_STATS` procedures and reports. You can use a rule filter to specify rules that Optimizer Statistics Advisor should check. However, you cannot write new rules.

Example 18-1 Listing Rules in V$STATS_ADVISOR_RULES

The following query, with sample output, lists a subset of the rules in `V$STATS_ADVISOR_RULES`. The rules may change from release to release.

```
SET LINESIZE 208
SET PAGESIZE 100
COL ID FORMAT 99
COL NAME FORMAT a33
COL DESCRIPTION FORMAT a62

SELECT RULE_ID AS ID, NAME, RULE_TYPE, DESCRIPTION
FROM   V$STATS_ADVISOR_RULES
WHERE  RULE_ID BETWEEN 1 AND 12
ORDER BY RULE_ID;

ID NAME                              RULE_TYPE DESCRIPTION
-- -------------------------------- --------- ------------------------------------------
 1 UseAutoJob                        SYSTEM    Use Auto Job for Statistics Collection
 2 CompleteAutoJob                   SYSTEM    Auto Statistics Gather Job should complete
                                               successfully
 3 MaintainStatsHistory              SYSTEM    Maintain Statistics History
 4 UseConcurrent                     SYSTEM    Use Concurrent preference for Statistics
                                               Collection
 5 UseDefaultPreference              SYSTEM    Use Default Preference for Stats Collection
 6 TurnOnSQLPlanDirective            SYSTEM    SQL Plan Directives should not be disabled
 7 AvoidSetProcedures                OPERATION Avoid Set Statistics Procedures
 8 UseDefaultParams                  OPERATION Use Default Parameters in Statistics
                                               Collection Proc.
 9 UseGatherSchemaStats              OPERATION Use gather_schema_stats procedure
10 AvoidInefficientStatsOprSeq       OPERATION Avoid inefficient statistics operation
                                               sequences
11 AvoidUnnecessaryStatsCollection OBJECT    Avoid unnecessary statistics collection
12 AvoidStaleStats                   OBJECT    Avoid objects with stale or no statistics

12 rows selected.
```

See Also:

Oracle Database Reference to learn more about `V$STATS_ADVISOR_RULES`

18.1.2.1.2 Findings for Optimizer Statistics Advisor

A finding results when Optimizer Statistics Advisor examines the evidence stored in the database and concludes that the rules were not followed.

To generate findings, Optimizer Statistics Advisor executes a task, which is invoked either automatically or manually. This task analyzes the statistics history stored in the data dictionary, the statistics operation log, and the current statistics footprint that exists in `SYSAUX`. For example, the advisor queries `DBA_TAB_STATISTICS` and `DBA_IND_STATISTICS` to determine whether statistics are stale, or whether a discrepancy exists between the numbers of rows.

Typically, Optimizer Statistics Advisor generates a finding when a specific rule is not followed or is violated, although some findings—such as object staleness—provide only information. For example, a finding may show that `DBMS_STATS.GATHER_TABLE_STATS` has used `ESTIMATE_PERCENT=>0.01`, which violates the `ESTIMATE_PERCENT=>AUTO_SAMPLE_SIZE` rule.

A finding corresponds to exactly one rule. However, a rule can generate many findings.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS`
- *Oracle Database Reference* to learn more about `ALL_TAB_STATISTICS`

18.1.2.1.3 Recommendations for Optimizer Statistics Advisor

Based on each finding, Optimizer Statistics Advisor makes recommendations on how to achieve better statistics.

For example, the advisor might discover a violation to the rule of not using sampling when gathering statistics, and recommend specifying `AUTO_SAMPLE_SIZE` instead. The advisor stores the recommendations in `DBA_ADVISOR_RECOMMENDATIONS`.

Multiple recommendations may exist for a single finding. In this case, you must investigate to determine which recommendation to follow. Each recommendation includes one or more rationales that explain why Optimizer Statistics Advisor makes its recommendation. In some cases, findings may not generate recommendations.

See Also:

- "Guideline for Setting the Sample Size" to learn the guideline for the sample size
- *Oracle Database Reference* to learn about `DBA_ADVISOR_RECOMMENDATIONS`

18.1.2.1.4 Actions for Optimizer Statistics Advisor

An Optimizer Statistics Advisor action is a SQL or PL/SQL script that implements recommendations. When feasible, recommendations have corresponding actions. The advisor stores actions in `DBA_ADVISOR_ACTIONS`.

For example, Optimizer Statistics Advisor executes a task that performs the following steps:

1. Checks rules

   The advisor checks conformity to the rule that stale statistics should be avoided.
2. Generates finding

   The advisor discovers that a number of objects have no statistics.
3. Generates recommendation

   The advisor recommends gathering statistics on the objects with no statistics.
4. Generates action

   The advisor generates a PL/SQL script that executes `DBMS_STATS.GATHER_DATABASE_STATS`, supplying a list of objects that need to have statistics gathered.

See Also:

- "Statistics Preference Overrides" to learn how to override statistics gathering preferences
- "Guideline for Setting the Sample Size" to learn more about `AUTO_SAMPLE_SIZE`
- *Oracle Database Reference* to learn about `DBA_ADVISOR_ACTIONS`

18.1.2.2 Operational Modes for Optimizer Statistics Advisor

Optimizer Statistics Advisor supports both an automated and manual mode.

- Automated

  The predefined task `AUTO_STATS_ADVISOR_TASK` runs automatically in the maintenance window once per day. The task runs as part of the automatic optimizer statistics collection client. The automated task generates findings and recommendations, but does not implement actions automatically.

  As for any other task, you can configure the automated task, and generate reports. If the report recommends actions, then you can implement the actions manually.

- Manual

  You can create your own task using the DBMS_STATS.CREATE_ADVISOR_TASK function, and then run it at any time using the EXECUTE_ADVISOR_TASK procedure.

  Unlike the automated task, the manual task can implement actions automatically. Alternatively, you can configure the task to generate a PL/SQL script, which you can then run manually.

See Also:

- "Configuring Automatic Optimizer Statistics Collection"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about DBMS_STATS.CREATE_ADVISOR_TASK

18.1.3 Command-Line Interface to Optimizer Statistics Advisor

Perform Optimizer Statistics Advisor tasks using the DBMS_STATS PL/SQL package.

Table 18-1 DBMS_STATS APIs for Task Creation and Deletion

PL/SQL Procedure or Function	Description
CREATE_ADVISOR_TASK	Creates an advisor task for Optimizer Statistics Advisor. If the task name is already specified, then the advisor uses the specified task name; otherwise, the advisor automatically generates a new task name.
DROP_ADVISOR_TASK	Deletes an Optimizer Statistics Advisor task and all its result data.

Table 18-2 DBMS_STATS APIs for Task Execution

PL/SQL Procedure or Function	Description
EXECUTE_ADVISOR_TASK	Executes a previously created Optimizer Statistics Advisor task.
INTERRUPT_ADVISOR_TASK	Interrupts a currently executing Optimizer Statistics Advisor task. The task ends its operations as it would in a normal exit, enabling you to access intermediate results. You can resume the task later.
CANCEL_ADVISOR_TASK	Cancels an Optimizer Statistics Advisor task execution, and removes all intermediate results of the current execution.
RESET_ADVISOR_TASK	Resets an Optimizer Statistics Advisor task execution to its initial state. Call this procedure on a task that is not currently executing.
RESUME_ADVISOR_TASK	Resumes the Optimizer Statistics Advisor task execution that was most recently interrupted.

Table 18-3 DBMS_STATS APIs for Advisor Reports

PL/SQL Procedure or Function	Description
`REPORT_STATS_ADVISOR_TASK`	Reports the results of an Optimizer Statistics Advisor task.
`GET_ADVISOR_RECS`	Generates a recommendation report on the given item.

Table 18-4 DBMS_STATS APIs for Task and Filter Configuration

PL/SQL Procedure or Function	Description
`CONFIGURE_ADVISOR_TASK`	Configures the Optimizer Statistics Advisor lists for the execution, reporting, script generation, and implementation of an advisor task.
`GET_ADVISOR_OPR_FILTER`	Creates an operation filter for a statistics operation.
`CONFIGURE_ADVISOR_RULE_FILTER`	Configures the rule filter for an Optimizer Statistics Advisor task.
`CONFIGURE_ADVISOR_OPR_FILTER`	Configures the operation filter for an Optimizer Statistics Advisor task.
`CONFIGURE_ADVISOR_OBJ_FILTER`	Configures the object filter for an Optimizer Statistics Advisor task.
`SET_ADVISOR_TASK_PARAMETER`	Updates the value of an Optimizer Statistics Advisor task parameter. Valid parameters are `TIME_LIMIT` and `OP_START_TIME`.

Table 18-5 DBMS_STATS APIs for Implementation of Recommended Actions

PL/SQL Procedure or Function	Description
`SCRIPT_ADVISOR_TASK`	Gets the script that implements the recommended actions for the problems found by the advisor. You can check this script, and then choose which actions to execute.
`IMPLEMENT_ADVISOR_TASK`	Implements the actions recommended by the advisor based on results from a specified Optimizer Statistics Advisor execution.

See Also:

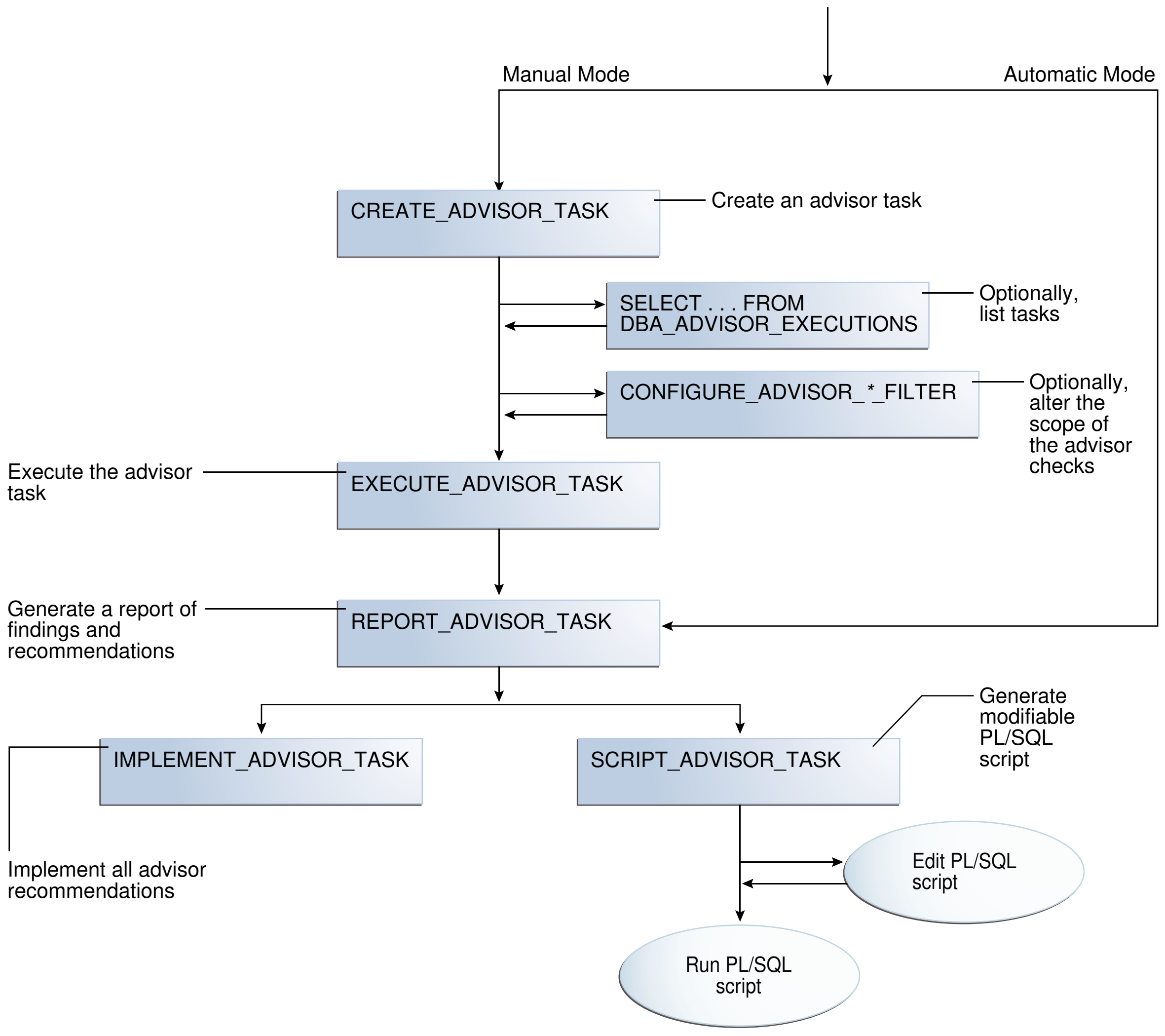
Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS` package

18.2 Basic Tasks for Optimizer Statistics Advisor

This section explains the basic workflow for using Optimizer Statistics Advisor. All procedures and functions are in the `DBMS_STATS` package.

The following figure shows the automatic and manual paths in the workflow. If `AUTO_STATS_ADVISOR_TASK` runs automatically in the maintenance window, then your workflow begins by querying the report. In the manual workflow, you must use PL/SQL to create and execute the tasks.

Figure 18-3 Basic Tasks for Optimizer Statistics Advisor


Manual Mode
Automatic Mode
CREATE_ADVISOR_TASK
Create an advisor task
SELECT . . . FROM DBA_ADVISOR_EXECUTIONS
Optionally, list tasks
CONFIGURE_ADVISOR_*_FILTER
Optionally, alter the scope of the advisor checks
Execute the advisor task
EXECUTE_ADVISOR_TASK
Generate a report of findings and recommendations
REPORT_ADVISOR_TASK
Generate modifiable PL/SQL script
IMPLEMENT_ADVISOR_TASK
SCRIPT_ADVISOR_TASK
Implement all advisor recommendations
Edit PL/SQL script
Run PL/SQL script

Typically, you perform Optimizer Statistics Advisor steps in the sequence shown in the following table.

Table 18-6 Optimizer Statistics Advisor Workflow

Step	Description	To Learn More
1	Create an Optimizer Advisor task using `DBMS_STATS.CREATE_ADVISOR_TASK` (manual workflow only).	"Creating an Optimizer Statistics Advisor Task"
2	Optionally, list executions of advisor tasks by querying `DBA_ADVISOR_EXECUTIONS`.	"Listing Optimizer Statistics Advisor Tasks"

Table 18-6 (Cont.) Optimizer Statistics Advisor Workflow

Step	Description	To Learn More
3	Optionally, configure a filter for the task using the `DBMS_STATS.CONFIGURE_ADVISOR_*_FILTER` procedures.	"Creating Filters for an Optimizer Advisor Task"
4	Execute the advisor task using `DBMS_STATS.EXECUTE_ADVISOR_TASK` (manual workflow only).	"Executing an Optimizer Statistics Advisor Task"
5	Generate an advisor report.	"Generating a Report for an Optimizer Statistics Advisor Task"
6	Implement the recommendations in either of following ways: • Implement all recommendations automatically using `DBMS_STATS.IMPLEMENT_ADVISOR_TASK`. • Generate a PL/SQL script that implements recommendations using `DBMS_STATS.SCRIPT_ADVISOR_TASK`, edit this script, and then run it manually.	"Implementing Actions Recommended by Optimizer Statistics Advisor" and "Generating a Script Using Optimizer Statistics Advisor"

Example 18-2 Basic Script for Optimizer Statistics Advisor in Manual Workflow

This script illustrates a basic Optimizer Statistics Advisor session. It creates a task, executes it, generates a report, and then implements the recommendations.

```
DECLARE
  v_tname   VARCHAR2(128) := 'my_task';
  v_ename   VARCHAR2(128) := NULL;
  v_report  CLOB := null;
  v_script  CLOB := null;
  v_implementation_result CLOB;
BEGIN
  -- create a task
  v_tname := DBMS_STATS.CREATE_ADVISOR_TASK(v_tname);

  -- execute the task
  v_ename := DBMS_STATS.EXECUTE_ADVISOR_TASK(v_tname);

  -- view the task report
  v_report := DBMS_STATS.REPORT_ADVISOR_TASK(v_tname);
  DBMS_OUTPUT.PUT_LINE(v_report);

  -- implement all recommendations
  v_implementation_result := DBMS_STATS.IMPLEMENT_ADVISOR_TASK(v_tname);
END;
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_STATS` package

18.2.1 Creating an Optimizer Statistics Advisor Task

The `DBMS_STATS.CREATE_ADVISOR_TASK` function creates a task for Optimizer Statistics Advisor. If you do not specify a task name, then Optimizer Statistics Advisor generates one automatically.

Prerequisites

To execute this subprogram, you must have the `ADVISOR` privilege.

Note:

This subprogram executes using invoker's rights.

To create an Optimizer Statistics Advisor task:

1. In SQL*Plus, log in to the database as a user with the necessary privileges.
2. Execute the `DBMS_STATS.CREATE_ADVISOR_TASK` function in the following form, where *`tname`* is the name of the task and *`ret`* is the variable that contains the returned output:

   ```
   EXECUTE ret := DBMS_STATS.CREATE_ADVISOR_TASK('tname');
   ```

   For example, to create the task `opt_adv_task1`, use the following code:

   ```
   DECLARE
     v_tname VARCHAR2(32767);
     v_ret   VARCHAR2(32767);
   BEGIN
     v_tname := 'opt_adv_task1';
     v_ret := DBMS_STATS.CREATE_ADVISOR_TASK(v_tname);
   END;
   /
   ```

3. Optionally, query `USER_ADVISOR_TASKS`:

   ```
   SELECT TASK_NAME, ADVISOR_NAME, CREATED, STATUS FROM
   USER_ADVISOR_TASKS;
   ```

Sample output appears below:

```
TASK_NAME       ADVISOR_NAME         CREATED   STATUS
--------------- -------------------- --------- -----------
OPT_ADV_TASK1   Statistics Advisor   05-SEP-16 INITIAL
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `CREATE_ADVISOR_TASK`

18.2.2 Listing Optimizer Statistics Advisor Tasks

The `DBA_ADVISOR_EXECUTIONS` view lists executions of Optimizer Statistics Advisor tasks.

To list Optimizer Statistics Advisor tasks:

1. In SQL*Plus, log in to the database as a user with administrator privileges.
2. Query `DBA_ADVISOR_EXECUTIONS` as follows:

```
COL EXECUTION_NAME FORMAT a14

SELECT EXECUTION_NAME, EXECUTION_END, STATUS
FROM   DBA_ADVISOR_EXECUTIONS
WHERE  TASK_NAME = 'AUTO_STATS_ADVISOR_TASK'
ORDER BY 2;
```

The following sample output shows 8 executions:

```
EXECUTION_NAME EXECUTION STATUS
-------------- --------- -----------
EXEC_1         27-AUG-16 COMPLETED
EXEC_17        28-AUG-16 COMPLETED
EXEC_42        29-AUG-16 COMPLETED
EXEC_67        30-AUG-16 COMPLETED
EXEC_92        01-SEP-16 COMPLETED
EXEC_117       02-SEP-16 COMPLETED
EXEC_142       03-SEP-16 COMPLETED
EXEC_167       04-SEP-16 COMPLETED

8 rows selected.
```

See Also:

Oracle Database Reference to learn more about `DBA_ADVISOR_EXECUTIONS`

18.2.3 Creating Filters for an Optimizer Advisor Task

Filters enable you to include or exclude objects, rules, and operations from Optimizer Statistics Advisor tasks.

18.2.3.1 About Filters for Optimizer Statistics Advisor

A filter is the use of `DBMS_STATS` to restrict an Optimizer Statistics Advisor task to a user-specified set of rules, schemas, or operations.

Filters are useful for including or excluding a specific set of results. For example, you can configure an advisor task to include only recommendations for the `sh` schema. Also, you can exclude all violations of the rule for stale statistics. The primary advantage of filters is the ability to ignore recommendations that you are not interested in, and reduce the overhead of the advisor task.

The simplest way to create filters is to use the following `DBMS_STATS` procedures either individually or in combination:

- `CONFIGURE_ADVISOR_OBJ_FILTER`

  Use this procedure to include or exclude the specified database schemas or objects. The object filter takes in an owner name and an object name, with wildcards (`%`) supported.
- `CONFIGURE_ADVISOR_RULE_FILTER`

  Use this procedure to include or exclude the specified rules. Obtain the names of rules by querying `V$STATS_ADVISOR_RULES`.
- `CONFIGURE_ADVISOR_OPR_FILTER`

  Use this procedure to include or exclude the specified `DBMS_STATS` operations. Obtain the IDs and names for operations by querying `DBA_OPTSTAT_OPERATIONS`.

For the preceding functions, you can specify the type of operation to which the filter applies: `EXECUTE`, `REPORT`, `SCRIPT`, and `IMPLEMENT`. You can also combine types, as in `EXECUTE + REPORT`. Null indicates that the filter applies to all types of advisor operations.

See Also:

- *Oracle Database Reference* to learn more about `V$STATS_ADVISOR_RULES`
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_STATS`

18.2.3.2 Creating an Object Filter for an Optimizer Advisor Task

The `DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER` function creates a rule filter for a specified Optimizer Statistics Advisor task. The function returns a CLOB that contains the updated values of the filter.

You can use either of the following basic strategies:

- Include findings for all objects (by default, all objects are considered), and then exclude findings for specified objects.
- Exclude findings for all objects, and then include findings only for specified objects.

Prerequisites

To use the `DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER` function, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.

Note:

This subprogram executes using invoker's rights.

To create an object filter:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Either exclude or include objects for a specified task using the `DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER` function.

   Invoke the function in the following form, where the placeholders are defined as follows:

   - *`report`* is the CLOB variable that contains the returned XML.
   - *`tname`* is the name of the task.
   - *`opr_type`* is the type of operation to perform.
   - *`rule`* is the name of the rule.
   - *`owner`* is the schema for the objects.
   - *`table`* is the name of the table.
   - *`action`* is the name of the action: `ENABLE`, `DISABLE`, `DELETE`, or `SHOW`.

```
BEGIN
  report := DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER(
    task_name          => 'tname'
  , stats_adv_opr_type => 'opr_type'
  , rule_name          => 'rule'
  , ownname            => 'owner'
  , tabname            => 'table'
  , action              => 'action' );
END;
```

Example 18-3   Including Only Objects in a Single Schema

In this example, for the task named `opt_adv_task1`, your goal is to disable recommendations for all objects except those in the `sh` schema. User account `sh` has been granted `ADVISOR` and `READ ANY TABLE` privileges. You perform the following steps:

1. Log in to the database as `sh`.
2. Drop any existing task named `opt_adv_task1`.

```
DECLARE
  v_tname VARCHAR2(32767);
BEGIN
  v_tname := 'opt_adv_task1';
  DBMS_STATS.DROP_ADVISOR_TASK(v_tname);
END;
/
```

3. Create a procedure named `sh_obj_filter` that restricts a specified task to objects in the `sh` schema.

```
CREATE OR REPLACE PROCEDURE sh_obj_filter(p_tname IN VARCHAR2) IS
   v_retc CLOB;
BEGIN
   -- Filter out all objects that are not in the sh schema
   v_retc := DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER(
               task_name          => p_tname
             , stats_adv_opr_type => 'EXECUTE'
             , rule_name          => NULL
             , ownname            => NULL
             , tabname            => NULL
             , action             => 'DISABLE' );

   v_retc := DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER(
               task_name          => p_tname
             , stats_adv_opr_type => 'EXECUTE'
             , rule_name          => NULL
             , ownname            => 'SH'
             , tabname            => NULL
             , action             => 'ENABLE' );
END;
/
SHOW ERRORS
```

4. Create a task named `opt_adv_task1`, and then execute the `sh_obj_filter` procedure for this task.

```
DECLARE
  v_tname VARCHAR2(32767);
  v_ret VARCHAR2(32767);
BEGIN
```

```
  v_tname := 'opt_adv_task1';
  v_ret   := DBMS_STATS.CREATE_ADVISOR_TASK(v_tname);
  sh_obj_filter(v_tname);
END;
/
```

5. Execute the task `opt_adv_task1`.

```
DECLARE
  v_tname VARCHAR2(32767);
  v_ret   VARCHAR2(32767);
begin
  v_tname := 'opt_adv_task1';
  v_ret   := DBMS_STATS.EXECUTE_ADVISOR_TASK(v_tname);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.CONFIGURE_ADVISOR_OBJ_FILTER`

18.2.3.3 Creating a Rule Filter for an Optimizer Advisor Task

The `DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER` function creates a rule filter for a specified Optimizer Statistics Advisor task. The function returns a CLOB that contains the updated values of the filter.

You can use either of the following basic strategies:

- Enable all rules (by default, all rules are enabled), and then disable specified rules.
- Disable all rules, and then enable only specified rules.

Prerequisites

To use the `DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER` function, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.

Note:

This subprogram executes using invoker's rights.

To create a rule filter:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.

2. Obtain the names of the advisor rules by querying `V$STATS_ADVISOR_RULES`.

   For example, query the view as follows (partial sample output included):

```
SET LINESIZE 200
SET PAGESIZE 100
COL ID FORMAT 99
COL NAME FORMAT a27
COL DESCRIPTION FORMAT a54

SELECT RULE_ID AS ID, NAME, RULE_TYPE, DESCRIPTION
FROM   V$STATS_ADVISOR_RULES
ORDER BY RULE_ID;

ID NAME                        RULE_TYPE DESCRIPTION
-- --------------------------- --------- ------------------------------------------
 1 UseAutoJob                  SYSTEM    Use Auto Job for Statistics Collection
 2 CompleteAutoJob             SYSTEM    Auto Statistics Gather Job should complete
                                         successfully
 3 MaintainStatsHistory        SYSTEM    Maintain Statistics History
 4 UseConcurrent               SYSTEM    Use Concurrent preference for Statistics
                                         Collection
...
```

3. Either exclude or include rules for a specified task using the `DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER` function.

   Invoke the function in the following form, where the placeholders are defined as follows:

   - *`tname`* is the name of the task.
   - *`report`* is the CLOB variable that contains the returned XML.
   - *`opr_type`* is the type of operation to perform.
   - *`rule`* is the name of the rule.
   - *`action`* is the name of the action: `ENABLE`, `DISABLE`, `DELETE`, or `SHOW`.

   ```
   BEGIN
     report := DBMS_STATS.DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
       task_name          => 'tname'
     , stats_adv_opr_type => 'opr_type'
     , rule_name          => 'rule'
     , action             => 'action' );
   END;
   ```

Example 18-4 Excluding the Rule for Stale Statistics

In this example, you know that statistics are stale because the automated statistics job did not run. You want to generate a report for the task named `opt_adv_task1`, but do not want to clutter it with recommendations about stale statistics.

1. You query `V$STATS_ADVISOR_RULES` for rules that deal with stale statistics (sample output included):

   ```
   COL NAME FORMAT a15
   SELECT RULE_ID AS ID, NAME, RULE_TYPE, DESCRIPTION
   FROM   V$STATS_ADVISOR_RULES
   WHERE  DESCRIPTION LIKE '%tale%'
   ORDER BY RULE_ID;

    ID NAME            RULE_TYPE DESCRIPTION
   --- --------------- --------- -----------------------------------------
    12 AvoidStaleStats OBJECT    Avoid objects with stale or no statistics
   ```

2. You configure a filter using `CONFIGURE_ADVISOR_RULE_FILTER`, specifying that task execution should exclude the rule `AvoidStaleStats`, but honor all other rules:

   ```
   VARIABLE b_ret CLOB
   BEGIN
      :b_ret := DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
        task_name          => 'opt_adv_task1'
   ,    stats_adv_opr_type => 'EXECUTE'
   ,    rule_name          => 'AvoidStaleStats'
   ,    action             => 'DISABLE' );
   END;
   /
   ```

Example 18-5 Including Only the Rule for Avoiding Stale Statistics

This example is the inverse of the preceding example. You want to generate a report for the task named `opt_adv_task1`, but want to see *only* recommendations about stale statistics.

1. Query `V$STATS_ADVISOR_RULES` for rules that deal with stale statistics (sample output included):

   ```
   COL NAME FORMAT a15

   SELECT RULE_ID AS ID, NAME, RULE_TYPE, DESCRIPTION
   FROM   V$STATS_ADVISOR_RULES
   WHERE  DESCRIPTION LIKE '%tale%'
   ORDER BY RULE_ID;

    ID NAME            RULE_TYPE DESCRIPTION
   --- --------------- --------- -----------------------------------------
    12 AvoidStaleStats OBJECT    Avoid objects with stale or no statistics
   ```

2. Configure a filter using `CONFIGURE_ADVISOR_RULE_FILTER`, specifying that task execution should *exclude* all rules:

   ```
   VARIABLE b_ret CLOB
   BEGIN
      :b_ret := DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
        task_name          => 'opt_adv_task1'
   ,    stats_adv_opr_type => 'EXECUTE'
   ,    rule_name          => null
   ,    action             => 'DISABLE' );
   ```

```
END;
/
```

3. Configure a filter that enables only the `AvoidStaleStats` rule:

```
BEGIN
   :b_ret := DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER(
     task_name          => 'opt_adv_task1'
,    stats_adv_opr_type => 'EXECUTE'
,    rule_name          => 'AvoidStaleStats'
,    action             => 'ENABLE' );
END;
/
```

See Also:

- *Oracle Database Reference* to learn more about `V$STATS_ADVISOR_RULES`
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `CONFIGURE_ADVISOR_RULE_FILTER`

18.2.3.4 Creating an Operation Filter for an Optimizer Advisor Task

The `DBMS_STATS.CONFIGURE_ADVISOR_OPR_FILTER` function creates an operation filter for a specified Optimizer Statistics Advisor task. The function returns a CLOB that contains the updated values of the filter.

You can use either of the following basic strategies:

- Disable all operations, and then enable only specified operations.
- Enable all operations (by default, all operations are enabled), and then disable specified operations.

The `DBA_OPTSTAT_OPERATIONS` view contains the IDs of statistics-related operations.

Prerequisites

To use `DBMS_STATS.CONFIGURE_ADVISOR_OPR_FILTER` function, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.

  Note:

  This subprogram executes using invoker's rights.

- You must be the owner of the task.
- To query the `DBA_OPTSTAT_OPERATIONS` view, you must have the `SELECT ANY TABLE` privilege.

To create an operation filter:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Query the types of operations.

   For example, list all distinct operations in `DBA_OPTSTAT_OPERATIONS` (sample output included):

   ```
   SQL> SELECT DISTINCT(OPERATION) FROM DBA_OPTSTAT_OPERATIONS ORDER BY
   OPERATION;

   OPERATION
   -----------------------
   gather_dictionary_stats
   gather_index_stats
   gather_schema_stats
   gather_table_stats
   purge_stats
   set_system_stats
   ```

3. Obtain the IDs of the operations to be filtered by querying `DBA_OPTSTAT_OPERATIONS`.

   For example, to obtain IDs for all statistics gathering operations for tables and indexes in the `SYS` and `sh` schemas, use the following query:

   ```
   SELECT ID
   FROM   DBA_OPTSTAT_OPERATIONS
   WHERE  (  OPERATION = 'gather_table_stats'
             OR OPERATION = 'gather_index_stats')
   AND    (  TARGET LIKE 'SH.%'
             OR TARGET LIKE 'SYS.%');
   ```

4. Exclude or include rules for a specified task using the `DBMS_STATS.CONFIGURE_ADVISOR_OPR_FILTER` function, specifying the IDs obtained in the previous step.

   Invoke the function in the following form, where the placeholders are defined as follows:

   - *`report`* is the CLOB variable that contains the returned XML.
   - *`tname`* is the name of the task.
   - *`opr_type`* is the type of operation to perform. This value cannot be null.
   - *`rule`* is the name of the rule.
   - *`opr_id`* is the ID (from `DBA_OPTSTAT_OPERATIONS.ID`) of the operation to perform. This value cannot be null.
   - *`action`* is the name of the action: `ENABLE`, `DISABLE`, `DELETE`, or `SHOW`.

   ```
   BEGIN
     report := DBMS_STATS.CONFIGURE_ADVISOR_OPR_FILTER(
       task_name          => 'tname'
     , stats_adv_opr_type => 'opr_type'
     , rule_name          => 'rule'
   ```

```
  , operation_id       => 'op_id'
  , action             => 'action' );
END;
```

Example 18-6 Excluding Operations for Gathering Table Statistics

In this example, your goal is to exclude operations that gather table statistics in the `hr` schema. User account `stats` has been granted the `DBA` role, `ADVISOR` privilege, and `SELECT ON DBA_OPTSTAT_OPERATIONS` privilege. You perform the following steps:

1. Log in to the database as `stats`.
2. Drop any existing task named `opt_adv_task1`.

```
DECLARE
  v_tname VARCHAR2(32767);
BEGIN
  v_tname := 'opt_adv_task1';
  DBMS_STATS.DROP_ADVISOR_TASK(v_tname);
END;
/
```

3. Create a procedure named `opr_filter` that configures a task to advise on all operations *except* those that gather statistics for tables in the `hr` schema.

```
CREATE OR REPLACE PROCEDURE opr_filter(p_tname IN VARCHAR2) IS
   v_retc CLOB;
BEGIN
   -- For all rules, prevent the advisor from operating
   -- on the operations selected in the following query
   FOR rec IN
     (SELECT ID FROM DBA_OPTSTAT_OPERATIONS WHERE OPERATION =
'gather_table_stats' AND TARGET LIKE 'HR.%')
   LOOP
     v_retc := DBMS_STATS.CONFIGURE_ADVISOR_OPR_FILTER(
                 task_name          => p_tname
               , stats_adv_opr_type => NULL
               , rule_name          => NULL
               , operation_id       => rec.id
               , action             => 'DISABLE');
   END LOOP;
END;
/
SHOW ERRORS
```

4. Create a task named `opt_adv_task1`, and then execute the `opr_filter` procedure for this task.

```
DECLARE
  v_tname VARCHAR2(32767);
  v_ret VARCHAR2(32767);
BEGIN
  v_tname := 'opt_adv_task1';
  v_ret   := DBMS_STATS.CREATE_ADVISOR_TASK(v_tname);
  opr_filter(v_tname);
```

```
END;
/
```

5. Execute the task `opt_adv_task1`.

```
DECLARE
  v_tname VARCHAR2(32767);
  v_ret   VARCHAR2(32767);
begin
  v_tname := 'opt_adv_task1';
  v_ret   := DBMS_STATS.EXECUTE_ADVISOR_TASK(v_tname);
END;
/
```

6. Print the report.

```
SPOOL /tmp/rep.txt
SET LONG 1000000
COLUMN report FORMAT A200
SET LINESIZE 250
SET PAGESIZE 1000

SELECT DBMS_STATS.REPORT_ADVISOR_TASK(
         task_name      => 'opt_adv_task1'
       , execution_name => NULL
       , type           => 'TEXT'
       , section        => 'ALL'
       ) AS report
FROM   DUAL;
SPOOL OFF
```

See Also:

- *Oracle Database Reference* to learn more about `DBA_OPTSTAT_OPERATIONS`
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `CONFIGURE_ADVISOR_OPR_FILTER`

18.2.4 Executing an Optimizer Statistics Advisor Task

The `DBMS_STATS.EXECUTE_ADVISOR_TASK` function executes a task for Optimizer Statistics Advisor. If you do not specify an execution name, then Optimizer Statistics Advisor generates one automatically.

The results of performing this task depend on the privileges of the executing user:

- `SYSTEM` level

  Only users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task on system-level rules.
- Operation level

The results depend on the following privileges:

  - Users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task for all statistics operations.
  - Users with the `ANALYZE ANY` privilege but *not* the `ANALYZE ANY DICTIONARY` privilege can perform this task for statistics operations related to any schema except `SYS`.
  - Users with the `ANALYZE ANY DICTIONARY` privilege but *not* the `ANALYZE ANY` privilege can perform this task for statistics operations related to their own schema and the `SYS` schema.
  - Users with neither the `ANALYZE ANY` nor the `ANALYZE ANY DICTIONARY` privilege can only perform this operation for statistics operations relating to their own schema.

- Object level

  Users can perform this task for any object for which they have statistics collection privileges.

Prerequisites

This task has the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.
- If you specify an execution name, then this name must not conflict with an existing execution.

Note:

This subprogram executes using invoker's rights.

To execute an Optimizer Statistics Advisor task:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Execute the `DBMS_STATS.EXECUTE_ADVISOR_TASK` function in the following form, where *`tname`* is the name of the task, *`execname`* is the optional name of the execution, and *`ret`* is the variable that contains the returned output:

   ```
   EXECUTE ret := DBMS_STATS.EXECUTE_ADVISOR_TASK('tname','execname');
   ```

   For example, to execute the task `opt_adv_task1`, use the following code:

   ```
   DECLARE
     v_tname VARCHAR2(32767);
     v_ret   VARCHAR2(32767);
   BEGIN
     v_tname := 'opt_adv_task1';
     v_ret := DBMS_STATS.EXECUTE_ADVISOR_TASK(v_tname);
   ```

```
END;
/
```

3. Optionally, obtain details about the execution by querying `USER_ADVISOR_EXECUTIONS`:

```
SELECT TASK_NAME, EXECUTION_NAME,
       EXECUTION_END, EXECUTION_TYPE AS TYPE, STATUS
FROM   USER_ADVISOR_EXECUTIONS;
```

   Sample output appears below:

```
TASK_NAME       EXECUTION_NAME       EXECUTION TYPE       STATUS
--------------- -------------------- --------- ---------- -----------
OPT_ADV_TASK1   EXEC_136             23-NOV-15 STATISTICS COMPLETED
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `EXECUTE_ADVISOR_TASK`

18.2.5 Generating a Report for an Optimizer Statistics Advisor Task

The `DBMS_STATS.REPORT_ADVISOR_TASK` function generates a report for an Optimizer Statistics Advisor task.

The report contains the following sections:

- General information

  This section describes the task name, execution name, creation date, and modification date.

- Summary

  This section summarizes the findings and rules violated by the findings.

- Findings

  Each finding section lists the relevant rule and findings. If the advisor has a recommendation, then the recommendation is described. In some cases, a recommendation also has a rationale.

The name of the automated Optimizer Statistics Advisor task is `AUTO_STATS_ADVISOR_TASK`. If you follow the automated workflow, then you only need to query the automatically generated report.

Prerequisites

To generate a report with the `DBMS_STATS.REPORT_ADVISOR_TASK` function, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.

> **Note:**
>
> This subprogram executes using invoker's rights.

The results of performing this task depend on the privileges of the executing user:

- `SYSTEM` level

  Only users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task on system-level rules.
- Operation level

  The results depend on the following privileges:
  - Users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task for all statistics operations.
  - Users with the `ANALYZE ANY` privilege but *not* the `ANALYZE ANY DICTIONARY` privilege can perform this task for statistics operations related to any schema except `SYS`.
  - Users with the `ANALYZE ANY DICTIONARY` privilege but *not* the `ANALYZE ANY` privilege can perform this task for statistics operations related to their own schema and the `SYS` schema.
  - Users with neither the `ANALYZE ANY` nor the `ANALYZE ANY DICTIONARY` privilege can only perform this operation for statistics operations relating to their own schema.
- Object level

  Users can perform this task for any object for which they have statistics collection privileges.

To generate an Optimizer Statistics Advisor report:

1. In SQL*Plus, log in to the database as a user with `ADVISOR` privileges.
2. Query the `DBMS_STATS.REPORT_ADVISOR_TASK` function output.

   Use the following query, where the placeholders have the following definitions:
   - *`tname`* is the name of the task.
   - *`exec`* is the name of the execution.
   - *`type`* is the type of output: `TEXT`, `HTML`, or `XML`.
   - *`sect`* is the section of the report: `SUMMARY`, `FINDINGS`, `ERRORS`, and `ALL`.
   - *`lvl`* is the format of the report: `BASIC`, `TYPICAL`, `ALL`, or `SHOW_HIDDEN`.

   ```
   SET LINESIZE 3000
   SET LONG 500000
   SET PAGESIZE 0
   SET LONGCHUNKSIZE 100000

   SELECT DBMS_STATS.REPORT_ADVISOR_TASK('tname', 'exec', 'type',
   ```

```
       'sect', 'lvl') AS REPORT
FROM   DUAL;
```

For example, to print a report for `AUTO_STATS_ADVISOR_TASK`, use the following query:

```
SELECT DBMS_STATS.REPORT_ADVISOR_TASK('AUTO_STATS_ADVISOR_TASK', NULL,
       'TEXT', 'ALL', 'ALL') AS REPORT
FROM   DUAL;
```

The following sample report shows four findings:

```
GENERAL INFORMATION
-------------------------------------------------------------------------------
 Task Name         : AUTO_STATS_ADVISOR_TASK
 Execution Name    : EXEC_136
 Created           : 09-05-16 02:52:34
 Last Modified     : 09-05-16 12:31:24
-------------------------------------------------------------------------------
SUMMARY
-------------------------------------------------------------------------------
 For execution EXEC_136 of task AUTO_STATS_ADVISOR_TASK, the Statistics Advisor
 has 4 findings. The findings are related to the following rules:
 AVOIDSETPROCEDURES, USEDEFAULTPARAMS, USEGATHERSCHEMASTATS, NOTUSEINCREMENTAL.
Please refer to the finding section for detailed information.

-------------------------------------------------------------------------------
FINDINGS
-------------------------------------------------------------------------------
 Rule Name:        AvoidSetProcedures
 Rule Description:  Avoid Set Statistics Procedures
 Finding:  There are 5 SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS procedures being
           used for statistics gathering.
 Recommendation:  Do not use SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS procedures.
                  Gather statistics instead of setting them.
 Rationale:  SET_[COLUMN|INDEX|TABLE|SYSTEM]_STATS will cause bad plans due to
             wrong or inconsistent statistics.
----------------------------------------------------
 Rule Name:        UseDefaultParams
 Rule Description:  Use Default Parameters in Statistics Collection Procedures
 Finding:  There are 367 statistics operations using nondefault parameters.
 Recommendation:  Use default parameters for statistics operations.
 Example:

 -- Gathering statistics for 'SH' schema using all default parameter values:
 BEGIN dbms_stats.gather_schema_stats('SH'); END;
 Rationale:  Using default parameter values for statistics gathering operations
             is more efficient.
----------------------------------------------------
 Rule Name:        UseGatherSchemaStats
 Rule Description:  Use gather_schema_stats procedure
 Finding:  There are 318 uses of GATHER_TABLE_STATS.
 Recommendation:  Use GATHER_SCHEMA_STATS instead of GATHER_TABLE_STATS.
 Example:
```

```
 -- Gather statistics for 'SH' schema:
 BEGIN dbms_stats.gather_schema_stats('SH'); END;
 Rationale:  GATHER_SCHEMA_STATS has more options available, including checking
             for staleness and gathering statistics concurrently. Also it is
             more maintainable for new tables added to the schema. If you only
             want to gather statistics for certain tables in the schema, specify
             them in the obj_filter_list parameter of
GATHER_SCHEMA_STATS.
----------------------------------------------------
 Rule Name:        NotUseIncremental

 Rule Description:  Statistics should not be maintained incrementally when it is
not
 Finding:  Incremental option has been turned on for 10 tables, which will not
benefit
           from using the incremental option.
 Schema:
 SH
 Objects:
 CAL_MONTH_SALES_MV
 CAL_MONTH_SALES_MV
 CHANNELS
 COUNTRIES
 CUSTOMERS
 DIMENSION_EXCEPTIONS
 FWEEK_PSCAT_SALES_MV
 FWEEK_PSCAT_SALES_MV
 PRODUCTS
 PROMOTIONS
 SUPPLEMENTARY_DEMOGRAPHICS
 TIMES

 Recommendation:  Do not use the incremental option for statistics gathering on
these
                  objects.
 Example:
 --
 Turn off the incremental option for 'SH.SALES':
 dbms_stats.set_table_prefs('SH', 'SALES', 'INCREMENTAL', 'FALSE');
 Rationale:  The overhead of using the incremental option on these tables
             outweighs the benefit of using the incremental option.
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `REPORT_ADVISOR_TASK`

18.2.6 Implementing Optimizer Statistics Advisor Recommendations

You can either implement all recommendations automatically using `DBMS_STATS.IMPLEMENT_ADVISOR_TASK`, or generate an editable script using `DBMS_STATS.SCRIPT_ADVISOR_TASK`.

18.2.6.1 Implementing Actions Recommended by Optimizer Statistics Advisor

The `DBMS_STATS.IMPLEMENT_ADVISOR_TASK` function implements the recommendations for a specified Optimizer Statistics Advisor task. If you do not specify an execution name, then Optimizer Statistics Advisor uses the most recent execution.

The simplest means of implementing recommendations is using `DBMS_STATS.IMPLEMENT_ADVISOR_TASK`. In this case, no generation of a script is necessary. You can specify that the advisor ignore the existing filters `(level=>'ALL')` or use the default, which honors the existing filters `(level=>'TYPICAL')`.

Prerequisites

To use `DBMS_STATS.IMPLEMENT_ADVISOR_TASK`, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.

Note:

This subprogram executes using invoker's rights.

The results of performing this task depend on the privileges of the executing user:

- `SYSTEM` level

  Only users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task on system-level rules.
- Operation level

  The results depend on the following privileges:
  - Users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task for all statistics operations.
  - Users with the `ANALYZE ANY` privilege but *not* the `ANALYZE ANY DICTIONARY` privilege can perform this task for statistics operations related to any schema except `SYS`.
  - Users with the `ANALYZE ANY DICTIONARY` privilege but *not* the `ANALYZE ANY` privilege can perform this task for statistics operations related to their own schema and the `SYS` schema.
  - Users with neither the `ANALYZE ANY` nor the `ANALYZE ANY DICTIONARY` privilege can only perform this operation for statistics operations relating to their own schema.
- Object level

Users can perform this task for any object for which they have statistics collection privileges.

To implement advisor actions:

1. In SQL*Plus, log in to the database as a user with the necessary privileges.
2. Execute the `DBMS_STATS.IMPLEMENT_ADVISOR_TASK` function in the following form, where the placeholders have the following definitions:
   - *`tname`* is the name of the task.
   - *`result`* is the CLOB variable that contains a list of the recommendations that have been successfully implemented.
   - *`fltr_lvl`* is the level of implementation: `TYPICAL` (existing filters honored) or `ALL` (filters ignored).

   ```
   BEGIN
     result := DBMS_STATS.IMPLEMENT_ADVISOR_TASK('tname', level => 
   fltr_lvl);
   END;
   ```

   For example, to implement all recommendations for the task `opt_adv_task1`, use the following code:

   ```
   VARIABLE b_ret CLOB
   DECLARE
     v_tname VARCHAR2(32767);
   BEGIN
     v_tname := 'opt_adv_task1';
     :b_ret := DBMS_STATS.IMPLEMENT_ADVISOR_TASK(v_tname);
   END;
   /
   ```

3. Optionally, print the XML output to confirm the implemented actions.

   For example, to print the XML returned in the previous step, use the following code (sample output included):

   ```
   SET LONG 10000
   SELECT XMLType(:b_ret) AS imp_results FROM DUAL;


   IMP_RESULTS
   ------------------------------------
   <implementation_results>
     <rule NAME="AVOIDSETPROCEDURES">
       <implemented>yes</implemented>
     </rule>
     <rule NAME="USEGATHERSCHEMASTATS">
       <implemented>yes</implemented>
     </rule>
     <rule NAME="AVOIDSETPROCEDURES">
       <implemented>yes</implemented>
     </rule>
   ```

```
    <rule NAME="USEGATHERSCHEMASTATS">
      <implemented>yes</implemented>
    </rule>
    <rule NAME="USEDEFAULTPARAMS">
      <implemented>no</implemented>
    </rule>
    <rule NAME="USEDEFAULTPARAMS">
      <implemented>yes</implemented>
    </rule>
    <rule NAME="NOTUSEINCREMENTAL">
      <implemented>yes</implemented>
    </rule>
  </implementation_results>
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.IMPLEMENT_ADVISOR_TASK`

18.2.6.2 Generating a Script Using Optimizer Statistics Advisor

The `DBMS_STATS.SCRIPT_ADVISOR_TASK` function generates an editable script with recommendations for a specified Optimizer Statistics Advisor task.

Unlike `IMPLEMENT_ADVISOR_TASK`, the `SCRIPT_ADVISOR_TASK` generates a script that you can edit before execution. The output script contains both comments and executable code. As with `IMPLEMENT_ADVISOR_TASK`, you can specify that the advisor ignore the existing filters `(level=>'ALL')` or use the default, which honors the existing filters `(level=>'TYPICAL')`. You can specify that the function returns the script as a CLOB and file, or only a CLOB.

Prerequisites

To use the `DBMS_STATS.SCRIPT_ADVISOR_TASK` function, you must meet the following prerequisites:

- To execute this subprogram, you must have the `ADVISOR` privilege.
- You must be the owner of the task.

Note:

This subprogram executes using invoker's rights.

The results of performing this task depend on the privileges of the executing user:

- `SYSTEM` level

  Only users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task on system-level rules.
- Operation level

The results depend on the following privileges:

  - Users with both the `ANALYZE ANY` and `ANALYZE ANY DICTIONARY` privileges can perform this task for all statistics operations.
  - Users with the `ANALYZE ANY` privilege but *not* the `ANALYZE ANY DICTIONARY` privilege can perform this task for statistics operations related to any schema except `SYS`.
  - Users with the `ANALYZE ANY DICTIONARY` privilege but *not* the `ANALYZE ANY` privilege can perform this task for statistics operations related to their own schema and the `SYS` schema.
  - Users with neither the `ANALYZE ANY` nor the `ANALYZE ANY DICTIONARY` privilege can only perform this operation for statistics operations relating to their own schema.

- Object level

  Users can perform this task for any object for which they have statistics collection privileges.

To generate an advisor script:

1. In SQL*Plus, log in to the database as a user with `ADVISOR` privileges.
2. Execute the `DBMS_STATS.SCRIPT_ADVISOR_TASK` function in the following form, where the placeholders have the following definitions:

   - *`tname`* is the name of the task.
   - *`exec`* is the name of the execution (default is null).
   - *`dir`* is the name of the directory (default is null).
   - *`result`* is the CLOB variable that contains a list of the recommendations that have been successfully implemented.
   - *`filter_lvl`* is the level of implementation: `TYPICAL` (existing filters honored) or `ALL` (filters ignored).

```
BEGIN
  result := DBMS_STATS.SCRIPT_ADVISOR_TASK('tname',
            execution_name => 'exec', dir_name => 'dir',
            level => 'filter_lvl');
END;
```

   For example, to generate a script that contains recommendations for the task `opt_adv_task1`, use the following code:

```
VARIABLE b_script CLOB
DECLARE
  v_tname VARCHAR2(32767);
BEGIN
  v_tname := 'opt_adv_task1';
  :b_script := DBMS_STATS.SCRIPT_ADVISOR_TASK(v_tname);
END;
/
```

Note:

If you do not specify an execution name, then Optimizer Statistics Advisor uses the most recent execution.

3. Print the script.

   For example, to print the script returned in the previous step, use the following code (sample output included):

```
DECLARE
  v_len    NUMBER(10);
  v_offset NUMBER(10) :=1;
  v_amount NUMBER(10) :=10000;
BEGIN
  v_len := DBMS_LOB.getlength(:b_script);
  WHILE (v_offset < v_len)
  LOOP
    DBMS_OUTPUT.PUT_LINE(DBMS_LOB.SUBSTR(:b_script,v_amount,v_offset));
    v_offset := v_offset + v_amount;
  END LOOP;
END;
/
```

   The following example shows a sample script:

```
-- Script generated for the recommendations from execution EXEC_23
-- in the statistics advisor task OPT_ADV_TASK1
-- Script version 12.2

-- No scripts will be provided for the rule AVOIDSETPROCEDURES.  Please check the
-- report for more details.
-- No scripts will be provided for the rule USEGATHERSCHEMASTATS. Please check the
-- report for more details.
-- No scripts will be provided for the rule AVOIDINEFFICIENTSTATSOPRSEQ. Please check
-- the report for more details.
-- No scripts will be provided for the rule AVOIDUNNECESSARYSTATSCOLLECTION. Please
-- check the report for more details.
-- No scripts will be provided for the rule GATHERSTATSAFTERBULKDML. Please check the
-- report for more details.
-- No scripts will be provided for the rule AVOIDDROPRECREATE. Please check the report
-- for more details.
-- No scripts will be provided for the rule AVOIDOUTOFRANGE. Please check the report
-- for more details.
-- No scripts will be provided for the rule AVOIDANALYZETABLE. Please check the report
-- for more details.
-- No scripts will be provided for the rule AVOIDSETPROCEDURES. Please check the
-- report for more details.
-- No scripts will be provided for the rule USEGATHERSCHEMASTATS. Please check the
-- report for more details.
-- No scripts will be provided for the rule AVOIDINEFFICIENTSTATSOPRSEQ. Please
-- check the report for more details.
-- No scripts will be provided for the rule AVOIDUNNECESSARYSTATSCOLLECTION. Please
```

```
check
-- the report for more details.
-- No scripts will be provided for the rule GATHERSTATSAFTERBULKDML. Please check
the
-- report for more details.
-- No scripts will be provided for the rule AVOIDDROPRECREATE. Please check the
report
-- for more details.
-- No scripts will be provided for the rule AVOIDOUTOFRANGE. Please check the
report
-- for more details.
-- No scripts will be provided for the rule AVOIDANALYZETABLE. Please check the
report
-- for more details.

-- Scripts for rule USEDEFAULTPARAMS
-- Rule Description: Use Default Parameters in Statistics Collection Procedures
-- Use the default preference value for parameters

begin dbms_stats.set_global_prefs('PREFERENCE_OVERRIDES_PARAMETER', 'TRUE'); end;
/

-- Scripts for rule USEDEFAULTOBJECTPREFERENCE
-- Rule Description: Use Default Object Preference for statistics collection
-- Setting object-level preferences to default values
-- setting CASCADE to default value for object level preference
-- setting ESTIMATE_PERCENT to default value for object level preference
-- setting METHOD_OPT to default value for object level preference
-- setting GRANULARITY to default value for object level preference
-- setting NO_INVALIDATE to default value for object levelpreference

-- Scripts for rule USEINCREMENTAL
-- Rule Description: Statistics should be maintained incrementally when it is
-- beneficial.
-- Turn on the incremental option for those objects for which using incremental is
-- helpful.

-- Scripts for rule UNLOCKNONVOLATILETABLE
-- Rule Description: Statistics for objects with non-volatile should not be locked
-- Unlock statistics for objects that are not volatile.

-- Scripts for rule LOCKVOLATILETABLE
-- Rule Description: Statistics for objects with volatile data should be locked
-- Lock statistics for volatile objects.

-- Scripts for rule NOTUSEINCREMENTAL
-- Rule Description: Statistics should not be maintained incrementally when it is
not
   beneficial
-- Turn off incremental option for those objects for which using incremental is not
-- helpful.

begin dbms_stats.set_table_prefs('SH', 'CAL_MONTH_SALES_MV', 'INCREMENTAL',
'FALSE'); end;
/
```

```
begin dbms_stats.set_table_prefs('SH', 'CHANNELS', 'INCREMENTAL', 'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'COUNTRIES', 'INCREMENTAL', 'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'CUSTOMERS', 'INCREMENTAL', 'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'DIMENSION_EXCEPTIONS', 'INCREMENTAL', 'FALSE');
end;
/
begin dbms_stats.set_table_prefs('SH', 'FWEEK_PSCAT_SALES_MV', 'INCREMENTAL', 'FALSE');
end;
/
begin dbms_stats.set_table_prefs('SH', 'PRODUCTS', 'INCREMENTAL', 'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'PROMOTIONS', 'INCREMENTAL', 'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'SUPPLEMENTARY_DEMOGRAPHICS', 'INCREMENTAL',
'FALSE'); end;
/
begin dbms_stats.set_table_prefs('SH', 'TIMES', 'INCREMENTAL', 'FALSE'); end;
/

-- Scripts for rule USEAUTODEGREE
-- Rule Description: Use Auto Degree for statistics collection
-- Turn on auto degree for those objects for which using auto degree is helpful.

-- Scripts for rule AVOIDSTALESTATS
-- Rule Description: Avoid objects with stale or no statistics
-- Gather statistics for those objcts that are missing or have no statistics.

-- Scripts for rule MAINTAINSTATSCONSISTENCY
-- Rule Description: Statistics of dependent objects should be consistent
-- Gather statistics for those objcts that are missing or have no statistics.
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_STATS.SCRIPT_ADVISOR_TASK`

Part VI
Optimizer Controls

You can use hints and initialization parameter to influence optimizer decisions and behavior.

19

Influencing the Optimizer

Optimizer defaults are adequate for most operations, but not all.

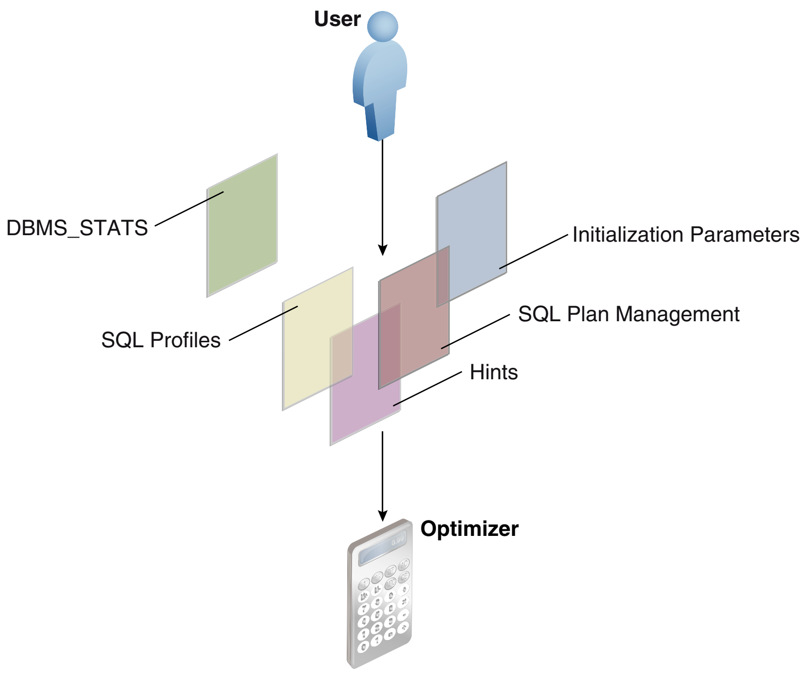
In some cases you may have information unknown to the optimizer, or need to tune the optimizer for a specific type of statement or workload. In such cases, influencing the optimizer may provide better performance.

19.1 Techniques for Influencing the Optimizer

You can influence the optimizer using several techniques, including SQL profiles, SQL Plan Management, initialization parameters, and hints.

The following figure shows the principal techniques for influencing the optimizer.

Figure 19-1 Techniques for Influencing the Optimizer



The overlapping squares in the preceding diagram show that SQL plan management uses both initialization parameters and hints. SQL profiles also technically include hints.

Note:

A stored outline is a legacy technique that serve a similar purpose to SQL plan baselines.

You can use the following techniques to influence the optimizer:

Table 19-1 Optimizer Techniques

Technique	Description	To Learn More
Initialization parameters	Parameters influence many types of optimizer behavior at the database instance and session level.	"Influencing the Optimizer with Initialization Parameters"
Hints	A hint is a commented instruction in a SQL statement. Hints control a wide range of behavior.	"Influencing the Optimizer with Hints"
`DBMS_STATS`	This package updates and manages optimizer statistics. The more accurate the statistics, the better the optimizer estimates. This chapter does not cover `DBMS_STATS`.	"Gathering Optimizer Statistics"
SQL profiles	A SQL profile is a database object that contains auxiliary statistics specific to a SQL statement. Conceptually, a SQL profile is to a SQL statement what a set of object-level statistics is to a table or index. A SQL profile can correct suboptimal optimizer estimates discovered during SQL tuning.	"Managing SQL Profiles"
SQL plan management and stored outlines	SQL plan management is a preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans. This chapter does not cover SQL plan management.	"Managing SQL Plan Baselines"

In some cases, multiple techniques optimize the same behavior. For example, you can set optimizer goals using both initialization parameters and hints.

See Also:

"Migrating Stored Outlines to SQL Plan Baselines" to learn how to migrate stored outlines to SQL plan baselines

19.2 Influencing the Optimizer with Initialization Parameters

This chapter explains which initialization parameters affect optimization, and how to set them.

19.2.1 About Optimizer Initialization Parameters

Oracle Database provides initialization parameters to influence various aspects of optimizer behavior, including cursor sharing, adaptive optimization, and the optimizer mode.

The following table lists some of the most important optimizer parameters. Note that this table does not include the approximate query initialization parameters, which are described in "Approximate Query Initialization Parameters".

Table 19-2 Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
`CURSOR_INVALIDATION`	Provides the default cursor invalidation level for DDL statements. `IMMEDIATE` sets the same cursor invalidation behavior for DDL as in releases before Oracle Database 12c Release 2 (12.2). This is the default. `DEFERRED` allows an application to take advantage of the reduced cursor invalidation for DDL without making any application changes. Deferred invalidation reduces the number of cursor invalidations and spreads the recompilation workload over time. For this reason, a cursor may run with a suboptimal plan until it is recompiled, and may incur small execution-time overhead. You can set this parameter at the `SYSTEM` or `SESSION` level. See "About the Life Cycle of Shared Cursors".
`CURSOR_SHARING`	Converts literal values in SQL statements to bind variables. Converting the values improves cursor sharing and can affect the execution plans of SQL statements. The optimizer generates the execution plan based on the presence of the bind variables and not the actual literal values. Set to `FORCE` to enable the creation of a new cursor when sharing an existing cursor, or when the cursor plan is not optimal. Set to `EXACT` to allow only statements with identical text to share the same cursor.
`DB_FILE_MULTIBLOCK_READ_COUNT`	Specifies the number of blocks that are read in a single I/O during a full table scan or index fast full scan. The optimizer uses the value of this parameter to calculate the cost of full table scans and index fast full scans. Larger values result in a lower cost for full table scans, which may result in the optimizer choosing a full table scan over an index scan. The default value of this parameter corresponds to the maximum I/O size that the database can perform efficiently. This value is platform-dependent and is 1 MB for most platforms. Because the parameter is expressed in blocks, it is set to a value equal to the maximum I/O size that can be performed efficiently divided by the standard block size. If the number of sessions is extremely large, then the multiblock read count value decreases to avoid the buffer cache getting flooded with too many table scan buffers.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
`OPTIMIZER_ADAPTIVE_PLANS`	Controls adaptive plans. An adaptive plan has alternative choices. The optimizer decides on a plan at run time based on statistics collected as the query executes. By default, this parameter is `true`, which means adaptive plans are enabled. Setting to this parameter to `false` disables the following features: • Nested loops and hash join selection • Star transformation bitmap pruning • Adaptive parallel distribution method See "About Adaptive Query Plans".
`OPTIMIZER_ADAPTIVE_REPORTING_ONLY`	Controls the reporting mode for automatic reoptimization and adaptive plans (see "Adaptive Query Plans"). By default, reporting mode is off (`false`), which means that adaptive optimizations are enabled. If set to `true`, then adaptive optimizations run in reporting-only mode. In this case, the database gathers information required for an adaptive optimization, but takes no action to change the plan. For example, an adaptive plan always choose the default plan, but the database collects information about which plan the database would use if the parameter were set to `false`. You can view the report by using `DBMS_XPLAN.DISPLAY_CURSOR`.
`OPTIMIZER_ADAPTIVE_STATISTICS`	Controls adaptive statistics. The optimizer can use adaptive statistics when query predicates are too complex to rely on base table statistics alone. By default, `OPTIMIZER_ADAPTIVE_STATISTICS` is `false`, which means that the following features are disabled: • SQL plan directives • Statistics feedback • Adaptive dynamic sampling See "Adaptive Statistics".
`OPTIMIZER_MODE`	Sets the optimizer mode at database instance startup. Possible values are `ALL_ROWS`, `FIRST_ROWS_`*`n`*, and `FIRST_ROWS`.
`OPTIMIZER_INDEX_CACHING`	Controls the cost analysis of an index probe with a nested loop. The range of values `0` to `100` indicates percentage of index blocks in the buffer cache, which modifies optimizer assumptions about index caching for nested loops and IN-list iterators. A value of `100` infers that 100% of the index blocks are likely to be found in the buffer cache, so the optimizer adjusts the cost of an index probe or nested loop accordingly. Use caution when setting this parameter because execution plans can change in favor of index caching.
`OPTIMIZER_INDEX_COST_ADJ`	Adjusts the cost of index probes. The range of values is `1` to `10000`. The default value is `100`, which means that the optimizer evaluates indexes as an access path based on the normal cost model. A value of `10` means that the cost of an index access path is one-tenth the normal cost of an index access path.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
`OPTIMIZER_INMEMORY_AWARE`	This parameter enables (`TRUE`) or disables (`FALSE`) all Oracle Database In-Memory (Database In-Memory) optimizer features, including the cost model for the IM column store, table expansion, Bloom filters, and so on. Setting the parameter to `FALSE` causes the optimizer to ignore the `INMEMORY` property of tables during the optimization of SQL statements.
`OPTIMIZER_REAL_TIME_STATISTICS`	When the `OPTIMIZER_REAL_TIME_STATISTICS` initialization parameter is set to `true`, Oracle Database automatically gathers real-time statistics during conventional DML operations. The default setting is `false`, which means real-time statistics are disabled.
`OPTIMIZER_SESSION_TYPE`	Determines whether the database verifies statements during automatic index verification. The default is `NORMAL`, which means statements are verified. `CRITICAL` takes precedence over `NORMAL`. By setting the `OPTIMIZER_SESSION_TYPE` initialization parameter to `ADHOC` in a session, you can suspend automatic indexing for queries in this session. The automatic indexing process does not identify index candidates, or create and verify indexes. This control may be useful for ad hoc queries or testing new functionality.
`OPTIMIZER_CAPTURE_SQL_QUARANTINE`	Enables or disables the automatic creation of SQL Quarantine configurations.To enable SQL Quarantine to create configurations automatically after the Resource Manager terminates a query, set the `OPTIMIZER_CAPTURE_SQL_QUARANTINE` initialization parameter to `TRUE` (the default is `FALSE`).
`OPTIMIZER_USE_INVISIBLE_INDEXES`	Enables or disables the use of invisible indexes.
`QUERY_REWRITE_ENABLED`	Enables or disables the query rewrite feature of the optimizer. `TRUE`, which is the default, enables the optimizer to utilize materialized views to enhance performance. `FALSE` disables the query rewrite feature of the optimizer and directs the optimizer not to rewrite queries using materialized views even when the estimated query cost of the unoptimized query is lower. `FORCE` enables the query rewrite feature of the optimizer and directs the optimizer to rewrite queries using materialized views even when the estimated query cost of the unoptimized query is lower.
`OPTIMIZER_USE_SQL_QUARANTINE`	Determines whether the optimizer considers SQL Quarantine configurations when choosing an execution plan for a SQL statement. To disable the use of existing SQL Quarantine configurations, set `OPTIMIZER_USE_SQL_QUARANTINE` to `FALSE` (the default is `TRUE`).

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
QUERY_REWRITE_INTEGRITY	Determines the degree to which query rewrite is enforced. By default, the integrity level is set to ENFORCED. In this mode, all constraints must be validated. The database does not use query rewrite transformations that rely on unenforced constraints. Therefore, if you use ENABLE NOVALIDATE RELY, some types of query rewrite might not work. To enable query rewrite when constraints are in NOVALIDATE mode, the integrity level must be TRUSTED or STALE_TOLERATED. In TRUSTED mode, the optimizer trusts that the relationships declared in dimensions and RELY constraints are correct. In STALE_TOLERATED mode, the optimizer uses materialized views that are valid but contain stale data as well as those that contain fresh data. This mode offers the maximum rewrite capability but creates the risk of generating inaccurate results.
RESULT_CACHE_MODE	Controls whether the database uses the SQL query result cache for all queries, or only for the queries that are annotated with the result cache hint. When set to MANUAL (default), you must use the RESULT_CACHE hint to specify that a specific result is to be stored in the cache. When set to FORCE, the database stores all results in the cache. The corresponding options MANUAL TEMP and FORCE TEMP specify that query results can reside in the temporary tablespace, unless prohibited by a hint. When setting this parameter, consider how the result cache handles PL/SQL functions. The database invalidates query results in the result cache using the same mechanism that tracks data dependencies for PL/SQL functions, but otherwise permits caching of queries that contain PL/SQL functions. Because PL/SQL function result cache invalidation does not track all kinds of dependencies (such as on sequences, SYSDATE, SYS_CONTEXT, and package variables), indiscriminate use of the query result cache on queries calling such functions can result in changes to results, that is, incorrect results. Thus, consider correctness and performance when choosing to enable the result cache, especially when setting RESULT_CACHE_MODE to FORCE.
RESULT_CACHE_MAX_SIZE	Specifies the maximum amount of SGA memory (in bytes) that can be used by the result cache. The default is derived from the values of SHARED_POOL_SIZE, SGA_TARGET, and MEMORY_TARGET. The value of this parameter is rounded to the largest multiple of 32 KB that is not greater than the specified value. The value 0 disables the cache.
RESULT_CACHE_MAX_RESULT	Specifies the percentage of RESULT_CACHE_MAX_SIZE that any single result can use. The default value is 5, but you can specify any percentage value between 1 and 100.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
RESULT_CACHE_MAX_TEMP_RESULT	Specifies the maximum percentage of temporary tablespace memory that one cached query can consume. The default value is 5. This parameter is only modifiable at the system level.
RESULT_CACHE_MAX_TEMP_SIZE	Specifies the maximum amount of temporary tablespace memory that the result cache can consume in a PDB. This parameter is only modifiable at the system level. The default is 10 times the default or initialized value of RESULT_CACHE_MAX_SIZE. Any positive value below 5 is rounded to 5. The specified value cannot exceed 10% of the currently estimated total free temporary tablespace in the SYS schema. The value 0 disables the feature.
RESULT_CACHE_REMOTE_EXPIRATION	Specifies the number of minutes for which a result that depends on remote database objects remains valid. The default is 0, which implies that the database should not cache results using remote objects. Setting this parameter to a nonzero value can produce stale answers, such as if a remote database modifies a table that is referenced in a result.
STAR_TRANSFORMATION_ENABLED	Enables the optimizer to cost a star transformation for star queries (if true). The star transformation combines the bitmap indexes on the various fact table columns.

See Also:

- *Oracle Database Performance Tuning Guide* to learn how to tune the query result cache
- *Oracle Database Data Warehousing Guide*

  to learn more about star transformations and query rewrite
- *Oracle Database In-Memory Guide* to learn more about Database In-Memory features
- *Oracle Database Reference* for complete information about the preceding initialization parameters

19.2.2 Enabling Optimizer Features

The OPTIMIZER_FEATURES_ENABLE initialization parameter (or hint) controls a set of optimizer-related features, depending on the database release.

The parameter accepts one of a list of valid string values corresponding to the release numbers, such as 11.2.0.2 or 12.2.0.1. You can use this parameter to preserve the old behavior of the optimizer after a database upgrade. For example, if you upgrade Oracle Database 12c Release 1 (12.1.0.2) to Oracle Database 12c Release 2 (12.2.0.1), then the default value of the `OPTIMIZER_FEATURES_ENABLE` parameter changes from `12.1.0.2` to `12.2.0.1`.

For backward compatibility, you may not want the execution plans to change because of new optimizer features in a new release. In such cases, you can set `OPTIMIZER_FEATURES_ENABLE` to an earlier version. If you upgrade to a new release, and if you *want* to enable the features in the new release, then you do *not* need to explicitly set the `OPTIMIZER_FEATURES_ENABLE` initialization parameter.

Caution:

Oracle does not recommend explicitly setting the `OPTIMIZER_FEATURES_ENABLE` initialization parameter to an earlier release. To avoid SQL performance regression that may result from execution plan changes, consider using SQL plan management instead.

Assumptions

This tutorial assumes the following:

- You recently upgraded the database from Oracle Database 12c Release 1 (12 1.0.2) to Oracle Database 12c Release 2 (12.2.0.1).
- You want to preserve the optimizer behavior from the earlier release.

To enable query optimizer features for a specific release:

1. Log in to the database with the appropriate privileges, and then query the current optimizer features settings.

   For example, run the following SQL*Plus command:

   ```
   SQL> SHOW PARAMETER optimizer_features_enable

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- --------
   optimizer_features_enable            string      12.2.0.1
   ```

2. Set the optimizer features setting at the instance or session level.

   For example, run the following SQL statement to set the optimizer version to `12.1.0.2`:

   ```
   SQL> ALTER SYSTEM SET OPTIMIZER_FEATURES_ENABLE='12.1.0.2';
   ```

   The preceding statement restores the optimizer functionality that existed in Oracle Database 12c Release 1 (12.1.0.2).

See Also:

- "Managing SQL Plan Baselines"
- *Oracle Database Reference* to learn about optimizer features enabled when you set `OPTIMIZER_FEATURES_ENABLE` to different release values

19.2.3 Choosing an Optimizer Goal

The **optimizer goal** is the prioritization of resource usage by the optimizer.

Using the `OPTIMIZER_MODE` initialization parameter, you can set the following optimizer goals:

- Best throughput (default)

  When you set the `OPTIMIZER_MODE` value to `ALL_ROWS`, the database uses the least amount of resources necessary to process all rows that the statement accessed.

  For batch applications such as Oracle Reports, optimize for best throughput. Usually, throughput is more important in batch applications because the user is only concerned with the time necessary for the application to complete. Response time is less important because the user does not examine the results of individual statements while the application is running.

- Best response time

  When you set the `OPTIMIZER_MODE` value to `FIRST_ROWS_n`, the database optimizes with a goal of best response time to return the first *n* rows, where *n* equals `1`, `10`, `100`, or `1000`.

  For interactive applications in Oracle Forms or SQL*Plus, optimize for response time. Usually, response time is important because the interactive user is waiting to see the first row or rows that the statement accessed.

Assumptions

This tutorial assumes the following:

- The primary application is interactive, so you want to set the optimizer goal for the database instance to minimize response time.
- For the current session only, you want to run a report and optimize for throughput.

To enable query optimizer features for a specific release:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the current optimizer mode.

   For example, run the following SQL*Plus command:

   ```
   dba1@PROD> SHOW PARAMETER OPTIMIZER_MODE

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- --------
   optimizer_mode                       string      ALL_ROWS
   ```

2. At the instance level, optimize for response time.

For example, run the following SQL statement to configure the system to retrieve the first 10 rows as quickly as possible:

```
SQL> ALTER SYSTEM SET OPTIMIZER_MODE='FIRST_ROWS_10';
```

3. At the session level only, optimize for throughput before running a report.

   For example, run the following SQL statement to configure only this session to optimize for throughput:

```
SQL> ALTER SESSION SET OPTIMIZER_MODE='ALL_ROWS';
```

See Also:

Oracle Database Reference to learn about the `OPTIMIZER_MODE` initialization parameter

19.2.4 Controlling Adaptive Optimization

In Oracle Database, **adaptive query optimization** is the process by which the optimizer adapts an execution plan based on statistics collected at run time.

Adaptive plans are enabled when the following initialization parameters are set:

- `OPTIMIZER_ADAPTIVE_PLANS` is `TRUE` (default)
- `OPTIMIZER_FEATURES_ENABLE` is `12.1.0.1` or later
- `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` is `FALSE` (default)

If `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` is set to `true`, then adaptive optimization runs in reporting-only mode. In this case, the database gathers information required for adaptive optimization, but does not change the plans. An adaptive plan always chooses the default plan, but the database collects information about the execution *as if* the parameter were set to `false`.

Adaptive statistics are enabled when the following initialization parameters are set:

- `OPTIMIZER_ADAPTIVE_STATISTICS` is `TRUE` (the default is `FALSE`)
- `OPTIMIZER_FEATURES_ENABLE` is `12.1.0.1` or later

Assumptions

This tutorial assumes the following:

- The `OPTIMIZER_FEATURES_ENABLE` initialization parameter is set to `12.1.0.1` or later.
- The `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` initialization parameter is set to `false` (default).
- You want to disable adaptive plans for testing purposes so that the database generates only reports.

To disable adaptive plans:

1. Connect SQL*Plus to the database as `SYSTEM`, and then query the current settings.

   For example, run the following SQL*Plus command:

   ```
   SHOW PARAMETER OPTIMIZER_ADAPTIVE_REPORTING_ONLY

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   optimizer_adaptive_reporting_only    boolean     FALSE
   ```

2. At the session level, set the `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` initialization parameter to `true`.

   For example, in SQL*Plus run the following SQL statement:

   ```
   ALTER SESSION SET OPTIMIZER_ADAPTIVE_REPORTING_ONLY=true;
   ```

3. Run a query.
4. Run `DBMS_XPLAN.DISPLAY_CURSOR` with the `+REPORT` parameter.

   When the `+REPORT` parameter is set, the report shows the plan the optimizer would have picked if automatic reoptimization had been enabled.

See Also:

- "About Adaptive Query Optimization"
- *Oracle Database Reference* to learn about the `OPTIMIZER_ADAPTIVE_REPORTING_ONLY` initialization parameter
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `+REPORT` parameter of the `DBMS_XPLAN.DISPLAY_CURSOR` function

19.3 Influencing the Optimizer with Hints

Optimizer hints are special comments in a SQL statement that pass instructions to the optimizer.

The optimizer uses hints to choose an execution plan for the statement unless prevented by some condition.

Note:

Oracle Database SQL Language Reference contains a complete reference for all SQL hints

19.3.1 About Optimizer Hints

A hint is embedded within a SQL comment.

The hint comment must immediately follow the first keyword of a SQL statement block. You can use either style of comment: a slash-star (`/*`) or pair of dashes (`--`). The plus-sign (`+`) hint delimiter must immediately follow the comment delimiter, with no space permitted before the plus sign, as in the following fragment:

```
SELECT /*+ hint_text */ ...
```

The space after the plus sign is optional. A statement block can have only one comment containing hints, but it can contain many space-separated hints. Separate multiple hints by at least one space, as in the following statement:

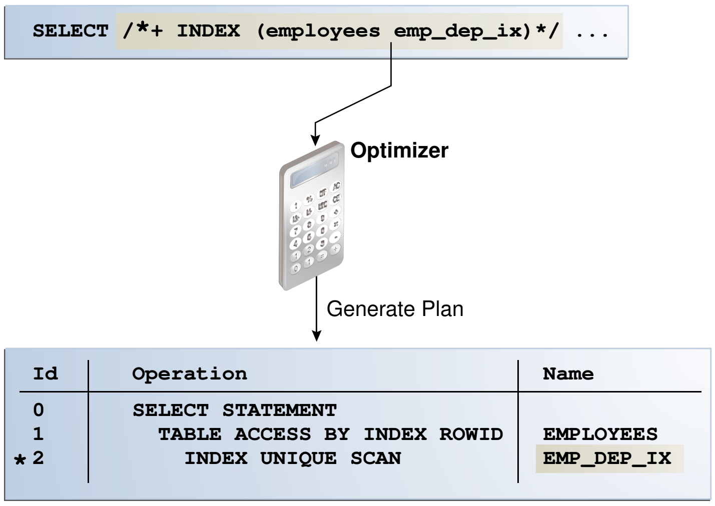
```
SELECT /*+ FULL (hr_emp) CACHE(hr_emp) */ last_name FROM employees
hr_emp;
```

19.3.1.1 Purpose of Hints

Hints enable you to make decisions normally made by the optimizer.

You can use hints to influence the optimizer mode, query transformation, access path, join order, and join methods. In a test environment, hints are useful for testing the performance of a specific access path. For example, you may know that an index is more selective for certain queries, leading to a better plan. The following figure shows how you can use a hint to tell the optimizer to use a specific index for a specific statement.

Figure 19-2 Optimizer Hint



```
SELECT /*+ INDEX (employees emp_dep_ix)*/ ...
```

```
  Id   Operation                          Name
   0   SELECT STATEMENT
   1     TABLE ACCESS BY INDEX ROWID      EMPLOYEES
 * 2       INDEX UNIQUE SCAN              EMP_DEP_IX
```

The disadvantage of hints is the extra code to manage, check, and control. Hints were introduced in Oracle7, when users had little recourse if the optimizer generated suboptimal plans. Because changes in the database and host environment can make hints obsolete or have negative consequences, a good practice is to test using hints, but use other techniques to manage execution plans.

Oracle provides several tools, including SQL Tuning Advisor, SQL plan management, and SQL Performance Analyzer, to address performance problems not solved by the optimizer. Oracle strongly recommends that you use these tools instead of hints because they provide fresh solutions as the data and database environment change.

19.3.1.2 Types of Hints

You can use hints for tables, query blocks, and statements.

Hints fall into the following types:

- Single-table

  Single-table hints are specified on one table or view. `INDEX` and `USE_NL` are examples of single-table hints. The following statement uses a single-table hint:

  ```
  SELECT /*+ INDEX (employees emp_department_ix)*/ employee_id,
  department_id
  FROM   employees
  WHERE  department_id > 50;
  ```

- Multitable

  Multitable hints are like single-table hints except that the hint can specify multiple tables or views. `LEADING` is an example of a multitable hint. The following statement uses a multitable hint:

  ```
  SELECT /*+ LEADING(e j) */ *
  FROM   employees e, departments d, job_history j
  WHERE  e.department_id = d.department_id
  AND    e.hire_date = j.start_date;
  ```

  Note:

  `USE_NL(table1 table2)` is not considered a multitable hint because it is a shortcut for `USE_NL(table1)` and `USE_NL(table2)`.

- Query block

  Query block hints operate on single query blocks. `STAR_TRANSFORMATION` and `UNNEST` are examples of query block hints. The following statement uses a query block hint to specify that the `FULL` hint applies only to the query block that references `employees`:

  ```
  SELECT /*+ INDEX(t1) FULL(@sel$2 t1) */ COUNT(*)
  FROM   jobs t1
  WHERE t1.job_id IN (SELECT job_id FROM employees t1);
  ```

- Statement

Statement hints apply to the entire SQL statement. `ALL_ROWS` is an example of a statement hint. The following statement uses a statement hint:

```
SELECT /*+ ALL_ROWS */ * FROM sales;
```

See Also:

Oracle Database SQL Language Reference for the most common hints by functional category.

19.3.1.3 Scope of Hints

When you specify a hint in a statement block, the hint applies to the appropriate query block, table, or entire statement in the statement block. The hint overrides any instance-level or session-level parameters.

A **statement block** is one of the following:

- A simple `MERGE`, `SELECT`, `INSERT`, `UPDATE`, or `DELETE` statement
- A parent statement or a subquery of a complex statement
- A part of a query using set operators (`UNION`, `MINUS`, `INTERSECT`)

Example 19-1 Query Using a Set Operator

The following query consists of two component queries and the `UNION` operator:

```
SELECT /*+ FIRST_ROWS(10) */ prod_id, time_id FROM 2010_sales
UNION ALL
SELECT /*+ ALL_ROWS */ prod_id, time_id FROM current_year_sales;
```

The preceding statement has two blocks, one for each component query. Hints in the first component query apply only to its optimization, not to the optimization of the second component query. For example, in the first week of 2015 you query current year and last year sales. You apply `FIRST_ROWS(10)` to the query of last year's (2014) sales and the `ALL_ROWS` hint to the query of this year's (2015) sales.

See Also:

Oracle Database SQL Language Reference for an overview of hints

19.3.2 Guidelines for Join Order Hints

In some cases, you can specify join order hints in a SQL statement so that it does not access rows that have no effect on the result.

The driving table in a join is the table to which other tables are joined. In general, the driving table contains the filter condition that eliminates the highest percentage of rows in the table. The **join order** can have a significant effect on the performance of a SQL statement.

Consider the following guidelines:

- Avoid a full table scan when an index retrieves the requested rows more efficiently.
- Avoid using an index that fetches many rows from the driving table when you can use a different index that fetches a small number of rows.
- Choose the join order so that you join fewer rows to tables later in the join order.

The following example shows how to tune join order effectively:

```
SELECT *
FROM   taba a,
       tabb b,
       tabc c
WHERE  a.acol BETWEEN   100 AND   200
AND    b.bcol BETWEEN 10000 AND 20000
AND    c.ccol BETWEEN 10000 AND 20000
AND    a.key1 = b.key1
AND    a.key2 = c.key2;
```

1. Choose the driving table and the driving index (if any).

   Each of the first three conditions in the previous example is a filter condition that applies to a single table. The last two conditions are join conditions.

   Filter conditions dominate the choice of driving table and index. In general, the driving table contains the filter condition that eliminates the highest percentage of rows. Because the range of 100 to 200 is narrow compared with the range of `acol`, but the ranges of 10000 and 20000 are relatively large, `taba` is the driving table, all else being equal.

   With nested loops joins, the joins occur through the join indexes, which are the indexes on the primary or foreign keys used to connect that table to an earlier table in the join tree. Rarely do you use the indexes on the non-join conditions, except for the driving table. Thus, after `taba` is chosen as the driving table, use the indexes on `b.key1` and `c.key2` to drive into `tabb` and `tabc`, respectively.

2. Choose the best join order, driving to the best unused filters earliest.

   You can reduce the work of the following join by first joining to the table with the best still-unused filter. Therefore, if `bcol BETWEEN` is more restrictive (rejects a higher percentage of the rows) than `ccol BETWEEN`, then the last join becomes easier (with fewer rows) if `tabb` is joined before `tabc`.

3. You can use the `ORDERED` or `STAR` hint to force the join order.

See Also:

Oracle Database Reference to learn about `OPTIMIZER_MODE`

19.3.3 Reporting on Hints

An explained plan includes a report showing which hints were used during plan generation.

19.3.3.1 Purpose of Hint Usage Reports

In releases before Oracle Database 19c, it could be difficult to determine why the optimizer did not use hints. The hint usage report solves this problem.

The optimizer uses the instructions encoded in hints to choose an execution plan for a statement, unless a condition prevents the optimizer from using the hint. The database does not issue error messages for hints that it ignores. The hint report shows which hints were used and ignored, and typically explains why hints were ignored. The most common reasons for ignoring hints are as follows:

- Syntax errors

  A hint can contain a typo or an invalid argument. If multiple hints appear in the same hint block, and if one hint has a syntax error, then the optimizer honors all hints before the hint with an error and ignores hints that appear afterward. For example, in the hint specification `/*+ INDEX(t1) FULL(t2) MERG(v) USE_NL(t2) */`, `MERG(v)` has a syntax error. The optimizer honors `INDEX(t1)` and `FULL(t2)`, but ignores `MERG(v)` and `USE_NL(t2)`. The hint usage report lists `MERG(v)` as having an error, but does not list `USE_NL(t2)` because it is not parsed.

- Unresolved hints

  An unresolved hint is invalid for a reason other than a syntax error. For example, a statement specifies `INDEX(employees emp_idx)`, where `emp_idx` is not a valid index name for table `employees`.

- Conflicting hints

  The database ignores combinations of conflicting hints, even if these hints are correctly specified. For example, a statement specifies `FULL(employees) INDEX(employees)`, but an index scan and full table scan are mutually exclusive. In most cases, the optimizer ignores both conflicting hints.

- Hints affected by transformations

  A transformation can make some hints invalid. For example, a statement specifies `PUSH_PRED(some_view) MERGE(some_view)`. When `some_view` merges into its containing query block, the optimizer cannot apply the `PUSH_PRED` hint because `some_view` is unavailable.

See Also:

Oracle Database SQL Language Reference to learn about the syntax rules for comments and hints

19.3.3.2 User Interface for Hint Usage Reports

The report includes the status of all optimizer hints. A subset of other hints, including `PARALLEL` and `INMEMORY`, are also included.

Report Access

Hint tracking is enabled by default. You can access the hint usage report by using the following `DBMS_XPLAN` functions:

- `DISPLAY`
- `DISPLAY_CURSOR`
- `DISPLAY_WORKLOAD_REPOSITORY`
- `DISPLAY_SQL_PLAN_BASELINE`
- `DISPLAY_SQLSET`

The preceding functions generate a report when you specify the value `HINT_REPORT` in the `format` parameter. The value `TYPICAL` displays only the hints that are not used in the final plan, whereas the value `ALL` displays both used and unused hints.

Report Format

Suppose that you explain the following hinted query:

```
SELECT /*+ INDEX(t1) FULL(@sel$2 t1) */ COUNT(*)
FROM   jobs t1
WHERE t1.job_id IN (SELECT /*+ FULL(t1) */ job_id FROM employees t1);
```

The following output of `DBMS_XPLAN.DISPLAY` shows the plan, including the hint report:

```
---------------------------------------------------------------------------
| Id| Operation            | Name      | Rows  | Bytes | Cost (%CPU)| Time|
---------------------------------------------------------------------------
| 0 | SELECT STATEMENT     |           |   1 |    17 |  3   (34)| 00:00:01 |
| 1 |  SORT AGGREGATE      |           |   1 |    17 |           |          |
| 2 |   NESTED LOOPS       |           |  19 |   323 |  3   (34)| 00:00:01 |
| 3 |    SORT UNIQUE       |           | 107 |   963 |  2    (0)| 00:00:01 |
| 4 |     TABLE ACCESS FULL| EMPLOYEES | 107 |   963 |  2    (0)| 00:00:01 |
|*5 |    INDEX UNIQUE SCAN | JOB_ID_PK |   1 |     8 |  0    (0)| 00:00:01 |
---------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$5DA710D3
   4 - SEL$5DA710D3 / "T1"@"SEL$2"
   5 - SEL$5DA710D3 / "T1"@"SEL$1"

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("T1"."JOB_ID"="JOB_ID")

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) COUNT(*)[22]
   2 - (#keys=0)
   3 - (#keys=1) "JOB_ID"[VARCHAR2,10]
   4 - (rowset=256) "JOB_ID"[VARCHAR2,10]

Hint Report (identified by operation id / Query Block Name / Object Alias):
```

```
Total hints for statement: 3 (U - Unused (1))
---------------------------------------------------------------------------

   4 -  SEL$5DA710D3 / "T1"@"SEL$2"
          U -  FULL(t1) / hint overridden by another in parent query block
            -  FULL(@sel$2 t1)

   5 -  SEL$5DA710D3 / "T1"@"SEL$1"
            -  INDEX(t1)
```

The report header shows the total number of hints in the report. In this case, the statement contained 3 total hints. If hints are unused, unresolved, or have syntax errors, then the header specifies their number. In this case, only 1 hint was unused.

The report displays the hints under the objects (for example, query blocks and tables) that appear in the plan. Before each object is a number that identifies the line in the plan where the object first appears. For example, the preceding report shows hints that apply to the following distinct objects: `T1@SEL$2`, and `T1@SEL$1`. The table `T1@SEL$2` appears in query block `SEL$5DA710D3` at line 4 of the plan. The table `T1@SEL$1` appears in the same query block at line 5 of the plan.

Hints can be specified incorrectly or associated with objects that are not present in the final plan. If a query block does not appear in the final plan, then the report assigns it line number `0`. In the preceding example, no hints have line number 0, so all query blocks appeared in the final plan.

The report shows the text of the hints. The hint may also have one of the following annotations:

- `E` indicates a syntax error.
- `N` indicates an unresolved hint.
- `U` indicates that the corresponding hint was not used in the final plan.

In the preceding example, `U - FULL(t1)` indicates that query block `SEL$5DA710D3` appeared in the final plan, but the `FULL(t1)` hint was not applied.

Within each object, unused hints appear at the beginning, followed by used hints. For example, the report first shows the `FULL(t1)` hint, which was not used, and then `FULL(@sel$2 t1)`, which was used. For many unused hints, the report explains why the optimizer did not apply the hints. In the preceding example, the report indicates that `FULL(t1)` was not used for the following reason: `hint overridden by another in parent query block`.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DBMS_XPLAN` package

19.3.3.3 Reporting on Hint Usage: Tutorial

You can use the `DBMS_XPLAN` display functions to report on hint usage.

Hint usage reporting is enabled by default. The steps for displaying a plan with hint information are the same as for displaying a plan normally.

Assumptions

This tutorial assumes the following:

- An index named `emp_emp_id_pk` exists on the `employees.employee_id` column.
- You want to query a specific employee.
- You want to use the `INDEX` hint to force the optimizer to use `emp_emp_id_pk`.

To report on hint usage:

1. Start SQL*Plus or SQL Developer, and log in to the database as user `hr`.
2. Explain the plan for the query of `employees`.

   For example, enter the following statement:

   ```
   EXPLAIN PLAN FOR
     SELECT /*+ INDEX(e emp_emp_id_pk) */ COUNT(*)
     FROM   employees e
     WHERE  e.employee_id = 5;
   ```

3. Query the plan table using a display function.

   You can specify any of the following values in the `format` parameter:

   - `ALL`
   - `TYPICAL`

   The following query displays all sections of the plan, including the hint usage information (sample output included):

   ```
   SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));

   PLAN_TABLE_OUTPUT
   ----------------------------------------------------------------------

   Plan hash value: 2637910222
   ----------------------------------------------------------------------
   |Id | Operation          | Name       |Rows|Bytes | Cost (%CPU)| Time|
   ----------------------------------------------------------------------
   | 0 | SELECT STATEMENT   |                | 1 | 4 | 0   (0)| 00:00:01 |
   | 1 |  SORT AGGREGATE    |                | 1 | 4 |        |          |
   |*2 |   INDEX UNIQUE SCAN| EMP_EMP_ID_PK  | 1 | 4 | 0   (0)| 00:00:01 |
   ----------------------------------------------------------------------

   Query Block Name / Object Alias (identified by operation id):
   ----------------------------------------------------------------------
   ```

```
   1 - SEL$1
   2 - SEL$1 / E@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("E"."EMPLOYEE_ID"=5)

Column Projection Information (identified by operation id):
--------------------------------------------------------------------
--

   1 - (#keys=0) COUNT(*)[22]

Hint Report (identified by operation id/Query Block Name/Object
Alias)
Total hints for statement: 1
--------------------------------------------------------------------
--

   2 -  SEL$1 / E@SEL$1
            -  INDEX(e emp_emp_id_pk)
```

The Hint Report section shows that the query block for the `INDEX(e emp_emp_id_pk)` hint is `SEL$1`. The table identifier is `E@SEL$1`. The line number of the plan line is `2`, which corresponds to the first line where the table `E@SEL$1` appears in the plan table.

See Also:

Oracle Database SQL Language Reference to learn more about `EXPLAIN PLAN`

19.3.3.4 Hint Usage Reports: Examples

These examples show various types of hint usage reports.

The following examples all show queries of tables in the `hr` schema.

Example 19-2 Statement-Level Unused Hint

The following example specifies an index range hint for the `emp_manager_ix` index:

```
EXPLAIN PLAN FOR
  SELECT /*+ INDEX_RS(e emp_manager_ix) */ COUNT(*)
  FROM   employees e
  WHERE  e.job_id < 5;
```

The following query of the plan table specifies the `format` value of `TYPICAL`, which shows only unused hints:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'TYPICAL'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 2731009358

---------------------------------------------------------------------------
| Id  | Operation        | Name       | Rows  | Bytes| Cost  (%CPU)| Time   |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT |            |    1 |    9  |    1    (0)| 00:00:01 |
|   1 |  SORT AGGREGATE  |            |    1 |     9 |            |          |
|*  2 |   INDEX FULL SCAN| EMP_JOB_IX |    5 |   45 |    1    (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

   2 - filter(TO_NUMBER("E"."JOB_ID")<5)

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 1 (U - Unused (1))
---------------------------------------------------------------------------

   2 -  SEL$1 / E@SEL$1
         U -  INDEX_RS(e emp_manager_ix)
```

The `U` in the preceding hint usage report indicates that the `INDEX_RS` hint was not used. The report shows the total number of unused hints: `U – Unused (1)`.

Example 19-3 Conflicting Hints

The following example specifies two hints, one for a skip scan and one for a fast full scan:

```
EXPLAIN PLAN FOR
  SELECT /*+ INDEX_SS(e emp_manager_ix) INDEX_FFS(e) */ COUNT(*)
  FROM   employees e
  WHERE  e.manager_id < 5;
```

The following query of the plan table specifies the `format` value of `TYPICAL`, which shows only unused hints:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'TYPICAL'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 2262146496

---------------------------------------------------------------------------
| Id| Operation         | Name           |Rows  |Bytes |Cost (%CPU)|Time   |
-----------------------------------------------------------------------
----
| 0 | SELECT STATEMENT  |                  | 1 |   4 |    1   (0)|
00:00:01 |
| 1 |  SORT AGGREGATE   |                  | 1 |   4 |
|          |
|*2 |   INDEX RANGE SCAN| EMP_MANAGER_IX | 1 |   4 |    1   (0)|
00:00:01 |
-----------------------------------------------------------------------
----

Predicate Information (identified by operation id):

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----

   2 - access("E"."MANAGER_ID"<5)

Hint Report (identified by operation id / Query Block Name / Object
Alias):
Total hints for statement: 2 (U - Unused (2))
-----------------------------------------------------------------------
----

   2 -  SEL$1 / E@SEL$1
    U -  INDEX_FFS(e) / hint conflicts with another in sibling query
block
    U -  INDEX_SS(e emp_manager_ix) / hint conflicts with another in
         sibling query block
```

The preceding report shows that the `INDEX_FFS(e)` and `INDEX_SS(e emp_manager_ix)` hints conflict with one other. Index skip scans and index fast full scans are mutually exclusive. The optimizer ignored both hints, as indicated by the text `U — Unused (2)`. Even though the optimizer ignored the hint specifying the `emp_manager_ix` index, the optimizer used this index anyway based on its cost-based analysis.

Example 19-4   Multitable Hints

The following example specifies four hints, one of which specifies two tables:

```
EXPLAIN PLAN FOR
  SELECT /*+ ORDERED USE_NL(t1, t2) INDEX(t2) NLJ_PREFETCH(t2) */
COUNT(*)
  FROM   jobs t1, employees t2
  WHERE  t1.job_id = t2.employee_id;
```

The following query of the plan table specifies the `format` value of `ALL`:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----
```

```
Plan hash value: 2668549896

---------------------------------------------------------------------------
| Id  | Operation            | Name           |Rows |Bytes |Cost (%CPU)|Time|
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                |  1 |  12 | 1   (0)| 00:00:01 |
|   1 |  SORT AGGREGATE      |                |  1 |  12 |        |          |
|   2 |   NESTED LOOPS       |                | 19 | 228 | 1   (0)| 00:00:01 |
|   3 |    INDEX FULL SCAN   | JOB_ID_PK      | 19 | 152 | 1   (0)| 00:00:01 |
|*  4 |    INDEX UNIQUE SCAN| EMP_EMP_ID_PK |  1 |    4 | 0   (0)| 00:00:01 |
---------------------------------------------------------------------------

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$1
   3 - SEL$1 / T1@SEL$1
   4 - SEL$1 / T2@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   4 - access("T2"."EMPLOYEE_ID"=TO_NUMBER("T1"."JOB_ID"))

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) COUNT(*)[22]
   2 - (#keys=0)
   3 - "T1"."JOB_ID"[VARCHAR2,10]

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 5 (U - Unused (2))
---------------------------------------------------------------------------

   1 -  SEL$1
          -  ORDERED

   3 -  SEL$1 / T1@SEL$1
         U -  USE_NL(t1, t2)

   4 -  SEL$1 / T2@SEL$1
         U -  NLJ_PREFETCH(t2)
          -  INDEX(t2)
          -  USE_NL(t1, t2)
```

The preceding report shows that two hints were not used: `USE_NL(t1, t2)` and `NLJ_PREFETCH(t2)`. Step 3 of the plan is an index full scan of the `jobs` table, which uses the alias `t1`. The report shows that the optimizer did not apply the `USE_NL(t1, t2)` hint for the access of `jobs`. Step 4 is an index unique scan of the `employees` table, which uses the alias t2. No U prefix exists for USE_NL(t1, t2), which means that the optimizer did use the hint for employees.

Example 19-5 Hints for Unused Query Blocks

The following example specifies two hints, UNNEST and SEMIJOIN, on a subquery:

```
EXPLAIN PLAN FOR
  SELECT COUNT(*), manager_id
  FROM   departments
  WHERE  manager_id IN (SELECT /*+ UNNEST SEMIJOIN */ manager_id FROM
employees)
  AND    ROWNUM <= 2
GROUP BY manager_id;
```

The following query of the plan table specifies the format value of ALL:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----
Plan hash value: 173733304
-----------------------------------------------------------------------
----
| Id  |Operation              | Name          |Rows  |Bytes |Cost (%CPU)|
Time|
-----------------------------------------------------------------------
----
| 0 | SELECT STATEMENT      |                |  2 | 14 | 3  (34)|
00:00:01 |
| 1 |  HASH GROUP BY        |                |  2 | 14 | 3  (34)|
00:00:01 |
|*2 |   COUNT STOPKEY       |                |    |    |
|          |
| 3 |    NESTED LOOPS SEMI  |                |  2 | 14 | 2   (0)|
00:00:01 |
|*4 |     TABLE ACCESS FULL| DEPARTMENTS    |  2 |  6 | 2   (0)|
00:00:01 |
|*5 |     INDEX RANGE SCAN | EMP_MANAGER_IX | 107|428 | 0   (0)|
00:00:01 |
-----------------------------------------------------------------------
----

PLAN_TABLE_OUTPUT
-----------------------------------------------------------------------
----

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$5DA710D3
   4 - SEL$5DA710D3 / DEPARTMENTS@SEL$1
   5 - SEL$5DA710D3 / EMPLOYEES@SEL$2
```

```
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter(ROWNUM<=2)
   4 - filter("MANAGER_ID" IS NOT NULL)
   5 - access("MANAGER_ID"="MANAGER_ID")

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=1) "MANAGER_ID"[NUMBER,22], COUNT(*)[22]
   2 - "MANAGER_ID"[NUMBER,22]
   3 - (#keys=0) "MANAGER_ID"[NUMBER,22]
   4 - "MANAGER_ID"[NUMBER,22]

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 2
---------------------------------------------------------------------------

   0 -  SEL$2
          -  SEMIJOIN
          -  UNNEST
```

In this example, the hints are specified in query block `SEL$2`, but `SEL$2` does not appear in the final plan. The report displays the hints under `SEL$2` with an associated line number of `0`.

Example 19-6 Overridden Hints

The following example specifies two `FULL` hints on the same table in the same query block:

```
EXPLAIN PLAN FOR
  SELECT /*+ INDEX(t1) FULL(@sel$2 t1) */ COUNT(*)
  FROM   jobs t1
  WHERE t1.job_id IN (SELECT /*+ FULL(t1) NO_MERGE */ job_id FROM employees 
t1);
```

The following query of the plan table specifies the `format` value of `ALL`:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(format => 'ALL'));

PLAN_TABLE_OUTPUT
---------------------------------------------------------------------------
Plan hash value: 3101158531

---------------------------------------------------------------------------
| Id  | Operation             | Name      |Rows  |Bytes |Cost (%CPU)|Time   |
---------------------------------------------------------------------------
|   0 | SELECT STATEMENT      |           |  1 |   17 |   3  (34)| 00:00:01 |
|   1 |  SORT AGGREGATE       |           |  1 |   17 |          |          |
|   2 |   NESTED LOOPS        |           | 19 |  323 |   3  (34)| 00:00:01 |
|   3 |    SORT UNIQUE        |           |107 |  963 |   2   (0)| 00:00:01 |
|   4 |     TABLE ACCESS FULL| EMPLOYEES |107 |  963 |   2   (0)| 00:00:01 |
|*  5 |    INDEX UNIQUE SCAN | JOB_ID_PK |  1 |    8 |   0   (0)| 00:00:01 |
---------------------------------------------------------------------------
```

```
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SEL$5DA710D3
   4 - SEL$5DA710D3 / T1@SEL$2
   5 - SEL$5DA710D3 / T1@SEL$1

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("T1"."JOB_ID"="JOB_ID")

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - (#keys=0) COUNT(*)[22]
   2 - (#keys=0)
   3 - (#keys=1) "JOB_ID"[VARCHAR2,10]
   4 - (rowset=256) "JOB_ID"[VARCHAR2,10]

Hint Report (identified by operation id / Query Block Name / Object
Alias):
Total hints for statement: 4 (U - Unused (1))
-----------------------------------------------------------------------
----

   0 -  SEL$2
          -  NO_MERGE

   4 -  SEL$5DA710D3 / T1@SEL$2
         U -  FULL(t1) / hint overridden by another in parent query
block
          -  FULL(@sel$2 t1)

   5 -  SEL$5DA710D3 / T1@SEL$1
          -  INDEX(t1)
```

Of the three hints specified, only one was unused. The hint `FULL(t1)` specified in query block `SEL$2` was overridden by the hint `FULL(@sel$2 T1)` specified in query block `SEL$1`. The `NO_MERGE` hint in query block `SEL$2` was used.

The following query of the plan table using the format setting of `TYPICAL` shows only unused hints:

```
SQL> select * from table(dbms_xplan.display(format => 'TYPICAL'));
Plan hash value: 3101158531

-----------------------------------------------------------------------
----
| Id | Operation            | Name      |Rows  |Bytes |Cost (%CPU)|
Time   |
-----------------------------------------------------------------------
----
```

```
|  0 | SELECT STATEMENT     |            |   1 |   17 |  3  (34)| 00:00:01 |
|  1 |  SORT AGGREGATE      |            |   1 |   17 |          |          |
|  2 |   NESTED LOOPS       |            |  19 |  323 |  3  (34)| 00:00:01 |
|  3 |    SORT UNIQUE       |            | 107 |  963 |  2   (0)| 00:00:01 |
|  4 |     TABLE ACCESS FULL| EMPLOYEES  | 107 |  963 |  2   (0)| 00:00:01 |
|* 5 |    INDEX UNIQUE SCAN | JOB_ID_PK  |   1 |    8 |  0   (0)| 00:00:01 |
---------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   5 - access("T1"."JOB_ID"="JOB_ID")

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 1 (U - Unused (1))
---------------------------------------------------------------------------

   4 -  SEL$5DA710D3 / T1@SEL$2
         U -  FULL(t1) / hint overridden by another in parent query block
```

Example 19-7   Multiple Hints

The following `UNION ALL` query specifies ten different hints:

```
SELECT /*+ FULL(t3) INDEX(t2) INDEX(t1) MERGE(@SEL$5) PARALLEL(2) */
t1.first_name
FROM   employees t1, jobs t2, job_history t3
WHERE  t1.job_id = t2.job_id
AND    t2.min_salary = 100000
AND    t1.department_id = t3.department_id
UNION ALL
SELECT /*+ INDEX(t3) USE_MERGE(t2) INDEX(t2) FULL(t1) NO_ORDER_SUBQ */
t1.first_name
FROM   departments t3, jobs t2, employees t1
WHERE  t1.job_id = t2.job_id
AND    t2.min_salary = 100000
AND    t1.department_id = t3.department_id;
```

The following query of the shared SQL area specifies the `format` value of `ALL` (note that the plan lines have been truncated for readability):

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR(format => 'ALL'))

...

---------------------------------------------------------------------------------------------
| Id  | Operation                                     | Name       |Rows  |Bytes |Cost (%CPU)|
---------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                              |             |      |      |  9 (100)|
|   1 |  UNION-ALL                                    |             |      |      |         |
|   2 |   PX COORDINATOR                              |             |      |      |         |
|   3 |    PX SEND QC (RANDOM)                        | :TQ10002    |    5 |  175 |  5   (0)|
|*  4 |     HASH JOIN                                 |             |    5 |  175 |  5   (0)|
|   5 |      PX RECEIVE                               |             |    3 |   93 |  3   (0)|
|   6 |       PX SEND BROADCAST                     | :TQ10001    |     3 |    93 |
3   (0)|
|   7 |        NESTED LOOPS                         |             |     3 |    93 |
3   (0)|
|   8 |         NESTED LOOPS                        |             |     6 |    93 |
3   (0)|
|*  9 |          TABLE ACCESS BY INDEX ROWID BATCHED| JOBS        |     1 |    12 |
2   (0)|
|  10 |           BUFFER SORT                       |             |       |
|         |
|  11 |            PX RECEIVE                       |             |    19 |       |
1   (0)|
|  12 |             PX SEND HASH (BLOCK ADDRESS)    | :TQ10000    |    19 |       |
1   (0)|
|  13 |              PX SELECTOR                    |             |       |
|         |
|  14 |               INDEX FULL SCAN               | JOB_ID_PK   |    19 |       |
1   (0)|
|* 15 |          INDEX RANGE SCAN                   | EMP_JOB_IX  |     6 |       |
0   (0)|
|  16 |         TABLE ACCESS BY INDEX ROWID         | EMPLOYEES   |     6 |   114 |
1   (0)|
|  17 |      PX BLOCK ITERATOR                      |             |    10 |    40 |
2   (0)|
|* 18 |       TABLE ACCESS FULL                     | JOB_HISTORY |    10 |    40 |
2   (0)|
|  19 |   PX COORDINATOR                            |             |       |
|         |
|  20 |    PX SEND QC (RANDOM)                      | :TQ20002    |     3 |    93 |
4   (0)|
|* 21 |     HASH JOIN                               |             |     3 |    93 |
4   (0)|
|  22 |      JOIN FILTER CREATE                     | :BF0000     |     1 |    12 |
2   (0)|
|  23 |       PX RECEIVE                            |             |     1 |    12 |
2   (0)|
|  24 |        PX SEND BROADCAST                    | :TQ20001    |     1 |    12 |
2   (0)|
|* 25 |         TABLE ACCESS BY INDEX ROWID BATCHED | JOBS        |     1 |    12 |
2   (0)|
|  26 |          BUFFER SORT                        |             |       |
|         |
|  27 |           PX RECEIVE                        |             |    19 |       |
1   (0)|
|  28 |            PX SEND HASH (BLOCK ADDRESS)     | :TQ20000    |    19 |       |
1   (0)|
|  29 |             PX SELECTOR                     |             |       |
|         |
|  30 |              INDEX FULL SCAN                | JOB_ID_PK   |    19 |       |
1   (0)|
|  31 |      JOIN FILTER USE                        | :BF0000     |   106 |  2014 |
2   (0)|
|  32 |       PX BLOCK ITERATOR                     |             |   106 |  2014 |
2   (0)|
|* 33 |        TABLE ACCESS FULL                    | EMPLOYEES   |   106 |  2014 |
2   (0)|
-------------------------------------------------------------------------------------------

Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------

   1 - SET$1
   4 - SEL$1
   9 - SEL$1        / T2@SEL$1
  14 - SEL$1        / T2@SEL$1
  15 - SEL$1        / T1@SEL$1
  16 - SEL$1        / T1@SEL$1
  18 - SEL$1        / T3@SEL$1
  21 - SEL$E0F432AE
  25 - SEL$E0F432AE / T2@SEL$2
  30 - SEL$E0F432AE / T2@SEL$2
  33 - SEL$E0F432AE / T1@SEL$2

Predicate Information (identified by operation id):
-----------------------------------------------------------

   4 - access("T1"."DEPARTMENT_ID"="T3"."DEPARTMENT_ID")
   9 - filter("T2"."MIN_SALARY"=100000)
  15 - access("T1"."JOB_ID"="T2"."JOB_ID")
  18 - access(:Z>=:Z AND :Z<=:Z)
  21 - access("T1"."JOB_ID"="T2"."JOB_ID")
  25 - filter("T2"."MIN_SALARY"=100000)
  33 - access(:Z>=:Z AND :Z<=:Z)
       filter(("T1"."DEPARTMENT_ID" IS NOT NULL AND
         SYS_OP_BLOOM_FILTER(:BF0000,"T1"."JOB_ID")))

Column Projection Information (identified by operation id):
-----------------------------------------------------------

   1 - STRDEF[20]
   2 - "T1"."FIRST_NAME"[VARCHAR2,20]
   3 - (#keys=0) "T1"."FIRST_NAME"[VARCHAR2,20]
   4 - (#keys=1; rowset=256) "T1"."FIRST_NAME"[VARCHAR2,20]
   5 - (rowset=256) "T1"."DEPARTMENT_ID"[NUMBER,22], "T1"."FIRST_NAME"[VARCHAR2,20]
   6 - (#keys=0) "T1"."DEPARTMENT_ID"[NUMBER,22], "T1"."FIRST_NAME"[VARCHAR2,20]
   7 - "T1"."FIRST_NAME"[VARCHAR2,20], "T1"."DEPARTMENT_ID"[NUMBER,22]
   8 - "T1".ROWID[ROWID,10]
   9 - "T2"."JOB_ID"[VARCHAR2,10]
  10 - (#keys=0) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  11 - (rowset=256) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  12 - (#keys=1) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  13 - "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  14 - "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  15 - "T1".ROWID[ROWID,10]
  16 - "T1"."FIRST_NAME"[VARCHAR2,20], "T1"."DEPARTMENT_ID"[NUMBER,22]
  17 - (rowset=256) "T3"."DEPARTMENT_ID"[NUMBER,22]
  18 - (rowset=256) "T3"."DEPARTMENT_ID"[NUMBER,22]
  19 - "T1"."FIRST_NAME"[VARCHAR2,20]
  20 - (#keys=0) "T1"."FIRST_NAME"[VARCHAR2,20]
  21 - (#keys=1; rowset=256) "T1"."FIRST_NAME"[VARCHAR2,20]
  22 - (rowset=256) "T2"."JOB_ID"[VARCHAR2,10]
  23 - (rowset=256) "T2"."JOB_ID"[VARCHAR2,10]
  24 - (#keys=0) "T2"."JOB_ID"[VARCHAR2,10]
  25 - "T2"."JOB_ID"[VARCHAR2,10]
  26 - (#keys=0) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  27 - (rowset=256) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  28 - (#keys=1) "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  29 - "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  30 - "T2".ROWID[ROWID,10], "T2"."JOB_ID"[VARCHAR2,10]
  31 - (rowset=256) "T1"."FIRST_NAME"[VARCHAR2,20], "T1"."JOB_ID"[VARCHAR2,10]
  32 - (rowset=256) "T1"."FIRST_NAME"[VARCHAR2,20], "T1"."JOB_ID"[VARCHAR2,10]
  33 - (rowset=256) "T1"."FIRST_NAME"[VARCHAR2,20], "T1"."JOB_ID"[VARCHAR2,10]

Hint Report (identified by operation id / Query Block Name / Object Alias):
Total hints for statement: 10 (U - Unused (2), N - Unresolved (1), E - Syntax error 
(1))
----------------------------------------------------------------------------------------
--

   0 -  STATEMENT
          -  PARALLEL(2)

   0 -  SEL$5
         N -  MERGE(@SEL$5)

   0 -  SEL$2
         E -  NO_ORDER_SUBQ

   9 -  SEL$1 / T2@SEL$1
          -  INDEX(t2)

  15 -  SEL$1 / T1@SEL$1
          -  INDEX(t1)

  18 -  SEL$1 / T3@SEL$1
          -  FULL(t3)

  21 -  SEL$E0F432AE / T3@SEL$2
         U -  INDEX(t3)

  25 -  SEL$E0F432AE / T2@SEL$2
         U -  USE_MERGE(t2)
          -  INDEX(t2)

  33 -  SEL$E0F432AE / T1@SEL$2
          -  FULL(t1)

Note
-----
   - Degree of Parallelism is 2 because of hint
```

The report indicates the following unused hints:

- Two unused hints (`U`)

The report indicates that `INDEX(t3)` and `USE_MERGE(t2)` were not used in query block `SEL$E0F432AE`.

- One unresolved hint (`N`)

  The hint `MERGE` is unresolved because the query block `SEL$5` does not exist.

- One syntax error (`E`)

  The hint `NO_ORDER_SUBQ` specified in `SEL$2` is not a valid hint.

20 Improving Real-World Performance Through Cursor Sharing

Cursor sharing can improve database application performance by orders of magnitude.

20.1 Overview of Cursor Sharing

Oracle Database can share cursors, which are pointers to private SQL areas in the shared pool.

20.1.1 About Cursors

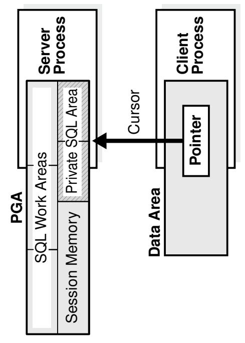
A **private SQL area** holds information about a parsed SQL statement and other session-specific information for processing.

When a server process executes SQL or PL/SQL code, the process uses the private SQL area to store bind variable values, query execution state information, and query execution work areas. The private SQL areas for each execution of a statement are not shared and may contain different values and data.

A cursor is a name or handle to a specific private SQL area. The cursor contains session-specific state information such as bind variable values and result sets.

As shown in the following graphic, you can think of a cursor as a pointer on the client side and as a state on the server side. Because cursors are closely associated with private SQL areas, the terms are sometimes used interchangeably.

Figure 20-1 Cursor



20.1.1.1 Private and Shared SQL Areas

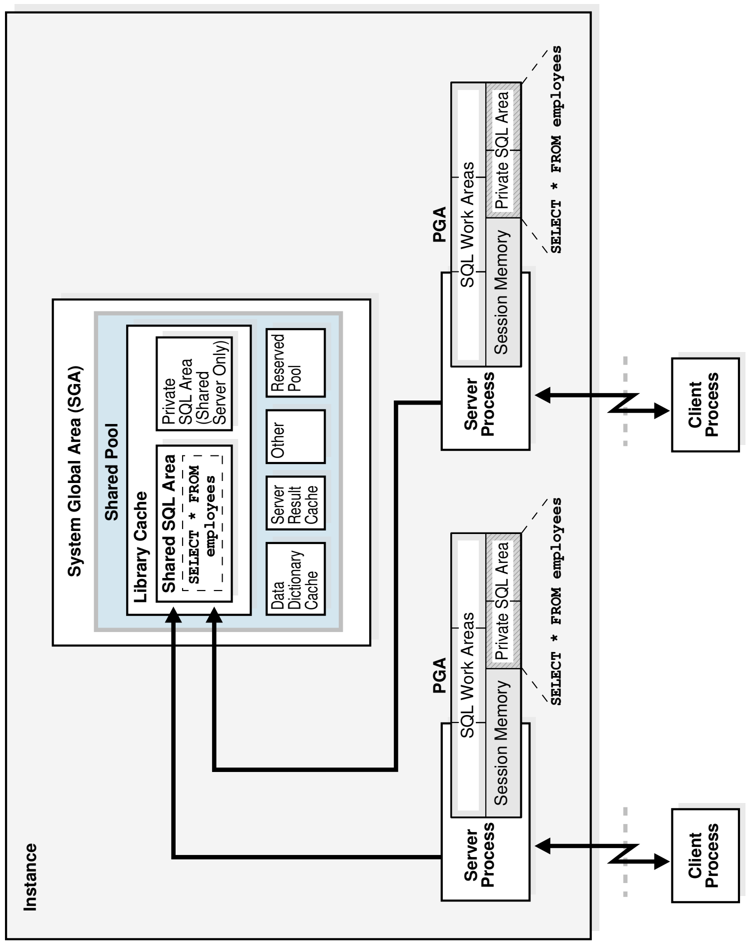
A cursor in the private SQL area points to a **shared SQL area** in the library cache.

Unlike the private SQL area, which contains session state information, the shared SQL area contains the parse tree and execution plan for the statement. For example, an execution of

`SELECT * FROM employees` has a plan and parse tree stored in one shared SQL area. An execution of `SELECT * FROM departments`, which differs both syntactically and semantically, has a plan and parse tree stored in a separate shared SQL area.

Multiple private SQL areas in the same or different sessions can reference a single shared SQL area, a phenomenon known as **cursor sharing**. For example, an execution of `SELECT * FROM employees` in one session and an execution of the `SELECT * FROM employees` (accessing the same table) in a different session can use the same parse tree and plan. A shared SQL area that is accessed by multiple statements is known as a shared cursor.

Figure 20-2 Cursor Sharing



Oracle Database automatically determines whether the SQL statement or PL/SQL block being issued is textually identical to another statement currently in the library cache, using the following steps:

1. The text of the statement is hashed.
2. The database looks for a matching hash value for an existing SQL statement in the shared pool. The following options are possible:
   - No matching hash value exists.

  In this case, the SQL statement does not currently exist in the shared pool, so the database performs a hard parse. This ends the shared pool check.

- A matching hash value exists.

  In this case, the database proceeds to the next step, which is a text match.

3. The database compares the text of the matched statement to the text of the hashed statement to determine whether they are identical. The following options are possible:

   - The textual match fails.

     In this case, the text match process stops, resulting in a hard parse.

   - The textual match succeeds.

     In this case, the database proceeds to the next step: determining whether the SQL can share an existing parent cursor.

     For a textual match to occur, the text of the SQL statements or PL/SQL blocks must be character-for-character identical, including spaces, case, and comments. For example, the following statements cannot use the same shared SQL area:

     ```
     SELECT * FROM employees;
     SELECT * FROM Employees;
     SELECT *  FROM employees;
     ```

     Usually, SQL statements that differ only in literals cannot use the same shared SQL area. For example, the following statements do not resolve to the same SQL area:

     ```
     SELECT count(1) FROM employees WHERE manager_id = 121;
     SELECT count(1) FROM employees WHERE manager_id = 247;
     ```

     The only exception to this rule is when the parameter `CURSOR_SHARING` has been set to `FORCE`, in which case similar statements can share SQL areas.

See Also:

- "Parent and Child Cursors"
- "Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix" to learn about the costs involved in using `CURSOR_SHARING`
- *Oracle Database Reference* to learn more about the `CURSOR_SHARING` initialization parameter

20.1.1.2 Parent and Child Cursors

Every parsed SQL statement has a parent cursor and one or more child cursors.

The parent cursor stores the text of the SQL statement. If the text of two statements is identical, then the statements share the same parent cursor. If the text is different, however, then the database creates a separate parent cursor.

Example 20-1 Parent Cursors

In this example, the first two statements are syntactically different (the letter “c” is lowercase in the first statement and uppercase in the second statement), but semantically identical. Because of the syntactic difference, these statements have different parent cursors. The third statement is syntactically identical to the first statement (lowercase “c”), but semantically different because it refers to a `customers` table in a different schema. Because of the syntactic identity, the third statement can share a parent cursor with the first statement.

```
SQL> CONNECT oe@inst1
Enter password: *******
Connected.
SQL> SELECT COUNT(*) FROM customers;

  COUNT(*)
----------
       319

SQL> SELECT COUNT(*) FROM Customers;

  COUNT(*)
----------
       319

SQL> CONNECT sh@inst1
Enter password:  *******
Connected.
SQL> SELECT COUNT(*) FROM customers;

  COUNT(*)
----------
    155500
```

The following query of `V$SQL` indicates the two parents. The statement with the SQL ID of `8h916vv2yw400`, which is the lowercase “c” version of the statement, has one parent cursor and two child cursors: child 0 and child 1. The statement with the SQL ID of `5rn2uxjtpz0wd`, which is the uppercase “c” version of the statement, has a different parent cursor and only one child cursor: child 0.

```
SQL> CONNECT SYSTEM@inst1
Enter password:  *******
Connected.

SQL> COL SQL_TEXT FORMAT a30
SQL> COL CHILD# FORMAT 99999
SQL> COL EXEC FORMAT 9999
SQL> COL SCHEMA FORMAT a6
SQL> SELECT SQL_ID, PARSING_SCHEMA_NAME AS SCHEMA, SQL_TEXT,
  2  CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC FROM V$SQL
  3  WHERE SQL_TEXT LIKE '%ustom%' AND SQL_TEXT NOT LIKE '%SQL_TEXT%'
ORDER BY SQL_ID;

SQL_ID        SCHEMA SQL_TEXT                        CHILD# EXEC
```

```
------------- ------ ------------------------------ ------ -----
5rn2uxjtpz0wd OE     SELECT COUNT(*) FROM Customers      0     1
8h916vv2yw400 OE     SELECT COUNT(*) FROM customers      0     1
8h916vv2yw400 SH     SELECT COUNT(*) FROM customers      1     1
```

20.1.1.2.1 Parent Cursors and V$SQLAREA

The V$SQLAREA view contains one row for every parent cursor.

In the following example, a query of V$SQLAREA shows two parent cursors, each identified with a different SQL_ID. The VERSION_COUNT indicates the number of child cursors.

```
COL SQL_TEXT FORMAT a30
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                       SQL_ID        VERSION_COUNT HASH_VALUE
------------------------------ ------------- ------------- ----------
SELECT * FROM Employees        5bzhzpaa0wy9m             1 2483976499
SELECT * FROM employees        4959aapufrm1k             2 1961610290
```

In the preceding output, the VERSION_COUNT of 2 for SELECT * FROM employees indicates multiple child cursors, which were necessary because the statement was executed against two different objects. In contrast, the statement SELECT * FROM Employees (note the capital "E") was executed once, and so has one parent cursor, and one child cursor (VERSION_COUNT of 1).

20.1.1.2.2 Child Cursors and V$SQL

Every parent cursor has one or more child cursors.

A **child cursor** contains the execution plan, bind variables, metadata about objects referenced in the query, optimizer environment, and other information. In contrast to the parent cursor, the child cursor does not store the text of the SQL statement.

If a statement is able to reuse a parent cursor, then the database checks whether the statement can reuse an existing child cursor. The database performs several checks, including the following:

- The database compares objects referenced in the issued statement to the objects referenced by the statement in the pool to ensure that they are all identical.

  References to schema objects in the SQL statements or PL/SQL blocks must resolve to the same object in the same schema. For example, if two users issue the following SQL statement, and if each user has its own employees table, then the following statement is not identical because the statement references different employees tables for each user:

  ```
  SELECT * FROM employees;
  ```

> **Note:**
>
> The database can share cursors for a private temporary table, but only within the same session. The database associates the session identifier as part of the cursor context. During a soft parse, the database can share the child cursor only when the current session ID matches with the session ID in the cursor context.

- The database determines whether the optimizer mode is identical.

  For example, SQL statements must be optimized using the same optimizer goal.

Example 20-2 Multiple Child Cursors

V$SQL describes the statements that currently reside in the library cache. It contains one row for every child cursor, as shown in the following example:

```
SELECT SQL_TEXT, SQL_ID, USERNAME AS USR, CHILD_NUMBER AS CHILD#,
       HASH_VALUE, PLAN_HASH_VALUE AS PLAN_HASHV
FROM   V$SQL s, DBA_USERS d
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND    d.USER_ID = s.PARSING_USER_ID;

SQL_TEXT                SQL_ID        USR CHILD# HASH_VALUE PLAN_HASHV
----------------------- ------------- --- ------ ---------- ----------
SELECT * FROM Employees 5bzhzpaa0wy9m  HR      0 2483976499 1445457117
SELECT * FROM employees 4959aapufrm1k  HR      0 1961610290 1445457117
SELECT * FROM employees 4959aapufrm1k  SH      1 1961610290 1445457117
```

In the preceding results, the CHILD# of the bottom two statements is different (0 and 1), even though the SQL_ID is the same. This means that the statements have the same parent cursor, but different child cursors. In contrast, the statement with the SQL_ID of 5bzhzpaa0wy9m has one parent and one child (CHILD# of 0). All three SQL statements use the same execution plan, as indicated by identical values in the PLAN_HASH_VALUE column.

Related Topics

- Types of Temporary Tables
  Temporary tables are classified as global, private, or cursor-duration.
- Choosing an Optimizer Goal
  The **optimizer goal** is the prioritization of resource usage by the optimizer.

20.1.1.2.3 Cursor Mismatches and V$SQL_SHARED_CURSOR

If a parent cursor has multiple children, then the V$SQL_SHARED_CURSOR view provides information about why the cursor was not shared. For several types of incompatibility, the TRANSLATION_MISMATCH column indicates a mismatch with the value Y or N.

Example 20-3 Translation Mismatch

In this example, the TRANSLATION_MISMATCH column shows that the two statements (SELECT * FROM employees) referenced different objects, resulting in a

TRANSLATION_MISMATCH value of Y for the last statement. Because sharing was not possible, each statement had a separate child cursor, as indicated by CHILD_NUMBER of 0 and 1.

```
SELECT S.SQL_TEXT, S.CHILD_NUMBER, s.CHILD_ADDRESS,
       C.TRANSLATION_MISMATCH
FROM   V$SQL S, V$SQL_SHARED_CURSOR C
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%'
AND S.CHILD_ADDRESS = C.CHILD_ADDRESS;

SQL_TEXT                       CHILD_NUMBER    CHILD_ADDRESS T
------------------------------ ------------ ---------------- -
SELECT * FROM employees                   0 0000000081EE8690 N
SELECT * FROM employees                   1 0000000081F22508 Y
```

20.1.2 About Cursors and Parsing

If an application issues a statement, and if Oracle Database cannot reuse a cursor, then it must build a new executable version of the application code. This operation is known as a **hard parse**.

A soft parse is any parse that is not a hard parse, and occurs when the database can reuse existing code. Some soft parses are less resource-intensive than others. For example, if a parent cursor for the statement already exists, then Oracle Database can perform various optimizations, and then store the child cursor in the shared SQL area. If a parent cursor does not exist, however, then Oracle Database must also store the parent cursor in the shared SQL area, which creates additional memory overhead.

Effectively, a hard parse recompiles a statement before running it. Hard parsing a SQL statement before every execution is analogous to recompiling a C program before every execution. A hard parse performs operations such as the following:

- Checking the syntax of the SQL statement
- Checking the semantics of the SQL statement
- Checking the access rights of the user issuing the statement
- Creating an execution plan
- Accessing the library cache and data dictionary cache numerous times to check the data dictionary

An especially resource-intensive aspect of hard parsing is accessing the library cache and data dictionary cache numerous times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change during the check. Latch contention increases statement execution time and decreases concurrency.

For all of the preceding reasons, the CPU and memory overhead of hard parses can create serious performance problems. The problems are especially evident in web applications that accept user input from a form, and then generate SQL statements dynamically. The Real-World Performance group strongly recommends reducing hard parsing as much as possible.

Video:

Video

Example 20-4 Finding Parse Information Using V$SQL

You can use various techniques to monitor hard and soft parsing. This example queries the session statistics to determine whether repeated executions of a `DBA_JOBS` query increase the hard parse count. The first execution of the statement increases the hard parse count to `49`, but the second execution does not change the hard parse count, which means that Oracle Database reused application code.

```
SQL> ALTER SYSTEM FLUSH SHARED_POOL;

System altered.

SQL> COL NAME FORMAT a18

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';

NAME                    VALUE
------------------ ----------
parse count (hard)         48

SQL> SELECT COUNT(*) FROM DBA_JOBS;

  COUNT(*)
----------
     0

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';

NAME                    VALUE
------------------ ----------
parse count (hard)         49

SQL> SELECT COUNT(*) FROM DBA_JOBS;

  COUNT(*)
----------
     0

SQL> SELECT s.NAME, m.VALUE
  2  FROM   V$STATNAME s, V$MYSTAT m
  3  WHERE  s.STATISTIC# = m.STATISTIC#
  4  AND    s.NAME LIKE '%(hard%';
NAME                        VALUE
------------------- ----------
parse count (hard)          49
```

Example 20-5   Finding Parse Information Using Trace Files

This example uses SQL Trace and the TKPROF utility to find parse information. You log in to the database with administrator privileges, and then query the directory location of the trace files (sample output included):

```
SET LINESIZE 120
COLUMN value FORMAT A80

SELECT value
FROM   v$diag_info
WHERE  name = 'Default Trace File';

VALUE
---------------------------------------------------------------------------
/disk1/oracle/log/diag/rdbms/orcl/orcl/trace/orcl_ora_23054.trc
```

You enable tracing, use the `TRACEFILE_IDENTIFIER` initialization parameter to give the trace file a meaningful name, and then query `hr.employees`:

```
EXEC DBMS_MONITOR.SESSION_TRACE_ENABLE(waits=>TRUE, binds=>TRUE);
ALTER SESSION SET TRACEFILE_IDENTIFIER = "emp_stmt";
SELECT * FROM hr.employees;
EXIT;
```

Search the default trace file directory for the trace file that you generated:

```
% ls *emp_stmt.trc
orcl_ora_17950_emp_stmt.trc
```

Use TKPROF to format the trace file, and then open the formatted file:

```
% tkprof orcl_ora_17950_emp_stmt.trc emp.out; vi emp.out
```

The formatted trace file contains the parse information for the query of `hr.employees`.

```
SQL ID: brmjpfs7dcnub Plan Hash: 1445457117

SELECT *
FROM
 hr.employees


call     count       cpu    lapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.07       0.08          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
```

```
Fetch        9     0.00      0.00         3       12        0       107
-------- -----   -------- ---------- --------- -------- --------  --------
total       11     0.07      0.08         3       12        0       107

Misses in library cache during parse: 1
Optimizer mode: ALL_ROWS
Parsing user id: SYSTEM
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  
---------------------------------------
       107        107        107  TABLE ACCESS FULL EMPLOYEES (cr=12 
pr=3 
                                  pw=0 time=497 us starts=1 cost=2 
                                  size=7383 card=107)
```

A library cache miss indicates a hard parse. Performing the same steps for a second execution of the same statement produces the following trace output, which shows no library cache misses:

```
SQL ID: brmjpfs7dcnub Plan Hash: 1445457117

SELECT *
FROM
 hr.employees


call     count     cpu   elapsed      disk    query   current      rows
------- ------  ------ ---------- --------- -------- ---------  --------
Parse        1    0.00      0.00         0        0         0         0
Execute      1    0.00      0.00         0        0         0         0
Fetch        9    0.00      0.00         3       12         0       107
------- ------  ------ ---------- --------- -------- ---------  --------
total       11    0.00      0.00         3       12         0       107

Misses in library cache during parse: 0
Optimizer mode: ALL_ROWS
Parsing user id: SYSTEM
Number of plan statistics captured: 1

Rows (1st) Rows (avg) Rows (max)  Row Source Operation
---------- ---------- ----------  
---------------------------------------
       107        107        107  TABLE ACCESS FULL EMPLOYEES (cr=12 
pr=3 
                                  pw=0 time=961 us starts=1 cost=2 
                                  size=7383 card=107)
```

See Also:

"Shared Pool Check"

20.1.3 About Literals and Bind Variables

Bind variables are essential to cursor sharing in Oracle database applications.

20.1.3.1 Literals and Cursors

When constructing SQL statements, some Oracle applications use literals instead of bind variables.

For example, the statement `SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101` uses the literal value `101` for the employee ID. By default, when similar statements do not use bind variables, Oracle Database cannot take advantage of cursor sharing. Thus, Oracle Database sees a statement that is identical except for the value `102`, or any other random value, as a completely new statement, requiring a hard parse.

The Real-World Performance group has determined that applications that use literals are a frequent cause of performance, scalability, and security problems. In the real world, it is not uncommon for applications to be written quickly, without considering cursor sharing. A classic example is a “screen scraping” application that copies the contents out of a web form, and then concatenates strings to construct the SQL statement dynamically.

Major problems that result from using literal values include the following:

- Applications that concatenate literals input by an end user are prone to SQL injection attacks. Only rewriting the applications to use bind variables eliminates this threat.
- If every statement is hard parsed, then cursors are not shared, and so the database must consume more memory to create the cursors.
- Oracle Database must latch the shared pool and library cache when hard parsing. As the number of hard parses increases, so does the number of processes waiting to latch the shared pool. This situation decreases concurrency and increases contention.

Video:

Video

Example 20-6 Literals and Cursor Sharing

Consider an application that executes the following statements, which differ only in literals:

```
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
```

The following query of V$SQLAREA shows that the three statements require three different parent cursors. As shown by VERSION_COUNT, each parent cursor requires its own child cursor.

```
COL SQL_TEXT FORMAT a30
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                       SQL_ID VERSION_COUNT HASH_VALUE
------------------------------ ------------- ------------- ----------
SELECT SUM(salary) FROM hr.emp b1tvfcc5qnczb             1  191509483
loyees WHERE employee_id < 165
SELECT SUM(salary) FROM hr.emp cn5250y0nqpym             1 2169198547
loyees WHERE employee_id < 101
SELECT SUM(salary) FROM hr.emp au8nag2vnfw67             1 3074912455
loyees WHERE employee_id < 120
```

See Also:

"Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix" to learn about SQL injection

20.1.3.2 Bind Variables and Cursors

You can develop Oracle applications to use bind variables instead of literals.

A bind variable is a placeholder in a query. For example, the statement `SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id` uses the bind variable`:emp_id` for the employee ID.

The Real-World Performance group has found that applications that use bind variables perform better, scale better, and are more secure. Major benefits that result from using bind variables include the following:

- Applications that use bind variables are not vulnerable to the same SQL injection attacks as applications that use literals.
- When identical statements use bind variables, Oracle Database can take advantage of cursor sharing, and share the plan and other information when different values are bound to the same statement.
- Oracle Database avoids the overhead of latching the shared pool and library cache required for hard parsing.

Video:

Video

Example 20-7 Bind Variables and Shared Cursors

The following example uses the `VARIABLE` command in SQL*Plus to create the `emp_id` bind variable, and then executes a query using three different bind values (`101`, `120`, and `165`):

```
VARIABLE emp_id NUMBER

EXEC :emp_id := 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
EXEC :emp_id := 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
EXEC :emp_id := 165;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
```

The following query of `V$SQLAREA` shows one unique SQL statement:

```
COL SQL_TEXT FORMAT a34
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                           SQL_ID        VERSION_COUNT HASH_VALUE
---------------------------------- ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employe 4318cbskba8yh             1 615850960
es WHERE employee_id < :emp_id
```

The `VERSION_COUNT` value of `1` indicates that the database reused the same child cursor rather than creating three separate child cursors. Using a bind variable made this reuse possible.

20.1.3.3 Bind Variable Peeking

In **bind variable peeking** (also known as **bind peeking**), the optimizer looks at the value in a bind variable when the database performs a hard parse of a statement.

The optimizer does not look at the bind variable values before every parse. Rather, the optimizer peeks only when the optimizer is *first* invoked, which is during the hard parse.

When a query uses literals, the optimizer can use the literal values to find the best plan. However, when a query uses bind variables, the optimizer must select the best plan without the presence of literals in the SQL text. This task can be extremely difficult. By peeking at bind values during the initial hard parse, the optimizer can determine the cardinality of a `WHERE` clause condition as if literals *had* been used, thereby improving the plan.

Because the optimizer only peeks at the bind value during the hard parse, the plan may not be optimal for all possible bind values. The following examples illustrate this principle.

Example 20-8 Literals Result in Different Execution Plans

Assume that you execute the following statements, which execute three different statements using different literals (`101`, `120`, and `165`), and then display the execution plans for each:

```
SET LINESIZE 167
SET PAGESIZE 0
```

```
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The database hard parsed all three statements, which were not identical. The `DISPLAY_CURSOR` output, which has been edited for clarity, shows that the optimizer chose the same index range scan plan for the first two statements, but a full table scan plan for the statement using literal `165`:

```
SQL_ID    cn5250y0nqpym, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101

Plan hash value: 2410354593

-------------------------------------------------------------------------------------
|Id| Operation                          | Name         |Rows|Bytes|Cost(%CPU)|Time|
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |              |  |    |2 (100)|          |
| 1|  SORT AGGREGATE                    |              |1 | 8 |        |          |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   |1 | 8 |2 (0)   | 00:00:01 |
|*3|    INDEX RANGE SCAN                | EMP_EMP_ID_PK |1 |    |1 (0)   | 00:00:01 |
-------------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<101)

SQL_ID    au8nag2vnfw67, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120

Plan hash value: 2410354593

-------------------------------------------------------------------------------------
|Id| Operation                          | Name         |Rows|Bytes|Cost(%CPU)|Time|
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                   |              |  |    |2 (100)|          |
| 1|  SORT AGGREGATE                    |              |1 | 8 |        |          |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES   |20|160|2 (0)   | 00:00:01 |
|*3|    INDEX RANGE SCAN                | EMP_EMP_ID_PK |20|    |1 (0)   | 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<120)

SQL_ID    b1tvfcc5qnczb, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165
```

```
Plan hash value: 1756381138

-------------------------------------------------------------------------
| Id  | Operation           | Name      |Rows| Bytes |Cost(%CPU)| Time    |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |           |    |       | 2 (100)|           |
|   1 |  SORT AGGREGATE     |           |  1 |     8 |        |           |
|*  2 |   TABLE ACCESS FULL| EMPLOYEES | 66 |   528 | 2   (0)| 00:00:01 |
-------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("EMPLOYEE_ID"<165)
```

The preceding output shows that the optimizer considers a full table scan more efficient than an index scan for the query that returns more rows.

Example 20-9 Bind Variables Result in Cursor Reuse

This example rewrites the queries executed in Example 20-8 to use bind variables instead of literals. You bind the same values (`101`, `120`, and `165`) to the bind variable `:emp_id`, and then display the execution plans for each:

```
VAR emp_id NUMBER

EXEC :emp_id := 101;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
EXEC :emp_id := 120;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
EXEC :emp_id := 165;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The `DISPLAY_CURSOR` output shows that the optimizer chose exactly the same plan for all three statements:

```
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id

Plan hash value: 2410354593

-------------------------------------------------------------------------------------
| Id | Operation                             | Name      |Rows|Bytes|Cost (%CPU)|Time|
-------------------------------------------------------------------------------------
|  0 | SELECT STATEMENT                      |              | |  |2 (100)|          |
|  1 |  SORT AGGREGATE                       |              |1|8 |       |          |
|  2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES    |1|8 |  2 (0)| 00:00:01 |
|* 3 |    INDEX RANGE SCAN                   | EMP_EMP_ID_PK |1|  |  1 (0)| 00:00:01 |
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
```

```
-------------------------------------------------

   3 - access("EMPLOYEE_ID"<:EMP_ID)
```

In contrast, when the preceding statements were executed with literals, the optimizer chose a lower-cost full table scan when the employee ID value was `165`. This is the problem solved by adaptive cursor sharing.

See Also:

"Adaptive Cursor Sharing"

20.1.4 About the Life Cycle of Shared Cursors

The database allocates a new shared SQL area when the optimizer parses a new SQL statement that is not DDL. The amount of memory required depends on the statement complexity.

The database can remove a shared SQL area from the shared pool even if this area corresponds to an open cursor that has been unused for a long time. If the open cursor is later used to run its statement, then the database reparses the statement and allocates a new shared SQL area. The database does not remove cursors whose statements are executing, or whose rows have not been completely fetched.

Shared SQL areas can become invalid because of changes to dependent schema objects or to optimizer statistics. Oracle Database uses two techniques to manage the cursor life cycle: invalidation and rolling invalidation.

See Also:

Oracle Database Concepts for an overview of memory allocation in the shared pool

20.1.4.1 Cursor Marked Invalid

When a shared SQL area is marked invalid, the database can remove it from the shared pool, along with valid cursors that have been unused for some time.

In some situations, the database must execute a statement that is associated with an invalid shared SQL area in the shared pool. In this case, the database performs a hard parse of the statement before execution.

The database immediately marks dependent shared SQL areas invalid when the following conditions are met:

- `DBMS_STATS` gathers statistics for a table, table cluster, or index when the `NO_INVALIDATE` parameter is `FALSE`.
- A SQL statement references a schema object, which is later modified by a DDL statement that uses immediate cursor invalidation (default).

You can manually specify immediate invalidation on statements such as `ALTER TABLE ... IMMEDIATE VALIDATION` and `ALTER INDEX ... IMMEDIATE VALIDATION`, or set the `CURSOR_INVALIDATION` initialization parameter to `IMMEDIATE` at the session or system level.

Note:

A DDL statement using the `DEFERRED VALIDATION` clause overrides the `IMMEDIATE` setting of the `CURSOR_INVALIDATION` initialization parameter.

When the preceding conditions are met, the database reparses the affected statements at next execution.

When the database invalidates a cursor, the `V$SQL.INVALIDATIONS` value increases (for example, from `0` to `1`), and `V$SQL.OBJECT_STATUS` shows `INVALID_UNAUTH`.

Example 20-10 Forcing Cursor Invalidation by Setting NO_INVALIDATE=FALSE

This example logs in as user `sh`, who has been granted administrator privileges. The example queries `sales`, and then gathers statistics for this table with `NO_INVALIDATE=FALSE`. Afterward, the `V$SQL.INVALIDATIONS` value changes from `0` to `1` for the cursor, indicating that the database flagged the cursor as invalid.

```
SQL> SELECT COUNT(*) FROM sales;

  COUNT(*)
----------
    918843

SQL> SELECT PREV_SQL_ID SQL_ID FROM V$SESSION WHERE SID =
SYS_CONTEXT('userenv', 'SID');

SQL_ID
-------------
1y17j786c7jbh

SQL> SELECT CHILD_NUMBER, EXECUTIONS,
  2  PARSE_CALLS, INVALIDATIONS, OBJECT_STATUS
  3  FROM V$SQL WHERE SQL_ID = '1y17j786c7jbh';

CHILD_NUMBER EXECUTIONS PARSE_CALLS INVALIDATIONS OBJECT_STATUS
------------ ---------- ----------- ------------- -------------
           0          1           1             0 VALID

SQL> EXEC DBMS_STATS.GATHER_TABLE_STATS(null,'sales',no_invalidate => FALSE);

PL/SQL procedure successfully completed.

SQL> SELECT CHILD_NUMBER, EXECUTIONS,
  2  PARSE_CALLS, INVALIDATIONS, OBJECT_STATUS
  3  FROM V$SQL WHERE SQL_ID = '1y17j786c7jbh';

CHILD_NUMBER EXECUTIONS PARSE_CALLS INVALIDATIONS OBJECT_STATUS
```

```
------------ ---------- ----------- ------------- --------------
           0          1           1             1 INVALID_UNAUTH
```

See Also:

- "About Optimizer Initialization Parameters"
- *Oracle Database SQL Language Reference* to learn more about `ALTER TABLE ... IMMEDIATE VALIDATION` and other DDL statements that permit immediate validation
- *Oracle Database Reference* to learn more about `V$SQL` and `V$SQLAREA` dynamic views
- *Oracle Database Reference* to learn more about the `CURSOR_INVALIDATION` initialization parameter

20.1.4.2 Cursors Marked Rolling Invalid

When cursors are marked rolling invalid (`V$SQL.IS_ROLLING_INVALID` is `Y`), the database gradually performs hard parses over an extended time.

Note:

When `V$SQL.IS_ROLLING_REFRESH_INVALID` is `Y`, the underlying object has changed, but recompilation of the cursor is not required. The database updates metadata in the cursor.

Purpose of Rolling Invalidation

Because a sharp increase in hard parses can significantly degrade performance, rolling invalidation—also called *deferred invalidation*—is useful for workloads that simultaneously invalidate many cursors. The database assigns each invalid cursor a randomly generated time period. SQL areas invalidated at the same time typically have different time periods.

A hard parse occurs only if a query accessing the cursor executes *after* the time period has expired. In this way, the database diffuses the overhead of hard parsing over time.

Note:

If parallel SQL statements are marked rolling invalid, then the database performs a hard parse at next execution, regardless of whether the time period has expired. In an Oracle Real Application Clusters (Oracle RAC) environment, this technique ensures consistency between execution plans of parallel execution servers and the query coordinator.

An analogy for rolling invalidation might be the gradual replacement of worn-out office furniture. Instead of replacing all the furniture at once, forcing a substantial financial outlay, a company assigns each piece a different expiration date. Over the course of a year, a piece stays in use until it is replaced, at which point a cost is incurred.

Specification of Deferred Invalidation

By default, DDL specifies that statements accessing the object use immediate cursor invalidation. For example, if you create a table or an index, then cursors that reference this table or index use immediate invalidation.

If a DDL statement supports deferred cursor invalidation, then you can override the default behavior by using statements such as `ALTER TABLE ... DEFERRED INVALIDATION`. The options depend on the DDL statement. For example, `ALTER INDEX` only supports `DEFERRED INVALIDATION` when the `UNUSABLE` or `REBUILD` option is also specified.

An alternative to DDL is setting the `CURSOR_INVALIDATION` initialization parameter to `DEFERRED` at the session or system level. A DDL statement using the `IMMEDIATE INVALIDATION` clause overrides the `DEFERRED` setting of the `CURSOR_INVALIDATION` initialization parameter.

When Rolling Invalidation Occurs

If the `DEFERRED INVALIDATION` attribute applies to an object, either as a result of DDL or an initialization parameter setting, then statements that access the object may be subject to deferred invalidation. The database marks shared SQL areas as rolling invalid in either of the following circumstances:

- `DBMS_STATS` gathers statistics for a table, table cluster, or index when the `NO_INVALIDATE` parameter is set to `DBMS_STATS.AUTO_INVALIDATE`. This is the default setting.
- One of the following statements is issued with `DEFERRED INVALIDATION` in circumstances that do not prevent the use of deferred invalidation:
  - `ALTER TABLE` on partitioned tables
  - `ALTER TABLE ... PARALLEL`
  - `ALTER INDEX ... UNUSABLE`
  - `ALTER INDEX ... REBUILD`
  - `CREATE INDEX`
  - `DROP INDEX`
  - `TRUNCATE TABLE` on partitioned tables

  A subset of DDL statements require immediate cursor invalidation for DML (`INSERT`, `UPDATE`, `DELETE`, or `MERGE`) but not `SELECT` statements. Many factors relating to the specific DDL statements and affected cursors determine whether Oracle Database uses deferred invalidation.

See Also:

- "About Optimizer Initialization Parameters"
- *Oracle Database SQL Language Reference* to learn more about `ALTER TABLE ... DEFERRED INVALIDATION` and other DDL statements that permit deferred invalidation
- *Oracle Database Reference* to learn more about `V$SQL` and `V$SQLAREA` dynamic views
- *Oracle Database Reference* to learn more about the `CURSOR_INVALIDATION` initialization parameter

20.2 CURSOR_SHARING and Bind Variable Substitution

This topic explains what the `CURSOR_SHARING` initialization parameter is, and how setting it to different values affects how Oracle Database uses bind variables.

20.2.1 CURSOR_SHARING Initialization Parameter

The `CURSOR_SHARING` initialization parameter controls how the database processes statements with bind variables.

In Oracle Database 12c, the parameter supports the following values:

- `EXACT`

  This is the default value. The database enables only textually identical statements to share a cursor. The database does not attempt to replace literal values with system-generated bind variables. In this case, the optimizer generates a plan for each statement based on the literal value.
- `FORCE`

  The database replaces all literals with system-generated bind variables. For statements that are identical after the bind variables replace the literals, the optimizer uses the same plan.

Note:

The `SIMILAR` value for `CURSOR_SHARING` is deprecated.

You can set `CURSOR_SHARING` at the system or session level, or use the `CURSOR_SHARING_EXACT` hint at the statement level.

See Also:

"Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix"

20.2.2 Parsing Behavior When CURSOR_SHARING = FORCE

When SQL statements use literals rather than bind variables, setting the `CURSOR_SHARING` initialization parameter to `FORCE` enables the database to replace literals with system-generated bind variables. Using this technique, the database can sometimes reduce the number of parent cursors in the shared SQL area.

Note:

If a statement uses an `ORDER BY` clause, then while the database may perform literal replacement in the clause, the cursor will not necessarily be sharable because different values of the column number as a literal imply different query results and potentially a different execution plan. The column number in the `ORDER BY` clause affects the query plan and execution, so the database cannot share two cursors having different column numbers.

When `CURSOR_SHARING` is set to `FORCE`, the database performs the following steps during the parse:

1. Copies *all* literals in the statement to the PGA, and replaces them with system-generated bind variables

   For example, an application could process the following statement:

   ```
   SELECT SUBSTR(last_name, 1, 4), SUM(salary)
   FROM   hr.employees
   WHERE  employee_id < 101 GROUP BY last_name
   ```

   The optimizer replaces literals, including the literals in the `SUBSTR` function, as follows:

   ```
   SELECT SUBSTR(last_name, :"SYS_B_0", :"SYS_B_1"), SUM(salary)
   FROM   hr.employees
   WHERE  employee_id < :"SYS_B_2" GROUP BY last_name
   ```

2. Searches for an identical statement (same SQL hash value) in the shared pool

   If an identical statement is *not* found, then the database performs a hard parse. Otherwise, the database proceeds to the next step.

3. Performs a soft parse of the statement

As the preceding steps indicate, setting the `CURSOR_SHARING` initialization parameter to `FORCE` does *not* reduce the parse count. Rather, in some cases, `FORCE` enables the database to perform a soft parse instead of a hard parse. Also, `FORCE` does not the prevent against SQL injection attacks because Oracle Database binds the values after any injection has already occurred.

Example 20-11 Replacement of Literals with System Bind Variables

This example sets `CURSOR_SHARING` to `FORCE` at the session level, executes three statements containing literals, and displays the plan for each statement:

```
ALTER SESSION SET CURSOR_SHARING=FORCE;
SET LINESIZE 170
SET PAGESIZE 0
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 101;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 120;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
SELECT SUM(salary) FROM hr.employees WHERE employee_id < 165;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR());
```

The following `DISPLAY_CURSOR` output, edited for readability, shows that all three statements used the same plan. The optimizer chose the plan, an index range scan, because it peeked at the *first* value (`101`) bound to the system bind variable, and picked this plan as the best for all values. In fact, this plan is not the best plan for all values. When the value is `165`, a full table scan is more efficient.

```
SQL_ID    cxx8n1cxr9khn, child number 0
-------------------------------------
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :"SYS_B_0"

Plan hash value: 2410354593

-----------------------------------------------------------------------------------
| Id  | Operation                            | Name         |Rows|Bytes|Cost(%CPU)|Time|
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT                     |               |   |    |2 (100)|        |
|   1 |  SORT AGGREGATE                      |               |1  | 8  |       |        |
|   2 |   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES     |1  | 8  |2  (0)  |00:00:01|
|*  3 |    INDEX RANGE SCAN                  | EMP_EMP_ID_PK |1  |    |1  (0)  |00:00:01|
-----------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("EMPLOYEE_ID"<101)
```

A query of `V$SQLAREA` confirms that Oracle Database replaced with the literal with system bind variable `:"SYS_B_0"`, and created one parent and one child cursor (`VERSION_COUNT=1`) for all three statements, which means that all executions shared the same plan.

```
COL SQL_TEXT FORMAT a36
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                              SQL_ID        VERSION_COUNT
```

```
HASH_VALUE
------------------------------------ ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employees cxx8n1cxr9khn             1  997509652
 WHERE employee_id < :"SYS_B_0"
```

See Also:

- "Private and Shared SQL Areas" for more details on the various checks performed
- *Oracle Database Reference* to learn about the `CURSOR_SHARING` initialization parameter

20.3 Adaptive Cursor Sharing

The **adaptive cursor sharing** feature enables a single statement that contains bind variables to use multiple execution plans.

Cursor sharing is "adaptive" because the cursor adapts its behavior so that the database does not always use the same plan for each execution or bind variable value.

20.3.1 Purpose of Adaptive Cursor Sharing

With bind peeking, the optimizer peeks at the values of user-defined bind variables on the first invocation of a cursor.

The optimizer determines the cardinality of any `WHERE` clause condition as if literals had been used instead of bind variables. If a column in a `WHERE` clause has skewed data, however, then a histogram may exist on this column. When the optimizer peeks at the value of the user-defined bind variable and chooses a plan, this plan may not be good for all values.

In adaptive cursor sharing, the database monitors data accessed over time for different bind values, ensuring the optimal choice of cursor for a specific bind value. For example, the optimizer might choose one plan for bind value `10` and a different plan for bind value `50`. Cursor sharing is "adaptive" because the cursor adapts its behavior so that the optimizer does not always choose the same plan for each execution or bind variable value. Thus, the optimizer automatically detects when different execution of a statement would benefit from different execution plans.

Note:

Adaptive cursor sharing is independent of the `CURSOR_SHARING` initialization parameter. Adaptive cursor sharing is equally applicable to statements that contain user-defined and system-generated bind variables. Adaptive cursor sharing does not apply to statements that contain only literals.

20.3.2 How Adaptive Cursor Sharing Works: Example

Adaptive cursor sharing monitors statements that use bind variables to determine whether a new plan is more efficient.

Assume that an application executes the following statement five times, binding different values every time:

```
SELECT * FROM employees WHERE salary = :sal AND department_id = :dept
```

Also assume in this example that a histogram exists on at least one of the columns in the predicate. The database processes this statement as follows:

1. The application issues the statement for the first time, which causes a hard parse. During the parse, the database performs the following tasks:
   - Peeks at the bind variables to generate the initial plan.
   - Marks the cursor as bind-sensitive. A bind-sensitive cursor is a cursor whose optimal plan may depend on the value of a bind variable. To determine whether a different plan is beneficial, the database monitors the behavior of a bind-sensitive cursor that uses different bind values.
   - Stores metadata about the predicate, including the cardinality of the bound values (in this example, assume that only 5 rows were returned).
   - Creates an execution plan (in this example, index access) based on the peeked values.
2. The database executes the cursor, storing the bind values and execution statistics in the cursor.
3. The application issues the statement a second time, using different bind variables, causing the database to perform a soft parse, and find the matching cursor in the library cache.
4. The database executes the cursor.
5. The database performs the following post-execution tasks:
   a. The database compares the execution statistics for the second execution with the first-execution statistics.
   b. The database observes the pattern of statistics over all previous executions, and then decides whether to mark the cursor as a bind-aware cursor. In this example, assume that the database decides the cursor is bind-aware.
6. The application issues the statement a third time, using different bind variables, which causes a soft parse. Because the cursor is bind-aware, the database does the following:
   - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is similar: 8 rows instead of 5 rows.
   - Reuses the execution plan in the existing child cursor.
7. The database executes the cursor.

8. The application issues the statement a fourth time, using different bind variables, causing a soft parse. Because the cursor is bind-aware, the database does the following:
   - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is vastly different: 102 rows (in a table with 107 rows) instead of 5 rows.
   - Does not find a matching child cursor.
9. The database performs a hard parse. As a result, the database does the following:
   - Creates a new child cursor with a *second* execution plan (in this example, a full table scan)
   - Stores metadata about the predicate, including the cardinality of the bound values, in the cursor
10. The database executes the new cursor.
11. The database stores the new bind values and execution statistics in the new child cursor.
12. The application issues the statement a fifth time, using different bind variables, which causes a soft parse. Because the cursor is bind-aware, the database does the following:
    - Determines whether the cardinality of the new values falls within the same range as the stored cardinality. In this example, the cardinality is 20.
    - Does not find a matching child cursor.
13. The database performs a hard parse. As a result, the database does the following:
    a. Creates a new child cursor with a *third* execution plan (in this example, index access)
    b. Determines that this index access execution plan is the same as the index access execution plan used for the first execution of the statement
    c. Merges the two child cursors containing index access plans, which involves storing the combined cardinality statistics into one child cursor, and deleting the other one
14. The database executes the cursor using the index access execution plan.

20.3.3 Bind-Sensitive Cursors

A **bind-sensitive cursor** is a cursor whose optimal plan may depend on the value of a bind variable.

The database has examined the bind value when computing cardinality, and considers the query “sensitive” to plan changes based on different bind values. The database monitors the behavior of a bind-sensitive cursor that uses different bind values to determine whether a different plan is beneficial.

The optimizer uses the following criteria to decide whether a cursor is bind-sensitive:

- The optimizer has peeked at the bind values to generate cardinality estimates.
- The bind is used in an equality or a range predicate.

For each execution of the query with a new bind value, the database records the execution statistics for the new value and compares them to the execution statistics for the previous value. If execution statistics vary greatly, then the database marks the cursor bind-aware.

Example 20-12   Column with Significant Data Skew

This example assumes that the `hr.employees.department_id` column has significant data skew. `SYSTEM` executes the following setup code, which adds 100,000 employees in department 50 to the `employees` table in the sample schema, for a total of 100,107 rows, and then gathers table statistics:

```
DELETE FROM hr.employees WHERE employee_id > 999;

ALTER TABLE hr.employees DISABLE NOVALIDATE CONSTRAINT emp_email_uk;

DECLARE
v_counter NUMBER(7) := 1000;
BEGIN
 FOR i IN 1..100000 LOOP
 INSERT INTO hr.employees
   VALUES (v_counter, null, 'Doe', 'Doe@example.com', null,'07-JUN-02',
     'AC_ACCOUNT', null, null, null, 50);
 v_counter := v_counter + 1;
 END LOOP;
END;
/
COMMIT;

BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (ownname = 'hr',tabname =>
'employees');
END;
/

ALTER SYSTEM FLUSH SHARED_POOL;
```

The following query shows a histogram on the `employees.department_id` column:

```
COL TABLE_NAME FORMAT a15
COL COLUMN_NAME FORMAT a20
COL HISTOGRAM FORMAT a9

SELECT TABLE_NAME, COLUMN_NAME, HISTOGRAM
FROM   DBA_TAB_COLS
WHERE  OWNER = 'HR'
AND    TABLE_NAME = 'EMPLOYEES'
AND    COLUMN_NAME = 'DEPARTMENT_ID';

TABLE_NAME      COLUMN_NAME          HISTOGRAM
--------------- -------------------- ---------
EMPLOYEES       DEPARTMENT_ID        FREQUENCY
```

Example 20-13 Low-Cardinality Query

This example continues the example in Example 20-12. The following query shows that the value `10` has extremely low cardinality for the column `department_id`, occupying .00099% of the rows:

```
VARIABLE dept_id NUMBER
EXEC :dept_id := 10;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE
department_id = :dept_id;
```

```
  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
         1              200
```

The optimizer chooses an index range scan, as expected for such a low-cardinality query:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 0
-------------------------------------
select COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------
| Id| Operation                           | Name            |Rows|Bytes|Cost (%CPU)|Time  |
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |                   |  |  |2(100)|        |
| 1|  SORT AGGREGATE                      |                   |1 |8 |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES         |1 |8 |2  (0)|00:00:01|
|*3|    INDEX RANGE SCAN                  | EMP_DEPARTMENT_IX |1 |  |1  (0)|00:00:01|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

The following query of `V$SQL` obtains information about the cursor:

```
COL BIND_AWARE FORMAT a10
COL SQL_TEXT FORMAT a22
COL CHILD# FORMAT 99999
COL EXEC FORMAT 9999
COL BUFF_GETS FORMAT 999999999
COL BIND_SENS FORMAT a9
COL SHARABLE FORMAT a9

SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHARABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHARABLE
---------------------- ------ ----- ---------- --------- ---------- --------
SELECT COUNT(*), MAX(e      0     1        196         Y          N        Y
mployee_id) FROM hr.em
```

```
ployees WHERE departme
nt_id = :dept_id
```

The preceding output shows one child cursor that has been executed once for the low-cardinality query. The cursor has been marked bind-sensitive because the optimizer believes the optimal plan may depend on the value of the bind variable.

When a cursor is marked bind-sensitive, Oracle Database monitors the behavior of the cursor using different bind values, to determine whether a different plan for different bind values is more efficient. The database marked this cursor bind-sensitive because the optimizer used the histogram on the `department_id` column to compute the selectivity of the predicate `WHERE department_id = :dept_id`. Because the presence of the histogram indicates that the column is skewed, different values of the bind variable may require different plans.

Example 20-14 High-Cardinality Query

This example continues the example in Example 20-13. The following code re-executes the same query using the value `50`, which occupies 99.9% of the rows:

```
EXEC :dept_id := 50;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE
department_id = :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
    100045           100999
```

Even though such an unselective query would be more efficient with a full table scan, the optimizer chooses the same index range scan used for `department_id=10`. This reason is that the database assumes that the existing plan in the cursor can be shared:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 0
-------------------------------------
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------
| Id| Operation                           | Name         |Rows|Bytes|Cost (%CPU)|Time |
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |                   |   |   |2(100)|        |
| 1|  SORT AGGREGATE                      |                   |1  |8  |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES         |1  |8  |2   (0)|00:00:01|
|*3|    INDEX RANGE SCAN                  | EMP_DEPARTMENT_IX |1  |   |1   (0)|00:00:01|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
```

```
   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

A query of `V$SQL` shows that the child cursor has now been executed twice:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHARABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHARABLE
---------------------- ------ ----- ---------- --------- ---------- --------
SELECT COUNT(*), MAX(e      0     2       1329         Y          N        Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

At this stage, the optimizer has not yet marked the cursor as bind-aware.

See Also:

Oracle Database Reference to learn about `V$SQL`

20.3.4 Bind-Aware Cursors

A **bind-aware cursor** is a bind-sensitive cursor that is eligible to use different plans for different bind values.

After a cursor has been made bind-aware, the optimizer chooses plans for future executions based on the bind value and its cardinality estimate. Thus, “bind-aware” means essentially “best plan for the current bind value.”

When a statement with a bind-sensitive cursor executes, the optimizer uses an internal algorithm to determine whether to mark the cursor bind-aware. The decision depends on whether the cursor produces significantly different data access patterns for different bind values, resulting in a performance cost that differs from expectations.

If the database marks the cursor bind-aware, then the *next* time that the cursor executes the database does the following:

- Generates a new plan based on the bind value
- Marks the original cursor generated for the statement as not sharable (`V$SQL.IS_SHAREABLE` is `N`). The original cursor is no longer usable and is eligible to age out of the library cache

When the same query repeatedly executes with different bind values, the database adds new bind values to the “signature” of the SQL statement (which includes the optimizer environment, NLS settings, and so on), and categorizes the values. The database examines the bind values, and considers whether the current bind value results in a significantly different data volume, or whether an existing plan is sufficient. The database does *not* need to create a new plan for each new value.

Consider a scenario in which you execute a statement with 12 distinct bind values (executing each distinct value twice), which causes the database to trigger 5 hard parses, and create 2 additional plans. Because the database performs 5 hard parses, it creates 5 new child cursors, even though some cursors have the same execution plan as existing cursors. The database marks the superfluous cursors as not usable, which means these cursors eventually age out of the library cache.

During the initial hard parses, the optimizer is essentially mapping out the relationship between bind values and the appropriate execution plan. After this initial period, the database eventually reaches a steady state. Executing with a new bind value results in picking the best child cursor in the cache, without requiring a hard parse. Thus, the number of parses does *not* scale with the number of different bind values.

Example 20-15 Bind-Aware Cursors

This example continues the example in "Bind-Sensitive Cursors". The following code issues a second query `employees` with the bind variable set to `50`:

```
EXEC :dept_id := 50;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE
department_id = :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
    100045           100999
```

During the first two executions, the database was monitoring the behavior of the queries, and determined that the different bind values caused the queries to differ significantly in cardinality. Based on this difference, the database adapts its behavior so that the same plan is not always shared for this query. Thus, the optimizer generates a new plan based on the current bind value, which is `50`:

```
SQL> SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 1
-------------------------------------
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE
department_id = :dept_id

Plan hash value: 1756381138

-----------------------------------------------------------------------
----
| Id  | Operation         | Name      | Rows   | Bytes | Cost(%CPU)|
Time   |
-----------------------------------------------------------------------
----
|  0 | SELECT STATEMENT   |           |        |      |254
(100)|          |
|  1 |  SORT AGGREGATE    |           |      1 |     8 |
|          |
|* 2 |   TABLE ACCESS FULL| EMPLOYEES |  100K | 781K |254  (15)| 00:00:01 |
---------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------

   2 - filter("DEPARTMENT_ID"=:DEPT_ID)
```

The following query of `V$SQL` obtains information about the cursor:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHAREABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT               CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHAREABLE
---------------------- ------ ----- ---------- --------- ---------- ---------
SELECT COUNT(*), MAX(e      0     2       1329         Y          N         N
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      1     1        800         Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

The preceding output shows that the database created an additional child cursor (`CHILD#` of `1`). Cursor `0` is now marked as not shareable. Cursor `1` shows a number of buffers gets lower than cursor `0`, and is marked both bind-sensitive and bind-aware. A bind-aware cursor may use different plans for different bind values, depending on the selectivity of the predicates containing the bind variable.

Example 20-16 Bind-Aware Cursors: Choosing the Optimal Plan

This example continues the example in "Example 20-15". The following code executes the same `employees` query with the value of `10`, which has extremely low cardinality (only one row):

```
EXEC :dept_id := 10;
SELECT COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id
= :dept_id;

  COUNT(*) MAX(EMPLOYEE_ID)
---------- ----------------
         1              200
```

The following output shows that the optimizer picked the best plan, which is an index scan, based on the low cardinality estimate for the current bind value of `10`:

```
SQL> SELECT * from TABLE(DBMS_XPLAN.DISPLAY_CURSOR);

PLAN_TABLE_OUTPUT
-------------------------------------
SQL_ID    a9upgaqqj7bn5, child number 2
-------------------------------------
select COUNT(*), MAX(employee_id) FROM hr.employees WHERE department_id = :dept_id

Plan hash value: 1642965905

-------------------------------------------------------------------------------------
| Id| Operation                           | Name          |Rows|Bytes|Cost (%CPU)|Time |
-------------------------------------------------------------------------------------
| 0| SELECT STATEMENT                     |                   |   |   |2(100)|        |
| 1|  SORT AGGREGATE                      |                   |1  |8  |      |        |
| 2|   TABLE ACCESS BY INDEX ROWID BATCHED| EMPLOYEES         |1  |8  |2   (0)|00:00:01|
|*3|    INDEX RANGE SCAN                  | EMP_DEPARTMENT_IX | 1|   |1   (0)|00:00:01|
-------------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   3 - access("DEPARTMENT_ID"=:DEPT_ID)
```

The `V$SQL` output now shows that three child cursors exist:

```
SELECT SQL_TEXT, CHILD_NUMBER AS CHILD#, EXECUTIONS AS EXEC,
       BUFFER_GETS AS BUFF_GETS, IS_BIND_SENSITIVE AS BIND_SENS,
       IS_BIND_AWARE AS BIND_AWARE, IS_SHAREABLE AS SHAREABLE
FROM   V$SQL
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                CHILD#  EXEC  BUFF_GETS BIND_SENS BIND_AWARE SHAREABLE
---------------------- ------ ----- ----------- --------- ---------- ---------
SELECT COUNT(*), MAX(e      0     2       1329          Y          N         N
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      1     1        800          Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id

SELECT COUNT(*), MAX(e      2     1          3          Y          Y         Y
mployee_id) FROM hr.em
ployees WHERE departme
nt_id = :dept_id
```

The database discarded the original cursor (`CHILD#` of `0`) when the cursor switched to bind-aware mode. This is a one-time overhead. The database marked cursor `0` as not shareable (`SHAREABLE` is `N`), which means that this cursor is unusable and will be among the first to age out of the cursor cache.

> **See Also:**
>
> *Oracle Database Reference* to learn about `V$SQL`

20.3.5 Cursor Merging

If the optimizer creates a plan for a bind-aware cursor, and if this plan is the same as an existing cursor, then the optimizer can perform **cursor merging**.

In this case, the database merges cursors to save space in the library cache. The database increases the selectivity range for the cursor to include the selectivity of the new bind value.

When a query uses a new bind variable, the optimizer tries to find a cursor that it thinks is a good fit based on similarity in the selectivity of the bind value. If the database cannot find such a cursor, then it creates a new one. If the plan for the new cursor is the same as one of the existing cursors, then the database merges the two cursors to save space in the library cache. The merge results in the database marking one cursor as not sharable. If the library cache is under space pressure, then the database ages out the non-sharable cursor first.

> **See Also:**
>
> "Example 20-12"

20.3.6 Adaptive Cursor Sharing Views

You can use the `V$` views for adaptive cursor sharing to see selectivity ranges, cursor information (such as whether a cursor is bind-aware or bind-sensitive), and execution statistics.

Specifically, use the following views:

- `V$SQL` shows whether a cursor is bind-sensitive or bind-aware.
- `V$SQL_CS_HISTOGRAM` shows the distribution of the execution count across a three-bucket execution history histogram.
- `V$SQL_CS_SELECTIVITY` shows the selectivity ranges stored for every predicate containing a bind variable if the selectivity was used to check cursor sharing. It contains the text of the predicates, and the low and high values for the selectivity ranges.
- `V$SQL_CS_STATISTICS` summarizes the information that the optimizer uses to determine whether to mark a cursor bind-aware. For a sample of executions, the database tracks the rows processed, buffer gets, and CPU time. The `PEEKED` column shows `YES` when the bind set was used to build the cursor; otherwise, the value is `NO`.

See Also:

Oracle Database Reference to learn about `V$SQL` and its related views

20.4 Real-World Performance Guidelines for Cursor Sharing

The Real-World Performance team has created guidelines for how to optimize cursor sharing in Oracle database applications.

20.4.1 Develop Applications with Bind Variables for Security and Performance

The Real-World Performance group strongly suggests that all enterprise applications use bind variables.

Oracle Database applications were intended to be written with bind variables. Avoid application designs that result in large numbers of users issuing dynamic, unshared SQL statements.
Whenever Oracle Database fails to find a match for a statement in the library cache, it must perform a hard parse. Despite the dangers of developing applications with literals, not all real-world applications use bind variables. Developers sometimes find that it is faster and easier to write programs that use literals. However, decreased development time does not lead to better performance and security after deployment.

Video:

Video

The primary benefits of using bind variables are as follows:

- Resource efficiency

  Compiling a program before every execution does not use resources efficiently, but this is essentially what Oracle Database does when it performs a hard parse. The database server must expend significant CPU and memory to create cursors, generate and evaluate execution plans, and so on. By enabling the database to share cursors, soft parsing consumes far fewer resources. If an application uses literals instead of bind variables, but executes only a few queries each day, then DBAs may not perceive the extra overhead as a performance problem. However, if an application executes hundreds or thousands of queries per second, then the extra resource overhead can easily degrade performance to unacceptable levels. Using bind variables enables the database to perform a hard parse only once, no matter how many times the statement executes.

- Scalability

  When the database performs a hard parse, the database spends more time acquiring and holding latches in the shared pool and library cache. Latches are low-level serialization devices. The longer and more frequently the database latches structures in shared memory, the longer the queue for these latches becomes. When multiple statements share the same execution plan, the requests for latches and the durations of latches go down. This behavior increases scalability.

- Throughput and response time

  When the database avoids constantly reparsing and creating cursors, more of its time is spent in user space. The Real-World Performance group has found that changing literals to use binds often leads to orders of magnitude improvements in throughput and user response time.

  See the following video:

  > **Video:**
  >
  > Video

- Security

  The only way to prevent SQL injection attacks is to use bind variables. Malicious users can exploit application that concatenate strings by “injecting” code into the application.

> **See Also:**
>
> *Oracle Database PL/SQL Language Reference* for an example of an application that fixes a security vulnerability created by literals

20.4.2 Do Not Use CURSOR_SHARING = FORCE as a Permanent Fix

The best practice is to write sharable SQL and use the default of `EXACT` for `CURSOR_SHARING`.

However, for applications with many similar statements, setting `CURSOR_SHARING` to `FORCE` can sometimes significantly improve cursor sharing. The replacement of literals with system-generated bind values can lead to reduced memory usage, faster parses, and reduced latch contention. However, `FORCE` is not meant to be a permanent development solution.
As a general guideline, the Real-World Performance group recommends *against* setting `CURSOR_SHARING` to `FORCE` exception in rare situations, and then only when all of the following conditions are met:

- Statements in the shared pool differ only in the values of literals.
- Response time is suboptimal because of a very high number of library cache misses.
- Your existing code has a serious security and scalability bug—the absence of bind variables—and you need a *temporary* band-aid until the source code can be fixed.
- You set this initialization parameter at the session level and not at the instance level.

Setting `CURSOR_SHARING` to `FORCE` has the following drawbacks:

- It indicates that the application does not use user-defined bind variables, which means that it is open to SQL injection. Setting `CURSOR_SHARING` to `FORCE` does *not* fix SQL injection bugs or render the code any more secure. The database binds values only after any malicious SQL text has already been injected.

Video:

Video

- The database must perform extra work during the soft parse to find a similar statement in the shared pool.
- The database removes every literal, which means that it can remove useful information. For example, the database strips out literal values in `SUBSTR` and `TO_DATE` functions. The use of system-generated bind variables where literals are more optimal can have a negative impact on execution plans.
- There is an increase in the maximum lengths (as returned by `DESCRIBE`) of any selected expressions that contain literals in a `SELECT` statement. However, the actual length of the data returned does not change.
- Star transformation is not supported.

See Also:

- "CURSOR_SHARING and Bind Variable Substitution"
- *Oracle Database Reference* to learn about the `CURSOR_SHARING` initialization parameter

20.4.3 Establish Coding Conventions to Increase Cursor Reuse

By default, any variation in the text of two SQL statements prevents the database from sharing a cursor, including the names of bind variables. Also, changes in the size of bind variables can cause cursor mismatches. For this reason, using bind variables in application code is not enough to *guarantee* cursor sharing.

The Real-World Performance group recommends that you standardize spacing and capitalization conventions for SQL statements and PL/SQL blocks. Also establish conventions for the naming and definitions of bind variables. If the database does not share cursors as expected, begin your diagnosis by querying `V$SQL_SHARED_CURSOR`.

Example 20-17 Variations in SQL Text

In this example, an application that uses bind variables executes 7 statements using the same bind variable value, but the statements are not textually identical:

```
VARIABLE emp_id NUMBER
EXEC :emp_id := 101;

SELECT SUM(salary) FROM hr.employees WHERE employee_id < :emp_id;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :EMP_ID;
SELECT SUM(salary) FROM hr.employees WHERE employee_id < :Emp_Id;
SELECT SUM(salary)  FROM hr.employees WHERE employee_id < :emp_id;
select sum(salary) from hr.employees where employee_id < :emp_id;
```

```
Select sum(salary) From hr.employees Where employee_id < :emp_id;
Select sum(salary) From hr.employees Where employee_id< :emp_id;
```

A query of `V$SQLAREA` shows that no cursor sharing occurred:

```
COL SQL_TEXT FORMAT a35
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                            SQL_ID        VERSION_COUNT HASH_VALUE
----------------------------------- ------------- ------------- ----------
SELECT SUM(salary) FROM hr.employee bkrfu3ggu5315             1 3751971877
s WHERE employee_id < :EMP_ID
SELECT SUM(salary) FROM hr.employee 70mdtwh7xj9gv             1  265856507
s WHERE employee_id < :Emp_Id
Select sum(salary) From hr.employee 18tt4ny9u5wkt             1 2476929625
s Where employee_id< :emp_id
SELECT SUM(salary)  FROM hr.employe b6b21tbyaf8aq             1 4238811478
es WHERE employee_id < :emp_id
SELECT SUM(salary) FROM hr.employee 4318cbskba8yh             1  615850960
s WHERE employee_id < :emp_id
select sum(salary) from hr.employee 633zpx3xm71kj             1 4214457937
s where employee_id < :emp_id
Select sum(salary) From hr.employee 1mqbbbnsrrw08             1  830205960
s Where employee_id < :emp_id

7 rows selected.
```

Example 20-18   Bind Length Mismatch

The following code defines a bind variable with different lengths, and then executes textually identical statements with the same bind values:

```
VARIABLE lname VARCHAR2(20)
EXEC :lname := 'Taylor';
SELECT SUM(salary) FROM hr.employees WHERE last_name = :lname;
VARIABLE lname VARCHAR2(100)
EXEC :lname := 'Taylor';
SELECT SUM(salary) FROM hr.employees WHERE last_name = :lname;
```

The following query shows that the database did not share the cursor:

```
COL SQL_TEXT FORMAT a35
SELECT SQL_TEXT, SQL_ID, VERSION_COUNT, HASH_VALUE
FROM   V$SQLAREA
WHERE  SQL_TEXT LIKE '%mployee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%';

SQL_TEXT                            SQL_ID           VERSION_COUNT HASH_VALUE
----------------------------------- ------------- ------------- ----------
```

```
SELECT SUM(salary) FROM hr.employee buh8j4557r0h1             2
1249608193
s WHERE last_name = :lname
```

The reason is because of the bind lengths:

```
COL BIND_LENGTH_UPGRADEABLE FORMAT a15
SELECT s.SQL_TEXT, s.CHILD_NUMBER, 
       c.BIND_LENGTH_UPGRADEABLE
FROM   V$SQL s, V$SQL_SHARED_CURSOR c
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%' 
AND    s.CHILD_ADDRESS = c.CHILD_ADDRESS;

SQL_TEXT                            CHILD_NUMBER BIND_LENGTH_UPG
----------------------------------- ------------ ---------------
SELECT SUM(salary) FROM hr.employee            0 N
s WHERE last_name = :lname
SELECT SUM(salary) FROM hr.employee            1 Y
s WHERE last_name = :lname
```

20.4.4 Minimize Session-Level Changes to the Optimizer Environment

A best practice is to prevent users of the application from changing the optimization approach and goal for their individual sessions. Any changes to the optimizer environment can prevent otherwise identical statements from sharing cursors.

Example 20-19 Environment Mismatches

This example shows two textually identical statements that nevertheless do not share a cursor:

```
VARIABLE emp_id NUMBER

EXEC :emp_id := 110;

ALTER SESSION SET OPTIMIZER_MODE = FIRST_ROWS;
SELECT salary FROM hr.employees WHERE employee_id < :emp_id;
ALTER SESSION SET OPTIMIZER_MODE = ALL_ROWS;
SELECT salary FROM hr.employees WHERE employee_id < :emp_id;
```

A query of `V$SQL_SHARED_CURSOR` shows a mismatch in the optimizer modes:

```
SELECT S.SQL_TEXT, S.CHILD_NUMBER, s.CHILD_ADDRESS, 
       C.OPTIMIZER_MODE_MISMATCH
FROM   V$SQL S, V$SQL_SHARED_CURSOR C
WHERE  SQL_TEXT LIKE '%employee%'
AND    SQL_TEXT NOT LIKE '%SQL_TEXT%' 
AND S.CHILD_ADDRESS = C.CHILD_ADDRESS;

SQL_TEXT                            CHILD_NUMBER CHILD_ADDRESS    O
----------------------------------- ------------ ---------------- -
SELECT salary FROM hr.employees WHE            0 0000000080293040 N
```

```
RE employee_id < :emp_id
SELECT salary FROM hr.employees WHE            1 000000008644E888 Y
RE employee_id < :emp_id
```

Part VII

Monitoring and Tracing SQL

Use `DBMS_MONITOR` to track database operations, SQL Test Case Builder to package information relating to a performance problem, and SQL Trace to generate diagnostic data for problem SQL statements.

21 Monitoring Database Operations

This chapter describes how to monitor SQL and PL/SQL.

21.1 About Monitoring Database Operations

The SQL monitoring feature is enabled by default when the `STATISTICS_LEVEL` initialization parameter is either set to `TYPICAL` (the default value) or `ALL`.

See Also:

Oracle Database Concepts for a brief conceptual overview of database operations

21.1.1 About Database Operations

A database operation is a set of database tasks. A typical task might be a batch job or Extraction, Transformation, and Loading (ETL) processing job.

Database operations are either simple or composite.

Simple Database Operation

A **simple database operation** is a single SQL statement or PL/SQL subprogram. When the SQL Monitor feature is enabled, the database monitors simple database operations automatically when any of the following conditions is true:

- A SQL statement or PL/SQL subprogram has consumed at least 5 seconds of CPU or I/O time in a single execution.
- A SQL statement executes in parallel.
- A SQL statement specifies the `/*+ MONITOR */` hint.
- The event `sql_monitor` specifies a list of SQL IDs for the statements to be monitored. For example, the following statement forces instance-level monitoring for SQL IDs `5hc07qvt8v737` and `9ht3ba3arrzt3`:

  ```
  ALTER SYSTEM SET EVENTS 'sql_monitor [sql: 5hc07qvt8v737|sql:
  9ht3ba3arrzt3] force=true'
  ```

At each step of the SQL execution plan, the database tracks statistics by performance metrics such as elapsed time, CPU time, number of reads and writes, and I/O wait time. These metrics are available in a graphical and interactive report called the SQL monitor active report.

Composite Database Operation

A **composite database operation** is defined by the user. It includes all SQL statements or PL/SQL subprograms that execute within a database session, with the beginning and end points defined using the procedures `DBMS_SQL_MONITOR.BEGIN_OPERATION` and `DBMS_SQL_MONITOR.END_OPERATION`. A composite database operation is uniquely identified by its name and execution ID, and can be executed multiple times.

> **Note:**
>
> A database session can participate in at most one database operation at a time.

Oracle Database automatically monitors a composite operation when either of the following conditions is true:

- The operation has consumed at least 5 seconds of CPU or I/O time.
- Tracking for the operations is forced by setting `FORCED_TRACKING` to `Y` in `DBMS_SQL_MONITOR.BEGIN_OPERATION`.

> **Note:**
>
> "Getting the Most Out of SQL Monitor" for a brief overview of SQL Monitor

21.1.2 Purpose of Monitoring Database Operations

For simple operations, Real-Time SQL Monitoring helps determine where a statement is spending its time.

You can also see the breakdown of time and resource usage for recently completed statements. In this way, you can better determine why a particular operation is expensive. Use cases for Real-Time SQL Monitoring include the following:

- A frequently executed SQL statement is executing more slowly than normal. You must identify the root cause of this problem.
- A database session is experiencing slow performance.
- A parallel SQL statement is taking a long time. You want to determine how the server processes are dividing the work.

In OLTP and data warehouse environments, a job often logically groups related SQL statements. The job can span multiple concurrent sessions. Database operations extend Real-Time SQL Monitoring by enabling you to treat a set of statements or procedures as a named, uniquely identified, and re-executable unit. Use cases for monitoring operations include the following:

- A periodic batch job containing many SQL statements must complete in a certain number of hours, but took longer than expected.

- After a database upgrade, the execution time of an important batch job increased. To resolve this problem, you must collect enough relevant statistical data from the batch job before and after the upgrade, compare the two sets of data, and then identify the changes.
- Packing a SQL tuning set (STS) took far longer than anticipated. To diagnose the problem, you need to know what was being executed over time. Because this issue cannot be easily reproduced, you need to monitor the process while it is running.

Related Topics

- Generating and Accessing SQL Monitor Reports
  By default, AWR automatically captures SQL monitoring reports in XML format.
- *Oracle Database Administrator's Guide*

See Also:

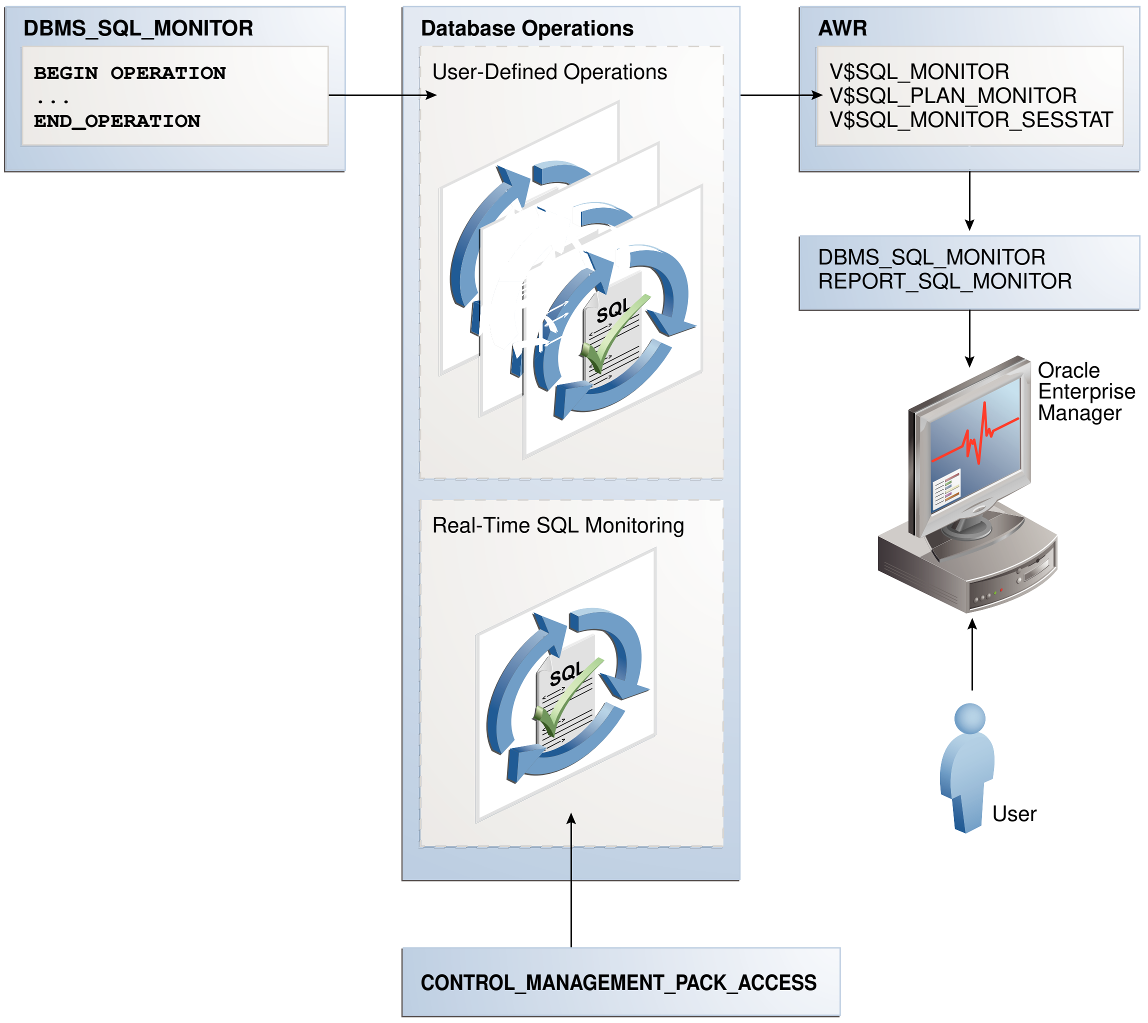
Why Use SQL Monitor for a video demonstrating some uses of SQL Monitor

21.1.3 How Database Monitoring Works

Real-Time SQL Monitoring is a built-in database infrastructure that helps you identify performance problems with long-running and parallel SQL statements.

The following figure gives an overview of the architecture for Real-Time SQL Monitoring.

Figure 21-1 Architecture for Database Operations Monitoring



As shown in the preceding graphic, the `DBMS_SQL_MONITOR` package defines database operations. After monitoring is initiated, the database stores metadata about the database operations in AWR, and the data in AWR and ASH. The database refreshes monitoring statistics in close to real time as each monitored statement executes, typically once every second. The database stores the operational data (the statements and metadata) in the SGA. After an operation completes, the database writes the SQL Monitor report to disk, where it can be queried using the `DBA_HIST_REPORTS` view.

Every monitored database operation has an entry in the `V$SQL_MONITOR` view. This entry tracks key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time, and various other wait times. The `V$SQL_PLAN_MONITOR` view includes monitoring statistics for each operation in the execution plan of the SQL statement being monitored. You can access reports by using `DBMS_SQL_MONITOR.REPORT_SQL_MONITOR`, Oracle Enterprise Manager Cloud Control (Cloud Control).

See Also:

- "Generating and Accessing SQL Monitor Reports"
- *Oracle Database Reference* to learn about `V$SQL_MONITOR`, `V$SQL_PLAN_MONITOR`, and `CONTROL_MANAGEMENT_PACK_ACCESS`
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQL_MONITOR` package

21.1.4 User Interfaces for Database Operations Monitoring

Real-Time SQL Monitoring is a feature of the Oracle Database Tuning Pack. Database operations are enabled when the `CONTROL_MANAGEMENT_PACK_ACCESS` initialization parameter is set to `DIAGNOSTIC+TUNING` (default).

21.1.4.1 Monitored SQL Executions Page in Cloud Control

The Monitored SQL Executions page in Cloud Control, also known as SQL Monitor, displays details of SQL execution. SQL Monitor is the recommended interface for reporting on database operations.

Statistics at each step of the execution plan are tracked by key performance metrics, including elapsed time, CPU time, number of reads and writes, I/O wait time, and various other wait times. These metrics enable DBAs to analyze SQL execution in depth and decide on the most appropriate tuning strategies for monitored SQL statements.

SQL Monitor Active Reports provide a flash-based interactive report that enables you to save data in an HTML file. You can save this file and view it offline.

21.1.4.1.1 Accessing the Monitored SQL Executions Page

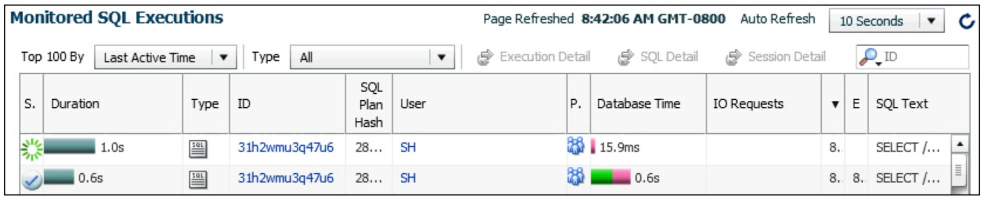
The Monitored SQL Executions shows information such as the SQL ID, database time, and I/O requests.

To access the Monitored SQL Executions page:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Performance** menu, select **SQL Monitoring**.

   The Monitored SQL Executions page appears.

Figure 21-2   Monitored SQL Executions



Monitored SQL Executions   Page Refreshed **8:42:06 AM GMT-0800**   Auto Refresh 10 Seconds

Top 100 By Last Active Time   Type All   Execution Detail   SQL Detail   Session Detail   ID

S.	Duration	Type	ID	SQL Plan Hash	User	P.	Database Time	IO Requests	▼	E	SQL Text
	1.0s		31h2wmu3q47u6	28...	SH		15.9ms		8.		SELECT /...
✓	0.6s		31h2wmu3q47u6	28...	SH		0.6s		8.	8.	SELECT /...

21.1.4.2 DBMS_SQL_MONITOR Package

The DBMS_SQL_MONITOR package defines the beginning and ending of a composite database operation, and generates a report of the database operations.

Table 21-1   DBMS_SQL_MONITOR

Subprogram	Description
BEGIN_OPERATION	This function starts a database operation in the current session. This function associates a session with a database operation. Starting in Oracle Database 12c Release 2 (12.2), you can use session_id and session_num to indicate the session in which to start monitoring.
END_OPERATION	This function ends a database operation in the current session. If the specified database operation does not exist, then this function has no effect.
REPORT_SQL_MONITOR	This function builds a detailed report with monitoring information for a SQL statement, PL/SQL block, or database operation. For each operation, it gives key information and associated global statistics. Use this function to get detailed monitoring information for a database operation. The target database operation for this report can be: • The last database operation monitored by Oracle Database (default, no parameter). • The last database operation executed in the specified session and monitored by Oracle Database. The session is identified by its session ID and optionally its serial number (-1 is current session). • The last execution of a specific database operation identified by its sql_id. • A specific execution of a database operation identified by the combination sql_id, sql_exec_start, and sql_exec_id. • The last execution of a specific database operation identified by dbop_name. • The specific execution of a database operation identified by the combination dbop_name, dbop_exec_id. Use the type parameter to specify the output type: TEXT (default), HTML, ACTIVE, or XML.
REPORT_SQL_MONITOR_XML	This function is identical to the REPORT_SQL_MONITOR function, except that the return type is XMLType.

Table 21-1 (Cont.) DBMS_SQL_MONITOR

Subprogram	Description
`REPORT_SQL_MONITOR_LIST`	This function builds a report for all or a subset of database operations that have been monitored by Oracle Database.
`REPORT_SQL_MONITOR_LIST_XML`	This function is identical to the `REPORT_SQL_MONITOR_LIST` function, except that it returns `XMLType`.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_SQL_MONITOR` package

21.1.4.3 Attributes of composite Database Operations

The `DBMS_SQL_MONITOR.BEGIN_OPERATION` function defines a database operation.

A composite database operation is uniquely identified by the following information:

- Database operation name

  This is a user-created name such as `daily_sales_report`. The operation name is the same for a job even if it is executed concurrently by different sessions or on different databases. Database operation names do not reside in different namespaces.

- Database operation execution ID

  Two or more occurrences of the same database operation can run at the same time, with the same name but different execution IDs. This numeric ID uniquely identifies different executions of the *same* database operation.

  The database automatically creates an execution ID when you begin a database operation. You can also specify a user-created execution ID.

Optionally, you can specify the session ID and session serial number in which to start the database operations. Thus, one database session can start a database operation defined in a different database session.

The database uses the following triplet of values to identify each SQL and PL/SQL statement monitored in the `V$SQL_MONITOR` view, regardless of whether the statement is included in a database operation:

- SQL identifier to identify the SQL statement (`SQL_ID`)
- Start execution timestamp (`SQL_EXEC_START`)
- An internally generated identifier to ensure that this primary key is truly unique (`SQL_EXEC_ID`)

You can use zero or more additional attributes to describe and identify the characteristics of a composite database operation. Every attribute has a name and value. For example, for database operation `daily_sales_report`, you might define the attribute `db_name` and assign it the value `prod`.

Related Topics

- *Oracle Database Reference*
- *Oracle Database PL/SQL Packages and Types Reference*

21.1.4.4 MONITOR and NO_MONITOR Hints

You can use the MONITOR and NO_MONITOR hints to control tracking for individual statements.

The `MONITOR` hint forces real-time SQL monitoring for the query, even if the statement is not long-running. This hint is valid only when the parameter `CONTROL_MANAGEMENT_PACK_ACCESS` is set to `DIAGNOSTIC+TUNING`. The following query forces SQL Monitor to enable tracking:

```
SELECT /*+ MONITOR */  prod_id, AVG(amount_sold), AVG(quantity_sold)
FROM   sales
GROUP BY prod_id
ORDER BY prod_id;
```

The `NO_MONITOR` hint disables real-time SQL monitoring for the query, even if the query is long running. The following query forces SQL Monitor to disable tracking:

```
SELECT /*+ NO_MONITOR */  prod_id, AVG(amount_sold),
AVG(quantity_sold)
FROM   sales
GROUP BY prod_id
ORDER BY prod_id;
```

Note:

Oracle Database SQL Language Reference to learn about the `MONITOR` and `NO_MONITOR` hints

21.1.4.5 Views for Monitoring and Reporting on Database Operations

You can obtain the statistics for database operations using `V$` and data dictionary view.

The following table summarizes these views.

Table 21-2 Views for Database Operations Monitoring

View	Description
DBA_HIST_REPORTS	This view displays metadata about XML reports captured in Automatic Workload Repository (AWR). Each XML report contains details about some activity of a component. For example, a SQL Monitor report contains a detailed report about a particular database operation. Important columns include: • The REPORT_SUMMARY column contains the summary of the report. • The COMPONENT_NAME column accepts the value sqlmonitor. • The REPORT_ID column provides the ID of the report, which you can specify in the RID parameter of DBMS_AUTO_REPORT.REPORT_REPOSITORY_DETAIL when generating the report. • The KEY1 column is the is the SQL ID for the statement. • The KEY2 column is the SQL execution ID for the statement AWR controls the retention period for SQL Monitor reports. Every SQL Monitor report in DBA_HIST_REPORTS is associated with an AWR SNAP_ID. Note that SQL Monitor reports are not exported or imported when you export or import the corresponding AWR data.
DBA_HIST_REPORTS_DETAILS	This view displays details about each report captured in AWR. Metadata for each report appears in the DBA_HIST_REPORTS view, whereas the actual report is available in the DBA_HIST_REPORTS_DETAILS view.
V$SQL_MONITOR	This view contains global, high-level information about simple and composite database operations. For simple database operations, monitoring statistics are not cumulative over several executions. In this case, one entry in V$SQL_MONITOR is dedicated to a single execution of a SQL statement. If the database monitors two executions of the same SQL statement, then each execution has a separate entry in V$SQL_MONITOR. For simple database operations, V$SQL_MONITOR has one entry for the parallel execution coordinator process and one entry for each parallel execution server process. Each entry has corresponding entries in V$SQL_PLAN_MONITOR. Because the processes allocated for the parallel execution of a SQL statement are cooperating for the same execution, these entries share the same execution key (the combination of SQL_ID, SQL_EXEC_START, and SQL_EXEC_ID). For composite database operations, each row contains an operation whose statistics are accumulated over the SQL statements and PL/SQL subprograms that run in the same session as part of the operation. The primary key is the combination of the columns DBOP_NAME and DBOP_EXEC_ID.

Table 21-2 (Cont.) Views for Database Operations Monitoring

View	Description
`V$SQL_MONITOR_SESSTAT`	This view contains the statistics for all sessions involved in the database operation. Most of the statistics are cumulative. The database stores the statistics in XML format instead of using each column for each statistic. This view is primarily intended for the report generator. Oracle recommends that you use `V$SESSTAT` instead of `V$SQL_MONITOR_SESSTAT`.
`V$SQL_PLAN_MONITOR`	This view contains monitoring statistics for each step in the execution plan of the monitored SQL statement. The database updates statistics in `V$SQL_PLAN_MONITOR` every second while the SQL statement is executing. Multiple entries exist in `V$SQL_PLAN_MONITOR` for every monitored SQL statement. Each entry corresponds to a step in the execution plan of the statement.

You can use the preceding `V$` views with the following views to get additional information about the monitored execution:

- `V$ACTIVE_SESSION_HISTORY`
- `V$SESSION`
- `V$SESSION_LONGOPS`
- `V$SQL`
- `V$SQL_PLAN`

See Also:

Oracle Database Reference to learn about the `V$` views for database operations monitoring

21.1.5 Basic Tasks in Database Operations Monitoring

This section explains the basic tasks in database operations monitoring.

Basic tasks are as follows:

- "Enabling and Disabling Monitoring of Database Operations"

  This task explains how you can enable automatic monitoring of database operations at the system and statement level.
- "Defining a Composite Database Operation"

  This section explains how you can define the beginning and end of a database operation using PL/SQL.
- "Generating and Accessing SQL Monitor Reports"

This section explains how you can generate and interpret reports on a database operation.

21.2 Enabling and Disabling Monitoring of Database Operations

Use initialization parameters to enable or disable monitoring.

21.2.1 Enabling Monitoring of Database Operations at the System Level

The SQL monitoring feature is enabled by default when the `STATISTICS_LEVEL` initialization parameter is either set to `TYPICAL` (the default value) or `ALL`. SQL monitoring starts automatically for all long-running queries.

Prerequisites

Because SQL monitoring is a feature of the Oracle Database Tuning Pack, the `CONTROL_MANAGEMENT_PACK_ACCESS` initialization parameter must be set to `DIAGNOSTIC+TUNING` (the default value).

Assumptions

This tutorial assumes the following:

- The `STATISTICS_LEVEL` initialization parameter is set to `BASIC`.
- You want to enable automatic monitoring of database operations.

To enable monitoring of database operations:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the current database operations settings.

   For example, run the following SQL*Plus command:

   ```
   SQL> SHOW PARAMETER statistics_level

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   statistics_level                     string      BASIC
   ```

2. Set the statistics level to `TYPICAL`.

   For example, run the following SQL statement:

   ```
   SQL> ALTER SYSTEM SET STATISTICS_LEVEL='TYPICAL';
   ```

See Also:

Oracle Database Reference to learn about the `STATISTICS_LEVEL` and `CONTROL_MANAGEMENT_PACK_ACCESS` initialization parameter

21.2.2 Enabling and Disabling Monitoring of Database Operations at the Statement Level

When the `CONTROL_MANAGEMENT_PACK_ACCESS` initialization parameter is set to `DIAGNOSTIC+TUNING`, you can use hints to enable or disable monitoring of specific SQL statements.

The database monitors SQL statements or PL/SQL subprograms automatically when they have consumed at least 5 seconds of CPU or I/O time in a single execution. The `MONITOR` hint is useful to enforce monitoring of statements or subprograms that do not meet the time criteria.

Two statement-level hints are available to force or prevent the database from monitoring a SQL statement. To force SQL monitoring, use the `MONITOR` hint:

```
SELECT /*+ MONITOR */ SYSDATE FROM DUAL;
```

This hint is effective only when the `CONTROL_MANAGEMENT_PACK_ACCESS` parameter is set to `DIAGNOSTIC+TUNING`. To prevent the hinted SQL statement from being monitored, use the `NO_MONITOR` reverse hint.

Assumptions

This tutorial assumes the following:

- Database monitoring is currently enabled at the system level.
- You want to disable automatic monitoring for the statement `SELECT * FROM sales ORDER BY time_id`.

To disable monitoring of database operations for a SQL statement:

1. Execute the query with the `NO_MONITOR` hint.

   For example, run the following statement:

   ```
   SQL> SELECT * /*+NO_MONITOR*/ FROM sales ORDER BY time_id;
   ```

See Also:

Oracle Database SQL Language Reference for information about using the `MONITOR` and `NO_MONITOR` hints

21.3 Defining a Composite Database Operation

Defining a database operation involves supplying a name and specifying its beginning and end times.

Start a database operation by using the `DBMS_SQL_MONITOR.BEGIN_OPERATION` function, and end it by using the `DBMS_SQL_MONITOR.END_OPERATION` procedure.

To begin the operation in a different session, specify the combination of `session_id` and `serial_num`. The `BEGIN_OPERATION` function returns the database operation execution ID. If `dbop_exec_id` is null, then the database generates a unique value.

A single namespace exists for database operations, which means that name collisions are possible. Oracle recommends the following naming convention: *`component_name.subcomponent_name.operation name`*. For operations inside the database, Oracle recommends using `ORA` for the component name. For example, a materialized view refresh could be named `ORA.MV.refresh`. An E-Business Suite payroll function could be named `EBIZ.payroll`.

To create a database operation in the current session:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Start the operation by using `DBMS_SQL_MONITOR.BEGIN_OPERATION`.

   This function returns the database operation execution ID. The following example creates the operation named `ORA.sales.agg`, and stores the execution ID in a SQL*Plus variable:

   ```
   VARIABLE exec_id NUMBER;
   BEGIN
     :exec_id := DBMS_SQL_MONITOR.BEGIN_OPERATION ( dbop_name =>
   'ORA.sales.agg' );
   END;
   /
   ```

3. Execute the SQL statements or PL/SQL programs that you want to monitor.
4. End the operation by using `DBMS_SQL_MONITOR.END_OPERATION`.

   The following example ends operation `ORA.sales.agg`:

   ```
   BEGIN
     DBMS_SQL_MONITOR.END_OPERATION ( dbop_name => 'ORA.sales.agg', dbop_eid
   => :exec_id );
   END;
   /
   ```

Example 21-1   Creating a Database Operation

The following example illustrates how to use the `DBMS_SQL_MONITOR` package to begin and end a database operation in a different session. This example assumes the following:

- You are an administrator and want to monitor statements in a session started by user `sh`.
- You want to monitor queries of the `sh.sales` table and `sh.customers` table.
- You want these two queries to be monitored as a database operation named `sh_count`.

Table 21-3 Creating a Database Operation

SYSTEM Session	SH Session	DESCRIPTION

`SQL> CONNECT SYSTEM`
`Enter password: *********`
`Connected.`
	n/a	Start SQL*Plus and connect as a user with the administrator privileges.

n/a	`SQL> CONNECT sh`
	`Enter password: ******`
	`Connected.`
		In a different terminal, start SQL*Plus and connect as a user as user `sh`.

`SELECT SID, SERIAL#`
`FROM   V$SESSION`
`WHERE  USERNAME = 'SH';`
 	
`       SID    SERIAL#`
`---------- ----------`
`       121      13397`
	n/a	In the `SYSTEM` session, query the session ID and serial number of the `sh` session.

`VARIABLE eid NUMBER`
 	
`BEGIN`
`:eid:=DBMS_SQL_MONITOR.BEGIN_OPERATION`
`      ('sh_count', null, null,`
`       null, '121', '13397');`
`END;`
`/`
 	
`PRINT eid`
 	
`       EID`
`----------`
`         2`
	n/a	In the `SYSTEM` session, begin a database operation, specifying the session ID and serial number for the `sh` session.

Table 21-3 (Cont.) Creating a Database Operation

SYSTEM Session	SH Session	DESCRIPTION

n/a	`SELECT count(*)`
	`FROM   sh.sales;`
	
	`  COUNT(*)`
	`----------`
	`    918843`
	
	`SELECT COUNT(*)`
	`FROM   sh.customers;`
	
	`  COUNT(*)`
	`----------`
	`     55500`
		In the `sh` session, query the `sales` and `customers` tables. These SQL queries are part of the `sh_count` operation.

`BEGIN`
`  DBMS_SQL_MONITOR.END_OPERATION`
`        ('sh_count',:eid);`
`END;`
`/`
	n/a	End the database operation by specifying the operation name and execution ID.

`COL DBOP_NAME FORMAT a10`
`COL STATUS FORMAT a10`
`COL ID FORMAT 999`
 	
`SELECT DBOP_NAME, DBOP_EXEC_ID AS ID,`
`       STATUS, CPU_TIME, BUFFER_GETS`
`FROM   V$SQL_MONITOR`
`WHERE DBOP_NAME IS NOT NULL`
`      ORDER BY DBOP_EXEC_ID;`
 	
`DBOP_NAME   ID     STATUS CPU_TIME GETS`
`---------- -- ---------- -------- ----`
`sh_count     1  EXECUTING    24997   65`
	n/a	Query the metadata for the `sh_count` database operation. The status of the operation is `EXECUTING` because the session has not picked up the new session status.

n/a	`SELECT SYSDATE FROM`
	`DUAL;`
		To collect changed session information, execute a query that performs a round trip to the database.

Table 21-3 (Cont.) Creating a Database Operation

SYSTEM Session	SH Session	DESCRIPTION

```
COL DBOP_NAME FORMAT a10
COL STATUS FORMAT a10
COL ID FORMAT 999

SELECT DBOP_NAME, DBOP_EXEC_ID AS ID,
       STATUS, CPU_TIME, BUFFER_GETS
FROM   V$SQL_MONITOR
WHERE DBOP_NAME IS NOT NULL
      ORDER BY DBOP_EXEC_ID;

DBOP_NAME  ID     STATUS CPU_TIME GETS
---------- -- ---------- -------- ----
sh_count    1       DONE    24997   65
```
	n/a	The status of the operation is now updated to `DONE`.

Related Topics

- *Oracle Database PL/SQL Packages and Types Reference*

21.4 Generating and Accessing SQL Monitor Reports

By default, AWR automatically captures SQL monitoring reports in XML format.

The reports capture only SQL statements that are not executing or queued and have finished execution since the last capture cycle. AWR captures reports only for the most expensive statements according to elapsed execution time. The SQL Monitor retention policy is controlled by the AWR policy. You can change the retention policy using the `DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS` procedure.

The Monitored SQL Executions page in Enterprise Manager Cloud Control (Cloud Control) summarizes the activity for monitored statements. You can use this page to drill down and obtain additional details about particular statements. The Monitored SQL Executions Details page uses data from several views, including the following:

- `GV$SQL_MONITOR`
- `GV$SQL_PLAN_MONITOR`
- `GV$SQL_MONITOR_SESSTAT`
- `GV$SQL`
- `GV$SQL_PLAN`
- `GV$ACTIVE_SESSION_HISTORY`
- `GV$SESSION_LONGOPS`
- `DBA_HIST_REPORTS`
- `DBA_HIST_REPORTS_DETAILS`

Note:

Starting in Oracle Database 19c, Oracle Database includes undocumented `V$` views that enable a database user *without* the `SELECT_CATALOG_ROLE` to see the plans and statistics for simple database operations (individual SQL and PL/SQL statements) executed within the session. A user without `SELECT_CATALOG_ROLE` cannot see SQL execution statistics and details for other users.

Assumptions

This tutorial assumes the following:

- The user `sh` is executing the following long-running parallel query of the sales made to each customer:

  ```
  SELECT c.cust_id, c.cust_last_name, c.cust_first_name,
         s.prod_id, p.prod_name, s.time_id
  FROM   sales s, customers c, products p
  WHERE  s.cust_id = c.cust_id
  AND    s.prod_id = p.prod_id
  ORDER BY c.cust_id, s.time_id;
  ```

- You want to ensure that the preceding query does not consume excessive resources. While the statement executes, you want to determine basic statistics about the database operation, such as the level of parallelism, the total database time, and number of I/O requests.
- You use Cloud Control to monitor statement execution.

  Note:

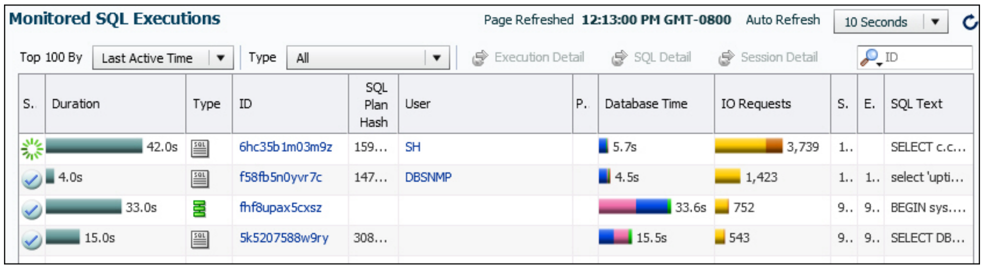
  To generate the SQL monitor report from the command line, run the `REPORT_SQL_MONITOR` function in the `DBMS_SQLTUNE` package, as in the following sample SQL*Plus script:

  ```
  VARIABLE my_rept CLOB
  BEGIN
    :my_rept :=DBMS_SQLTUNE.REPORT_SQL_MONITOR();
  END;
  /
  PRINT :my_rept
  ```

To monitor SQL executions:

1. Access the Monitored SQL Executions page, as described in "Monitored SQL Executions Page in Cloud Control".

   In the following graphic, the top row shows the parallel query.



Monitored SQL Executions — Page Refreshed 12:13:00 PM GMT-0800 — Auto Refresh: 10 Seconds

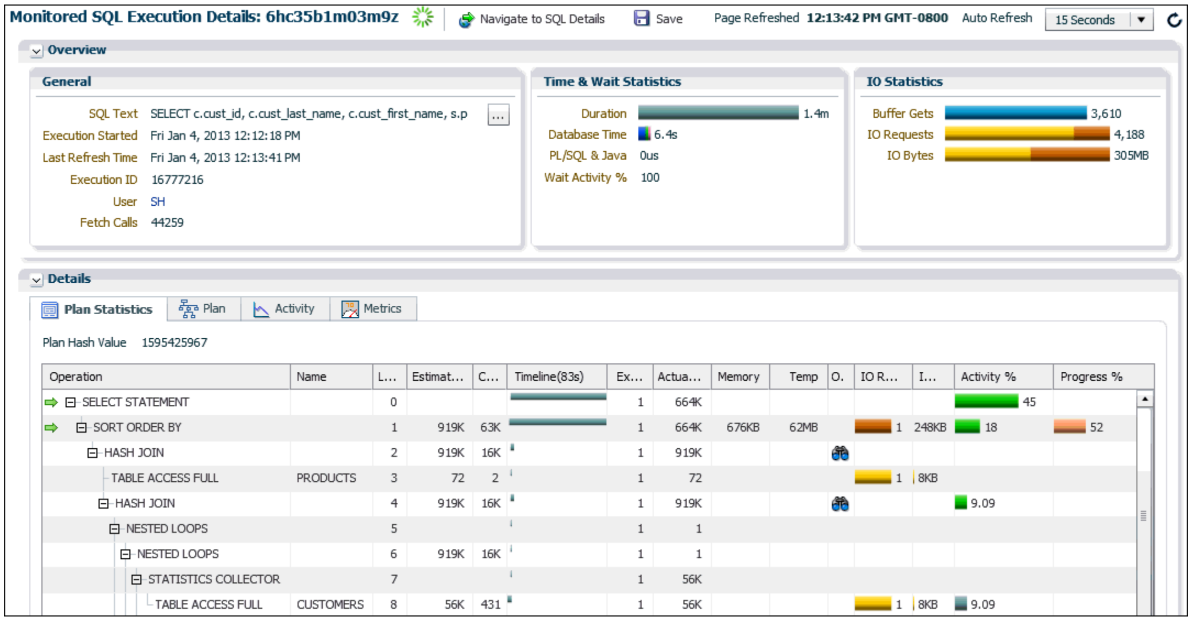
Top 100 By: Last Active Time — Type: All

S.	Duration	Type	ID	SQL Plan Hash	User	P.	Database Time	IO Requests	S.	E.	SQL Text
	42.0s		6hc35b1m03m9z	159...	SH		5.7s	3,739	1..		SELECT c.c...
✓	4.0s		f58fb5n0yvr7c	147...	DBSNMP		4.5s	1,423	1..	1..	select 'upti...
✓	33.0s		fhf8upax5cxsz				33.6s	752	9..	9..	BEGIN sys....
✓	15.0s		5k5207588w9ry	308...			15.5s	543	9..	9..	SELECT DB...

In this example, the query has been executing for 1.4 minutes.

2. Click the value in the SQL ID column to see details about the statement.

   The Monitored SQL Details page appears.



Monitored SQL Execution Details: 6hc35b1m03m9z — Navigate to SQL Details — Save — Page Refreshed 12:13:42 PM GMT-0800 — Auto Refresh: 15 Seconds

Overview

General

SQL Text	SELECT c.cust_id, c.cust_last_name, c.cust_first_name, s.p
Execution Started	Fri Jan 4, 2013 12:12:18 PM
Last Refresh Time	Fri Jan 4, 2013 12:13:41 PM
Execution ID	16777216
User	SH
Fetch Calls	44259

Time & Wait Statistics

Duration	1.4m
Database Time	6.4s
PL/SQL & Java	0us
Wait Activity %	100

IO Statistics

Buffer Gets	3,610
IO Requests	4,188
IO Bytes	305MB

Details

Plan Statistics | Plan | Activity | Metrics

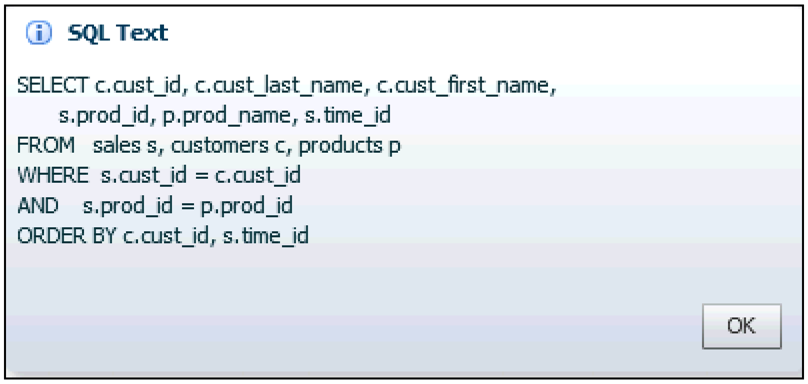
Plan Hash Value 1595425967

Operation	Name	L...	Estimat...	C...	Timeline(83s)	Ex...	Actua...	Memory	Temp	O.	IO R...	I...	Activity %	Progress %
SELECT STATEMENT		0				1	664K						45	
SORT ORDER BY		1	919K	63K		1	664K	676KB	62MB		1	248KB	18	52
HASH JOIN		2	919K	16K		1	919K							
TABLE ACCESS FULL	PRODUCTS	3	72	2		1	72				1	8KB		
HASH JOIN		4	919K	16K		1	919K						9.09	
NESTED LOOPS		5				1	1							
NESTED LOOPS		6	919K	16K		1	1							
STATISTICS COLLECTOR		7				1	56K							
TABLE ACCESS FULL	CUSTOMERS	8	56K	431		1	56K				1	8KB	9.09	

The preceding report shows the execution plan and statistics relating to statement execution. For example, the Timeline column shows when each step of the execution plan was active. Times are shown relative to the beginning and end of the statement execution. The Executions column shows how many times an operation was executed.

3. In the Overview section, click the link next to the SQL text.

   A message shows the full text of the SQL statement.

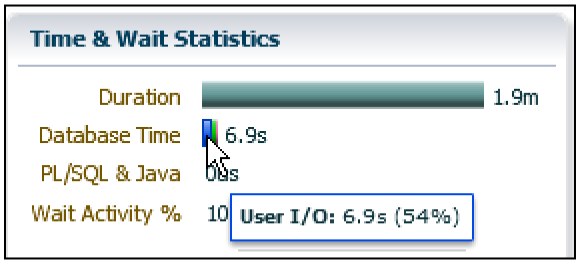


SQL Text

```
SELECT c.cust_id, c.cust_last_name, c.cust_first_name,
    s.prod_id, p.prod_name, s.time_id
FROM   sales s, customers c, products p
WHERE  s.cust_id = c.cust_id
AND    s.prod_id = p.prod_id
ORDER BY c.cust_id, s.time_id
```

4. In the Time & Wait Statistics section, next to Database Time, move the cursor over the largest portion on the bar graph.

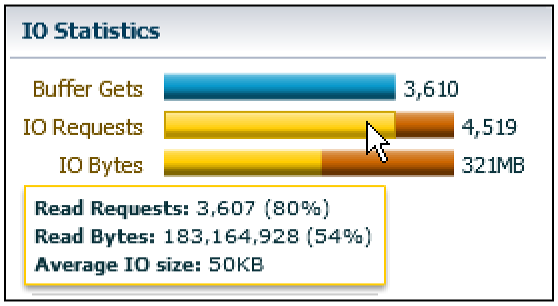
   A message shows that user I/O is consuming over half of database time.



   Database Time measures the amount of time the database has spent working on this SQL statement. This value includes CPU and wait times, such as I/O time. The bar graph is divided into several color-coded portions to highlight CPU resources, user I/O resources, and other resources. You can move the cursor over any portion to view the percentage value of the total.

5. In the Details section, in the IO Requests column, move the cursor over the I/O requests bar to view the percentage value of the total.

   A message appears.



Read Requests: 3,607 (80%)
Read Bytes: 183,164,928 (54%)
Average IO size: 50KB

   In the preceding graphic, the IO Requests message shows the total number of read requests issued by the monitored SQL. The message shows that read requests form 80% of the total I/O requests.

See Also:

- Cloud Control Online Help for descriptions of the elements on the Monitored SQL Executions Details page, and for complete descriptions of all statistics in the report.
- *Oracle Database Reference* to learn about `V$SQL_MONITOR` and related views for database operations monitoring
- Why Use SQL Monitor for a video demonstrating useful features of the Monitored SQL Details report

21.5 Monitoring Database Operations: Scenarios

In these scenarios, you report on both simple and composite database operations.

21.5.1 Reporting on a Simple Database Operation: Scenario

In this scenario, a query is expected to complete in seconds, but is continuing to execute.

In this example, assume that you log in to the database as user `sh`, and then run the following query:

```
SELECT /*+ MONITOR */ s.prod_id, c.cust_last_name FROM sales s,
customers c ORDER BY prod_id
```

The query is not completing. Starting in Oracle Database 19c, low-privileged users such as `sh` can generate a SQL Monitor report for simple database operations (individual SQL and PL/SQL statements) in their session. To identify the source of the problem, you use SQL Monitor for diagnosis as follows:

1. Cancel the query.
2. Obtain a text report by invoking `DBMS_SQL_MONITOR.REPORT_SQL_MONITOR`:

   ```
   SET LONG 1000000
   VARIABLE my_rept CLOB;
   BEGIN
     :my_rept := DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
                   report_level => 'ALL',
                   TYPE         => 'text');
   END;
   /
   PRINT :my_rept
   ```

   Partial sample output appears below:

```
SQL Text
------------------------------
SELECT /*+ MONITOR */ s.prod_id, c.cust_last_name FROM sales s, customers c
ORDER BY prod_id
```

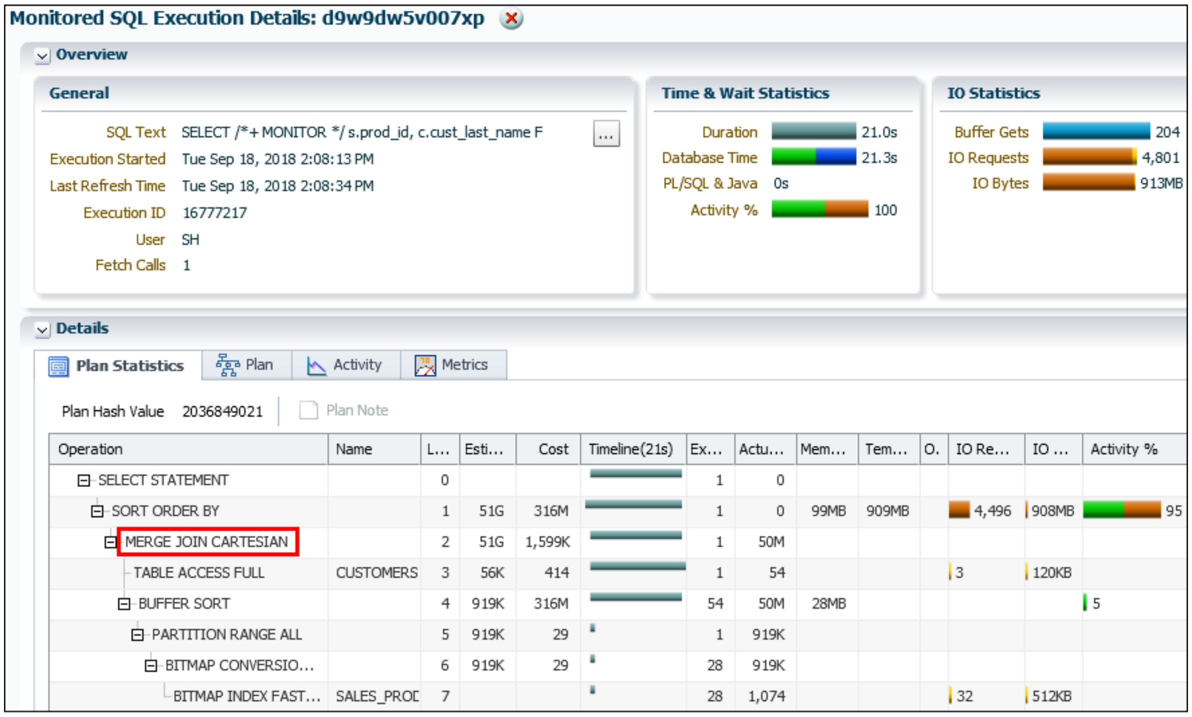
```
Global Information
------------------------------
 Status              :  DONE (ERROR)
 Instance ID         :  1
 Session             :  SH (42:3617)
 SQL ID              :  d9w9dw5v007xp
 SQL Execution ID    :  16777217
 Execution Started   :  09/18/2018 14:08:13
 First Refresh Time  :  09/18/2018 14:08:13
 Last Refresh Time   :  09/18/2018 14:08:34

MY_REPT
------------------------------------------------------------------------
 Duration            :  21s
 Module/Action       :  SQL*Plus/-
 Service             :  SYS$USERS
 Program             :  sqlplus@slc16iva (TNS V1-V3)
 Fetch Calls         :  1
Global Stats

MY_REPT
------------------------------------------------------------------------
| Time(s)|Time(s)|Waits(s)  |Calls | Gets |Reqs | Bytes | Reqs  | Bytes |
========================================================================
|   21 |   11 |      10 |      1 |    204 |  233 |    3MB |  4568 | 909MB |
========================================================================


SQL Plan Monitoring Details (Plan Hash Value=2036849021)
========================================================================

MY_REPT
------------------------------------------------------------------------
========================================================================
| Id |             Operation             |      Name       |  Rows   | Cost |
Time     | Start  | Execs |    Rows    | Read | Read   | Write | Write |   Mem   | Tem
p  | Activity | Activity Detail |
|    |                                   |                 | (Estim) |      | Ac
tive(s) | Active |       | (Actual) | Reqs | Bytes | Reqs  | Bytes |  (Max)  |  (Ma
x) |    (%)     |    (# samples)    |
================================================================================
================================================================================
=================================

MY_REPT
--------------------------------------------------------------------------------
| 0  | SELECT STATEMENT                  |                 |         |      |
     19 |      +2 |      1 |         0 |      |        |       |       |     . |
 . |          |          |
| 1 |   SORT ORDER BY                    |                 |     51G |316M |
     20 |      +1 |      1 |         0 |      |        |  4496 | 908MB |   99MB | 909
MB |          |          |
| 2 |    MERGE JOIN CARTESIAN            |                 |     51G |   2M |
     19 |      +2 |      1 |       50M |      |        |       |       |     . |
 . |          |          |
| 3 |     TABLE ACCESS FULL              | CUSTOMERS       |   55500 | 414 |
     20 |      +2 |      1 |        54 |    3 | 120KB |       |       |     . |
 . |          |          |
| 4 |     BUFFER SORT                    |                 |    919K |316M |
     19 |      +2 |     54 |       50M |      |        |       |       |   28MB |
 . |          |          |
| 5 |      PARTITION RANGE ALL           |                 |    919K |   29 |
      1 |      +2 |      1 |      919K |      |        |       |       |     . |
 . |          |          |
| 6 |       BITMAP CONVERSION TO ROWIDS  |                 |    919K |   29 |
      1 |      +2 |     28 |      919K |      |        |       |       |     . |
 . |          |          |
| 7 |        BITMAP INDEX FAST FULL SCAN | SALES_PROD_BIX  |         |      |
      1 |      +2 |     28 |      1074 |   32 | 512KB |       |       |     . |
 . |          |         |
================================================================================
```

Because of the formatting, the preceding output is difficult to read. You decide to create an active SQL Monitor report, which is graphical.

3. Create a SQL script containing the following commands:

```
SET FEEDBACK OFF
SET TERMOUT OFF
SET TRIMSPOOL ON
SET TRIM ON
SET PAGES 0
SET LINESIZE 1000
SET LONG 1000000
SET LONGCHUNKSIZE 1000000

SPOOL /tmp/long_sql.htm
SELECT DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
         report_level => 'ALL',
         TYPE         => 'active')
FROM   DUAL;
SPOOL OFF
```

4. In SQL*Plus, run the SQL script that you created in the preceding step.
5. Open the output HTML file in a browser, and then review the report:



Monitored SQL Execution Details: d9w9dw5v007xp

Overview

General

SQL Text	SELECT /*+ MONITOR */ s.prod_id, c.cust_last_name F
Execution Started	Tue Sep 18, 2018 2:08:13 PM
Last Refresh Time	Tue Sep 18, 2018 2:08:34 PM
Execution ID	16777217
User	SH
Fetch Calls	1

Time & Wait Statistics

Duration	21.0s
Database Time	21.3s
PL/SQL & Java	0s
Activity %	100

IO Statistics

Buffer Gets	204
IO Requests	4,801
IO Bytes	913MB

Details

Plan Hash Value 2036849021

Operation	Name	L...	Esti...	Cost	Timeline(21s)	Ex...	Actu...	Mem...	Tem...	O.	IO Re...	IO ...	Activity %
SELECT STATEMENT		0				1	0						
SORT ORDER BY		1	51G	316M		1	0	99MB	909MB		4,496	908MB	95
MERGE JOIN CARTESIAN		2	51G	1,599K		1	50M						
TABLE ACCESS FULL	CUSTOMERS	3	56K	414		1	54				3	120KB	
BUFFER SORT		4	919K	316M		54	50M	28MB					5
PARTITION RANGE ALL		5	919K	29		1	919K						
BITMAP CONVERSIO...		6	919K	29		28	919K						
BITMAP INDEX FAST...	SALES_PROD	7				28	1,074				32	512KB	

The cause of the performance problem is shown in Line 2: a Cartesian join. The author of the query inadvertently left off the `WHERE` clause. Instead of returning around 1 million rows as it would for an inner join of `sales` and `customers`, the query returned 50 million rows before it was canceled. Sorting the joined data from the two tables is consuming most of the DB time (Line 1).

21.5.2 Reporting on Composite Database Operation: Scenario

This scenario uses `DBMS_SQL_MONITOR` to define a database operation and generates an active report.

Your goal is to group four queries of tables in the `sh` schema into an operation, and then generate a report.

1. In SQL*Plus, log on as an administrative user `SAM`. Begin an operation named `SHOP` (specifying `forced_tracking` to ensure that SQL Monitor tracks the SQL), run four queries, and then end the operation as follows:

```
VARIABLE exec_id NUMBER;
BEGIN
  :exec_id := DBMS_SQL_MONITOR.BEGIN_OPERATION ( dbop_name => 'SHOP',
forced_tracking => 'Y' );
END;
/

SELECT COUNT(*) FROM sh.sales;
SELECT COUNT(*) FROM sh.customers;

SELECT prod_id, cust_id
FROM   sh.sales
WHERE  prod_id < 26
ORDER BY prod_id;

SELECT cust_id, cust_first_name, cust_last_name, cust_city
FROM   sh.customers
WHERE  cust_id < 30000
ORDER BY cust_id;

BEGIN
  DBMS_SQL_MONITOR.END_OPERATION ( dbop_name => 'SHOP', dbop_eid
=> :exec_id );
END;
/
```

2. To obtain metadata about the operation, including its status and metadata, query `V$SQL_MONITOR` (sample output included):

```
COL STATUS FORMAT a10
COL DBOP_NAME FORMAT a10
COL CON_NAME FORMAT a5

SELECT STATUS, SQL_ID, DBOP_NAME, DBOP_EXEC_ID,
       TO_CHAR(ELAPSED_TIME/1000000,'000.00') AS ELA_SEC
FROM   V$SQL_MONITOR
WHERE  DBOP_NAME = 'SHOP';

STATUS     SQL_ID        DBOP_NAME  DBOP_EXEC_ID ELA_SEC
```

```
---------- ------------- ---------- ------------ -------
DONE                      SHOP                   3  001.34
```

3. To obtain metadata about the SQL Monitor report, call `DBMS_SQL_MONITOR.REPORT_SQL_MONITOR` (sample output included):

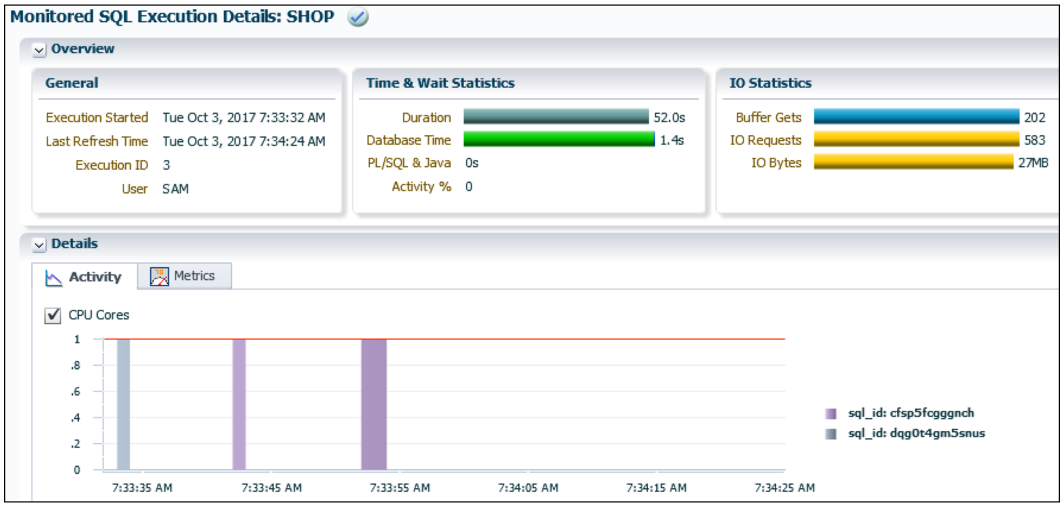
```
SET LONG 10000000
SET LONGCHUNKSIZE 10000000
SET PAGES 0
SELECT DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(
  dbop_name => 'SHOP', type => 'TEXT', report_level => 'ALL') AS rpt
FROM DUAL;

SQL Monitoring Report

Global Information
------------------------------
 Status               :  DONE
 Instance ID          :  1
 Session              :  SAM (87:6406)
 DBOP Name            :  SHOP
 DBOP Execution ID    :  3
 First Refresh Time   :  10/03/2017 07:33:32
 Last Refresh Time    :  10/03/2017 07:34:24
 Duration             :  52s
 Module/Action        :  sqlplus@myhost (TNS V1-V3)/-
 Service              :  MYSERVICE
 Program              :  sqlplus@myhost (TNS V1-V3)

Global Stats
=========================================================
| Elapsed |   Cpu   |    IO    | Buffer | Read | Read  |
| Time(s) | Time(s) | Waits(s) |  Gets  | Reqs | Bytes |
=========================================================
|    1.36 |    1.34 |     0.02 |    202 |  583 |  27MB |
=========================================================
```

4. To generate an active HTML report, pass the name of the operation to `DBMS_SQL_MONITOR.REPORT_SQL_MONITOR`:

```
SET TRIMSPOOL ON
SET TRIM ON
SET PAGES 0
SET LINESIZE 1000
SET LONG 1000000
SET LONGCHUNKSIZE 1000000

SPOOL /tmp/shop.htm
SELECT
DBMS_SQL_MONITOR.REPORT_SQL_MONITOR(dbop_name=>'SHOP',report_level=>
'ALL',TYPE=>'active')
FROM   DUAL;
SPOOL OFF
```

The following graphic shows the active report:

Figure 21-3 SQL Monitor Report



Monitored SQL Execution Details: SHOP

Overview

General

Execution Started	Tue Oct 3, 2017 7:33:32 AM
Last Refresh Time	Tue Oct 3, 2017 7:34:24 AM
Execution ID	3
User	SAM

Time & Wait Statistics

Duration	52.0s
Database Time	1.4s
PL/SQL & Java	0s
Activity %	0

IO Statistics

Buffer Gets	202
IO Requests	583
IO Bytes	27MB

Details

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_SQL_MONITOR`

22 Gathering Diagnostic Data with SQL Test Case Builder

SQL Test Case Builder is a tool that automatically gathers information needed to reproduce the problem in a different database instance.

A **SQL test case** is a set of information that enables a developer to reproduce the execution plan for a specific SQL statement that has encountered a performance problem.

This chapter contains the following topics:

22.1 Purpose of SQL Test Case Builder

SQL Test Case Builder automates the process of gathering and reproducing information about a problem and the environment in which it occurred.

For most SQL components, obtaining a reproducible test case is the most important factor in bug resolution speed. It is also the longest and most painful step for users. The goal of SQL Test Case Builder is to gather as much as information related to an SQL incident as possible, and then package it in a way that enables Oracle staff to reproduce the problem on a different system.

The output of SQL Test Case Builder is a set of scripts in a predefined directory. These scripts contain the commands required to re-create all the necessary objects and the environment on another database instance. After the test case is ready, you can create a zip file of the directory and move it to another database, or upload the file to Oracle Support.

22.2 Concepts for SQL Test Case Builder

Key concepts for SQL Test Case Builder include SQL incidents, types of information recorded, and the form of the output.

This section contains the following topics:

22.2.1 SQL Incidents

In the fault diagnosability infrastructure of Oracle Database, an **incident** is a single occurrence of a problem.

A SQL incident is a SQL-related problem. When a problem (critical error) occurs multiple times, the database creates an incident for each occurrence. Incidents are timestamped and tracked in the Automatic Diagnostic Repository (ADR). Each incident has a numeric incident ID, which is unique within the ADR.

SQL Test Case Builder is accessible any time on the command line. In Oracle Enterprise Manager Cloud Control (Cloud Control), the SQL Test Case pages are only available after a SQL incident is found.

22.2.2 What SQL Test Case Builder Captures

SQL Test Case Builder captures permanent information about a SQL query and its environment.

The information includes the query being executed, table and index definitions (but not the actual data), PL/SQL packages and program units, optimizer statistics, SQL plan baselines, and initialization parameter settings. Starting with Oracle Database 12c, SQL Test Case Builder also captures and replays transient information, including information only available as part of statement execution.

SQL Test Case Builder supports the following:

- Adaptive plans

  SQL Test Case Builder captures inputs to the decisions made regarding adaptive plans, and replays them at each decision point. For adaptive plans, the final statistics value at each buffering statistics collector is sufficient to decide on the final plan.

- Automatic memory management

  The database automatically handles the memory requested for each SQL operation. Actions such as sorting can affect performance significantly. SQL Test Case Builder keeps track of the memory activities, for example, where the database allocated memory and how much it allocated.

- Dynamic statistics

  Dynamic statistics is an optimization technique in which the database executes a recursive SQL statement to scan a small random sample of a table's blocks to estimate predicate selectivities. Regathering dynamic statistics on a different database does not always generate the same results, for example, when data is missing. To reproduce the problem, SQL Test Case Builder exports the dynamic statistics result from the source database. In the testing database, SQL Test Case Builder reuses the same values captured from the source database instead of regathering dynamic statistics.

- Multiple execution support

  SQL Test Case Builder can capture dynamic information accumulated during multiple executions of the query. This capability is important for automatic reoptimization.

- Compilation environment and bind values replay

  The compilation environment setting is an important part of the query optimization context. SQL Test Case Builder captures nondefault settings altered by the user when running the problem query in the source database. If any nondefault parameter values are used, SQL Test Case Builder re-establishes the same values before running the query.

- Object statistics history

  The statistics history for objects is helpful to determine whether a plan change was caused by a change in statistics values. `DBMS_STATS` stores the history in the data dictionary. SQL Test Case Builder stores this statistics data into a staging table during export. During import, SQL Test Case Builder automatically reloads the statistics history data into the target database from the staging table.

- Statement history

  The statement history is important for diagnosing problems related to adaptive cursor sharing, statistics feedback, and cursor sharing bugs. The history includes execution plans and compilation and execution statistics.

See Also:

- *Oracle Database SQL Tuning Guide* for more information about adaptive query plans, supplemental dynamic statistics, automatic reoptimization, and SQL plan baselines
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_STATS` package

22.2.3 Output of SQL Test Case Builder

The output of SQL Test Case Builder is a set of files that contains commands required to re-create the environment and all necessary objects.

By default, SQL Test Case Builder stores the files in the following directory, where *incnum* refers to the incident number and *runnum* refers to the run number:

```
$ADR_HOME/incident/incdir_incnum/SQLTCB_runnum
```

For example, a valid output file name could be as follows:

```
$ORACLE_HOME/log/diag/rdbms/dbsa/dbsa/incident/incdir_2657/SQLTCB_1
```

You can also specify a particular directory for storing the SQL Test Case Builder files by creating a directory object with the name `SQL_TCB_DIR` and running the procedure `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE` as shown in the following example:

```
CREATE OR REPLACE DIRECTORY SQL_TCB_DIR '/tmp';

DECLARE
tc CLOB;
BEGIN
  DBMS_SQLDIAG.EXPORT_SQL_TESTCASE (
    directory => 'SQL_TCB_DIR',
    sql_text  => 'select * from hr_table',
    testcase  => tc);
END;
```

Note:

The database administrator must have read and write access permissions to the operating system directory specified in the directory object `SQL_TCB_DIR`.

You can also specify a name for a test case using the `testcase_name` parameter of the `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE` procedure. A test case name is used as a prefix for all the files generated by SQL Test Case Builder.

If you do not specify a test case name, then a default test case name having the following format is used by SQL Test Case Builder:

```
oratcb_connectionId_sqlId_sequenceNumber_sessionId
```

Here, *connectionId* is the database connection ID, *sqlId* is the SQL statement ID, *sequenceNumber* is the internal sequence number, and *sessionId* is the database session ID.

You can also specify any additional information to include in the output of SQL Test Case Builder using the `ctrlOptions` parameter of the `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE` procedure. The following are some of the options that you can specify in the `ctrlOptions` parameter:

- `compress`: This option is used to compress the SQL Test Case Builder output files into a zip file.
- `diag_event`: This option is used to specify the level of trace information to include in the SQL Test Case Builder output.
- `problem_type`: This option is used to assign an issue type for a SQL Test Case Builder test case. For example, if a test case is related to performance regression issue, then you can assign the value of `PERFORMANCE` to the `problem_type` option.

You can view the information about all the test cases generated by SQL Test Case Builder by querying the `V$SQL_TESTCASES` view as shown in the following example:

```
select testcase_name, sql_text from v$sql_testcases;

TESTCASE_NAME                         SQL_TEXT
------------------------------------- ----------------------
oratcb_0_am8q8kudm02v9_1_00244CC50001 select * from hr_table
```

Note:

The `V$SQL_TESTCASES` view requires the existence of a SQL Test Case Builder root directory object named `SQL_TCB_DIR`. In Oracle Autonomous Database environments, this directory object is created automatically on each POD during provisioning. For on-premises databases, you must explicitly create the SQL Test Case Builder root directory object `SQL_TCB_DIR`, otherwise the `V$SQL_TESTCASES` view will not display any information. The database administrator must have read and write access permissions to the operating system directory specified in the directory object `SQL_TCB_DIR`.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SQLDIAG.EXPORT_SQL_TESTCASE` procedure
- *Oracle Database Reference* for more information about the `V$SQL_TESTCASES` view

22.3 User Interfaces for SQL Test Case Builder

You can access SQL Test Case Builder either through Cloud Control or using PL/SQL on the command line.

This section contains the following topics:

22.3.1 Graphical Interface for SQL Test Case Builder

Within Cloud Control, you can access SQL Test Case Builder from the Incident Manager page or the Support Workbench page.

This section contains the following topics:

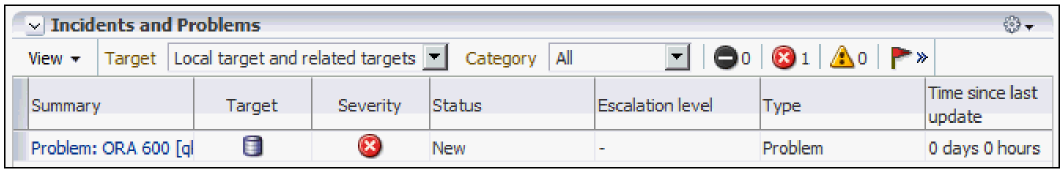
22.3.1.1 Accessing the Incident Manager

From the Incidents and Problems section on the Database Home page, you can navigate to the Incident Manager.

To access the Incident Manager:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. In the Incidents and Problems section, locate the SQL incident to be investigated.

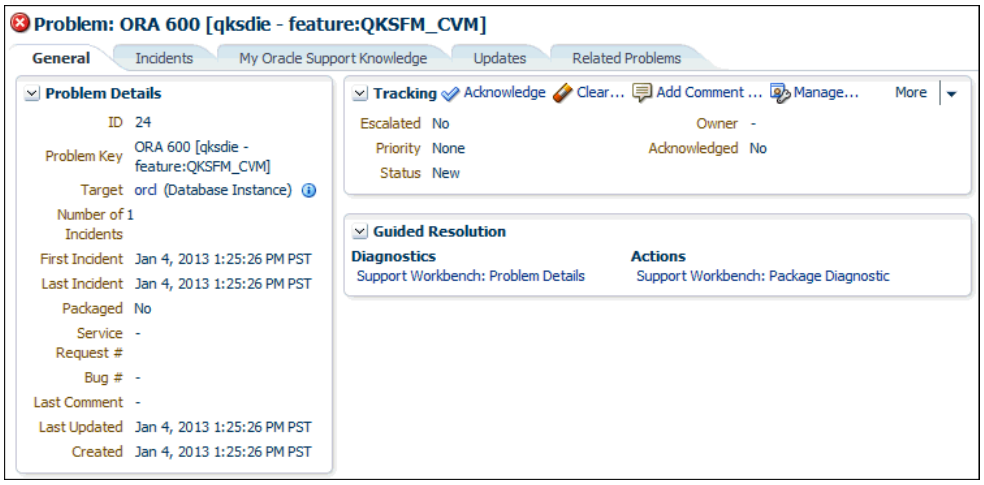
   In the following example, the `ORA 600` error is a SQL incident.



Summary	Target	Severity	Status	Escalation level	Type	Time since last update
Problem: ORA 600 [ql			New	-	Problem	0 days 0 hours

6. Click the summary of the incident.

   The Problem Details page of the Incident Manager appears.



Problem: ORA 600 [qksdie - feature:QKSFM_CVM]

Problem Details

ID	24
Problem Key	ORA 600 [qksdie - feature:QKSFM_CVM]
Target	orcl (Database Instance)
Number of Incidents	1
First Incident	Jan 4, 2013 1:25:26 PM PST
Last Incident	Jan 4, 2013 1:25:26 PM PST
Packaged	No
Service Request #	-
Bug #	-
Last Comment	-
Last Updated	Jan 4, 2013 1:25:26 PM PST
Created	Jan 4, 2013 1:25:26 PM PST

Tracking

Escalated	No	Owner	-
Priority	None	Acknowledged	No
Status	New		

Guided Resolution

Diagnostics	Actions
Support Workbench: Problem Details	Support Workbench: Package Diagnostic

The Support Workbench page appears, with the incidents listed in a table.

22.3.1.2 Accessing the Support Workbench

From the Oracle Database menu, you can navigate to the Support Workbench.

To access the Support Workbench:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Oracle Database** menu, select **Diagnostics**, then **Support Workbench**.

   The Support Workbench page appears, with the incidents listed in a table.

22.4 Command-Line Interface for SQL Test Case Builder

The `DBMS_SQLDIAG` package performs tasks relating to SQL Test Case Builder.

This package consists of various subprograms for the SQL Test Case Builder, some of which are listed in the following table.

Table 22-1 SQL Test Case Functions in the DBMS_SQLDIAG Package

Procedure	Description
`EXPORT_SQL_TESTCASE`	Exports a SQL test case to a user-specified directory
`EXPORT_SQL_TESTCASE_DIR_BY_INC`	Exports a SQL test case corresponding to the incident ID passed as an argument
`EXPORT_SQL_TESTCASE_DIR_BY_TXT`	Exports a SQL test case corresponding to the SQL text passed as an argument

Table 22-1 (Cont.) SQL Test Case Functions in the DBMS_SQLDIAG Package

Procedure	Description
`IMPORT_SQL_TESTCASE`	Imports a SQL test case into a schema
`REPLAY_SQL_TESTCASE`	Automates reproduction of a SQL test case
`EXPLAIN_SQL_TESTCASE`	Explains a SQL test case

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DBMS_SQLDIAG` package

22.5 Running SQL Test Case Builder

You can run SQL Test Case Builder using Cloud Control.

Assumptions

This tutorial assumes the following:

- You ran the following `EXPLAIN PLAN` statement as user `sh`, which causes an internal error:

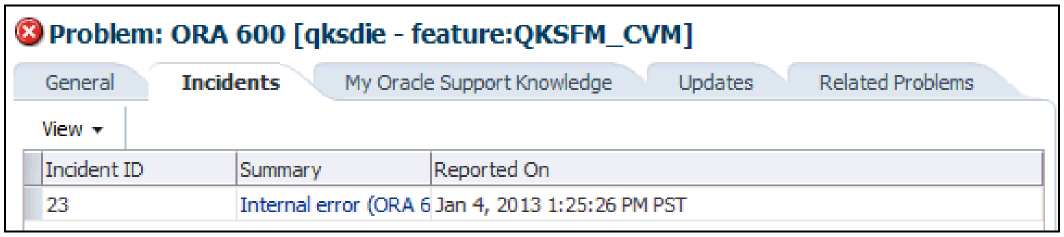
```
EXPLAIN PLAN FOR
  SELECT unit_cost, sold
  FROM   costs c,
         ( SELECT /*+ merge */ p.prod_id, SUM(quantity_sold) AS sold
           FROM   products p, sales s
           WHERE  p.prod_id = s.prod_id
           GROUP BY p.prod_id ) v
  WHERE  c.prod_id = v.prod_id;
```

- In the Incidents and Problems section on the Database Home page, a SQL incident generated by the internal error appears.
- You access the Incident Details page, as explained in "Accessing the Incident Manager".

To run SQL Test Case Builder:

1. Click the Incidents tab.

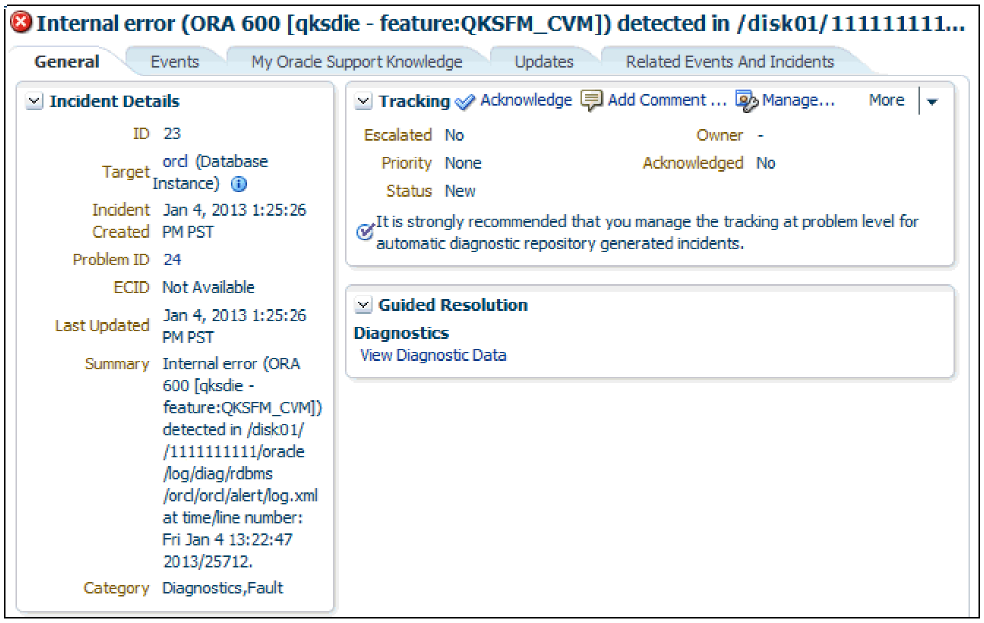
   The Problem Details page appears.


Problem: ORA 600 [qksdie - feature:QKSFM_CVM]

Incident ID	Summary	Reported On
23	Internal error (ORA 6	Jan 4, 2013 1:25:26 PM PST

2. Click the summary for the incident.

   The Incident Details page appears.



Internal error (ORA 600 [qksdie - feature:QKSFM_CVM]) detected in /disk01/111111111...

Incident Details

ID	23
Target	orcl (Database Instance)
Incident Created	Jan 4, 2013 1:25:26 PM PST
Problem ID	24
ECID	Not Available
Last Updated	Jan 4, 2013 1:25:26 PM PST
Summary	Internal error (ORA 600 [qksdie - feature:QKSFM_CVM]) detected in /disk01/ /1111111111/oracle /log/diag/rdbms /orcl/orcl/alert/log.xml at time/line number: Fri Jan 4 13:22:47 2013/25712.
Category	Diagnostics,Fault

Tracking

Escalated	No	Owner	-
Priority	None	Acknowledged	No
Status	New		

It is strongly recommended that you manage the tracking at problem level for automatic diagnostic repository generated incidents.

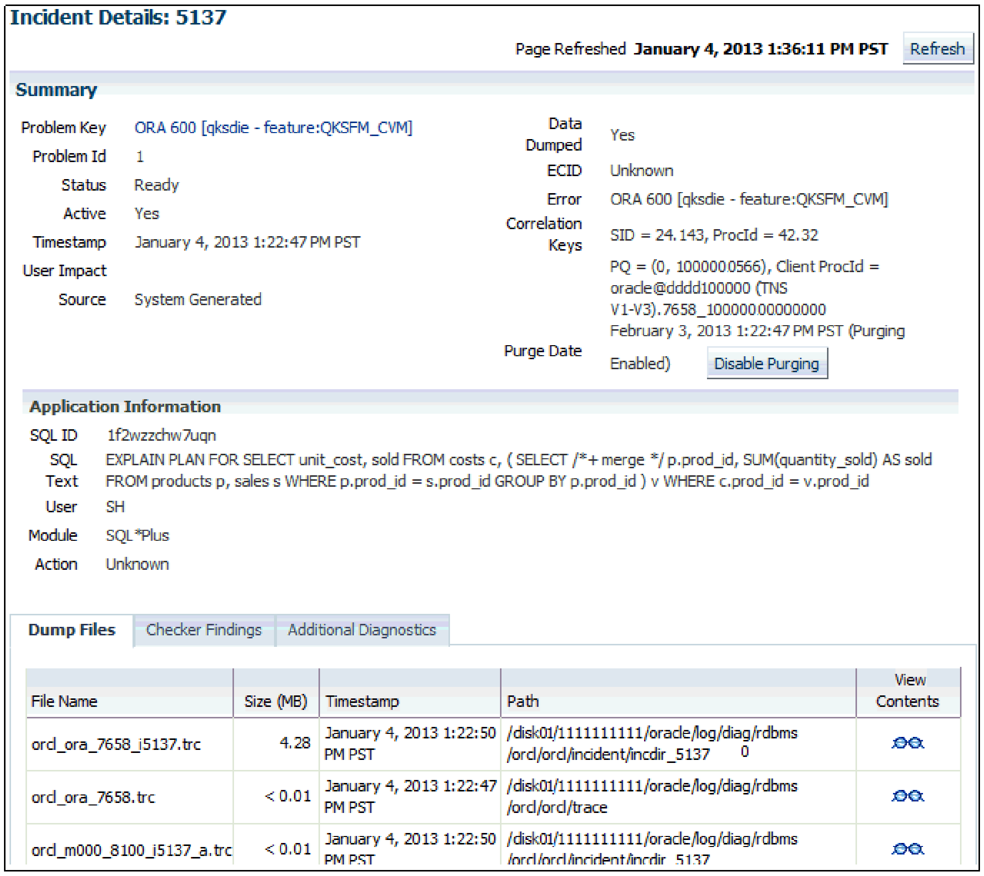
Guided Resolution

Diagnostics

View Diagnostic Data

3. In Guided Resolution, click **View Diagnostic Data**.

   The Incident Details: *incident_number* page appears.



Incident Details: 5137

Page Refreshed **January 4, 2013 1:36:11 PM PST**

Summary

Problem Key	ORA 600 [qksdie - feature:QKSFM_CVM]	Data Dumped	Yes
Problem Id	1	ECID	Unknown
Status	Ready	Error Correlation Keys	ORA 600 [qksdie - feature:QKSFM_CVM]
Active	Yes		SID = 24.143, ProcId = 42.32
Timestamp	January 4, 2013 1:22:47 PM PST		PQ = (0, 1000000566), Client ProcId = oracle@dddd100000 (TNS V1-V3).7658_10000000000000
User Impact		Purge Date	February 3, 2013 1:22:47 PM PST (Purging Enabled)
Source	System Generated		

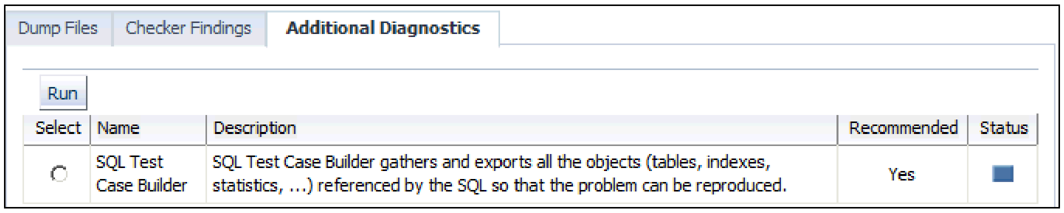
Application Information

SQL ID	1f2wzzchw7uqn
SQL Text	EXPLAIN PLAN FOR SELECT unit_cost, sold FROM costs c, (SELECT /*+ merge */ p.prod_id, SUM(quantity_sold) AS sold FROM products p, sales s WHERE p.prod_id = s.prod_id GROUP BY p.prod_id) v WHERE c.prod_id = v.prod_id
User	SH
Module	SQL*Plus
Action	Unknown

File Name	Size (MB)	Timestamp	Path	View Contents
orcl_ora_7658_i5137.trc	4.28	January 4, 2013 1:22:50 PM PST	/disk01/1111111111/oracle/log/diag/rdbms /orcl/orcl/incident/incdir_5137 0	👓
orcl_ora_7658.trc	< 0.01	January 4, 2013 1:22:47 PM PST	/disk01/1111111111/oracle/log/diag/rdbms /orcl/orcl/trace	👓
orcl_m000_8100_i5137_a.trc	< 0.01	January 4, 2013 1:22:50 PM PST	/disk01/1111111111/oracle/log/diag/rdbms /orcl/orcl/incident/incdir_5137	👓

4. In the Application Information section, click **Additional Diagnostics**.

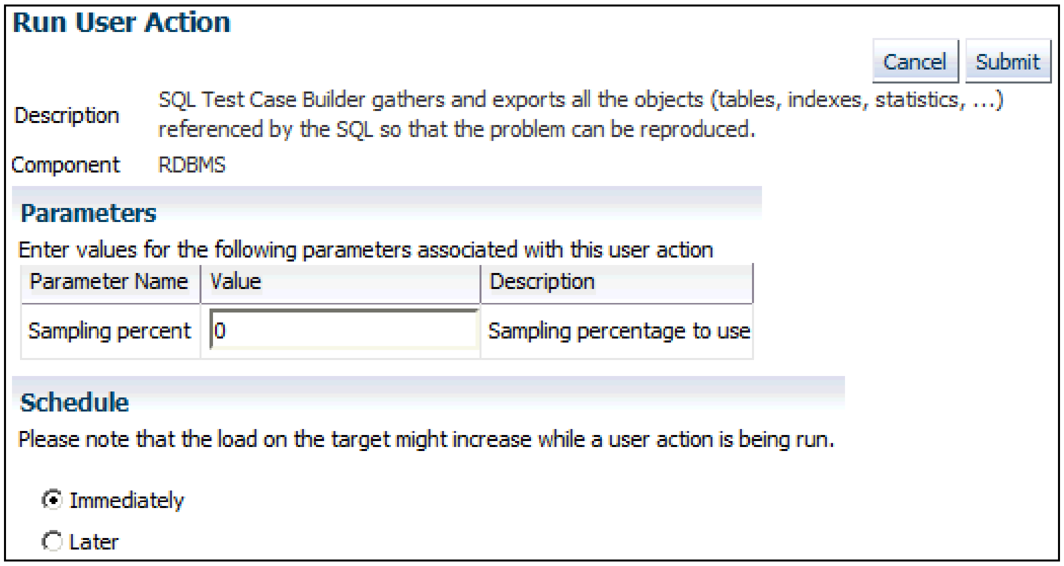
   The Additional Diagnostics subpage appears.



Select	Name	Description	Recommended	Status
○	SQL Test Case Builder	SQL Test Case Builder gathers and exports all the objects (tables, indexes, statistics, ...) referenced by the SQL so that the problem can be reproduced.	Yes	■

5. Select **SQL Test Case Builder**, and then click **Run**.

   The Run User Action page appears.



Description SQL Test Case Builder gathers and exports all the objects (tables, indexes, statistics, ...) referenced by the SQL so that the problem can be reproduced.

Component RDBMS

Parameters

Enter values for the following parameters associated with this user action

Parameter Name	Value	Description
Sampling percent	0	Sampling percentage to use

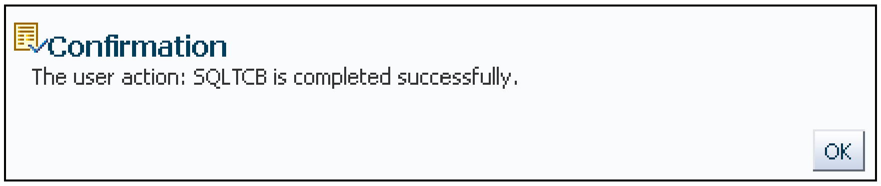
Schedule

Please note that the load on the target might increase while a user action is being run.

[✓ Immediately] Later

6. Select a sampling percentage (optional), and then click **Submit**.

   After processing completes, the Confirmation page appears.



The user action: SQLTCB is completed successfully.

7. Access the SQL Test Case files in the location described in "Output of SQL Test Case Builder".

23 Performing Application Tracing

This chapter explains what end-to-end application tracing is, and how to generate and read trace files.

See Also:

*SQL*Plus User's Guide and Reference* to learn about the use of Autotrace to trace and tune SQL*Plus statements

23.1 Overview of End-to-End Application Tracing

End-to-end application tracing can identify the source of an excessive database workload, such as a high load SQL statement, by client identifier, service, module, action, session, instance, or an entire database.

In multitier environments, the middle tier routes a request from an end client to different database sessions, making it difficult to track a client across database sessions. End-to-end application tracing is an infrastructure that uses a client ID to uniquely trace a specific end-client through all tiers to the database and provides information about the operation that an end client is performing in the database.

23.1.1 Purpose of End-to-End Application Tracing

End-to-end application tracing simplifies diagnosing performance problems in multitier environments.

For example, you can identify the source of an excessive database workload, such as a high-load SQL statement, and contact the user responsible. Also, a user having problems can contact you. You can then identify what this user session is doing at the PDB level

End-to-end application tracing also simplifies management of application workloads by tracking specific modules and actions in a service. The module and action names are set by the application developer. For example, you would use the `SET_MODULE` and `SET_ACTION` procedures in the `DBMS_APPLICATION_INFO` package to set these values in a PL/SQL program.

End-to-end application tracing can identify workload problems in a database for:

- Client identifier

  Specifies an end user based on the logon ID, such as `HR.HR`
- Service

  Specifies a group of applications with common attributes, service level thresholds, and priorities; or a single application, such as `ACCTG` for an accounting application
- Module

Specifies a functional block, such as Accounts Receivable or General Ledger, of an application

- Action

  Specifies an action, such as an `INSERT` or `UPDATE` operation, in a module

- Session

  Specifies a session based on a given database session identifier (SID), on the local instance

- Instance

  Specifies a given database instance based on the instance name

- Container

  Specifies the container

23.1.2 End-to-End Application Tracing for PDBs

`V$` views enable read access to trace files that are specific to a container.

The primary use cases are as follows:

- CDB administrators must view trace files from a specific PDB.

  The `V$DIAG_TRACE_FILE` view lists all trace files in the ADR trace directory that contain trace data from a specific PDB. `V$DIAG_TRACE_FILE_CONTENTS` displays the contents of the trace files.

- PDB administrators must view trace files from a specific PDB.

  You can use SQL Trace to collect diagnostic data for the SQL statements executing in a PDB application. The trace data includes SQL tracing (event 10046) and optimizer tracing (event 10053). Using `V$` views, developers can access only SQL or optimizer trace records without accessing the entire trace file.

To enable users and tools to determine which PDB is associated with a file or a part of a file, PDB annotations exist in trace files, incident dumps, and log files. The PDB information is part of the structured metadata that is stored in the `.trm` file for each trace file. Each record captures the following attributes:

- `CON_ID`, which is the ID of the container associated with the data
- `CON_UID`, which is the unique ID of the container
- `NAME`, which is the name of the container

See Also:

"Views for Application Tracing"

23.1.3 Tools for End-to-End Application Tracing

The SQL Trace facility and TKPROF are two basic performance diagnostic tools that can help you accurately assess the efficiency of the SQL statements an application runs.

For best results, use these tools with `EXPLAIN PLAN` rather than using `EXPLAIN PLAN` alone. After tracing information is written to files, you can consolidate this data with the TRCSESS utility, and then diagnose it with TKPROF or SQL Trace.

The recommended interface for end-to-end application tracing is Oracle Enterprise Manager Cloud Control (Cloud Control). Using Cloud Control, you can view the top consumers for each consumer type, and enable or disable statistics gathering and SQL tracing for specific consumers. If Cloud Control is unavailable, then you can manage this feature using the `DBMS_MONITOR` APIs.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_MONITOR`, `DBMS_SESSION`, `DBMS_SERVICE`, and `DBMS_APPLICATION_INFO` packages

23.1.3.1 Overview of the SQL Trace Facility

The SQL Trace facility provides performance information on individual SQL statements.

SQL Trace generates the following statistics for each statement:

- Parse, execute, and fetch counts
- CPU and elapsed times
- Physical reads and logical reads
- Number of rows processed
- Misses on the library cache
- User name under which each parse occurred
- Each commit and rollback
- Wait event data for each SQL statement, and a summary for each trace file

If the cursor for the SQL statement is closed, then SQL Trace also provides row source information that includes:

- Row operations showing the actual execution plan of each SQL statement
- Number of rows, number of consistent reads, number of physical reads, number of physical writes, and time elapsed for each operation on a row

Although you can enable the SQL Trace facility for a session or an instance, Oracle recommends that you use the `DBMS_SESSION` or `DBMS_MONITOR` packages instead. When the SQL Trace facility is enabled for a session or for an instance, performance statistics for all SQL statements executed in a user session or in the instance are placed into trace files. Using the SQL Trace facility can affect performance and may result in increased system overhead, excessive CPU usage, and inadequate disk space.

The TRCSESS command-line utility consolidates tracing information from several trace files based on specific criteria, such as session or client ID.

See Also:

- "TRCSESS"
- "Enabling End-to-End Application Tracing" to learn how to use the DBMS_SESSION or DBMS_MONITOR packages to enable SQL tracing for a session or an instance

23.1.3.2 Overview of TKPROF

To format the contents of the trace file and place the output into a readable output file, run the TKPROF program.

TKPROF can also do the following:

- Create a SQL script that stores the statistics in the database
- Determine the execution plans of SQL statements

TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows which it processed. This information enables you to locate those statements that are using the greatest resource. With baselines available, you can assess whether the resources used are reasonable given the work done.

23.2 Enabling Statistics Gathering for End-to-End Tracing

To gather the appropriate statistics using PL/SQL, you must enable statistics gathering for client identifier, service, module, or action using procedures in DBMS_MONITOR.

The default level is the session-level statistics gathering. Statistics gathering is global for the database and continues after a database instance is restarted.

23.2.1 Enabling Statistics Gathering for a Client ID

The procedure CLIENT_ID_STAT_ENABLE enables statistics gathering for a given client ID, whereas the procedure CLIENT_ID_STAT_DISABLE disables it.

You can view client identifiers in the CLIENT_IDENTIFIER column in V$SESSION.

Assumptions

This tutorial assumes that you want to enable and then disable statistics gathering for the client with the ID oe.oe.

To enable and disable statistics gathering for a client identifier:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable statistics gathering for oe.oe.

   For example, run the following command:

   ```
   EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_ENABLE(client_id => 'OE.OE');
   ```

3. Disable statistics gathering for `oe.oe`.

   For example, run the following command:

   ```
   EXECUTE DBMS_MONITOR.CLIENT_ID_STAT_DISABLE(client_id => 'OE.OE');
   ```

23.2.2 Enabling Statistics Gathering for Services, Modules, and Actions

The procedure `SERV_MOD_ACT_STAT_ENABLE` enables statistic gathering for a combination of service, module, and action, whereas the procedure `SERV_MOD_ACT_STAT_DISABLE` disables statistic gathering for a combination of service, module, and action.

When you change the module or action using the preceding `DBMS_MONITOR` procedures, the change takes effect when the next user call is executed in the session. For example, if a module is set to `module1` in a session, and if the module is reset to `module2` in a user call in the session, then the module remains `module1` during this user call. The module is changed to `module2` in the next user call in the session.

Assumptions

This tutorial assumes that you want to gather statistics as follows:

- For the `ACCTG` service
- For all actions in the `PAYROLL` module
- For the `INSERT ITEM` action within the `GLEDGER` module

To enable and disable statistics gathering for a service, module, and action:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable statistics gathering for the desired service, module, and action.

   For example, run the following commands:

   ```
   BEGIN
     DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
       service_name => 'ACCTG'
   ,   module_name  => 'PAYROLL' );
   END;

   BEGIN
     DBMS_MONITOR.SERV_MOD_ACT_STAT_ENABLE(
       service_name => 'ACCTG'
   ,   module_name  => 'GLEDGER'
   ,   action_name  => 'INSERT ITEM' );
   END;
   ```

3. Disable statistic gathering for the previously specified combination of service, module, and action.

   For example, run the following command:

   ```
   BEGIN
     DBMS_MONITOR.SERV_MOD_ACT_STAT_DISABLE(
       service_name => 'ACCTG'
   ```

```
,   module_name  => 'GLEDGER'
,   action_name  => 'INSERT ITEM' );
END;
```

23.3 Enabling End-to-End Application Tracing

To enable tracing for client identifier, service, module, action, session, instance or database, execute the appropriate procedures in the `DBMS_MONITOR` package.

With the criteria that you provide, specific trace information is captured in a set of trace files and combined into a single output trace file. You can enable tracing for specific diagnosis and workload management by the following criteria:

See Also:

Oracle Database Administrator's Guide for information about how to locate trace files

23.3.1 Enabling Tracing for a Client Identifier

To enable tracing globally for the database for a specified client identifier, use the `DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE` procedure.

The `CLIENT_ID_TRACE_DISABLE` procedure disables tracing globally for the database for a given client identifier.

Assumptions

This tutorial assumes the following:

- `OE.OE` is the client identifier for which SQL tracing is to be enabled.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.

To enable and disable tracing for a client identifier:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable tracing for the client.

   For example, execute the following program:

   ```
   BEGIN
     DBMS_MONITOR.CLIENT_ID_TRACE_ENABLE(
       client_id => 'OE.OE' ,
       waits     => true    ,
       binds     => false   );
   END;
   ```

3. Disable tracing for the client.

For example, execute the following command:

```
EXECUTE DBMS_MONITOR.CLIENT_ID_TRACE_DISABLE(client_id => 'OE.OE');
```

23.3.2 Enabling Tracing for a Service, Module, and Action

The `DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE` procedure enables SQL tracing for a specified combination of service name, module, and action globally for a database, unless the procedure specifies a database instance name.

The `SERV_MOD_ACT_TRACE_DISABLE` procedure disables the trace at all enabled instances for a given combination of service name, module, and action name globally.

Assumptions

This tutorial assumes the following:

- You want to enable tracing for the service `ACCTG`.
- You want to enable tracing for all actions for the combination of the `ACCTG` service and the `PAYROLL` module.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.
- You want to enable tracing only for the `inst1` instance.

To enable and disable tracing for a service, module, and action:

1. Start SQL*Plus, and then connect to the database with the appropriate privileges.
2. Enable tracing for the service, module, and actions.

   For example, execute the following command:

```
BEGIN
  DBMS_MONITOR.SERV_MOD_ACT_TRACE_ENABLE(
    service_name  => 'ACCTG'   ,
    module_name   => 'PAYROLL' ,
    waits         =>  true     ,
    binds         =>  false    ,
    instance_name => 'inst1'   );
END;
```

3. Disable tracing for the service, module, and actions.

   For example, execute the following command:

```
BEGIN
  DBMS_MONITOR.SERV_MOD_ACT_TRACE_DISABLE(
    service_name  => 'ACCTG'   ,
    module_name   => 'PAYROLL' ,
    instance_name => 'inst1'   );
END;
```

23.3.3 Enabling Tracing for a Session

The `SESSION_TRACE_ENABLE` procedure enables the trace for a given database session identifier (SID) on the local instance.

Whereas the `DBMS_MONITOR` package can only be invoked by a user with the DBA role, users can also enable SQL tracing for their own session by invoking the `DBMS_SESSION.SESSION_TRACE_ENABLE` procedure, as in the following example:

```
BEGIN
  DBMS_SESSION.SESSION_TRACE_ENABLE(
    waits => true
  , binds => false);
END;
```

Assumptions

This tutorial assumes the following:

- You want to log in to the database with administrator privileges.
- User `OE` has one active session.
- You want to temporarily enable tracing for the `OE` session.
- You want to include wait information in the trace.
- You want to exclude bind information from the trace.

To enable and disable tracing for a session:

1. Start SQL*Plus, and then log in to the database with the administrator privileges.
2. Determine the session ID and serial number values for the session to trace.

   For example, query `V$SESSION` as follows:

   ```
   SELECT SID, SERIAL#, USERNAME
   FROM   V$SESSION
   WHERE  USERNAME = 'OE';

          SID    SERIAL# USERNAME
   ---------- ---------- ------------------------------
           27         60 OE
   ```

3. Use the values from the preceding step to enable tracing for a specific session.

   For example, execute the following program to enable tracing for the `OE` session, where the `true` argument includes wait information in the trace and the `false` argument excludes bind information from the trace:

   ```
   BEGIN
     DBMS_MONITOR.SESSION_TRACE_ENABLE(
       session_id => 27
     , serial_num => 60
     , waits      => true
   ```

```
  , binds       => false);
END;
```

4. Disable tracing for the session.

   The `SESSION_TRACE_DISABLE` procedure disables the trace for a given database session identifier (SID) and serial number. For example:

```
BEGIN
  DBMS_MONITOR.SESSION_TRACE_DISABLE(
    session_id => 27
  , serial_num => 60);
END;
```

23.3.4 Enabling Tracing for an Instance or Database

The `DBMS_MONITOR.DATABASE_TRACE_ENABLE` procedure overrides all other session-level traces, but is complementary to the client identifier, service, module, and action traces. Tracing is enabled for all current and future sessions.

All new sessions inherit the wait and bind information specified by this procedure until you call the `DATABASE_TRACE_DISABLE` procedure. When you invoke this procedure with the `instance_name` parameter, the procedure resets the session-level SQL trace for the named instance. If you invoke this procedure without the `instance_name` parameter, then the procedure resets the session-level SQL trace for the entire database.

Prerequisites

You must have administrative privileges to execute the `DATABASE_TRACE_ENABLE` procedure.

Assumptions

This tutorial assumes the following:

- You want to enable tracing for all SQL the `inst1` instance.
- You want wait information to be in the trace.
- You do not want bind information in the trace.

To enable and disable tracing for an instance or database:

1. Start SQL*Plus, and then log in to the database with the necessary privileges.
2. Call the `DATABASE_TRACE_ENABLE` procedure to enable SQL tracing for a given instance or an entire database.

   For example, execute the following program, where the `true` argument specifies that wait information is in the trace, and the `false` argument specifies that no bind information is in the trace:

```
BEGIN
  DBMS_MONITOR.DATABASE_TRACE_ENABLE(
    waits         => true
  , binds         => false
  , instance_name => 'inst1' );
END;
```

3. Disable tracing.

   The `DATABASE_TRACE_DISABLE` procedure disables the trace. For example, the following program disables tracing for `inst1`:

   ```
   EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE(instance_name =>
   'inst1');
   ```

   To disable SQL tracing for an entire database, invoke the `DATABASE_TRACE_DISABLE` procedure without specifying the `instance_name` parameter:

   ```
   EXECUTE DBMS_MONITOR.DATABASE_TRACE_DISABLE();
   ```

23.4 Generating Output Files Using SQL Trace and TKPROF

This section explains the basic procedure for using SQL Trace and TKPROF.

The procedure for generating output files is as follows:

1. Set initialization parameters for trace file management.

   See "Step 1: Setting Initialization Parameters for Trace File Management".
2. Enable the SQL Trace facility for the desired session, and run the application. This step produces a trace file containing statistics for the SQL statements issued by the application.

   See "Step 2: Enabling the SQL Trace Facility".
3. Run TKPROF to translate the trace file created in Step 2 into a readable output file. This step can optionally create a SQL script that you can use to store the statistics in a database.

   See "Step 3: Generating Output Files with TKPROF".
4. Optionally, run the SQL script produced in Step 3 to store the statistics in the database.

   See "Step 4: Storing SQL Trace Facility Statistics".

23.4.1 Step 1: Setting Initialization Parameters for Trace File Management

To enable trace files, you must ensure that specific initialization parameters are set.

When the SQL Trace facility is enabled for a session, Oracle Database generates a trace file containing statistics for traced SQL statements for that session. When the SQL Trace facility is enabled for an instance, Oracle Database creates a separate trace file for each process.

To set initialization parameters for trace file management:

1. Check the settings of the `TIMED_STATISTICS`, `MAX_DUMP_FILE_SIZE`, and `DIAGNOSTIC_DEST` initialization parameters, as indicated in "Table 23-1".

Table 23-1 Initialization Parameters to Check Before Enabling SQL Trace

Parameter	Description
`DIAGNOSTIC_DEST`	Specifies the location of the Automatic Diagnostic Repository (ADR) Home. The diagnostic files for each database instance are located in this dedicated directory.
`MAX_DUMP_FILE_SIZE`	When the SQL Trace facility is enabled at the database instance level, every call to the database writes a text line in a file in the operating system's file format. The maximum size of these files in operating system blocks is limited by this initialization parameter. The default is `UNLIMITED`.
`TIMED_STATISTICS`	Enables and disables the collection of timed statistics, such as CPU and elapsed times, by the SQL Trace facility, and the collection of various statistics in the `V$` views. If `STATISTICS_LEVEL` is set to `TYPICAL` or `ALL`, then the default value of `TIMED_STATISTICS` is `true`. If `STATISTICS_LEVEL` is set to `BASIC`, then the default value of `TIMED_STATISTICS` is `false`. Enabling timing causes extra timing calls for low-level operations. This is a dynamic parameter. It is also a session parameter.

2. Devise a way of recognizing the resulting trace file.

   Be sure you know how to distinguish the trace files by name. You can tag trace files by including in your programs a statement such as `SELECT` *`program_name`*`' FROM DUAL`. You can then trace each file back to the process that created it.

   You can also set the `TRACEFILE_IDENTIFIER` initialization parameter to specify a custom identifier that becomes part of the trace file name. For example, you can add *`my_trace_id`* to subsequent trace file names for easy identification with the following:

```
ALTER SESSION SET TRACEFILE_IDENTIFIER = 'my_trace_id';
```

3. If the operating system retains multiple versions of files, then ensure that the version limit is high enough to accommodate the number of trace files you expect the SQL Trace facility to generate.
4. If the generated trace files can be owned by an operating system user other than yourself, then ensure that you have the necessary permissions to use TKPROF to format them.

See Also:

- *Oracle Database Reference* to learn about the `DIAGNOSTIC_DEST`, `STATISTICS_LEVEL`, `TIMED_STATISTICS`, and `TRACEFILE_IDENTIFIER` initialization parameters
- *Oracle Database Administrator's Guide* to learn how to control the trace file size

23.4.2 Step 2: Enabling the SQL Trace Facility

You can enable the SQL Trace facility at the instance or session level.

The package to use depends on the level:

- Database instance

  Use `DBMS_MONITOR.DATABASE_TRACE_ENABLE` procedure to enable tracing, and `DBMS_MONITOR.DATABASE_TRACE_DISABLE` procedure to disable tracing.

- Database session

  Use `DBMS_SESSION.SET_SQL_TRACE` procedure to enable tracing (`true`) or disable tracing (`false`).

> **Note:**
>
> Because running the SQL Trace facility increases system overhead, enable it only when tuning SQL statements, and disable it when you are finished.

To enable and disable tracing at the database instance level:

1. Start SQL*Plus, and connect to the database with administrator privileges.
2. Enable tracing at the database instance level.

   The following example enables tracing for the `orcl` instance:

   ```
   EXEC DBMS_MONITOR.DATABASE_TRACE_ENABLE(INSTANCE_NAME => 'orcl');
   ```

3. Execute the statements to be traced.
4. Disable tracing for the database instance.

   The following example disables tracing for the `orcl` instance:

   ```
   EXEC DBMS_MONITOR.DATABASE_TRACE_DISABLE(INSTANCE_NAME => 'orcl');
   ```

To enable and disable tracing at the session level:

1. Start SQL*Plus, and connect to the database with the desired credentials.
2. Enable tracing for the current session.

   The following example enables tracing for the current session:

   ```
   EXEC DBMS_SESSION.SET_SQL_TRACE(sql_trace => true);
   ```

3. Execute the statements to be traced.
4. Disable tracing for the current session.

The following example disables tracing for the current session:

```
EXEC DBMS_SESSION.SET_SQL_TRACE(sql_trace => false);
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about `DBMS_MONITOR.DATABASE_TRACE_ENABLE`

23.4.3 Step 3: Generating Output Files with TKPROF

TKPROF accepts as input a trace file produced by the SQL Trace facility, and it produces a formatted output file. TKPROF can also generate execution plans.

After the SQL Trace facility has generated trace files, you can:

- Run TKPROF on each individual trace file, producing several formatted output files, one for each session.
- Concatenate the trace files, and then run TKPROF on the result to produce a formatted output file for the entire instance.
- Run the TRCSESS command-line utility to consolidate tracing information from several trace files, then run TKPROF on the result.

TKPROF does not report `COMMIT` and `ROLLBACK` statements recorded in the trace file.

Note:

The following SQL statements are truncated to 25 characters in the SQL Trace file:

```
SET ROLE
GRANT
ALTER USER
ALTER ROLE
CREATE USER
CREATE ROLE
```

Example 23-1 TKPROF Output

```
SELECT * FROM emp, dept
WHERE emp.deptno = dept.deptno;

call    count       cpu    elapsed      disk    query current     rows
---- -------  -------  --------- --------- --------- -------  ------
Parse       1      0.16       0.29         3       13        0        0
Execute     1      0.00       0.00         0        0        0        0
Fetch       1      0.03       0.26         2        2        4       14

Misses in library cache during parse: 1
```

```
Parsing user id: (8) SCOTT

Rows     Execution Plan
-------  --------------------------------------------------- 14  MERGE
JOIN
 4    SORT JOIN
 4      TABLE ACCESS (FULL) OF 'DEPT'
14     SORT JOIN
14       TABLE ACCESS (FULL) OF 'EMP'
```

For this statement, TKPROF output includes the following information:

- The text of the SQL statement
- The SQL Trace statistics in tabular form
- The number of library cache misses for the parsing and execution of the statement.
- The user initially parsing the statement.
- The execution plan generated by `EXPLAIN PLAN`.

TKPROF also provides a summary of user level statements and recursive SQL calls for the trace file.

23.4.4 Step 4: Storing SQL Trace Facility Statistics

You might want to keep a history of the statistics generated by the SQL Trace facility for an application, and compare them over time.

`TKPROF` can generate a SQL script that creates a table and inserts rows of statistics into it. This script contains the following:

- A `CREATE TABLE` statement that creates an output table named `TKPROF_TABLE`.
- `INSERT` statements that add rows of statistics, one for each traced SQL statement, to `TKPROF_TABLE`.

After running `TKPROF`, run this script to store the statistics in the database.

23.4.4.1 Generating the TKPROF Output SQL Script

When you run `TKPROF`, use the `INSERT` parameter to specify the name of the generated SQL script.

If you omit the `INSERT` parameter, then `TKPROF` does not generate a script.

23.4.4.2 Editing the TKPROF Output SQL Script

After `TKPROF` has created the SQL script, you might want to edit the script before running it.

If you have created an output table for previously collected statistics, and if you want to add new statistics to this table, then remove the `CREATE TABLE` statement from the script. The script then inserts the new rows into the existing table. If you have created multiple output tables, perhaps to store statistics from different databases in different tables, then edit the CREATE TABLE and INSERT statements to change the name of the output table.

23.4.4.3 Querying the Output Table

After you have created the output table, query using a SELECT statement.

The following CREATE TABLE statement creates the TKPROF_TABLE:

```
CREATE TABLE TKPROF_TABLE (

DATE_OF_INSERT     DATE,
CURSOR_NUM         NUMBER,
DEPTH              NUMBER,
USER_ID            NUMBER,
PARSE_CNT          NUMBER,
PARSE_CPU          NUMBER,
PARSE_ELAP         NUMBER,
PARSE_DISK         NUMBER,
PARSE_QUERY        NUMBER,
PARSE_CURRENT      NUMBER,
PARSE_MISS         NUMBER,
EXE_COUNT          NUMBER,
EXE_CPU            NUMBER,
EXE_ELAP           NUMBER,
EXE_DISK           NUMBER,
EXE_QUERY          NUMBER,
EXE_CURRENT        NUMBER,
EXE_MISS           NUMBER,
EXE_ROWS           NUMBER,
FETCH_COUNT        NUMBER,
FETCH_CPU          NUMBER,
FETCH_ELAP         NUMBER,
FETCH_DISK         NUMBER,
FETCH_QUERY        NUMBER,
FETCH_CURRENT      NUMBER,
FETCH_ROWS         NUMBER,
CLOCK_TICKS        NUMBER,
SQL_STATEMENT      LONG);
```

Most output table columns correspond directly to the statistics that appear in the formatted output file. For example, the PARSE_CNT column value corresponds to the count statistic for the parse step in the output file.

The columns in the following table help you identify a row of statistics.

Table 23-2 TKPROF_TABLE Columns for Identifying a Row of Statistics

Column	Description
SQL_STATEMENT	This is the SQL statement for which the SQL Trace facility collected the row of statistics. Because this column has data type LONG, you cannot use it in expressions or WHERE clause conditions.

Table 23-2 (Cont.) TKPROF_TABLE Columns for Identifying a Row of Statistics

Column	Description
`DATE_OF_INSERT`	This is the date and time when the row was inserted into the table. This value is different from the time when the SQL Trace facility collected the statistics.
`DEPTH`	This indicates the level of recursion at which the SQL statement was issued. For example, a value of 0 indicates that a user issued the statement. A value of 1 indicates that Oracle Database generated the statement as a recursive call to process a statement with a value of 0 (a statement issued by a user). A value of *n* indicates that Oracle Database generated the statement as a recursive call to process a statement with a value of *n*-1.
`USER_ID`	This identifies the user issuing the statement. This value also appears in the formatted output file.
`CURSOR_NUM`	Oracle Database uses this column value to keep track of the cursor to which each SQL statement was assigned.

The output table does not store the statement's execution plan. The following query returns the statistics from the output table. These statistics correspond to the formatted output shown in "Example 23-7".

```
SELECT * FROM TKPROF_TABLE;
```

Sample output appears as follows:

```
DATE_OF_INSERT CURSOR_NUM DEPTH USER_ID PARSE_CNT PARSE_CPU PARSE_ELAP
-------------- ---------- ----- ------- --------- --------- ----------
21-DEC-2017             1     0       8         1        16         22

PARSE_DISK PARSE_QUERY PARSE_CURRENT PARSE_MISS EXE_COUNT EXE_CPU
---------- ----------- ------------- ---------- --------- -------
    3           11             0             1          1       0

EXE_ELAP EXE_DISK EXE_QUERY EXE_CURRENT EXE_MISS EXE_ROWS FETCH_COUNT
-------- -------- --------- ----------- -------- -------- -----------
    0        0        0          0          0        0         1

FETCH_CPU FETCH_ELAP FETCH_DISK FETCH_QUERY FETCH_CURRENT FETCH_ROWS
--------- ---------- ---------- ----------- ------------- ----------
     2         20          2          2             4            10

SQL_STATEMENT
---------------------------------------------------------------------
SELECT * FROM EMP, DEPT WHERE EMP.DEPTNO = DEPT.DEPTNO
```

23.5 Guidelines for Interpreting TKPROF Output

While `TKPROF` provides a useful analysis, the most accurate measure of efficiency is the performance of the application. At the end of the `TKPROF` output is a summary of the work that the process performed during the period that the trace was running.

23.5.1 Guideline for Interpreting the Resolution of Statistics

Timing statistics have a resolution of one hundredth of a second. Therefore, any operation on a cursor that takes a hundredth of a second or less might not be timed accurately.

Keep the time limitation in mind when interpreting statistics. In particular, be careful when interpreting the results from simple queries that execute very quickly.

23.5.2 Guideline for Recursive SQL Statements

Recursive SQL is additional SQL that Oracle Database must issue to execute a SQL statement issued by a user.

Conceptually, recursive SQL is “side-effect SQL.” For example, if a session inserts a row into a table that has insufficient space to hold that row, then the database makes recursive SQL calls to allocate the space dynamically. The database also generates recursive calls when data dictionary information is not available in memory and so must be retrieved from disk.

If recursive calls occur while the SQL Trace facility is enabled, then `TKPROF` produces statistics for the recursive SQL statements and marks them clearly as recursive SQL statements in the output file. You can suppress the listing of Oracle Database internal recursive calls (for example, space management) in the output file by setting the `SYS` command-line parameter to `NO`. The statistics for a recursive SQL statement are included in the listing for that statement, not in the listing for the SQL statement that caused the recursive call. So, when you are calculating the total resources required to process a SQL statement, consider the statistics for that statement and those for recursive calls caused by that statement.

Note:

Recursive SQL statistics are not included for SQL-level operations.

23.5.3 Guideline for Deciding Which Statements to Tune

You must determine which SQL statements use the most CPU or disk resource.

If the `TIMED_STATISTICS` parameter is enabled, then you can find high CPU activity in the `CPU` column. If `TIMED_STATISTICS` is not enabled, then check the `QUERY` and `CURRENT` columns.

With the exception of locking problems and inefficient PL/SQL loops, neither the CPU time nor the elapsed time is necessary to find problem statements. The key is the number of block visits, both query (that is, subject to read consistency) and current (that is, not subject to read consistency). Segment headers and blocks that are going to be updated are acquired in current mode, but all query and subquery processing requests the data in query mode. These are precisely the same measures as the instance statistics `CONSISTENT GETS` and `DB BLOCK GETS`. You can find high disk activity in the `disk` column.

The following listing shows `TKPROF` output for one SQL statement as it appears in the output file:

```
SELECT *
FROM emp, dept
WHERE emp.deptno = dept.deptno;

call   count      cpu    elapsed     disk    query current    rows
---- -------  -------  --------- -------- -------- -------  ------
Parse     11     0.08       0.18        0        0       0       0
Execute   11     0.23       0.66        0        3       6       0
Fetch     35     6.70       6.83      100    12326       2     824
------------------------------------------------------------------
total     57     7.01       7.67      100    12329       8     826

Misses in library cache during parse: 0
```

If it is acceptable to have 7.01 CPU seconds and to retrieve 824 rows, then you need not look any further at this trace output. In fact, a major use of `TKPROF` reports in a tuning exercise is to eliminate processes from the detailed tuning phase.

The output indicates that 10 unnecessary parse call were made (because 11 parse calls exist for this single statement) and that array fetch operations were performed. More rows were fetched than there were fetches performed. A large gap between `CPU` and `elapsed` timings indicates Physical I/Os.

See Also:

"Example 23-4"

23.5.4 Guidelines for Avoiding Traps in TKPROF Interpretation

When interpreting `TKPROF` output, it helps to be aware of common traps.

23.5.4.1 Guideline for Avoiding the Argument Trap

If you are not aware of the values being bound at run time, then it is possible to fall into the argument trap.

`EXPLAIN PLAN` cannot determine the type of a bind variable from the text of SQL statements, and it always assumes that the type is `VARCHAR`. If the bind variable is actually a number or a date, then `TKPROF` can cause implicit data conversions, which can cause inefficient plans to be executed. To avoid this situation, experiment with different data types in the query, and perform the conversion yourself.

23.5.4.2 Guideline for Avoiding the Read Consistency Trap

Without knowing that an uncommitted transaction had made a series of updates to a column, it is difficult to see why so many block visits would be incurred.

Such cases are not normally repeatable. If the process were run again, it is unlikely that another transaction would interact with it in the same way.

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call     count     cpu     elapsed     disk     query current     rows
----     -----     ---     -------     ----     ----- -------     ----
Parse        1    0.10        0.18        0         0       0        0
Execute      1    0.00        0.00        0         0       0        0
Fetch        1    0.11        0.21        2       101       0        1

Misses in library cache during parse: 1
Parsing user id: 01 (USER1)

Rows     Execution Plan
----     --------- ----
   0     SELECT STATEMENT
   1       TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
   2         INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON_UNIQUE)
```

23.5.4.3 Guideline for Avoiding the Schema Trap

In some cases, an apparently straightforward indexed query looks at many database blocks and accesses them in current mode.

The following example shows an extreme (and thus easily detected) example of the schema trap:

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call        count         cpu      elapsed     disk  query current rows
--------  -------   ---------    ---------  -------- ------ ------- ----
Parse           1        0.06         0.10        0      0       0    0
Execute         1        0.02         0.02        0      0       0    0
Fetch           1        0.23         0.30       31     31       3    1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
   2340    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      0      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

Two statistics suggest that the query might have been executed with a full table scan: the current mode block visits, and the number of rows originating from the Table Access row source in the plan. The explanation is that the required index was built after the trace file had been produced, but before `TKPROF` had been run. Generating a new trace file gives the following data:

```
SELECT name_id
FROM cq_names
WHERE name = 'FLOOR';

call     count    cpu   elapsed  disk  query current     rows
-----   ------ ------  --------- ----- ------ -------    -----
Parse        1   0.01      0.02     0      0       0        0
Execute      1   0.00      0.00     0      0       0        0
Fetch        1   0.00      0.00     0      2       0        1

Misses in library cache during parse: 0
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  SELECT STATEMENT
      1    TABLE ACCESS (BY ROWID) OF 'CQ_NAMES'
      2      INDEX (RANGE SCAN) OF 'CQ_NAMES_NAME' (NON-UNIQUE)
```

In the correct version, the parse call took 10 milliseconds of CPU time and 20 milliseconds of elapsed time, but the query apparently took no time to execute and perform the fetch. These anomalies arise because the clock tick of 10 milliseconds is too long relative to the time taken to execute and fetch the data. In such cases, it is important to get many executions of the statements, so that you have statistically valid numbers.

23.5.4.4 Guideline for Avoiding the Time Trap

In some cases, a query takes an inexplicably long time.

For example, the following update of 7 rows executes in 19 seconds:

```
UPDATE cq_names
  SET ATTRIBUTES = lower(ATTRIBUTES)
  WHERE ATTRIBUTES = :att

call        count       cpu    elapsed      disk    query current
rows
-------- -------  --------  --------- -------- --------- -------
----------
Parse           1      0.06       0.24        0        0
0           0
Execute         1      0.62      19.62       22      526
12           7
Fetch           0      0.00       0.00        0        0
0           0

Misses in library cache during parse: 1
```

```
Parsing user id: 02  (USER2)

Rows     Execution Plan
-------  ---------------------------------------------------
      0  UPDATE STATEMENT
   2519  TABLE ACCESS (FULL) OF 'CQ_NAMES'
```

The explanation is interference from another transaction. In this case, another transaction held a shared lock on the table `cq_names` for several seconds before and after the update was issued. It takes experience to diagnose that interference effects are occurring. On the one hand, comparative data is essential when the interference is contributing only a short delay (or a small increase in block visits in the previous example). However, if the interference contributes only modest overhead, and if the statement is essentially efficient, then its statistics may not require analysis.

23.6.1 Application Tracing Utilities

The Oracle tracing utilities are TKPROF and TRCSESS.

23.6.1.1 TRCSESS

The TRCSESS utility consolidates trace output from selected trace files based on user-specified criteria.

After TRCSESS merges the trace information into a single output file, TKPROF can process the output file.

23.6.1.1.1 Purpose

TRCSESS is useful for consolidating the tracing of a particular session for performance or debugging purposes.

Tracing a specific session is usually not a problem in the dedicated server model because one process serves a session during its lifetime. You can see the trace information for the session from the trace file belonging to the server process. However, in a shared server configuration, a user session is serviced by different processes over time. The trace for the user session is scattered across different trace files belonging to different processes, which makes it difficult to get a complete picture of the life cycle of a session.

23.6.1.1.2 Guidelines

You must specify one of the `session`, `clientid`, `service`, `action`, or `module` options.

If you specify multiple options, then TRCSESS consolidates all trace files that satisfy the specified criteria into the output file.

23.6.1.1.3 Syntax

```
trcsess  [output=output_file_name]
         [session=session_id]
         [clientid=client_id]
         [service=service_name]
         [action=action_name]
         [module=module_name]
         [trace_files]
```

23.6.1.1.4 Options

TRCSESS supports a number of command-line options.

Argument	Description
`output`	Specifies the file where the output is generated. If this option is not specified, then the utility writes to standard output.
`session`	Consolidates the trace information for the session specified. The session identifier is a combination of session index and session serial number, such as `21.2371`. You can locate these values in the `V$SESSION` view.
`clientid`	Consolidates the trace information for the specified client ID.
`service`	Consolidates the trace information for the specified service name.
`action`	Consolidates the trace information for the specified action name.
`module`	Consolidates the trace information for the specified module name.
`trace_files`	Lists the trace file names, separated by spaces, in which TRCSESS should look for trace information. You can use the wildcard character (`*`) to specify the trace file names. If you do not specify trace files, then TRCSESS uses all files in the current directory as input.

23.6.1.1.5 Examples

This section demonstrates common TRCSESS use cases.

Example 23-2 Tracing a Single Session

This sample output of TRCSESS shows the container of traces for a particular session. In this example, the session index and serial number equals `21.2371`. All files in current directory are taken as input.

```
trcsess session=21.2371
```

Example 23-3 Specifying Multiple Trace Files

The following example specifies two trace files:

```
trcsess session=21.2371 main_12359.trc main_12995.trc
```

The sample output is similar to the following:

```
[PROCESS ID = 12359]
*** 2014-04-02 09:48:28.376
PARSING IN CURSOR #1 len=17 dep=0 uid=27 oct=3 lid=27 tim=868373970961
hv=887450622 ad='22683fb4'
select * from cat
END OF STMT
PARSE #1:c=0,e=339,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373970944
EXEC #1:c=0,e=221,p=0,cr=0,cu=0,mis=0,r=0,dep=0,og=4,tim=868373971411
```

```
FETCH #1:c=0,e=791,p=0,cr=7,cu=0,mis=0,r=1,dep=0,og=4,tim=868373972435
FETCH #1:c=0,e=1486,p=0,cr=20,cu=0,mis=0,r=6,dep=0,og=4,tim=868373986238
*** 2014-04-02 10:03:58.058
XCTEND rlbk=0, rd_only=1
STAT #1 id=1 cnt=7 pid=0 pos=1 obj=0 op='FILTER  '
STAT #1 id=2 cnt=7 pid=1 pos=1 obj=18 op='TABLE ACCESS BY INDEX ROWID OBJ$ '
STAT #1 id=3 cnt=7 pid=2 pos=1 obj=37 op='INDEX RANGE SCAN I_OBJ2 '
STAT #1 id=4 cnt=0 pid=1 pos=2 obj=4 op='TABLE ACCESS CLUSTER TAB$J2 '
STAT #1 id=5 cnt=6 pid=4 pos=1 obj=3 op='INDEX UNIQUE SCAN I_OBJ# '
[PROCESS ID=12995]
*** 2014-04-02 10:04:32.738
Archiving is disabled
```

23.6.1.2 TKPROF

The TKPROF program formats the contents of the trace file and places the output into a readable output file.

TKPROF can also do the following:

- Create a SQL script that stores the statistics in the database
- Determine the execution plans of SQL statements

> **Note:**
>
> If the cursor for a SQL statement is not closed, then TKPROF output does not automatically include the actual execution plan of the SQL statement. In this situation, use the `EXPLAIN` option with TKPROF to generate an execution plan.

TKPROF reports each statement executed with the resources it has consumed, the number of times it was called, and the number of rows which it processed.

23.6.1.2.1 Purpose

TKPROF can locate statements that are consuming the greatest resources.

With baselines available, you can assess whether the resources used are reasonable given the work performed.

23.6.1.2.2 Guidelines

The input and output files are the only required arguments.

If you invoke TKPROF without arguments, then the tool displays online help.

23.6.1.2.3 Syntax

```
tkprof input_file output_file
  [ waits=yes|no ]
  [ sort=option ]
  [ print=n ]
  [ aggregate=yes|no ]
  [ insert=filename3 ]
  [ sys=yes|no ]
  [ table=schema.table ]
  [ explain=user/password ]
  [ record=filename4 ]
  [ width=n ]
```

23.6.1.2.4 Options

TKPROF supports a number of command-line options.

Table 23-3 TKPROF Arguments

Argument	Description
input_file	Specifies the input file, a trace file containing statistics produced by the SQL Trace facility. This file can be either a trace file produced for a single session, or a file produced by concatenating individual trace files from multiple sessions.
output_file	Specifies the file to which `TKPROF` writes its formatted output.
`WAITS`	Specifies whether to record summary for any wait events found in the trace file. Valid values are `YES` (default) and `NO`.
`SORT`	Sorts traced SQL statements in descending order of specified sort option before listing them in the output file. If multiple options are specified, then the output is sorted in descending order by the sum of the values specified in the sort options. If you omit this parameter, then `TKPROF` lists statements into the output file in order of first use. Sort options are listed as follows: • `PRSCNT` - Number of times parsed • `PRSCPU` - CPU time spent parsing • `PRSELA` - Elapsed time spent parsing • `PRSDSK` - Number of physical reads from disk during parse • `PRSQRY` - Number of consistent mode block reads during parse • `PRSCU` - Number of current mode block reads during parse • `PRSMIS` - Number of library cache misses during parse • `EXECNT` - Number of executions • `EXECPU` - CPU time spent executing • `EXEELA` - Elapsed time spent executing • `EXEDSK` - Number of physical reads from disk during execute • `EXEQRY` - Number of consistent mode block reads during execute • `EXECU` - Number of current mode block reads during execute • `EXEROW` - Number of rows processed during execute • `EXEMIS` - Number of library cache misses during execute • `FCHCNT` - Number of fetches • `FCHCPU` - CPU time spent fetching • `FCHELA` - Elapsed time spent fetching • `FCHDSK` - Number of physical reads from disk during fetch • `FCHQRY` - Number of consistent mode block reads during fetch • `FCHCU` - Number of current mode block reads during fetch • `FCHROW` - Number of rows fetched • `USERID` - ID of user that parsed the cursor

Table 23-3 (Cont.) TKPROF Arguments

Argument	Description
`PRINT`	Lists only the first integer sorted SQL statements from the output file. If you omit this parameter, then `TKPROF` lists all traced SQL statements. This parameter does not affect the optional SQL script. The SQL script always generates insert data for all traced SQL statements.
`AGGREGATE`	If you specify `AGGREGATE` = `NO`, then `TKPROF` does not aggregate multiple users of the same SQL text.
`INSERT`	Creates a SQL script that stores the trace file statistics in the database. `TKPROF` creates this script with the name *`filename3`*. This script creates a table and inserts a row of statistics for each traced SQL statement into the table.
`SYS`	Enables and disables the listing of SQL statements issued by the user `SYS`, or recursive SQL statements, into the output file. The default value of `YES` causes `TKPROF` to list these statements. The value of `NO` causes `TKPROF` to omit them. This parameter does not affect the optional SQL script. The SQL script always inserts statistics for all traced SQL statements, including recursive SQL statements.
`TABLE`	Specifies the schema and name of the table into which `TKPROF` temporarily places execution plans before writing them to the output file. If the specified table exists, then `TKPROF` deletes all rows in the table, uses it for the `EXPLAIN PLAN` statement (which writes more rows into the table), and then deletes those rows. If this table does not exist, then `TKPROF` creates it, uses it, and then drops it. The specified user must be able to issue `INSERT`, `SELECT`, and `DELETE` statements against the table. If the table does not exist, then the user must also be able to issue `CREATE TABLE` and `DROP TABLE` statements. This option enables multiple users to run `TKPROF` concurrently with the same database user account in the `EXPLAIN` value. These users can specify different `TABLE` values and avoid destructively interfering with each other when processing the temporary plan table. `TKPROF` supports the following combinations: • The `EXPLAIN` parameter without the `TABLE` parameter `TKPROF` uses the table `PROF$PLAN_TABLE` in the schema of the user specified by the `EXPLAIN` parameter • The `TABLE` parameter without the `EXPLAIN` parameter `TKPROF` ignores the `TABLE` parameter. If no plan table exists, then `TKPROF` creates the table `PROF$PLAN_TABLE` and then drops it at the end.

Table 23-3 (Cont.) TKPROF Arguments

Argument	Description
EXPLAIN	Determines the execution plan for each SQL statement in the trace file and writes these execution plans to the output file. TKPROF also displays the number of rows processed by each step of the execution plan. TKPROF determines execution plans by issuing the EXPLAIN PLAN statement after connecting to Oracle Database with the user and password specified in this parameter. The specified user must have CREATE SESSION system privileges. TKPROF takes longer to process a large trace file if the EXPLAIN option is used. **Note:** Trace files generated immediately after instance startup contain data that reflects the activity of the startup process. In particular, they reflect a disproportionate amount of I/O activity as caches in the system global area (SGA) are filled. For the purposes of tuning, ignore such trace files.
RECORD	Creates a SQL script with the specified *filename* with all of the nonrecursive SQL in the trace file. You can use this script to replay the user events from the trace file.
WIDTH	An integer that controls the output line width of some TKPROF output, such as the explain plan. This parameter is useful for post-processing of TKPROF output.

23.6.1.2.5 Output

This section explains the TKPROF output.

23.6.1.2.5.1 Identification of User Issuing the SQL Statement in TKPROF

TKPROF lists the user ID of the user issuing each SQL statement.

If the SQL Trace input file contained statistics from multiple users, and if the statement was issued by multiple users, then TKPROF lists the ID of the last user to parse the statement. The user ID of all database users appears in the data dictionary in the column ALL_USERS.USER_ID.

23.6.1.2.5.2 Tabular Statistics in TKPROF

TKPROF lists the statistics for a SQL statement returned by the SQL Trace facility in rows and columns.

Each row corresponds to one of three steps of SQL statement processing. Statistics are identified by the value of the CALL column. See Table 23-4.

Table 23-4 CALL Column Values

CALL Value	Meaning
PARSE	Translates the SQL statement into an execution plan, including checks for proper security authorization and checks for the existence of tables, columns, and other referenced objects.

Table 23-4 (Cont.) CALL Column Values

CALL Value	Meaning
EXECUTE	Actual execution of the statement by Oracle Database. For INSERT, UPDATE, DELETE, and MERGE statements, this modifies the data. For SELECT statements, this identifies the selected rows.
FETCH	Retrieves rows returned by a query. Fetches are only performed for SELECT statements.

The other columns of the SQL Trace facility output are combined statistics for all parses, executions, and fetches of a statement. The sum of `query` and `current` is the total number of buffers accessed, also called Logical I/Os (LIOs). See Table 23-5.

Table 23-5 SQL Trace Statistics for Parses, Executes, and Fetches.

SQL Trace Statistic	Meaning
COUNT	Number of times a statement was parsed, executed, or fetched.
CPU	Total CPU time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not enabled.
ELAPSED	Total elapsed time in seconds for all parse, execute, or fetch calls for the statement. This value is zero (0) if TIMED_STATISTICS is not enabled.
DISK	Total number of data blocks physically read from the data files on disk for all parse, execute, or fetch calls.
QUERY	Total number of buffers retrieved in consistent mode for all parse, execute, or fetch calls. Usually, buffers are retrieved in consistent mode for queries.
CURRENT	Total number of buffers retrieved in current mode. Buffers are retrieved in current mode for statements such as INSERT, UPDATE, and DELETE.

Statistics about the processed rows appear in the `ROWS` column. The column shows the number of rows processed by the SQL statement. This total does not include rows processed by subqueries of the SQL statement. For `SELECT` statements, the number of rows returned appears for the fetch step. For `UPDATE`, `DELETE`, and `INSERT` statements, the number of rows processed appears for the execute step.

Note:

The row source counts are displayed when a cursor is closed. In SQL*Plus, there is only one user cursor, so each statement executed causes the previous cursor to be closed; therefore, the row source counts are displayed. PL/SQL has its own cursor handling and does not close child cursors when the parent cursor is closed. Exiting or reconnecting causes the counts to be displayed.

23.6.1.2.5.3 Library Cache Misses in TKPROF

TKPROF also lists the number of library cache misses resulting from parse and execute steps for each SQL statement.

These statistics appear on separate lines following the tabular statistics. If the statement resulted in no library cache misses, then `TKPROF` does not list the statistic. In "Examples", the statement resulted in one library cache miss for the parse step and no misses for the execute step.

23.6.1.2.5.4 Row Source Operations in TKPROF

In the TKPROF output, row source operations show the number of rows processed for each operation executed on the rows, and additional row source information, such as physical reads and writes.

Table 23-6 Row Source Operations

Row Source Operation	Meaning
`cr`	Consistent reads performed by the row source.
`r`	Physical reads performed by the row source
`w`	Physical writes performed by the row source
`time`	Time in microseconds

In the following sample `TKPROF` output, note the `cr`, `r`, `w`, and `time` values under the Row Source Operation column:

```
Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                      (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)
```

23.6.1.2.5.5 Wait Event Information in TKPROF

If wait event information exists, then the `TKPROF` output includes a section on wait events.

Output looks similar to the following:

```
Elapsed times include waiting on following events:

  Event waited on                Times Waited   Max. Wait  Total
Waited
  ---------------------------     ------------   ---------
------------
  db file sequential read                8084        0.12
5.34
  direct path write                       834        0.00
0.00
  direct path write temp                  834        0.00
0.05
  db file parallel read                     8        1.53
5.51
```

```
  db file scattered read                  4180        0.07          1.45
  direct path read                        7082        0.00          0.05
  direct path read temp                   7082        0.00          0.44
  rdbms ipc reply                           20        0.00          0.01
  SQL*Net message to client                  1        0.00          0.00
  SQL*Net message from client                1        0.00          0.00
```

In addition, wait events are summed for the entire trace file at the end of the file.

To ensure that wait events information is written to the trace file for the session, run the following SQL statement:

```
ALTER SESSION SET EVENTS '10046 trace name context forever, level 8';
```

23.6.1.2.6 Examples

This section demonstrates common TKPROF use cases.

Example 23-4 Printing the Most Resource-Intensive Statements

If you are processing a large trace file using a combination of `SORT` parameters and the `PRINT` parameter, then you can produce a `TKPROF` output file containing only the highest resource-intensive statements. The following statement prints the 10 statements in the trace file that have generated the most physical I/O:

```
TKPROF ora53269.trc ora53269.prf SORT = (PRSDSK, EXEDSK, FCHDSK) PRINT = 10
```

Example 23-5 Generating a SQL Script

This example runs `TKPROF`, accepts a trace file named `examp12_jane_fg_sqlplus_007.trc`, and writes a formatted output file named `outputa.prf`:

```
TKPROF examp12_jane_fg_sqlplus_007.trc OUTPUTA.PRF EXPLAIN=hr
  TABLE=hr.temp_plan_table_a INSERT=STOREA.SQL SYS=NO SORT=(EXECPU,FCHCPU)
```

This example is likely to be longer than a single line on the screen, and you might need to use continuation characters, depending on the operating system.

Note the other parameters in this example:

- The `EXPLAIN` value causes `TKPROF` to connect as the user `hr` and use the `EXPLAIN PLAN` statement to generate the execution plan for each traced SQL statement. You can use this to get access paths and row source counts.

  Note:

  If the cursor for a SQL statement is not closed, then `TKPROF` output does not automatically include the actual execution plan of the SQL statement. In this situation, you can use the `EXPLAIN` option with `TKPROF` to generate an execution plan.

- The `TABLE` value causes `TKPROF` to use the table `temp_plan_table_a` in the schema `scott` as a temporary plan table.
- The `INSERT` value causes `TKPROF` to generate a SQL script named `STOREA.SQL` that stores statistics for all traced SQL statements in the database.
- The `SYS` parameter with the value of `NO` causes `TKPROF` to omit recursive SQL statements from the output file. In this way, you can ignore internal Oracle Database statements such as temporary table operations.
- The `SORT` value causes `TKPROF` to sort the SQL statements in order of the sum of the CPU time spent executing and the CPU time spent fetching rows before writing them to the output file. For greatest efficiency, always use `SORT` parameters.

Example 23-6 TKPROF Header

This example shows a sample header for the TKPROF report.

```
TKPROF: Release 12.1.0.0.2

Copyright (c) 1982, 2012, Oracle and/or its affiliates.  All rights
reserved.

Trace file: /disk1/oracle/log/diag/rdbms/orcla/orcla/trace/
orcla_ora_917.trc
Sort options: default

***********************************************************************
****
count    = number of times OCI procedure was executed
cpu      = cpu time in seconds executing
elapsed  = elapsed time in seconds executing
disk     = number of physical reads of buffers from disk
query    = number of buffers gotten for consistent read
current  = number of buffers gotten in current mode (usually for
update)
rows     = number of rows processed by the fetch or execute call
***********************************************************************
****
```

Example 23-7 TKPROF Body

This example shows a sample body for a TKPROF report.

```
call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.01       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        0      0.00       0.00          0          0          0           0
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        2      0.01       0.00          0          0          0           0

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 44
```

```
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1       28.59         28.59
********************************************************************************

select condition
from
 cdef$ where rowid=:1

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
Optimizer mode: CHOOSE
Parsing user id: SYS   (recursive depth: 1)

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS BY USER ROWID OBJ#(31) (cr=1 r=0 w=0 time=151 us)

********************************************************************************

SELECT last_name, job_id, salary
  FROM employees
WHERE salary =
  (SELECT max(salary) FROM employees)

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.02       0.01          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        4      0.02       0.01          0         15          0           1

Misses in library cache during parse: 1
Optimizer mode: FIRST_ROWS
Parsing user id: 44

Rows     Row Source Operation
-------  ---------------------------------------------------
      1  TABLE ACCESS FULL EMPLOYEES (cr=15 r=0 w=0 time=1743 us)
      1   SORT AGGREGATE (cr=7 r=0 w=0 time=777 us)
    107    TABLE ACCESS FULL EMPLOYEES (cr=7 r=0 w=0 time=655 us)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                          2        0.00          0.00
  SQL*Net message from client                        2        9.62          9.62
********************************************************************************

********************************************************************************
 delete
         from stats$sqltext st
        where (hash_value, text_subset) not in
             (select --+ hash_aj
                     hash_value, text_subset
                from stats$sql_summary ss
               where (    (   snap_id     < :lo_snap
                           or snap_id     > :hi_snap
                          )
                          and dbid            = :dbid
                          and instance_number = :inst_num
                      )
                  or (    dbid            != :dbid
                      or instance_number != :inst_num)
              )

call     count       cpu    elapsed       disk      query    current rows
------- ------  -------- ---------- ---------- ---------- ---------- ----------
Parse        1      0.00       0.00          0          0          0          0
Execute      1     29.60      60.68     266984      43776     131172      28144
Fetch        0      0.00       0.00          0          0          0          0
------- ------  -------- ---------- ---------- ---------- ---------- ----------
total        2     29.60      60.68     266984      43776     131172      28144

Misses in library cache during parse: 1
Misses in library cache during execute: 1
Optimizer mode: CHOOSE
Parsing user id: 22

Rows     Row Source Operation
-------  ---------------------------------------------------
      0  DELETE  (cr=43141 r=266947 w=25854 time=60235565 us)
  28144   HASH JOIN ANTI (cr=43057 r=262332 w=25854 time=48830056 us)
  51427    TABLE ACCESS FULL STATS$SQLTEXT (cr=3465 r=3463 w=0 time=865083 us)
 647529    INDEX FAST FULL SCAN STATS$SQL_SUMMARY_PK
                      (cr=39592 r=39325 w=0 time=10522877 us) (object id 7409)

Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  db file sequential read                      8084        0.12          5.34
  direct path write                             834        0.00          0.00
  direct path write temp                        834        0.00          0.05
  db file parallel read                           8        1.53          5.51
  db file scattered read                       4180        0.07          1.45
  direct path read                             7082        0.00          0.05
  direct path read temp                        7082        0.00          0.44
  rdbms ipc reply                                20        0.00          0.01
  SQL*Net message to client                       1        0.00          0.00
  SQL*Net message from client                     1        0.00          0.00
********************************************************************************
```

Example 23-8   TKPROF Summary

This example that shows a summary for the TKPROF report.

```
OVERALL TOTALS FOR ALL NON-RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        4      0.04       0.01          0          0          0           0
Execute      5      0.00       0.04          0          0          0           0
Fetch        2      0.00       0.00          0         15          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total       11      0.04       0.06          0         15          0           1

Misses in library cache during parse: 4
Misses in library cache during execute: 1
Elapsed times include waiting on following events:
  Event waited on                             Times   Max. Wait  Total Waited
  ----------------------------------------   Waited  ----------  ------------
  SQL*Net message to client                       6        0.00          0.00
  SQL*Net message from client                     5       77.77        128.88

OVERALL TOTALS FOR ALL RECURSIVE STATEMENTS

call     count       cpu    elapsed       disk      query    current        rows
------- ------  -------- ---------- ---------- ---------- ----------  ----------
Parse        1      0.00       0.00          0          0          0           0
Execute      1      0.00       0.00          0          0          0           0
Fetch        1      0.00       0.00          0          2          0           1
------- ------  -------- ---------- ---------- ---------- ----------  ----------
total        3      0.00       0.00          0          2          0           1

Misses in library cache during parse: 1
    5  user  SQL statements in session.
    1  internal SQL statements in session.
    6  SQL statements in session.
********************************************************************************
Trace file: main_ora_27621.trc
Trace file compatibility: 9.00.01
Sort options: default
       1  session in tracefile.
       5  user  SQL statements in trace file.
       1  internal SQL statements in trace file.
       6  SQL statements in trace file.
       6  unique SQL statements in trace file.
      76  lines in trace file.
     128  elapsed seconds in trace file.
```

23.7.1 Views for Application Tracing

You can use data dictionary and V$ views to monitor tracing.

This section includes the following topics:

23.7.1.1 Views Relevant for Trace Statistics

You can display the statistics that have been gathered with the following V$ and DBA views.

Table 23-7 Diagnostic Views

View	Description
DBA_ENABLED_AGGREGATIONS	Accumulated global statistics for the currently enabled statistics
V$CLIENT_STATS	Accumulated statistics for a specified client identifier
V$SERVICE_STATS	Accumulated statistics for a specified service
V$SERV_MOD_ACT_STATS	Accumulated statistics for a combination of specified service, module, and action
V$SERVICEMETRIC	Accumulated statistics for elapsed time of database calls and for CPU use
V$DIAG_TRACE_FILE	Information about all trace files in ADR for the current container
V$DIAG_APP_TRACE_FILE	Information about all trace files that contain application trace data (SQL_TRACE or OPTIMIZER_TRACE event data) in ADR for the current container
V$DIAG_TRACE_FILE_CONTENTS	Trace data in the trace files in ADR
V$DIAG_SQL_TRACE_RECORDS	SQL_TRACE data in the trace files in ADR
V$DIAG_OPT_TRACE_RECORDS	Optimizer trace event data in the trace files in ADR
V$DIAG_SESS_SQL_TRACE_RECORDS	SQL_TRACE data in the trace files in ADR for the current user session
V$DIAG_SESS_OPT_TRACE_RECORDS	Optimizer trace event data in the trace files in ADR for the current user session
V$DIAG_ALERT_EXT	Contents of the XML-based alert log in ADR for the current container

See Also:

Oracle Database Reference for information about V$ and data dictionary views

23.7.1.2 Views Related to Enabling Tracing

A Cloud Control report or the DBA_ENABLED_TRACES view can display outstanding traces.

In the DBA_ENABLED_TRACES view, you can determine detailed information about how a trace was enabled, including the trace type. The trace type specifies whether the trace is enabled for client identifier, session, service, database, or a combination of service, module, and action.

Part VIII
Automatic SQL Tuning

SQL Tuning Advisor and SQL Access Advisor are built-in tools that provide SQL tuning recommendations.

24
Capturing Workloads in SQL Tuning Sets

A SQL tuning set is a mechanism to collect, maintain, and access SQL workload data for SQL performance monitoring and tuning.

24.1 About SQL Tuning Sets

A **SQL tuning set (STS)** is a database object that you can use as input to tuning tools.

An STS includes the following components:

- A set of SQL statements
- Associated execution context, such as user schema, application module name and action, list of bind values, and the environment for SQL compilation of the cursor
- Associated basic execution statistics, such as elapsed time, CPU time, buffer gets, disk reads, rows processed, cursor fetches, the number of executions, the number of complete executions, optimizer cost, and the command type
- Associated execution plans and row source statistics for each SQL statement (optional)

Note:

Data visibility and privilege requirements may differ when using an STS with pluggable databases.

24.1.1 Purpose of SQL Tuning Sets

An STS enables you to group SQL statements and related metadata in a single database object, which you can use to meet your tuning goals.

Specifically, SQL tuning sets achieve the following goals:

- Providing input to the performance tuning advisors

  You can use an STS as input to multiple database advisors, including SQL Tuning Advisor, SQL Access Advisor, and SQL Performance Analyzer.
- Transporting SQL between databases

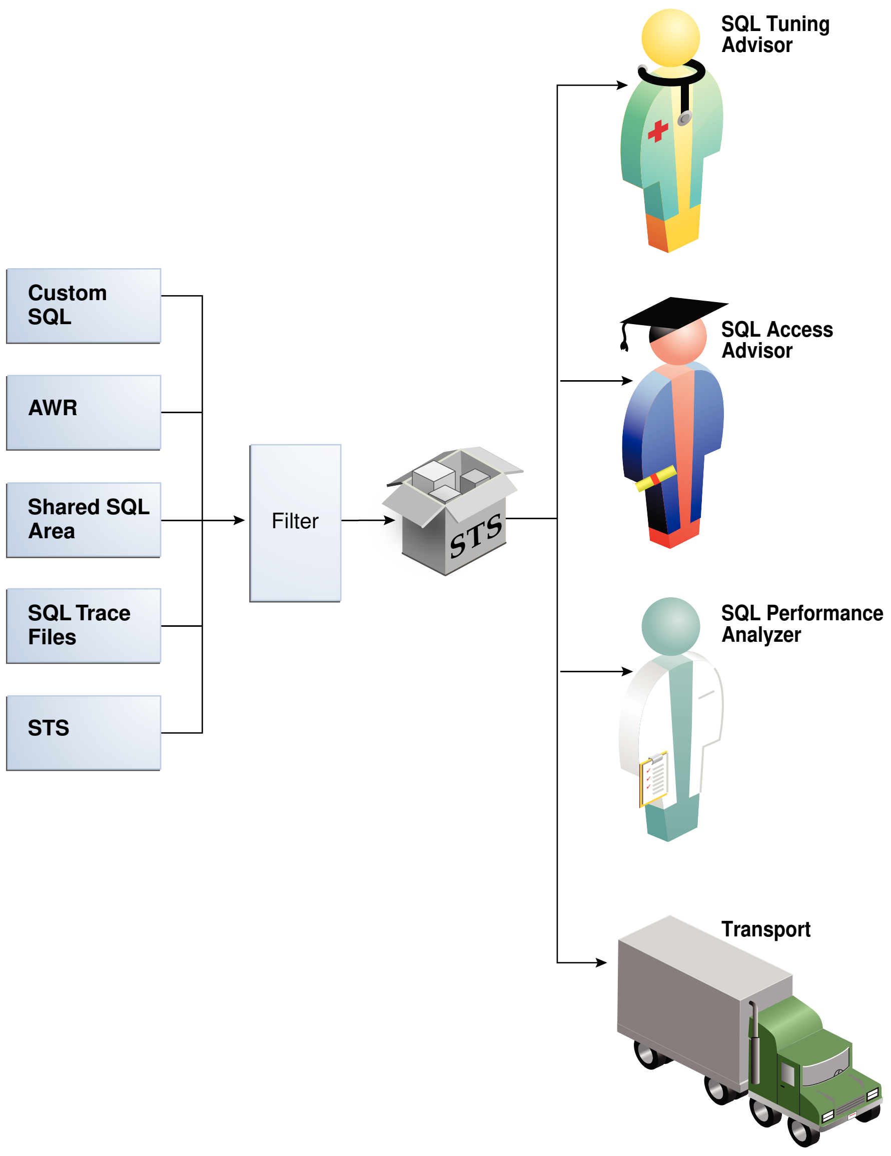
  You can export SQL tuning sets from one database to another, enabling transfer of SQL workloads between databases for remote performance diagnostics and tuning. When suboptimally performing SQL statements occur on a production database, developers may not want to investigate and tune directly on the production database. The DBA can transport the problematic SQL statements to a test database where the developers can safely analyze and tune them.

24.1.2 Concepts for SQL Tuning Sets

To create an STS, you must load SQL statements into an STS from a source.

As shown in the following figure, the source can be the Automatic Workload Repository (AWR), shared SQL area, customized SQL provided by the user, trace files, or another STS.

Figure 24-1 SQL Tuning Sets



SQL tuning sets can do the following:

- Filter SQL statements using the application module name and action, or any execution statistics
- Rank SQL statements based on any combination of execution statistics

- Serve as input to the advisors or transport it to a different database

See Also:

Oracle Database Performance Tuning Guide to learn about AWR

24.1.3 User Interfaces for SQL Tuning Sets

You can use either Oracle Enterprise Manager Cloud Control (Cloud Control) or PL/SQL packages to manage SQL tuning sets. Oracle recommends Cloud Control.

24.1.3.1 Accessing the SQL Tuning Sets Page in Cloud Control

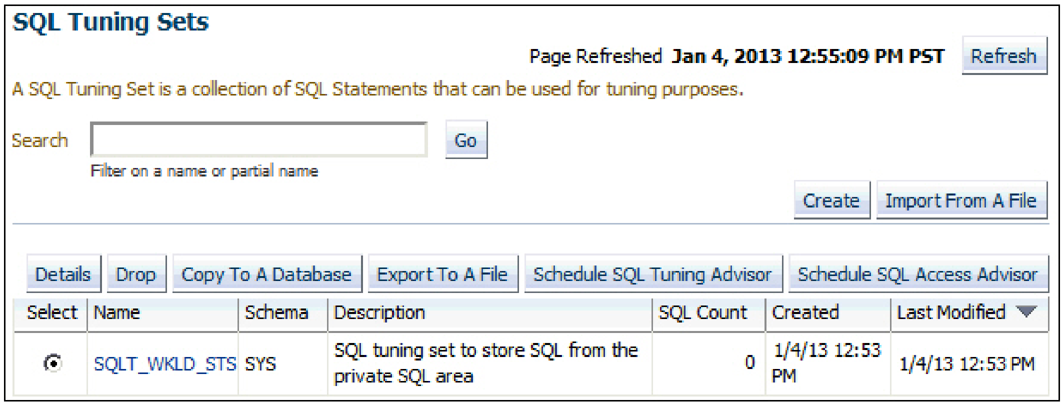
The SQL Tuning Sets page in Cloud Control is the starting page from which you can perform most operations relating to SQL tuning sets.

To access the SQL Tuning Sets page:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Performance** menu, select **SQL**, then **SQL Tuning Sets**.

   The SQL Tuning Sets page appears, as shown in Figure 24-2.

Figure 24-2 SQL Tuning Sets



Page Refreshed **Jan 4, 2013 12:55:09 PM PST**

A SQL Tuning Set is a collection of SQL Statements that can be used for tuning purposes.

Select	Name	Schema	Description	SQL Count	Created	Last Modified
●	SQLT_WKLD_STS	SYS	SQL tuning set to store SQL from the private SQL area	0	1/4/13 12:53 PM	1/4/13 12:53 PM

See Also:

Oracle Database 2 Day + Performance Tuning Guide

24.1.3.2 Command-Line Interface to SQL Tuning Sets

On the command line, you can use the `DBMS_SQLTUNE` or `DBMS_SQLSET` packages to manage SQL tuning sets.

You must have the `ADMINISTER SQL TUNING SET` system privilege to manage SQL tuning sets that you own, or the `ADMINISTER ANY SQL TUNING SET` system privilege to manage any SQL tuning sets.

The traditional package for managing SQL tuning sets is `DBMS_SQLTUNE`, which requires the Oracle Tuning Pack. Starting in Oracle Database 18c, you can perform the same tasks with `DBMS_SQLSET`, which does *not* require the Oracle Tuning Pack. In most cases, the name of the subprogram in `DBMS_SQLSET` is identical to the name of the equivalent subprogram in `DBMS_SQLTUNE`. The following table shows only the subprograms whose names differ.

Table 24-1 Naming Differences for SQL Tuning Set Subprograms

DBMS_SQLTUNE	DBMS_SQLSET
`ADD_SQLSET_REFERENCE`	`ADD_REFERENCE`
`CAPTURE_CURSOR_CACHE_SQLSET`	`CAPTURE_CURSOR_CACHE`
`CREATE_STGTAB_SQLSET`	`CREATE_STGTAB`
`PACK_STGTAB_SQLSET`	`PACK_STGTAB`
`REMAP_STGTAB_SQLSET`	`REMAP_STGTAB`
`REVOVE_SQLSET_REFERENCE`	`REMOVE_REFERENCE`
`UNPACK_STGTAB_SQLSET`	`UNPACK_STGTAB`

See Also:

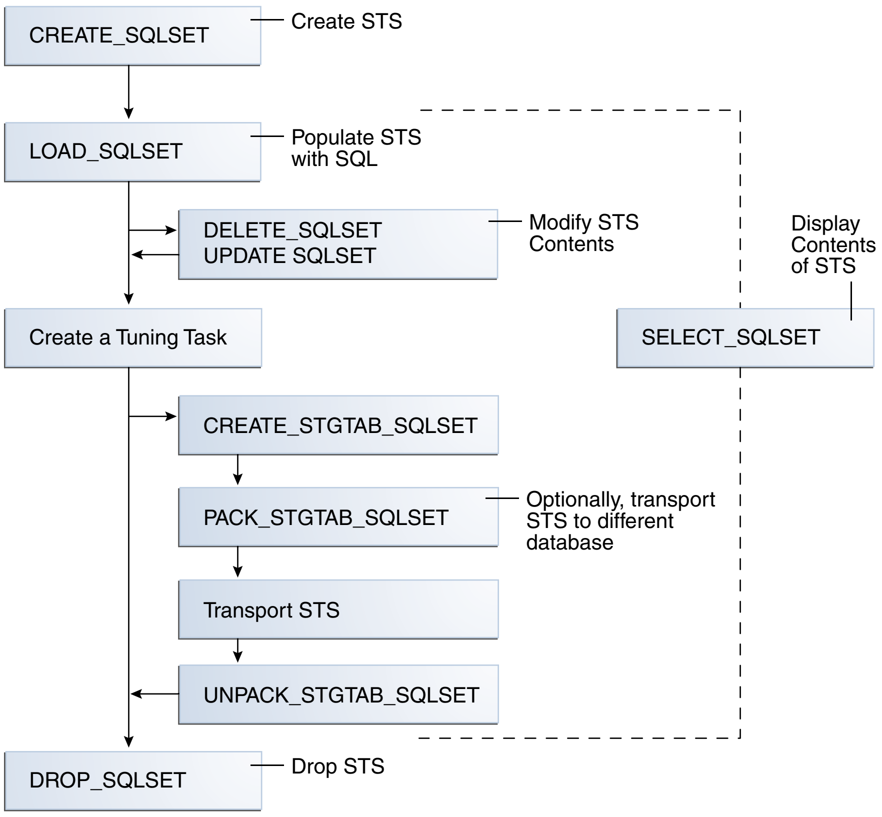
Oracle Database PL/SQL Packages and Types Reference to learn about `DBMS_SQLTUNE` and `DBMS_SQLSET`

24.1.4 Basic Tasks for Managing SQL Tuning Sets

You can use `DBMS_SQLTUNE` or `DBMS_SQLSET` to create, use, and delete SQL tuning sets. In most cases, the relevant subprograms in these packages have identical names.

The following graphic shows the basic workflow.

Figure 24-3 SQL Tuning Sets APIs



Typically, you perform STS operations in the following sequence:

1. Create a new STS.

   "Creating a SQL Tuning Set Using CREATE_SQLSET" describes this task.

2. Load the STS with SQL statements and associated metadata.

   "Loading a SQL Tuning Set Using LOAD_SQLSET" describes this task.

3. Optionally, display the contents of the STS.

   "Querying a SQL Tuning Set" describes this task.

4. Optionally, update or delete the contents of the STS.

   "Modifying a SQL Tuning Set Using UPDATE_SQLSET" describes this task.

5. Create a tuning task with the STS as input.

6. Optionally, transport the STS to another database.

   "Transporting a SQL Tuning Set" describes this task.

7. Drop the STS when finished.

   "Dropping a SQL Tuning Set Using DROP_SQLSET" describes this task.

See Also:

"Command-Line Interface to SQL Tuning Sets" for the names of the equivalent DBMS_SQLSET subprograms

24.2 Creating a SQL Tuning Set Using CREATE_SQLSET

Use the CREATE_SQLSET procedure in DBMS_SQLTUNE or DBMS_SQLSET to create an empty STS in the database.

Using the function instead of the procedure causes the database to generate a name for the STS. The following table describes some procedure parameters.

Table 24-2 DBMS_SQLSET.CREATE_SQLSET Parameters

Parameter	Description
sqlset_name	Name of the STS
description	Optional description of the STS

Assumptions

This tutorial assumes that

- You want to create an STS named SQLT_WKLD_STS.
- You use DBMS_SQLTUNE instead of DBMS_SQLSET.

To create an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use the DBMS_SQLSET.CREATE_SQLSET procedure.

   For example, execute the following PL/SQL program:

```
BEGIN
  DBMS_SQLSET.CREATE_SQLSET (
    sqlset_name  => 'SQLT_WKLD_STS'
,   description  => 'STS to store SQL from the private SQL area'
);
END;
```

3. Optionally, confirm that the STS was created.

   The following example queries the status of all SQL tuning sets owned by the current user:

```
COLUMN NAME FORMAT a20
COLUMN COUNT FORMAT 99999
COLUMN DESCRIPTION FORMAT a11
```

```
SELECT NAME, STATEMENT_COUNT AS "SQLCNT", DESCRIPTION
FROM   USER_SQLSET;
```

Sample output appears below:

```
NAME                 SQLCNT DESCRIPTION
-------------------- ------ -----------
SQLT_WKLD_STS             2 SQL Cache
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information

24.3 Loading a SQL Tuning Set Using LOAD_SQLSET

To load an STS with SQL statements, use the `LOAD_SQLSET` procedure in the `DBMS_SQLTUNE` or `DBMS_SQLSET` package.

The standard sources for populating an STS are AWR, another STS, or the shared SQL area. For both the workload repository and SQL tuning sets, predefined table functions can select columns from the source to populate a new STS.

The following table describes some `DBMS_SQLSET.LOAD_SQLSET` procedure parameters.

Table 24-3 DBMS_SQLSET.LOAD_SQLSET Parameters

Parameter	Description
`populate_cursor`	Specifies the cursor reference from which to populate the STS.
`load_option`	Specifies how the statements are loaded into the STS. The possible values are `INSERT` (default), `UPDATE`, and `MERGE`.

The `DBMS_SQLSET.SELECT_CURSOR_CACHE` function collects SQL statements from the shared SQL area according to the specified filter. This function returns one `SQLSET_ROW` per SQL ID or `PLAN_HASH_VALUE` pair found in each data source.

Use the `DBMS_SQLSET.CAPTURE_CURSOR_CACHE_SQLSET` function (or the equivalent `DBMS_SQLSET.CAPTURE_CURSOR_CACHE`) to repeatedly poll the shared SQL area over a specified interval. This function is more efficient than repeatedly calling the `SELECT_CURSOR_CACHE` and `LOAD_SQLSET` procedures. This function effectively captures the entire workload, as opposed to the AWR, which only captures the workload of high-load SQL statements, or the `LOAD_SQLSET` procedure, which accesses the data source only once.

Prerequisites

This tutorial has the following prerequisites:

- Filters provided to the `SELECT_CURSOR_CACHE` function are evaluated as part of SQL statements run by the current user. As such, they are executed with that user's security privileges and can contain any constructs and subqueries that user can access, but no more.

- The current user must have privileges on the shared SQL area views.

Assumptions

This tutorial assumes the following:

- You want to load the SQL tuning set named `SQLT_WKLD_STS` with statements from the shared SQL area.
- You want to use `DBMS_SQLSET` rather than `DBMS_SQLTUNE` to load the STS.

To load an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Run the `DBMS_SQLSET.LOAD_SQLSET` procedure.

   For example, execute the following PL/SQL program to populate a SQL tuning set with all cursor cache statements that belong to the `sh` schema:

```
DECLARE
  c_sqlarea_cursor DBMS_SQLSET.SQLSET_CURSOR;
BEGIN
 OPEN c_sqlarea_cursor FOR
   SELECT VALUE(p)
   FROM   TABLE(
            DBMS_SQLSET.SELECT_CURSOR_CACHE(
            ' module = ''SQLT_WKLD'' AND parsing_schema_name =
''SH'' ')
          ) p;
-- load the tuning set
  DBMS_SQLSET.LOAD_SQLSET (
    sqlset_name     => 'SQLT_WKLD_STS'
,   populate_cursor =>  c_sqlarea_cursor
);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information.

24.4 Querying a SQL Tuning Set

To read the contents of an STS after it has been created and populated, use the `SELECT_SQLSET` function of `DBMS_SQLTUNE` or `DBMS_SQLSET`, optionally using filtering criteria.

Select the output of `SELECT_SQLSET` using a PL/SQL pipelined table function, which accepts a collection of rows as input. You invoke the table function as the operand of the table operator in the `FROM` list of a `SELECT` statement. The following table describes some `SELECT_SQLSET` function parameters.

Table 24-4 DBMS_SQLTUNE.SELECT_SQLSET Parameters

Parameter	Description
`basic_filter`	The SQL predicate to filter the SQL from the STS defined on attributes of the `SQLSET_ROW`
`object_filter`	Specifies the objects that exist in the object list of selected SQL from the shared SQL area

The following table describes some attributes of the `SQLSET_ROW` object. These attributes appears as columns when you query `TABLE(DBMS_SQLTUNE.SELECT_SQLSET())`.

Table 24-5 SQLSET_ROW Attributes

Parameter	Description
`parsing_schema_name`	Schema in which the SQL is parsed
`elapsed_time`	Sum of the total number of seconds elapsed for this SQL statement
`buffer_gets`	Total number of buffer gets (number of times the database accessed a block) for this SQL statement

Assumptions

This tutorial assumes the following:

- You want to display the contents of an STS named `SQLT_WKLD_STS`.
- You are using `DBMS_SQLTUNE` instead of `DBMS_SQLSET`.

To display the contents of an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Query the STS contents using the `TABLE` function.

   For example, execute the following query:

```
COLUMN SQL_TEXT FORMAT a30
COLUMN SCH FORMAT a3
COLUMN ELAPSED FORMAT 999999999

SELECT SQL_ID, PARSING_SCHEMA_NAME AS "SCH", SQL_TEXT,
       ELAPSED_TIME AS "ELAPSED", BUFFER_GETS
FROM   TABLE( DBMS_SQLTUNE.SELECT_SQLSET( 'SQLT_WKLD_STS' ) );
```

Sample output appears below:

```
SQL_ID        SCH SQL_TEXT                          ELAPSED
BUFFER_GETS
------------- --- ------------------------------ ----------
-----------
79f8shn041a1f SH  select * from sales where quan    8373148
24016
                  tity_sold < 5 union select * f
                  rom sales where quantity_sold
                  > 500

2cqsw036j5u7r SH  select promo_name, count(*) c     3557373
309
                  from promotions p, sales s whe
                  re s.promo_id = p.promo_id and
                   p.promo_category = 'internet'
                   group by p.promo_name order b
                  y c desc

fudq5z56g642p SH  select sum(quantity_sold) from    4787891
12118
                   sales s, products p where s.p
                  rod_id = p.prod_id and s.amoun
                  t_sold > 20000 and p.prod_name
                   = 'Linen Big Shirt'

bzmnj0nbvmz8t SH  select * from sales where amou     442355
15281
                  nt_sold = 4
```

3. Optionally, filter the results based on user-specific criteria.

   The following example displays statements with a disk reads to buffer gets ratio greater than or equal to 50%:

```
COLUMN SQL_TEXT FORMAT a30
COLUMN SCH FORMAT a3
COLUMN BUF_GETS FORMAT 99999999
COLUMN DISK_READS FORMAT 99999999
COLUMN %_DISK FORMAT 9999.99
SELECT sql_id, parsing_schema_name as "SCH", sql_text,
       buffer_gets as "B_GETS",
       disk_reads as "DR", ROUND(disk_reads/buffer_gets*100,2)
"%_DISK"
FROM TABLE( DBMS_SQLTUNE.SELECT_SQLSET(
            'SQLT_WKLD_STS',
            '(disk_reads/buffer_gets) >= 0.50' ) );
```

Sample output appears below:

```
SQL_ID        SCH SQL_TEXT                       B_GETS DR      %_DISK
------------- --- ------------------------------ ------ ------- --------
79f8shn041a1f SH  select * from sales where quan  24016   17287  71.98
                  tity_sold < 5 union select * f
                  rom sales where quantity_sold
                  > 500

fudq5z56g642p SH  select sum(quantity_sold) from  12118    6355  52.44
                   sales s, products p where s.p
                  rod_id = p.prod_id and s.amoun
                  t_sold > 20000 and p.prod_name
                   = 'Linen Big Shirt'
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information

24.5 Modifying a SQL Tuning Set Using UPDATE_SQLSET

Use the `UPDATE_SQLSET` procedure in `DBMS_SQLTUNE` or `DBMS_SQLSET` to delete SQL statements from an STS.

You can use the `UPDATE_SQLSET` procedure to update the attributes of SQL statements (such as `PRIORITY` or `OTHER`) in an existing STS identified by STS name and SQL ID.

Assumptions

This tutorial assumes that you want to modify `SQLT_WKLD_STS` as follows:

- You want to delete all SQL statements with fetch counts over 100.
- You want to change the priority of the SQL statement with ID `fudq5z56g642p` to `1`. You can use priority as a ranking criteria when running SQL Tuning Advisor.
- You use `DBMS_SQLSET` instead of `DBMS_SQLTUNE`.

To modify the contents of an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Optionally, query the STS contents using the `TABLE` function.

   For example, execute the following query:

   ```
   SELECT SQL_ID, ELAPSED_TIME, FETCHES, EXECUTIONS
   FROM   TABLE(DBMS_SQLSET.SELECT_SQLSET('SQLT_WKLD_STS'));
   ```

Sample output appears below:

```
SQL_ID         ELAPSED_TIME    FETCHES EXECUTIONS
------------- ------------ ---------- ----------
2cqsw036j5u7r      3407459          2          1
79f8shn041a1f      9453965      61258          1
bzmnj0nbvmz8t       401869          1          1
fudq5z56g642p      5300264          1          1
```

3. Delete SQL statements based on user-specified criteria.

   Use the `basic_filter` predicate to filter the SQL from the STS defined on attributes of the `SQLSET_ROW`. The following example deletes all statements in the STS with fetch counts over 100:

```
BEGIN
  DBMS_SQLSET.DELETE_SQLSET (
      sqlset_name  => 'SQLT_WKLD_STS'
,     basic_filter => 'fetches > 100'
);
END;
/
```

4. Set attribute values for SQL statements.

   The following example sets the priority of statement `2cqsw036j5u7r` to `1`:

```
BEGIN
  DBMS_SQLSET.UPDATE_SQLSET (
      sqlset_name     => 'SQLT_WKLD_STS'
,     sql_id          => '2cqsw036j5u7r'
,     attribute_name  => 'PRIORITY'
,     attribute_value =>  1
);
END;
/
```

5. Optionally, query the STS to confirm that the intended modifications were made.

   For example, execute the following query:

```
SELECT SQL_ID, ELAPSED_TIME, FETCHES, EXECUTIONS, PRIORITY
FROM   TABLE(DBMS_SQLSET.SELECT_SQLSET('SQLT_WKLD_STS'));
```

   Sample output appears below:

```
SQL_ID         ELAPSED_TIME    FETCHES EXECUTIONS   PRIORITY
------------- ------------ ---------- ---------- ----------
2cqsw036j5u7r      3407459          2          1          1
bzmnj0nbvmz8t       401869          1          1
fudq5z56g642p      5300264          1          1
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for more information

24.6 Transporting a SQL Tuning Set

You can transport an STS to any database created in Oracle Database 10*g* Release 2 (10.2) or later. This technique is useful when using SQL Performance Analyzer to tune regressions on a test database.

24.6.1 About Transporting SQL Tuning Sets

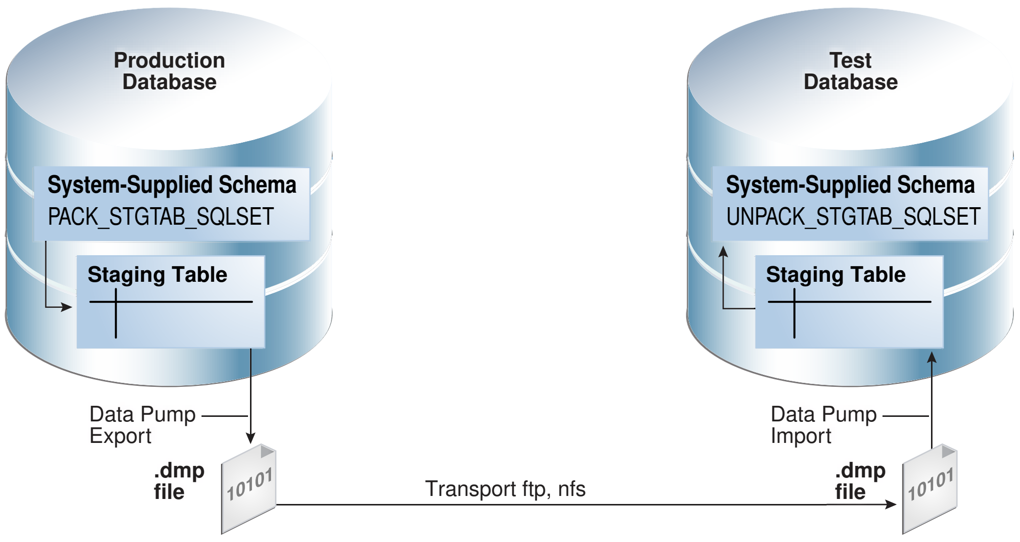
Transporting SQL tuning sets between databases means copying the SQL tuning sets to and from a staging table, and then using other tools to move the staging table to the destination database. The most common tools are Oracle Data Pump or a database link.

24.6.1.1 Basic Steps for Transporting SQL Tuning Sets

Transporting SQL tuning sets requires exporting the STS, transporting the dump file, and then importing the dump file.

The following graphic shows the process using Oracle Data Pump and `ftp`.

Figure 24-4 Transporting SQL Tuning Sets



As shown in Figure 24-4, the steps are as follows:

1. In the production database, pack the STS into a staging table using `DBMS_SQLTUNE.PACK_STGTAB_SQLSET` or `DBMS_SQLSET.PACK_STGTAB`.
2. Export the STS from the staging table to a `.dmp` file using Oracle Data Pump.

3. Transfer the `.dmp` file from the production host to the test host using a transfer tool such as `ftp`.
4. In the test database, import the STS from the `.dmp` file to a staging table using Oracle Data Pump.
5. Unpack the STS from the staging table using `DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET` or `DBMS_SQLSET.UNPACK_STGTAB`.

24.6.1.2 Basic Steps for Transporting SQL Tuning Sets When the CON_DBID Values Differ

When transporting an STS, you must remap the `con_dbid` of each SQL statement in the STS when the `con_dbid` of the source database and the destination database are different.

Situations that cause the `con_dbid` value to differ include the following:

- A single-instance database whose instance has been restarted
- Different instances of an Oracle RAC database
- Different PDBs

The basic steps for remapping are as follows:

1. Pack the STS into a staging table using `DBMS_SQLTUNE.PACK_STGTAB_SQLSET` or `DBMS_SQLSET.PACK_STGTAB`.
2. Remap each `con_dbid` in the staging table using `DBMS_SQLTUNE.REMAP_STGTAB_SQLSET` or `DBMS_SQLSET.REMAP_STGTAB`.
3. Export the STS.
4. Unpack the STS in the destination CDB.

Example 24-1 Remapping a CON_DBID When Transporting an STS from one PDB to Another

In this example, you intend to transport an STS named `STS_for_transport` from one PDB to a different PDB. On the source PDB, you have already packed the STS into source staging table `src_stg_tbl` using the `DBMS_SQLTUNE.PACK_STGTAB_SQLSET` procedure. The container ID of the destination PDB is `12345`.

In the source PDB, you execute the following commands:

```
VARIABLE con_dbid_src NUMBER;

EXEC SELECT UNIQUE con_dbid INTO :con_dbid_src FROM src_stg_tbl;

BEGIN
  DBMS_SQLTUNE.REMAP_STGTAB_SQLSET (
    staging_table_name   => 'src_stg_tbl'
,   staging_schema_owner => 'dba1'
,   old_sqlset_name      => 'STS_for_transport'
,   old_con_dbid         => :con_dbid_src
,   new_con_dbid         => 12345);
END;
```

You can now export the contents of the staging table, and then continue using the normal transport procedure.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about `REMAP_STGTAB_SQLSET`

24.6.2 Transporting SQL Tuning Sets with DBMS_SQLTUNE

You can transport SQL tuning sets using three subprograms in the `DBMS_SQLTUNE` or `DBMS_SQLSET` package.

The following table describes the procedures relevant for transporting SQL tuning sets.

Table 24-6 Procedures for Transporting SQL Tuning Sets

DBMS_SQLTUNE Procedure	Equivalent DBMS_SQLSET Procedure	Description
`CREATE_STGTAB_SQLSET`	`CREATE_STGTAB`	Create a staging table to hold the exported SQL tuning sets
`PACK_STGTAB_SQLSET`	`PACK_STGTAB`	Populate a staging table with SQL tuning sets
`UNPACK_STGTAB_SQLSET`	`UNPACK_STGTAB`	Copy the SQL tuning sets from the staging table into a database

Assumptions

This tutorial assumes the following:

- An STS with regressed SQL resides in a production database created in the current release.
- You run SQL Performance Analyzer trials on a remote test database created in Oracle Database 11g Release 2 (11.2).
- You want to copy the STS from the production database to the test database and tune the regressions from the SQL Performance Analyzer trials.
- You want to use Oracle Database Pump to transfer the SQL tuning sets between database hosts.
- You use `DBMS_SQLTUNE` rather than `DBMS_SQLSET`.

To transport an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with administrative privileges.
2. Use the `CREATE_STGTAB_SQLSET` procedure to create a staging table to hold the exported SQL tuning sets.

The following example creates `my_11g_staging_table` in the `dba1` schema and specifies the format of the staging table as 11.2:

```
BEGIN
  DBMS_SQLTUNE.CREATE_STGTAB_SQLSET (
    table_name  => 'my_10g_staging_table'
,   schema_name => 'dba1'
,   db_version  => DBMS_SQLTUNE.STS_STGTAB_11_2_VERSION
);
END;
/
```

3. Use the `PACK_STGTAB_SQLSET` procedure to populate the staging table with SQL tuning sets.

   The following example populates `dba1.my_11g_staging_table` with the STS `my_sts` owned by `hr`:

```
BEGIN
  DBMS_SQLTUNE.PACK_STGTAB_SQLSET (
    sqlset_name          => 'sqlt_wkld_sts'
,   sqlset_owner         => 'sh'
,   staging_table_name   => 'my_11g_staging_table'
,   staging_schema_owner => 'dba1'
,   db_version           => DBMS_SQLTUNE.STS_STGTAB_11_2_VERSION
);
END;
/
```

4. If necessary, remap the container ID values for the statements in the STS as described in "Basic Steps for Transporting SQL Tuning Sets When the CON_DBID Values Differ".
5. Use Oracle Data Pump to export the contents of the staging table.

   For example, run the `expdp` command at the operating system prompt:

```
expdp dba1 DIRECTORY=dpump_dir1 DUMPFILE=sts.dmp
TABLES=my_11g_staging_table
```

6. Transfer the dump file to the test database host.
7. Log in to the test host as an administrator, and then use Oracle Data Pump to import the contents of the staging table.

   For example, run the `impdp` command at the operating system prompt:

```
impdp dba1 DIRECTORY=dpump_dir1 DUMPFILE=sts.dmp
TABLES=my_11g_staging_table
```

8. On the test database, execute the `UNPACK_STGTAB_SQLSET` procedure to copy the SQL tuning sets from the staging table into the database.

   The following example shows how to unpack the SQL tuning sets:

```
BEGIN
  DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET (
    sqlset_name        => '%'
,   replace            => true
,   staging_table_name => 'my_11g_staging_table');
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `DBMS_SQLTUNE.UNPACK_STGTAB_SQLSET`

24.7 Dropping a SQL Tuning Set Using DROP_SQLSET

To drop an STS from the database, use the `DROP_SQLSET` procedure in the `DBMS_SQLTUNE` or `DBMS_SQLSET` package.

Prerequisites

Ensure that no tuning task is currently using the STS to be dropped. If an existing tuning task is using this STS, then drop the task before dropping the STS. Otherwise, the database issues an `ORA-13757` error.

Assumptions

This tutorial assumes the following:

- You want to drop an STS named `SQLT_WKLD_STS`.
- You use `DBMS_SQLSET` instead of `DBMS_SQLTUNE`.

To drop an STS:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use the `DBMS_SQLSET.DROP_SQLSET` procedure.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_SQLSET.DROP_SQLSET( sqlset_name => 'SQLT_WKLD_STS' );
   END;
   /
   ```

3. Optionally, confirm that the STS was deleted.

   The following example counts the number of SQL tuning sets named `SQLT_WKLD_STS` owned by the current user (sample output included):

   ```
   SELECT  COUNT(*)
   FROM    USER_SQLSET
   WHERE   NAME = 'SQLT_WKLD_STS';

     COUNT(*)
   ```

```
----------
         0
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the STS procedures in `DBMS_SQLSET`

25
Analyzing SQL with SQL Tuning Advisor

Use SQL Tuning Advisor to obtain recommendations for improving performance of high-load SQL statements, and prevent regressions by only executing optimal plans.

25.1 About SQL Tuning Advisor

SQL Tuning Advisor is SQL diagnostic software in the Oracle Database Tuning Pack.

You can submit one or more SQL statements as input to the advisor and receive advice or recommendations for how to tune the statements, along with a rationale and expected benefit.

25.1.1 Purpose of SQL Tuning Advisor

SQL Tuning Advisor is a mechanism for resolving problems related to suboptimally performing SQL statements.

Use SQL Tuning Advisor to obtain recommendations for improving performance of high-load SQL statements, and prevent regressions by only executing optimal plans.

Tuning recommendations include:

- Collection of object statistics
- Creation of indexes
- Rewriting SQL statements
- Creation of SQL profiles
- Creation of SQL plan baselines

The recommendations generated by SQL Tuning Advisor help you achieve the following specific goals:

- Avoid labor-intensive manual tuning

  Identifying and tuning high-load SQL statements is challenging even for an expert. SQL Tuning Advisor uses the optimizer to tune SQL for you.
- Generate recommendations and implement SQL profiles automatically

  You can configure an Automatic SQL Tuning task to run nightly in maintenance windows. When invoked in this way, the advisor can generate recommendations and also implement SQL profiles automatically.
- Analyze database-generated statistics to achieve optimal plans

  The database contains a vast amount of statistics about its own operations. SQL Tuning Advisor can perform deep mining and analysis of internal information to improve execution plans.
- Enable developers to tune SQL on a test system instead of the production system

When suboptimally performing SQL statements occur on a production database, developers may not want to investigate and tune directly on the production database. The DBA can transport the problematic SQL statements to a test database where the developers can safely analyze and tune them.

When tuning multiple statements, SQL Tuning Advisor does not recognize interdependencies between the statements. Instead, SQL Tuning Advisor offers a convenient way to get tuning recommendations for many statements.

Note:

A common user whose current container is the CDB root can run SQL Tuning Advisor manually for SQL statements from any PDB. When a statement is tuned, it is tuned in any container that runs the statement.

A user whose current container is a PDB can run SQL Tuning Advisor manually for SQL statements that have been run in this PDB. In this case, a statement is tuned only for the current PDB. The results related to a PDB are stored in the PDB and are included if the PDB is unplugged. When SQL Tuning Advisor is run manually by a user whose current container is a PDB, the results are only visible to a user whose current container is that PDB.

See Also:

"Managing SQL Plan Baselines" to learn about SQL plan management

25.1.2 SQL Tuning Advisor Architecture

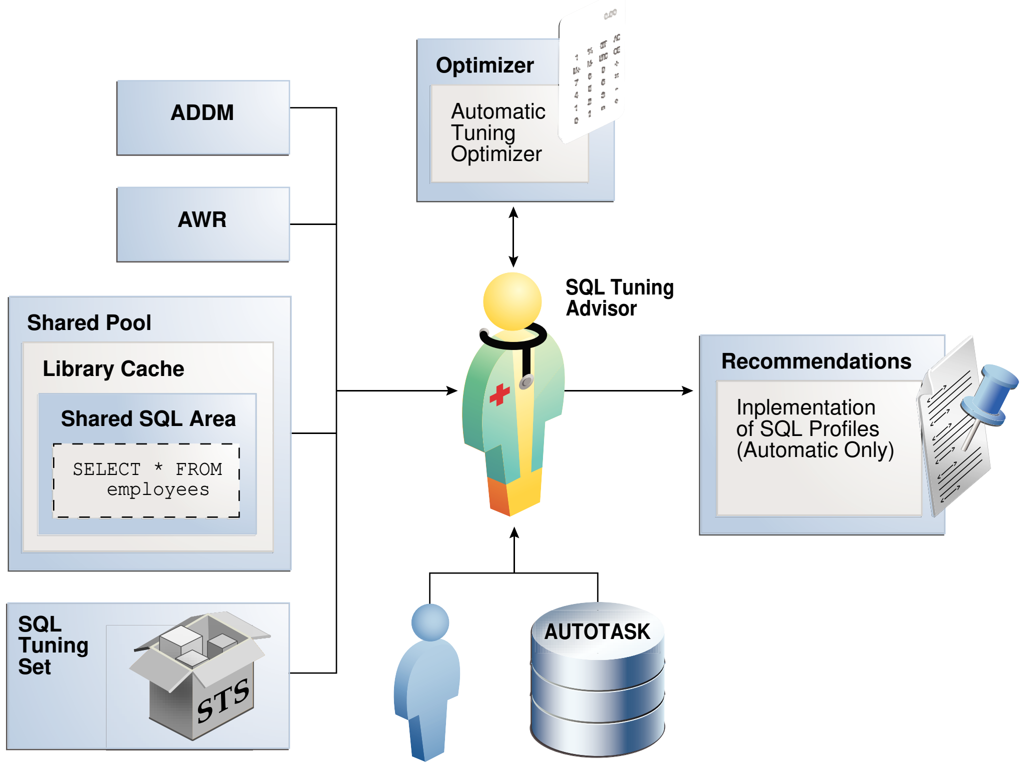
Automatic Tuning Optimizer is the central tool used by SQL Tuning Advisor. The advisor can receive SQL statements as input from multiple sources, analyze these statements using the optimizer, and then make recommendations.

Invoking Automatic Tuning Optimizer for every hard parse consumes significant time and resources. Tuning mode is meant for complex and high-load SQL statements that significantly affect database performance.

Manageability advisors such as SQL tuning advisor use a common infrastructure called the advisor framework. This framework provides a common schema and interface for storing task objects. An advisor schema is a set of tables to store the data from advisors. SQL Tuning Advisor receives tuning input, and then writes to the advisor schemas by means of the advisor framework. SQL Tuning Advisor reads data from advisor schema when it produces its reports.

The following figure shows the basic architecture of SQL Tuning Advisor.

Figure 25-1 SQL Tuning Advisor Architecture



See Also:

"SQL Parsing"

25.1.2.1 Input to SQL Tuning Advisor

Input for SQL Tuning Advisor can come from several sources, including ADDM, AWR, the shared SQL area, and SQL tuning sets.

SQL Tuning Advisor uses its input sources as follows:

- Automatic Database Diagnostic Monitor (ADDM)

  The primary input source for SQL Tuning Advisor is ADDM (pronounced *Adam*). By default, ADDM runs proactively once every hour. To identify performance problems involving high-load SQL statements, ADDM analyzes key statistics gathered by Automatic Workload Repository (AWR) over the last hour . If a high-load SQL statement is identified, then ADDM recommends running SQL Tuning Advisor on the SQL.

- AWR

  AWR takes regular snapshots of system activity, including high-load SQL statements ranked by relevant statistics, such as CPU consumption and wait time.

  You can view the AWR and manually identify high-load SQL statements. You can run SQL Tuning Advisor on these statements, although Oracle Database automatically performs this work as part of automatic SQL tuning. By default, AWR retains data for the last eight days. You can locate and tune any high-load SQL that ran within the retention period of AWR using this technique.

- Shared SQL area

  The database uses the shared SQL area to tune recent SQL statements that have yet to be captured in AWR. The shared SQL area and AWR provide the capability to identify and tune high-load SQL statements from the current time going as far back as the AWR retention allows, which by default is at least 8 days.

- SQL tuning set

  A SQL tuning set (STS) is a database object that stores SQL statements along with their execution context. An STS can include SQL statements that are yet to be deployed, with the goal of measuring their individual performance, or identifying the ones whose performance falls short of expectation. When a set of SQL statements serve as input, the database must first construct and use an STS.

See Also:

- "About SQL Tuning Sets"
- *Oracle Database Performance Tuning Guide* to learn about ADDM
- *Oracle Database Performance Tuning Guide* to learn about AWR
- *Oracle Database Concepts* to learn about the shared SQL area

25.1.2.2 Output of SQL Tuning Advisor

After analyzing the SQL statements, SQL Tuning Advisor publishes recommendations.

Specifically, SQL Tuning Advisor produces the following types of output:

- Advice on optimizing the execution plan
- Rationale for the proposed optimization
- Estimated performance benefit
- SQL statement to implement the advice

The benefit percentage shown for each recommendation is calculated using the following formula:

```
abnf% = (time_old - time_new)/(time_old)
```

For example, assume that before tuning the execution time was 100 seconds, and after implementing the recommendation the new execution time is expected to be 33 seconds. This benefit calculation for this performance improvement is as follows:

```
67% = (100 - 33)/(100)
```

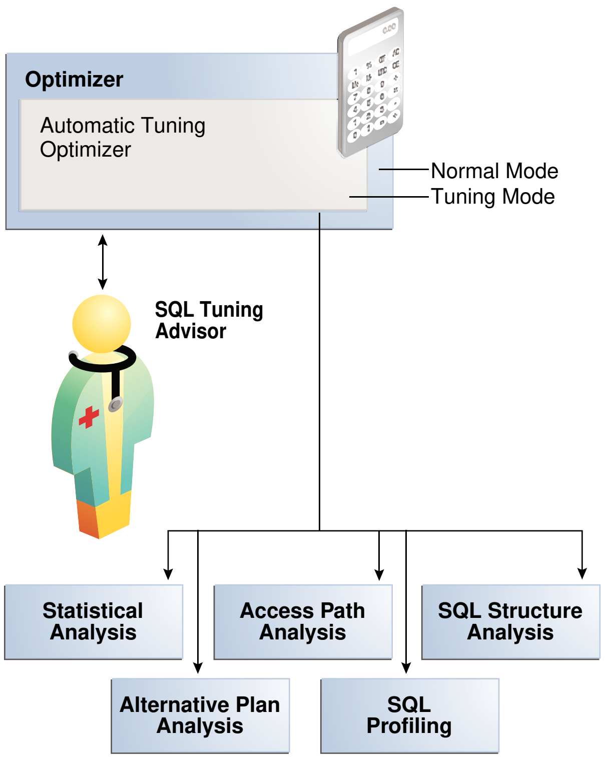
You choose whether to accept the recommendations to optimize the SQL statements. Depending on how it is configured, Automatic SQL Tuning Advisor can implement the SQL profile recommendations to tune the statement *without* user intervention. When invoked on demand, SQL Tuning Advisor can recommend that the user implement a SQL profile, but can never implement it automatically.

25.1.2.3 Automatic Tuning Optimizer Analyses

In tuning mode, the optimizer has more time to consider options and gather statistics. For example, Automatic Tuning Optimizer can use dynamic statistics and partial statement execution.

The following graphic depicts the different types of analysis that Automatic Tuning Optimizer performs.

Figure 25-2 Automatic Tuning Optimizer



See Also:

"Query Optimizer Concepts "

25.1.2.3.1 Statistical Analysis

The optimizer relies on statistics to generate execution plans.

If these statistics are stale or missing, then the optimizer can generate suboptimal plans. Automatic Tuning Optimizer checks for missing or stale statistics, and recommends gathering fresh statistics if needed.

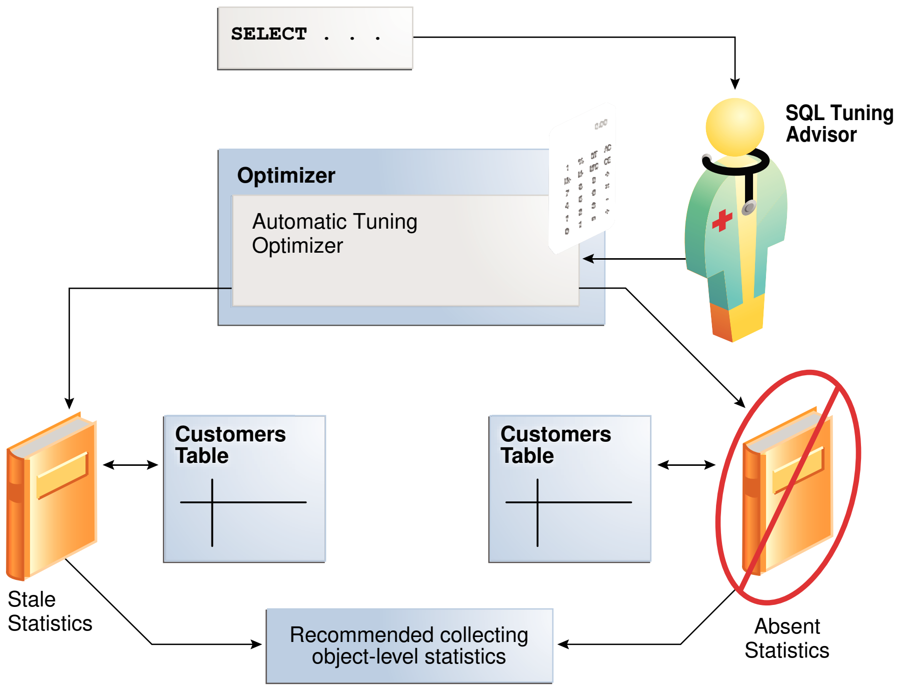
- Object statistics

  The optimizer checks the statistics for each object referenced in the query.
- System statistics

  On Oracle Exadata Database Machine, the cost of smart scans depends on the system statistics I/O seek time, multiblock read count, and I/O transfer speed. The values of these system statistics are usually different on Oracle Exadata Database Machine, so an analysis to determines whether these system statistics are not up to date. If gathering these statistics would improve the plan, then SQL Tuning Advisor recommends accepting a SQL profile.

The following graphic depicts the analysis of object-level statistics.

Figure 25-3 Statistical Analysis by Automatic Tuning Optimizer



See Also:

Oracle Exadata Database Machine System Overview

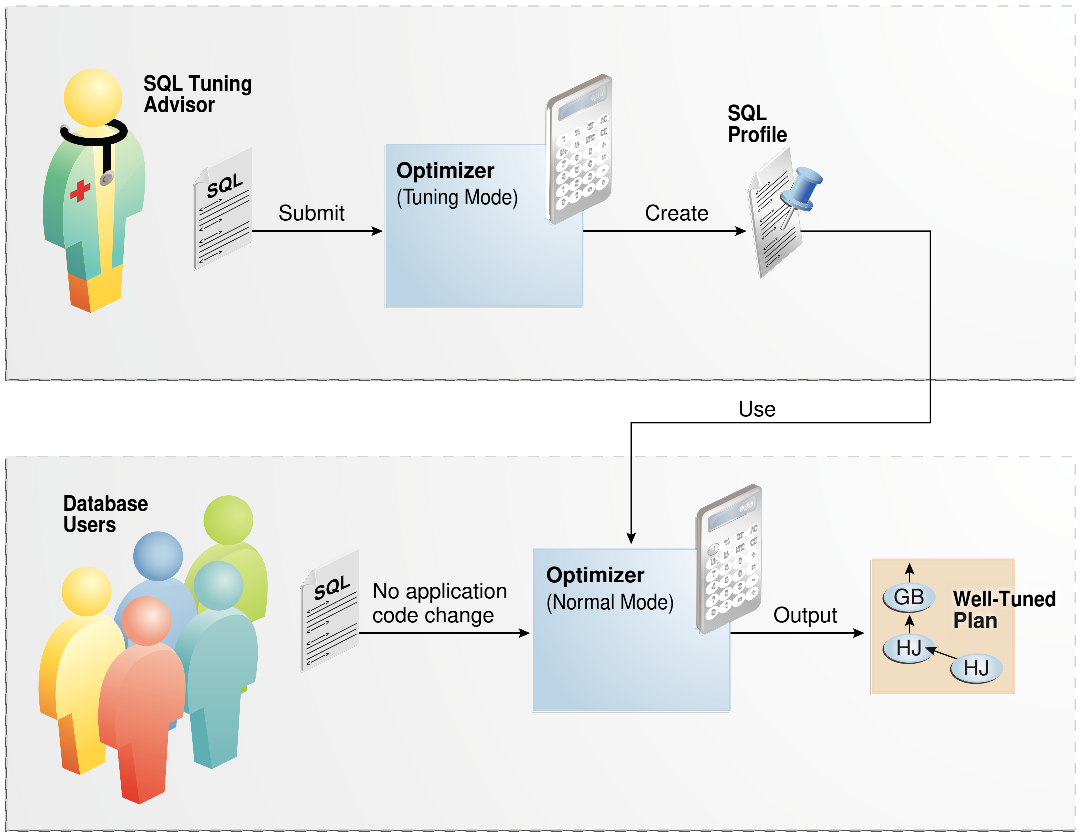
25.1.2.3.2 SQL Profiling

SQL profiling is the verification by the Automatic Tuning Optimizer of its own estimates.

By reviewing execution history and testing the SQL, the optimizer can ensure that it has the most accurate information available to generate execution plans. SQL profiling is related to but distinct from the steps of generating SQL Tuning Advisor recommendations and implementing these recommendations.

The following graphic shows SQL Tuning Advisor recommending a SQL profile and automatically implementing it. After creating the profile, the optimizer can use it as additional input when generating execution plans.

Figure 25-4 SQL Profile



See Also:

"About SQL Profiles"

25.1.2.3.2.1 How SQL Profiling Works

The database can profile some DML and DDL statements.

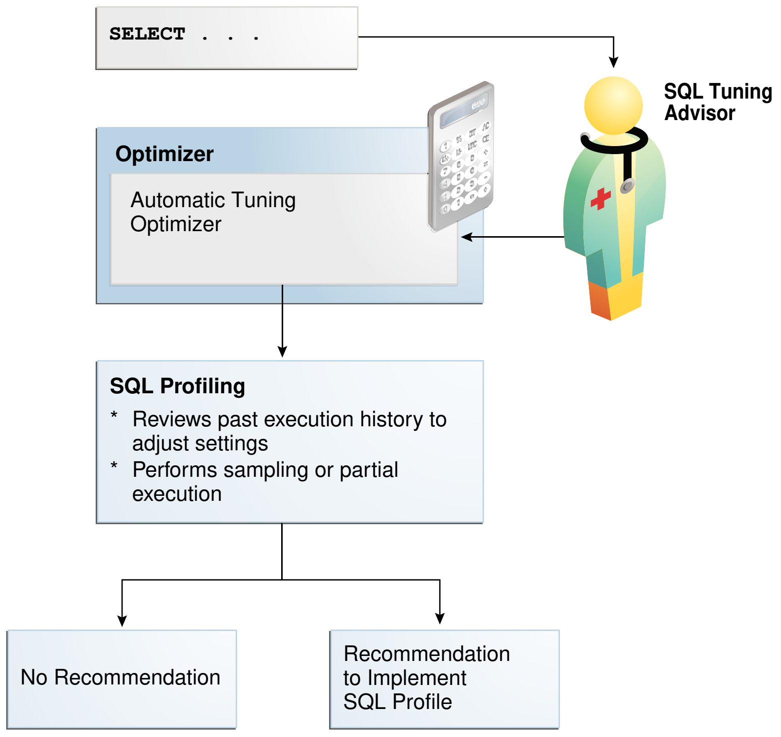
Specifically, SQL Tuning Advisor can profile the following types of statement:

- DML statements (`SELECT`, `INSERT` with a `SELECT` clause, `UPDATE`, `DELETE`, and the update or insert operations of `MERGE`)
- `CREATE TABLE` statements (only with the `AS SELECT` clause)

After performing its analysis, SQL Tuning Advisor either recommends or does not recommend implementing a SQL profile.

The following graphic shows the SQL profiling process.

Figure 25-5 SQL Profiling


SELECT . . .
SQL Tuning Advisor
Optimizer
Automatic Tuning Optimizer
SQL Profiling
* Reviews past execution history to adjust settings
* Performs sampling or partial execution
No Recommendation
Recommendation to Implement SQL Profile

During SQL profiling, the optimizer verifies cost, selectivity, and cardinality for a statement. The optimizer uses either of the following methods:

- Samples the data and applies appropriate predicates to the sample

  The optimizer compares the new estimate to the regular estimate and, if the difference is great enough, applies a correction factor.

- Executes a fragment of the SQL statement

  This method is more efficient than the sampling method when the predicates provide efficient access paths.

The optimizer uses the past statement execution history to determine correct settings. For example, if the history indicates that a SQL statement is usually executed only partially, then the optimizer uses `FIRST_ROWS` instead of `ALL_ROWS` optimization.

See Also:

"Choosing an Optimizer Goal"

25.1.2.3.2.2 SQL Profile Implementation

If the optimizer generates auxiliary information during statistical analysis or SQL profiling, then the optimizer recommends implementing a SQL profile.

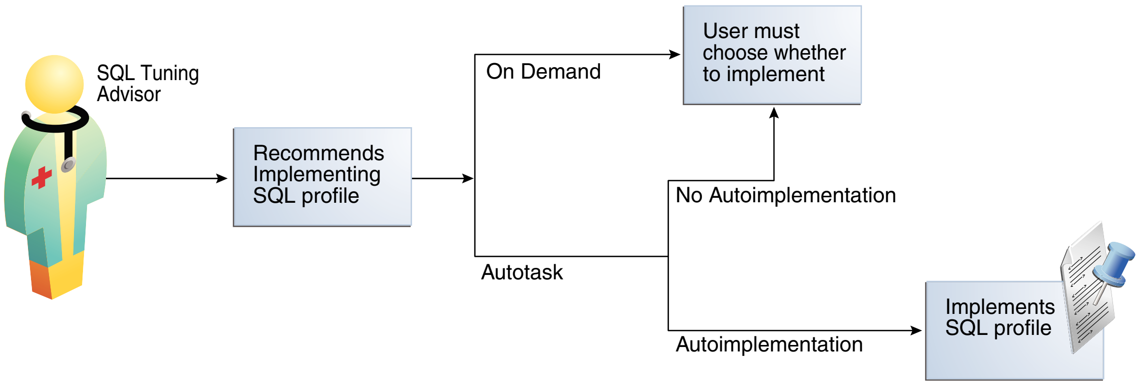
As shown in Figure 25-6, the following options are possible:

- When SQL Tuning Advisor is run on demand, the user must choose whether to implement the SQL profile.
- When the Automatic SQL Tuning task is configured to implement SQL profiles automatically, advisor behavior depends on the setting of the `ACCEPT_SQL_PROFILE` tuning task parameter:
  – If set to `true`, then the advisor implements SQL profiles automatically.
  – If set to `false`, then user intervention is required.
  – If set to `AUTO` (default), then the setting is `true` when at least one SQL statement exists with a SQL profile, and `false` when this condition is not satisfied.

Note:

The Automatic SQL Tuning task cannot automatically create SQL plan baselines or add plans to them.

Figure 25-6 Implementing SQL Profiles



At any time during or after automatic SQL tuning, you can view a report. This report describes in detail the SQL statements that were analyzed, the recommendations generated, and any SQL profiles that were automatically implemented.

See Also:

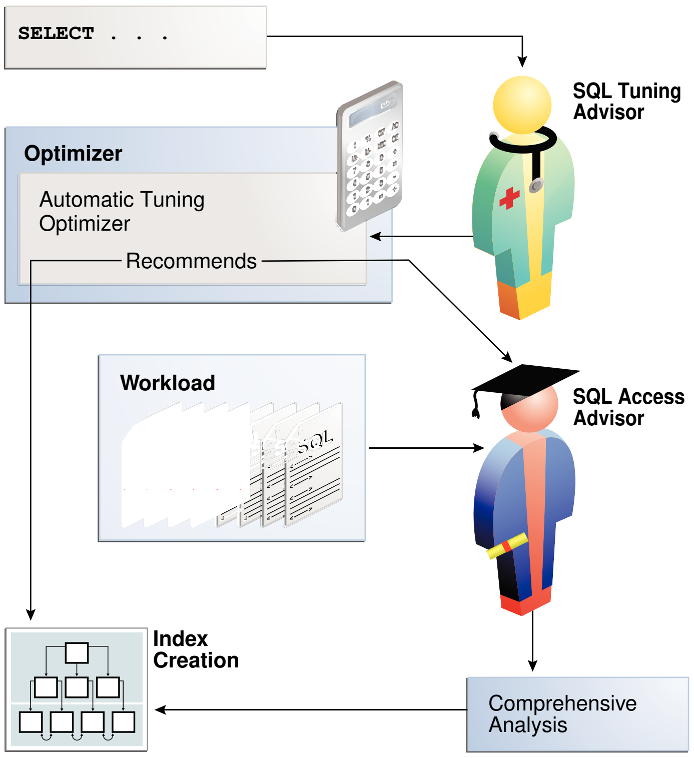
- "Configuring the Automatic SQL Tuning Task Using the Command Line"
- "Plan Evolution"
- "About SQL Profiles"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `ACCEPT_SQL_PROFILE`

25.1.2.3.3 Access Path Analysis

An **access path** is the means by which the database retrieves data.

For example, a query using an index and a query using a full table scan use different access paths. In some cases, indexes can greatly enhance the performance of a SQL statement by eliminating full table scans. The following graphic illustrates access path analysis.

Figure 25-7 Access Path Analysis



Automatic Tuning Optimizer explores whether a new index can significantly enhance query performance and recommends either of the following:

- Creating an index

  Index recommendations are specific to the SQL statement processed by SQL Tuning Advisor. Sometimes a new index provides a quick solution to the performance problem associated with a single SQL statement.

- Running SQL Access Advisor

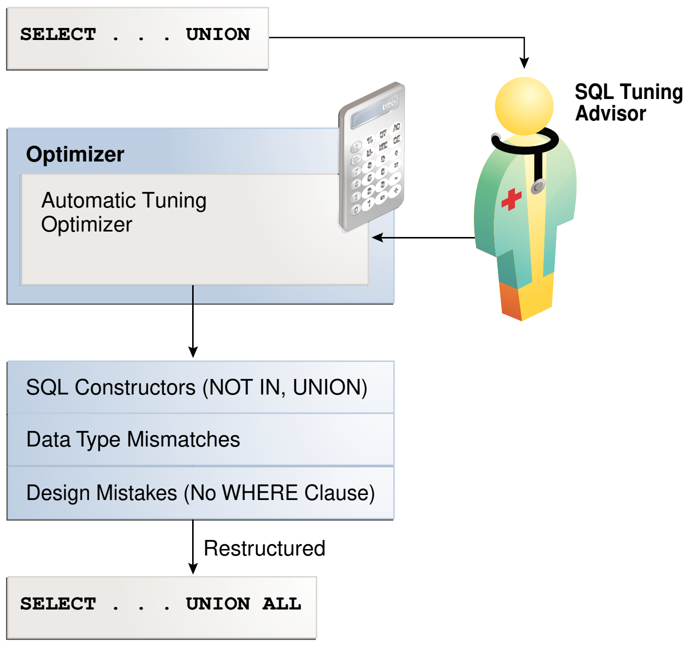
  Because the Automatic Tuning Optimizer does not analyze how its index recommendation can affect the entire SQL workload, it also recommends running SQL Access Advisor on the SQL statement along with a representative SQL workload. SQL Access Advisor examines the effect of creating an index on the SQL workload before making recommendations.

25.1.2.3.4 SQL Structural Analysis

During structural analysis, Automatic Tuning Optimizer tries to identify syntactic, semantic, or design problems that can lead to suboptimal performance. The goal is to identify poorly written SQL statements and to advise you how to restructure them.

The following graphic illustrates structural analysis.

Figure 25-8 Structural Analysis



Some syntax variations negatively affect performance. In structural analysis, the automatic tuning optimizer evaluates statements against a set of rules, identifies inefficient coding techniques, and recommends an alternative statement if possible.

As shown in Figure 25-8, Automatic Tuning Optimizer identifies the following categories of structural problems:

- Inefficient use of SQL constructors

  A suboptimally performing statement may be using `NOT IN` instead of `NOT EXISTS`, or `UNION` instead of `UNION ALL`. The `UNION` operator, as opposed to the `UNION ALL` operator, uses a unique sort to ensure that no duplicate rows are in the result set. If you know that two queries do not return duplicates, then use `UNION ALL`.

- Data type mismatches

  If the indexed column and the compared value have a data type mismatch, then the database does not use the index because of the implicit data type conversion. Also, the database must expend additional resources converting data types, and some SQL statements may fail because data values do not convert correctly. Common mistakes include columns that contain numeric data but are never used for arithmetic operations: telephone numbers, credit card numbers, and check numbers. To avoid poor cardinality estimates, suboptimal plans, and `ORA-01722` errors, developers must ensure that bind variables are type `VARCHAR2` and not numbers.

- Design mistakes

  A classic example of a design mistake is a missing join condition. If *n* is the number of tables in a query block, then *n*-1 join conditions must exist to avoid a Cartesian product.

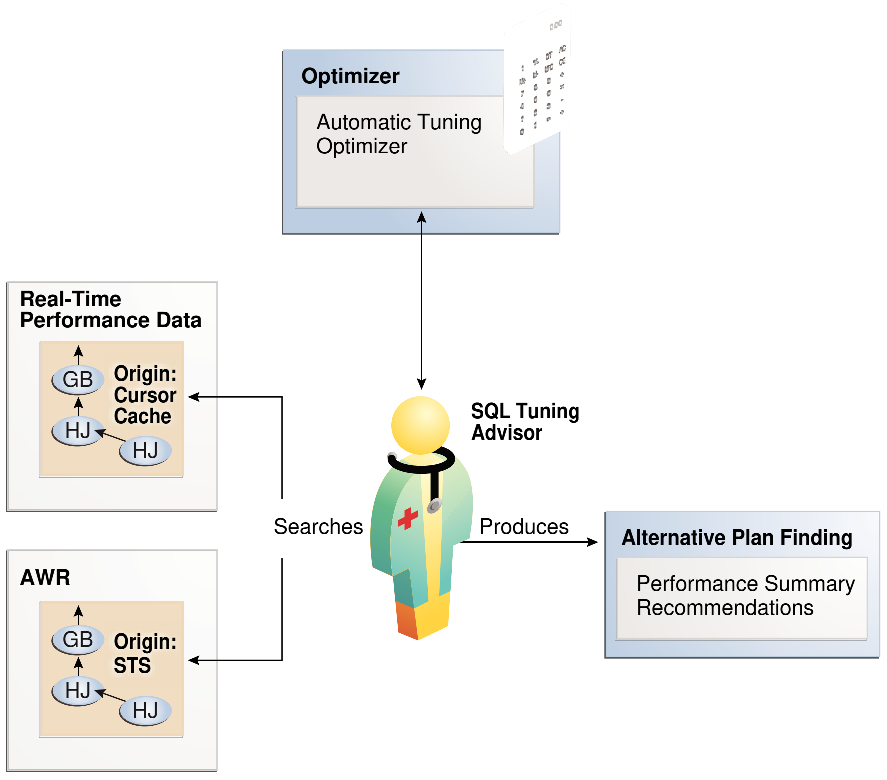
In each case, Automatic Tuning Optimizer makes relevant suggestions to restructure the statements. The suggested alternative statement is similar, but not equivalent, to the original statement. For example, the suggested statement may use `UNION ALL` instead of `UNION`. You can then determine if the advice is sound.

25.1.2.3.5 Alternative Plan Analysis

While tuning a SQL statement, SQL Tuning Advisor searches real-time and historical performance data for **alternative execution plans** for the statement.

If plans other than the original plan exist, then SQL Tuning Advisor reports an alternative plan finding. The follow graphic shows SQL Tuning Advisor finding two alternative plans and generating an alternative plan finding.

Figure 25-9 Alternative Plan Analysis



SQL Tuning Advisor validates the alternative execution plans and notes any plans that are not reproducible. When reproducible alternative plans are found, you can create a SQL plan baseline to instruct the optimizer to choose these plans in the future.

Example 25-1 Alternative Plan Finding

The following example shows an alternative plan finding for a `SELECT` statement:

```
2- Alternative Plan Finding
---------------------------
  Some alternative execution plans for this statement were found by searching
  the system's real-time and historical performance data.

  The following table lists these plans ranked by their average elapsed time.
  See section "ALTERNATIVE PLANS SECTION" for detailed information on each
  plan.
```

```
id plan hash  last seen            elapsed (s)  origin          note
-- ---------- -------------------- ------------ --------------- ----------------
 1 1378942017  2009-02-05/23:12:08        0.000 Cursor Cache    original plan
 2 2842999589  2009-02-05/23:12:08        0.002 STS

Information
-----------
- The Original Plan appears to have the best performance, based on the
  elapsed time per execution.  However, if you know that one alternative
  plan is better than the Original Plan, you can create a SQL plan baseline
  for it. This will instruct the Oracle optimizer to pick it over any other
  choices in the future.
  execute dbms_sqltune.create_sql_plan_baseline(task_name => 'TASK_XXXXX',
          object_id => 2, task_owner => 'SYS', plan_hash => xxxxxxxx);
```

The preceding example shows that SQL Tuning Advisor found two plans, one in the shared SQL area and one in a SQL tuning set. The plan in the shared SQL area is the same as the original plan.

SQL Tuning Advisor only recommends an alternative plan if the elapsed time of the original plan is worse than alternative plans. In this case, SQL Tuning Advisor recommends that users create a SQL plan baseline on the plan with the best performance. In Example 25-1, the alternative plan did not perform as well as the original plan, so SQL Tuning Advisor did not recommend using the alternative plan.

Example 25-2   Alternative Plans Section

In this example, the alternative plans section of the SQL Tuning Advisor output includes both the original and alternative plans and summarizes their performance. The most important statistic is elapsed time. The original plan used an index, whereas the alternative plan used a full table scan, increasing elapsed time by .002 seconds.

```
Plan 1
------

  Plan Origin                 :Cursor Cache
  Plan Hash Value             :1378942017
  Executions                  :50
  Elapsed Time                :0.000 sec
  CPU Time                    :0.000 sec
  Buffer Gets                 :0
  Disk Reads                  :0
  Disk Writes                 :0

Notes:
  1. Statistics shown are averaged over multiple executions.
  2. The plan matches the original plan.

--------------------------------------------
| Id  | Operation             | Name         |
--------------------------------------------
|   0 | SELECT STATEMENT      |              |
|   1 |  SORT AGGREGATE       |              |
|   2 |   MERGE JOIN          |              |
|   3 |    INDEX FULL SCAN    | TEST1_INDEX |
|   4 |     SORT JOIN         |              |
|   5 |      TABLE ACCESS FULL| TEST         |
--------------------------------------------

Plan 2
------

  Plan Origin                 :STS
  Plan Hash Value             :2842999589
  Executions                  :10
  Elapsed Time                :0.002 sec
  CPU Time                    :0.002 sec
  Buffer Gets                 :3
  Disk Reads                  :0
  Disk Writes                 :0

Notes:
  1. Statistics shown are averaged over multiple executions.

-------------------------------------
| Id  | Operation            | Name  |
-------------------------------------
|   0 | SELECT STATEMENT     |       |
|   1 |  SORT AGGREGATE      |       |
|   2 |   HASH JOIN          |       |
|   3 |    TABLE ACCESS FULL| TEST   |
|   4 |    TABLE ACCESS FULL| TEST1  |
-------------------------------------
```

To adopt an alternative plan regardless of whether SQL Tuning Advisor recommends it, call `DBMS_SQLTUNE.CREATE_SQL_PLAN_BASELINE`. You can use this procedure to create a SQL plan baseline on any existing reproducible plan.

See Also:

"Differences Between SQL Plan Baselines and SQL Profiles"

25.1.3 SQL Tuning Advisor Operation

You can run SQL Tuning Advisor automatically or on demand. You can also run the advisor on a local or remote database.

25.1.3.1 Automatic and On-Demand SQL Tuning

Configure SQL Tuning Advisor to run automatically using `DBMS_AUTO_SQLTUNE`, or on demand using `DBMS_SQLTUNE`.

The methods of invocation differ as follows:

- Automatically

You can configure SQL Tuning Advisor to run during nightly system maintenance windows. When run by AUTOTASK, the advisor is known as Automatic SQL Tuning Advisor and performs automatic SQL tuning.

- On-Demand

  In on-demand SQL tuning, you manually invoke SQL Tuning Advisor to diagnose and fix SQL-related performance problems *after* they have been discovered. Oracle Enterprise Manager Cloud Control (Cloud Control) is the preferred interface for tuning SQL on demand, but you can also use the DBMS_SQLTUNE PL/SQL package.

SQL Tuning Advisor uses Automatic Tuning Optimizer to perform its analysis. This optimization is "automatic" because the optimizer analyzes the SQL instead of the user. Do not confuse Automatic Tuning Optimizer with automatic SQL tuning, which in this document refers *only* to the work performed by the Automatic SQL Tuning task.

See Also:

- "Running SQL Tuning Advisor On Demand"
- "Managing the Automatic SQL Tuning Task"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about DBMS_SQLTUNE

25.1.3.2 SQL Tuning on Active Data Guard Databases

You can run SQL Tuning Advisor either in a primary database or in a Data Guard Active Standby Database. In a Standby database, the SQL statement being tuned must be created in the standby database and you use a database link to write tuning details and results in the primary database.

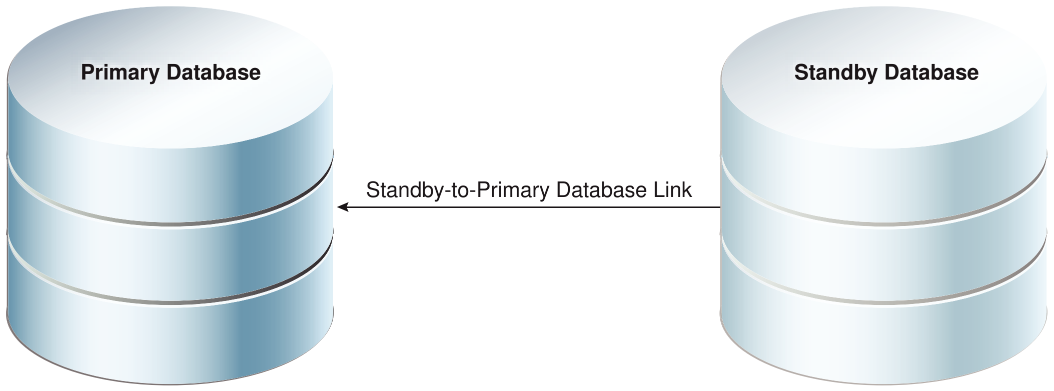
In the simplest case, SQL Tuning Advisor accepts input, executes, and stores results within a single database. Local mode is appropriate for databases in which the performance overhead of SQL Tuning Advisor execution is acceptable.

In remote tuning, you issue SQL Tuning Advisor tasks on a standby database in order to tune individual SQL statements. The SQL statement being tuned must be created on the primary database. A private, secure standby-to-primary database link is used to write data to and read data from the primary database. The link is necessary because the standby database, which is read-only, cannot write the SQL tuning data locally.

You can do remote tuning on the standby database by manually running DBMS_SQLTUNE statements on the command line. Starting with Oracle Database 19c, you also can use Oracle Enterprise Manager Cloud Control on the standby to create and execute tuning tasks and implement SQL profile recommendations.

The following below the general setup for tuning a standby database workload on a primary database. This technique requires a standby-to-primary database link.

Figure 25-10 Tuning a Standby Workload on a Primary Database



See Also:

- "Configuring a SQL Tuning Task"
- *Oracle Data Guard Concepts and Administration* to learn how to perform remote tuning in a Data Guard environment
- *Oracle Database PL/SQL Packages and Types Reference* for details about `DBMS_SQLTUNE`.

25.1.3.2.1 Using DBMS_SQLTUNE to Tune the Primary Database Remotely

If you want to tune a SQL statement written on the primary database on the standby database, then on the standby, specify the `database_link_to` parameter in `DBMS_SQLTUNE` procedures. By default, the `database_link_to` parameter is null, which means that tuning is local.

The `database_link_to` parameter must specify a private database link. This link must be owned by `SYS` and accessed by the default privileged user `SYS$UMF`. The following sample statement creates a link named `lnk_to_pri`:

```
CREATE DATABASE LINK lnk_to_pri CONNECT TO SYS$UMF IDENTIFIED BY
password USING 'inst1';
```

The following table illustrates a typical remote tuning session. You issue the SQL tuning statement on the standby database. `DBMS_SQLTUNE` uses the database link both to fetch data from the primary database, and store data in the primary database.

Table 25-1 Using DBMS_SQLTUNE on a Standby Database to Remotely Tune the Primary Database

Step	Statement Issued on Standby Database	Result
1	`CREATE_TUNING_TASK`	`DBMS_SQLTUNE` creates the task data in the primary database using the standby-to-primary database link.

Table 25-1 (Cont.) Using DBMS_SQLTUNE on a Standby Database to Remotely Tune the Primary Database

Step	Statement Issued on Standby Database	Result
2	`EXECUTE_TUNING_TASK`	`DBMS_SQLTUNE` uses the database link to read the SQL Tuning Advisor task data stored in the primary database. The tuning analysis occurs on the standby database, but `DBMS_SQLTUNE` writes the results remotely to the primary database.
3	`REPORT_TUNING_TASK`	`DBMS_SQLTUNE` uses the database link to read the SQL Tuning Advisor report data from the primary database, and then constructs the report locally on the standby database.
4	`ACCEPT_SQL_PROFILE`	`DBMS_SQLTUNE` uses the database link to write the SQL profile data remotely to the primary database.

25.1.3.2.2 Using Enterprise Manager Cloud Control to Tune an Active Standby Query Workload

Prerequisites

- Enterprise Manager must be discovered and must be connected to an Active Data Guard Standby Database.
- The SQL tuning statement used must be created on the primary database.
- A non-public database link that points to the primary database must be created on the primary database for user `SYS$UMF`. In each case, in order to execute tuning tasks from an Active Data Guard Standby Database in a PDB, a separate database link must be created for that PDB. For example:

  ```
  CREATE DATABASE LINK lnk_to_pri CONNECT TO SYS$UMF IDENTIFIED BY
  <password> USING \
  '(DESCRIPTION=(ADDRESS=(PROTOCOL=TCP)(HOST=<fully_qualified_hostname>)
  (PORT=<port_number>)) \
  (CONNECT_DATA=(SERVICE_NAME=<fully_qualified_service_name>)))';
  ```

  You select this database link in SQL Tuning Advisor on the standby.
- The `SYS$UMF` account must be unlocked with password set to the same password as the database link.

Note:

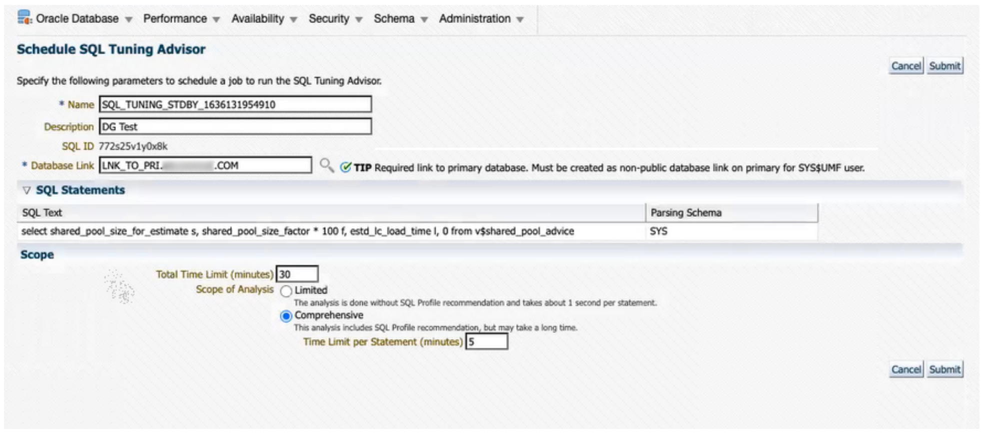
In all cases in this section, the "standby" refers to an Active Data Guard Standby Database.

See Also:

Oracle Data Guard Concepts and Administration.

Steps

1. Log in to the standby database as a user with privileges to execute `DBMS_SQLTUNE` procedures, such as the `SYS` user.
2. There are several different ways to select the query you want to tune. From the **Performance** menu, one way is to click **Performance Hub**, and then **Ash Analytics**. Find and select the query, then click **Tune SQL** to bring the query into SQL Tuning Advisor.



Schedule SQL Tuning Advisor

Specify the following parameters to schedule a job to run the SQL Tuning Advisor.

* Name	SQL_TUNING_STDBY_1636131954910
Description	DG Test
SQL ID	772s25v1y0x8k
* Database Link	LNK_TO_PRI.[redacted].COM

✓ **TIP** Required link to primary database. Must be created as non-public database link on primary for SYS$UMF user.

▽ **SQL Statements**

SQL Text	Parsing Schema
select shared_pool_size_for_estimate s, shared_pool_size_factor * 100 f, estd_lc_load_time l, 0 from v$shared_pool_advice	SYS

Scope

Total Time Limit (minutes) 30

Scope of Analysis [ ] Limited
The analysis is done without SQL Profile recommendation and takes about 1 second per statement.
[✓ Comprehensive]
This analysis includes SQL Profile recommendation, but may take a long time.
Time Limit per Statement (minutes) 5

> **Note:**
>
> The **Database Link** field shown above appears only in SQL Tuning Advisor on the standby. It is not visible from a primary database.

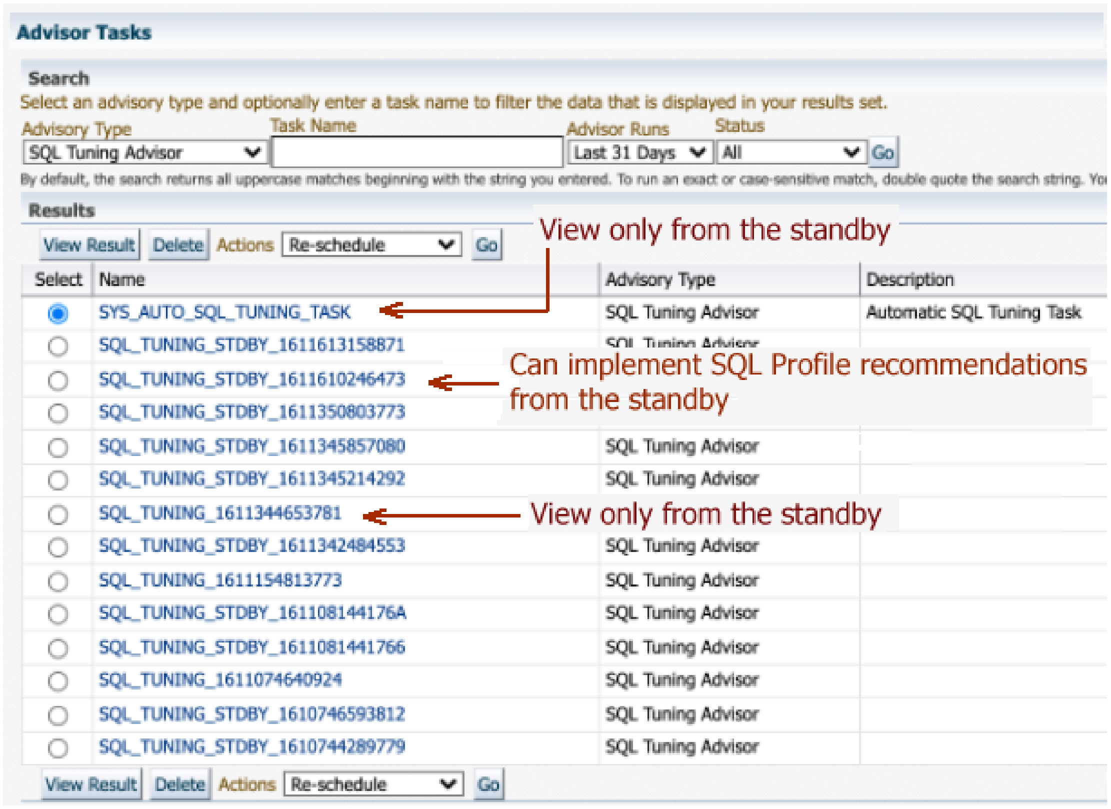
3. You can either accept the default tuning task name or change it. The default name includes an " `_STDBY_`" segment so that you can distinguish tasks created on the standby from those created on the primary. If you change it, be sure to give it a name which indicates that the task was created from the standby.
4. In the **Database Link** field, select the database link to the primary database. This must be a `SYS$UMF` link.
5. Click **Submit**.
6. When the task is done, the **SQL Tuning Result Summary** page displays. After reviewing the results, click **SQL Profile**. (See *Viewing the Result of a SQL Tuning Task* at the end of this topic, which shows an example of a result summary.)
7. On the **SQL Result Details** page, click **View Recommendations**.
8. **On the Recommendation for SQL <*ID name*>** page, click **Implement**.
9. On the **Confirmation** page, click **Yes**. (You may choose to implement the new profile with forced matching if you want this to impact the same SQL, but with different values).

> **Note:**
>
> You can access previously-created SQL tuning tasks by navigating to **Advisors Home**. From there, you can view all tasks that were created on both the primary database and on the standby. You can select and implement SQL Profile recommendations for tuning tasks that were created from the standby.

SQL Tuning Advisor Limitations From the Active Data Guard Standby Database

- You can only implement SQL Profile recommendations for tasks that were created on the standby. (This is why it is important that the name of a task includes some indicator that it was created on the standby.)
- You cannot implement recommendations for the `SYS_AUTO_SQL_TUNING_TASK` or any other task that was created from the primary database.
- You cannot collect optimizer statistics.
- You cannot implement index recommendations.
- A tuning task can tune only a single SQL statement (not a SQL tuning set).

The image below shows the **Advisor Tasks** section of **Advisors Home**. By default, user-created tasks include "_STDBY_" in the task name, as shown in this list. The tasks that do not include that segment in the name are presumed to tasks that were created from the primary database. SQL Profile recommendations for these tasks cannot be implemented from the standby database. The SYS_AUTO_SQL_TUNING_TASK also cannot be implemented from the standby database.



Advisor Tasks

Search

Select an advisory type and optionally enter a task name to filter the data that is displayed in your results set.

Advisory Type	Task Name	Advisor Runs	Status
SQL Tuning Advisor		Last 31 Days	All

By default, the search returns all uppercase matches beginning with the string you entered. To run an exact or case-sensitive match, double quote the search string. Yo

Results

Select	Name	Advisory Type	Description
●	SYS_AUTO_SQL_TUNING_TASK	SQL Tuning Advisor	Automatic SQL Tuning Task
○	SQL_TUNING_STDBY_1611613158871	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_1611610246473		
○	SQL_TUNING_STDBY_1611350803773		
○	SQL_TUNING_STDBY_1611345857080	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_1611345214292	SQL Tuning Advisor	
○	SQL_TUNING_1611344653781		
○	SQL_TUNING_STDBY_1611342484553	SQL Tuning Advisor	
○	SQL_TUNING_1611154813773	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_161108144176A	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_1611081441766	SQL Tuning Advisor	
○	SQL_TUNING_1611074640924	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_1610746593812	SQL Tuning Advisor	
○	SQL_TUNING_STDBY_1610744289779	SQL Tuning Advisor	

View Result
Delete
Actions
Re-schedule
Go
View only from the standby
Can implement SQL Profile recommendations from the standby
View only from the standby

Viewing the Result of a SQL Tuning Task

The **SQL Tuning Task Result Summary** displays after a SQL Tuning Task completes and also when you select a tuning task from the **Advisor Central** page.

On an Active Data Guard Standby database you can view the result summary of a completed SQL tuning task that was executed on the same standby. You can click **SQL Profiles** view and implement recommended SQL profiles. The **Index** and **Statistics** links are view only.

25.2 Managing the Automatic SQL Tuning Task

When your goal is to identify SQL performance problems proactively, configuring SQL Tuning Advisor as an automated task is a simple solution. The task processes selected high-load SQL statements from AWR that qualify as tuning candidates.

See Also:

Oracle Database Administrator’s Guide to learn more about automated maintenance tasks

25.2.1 About the Automatic SQL Tuning Task

By default, the Automatic SQL Tuning task runs for in a nightly maintenance window.

See Also:

Oracle Database Administrator’s Guide to learn more about automatic maintenance tasks

25.2.1.1 Purpose of Automatic SQL Tuning

Configuring automatic SQL tuning instead of tuning manually decreases cost and increases manageability

Many DBAs do not have the time needed for the intensive analysis required for SQL tuning. Even when they do, SQL tuning involves several manual steps. Because several different SQL statements may be high load on any given day, DBAs may have to expend considerable effort to monitor and tune them. .

The automated SQL tuning task does *not* process the following types of SQL:

- Ad hoc SQL statements or SQL statements that do not repeat within a week
- Parallel queries
- Queries that take too long to run after being SQL profiled, so that it is not practical for SQL Tuning Advisor to test execution
- Recursive SQL

You can run SQL Tuning Advisor on demand to tune the preceding types of SQL statements.

25.2.1.2 Automatic SQL Tuning Concepts

Oracle Scheduler uses the automated maintenance tasks infrastructure (known as *AutoTask*) to schedules tasks to run automatically.

By default, the Automatic SQL Tuning task runs for at most one hour in a nightly maintenance window. You can customize attributes of the maintenance windows, including start and end time, frequency, and days of the week.

Automatic SQL Tuning Advisor data is stored in the CDB root and is only visible to a common user whose current container is the root. It might have results about SQL statements executed in a PDB that were analyzed by the advisor, but these results are not included if the PDB is unplugged. The results cannot be viewed when the current container is a PDB.

See Also:

- *Oracle Database Administrator's Guide* to learn about Oracle Scheduler
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `DBMS_AUTO_TASK_ADMIN`

25.2.1.3 Command-Line Interface to SQL Tuning Advisor

On the command line, you can use PL/SQL packages to perform SQL tuning tasks.

The following table describes the most relevant packages.

Table 25-2 SQL Tuning Advisor Packages

Package	Description
`DBMS_AUTO_SQLTUNE`	Enables you run SQL Tuning Advisor, manage SQL profiles, manage SQL tuning sets, and perform real-time SQL performance monitoring. To use this API, you must have the `ADVISOR` privilege.
`DBMS_AUTO_TASK_ADMIN`	Provides an interface to `AUTOTASK`. You can use this interface to enable and disable the Automatic SQL Tuning task.

See Also:

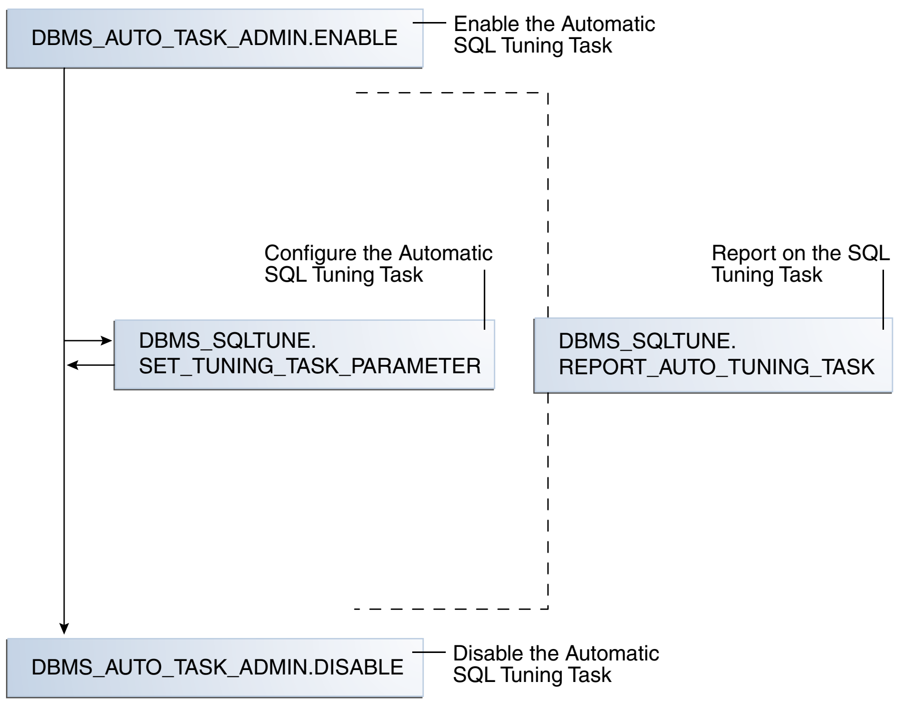
Oracle Database PL/SQL Packages and Types Reference to learn about `DBMS_SQLTUNE` ad `DBMS_AUTO_TASK_ADMIN`

25.2.1.4 Basic Tasks for Automatic SQL Tuning

This section explains the basic tasks in running SQL Tuning Advisor as an automatic task.

The following graphic shows the basic workflow.

Figure 25-11 Automatic SQL Tuning APIs


DBMS_AUTO_TASK_ADMIN.ENABLE
Enable the Automatic SQL Tuning Task
Configure the Automatic SQL Tuning Task
Report on the SQL Tuning Task
DBMS_SQLTUNE. SET_TUNING_TASK_PARAMETER
DBMS_SQLTUNE. REPORT_AUTO_TUNING_TASK
DBMS_AUTO_TASK_ADMIN.DISABLE
Disable the Automatic SQL Tuning Task

As shown in Figure 25-12, the basic procedure is as follows:

1. Enable the Automatic SQL Tuning task.

   See "Enabling and Disabling the Automatic SQL Tuning Task".
2. Optionally, configure the Automatic SQL Tuning task.

   See "Configuring the Automatic SQL Tuning Task".
3. Display the results of the Automatic SQL Tuning task.

   See "Viewing Automatic SQL Tuning Reports".
4. Disable the Automatic SQL Tuning task.

   See "Enabling and Disabling the Automatic SQL Tuning Task".

25.2.2 Enabling and Disabling the Automatic SQL Tuning Task

You can enable or disable the Automatic SQL Tuning task using Cloud Control (preferred) or a command-line interface.

25.2.2.1 Enabling and Disabling the Automatic SQL Tuning Task Using Cloud Control

You can enable and disable all automatic maintenance tasks, including the Automatic SQL Tuning task, using Cloud Control.

To enable or disable the Automatic SQL Tuning task using Cloud Control:

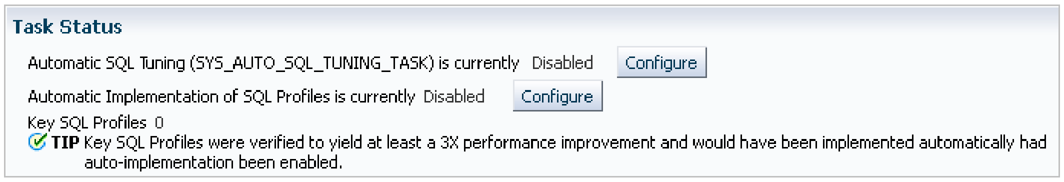
1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Administration** menu, select **Oracle Scheduler**, then **Automated Maintenance Tasks**.

   The Automated Maintenance Tasks page appears.

   This page shows the predefined tasks. You access each task by clicking the corresponding link to get more information about the task.
6. Click **Automatic SQL Tuning**.

   The Automatic SQL Tuning Result Summary page appears.

   The Task Status section shows whether the Automatic SQL Tuning Task is enabled or disabled. In the following graphic, the task is disabled:



Task Status

Automatic SQL Tuning (SYS_AUTO_SQL_TUNING_TASK) is currently Disabled

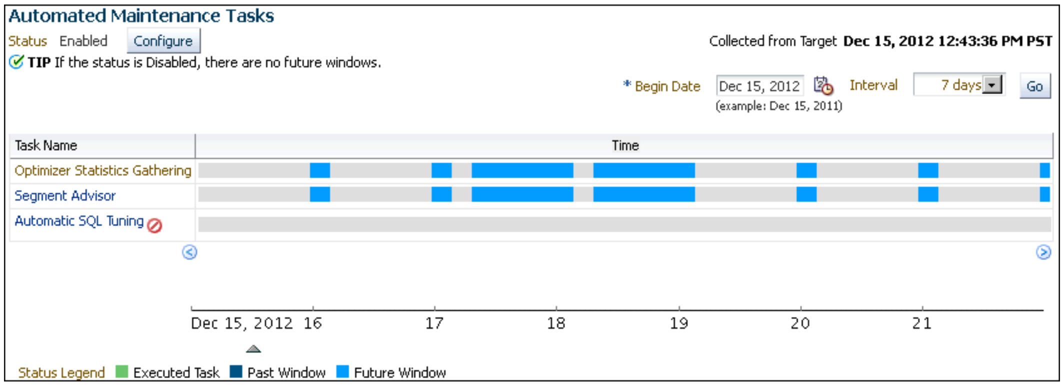
Automatic Implementation of SQL Profiles is currently Disabled

Key SQL Profiles 0

✓ **TIP** Key SQL Profiles were verified to yield at least a 3X performance improvement and would have been implemented automatically had auto-implementation been enabled.

7. In Automatic SQL Tuning, click **Configure**.

   The Automated Maintenance Tasks Configuration page appears.



Automated Maintenance Tasks

Status Enabled

Collected from Target **Dec 15, 2012 12:43:36 PM PST**

✓ **TIP** If the status is Disabled, there are no future windows.

* Begin Date Dec 15, 2012 (example: Dec 15, 2011) Interval 7 days

Task Name	Time
Optimizer Statistics Gathering	
Segment Advisor	
Automatic SQL Tuning	

By default, Automatic SQL Tuning executes in all predefined maintenance windows in `MAINTENANCE_WINDOW_GROUP`.

8. Perform the following steps:
   a. In the Task Settings for Automatic SQL Tuning, select either **Enabled** or **Disabled** to enable or disable the automated task.
   b. To disable Automatic SQL Tuning for specific days in the week, check the appropriate box next to the window name.
   c. To change the characteristics of a window, click **Edit Window Group**.
   d. Click **Apply**.

25.2.2.2 Enabling and Disabling the Automatic SQL Tuning Task from the Command Line

If you do not use Cloud Control to enable and disable the Automatic SQL Tuning task, then you must use the command line.

You have the following options:

- Run the `ENABLE` or `DISABLE` procedure in the `DBMS_AUTO_TASK_ADMIN` PL/SQL package.

  This package is the recommended command-line technique. For both the `ENABLE` and `DISABLE` procedures, you can specify a particular maintenance window with the `window_name` parameter.
- Set the `STATISTICS_LEVEL` initialization parameter to `BASIC` to disable collection of *all* advisories and statistics, including Automatic SQL Tuning Advisor.

  Because monitoring and many automatic features are disabled, Oracle strongly recommends that you do not set `STATISTICS_LEVEL` to `BASIC`.

To enable or disable Automatic SQL Tuning using DBMS_AUTO_TASK_ADMIN:

1. Connect SQL*Plus to the database with administrator privileges, and then do one of the following:
   - To enable the automated task, execute the following PL/SQL block:

     ```
     BEGIN
       DBMS_AUTO_TASK_ADMIN.ENABLE (
         client_name => 'sql tuning advisor'
     ,   operation   => NULL
     ,   window_name => NULL
     );
     END;
     /
     ```

   - To disable the automated task, execute the following PL/SQL block:

     ```
     BEGIN
       DBMS_AUTO_TASK_ADMIN.DISABLE (
         client_name => 'sql tuning advisor'
     ,   operation   => NULL
     ,   window_name => NULL
     ```

```
);
END;
/
```

2. Query the data dictionary to confirm the change.

   For example, query `DBA_AUTOTASK_CLIENT` as follows (sample output included):

   ```
   COL CLIENT_NAME FORMAT a20

   SELECT CLIENT_NAME, STATUS
   FROM   DBA_AUTOTASK_CLIENT
   WHERE  CLIENT_NAME = 'sql tuning advisor';

   CLIENT_NAME          STATUS
   -------------------- --------
   sql tuning advisor   ENABLED
   ```

To disable collection of all advisories and statistics:

1. Connect SQL*Plus to the database with administrator privileges, and then query the current statistics level setting.

   The following SQL*Plus command shows that `STATISTICS_LEVEL` is set to `ALL`:

   ```
   sys@PROD> SHOW PARAMETER statistics_level

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   statistics_level                     string      ALL
   ```

2. Set `STATISTICS_LEVEL` to `BASIC` as follows:

   ```
   sys@PROD> ALTER SYSTEM SET STATISTICS_LEVEL ='BASIC';

   System altered.
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information

25.2.3 Configuring the Automatic SQL Tuning Task

You can configure the Automatic SQL Tuning task using Cloud Control or the command line.

25.2.3.1 Configuring the Automatic SQL Tuning Task Using Cloud Control

You can enable and disable all automatic maintenance tasks, including the Automatic SQL Tuning task, using Cloud Control. You must perform the operation as `SYS` or have the `EXECUTE` privilege on the PL/SQL package `DBMS_AUTO_SQLTUNE`.

To configure the Automatic SQL Tuning task using Cloud Control:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Administration** menu, click **Oracle Scheduler**, then **Automated Maintenance Tasks**.

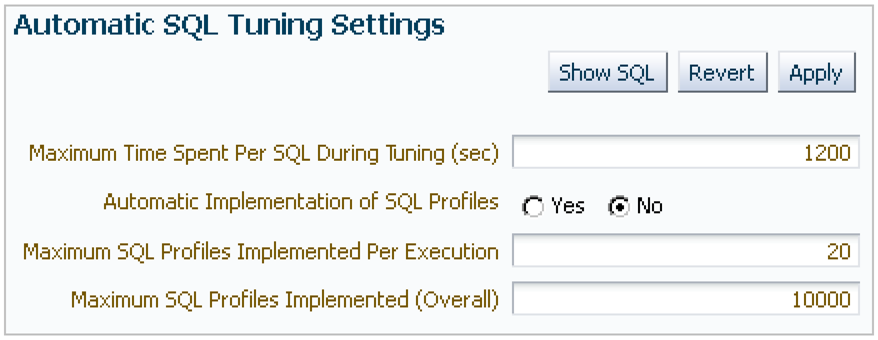
   The Automated Maintenance Tasks page appears.

   This page shows the predefined tasks. You access each task by clicking the corresponding link to get more information about the task itself.
6. Click **Automatic SQL Tuning**.

   The Automatic SQL Tuning Result Summary page appears.
7. Under Task Settings, click **Configure** next to Automatic SQL Tuning (`SYS_AUTO_SQL_TUNING_TASK`).

   The Automated Maintenance Tasks Configuration page appears.
8. Under Task Settings, click **Configure** next to Automatic SQL Tuning.

   The Automatic SQL Tuning Settings page appears.



Maximum Time Spent Per SQL During Tuning (sec)	1200
Automatic Implementation of SQL Profiles	Yes [✓ No]
Maximum SQL Profiles Implemented Per Execution	20
Maximum SQL Profiles Implemented (Overall)	10000

9. Make the desired changes and click **Apply**.

25.2.3.2 Configuring the Automatic SQL Tuning Task Using the Command Line

The `DBMS_AUTO_SQLTUNE` package enables you to configure automatic SQL tuning by specifying the task parameters using the `SET_AUTO_TUNING_TASK_PARAMETER` procedure.

Because the task is owned by `SYS`, only `SYS` can set task parameters.

The ACCEPT_SQL_PROFILE tuning task parameter specifies whether to implement SQL profiles automatically (true) or require user intervention (false). The default is AUTO, which means true if at least one SQL statement exists with a SQL profile and false if this condition is not satisfied.

Note:

When automatic implementation is enabled, the advisor only implements recommendations to create SQL profiles. Recommendations such as creating new indexes, gathering optimizer statistics, and creating SQL plan baselines are not automatically implemented.

Assumptions

This tutorial assumes the following:

- You want the database to implement SQL profiles automatically, but to implement no more than 50 SQL profiles per execution, and no more than 50 profiles total on the database.
- You want the task to time out after 1200 seconds per execution.

To set Automatic SQL Tuning task parameters:

1. Connect SQL*Plus to the database with the appropriate privileges, and then optionally query the current task settings.

   For example, connect SQL*Plus to the database with administrator privileges and execute the following query:

```
COL PARAMETER_NAME FORMAT a25
COL VALUE FORMAT a10

SELECT PARAMETER_NAME, PARAMETER_VALUE AS "VALUE"
FROM   DBA_ADVISOR_PARAMETERS
WHERE  ( (TASK_NAME = 'SYS_AUTO_SQL_TUNING_TASK') AND
         ( (PARAMETER_NAME LIKE '%PROFILE%') OR
           (PARAMETER_NAME = 'LOCAL_TIME_LIMIT') OR
           (PARAMETER_NAME = 'EXECUTION_DAYS_TO_EXPIRE') ) );
```

   Sample output appears as follows:

```
PARAMETER_NAME            VALUE
------------------------- ----------
EXECUTION_DAYS_TO_EXPIRE  30
LOCAL_TIME_LIMIT          1000
ACCEPT_SQL_PROFILES       FALSE
MAX_SQL_PROFILES_PER_EXEC 20
MAX_AUTO_SQL_PROFILES     10000
```

2. Set parameters using PL/SQL code of the following form:

   ```
   BEGIN
     DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER (
       task_name => 'SYS_AUTO_SQL_TUNING_TASK'
   ,   parameter => parameter_name
   ,   value     => value
   );
   END;
   /
   ```

Example 25-3 Setting SQL Tuning Task Parameters

The following PL/SQL block sets a time limit to 20 minutes, and also automatically implements SQL profiles and sets limits for these profiles:

```
BEGIN
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER('SYS_AUTO_SQL_TUNING_TASK',
    'LOCAL_TIME_LIMIT', 1200);
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER('SYS_AUTO_SQL_TUNING_TASK',
    'ACCEPT_SQL_PROFILES', 'true');
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER('SYS_AUTO_SQL_TUNING_TASK',
    'MAX_SQL_PROFILES_PER_EXEC', 50);
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER('SYS_AUTO_SQL_TUNING_TASK',
    'MAX_AUTO_SQL_PROFILES', 10002);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information for `DBMS_AUTO_SQLTUNE`

25.2.4 Viewing Automatic SQL Tuning Reports

At any time during or after the running of the Automatic SQL Tuning task, you can view a tuning report.

The tuning report contains information about all executions of the automatic SQL tuning task. Depending on the sections that were included in the report, you can view information in the following sections:

- General information

  This section has a high-level description of the automatic SQL tuning task, including information about the inputs given for the report, the number of SQL statements tuned during the maintenance, and the number of SQL profiles created.

- Summary

  This section lists the SQL statements (by their SQL identifiers) that were tuned during the maintenance window and the estimated benefit of each SQL profile, or the execution statistics after performing a test execution of the SQL statement with the SQL profile.

- Tuning findings

  This section contains the following information about each SQL statement analyzed by SQL Tuning Advisor:

  – All findings associated with each SQL statement
  – Whether the profile was implemented on the database, and why
  – Whether the SQL profile is currently enabled on the database
  – Detailed execution statistics captured when testing the SQL profile

- Explain plans

  This section shows the old and new explain plans used by each SQL statement analyzed by SQL Tuning Advisor.

- Errors

  This section lists all errors encountered by the automatic SQL tuning task.

25.2.4.1 Viewing Automatic SQL Tuning Reports Using the Command Line

To generate a SQL tuning report as a `CLOB`, execute the `DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK` function.

You can store the `CLOB` in a variable and then print the variable to view the report.

Assumptions

This section assumes that you want to show all SQL statements that were analyzed in the most recent execution, including recommendations that were not implemented.

To create and access an Automatic SQL Tuning Advisor report:

1. Connect SQL*Plus to the database with administrator privileges, and then execute the `DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK` function.

   The following example generates a text report to show all SQL statements that were analyzed in the most recent execution, including recommendations that were not implemented:

```
VARIABLE my_rept CLOB;
BEGIN
  :my_rept :=DBMS_SQLTUNE.REPORT_AUTO_TUNING_TASK (
    begin_exec   => NULL
,   end_exec     => NULL
,   type         => 'TEXT'
,   level        => 'TYPICAL'
,   section      => 'ALL'
,   object_id    => NULL
,   result_limit => NULL
);
END;
/
```

```
PRINT :my_rept
```

2. Read the general information section for an overview of the tuning execution.

   The following sample shows the Automatic SQL Tuning task analyzed 17 SQL statements in just over 7 minutes:

```
MY_REPT
---------------------------------------------------------------------
----
GENERAL INFORMATION SECTION
---------------------------------------------------------------------
----
Tuning Task Name                     : SYS_AUTO_SQL_TUNING_TASK
Tuning Task Owner                    : SYS
Workload Type                        : Automatic High-Load SQL
Workload
Execution Count                      : 6
Current Execution                    : EXEC_170
Execution Type                       : TUNE SQL
Scope                                : COMPREHENSIVE
Global Time Limit(seconds)           : 3600
Per-SQL Time Limit(seconds)          : 1200
Completion Status                    : COMPLETED
Started at                           : 04/16/2012 10:00:00
Completed at                         : 04/16/2012 10:07:11
Number of Candidate SQLs             : 17
Cumulative Elapsed Time of SQL (s)   : 8
```

3. Look for findings and recommendations.

   If SQL Tuning Advisor makes a recommendation, then weigh the pros and cons of accepting it.

   The following example shows that SQL Tuning Advisor found a plan for a statement that is potentially better than the existing plan. The advisor recommends implementing a SQL profile.

```
---------------------------------------------------------------------
----
SQLs with Findings Ordered by Maximum (Profile/Index) Benefit,
Object ID
---------------------------------------------------------------------
----
ob ID  SQL ID        stats profile(benefit) index(benefit)
restructure
------ ------------- ----- ---------------- --------------
-----------
    82 dqjcc345dd4ak                 58.03%
    72
51bbkcd9zwsjw                                              2
    81 03rxjf8gb18jg

---------------------------------------------------------------------
----
```

```
DETAILS SECTION
-------------------------------------------------------------------------
 Statements with Results Ordered by Max (Profile/Index) Benefit, Obj ID
-------------------------------------------------------------------------
Object ID  : 82
Schema Name: DBA1
SQL ID     : dqjcc345dd4ak
SQL Text   : SELECT status FROM dba_autotask_client WHERE client_name=:1

-------------------------------------------------------------------------
FINDINGS SECTION (1 finding)
-------------------------------------------------------------------------

1- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.
  The SQL profile was not automatically created because the verified
  benefit was too low.

  Recommendation (estimated benefit: 58.03%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile.
    execute dbms_sqltune.accept_sql_profile(task_name =>
         'SYS_AUTO_SQL_TUNING_TASK', object_id => 82, replace => TRUE);

  Validation results
  ------------------
  The SQL profile was tested by executing its plan and the original
  plan and measuring their respective execution statistics. A plan
  may have been only partially executed if the other could be run
  to completion in less time.

                           Original Plan  With SQL Profile  % Improved
                           -------------  ----------------  ----------
  Completion Status:            COMPLETE          COMPLETE
  Elapsed Time(us):                26963              8829     67.25 %
  CPU Time(us):                    27000              9000     66.66 %
  User I/O Time(us):                  25                14        44 %
  Buffer Gets:                       905               380     58.01 %
  Physical Read Requests:              0                 0
  Physical Write Requests:             0                 0
  Physical Read Bytes:                 0                 0
  Physical Write Bytes:             7372              7372         0 %
  Rows Processed:                      1                 1
  Fetches:                             1                 1
  Executions:                          1                 1

  Notes
  -----
  1. The original plan was first executed to warm the buffer cache.
  2. Statistics for original plan were averaged over next 9 executions.
  3. The SQL profile plan was first executed to warm the buffer cache.
  4. Statistics for the SQL profile plan were averaged over
     next 9 executions.
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information.

25.3 Running SQL Tuning Advisor On Demand

You can run SQL Tuning Advisor on demand.

25.3.1 About On-Demand SQL Tuning

On-demand SQL tuning is defined as any invocation of SQL Tuning Advisor that does not result from the Automatic SQL Tuning task.

25.3.1.1 Purpose of On-Demand SQL Tuning

Typically, you invoke SQL Tuning Advisor to run ADDM proactively, or to tune SQL statement reactively when users complain about suboptimal performance.

In both the proactive and reactive scenarios, running SQL Tuning Advisor is usually the quickest way to fix unexpected SQL performance problems.

25.3.1.2 User Interfaces for On-Demand SQL Tuning

The recommended user interface for running SQL Tuning Advisor manually is Cloud Control.

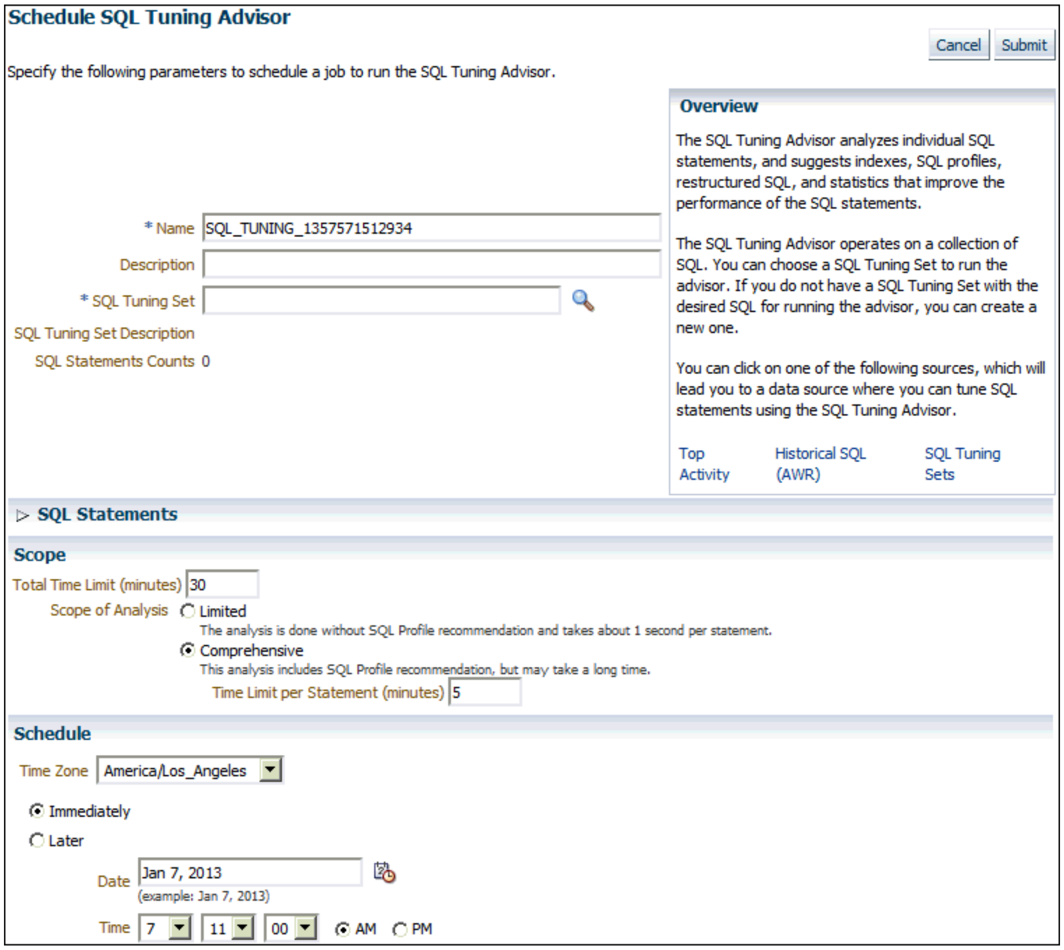
25.3.1.2.1 Accessing the SQL Tuning Advisor Using Cloud Control

Automatic Database Diagnostic Monitor (ADDM) automatically identifies high-load SQL statements. If ADDM identifies such statements, then click **Schedule/Run SQL Tuning Advisor** on the Recommendation Detail page to run SQL Tuning Advisor.

To tune SQL statements manually using SQL Tuning Advisor:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Performance** menu, click **SQL**, then **SQL Tuning Advisor**.

   The Schedule SQL Tuning Advisor page appears.



Schedule SQL Tuning Advisor

Cancel Submit

Specify the following parameters to schedule a job to run the SQL Tuning Advisor.

* Name	SQL_TUNING_1357571512934
Description	
* SQL Tuning Set	
SQL Tuning Set Description	
SQL Statements Counts	0

Overview

The SQL Tuning Advisor analyzes individual SQL statements, and suggests indexes, SQL profiles, restructured SQL, and statistics that improve the performance of the SQL statements.

The SQL Tuning Advisor operates on a collection of SQL. You can choose a SQL Tuning Set to run the advisor. If you do not have a SQL Tuning Set with the desired SQL for running the advisor, you can create a new one.

You can click on one of the following sources, which will lead you to a data source where you can tune SQL statements using the SQL Tuning Advisor.

Top Activity | Historical SQL (AWR) | SQL Tuning Sets

▷ SQL Statements

Scope

Total Time Limit (minutes) 30

Scope of Analysis [ ] Limited
The analysis is done without SQL Profile recommendation and takes about 1 second per statement.
[✓ Comprehensive]
This analysis includes SQL Profile recommendation, but may take a long time.
Time Limit per Statement (minutes) 5

Schedule

Time Zone America/Los_Angeles

[✓ Immediately]
Later

Date Jan 7, 2013
(example: Jan 7, 2013)

Time 7 11 00 [✓ AM] PM

See Also:

Oracle Database 2 Day + Performance Tuning Guide to learn how to configure and run SQL Tuning Advisor using Cloud Control.

25.3.1.2.2 Command-Line Interface to On-Demand SQL Tuning

If Cloud Control is unavailable, then you can run SQL Tuning Advisor using procedures in the `DBMS_SQLTUNE` package.

To use the APIs, the user must have the `ADVISOR` privilege.

See Also:

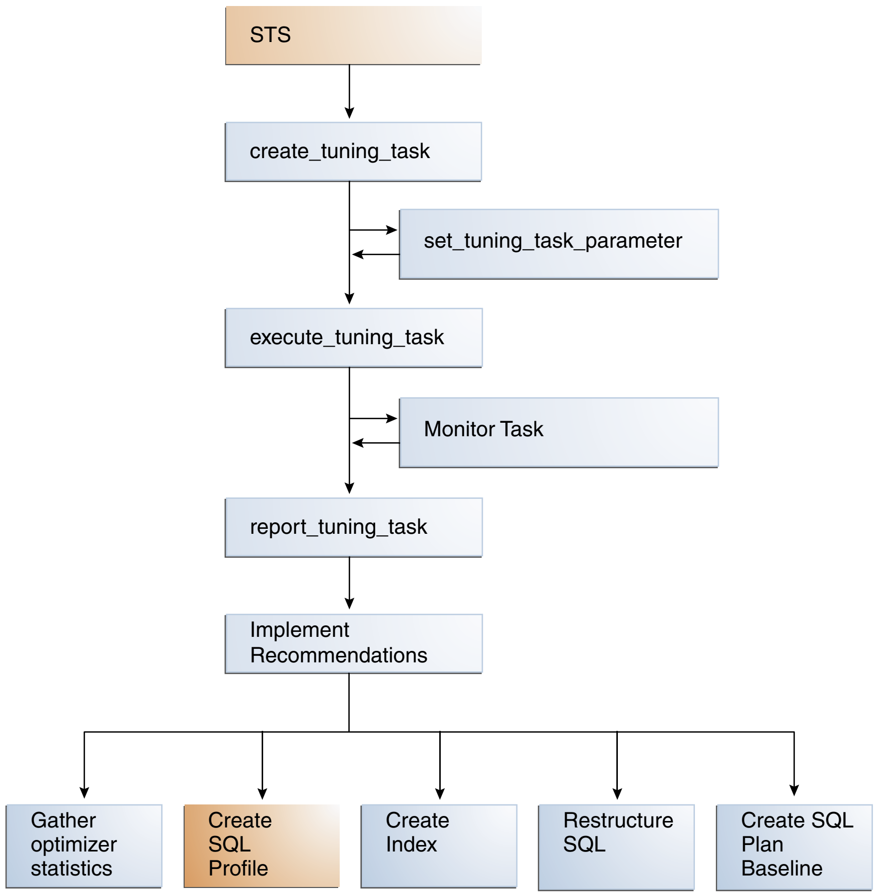
Oracle Database PL/SQL Packages and Types Reference for complete reference information

25.3.1.3 Basic Tasks in On-Demand SQL Tuning

This section explains the basic tasks in running SQL Tuning Advisor using the `DBMS_SQLTUNE` package.

The following graphic shows the basic workflow when using the PL/SQL APIs.

Figure 25-12   SQL Tuning Advisor APIs



As shown in Figure 25-12, the basic procedure is as follows:

1. Prepare or create the input to SQL Tuning Advisor. The input can be either:
   - The text of a single SQL statement
   - A SQL tuning set that contains one or more statements
2. Create a SQL tuning task.

   See "Creating a SQL Tuning Task".
3. Optionally, configure the SQL tuning task that you created.

   See "Configuring a SQL Tuning Task".
4. Execute a SQL tuning task.

   See "Executing a SQL Tuning Task".

5. Optionally, check the status or progress of a SQL tuning task.

   "Monitoring a SQL Tuning Task".
6. Display the results of a SQL tuning task.

   "Displaying the Results of a SQL Tuning Task".
7. Implement recommendations as appropriate.

See Also:

Oracle Database 2 Day + Performance Tuning Guide to learn how to tune SQL using Cloud Control

25.3.2 Creating a SQL Tuning Task

To create a SQL tuning task execute the `DBMS_SQLTUNE.CREATE_TUNING_TASK` function.

You can create tuning tasks from any of the following:

- The text of a single SQL statement
- A SQL tuning set containing multiple statements
- A SQL statement selected by SQL identifier from the shared SQL area
- A SQL statement selected by SQL identifier from AWR

The `scope` parameter is one of the most important for this function. You can set this parameter to the following values:

- `LIMITED`

  SQL Tuning Advisor produces recommendations based on statistical checks, access path analysis, and SQL structure analysis. SQL profile recommendations are not generated.
- `COMPREHENSIVE`

  SQL Tuning Advisor carries out all the analysis it performs under limited scope plus SQL profiling.

Assumptions

This tutorial assumes the following:

- You want to tune as user `hr`, who has the `ADVISOR` privilege.
- You want to tune the following query:

```
SELECT /*+ ORDERED */ *
FROM   employees e, locations l, departments d
WHERE  e.department_id = d.department_id
AND    l.location_id = d.location_id
AND    e.employee_id < :bnd;
```

- You want to pass the bind variable `100` to the preceding query.
- You want SQL Tuning Advisor to perform SQL profiling.

- You want the task to run no longer than 60 seconds.

To create a SQL tuning task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then run the `DBMS_SQLTUNE.CREATE_TUNING_TASK` function.

   For example, execute the following PL/SQL program:

```
DECLARE
  my_task_name VARCHAR2(30);
  my_sqltext   CLOB;
BEGIN
  my_sqltext := 'SELECT /*+ ORDERED */ * '                      ||
                'FROM employees e, locations l, departments d ' ||
                'WHERE e.department_id = d.department_id AND '  ||
                      'l.location_id = d.location_id AND '      ||
                      'e.employee_id < :bnd';

  my_task_name := DBMS_SQLTUNE.CREATE_TUNING_TASK (
          sql_text    => my_sqltext
,         bind_list   => sql_binds(anydata.ConvertNumber(100))
,         user_name   => 'HR'
,         scope       => 'COMPREHENSIVE'
,         time_limit  => 60
,         task_name   => 'STA_SPECIFIC_EMP_TASK'
,         description => 'Task to tune a query on a specified
employee'
);
END;
/
```

2. Optionally, query the status of the task.

   The following example queries the status of all tasks owned by the current user, which in this example is `hr`:

```
COL TASK_ID FORMAT 999999
COL TASK_NAME FORMAT a25
COL STATUS_MESSAGE FORMAT a33

SELECT TASK_ID, TASK_NAME, STATUS, STATUS_MESSAGE
FROM   USER_ADVISOR_LOG;
```

   Sample output appears below:

```
TASK_ID TASK_NAME                 STATUS      STATUS_MESSAGE
------- ------------------------- ----------- --------------
    884 STA_SPECIFIC_EMP_TASK     INITIAL
```

   In the preceding output, the `INITIAL` status indicates that the task has not yet started execution.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_SQLTUNE.CREATE_TUNING_TASK` function

25.3.3 Configuring a SQL Tuning Task

To change the parameters of a tuning task after it has been created, execute the `DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER` function.

This tutorial assumes the following:

- You want to tune with user account `hr`, which has been granted the `ADVISOR` privilege.
- You want to tune the `STA_SPECIFIC_EMP_TASK` created in "Creating a SQL Tuning Task".
- You want to change the maximum time that the SQL tuning task can run to 300 seconds.

To configure a SQL tuning task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then run the `DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER` function.

   For example, execute the following PL/SQL program to change the time limit of the tuning task to 300 seconds:

```
BEGIN
  DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER (
    task_name => 'STA_SPECIFIC_EMP_TASK'
,   parameter => 'TIME_LIMIT'
,   value     => 300
);
END;
/
```

2. Optionally, verify that the task parameter was changed.

   The following example queries the values of all used parameters in task `STA_SPECIFIC_EMP_TASK`:

```
COL PARAMETER_NAME FORMAT a25
COL VALUE FORMAT a15

SELECT PARAMETER_NAME, PARAMETER_VALUE AS "VALUE"
FROM   USER_ADVISOR_PARAMETERS
WHERE  TASK_NAME = 'STA_SPECIFIC_EMP_TASK'
AND    PARAMETER_VALUE != 'UNUSED'
ORDER BY PARAMETER_NAME;
```

   Sample output appears below:

```
PARAMETER_NAME            VALUE
------------------------- ---------------
DAYS_TO_EXPIRE            30
```

```
    DEFAULT_EXECUTION_TYPE      TUNE SQL
    EXECUTION_DAYS_TO_EXPIRE    UNLIMITED
    JOURNALING                  INFORMATION
    MODE                        COMPREHENSIVE
    SQL_LIMIT                   -1
    SQL_PERCENTAGE              1
    TARGET_OBJECTS              1
    TEST_EXECUTE                AUTO
    TIME_LIMIT                  300
```

Example 25-4   Tuning a Standby Database Workload Using a Database Link

Starting in Oracle Database 12c Release 2 (12.2), you can tune a standby database workload by specifying a database link in the `database_link_to` parameter. For security reasons, Oracle recommends using a private database link. The link must be owned by `SYS` and accessed by a privileged user. Oracle Database includes a default privileged user named `SYS$UMF`.

The following program, which is issued on the standby database, shows a sample SQL tuning session for a query of `table1`. The `database_link_to` parameter specifies the name of the standby-to-primary database link.

```
VARIABLE tname VARCHAR2(30);
VARIABLE query VARCHAR2(500);

EXEC :tname := 'my_task';
EXEC :query := 'SELECT /*+ FULL(t)*/ col1 FROM table1 t WHERE
col1=9000';

BEGIN
:tname := DBMS_SQLTUNE.CREATE_TUNING_TASK(
            sql_text         => :query
          , task_name        => :tname
          , database_link_to => 'lnk_to_pri' );
END;
/

EXEC DBMS_SQLTUNE.EXECUTE_TUNING_TASK(:tname);

SELECT DBMS_SQLTUNE.REPORT_TUNING_TASK(:tname) FROM DUAL;

BEGIN
  DBMS_SQLTUNE.ACCEPT_SQL_PROFILE(
    task_name        => :tname
,   name             => 'prof'
,   task_owner       => 'SYS'
,   replace          => TRUE
,   database_link_to => 'lnk_to_pri' );
END;
/
```

Note that the `bind_list` parameter of `CREATE_TUNING_TASK` is not supported on a standby database.

See Also:

- "SQL Tuning on Active Data Guard Databases"
- *Oracle Data Guard Concepts and Administration* to learn how to perform remote tuning in a Data Guard environment
- *Oracle Database PL/SQL Packages and Types Reference* for `DBMS_SQLTUNE.SET_TUNING_TASK_PARAMETER` syntax and semantics

25.3.4 Executing a SQL Tuning Task

To execute a SQL tuning task, use the `DBMS_SQLTUNE.EXECUTE_TUNING_TASK` function. The most important parameter is `task_name`.

Note:

You can also execute the automatic tuning task `SYS_AUTO_SQL_TUNING_TASK` using the `EXECUTE_TUNING_TASK` API. SQL Tuning Advisor performs the same analysis and actions as it would when run automatically.

Assumptions

This tutorial assumes the following:

- You want to tune as user `hr`, who has the `ADVISOR` privilege.
- You want to execute the `STA_SPECIFIC_EMP_TASK` created in "Creating a SQL Tuning Task".

To execute a SQL tuning task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then run the `DBMS_SQLTUNE.EXECUTE_TUNING_TASK` function.

   For example, execute the following PL/SQL program:

   ```
   BEGIN
     DBMS_SQLTUNE.EXECUTE_TUNING_TASK(task_name=>'STA_SPECIFIC_EMP_TASK');
   END;
   /
   ```

2. Optionally, query the status of the task.

   The following example queries the status of all tasks owned by the current user, which in this example is `hr`:

   ```
   COL TASK_ID FORMAT 999999
   COL TASK_NAME FORMAT a25
   COL STATUS_MESSAGE FORMAT a33
   ```

```
SELECT TASK_ID, TASK_NAME, STATUS, STATUS_MESSAGE
FROM   USER_ADVISOR_LOG;
```

Sample output appears below:

```
TASK_ID TASK_NAME                 STATUS      STATUS_MESSAGE
------- ------------------------- ----------- --------------
    884 STA_SPECIFIC_EMP_TASK     COMPLETED
```

See Also:

Oracle Database PL/SQL Packages and Types Reference for complete reference information about the `DBMS_SQLTUNE.EXECUTE_TUNING_TASK` function

25.3.5 Monitoring a SQL Tuning Task

When you create a SQL tuning task in Cloud Control, no separate monitoring step is necessary. Cloud Control displays the status page automatically.

If you do not use Cloud Control, then you can monitor currently executing SQL tuning tasks by querying the data dictionary and dynamic performance views. The following table describes the relevant views.

Table 25-3 DBMS_SQLTUNE.EXECUTE_TUNING_TASK Parameters

View	Description
`USER_ADVISOR_TASKS`	Displays information about tasks owned by the current user. The view contains one row for each task. Each task has a name that is unique to the owner. Task names are just informational and no uniqueness is enforced within any other namespace.
`V$ADVISOR_PROGRESS`	Displays information about the progress of advisor execution.

Assumptions

This tutorial assumes the following:

- You tune as user `hr`, who has the `ADVISOR` privilege.
- You monitor the `STA_SPECIFIC_EMP_TASK` that you executed in "Executing a SQL Tuning Task".

To monitor a SQL tuning task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then determine whether the task is executing or completed.

For example, query the status of `STA_SPECIFIC_EMP_TASK` as follows:

```
SELECT STATUS
FROM   USER_ADVISOR_TASKS
WHERE  TASK_NAME = 'STA_SPECIFIC_EMP_TASK';
```

The following output shows that the task has completed:

```
STATUS
-----------
EXECUTING
```

2. Determine the progress of an executing task.

   The following example queries the status of the task with task ID `884`:

```
VARIABLE my_tid NUMBER;
EXEC :my_tid := 884
COL ADVISOR_NAME FORMAT a20
COL SOFAR FORMAT 999
COL TOTALWORK FORMAT 999

SELECT TASK_ID, ADVISOR_NAME, SOFAR, TOTALWORK,
       ROUND(SOFAR/TOTALWORK*100,2) "%_COMPLETE"
FROM   V$ADVISOR_PROGRESS
WHERE  TASK_ID = :my_tid;
```

   Sample output appears below:

```
   TASK_ID ADVISOR_NAME         SOFAR TOTALWORK %_COMPLETE
---------- -------------------- ----- --------- ----------
       884 SQL Tuning Advisor       1         2         50
```

See Also:

Oracle Database Reference to learn about the `V$ADVISOR_PROGRESS` view

25.3.6 Displaying the Results of a SQL Tuning Task

To report the results of a tuning task, use the `DBMS_SQLTUNE.REPORT_TUNING_TASK` function.

The report contains all the findings and recommendations of SQL Tuning Advisor. For each proposed recommendation, the report provides the rationale and benefit along with the SQL statements needed to implement the recommendation.

Assumptions

This tutorial assumes the following:

- You want to tune as user `hr`, who has the `ADVISOR` privilege.

- You want to access the report for the `STA_SPECIFIC_EMP_TASK` executed in "Executing a SQL Tuning Task".

To view the report for a SQL tuning task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then run the `DBMS_SQLTUNE.REPORT_TUNING_TASK` function.

   For example, you run the following statements:

   ```
   SET LONG 1000
   SET LONGCHUNKSIZE 1000
   SET LINESIZE 100
   SELECT DBMS_SQLTUNE.REPORT_TUNING_TASK( 'STA_SPECIFIC_EMP_TASK' )
   FROM   DUAL;
   ```

   Truncated sample output appears below:

   ```
   DBMS_SQLTUNE.REPORT_TUNING_TASK('STA_SPECIFIC_EMP_TASK')
   --------------------------------------------------------------------
   ----
   GENERAL INFORMATION SECTION
   --------------------------------------------------------------------
   ----
   Tuning Task Name   : STA_SPECIFIC_EMP_TASK
   Tuning Task Owner  : HR
   Workload Type      : Single SQL Statement
   Execution Count    : 11
   Current Execution  : EXEC_1057
   Execution Type     : TUNE SQL
   Scope              : COMPREHENSIVE
   Time Limit(seconds): 300
   Completion Status  : COMPLETED
   Started at         : 04/22/2012 07:35:49
   Completed at       : 04/22/2012 07:35:50

   --------------------------------------------------------------------
   ----
   Schema Name: HR
   SQL ID     : dg7nfaj0bdcvk
   SQL Text   : SELECT /*+ ORDERED */ * FROM employees e, locations l,
                departments d WHERE e.department_id = d.department_id
   AND
                l.location_id = d.location_id AND e.employee_id < :bnd
   Bind Variables :
    1 -  (NUMBER):100

   --------------------------------------------------------------------
   ----
   FINDINGS SECTION (4 findings)
   -----------------------------------------------
   ```

2. Interpret the results.

See Also:

- "Viewing Automatic SQL Tuning Reports Using the Command Line"
- *Oracle Database PL/SQL Packages and Types Reference* for complete reference information

25.4 The Automatic SQL Tuning Set

The Automatic SQL Tuning Set (ASTS) is a system-maintained record of SQL execution plans and SQL statement performance metrics seen by the database. It uses automatic SQL plan management to analyze this data and quickly repair SQL performance regressions.

ASTS resolves SQL statement performance regressions quickly and with minimal manual intervention. It mitigates the risks associated with database changes, system configuration changes, and upgrades. It is self-maintaining, with no configuration required. ASTS is enabled by default. Oracle recommends that you keep it enabled.

Note:

Do not disable ASTS task if you are using Automatic Indexing (which relies on ASTS). Automatic SQL plan management uses ASTS as a source of alternative SQL execution plans.

Over time, the ASTS accumulates information on all SQL statements executed on the system (including details of all execution plans).

ASTS and AWR

ASTS is complementary to AWR. Both are core manageability infrastructure components of Oracle Database.

Like AWR, the ASTS is a historic record of SQL execution plans and SQL statement performance metrics. It differs from the Automatic Workload Repository (AWR) because it is not limited to statements that consume significant system resources. Over time, ASTS will include examples of all queries seen on the system. However, it does impose a limit on the collection of non-reusable statements such as ad-hoc queries or statements that use literals instead of bind variables.

ASTS is particularly useful for diagnosing and potentially correcting SQL performance regressions in situations where the regression is caused by a plan change. In cases like this, the better plan is unlikely to be available in AWR, but it will be available in ASTS. This is significant because, for example, SQL plan management can be used to locate, test, and enforce better SQL execution plans contained in ASTS. Automatic SQL plan management implements this entire workflow without manual intervention.

Example 25-5 How to View Captured SQL statements in ASTS

```
Select sql_text
from dba_sqlset_Statements
where sqlset_name = 'SYS_AUTO_STS';
```

Example 25-6 Enabling the ASTS Task

```
Begin
   DBMS_Auto_Task_Admin.Enable(
       Client_Name => 'Auto STS Capture Task',
       Operation => NULL,
       Window_name => NULL);
End;
/
```

Example 25-7 Disabling the ASTS Task

```
Begin
   DBMS_Auto_Task_Admin.Disable(
       Client_Name => 'Auto STS Capture Task',
       Operation => NULL,
       Window_name => NULL);
End;
/
```

Example 25-8 Viewing Task Status

```
Select Task_Name, Enabled
From   DBA_AutoTask_Schedule_Control
Where  Task_Name = 'Auto STS Capture Task';
```

See Also:

The Database Licensing Information User Manual for availability details.

The *Oracle Database Reference* for the following views.

- DBA_AUTOTASK_SCHEDULE_CONTROL
- DBA_AUTOTASK_SETTINGS

26

Optimizing Access Paths with SQL Access Advisor

SQL Access Advisor is diagnostic software that identifies and helps resolve SQL performance problems by recommending indexes, materialized views, materialized view logs, or partitions to create, drop, or retain.

26.1 About SQL Access Advisor

SQL Access Advisor accepts input from several sources, including SQL tuning sets, and then issues recommendations.

Note:

Data visibility and privilege requirements may differ when using SQL Access Advisor with pluggable databases.

See Also:

Oracle Database Administrator’s Guide for a table that summarizes how manageability features work in a container database (CDB)

26.1.1 Purpose of SQL Access Advisor

SQL Access Advisor recommends the proper set of materialized views, materialized view logs, partitions, and indexes for a specified workload.

Materialized views, partitions, and indexes are essential when tuning a database to achieve optimum performance for complex, data-intensive queries. SQL Access Advisor takes an actual workload as input, or derives a hypothetical workload from a schema. The advisor then recommends access structures for faster execution path. The advisor provides the following advantages:

- Does not require you to have expert knowledge
- Makes decisions based on rules that reside in the optimizer
- Covers all aspects of SQL access in a single advisor
- Provides simple, user-friendly GUI wizards in Cloud Control
- Generates scripts for implementation of recommendations

See Also:

- *Oracle Database 2 Day + Performance Tuning Guide* to learn how to use SQL Access Advisor with Cloud Control
- *Oracle Database Administrator's Guide* to learn more about automated indexing
- *Oracle Database Licensing Information User Manual* for details on whether automated indexing is supported for different editions and services

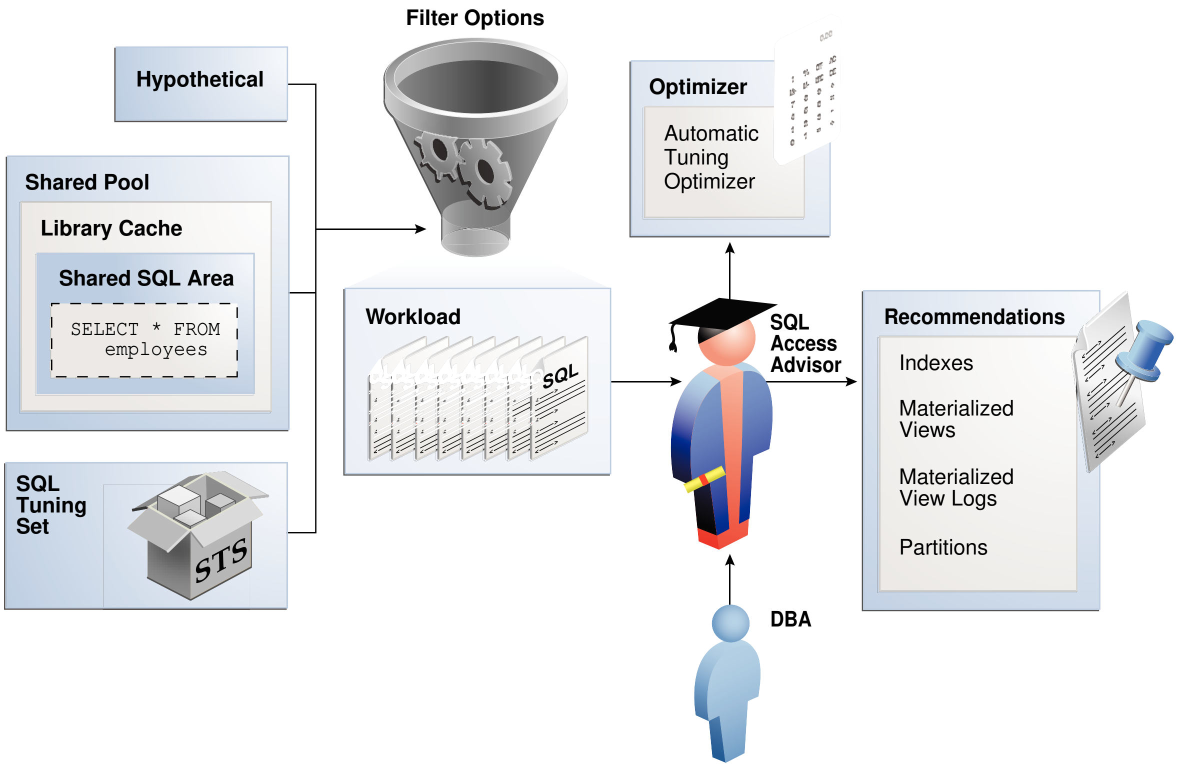
26.1.2 SQL Access Advisor Architecture

Automatic Tuning Optimizer is the central tool used by SQL Access Advisor.

The advisor can receive SQL statements as input from the sources shown in Figure 26-1, analyze these statements using the optimizer, and then make recommendations.

Figure 26-1 shows the basic architecture of SQL Access Advisor.

Figure 26-1 SQL Access Advisor Architecture



See Also:

"About Automatic Tuning Optimizer"

26.1.2.1 Input to SQL Access Advisor

SQL Access Advisor requires a workload, which consists of one or more SQL statements, plus statistics and attributes that fully describe each statement.

A full workload contains all SQL statements from a target business application. A partial workload contains a subset of SQL statements.

As shown in Figure 26-1, SQL Access Advisor input can come from the following sources:

- Shared SQL area

  The database uses the shared SQL area to analyze recent SQL statements that are currently in `V$SQL`.

- SQL tuning set

  A SQL tuning set (STS) is a database object that stores SQL statements along with their execution context. When a set of SQL statements serve as input, the database must first construct and use an STS.

  Note:

  For best results, provide a workload as a SQL tuning set. The `DBMS_SQLTUNE` package provides helper functions that can create SQL tuning sets from common workload sources, such as the SQL cache, a user-defined workload stored in a table, and a hypothetical workload.

- Hypothetical workload

  You can create a hypothetical workload from a schema by analyzing dimensions and constraints. This option is useful when you are initially designing your application.

See Also:

- "About SQL Tuning Sets"
- *Oracle Database Concepts* to learn about the shared SQL area

26.1.2.2 Filter Options for SQL Access Advisor

You can apply a filter to a workload to restrict what is analyzed.

For example, specify that the advisor look at only the 30 most resource-intensive statements in the workload, based on optimizer cost. This restriction can generate different sets of recommendations based on different workload scenarios.

SQL Access Advisor parameters control the recommendation process and customization of the workload. These parameters control various aspects of the process, such as the type of recommendation required and the naming conventions for what it recommends.

To set these parameters, use the `DBMS_ADVISOR.SET_TASK_PARAMETER` procedure. Parameters are persistent in that they remain set for the life span of the task. When a parameter value is set using `DBMS_ADVISOR.SET_TASK_PARAMETER`, the value does not change until you make another call to this procedure.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_ADVISOR.SET_TASK_PARAMETER` procedure

26.1.2.3 SQL Access Advisor Recommendations

A task recommendation can range from a simple to a complex solution.

The advisor can recommend that you create database objects such as the following:

- Indexes

  SQL Access Advisor index recommendations include bitmap, function-based, and B-tree indexes. A bitmap index offers a reduced response time for many types of ad hoc queries and reduced storage requirements compared to other indexing techniques. B-tree indexes are most commonly used in a data warehouse to index unique or near-unique keys. SQL Access Advisor materialized view recommendations include fast refreshable and full refreshable materialized views, for either general rewrite or exact text match rewrite.
- Materialized views

  SQL Access Advisor, using the `TUNE_MVIEW` procedure, also recommends how to optimize materialized views so that they can be fast refreshable and take advantage of general query rewrite.
- Materialized view logs

  A materialized view log is a table at the materialized view's primary site or primary materialized view site that records all DML changes to the primary table or primary materialized view. A fast refresh of a materialized view is possible only if the materialized view's primary has a materialized view log.
- Partitions

  SQL Access Advisor can recommend partitioning on an existing unpartitioned base table to improve performance. Furthermore, it may recommend new indexes and materialized views that are themselves partitioned.

  While creating new partitioned indexes and materialized view is no different from the unpartitioned case, partition existing base tables with care. This is especially true when indexes, views, constraints, or triggers are defined on the table.

To make recommendations, SQL Access Advisor relies on structural statistics about table and index cardinalities of dimension level columns, `JOIN KEY` columns, and fact table key columns. You can gather exact or estimated statistics with the `DBMS_STATS` package.

Because gathering statistics is time-consuming and full statistical accuracy is not required, it is usually preferable to estimate statistics. Without gathering statistics on a specified table, queries referencing this table are marked as invalid in the workload, resulting in no recommendations for these queries. It is also recommended that all existing indexes and materialized views have been analyzed.

See Also:

- "About Manual Statistics Collection with DBMS_STATS"
- *Oracle Database Data Warehousing Guide* to learn more about materialized views
- *Oracle Database VLDB and Partitioning Guide* to learn more about partitions

26.1.2.4 SQL Access Advisor Actions

In general, each recommendation provides a benefit for a set of queries.

All individual actions in a recommendation must be implemented together to achieve the full benefit. Recommendations can share actions.

For example, a `CREATE INDEX` statement could provide a benefit for several queries, but some queries might benefit from an additional `CREATE MATERIALIZED VIEW` statement. In that case, the advisor would generate two recommendations: one for the set of queries that require only the index, and another one for the set of queries that require both the index and the materialized view.

26.1.2.4.1 Types of Actions

SQL Access Advisor makes several different types of recommendations.

Recommendations include the following types of actions:

- `PARTITION BASE TABLE`

  This action partitions an existing unpartitioned base table.

- `CREATE|DROP|RETAIN {MATERIALIZED VIEW|MATERIALIZED VIEW LOG|INDEX}`

  The `CREATE` actions corresponds to new access structures. `RETAIN` recommends keeping existing access structures. SQL Access Advisor only recommends `DROP` when the `WORKLOAD_SCOPE` parameter is set to `FULL`.

- `GATHER STATS`

  This action generates a call to a `DBMS_STATS` procedure to gather statistics on a newly generated access structure.

Multiple recommendations may refer to the same action. However, when generating a script for the recommendation, you only see each action once.

See Also:

- "About Manual Statistics Collection with DBMS_STATS"
- "Viewing SQL Access Advisor Task Results" to learn how to view actions and recommendations

26.1.2.4.2 Guidelines for Interpreting Partitioning Recommendations

When SQL Access Advisor determines that partitioning a base table would improve performance, the advisor adds a partition action to every recommendation containing a query referencing the table. In this way, index and materialized view recommendations are implemented on the correctly partitioned tables.

SQL Access Advisor may recommend partitioning an existing nonpartitioned base table. When the advisor implementation script contains partition recommendations, note the following issues:

- Partitioning an existing table is a complex and extensive operation, which may take considerably longer than implementing a new index or materialized view. Sufficient time should be reserved for implementing this recommendation.
- While index and materialized view recommendations are easy to reverse by deleting the index or view, a table, after being partitioned, cannot easily be restored to its original state. Therefore, ensure that you back up the database before executing a script containing partition recommendations.
- While repartitioning a base table, SQL Access Advisor scripts make a temporary copy of the original table, which occupies the same amount of space as the original table. Therefore, the repartitioning process requires sufficient free disk space for another copy of the largest table to be repartitioned. Ensure that such space is available before running the implementation script.

  The partition implementation script attempts to migrate dependent objects such as indexes, materialized views, and constraints. However, some object cannot be automatically migrated. For example, PL/SQL stored procedures defined against a repartitioned base table typically become invalid and must be recompiled.
- If you decide not to implement a partition recommendation, then all other recommendations on the same table in the same script (such as `CREATE INDEX` and `CREATE MATERIALIZED VIEW` recommendations) depend on the partitioning recommendation. To obtain accurate recommendations, do not simply remove the partition recommendation from the script. Rather, rerun the advisor with partitioning disabled, for example, by setting parameter `ANALYSIS_SCOPE` to a value that does not include the keyword `TABLE`.

See Also:

- *Oracle Database SQL Language Reference* for `CREATE DIRECTORY` syntax
- *Oracle Database PL/SQL Packages and Types Reference* for detailed information about the `DBMS_ADVISOR.GET_TASK_SCRIPT` function

26.1.2.5 SQL Access Advisor Repository

Information required and generated by SQL Access Advisor resides in the Advisor repository, which is in the data dictionary.

The SQL Access Advisor repository has the following benefits:

- Collects a complete workload for SQL Access Advisor
- Supports historical data
- Is managed by the database

26.1.3 User Interfaces for SQL Access Advisor

Oracle recommends that you use SQL Access Advisor through its GUI wizard, which is available in Cloud Control.

You can also invoke SQL Access Advisor through the `DBMS_ADVISOR` package. This chapter explains how to use the API.

See Also:

- *Oracle Database 2 Day + Performance Tuning Guide* explains how to use the SQL Access Advisor wizard.
- See *Oracle Database PL/SQL Packages and Types Reference* for complete semantics and syntax.

26.1.3.1 Accessing the SQL Access Advisor: Initial Options Page Using Cloud Control

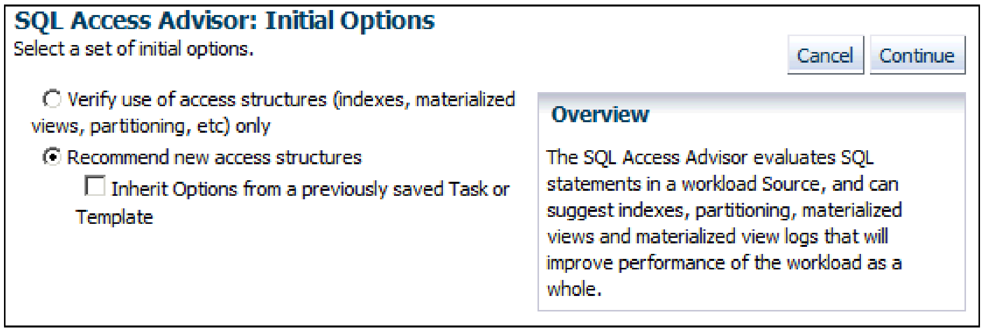
The SQL Access Advisor: Initial Options page in Cloud Control is the starting page for a wizard that guides you through the process of obtaining recommendations.

To access the SQL Access Advisor: Initial Options page:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Performance** menu, select **SQL**, then **SQL Access Advisor**.

   The SQL Access Advisor: Initial Options page appears., shown in Figure 26-2.

Figure 26-2 SQL Access Advisor: Initial Options



SQL Access Advisor: Initial Options

Select a set of initial options.

( ) Verify use of access structures (indexes, materialized views, partitioning, etc) only

[✓ Recommend new access structures]

☐ Inherit Options from a previously saved Task or Template

Overview

The SQL Access Advisor evaluates SQL statements in a workload Source, and can suggest indexes, partitioning, materialized views and materialized view logs that will improve performance of the workload as a whole.

You can perform most SQL plan management tasks in this page or in pages accessed through this page.

See Also:

- Cloud Control context-sensitive online help to learn about the options on the SQL Access Advisor: Initial Options page
- *Oracle Database 2 Day + Performance Tuning Guide* to learn how to configure and run SQL Tuning Advisor using Cloud Control

26.1.3.2 Command-Line Interface to SQL Tuning Sets

On the command line, you can use the `DBMS_ADVISOR` package to manage SQL Tuning Advisor.

The `DBMS_ADVISOR` package consists of a collection of analysis and advisory functions and procedures callable from any PL/SQL program. You must have the `ADVISOR` privilege to use `DBMS_ADVISOR`.

See Also:

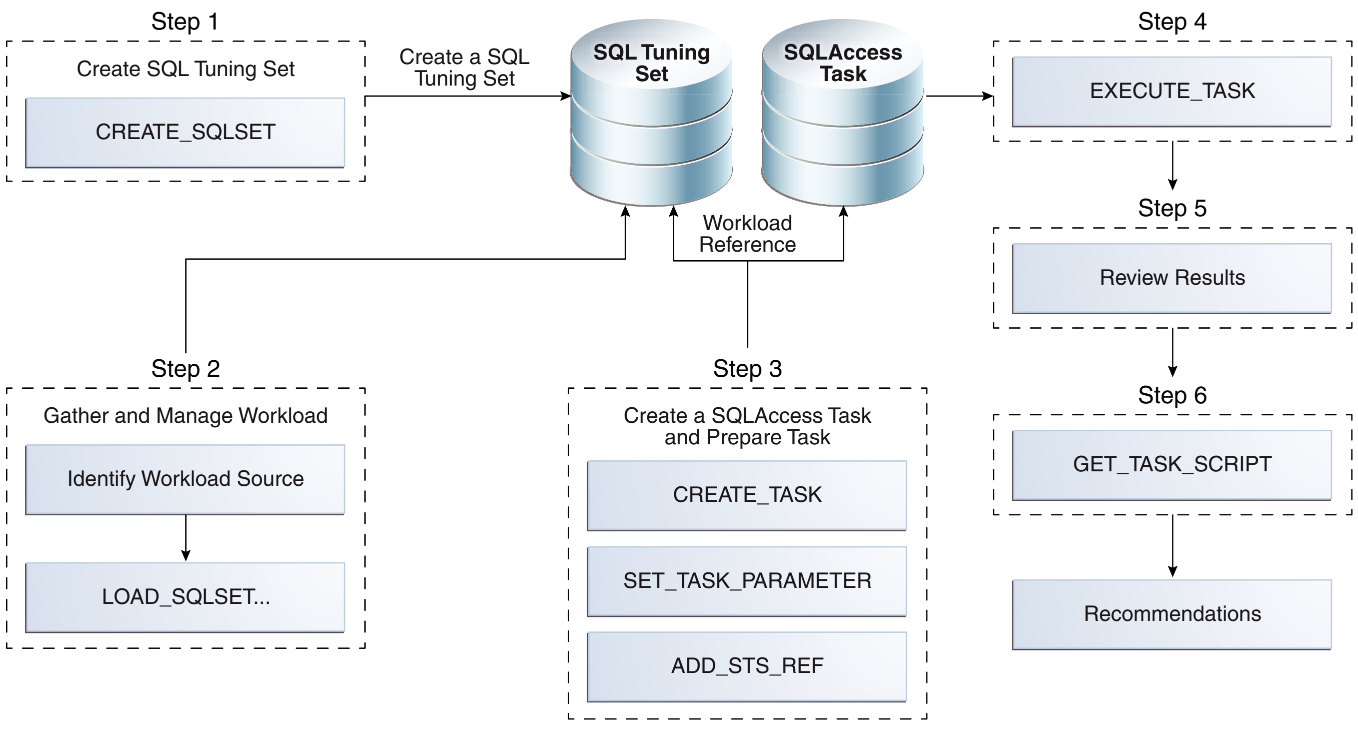
Oracle Database PL/SQL Packages and Types Reference to learn about `DBMS_ADVISOR`

26.2 Using SQL Access Advisor: Basic Tasks

Basic tasks include creating an STS, loading it, creating a SQL Access Advisor task, and then executing the task.

The following graphic shows the basic workflow for SQL Access Advisor.

Figure 26-3 Using SQL Access Advisor



Typically, you use SQL Access Advisor by performing the following steps:

1. Create a SQL tuning set

   The input workload source for SQL Access Advisor is a SQL tuning set (STS). Use `DBMS_SQLTUNE.CREATE_SQLSET` or `DBMS_SQLSET.CREATE_SQLSET` to create a SQL tuning set.

   "Creating a SQL Tuning Set as Input for SQL Access Advisor" describes this task.

2. Load the SQL tuning set

   SQL Access Advisor performs best when a workload based on actual usage is available. Use `DBMS_SQLTUNE.LOAD_SQLSET` or `DBMS_SQLSET.LOAD_SQLSET` to populate the SQL tuning set with your workload.

   "Populating a SQL Tuning Set with a User-Defined Workload" describes this task.

3. Create and configure a task

   In the task, you define what SQL Access Advisor must analyze and the location of the analysis results. Create a task using the `DBMS_ADVISOR.CREATE_TASK` procedure. You can then define parameters for the task using the `SET_TASK_PARAMETER` procedure, and then link the task to an STS by using the `DBMS_ADVISOR.ADD_STS_REF` procedure.

   "Creating and Configuring a SQL Access Advisor Task" describes this task.

4. Execute the task

   Run the `DBMS_ADVISOR.EXECUTE_TASK` procedure to generate recommendations. Each recommendation specifies one or more actions. For example, a recommendation could be to create several materialized view logs, create a materialized view, and then analyze it to gather statistics.

"Executing a SQL Access Advisor Task" describes this task.

5. View the recommendations

   You can view the recommendations by querying data dictionary views.

   "Viewing SQL Access Advisor Task Results" describes this task.

6. Optionally, generate and execute a SQL script that implements the recommendations.

   "Generating and Executing a Task Script" that describes this task.

26.2.1 Creating a SQL Tuning Set as Input for SQL Access Advisor

The input workload source for SQL Access Advisor is an STS.

Because an STS is stored as a separate entity, multiple advisor tasks can share it. Create an STS with the `DBMS_SQLTUNE.CREATE_SQLSET` or `DBMS_SQLSET.CREATE_SQLSET` procedure.

After an advisor task has referenced an STS, you cannot delete or modify the STS until all advisor tasks have removed their dependency on it. A workload reference is removed when a parent advisor task is deleted, or when you manually remove the workload reference from the advisor task.

Prerequisites

The user creating the STS must have been granted the `ADMINISTER SQL TUNING SET` privilege. To run SQL Access Advisor on SQL tuning sets owned by other users, the user must have the `ADMINISTER ANY SQL TUNING SET` privilege.

Assumptions

This tutorial assumes the following:

- You want to create an STS named `MY_STS_WORKLOAD`.
- You want to use this STS as input for a workload derived from the `sh` schema.
- You use `DBMS_SQLTUNE` rather than `DBMS_SQLSET`.

To create an STS :

1. In SQL*Plus, log in to the database as user `sh`.
2. Set SQL*Plus variables.

   For example, enter the following commands:

```
SET SERVEROUTPUT ON;
VARIABLE task_id NUMBER;
VARIABLE task_name VARCHAR2(255);
VARIABLE workload_name VARCHAR2(255);
```

3. Create the SQL tuning set.

For example, assign a value to the `workload_name` variable and create the STS as follows:

```
EXECUTE :workload_name := 'MY_STS_WORKLOAD';
EXECUTE DBMS_SQLTUNE.CREATE_SQLSET(:workload_name, 'test purpose');
```

See Also:

- "About SQL Tuning Sets"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about `CREATE_SQLSET`

26.2.2 Populating a SQL Tuning Set with a User-Defined Workload

A workload consists of one or more SQL statements, plus statistics and attributes that fully describe each statement.

A full workload contains all SQL statements from a target business application. A partial workload contains a subset of SQL statements. The difference is that for full workloads SQL Access Advisor may recommend dropping unused materialized views and indexes.

You cannot use SQL Access Advisor without a workload. SQL Access Advisor ranks the entries according to a specific statistic, business importance, or combination of the two, which enables the advisor to process the most important SQL statements first.

SQL Access Advisor performs best with a workload based on actual usage. You can store multiple workloads in the form of SQL tuning sets, so that you can view the different uses of a real-world data warehousing or OLTP environment over a long period and across the life cycle of database instance startup and shutdown.

The following table describes procedures that you can use to populate an STS with a user-defined workload.

Table 26-1 Procedures for Loading an STS

Procedure	Description	To Learn More
`DBMS_SQLTUNE.LOAD_SQLSET` or `DBMS_SQLSET.LOAD_SQLSET`	Populates the SQL tuning set with a set of selected SQL. You can call the procedure multiple times to add new SQL statements or replace attributes of existing statements.	*Oracle Database PL/SQL Packages and Types Reference*
`DBMS_ADVISOR.COPY_SQLWKLD_TO_STS`	Copies SQL workload data to a user-designated SQL tuning set. The user must have the required SQL tuning set privileges and the required `ADVISOR` privilege.	*Oracle Database PL/SQL Packages and Types Reference*

Assumptions

This tutorial assumes that you want to do the following:

- Create a table named `sh.user_workload` to store information about SQL statements
- Load the `sh.user_workload` table with information about three queries of tables in the `sh` schema
- Populate the STS created in "Creating a SQL Tuning Set as Input for SQL Access Advisor" with the workload contained in `sh.user_workload`
- Use `DBMS_SQLTUNE.LOAD_SQLSET` instead of `DBMS_SQLSET.LOAD_SQLSET`

To populate an STS with a user-defined workload:

1. In SQL*Plus, log in to the database as user `sh`.
2. Create the `user_workload` table.

   For example, enter the following commands:

```
DROP TABLE user_workload;
CREATE TABLE user_workload
(
  username              varchar2(128),  /* User who executes
statement */
  module                varchar2(64),       /* Application module
name */
  action                varchar2(64),       /* Application action
name */
  elapsed_time          number,              /* Elapsed time for
query */
  cpu_time              number,                  /* CPU time for
query */
  buffer_gets           number,       /* Buffer gets consumed by
query */
  disk_reads            number,        /* Disk reads consumed by
query */
  rows_processed        number,        /* # of rows processed by
query */
  executions            number,           /* # of times query
executed */
  optimizer_cost        number,            /* Optimizer cost for
query */
  priority              number,            /* User-priority (1,2 or
3) */
  last_execution_date   date,              /* Last time query
executed */
  stat_period           number,         /* Window exec time in
seconds */
  sql_text              clob                          /* Full SQL
Text */
);
```

3. Load the `user_workload` table with information about queries.

For example, execute the following statements:

```
-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
'SELECT   t.week_ending_day, p.prod_subcategory,
          SUM(s.amount_sold) AS dollars, s.channel_id, s.promo_id
 FROM     sales s, times t, products p
 WHERE    s.time_id = t.time_id
 AND      s.prod_id = p.prod_id
 AND      s.prod_id > 10
 AND      s.prod_id < 50
 GROUP BY t.week_ending_day, p.prod_subcategory, s.channel_id,
s.promo_id')
/

-- aggregation with selection
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
 'SELECT   t.calendar_month_desc, SUM(s.amount_sold) AS dollars
  FROM     sales s , times t
  WHERE    s.time_id = t.time_id
  AND      s.time_id BETWEEN TO_DATE(''01-JAN-2000'', ''DD-MON-YYYY'')
  AND      TO_DATE(''01-JUL-2000'', ''DD-MON-YYYY'')
  GROUP BY t.calendar_month_desc')
/

-- order by
INSERT INTO user_workload (username, module, action, priority, sql_text)
VALUES ('SH', 'Example1', 'Action', 2,
 'SELECT   c.country_id, c.cust_city, c.cust_last_name
  FROM     customers c
  WHERE    c.country_id IN (52790, 52789)
  ORDER BY c.country_id, c.cust_city, c.cust_last_name')
/
COMMIT;
```

4. Execute a PL/SQL program that fills a cursor with rows from the `user_workload` table, and then loads the contents of this cursor into the STS named `MY_STS_WORKLOAD`.

   For example, execute the following PL/SQL program:

```
DECLARE
  sqlset_cur DBMS_SQLTUNE.SQLSET_CURSOR;
BEGIN
  OPEN sqlset_cur FOR
    SELECT SQLSET_ROW(null,null, SQL_TEXT, null, null, 'SH', module,
                     'Action', 0, 0, 0, 0, 0, 1, 0, 1, 0, 0, null, 2, 3,
                     sysdate, 0, 0, null, 0, null, null)
    FROM USER_WORKLOAD;
  DBMS_SQLTUNE.LOAD_SQLSET('MY_STS_WORKLOAD', sqlset_cur);
END;
/
```

26.2.3 Creating and Configuring a SQL Access Advisor Task

Use the `DBMS_ADVISOR.CREATE_TASK` procedure to create a SQL Access Advisor task.

In the SQL Access Advisor task, you define what the advisor must analyze and the location of the results. You can create multiple tasks, each with its own specialization. All are based on the same Advisor task model and share the same repository.

Configuring the task involves the following steps:

- Defining task parameters

  At the time the recommendations are generated, you can apply a filter to the workload to restrict what is analyzed. This restriction provides the ability to generate different sets of recommendations based on different workload scenarios.

  SQL Access Advisor parameters control the recommendation process and customization of the workload. These parameters control various aspects of the process, such as the type of recommendation required and the naming conventions for what it recommends.

  If parameters are not defined, then the database uses the defaults. You can set task parameters by using the `DBMS_ADVISOR.SET_TASK_PARAMETER` procedure. Parameters are persistent in that they remain set for the life span of the task. When a parameter value is set using `SET_TASK_PARAMETER`, it does not change until you make another call to this procedure.

- Linking the task to the workload

  Because the workload is independent, you must link it to a task using the `DBMS_ADVISOR.ADD_STS_REF` procedure. After this link has been established, you cannot delete or modify the workload until all advisor tasks have removed their dependency on the workload. A workload reference is removed when a user deletes a parent advisor task or manually removes the workload reference from the task by using the `DBMS_ADVISOR.DELETE_STS_REF` procedure.

Prerequisites

The user creating the task must have been granted the `ADVISOR` privilege.

Assumptions

This tutorial assumes the following:

- You want to create a task named `MYTASK`.
- You want to use this task to analyze the workload that you defined in "Populating a SQL Tuning Set with a User-Defined Workload".
- You want to terminate the task if it takes longer than 30 minutes to execute.
- You want to SQL Access Advisor to only consider indexes.

To create and configure a SQL Access Advisor task:

1. Connect SQL*Plus to the database as user `sh`, and then create the task.

For example, enter the following commands:

```
EXEC :task_name := 'MYTASK';
EXEC DBMS_ADVISOR.CREATE_TASK('SQL Access Advisor', :task_id, :task_name);
```

2. Set task parameters.

   For example, execute the following statements:

```
EXEC DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'TIME_LIMIT', 30);
EXEC DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'ANALYSIS_SCOPE', 'ALL');
```

3. Link the task to the workload.

   For example, execute the following statement:

```
EXECUTE DBMS_ADVISOR.ADD_STS_REF(:task_name, 'SH', :workload_name);
```

See Also:

- "Categories for SQL Access Advisor Task Parameters"
- "Deleting SQL Access Advisor Tasks"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_ADVISOR.CREATE_TASK`, `DBMS_ADVISOR.SET_TASK_PARAMETER`, and `DBMS_ADVISOR.ADD_STS_REF` procedures

26.2.4 Executing a SQL Access Advisor Task

The `DBMS_ADVISOR.EXECUTE_TASK` procedure performs SQL Access Advisor analysis or evaluation for the specified task.

Task execution is a synchronous operation, so the database does not return control to the user until the operation has completed, or the database detects a user interrupt. After the return or execution of the task, you can check the `DBA_ADVISOR_LOG` table for the execution status.

Running `EXECUTE_TASK` generates recommendations. A recommendation includes one or more actions, such as creating a materialized view log or a materialized view.

Prerequisites

When processing a workload, SQL Access Advisor attempts to validate each statement to identify table and column references. The database achieves validation by processing each statement as if it were being executed by the statement's original user.

If the user does not have `SELECT` privileges to a particular table, then SQL Access Advisor bypasses the statement referencing the table. This behavior can cause many statements to be excluded from analysis. If SQL Access Advisor excludes all statements in a workload, then the workload is invalid. SQL Access Advisor returns the following message:

```
QSM-00774, there are no SQL statements to process for task TASK_NAME
```

To avoid missing critical workload queries, the current database user must have SELECT privileges on the tables targeted for materialized view analysis. For these tables, these SELECT privileges cannot be obtained through a role.

Assumptions

This tutorial assumes that you want to execute the task you configured in "Creating and Configuring a SQL Access Advisor Task".

To create and configure a SQL Access Advisor task:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Execute the task.

   For example, execute the following statement:

   ```
   EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:task_name);
   ```

3. Optionally, query USER_ADVISOR_LOG to check the status of the task.

   For example, execute the following statements (sample output included):

   ```
   COL TASK_ID FORMAT 999
   COL TASK_NAME FORMAT a25
   COL STATUS_MESSAGE FORMAT a25

   SELECT TASK_ID, TASK_NAME, STATUS, STATUS_MESSAGE
   FROM   USER_ADVISOR_LOG;

   TASK_ID TASK_NAME                  STATUS      STATUS_MESSAGE
   ------- ------------------------- -----------
   -------------------------
       103 MYTASK                     COMPLETED   Access advisor
   execution
                                                  completed
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the EXECUTE_TASK procedure and its parameters

26.2.5 Viewing SQL Access Advisor Task Results

You can view each recommendation generated by SQL Access Advisor using several data dictionary views.

The views are summarized in Table 26-2. However, it is easier to use the DBMS_ADVISOR.GET_TASK_SCRIPT procedure or Cloud Control, which graphically displays the recommendations and provides hyperlinks to quickly see which SQL statements benefit from a recommendation.

Each recommendation produced by SQL Access Advisor is linked to the SQL statement it benefits. Each recommendation corresponds to one or more actions. Each action has one or more attributes.

Each action has attributes pertaining to the access structure properties. The name and tablespace for each applicable access structure are in the `ATTR1` and `ATTR2` columns of `USER_ADVISOR_ATTRIBUTES`. The space occupied by each new access structure is in the `NUM_ATTR1` column. Other attributes are different for each action.

Table 26-2 Views Showing Task Results

Data Dictionary View (DBA, USER)	Description
`DBA_ADVISOR_TASKS`	Displays information about advisor tasks. To see SQL Access Advisor tasks, select where `ADVISOR_NAME = 'SQL Access Advisor'`.
`DBA_ADVISOR_RECOMMENDATIONS`	Displays the results of an analysis of all recommendations in the database. A recommendation can have multiple actions associated with it. The `DBA_ADVISOR_ACTIONS` view describe the actions. A recommendation also points to a set of rationales that present a justification/reasoning for that recommendation. The `DBA_ADVISOR_RATIONALE` view describes the rationales.
`DBA_ADVISOR_ACTIONS`	Displays information about the actions associated with all recommendations in the database. Each action is specified by the `COMMAND` and `ATTR1` through `ATTR6` columns. Each command defines how to use the attribute columns.
`DBA_ADVISOR_RATIONALE`	Displays information about the rationales for all recommendations in the database.
`DBA_ADVISOR_SQLA_WK_STMTS`	Displays information about all workload objects in the database after a SQL Access Advisor analysis. The precost and postcost numbers are in terms of the estimated optimizer cost (shown in `EXPLAIN PLAN`) without and with the recommended access structure.

Assumptions

This tutorial assumes that you want to view results of the task you executed in "Executing a SQL Access Advisor Task".

To view the results of a SQL Access Advisor task:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the advisor recommendations.

   For example, execute the following statements (sample output included):

```
VARIABLE workload_name VARCHAR2(255);
VARIABLE task_name VARCHAR2(255);
EXECUTE :task_name := 'MYTASK';
EXECUTE :workload_name := 'MY_STS_WORKLOAD';
```

```
SELECT REC_ID, RANK, BENEFIT
FROM   USER_ADVISOR_RECOMMENDATIONS
WHERE  TASK_NAME = :task_name
ORDER BY RANK;

    REC_ID       RANK    BENEFIT
---------- ---------- ----------
         1          1        236
         2          2        356
```

The preceding output shows the recommendations (`rec_id`) produced by an SQL Access Advisor run, with their rank and total benefit. The rank is a measure of the importance of the queries that the recommendation helps. The benefit is the total improvement in execution cost (in terms of optimizer cost) of all queries using the recommendation.

2. Identify which query benefits from which recommendation.

   For example, execute the following query of `USER_ADVISOR_SQLA_WK_STMTS` (sample output included):

```
SELECT SQL_ID, REC_ID, PRECOST, POSTCOST,
       (PRECOST-POSTCOST)*100/PRECOST AS PERCENT_BENEFIT
FROM   USER_ADVISOR_SQLA_WK_STMTS
WHERE  TASK_NAME = :task_name
AND    WORKLOAD_NAME = :workload_name
ORDER BY percent_benefit DESC;

SQL_ID            REC_ID    PRECOST   POSTCOST PERCENT_BENEFIT
------------- ---------- ---------- ---------- ---------------
fn4bsxdm98w3u          2        578        222      61.5916955
29bbju72rv3t2          1       5750       5514      4.10434783
133ym38r6gbar          0        772        772               0
```

   The precost and postcost numbers are in terms of the estimated optimizer cost (shown in `EXPLAIN PLAN`) both without and with the recommended access structure changes.

3. Display the number of distinct actions for this set of recommendations.

   For example, use the following query (sample output included):

```
SELECT 'Action Count', COUNT(DISTINCT action_id) cnt
FROM   USER_ADVISOR_ACTIONS
WHERE  TASK_NAME = :task_name;

'ACTIONCOUNT        CNT
------------ ----------
Action Count          4
```

4. Display the actions for this set of recommendations.

   For example, use the following query (sample output included):

```
SELECT REC_ID, ACTION_ID, SUBSTR(COMMAND,1,30) AS command
FROM   USER_ADVISOR_ACTIONS
```

```
WHERE  TASK_NAME = :task_name
ORDER BY rec_id, action_id;

    REC_ID  ACTION_ID COMMAND
---------- ---------- ------------------------------
         1          1 PARTITION TABLE
         1          2 RETAIN INDEX
         2          1 PARTITION TABLE
         2          3 RETAIN INDEX
         2          4 RETAIN INDEX
```

5. Display attributes of the recommendations.

   For example, create the following PL/SQL procedure `show_recm`, and then execute it to see attributes of the actions:

```
CREATE OR REPLACE PROCEDURE show_recm (in_task_name IN VARCHAR2) IS
CURSOR curs IS
  SELECT DISTINCT action_id, command, attr1, attr2, attr3, attr4
  FROM user_advisor_actions
  WHERE task_name = in_task_name
  ORDER BY action_id;
  v_action        number;
  v_command     VARCHAR2(32);
  v_attr1       VARCHAR2(4000);
  v_attr2       VARCHAR2(4000);
  v_attr3       VARCHAR2(4000);
  v_attr4       VARCHAR2(4000);
  v_attr5       VARCHAR2(4000);
BEGIN
  OPEN curs;
  DBMS_OUTPUT.PUT_LINE('=========================================');
  DBMS_OUTPUT.PUT_LINE('Task_name = ' || in_task_name);
  LOOP
     FETCH curs INTO
       v_action, v_command, v_attr1, v_attr2, v_attr3, v_attr4 ;
   EXIT when curs%NOTFOUND;
   DBMS_OUTPUT.PUT_LINE('Action ID: ' || v_action);
   DBMS_OUTPUT.PUT_LINE('Command : ' || v_command);
   DBMS_OUTPUT.PUT_LINE('Attr1 (name)      : ' || SUBSTR(v_attr1,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr2 (tablespace): ' || SUBSTR(v_attr2,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr3             : ' || SUBSTR(v_attr3,1,30));
   DBMS_OUTPUT.PUT_LINE('Attr4             : ' || v_attr4);
   DBMS_OUTPUT.PUT_LINE('Attr5             : ' || v_attr5);
   DBMS_OUTPUT.PUT_LINE('----------------------------------------');
   END LOOP;
   CLOSE curs;
   DBMS_OUTPUT.PUT_LINE('=========END RECOMMENDATIONS============');
END show_recm;
/

SET SERVEROUTPUT ON SIZE 99999
EXECUTE show_recm(:task_name);
```

The following output shows attributes of actions in the recommendations:

```
=========================================
Task_name = MYTASK
Action ID: 1
Command : PARTITION TABLE
Attr1 (name)       : "SH"."SALES"
Attr2 (tablespace):
Attr3              : ("TIME_ID")
Attr4              : INTERVAL
Attr5              :
----------------------------------------
Action ID: 2
Command : RETAIN INDEX
Attr1 (name)       : "SH"."PRODUCTS_PK"
Attr2 (tablespace):
Attr3              : "SH"."PRODUCTS"
Attr4              : BTREE
Attr5              :
----------------------------------------
Action ID: 3
Command : RETAIN INDEX
Attr1 (name)       : "SH"."TIMES_PK"
Attr2 (tablespace):
Attr3              : "SH"."TIMES"
Attr4              : BTREE
Attr5              :
----------------------------------------
Action ID: 4
Command : RETAIN INDEX
Attr1 (name)       : "SH"."SALES_TIME_BIX"
Attr2 (tablespace):
Attr3              : "SH"."SALES"
Attr4              : BITMAP
Attr5              :
----------------------------------------
=========END RECOMMENDATIONS============
```

See Also:

- "Action Attributes in the DBA_ADVISOR_ACTIONS View"
- *Oracle Database PL/SQL Packages and Types Reference* for details regarding `Attr5` and `Attr6`

26.2.6 Generating and Executing a Task Script

You can use the procedure `DBMS_ADVISOR.GET_TASK_SCRIPT` to create a script of the SQL statements for the SQL Access Advisor recommendations. The script is an executable SQL file that can contain `DROP`, `CREATE`, and `ALTER` statements.

For new objects, the names of the materialized views, materialized view logs, and indexes are automatically generated by using the user-specified name template. Review the generated SQL script before attempting to execute it.

Assumptions

This tutorial assumes that you want to save and execute a script that contains the recommendations generated in "Executing a SQL Access Advisor Task".

To save and execute a SQL script:

1. Connect SQL*Plus to the database as an administrator.
2. Create a directory object and grant permissions to read and write to it.

   For example, use the following statements:

   ```
   CREATE DIRECTORY ADVISOR_RESULTS AS '/tmp';
   GRANT READ ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
   GRANT WRITE ON DIRECTORY ADVISOR_RESULTS TO PUBLIC;
   ```

3. Connect to the database as `sh`, and then save the script to a file.

   For example, use the following statement:

   ```
   EXECUTE DBMS_ADVISOR.CREATE_FILE(DBMS_ADVISOR.GET_TASK_SCRIPT('MYTASK'),
   'ADVISOR_RESULTS', 'advscript.sql');
   ```

4. Use a text editor to view the contents of the script.

   The following is a fragment of a script generated by this procedure:

   ```
   Rem  Username:        SH
   Rem  Task:            MYTASK
   Rem  Execution date:
   Rem

   Rem
   Rem  Repartitioning table "SH"."SALES"
   Rem

   SET SERVEROUTPUT ON
   SET ECHO ON

   Rem
   Rem Creating new partitioned table
   Rem
     CREATE TABLE "SH"."SALES1"
      (    "PROD_ID" NUMBER,
           "CUST_ID" NUMBER,
   ```

```
        "TIME_ID" DATE,
        "CHANNEL_ID" NUMBER,
        "PROMO_ID" NUMBER,
        "QUANTITY_SOLD" NUMBER(10,2),
        "AMOUNT_SOLD" NUMBER(10,2)
   ) PCTFREE 5 PCTUSED 40 INITRANS 1 MAXTRANS 255
 NOCOMPRESS  NOLOGGING
  TABLESPACE "EXAMPLE"
PARTITION BY RANGE ("TIME_ID") INTERVAL( NUMTOYMINTERVAL( 1,
'MONTH'))
( PARTITION VALUES LESS THAN (TO_DATE(' 1998-02-01 00:00:00',
'SYYYY-MM-DD HH24:MI:SS', 'NLS_CALENDAR=GREGORIAN')) );
.
.
.
```

5. Optionally, in SQL*Plus, run the SQL script.

   For example, enter the following command:

   ```
   @/tmp/advscript.sql
   ```

See Also:

- *Oracle Database SQL Language Reference* for `CREATE DIRECTORY` syntax
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `GET_TASK_SCRIPT` function

26.3 Performing a SQL Access Advisor Quick Tune

To tune a single SQL statement, the `DBMS_ADVISOR.QUICK_TUNE` procedure accepts as its input a `task_name` and a single SQL statement.

The `DBMS_ADVISOR.QUICK_TUNE` procedure creates a task and workload and executes this task. `EXECUTE_TASK` and `QUICK_TUNE` produce the same results. However, `QUICK_TUNE` is easier when tuning a single SQL statement.

Assumptions

This tutorial assumes the following:

- You want to tune a single SQL statement.
- You want to name the task `MY_QUICKTUNE_TASK`.

To create a template and base a task on this template:

1. Connect SQL*Plus to the database as user `sh`, and then initialize SQL*Plus variables for the SQL statement and task name.

For example, enter the following commands:

```
VARIABLE t_name VARCHAR2(255);
VARIABLE sq VARCHAR2(4000);
EXEC :sq := 'SELECT COUNT(*) FROM customers WHERE cust_state_province
=''CA''';
EXECUTE :t_name := 'MY_QUICKTUNE_TASK';
```

2. Perform the quick tune.

   For example, the following statement executes `MY_QUICKTUNE_TASK`:

```
EXEC DBMS_ADVISOR.QUICK_TUNE(DBMS_ADVISOR.SQLACCESS_ADVISOR,:t_name,:sq);
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `QUICK_TUNE` procedure and its parameters

26.4 Using SQL Access Advisor: Advanced Tasks

This section describes advanced tasks involving SQL Access Advisor.

26.4.1 Evaluating Existing Access Structures

SQL Access Advisor operates in two modes: problem-solving and evaluation.

By default, SQL Access Advisor attempts to solve access method problems by looking for enhancements to index structures, partitions, materialized views, and materialized view logs. For example, a problem-solving run may recommend creating a new index, adding a new column to a materialized view log, and so on.

When you set the `ANALYSIS_SCOPE` parameter to `EVALUATION`, SQL Access Advisor comments only on which access structures the supplied workload uses. An evaluation-only run may only produce recommendations such as retaining an index, retaining a materialized view, and so on. The evaluation mode can be useful to see exactly which indexes and materialized views a workload is using. SQL Access Advisor does not evaluate the performance impact of existing base table partitioning.

To create a task and set it to evaluation mode:

1. Connect SQL*Plus to the database with the appropriate privileges, and then create a task.

   For example, enter the following statement, where `t_name` is a SQL*Plus variable set to the name of the task:

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK(:t_name);
```

2. Perform the quick tune.

For example, the following statement sets the previous task to evaluation mode:

```
EXECUTE
DBMS_ADVISOR.SET_TASK_PARAMETER(:t_name,'ANALYSIS_SCOPE','EVALUATION
');
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `SET_TASK_PARAMETER` procedure and its parameters

26.4.2 Updating SQL Access Advisor Task Attributes

You can use the `DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES` procedure to set attributes for the task.

You can set the following attributes:

- Change the name of a task.
- Give a task a description.
- Set the task to be read-only so it cannot be changed.
- Make the task a template upon which you can define other tasks.
- Changes various attributes of a task or a task template.

Assumptions

This tutorial assumes the following:

- You want to change the name of existing task `MYTASK` to `TUNING1`.
- You want to make the task `TUNING1` read-only.

To update task attributes:

1. Connect SQL*Plus to the database as user `sh`, and then change the name of the task.

   For example, use the following statement:

   ```
   EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('MYTASK', 'TUNING1');
   ```

2. Set the task to read-only.

   For example, use the following statement:

   ```
   EXECUTE DBMS_ADVISOR.UPDATE_TASK_ATTRIBUTES('TUNING1',
     read_only => 'true');
   ```

See Also:

- "Creating and Using SQL Access Advisor Task Templates"
- *Oracle Database PL/SQL Packages and Types Reference* for more information regarding the `UPDATE_TASK_ATTRIBUTES` procedure and its parameters

26.4.3 Creating and Using SQL Access Advisor Task Templates

A **task template** is a saved configuration on which to base future tasks and workloads.

A template enables you to set up any number of tasks or workloads that can serve as starting points or templates for future task creation. By setting up a template, you can save time when performing tuning analysis. This approach also enables you to custom fit a tuning analysis to the business operation.

Physically, there is no difference between a task and a template. However, a template cannot be executed. To create a task from a template, you specify the template to be used when a new task is created. At that time, SQL Access Advisor copies the data and parameter settings from the template into the newly created task. You can also set an existing task to be a template by setting the template attribute when creating the task or later using the `UPDATE_TASK_ATTRIBUTE` procedure.

The following table describes procedures that you can use to manage task templates.

Table 26-3 DBMS_ADVISOR Procedures for Task Templates

Procedure	Description
`CREATE_TASK`	The `template` parameter is an optional task name of an existing task or task template. To specify built-in SQL Access Advisor templates, use the template name as described in Table 26-6. `is_template` is an optional parameter that enables you to set the newly created task as a template. Valid values are `true` and `false`.
`SET_TASK_PARAMETER`	The `INDEX_NAME_TEMPLATE` parameter specifies the method by which new index names are formed. The `MVIEW_NAME_TEMPLATE` parameter specifies the method by which new materialized view names are formed. The `PARTITION_NAME_TEMPLATE` parameter specifies the method by which new partition names are formed.
`UPDATE_TASK_ATTRIBUTES`	`is_template` marks the task as a template. Physically, there is no difference between a task and a template; however, a template cannot be executed. Possible values are: `true` and `false`. If the value is `NULL` or contains the value `ADVISOR_UNUSED`, then the setting is not changed.

Assumptions

This tutorial assumes the following:

- You want to create a template named `MY_TEMPLATE`.
- You want to set naming conventions for indexes and materialized views that are recommended by tasks based on `MY_TEMPLATE`.
- You want to create task `NEWTASK` based on `MY_TEMPLATE`.

To create a template and base a task on this template:

1. Connect SQL*Plus to the database as user `sh`, and then create a task as a template.

   For example, create a template named `MY_TEMPLATE` as follows:

   ```
   VARIABLE template_id NUMBER;
   VARIABLE template_name VARCHAR2(255);
   EXECUTE :template_name := 'MY_TEMPLATE';
   BEGIN
     DBMS_ADVISOR.CREATE_TASK (
       'SQL Access Advisor'
   ,   :template_id
   ,   :template_name
   ,   is_template => 'true'
   );
   END;
   ```

2. Set template parameters.

   For example, the following statements set the naming conventions for recommended indexes and materialized views:

   ```
   -- set naming conventions for recommended indexes/mvs
   BEGIN
     DBMS_ADVISOR.SET_TASK_PARAMETER (
       :template_name
   ,   'INDEX_NAME_TEMPLATE'
   ,   'SH_IDX$$_<SEQ>'
   );
   END;

   BEGIN
     DBMS_ADVISOR.SET_TASK_PARAMETER (
       :template_name
   ,   'MVIEW_NAME_TEMPLATE'
   ,   'SH_MV$$_<SEQ>'
   );
   END;
   ```

3. Create a task based on a preexisting template.

   For example, enter the following commands to create `NEWTASK` based on `MY_TEMPLATE`:

   ```
   VARIABLE task_id NUMBER;
   VARIABLE task_name VARCHAR2(255);
   EXECUTE :task_name := 'NEWTASK';
   BEGIN
     DBMS_ADVISOR.CREATE_TASK (
       'SQL Access Advisor'
   ,   :task_id
   ,   :task_name
   ,   template=>'MY_TEMPLATE'
   ```

```
    );
    END;
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `CREATE_TASK` and `SET_TASK_PARAMETER` procedures

26.4.4 Terminating SQL Access Advisor Task Execution

SQL Access Advisor enables you to interrupt the recommendation process or allow it to complete.

An interruption signals SQL Access Advisor to stop processing and marks the task as `INTERRUPTED`. At that point, you may update recommendation attributes and generate scripts.

Intermediate results represent recommendations for the workload contents up to that point in time. If recommendations must be sensitive to the entire workload, then Oracle recommends that you let the task complete. Additionally, recommendations made by the advisor early in the recommendation process do not contain base table partitioning recommendations. The partitioning analysis requires a large part of the workload to be processed before it can determine whether partitioning would be beneficial. Therefore, if SQL Access Advisor detects a benefit, then only later intermediate results contain base table partitioning recommendations.

26.4.4.1 Interrupting SQL Access Advisor Tasks

The `DBMS_ADVISOR.INTERRUPT_TASK` procedure causes a SQL Access Advisor task execution to terminate as if it had reached its normal end.

Thus, you can see any recommendations that have been formed up to the point of the interruption. An interrupted task cannot be restarted. The syntax is as follows:

```
DBMS_ADVISOR.INTERRUPT_TASK (task_name IN VARCHAR2);
```

Assumptions

This tutorial assumes the following:

- Long-running task `MYTASK` is currently executing.
- You want to interrupt this task, and then view the recommendations.

To interrupt a currently executing task:

1. Connect SQL*Plus to the database as `sh`, and then interrupt the task.

   For example, create a template named `MY_TEMPLATE` as follows:

   ```
   EXECUTE DBMS_ADVISOR.INTERRUPT_TASK ('MYTASK');
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `INTERRUPT_TASK` procedure

26.4.4.2 Canceling SQL Access Advisor Tasks

You can stop task execution by calling the `DBMS_ADVISOR.CANCEL_TASK` procedure and passing in the task name for this recommendation process.

SQL Access Advisor may take a few seconds to respond to this request. Because all advisor task procedures are synchronous, to cancel an operation, you must use a separate database session. If you use `CANCEL_TASK`, then SQL Access Advisor makes no recommendations.

A cancel command effective restores the task to its condition before the start of the canceled operation. Therefore, a canceled task or data object cannot be restarted. However, you can reset the task using `DBMS_ADVISOR.RESET_TASK`, and then execute it again. The `CANCEL_TASK` syntax is as follows:

```
DBMS_ADVISOR.CANCEL_TASK (task_name   IN  VARCHAR2);
```

The `RESET_TASK` procedure resets a task to its initial starting point, which has the effect of removing all recommendations and intermediate data from the task. The task status is set to `INITIAL`. The syntax is as follows:

```
DBMS_ADVISOR.RESET_TASK (task_name     IN VARCHAR2);
```

Assumptions

This tutorial assumes the following:

- Long-running task `MYTASK` is currently executing. This task is set to make partitioning recommendations.
- You want to cancel this task, and then reset it so that the task makes only index recommendations.

To cancel a currently executing task:

1. Connect SQL*Plus to the database as user `sh`, and then cancel the task.

   For example, create a template named `MY_TEMPLATE` as follows:

   ```
   EXECUTE DBMS_ADVISOR.CANCEL_TASK ('MYTASK');
   ```

2. Reset the task.

   For example, execute the `RESET_TASK` procedure as follows:

   ```
   EXECUTE DBMS_ADVISOR.RESET_TASK('MYTASK');
   ```

3. Set task parameters.

For example, change the analysis scope to `INDEX` as follows:

```
EXECUTE DBMS_ADVISOR.SET_TASK_PARAMETER(:task_name, 'ANALYSIS_SCOPE',
'INDEX');
```

4. Execute the task.

   For example, execute `MYTASK` as follows:

```
EXECUTE DBMS_ADVISOR.EXECUTE_TASK ('MYTASK');
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about `RESET_TASK` and `CANCEL_TASK`

26.4.5 Deleting SQL Access Advisor Tasks

The `DBMS_ADVISOR.DELETE_TASK` procedure deletes existing SQL Access Advisor tasks from the repository.

The syntax for SQL Access Advisor task deletion is as follows:

```
DBMS_ADVISOR.DELETE_TASK (task_name  IN VARCHAR2);
```

If a task is linked to an STS workload, and if you want to delete the task or workload, then you must remove the link between the task and the workload using the `DELETE_STS_REF` procedure. The following example deletes the link between task `MYTASK` and the current user's SQL tuning set `MY_STS_WORKLOAD`:

```
EXECUTE DBMS_ADVISOR.DELETE_STS_REF('MYTASK', null, 'MY_STS_WORKLOAD');
```

Assumptions

This tutorial assumes the following:

- User `sh` currently owns multiple SQL Access Advisor tasks.
- You want to delete `MYTASK`.
- The task `MYTASK` is currently linked to workload `MY_STS_WORKLOAD`.

To delete a SQL Access Advisor task:

1. Connect SQL*Plus to the database as user `sh`, and then query existing SQL Access Advisor tasks.

   For example, query the data dictionary as follows (sample output included):

```
SELECT TASK_NAME
FROM   USER_ADVISOR_TASKS
WHERE  ADVISOR_NAME = 'SQL Access Advisor';
```

```
TASK_NAME
-------------------------
MYTASK
NEWTASK
```

2. Delete the link between `MYTASK` and `MY_STS_WORKLOAD`.

   For example, delete the reference as follows:

   ```
   EXECUTE DBMS_ADVISOR.DELETE_STS_REF('MYTASK', null,
   'MY_STS_WORKLOAD');
   ```

3. Delete the desired task.

   For example, delete `MYTASK` as follows:

   ```
   EXECUTE DBMS_ADVISOR.DELETE_TASK('MYTASK');
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DELETE_TASK` procedure and its parameters

26.4.6 Marking SQL Access Advisor Recommendations

By default, all SQL Access Advisor recommendations are ready to be implemented. However, you can choose to skip or exclude selected recommendations by using the `DBMS_ADVISOR.MARK_RECOMMENDATION` procedure.

`MARK_RECOMMENDATION` enables you to annotate a recommendation with a `REJECT` or `IGNORE` setting, which causes the `GET_TASK_SCRIPT` to skip it when producing the implementation procedure.

If SQL Access Advisor makes a recommendation to partition one or multiple previously nonpartitioned base tables, then consider carefully before skipping this recommendation. Changing a table's partitioning scheme affects the cost of all queries, indexes, and materialized views defined on the table. Therefore, if you skip the partitioning recommendation, then the advisor's remaining recommendations on this table are no longer optimal. To see recommendations on your workload that do not contain partitioning, reset the advisor task and rerun it with the `ANALYSIS_SCOPE` parameter changed to exclude partitioning recommendations.

The syntax is as follows:

```
DBMS_ADVISOR.MARK_RECOMMENDATION (
   task_name           IN VARCHAR2
   id                  IN NUMBER,
   action              IN VARCHAR2);
```

Assumptions

This tutorial assumes the following:

- You are reviewing the recommendations as described in tutorial "Viewing SQL Access Advisor Task Results".
- You want to reject the first recommendation, which partitions a table.

To mark a recommendation:

1. Connect SQL*Plus to the database as user `sh`, and then mark the recommendation.

   For example, reject recommendation `1` as follows:

   ```
   EXECUTE DBMS_ADVISOR.MARK_RECOMMENDATION('MYTASK', 1, 'REJECT');
   ```

   This recommendation and any dependent recommendations do not appear in the script.
2. Generate the script as explained in "Generating and Executing a Task Script".

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `MARK_RECOMMENDATIONS` procedure and its parameters

26.4.7 Modifying SQL Access Advisor Recommendations

Using the `UPDATE_REC_ATTRIBUTES` procedure, SQL Access Advisor names and assigns ownership to new objects such as indexes and materialized views during analysis.

SQL Access Advisor may not necessarily choose appropriate names. In this case, you may choose to manually set the owner, name, and tablespace values for new objects. For recommendations referencing existing database objects, owner and name values cannot be changed. The syntax is as follows:

```
DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES (
   task_name             IN VARCHAR2
   rec_id                IN NUMBER,
   action_id             IN NUMBER,
   attribute_name        IN VARCHAR2,
   value                 IN VARCHAR2);
```

The `attribute_name` parameter can take the following values:

- `OWNER`

  Specifies the owner name of the recommended object.
- `NAME`

  Specifies the name of the recommended object.
- `TABLESPACE`

Specifies the tablespace of the recommended object.

Assumptions

This tutorial assumes the following:

- You are reviewing the recommendations as described in tutorial "Viewing SQL Access Advisor Task Results".
- You want to change the tablespace for recommendation 1, action 1 to `SH_MVIEWS`.

To mark a recommendation:

1. Connect SQL*Plus to the database as user `sh`, and then update the recommendation attribute.

   For example, change the tablespace name to `SH_MVIEWS` as follows:

   ```
   BEGIN
     DBMS_ADVISOR.UPDATE_REC_ATTRIBUTES (
       'MYTASK'
   ,   1
   ,   1
   ,   'TABLESPACE'
   ,   'SH_MVIEWS'
   );
   END;
   ```

2. Generate the script as explained in "Generating and Executing a Task Script".

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `UPDATE_REC_ATTRIBUTES` procedure and its parameters

26.5 SQL Access Advisor Examples

Oracle Database provides a script that contains several SQL Access Advisor examples that you can run on a test database.

The script is named *`ORACLE_HOME`*`/rdbms/demo/aadvdemo.sql`.

26.6 SQL Access Advisor Reference

You can access metadata about SQL Access Advisor using data dictionary views.

26.6.1 Action Attributes in the DBA_ADVISOR_ACTIONS View

The DBA_ADVISOR_ACTIONS view displays information about the actions associated with all recommendations in the database. Each action is specified by the `COMMAND` and `ATTR1` through `ATTR6` columns.

The following table maps SQL Access Advisor actions to attribute columns in the `DBA_ADVISOR_ACTIONS` view. In the table, `MV` refers to a materialized view.

Table 26-4 SQL Access Advisor Action Attributes

Action	ATTR1 Column	ATTR2 Column	ATTR3 Column	ATTR4 Column	ATTR5 Column	ATTR6 Column	NUM_ATTR1 Column
`CREATE INDEX`	Index name	Index tablespace	Target table	`BITMAP` or`BTREE`	Index column list / expression	Unused	Storage size in bytes for the index
`CREATE MATERIALIZED VIEW`	MV name	MV tablespace	`REFRESH COMPLETE, REFRESH FAST,REFRESH FORCE, NEVER REFRESH`	`ENABLE QUERY REWRITE, DISABLE QUERY REWRITE`	SQL `SELECT` statement	Unused	Storage size in bytes for the MV
`CREATE MATERIALIZED VIEW LOG`	Target table name	MV log tablespace	`ROWID PRIMARY KEY,SEQUENCE OBJECT ID`	`INCLUDING NEW VALUES, EXCLUDING NEW VALUES`	Table column list	Partitioning subclauses	Unused
`CREATE REWRITE EQUIVALENCE`	Name of equivalence	Checksum value	Unused	Unused	Source SQL statement	Equivalent SQL statement	Unused
`DROP INDEX`	Index name	Unused	Unused	Unused	Index columns	Unused	Storage size in bytes for the index
`DROP MATERIALIZED VIEW`	MV name	Unused	Unused	Unused	Unused	Unused	Storage size in bytes for the MV
`DROP MATERIALIZED VIEW LOG`	Target table name	Unused	Unused	Unused	Unused	Unused	Unused

Table 26-4 (Cont.) SQL Access Advisor Action Attributes

Action	ATTR1 Column	ATTR2 Column	ATTR3 Column	ATTR4 Column	ATTR5 Column	ATTR6 Column	NUM_ATTR1 Column
PARTITION TABLE	Table name	RANGE, INTERVAL, LIST, HASH, RANGE-HASH, RANGE-LIST	Partition key for partitioning (column name or list of column names)	Partition key for subpartitioning (column name or list of column names)	SQL PARTITION clause	SQL SUBPARTITION clause	Unused
PARTITION INDEX	Index name	LOCAL, RANGE, HASH	Partition key for partitioning (list of column names)	Unused	SQL PARTITION clause	Unused	Unused
PARTITION ON MATERIALIZED VIEW	MV name	RANGE, INTERVAL, LIST, HASH, RANGE-HASH, RANGE-LIST	Partition key for partitioning (column name or list of column names)	Partition key for subpartitioning (column name or list of column names)	SQL SUBPARTITION clause	SQL SUBPARTITION clause	Unused
RETAIN INDEX	Index name	Unused	Target table	BITMAP or BTREE	Index columns	Unused	Storage size in bytes for the index
RETAIN MATERIALIZED VIEW	MV name	Unused	REFRESH COMPLETE or REFRESH FAST	Unused	SQL SELECT statement	Unused	Storage size in bytes for the MV
RETAIN MATERIALIZED VIEW LOG	Target table name	Unused	Unused	Unused	Unused	Unused	Unused

26.6.2 Categories for SQL Access Advisor Task Parameters

SQL Access Advisor task parameters fall into the following categories: workload filtering, task configuration, schema attributes, and recommendation options.

The following table groups the most relevant SQL Access Advisor task parameters into categories. All task parameters for workload filtering are deprecated.

Table 26-5 *Types of Advisor Task Parameters And Their Uses*

Workload Filtering	Task Configuration	Schema Attributes	Recommendation Options
END_TIME	DAYS_TO_EXPIRE	DEF_INDEX_OWNER	ANALYSIS_SCOPE
INVALID_ACTION_LIST	JOURNALING	DEF_INDEX_TABLESPACE	COMPATIBILITY

Table 26-5 (Cont.) *Types of Advisor Task Parameters And Their Uses*

Workload Filtering	Task Configuration	Schema Attributes	Recommendation Options
INVALID_MODULE_LIST	REPORT_DATE_FORMAT	DEF_MVIEW_OWNER	CREATION_COST
INVALID_SQLSTRING_LIMIT		DEF_MVIEW_TABLESPACE	DML_VOLATILITY
INVALID_TABLE_LIST		DEF_MVLOG_TABLESPACE	LIMIT_PARTITION_SCHEMES
INVALID_USERNAME_LIST		DEF_PARTITION_TABLESPACE	MODE
RANKING_MEASURE		INDEX_NAME_TEMPLATE	PARTITIONING_TYPES
SQL_LIMIT		MVIEW_NAME_TEMPLATE	REFRESH_MODE
START_TIME			STORAGE_CHANGE
TIME_LIMIT			USE_SEPARATE_TABLESPACES
VALID_ACTION_LIST			WORKLOAD_SCOPE
VALID_MODULE_LIST			
VALID_SQLSTRING_LIST			
VALID_TABLE_LIST			
VALID_USERNAME_LIST			

26.6.3 SQL Access Advisor Constants

DBMS_ADVISOR provides a number of constants.

You can use the constants shown in the following table with SQL Access Advisor.

Table 26-6 SQL Access Advisor Constants

Constant	Description
ADVISOR_ALL	A value that indicates all possible values. For string parameters, this value is equivalent to the wildcard (%) character.
ADVISOR_CURRENT	Indicates the current time or active set of elements. Typically, this is used in time parameters.
ADVISOR_DEFAULT	Indicates the default value. Typically used when setting task or workload parameters.
ADVISOR_UNLIMITED	A value that represents an unlimited numeric value.
ADVISOR_UNUSED	A value that represents an unused entity. When a parameter is set to ADVISOR_UNUSED, it has no effect on the current operation. A typical use for this constant is to set a parameter as unused for its dependent operations.
SQLACCESS_GENERAL	Specifies the name of a default SQL Access general-purpose task template. This template sets the DML_VOLATILITY task parameter to true and ANALYSIS_SCOPE to INDEX, MVIEW.

Table 26-6 (Cont.) SQL Access Advisor Constants

Constant	Description
`SQLACCESS_OLTP`	Specifies the name of a default SQL Access OLTP task template. This template sets the `DML_VOLATILITY` task parameter to `true` and `ANALYSIS_SCOPE` to `INDEX`.
`SQLACCESS_WAREHOUSE`	Specifies the name of a default SQL Access warehouse task template. This template sets the `DML_VOLATILITY` task parameter to `false` and `EXECUTION_TYPE` to `INDEX`, `MVIEW`.
`SQLACCESS_ADVISOR`	Contains the formal name of SQL Access Advisor. You can specify this name when procedures require the Advisor name as an argument.

Part IX
SQL Management Objects

A **SQL management object** is a feature that stabilizes the execution plans of individual SQL statements. SQL profiles and SQL plan baselines are SQL management objects.

27

Managing SQL Profiles

When warranted, SQL Tuning Advisor recommends a SQL profile. You can use `DBMS_SQLTUNE` to implement, alter, drop, and transport SQL profiles.

27.1 About SQL Profiles

A **SQL profile** is a database object that contains auxiliary statistics specific to a SQL statement.

Conceptually, a SQL profile is to a SQL statement what object-level statistics are to a table or index. SQL profiles are created when a DBA invokes SQL Tuning Advisor.

See Also:

"About SQL Tuning Advisor"

27.1.1 Purpose of SQL Profiles

When profiling a SQL statement, SQL Tuning Advisor uses a specific set of bind values as input.

The advisor compares the optimizer estimate with values obtained by executing fragments of the statement on a data sample. When significant variances are found, SQL Tuning Advisor bundles corrective actions together in a SQL profile, and then recommends its acceptance.

The corrected statistics in a SQL profile can improve optimizer cardinality estimates, which in turn leads the optimizer to select better plans. SQL profiles provide the following benefits over other techniques for improving plans:

- Unlike hints and stored outlines, SQL profiles do not tie the optimizer to a specific plan or subplan. SQL profiles fix incorrect estimates while giving the optimizer the flexibility to pick the best plan in different situations.
- Unlike hints, no changes to application source code are necessary when using SQL profiles. The use of SQL profiles by the database is transparent to the user.

See Also:

- "Influencing the Optimizer with Hints"
- "Analyzing SQL with SQL Tuning Advisor"

27.1.2 Concepts for SQL Profiles

A SQL profile is a collection of auxiliary statistics on a query, including all tables and columns referenced in the query.

The profile is stored in an internal format in the data dictionary. The user interface is the `DBA_SQL_PROFILES` dictionary view. The optimizer uses this information during optimization to determine the most optimal plan.

See Also:

Oracle Database Reference to learn more about `DBA_SQL_PROFILES`

27.1.2.1 Statistics in SQL Profiles

A SQL profile contains, among other statistics, a set of cardinality adjustments.

The cardinality measure is based on sampling the `WHERE` clause rather than on statistical projection. A profile uses parts of the query to determine whether the estimated cardinalities are close to the actual cardinalities and, if a mismatch exists, uses the corrected cardinalities. For example, if a SQL profile exists for `SELECT * FROM t WHERE x=5 AND y=10`, then the profile stores the actual number of rows returned.

Starting in Oracle Database 18c, SQL Tuning Advisor can recommend an Exadata-aware SQL profile. On Oracle Exadata Database Machine, the cost of smart scans depends on the system statistics I/O seek time (`ioseektim`), multiblock read count (`mbrc`), and I/O transfer speed (`iotfrspeed`). The values of these statistics usually differ on Exadata and can thus influence the choice of plan. If system statistics are stale, and if gathering them improves performance, then SQL Tuning Advisor recommends accepting an Exadata-aware SQL profile.

See Also:

- "Permanent Table Statistics"
- *Oracle Database Performance Tuning Guide* to learn about system statistics
- *Oracle Exadata Database Machine System Overview*

27.1.2.2 SQL Profiles and Execution Plans

The SQL profile contains supplemental statistics for the entire *statement*, not individual *plans*. The profile does not itself determine a specific plan.

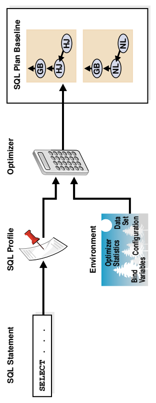
Internally, a SQL profile is implemented using hints that address different types of problems. These hints do not specify any particular plan. Rather, the hints correct errors in the optimizer estimation algorithm that lead to suboptimal plans. For example, a profile may use the `TABLE_STATS` hint to set object statistics for tables when the statistics are missing or stale.

When choosing plans, the optimizer has the following sources of information:

- The environment, which contains the database configuration, system statistics, bind variable values, optimizer statistics, data set, and so on
- The supplemental statistics in the SQL profile

The following figure shows the relationship between a SQL statement and the SQL profile for this statement. The optimizer uses the SQL profile and the environment to generate an execution plan. In this example, the plan is in the SQL plan baseline for the statement.

Figure 27-1 SQL Profile



If either the optimizer environment or SQL profile changes, then the optimizer can create a new plan. As tables grow, or as indexes are created or dropped, the plan for a SQL profile can change. The profile continues to be relevant even if the data distribution or access path of the corresponding statement changes.

In general, you do not need to refresh SQL profiles. Over time, however, profile content can become outdated. In this case, performance of the SQL statement may degrade. The statement may appear as high-load or top SQL. In this case, the Automatic SQL Tuning task again captures the statement as high-load SQL. You can implement a new SQL profile for the statement.

See Also:

- "Differences Between SQL Plan Baselines and SQL Profiles"
- "Introduction to Optimizer Statistics"
- *Oracle Database SQL Language Reference* to learn about SQL hints

27.1.2.3 SQL Profile Recommendations

SQL Tuning Advisor invokes Automatic Tuning Optimizer to generate SQL profile recommendations.

Recommendations to implement SQL profiles occur in a finding, which appears in a separate section of the SQL Tuning Advisor report. When you implement (or accept) a SQL profile, the database creates the profile and stores it persistently in the data dictionary. However, the SQL profile information is not exposed through regular dictionary views.

Example 27-1 SQL Profile Recommendation

In this example, the database found a better plan for a `SELECT` statement that uses several expensive joins. The database recommends running `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` to implement the profile, which enables the statement to run 98.53% faster.

```
-----------------------------------------------------------------------
----
FINDINGS SECTION (2 findings)
-----------------------------------------------------------------------
----

1- SQL Profile Finding (see explain plans section below)
--------------------------------------------------------
  A potentially better execution plan was found for this statement.
Choose
  one of the following SQL profiles to implement.

  Recommendation (estimated benefit: 99.45%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile.
    execute dbms_sqltune.accept_sql_profile(task_name => 'my_task',
            object_id => 3, task_owner => 'SH', replace => TRUE);

  Validation results
  ------------------
  The SQL profile was tested by executing both its plan and the
original
  plan and measuring their respective execution statistics. A plan may
  have been only partially executed if the other could be run to
  completion in less time.

                         Original Plan  With SQL Profile  % Improved
                         -------------  ----------------  ----------
  Completion Status:           PARTIAL          COMPLETE
  Elapsed Time(us):           15467783            226902     98.53 %
  CPU Time(us):               15336668            226965     98.52 %
  User I/O Time(us):                 0                 0
  Buffer Gets:                 3375243             18227     99.45 %
  Disk Reads:                        0                 0
  Direct Writes:                     0                 0
  Rows Processed:                    0               109
  Fetches:                           0               109
  Executions:                        0                 1

  Notes
  -----
  1. The SQL profile plan was first executed to warm the buffer cache.
  2. Statistics for the SQL profile plan were averaged over next 3
executions.
```

Sometimes SQL Tuning Advisor may recommend implementing a profile that uses the Automatic Degree of Parallelism (Auto DOP) feature. A parallel query profile is only recommended when the original plan is serial and when parallel execution can significantly reduce the elapsed time for a long-running query.

When it recommends a profile that uses Auto DOP, SQL Tuning Advisor gives details about the performance overhead of using parallel execution for the SQL statement in the report. For parallel execution recommendations, SQL Tuning Advisor may provide two SQL profile recommendations, one using serial execution and one using parallel.

The following example shows a parallel query recommendation. In this example, a degree of parallelism of 7 improves response time significantly at the cost of increasing resource consumption by almost 25%. You must decide whether the reduction in database throughput is worth the increase in response time.

```
  Recommendation (estimated benefit: 99.99%)
  ------------------------------------------
  - Consider accepting the recommended SQL profile to use parallel
    execution for this statement.
    execute dbms_sqltune.accept_sql_profile(task_name => 'gfk_task',
            object_id => 3, task_owner => 'SH', replace => TRUE,
            profile_type => DBMS_SQLTUNE.PX_PROFILE);

  Executing this query parallel with DOP 7 will improve its response time
  82.22% over the SQL profile plan. However, there is some cost in enabling
  parallel execution. It will increase the statement's resource
  consumption by an estimated 24.43% which may result in a reduction of
  system throughput. Also, because these resources are consumed over a
  much   smaller duration, the response time of concurrent statements
  might be negatively impacted if sufficient hardware capacity is not
  available.

  The following data shows some sampled statistics for this SQL from the
  past week and projected weekly values when parallel execution is enabled.

                                 Past week sampled statistics for this SQL
                                 -----------------------------------------
  Number of executions                                                   0
  Percent of total activity                                            .29
  Percent of samples with #Active Sessions > 2*CPU                       0
  Weekly DB time (in sec)                                            76.51

                              Projected statistics with Parallel Execution
                              --------------------------------------------
  Weekly DB time (in sec)                                            95.21
```

See Also:

- "SQL Profiling"
- *Oracle Database VLDB and Partitioning Guide* to learn more about Auto DOP
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` procedure

27.1.2.4 SQL Profiles and SQL Plan Baselines

You can use SQL profiles with or without SQL plan management.

No strict relationship exists between the SQL profile and a SQL plan baseline. If a statement has multiple plans in a SQL plan baseline, then a SQL profile is useful because it enables the optimizer to choose the lowest-cost plan in the baseline.

See Also:

"Overview of SQL Plan Management"

27.1.3 User Interfaces for SQL Profiles

Oracle Enterprise Manager Cloud Control (Cloud Control) usually handles SQL profiles as part of automatic SQL tuning.

On the command line, you can manage SQL profiles with the `DBMS_SQLTUNE` package. To use the APIs, you must have the `ADMINISTER SQL MANAGEMENT OBJECT` privilege.

See Also:

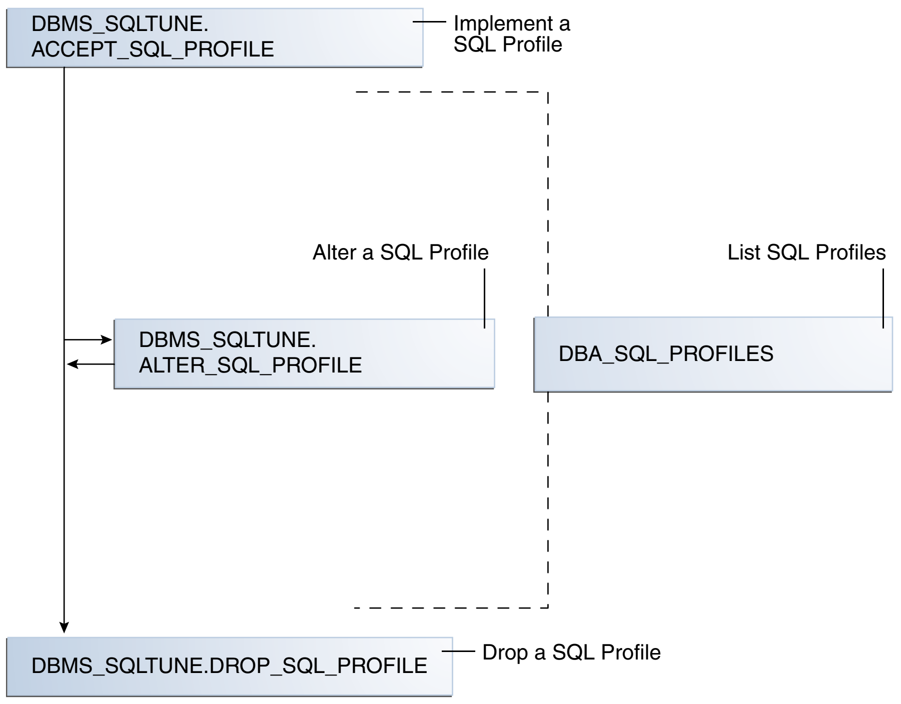
- *Oracle Database PL/SQL Packages and Types Reference* for information about the `DBMS_SQLTUNE` package
- *Oracle Database 2 Day + Performance Tuning Guide* to learn how to manage SQL profiles with Cloud Control

27.1.4 Basic Tasks for SQL Profiles

Basic tasks include accepting (implementing) a SQL profile, altering it, listing it, and dropping it.

The following graphic shows the basic workflow.

Figure 27-2   Managing SQL Profiles



Typically, you manage SQL profiles in the following sequence:

1. Implement a recommended SQL profile.

   "Implementing a SQL Profile" describes this task.

2. Obtain information about SQL profiles stored in the database.

   "Listing SQL Profiles" describes this task.

3. Optionally, modify the implemented SQL profile.

   "Altering a SQL Profile" describes this task.

4. Drop the implemented SQL profile when it is no longer needed.

   "Dropping a SQL Profile" describes this task.

To tune SQL statements on another database, you can transport both a SQL tuning set and a SQL profile to a separate database. "Transporting a SQL Profile" describes this task.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `DBMS_SQLTUNE` package

27.2 Implementing a SQL Profile

Implementing a SQL profile means storing it persistently in the database.

Implementing a profile is the same as accepting it. A profile must be accepted before the optimizer can use it as input when generating plans.

27.2.1 About SQL Profile Implementation

As a rule of thumb, implement a SQL profile recommended by SQL Tuning Advisor.

If the database recommends both an index and a SQL profile, then either use both or use the SQL profile only. If you create an index, then the optimizer may need the profile to pick the new index.

In some situations, SQL Tuning Advisor may find an improved serial plan in addition to an even better parallel plan. In this case, the advisor recommends both a standard and a parallel SQL profile, enabling you to choose between the best serial and best parallel plan for the statement. Implement a parallel plan only if the increase in response time is worth the decrease in throughput.

To implement a SQL profile, execute the `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` procedure. Some important parameters are as follows:

- `profile_type`

  Set this parameter to `REGULAR_PROFILE` for a SQL profile without a change to parallel execution, or `PX_PROFLE` for a SQL profile with a change to parallel execution.

- `force_match`

  This parameter controls statement matching. Typically, an accepted SQL profile is associated with the SQL statement through a SQL signature that is generated using a hash function. This hash function changes the SQL statement to upper case and removes all extra whitespace before generating the signature. Thus, the same SQL profile works for all SQL statements in which the only difference is case and white spaces.

  By setting `force_match` to `true`, the SQL profile additionally targets all SQL statements that have the same text after the literal values in the `WHERE` clause have been replaced by bind variables. This setting may be useful for applications that use only literal values because it enables SQL with text differing only in its literal values to share a SQL profile. If both literal values and bind variables are in the SQL text, or if `force_match` is set to `false` (default), then the literal values in the `WHERE` clause are not replaced by bind variables.

See Also:

Oracle Database PL/SQL Packages and Types Reference for information about the `ACCEPT_SQL_PROFILE` procedure

27.2.2 Implementing a SQL Profile

To implement a SQL profile, use the `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` procedure.

Assumptions

This tutorial assumes the following:

- The SQL Tuning Advisor task `STA_SPECIFIC_EMP_TASK` includes a recommendation to create a SQL profile.
- The name of the SQL profile is `my_sql_profile`.
- The PL/SQL block accepts a profile that uses parallel execution (`profile_type`).
- The profile uses force matching.

To implement a SQL profile:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Execute the `ACCEPT_SQL_PROFILE` function.

   For example, execute the following PL/SQL:

```
DECLARE
  my_sqlprofile_name VARCHAR2(30);
BEGIN
  my_sqlprofile_name := DBMS_SQLTUNE.ACCEPT_SQL_PROFILE (
    task_name    => 'STA_SPECIFIC_EMP_TASK'
,   name         => 'my_sql_profile'
,   profile_type => DBMS_SQLTUNE.PX_PROFILE
,   force_match  => true
);
END;
/
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DBMS_SQLTUNE.ACCEPT_SQL_PROFILE` procedure

27.3 Listing SQL Profiles

The data dictionary view `DBA_SQL_PROFILES` stores SQL profiles persistently in the database.

The profile statistics are in an Oracle Database internal format, so you cannot query profiles directly. However, you can list profiles.

To list SQL profiles:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Query the `DBA_SQL_PROFILES` view.

   For example, execute the following query:

   ```
   COLUMN category FORMAT a10
   COLUMN sql_text FORMAT a20

   SELECT NAME, SQL_TEXT, CATEGORY, STATUS
   FROM   DBA_SQL_PROFILES;
   ```

   Sample output appears below:

   ```
   NAME                           SQL_TEXT             CATEGORY
   STATUS
   ------------------------------ -------------------- ----------
   --------
   SYS_SQLPROF_01285f6d18eb0000   select promo_name, c DEFAULT
   ENABLED
                                  ount(*) c from promo
                                  tions p, sales s whe
                                  re s.promo_id = p.pr
                                  omo_id and p.promo_c
                                  ategory = 'internet'
                                   group by p.promo_na
                                  me order by c desc
   ```

See Also:

Oracle Database Reference to learn about the `DBA_SQL_PROFILES` view

27.4 Altering a SQL Profile

You can alter attributes of an existing SQL profile using the `attribute_name` parameter of the `ALTER_SQL_PROFILE` procedure.

The `CATEGORY` attribute determines which sessions can apply a profile. View the `CATEGORY` attribute by querying `DBA_SQL_PROFILES.CATEGORY`. By default, all profiles are in the `DEFAULT` category, which means that all sessions in which the `SQLTUNE_CATEGORY` initialization parameter is set to `DEFAULT` can use the profile.

By altering the category of a SQL profile, you determine which sessions are affected by profile creation. For example, by setting the category to `DEV`, only sessions in which the `SQLTUNE_CATEGORY` initialization parameter is set to `DEV` can use the profile. Other sessions do not have access to the SQL profile and execution plans for SQL statements are not impacted by the SQL profile. This technique enables you to test a profile in a restricted environment before making it available to other sessions.

The example in this section assumes that you want to change the category of the SQL profile so it is used only by sessions with the SQL profile category set to `TEST`, run the SQL statement, and then change the profile category back to `DEFAULT`.

To alter a SQL profile:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use the `ALTER_SQL_PROFILE` procedure to set the `attribute_name`.

   For example, execute the following code to set the attribute `CATEGORY` to `TEST`:

   ```
   VARIABLE pname my_sql_profile
   BEGIN DBMS_SQLTUNE.ALTER_SQL_PROFILE (
      name            =>  :pname
   ,  attribute_name  =>  'CATEGORY'
   ,  value           =>  'TEST'
   );
   END;
   ```

3. Change the initialization parameter setting in the current database session.

   For example, execute the following SQL:

   ```
   ALTER SESSION SET SQLTUNE_CATEGORY = 'TEST';
   ```

4. Test the profiled SQL statement.
5. Use the `ALTER_SQL_PROFILE` procedure to set the `attribute_name`.

   For example, execute the following code to set the attribute `CATEGORY` to `DEFAULT`:

   ```
   VARIABLE pname my_sql_profile
   BEGIN
     DBMS_SQLTUNE.ALTER_SQL_PROFILE (
         name            =>  :pname
   ,     attribute_name  =>  'CATEGORY'
   ,     value           =>  'DEFAULT'
   );
   END;
   ```

See Also:

- *Oracle Database Reference* to learn about the `SQLTUNE_CATEGORY` initialization parameter
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `ALTER_SQL_PROFILE` procedure

27.5 Dropping a SQL Profile

You can drop a SQL profile with the `DROP_SQL_PROFILE` procedure.

Assumptions

This section assumes the following:

- You want to drop `my_sql_profile`.
- You want to ignore errors raised if the name does not exist.

To drop a SQL profile:

1. In SQL*Plus or SQL Developer, log in to the database as a user with the necessary privileges.
2. Use the `DBMS_SQLTUNE.DROP_SQL_PROFILE` procedure.

   The following example drops the profile named `my_sql_profile`:

```
BEGIN
  DBMS_SQLTUNE.DROP_SQL_PROFILE (
    name => 'my_sql_profile'
);
END;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DROP_SQL_PROFILE` procedure
- *Oracle Database Reference* to learn about the `SQLTUNE_CATEGORY` initialization parameter

27.6 Transporting a SQL Profile

You can export a SQL profile from the `SYS` schema in one database to a staging table, and then import it from the staging table into another database. You can transport a SQL profile to any Oracle database created in the same release or later.

Table 27-1 shows the main procedures and functions for managing SQL profiles.

Table 27-1 APIs for Transporting SQL Profiles

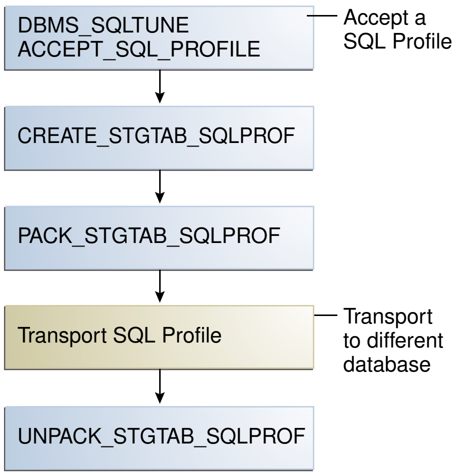
Procedure or Function	Description
`CREATE_STGTAB_SQLPROF`	Creates the staging table used for copying SQL profiles from one system to another.

Table 27-1 (Cont.) APIs for Transporting SQL Profiles

Procedure or Function	Description
`PACK_STGTAB_SQLPROF`	Moves profile data out of the `SYS` schema into the staging table.
`UNPACK_STGTAB_SQLPROF`	Uses the profile data stored in the staging table to create profiles on this system.

The following graphic shows the basic workflow of transporting SQL profiles.

Figure 27-3 Transporting SQL Profiles



Assumptions

This tutorial assumes the following:

- You want to transport `my_profile` from a production database to a test database.
- You want to create the staging table in the `dba1` schema.

To transport a SQL profile:

1. Connect SQL*Plus to the database with the appropriate privileges, and then use the `CREATE_STGTAB_SQLPROF` procedure to create a staging table to hold the SQL profiles.

   The following example creates `my_staging_table` in the `dba1` schema:

   ```
   BEGIN
     DBMS_SQLTUNE.CREATE_STGTAB_SQLPROF (
       table_name  => 'my_staging_table'
   ,   schema_name => 'dba1'
   );
   END;
   /
   ```

2. Use the `PACK_STGTAB_SQLPROF` procedure to export SQL profiles into the staging table.

The following example populates `dba1.my_staging_table` with the SQL profile `my_profile`:

```
BEGIN
  DBMS_SQLTUNE.PACK_STGTAB_SQLPROF (
    profile_name         => 'my_profile'
,   staging_table_name   => 'my_staging_table'
,   staging_schema_owner => 'dba1'
);
END;
/
```

3. Move the staging table to the database where you plan to unpack the SQL profiles.

   Move the table using your utility of choice. For example, use Oracle Data Pump or a database link.

4. On the database where you plan to import the SQL profiles, use `UNPACK_STGTAB_SQLPROF` to unpack SQL profiles from the staging table.

   The following example shows how to unpack SQL profiles in the staging table:

```
BEGIN
  DBMS_SQLTUNE.UNPACK_STGTAB_SQLPROF(
     replace            => true
,    staging_table_name => 'my_staging_table'
);
END;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for complete reference information about `DBMS_SQLTUNE`
- *Oracle Database Utilities* to learn how to use Oracle Data Pump

28
Overview of SQL Plan Management

SQL plan management is a preventative mechanism that enables the optimizer to automatically manage execution plans, ensuring that the database uses only known or verified plans.

28.1 About SQL Plan Baselines

SQL plan management uses a mechanism called a **SQL plan baseline**, which is a set of accepted plans that the optimizer is allowed to use for a SQL statement.

In this context, a plan includes all plan-related information (for example, SQL plan identifier, set of hints, bind values, and optimizer environment) that the optimizer needs to reproduce an execution plan. The baseline is implemented as a set of plan rows and the outlines required to reproduce the plan. An outline is a set of optimizer hints used to force a specific plan.

The main components of SQL plan management are as follows:

- Plan capture

  This component stores relevant information about plans for a set of SQL statements.

- Plan selection

  This component is the detection by the optimizer of plan changes based on stored plan history, and the use of SQL plan baselines to select appropriate plans to avoid potential performance regressions.

- Plan evolution

  This component is the process of adding new plans to existing SQL plan baselines, either manually or automatically. In the typical use case, the database accepts a plan into the plan baseline only after verifying that the plan performs well.

28.2 Purpose of SQL Plan Management

SQL plan management prevents performance regressions caused by plan changes.

A secondary goal is to gracefully adapt to changes such as new optimizer statistics or indexes by verifying and accepting only plan changes that improve performance.

> **Note:**
>
> SQL plan baselines cannot help when an event has caused irreversible execution plan changes, such as dropping an index.

28.2.1 Benefits of SQL Plan Management

SQL plan management can improve or preserve SQL performance in database upgrades and system and data changes.

Specifically, benefits include:

- A database upgrade that installs a new optimizer version usually results in plan changes for a small percentage of SQL statements.

  Most plan changes result in either improvement or no performance change. However, some plan changes may cause performance regressions. SQL plan baselines significantly minimize potential regressions resulting from an upgrade.

  When you upgrade, the database only uses plans from the plan baseline. The database puts new plans that are not in the current baseline into a holding area, and later evaluates them to determine whether they use fewer resources than the current plan in the baseline. If the plans perform better, then the database promotes them into the baseline; otherwise, the database does not promote them.
- Ongoing system and data changes can affect plans for some SQL statements, potentially causing performance regressions.

  SQL plan baselines help minimize performance regressions and stabilize SQL performance.
- Deployment of new application modules introduces new SQL statements into the database.

  The application software may use appropriate SQL execution plans developed in a standard test configuration for the new statements. If the system configuration is significantly different from the test configuration, then the database can evolve SQL plan baselines over time to produce better performance.

See Also:

Oracle Database Upgrade Guide to learn how to upgrade an Oracle database

28.2.2 Differences Between SQL Plan Baselines and SQL Profiles

Both SQL profiles and SQL plan baselines help improve the performance of SQL statements by ensuring that the optimizer uses only optimal plans.

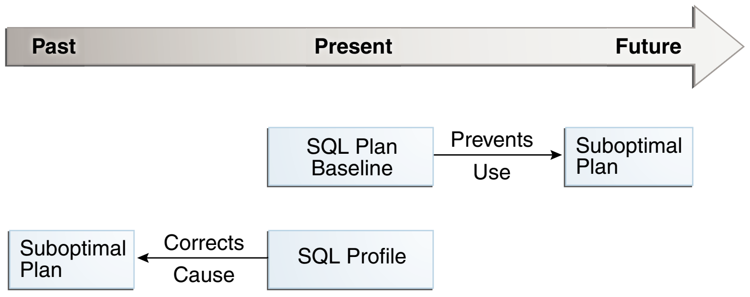
Both profiles and baselines are internally implemented using hints. However, these mechanisms have significant differences, including the following:

- In general, SQL plan baselines are proactive, whereas SQL profiles are reactive.

  Typically, you create SQL plan baselines *before* significant performance problems occur. SQL plan baselines prevent the optimizer from using suboptimal plans in the future.

  The database creates SQL profiles when you invoke SQL Tuning Advisor, which you do typically only *after* a SQL statement has shown high-load symptoms. SQL profiles are primarily useful by providing the ongoing resolution of optimizer mistakes that have led to suboptimal plans. Because the SQL profile mechanism is reactive, it cannot guarantee stable performance as drastic database changes occur.

Figure 28-1 SQL Plan Baselines and SQL Profiles



- SQL plan baselines reproduce a specific plan, whereas SQL profiles correct optimizer cost estimates.

  A SQL plan baseline is a set of accepted plans. Each plan is implemented using a set of outline hints that fully specify a particular plan. SQL profiles are also implemented using hints, but these hints do not specify any specific plan. Rather, the hints correct miscalculations in the optimizer estimates that lead to suboptimal plans. For example, a hint may correct the cardinality estimate of a table.

  Because a profile does not constrain the optimizer to any one plan, a SQL profile is more flexible than a SQL plan baseline. For example, changes in initialization parameters and optimizer statistics enable the optimizer to choose a better plan.

Oracle recommends that you use SQL Tuning Advisor. In this way, you follow the recommendations made by the advisor for SQL profiles and plan baselines rather than trying to determine which mechanism is best for each SQL statement.

See Also:

- "About Optimizer Hints"
- "Managing SQL Profiles"
- "Analyzing SQL with SQL Tuning Advisor"

28.3 Plan Capture

SQL plan capture refers to techniques for capturing and storing relevant information about plans in the SQL Management Base for a set of SQL statements.

Capturing a plan means making SQL plan management aware of this plan. You can configure initial plan capture to occur automatically by setting an initialization parameter, or you can capture plans manually by using the `DBMS_SPM` package.

28.3.1 Automatic Initial Plan Capture

When enabled, the database checks whether executed SQL statements are eligible for automatic capture.

You can enable **automatic initial plan capture** by setting `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` to `true` (the default is `false`). Note that the initialization parameter `OPTIMIZER_USE_SQL_PLAN_BASELINES` is independent. For example, if `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` is `true`, then the database creates initial plan baselines regardless of whether `OPTIMIZER_USE_SQL_PLAN_BASELINES` is `true` or `false`.

See Also:

- "Plan Selection"
- *Oracle Database Reference* to learn about the `OPTIMIZER_USE_SQL_PLAN_BASELINES` initialization parameter

28.3.1.1 Eligibility for Automatic Initial Plan Capture

To be eligible for automatic plan capture, an executed statement must be repeatable, and it must not be excluded by any capture filters.

By default, the database considers all repeatable SQL statements as eligible for capture, with the following exceptions:

- `CREATE TABLE` when the `AS SELECT` clause is *not* specified
- `DROP TABLE`
- `INSERT INTO ... VALUES`

The first check for eligibility is repeated execution. If a statement is executed less than twice, then the database does not consider it eligible for a SQL plan baseline. If a statement is executed at least twice, then it is by definition repeatable, and so the database considers it eligible for further checking.

Note:

SQL plan management does not protect statements that have been explained using `EXPLAIN PLAN` but have not been executed.

For repeatable statements, the `DBMS_SPM.CONFIGURE` procedure enables you to create an automatic capture filter. Thus, you can capture only statements that you want, and exclude noncritical statements, thereby saving space in the `SYSAUX` tablespace. Noncritical queries often have the following characteristics:

- Not executed often enough to be significant
- Not resource-intensive
- Not sufficiently complex to benefit from SQL plan management

For a specified parameter, a filter either includes (`allow=>TRUE`) or excludes (`allow=>FALSE`) plans for statements with the specified values. To be eligible for capture, a repeatable statement must not be *excluded* by any filter. The

DBMS_SPM.CONFIGURE procedure supports filters for SQL text, parsing schema name, module, and action.

A null value for any parameter removes the filter. By using `parameter_value=>'%'` in combination with `allow=FALSE`, you can filter out all values for a parameter, and then create a separate filter to include only specified values. The `DBA_SQL_MANAGEMENT_CONFIG` view shows the current filters.

See Also:

- "Configuring Filters for Automatic Plan Capture"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_SPM.CONFIGURE` procedure
- *Oracle Database Reference* to learn more about the `DBA_SQL_MANAGEMENT_CONFIG` view

28.3.1.2 Plan Matching for Automatic Initial Plan Capture

If the database executes a repeatable SQL statement, and if this statement passes through the `DBMS_SPM.CONFIGURE` filters, then the database attempts to match a plan in the SQL plan baseline.

For automatic initial plan capture, the plan matching algorithm is as follows:

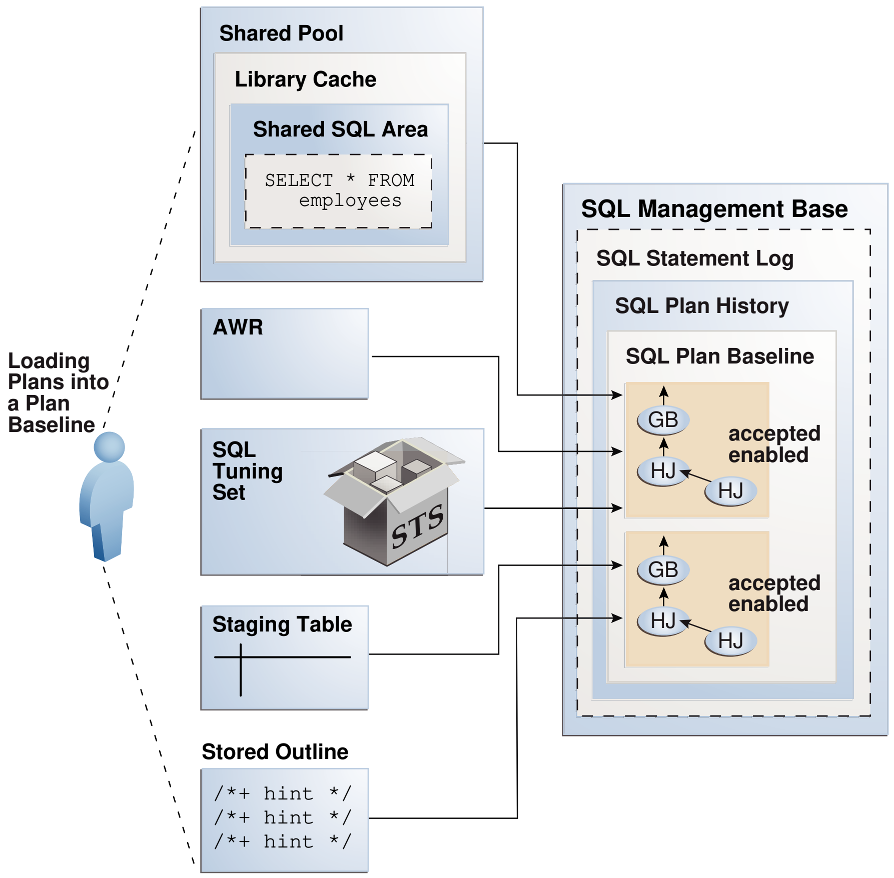
- If a SQL plan baseline does *not* exist, then the optimizer creates a plan history and SQL plan baseline for the statement, marking the initial plan for the statement as accepted and adding it to the SQL plan baseline.
- If a SQL plan baseline exists, then the optimizer behavior depends on the cost-based plan derived at parse time:
  – If this plan does *not* match a plan in the SQL plan baseline, then the optimizer marks the new plan as unaccepted and adds it to the SQL plan baseline.
  – If this plan *does* match a plan in the SQL plan baseline, then nothing is added to the SQL plan baseline.

28.3.2 Manual Plan Capture

In SQL plan management, **manual plan capture** refers to the user-initiated bulk load of existing plans into a SQL plan baseline.

Use Cloud Control or PL/SQL to load the execution plans for SQL statements from AWR, a SQL tuning set (STS), the shared SQL area, a staging table, or a stored outline.

Figure 28-2   Loading Plans into a SQL Plan Baseline



The loading behavior varies depending on whether a SQL plan baseline exists for each statement represented in the bulk load:

- If a baseline for the statement does not exist, then the database does the following:
  1. Creates a plan history and plan baseline for the statement
  2. Marks the initial plan for the statement as accepted
  3. Adds the plan to the new baseline
- If a baseline for the statement exists, then the database does the following:
  1. Marks the loaded plan as accepted
  2. Adds the plan to the plan baseline for the statement *without* verifying the plan's performance

Manually loaded plans are always marked accepted because the optimizer assumes that any plan loaded manually by the administrator has acceptable performance. You can load plans without enabling them by setting the `enabled` parameter to `NO` in the `DBMS_SPM.LOAD_PLANS_FROM_%` functions.

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DBMS_SPM.LOAD_PLANS_FROM_%` functions

28.4 Plan Selection

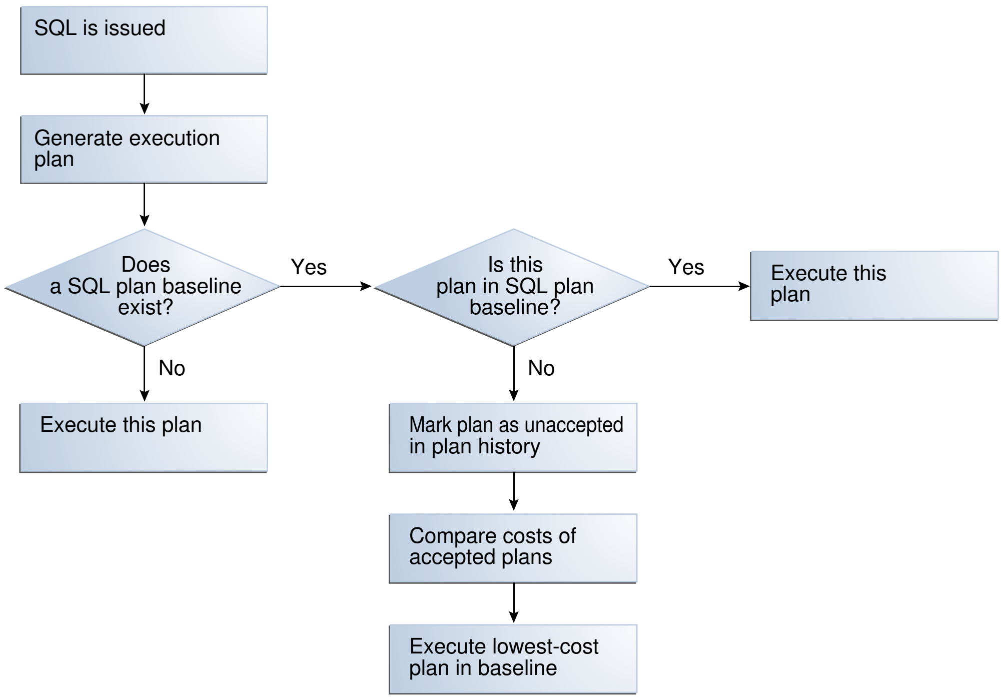
SQL plan selection is the optimizer ability to detect plan changes based on stored plan history, and the use of SQL plan baselines to select plans to avoid potential performance regressions.

When the database performs a hard parse of a SQL statement, the optimizer generates a best-cost plan. By default, the optimizer then attempts to find a matching plan in the SQL plan baseline for the statement. If no plan baseline exists, then the database runs the statement with the best-cost plan.

If a plan baseline exists, then the optimizer behavior depends on whether the newly generated plan is in the plan baseline:

- If the new plan is in the baseline, then the database executes the statement using the found plan.
- If the new plan is *not* in the baseline, then the optimizer marks the newly generated plan as unaccepted and adds it to the plan history. Optimizer behavior depends on the contents of the plan baseline:
  - If fixed plans exist in the plan baseline, then the optimizer uses the fixed plan with the lowest cost.
  - If no fixed plans exist in the plan baseline, then the optimizer uses the baseline plan with the lowest cost.
  - If no reproducible plans exist in the plan baseline, which could happen if every plan in the baseline referred to a dropped index, then the optimizer uses the newly generated cost-based plan.

Figure 28-3 Decision Tree for SQL Plan Selection


SQL is issued
Generate execution plan
Does a SQL plan baseline exist?
Yes
No
Execute this plan
Is this plan in SQL plan baseline?
Yes
Execute this plan
No
Mark plan as unaccepted in plan history
Compare costs of accepted plans
Execute lowest-cost plan in baseline

See Also:

"Fixed Plans"

28.5 Plan Evolution

In general, SQL **plan evolution** is the process by which the optimizer verifies new plans and adds them to an existing SQL plan baseline.

28.5.1 Purpose of Plan Evolution

Typically, a SQL plan baseline for a statement starts with one accepted plan.

However, some SQL statements perform well when executed with different plans under different conditions. For example, a SQL statement with bind variables whose values result in different selectivities may have several optimal plans. Creating a materialized view or an index or repartitioning a table may make current plans more expensive than other plans.

If new plans were never added to SQL plan baselines, then the performance of some SQL statements might degrade. Thus, it is sometimes necessary to evolve newly accepted plans into SQL plan baselines. Plan evolution prevents performance regressions by verifying the performance of a new plan before including it in a SQL plan baseline.

28.5.2 How Plan Evolution Works

Plan evolution involves both verifying and adding plans.

Specifically, plan evolution consists of the following distinct steps:

1. Verifying

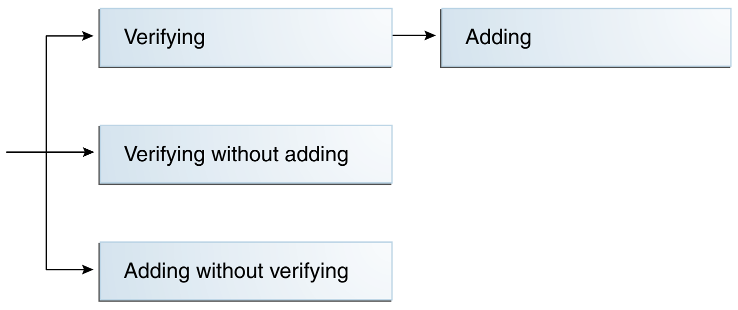
   The optimizer ensures that unaccepted plans perform at least as well as accepted plans in a SQL plan baseline (known as plan verification).

2. Adding

   After the database has proved that unaccepted plans perform as well as accepted plans, the database adds the plans to the baseline.

In the standard case of plan evolution, the optimizer performs the preceding steps sequentially, so that a new plan is not usable by SQL plan management until the optimizer verifies plan performance relative to the SQL plan baseline. However, you can configure SQL plan management to perform one step without performing the other. The following graphic shows the possible paths for plan evolution.

Figure 28-4   Plan Evolution



28.5.3 PL/SQL Subprograms for Plan Evolution

The `DBMS_SPM` package provides procedures and functions for plan evolution.

These subprograms use the task infrastructure. For example, `CREATE_EVOLVE_TASK` creates an evolution task, whereas `EXECUTE_EVOLVE_TASK` executes it. All task evolution subprograms have the string `EVOLVE_TASK` in the name.

Use the evolve procedures on demand, or configure the subprograms to run automatically. The automatic maintenance task `SYS_AUTO_SPM_EVOLVE_TASK` executes daily in the scheduled maintenance window. The task perform the following actions automatically:

1. Selects and ranks unaccepted plans for verification
2. Accepts each plan if it satisfies the performance threshold

See Also:

- "Managing the SPM Evolve Advisor Task"
- "Evolving SQL Plan Baselines Manually"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package

28.6 Storage Architecture for SQL Plan Management

The SQL plan management infrastructure records the signatures of parsed statements, and both accepted and unaccepted plans.

28.6.1 SQL Management Base

The **SQL management base (SMB)** is a logical repository in the data dictionary.

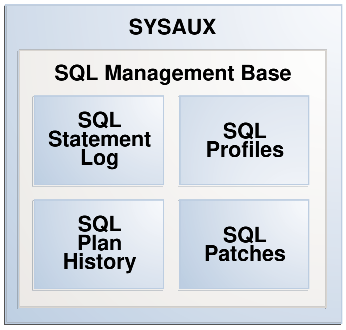
The SMB contains the following:

- SQL statement log, which contains only SQL IDs
- SQL plan history, which includes the SQL plan baselines
- SQL profiles
- SQL patches

The SMB stores information that the optimizer can use to maintain or improve SQL performance.

The SMB resides in the `SYSAUX` tablespace and uses automatic segment-space management. Because the SMB is located entirely within the `SYSAUX` tablespace, the database does not use SQL plan management and SQL tuning features when this tablespace is unavailable.

Figure 28-5 SMB Architecture



SMB data related to a PDB is stored in the PDB, and is included if the PDB is unplugged. A common user whose current container is the CDB root can view SMB data for PDBs. A user whose current container is a PDB can view the SMB data for the PDB only.

See Also:

Oracle Database Administrator’s Guide to learn about the `SYSAUX` tablespace

28.6.2 SQL Statement Log

When automatic SQL plan capture is enabled, the **SQL statement log** contains the signature of statements that the optimizer has evaluated over time.

A SQL signature is a numeric hash value computed using a SQL statement text that has been normalized for case insensitivity and white space. When the optimizer parses a statement, it creates signature.

During automatic capture, the database matches this signature against the SQL statement log (`SQLLOG$`) to determine whether the signature has been observed before. If it has not, then the database adds the signature to the log. If the signature is already in the log, then the database has confirmation that the statement is a repeatable SQL statement.

Note:

If a filter excludes a statement, then its signature is also excluded from the log.

Example 28-1 Logging SQL Statements

This example illustrates how the database tracks statements in the statement log and creates baselines automatically for repeatable statements. An initial query of the statement log shows no tracked SQL statements. After a query of `hr.jobs` for `AD_PRES`, the log shows one tracked statement.

```
SQL> ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true;

System altered.

SQL> SELECT * FROM SQLLOG$;

no rows selected

SQL> SELECT job_title FROM hr.jobs WHERE job_id = 'AD_PRES';

JOB_TITLE
-----------------------------------
President

SQL> SELECT * FROM SQLLOG$;

 SIGNATURE     BATCH#
---------- ----------
1.8096E+19          1
```

Now the session executes a different `jobs` query. The log shows two tracked statements:

```
SQL> SELECT job_title FROM hr.jobs WHERE job_id='PR_REP';

JOB_TITLE
-----------------------------------
Public Relations Representative

SQL> SELECT * FROM SQLLOG$;

 SIGNATURE     BATCH#
---------- ----------
1.7971E+19          1
1.8096E+19          1
```

A query of `DBA_SQL_PLAN_BASELINES` shows that no baseline for either statement exists because neither statement is repeatable:

```
SQL> SELECT SQL_HANDLE, SQL_TEXT
  2  FROM DBA_SQL_PLAN_BASELINES
  3  WHERE SQL_TEXT LIKE 'SELECT job_title%';

no rows selected
```

The session executes the query for `job_id='PR_REP'` a second time. Because this statement is now repeatable, and because automatic SQL plan capture is enabled, the database creates a plan baseline for this statement. The query for `job_id='AD_PRES'` has only been executed once, so no plan baseline exists for it.

```
SQL> SELECT job_title FROM hr.jobs WHERE job_id='PR_REP';

JOB_TITLE
-----------------------------------
Public Relations Representative

SQL> SELECT SQL_HANDLE, SQL_TEXT
  2  FROM DBA_SQL_PLAN_BASELINES
  3  WHERE SQL_TEXT LIKE 'SELECT job_title%';

SQL_HANDLE           SQL_TEXT
-------------------- --------------------
SQL_f9676a330f972dd5 SELECT job_title FRO
                     M hr.jobs WHERE job_
                     id='PR_REP'
```

See Also:

- "Automatic Initial Plan Capture"
- *Oracle Database Reference* to learn about `DBA_SQL_PLAN_BASELINES`

28.6.3 SQL Plan History

The **SQL plan history** is the set of captured SQL execution plans. The history contains both SQL plan baselines and unaccepted plans.

In SQL plan management, the database detects new SQL execution plans for existing SQL plan baselines and records the new plan in the history so that they can be evolved (verified). Evolution is initiated automatically by the database or manually by the DBA.

Starting in Oracle Database 12c, the SMB stores the execution plans for all SQL statements in the SQL plan history. The `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE` function fetches and displays the plan from the SMB. For plans created before Oracle Database 12c, the function must compile the SQL statement and generate the plan because the SMB does not store it.

See Also:

- "Displaying Plans in a SQL Plan Baseline"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE` function

28.6.3.1 Enabled Plans

An **enabled plan** is a plan that is eligible for use by the optimizer.

When plans are loaded with the `enabled` parameter set to `YES` (default), the database automatically marks the resulting SQL plan baselines as enabled, even if they are unaccepted. You can manually change an enabled plan to a disabled plan, which means the optimizer can no longer use the plan even if it is accepted.

28.6.3.2 Accepted Plans

An **accepted plan** is a plan that is in a SQL plan baseline for a SQL statement and thus available for use by the optimizer. An accepted plan contains a set of hints, a plan hash value, and other plan-related information.

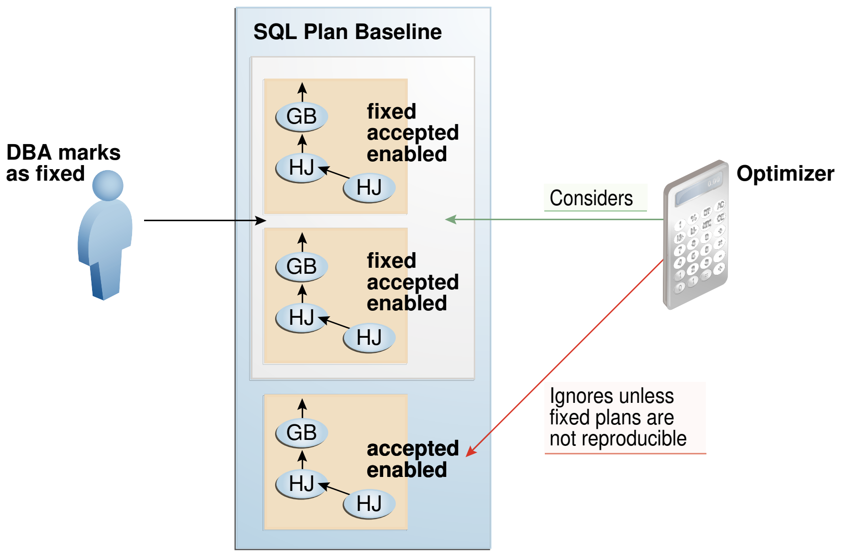
The SQL plan history for a statement contains all plans, both accepted and unaccepted. After the optimizer generates the first accepted plan in a plan baseline, every subsequent unaccepted plan is added to the plan history, awaiting verification, but is not in the SQL plan baseline.

28.6.3.3 Fixed Plans

A **fixed plan** is an accepted plan that is marked as preferred, so that the optimizer considers only the fixed plans in the baseline. Fixed plans influence the plan selection process of the optimizer.

Assume that three plans exist in the SQL plan baseline for a statement. You want the optimizer to give preferential treatment to only two of the plans. As shown in the following figure, you mark these two plans as fixed so that the optimizer uses only the best plan from these two, ignoring the other plans.

Figure 28-6 Fixed Plans



If new plans are added to a baseline that contains at least one enabled fixed plan, then the optimizer cannot use the new plans until you manually declare them as fixed.

29
Managing SQL Plan Baselines

This chapter explains the concepts and tasks relating to SQL plan management using the `DBMS_SPM` package.

See Also:

- "Migrating Stored Outlines to SQL Plan Baselines"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about `DBMS_SPM`

29.1 About Managing SQL Plan Baselines

This topic describes the available interfaces and basic tasks for SQL plan management.

29.1.1 User Interfaces for SQL Plan Management

You can access the `DBMS_SPM` package through Cloud Control or through the command line.

29.1.1.1 Accessing the SQL Plan Baseline Page in Cloud Control

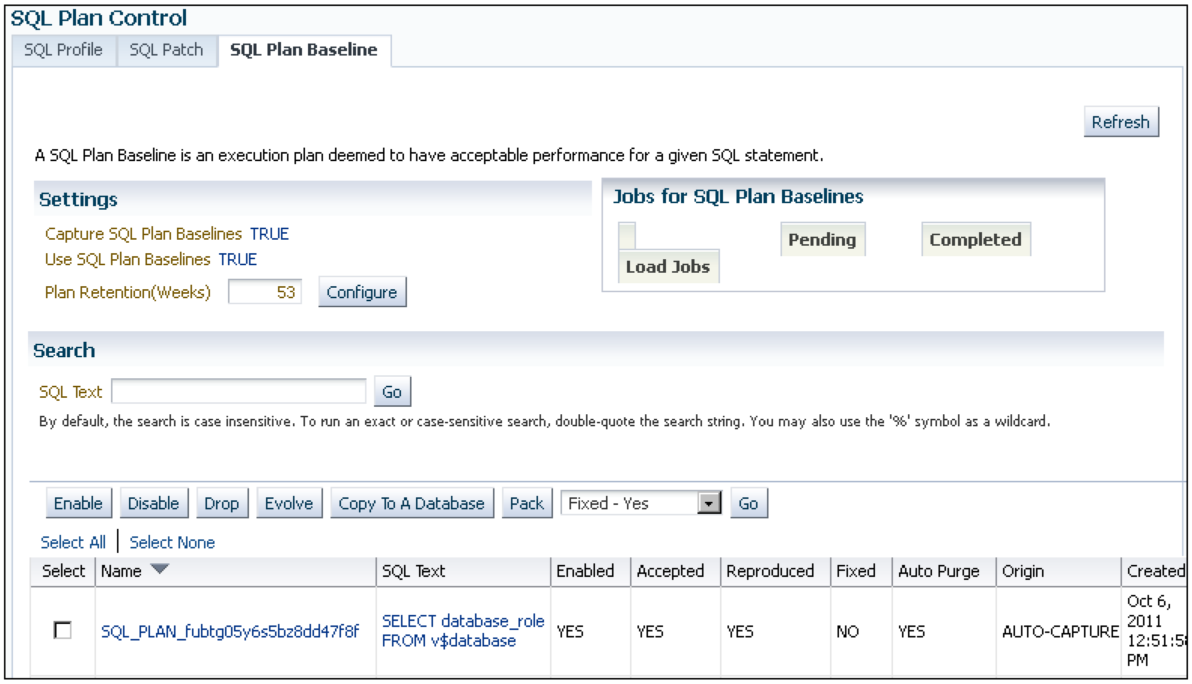
The SQL Plan Control page in Cloud Control is a GUI that shows information about SQL profiles, SQL patches, and SQL plan baselines.

To access the SQL Plan Baseline page:

1. Log in to Cloud Control with the appropriate credentials.
2. Under the **Targets** menu, select **Databases**.
3. In the list of database targets, select the target for the Oracle Database instance that you want to administer.
4. If prompted for database credentials, then enter the minimum credentials necessary for the tasks you intend to perform.
5. From the **Performance** menu, select **SQL**, then **SQL Plan Control**.

   The SQL Plan Control page appears.
6. Click **Files** to view the SQL Plan Baseline subpage, shown in Figure 29-1.

Figure 29-1 SQL Plan Baseline Subpage



A SQL Plan Baseline is an execution plan deemed to have acceptable performance for a given SQL statement.

Capture SQL Plan Baselines TRUE
Use SQL Plan Baselines TRUE
Plan Retention(Weeks) 53

SQL Text

By default, the search is case insensitive. To run an exact or case-sensitive search, double-quote the search string. You may also use the '%' symbol as a wildcard.

Select	Name	SQL Text	Enabled	Accepted	Reproduced	Fixed	Auto Purge	Origin	Created
☐	SQL_PLAN_fubtg05y6s5bz8dd47f8f	SELECT database_role FROM v$database	YES	YES	YES	NO	YES	AUTO-CAPTURE	Oct 6, 2011 12:51:5 PM

You can perform most SQL plan management tasks in this page or in pages accessed through this page.

See Also:

- Cloud Control context-sensitive online help to learn about the options on the SQL Plan Baseline subpage
- "Managing the SPM Evolve Advisor Task"

29.1.1.2 DBMS_SPM Package

On the command line, use the `DBMS_SPM` and `DBMS_XPLAN` PL/SQL packages to perform most SQL plan management tasks.

The following table describes the most relevant `DBMS_SPM` procedures and functions for creating, dropping, and loading SQL plan baselines.

Table 29-1 DBMS_SPM Procedures and Functions

Procedure or Function	Description
`CONFIGURE`	This procedure changes configuration options for the SMB in name/value format.
`CREATE_STGTAB_BASELINE`	This procedure creates a staging table that enables you to transport SQL plan baselines from one database to another.

Table 29-1 (Cont.) DBMS_SPM Procedures and Functions

Procedure or Function	Description
`DROP_SQL_PLAN_BASELINE`	This function drops some or all plans in a plan baseline.
`LOAD_PLANS_FROM_CURSOR_CACHE`	This function loads plans in the shared SQL area (also called the **cursor cache**) into SQL plan baselines.
`LOAD_PLANS_FROM_SQLSET`	This function loads plans in an STS into SQL plan baselines.
`LOAD_PLANS_FROM_AWR`	This function loads plans from AWR into SQL plan baselines.
`PACK_STGTAB_BASELINE`	This function packs SQL plan baselines, which means that it copies them from the SMB into a staging table.
`UNPACK_STGTAB_BASELINE`	This function unpacks SQL plan baselines, which means that it copies SQL plan baselines from a staging table into the SMB.

Also, you can use `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE` to show one or more execution plans for the SQL statement identified by SQL handle.

See Also:

- "About the DBMS_SPM Evolve Functions" describes the functions related to SQL plan evolution.
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` and `DBMS_XPLAN` packages

29.1.2 Basic Tasks in SQL Plan Management

This topic explains the basic tasks in using SQL plan management to prevent performance regressions and enable the optimizer to consider new execution plans.

The tasks are as follows:

- Set initialization parameters to control whether the database captures and uses SQL plan baselines, and whether it evolves new plans.

  See "Configuring SQL Plan Management".
- Display plans in a SQL plan baseline.

  See "Displaying Plans in a SQL Plan Baseline".
- Manually load plans into SQL plan baselines.

  Load plans from AWR, SQL tuning sets, the shared SQL area, a staging table, or stored outlines.

  See "Loading SQL Plan Baselines".
- Manually evolve plans into SQL plan baselines.

  Use PL/SQL to verify the performance of specified plans and add them to plan baselines.

  See "Evolving SQL Plan Baselines Manually".

- Drop all or some plans in SQL plan baselines.

  See "Dropping SQL Plan Baselines".

- Manage the SMB.

  Alter disk space limits and change the length of the plan retention policy.

  See "Managing the SQL Management Base".

- Migrate stored outlines to SQL plan baselines.

  See "Migrating Stored Outlines to SQL Plan Baselines".

29.2 Configuring SQL Plan Management

You can configure the capture and use of SQL plan baselines, and the SPM Evolve Advisor task.

29.2.1 Configuring the Capture and Use of SQL Plan Baselines

You control SQL plan management with the initialization parameters `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` and `OPTIMIZER_USE_SQL_PLAN_BASELINES`.

The default values are as follows:

- `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=false`

  For any repeatable SQL statement that does not already exist in the plan history, the database does *not* automatically create an initial SQL plan baseline for the statement.

  If `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true`, then you can use the `DBMS_SPM.CONFIGURE` procedure to configure filters that determine which statements are eligible for plan capture. By default, no filters are configured, which means that all repeatable statements are eligible for plan capture.

- `OPTIMIZER_USE_SQL_PLAN_BASELINES=true`

  For any SQL statement that has an existing SQL plan baseline, the database automatically adds new plans to the SQL plan baseline as unaccepted plans.

Note:

The settings of the preceding parameters are independent of each other. For example, if `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` is `true`, then the database creates initial plan baselines for new statements even if `OPTIMIZER_USE_SQL_PLAN_BASELINES` is `false`.

If the default behavior is what you intend, then skip this section.

The following sections explain how to change the default parameter settings from the command line. If you use Cloud Control, then set these parameters in the SQL Plan Baseline subpage.

See Also:

- "Figure 29-1"
- "Automatic Initial Plan Capture"
- "Plan Selection"

29.2.1.1 Enabling Automatic Initial Plan Capture for SQL Plan Management

Setting the `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` initialization parameter to `true` is all that is necessary for the database to automatically create an initial SQL plan baseline for any eligible SQL statement not already in the plan history.

By default, the database considers all repeatable SQL statements as eligible for capture, with the following exceptions:

- `CREATE TABLE` when the `AS SELECT` clause is *not* specified
- `DROP TABLE`
- `INSERT INTO ... VALUES`

Caution:

By default, when automatic baseline capture is enabled, the database creates a SQL plan baseline for *every* eligible repeatable statement, including all recursive SQL and monitoring SQL. Thus, automatic capture may result in the creation of an extremely large number of plan baselines. To limit the statements that are eligible for plan baselines, configure filters using the `DBMS_SPM.CONFIGURE` procedure.

The `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` parameter does not control the automatic addition of newly discovered plans to a previously created SQL plan baseline.

To enable automatic initial plan capture for SQL plan management:

1. In SQL*Plus, log in to the database with the necessary privileges.
2. Show the current settings for SQL plan management.

   For example, connect SQL*Plus to the database with administrator privileges and execute the following command (sample output included):

   ```
   SHOW PARAMETER SQL_PLAN
   ```

   The following sample output shows that automatic initial plan capture is disabled:

   ```
   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   optimizer_capture_sql_plan_baselines boolean     FALSE
   optimizer_use_sql_plan_baselines     boolean     TRUE
   ```

If the parameters are set as you intend, then skip the remaining steps.

3. To enable the automatic recognition of repeatable SQL statements and the generation of SQL plan baselines for these statements, enter the following statement:

```
ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true;
```

See Also:

Oracle Database Reference to learn more about `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES`

29.2.1.2 Configuring Filters for Automatic Plan Capture

If `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true`, then you can use the `DBMS_SPM.CONFIGURE` procedure to create an **automatic capture filter** for repeatable statements.

An automatic filter enables you to capture only statements that you want, and exclude noncritical statements. This technique saves space in the `SYSAUX` tablespace.

The following table describes the relevant parameters of the `DBMS_SPM.CONFIGURE` procedure.

Table 29-2 DBMS_SPM.CONFIGURE Parameters

Parameter	Description
`parameter_name`	The type of filter for automatic capture. Possible values are `AUTO_CAPTURE_SQL_TEXT`, `AUTO_CAPTURE_PARSING_SCHEMA_NAME`, `AUTO_CAPTURE_MODULE`, and `AUTO_CAPTURE_ACTION`.
`parameter_value`	The search criteria for the automatic capture filter. When `parameter_name` is set to `AUTO_CAPTURE_SQL_TEXT`, the search pattern depends on the `allow` setting: • `LIKE` The parameter uses this pattern when `allow=>true`. • `NOT LIKE` The parameter uses this pattern when `allow=>false`. For all other non-null `parameter_name` values, the search pattern depends on the `allow` setting: • `=` The parameter uses this pattern when `allow=>true`. • `<>` The parameter uses this pattern when `allow=>false`. A null value removes the filter for `parameter_name` entirely.
`allow`	Whether to include (`true`) or exclude (`false`) matching SQL statements and plans. If null, then the procedure ignores the specified parameter.

You can configure multiple parameters of different types. Also, you can specify multiple values for the same parameter in separate statements, which the database combines. The settings are additive: one parameter setting does not override a previous setting. For example, the following filter captures SQL in the parsing schema `SYS` or `SYSTEM`:

```
EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_PARSING_SCHEMA_NAME','SYS',true);
EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_PARSING_SCHEMA_NAME','SYSTEM',true);
```

However, you cannot configure multiple values for the same parameter in the same procedure. For example, you cannot specify multiple SQL text strings for `AUTO_CAPTURE_SQL_TEXT`.

The `DBA_SQL_MANAGEMENT_CONFIG` view shows the current parameter values.

This tutorial assumes the following:

- The `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` initialization parameter is set to `true`.
- You want to include only statements parsed in the `sh` schema to be eligible for baselines.
- You want to exclude statements that contain the text `TEST_ONLY`.

To filter out all statements except those parsed in the sh schema:

1. Connect SQL*Plus to the database with the appropriate privileges.
2. To remove any existing filters for parsing schema and SQL text, execute the following PL/SQL programs:

   ```
   EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_PARSING_SCHEMA_NAME',null,true);
   EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_SQL_TEXT',null,true);
   ```

3. Include only statements parsed in the `sh` schema for consideration for automatic capture:

   ```
   EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_PARSING_SCHEMA_NAME','sh',true);
   ```

4. Exclude any statement that contains the text `TEST_ONLY` from consideration for automatic capture:

   ```
   EXEC DBMS_SPM.CONFIGURE('AUTO_CAPTURE_SQL_TEXT','%TEST_ONLY%',false);
   ```

5. Optionally, to confirm the filters, query `DBA_SQL_MANAGEMENT_CONFIG`.

   For example, use the following query (sample output included):

   ```
   COL PARAMETER_NAME FORMAT a32
   COL PARAMETER_VALUE FORMAT a32

   SELECT PARAMETER_NAME, PARAMETER_VALUE
   FROM   DBA_SQL_MANAGEMENT_CONFIG
   WHERE  PARAMETER_NAME LIKE '%AUTO%';

   PARAMETER_NAME                   PARAMETER_VALUE
   -------------------------------- --------------------------------
   AUTO_CAPTURE_PARSING_SCHEMA_NAME parsing_schema IN (SH)
   AUTO_CAPTURE_MODULE
   ```

```
AUTO_CAPTURE_ACTION
AUTO_CAPTURE_SQL_TEXT            (sql_text NOT LIKE %TEST_ONLY%)
```

See Also:

- "Automatic Initial Plan Capture"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_SPM.CONFIGURE` procedure
- *Oracle Database Reference* to learn more about the `DBA_SQL_MANAGEMENT_CONFIG` view

29.2.1.3 Disabling All SQL Plan Baselines

When you set the `OPTIMIZER_USE_SQL_PLAN_BASELINES` initialization parameter to `false`, the database does not use *any* plan baselines in the database.

Typically, you might want to disable one or two plan baselines, but not all of them. A possible use case might be testing the benefits of SQL plan management.

To disable all SQL plan baselines in the database:

1. Connect SQL*Plus to the database with the appropriate privileges, and then show the current settings for SQL plan management.

   For example, connect SQL*Plus to the database with administrator privileges and execute the following command (sample output included):

   ```
   SQL> SHOW PARAMETER SQL_PLAN

   NAME                                 TYPE        VALUE
   ------------------------------------ ----------- -----
   optimizer_capture_sql_plan_baselines boolean     FALSE
   optimizer_use_sql_plan_baselines     boolean     TRUE
   ```

   If the parameters are set as you intend, then skip the remaining steps.

2. To ignore all existing plan baselines enter the following statement:

   ```
   SQL> ALTER SYSTEM SET OPTIMIZER_USE_SQL_PLAN_BASELINES=false
   ```

See Also:

Oracle Database Reference to learn about the SQL plan baseline initialization parameters

29.2.2 Managing the SPM Evolve Advisor Task

SPM Evolve Advisor is a SQL advisor that evolves plans that have recently been added to the SQL plan baseline. The advisor simplifies plan evolution by eliminating the requirement to do it manually.

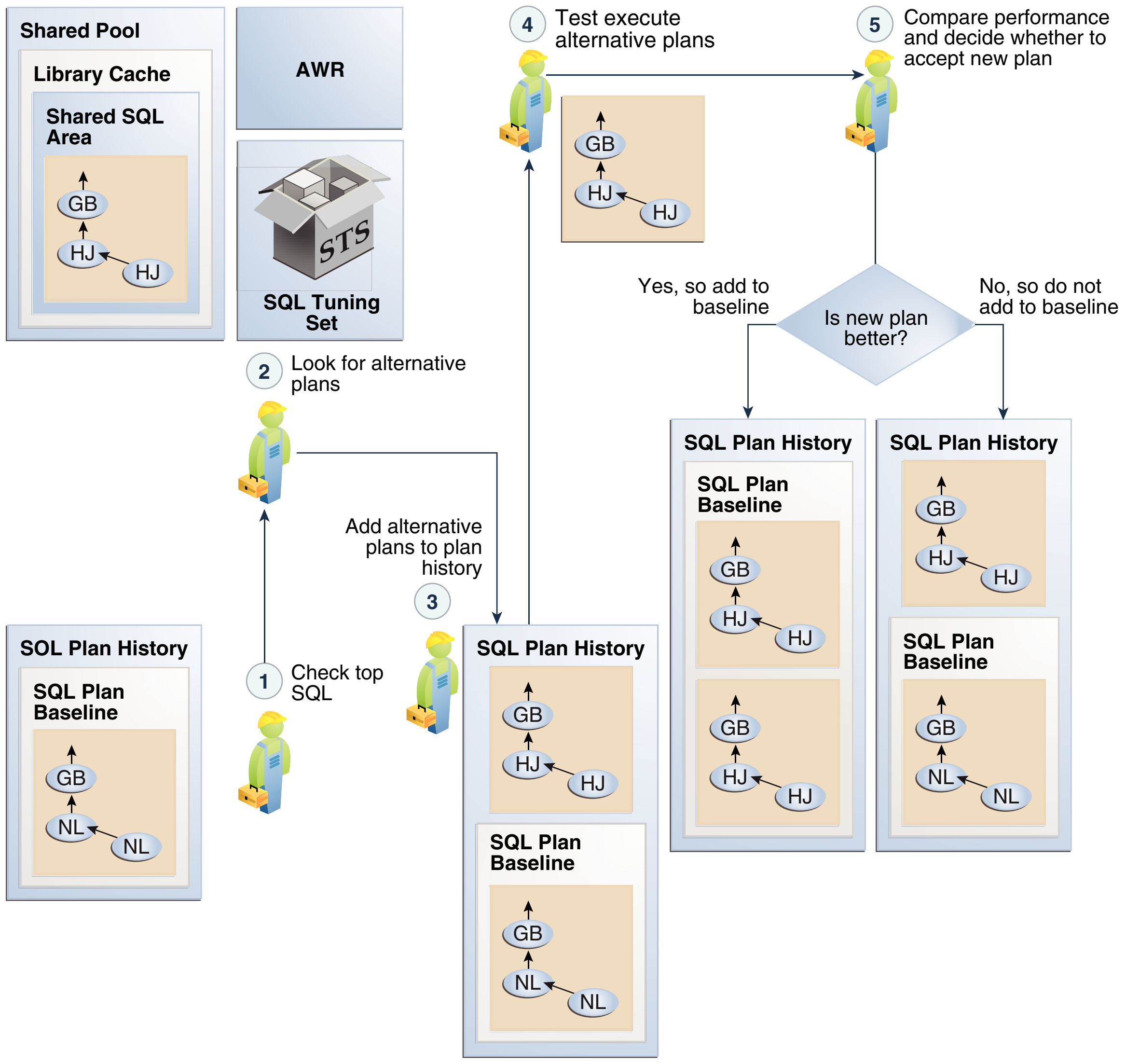
29.2.2.1 About the SPM Evolve Advisor Task

By default, `SYS_AUTO_SPM_EVOLVE_TASK` runs daily in the scheduled maintenance window. Optionally, you can configure it to run hourly.

A SQL plan baseline prevents performance regressions caused by suboptimal plans. If a SQL statement does not have a SQL plan baseline, and if the `alternate_plan_baseline` parameter is set to `AUTO`, then SQM Evolve Advisor can sometimes resolve such performance regressions automatically. The advisor compares all available plans and chooses the best-performing plan as the baseline.

The following figure shows the workflow for Automatic SPM Evolve Advisor:

Figure 29-2 Automatic SPM Evolve Advisor


Shared Pool
Library Cache
Shared SQL Area
GB
HJ
HJ
AWR
STS
SQL Tuning Set
4 Test execute alternative plans
5 Compare performance and decide whether to accept new plan
Yes, so add to baseline
Is new plan better?
No, so do not add to baseline
2 Look for alternative plans
Add alternative plans to plan history
3
1 Check top SQL
SQL Plan History
SQL Plan Baseline
NL

Whenever it runs in the maintenance window, SPM Evolve Advisor performs the following tasks:

1. Checks AWR for top SQL

   AWR stores the most resource-intensive SQL statements. SPM Evolve Advisor searches AWR for statements that are most likely to benefit from SQL plan baselines, and then adds plans for these statements to the baselines.

2. Looks for alternative plans in all available sources

   By default (`alternate_plan_source=AUTO`), the automatic task searches all available repositories for plans that are not yet in the SMB plan history. The setting for `alternate_plan_source` is shown in the `DBA_ADVISORS_PARAMETERS` view.

3. Adds unaccepted plans to the plan history

   These plans are not yet in the SQL plan baseline for any SQL statement.

4. Tests the execution of as many plans as possible during the maintenance window

For every alternative plan, the database test executes the statement and records the performance statistics. The goal is to use a cost-based algorithm to compare the performance of every alternative plan with the plan that the optimizer would otherwise choose.

5. Performs either of the following actions, depending on whether the alternative plan performs better than the current plan:
    - If performance is better, then SPM Evolve Advisor accepts the plan. The alternative plan is now in the baseline.
    - If performance is worse, then the plan remains in the statement history, but not the baseline.

Example 29-1 Out-of-Range Query

Assume that an application issues a new, long-running query of `sh.products`. The query references the `prod_list_price` column using a bind variable. The database performs the following steps:

1. The optimizer chooses an optimal plan for this query based on fresh statistics, which show `prod_list_price` with a maximum list price of $1299.99. The optimal plan resides in AWR.
2. An application adds more products to the `sh.products` table, and changes the list prices of many products.
3. The application issues the original query, which the optimizer reparses.

    In this execution of the query, the bind variable sets the list price to $1500, which is higher than the $1299.99 maximum value recorded in the table statistics. This is known as an *out-of-range query*.
4. The optimizer chooses a suboptimal plan for the out-of-range query, causing a performance regression.

    The optimizer attempts to avoid performance regressions by allowing for out-of-range conditions, but is sometimes unsuccessful, as in this example. The result is a suboptimal plan.
5. The database resolves the performance regression as follows:
    - SPM Evolve Advisor identifies the long-running query as a candidate for a SQL plan baseline.
    - SPM Evolve Advisor finds both plans, the original plan in AWR and the suboptimal plan for the out-of-range query, and determines that the original plan performs better.
    - SQL plan management adds the original plan for the query to the SQL plan baseline. Consequently, the optimizer will not use the regressed plan.

See Also:

- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_SPM.SET_EVOLVE_TASK_PARAMETER` procedure

29.2.2.2 Enabling and Disabling the Automatic SPM Evolve Advisor Task

No separate scheduler client exists for the Automatic SPM Evolve Advisor task.

One client controls both Automatic SQL Tuning Advisor and Automatic SPM Evolve Advisor. Thus, the same task enables or disables both. You can also disable it using `DBMS_SPM.SET_EVOLVE_TASK_PARAMETER`.

To disable the Automatic SPM Evolve Advisor task:

1. Log in to the database with the appropriate privileges.
2. Set the `ALTERNATE_PLAN_BASELINE` parameter to null:

```
BEGIN
   DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
      task_name => 'SYS_AUTO_SPM_EVOLVE_TASK',
      parameter => 'ALTERNATE_PLAN_BASELINE',
      value => '');
END;
/
```

3. Set the `ALTERNATE_PLAN_SOURCE` parameter to an empty string:

```
BEGIN
   DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
      task_name => 'SYS_AUTO_SPM_EVOLVE_TASK',
      parameter => 'ALTERNATE_PLAN_SOURCE',
      value => '');
END;
/
```

See Also:

"Enabling and Disabling the Automatic SQL Tuning Task" to learn how to enable and disable Automatic SPM Evolve Advisor

29.2.2.3 Configuring the Automatic SPM Evolve Advisor Task

Configure automatic plan evolution by using the `DBMS_SPM` package.

Overview of the Automatic SPM Evolve Advisor Task

Specify the automatic task parameters using the `SET_EVOLVE_TASK_PARAMETER` procedure. The following table describes some procedure parameters.

Table 29-3 DBMS_SPM.SET_EVOLVE_TASK_PARAMETER Parameters

Parameter	Description	Default
`alternate_plan_source`	Determines which sources to search for additional plans: • `AUTO` (the database selects the source automatically) • `AUTOMATIC_WORKLOAD_REPOSITORY` • `CURSOR_CACHE` • `SQL_TUNING_SET` You can combine multiple values with the plus sign (`+`).	The default depends on whether the SPM Evolve Advisor task is automated or manual: • If automated, the default is `AUTO`. • If manual, the default is `CURSOR_CACHE+AUTOMATIC_WORKLOAD_REPOSITORY`.
`alternate_plan_baseline`	Determines which alternative plans should be loaded: • `AUTO` lets Autonomous Database choose whether to load plans for statements with or without baselines. • `EXISTING` loads alternate plans with for statements with existing baselines. • `NEW` loads alternative plans for statements without a baseline, in which case a new baseline is created. You can combine multiple values with the plus sign (`+`), as in `EXISTING+NEW`.	`EXISTING`
`alternate_plan_limit`	Specifies the maximum number of plans to load in total (that is, not the limit for each SQL statement).	The default depends on whether the SPM Evolve Advisor task is automated or manual: • If automated, the default is `UNLIMITED`. • If manual, the default is `10`.
`accept_plans`	Specifies whether to accept recommended plans automatically. When `ACCEPT_PLANS` is `true`, SQL plan management automatically accepts all plans recommended by the task. When `ACCEPT_PLANS` is `false`, the task verifies the plans and generates a report of its findings, but does not evolve the plans automatically. You can use a report to identify new SQL plan baselines and accept them manually.	`true` (regardless of whether the advisor is run automatically or manually)

Table 29-3 (Cont.) DBMS_SPM.SET_EVOLVE_TASK_PARAMETER Parameters

Parameter	Description	Default
`time_limit`	Global time limit in seconds. This is the total time allowed for the task.	The default depends on whether the SPM Evolve Advisor task is automated or manual: • If automated, the default is `3600`. • If manual, the default is `2147483646`.

> **Note:**
>
> See *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services

Assumptions

The tutorial in this section assumes the following:

- You can log in to the database as `SYS`. Because the `SYS_AUTO_SPM_EVOLVE_TASK` task is owned by `SYS`, only `SYS` can set task parameters.
- You want the database to accept plans automatically.
- You want the task to time out after 1200 seconds per execution.
- You want the evolve task to look for up to a maximum of 500 plans in the shared SQL area and AWR repository

To set automatic evolution task parameters:

1. Start SQL*Plus, and then log in to the database as `SYS`.
2. Query the current parameter settings for `SYS_AUTO_SPM_EVOLVE_TASK`.

   For example, connect SQL*Plus to the database with administrator privileges and execute the following query:

   ```
   COL PARAMETER_NAME FORMAT a25
   COL VALUE FORMAT a42
   SELECT PARAMETER_NAME, PARAMETER_VALUE AS "VALUE"
   FROM   DBA_ADVISOR_PARAMETERS
   WHERE  ( (TASK_NAME = 'SYS_AUTO_SPM_EVOLVE_TASK') AND
            ( (PARAMETER_NAME = 'ACCEPT_PLANS') OR
              (PARAMETER_NAME LIKE '%ALT%') OR
              (PARAMETER_NAME = 'TIME_LIMIT') ) );
   ```

   Sample output appears as follows:

   ```
   PARAMETER_NAME            VALUE
   ------------------------- ------------------------------------------
   ALTERNATE_PLAN_LIMIT      0
   ```

```
ALTERNATE_PLAN_SOURCE      CURSOR_CACHE+AUTOMATIC_WORKLOAD_REPOSITORY
ALTERNATE_PLAN_BASELINE    EXISTING
ACCEPT_PLANS               true
TIME_LIMIT                 3600
```

3. Set parameters using PL/SQL code of the following form:

   ```
   BEGIN
     DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
       task_name => 'SYS_AUTO_SPM_EVOLVE_TASK'
   ,   parameter => parameter_name
   ,   value     => value
   );
   END;
   /
   ```

   For example, the following PL/SQL block configures the `SYS_AUTO_SPM_EVOLVE_TASK` task to automatically accept plans, seek up a maximum of 500 plans in the shared SQL area and AWR repository, and time out after 20 minutes:

   ```
   BEGIN
     DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
       task_name => 'SYS_AUTO_SPM_EVOLVE_TASK'
   ,   parameter => 'TIME_LIMIT'
   ,   value     => '1200'
   );
     DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
       task_name => 'SYS_AUTO_SPM_EVOLVE_TASK'
   ,   parameter => 'ACCEPT_PLANS'
   ,   value     => 'true'
   );
     DBMS_SPM.SET_EVOLVE_TASK_PARAMETER(
       task_name => 'SYS_AUTO_SPM_EVOLVE_TASK'
   ,   parameter => 'ALTERNATE_PLAN_LIMIT'
   ,   value     => '500'
   );
   END;
   /
   ```

4. Optionally, confirm your changes by querying the current parameter settings for `SYS_AUTO_SPM_EVOLVE_TASK`.

   For example, execute the following query:

   ```
   SELECT PARAMETER_NAME, PARAMETER_VALUE AS "VALUE"
   FROM   DBA_ADVISOR_PARAMETERS
   WHERE  ( (TASK_NAME = 'SYS_AUTO_SPM_EVOLVE_TASK') AND
            ( (PARAMETER_NAME = 'ACCEPT_PLANS') OR
              (PARAMETER_NAME LIKE '%ALT%') OR
              (PARAMETER_NAME = 'TIME_LIMIT') ) );
   ```

Sample output appears as follows:

```
PARAMETER_NAME            VALUE
------------------------- -----------------------------------------
ALTERNATE_PLAN_LIMIT      500
ALTERNATE_PLAN_SOURCE     CURSOR_CACHE+AUTOMATIC_WORKLOAD_REPOSITORY
ALTERNATE_PLAN_BASELINE   EXISTING
ACCEPT_PLANS              true
TIME_LIMIT                1200
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for complete reference information for `DBMS_SPM.SET_EVOLVE_TASK_PARAMETER`
- *Oracle Database Reference* to learn more about the `DBA_ADVISOR_PARAMETERS` view

29.2.2.4 Configuring the High-Frequency Automatic SPM Evolve Advisor Task

You can configure automatic plan evolution to occur more frequently.

See Also:

Oracle Database Licensing Information User Manual for details on which features are supported for different editions and services

29.2.2.4.1 About the High-Frequency Automatic SPM Evolve Advisor Task

The high-frequency SPM Evolve Advisor task complements the standard Automatic SPM Evolve Advisor task.

By default, `SYS_AUTO_SPM_EVOLVE_TASK` runs daily in the scheduled AutoTask maintenance window. If data changes frequently between two consecutive task executions, then the optimizer may choose suboptimal plans. For example, if product list prices change more frequently than executions of `SYS_AUTO_SPM_EVOLVE_TASK`, then more out-of-range queries may occur, possibly leading to suboptimal plans.

When you enable the high-frequency Automatic SPM Evolve Advisor task, `SYS_AUTO_SPM_EVOLVE_TASK` runs more frequently, performing the same operations during every execution. The high-frequency task runs every hour and runs for no longer than 30 minutes. These settings are not configurable. The frequent executions mean that the optimizer has more opportunities to find and evolve better performing plans.

Both the standard Automatic SPM Evolve Advisor task and high-frequency task have the same name: `SYS_AUTO_SPM_EVOLVE_TASK`. In `DBA_ADVISOR_EXECUTIONS`, the two tasks are distinguished by execution name. The name of the standard task execution has the form `EXEC_number`, whereas the name of the high-frequency execution has the form `SYS_SPM_timestamp`.

`DBMS_SPM.CONFIGURE` enables the high-frequency task, but has no dependency on the SPM Evolve Advisor. The standard task and high-frequency task are independent and are scheduled through two different frameworks.

See Also:

- *Oracle Database Licensing Information User Manual* for details on which features are supported for different editions and services
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the `DBMS_SPM.CONFIGURE` procedure

29.2.2.4.2 Enabling the High-Frequency Automatic SPM Evolve Advisor Task: Tutorial

To enable and disable the high-frequency Automatic SPM Evolve Advisor task, use the `DBMS_SPM.CONFIGURE` procedure.

You can set `auto_spm_evolve_task` to any of the following values:

- `ON` — Enables the high-frequency SPM Evolve Advisor task.
- `OFF` — Disables the high-frequency SPM Evolve Advisor task. This is the default.
- `AUTO` — Allows the database to determine when to execute the high-frequency SPM Evolve Advisor task. In this release, `AUTO` is equivalent to `OFF`.

Note that the task interval and runtime are fixed and cannot be adjusted by the user.

To enable the high-frequency SPM Evolve Advisor task:

1. Log in to the database as `SYS`.
2. Query the current setting for `DBMS_SPM.CONFIGURE` (sample output included):

```
COL PARAMETER_NAME FORMAT a32
COL PARAMETER_VALUE FORMAT a32
SELECT PARAMETER_NAME, PARAMETER_VALUE
FROM   DBA_SQL_MANAGEMENT_CONFIG
WHERE  PARAMETER_NAME LIKE '%SPM%';

PARAMETER_NAME                   PARAMETER_VALUE
-------------------------------- --------------------------------
AUTO_SPM_EVOLVE_TASK             OFF
AUTO_SPM_EVOLVE_TASK_INTERVAL    3600
AUTO_SPM_EVOLVE_TASK_MAX_RUNTIME 1800
```

3. Enable the task.

   Execute the following PL/SQL code:

```
EXEC DBMS_SPM.CONFIGURE('AUTO_SPM_EVOLVE_TASK', 'ON');
```

4. To confirm that the task is enabled, query the current setting for `AUTO_SPM_EVOLVE_TASK` (sample output included):

```
COL PARAMETER_NAME FORMAT a32
COL PARAMETER_VALUE FORMAT a32
SELECT PARAMETER_NAME, PARAMETER_VALUE
FROM   DBA_SQL_MANAGEMENT_CONFIG
WHERE  PARAMETER_NAME = 'AUTO_SPM_EVOLVE_TASK';

PARAMETER_NAME                   PARAMETER_VALUE
-------------------------------- --------------------------------
AUTO_SPM_EVOLVE_TASK             ON
```

5. Optionally, wait a few hours, and then query the status of the task executions:

```
SET LINESIZE 150
COL TASK_NAME FORMAT a30
COL EXECUTION_NAME FORMAT a30

SELECT TASK_NAME, EXECUTION_NAME, STATUS
FROM   DBA_ADVISOR_EXECUTIONS
WHERE  TASK_NAME LIKE '%SPM%'
AND    (EXECUTION_NAME LIKE 'SYS_SPM%' OR EXECUTION_NAME LIKE
'EXEC_%')
ORDER BY EXECUTION_END;

TASK_NAME                      EXECUTION_NAME                 STATUS
------------------------------ ------------------------------
---------
SYS_AUTO_SPM_EVOLVE_TASK       SYS_SPM_2019-06-03/13:15:26
COMPLETED
SYS_AUTO_SPM_EVOLVE_TASK       SYS_SPM_2019-06-03/14:16:04
COMPLETED
SYS_AUTO_SPM_EVOLVE_TASK       EXEC_6
COMPLETED
SYS_AUTO_SPM_EVOLVE_TASK       SYS_SPM_2019-06-03/15:16:32
COMPLETED
SYS_AUTO_SPM_EVOLVE_TASK       SYS_SPM_2019-06-03/16:17:00
COMPLETED
...
```

In the preceding output, `EXEC_6` is the execution name of the standard SPM Automatic Advisor task. The other executions are of the high-frequency task.

29.3 Displaying Plans in a SQL Plan Baseline

To view the plans stored in the SQL plan baseline for a specific statement, use the `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE` function.

This function uses plan information stored in the plan history to display the plans. The following table describes the relevant parameters for `DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE`.

Table 29-4 DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE Parameters

Function Parameter	Description
`sql_handle`	SQL handle of the statement. Retrieve the SQL handle by joining the `V$SQL.SQL_PLAN_BASELINE` and `DBA_SQL_PLAN_BASELINES` views on the `PLAN_NAME` columns.
`plan_name`	Name of the plan for the statement.

This section explains how to show plans in a baseline from the command line. If you use Cloud Control, then display plan baselines from the SQL Plan Baseline subpage shown in Figure 29-1.

To display plans in a SQL plan baselines:

1. Connect SQL*Plus to the database with the appropriate privileges, and then obtain the SQL ID of the query whose plan you want to display.

   For example, assume that a SQL plan baseline exists for a `SELECT` statement with the SQL ID `31d96zzzpcys9`.

2. Query the plan by SQL ID.

   The following query displays execution plans for the statement with the SQL ID `31d96zzzpcys9`:

```
SELECT  PLAN_TABLE_OUTPUT
FROM    V$SQL s, DBA_SQL_PLAN_BASELINES b,
        TABLE(

DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE(b.sql_handle,b.plan_name,'basic')
        ) t
WHERE   s.EXACT_MATCHING_SIGNATURE=b.SIGNATURE
AND     b.PLAN_NAME=s.SQL_PLAN_BASELINE
AND     s.SQL_ID='31d96zzzpcys9';
```

   The sample query results are as follows:

```
PLAN_TABLE_OUTPUT
------------------------------------------------------------------------

------------------------------------------------------------------------
SQL handle: SQL_513f7f8a91177b1a
SQL text: select * from hr.employees where employee_id=100
------------------------------------------------------------------------

------------------------------------------------------------------------
Plan name: SQL_PLAN_52gvzja8jfysuc0e983c6         Plan id: 3236529094
Enabled: YES     Fixed: NO      Accepted: YES     Origin: AUTO-CAPTURE
------------------------------------------------------------------------

Plan hash value: 3236529094

-----------------------------------------------------
```

```
| Id  | Operation                   | Name          |
-----------------------------------------------------
|   0 | SELECT STATEMENT            |               |
|   1 |  TABLE ACCESS BY INDEX ROWID| EMPLOYEES     |
|   2 |   INDEX UNIQUE SCAN         | EMP_EMP_ID_PK |
-----------------------------------------------------
```

The results show that the plan for SQL ID `31d96zzzpcys` is named `SQL_PLAN_52gvzja8jfysuc0e983c6` and was captured automatically.

See Also:

- "SQL Management Base"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about additional parameters used by the `DISPLAY_SQL_PLAN_BASELINE` function

29.4 Loading SQL Plan Baselines

Using `DBMS_SPM`, you can initiate the bulk load of a set of existing plans into a SQL plan baseline.

29.4.1 About Loading SQL Plan Baselines

The `DBMS_SPM` package enables you to load plans from multiple sources.

The goal of this task is to load plans from the following sources:

- AWR

  Load plans from Automatic Workload Repository (AWR) snapshots. You must specify the beginning and ending of the snapshot range. Optionally, you can apply a filter to load only plan that meet specified criteria. By default, the optimizer uses the loaded plans the next time that the database executes the SQL statements.
- Shared SQL area

  Load plans for statements directly from the shared SQL area, which is in the shared pool of the SGA. By applying a filter on the module name, the schema, or the SQL ID you identify the SQL statement or set of SQL statements to capture. The optimizer uses the plans the next time that the database executes the SQL statements.

  Loading plans directly from the shared SQL area is useful when application SQL has been hand-tuned using hints. Because you probably cannot change the SQL to include the hint, populating the SQL plan baseline ensures that the application SQL uses optimal plans.
- SQL tuning set (STS)

  Capture the plans for a SQL workload into an STS, and then load the plans into the SQL plan baselines. The optimizer uses the plans the next time that the database executes the SQL statements. Bulk loading execution plans from an STS is an effective way to prevent plan regressions after a database upgrade.

- Staging table

  Use the DBMS_SPM package to define a staging table, DBMS_SPM.PACK_STGTAB_BASELINE to copy the baselines into a staging table, and Oracle Data Pump to transfer the table to another database. On the destination database, use DBMS_SPM.UNPACK_STGTAB_BASELINE to unpack the plans from the staging table and put the baselines into the SMB.

  A use case is the introduction of new SQL statements into the database from a new application module. A vendor can ship application software with SQL plan baselines for the new SQL. In this way, the new SQL uses plans that are known to give optimal performance under a standard test configuration. Alternatively, if you develop or test an application in-house, export the correct plans from the test database and import them into the production database.
- Stored outline

  Migrate stored outlines to SQL plan baselines. After the migration, you maintain the same plan stability that you had using stored outlines while being able to use the more advanced features provided by SQL Plan Management, such as plan evolution. See .

See Also:

- "Migrating Stored Outlines to SQL Plan Baselines"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the DBMS_SPM.PACK_STGTAB_BASELINE Function

29.4.2 Loading Plans from AWR

This topic explains how to load plans from AWR using PL/SQL.

Load plans with the LOAD_PLANS_FROM_AWR function of the DBMS_SPM package. The following table describes some function parameters.

Table 29-5 LOAD_PLANS_FROM_AWR Parameters

Function Parameter	Description
begin_snap	Number of the beginning snapshot in the range. Required.
end_snap	Number of the ending snapshot in the range. Required.
basic_filter	A filter applied to AWR to select only qualifying plans to be loaded. The default null means that all plans in AWR are selected. The filter can take the form of any WHERE clause predicate that can be specified against the column DBA_HIST_SQLTEXT.SQL_TEXT. An example is basic_filter => 'sql_text like ''SELECT /*LOAD_STS*/%'''.
fixed	Default NO means the loaded plans are used as nonfixed plans. YES means the loaded plans are fixed plans. "Plan Selection" explains that the optimizer chooses a fixed plan in the plan baseline over a nonfixed plan.

This section explains how to load plans using the command line. In Cloud Control, go to the SQL Plan Baseline subpage (shown in Figure 29-1) and click **Load** to load plan baselines from AWR.

This tutorial assumes the following:

- You want to load plans for the following query into the SMB:

  ```
  SELECT /*LOAD_AWR*/ *
  FROM   sh.sales
  WHERE  quantity_sold > 40
  ORDER BY prod_id;
  ```

- You want the loaded plans to be nonfixed.
- The user `sh` has privileges to query `DBA_HIST_SNAPSHOT` and `DBA_SQL_PLAN_BASELINES`, execute `DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT`, and execute `DBMS_SPM.LOAD_PLANS_FROM_AWR`.

To load plans from the shared SQL area:

1. Log in to the database with the appropriate privileges, and then query the most recent 3 AWR snapshots.

   For example, query `DBA_HIST_SNAPSHOT` as follows:

   ```
   SELECT *
   FROM   (SELECT SNAP_ID, SNAP_LEVEL,
                  TO_CHAR(BEGIN_INTERVAL_TIME, 'DD/MM/YY HH24:MI:SS')
   BEGIN
           FROM   DBA_HIST_SNAPSHOT
           ORDER BY SNAP_ID DESC)
   WHERE   ROWNUM <= 3;

      SNAP_ID SNAP_LEVEL BEGIN
   ---------- ---------- -----------------
          212          1 10/12/15 06:00:02
          211          1 10/12/15 05:00:11
          210          1 10/12/15 04:00:59
   ```

2. Query `sh.sales`, using the `LOAD_AWR` tag to identify the SQL statement.

   For example, use the following query:

   ```
   SELECT /*LOAD_AWR*/ *
   FROM   sh.sales
   WHERE  quantity_sold > 40
   ORDER BY prod_id;
   ```

3. Take a new AWR snapshot.

   For example, use the following program:

   ```
   EXEC DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT;
   ```

4. Query the most recent 3 AWR snapshots to confirm that a new snapshot was taken.

For example, query DBA_HIST_SNAPSHOT as follows:

```
SELECT *
FROM   (SELECT SNAP_ID, SNAP_LEVEL,
               TO_CHAR(BEGIN_INTERVAL_TIME, 'DD/MM/YY HH24:MI:SS') BEGIN
        FROM   DBA_HIST_SNAPSHOT
        ORDER BY SNAP_ID DESC)
WHERE   ROWNUM <= 3;

   SNAP_ID SNAP_LEVEL BEGIN
---------- ---------- -----------------
       213          1 10/12/15 06:24:53
       212          1 10/12/15 06:00:02
       211          1 10/12/15 05:00:11
```

5. Load the plans for the most recent 2 snapshots from AWR.

   For example, execute the LOAD_PLANS_FROM_AWR function in SQL*Plus to load the plans from snapshot 212 to 213:

```
VARIABLE v_plan_cnt NUMBER
EXEC :v_plan_cnt := DBMS_SPM.LOAD_PLANS_FROM_AWR(begin_snap => 212,
end_snap => 213);
```

   In the preceding example, the variable v_plan_cnt contains the number of plans that were loaded.

6. Query the data dictionary to ensure that the plans were loaded into the baseline for the LOAD_AWR statement.

   The following statement queries DBA_SQL_PLAN_BASELINES (sample output included):

```
COL SQL_HANDLE FORMAT a20
COL SQL_TEXT FORMAT a20
COL PLAN_NAME FORMAT a30
COL ORIGIN FORMAT a20

SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME,
       ORIGIN, ENABLED, ACCEPTED
FROM   DBA_SQL_PLAN_BASELINES
WHERE SQL_TEXT LIKE '%LOAD_AWR%';

SQL_HANDLE           SQL_TEXT          PLAN_NAME                 ORIGIN      ENA ACC
-------------------- ----------------- -------------------------- ----------- --- ---
SQL_495d29c5f4612cda SELECT /*LOAD_AWR SQL_PLAN_4kr99sru62b6u54bc MANUAL-LOAD- YES YES
                     */ * FROM         8843                       FROM-AWR
                     sh.sales WHERE
                     quantity_sold
                     > 40
                     ORDER BY prod_id
```

   The output shows that the plan is accepted, which means that it is in the plan baseline for the statement. Also, the origin is MANUAL-LOAD-FROM-AWR, which means that the statement was loaded manually from AWR rather than automatically captured.

See Also:

- "Fixed Plans"
- *Oracle Database PL/SQL Packages and Types Reference* to learn how to use the `DBMS_SPM.LOAD_PLANS_FROM_AWR` function
- *Oracle Database Reference* to learn more about the `DBA_SQL_PLAN_BASELINES` view

29.4.3 Loading Plans from the Shared SQL Area

This topic explains how to load plans from the shared SQL area, also called the cursor cache, using PL/SQL.

Load plans with the `LOAD_PLANS_FROM_CURSOR_CACHE` function of the `DBMS_SPM` package. The following table describes some function parameters.

Table 29-6 LOAD_PLANS_FROM_CURSOR_CACHE Parameters

Function Parameter	Description
`sql_id`	SQL statement identifier. Identifies a SQL statement in the shared SQL area.
`fixed`	Default `NO` means the loaded plans are used as nonfixed plans. `YES` means the loaded plans are fixed plans. "Plan Selection" explains that the optimizer chooses a fixed plan in the plan baseline over a nonfixed plan.

This section explains how to load plans using the command line. In Cloud Control, go to the SQL Plan Baseline subpage (shown in Figure 29-1) and click **Load** to load plan baselines from the shared SQL area.

This tutorial assumes the following:

- You have executed the following query:

  ```
  SELECT /*LOAD_CC*/ *
  FROM   sh.sales
  WHERE  quantity_sold > 40
  ORDER BY prod_id;
  ```

- You want the loaded plans to be nonfixed.

To load plans from the shared SQL area:

1. Connect SQL*Plus to the database with the appropriate privileges, and then determine the SQL IDs of the relevant statements in the shared SQL area.

   For example, query `V$SQL` for the SQL ID of the `sh.sales` query (sample output included):

   ```
   SELECT   SQL_ID, CHILD_NUMBER AS "Child Num",
            PLAN_HASH_VALUE AS "Plan Hash",
   ```

```
         OPTIMIZER_ENV_HASH_VALUE AS "Opt Env Hash"
FROM     V$SQL
WHERE    SQL_TEXT LIKE 'SELECT /*LOAD_CC*/%';

SQL_ID         Child Num  Plan Hash Opt Env Hash
------------- ---------- ---------- ------------
27m0sdw9snw59          0 1421641795   3160571937
```

The preceding output shows that the SQL ID of the statement is `27m0sdw9snw59`.

2. Load the plans for the specified statements into the SQL plan baseline.

   For example, execute the `LOAD_PLANS_FROM_CURSOR_CACHE` function in SQL*Plus to load the plan for the statement with the SQL ID `27m0sdw9snw59`:

```
VARIABLE v_plan_cnt NUMBER
BEGIN
  :v_plan_cnt:=DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE(
    sql_id => '27m0sdw9snw59');
END;
```

   In the preceding example, the variable `v_plan_cnt` contains the number of plans that were loaded.

3. Query the data dictionary to ensure that the plans were loaded into the baseline for the statement.

   The following statement queries `DBA_SQL_PLAN_BASELINES` (sample output included):

```
SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME,
       ORIGIN, ENABLED, ACCEPTED
FROM   DBA_SQL_PLAN_BASELINES;

SQL_HANDLE            SQL_TEXT             PLAN_NAME      ORIGIN              ENA ACC
--------------------- -------------------- -------------- ------------------- --- ---
SQL_a8632bd857a4a25e  SELECT /*LOAD_CC*/   SQL_PLAN_gdkvz MANUAL-LOAD-FROM-CC YES YES
                      *                    fhrgkda71694fc
                      FROM sh.sales        6b
                      WHERE quantity_sold
                      > 40
                      ORDER BY prod_id
```

   The output shows that the plan is accepted, which means that it is in the plan baseline for the statement. Also, the origin is `MANUAL-LOAD-FROM-CC`, which means that the statement was loaded manually from the shared SQL area rather than automatically captured.

See Also:

- "Fixed Plans"
- *Oracle Database PL/SQL Packages and Types Reference* to learn how to use the `DBMS_SPM.LOAD_PLANS_FROM_CURSOR_CACHE` function
- *Oracle Database Reference* to learn more about the `DBA_SQL_PLAN_BASELINES` view

29.4.4 Loading Plans from a SQL Tuning Set

A **SQL tuning set** (STS) is a database object that includes one or more SQL statements, execution statistics, and execution context. This topic explains how to load plans from an STS.

Load plans with the `DBMS_SPM.LOAD_PLANS_FROM_SQLSET` function or using Cloud Control. The following table describes some function parameters.

Table 29-7 LOAD_PLANS_FROM_SQLSET Parameters

Function Parameter	Description
`sqlset_name`	Name of the STS from which the plans are loaded into SQL plan baselines.
`basic_filter`	A filter applied to the STS to select only qualifying plans to be loaded. The filter can take the form of any `WHERE` clause predicate that can be specified against the view `DBA_SQLSET_STATEMENTS`. An example is `basic_filter => 'sql_text like ''SELECT /*LOAD_STS*/%'''`.
`fixed`	Default `NO` means the loaded plans are used as nonfixed plans. `YES` means the loaded plans are fixed plans. "Plan Selection" explains that the optimizer chooses a fixed plan in the plan baseline over a nonfixed plan.

This section explains how to load plans from the command line. In Cloud Control, go to the SQL Plan Baseline subpage (shown in Figure 29-1) and click **Load** to load plan baselines from SQL tuning sets.

Assumptions

This tutorial assumes the following:

- You want the loaded plans to be nonfixed.
- You have executed the following query:

```
SELECT /*LOAD_STS*/ *
FROM   sh.sales
WHERE  quantity_sold > 40
ORDER BY prod_id;
```

- You have loaded the plan from the shared SQL area into the SQL tuning set named `SPM_STS`, which is owned by user `SPM`.

- After the operation, you want to drop the STS using `DBMS_SQLTUNE.DROP_SQLSET` rather than the equivalent D`BMS_SQLSET.DROP_SQLSET`.

To load plans from a SQL tuning set:

1. Connect SQL*Plus to the database with the appropriate privileges, and then verify which plans are in the SQL tuning set.

   For example, query `DBA_SQLSET_STATEMENTS` for the STS name (sample output included):

   ```
   SELECT SQL_TEXT
   FROM   DBA_SQLSET_STATEMENTS
   WHERE  SQLSET_NAME = 'SPM_STS';

   SQL_TEXT
   --------------------
   SELECT /*LOAD_STS*/
   *
   FROM sh.sales
   WHERE quantity_sold
   > 40
   ORDER BY prod_id
   ```

   The output shows that the plan for the `select /*LOAD_STS*/` statement is in the STS.

2. Load the plan from the STS into the SQL plan baseline.

   For example, in SQL*Plus execute the function as follows:

   ```
   VARIABLE v_plan_cnt NUMBER
   EXECUTE :v_plan_cnt := DBMS_SPM.LOAD_PLANS_FROM_SQLSET( -
              sqlset_name  => 'SPM_STS', -
              basic_filter => 'sql_text like ''SELECT /*LOAD_STS*/%''' );
   ```

   The `basic_filter` parameter specifies a `WHERE` clause that loads only the plans for the queries of interest. The variable `v_plan_cnt` stores the number of plans loaded from the STS.

3. Query the data dictionary to ensure that the plan was loaded into the baseline for the statement.

   The following statement queries the `DBA_SQL_PLAN_BASELINES` view (sample output included).

```
SQL> SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME,
  2         ORIGIN, ENABLED, ACCEPTED
  3  FROM   DBA_SQL_PLAN_BASELINES;

SQL_HANDLE            SQL_TEXT         PLAN_NAME        ORIGIN               ENA ACC
--------------------- ---------------- ---------------- -------------------- --- ---
SQL_a8632bd857a4a25e  SELECT           SQL_PLAN_ahstb   MANUAL-LOAD-FROM-STS YES YES
                      /*LOAD_STS*/*    v1bu98ky1694fc6b
                      FROM sh.sales
                      WHERE
                      quantity_sold
```

```
> 40 ORDER BY
prod_id
```

The output shows that the plan is accepted, which means that it is in the plan baseline. Also, the origin is `MANUAL-LOAD-FROM-STS`, which means that the plan was loaded manually from a SQL tuning set rather than automatically captured.

4. Optionally, drop the STS.

   For example, execute `DBMS_SQLTUNE.DROP_SQLSET` to drop the `SPM_STS` tuning set as follows:

```
EXEC SYS.DBMS_SQLTUNE.DROP_SQLSET( sqlset_name  => 'SPM_STS', -
                                   sqlset_owner => 'SPM' );
```

See Also:

- "Command-Line Interface to SQL Tuning Sets"
- *Oracle Database Reference* to learn about the `DBA_SQL_PLAN_BASELINES` view
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM.LOAD_PLANS_FROM_SQLSET` function

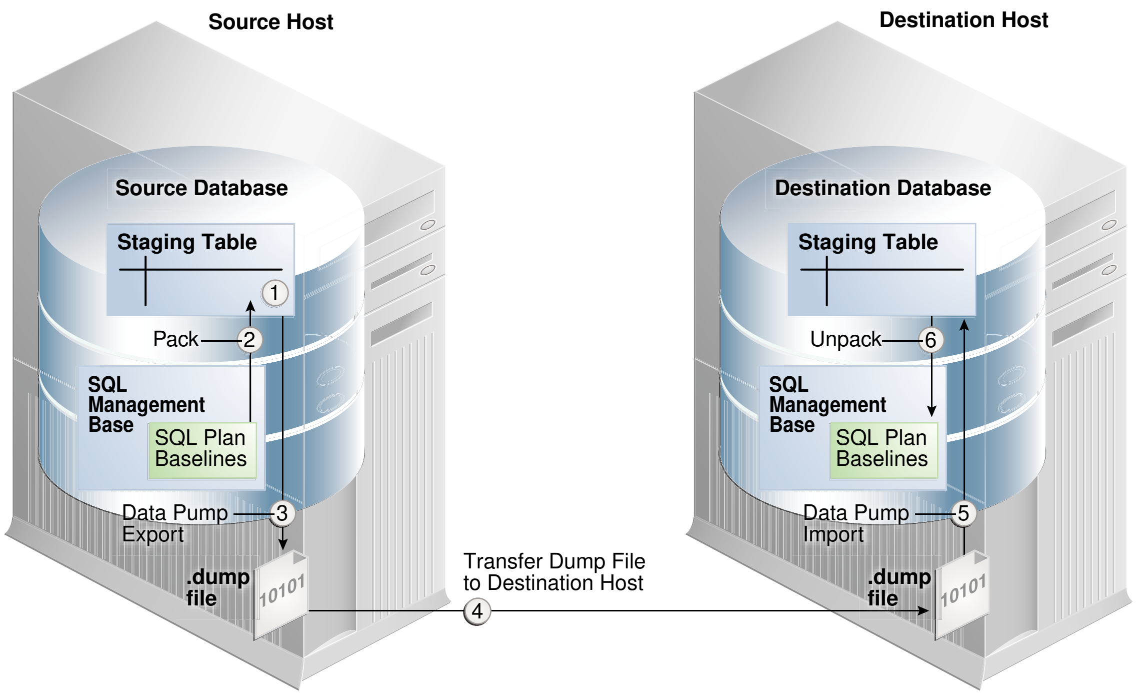
29.4.5 Loading Plans from a Staging Table

You may want to transfer optimal plans from a source database to a different destination database.

For example, you may have investigated a set of plans on a test database and confirmed that they have performed well. You may then want to load these plans into a production database.

A staging table is a table that, for the duration of its existence, stores plans so that the plans do not disappear from the table while you are unpacking them. Use the `DBMS_SPM.CREATE_STGTAB_BASELINE` procedure to create a staging table. To pack (insert row into) and unpack (extract rows from) the staging table, use the `PACK_STGTAB_BASELINE` and `UNPACK_STGTAB_BASELINE` functions of the `DBMS_SPM` package. Oracle Data Pump Import and Export enable you to copy the staging table to a different database.

Figure 29-3 Loading Plans from a Staging Table



Export plans with the `PACK_STGTAB_BASELINE` function of the `DBMS_SPM` package, and then import them with `UNPACK_STGTAB_BASELINE`. The following table describes some function parameters.

Table 29-8 PACK_STGTAB_BASELINE and UNPACK_STGTAB_BASELINE Parameters

Function Parameter	Description
`table_name`	Specifies the table to be imported or exported.
`origin`	Origin of SQL plan baseline. These procedures accept all possible values of `DBA_SQL_PLAN_BASELINES.ORIGIN` as the `origin` argument. For example, the `origin` parameter permits `MANUAL-LOAD-FROM-STS`, `MANUAL-LOAD-FROM-AWR`, and `MANUAL-LOAD-FROM-CC`.

This tutorial assumes the following:

- You want to create a staging table named `stage1` in the source database.
- You want to load all plans owned by user `spm` into the staging table.
- You want to transfer the staging table to a destination database.
- You want to load the plans in `stage1` as fixed plans.

To transfer a set of SQL plan baselines from one database to another:

1. Connect SQL*Plus to the source database with the appropriate privileges, and then create a staging table using the `CREATE_STGTAB_BASELINE` procedure.

The following example creates a staging table named `stage1`:

```
BEGIN
  DBMS_SPM.CREATE_STGTAB_BASELINE (
    table_name => 'stage1');
END;
/
```

2. On the source database, pack the SQL plan baselines you want to export from the SQL management base into the staging table.

   The following example packs enabled plan baselines created by user `spm` into staging table `stage1`. Select SQL plan baselines using the plan name (`plan_name`), SQL handle (`sql_handle`), or any other plan criteria. The `table_name` parameter is mandatory.

```
DECLARE
  v_plan_cnt NUMBER;
BEGIN
  v_plan_cnt := DBMS_SPM.PACK_STGTAB_BASELINE (
    table_name => 'stage1'
,   enabled    => 'yes'
,   creator    => 'spm'
);
END;
/
```

3. Export the staging table `stage1` into a dump file using Oracle Data Pump Export.
4. Transfer the dump file to the host of the destination database.
5. On the destination database, import the staging table `stage1` from the dump file using the Oracle Data Pump Import utility.
6. On the destination database, unpack the SQL plan baselines from the staging table into the SQL management base.

   The following example unpacks all fixed plan baselines stored in the staging table `stage1`:

```
DECLARE
  v_plan_cnt NUMBER;
BEGIN
  v_plan_cnt := DBMS_SPM.UNPACK_STGTAB_BASELINE (
    table_name => 'stage1'
,   fixed      => 'yes'
);
END;
/
```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for more information about using the `DBMS_SPM` package
- *Oracle Database Reference* to learn more about the `DBA_SQL_PLAN_BASELINES` view
- *Oracle Database Utilities* for detailed information about using the Data Pump Export and Import utilities

29.5 Evolving SQL Plan Baselines Manually

You can use PL/SQL or Cloud Control to manually evolve an unaccepted plan to determine whether it performs better than any plan currently in the plan baseline.

See Also:

"Managing the SPM Evolve Advisor Task"

29.5.1 About the DBMS_SPM Evolve Functions

This topic describes the most relevant `DBMS_SPM` functions for managing plan evolution. Execute evolution tasks manually or schedule them to run automatically.

Table 29-9 DBMS_SPM Functions and Procedures for Managing Plan Evolution Tasks

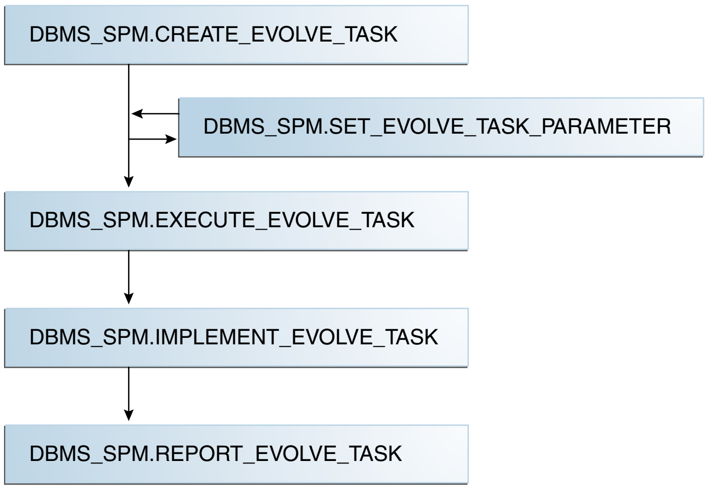
Procedure or Function	Description
`ACCEPT_SQL_PLAN_BASELINE`	This function accepts one recommendation to evolve a single plan into a SQL plan baseline.
`CREATE_EVOLVE_TASK`	This function creates an advisor task to prepare the plan evolution of one or more plans for a specified SQL statement. The input parameters can be a SQL handle, plan name or a list of plan names, time limit, task name, and description.
`EXECUTE_EVOLVE_TASK`	This function executes an evolution task. The input parameters can be the task name, execution name, and execution description. If not specified, the advisor generates the name, which is returned by the function.
`IMPLEMENT_EVOLVE_TASK`	This function implements all recommendations for an evolve task. Essentially, this function is equivalent to using `ACCEPT_SQL_PLAN_BASELINE` for all recommended plans. Input parameters include task name, plan name, owner name, and execution name.

Table 29-9 (Cont.) DBMS_SPM Functions and Procedures for Managing Plan Evolution Tasks

Procedure or Function	Description
`REPORT_EVOLVE_TASK`	This function displays the results of an evolve task as a `CLOB`. Input parameters include the task name and section of the report to include.
`SET_EVOLVE_TASK_PARAMETER`	This function updates the value of an evolve task parameter.

Oracle recommends that you configure SPM Evolve Advisor to run automatically. You can also evolve SQL plan baselines manually. The following graphic shows the basic workflow for managing SQL plan management tasks.

Figure 29-4 Evolving SQL Plan Baselines


DBMS_SPM.CREATE_EVOLVE_TASK
DBMS_SPM.SET_EVOLVE_TASK_PARAMETER
DBMS_SPM.EXECUTE_EVOLVE_TASK
DBMS_SPM.IMPLEMENT_EVOLVE_TASK
DBMS_SPM.REPORT_EVOLVE_TASK

Typically, you manage SQL plan evolution tasks in the following sequence:

1. Create an evolve task
2. Optionally, set evolve task parameters
3. Execute the evolve task
4. Implement the recommendations in the task
5. Report on the task outcome

See Also:

- "Configuring the Automatic SPM Evolve Advisor Task" to learn about `SET_EVOLVE_TASK_PARAMETER`
- *Oracle Database PL/SQL Packages and Types Reference* for more information about the `DBMS_SPM` package

29.5.2 Managing an Evolve Task

This topic describes a typical use case in which you create and execute a task, and then implement its recommendations.

The following table describes some parameters of the `CREATE_EVOLVE_TASK` function.

Table 29-10 DBMS_SPM.CREATE_EVOLVE_TASK Parameters

Function Parameter	Description
`sql_handle`	SQL handle of the statement. The default `NULL` considers all SQL statements with unaccepted plans.
`plan_name`	Plan identifier. The default `NULL` means consider all unaccepted plans of the specified SQL handle or all SQL statements if the SQL handle is `NULL`.
`time_limit`	Time limit in number of minutes. The time limit for first unaccepted plan equals the input value. The time limit for the second unaccepted plan equals the input value minus the time spent in first plan verification, and so on. The default `DBMS_SPM.AUTO_LIMIT` means let the system choose an appropriate time limit based on the number of plan verifications required to be done.
`task_name`	User-specified name of the evolution task.

This section explains how to evolve plan baselines from the command line. In Cloud Control, from the SQL Plan Baseline subpage, select a plan, and then click **Evolve**.

This tutorial assumes the following:

- You do not have the automatic evolve task enabled.
- You want to create a SQL plan baseline for the following query:

  ```
  SELECT /* q1_group_by */ prod_name, sum(quantity_sold)
  FROM   products p, sales s
  WHERE  p.prod_id = s.prod_id
  AND    p.prod_category_id =203
  GROUP BY prod_name;
  ```

- You want to create two indexes to improve the query performance, and then evolve the plan that uses these indexes if it performs better than the plan currently in the plan baseline.

To evolve a specified plan:

1. Perform the initial setup as follows:
   a. Connect SQL*Plus to the database with administrator privileges, and then prepare for the tutorial by flushing the shared pool and the buffer cache:

      ```
      ALTER SYSTEM FLUSH SHARED_POOL;
      ALTER SYSTEM FLUSH BUFFER_CACHE;
      ```

   b. Enable the automatic capture of SQL plan baselines.

For example, enter the following statement:

```
ALTER SYSTEM SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES=true;
```

c. Connect to the database as user `sh`, and then set SQL*Plus display parameters:

```
CONNECT sh
-- enter password
SET PAGES 10000 LINES 140
SET SERVEROUTPUT ON
COL SQL_TEXT FORMAT A20
COL SQL_HANDLE FORMAT A20
COL PLAN_NAME FORMAT A30
COL ORIGIN FORMAT A12
SET LONGC 60535
SET LONG 60535
SET ECHO ON
```

2. Execute the `SELECT` statements so that SQL plan management captures them:

   a. Execute the `SELECT /* q1_group_by */` statement for the first time.

      Because the database only captures plans for repeatable statements, the plan baseline for this statement is empty.

   b. Query the data dictionary to confirm that no plans exist in the plan baseline.

      For example, execute the following query (sample output included):

```
SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME, ORIGIN, ENABLED,
       ACCEPTED, FIXED, AUTOPURGE
FROM   DBA_SQL_PLAN_BASELINES
WHERE  SQL_TEXT LIKE '%q1_group%';

no rows selected
```

      SQL plan management only captures repeatable statements, so this result is expected.

   c. Execute the `SELECT /* q1_group_by */` statement for the second time.

3. Query the data dictionary to ensure that the plans were loaded into the plan baseline for the statement.

   The following statement queries `DBA_SQL_PLAN_BASELINES` (sample output included):

```
SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME,
       ORIGIN, ENABLED, ACCEPTED, FIXED
FROM   DBA_SQL_PLAN_BASELINES
WHERE  SQL_TEXT LIKE '%q1_group%';

SQL_HANDLE           SQL_TEXT         PLAN_NAME              ORIGIN       ENA ACC
FIX
-------------------- ---------------- ---------------------- ------------ --- ---
---
```

```
SQL_07f16c76ff893342 SELECT /* q1_gro SQL_PLAN_0gwbcfvzskcu2 AUTO-CAPTURE YES YES NO
                     up_by */ prod_na 42949306
                     me, sum(quantity
                     _sold) FROM
                     products p,
                     sales s WHERE
                     p.prod_id =
                     s.prod_id AND
                     p.prod_category
                     _id =203 GROUP
                     BY prod_name
```

The output shows that the plan is accepted, which means that it is in the plan baseline for the statement. Also, the origin is `AUTO-CAPTURE`, which means that the statement was automatically captured and not manually loaded.

4. Explain the plan for the statement and verify that the optimizer is using this plan.

   For example, explain the plan as follows, and then display it:

```
EXPLAIN PLAN FOR
  SELECT /* q1_group_by */ prod_name, sum(quantity_sold)
  FROM   products p, sales s
  WHERE  p.prod_id = s.prod_id
  AND    p.prod_category_id =203
  GROUP BY prod_name;

SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(null, null, 'basic +note'));
```

   Sample output appears below:

```
Plan hash value: 1117033222

------------------------------------------
| Id  | Operation             | Name     |
------------------------------------------
|   0 | SELECT STATEMENT      |          |
|   1 |  HASH GROUP BY        |          |
|   2 |   HASH JOIN           |          |
|   3 |    TABLE ACCESS FULL  | PRODUCTS |
|   4 |    PARTITION RANGE ALL|          |
|   5 |     TABLE ACCESS FULL | SALES    |
------------------------------------------

Note
-----
   - SQL plan baseline "SQL_PLAN_0gwbcfvzskcu242949306" used for this
statement
```

   The note indicates that the optimizer is using the plan shown with the plan name listed in the previous step.

5. Create two indexes to improve the performance of the `SELECT /* q1_group_by */` statement.

For example, use the following statements:

```
CREATE INDEX ind_prod_cat_name
  ON products(prod_category_id, prod_name, prod_id);
CREATE INDEX ind_sales_prod_qty_sold
  ON sales(prod_id, quantity_sold);
```

6. Execute the `select /* q1_group_by */` statement again.

   Because automatic capture is enabled, the plan baseline is populated with the new plan for this statement.

7. Query the data dictionary to ensure that the plan was loaded into the SQL plan baseline for the statement.

   The following statement queries `DBA_SQL_PLAN_BASELINES` (sample output included).

```
SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME, ORIGIN, ENABLED, ACCEPTED
FROM   DBA_SQL_PLAN_BASELINES
WHERE  SQL_HANDLE IN ('SQL_07f16c76ff893342')
ORDER BY SQL_HANDLE, ACCEPTED;

SQL_HANDLE           SQL_TEXT             PLAN_NAME              ORIGIN       ENA
ACC
-------------------- -------------------- ---------------------- ------------ ---
---
SQL_07f16c76ff893342 SELECT /* q1_group_b SQL_PLAN_0gwbcfvzskcu2 AUTO-CAPTURE YES
NO
                     y */ prod_name, sum( 0135fd6c
                     quantity_sold)
                     FROM   products p, s
                     ales s
                     WHERE  p.prod_id = s
                     .prod_id
                     AND    p.prod_catego
                     ry_id =203
                     GROUP BY prod_name

SQL_07f16c76ff893342 SELECT /* q1_group_b SQL_PLAN_0gwbcfvzskcu2 AUTO-CAPTURE YES
YES
                     y */ prod_name, sum( 42949306
                     quantity_sold)
                     FROM   products p, s
                     ales s
                     WHERE  p.prod_id = s
                     .prod_id
                     AND    p.prod_catego
                     ry_id =203
                     GROUP BY prod_name
```

   The output shows that the new plan is unaccepted, which means that it is in the statement history but not the SQL plan baseline.

8. Explain the plan for the statement and verify that the optimizer is using the original unindexed plan.

For example, explain the plan as follows, and then display it:

```
EXPLAIN PLAN FOR
  SELECT /* q1_group_by */ prod_name, sum(quantity_sold)
  FROM   products p, sales s
  WHERE  p.prod_id = s.prod_id
  AND    p.prod_category_id =203
  GROUP BY prod_name;
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY(null, null, 'basic +note'));
```

Sample output appears below:

```
Plan hash value: 1117033222

------------------------------------------
| Id  | Operation             | Name     |
------------------------------------------
|   0 | SELECT STATEMENT      |          |
|   1 |  HASH GROUP BY        |          |
|   2 |   HASH JOIN           |          |
|   3 |    TABLE ACCESS FULL  | PRODUCTS |
|   4 |    PARTITION RANGE ALL|          |
|   5 |     TABLE ACCESS FULL | SALES    |
------------------------------------------

Note
-----
   - SQL plan baseline "SQL_PLAN_0gwbcfvzskcu242949306" used for this
statement
```

The note indicates that the optimizer is using the plan shown with the plan name listed in Step 3.

9. Connect as an administrator, and then create an evolve task that considers all SQL statements with unaccepted plans.

   For example, execute the `DBMS_SPM.CREATE_EVOLVE_TASK` function and then obtain the name of the task:

```
CONNECT / AS SYSDBA
VARIABLE cnt NUMBER
VARIABLE tk_name VARCHAR2(50)
VARIABLE exe_name VARCHAR2(50)
VARIABLE evol_out CLOB

EXECUTE :tk_name := DBMS_SPM.CREATE_EVOLVE_TASK(
  sql_handle => 'SQL_07f16c76ff893342',
  plan_name  => 'SQL_PLAN_0gwbcfvzskcu20135fd6c');

SELECT :tk_name FROM DUAL;
```

The following sample output shows the name of the task:

```
:EVOL_OUT
--------------------------------------------------------------------
----
TASK_11
```

Now that the task has been created and has a unique name, execute the task.

10. Execute the task.

    For example, execute the `DBMS_SPM.EXECUTE_EVOLVE_TASK` function (sample output included):

```
EXECUTE :exe_name :=DBMS_SPM.EXECUTE_EVOLVE_TASK(task_name=>:tk_name
);
SELECT :exe_name FROM DUAL;

:EXE_NAME
--------------------------------------------------------------------
----
EXEC_1
```

11. View the report.

    For example, execute the `DBMS_SPM.REPORT_EVOLVE_TASK` function (sample output included):

```
EXECUTE :evol_out :=
DBMS_SPM.REPORT_EVOLVE_TASK( task_name=>:tk_name,
execution_name=>:exe_name );
SELECT :evol_out FROM DUAL;

GENERAL INFORMATION SECTION
--------------------------------------------------------------------
----

 Task Information:
 ---------------------------------------------
 Task Name            : TASK_11
 Task Owner           : SYS
 Execution Name       : EXEC_1
 Execution Type       : SPM EVOLVE
 Scope                : COMPREHENSIVE
 Status               : COMPLETED
 Started              : 01/09/2012 12:21:27
 Finished             : 01/09/2012 12:21:29
 Last Updated         : 01/09/2012 12:21:29
 Global Time Limit    : 2147483646
 Per-Plan Time Limit  : UNUSED
 Number of Errors     : 0
--------------------------------------------------------------------
----

SUMMARY SECTION
```

```
---------------------------------------------------------------------------
  Number of plans processed  : 1
  Number of findings         : 1
  Number of recommendations  : 1
  Number of errors           : 0
---------------------------------------------------------------------------

DETAILS SECTION
---------------------------------------------------------------------------
 Object ID          : 2
 Test Plan Name     : SQL_PLAN_0gwbcfvzskcu20135fd6c
 Base Plan Name     : SQL_PLAN_0gwbcfvzskcu242949306
 SQL Handle         : SQL_07f16c76ff893342
 Parsing Schema     : SH
 Test Plan Creator  : SH
 SQL Text           : SELECT /*q1_group_by*/ prod_name,
                      sum(quantity_sold)
                      FROM products p, sales s
                      WHERE p.prod_id=s.prod_id AND p.prod_category_id=203
                      GROUP BY prod_name

Execution Statistics:
-----------------------------
                    Base Plan                      Test Plan
                    ----------------------------   ---------
 Elapsed Time (s):  .044336                        .012649
 CPU Time (s):      .044003                        .012445
 Buffer Gets:       360                            99
 Optimizer Cost:    924                            891
 Disk Reads:        341                            82
 Direct Writes:     0                              0
 Rows Processed:    4                              2
 Executions:        5                              9


FINDINGS SECTION
---------------------------------------------------------------------------

Findings (1):
-----------------------------
 1. The plan was verified in 2.18 seconds. It passed the benefit
    criterion because its verified performance was 2.01 times
    better than that of the baseline plan.

Recommendation:
-----------------------------
 Consider accepting the plan. Execute
 dbms_spm.accept_sql_plan_baseline(task_name => 'TASK_11',
 object_id => 2, task_owner => 'SYS');

EXPLAIN PLANS SECTION
---------------------------------------------------------------------------

Baseline Plan
-----------------------------
```

```
 Plan Id          : 1
 Plan Hash Value  : 1117033222

--------------------------------------------------------------------
---
| Id| Operation                | Name    | Rows | Bytes    |Cost |
Time  |
--------------------------------------------------------------------
---
| 0 | SELECT STATEMENT         |         |   21 |     861 |924|
00:00:12|
| 1 |   HASH GROUP BY          |         |   21 |     861 |924|
00:00:12|
| *2|    HASH JOIN             |         |267996|10987836 |742|
00:00:09|
| *3|     TABLE ACCESS FULL    |PRODUCTS|   21 |     714 |  2|
00:00:01|
| 4 |     PARTITION RANGE ALL |         |918843| 6431901 |662|
00:00:08|
| 5 |      TABLE ACCESS FULL  |SALES    |918843| 6431901 |662|
00:00:08|
--------------------------------------------------------------------
---

Predicate Information (identified by operation id):
------------------------------------------
* 2 - access("P"."PROD_ID"="S"."PROD_ID")
* 3 - filter("P"."PROD_CATEGORY_ID"=203)

Test Plan
-----------------------------
 Plan Id          : 2
 Plan Hash Value  : 20315500

--------------------------------------------------------------------
----
|Id| Operation              | Name              | Rows |Bytes |Cost|
Time |
--------------------------------------------------------------------
----
| 0|SELECT STATEMENT        |                   |   21|   861|891|
00:00:11|
| 1|  SORT GROUP BY NOSORT|                   |   21|   861|891|
00:00:11|
| 2|   NESTED LOOPS         |                   |267K |10987K|891|
00:00:11|
|*3|    INDEX RANGE SCAN    |IND_PROD_CAT_NAME |   21|   714|  1|
00:00:01|
|*4|    INDEX RANGE SCAN    |IND_SALES_PROD_QTY|12762|  9334| 42|
00:00:01|
--------------------------------------------------------------------
----

Predicate Information (identified by operation id):
------------------------------------------
```

```
* 3 - access("P"."PROD_CATEGORY_ID"=203)
* 4 - access("P"."PROD_ID"="S"."PROD_ID")
```

This report indicates that the new execution plan, which uses the two new indexes, performs better than the original plan.

12. Implement the recommendations of the evolve task.

    For example, execute the `IMPLEMENT_EVOLVE_TASK` function:

```
BEGIN
  :cnt := DBMS_SPM.IMPLEMENT_EVOLVE_TASK(
    task_name=>:tk_name, execution_name=>:exe_name );
END;
```

13. Query the data dictionary to ensure that the new plan is accepted.

    The query provides the following sample output:

```
SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME, ORIGIN, ENABLED, ACCEPTED
FROM   DBA_SQL_PLAN_BASELINES
WHERE  SQL_HANDLE IN ('SQL_07f16c76ff893342')
ORDER BY SQL_HANDLE, ACCEPTED;

SQL_HANDLE           SQL_TEXT             PLAN_NAME              ORIGIN       ENA ACC
-------------------- -------------------- ---------------------- ------------ --- ---
SQL_07f16c76ff893342 SELECT /* q1_group_b SQL_PLAN_0gwbcfvzskcu2 AUTO-CAPTURE YES YES
                     y */ prod_name, sum( 0135fd6c
                     quantity_sold)
                     FROM    products p, s
                     ales s
                     WHERE   p.prod_id = s
                     .prod_id
                     AND     p.prod_catego
                     ry_id =203
                     GROUP BY prod_name

SQL_07f16c76ff893342 SELECT /* q1_group_b SQL_PLAN_0gwbcfvzskcu2 AUTO-CAPTURE YES YES
                     y */ prod_name, sum( 42949306
                     quantity_sold)
                     FROM    products p, s
                     ales s
                     WHERE   p.prod_id = s
                     .prod_id
                     AND     p.prod_catego
                     ry_id =203
                     GROUP BY prod_name
```

The output shows that the new plan is accepted.

14. Clean up after the example.

    For example, enter the following statements:

```
EXEC :cnt := DBMS_SPM.DROP_SQL_PLAN_BASELINE('SQL_07f16c76ff893342');
EXEC :cnt := DBMS_SPM.DROP_SQL_PLAN_BASELINE('SQL_9049245213a986b3');
EXEC :cnt :=
DBMS_SPM.DROP_SQL_PLAN_BASELINE('SQL_bb77077f5f90a36b');
EXEC :cnt :=
DBMS_SPM.DROP_SQL_PLAN_BASELINE('SQL_02a86218930bbb20');
DELETE FROM SQLLOG$;
CONNECT sh
-- enter password
DROP INDEX IND_SALES_PROD_QTY_SOLD;
DROP INDEX IND_PROD_CAT_NAME;
```

See Also:

- "Managing the SPM Evolve Advisor Task"
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about the DBMS_SPM evolve functions

29.6 Dropping SQL Plan Baselines

You can remove some or all plans from a SQL plan baseline. This technique is sometimes useful when testing SQL plan management.

Drop plans with the DBMS_SPM.DROP_SQL_PLAN_BASELINE function. This function returns the number of dropped plans. The following table describes input parameters.

Table 29-11 DROP_SQL_PLAN_BASELINE Parameters

Function Parameter	Description
sql_handle	SQL statement identifier.
plan_name	Name of a specific plan. Default NULL drops all plans associated with the SQL statement identified by sql_handle.

This section explains how to drop baselines from the command line. In Cloud Control, from the SQL Plan Baseline subpage, select a plan, and then click **Drop**.

This tutorial assumes that you want to drop all plans for the following SQL statement, effectively dropping the SQL plan baseline:

```
SELECT /* repeatable_sql */ COUNT(*) FROM hr.jobs;
```

To drop a SQL plan baseline:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the data dictionary for the plan baseline.

The following statement queries `DBA_SQL_PLAN_BASELINES` (sample output included):

```
SQL> SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME, ORIGIN,
  2         ENABLED, ACCEPTED
  3  FROM   DBA_SQL_PLAN_BASELINES
  4  WHERE  SQL_TEXT LIKE 'SELECT /* repeatable_sql%';

SQL_HANDLE           SQL_TEXT             PLAN_NAME              ORIGIN       ENA ACC
-------------------- -------------------- ---------------------- ------------ --- ---
SQL_b6b0d1c71cd1807b SELECT /* repeatable SQL_PLAN_bdc6jswfd303v AUTO-CAPTURE YES YES
                     _sql */ count(*)     2f1e9c20
                     from hr.jobs
```

2. Drop the SQL plan baseline for the statement.

   The following example drops the plan baseline with the SQL handle `SQL_b6b0d1c71cd1807b`, and returns the number of dropped plans. Specify plan baselines using the plan name (`plan_name`), SQL handle (`sql_handle`), or any other plan criteria. The `table_name` parameter is mandatory.

   ```
   DECLARE
     v_dropped_plans number;
   BEGIN
     v_dropped_plans := DBMS_SPM.DROP_SQL_PLAN_BASELINE (
        sql_handle => 'SQL_b6b0d1c71cd1807b'
   );
     DBMS_OUTPUT.PUT_LINE('dropped ' || v_dropped_plans || ' plans');
   END;
   /
   ```

3. Confirm that the plans were dropped.

   For example, execute the following query:

   ```
   SELECT SQL_HANDLE, SQL_TEXT, PLAN_NAME, ORIGIN,
          ENABLED, ACCEPTED
   FROM   DBA_SQL_PLAN_BASELINES
   WHERE  SQL_TEXT LIKE 'SELECT /* repeatable_sql%';

   no rows selected
   ```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn about the `DROP_SQL_PLAN_BASELINE` function

29.7 Managing the SQL Management Base

The SQL management base is a part of the data dictionary that resides in the `SYSAUX` tablespace. It stores statement logs, plan histories, SQL plan baselines, and SQL profiles.

29.7.1 About Managing the SMB

Use the DBMS_SPM.CONFIGURE procedure to set configuration options for the SMB and the maintenance of SQL plan baselines.

The DBA_SQL_MANAGEMENT_CONFIG view shows the current configuration settings for the SMB. The following table describes the parameters in the PARAMETER_NAME column.

Table 29-12 Parameters in DBA_SQL_MANAGEMENT_CONFIG.PARAMETER_NAME

Parameter	Description
SPACE_BUDGET_PERCENT	Maximum percent of SYSAUX space that the SQL management base can use. The default is 10. The allowable range for this limit is between 1% and 50%.
PLAN_RETENTION_WEEKS	Number of weeks to retain unused plans before they are purged. The default is 53.
AUTO_CAPTURE_PARSING_SCHEMA_NAME	A list of the form (% LIKE *a* OR % LIKE *b* ...) AND (% NOT LIKE *c* AND % NOT LIKE *d* ...), which is the internal representation of the parsing schema name filter. If no parsing schema filters exist, then one side of the outer conjunction will be absent.
AUTO_CAPTURE_MODULE	A list of the form (% LIKE *a* OR % LIKE *b* ...) AND (% NOT LIKE *c* AND % NOT LIKE *d* ...), which is the internal representation of the module filter. If no module filters exist, then one side of the outer conjunction will be absent.
AUTO_CAPTURE_ACTION	A list of the form (% LIKE *a* OR % LIKE *b* ...) AND (% NOT LIKE *c* AND % NOT LIKE *d* ...), which is the internal representation of the action filter. If no action filters exist, then one side of the outer conjunction will be absent.
AUTO_CAPTURE_SQL_TEXT	A list of the form (% LIKE *a* OR % LIKE *b* ...) AND (% NOT LIKE *c* AND % NOT LIKE *d* ...), which is the internal representation of the SQL text filter. If no SQL text filters exist, then one side of the outer conjunction will be absent.

See Also:

- "Eligibility for Automatic Initial Plan Capture"
- *Oracle Database Reference* to learn more about DBA_SQL_MANAGEMENT_CONFIG
- *Oracle Database PL/SQL Packages and Types Reference* to learn more about DBMS_SPM.CONFIGURE

29.7.2 Changing the Disk Space Limit for the SMB

A weekly background process measures the total space occupied by the SMB.

When the defined limit is exceeded, the process writes a warning to the alert log. The database generates alerts weekly until either the SMB space limit is increased, the size of the `SYSAUX` tablespace is increased, or the disk space used by the SMB is decreased by purging SQL management objects (SQL plan baselines or SQL profiles). This task explains how to change the limit with the `DBMS_SPM.CONFIGURE` procedure.

Assumptions

This tutorial assumes the following:

- The current SMB space limit is the default of 10%.
- You want to change the percentage limit to 30%

To change the percentage limit of the SMB:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the data dictionary to see the current space budget percent.

   For example, execute the following query (sample output included):

```
SELECT PARAMETER_NAME, PARAMETER_VALUE AS "%_LIMIT",
       ( SELECT sum(bytes/1024/1024) FROM DBA_DATA_FILES
         WHERE TABLESPACE_NAME = 'SYSAUX' ) AS SYSAUX_SIZE_IN_MB,
       PARAMETER_VALUE/100 *
       ( SELECT sum(bytes/1024/1024) FROM DBA_DATA_FILES
         WHERE TABLESPACE_NAME = 'SYSAUX' ) AS "CURRENT_LIMIT_IN_MB"
FROM DBA_SQL_MANAGEMENT_CONFIG
WHERE PARAMETER_NAME = 'SPACE_BUDGET_PERCENT';

PARAMETER_NAME          %_LIMIT SYSAUX_SIZE_IN_MB CURRENT_LIMIT_IN_MB
-------------------- ---------- ----------------- -------------------
SPACE_BUDGET_PERCENT         10          211.4375            21.14375
```

2. Change the percentage setting.

   For example, execute the following command to change the setting to 30%:

```
EXECUTE DBMS_SPM.CONFIGURE('space_budget_percent',30);
```

3. Query the data dictionary to confirm the change.

   For example, execute the following join (sample output included):

```
SELECT PARAMETER_NAME, PARAMETER_VALUE AS "%_LIMIT",
       ( SELECT sum(bytes/1024/1024) FROM DBA_DATA_FILES
         WHERE TABLESPACE_NAME = 'SYSAUX' ) AS SYSAUX_SIZE_IN_MB,
       PARAMETER_VALUE/100 *
       ( SELECT sum(bytes/1024/1024) FROM DBA_DATA_FILES
         WHERE TABLESPACE_NAME = 'SYSAUX' ) AS "CURRENT_LIMIT_IN_MB"
FROM   DBA_SQL_MANAGEMENT_CONFIG
WHERE  PARAMETER_NAME = 'SPACE_BUDGET_PERCENT';
```

```
PARAMETER_NAME          %_LIMIT SYSAUX_SIZE_IN_MB
CURRENT_LIMIT_IN_MB
-------------------- ---------- -----------------
-------------------
SPACE_BUDGET_PERCENT         30          211.4375
63.43125
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `DBMS_SPM.CONFIGURE` procedure

29.7.3 Changing the Plan Retention Policy in the SMB

A weekly scheduled purging task manages disk space used by SQL plan management.

The task runs as an automated task in the maintenance window. The database purges plans that have not been used for longer than the plan retention period, as identified by the `LAST_EXECUTED` timestamp stored in the SMB for that plan. The default retention period is 53 weeks. The period can range between 5 and 523 weeks.

This task explains how to change the plan retention period with the `DBMS_SPM.CONFIGURE` procedure. In Cloud Control, set the plan retention policy in the SQL Plan Baseline subpage (shown in Figure 29-1).

To change the plan retention period for the SMB:

1. Connect SQL*Plus to the database with the appropriate privileges, and then query the data dictionary to see the current plan retention period.

   For example, execute the following query (sample output included):

   ```
   SQL> SELECT PARAMETER_NAME, PARAMETER_VALUE
     2  FROM   DBA_SQL_MANAGEMENT_CONFIG
     3  WHERE  PARAMETER_NAME = 'PLAN_RETENTION_WEEKS';

   PARAMETER_NAME                 PARAMETER_VALUE
   ------------------------------ ---------------
   PLAN_RETENTION_WEEKS                        53
   ```

2. Change the retention period.

   For example, execute the `CONFIGURE` procedure to change the period to 105 weeks:

   ```
   EXECUTE DBMS_SPM.CONFIGURE('plan_retention_weeks',105);
   ```

3. Query the data dictionary to confirm the change.

For example, execute the following query:

```
SQL> SELECT PARAMETER_NAME, PARAMETER_VALUE
  2  FROM   DBA_SQL_MANAGEMENT_CONFIG
  3  WHERE  PARAMETER_NAME = 'PLAN_RETENTION_WEEKS';

PARAMETER_NAME                 PARAMETER_VALUE
------------------------------ ---------------
PLAN_RETENTION_WEEKS                       105
```

See Also:

Oracle Database PL/SQL Packages and Types Reference to learn more about the `CONFIGURE` procedure

30
Migrating Stored Outlines to SQL Plan Baselines

Stored outline migration is the user-initiated process of converting stored outlines to SQL plan baselines. A SQL plan baseline is a set of plans proven to provide optimal performance.

Note:

Starting in Oracle Database 12c, stored outlines are deprecated. See "Migrating Stored Outlines to SQL Plan Baselines" for an alternative.

This chapter explains the concepts and tasks relating to stored outline migration.

30.1 About Stored Outline Migration

A **stored outline** is a set of hints for a SQL statement.

The hints direct the optimizer to choose a specific plan for the statement. A stored outline is a legacy technique for providing plan stability.

30.1.1 Purpose of Stored Outline Migration

If you rely on stored outlines to maintain plan stability and prevent performance regressions, then you can safely migrate from stored outlines to SQL plan baselines. After the migration, you can maintain the same plan stability while benefiting from the features provided by the SQL Plan Management framework.

Stored outlines have the following disadvantages:

- Stored outlines cannot automatically evolve over time. Consequently, a stored outline may be optimal when you create it, but become a suboptimal plan after a database change, leading to performance degradation.
- Hints in a stored outline can become invalid, as with an index hint on a dropped index. In such cases, the database still uses the outlines but excludes the invalid hints, producing a plan that is often worse than the original plan or the current best-cost plan generated by the optimizer.
- For a SQL statement, the optimizer can only choose the plan defined in the stored outline in the currently specified category. The optimizer cannot choose from other stored outlines in different categories or the current cost-based plan even if they improve performance.
- Stored outlines are a reactive tuning technique, which means that you only use a stored outline to address a performance problem after it has occurred. For example, you may implement a stored outline to correct the plan of a SQL statement that became high-load.

In this case, you used stored outlines instead of proactively tuning the statement before it became high-load.

The stored outline migration PL/SQL API helps solve the preceding problems in the following ways:

- SQL plan baselines enable the optimizer to use the same optimal plan and allow this plan to evolve over time.

  For a specified SQL statement, you can add new plans as SQL plan baselines after they are verified not to cause performance regressions.
- SQL plan baselines prevent plans from becoming unreproducible because of invalid hints.

  If hints stored in a plan baseline become invalid, then the plan may not be reproducible by the optimizer. In this case, the optimizer selects an alternative reproducible plan baseline or the current best-cost plan generated by optimizer.
- For a specific SQL statement, the database can maintain multiple plan baselines.

  The optimizer can choose from a set of optimal plans for a specific SQL statement instead of being restricted to a single plan per category, as required by stored outlines.

30.1.2 How Stored Outline Migration Works

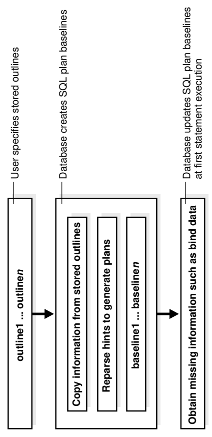
Stored outline migration is a user-initiated process that goes through multiple stages.

This section explains how the database migrates stored outlines to SQL plan baselines. This information is important for performing the task of migrating stored outlines.

30.1.2.1 Stages of Stored Outline Migration

To migrate stored outlines, you specify the stores outlines. The database then creates and updates SQL plan baselines.

Figure 30-1 Stages of Stored Outline Migration


User specifies stored outlines
Database creates SQL plan baselines
Database updates SQL plan baselines at first statement execution
outline1 ... outlinen
Copy information from stored outlines
Reparse hints to generate plans
baseline1 ... baselinen
Obtain missing information such as bind data

The migration process has the following stages:

1. The user invokes a function that specifies which outlines to migrate.
2. The database processes the outlines as follows:
   a. The database copies information in the outline needed by the plan baseline.

      The database copies it directly or calculates it based on information in the outline. For example, the text of the SQL statement exists in both schemas, so the database can copy the text from outline to baseline.
   b. The database reparses the hints to obtain information not in the outline.

      The plan hash value and plan cost cannot be derived from the existing information in the outline, which necessitates reparsing the hints.
   c. The database creates the baselines.
3. The database obtains missing information when it chooses the SQL plan baseline for the first time to execute the SQL statement.

   The compilation environment and execution statistics are only available during execution when the plan baseline is parsed and compiled.

The migration is complete only after the preceding phases complete.

30.1.2.2 Outline Categories and Baseline Modules

An outline is a set of hints, whereas a SQL plan baseline is a set of plans.

Because they are different technologies, some functionality of outlines does not map exactly to functionality of baselines. For example, a single SQL statement can have multiple outlines, each of which is in a different outline **category**, but the only category that currently exists for baselines is `DEFAULT`.

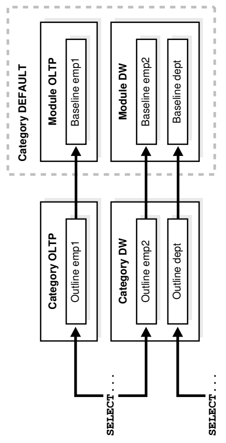
The equivalent of a category for an outline is a module for a SQL plan baseline. Table 30-1 explains how outline categories map to modules.

Table 30-1 Outline Categories

Concept	Description	Default Value
Outline Category	Specifies a user-defined grouping for a set of stored outlines. You can use categories to maintain different stored outlines for a SQL statement. For example, a single statement can have an outline in the `OLTP` category and the `DW` category. Each SQL statement can have one or more stored outlines. Each stored outline is in one and only one outline category. A statement can have multiple stored outlines in different categories, but only one stored outline exists for each category of each statement. During migration, the database maps each outline category to a SQL plan baseline module.	`DEFAULT`
Baseline Module	Specifies a high-level function being performed. A SQL plan baseline can belong to one and only one module.	After an outline is migrated to a SQL plan baseline, the module name defaults to outline category name.
Baseline Category	Only one SQL plan baseline category exists. This category is named `DEFAULT`. During stored outline migration, the module name of the SQL plan baseline is set to the category name of the stored outline. A statement can have multiple SQL plan baselines in the `DEFAULT` category.	`DEFAULT`

When migrating stored outlines to SQL plan baselines, Oracle Database maps every outline category to a SQL plan baseline module with the same name. As shown in the following diagram, the outline category `OLTP` is mapped to the baseline module `OLTP`. After migration, `DEFAULT` is a super-category that contains all SQL plan baselines.

Figure 30-2 DEFAULT Category



30.1.3 User Interface for Stored Outline Migration

You can use the `DBMS_SPM` package to perform the stored outline migration.

Table 30-2 DBMS_SPM Functions Relating to Stored Outline Migration

DBMS_SPM Function	Description
`MIGRATE_STORED_OUTLINE`	Migrates existing stored outlines to plan baselines. Use either of the following formats: • Specify outline name, SQL text, outline category, or all stored outlines. • Specify a list of outline names.
`ALTER_SQL_PLAN_BASELINE`	Changes an attribute of a single plan or all plans associated with a SQL statement.
`DROP_MIGRATED_STORED_OUTLINE`	Drops stored outlines that have been migrated to SQL plan baselines. The function finds stored outlines marked as `MIGRATED` in the `DBA_OUTLINES` view, and then drops these outlines from the database.

You can control stored outline and plan baseline behavior with initialization and session parameters. Table 30-3 describes the relevant parameters. See Table 30-5 and Table 30-6 for an explanation of how these parameter settings interact.

Table 30-3 Parameters Relating to Stored Outline Migration

Initialization or Session Parameter	Description	Parameter Type
`CREATE_STORED_OUTLINES`	Determines whether Oracle Database automatically creates and stores an outline for each query submitted during the session.	Initialization parameter
`OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES`	Enables or disables the automatic recognition of repeatable SQL statement and the generation of SQL plan baselines for these statements.	Initialization parameter

Table 30-3 (Cont.) Parameters Relating to Stored Outline Migration

Initialization or Session Parameter	Description	Parameter Type
`OPTIMIZER_USE_SQL_PLAN_BASELINES`	Enables or disables the use of SQL plan baselines stored in SQL Management Base.	Initialization parameter
`USE_STORED_OUTLINES`	Determines whether the optimizer uses stored outlines to generate execution plans. **Note:** This is a *session* parameter, not an initialization parameter.	Session

You can use database views to access information relating to stored outline migration. Table 30-4 describes the following main views.

Table 30-4 Views Relating to Stored Outline Migration

View	Description
`DBA_OUTLINES`	Describes all stored outlines in the database. The `MIGRATED` column is important for outline migration and shows one of the following values: `NOT-MIGRATED` and `MIGRATED`. When `MIGRATED`, the stored outline has been migrated to a plan baseline and is not usable.
`DBA_SQL_PLAN_BASELINES`	Displays information about the SQL plan baselines currently created for specific SQL statements. The `ORIGIN` column indicates how the plan baseline was created. The value `STORED-OUTLINE` indicates the baseline was created by migrating an outline.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package
- *Oracle Database Reference* to learn about the `CREATE_STORED_OUTLINES` initialization parameter

30.1.4 Basic Steps in Stored Outline Migration

The basic process is to prepare, migrate, and then clean up.

The basic steps are as follows:

1. Prepare for stored outline migration.

   Review the migration prerequisites and determine how you want the migrated plan baselines to behave.

   See "Preparing for Stored Outline Migration".

2. Perform either of the following tasks:
   - Migrate to baselines to use SQL Plan Management features.

     See "Migrating Outlines to Utilize SQL Plan Management Features".
   - Migrate to baselines while exactly preserving the behavior of the stored outlines.

     See "Migrating Outlines to Preserve Stored Outline Behavior".
3. Perform post-migration confirmation and cleanup.

   See "Performing Follow-Up Tasks After Stored Outline Migration".

30.2 Preparing for Stored Outline Migration

The goal is preparation is determining which stored outlines are eligible for migration.

To prepare for stored outline migration:

1. Connect SQL*Plus to the database with `SYSDBA` privileges or the `EXECUTE` privilege on the `DBMS_SPM` package.
2. Query the stored outlines in the database.

   The following example queries all stored outlines that have not been migrated to SQL plan baselines:

   ```
   SELECT NAME, CATEGORY, SQL_TEXT
   FROM   DBA_OUTLINES
   WHERE  MIGRATED = 'NOT-MIGRATED';
   ```
3. Determine which stored outlines meet the following prerequisites for migration eligibility:
   - The statement must *not* be a run-time `INSERT AS SELECT` statement.
   - The statement must *not* reference a remote object.
   - This statement must *not* be a private stored outline.
4. Decide whether to migrate all outlines, specified stored outlines, or outlines belonging to a specified outline category.

   If you do not decide to migrate all outlines, then identify the outlines or categories that you intend to migrate.
5. Decide whether the stored outlines migrated to SQL plan baselines use **fixed plans** or **nonfixed plans**:
   - Fixed plans

     A fixed plan is frozen. If a fixed plan is reproducible using the hints stored in plan baseline, then the optimizer always chooses the lowest-cost fixed plan baseline over plan baselines that are not fixed. Essentially, a fixed plan baseline acts as a stored outline with valid hints.

     A fixed plan is reproducible when the database can parse the statement based on the hints stored in the plan baseline and create a plan with the same plan hash value as the one in the plan baseline. If one of more of the hints become invalid, then the database may not be able to create a plan with the same plan hash value. In this case, the plan is nonreproducible.

If a fixed plan cannot be reproduced when parsed using its hints, then the optimizer chooses a different plan, which can be either of the following:

  – Another plan for the SQL plan baseline
  – The current cost-based plan created by the optimizer

  In some cases, a performance regression occurs because of the different plan, requiring SQL tuning.

- Nonfixed plans

  If a plan baseline does not contain fixed plans, then SQL Plan Management considers the plans equally when picking a plan for a SQL statement.

6. Before beginning the actual migration, ensure that the Oracle database meets the following prerequisites:
   - The database must be Enterprise Edition.
   - The database must be open and *not* in a suspended state.
   - The database must *not* be in restricted access (DBA), read-only, or migrate mode.
   - Oracle Call Interface (OCI) must be available.

See Also:

- *Oracle Database Administrator’s Guide* to learn about administrator privileges
- *Oracle Database Reference* to learn about the `DBA_OUTLINES` views

30.3 Migrating Outlines to Utilize SQL Plan Management Features

You can migrate stored outline to SQL plan baselines.

The goals of this task are as follows:

- To allow SQL Plan Management to select from all plans in a plan baseline for a SQL statement instead of applying the same fixed plan after migration
- To allow the SQL plan baseline to evolve in the face of database changes by adding new plans to the baseline

Assumptions

This tutorial assumes the following:

- You migrate all outlines.

  To migrate specific outlines, use the `DBMS_SPM.MIGRATE_STORED_OUTLINE` function.
- You want the module names of the baselines to be identical to the category names of the migrated outlines.
- You do *not* want the SQL plans to be fixed.

By default, generated plans are not fixed and SQL Plan Management considers all plans equally when picking a plan for a SQL statement. This situation permits the advanced feature of plan evolution to capture new plans for a SQL statement, verify their performance, and accept these new plans into the plan baseline.

To migrate stored outlines to SQL plan baselines:

1. Connect SQL*Plus to the database with the appropriate privileges.
2. Call PL/SQL function `MIGRATE_STORED_OUTLINE`.

   The following sample PL/SQL block migrates all stored outlines to fixed baselines:

   ```
   DECLARE
     my_report CLOB;
   BEGIN
     my_outlines := DBMS_SPM.MIGRATE_STORED_OUTLINE(
                        attribute_name => 'all' );
   END;
   /
   ```

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package
- *Oracle Database SQL Language Reference* to learn about the `ALTER SYSTEM` statement

30.4 Migrating Outlines to Preserve Stored Outline Behavior

You can migrate stored outlines to SQL plan baselines and preserve the original behavior of the stored outlines by creating fixed plan baselines.

A fixed plan has higher priority over other plans for the same SQL statement. If a plan is fixed, then the plan baseline cannot be evolved. The database does not add new plans to a plan baseline that contains a fixed plan.

Assumptions

This tutorial assumes the following:

- You want to migrate only the stored outlines in the category named `firstrow`.
- You want the module names of the baselines to be identical to the category names of the migrated outlines.

To migrate stored outlines to plan baselines:

1. Connect SQL*Plus to the database with the appropriate privileges.
2. Call PL/SQL function `MIGRATE_STORED_OUTLINE`.

The following sample PL/SQL block migrates stored outlines in the category `firstrow` to fixed baselines:

```
DECLARE
  my_report CLOB;
BEGIN
  my_outlines := DBMS_SPM.MIGRATE_STORED_OUTLINE(
    attribute_name => 'category',
    attribute_value => 'firstrow',
    fixed => 'YES' );
END;
/
```

After migration, the SQL plan baselines is in module `firstrow` and category `DEFAULT`.

See Also:

- *Oracle Database PL/SQL Packages and Types Reference* for syntax and semantics of the `DBMS_SPM.MIGRATE_STORED_OUTLINE` function
- *Oracle Database SQL Language Reference* to learn about the `ALTER SYSTEM` statement

30.5 Performing Follow-Up Tasks After Stored Outline Migration

After migrating outlines to SQL plan baselines, you must perform some follow-up work.

The goals of this task are as follows:

- To configure the database to use plan baselines instead of stored outlines for stored outlines that have been migrated to SQL plan baselines
- To create SQL plan baselines instead of stored outlines for future SQL statements
- To drop the stored outlines that have been migrated to SQL plan baselines

This section explains how to set initialization parameters relating to stored outlines and plan baselines. The `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` and `CREATE_STORED_OUTLINES` initialization parameters determine how and when the database creates stored outlines and SQL plan baselines. Table 30-5 explains the interaction between these parameters.

Table 30-5 Creation of Outlines and Baselines

CREATE_STORED_OUTLINES Initialization Parameter	OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES Initialization Parameter	Database Behavior
`false`	`false`	When executing a SQL statement, the database does not create stored outlines or SQL plan baselines.
`false`	`true`	The automatic recognition of repeatable SQL statements and the generation of SQL plan baselines for these statements is enabled. When executing a SQL statement, the database creates only new SQL plan baselines (if they do not exist) with the category name `DEFAULT` for the statement.
`true`	`false`	Oracle Database automatically creates and stores an outline for each query submitted during the session. When executing a SQL statement, the database creates only new stored outlines (if they do not exist) with the category name `DEFAULT` for the statement.
`category`	`false`	When executing a SQL statement, the database creates only new stored outlines (if they do not exist) with the specified category name for the statement.
`true`	`true`	Oracle Database automatically creates and stores an outline for each query submitted during the session. The automatic recognition of repeatable SQL statements and the generation of SQL plan baselines for these statements is also enabled. When executing a SQL statement, the database creates both stored outlines and SQL plan baselines with the category name `DEFAULT`.
`category`	`true`	Oracle Database automatically creates and stores an outline for each query submitted during the session. The automatic recognition of repeatable SQL statements and the generation of SQL plan baselines for these statements is also enabled. When executing a SQL statement, the database creates stored outlines with the specified category name and SQL plan baselines with the category name `DEFAULT`.

The `USE_STORED_OUTLINES` session parameter (it is *not* an initialization parameter) and `OPTIMIZER_USE_SQL_PLAN_BASELINES` initialization parameter determine how the database uses stored outlines and plan baselines. Table 30-6 explains how these parameters interact.

Table 30-6 Use of Stored Outlines and SQL Plan Baselines

USE_STORED_OUTLINES Session Parameter	OPTIMIZER_USE_SQL_ PLAN_BASELINES Initialization Parameter	Database Behavior
`false`	`false`	When choosing a plan for a SQL statement, the database does not use stored outlines or plan baselines.
`false`	`true`	When choosing a plan for a SQL statement, the database uses only SQL plan baselines.
`true`	`false`	When choosing a plan for a SQL statement, the database uses stored outlines with the category name `DEFAULT`.
`category`	`false`	When choosing a plan for a SQL statement, the database uses stored outlines with the specified category name. If a stored outline with the specified category name does not exist, then the database uses a stored outline in the `DEFAULT` category if it exists.
`true`	`true`	When choosing a plan for a SQL statement, stored outlines take priority over plan baselines. If a stored outline with the category name `DEFAULT` exists for the statement and is applicable, then the database applies the stored outline. Otherwise, the database uses SQL plan baselines. However, if the stored outline has the property `MIGRATED`, then the database does not use the outline and uses the corresponding SQL plan baseline instead (if it exists).
`category`	`true`	When choosing a plan for a SQL statement, stored outlines take priority over plan baselines. If a stored outline with the specified category name or the `DEFAULT` category exists for the statement and is applicable, then the database applies the stored outline. Otherwise, the database uses SQL plan baselines. However, if the stored outline has the property `MIGRATED`, then the database does not use the outline and uses the corresponding SQL plan baseline instead (if it exists).

Assumptions

This tutorial assumes the following:

- You have completed the basic steps in the stored outline migration.
- Some stored outlines may have been created before Oracle Database 10*g*.

  Hints in releases before Oracle Database 10*g* use a local hint format. After migration, hints stored in a plan baseline use the global hints format introduced in Oracle Database 10*g*.

To place the database in the proper state after the migration:

1. Connect SQL*Plus to the database with the appropriate privileges, and then check that SQL plan baselines have been created as the result of migration.

   Ensure that the plans are enabled and accepted. For example, enter the following query (partial sample output included):

   ```
   SELECT SQL_HANDLE, PLAN_NAME, ORIGIN, ENABLED, ACCEPTED, FIXED, MODULE
   FROM   DBA_SQL_PLAN_BASELINES;

   SQL_HANDLE                PLAN_NAME  ORIGIN         ENA ACC FIX MODULE
   ------------------------- ---------- -------------- --- --- --- ------
   SYS_SQL_f44779f7089c8fab  STMT01     STORED-OUTLINE YES YES NO  DEFAULT
   .
   .
   .
   ```

2. Optionally, change the attributes of the SQL plan baselines.

   For example, the following statement changes the status of the baseline for the specified SQL statement to `fixed`:

   ```
   DECLARE
     v_cnt PLS_INTEGER;
   BEGIN
     v_cnt := DBMS_SPM.ALTER_SQL_PLAN_BASELINE(
                             sql_handle=>'SYS_SQL_f44779f7089c8fab',
                             attribute_name=>'FIXED',
                             attribute_value=>'NO');
     DBMS_OUTPUT.PUT_LINE('Plans altered: ' || v_cnt);
   END;
   /
   ```

3. Check the status of the original stored outlines.

   For example, enter the following query (partial sample output included):

   ```
   SELECT NAME, OWNER, CATEGORY, USED, MIGRATED
   FROM   DBA_OUTLINES
   ORDER BY NAME;

   NAME       OWNER      CATEGORY   USED   MIGRATED
   ---------- ---------- ---------- ------ ------------
   STMT01     SYS        DEFAULT    USED   MIGRATED
   ```

```
STMT02      SYS         DEFAULT     USED    MIGRATED
.
.
.
```

4. Drop all stored outlines that have been migrated to SQL plan baselines.

   For example, the following statements drops all stored outlines with status `MIGRATED` in `DBA_OUTLINES`:

   ```
   DECLARE
     v_cnt PLS_INTEGER;
   BEGIN
     v_cnt := DBMS_SPM.DROP_MIGRATED_STORED_OUTLINE();
     DBMS_OUTPUT.PUT_LINE('Migrated stored outlines dropped: ' ||
   v_cnt);
   END;
   /
   ```

5. Set initialization parameters so that:

   - When executing a SQL statement, the database creates plan baselines but does not create stored outlines.
   - The database only uses stored outlines when the equivalent SQL plan baselines do not exist.

   For example, the following SQL statements instruct the database to create SQL plan baselines instead of stored outlines when a SQL statement is executed. The example also instructs the database to apply a stored outline in category `allrows` or `DEFAULT` only if it exists and has not been migrated to a SQL plan baseline. In other cases, the database applies SQL plan baselines instead.

   ```
   ALTER SYSTEM
     SET CREATE_STORED_OUTLINE = false SCOPE = BOTH;

   ALTER SYSTEM
     SET OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES = true SCOPE = BOTH;

   ALTER SYSTEM
      SET OPTIMIZER_USE_SQL_PLAN_BASELINES = true SCOPE = BOTH;

   ALTER SESSION
      SET USE_STORED_OUTLINES = allrows SCOPE = BOTH;
   ```

See Also:

- "Basic Steps in Stored Outline Migration"
- *Oracle Database PL/SQL Packages and Types Reference* to learn about the `DBMS_SPM` package
- *Oracle Database Reference* to learn about the `CREATE_STORED_OUTLINES` initialization parameter

Glossary

accepted plan

In the context of SQL plan management, a plan that is in a SQL plan baseline for a SQL statement and thus available for use by the optimizer. An accepted plan contains a set of hints, a plan hash value, and other plan-related information.

access path

The means by which the database retrieves data from a database. For example, a query using an index and a query using a full table scan use different access paths.

adaptive cursor sharing

A feature that enables a single statement that contains bind variables to use multiple execution plans. Cursor sharing is "adaptive" because the **cursor** adapts its behavior so that the database does not always use the same plan for each execution or bind variable value.

adaptive dynamic sampling

A feature of the adaptive optimizer that enables the automatic adjustment of the dynamic statistics level.

adaptive optimizer

A feature of the optimizer that enables it to adapt plans based on run-time statistics.

adaptive query plan

An execution plan that changes after optimization because run-time conditions indicate that optimizer estimates are inaccurate. An adaptive query plan has different built-in plan options. During the first execution, before a specific subplan becomes active, the optimizer makes a final decision about which option to use. The optimizer bases its choice on observations made during the execution up to this point. Thus, an adaptive query plan enables the final plan for a statement to differ from the default plan.

adaptive query optimization

A set of capabilities that enables the **adaptive optimizer** to make run-time adjustments to execution plans and discover additional information that can lead to better statistics. Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan.

ADDM

See **Automatic Database Diagnostic Monitor (ADDM)**.

antijoin

A join that returns rows that fail to match the subquery on the right side. For example, an antijoin can list departments with no employees. Antijoins use the `NOT EXISTS` or `NOT IN` constructs.

approximate query processing

A set of optimization techniques that speed analytic queries by calculating results within an acceptable range of error.

automatic capture filter

A SQL plan management feature that enables you to specify the eligibility criteria for automatic initial plan capture.

Automatic Database Diagnostic Monitor (ADDM)

ADDM is self-diagnostic software built into Oracle Database. ADDM examines and analyzes data captured in **Automatic Workload Repository (AWR)** to determine possible database performance problems.

automatic optimizer statistics collection

The automatic running of the `DBMS_STATS` package to collect optimizer statistics for all schema objects for which statistics are missing or stale.

automatic initial plan capture

The mechanism by which the database automatically creates a SQL plan baseline for any repeatable SQL statement executed on the database. Enable automatic initial plan capture by setting the `OPTIMIZER_CAPTURE_SQL_PLAN_BASELINES` initialization parameter to `true` (the default is `false`).

See **repeatable SQL statement**.

automatic reoptimization

The ability of the optimizer to automatically change a plan on subsequent executions of a SQL statement. Automatic reoptimization can fix any suboptimal plan chosen due to incorrect optimizer estimates, from a suboptimal distribution method to an incorrect choice of degree of parallelism.

automatic SQL tuning

The work performed by Automatic SQL Tuning Advisor it runs as an automated task within system maintenance windows.

Automatic SQL Tuning Advisor

SQL Tuning Advisor when run as an automated maintenance task. Automatic SQL Tuning runs during system maintenance windows as an automated maintenance task, searching for ways to improve the execution plans of high-load SQL statements.

See **SQL Tuning Advisor**.

Automatic Tuning Optimizer

The optimizer when invoked by SQL Tuning Advisor. In SQL tuning mode, the optimizer performs additional analysis to check whether it can further improve the plan produced in normal mode. The optimizer output is not an execution plan, but a series of actions, along with their rationale and expected benefit for producing a significantly better plan.

Automatic Workload Repository (AWR)

The infrastructure that provides services to Oracle Database components to collect, maintain, and use statistics for problem detection and self-tuning.

AWR

See **Automatic Workload Repository (AWR)**.

AWR snapshot

A set of data for a specific time that is used for performance comparisons. The delta values captured by the snapshot represent the changes for each statistic over the time period. Statistics gathered by are queried from memory. You can display the gathered data in both reports and views.

band join

A special type of nonequijoin in which key values in one data set must fall within the specified range ("band") of the second data set.

base cardinality

For a table, the total number of rows in the table.

baseline

In the context of **AWR**, the interval between two AWR snapshots that represent the database operating at an optimal level.

bind-aware cursor

A bind-sensitive cursor that is eligible to use different plans for different bind values. After a cursor has been made bind-aware, the optimizer chooses plans for future executions based on the bind value and its **cardinality** estimate.

bind-sensitive cursor

A cursor whose optimal plan may depend on the value of a bind variable. The database monitors the behavior of a bind-sensitive cursor that uses different bind values to determine whether a different plan is beneficial.

bind variable

A placeholder in a SQL statement that must be replaced with a valid value or value address for the statement to execute successfully. By using bind variables, you can write a SQL statement that accepts inputs or parameters at run time. The following query uses `v_empid` as a bind variable:

```
SELECT * FROM employees WHERE employee_id = :v_empid;
```

bind variable peeking

The ability of the optimizer to look at the value in a bind variable during a **hard parse**. By peeking at bind values, the optimizer can determine the selectivity of a `WHERE` clause condition as if literals *had* been used, thereby improving the plan.

bitmap join index

A bitmap index for the join of two or more tables.

bitmap piece

A subcomponent of a single bitmap index entry. Each indexed column value may have one or more bitmap pieces. The database uses bitmap pieces to break up an index entry that is large in relation to the size of a block.

B-tree index

An index organized like an upside-down tree. A B-tree index has two types of blocks: branch blocks for searching and leaf blocks that store values. The leaf blocks contain every indexed data value and a corresponding rowid used to locate the actual row. The "B" stands for "balanced" because all leaf blocks automatically stay at the same depth.

bulk load

A `CREATE TABLE AS SELECT` or `INSERT INTO ... SELECT` operation.

bushy join tree

A join tree in which the left or the right child of an internal node can be a join node. Nodes in a bushy join tree may have recursive structures in both its descendents.

cardinality

The number of rows that is expected to be or is returned by an operation in an execution plan.

Cartesian join

A join in which one or more of the tables does not have any join conditions to any other tables in the statement. The **optimizer** joins every row from one data source with every row from the other data source, creating the Cartesian product of the two sets.

child cursor

The cursor containing the plan, compilation environment, and other information for a statement whose text is stored in a **parent cursor**. The parent cursor is number `0`, the first child is number `1`, and so on. Child cursors reference the same SQL text as the parent cursor, but are different. For example, two queries with the text `SELECT * FROM t` use different cursors when they reference two different tables named `t`.

cluster scan

An access path for a table cluster. In an indexed table cluster, Oracle Database first obtains the rowid of one of the selected rows by scanning the cluster index. Oracle Database then locates the rows based on this rowid.

column group

A set of columns that is treated as a unit.

column group statistics

Extended statistics gathered on a group of columns treated as a unit.

column statistics

Statistics about columns that the **optimizer** uses to determine optimal execution plans. Column statistics include the number of distinct column values, low value, high value, and number of nulls.

complex view merging

The merging of views containing the `GROUP BY` or `DISTINCT` keywords.

composite database operation

In a **database operation**, the activity between two points in time in a database session, with each session defining its own beginning and end points. A session can participate in at most one composite database operation at a time.

concurrency

Simultaneous access of the same data by many users. A multiuser database management system must provide adequate concurrency controls so that data cannot be updated or changed improperly, compromising data integrity.

concurrent statistics gathering mode

A mode that enables the database to simultaneously gather optimizer statistics for multiple tables in a schema, or multiple partitions or subpartitions in a table. Concurrency can reduce the overall time required to gather statistics by enabling the database to fully use multiple CPUs.

condition

A combination of one or more expressions and logical operators that returns a value of `TRUE`, `FALSE`, or `UNKNOWN`.

cost

A numeric internal measure that represents the estimated resource usage for an **execution plan**. The lower the cost, the more efficient the plan.

cost-based optimizer (CBO)

The legacy name for the optimizer. In earlier releases, the cost-based optimizer was an alternative to the rule-based optimizer (RBO).

cost model

The internal optimizer model that accounts for the **cost** of the I/O, CPU, and network resources that a query is predicted to use.

cumulative statistics

A count such as the number of block reads. Oracle Database generates many types of cumulative statistics for the system, sessions, and individual SQL statements.

cursor

A handle or name for a **private SQL area** in the PGA. Because cursors are closely associated with private SQL areas, the terms are sometimes used interchangeably.

cursor cache

See **shared SQL area**.

cursor merging

Combining cursors to save space in the shared SQL area. If the optimizer creates a plan for a **bind-aware cursor**, and if this plan is the same as an existing cursor, then the optimizer can merge the cursors.

cursor-duration temporary table

A temporary, in-memory table that stores query results for the duration of a cursor. For complex operations such as `WITH` clause queries and star transformations, this optimization enhances the materialization of intermediate results from repetitively used subqueries. In this way, cursor-duration temporary tables improve performance and optimizes I/O.

data flow operator (DFO)

The unit of work between data redistribution stages in a parallel query.

data skew

Large variations in the number of duplicate values in a column.

database operation

A set of database tasks defined by end users or application code, for example, a batch job or ETL processing.

default plan

For an adaptive plan, the execution plan initially chosen by the optimizer using the statistics from the data dictionary. The default plan can differ from the **final plan**.

disabled plan

A plan that a database administrator has manually marked as ineligible for use by the optimizer.

degree of parallelism (DOP)

The number of parallel execution servers associated with a single operation. Parallel execution is designed to effectively use multiple CPUs. Oracle Database parallel execution framework enables you to either explicitly choose a specific degree of parallelism or to rely on Oracle Database to automatically control it.

dense key

A numeric key that is stored as a native integer and has a range of values.

dense grouping key

A key that represents all grouping keys whose grouping columns come from a specific fact table or dimension.

dense join key

A key that represents all join keys whose join columns come from a particular fact table or dimension.

density

A decimal number between `0` and `1` that measures the selectivity of a column. Values close to `1` indicate that the column is unselective, whereas values close to `0` indicate that this column is more selective.

direct path read

A single or multiblock read into the PGA, bypassing the SGA.

driving table

The table to which other tables are joined. An analogy from programming is a for loop that contains another for loop. The outer for loop is the analog of a driving table, which is also called an **outer table**.

dynamic performance view

A view created on dynamic performance tables, which are virtual tables that record current database activity. The dynamic performance views are called fixed views because they cannot be altered or removed by the database administrator. They are also called `V$` views because they begin with the string `V$` (`GV$` in Oracle RAC).

dynamic plan

A set of subplan groups. A subplan group is set of subplans. In an adaptive query plan, the optimizer chooses a subplan at run time depending on the statistics obtained by the statistics collector.

dynamic statistics

An optimization technique in which the database executes a **recursive SQL** statement to scan a small random sample of a table's blocks to estimate predicate selectivities.

dynamic statistics level

The level that controls both when the database gathers dynamic statistics, and the size of the sample that the optimizer uses to gather the statistics. Set the dynamic statistics level using either the `OPTIMIZER_DYNAMIC_SAMPLING` initialization parameter or a statement **hint**.

enabled plan

In **SQL plan management**, a plan that is eligible for use by the optimizer.

endpoint number

A number that uniquely identifies a bucket in a **histogram**. In frequency and hybrid histograms, the endpoint number is the cumulative frequency of endpoints. In height-balanced histograms, the endpoint number is the bucket number.

endpoint repeat count

In a hybrid histogram, the number of times the **endpoint value** is repeated, for each endpoint (bucket) in the histogram. By using the repeat count, the optimizer can obtain accurate estimates for almost popular values.

endpoint value

An endpoint value is the highest value in the range of values in a **histogram** bucket.

equijoin

A join whose join condition contains an equality operator.

estimator

The component of the optimizer that determines the overall cost of a given **execution plan**.

execution plan

The combination of steps used by the database to execute a SQL statement. Each step either retrieves rows of data physically from the database or prepares them for the session issuing the statement. You can override execution plans by using a hint.

execution tree

A tree diagram that shows the flow of row sources from one step to another in an **execution plan**.

expected cardinality

For a table, the estimated number of rows the table has after all filter predicates have been applied to the table.

expression

A combination of one or more values, operators, and SQL functions that evaluates to a value. For example, the expression `2*2` evaluates to `4`. In general, expressions assume the data type of their components.

expression statistics

A type of extended statistics that improves optimizer estimates when a `WHERE` clause has predicates that use expressions.

extended statistics

A type of optimizer statistics that improves estimates for cardinality when multiple predicates exist or when predicates contain an expression.

extensible optimizer

An optimizer capability that enables authors of user-defined functions and indexes to create statistics collection, selectivity, and cost functions that the optimizer uses when choosing an execution plan. The optimizer cost model is extended to integrate information supplied by the user to assess CPU and I/O cost.

extension

A column group or an expression. The statistics collected for column groups and expressions are called **extended statistics**.

external table

A read-only table whose metadata is stored in the database but whose data in stored in files outside the database. The database uses the metadata describing external tables to expose their data as if they were relational tables.

filter condition

A `WHERE` clause component that eliminates rows from a single object referenced in a SQL statement.

final plan

In an adaptive plan, the plan that executes to completion. The **default plan** can differ from the final plan.

fixed object

A dynamic performance table or its index. The fixed objects are owned by `SYS`. Fixed object tables have names beginning with `X$` and are the base tables for the `V$` views.

fixed plan

An **accepted plan** that is marked as preferred, so that the optimizer considers only the fixed plans in the **SQL plan baseline**. You can use fixed plans to influence the plan selection process of the optimizer.

frequency histogram

A type of histogram in which each distinct column value corresponds to a single bucket. An analogy is sorting coins: all pennies go in bucket 1, all nickels go in bucket 2, and so on.

full outer join

A combination of a left and right outer join. In addition to the inner join, the database uses nulls to preserve rows from both tables that have not been returned in the result of the inner join. In other words, full outer joins join tables together, yet show rows with no corresponding rows in the joined tables.

full table scan

A scan of table data in which the database sequentially reads all rows from a table and filters out those that do not meet the selection criteria. All data blocks under the high water mark are scanned.

global temporary table

A special temporary table that stores intermediate session-private data for a specific duration.

hard parse

The steps performed by the database to build a new executable version of application code. The database must perform a hard parse instead of a soft parse if the parsed representation of a submitted statement does not exist in the **shared SQL area**.

hash cluster

A type of table cluster that is similar to an indexed cluster, except the index key is replaced with a hash function. No separate cluster index exists. In a hash cluster, the data is the index.

hash collision

Hashing multiple input values to the same output value.

hash function

A function that operates on an arbitrary-length input value and returns a fixed-length hash value.

hash join

A method for joining large data sets. The database uses the smaller of two data sets to build a hash table on the join key in memory. It then scans the larger data set, probing the hash table to find the joined rows.

hash scan

An access path for a table cluster. The database uses a hash scan to locate rows in a hash cluster based on a hash value. In a hash cluster, all rows with the same hash value are stored in the same data block. To perform a hash scan, Oracle Database first obtains the hash value by applying a hash function to a cluster key value specified by the statement, and then scans the data blocks containing rows with that hash value.

hash table

An in-memory data structure that associates join keys with rows in a hash join. For example, in a join of the `employees` and `departments` tables, the join key might be the department ID. A hash function uses the join key to generate a hash value. This hash value is an index in an array, which is the hash table.

hash value

In a hash cluster, a unique numeric ID that identifies a bucket. Oracle Database uses a hash function that accepts an infinite number of hash key values as input and sorts them into a finite number of buckets. Each hash value maps to the database block address for the block that stores the rows corresponding to the hash key value (department 10, 20, 30, and so on).

hashing

A mathematical technique in which an infinite set of input values is mapped to a finite set of output values, called hash values. Hashing is useful for rapid lookups of data in a hash table.

heap-organized table

A table in which the data rows are stored in no particular order on disk. By default, `CREATE TABLE` creates a heap-organized table.

height-balanced histogram

A histogram in which column values are divided into buckets so that each bucket contains approximately the same number of rows.

hint

An instruction passed to the **optimizer** through comments in a SQL statement. The optimizer uses hints to choose an execution plan for the statement.

histogram

A special type of column statistic that provides more detailed information about the data distribution in a table column.

hybrid hash distribution technique

An adaptive parallel data distribution that does not decide the final data distribution method until execution time.

hybrid histogram

An enhanced height-based histogram that stores the exact frequency of each endpoint in the sample, and ensures that a value is never stored in multiple buckets.

hybrid partitioned table

A table in which some partitions are stored in data file segments and some are stored in external data source.

implicit query

A component of a DML statement that retrieves data without a **subquery**. An `UPDATE`, `DELETE`, or `MERGE` statement that does not explicitly include a `SELECT` statement uses an implicit query to retrieve the rows to be modified.

In-Memory scan

A table scan that retrieves rows from the In-Memory Column Store (IM column store).

incremental statistics maintenance

The ability of the database to generate global statistics for a partitioned table by aggregating partition-level statistics.

index

Optional schema object associated with a nonclustered table, table partition, or table cluster. In some cases indexes speed data access.

index cluster

An table cluster that uses an index to locate data. The cluster index is a B-tree index on the cluster key.

index clustering factor

A measure of row order in relation to an indexed value such as employee last name. The more scattered the rows among the data blocks, the lower the clustering factor.

index fast full scan

A scan of the index blocks in unsorted order, as they exist on disk. This scan reads the index instead of the table.

index full scan

The scan of an entire index in key order.

index-organized table

A table whose storage organization is a variant of a primary B-tree index. Unlike a heap-organized table, data is stored in primary key order.

index range scan

An index range scan is an ordered scan of an index that has the following characteristics:

- One or more leading columns of an index are specified in conditions.
- 0, 1, or more values are possible for an index key.

index range scan descending

An index range scan in which the database returns rows in descending order.

index skip scan

An index scan occurs in which the initial column of a composite index is "skipped" or not specified in the query. For example, if the composite index key is `(cust_gender,cust_email)`, then the query predicate does not reference the `cust_gender` column.

index statistics

Statistics about indexes that the **optimizer** uses to determine whether to perform a full table scan or an index scan. Index statistics include B-tree levels, leaf block counts, the index clustering factor, distinct keys, and number of rows in the index.

index unique scan

A scan of an index that returns either 0 or 1 rowid.

indextype

An object that specifies the routines that manage a domain (application-specific) index.

inner join

A join of two or more tables that returns only those rows that satisfy the join condition.

inner table

In a nested loops join, the table that is not the **outer table** (driving table). For every row in the outer table, the database accesses all rows in the inner table. The outer loop is for every row in the outer table and the inner loop is for every row in the inner table.

join

A statement that retrieves data from multiple tables specified in the `FROM` clause of a SQL statement. Join types include inner joins, outer joins, and Cartesian joins.

join condition

A condition that compares two row sources using an expression. The database combines pairs of rows, each containing one row from each row source, for which the join condition evaluates to `true`.

join elimination

The removal of redundant tables from a query. A table is redundant when its columns are only referenced in join predicates, and those joins are guaranteed to neither filter nor expand the resulting rows.

join factorization

A cost-based transformation that can factorize common computations from branches of a `UNION ALL` query. Without join factorization, the optimizer evaluates each branch of a `UNION ALL` query independently, which leads to repetitive processing, including data access and joins. Avoiding an extra scan of a large base table can lead to a huge performance improvement.

join group

A user-created database object that specifies a group of columns that participate in an join. Join groups are only supported in the In-Memory column store.

join method

A method of joining a pair of row sources. The possible join methods are nested loop, sort merge, and hash joins. A Cartesian join requires one of the preceding join methods

join order

The order in which multiple tables are joined together. For example, for each row in the `employees` table, the database can read each row in the `departments` table. In an alternative join order, for each row in the `departments` table, the database reads each row in the `employees` table.

To execute a statement that joins more than two tables, Oracle Database joins two of the tables and then joins the resulting row source to the next table. This process continues until all tables are joined into the result.

join predicate

A predicate in a `WHERE` or `JOIN` clause that combines the columns of two tables in a join.

key vector

A data structure that maps between dense join keys and dense grouping keys.

latch

A low-level serialization control mechanism used to protect shared data structures in the SGA from simultaneous access.

left deep join tree

A join tree in which the left input of every join is the result of a previous join.

left table

In an outer join, the table specified on the left side of the `OUTER JOIN` keywords (in ANSI SQL syntax).

library cache

An area of memory in the shared pool. This cache includes the shared SQL areas, private SQL areas (in a shared server configuration), PL/SQL procedures and packages, and control structures such as locks and library cache handles.

library cache hit

The reuse of SQL statement code found in the library cache.

library cache miss

During SQL processing, the act of searching for a usable plan in the library cache and not finding it.

maintenance window

A contiguous time interval during which automated maintenance tasks run. The maintenance windows are Oracle Scheduler windows that belong to the window group named `MAINTENANCE_WINDOW_GROUP`.

manual plan capture

The user-initiated bulk load of existing plans into a **SQL plan baseline**.

materialized view

A schema object that stores a query result. All materialized views are either read-only or updatable.

multiblock read

An I/O call that reads multiple database blocks. Multiblock reads can significantly speed up full table scans. For example, a data block might be 8 KB, but the operating system can read 1024 KB in a single I/O. For some queries, the optimizer may decide that it is more cost-efficient to read 128 data blocks in one I/O than in 128 sequential I/Os.

NDV

Number of distinct values. The NDV is important in generating cardinality estimates.

nested loops join

A type of join method. A nested loops join determines the outer table that drives the join, and for every row in the outer table, probes each row in the inner table. The outer loop is for each row in the outer table and the inner loop is for each row in the inner table. An analogy from programming is a `for` loop inside of another `for` loop.

nonequijoin

A join whose join condition does not contain an equality operator.

nonjoin column

A predicate in a `WHERE` clause that references only one table.

nonpopular value

In a histogram, any value that does *not* span two or more endpoints. Any value that is not nonpopular is a **popular value**.

noworkload statistics

Optimizer system statistics gathered when the database simulates a workload.

on-demand SQL tuning

The manual invocation of SQL Tuning Advisor. Any invocation of SQL Tuning Advisor that is not the result of an Automatic SQL Tuning task is on-demand tuning.

optimization

The overall process of choosing the most efficient means of executing a SQL statement.

optimizer

Built-in database software that determines the most efficient way to execute a SQL statement by considering factors related to the objects referenced and the conditions specified in the statement.

optimizer cost model

The model that the optimizer uses to select an execution plan. The optimizer selects the execution plan with the lowest cost, where cost represents the estimated resource usage for that plan. The optimizer cost model accounts for the I/O, CPU, and network resources that the query will use.

optimizer environment

The totality of session settings that can affect execution plan generation, such as the work area size or optimizer settings (for example, the optimizer mode).

optimizer goal

The prioritization of resource usage by the optimizer. Using the `OPTIMIZER_MODE` initialization parameter, you can set the optimizer goal best throughput or best response time.

optimizer statistics

Details about the database its object used by the optimizer to select the best **execution plan** for each SQL statement. Categories include **table statistics** such as numbers of rows, **index statistics** such as B-tree levels, **system statistics** such as CPU and I/O performance, and **column statistics** such as number of nulls.

Optimizer Statistics Advisor

A tool that inspects statistics gathering practices, automatically diagnoses problems with these practices, and generates a report of findings and recommendations.

Optimizer Statistics Advisor rules

System-supplied standards by which Optimizer Statistics Advisor performs its checks.

optimizer statistics collection

The gathering of optimizer statistics for database objects. The database can collect these statistics automatically, or you can collect them manually by using the system-supplied `DBMS_STATS` package.

optimizer statistics collector

A row source inserted into an execution plan at key points to collect run-time statistics for use in adaptive plans.

optimizer statistics preferences

The default values of the parameters used by automatic statistics collection and the `DBMS_STATS` statistics gathering procedures.

outer join

A join condition using the outer join operator (`+`) with one or more columns of one of the tables. The database returns all rows that meet the join condition. The database also returns all rows from the table without the outer join operator for which there are no matching rows in the table with the outer join operator.

outer table

See **driving table**

parallel execution

The application of multiple CPU and I/O resources to the execution of a single database operation.

parallel query

A query in which multiple processes work together simultaneously to run a single SQL query. By dividing the work among multiple processes, Oracle Database can run the statement more quickly. For example, four processes retrieve rows for four different quarters in a year instead of one process handling all four quarters by itself.

parent cursor

The cursor that stores the SQL text and other minimal information for a SQL statement. The **child cursor** contains the plan, compilation environment, and other information. When a statement first executes, the database creates both a parent and child cursor in the shared pool.

parse call

A call to Oracle to prepare a SQL statement for execution. The call includes syntactically checking the SQL statement, optimizing it, and then building or locating an executable form of that statement.

parsing

The stage of **SQL processing** that involves separating the pieces of a SQL statement into a data structure that can be processed by other routines.

A hard parse occurs when the SQL statement to be executed is either not in the shared pool, or it is in the shared pool but it cannot be shared. A soft parse occurs when a session attempts to execute a SQL statement, and the statement is already in the shared pool, and it can be used.

partition maintenance operation

A partition-related operation such as adding, exchanging, merging, or splitting table partitions.

partition-wise join

A join optimization that divides a large join of two tables, one of which must be partitioned on the join key, into several smaller joins.

pending statistics

Unpublished optimizer statistics. By default, the optimizer uses published statistics but does not use pending statistics.

performance feedback

This form of **automatic reoptimization** helps improve the degree of parallelism automatically chosen for repeated SQL statements when `PARALLEL_DEGREE_POLICY` is set to `ADAPTIVE`.

pipelined table function

A PL/SQL function that accepts a collection of rows as input. You invoke the table function as the operand of the table operator in the `FROM` list of a `SELECT` statement.

plan evolution

The manual change of an **unaccepted plan** in the **SQL plan history** into an **accepted plan** in the **SQL plan baseline**.

plan generator

The part of the optimizer that tries different access paths, join methods, and join orders for a given query block to find the plan with the lowest cost.

plan selection

The attempt to find a matching plan in the **SQL plan baseline** for a statement after performing a hard parse.

plan verification

Comparing the performance of an **unaccepted plan** to a plan in a **SQL plan baseline** and ensuring that it performs better.

popular value

In a histogram, any value that spans two or more endpoints. Any value that is not popular is an **nonpopular value**.

predicate pushing

A transformation technique in which the optimizer "pushes" the relevant predicates from the containing query block into the view query block. For views that are not merged, this technique improves the subplan of the unmerged view because the database can use the pushed-in predicates to access indexes or to use as filters.

private SQL area

An area in memory that holds a parsed statement and other information for processing. The private SQL area contains data such as **bind variable** values, query execution state information, and query execution work areas.

private temporary table

A memory-only temporary table whose data and metadata is session-private.

proactive SQL tuning

Using SQL tuning tools to identify SQL statements that are candidates for tuning *before* users have complained about a performance problem.

See **reactive SQL tuning**, **SQL tuning**.

projection view

An optimizer-generated view that appear in queries in which a `DISTINCT` view has been merged, or a `GROUP BY` view is merged into an outer query block that also contains `GROUP BY`, `HAVING`, or aggregates.

See **simple view merging**, **complex view merging**.

query

An operation that retrieves data from tables or views. For example, `SELECT * FROM employees` is a query.

query block

A top-level `SELECT` statement, **subquery**, or unmerged view

query optimizer

See **optimizer**.

reactive SQL tuning

Diagnosing and fixing SQL-related performance problems *after* users have complained about them.

See **proactive SQL tuning**, **SQL tuning**.

real-time statistics

Supplemental statistics collected automatically during conventional DML operations.

recursive SQL

Additional SQL statements that the database must issue to execute a SQL statement issued by a user. The generation of recursive SQL is known as a recursive call. For example, the database generates recursive calls when data dictionary information is not available in memory and so must be retrieved from disk.

reoptimization

See **automatic reoptimization**.

repeatable SQL statement

A statement that the database parses or executes after recognizing that it is tracked in the **SQL statement log**.

response time

The time required to complete a unit of work.

See **throughput**.

result set

In a query, the set of rows generated by the execution of a cursor.

right deep join tree

A join tree in which the right input of every join is the result of a previous join, and the left child of every internal node of a join tree is a table.

right table

In an outer join, the table specified on the right side of the `OUTER JOIN` keywords (in ANSI SQL syntax).

rowid

A globally unique address for a row in a table.

row set

A set of rows returned by a step in an **execution plan**.

row source

An iterative control structure that processes a set of rows in an iterated manner and produces a row set.

row source generator

Software that receives the optimal plan from the optimizer and outputs the execution plan for the SQL statement.

row source tree

A collection of row sources produced by the row source generator. The row source tree for a SQL statement shows information such as table order, access methods, join methods, and data operations such as filters and sorts.

rule filter

The use of `DBMS_STATS.CONFIGURE_ADVISOR_RULE_FILTER` to restrict an Optimizer Statistics Advisor task to a user-specified set of rules. For example, you might exclude the rule that checks for stale statistics.

sample table scan

A scan that retrieves a random sample of data from a simple table or a complex `SELECT` statement, such as a statement involving joins and views.

sampling

Gathering statistics from a random subset of rows in a table.

selectivity

A value indicating the proportion of a row set retrieved by a predicate or combination of predicates, for example, `WHERE last_name = 'Smith'`. A selectivity of `0` means that no rows pass the predicate test, whereas a value of `1` means that all rows pass the test.

The adjective *selective* means roughly "choosy." Thus, a highly selective query returns a low proportion of rows (selectivity close to `0`), whereas an **unselective** query returns a high proportion of rows (selectivity close to `1`).

semijoin

A join that returns rows from the first table when at least one match exists in the second table. For example, you list departments with at least one employee. The difference between a semijoin and a conventional join is that rows in the first table are returned at most once. Semijoins use the `EXISTS` or `IN` constructs.

shared cursor

A shared SQL area that is used by multiple SQL statements.

shared pool

Portion of the SGA that contains shared memory constructs such as shared SQL areas.

shared SQL area

An area in the shared pool that contains the parse tree and **execution plan** for a SQL statement. Only one shared SQL area exists for a unique statement. The shared SQL area is sometimes referred to as the **cursor cache**.

simple database operation

A database operation consisting of a single SQL statement or PL/SQL procedure or function.

simple view merging

The merging of select-project-join views. For example, a query joins the `employees` table to a subquery that joins the `departments` and `locations` tables.

SMB

See SQL management base (SMB).

snowflake schema

A star schema in which dimension tables reference other tables.

snowstorm schema

A combination of multiple snowflake schemas.

soft parse

Any parse that is not a hard parse. If a submitted SQL statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code. This reuse of code is also called a **library cache hit**.

sort merge join

A type of join method. The join consists of a sort join, in which both inputs are sorted on the join key, followed by a merge join, in which the sorted lists are merged.

SQL Access Advisor

SQL Access Advisor is internal diagnostic software that recommends which materialized views, indexes, and materialized view logs to create, drop, or retain.

SQL compilation

In the context of Oracle SQL processing, this term refers collectively to the phases of parsing, optimization, and plan generation.

SQL handle

A string value derived from the numeric SQL signature. Like the signature, the handle uniquely identifies a SQL statement. It serves as a SQL search key in user APIs.

SQL ID

For a specific SQL statement, the unique identifier of the parent cursor in the library cache. A hash function applied to the text of the SQL statement generates the SQL ID. The `V$SQL.SQL_ID` column displays the SQL ID.

SQL incident

In the fault diagnosability infrastructure of Oracle Database, a single occurrence of a SQL-related problem. When a problem (critical error) occurs multiple times, the database creates an incident for each occurrence. Incidents are timestamped and tracked in the Automatic Diagnostic Repository (ADR).

SQL management base (SMB)

A logical repository that stores statement logs, plan histories, SQL plan baselines, and SQL profiles. The SMB is part of the data dictionary and resides in the `SYSAUX` tablespace.

SQL management object

A feature that stabilizes the execution plans of individual SQL statements. Examples include SQL profiles, SQL plan baselines, and SQL patches.

SQL plan baseline

A set of one or more accepted plans for a repeatable SQL statement. Each accepted plan contains a set of hints, a plan hash value, and other plan-related information. SQL plan management uses SQL plan baselines to record and evaluate the execution plans of SQL statements over time.

SQL plan capture

Techniques for capturing and storing relevant information about plans in the SQL management base (SMB) for a set of SQL statements. Capturing a plan means making SQL plan management aware of this plan.

SQL plan directive

Additional information and instructions that the optimizer can use to generate a more optimal plan. For example, a SQL plan directive might instruct the optimizer to collect missing statistics or gather dynamic statistics.

SQL plan history

The set of captured execution plans. The history contains both SQL plan baselines and unaccepted plans.

SQL plan management

SQL plan management is a preventative mechanism that records and evaluates the execution plans of SQL statements over time. SQL plan management can prevent SQL plan regressions caused by environmental changes such as a new optimizer version, changes to optimizer statistics, system settings, and so on.

SQL processing

The stages of parsing, optimization, row source generation, and execution of a SQL statement.

SQL profile

A set of auxiliary information built during automatic tuning of a SQL statement. A SQL profile is to a SQL statement what statistics are to a table. The optimizer can use SQL profiles to improve cardinality and selectivity estimates, which in turn leads the optimizer to select better plans.

SQL profiling

The verification and validation by the Automatic Tuning Advisor of its own estimates.

SQL signature

A numeric hash value computed using a SQL statement text that has been normalized for case insensitivity and white space. It uniquely identifies a SQL statement. The database uses this signature as a key to maintain SQL management objects such as SQL profiles, SQL plan baselines, and SQL patches.

SQL statement log

When automatic SQL plan capture is enabled, a log that contains the SQL ID of SQL statements that the optimizer has evaluated over time. A statement is tracked when it exists in the log.

SQL test case

A problematic SQL statement and related information needed to reproduce the execution plan in a different environment. A SQL test case is stored in an Oracle Data Pump file.

SQL test case builder

A database feature that gathers information related to a SQL statement and packages it so that a user can reproduce the problem on a different database. The `DBMS_SQLDIAG` package is the interface for SQL test case builder.

SQL trace file

A server-generated file that provides performance information on individual SQL statements. For example, the trace file contains parse, execute, and fetch counts, CPU and elapsed times, physical reads and logical reads, and misses in the library cache.

SQL tuning

The process of improving SQL statement efficiency to meet measurable goals.

SQL Tuning Advisor

Built-in database diagnostic software that optimizes high-load SQL statements.

See Automatic SQL Tuning Advisor.

SQL tuning set (STS)

A database object that includes one or more SQL statements along with their execution statistics and execution context.

star schema

A relational schema whose design represents a dimensional data model. The star schema consists of one or more fact tables and one or more dimension tables that are related through foreign keys.

statistics feedback

A form of automatic reoptimization that automatically improves plans for repeated queries that have cardinality misestimates. The optimizer may estimate cardinalities incorrectly for many reasons, such as missing statistics, inaccurate statistics, or complex predicates.

stored outline

A set of hints for a SQL statement. The hints in stored outlines direct the optimizer to choose a specific plan for the statement.

subplan

A portion of an adaptive plan that the optimizer can switch to as an alternative at run time. A subplan can consist of multiple operations in the plan. For example, the optimizer can treat a join method and the corresponding access path as one unit when determining whether to change the plan at run time.

subplan group

A set of subplans in an adaptive query plan.

subquery

A **query** nested within another SQL statement. Unlike implicit queries, subqueries use a `SELECT` statement to retrieve data.

subquery unnesting

A transformation technique in which the optimizer transforms a nested query into an equivalent join statement, and then optimizes the join.

synopsis

A set of auxiliary statistics gathered on a partitioned table when the `INCREMENTAL` value is set to `true`.

system statistics

Statistics that enable the **optimizer** to use CPU and I/O characteristics. Index statistics include B-tree levels, leaf block counts, clustering factor, distinct keys, and number of rows in the index.

table cluster

A schema object that contains data from one or more tables, all of which have one or more columns in common. In table clusters, the database stores together all the rows from all tables that share the same cluster key.

table expansion

A transformation technique that enables the optimizer to generate a plan that uses indexes on the read-mostly portion of a partitioned table, but not on the active portion of the table.

table statistics

Statistics about tables that the **optimizer** uses to determine table access cost, join cardinality, join order, and so on. Table statistics include row counts, block counts, empty blocks, average free space per block, number of chained rows, average row length, and staleness of the statistics on the table.

throughput

The amount of work completed in a unit of time.

See **response time**.

top frequency histogram

A variation of a **frequency histogram** that ignores nonpopular values that are statistically insignificant, thus producing a better histogram for highly popular values.

tuning mode

One of the two optimizer modes. When running in tuning mode, the optimizer is known as the **Automatic Tuning Optimizer**. In tuning mode, the optimizer determines whether it can further improve the plan produced in normal mode. The optimizer output is not an execution plan, but a series of actions, along with their rationale and expected benefit for producing a significantly better plan.

unaccepted plan

A plan for a statement that is in the **SQL plan history** but has not been added to the **SQL plan management**.

unselective

A relatively large fraction of rows from a row set. A query becomes more unselective as the **selectivity** approaches `1`. For example, a query that returns 999,999 rows from a table with one million rows is unselective. A query of the same table that returns one row is selective.

user response time

The time between when a user submits a command and receives a response.

See **throughput**.

V$ view

See **dynamic performance view**.

vector I/O

A type of I/O in which the database obtains a set of rowids, sends them batched in an array to the operating system, which performs the read.

view merging

The merging of a query block representing a view into the query block that contains it. View merging can improve plans by enabling the optimizer to consider additional join orders, access methods, and other transformations.

workload statistics

Optimizer statistics for system activity in a specified time period.

Index