19

Influencing the Optimizer

Optimizer defaults are adequate for most operations, but not all.

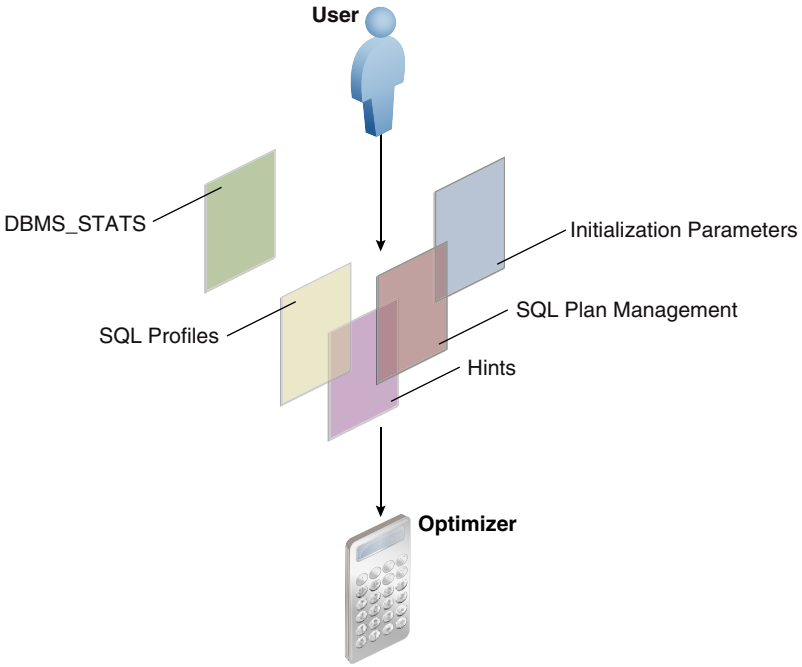
In some cases you may have information unknown to the optimizer, or need to tune the optimizer for a specific type of statement or workload. In such cases, influencing the optimizer may provide better performance.

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

Influencing the Optimizer with Initialization Parameters

This chapter explains which initialization parameters affect optimization, and how to set them.

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.
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".

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
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.
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.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
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).
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.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
RESULT_CACHE_INTEGRITY	Determines whether queries with objects that cannot be identified as being deterministic (such as PL/SQL function not declared as deterministic) will be considered for result caching. Enforcing the requirement rules out the chance of accidentally caching objects that should not be cached. `RESULT_CACHE_INTEGRITY = { ENFORCED | TRUSTED }` Values: • ENFORCED Regardless of the setting of RESULT_CACHE_MODE or specified hints, only deterministic constructs are eligible for result caching. For example, when the value is ENFORCED, queries using PL/SQL functions that are not declared as deterministic are never cached and must be declared as deterministic. • TRUSTED The database honors the setting of RESULT_CACHE_MODE and specified hints and will consider queries using possibly nondeterministic constructs as candidates for result caching. For example, queries using PL/SQL functions that are not declared as deterministic can be cached. However, results that are known to be nondeterministic such as those from queries using SYSDATE in any form are not cached. The default value of RESULT_CACHE_INTEGRITY is TRUSTED, except for Autonomous Database. The default for Autonomous Database is ENFORCED. **Note:** Setting RESULT_CACHE_INTEGRITY to TRUSTED achieves backward compatibility with Oracle Databases prior to Release 23ai, before the introduction of this parameter.

Table 19-2 (Cont.) Initialization Parameters That Control Optimizer Behavior

Initialization Parameter	Description
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.
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

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

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

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

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

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:

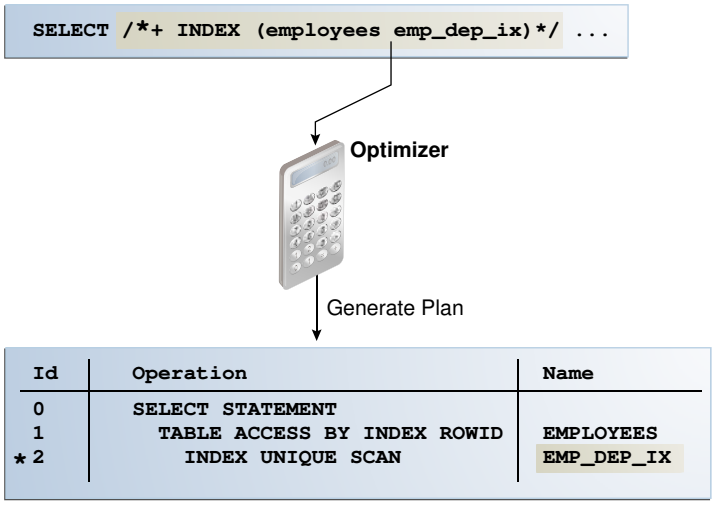
```
SELECT /*+ FULL (hr_emp) CACHE(hr_emp) */ last_name FROM employees hr_emp;
```

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



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.

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.

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

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`

Reporting on Hints

An explain plan includes a report showing which hints were used during plan generation.

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

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

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`

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.

SQL Analysis Report

SQL analysis checks your SQL statement and reports on changes that will improve SQL execution performance and also constructs that hinder the ability of the optimizer to generate optimal SQL execution plans.

SQL analysis is enabled by default.

User Interface for SQL Analysis Report

The examples below describe the `DBMS_XPLAN` interface that provides the SQL Analysis Report. The report includes the analysis of the SQL statement. Note that there is also an SQL Analysis section in the SQL Monitor Report, which you can view with `DBMS_SQLTUNE.REPORT_SQL_MONITOR`.

Report Access

You can generate variants of the report by using the following DBMS_XPLAN functions:

- `DISPLAY`
- `DISPLAY_CURSOR`
- `DISPLAY_WORKLOAD_REPOSITORY`
- `DISPLAY_SQL_PLAN_BASELINE`
- `DISPLAY_SQLSET`

In the examples below, the `DBMS_XPLAN.DISPLAY_CURSOR` function is used to show the execution plan of the loaded cursor. The plan includes the SQL analysis report.

Formatting the SQL Analysis Report

The SQL analysis report has two format options: `TYPICAL` and `BASIC`. The default value is `'TYPICAL'`:

```
SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR());
```

You can also make the format value explicit:

```
SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(format=>'typical'));
```

The report can be removed from the `TYPICAL` format as follows. This excludes thet SQL analysis report.

```
SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(format=>'typical -
sql_analysis_report'));
```

To display the SQL analysis report with a `BASIC` format, specify the additional values `ALIAS` and `SQL_ANALYSIS_REPORT` in the format parameter. For example:

```
SELECT * FROM table(DBMS_XPLAN.DISPLAY_CURSOR(format=>'basic alias
sql_analysis_report'));
```

SQL Analysis Report Examples

Example 19-8 SQL Analysis Report Finds an Expensive Cartesian Product and Recommends JOINs or Removal of Disconnected Tables and Views

```
SQL> explain plan for
  2  select sum(quantity_sold)
  3  from   sales s,
  4         products p
  5  where  p.prod_name = 'Lion trouser Set'
  6  /

Explained.

SQL>
SQL> SELECT * FROM table(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
----------------------
Plan hash value: 1967659834

--------------------------------------------------------------------------------
--------------------
| Id  | Operation              | Name     | Rows  | Bytes | Cost (%CPU)|
Time     | Pstart| Pstop |
--------------------------------------------------------------------------------
--------------------
```

```
|   0 | SELECT STATEMENT       |          |     1 |    30 |    17   (0)|
00:00:01 |       |       |
|   1 |  SORT AGGREGATE        |          |     1 |    30 |
|          |       |       |
|   2 |   MERGE JOIN CARTESIAN |          |  1848 | 55440 |    17   (0)|
00:00:01 |       |       |
|*  3 |    TABLE ACCESS FULL   | PRODUCTS |     2 |    54 |     9   (0)|
00:00:01 |       |       |
|   4 |    BUFFER SORT         |          |   960 |  2880 |     8   (0)|
00:00:01 |       |       |
|   5 |     PARTITION RANGE ALL|          |   960 |  2880 |     4   (0)|
00:00:01 |     1 |    16 |
|   6 |      TABLE ACCESS FULL | SALES    |   960 |  2880 |     4   (0)|
00:00:01 |     1 |    16 |
--------------------------------------------------------------------------------
--------------------

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

   3 - filter("P"."PROD_NAME"='Lion trouser Set')

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

   1 -  SEL$1
          -  The query block has 1 cartesian product which may be
             expensive. Consider adding join conditions or removing the
             disconnected tables or views.

26 rows selected.
```

Example 19-9 SQL Analysis Report Finds and Expensive UNION and Recommends UNION ALL

```
SQL>
SQL> explain plan for
  2  select prod_name
  3  from   products
  4  where  prod_subcategory = 'Shirts - Girls'
  5  union
  6  select prod_name
  7  from   products
  8  where  prod_subcategory = 'Shirts - Boys'
  9  /

Explained.

SQL>
SQL> SELECT * FROM table(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
```

```
-------------------------------------------------------------------------------
----------------------
Plan hash value: 699097881

-------------------------------------------------------------------------------
--
| Id  | Operation           | Name     | Rows  | Bytes | Cost (%CPU)|
Time     |
-------------------------------------------------------------------------------
--
|   0 | SELECT STATEMENT    |          |    53 |  2385 |    20  (10)|
00:00:01 |
|   1 |  HASH UNIQUE        |          |    53 |  2385 |    20  (10)|
00:00:01 |
|   2 |   UNION-ALL         |          |    53 |  2385 |    20  (10)|
00:00:01 |
|*  3 |    TABLE ACCESS FULL| PRODUCTS |    24 |  1080 |     9   (0)|
00:00:01 |
|*  4 |    TABLE ACCESS FULL| PRODUCTS |    29 |  1305 |     9   (0)|
00:00:01 |
-------------------------------------------------------------------------------
--

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

   3 - filter("PROD_SUBCATEGORY"='Shirts - Girls')
   4 - filter("PROD_SUBCATEGORY"='Shirts - Boys')

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

   1 -  SET$1
           -  The query block contains UNION which may be expensive.
              Consider using UNION ALL if duplicates are allowed or
              uniqueness is guaranteed.

25 rows selected.
```

Example 19-10 SQL Analysis Report Finds and Reports Columns Containing Predicates That Should be Rewritten

```
SQL>
SQL> explain plan for
  2  select prod_name
  3  from   products
  4  where  prod_subcategory like '%Girls%'
  5  /

Explained.

SQL>
```

```
SQL> SELECT * FROM table(DBMS_XPLAN.DISPLAY());

PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------
----------------------
Plan hash value: 1954719464

-------------------------------------------------------------------------------
| Id  | Operation          | Name     | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |          |    38 |  1710 |     9   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| PRODUCTS |    38 |  1710 |     9   (0)| 00:00:01 |
-------------------------------------------------------------------------------

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

   1 - filter("PROD_SUBCATEGORY" LIKE '%Girls%')

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

   1 -  SEL$1 / "PRODUCTS"@"SEL$1"
           -  The following columns have predicates which preclude their
              use as keys in index range scan. Consider rewriting the
              predicates.
                "PROD_SUBCATEGORY"

22 rows selected.
```