6

# Explaining and Displaying Execution Plans

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

## Introduction to Execution Plans

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

### 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

## 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

### 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.

### 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

# 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.

## 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

## 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.

## 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

# 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
```

Plan operations request data from their children. The execution order in EXPLAIN PLAN output is as follows:

1. Execution starts at the first operation with no children, which in the example above is the full scan of `EMPLOYEES` (Id 3).
2. `EMPLOYEES` returns its data to the parent (Id 2).
3. Execution then proceeds to next child of the hash join and does a full scan of `DEPARTMENTS` (Id 4).
4. `DEPARTMENTS` has no children and so returns data to the parent (Id 2).
5. The hash join combines the rows from the two tables and passes them up to the `SORT ORDER BY` (Id 1)
6. Finally the `SELECT` returns the data to the client.

> **Note:**
>
> If this example included more operations such as additional joins, execution would continue from Step 5 following the same pattern for each operation down to the end of the plan, the final step where the `SELECT` returns data to the client.
> The steps in the EXPLAIN PLAN output as described here may be different on some of your databases. This is because the optimizer may choose a different EXECUTION PLAN, depending on the database configuration.

**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`
- *How to Read an Execution Plan*. This Oracle blog post describes how to read an EXECUTION PLAN, but the same order of execution applies to an EXPLAIN PLAN, so it may give you a better understanding of the process in both types of plan.

## 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;
```

## 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`

# 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.

# Displaying Execution Plans

The easiest way to display execution plans is to use DBMS_XPLAN display functions or V$ views.

## 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

## 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.<br>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$SQL_PLAN_STATISTICS_ALL` if statistics are included). You can apply a predicate on the specified table to select rows of the plan to display.<br>The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, and `ALL`. |
| `DISPLAY_AWR` | This table function displays the contents of an execution plan stored in AWR.<br>The `format` parameter controls the level of the plan. It accepts the values `BASIC`, `TYPICAL`, `SERIAL`, and `ALL`. |
| `DISPLAY_CURSOR` | This table function displays the explain plan of any cursor loaded in the cursor cache. In addition to the explain plan, various plan statistics (such as. I/O, memory and timing) can be reported (based on the `V$SQL_PLAN_STATISTICS_ALL VIEWS`).<br>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:<br>• 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.<br>• Recommended plan. In reporting mode, the output includes the plan that would be chosen based on execution statistics.<br>• Dynamic plan. The output summarizes the portions of the plan that differ from the default plan chosen by the optimizer.<br>• Reoptimization. The output displays the plan that would be chosen on a subsequent execution because of reoptimization. |

**Table 6-1 (Cont.) DBMS_XPLAN Display Functions**

| Display Functions | Notes |
|---|---|
| DISPLAY_PLAN | This table function displays the contents of the plan table in a variety of formats with CLOB output type.<br>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.<br>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.<br>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.<br>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

## 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$SQL | Lists statistics for cursors and contains one row for each child of the original SQL text entered.<br>Starting in Oracle Database 19c, V$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$SQL_SHARED_CURSOR | Explains why a particular child cursor is not shared with existing child cursors. Each column identifies a specific reason why the cursor cannot be shared.<br>The USE_FEEDBACK_STATS column shows whether a child cursor fails to match because of reoptimization. |
| V$SQL_PLAN | Contains the plan for every statement stored in the shared SQL area.<br>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.<br>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. |
| 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.<br>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

## 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

# 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`

# Display Execution Plans: Examples

These examples show different ways of displaying execution plans.

## 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.

# Displaying Parallel Execution Plans: Example

Plans for parallel queries differ in important ways from plans for serial queries.

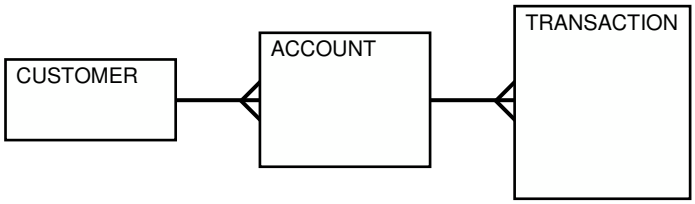
## About EXPLAIN PLAN and Parallel Queries

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"

## 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.

## 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
```

## 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
```

## 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.

### 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.

## 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.

## 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.

## 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`.

## 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.

### 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.

## 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)
```

## 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
```

# 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
```

# 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.

## 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.

## 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 `generic_plan_object` 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:<br>• `owner`—The owner of the plan table<br>• `plan_table_name`—The name of the plan table<br>• `statement_id`—The ID of the statement (optional)<br>• `plan_id`—The ID of the plan (optional) |
| Cursor cache | `cursor_cache_object(sql_id, child_number)` | The parameters are as follows:<br>• `sql_id`—The SQL ID of the plan<br>• `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:<br>• `sql_id`—The SQL ID of the plan<br>• `dbid`—The database ID (optional)<br>• `con_dbid`—The CDB ID (optional)<br>• `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:<br>• `sqlset_owner`—The owner of the SQL tuning set<br>• `sqlset_name`—The name of the SQL tuning set<br>• `sql_id`—The SQL ID of the plan<br>• `plan_hash_value`—The hash value of the plan (optional) |
| SQL plan management | `spm_object (sql_handle, plan_name)` | The parameters are as follows:<br>• `sql_handle`—The SQL handle of plans protected by SQL plan management<br>• `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:<br>• `task_name`—The name of the advisor task<br>• `execution_name`—The name of the task execution<br>• `sql_id`—The SQL ID of the plan<br>• `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`. |
| `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

# 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:**

> **Note:**
>
> As of Oracle Database 23ai, for `SELECT`s that do not require table access, `FROM DUAL` is now implicit when there is no `FROM` clause. `FROM DUAL` is supported but is no longer required. `SELECT <expr_list>;` is sufficient. Also note that these operations still appear in the execution plan as `FAST DUAL`.

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

# 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)
-----------------------------------------------------------------------------
```