Interpreting Execution Plans

Chapter 4

Interpreting Execution Plans

4

ORACLE

Objectives

After completing this lesson, you should be able to:

- Gather execution plans
- Display execution plans
- Interpret execution plans

What Is an Execution Plan?

- The execution plan of a SQL statement is composed of small building blocks called row sources for serial execution plans.
- The combination of row sources for a statement is called the execution plan.
- By using parent-child relationships, the execution plan can be displayed in a tree-like structure (text or graphical).

ORACLE

What Is an Execution Plan?

An execution plan is the output of the optimizer and is presented to the execution engine for implementation. It instructs the execution engine about the operations it must perform for retrieving the data required by a query most efficiently.

The `EXPLAIN PLAN` statement gathers execution plans chosen by the Oracle optimizer for the `SELECT`, `UPDATE`, `INSERT`, and `DELETE` statements. The steps of the execution plan are not performed in the order in which they are numbered. There is a parent-child relationship between steps. The row source tree is the core of the execution plan. It shows the following information:

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

In addition to the row source tree (or data flow tree for parallel operations), 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

The EXPLAIN PLAN results help you determine whether the optimizer selects a particular execution plan, such as nested loops join.

Where to Find Execution Plans?

- `PLAN_TABLE` (SQL Developer or SQL*Plus)
- `V$SQL_PLAN` (Library Cache)
- `V$SQL_PLAN_MONITOR` (11*g*)
- `DBA_HIST_SQL_PLAN` (AWR)
- `STATS$SQL_PLAN` (Statspack)
- SQL management base (SQL plan baselines)
- SQL tuning set
- Trace files generated by `DBMS_MONITOR`
- Event 10053 trace file
- Process state dump trace file since 10*g*R2

ORACLE

Where to Find Execution Plans?

There are many ways to retrieve execution plans inside the database. The most well-known ones are listed in the slide:

- The `EXPLAIN PLAN` command enables you to view the execution plan that the optimizer might use to execute a SQL statement. This command is very useful because it outlines the plan that the optimizer may use and inserts it in a table called `PLAN_TABLE` without executing the SQL statement. This command is available from SQL*Plus or SQL Developer.
- `V$SQL_PLAN` provides a way to examine the execution plan for cursors that were recently executed. Information in `V$SQL_PLAN` is very similar to the output of an `EXPLAIN PLAN` statement. However, while `EXPLAIN PLAN` shows a theoretical plan that can be used if this statement was executed, `V$SQL_PLAN` contains the actual plan used.
- `V$SQL_PLAN_MONITOR` displays plan-level monitoring statistics for each SQL statement found in `V$SQL_MONITOR`. Each row in `V$SQL_PLAN_MONITOR` corresponds to an operation of the execution plan that is monitored.
- The Automatic Workload Repository (AWR) infrastructure and Statspack store execution plans of top SQL statements. Plans are recorded into `DBA_HIST_SQL_PLAN` or `STATS$SQL_PLAN`.

- Plan and row source operations are dumped in trace files generated by DBMS_MONITOR.
- The SQL management base (SMB) is a part of the data dictionary that resides in the SYSAUX tablespace. It stores statement log, plan histories, and SQL plan baselines, as well as SQL profiles.
- The event 10053, which is used to dump cost-based optimizer (CBO) computations may include a plan.
- Starting with Oracle Database 10*g*, Release 2, when you dump process state (or errorstack from a process), execution plans are included in the trace file that is generated.

Viewing Execution Plans

- The EXPLAIN PLAN command followed by:
  - SELECT from PLAN_TABLE
  - DBMS_XPLAN.DISPLAY()
- SQL*Plus Autotrace: SET AUTOTRACE ON
- DBMS_XPLAN.DISPLAY_CURSOR()
- DBMS_XPLAN.DISPLAY_AWR()
- DBMS_XPLAN.DISPLAY_SQLSET()
- DBMS_XPLAN.DISPLAY_SQL_PLAN_BASELINE()

Viewing Execution Plans

If you execute the EXPLAIN PLAN SQL*Plus command, you can then SELECT from the PLAN_TABLE to view the execution plan. There are several SQL*Plus scripts available to format the plan table output. The easiest way to view an execution plan is to use the DBMS_XPLAN package. The DBMS_XPLAN package supplies five table functions:

- DISPLAY: To format and display the contents of a plan table
- DISPLAY_AWR: To format and display the contents of the execution plan of a stored SQL statement in the AWR
- DISPLAY_CURSOR: To format and display the contents of the execution plan of any loaded cursor
- DISPLAY_SQL_PLAN_BASELINE: To display one or more execution plans for the SQL statement identified by SQL handle
- DISPLAY_SQLSET: To format and display the contents of the execution plan of statements stored in a SQL tuning set

An advantage of using the DBMS_XPLAN package table functions is that the output is formatted consistently without regard to the source.

The EXPLAIN PLAN Command

The `EXPLAIN PLAN` Command

- Generates an optimizer execution plan
- Stores the plan in `PLAN_TABLE`
- Does not execute the statement itself

ORACLE

The `EXPLAIN PLAN` Command

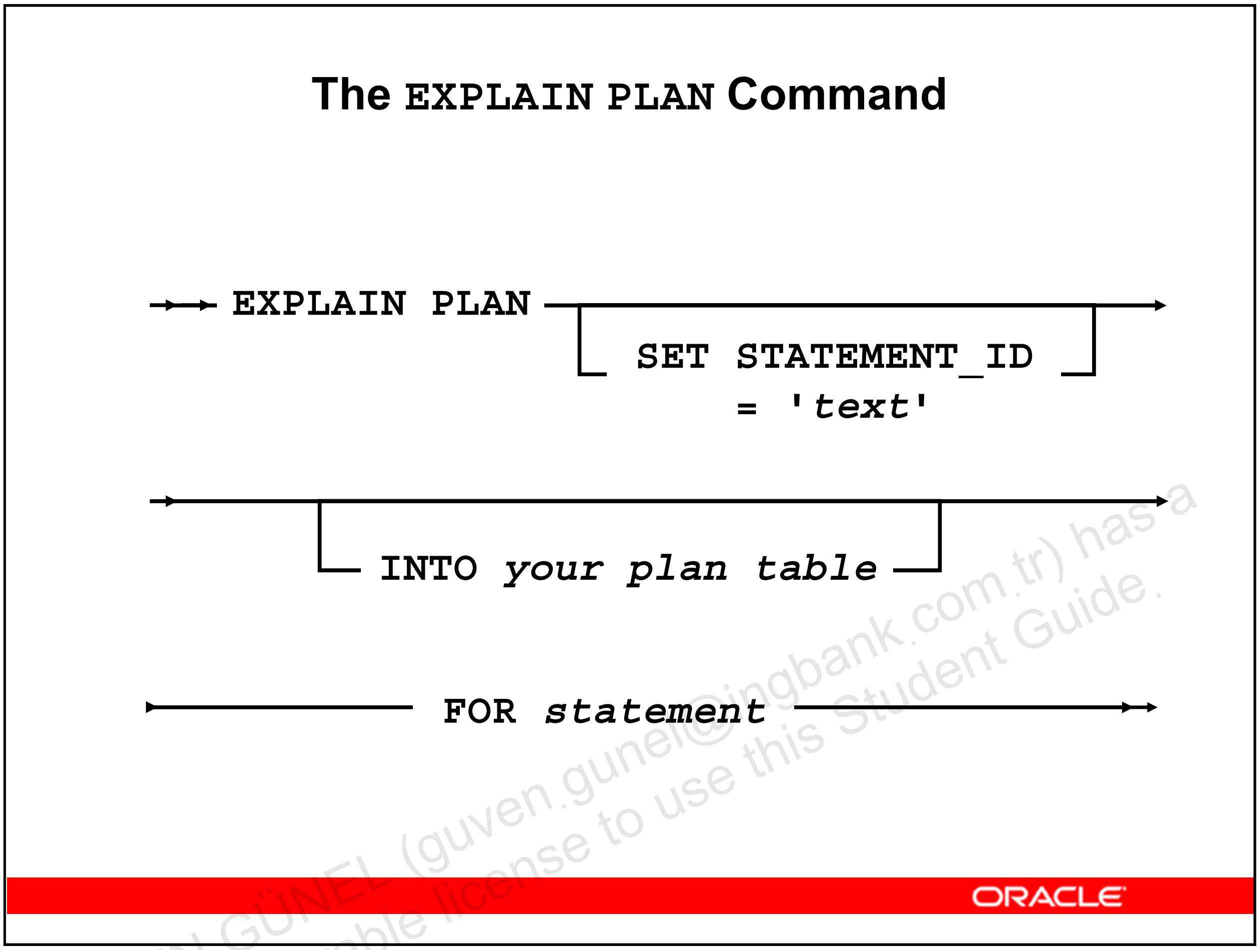
The `EXPLAIN PLAN` command is used to generate the execution plan that the optimizer uses to execute a SQL statement. It does not execute the statement, but simply produces the plan that may be used, and inserts this plan into a table. If you examine the plan, you can see how the Oracle Server executes the statement.

Using EXPLAIN PLAN

- First use the `EXPLAIN PLAN` command to explain a SQL statement.
- Then retrieve the plan steps by querying `PLAN_TABLE`.

`PLAN_TABLE` is automatically created as a global temporary table to hold the output of an `EXPLAIN PLAN` statement for all users. `PLAN_TABLE` is the default sample output table into which the `EXPLAIN PLAN` statement inserts rows describing execution plans.

Note: You can create your own `PLAN_TABLE` using the `$ORACLE_HOME/rdbms/admin/utlxplan.sql` script if you want to keep the execution plan information for a long term.



The `EXPLAIN PLAN` Command (continued)

This command inserts a row in the plan table for each step of the execution plan.
In the syntax diagram in the slide, the fields in italics have the following meanings:

The EXPLAIN PLAN Command: Example

```
SQL> EXPLAIN PLAN
  2  SET STATEMENT_ID = 'demo01' FOR
  3  SELECT e.last_name, d.department_name
  4  FROM hr.employees e, hr.departments d
  5  WHERE e.department_id = d.department_id;

Explained.

SQL>
```

Note: The `EXPLAIN PLAN` command does not actually execute the statement.

The EXPLAIN PLAN Command: Example

This command inserts the execution plan of the SQL statement in the plan table and adds the optional `demo01` name tag for future reference. You can also use the following syntax:

```
EXPLAIN PLAN
FOR
SELECT e.last_name, d.department_name
   FROM hr.employees e, hr.departments d
WHERE e.department_id =d.department_id;
```

PLAN_TABLE

- PLAN_TABLE:
  - Is automatically created to hold the EXPLAIN PLAN output.
  - You can create your own using utlxplan.sql.
  - Advantage: SQL is not executed
  - Disadvantage: May not be the actual execution plan
- PLAN_TABLE is hierarchical.
- Hierarchy is established with the ID and PARENT_ID columns.

PLAN_TABLE

There are various available methods to gather execution plans. Now, you are introduced only to the EXPLAIN PLAN statement. This SQL statement gathers the execution plan of a SQL statement without executing it, and outputs its result in the PLAN_TABLE table. Whatever the method to gather and display the explain plan, the basic format and goal are the same. However, PLAN_TABLE just shows you a plan that might not be the one chosen by the optimizer. PLAN_TABLE is automatically created as a global temporary table and is visible to all users. PLAN_TABLE is the default sample output table into which the EXPLAIN PLAN statement inserts rows describing execution plans. PLAN_TABLE is organized in a tree-like structure and you can retrieve that structure by using both the ID and PARENT_ID columns with a CONNECT BY clause in a SELECT statement. While a PLAN_TABLE table is automatically set up for each user, you can use the utlxplan.sql SQL script to manually create a local PLAN_TABLE in your schema and use it to store the results of EXPLAIN PLAN. The exact name and location of this script depends on your operating system. On UNIX, it is located in the $ORACLE_HOME/rdbms/admin directory. It is recommended that you drop and rebuild your local PLAN_TABLE table after upgrading the version of the database because the columns might change. This can cause scripts to fail or cause TKPROF to fail, if you are specifying the table.

Note: If you want an output table with a different name, first create PLAN_TABLE manually with the utlxplan.sql script, and then rename the table with the RENAME SQL statement.

Displaying from PLAN_TABLE: Typical

```
SQL> EXPLAIN PLAN SET STATEMENT_ID = 'demo01' FOR SELECT * FROM emp
  2  WHERE ename = 'KING';

Explained.

SQL> SET LINESIZE 130
SQL> SET PAGESIZE 0
SQL> select * from table(DBMS_XPLAN.DISPLAY());

Plan hash value: 3956160932
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    37 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
```

ORACLE

Displaying from PLAN_TABLE: Typical

In the example in the slide, the EXPLAIN PLAN command inserts the execution plan of the SQL statement in PLAN_TABLE and adds the optional demo01 name tag for future reference. The DISPLAY function of the DBMS_XPLAN package can be used to format and display the last statement stored in PLAN_TABLE. You can also use the following syntax to retrieve the same result: SELECT * FROM TABLE(dbms_xplan.display('plan_table','demo01','typical',null));

The output is the same as shown in the slide. In this example, you can substitute the name of another plan table instead of PLAN_TABLE and demo01 represents the statement ID. TYPICAL displays the most relevant information in the plan: operation ID, name and option, number of rows, bytes, and optimizer cost. The last parameter for the DISPLAY function is the one corresponding to filter_preds. This parameter represents a filter predicate or predicates to restrict the set of rows selected from the table where the plan is stored. When value is null (the default), the plan displayed corresponds to the last executed explain plan. This parameter can reference any column of the table where the plan is stored and can contain any SQL construct—for example, subquery or function calls.

Note: Alternatively, you can run the utlxpls.sql (or utlxplp.sql for parallel queries) script (located in the ORACLE_HOME/rdbms/admin/ directory) to display the execution plan stored in PLAN_TABLE for the last statement explained. This script uses the DISPLAY table function from the DBMS_XPLAN package.

Displaying from PLAN_TABLE: ALL

```
SQL> select * from table(DBMS_XPLAN.DISPLAY(null,null,'ALL'));

Plan hash value: 3956160932
--------------------------------------------------------------------------
| Id  | Operation         | Name | Rows  | Bytes | Cost (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |    37 |     3   (0)| 00:00:01 |
|*  1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |     3   (0)| 00:00:01 |
--------------------------------------------------------------------------
Query Block Name / Object Alias (identified by operation id):
-------------------------------------------------------------
   1 - SEL$1 / EMP@SEL$1
Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("ENAME"='KING')
Column Projection Information (identified by operation id):
-----------------------------------------------------------
 1 - "EMP"."EMPNO"[NUMBER,22], "ENAME"[VARCHAR2,10], "EMP"."JOB"[VARCHAR2,9],
     "EMP"."MGR"[NUMBER,22], "EMP"."HIREDATE"[DATE,7], "EMP"."SAL"[NUMBER,22],
     "EMP"."COMM"[NUMBER,22], "EMP"."DEPTNO"[NUMBER,22]
```

ORACLE

Displaying from PLAN_TABLE: ALL

Here you use the same EXPLAIN PLAN command example as in the previous slide. The ALL option used with the DISPLAY function allows you to output the maximum user level information. It includes information displayed with the TYPICAL level, with additional information such as PROJECTION, ALIAS, and information about REMOTE SQL, if the operation is distributed.

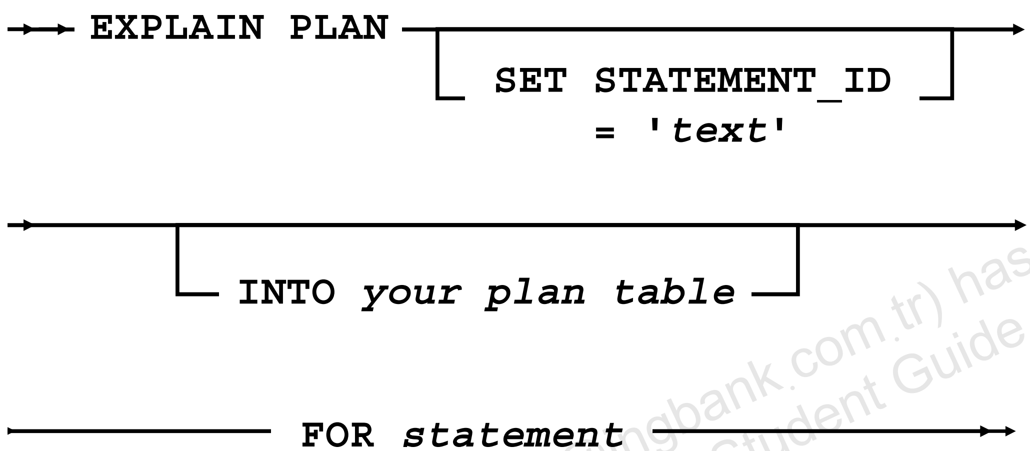
For finer control on the display output, the following keywords can be added to the format parameter to customize its default behavior. Each keyword either represents a logical group of plan table columns (such as PARTITION) or logical additions to the base plan table output (such as PREDICATE). Format keywords must be separated by either a comma or a space:

- ROWS: If relevant, shows the number of rows estimated by the optimizer
- BYTES: If relevant, shows the number of bytes estimated by the optimizer
- COST: If relevant, shows optimizer cost information
- PARTITION: If relevant, shows partition pruning information
- PARALLEL: If relevant, shows PX information (distribution method and table queue information)
- PREDICATE: If relevant, shows the predicate section

- `PROJECTION`: If relevant, shows the projection section

The EXPLAIN PLAN Command

The `EXPLAIN PLAN` Command



ORACLE

Displaying from `PLAN_TABLE`: `ALL` (continued)

- `ALIAS`: If relevant, shows the “Query Block Name/Object Alias” section
- `REMOTE`: If relevant, shows the information for the distributed query (for example, remote from serial distribution and remote SQL)
- `NOTE`: If relevant, shows the note section of the explain plan

If the target plan table also stores plan statistics columns (for example, it is a table used to capture the content of the fixed view `V$SQL_PLAN_STATISTICS_ALL`), additional format keywords can be used to specify which class of statistics to display when using the `DISPLAY` function. These additional format keywords are `IOSTATS`, `MEMSTATS`, `ALLSTATS` and `LAST`.

Note: Format keywords can be prefixed with the “`-`” sign to exclude the specified information. For example, “`-PROJECTION`” excludes projection information.

Displaying from PLAN_TABLE: ADVANCED

```
select plan_table_output from table(DBMS_XPLAN.DISPLAY(null,null,'ADVANCED
 -PROJECTION -PREDICATE -ALIAS'));
Plan hash value: 3956160932
--------------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost  (%CPU)| Time     |
--------------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |     1 |    37 |      3   (0)| 00:00:01 |
|   1 |  TABLE ACCESS FULL| EMP  |     1 |    37 |      3   (0)| 00:00:01 |
--------------------------------------------------------------------------
Outline Data
-------------
  /*+
      BEGIN_OUTLINE_DATA
      FULL(@"SEL$1" "EMP"@"SEL$1")
      OUTLINE_LEAF(@"SEL$1")
      ALL_ROWS
      DB_VERSION('11.1.0.6')
      OPTIMIZER_FEATURES_ENABLE('11.1.0.6')
      IGNORE_OPTIM_EMBEDDED_HINTS
      END_OUTLINE_DATA
  */
```

ORACLE

Displaying from PLAN_TABLE: ADVANCED

The ADVANCED format is available only from Oracle Database 10*g*, Release 2 and later versions.

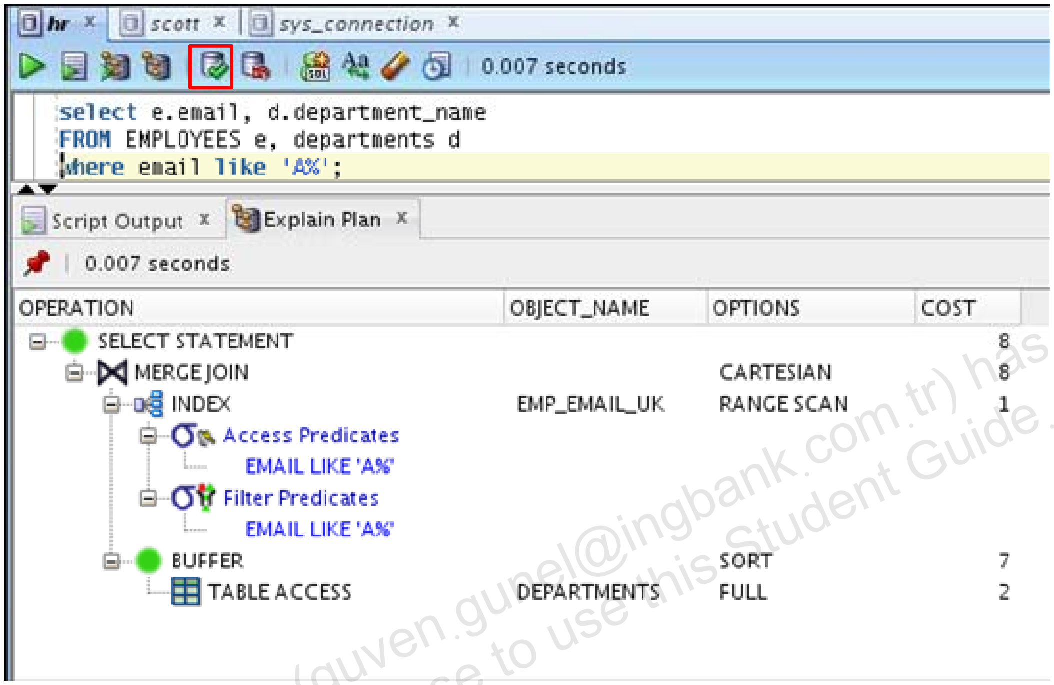
This output format includes all sections from the ALL format plus the outline data that represents a set of hints to reproduce that particular plan.

This section may be useful if you want to reproduce a particular execution plan in a different environment.

This is the same section, which is displayed in the trace file for event 10053.

Note: When the ADVANCED format is used with V$SQL_PLAN, there is one more section called Peeked Binds (identified by position).

Explain Plan Using SQL Developer



Explain Plan Using SQL Developer

The Explain Plan icon generates the execution plan, which you can see in the Explain tab. An execution plan shows a row source tree with the hierarchy of operations that make up the statement. For each operation, it shows the ordering of the tables referenced by the statement, access method for each table mentioned in the statement, join method for tables affected by join operations in the statement, and data operations such as filter, sort, or aggregation. In addition to the row source tree, the plan table displays information about optimization (such as the cost and cardinality of each operation), partitioning (such as the set of accessed partitions), and parallel execution (such as the distribution method of join inputs).

AUTOTRACE

- Is a SQL*Plus and SQL Developer facility
- Was introduced with Oracle 7.3
- Needs a `PLAN_TABLE`
- Needs the `PLUSTRACE` role to retrieve statistics from some `V$` views
- By default, produces the execution plan and statistics after running the query
- May not be the execution plan used by the optimizer when using bind peeking (recursive `EXPLAIN PLAN`)

AUTOTRACE

When running SQL statements under SQL*Plus or SQL Developer, you can automatically get a report on the execution plan and the statement execution statistics. The report is generated after successful SQL DML (that is, `SELECT`, `DELETE`, `UPDATE`, and `INSERT`) statements. It is useful for monitoring and tuning the performance of these statements.

To use this feature, you must have a `PLAN_TABLE` available in your schema, and then have the `PLUSTRACE` role granted to you. The database administrator (DBA) privileges are required to grant the `PLUSTRACE` role. The `PLUSTRACE` role is created and granted to the DBA role by running the supplied `$ORACLE_HOME/sqlplus/admin/plustrce.sql` script.

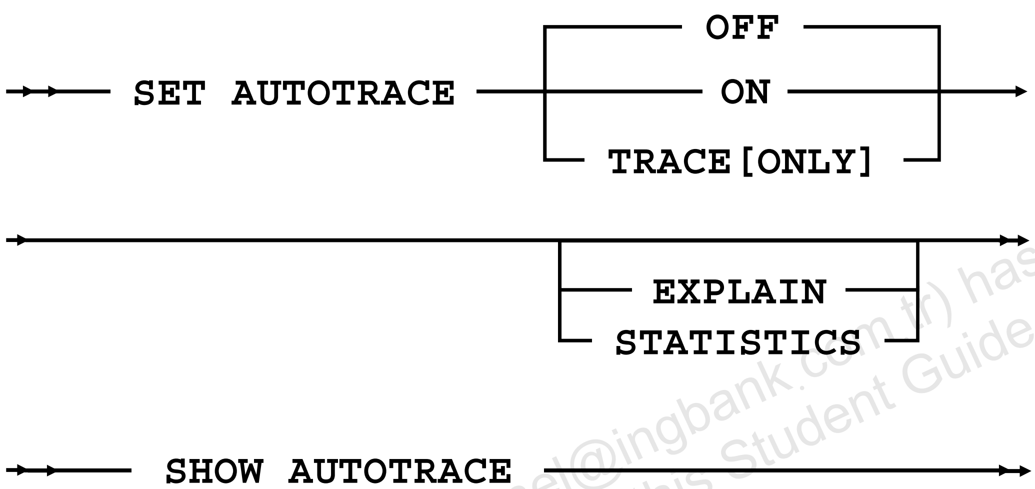
On some versions and platforms, this is run by the database creation scripts. If this is not the case on your platform, connect as `SYSDBA` and run the `plustrce.sql` script.

The `PLUSTRACE` role contains the select privilege on three `V$` views. These privileges are necessary to generate `AUTOTRACE` statistics.

`AUTOTRACE` is an excellent diagnostic tool for SQL statement tuning. Because it is purely declarative, it is easier to use than `EXPLAIN PLAN`.

Note: The system 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.

The AUTOTRACE Syntax



ORACLE

The AUTOTRACE Syntax

You can enable AUTOTRACE in various ways using the syntax shown in the slide. The command options are as follows:

- OFF: Disables autotracing SQL statements
- ON: Enables autotracing SQL statements
- TRACE or TRACE [ONLY] : Enables autotracing SQL statements and suppresses statement output
- EXPLAIN: Displays execution plans, but does not display statistics
- STATISTICS: Displays statistics, but does not display execution plans

Note: If both the EXPLAIN and STATISTICS command options are omitted, execution plans and statistics are displayed by default.

AUTOTRACE: Examples

- To start tracing statements using AUTOTRACE:

```
SQL> set autotrace on
```

- To display the execution plan only without execution:

```
SQL> set autotrace traceonly explain
```

- To display rows and statistics:

```
SQL> set autotrace on statistics
```

- To get the plan and the statistics only (suppress rows):

```
SQL> set autotrace traceonly
```

ORACLE

AUTOTRACE: Examples

You can control the report by setting the AUTOTRACE system variable. The following are some examples:

- SET AUTOTRACE ON: The AUTOTRACE report includes both the optimizer execution plan and the SQL statement execution statistics.
- SET AUTOTRACE TRACEONLY EXPLAIN: The AUTOTRACE report shows only the optimizer execution path without executing the statement.
- SET AUTOTRACE ON STATISTICS: The AUTOTRACE report shows the SQL statement execution statistics and rows.
- SET AUTOTRACE TRACEONLY: This is similar to SET AUTOTRACE ON, but it suppresses the printing of the user’s query output, if any. If STATISTICS is enabled, the query data is still fetched, but not printed.
- SET AUTOTRACE OFF: No AUTOTRACE report is generated. This is the default.

AUTOTRACE: Statistics

```
SQL> show autotrace
autotrace OFF
SQL> set autotrace traceonly statistics
SQL> SELECT * FROM oe.products;

288 rows selected.

Statistics
--------------------------------------------------------
       1334  recursive calls
          0  db block gets
        686  consistent gets
        394  physical reads
          0  redo size
     103919  bytes sent via SQL*Net to client
        629  bytes received via SQL*Net from client
         21  SQL*Net roundtrips to/from client
         22  sorts (memory)
          0  sorts (disk)
        288  rows processed
```

ORACLE

AUTOTRACE: Statistics

The statistics are recorded by the server when your statement executes and indicate the system resources required to execute your statement. The results include the following statistics:

- `recursive calls` is the number of recursive calls generated at both the user and system level. Oracle Database maintains tables used for internal processing. When Oracle Database needs to make a change to these tables, it internally generates an internal SQL statement, which in turn generates a recursive call.
- `db block gets` is the number of times a `CURRENT` block was requested.
- `consistent gets` is the number of times a consistent read was requested for a block.
- `physical reads` is the total number of data blocks read from disk. This number equals the value of “physical reads direct” plus all reads into buffer cache.
- `redo size` is the total amount of redo generated in bytes.
- `bytes sent via SQL*Net to client` is the total number of bytes sent to the client from the foreground processes.
- `bytes received via SQL*Net from client` is the total number of bytes received from the client over Oracle Net.

- `SQL*Net roundtrips to/from client` is the total number of Oracle Net messages sent to and received from the client.

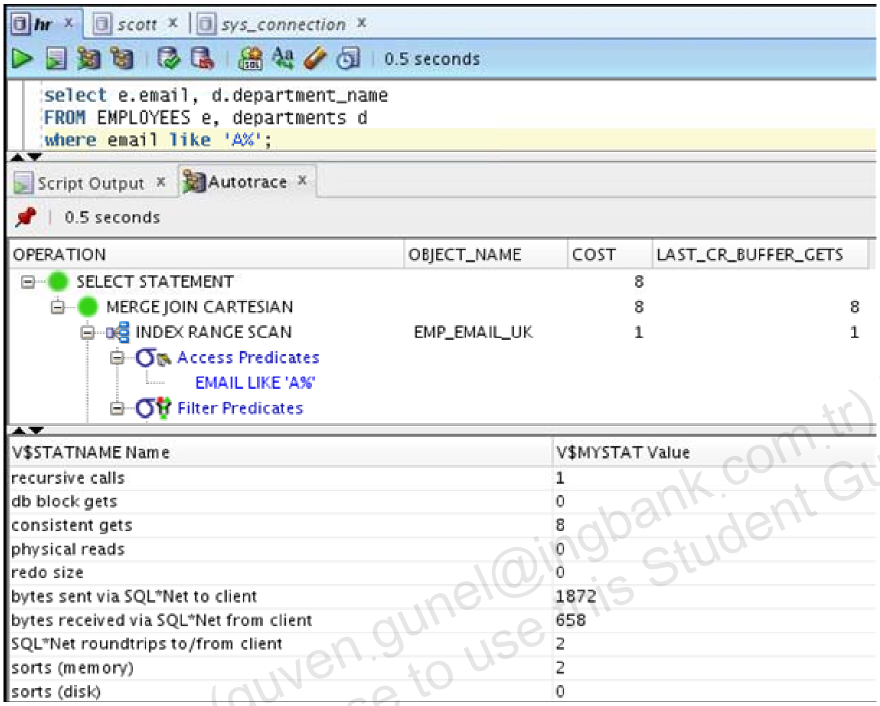
Note: The statistics printed by `AUTOTRACE` are retrieved from `V$SESSTAT`.

- `sorts (memory)` is the number of sort operations that were performed completely in memory and did not require any disk writes.
- `sorts (disk)` is the number of sort operations that required at least one disk write.
- `rows processed` is the number of rows processed during the operation.

The client referred to in the statistics is SQL*Plus. Oracle Net refers to the generic process communication between SQL*Plus and the server, regardless of whether Oracle Net is installed. You cannot change the default format of the statistics report.

Note: `db block gets` indicates reads of the current block from the database. `consistent gets` are reads of blocks that must satisfy a particular system change number (SCN). `physical reads` indicates reads of blocks from disk. `db block gets` and `consistent gets` are the two statistics that are usually monitored. These should be low compared to the number of rows retrieved. Sorts should be performed in memory rather than on disk.

`AUTOTRACE` Using SQL Developer



`AUTOTRACE` Using SQL Developer

The Autotrace pane displays trace-related information when you execute the SQL statement by clicking the Autotrace icon. This information can help you to identify SQL statements that will benefit from tuning.

Using the V$SQL_PLAN View

- V$SQL_PLAN provides a way of examining the execution plan for cursors that are still in the library cache.
- V$SQL_PLAN is very similar to PLAN_TABLE:
  - PLAN_TABLE shows a theoretical plan that can be used if this statement were to be executed.
  - V$SQL_PLAN contains the actual plan used.
- It contains the execution plan of every cursor in the library cache (including child).
- Link to V$SQL:
  - ADDRESS, HASH_VALUE, and CHILD_NUMBER

ORACLE

Using the V$SQL_PLAN View

This view displays the execution plan for cursors that are still in the library cache. The information in this view is very similar to the information in PLAN_TABLE. However, V$SQL_PLAN contains the actual plan used. The execution plan obtained by the EXPLAIN PLAN statement can be different from the execution plan used to execute the cursor. This is because the cursor might have been compiled with different values of session parameters or bind variables..

V$SQL_PLAN shows the plan for a cursor rather than for all cursors associated with a SQL statement. The difference is that a SQL statement can have more than one cursor associated with it, with each cursor further identified by a CHILD_NUMBER. For example, the same statement executed by different users has different cursors associated with it if the object that is referenced is in a different schema. Similarly, different hints can cause different cursors. The V$SQL_PLAN table can be used to see the different plans for different child cursors of the same statement.

Note: Another useful view is V$SQL_PLAN_STATISTICS, which provides the execution statistics of each operation in the execution plan for each cached cursor. Also, the V$SQL_PLAN_STATISTICS_ALL view concatenates information from V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.

The `V$SQL_PLAN` Columns

HASH_VALUE	Hash value of the parent statement in the library cache
ADDRESS	Address of the handle to the parent for this cursor
CHILD_NUMBER	Child cursor number using this execution plan
POSITION	Order of processing for all operations that have the same `PARENT_ID`
PARENT_ID	ID of the next execution step that operates on the output of the current step
ID	Number assigned to each step in the execution plan
PLAN_HASH_VALUE	Numerical representation of the SQL plan for the cursor

Note: This is only a partial listing of the columns.

ORACLE

The `V$SQL_PLAN` Columns

The view contains many of the `PLAN_TABLE` columns, plus several others. The columns that are also present in `PLAN_TABLE` have the same values:

- `ADDRESS`
- `HASH_VALUE`

The `ADDRESS` and `HASH_VALUE` columns can be used to join with `V$SQLAREA` to add the cursor-specific information.

The `ADDRESS`, `HASH_VALUE`, and `CHILD_NUMBER` columns can be used to join with `V$SQL` to add the child cursor–specific information.

The `PLAN_HASH VALUE` column is a numerical representation of the SQL plan for the cursor. By comparing one `PLAN_HASH_VALUE` with another, you can easily identify whether the two plans are the same or not (rather than comparing the two plans line-by-line).

Note: Since Oracle Database 10*g*, `SQL_HASH_VALUE` in `V$SESSION` has been complemented with `SQL_ID`, which you retrieve in many other `V$` views. `SQL_HASH_VALUE` is a 32-bit value and is not unique enough for large repositories of AWR data. `SQL_ID` is a 64-bit hash value, which is more unique, the bottom 32 bits of which are `SQL_HASH_VALUE`. It is normally represented as a character string to make it more manageable.

The V$SQL_PLAN_STATISTICS View

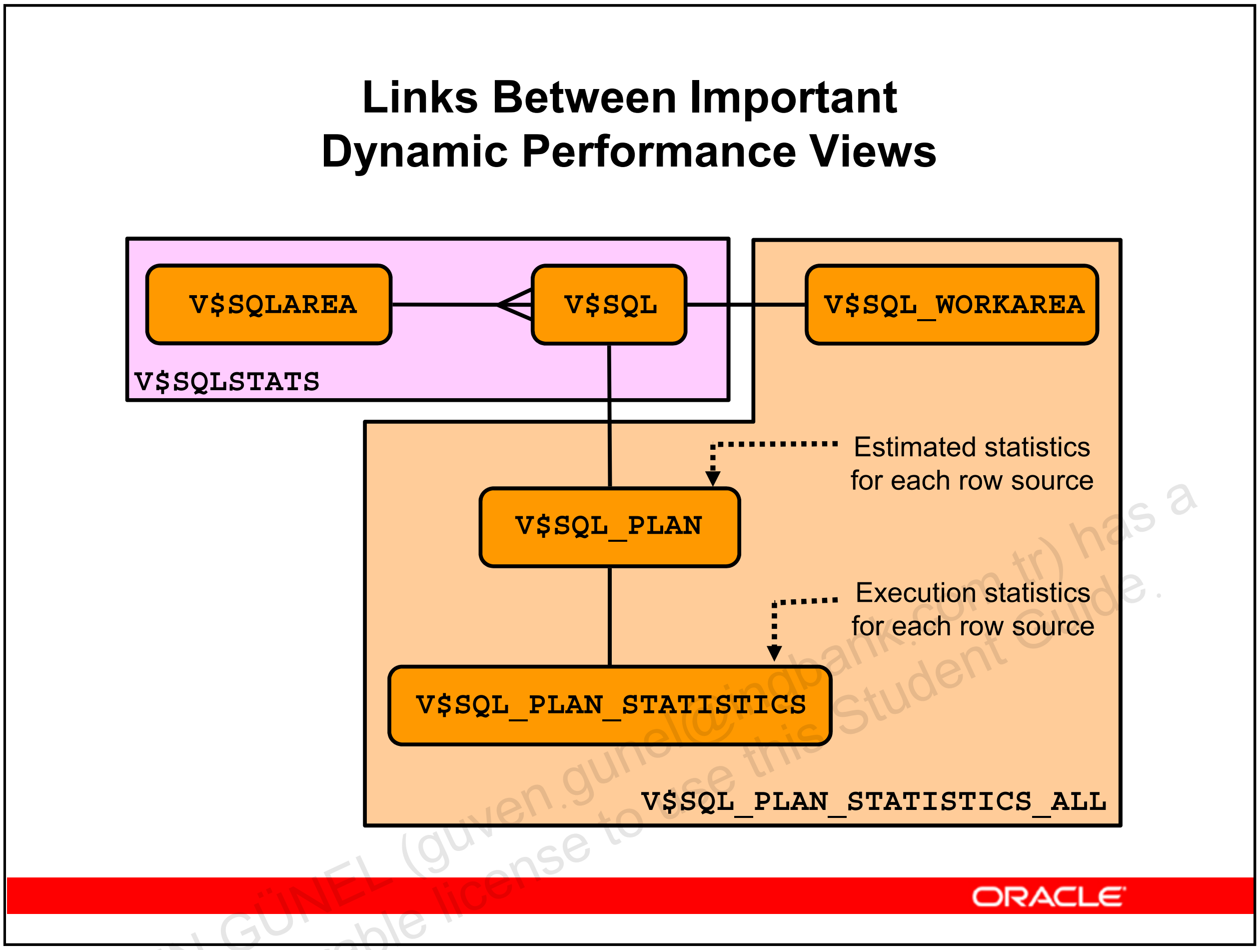
- V$SQL_PLAN_STATISTICS provides actual execution statistics:
  - STATISTICS_LEVEL set to ALL
  - The GATHER_PLAN_STATISTICS hint
- V$SQL_PLAN_STATISTICS_ALL enables side-by-side comparisons of the optimizer estimates with the actual execution statistics.

The V$SQL_PLAN_STATISTICS View

The V$SQL_PLAN_STATISTICS view provides the actual execution statistics for every operation in the plan, such as the number of output rows, and elapsed time. All statistics, except the number of output rows, are cumulative. For example, the statistics for a join operation also include the statistics for its two inputs. The statistics in V$SQL_PLAN_STATISTICS are available for cursors that have been compiled with the STATISTICS_LEVEL initialization parameter set to ALL or using the GATHER_PLAN_STATISTICS hint.

The V$SQL_PLAN_STATISTICS_ALL view contains memory-usage statistics for row sources that use SQL memory (sort or hash join). This view concatenates information in V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.

Links Between Important Dynamic Performance Views



Links Between Important Dynamic Performance Views

`V$SQLAREA` displays statistics on shared SQL areas and contains one row per SQL string. It provides statistics on SQL statements that are in memory, parsed, and ready for execution:

- `SQL_ID` is the SQL identifier of the parent cursor in the library cache.
- `VERSION_COUNT` is the number of child cursors that are present in the cache under this parent.

`V$SQL` lists statistics on shared SQL areas and contains one row for each child of the original SQL text entered:

- `ADDRESS` represents the address of the handle to the parent for this cursor.
- `HASH_VALUE` is the value of the parent statement in the library cache.
- `SQL_ID` is the SQL identifier of the parent cursor in the library cache.
- `PLAN_HASH_VALUE` is a numeric representation of the SQL plan for this cursor. By comparing one `PLAN_HASH_VALUE` with another, you can easily identify if the two plans are the same or not (rather than comparing the two plans line-by-line).
- `CHILD_NUMBER` is the number of this child cursor.

Statistics displayed in V$SQL are normally updated at the end of query execution. However, for long-running queries, they are updated every five seconds. This makes it easy to see the impact of long-running SQL statements while they are still in progress.

V$SQL_PLAN contains the execution plan information for each child cursor loaded in the library cache. The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to add the child cursor–specific information.

V$SQL_PLAN_STATISTICS provides execution statistics at the row source level for each child cursor. The ADDRESS and HASH_VALUE columns can be used to join with V$SQLAREA to locate the parent cursor. The ADDRESS, HASH_VALUE, and CHILD_NUMBER columns can be used to join with V$SQL to locate the child cursor using this area.

V$SQL_PLAN_STATISTICS_ALL contains memory usage statistics for row sources that use SQL memory (sort or hash join). This view concatenates information in V$SQL_PLAN with execution statistics from V$SQL_PLAN_STATISTICS and V$SQL_WORKAREA.

V$SQL_WORKAREA displays information about work areas used by SQL cursors. Each SQL statement stored in the shared pool has one or more child cursors that are listed in the V$SQL view. V$SQL_WORKAREA lists all work areas needed by these child cursors. V$SQL_WORKAREA can be joined with V$SQLAREA on (ADDRESS, HASH_VALUE) and with V$SQL on (ADDRESS, HASH_VALUE, CHILD_NUMBER).

You can use this view to find answers to the following questions:

- What are the top 10 work areas that require the most cache area?
- For work areas allocated in the AUTO mode, what percentage of work areas run using maximum memory?

V$SQLSTATS displays basic performance statistics for SQL cursors, with each row representing the data for a unique combination of SQL text and optimizer plan (that is, unique combination of SQL_ID and PLAN_HASH_VALUE). The column definitions for columns in V$SQLSTATS are identical to those in the V$SQL and V$SQLAREA views. However, the V$SQLSTATS view differs from V$SQL and V$SQLAREA in that it is faster, more scalable, and has a greater data retention (the statistics may still appear in this view, even after the cursor has been aged out of the shared pool). Note that V$SQLSTATS contains a subset of columns that appear in V$SQL and V$SQLAREA.

Querying V$SQL_PLAN

Querying V$SQL_PLAN

```
SELECT PLAN_TABLE_OUTPUT FROM
TABLE(DBMS_XPLAN.DISPLAY_CURSOR('47ju6102uvq5q'));
```

```
SQL_ID  47ju6102uvq5q, child number 0
-------------------------------------
SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d WHERE
e.department_id =d.department_id

Plan hash value: 2933537672

--------------------------------------------------------------------------------
| Id  | Operation                    | Name        | Rows  | Bytes | Cost (%CPU|
--------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |       |       |     6 (100|
|   1 |  MERGE JOIN                  |             |   106 |  2862 |     6  (17|
|   2 |   TABLE ACCESS BY INDEX ROWID| DEPARTMENTS |    27 |   432 |     2   (0|
|   3 |    INDEX FULL SCAN           | DEPT_ID_PK  |    27 |       |     1   (0|
|*  4 |   SORT JOIN                  |             |   107 |  1177 |     4  (25|
|   5 |    TABLE ACCESS FULL         | EMPLOYEES   |   107 |  1177 |     3   (0|
--------------------------------------------------------------------------------
Predicate Information (identified by operation id):
---------------------------------------------------


   4 - access("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")
       filter("E"."DEPARTMENT_ID"="D"."DEPARTMENT_ID")

24 rows selected.
```


Querying V$SQL_PLAN

You can query `V$SQL_PLAN` using the `DBMS_XPLAN.DISPLAY_CURSOR()` function to display the current or last executed statement (as shown in the example). You can pass the value of `SQL_ID` for the statement as a parameter to obtain the execution plan for a given statement. `SQL_ID` is the `SQL_ID` of the SQL statement in the cursor cache. You can retrieve the appropriate value by querying the `SQL_ID` column in `V$SQL` or `V$SQLAREA`. Alternatively, you could select the `PREV_SQL_ID` column for a specific session out of `V$SESSION`. This parameter defaults to null in which case the plan of the last cursor executed by the session is displayed. To obtain `SQL_ID`, execute the following query:

```
SELECT e.last_name, d.department_name
FROM hr.employees e, hr.departments d
WHERE  e.department_id =d.department_id;

SELECT SQL_ID, SQL_TEXT FROM V$SQL
WHERE SQL_TEXT LIKE '%SELECT e.last_name,%' ;

13saxr0mmz1s3  select SQL_id, sql_text from v$SQL …
47ju6102uvq5q  SELECT e.last_name, d.department_name …
```

CHILD_NUMBER is the child number of the cursor to display. If not supplied, the execution plan of all cursors matching the supplied SQL_ID parameter are displayed. CHILD_NUMBER can be specified only if SQL_ID is specified.

The FORMAT parameter controls the level of detail for the plan. In addition to the standard values (BASIC, TYPICAL, SERIAL, ALL, and ADVANCED), there are additional supported values to display run-time statistics for the cursor:

- IOSTATS: Assuming that the basic plan statistics are collected when SQL statements are executed (either by using the GATHER_PLAN_STATISTICS hint or by setting the statistics_level parameter to ALL), this format shows I/O statistics for ALL (or only for LAST) executions of the cursor.
- MEMSTATS: Assuming that the Program Global Area (PGA) memory management is enabled (that is, the pga_aggregate_target parameter is set to a nonzero value), this format allows to display memory management statistics (for example, execution mode of the operator, how much memory was used, number of bytes spilled to disk, and so on). These statistics only apply to memory-intensive operations, such as hash joins, sort or some bitmap operators.
- ALLSTATS: A shortcut for 'IOSTATS MEMSTATS'
- LAST: By default, plan statistics are shown for all executions of the cursor. The LAST keyword can be specified to see only the statistics for the last execution.

Automatic Workload Repository (AWR)

- Collects, processes, and maintains performance statistics for problem-detection and self-tuning purposes
- Statistics include:
  - Object statistics
  - Time-model statistics
  - Some system and session statistics
  - Active Session History (ASH) statistics
- Automatically generates snapshots of the performance data

Automatic Workload Repository (AWR)

The AWR is part of the intelligent infrastructure introduced with Oracle Database 10*g*. This infrastructure is used by many components, such as Automatic Database Diagnostic Monitor (ADDM) for analysis. The AWR automatically collects, processes, and maintains system-performance statistics for problem-detection and self-tuning purposes and stores the statistics persistently in the database.

The statistics collected and processed by the AWR include:

- Object statistics that determine both access and usage statistics of database segments
- Time-model statistics based on time usage for activities, displayed in the `V$SYS_TIME_MODEL` and `V$SESS_TIME_MODEL` views
- Some of the system and session statistics collected in the `V$SYSSTAT` and `V$SESSTAT` views
- SQL statements that produce the highest load on the system, based on criteria, such as elapsed time, CPU time, buffer gets, and so on
- ASH statistics, representing the history of recent sessions

The database automatically generates snapshots of the performance data once every hour and collects the statistics in the workload repository. The data in the snapshot interval is then analyzed by ADDM. The ADDM compares the differences between snapshots to determine which SQL statements to capture based on the effect on the system load. This reduces the number of SQL statements that need to be captured over time.

Note: By using PL/SQL packages, such as `DBMS_WORKLOAD_REPOSITORY` or Oracle Enterprise Manager, you can manage the frequency and retention period of SQL that is stored in the AWR.

Managing AWR with PL/SQL

- Creating snapshots:

```
SQL> exec DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT ('ALL');
```

- Dropping snapshots:

```
SQL> exec DBMS_WORKLOAD_REPOSITORY.DROP_SNAPSHOT_RANGE -
    (low_snap_id => 22, high_snap_id => 32, dbid => 3310949047);
```

- Managing snapshot settings:

```
SQL> exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS -
     (retention => 43200, interval => 30, dbid => 3310949047);
```


Managing AWR with PL/SQL

Although the primary interface for managing the AWR is Enterprise Manager, monitoring functions can be managed with procedures in the `DBMS_WORKLOAD_REPOSITORY` package.

Snapshots are automatically generated for an Oracle Database; however, you can use `DBMS_WORKLOAD_REPOSITORY` procedures to manually create, drop, and modify the snapshots and baselines that are used by the ADDM. Snapshots and baselines are sets of historical data for specific time periods that are used for performance comparisons. To invoke these procedures, a user must be granted the DBA role.

Creating Snapshots

You can manually create snapshots with the `CREATE_SNAPSHOT` procedure if you want to capture statistics at times different than those of the automatically generated snapshots. Here is an example:

```
Exec DBMS_WORKLOAD_REPOSITORY.CREATE_SNAPSHOT ('ALL');
```

In this example, a snapshot for the instance is created immediately with the flush level specified to the default flush level of `TYPICAL`. You can view this snapshot in the `DBA_HIST_SNAPSHOT` view.

Dropping Snapshots

You can drop a range of snapshots using the DROP_SNAPSHOT_RANGE procedure. To view a list of the snapshot IDs along with database IDs, check the DBA_HIST_SNAPSHOT view. For example, you can drop the following range of snapshots:

```
Exec DBMS_WORKLOAD_REPOSITORY.DROP_SNAPSHOT_RANGE - (low_snap_id =>
22, high_snap_id => 32, dbid => 3310949047);
```

In the example, the range of snapshot IDs to drop is specified from 22 to 32. The optional database identifier is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value.

ASH data that belongs to the time period specified by the snapshot range is also purged when the DROP_SNAPSHOT_RANGE procedure is called.

Modifying Snapshot Settings

You can adjust the interval and retention of snapshot generation for a specified database ID. However, note that this can affect the precision of the Oracle diagnostic tools.

The INTERVAL setting specifies how often (in minutes) snapshots are automatically generated. The RETENTION setting specifies how long (in minutes) snapshots are stored in the workload repository. To adjust the settings, use the MODIFY_SNAPSHOT_SETTINGS procedure, as in the following example:

```
Exec DBMS_WORKLOAD_REPOSITORY.MODIFY_SNAPSHOT_SETTINGS( -retention
   => 43200, interval => 30, dbid => 3310949047);
```

In this example, the retention period is specified as 43,200 minutes (30 days), and the interval between each snapshot is specified as 30 minutes. If NULL is specified, the existing value is preserved. The optional database identifier is 3310949047. If you do not specify a value for dbid, the local database identifier is used as the default value. You can check the current settings for your database instance with the DBA_HIST_WR_CONTROL view.

Important AWR Views

- V$ACTIVE_SESSION_HISTORY
- V$ metric views
- DBA_HIST views:
  - DBA_HIST_ACTIVE_SESS_HISTORY
  - DBA_HIST_BASELINE DBA_HIST_DATABASE_INSTANCE
  - DBA_HIST_SNAPSHOT
  - DBA_HIST_SQL_PLAN
  - DBA_HIST_WR_CONTROL

Important AWR Views

You can view the AWR data on Oracle Enterprise Manager screens or in AWR reports. However, you can also view the statistics directly from the following views:

V$ACTIVE_SESSION_HISTORY: This view displays active database session activity, sampled once every second.

V$ metric views provide metric data to track the performance of the system. The metric views are organized into various groups, such as event, event class, system, session, service, file, and tablespace metrics. These groups are identified in the V$METRICGROUP view.

The DBA_HIST views contain historical data stored in the database. This group of views includes:

- DBA_HIST_ACTIVE_SESS_HISTORY displays the history of the contents of the sampled in-memory active session history for recent system activity.
- DBA_HIST_BASELINE displays information about the baselines captured in the system.
- DBA_HIST_DATABASE_INSTANCE displays information about the database environment.
- DBA_HIST_SNAPSHOT displays information about snapshots in the system.
- DBA_HIST_SQL_PLAN displays SQL execution plans.

- DBA_HIST_WR_CONTROL displays the settings for controlling AWR.

Querying the AWR

- Retrieve all execution plans stored for a particular `SQL_ID`.

```
SQL> SELECT PLAN_TABLE_OUTPUT FROM TABLE (DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w'));

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------------
SQL_ID 454rug2yva18w
--------------------
select /* example */ * from hr.employees natural join hr.departments

Plan hash value: 4179021502

-----------------------------------------------------------------------------------
| Id  | Operation           | Name        | Rows  | Bytes | Cost (%CPU)| Time     |
-----------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |             |       |       |     6 (100)|          |
|   1 |  HASH JOIN          |             |    11 |   968 |     6  (17)| 00:00:01 |
|   2 |   TABLE ACCESS FULL| DEPARTMENTS |    11 |   220 |     2   (0)| 00:00:01 |
|   3 |   TABLE ACCESS FULL| EMPLOYEES   |   107 |  7276 |     3   (0)| 00:00:01 |
-----------------------------------------------------------------------------------
```

- Display all execution plans of all statements containing "JF."

```
SELECT tf.* FROM DBA_HIST_SQLTEXT ht, table
    (DBMS_XPLAN.DISPLAY_AWR(ht.sql_id,null, null,  'ALL' )) tf
 WHERE ht.sql_text like '%JF%';
```

ORACLE

Querying the AWR

You can use the `DBMS_XPLAN.DISPLAY_AWR()` function to display all stored plans in the AWR. In the example in the slide, you pass in a `SQL_ID` as an argument. `SQL_ID` is the `SQL_ID` of the SQL statement in the cursor cache. The `DISPLAY_AWR()` function also takes the `PLAN_HASH_VALUE`, `DB_ID`, and `FORMAT` parameters.

The steps to complete this example are as follows:

1. Execute the SQL statement:

```
SQL> select /* example */ * from hr.employees natural
     join hr.departments;
```

2. Query `V$SQL_TEXT` to obtain the `SQL_ID`:

```
SQL> select sql_id, sql_text from v$SQL
       where sql_text
       like '%example%';
SQL_ID        SQL_TEXT
------------- -----------------------------------------
F8tc4anpz5cdb select sql_id, sql_text from v$SQL …
454rug2yva18w select /* example */ * from …
```

3. Using the `SQL_ID`, verify that this statement has been captured in the `DBA_HIST_SQLTEXT` dictionary view. If the query does not return rows, it indicates that the statement has not yet been loaded in the AWR.

```
SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext WHERE SQL_ID
=' 454rug2yva18w';

no rows selected
```

You can take a manual AWR snapshot rather than wait for the next snapshot (which occurs every hour). Then check to see if it has been captured in `DBA_HIST_SQLTEXT`:

```
SQL> exec dbms_workload_repository.create_snapshot;


PL/SQL procedure successfully completed.


SQL> SELECT SQL_ID, SQL_TEXT FROM dba_hist_sqltext WHERE SQL_ID
   =' 454rug2yva18w';

SQL_ID          SQL_TEXT
-------------   ------------------------------

454rug2yva18w   select /* example */ * from …
```

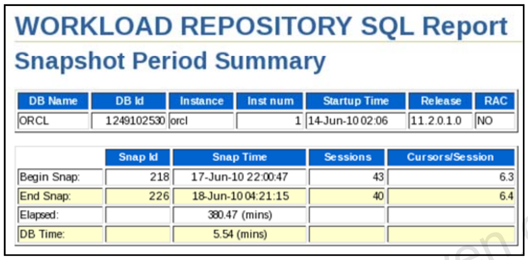
4. Use the `DBMS_XPLAN.DISPLAY_AWR ()` function to retrieve the execution plan:

```
SQL>SELECT PLAN_TABLE_OUTPUT FROM TABLE
   (DBMS_XPLAN.DISPLAY_AWR('454rug2yva18w’));
```

Generating SQL Reports from AWR Data

```
SQL> @$ORACLE_HOME/rdbms/admin/awrsqrpt

Specify the Report Type …
Would you like an HTML report, or a plain text report?
Specify the number of days of snapshots to choose from
Specify the Begin and End Snapshot Ids …
Specify the SQL Id …
Enter value for sql_id: dvza55c7zu0yv
Specify the Report Name …
```

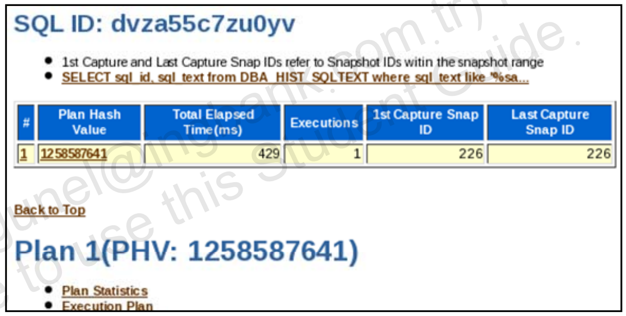


WORKLOAD REPOSITORY SQL Report

Snapshot Period Summary

DB Name	DB Id	Instance	Inst num	Startup Time	Release	RAC
ORCL	1249102530	orcl	1	14-Jun-10 02:06	11.2.0.1.0	NO

	Snap Id	Snap Time	Sessions	Cursors/Session
Begin Snap:	218	17-Jun-10 22:00:47	43	6.3
End Snap:	226	18-Jun-10 04:21:15	40	6.4
Elapsed:		380.47 (mins)		
DB Time:		5.54 (mins)		



SQL ID: dvza55c7zu0yv

- 1st Capture and Last Capture Snap IDs refer to Snapshot IDs witin the snapshot range
- SELECT sql_id, sql_text from DBA_HIST_SQLTEXT where sql_text like '%sa...

#	Plan Hash Value	Total Elapsed Time(ms)	Executions	1st Capture Snap ID	Last Capture Snap ID
1	1258587641	429	1	226	226

Back to Top

Plan 1(PHV: 1258587641)

- Plan Statistics
- Execution Plan

ORACLE

Generating SQL Reports from AWR Data

Since Oracle Database 10*g*, Release 2, it is possible to generate SQL reports from AWR data, basically, the equivalent to `sqrepsql.sql` with Statspack. In 10.1.0.4.0, the equivalent to `sprepsql.sql` is *not* available in AWR. However, in 10*g*R2, the equivalent of `sprepsql.sql` is available. In 10*g*R2, the AWR SQL report can be generated by calling the `$ORACLE_HOME/rdbms/admin/awrsqrpt.sql` file.

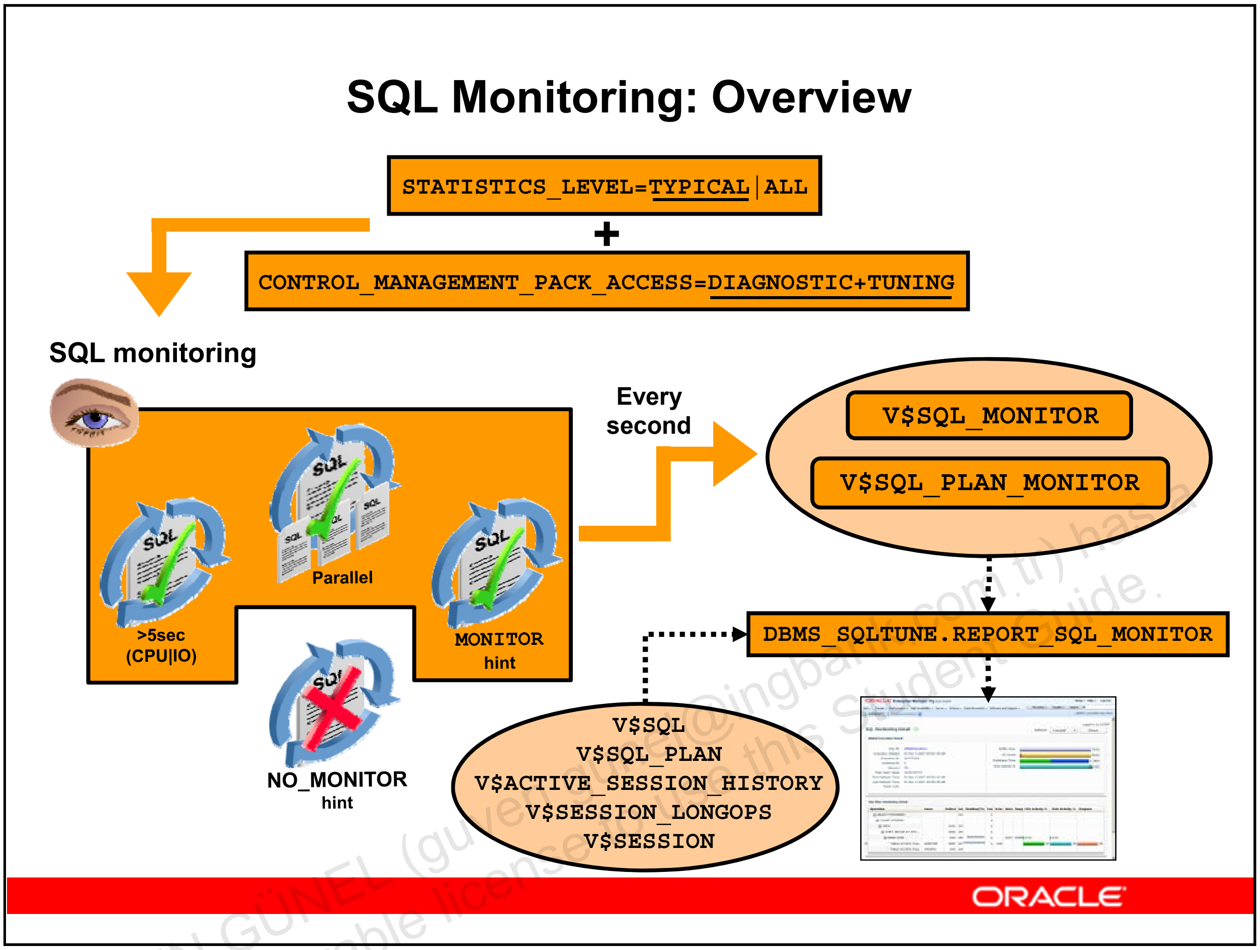
You can display the plan information in AWR by using the `display_awr` table function in the `dbms_xplan` PL/SQL package.

For example, this displays the plan information for a `SQL_ID` in AWR:

```
select * from table(dbms_xplan.display_awr('dvza55c7zu0yv'));
```

You can retrieve the appropriate value for the SQL statement of interest by querying `SQL_ID` in the `DBA_HIST_SQLTEXT` column.

SQL Monitoring: Overview



SQL Monitoring: Overview

The SQL monitoring feature is enabled by default when the STATISTICS_LEVEL initialization parameter is either set to ALL or TYPICAL (the default value).

Additionally, the CONTROL_MANAGEMENT_PACK_ACCESS parameter must be set to DIAGNOSTIC+TUNING (the default value) because SQL monitoring is a feature of the Oracle Database Tuning Pack.

By default, SQL monitoring is automatically started when a SQL statement runs parallel, or when it has consumed at least five seconds of the CPU or I/O time in a single execution.

As mentioned, SQL monitoring is active by default. However, two statement-level hints are available to force or prevent a SQL statement from being monitored. To force SQL monitoring, use the MONITOR hint. To prevent the hinted SQL statement from being monitored, use the NO_MONITOR hint.

You can monitor the statistics for SQL statement execution using the V$SQL_MONITOR and V$SQL_PLAN_MONITOR views.

After monitoring is initiated, an entry is added to the dynamic performance V$SQL_MONITOR view. This entry tracks key performance metrics collected for the execution, including the elapsed time, CPU time, number of reads and writes, I/O wait time, and various other wait times. These statistics are refreshed in near real time as the statement executes, generally once every second.

After the execution ends, monitoring information is not deleted immediately, but is kept in the `V$SQL_MONITOR` view for at least one minute. The entry is eventually deleted so its space can be reclaimed as new statements are monitored.

The `V$SQL_MONITOR` and `V$SQL_PLAN_MONITOR` views can be used in conjunction with the following views to get additional information about the execution that is monitored:

`V$SQL`, `V$SQL_PLAN`, `V$ACTIVE_SESSION_HISTORY`, `V$SESSION_LONGOPS`, and `V$SESSION`

Instead, you can use the SQL monitoring report to view SQL monitoring data.

The SQL monitoring report is also available in a GUI version through Enterprise Manager and SQL Developer

SQL Monitoring Report: Example

```
SQL> set long 10000000
SQL> set longchunksize 10000000
SQL> set linesize 200
SQL> select dbms_sqltune.report_sql_monitor from dual;

SQL Monitoring Report

SQL Text
--------------------------
select count(*) from sales

Global Information
 Status               :  EXECUTING
 Instance ID          :  1
 Session ID           :  125
 SQL ID               :  fazrk33ng71km
 SQL Execution ID     :  16777216
 Plan Hash Value      :  1047182207
 Execution Started    :  02/19/2008 21:01:18
 First Refresh Time   :  02/19/2008 21:01:22
 Last Refresh Time    :  02/19/2008 21:01:42
------------------------------------------------------------
| Elapsed |   Cpu   |    IO    |  Other   | Buffer | Reads |
| Time(s) | Time(s) | Waits(s) | Waits(s) |  Gets  |       |
------------------------------------------------------------
|      22 |    3.36 |     0.01 |       19 |   259K |  199K |
------------------------------------------------------------
```

In a different session

```
SQL> select count(*) from sales;
```

ORACLE

SQL Monitoring Report: Example

In this example, it is assumed that you SELECT from SALES from a different session than the one used to print the SQL monitoring report.

The DBMS_SQLTUNE.REPORT_SQL_MONITOR function accepts several input parameters to specify the execution, the level of detail in the report, and the report type (TEXT, HTML, or XML). By default, a text report is generated for the last execution that was monitored if no parameters are specified as shown in the example in the slide.

After the SELECT statement is started, and while it executes, you print the SQL monitoring report from a second session.

From the report, you can see that the SELECT statement executes currently.

The Global Information section gives you some important information:

- To uniquely identify two executions of the same SQL statement, a composite key called an execution key is generated. This execution key consists of three attributes, each corresponding to a column in V$SQL_MONITOR:
  - SQL identifier to identify the SQL statement (SQL_ID)
  - An internally generated identifier to ensure that this primary key is truly unique (SQL_EXEC_ID)

- A start execution time stamp (SQL_EXEC_START)

The report also shows you some important statistics calculated so far.

SQL Monitoring Report: Example

```
SQL Plan Monitoring Details
====================================================================================
| Id    |      Operation         | Name   |  Rows   | Cost   |   Time    | Start  |
|       |                        |        | (Estim) |        | Active(s) | Active |
====================================================================================
|     0 | SELECT STATEMENT       |        |         | 78139  |           |        |
|     1 |   SORT AGGREGATE       |       |       1 |        |           |        |
|  -> 2 |    TABLE ACCESS FULL   | SALES  | 53984K  | 78139  |        23 |     +1 |
|       |                        |        |         |        |           |        |
====================================================================================


==================================================================================
Starts |   Rows   | Activity |          Activity Detail           | Progress    |
       | (Actual) | (percent)|             (sample #)             |             |
     1 |          |          |                                    |             |
     1 |          |          |                                    |             |
     1 |  42081K  |   100.00 | Cpu (4)                            | 74%         |
==================================================================================
```

ORACLE

SQL Monitoring Report: Example (continued)

The report then displays the execution path currently used by your statement. SQL monitoring gives you the display of the current operation that executes in the plan. This enables you to detect parts of the plan that are the most time consuming, so that you can focus your analysis on those parts. The running operation is marked by an arrow in the Id column of the report.

The Time Active(s) column shows how long the operation has been active (the delta in seconds between the first and the last active time).

The Start Active column shows, in seconds, when the operation in the execution plan started relative to the SQL statement execution start time. In this report, the table access full operation at Id 2 was the first to start (+1s Start Active) and ran for the first 23 seconds so far.

The Starts column shows the number of times the operation in the execution plan was executed.

The Rows (Actual) column indicates the number of rows produced, and the Rows (Estim) column shows the estimated cardinality from the optimizer.

The Activity (percent) and Activity Detail (sample #) columns are derived by joining the `V$SQL_PLAN_MONITOR` and `V$ACTIVE_SESSION_HISTORY` views. Activity (percent) shows the percentage of database time consumed by each operation of the execution plan. Activity Detail (sample#) shows the nature of that activity (such as CPU or wait event).

In this report, the Activity Detail (sample #) column shows that most of the database time, 100%, is consumed by operation Id 2 (`TABLE ACCESS FULL of SALES`). So far, this activity consists of 4 samples, which are only attributed to CPU.

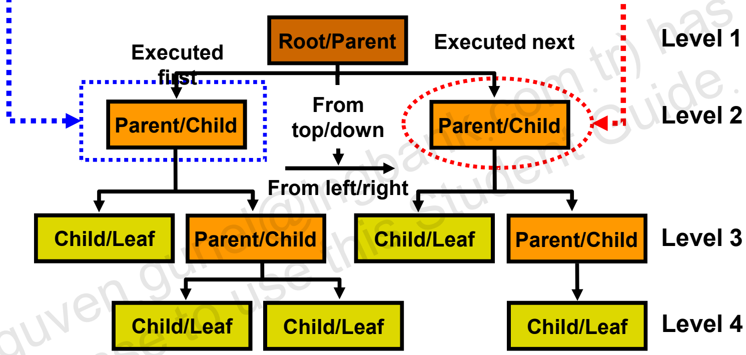
The last column, Progress, shows progress monitoring information for the operation from the `V$SESSION_LONGOPS` view. In this report, it shows that, so far, the `TABLE ACCESS FULL` operation is 74% complete. This column only appears in the report after a certain amount of time, and only for the instrumented row sources.

Note: Not shown by this particular report, the Memory and Temp columns indicate the amount of memory and temporary space consumed by corresponding operation of the execution plan.

Interpreting an Execution Plan

```
id= 1  (pid= )           root/parent
id= 2  (pid=1) (pos=1)    parent/child
id= 3  (pid=2) (pos=1)      child/leaf
id= 4  (pid=2) (pos=2)    parent/child
id= 5  (pid=4) (pos=1)      child/leaf
id= 6  (pid=4) (pos=2)      child/leaf
id= 7  (pid=1) (pos=2)    parent/child
id= 8  (pid=7) (pos=1)      child/leaf
id= 9  (pid=7) (pos=2)    parent/child
id=10  (pid=9) (pos=1)      child/leaf
```

Transform it into a tree.



Interpreting an Execution Plan

Explain plan output is a representation of a tree of row sources.

Each step (line in the execution plan or node in the tree) represents a row source.

The explain plan utility indents nodes to indicate that they are the children of the parent above it.

The order of the nodes under the parent indicates the order of execution of the nodes within that level. If two steps are indented at the same level, the first one is executed first.

In the tree format, the leaf at the left on each level of the tree is where the execution starts.

The steps of the execution plan are not performed in the order in which they are numbered. there is a parent–child relationship between steps.

In `PLAN_TABLE` and `V$SQL_PLAN`, the important elements to retrieve the tree structure are the `ID`, `PARENT_ID`, and `POSITION` columns. In a trace file, these columns correspond to the `id`, `pid`, and `pos` fields, respectively.

One way to read an execution plan is by converting it into a graph that has a tree structure. You can start from the top, with `id=1`, which is the root node in the tree. Next, you must find the operations that feed this root node. That is accomplished by operations, which have `parent_id` or `pid` with value 1.

Note: The course focuses on serial plans and does not discusses parallel execution plans.

To draw plan as a tree, do the following:

1. Take the ID with the lowest number and place it at the top.
2. Look for rows which have a PID (parent) equal to this value.
3. Place these in the tree below the Parent according to their POS values from the lowest to the highest, ordered from left to right.
4. After all the IDs for a parent have been found, move down to the next ID and repeat the process, finding new rows with the same PID.

The first thing to determine in an explain plan is which node is executed first. The method in the slide explains this, but sometimes with complicated plans it is difficult to do this and also difficult to follow the steps through to the end. Large plans are exactly the same as smaller ones, but with more entries. The same basic rules apply. You can always collapse the plan to hide a branch of the tree which does not consume much of the resources.

Standard explain plan interpretation:

1. Start at the top.
2. Move down the row sources until you get to one which produces data, but does not consume any. This is the start row source.
3. Look at the siblings of this row source. These row sources are executed next.
4. After the children are executed, the parent is executed next.
5. Now that this parent and its children are completed, work back up the tree, and look at the siblings of the parent row source and its parents. Execute as before.
6. Move back up the plan until all row sources are exhausted.

Standard tree interpretation:

1. Start at the top.
2. Move down the tree to the left until you reach the left node. This is executed first.
3. Look at the siblings of this row source. These row sources are executed next.
4. After the children are executed, the parent is executed next.
5. Now that this parent and its children are completed, work back up the tree, and look at the siblings of the parent row source and its parents. Execute as before.
6. Move back up the tree until all row sources are exhausted.

If you remember the few basic rules of explain plans and with some experience, you can read most plans easily.

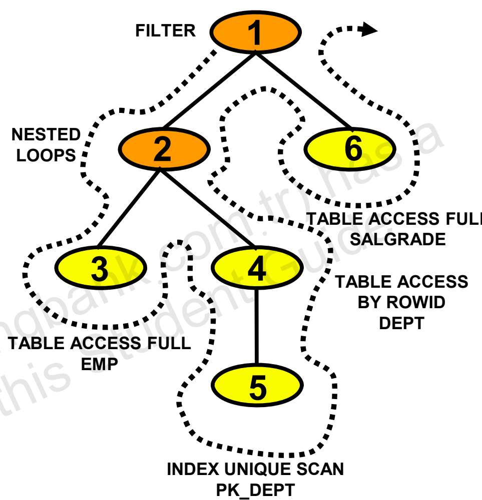
Execution Plan Interpretation: Example 1

```
SELECT /*+ RULE */ ename,job,sal,dname
FROM emp,dept
WHERE dept.deptno=emp.deptno and not exists(SELECT *
                                            FROM salgrade
                                            WHERE emp.sal between losal and hisal);
```

```
--------------------------------------------------
| Id  | Operation                     | Name     |
--------------------------------------------------
|   0 | SELECT STATEMENT              |          |
|*  1 |  FILTER                       |          |
|   2 |   NESTED LOOPS                |          |
|   3 |    TABLE ACCESS FULL          | EMP      |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT     |
|*  5 |     INDEX UNIQUE SCAN         | PK_DEPT  |
|*  6 |   TABLE ACCESS FULL           | SALGRADE |
--------------------------------------------------

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

   1 - filter( NOT EXISTS
     (SELECT 0 FROM "SALGRADE" "SALGRADE" WHERE
      "HISAL">=:B1 AND "LOSAL"<=:B2))
   5 - access("DEPT"."DEPTNO"="EMP"."DEPTNO")
   6 - filter("HISAL">=:B1 AND "LOSAL"<=:B2)
```



Execution Plan Interpretation: Example 1

You start with an example query to illustrate how to interpret an execution plan. The slide shows a query with its associated execution plan and the same plan in the tree format.

The query tries to find employees who have salaries outside the range of salaries in the salary grade table. The query is a `SELECT` statement from two tables with a subquery based on another table to check the salary grades.

See the execution order for this query. Based on the example in the slide, and from the previous slide, the execution order is 3 – 5 – 4 – 2 – 6 – 1:

- **3:** The plan starts with a full table scan of `EMP` (`ID=3`).
- **5:** The rows are passed back to the controlling nested loops join step (`ID=2`), which uses them to execute the lookup of rows in the `PK_DEPT` index in `ID=5`.
- **4:** The `ROWID`s from the index are used to lookup the other information from the `DEPT` table in `ID=4`.
- **2:** `ID=2`, the nested loops join step, is executed until completion.
- **6:** After `ID=2` has exhausted its row sources, a full table scan of `SALGRADE` in `ID=6` (at the same level in the tree as `ID=2`, therefore, its sibling) is executed.
- **1:** This is used to filter the rows from `ID2` and `ID6`.

Note that children are executed before parents, so although structures for joins must be set up before the child execution, the children are notated as executed first. Probably, the easiest way is to consider it as the order in which execution completes, so for the `NESTED LOOPS` join at `ID=2`, the two children {`ID=3` and `ID=4` (together with its child)} must have completed their execution before `ID=2` can be completed.

Execution Plan Interpretation: Example 1

```
SQL> alter session set statistics_level=ALL;

Session altered.

SQL> select /*+ RULE to make sure it reproduces 100% */ ename,job,sal,dname
from emp,dept where dept.deptno = emp.deptno and not exists (select * from salgrade
where emp.sal between losal and hisal);

no rows selected

SQL> select * from table(dbms_xplan.display_cursor(null,null,'TYPICAL IOSTATS
LAST'));

SQL_ID  274019myw3vuf, child number 0
-------------------------------------
…
Plan hash value: 1175760222
--------------------------------------------------------------------------------
| Id  | Operation                      | Name      | Starts | A-Rows | Buffers |
--------------------------------------------------------------------------------
|*  1 |  FILTER                        |           |      1 |      0 |      61 |
|   2 |   NESTED LOOPS                 |           |      1 |     14 |      25 |
|   3 |    TABLE ACCESS FULL           | EMP       |      1 |     14 |       7 |
|   4 |    TABLE ACCESS BY INDEX ROWID| DEPT      |     14 |     14 |      18 |
|*  5 |     INDEX UNIQUE SCAN          | PK_DEPT   |     14 |     14 |       4 |
|*  6 |   TABLE ACCESS FULL            | SALGRADE  |     12 |     12 |      36 |
--------------------------------------------------------------------------------
…
```

ORACLE

Execution Plan Interpretation: Example 1 (continued)

The example in the slide is a plan dump from V$SQL_PLAN with STATISTICS_LEVEL set to ALL. This report shows you some important additional information compared to the output of the EXPLAIN PLAN command:

- A-Rows corresponds to the number of rows produced by the corresponding row source.
- Buffers corresponds to the number of consistent reads done by the row source.
- Starts indicates how many times the corresponding operation was processed.

For each row from the EMP table, the system gets its ENAME, SAL, JOB, and DEPTNO.

Then the system accesses the DEPT table by its unique index (PK_DEPT) to get DNAME using DEPTNO from the previous result set.

If you observe the statistics closely, the TABLE ACCESS FULL operation on the EMP table (ID=3) is started once. However, operations from ID 5 and 4 are started 14 times; once for each EMP rows. At this step (ID=2), the system gets all ENAME, SAL, JOB, and DNAME.

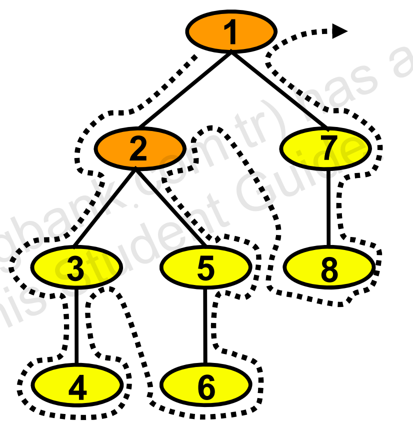
The system now must filter out employees who have salaries outside the range of salaries in the salary grade table. To do that, for each row from ID=2, the system accesses the SALGRADE table using a FULL TABLE SCAN operation to check if the employee’s salary is outside the salary range. This operation only needs to be done 12 times in this case because at run time the system does the check for each distinct salary, and there are 12 distinct salaries in the EMP table.

Execution Plan Interpretation: Example 2

Execution Plan Interpretation: Example 2

```
SQL> select /*+ USE_NL(d) use_nl(m) */ m.last_name as dept_manager
  2  ,      d.department_name
  3  ,      l.street_address
  4  from   hr.employees m   join
  5         hr.departments d on (d.manager_id = m.employee_id)
  6         natural join
  7         hr.locations l
  8  where  l.city = 'Seattle';
```

```
0    SELECT STATEMENT
1 0    NESTED LOOPS
2 1     NESTED LOOPS
3 2      TABLE ACCESS BY INDEX ROWID LOCATIONS
4 3       INDEX RANGE SCAN            LOC_CITY_IX
5 2      TABLE ACCESS BY INDEX ROWID DEPARTMENTS
6 5       INDEX RANGE SCAN            DEPT_LOCATION_IX
7 1     TABLE ACCESS BY INDEX ROWID  EMPLOYEES
8 7      INDEX UNIQUE SCAN            EMP_EMP_ID_PK
```



Execution Plan Interpretation: Example 2

This query retrieves names, department names, and addresses for employees whose departments are located in Seattle and who have managers.

For formatting reasons, the explain plan has the `ID` in the first column, and `PID` in the second column. The position is reflected by the indentation. The execution plan shows two nested loops join operations.

You follow the steps from the previous example:

1. Start at the top. `ID=0`
2. Move down the row sources until you get to the one, which produces data, but does not consume any. In this case, `ID 0`, `1`, `2`, and `3` consume data. `ID=4` is the first row source that does not consume any. This is the start row source. `ID=4` is executed first. The index range scan produces `ROWID`s, which are used to lookup in the `LOCATIONS` table in `ID=3`.
3. Look at the siblings of this row source. These row sources are executed next. The sibling at the same level as `ID=3` is `ID=5`. Node `ID=5` has a child `ID=6`, which is executed before it. This is another index range scan producing `ROWID`s, which are used to lookup in the `DEPARTMENTS` table in `ID=5`.

4. After the children operation, the parent operation is next. The NESTED LOOPS join at ID=2 is executed next bringing together the underlying data.
5. Now that this parent and its children are completed, walk back up the tree, and look at the siblings of the parent row source and its parents. Execute as before. The sibling of ID=2 at the same level in the plan is ID=7. This has a child ID=8, which is executed first. The index unique scan produces ROWIDs, which are used to lookup in the EMPLOYEES table in ID=7.
6. Move back up the plan until all row sources are exhausted. Finally this is brought together with the NESTED LOOPS at ID=1, which passes the results back to ID=0.
7. The execution order is: 4 – 3 – 6 – 5 – 2 – 8 – 7 – 1 – 0

Here is the complete description of this plan:

The inner nested loops is executed first using LOCATIONS as the driving table, using an index access on the CITY column. This is because you search for departments in Seattle only.

The result is joined with the DEPARTMENTS table, using the index on the LOCATION_ID join column; the result of this first join operation is the driving row source for the second nested loops join.

The second join probes the index on the EMPLOYEE_ID column of the EMPLOYEES table. The system can do that because it knows (from the first join) the employee ID of all managers of departments in Seattle. Note that this is a unique scan because it is based on the primary key.

Finally, the EMPLOYEES table is accessed to retrieve the last name.

Execution Plan Interpretation: Example 3

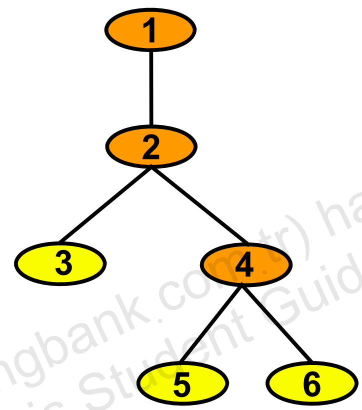
Execution Plan Interpretation: Example 3

```
select /*+ ORDERED USE_HASH(b) SWAP_JOIN_INPUTS(c) */ max(a.i)
from t1 a, t2 b, t3 c
where a.i = b.i and a.i = c.i;
```

```
0   SELECT STATEMENT
1    SORT AGGREGATE
2 1   HASH JOIN
3 2    TABLE ACCESS FULL T3
4 2    HASH JOIN
5 4     TABLE ACCESS FULL T1
6 4     TABLE ACCESS FULL T2
```

Expand All | Collapse All

Operation	Object	Order
SELECT STATEMENT		7
SORT AGGREGATE		6
HASH JOIN		5
TABLE ACCESS FULL	T3	1
HASH JOIN		4
TABLE ACCESS FULL	T1	2
TABLE ACCESS FULL	T2	3



Join order is: T1 - T2 - T3

ORACLE

Execution Plan Interpretation: Example 3

See the execution plan in the slide. Try to find the order in which the plan is executed and deduce what is the join order (order in which the system joins tables). Again, `ID` is in the first column and `PID` in the second column. The position is reflected by the indentation. It is important to recognize what the join order of an execution plan is, to be able to find your plan in a 10053 event trace file.

Here is the interpretation of this plan:

- The system first hashes the `T3` table (Operation `ID=3`) into memory.
- Then it hashes the `T1` table (Operation `ID=5`) into memory.
- Then the scan of the `T2` table begins (Operation `ID=6`).
- The system picks a row from `T2` and probes `T1` (`T1.i=T2.i`).
- If the row survives, the system probes `T3` (`T1.i=T3.i`).
- If the row survives, the system sends it to next operation.
- The system outputs the maximum value from the previous result set.

In conclusion, the execution order is : `3 – 5 – 6 – 4 – 2 – 1`

The join order is: `T1 – T2 – T3`

You can also use Enterprise Manager to understand execution plans, especially because it displays the Order column.

Note: A special hint was used to make sure `T3` would be first in the plan.

Reading More Complex Execution Plans

Reading More Complex Execution Plans

```
SELECT owner , segment_name , segment_type
FROM dba_extents
WHERE file_id = 1
AND   123213 BETWEEN block_id AND block_id + blocks -1;
```

Expand All | Collapse All

Operation	Object	Order	Rows	Bytes	Cost	CPU (%)	Time
SELECT STATEMENT		113			2,834	100	
VIEW	SYS.DBA_EXTENTS	112	2	140	2,834	0	0:0:35
UNION-ALL		111					
NESTED LOOPS		56	1	214	1,391	0	0:0:17
NESTED LOOPS		110	1	196	1,442	0	0:0:18

Collapse using indentation
and
focus on operations consuming most resources.

ORACLE

Reading More Complex Execution Plans

The plan at the left comes from the query (in the slide) on the data dictionary. It is so long that it is very difficult to apply the previous method to interpret it and locate the first operation.

You can always collapse a plan to make it readable. This is illustrated at the right where you can see the same plan collapsed. As shown, this is easy to do when using the Enterprise Manager or SQL Developer graphical interface. You can clearly see that this plan is a `UNION ALL` of two branches. Your knowledge about the data dictionary enables you to understand that the two branches correspond to dictionary-managed tablespaces and locally-managed ones. Your knowledge about your database enables you to know that there are no dictionary-managed tablespaces. So, if there is a problem, it must be on the second branch. To get confirmation, you must look at the plan information and execution statistics of each row source to locate the part of the plan that consumes most resources. Then, you just need to expand the branch you want to investigate (where time is being spent). To use this method, you must look at the execution statistics that are generally found in `V$SQL_PLAN_STATISTICS` or in the `tkprof` reports generated from trace files. For example, `tkprof` cumulates for each parent operation the time it takes to execute itself plus the sum of all its child operation time.

Reviewing the Execution Plan

- Drive from the table that has most selective filter.
- Look for the following:
  - Driving table has the best filter
  - Fewest number of rows are returned to the next step
  - The join method is appropriate for the number of rows returned
  - Views are correctly used
  - Unintentional Cartesian products
  - Tables accessed efficiently

ORACLE

Reviewing the Execution Plan

When you tune a SQL statement in an online transaction processing (OLTP) environment, the goal is to drive from the table that has the most selective filter. This means that there are fewer rows passed to the next step. If the next step is a join, this means fewer rows are joined. Check to see whether the access paths are optimal. When you examine the optimizer execution plan, look for the following:

- The plan is such that the driving table has the best filter.
- The join order in each step means that the fewest number of rows are returned to the next step (that is, the join order should reflect going to the best not-yet-used filters).
- The join method is appropriate for the number of rows being returned. For example, nested loop joins through indexes may not be optimal when many rows are returned.
- Views are used efficiently. Look at the `SELECT` list to see whether access to the view is necessary.
- There are any unintentional Cartesian products (even with small tables).
- Each table is being accessed efficiently: Consider the predicates in the SQL statement and the number of rows in the table. Look for suspicious activity, such as a full table scans on tables with large number of rows, which have predicates in the `WHERE` clause.

Also, a full table scan might be more efficient on a small table, or to leverage a better join method (for example, hash join) for the number of rows returned.

If any of these conditions are not optimal, consider restructuring the SQL statement or the indexes available on the tables.

Looking Beyond Execution Plans

- An execution plan alone cannot tell you whether a plan is good or not.
- May need additional testing and tuning:
  - SQL Tuning Advisor
  - SQL Access Advisor
  - SQL Performance Analyzer
  - SQL Monitoring
  - Tracing

ORACLE

Looking Beyond Execution Plans

The execution plan alone cannot differentiate between well-tuned statements and those that perform poorly. For example, an `EXPLAIN PLAN` output that shows that a statement uses an index does not necessarily mean that the statement runs efficiently. Sometimes indexes can be extremely inefficient.

It is best to use `EXPLAIN PLAN` to determine an access plan, and then later prove that it is the optimal plan through testing. When evaluating a plan, you should examine the statement's actual resource consumption.

The rest of this course is intended to show you various methods to achieve this.

Quiz

A user needs to be granted some specialized privileges to generate `AUTOTRACE` statistics.

a. True
b. False

Answer: a

Quiz

An `EXPLAIN PLAN` command executes the statement and inserts the plan used by the optimizer into a table.

a. True

b. False

ORACLE

Answer: b

Quiz

Which of the following is not true about a PLAN_TABLE?

a. The PLAN_TABLE is automatically created to hold the EXPLAIN PLAN output.
b. You cannot create your own PLAN_TABLE.
c. The actual SQL command is not executed.
d. The plan in the PLAN_TABLE may not be the actual execution plan.

Answer: b

Quiz

Quiz

After monitoring is initiated, an entry is added to the _______view. This entry tracks key performance metrics collected for the execution.

a. `V$SQL_MONITOR`

b. `V$PLAN_MONITOR`

c. `ALL_SQL_MONITOR`

d. `ALL_SQL_PLAN_MONITOR`

Answer: b

Summary

Summary

In this lesson, you should have learned how to:

- Gather execution plans
- Display execution plans
- Interpret execution plans

Practice 4: Overview

This practice covers the following topics:

- Using different techniques to extract execution plans
- Using SQL monitoring

ORACLE