Oracle 12c Database  Performance Tuning

By David Itshak
shaked19@gmail.com

http://www.ildba.co.il/author/cimid/
http://www.sqlserver.co.il/?cat=940

Global Hebrew Virtual PASS Chapter :
https://www.youtube.com/watch?v=x4hGjYGBfkc
https://www.youtube.com/watch?v=eJO8G9if3EY

SqlSaturday Israel 2016 :
Reference and Credits

Oracle® Database Concepts 12c Release 1 (12.1) E41396-13
https://docs.oracle.com/database/121/CNCPT/toc.htm

Oracle® Database SQL Language Reference 12c Release 1 (12.1) E41329-20

Oracle® Database SQL Tuning Guide 12c Release 1 (12.1) E49106-09
https://docs.oracle.com/database/121/TGSQL/title.htm

Oracle Database 12c Performance Tuning Recipes, Apress


Agenda

- Vocabulary
- Purpose of SQL Tuning
- Oracle database tuning tools
- Troubleshooting Poor Performance
- Oracle Database Tuning Tools
- Cursors
- Soft & Hard Parse
- SQL Processing
- Query Optimizer
- Selectivity and cardinality
- execution Plans
- Joins
- Histograms
Vocabulary

- **Bottleneck**: Area where resource contention is highest
- **Throughput**: Work per unit
- **Response Time**: Time to complete a given workload
- **Baseline**: Should Capture usage highs and lows
- **Snapshot**: Can be overlaid to compare against Baseline(s)
Purpose of SQL Tuning

• **Reduce user response time**: decreasing the time between when a user issues a statement and receives a response.

• **Improve throughput**: using the least amount of resources necessary to process all rows accessed by a statement.
Purpose of SQL Tuning

• **Response time**: An online book seller application that hangs for three minutes after a customer updates the shopping cart.

• **Throughput**: Three minute **parallel query** in a data warehouse that consumes all of the database host CPU, preventing other queries from running.

• In each case user response time is 3 minutes, but cause of the problem is different, and so is the tuning goal.
Working on the problem

Where is time spent? First, you have to identify where time goes. For example, if a specific operation takes ten seconds, you have to find out which module or component most of these ten seconds are used up in.

How is time spent? Once you know where the time goes, you have to find out how that time is spent. For example, you may find out that the component spends 4.2 seconds on CPU, 0.4 seconds doing disk I/O operations, and 5.1 seconds waiting for dequeuing a message coming from another component.

How can time spent be reduced? Finally, it is time to find out how the operation can be made faster. To do so, it is essential to focus on the most time-consuming part of the processing. For example, if disk I/O operations take 4% of the overall processing time, it makes no sense to start optimizing them, even if they are very slow.

• Throughput: Three minute parallel query in a data warehouse that consumes all of the database host CPU, preventing other queries from running.

In each case user response time is 3 minutes, but cause of the problem is different, and so is the tuning goal.

A typical web application consists of several components deployed on multiple systems.
Oracle Database Tuning Tools

- **Oracle Database 12C**
  - Enterprise Edition

- **Oracle Enterprise Manager Cloud Control**
  - Separate installation /licensing from DB

- **Oracle Diagnostics Pack**
  - Separate installation /licensing from DB

- **Oracle gives you all the “toys”, but be carefull**
  - License audits can be costly
What Is a Cursor?

• A cursor is a handle (a memory structure that enables a program to access a resource) that references a private SQL area with an associated shared SQL area.

• Although the handle is a client-side memory structure, it references a memory structure allocated by a server process that, in turn, references a memory structure stored in the SGA, and more precisely in the library cache.

A cursor is a handle to a private SQL area with an associated shared SQL area.
What Is a Cursor?

- **Private SQL area**:
  1. Stores data such as bind variable values and query execution state information.
  2. Belongs to a specific session.
  3. The session memory used to store private SQL areas is called user global area (UGA).

- **A shared SQL area**:
  - Consists of 2 separate structures: *parent cursor* and *child cursor*.
    1. **Parent cursor**: stores text of the SQL statement associated with the cursor
    2. **Child cursor**: stores the execution environment and the execution plan. These elements specify how the processing is carried out.
  - A shared SQL area can be used by several sessions, and therefore it’s stored in the library cache.
What Is a Cursor?

A PL/SQL block taking advantage of an implicit cursor; basically, the PL/SQL block delegates the control over the cursor to the PL/SQL compiler:

```sql
DECLARE
    l_ename emp.ename%TYPE := 'SCOTT';
    l_empno emp.empno%TYPE;
BEGIN
    SELECT empno INTO l_empno FROM emp
    WHERE ename = l_ename;
    dbms_output.put_line(l_empno);
END;
```
What Is a Cursor?

1. Once stored in the library cache, parent and child cursors are externalized through the v$sqlarea and v$sql views, respectively.

2. In most cases, cursors are identified with two columns: sql_id and child_number. sql_id column identifies parent cursors. Both values together identify child cursors.

Steps carried out during the parse phase
Soft & Hard Parse

- When shareable parent and child cursors are available and, consequently, only the first two operations are carried out, the parse is called a **soft parse**.

- When all operations are carried out, it's called a **hard parse**.

- From a performance point of view, you should avoid hard parses as much as possible. This is precisely why the database engine stores shareable cursors in the library cache. In this way, every process belonging to the instance might be able to reuse them.

**Hard parses should be avoided!**

- Generation of an execution plan is a very **CPU-intensive** operation.

- **Memory** in the shared pool is needed for storing the parent and child cursors in the library cache.

- Because the shared pool is shared over all sessions, memory allocations in the shared pool are serialized (with mutexes)
Shareable Cursors

SQL> SELECT * FROM t WHERE n = 1234;
SQL> select * from t where n = 1234;
SQL> SELECT * FROM t WHERE n=1234;
SQL> SELECT * FROM t WHERE n = 1234

SQL> SELECT sql_id, sql_text, executions
2 FROM v$sqlarea
3 WHERE sql_text LIKE '%1234';

<table>
<thead>
<tr>
<th>SQL_ID</th>
<th>SQL_TEXT</th>
<th>EXECUTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>2254m1487jg50</td>
<td>select * from t where n = 1234</td>
<td>1</td>
</tr>
<tr>
<td>g9y3jtp6ru4cb</td>
<td>SELECT * FROM t WHERE n = 1234</td>
<td>2</td>
</tr>
<tr>
<td>7n8p5s2udfdsn</td>
<td>SELECT * FROM t WHERE n=1234</td>
<td>1</td>
</tr>
</tbody>
</table>
Shareable Cursors

SQL> ALTER SESSION SET optimizer_mode = all_rows;
SQL> SELECT count(*) FROM t;
COUNT(*)
--------
1000

SQL> ALTER SESSION SET optimizer_mode = first_rows_1;
SQL> SELECT count(*) FROM t;
COUNT(*)
--------
1000

SQL> SELECT sql_id, child_number, optimizer_mode, plan_hash_value
2 FROM v$sql
3 WHERE sql_text = 'SELECT count(*) FROM t';

<table>
<thead>
<tr>
<th>SQL_ID</th>
<th>CHILD_NUMBER</th>
<th>OPTIMIZER_MODE</th>
<th>PLAN_HASH_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>5tjqf7sx5dzmj</td>
<td>0</td>
<td>ALL_ROWS</td>
<td>2966233522</td>
</tr>
<tr>
<td>5tjqf7sx5dzmj</td>
<td>1</td>
<td>FIRST_ROWS</td>
<td>2966233522</td>
</tr>
</tbody>
</table>

double parent cursor (5tjqf7sx5dzmj) and two child cursors (0 and 1)

New child cursor was created because of a new individual execution environment
The advantage of bind variables for performance is that they allow the sharing of parent cursors in the library cache and that way avoid hard parses and the overhead associated with them.

Example: three INSERT statements that, thanks to bind variables, share the same cursor in the library cache:
SQL Processing

- SQL Processing
- Query Optimizer Concepts
- Query Transformations
Stages SQL Processing

SQL Parsing
- Syntax Check
- Semantic Check
- Shared Pool Check

When an application issues a SQL statement, the application makes a parse call to the database to prepare the statement for execution.

The parse call opens or creates a cursor, which is a handle for the session-specific private SQL area that holds a parsed SQL statement and other processing information.

The cursor and private SQL area are in the program global area (PGA).

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

Semantic Check
SQL> SELECT * FROM nonexistent_table;
SELECT * FROM nonexistent_table
*
ERROR at line 1:
ORA-00942: table or view does not exist
Shared Pool Check

**SGA**
Shared components:
1. **Shared pool**: Caches the most recently used SQL statements that have been issued by database users.
2. **Database buffer cache**: Caches the data that has been most recently accessed by database users.
3. **Redo log buffer**: Stores transaction information for recovery purposes.

**Program Global Area (PGA)**
- Not shared.
- Components:
  1. **SQL Work Area**: Used for memory-intensive operations such as sorting or building a hash table during join operations.
  2. **Private SQL Area**: Holds information about SQL statement and bind variable values.
Shared Pool Check

- DB uses a hashing algorithm to generate a hash value for every SQL statement.

- The statement hash value is the SQL ID shown in V$SQL.SQL_ID.

- This hash value is deterministic within a version of Oracle Database.

- When a user submits a SQL statement, the database searches the shared SQL area to see if an existing parsed statement has the same hash value:

  1. **Memory address for the statement**: Oracle Database uses the SQL ID to perform a keyed read in a lookup table. In this way, the database obtains possible memory addresses of the statement.

  2. **Hash value of an execution plan for the statement**: A SQL statement can have multiple plans in the shared pool. Typically, each plan has a different hash value. If the same SQL ID has multiple plan hash values, then the database knows that multiple plans exist for this SQL ID.
Shared Pool Check

Parse Categories:

1. Hard parse/ library cache miss
   - If Oracle Database cannot reuse existing code, then it must build a new executable version of the application code.
   - DB always perform a hard parse of DDL.
   - During the hard parse, the database accesses the library cache and data dictionary cache numerous times to check the data dictionary. When the database accesses these areas, it uses a serialization device called a latch on required objects so that their definition does not change. Latch contention increases statement execution time and decreases concurrency.

2. Soft parse/ a library cache hit
   - Any parse that is not a hard parse.
   - If the submitted statement is the same as a reusable SQL statement in the shared pool, then Oracle Database reuses the existing code.
   - Soft parse is preferable to a hard parse because the database skips the optimization and row source generation steps, proceeding straight to execution.
Query Optimizer Concepts

- Purpose of the Query Optimizer
- Cost-Based Optimization
- Execution Plans
Purpose of the Query Optimizer

• The optimizer attempts to generate the best execution plan for a SQL statement.

• The best execution plan is defined as the plan with the lowest cost among all considered candidate plans.

• The cost computation accounts for factors of query execution such as I/O, CPU, and communication.

• The best method of execution depends on myriad conditions including how the query is written, the size of the data set, the layout of the data, and which access structures exist.

• The optimizer determines the best plan for a SQL statement by examining multiple access methods, such as full table scan or index scans, and different join methods such as nested loops and hash joins.

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

• Consider a user who queries records for employees who are managers. If the database statistics indicate that 80% of employees are managers, then the optimizer may decide that a full table scan is most efficient. However, if statistics indicate that few employees are managers, then reading an index followed by a table access by rowid may be more efficient than a full table scan.
Cost-Based Optimization

- The overall process of choosing the most efficient means of executing a SQL statement

- SQL is a nonprocedural language, so the optimizer is free to merge, reorganize, and process in any order.

- Based on statistics collected about the accessed data.

- Factors considered by the optimizer include:
  - System resources, which includes I/O, CPU, and memory
  - Number of rows returned
  - Size of the initial data sets
Cost-Based Optimization

- The cost is a number that represents the estimated resource usage for an execution plan.

- The optimizer’s cost model accounts for the I/O, CPU, and network resources that the database requires to execute the query.

- The optimizer assigns a cost to each possible plan, and then chooses the plan with the lowest cost.

- For this reason, the optimizer is sometimes called the cost-based optimizer (CBO) to contrast it with the legacy rule-based optimizer (RBO).
An execution plan describes a recommended method of execution for a SQL statement.

The plans show the combination of the steps Oracle Database uses to execute a SQL statement.

Each step either retrieves rows of data physically from the database or prepares them for the user issuing the statement.

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

• Each SELECT block in the original SQL statement is represented internally
  by a query block. A query block can be a top-level statement, subquery, or unmerged
• Ex: SQL statement consists of two query blocks

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

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

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

• The number of possible plans for a query block is proportional to the number of objects in the FROM clause. This number rises exponentially with the number of objects.

• For example, the possible plans for a join of five tables are significantly higher than the possible plans for a join of two tables.
```
SELECT * 
FROM sales 
WHERE promo_id=33 
OR prod_id=136;
```

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

- The **selectivity** represents a fraction of rows from a row set. The row set can be a base table, a view, or the result of a join.

- The selectivity is tied to a query predicate, such as last_name = 'Smith', or a combination of predicates, such as last_name = 'Smith' AND job_id = 'SH_CLERK'.

- **Note:** Selectivity is an internal calculation that is not visible in execution plans.
Selectivity and Cardinality

• \( \text{cardinality} = \text{selectivity} \times \text{num\_rows} \)

SQL> SELECT * FROM t;
10000 rows selected

• cardinality of the operation accessing the table is 2,601 selectivity is 0.2601 (2,601 rows returned out of 10,000):

SQL> SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000;
2601 rows selected.

• Cardinality = 0 => selectivity = 0

SQL> SELECT * FROM t WHERE n1 = 19;
no rows selected
Selectivity and Cardinality

- Execution plan contains at least one aggregate operation

```
SQL> SELECT sum(n2) FROM t WHERE n1 BETWEEN 6000 AND 7000;
SUM(N2)
----------
70846
```

- Executed following query to find out how many rows are returned by the access operation and passed as input to the aggregate

```
SQL> SELECT * FROM t WHERE n1 BETWEEN 6000 AND 7000;
2601 rows selected.
```

- Selectivity is 0.2601 (2,601/10,000)
Selectivity

- $0.0 \leq \text{Selectivity} < 1.0$.

- $\text{selectivity} = 0.0$ means that no rows.

- selectivity of 1.0 means that all rows are selected.

- A predicate becomes more selective as the value approaches 0.0 and less selective (or more unselective) as the value approaches 1.0.
Selectivity

• Optimizer estimates selectivity depending on whether statistics are available:
  
• **Statistics not available**

1. Depending on the value of the OPTIMIZER_DYNAMIC_SAMPLING initialization parameter, the optimizer either uses *dynamic statistics* or an internal default value.

1. DB uses different internal defaults depending on the predicate type. For example, the internal default for an equality predicate (last_name = 'Smith') is lower than for a range predicate (last_name > 'Smith') because an equality predicate is expected to return a smaller fraction of rows.
Cardinality

- **Number of rows returned by each operation in an execution plan**

- The optimizer determines the cardinality for each operation based on a complex set of formulas that use both table and column level statistics, or dynamic statistic.

- **With no histogram** optimizer assumes a uniform distribution. Ex:

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

  - Employees table contains **107 rows**.
  - Current DB statistics indicate that the **number of distinct values in the salary column is 58**.
  - Optimizer calculates the cardinality of the result set as 2, using the formula **107/58=1.84**.
Cardinality Importance

• Cardinality estimates must be as accurate as possible because they influence all aspects of the execution plan.

• Cardinality is important when the optimizer determines the cost of a join

• For example, in a nested loops join of the employees and departments tables, the number of rows in employees determines how often the database must probe the departments table.

• Cardinality is also important for determining the cost of sorts.
Cost

- The **optimizer cost model** accounts for the I/O, CPU, and network resources that a query is predicted to use.
- An internal numeric measure that represents the estimated resource usage for a plan.
- The lower the cost, the more efficient the plan.
- You cannot tune or change it.

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Cost (%CPU)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>14 (100)</td>
</tr>
<tr>
<td>1</td>
<td>HASH GROUP BY</td>
<td></td>
<td>14 (22)</td>
</tr>
<tr>
<td>2</td>
<td>HASH JOIN</td>
<td></td>
<td>13 (16)</td>
</tr>
<tr>
<td>3</td>
<td>VIEW</td>
<td>index$_join$_002</td>
<td>7 (15)</td>
</tr>
<tr>
<td>4</td>
<td>HASH JOIN</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>INDEX FAST FULL SCAN</td>
<td>PRODUCTS_PK</td>
<td>4 (0)</td>
</tr>
<tr>
<td>6</td>
<td>INDEX FAST FULL SCAN</td>
<td>PRODUCTS_PROD_CAT_IX</td>
<td>4 (0)</td>
</tr>
<tr>
<td>7</td>
<td>PARTITION RANGE ALL</td>
<td></td>
<td>5 (0)</td>
</tr>
<tr>
<td>8</td>
<td>TABLE ACCESS FULL</td>
<td>SALES</td>
<td>5 (0)</td>
</tr>
</tbody>
</table>
Cost of Access Path

- **Table scan or fast full index scan**
  - DB reads multiple blocks from disk in a single I/O. Cost of the scan depends on the number of blocks to be scanned and the multiblock read count value.

- **Index scan**
  - Cost of an index scan depends on the levels in the B-tree, the number of index leaf blocks to be scanned, and the number of rows to be fetched using the rowid in the index keys. The cost of fetching rows using rowids depends on the **index clustering factor**.
Clustering Factor

- For a B-tree index, the index clustering factor measures the physical grouping of rows in relation to an index value, such as last name.
- The index clustering factor helps the optimizer decide whether an index scan or full table scan is more efficient for certain queries.
- A low clustering factor indicates an efficient index scan.
- When a table is accessed through an index the estimated cost of that access is determined by index—not table—statistics.
- **knowing the selectivity isn’t sufficient for the CBO to estimate the cost**

Table blocks from a weakly clustered index

Table blocks from a strongly clustered index

Imaginary table with ten blocks that each contain 20 rows, making a total of 200 rows in the table. Let us assume that 20 rows in the table have a specific value for a specific column and that column is indexed. If these 20 rows are scattered around the table then there would be approximately two matching rows per block. As you can see, every block in the table would need to be read to obtain these 20 rows, so a full table scan would be more efficient than an indexed access, primarily because multi-block reads could be used to access the table, rather than single block reads.

In this case all 20 rows that we select through the index appear in a single block, and now an indexed access would be far more efficient, as only one table block needs to be read. selectivity is 10% (20 rows from 200) so knowing the selectivity isn’t sufficient for the CBO to estimate the cost of access to a table via an index. Enter the clustering factor.
The estimator uses three different measures to determine cost:

1. **Selectivity**: More selective as the selectivity value approaches 0 and less selective (or more unselective) as the value approaches 1.

2. **Cardinality**: The Rows column in an execution plan shows the estimated cardinality.

3. **Cost**: This measure represents units of work or resource used. The query optimizer uses disk I/O, CPU usage, and memory usage as units of work.
The plan generator explores various plans for a query block by trying out different access paths, join methods, and join orders.

Many plans are possible because of the various combinations that the database can use to produce the same result.

The optimizer picks the plan with the lowest cost.
Displaying Execution Plans

- **ID.** A number to uniquely identify the operations.

- **Operation.** There are some 200 operation codes.

- **Name.** DB object that is being operated on.

- **Rows.** Estimated number of rows that the operation is finally going to produce (a *cardinality estimate*). Applies to a single invocation of that operation and not the total number of rows returned by all invocations. In this case, operations 1 and 2 correspond to the subquery in the select list and are expected to return an average of about one row each time they are invoked.

- **Bytes** column shows an estimate of the number of bytes returned by one invocation of the operation.

- **Cost** column and time column both display the estimated elapsed time for the statement. The cost column expresses this elapsed time in terms of single-block read units.

### Example

```sql
SET LINES 200 PAGESIZE 0 FEEDBACK OFF
EXPLAIN PLAN FOR
  SELECT *
  ,d.dname
  ,d.loc
  , (SELECT COUNT (*)
  FROM scott.emp i
  WHERE i.deptno = e.deptno) dept_count
FROM scott.emp e, scott.dept d
WHERE e.deptno = d.deptno;

SELECT * FROM TABLE (DBMS_XPLAN.display);
```

### Execution Plan Table

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU)</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>14</td>
<td>1638</td>
<td>7 (15)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>13</td>
<td>1 (15)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 2</td>
<td>TABLE ACCESS FULL EMP</td>
<td></td>
<td>1</td>
<td>13</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>* 3</td>
<td>HASH JOIN</td>
<td></td>
<td>14</td>
<td>1638</td>
<td>7 (15)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL DEPT</td>
<td></td>
<td>4</td>
<td>120</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL EMP</td>
<td></td>
<td>14</td>
<td>1218</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

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

- 2 - filter("I"."DEPTNO"=:B1)
- 3 - access("E"."DEPTNO"="D"."DEPTNO")

### Note

Join of EMP and DEPT tables
How Operations Interact

- **Operation 0: SELECT STATEMENT**: This particular operation always has at least one child and calls the last child first.

- **Operation 1: SORT AGGREGATE**

- **Operation 2: TABLE ACCESS FULL**

- **Operation 3: HASH JOIN**

- **Operations 4 and 5: TABLE FULL SCANS**

#### Parent-child relationships in an execution plan

**Join of EMP and DEPT tables**

```
SET LINES 200 PAGESIZE 0 FEEDBACK OFF

EXPLAIN PLAN FOR
    SELECT e.*,
        d.dname,
        d.loc,
        (SELECT COUNT (*)
            FROM scott.emp i
            WHERE i.deptno = e.deptno)
        dept_count
    FROM scott.emp e, scott.dept d
    WHERE e.deptno = d.deptno;

SELECT * FROM TABLE (DBMS_XPLAN.display);
```

Plan hash value: 4153889731

<table>
<thead>
<tr>
<th>Id</th>
<th>Operation</th>
<th>Name</th>
<th>Rows</th>
<th>Bytes</th>
<th>Cost (%CPU</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SELECT STATEMENT</td>
<td></td>
<td>14</td>
<td>1638</td>
<td>7 (15)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>1</td>
<td>SORT AGGREGATE</td>
<td></td>
<td>1</td>
<td>13</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>2</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>1</td>
<td>13</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>3</td>
<td>HASH JOIN</td>
<td></td>
<td>14</td>
<td>1638</td>
<td>7 (15)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>4</td>
<td>TABLE ACCESS FULL</td>
<td>DEPT</td>
<td>4</td>
<td>120</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
<tr>
<td>5</td>
<td>TABLE ACCESS FULL</td>
<td>EMP</td>
<td>14</td>
<td>1218</td>
<td>3 (0)</td>
<td>00:00:01</td>
</tr>
</tbody>
</table>

Predicate Information (identified by operation id):

2 - filter("I"."DEPTNO":B1)
3 - access("E"."DEPTNO"="D"."DEPTNO")
How Operations Interact

• **Operation 0: SELECT STATEMENT**: This particular operation always has at least one child and calls the *last* child first.

• **Operation 1: SORT AGGREGATE**

• **Operation 2: TABLE ACCESS FULL**

• **Operation 3: HASH JOIN**

• Operations 4 and 5: TABLE FULL SCANs

Parent-child relationships in an execution plan

Join of EMP and DEPT tables
Joins : Nested Loop Join

- simplest join algorithm.
- Accepts two inputs, which are outer and inner tables.

- Inner nested loop join algorithm :
  
  for each row R1 in outer table
  
   for each row R2 in inner table
   
    if R1 joins with R2
    
     return join (R1, R2)

- The cost grows quickly with the size of the inputs; therefore, a nested loop join is efficient when at least one of the inputs is small.
Joins : Merge Join

• The *merge join* works with two sorted inputs. It compares two rows, one at time, and returns their join to the client if they are equal. Otherwise, it discards the lesser value and moves on to the next row in the input.

• The cost is proportional to the sum of the sizes of both inputs.

• More efficient on large inputs as compared to a nested loop join.

• Requires both inputs to be sorted, which is often the case when inputs are indexed on the join key column.

/* Prerequisites: Inputs I1 and I2 are sorted */
get first row R1 from input I1
get first row R2 from input I2
while not end of either input
begin
   if R1 joins with R2
   begin
      return join (R1, R2)
      get next row R2 from I2
   end
   else if R1 < R2
   get next row R1 from I1
   else /* R1 > R2 */
   get next row R2 from I2
end
Joins : Hash Join

- Designed to handle large unsorted inputs
- During the first, or build, phase, a hash join scans one of inputs, calculates the hash values of the join key, and places it into the hash table.
- Next, in the second, or probe, phase, it scans the second input, and checks, or probes, if the hash value of the join key from second input exists in the hash table. When this is the case, Oracle evaluates the join predicate for the row from the second input and all rows from the first input, which belong to the same hash bucket.
- This comparison must be done because the algorithm that calculates the hash values does not guarantee the uniqueness of the hash value of individual keys, which leads to hash collision when multiple different keys generate the same hash. Even though there is the possibility of additional overhead from the extra comparison operations due to hash collisions, those situations are relatively rare.

/* Build Phase */
for each row R1 in input I1
begin
    calculate hash value on R1 join key
    insert hash value to appropriate bucket in hash table
end

/* Probe Phase */
for each row R2 in input I2
begin
    calculate hash value on R2 join key
    for each row R1 in hash table bucket
        if R1 joins with R2
            return join (R1, R2)
    end
end
Comparing Join Types

<table>
<thead>
<tr>
<th></th>
<th>Nested Loop Join</th>
<th>Merge Join</th>
<th>Hash Join</th>
</tr>
</thead>
<tbody>
<tr>
<td>Best use-case</td>
<td>Small inputs. Preferable with index on join key in inner table.</td>
<td>Medium to large inputs sorted on index key.</td>
<td>Medium to large inputs.</td>
</tr>
<tr>
<td>Requires sorted input</td>
<td>No</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Requires equality predicate</td>
<td>No</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>Blocking operator</td>
<td>No</td>
<td>No</td>
<td>Yes (Build phase only)</td>
</tr>
<tr>
<td>Uses memory</td>
<td>No</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Uses tempdb</td>
<td>No</td>
<td>No (with exception of many-to-many joins)</td>
<td>Yes in case of spills</td>
</tr>
<tr>
<td>Preserves order</td>
<td>Yes (outer input)</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

- One of the common mistakes people make during performance tuning is relying strictly on the number of logical reads produced by the query. For example, it is entirely possible that a hash join produces fewer reads as compared to a nested loop. However, it would not factor in CPU usage and memory overhead or the performance implication in the case of tempdb spills and bailouts.

- Merge join is more efficient than a nested loop on sorted inputs.
Adaptive Query Optimization

- A set of capabilities that enables the optimizer to make run-time adjustments to execution plans and discover additional information that can lead to better statistics.

- Adaptive optimization is helpful when existing statistics are not sufficient to generate an optimal plan.
Optimizer Statistics

- Data stored in the data dictionary used by the Oracle cost-based optimizer (CBO) to determine query execution plans
  - View by using V$USER_TAB_STATISTICS

- Table statistics: Number of rows, number of blocks, average row length

- Column statistics: Number of distinct values, minimum and maximum values

- Index statistics: Number of distinct values; depth of the index, number of leaf blocks, clustering factor
Gathering Statistics

- **DBMS_STATS PL/SQL package**
  - Works on tables, partitions, indexes, and columns

- **EM Cloud Control 12c**
  - Executes PL/SQL under the hood

- **When you use DBCA, an Automatic Maintenance Task (AutoTask) is created using Oracle Scheduler**

- **Oracle considers stats “stale” when more than 10 percent of the table data has changed since stats were last gathered**
Gathering Statistics

TIP: The Objects step will be skipped when Database, Fixed Objects or Dictionary Objects is selected.

Options for Scope: Schema

- Use Oracle-recommended option settings
  Oracle will select objects for which to gather optimizer statistics based on the activity on the objects. Also, Oracle will use the best options for generating the statistics. The Customize Options step will be skipped if you choose this option.

- View Oracle-recommended option settings

Validate With SQL Performance Analyzer

- Validate impact of stats on SQL performance prior to publishing (recommended). The database global statistics gathering option PUBLISH will be set to FALSE temporarily during the process.
**Histograms**

- The additional information needed by the query optimizer to get information about the nonuniform distribution of data is called a histogram.

- Prior to version 12.1, two types of histograms are available: **frequency histograms** and **height-balanced histograms**. Oracle Database 12.1 introduces two additional types to replace height-balanced histograms: **top frequency histograms** and **hybrid histograms**.

```sql
SQL> SELECT val2, count(*)
2 FROM t
3 GROUP BY val2
4 ORDER BY val2;

<table>
<thead>
<tr>
<th>VAL2</th>
<th>COUNT(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>8</td>
</tr>
<tr>
<td>102</td>
<td>25</td>
</tr>
<tr>
<td>103</td>
<td>68</td>
</tr>
<tr>
<td>104</td>
<td>185</td>
</tr>
<tr>
<td>105</td>
<td>502</td>
</tr>
<tr>
<td>106</td>
<td>212</td>
</tr>
</tbody>
</table>
```

If data isn’t uniformly distributed, the query optimizer can’t compute acceptable estimations without additional information. For example, given the data set stored in the val2 column, how could the query optimizer make a meaningful estimation for a predicate like `val2=105`? It can’t, because it has no clue that about 50 percent of the rows fulfill that predicate:
Histograms

- The additional information needed by the query optimizer to get information about the nonuniform distribution of data is called a histogram.

- Prior to version 12.1, two types of histograms are available: frequency histograms and height-balanced histograms. Oracle Database 12.1 introduces two additional types to replace height-balanced histograms: top frequency histograms and hybrid histograms.

```
SQL> SELECT val2, count(*)
2  FROM t
3  GROUP BY val2
4  ORDER BY val2;

VAL2   COUNT(*)
-------    -------
101       8
102      25
103      68
104     185
105     502
106     212
```

If data isn’t uniformly distributed, the query optimizer can’t compute acceptable estimations without additional information. For example, given the data set stored in the val2 column, how could the query optimizer make a meaningful estimation for a predicate like val2=105? It can’t, because it has no clue that about 50 percent of the rows fulfill that predicate.
Ex: Frequency Histograms

- The frequency histogram stored in the data dictionary is similar to this representation.

```
SQL> SELECT endpoint_value, endpoint_number,
           2 endpoint_number - lag(endpoint_number,1,0)
  3 OVER (ORDER BY endpoint_number) AS frequency
  4 FROM user_tab_histograms
  5 WHERE table_name = 'T'
  6 AND column_name = 'VAL2'
  7 ORDER BY endpoint_number;
```