SQL Code Performance & Tuning

Oracle Academy Study Materials

Optimizer Statistics

- Describe the database and the objects in the database
- Information used by the query optimizer to estimate:
  - Selectivity of predicates
  - Cost of each execution plan
  - Access method, join order, and join method
  - CPU and input/output (I/O) costs
- Refreshing optimizer statistics whenever they are stale is as important as gathering them:
  - Automatically gathered by the system
  - Manually gathered by the user with DBMS_STATS

Optimizer Statistics

Types of Optimizer Statistics

- Table statistics:
  - Number of rows
  - Number of blocks
  - Average row length
- Index Statistics:
  - B*-tree level
  - Distinct keys
  - Number of leaf blocks
  - Clustering factor
- System statistics
  - I/O performance and utilization
  - CPU performance and utilization
- Column statistics
  - Basic: Number of distinct values, number of nulls, average length, min, max
  - Histograms (data distribution when the column data is skewed)
  - Extended statistics

Optimizer Statistics

Table Statistics (`DBA_TAB_STATISTICS`)

- Used to determine:
  - Table access cost
  - Join cardinality
  - Join order
- Some of the statistics gathered are:
  - Row count (NUM_ROWS)
  - Block count (BLOCKS) Exact
  - Empty blocks (EMPTY_BLOCKS) Exact
  - Average free space per block (AVG_SPACE)
  - Average row length (AVG_ROW_LEN)
  - Statistics status (STALE_STATS)

Optimizer Statistics

Index Statistics (`DBA_IND_STATISTICS`)

- Used to decide:
  - Full table scan versus index scan
- Statistics gathered are:
  - B*-tree level (BLEVEL) Exact
  - Leaf block count (LEAF_BLOCKS)
  - Distinct keys (DISTINCT_KEYS)
  - Average number of leaf blocks in which each distinct value in the index appears (AVG_LEAF_BLOCKS_PER_KEY)
  - Average number of data blocks in the table pointed to by a distinct value in the index (AVG_DATA_BLOCKS_PER_KEY)
  - Number of rows in the index (NUM_ROWS)

Optimizer Statistics

Column Statistics (`DBA_TAB_COL_STATISTICS`)

- Count of distinct values of the column (NUM_DISTINCT)
- Low value (LOW_VALUE) Exact
- High value (HIGH_VALUE) Exact
- Number of nulls (NUM_NULLS)
- Selectivity estimate for nonpopular values (DENSITY)
- Number of histogram buckets (NUM_BUCKETS)
- Type of histogram (HISTOGRAM) - histograms are useful when you have a high degree of skew in the column distribution

Mechanisms for Gathering Statistics

- Automatic statistics gathering
  - gather_stats_prog automated task
- Manual statistics gathering
  - DBMS_STATS package
- When the system encounters a table with missing statistics, it dynamically gathers the necessary statistics needed by the optimizer.
- However, for certain types of tables, it does not perform dynamic sampling. These include remote tables and external tables.
- In those cases and also when dynamic sampling has been disabled, the optimizer uses default values for its statistics.

When to Gather Statistics Manually

- Rely mostly on automatic statistics collection:
  - Change the frequency of automatic statistics collection to meet your needs.
  - Remember that STATISTICS_LEVEL should be set to TYPICAL or ALL for automatic statistics collection to work properly.
- Gather statistics manually for:
  - Objects that are volatile
  - Objects modified in batch operations
  - External tables, system statistics, fixed objects
  - Objects modified in batch operations: Gather statistics as part of the batch operation.
  - New objects: Gather statistics right after object creation.

Optimizer Hints

- Hints enable you to influence decisions made by the optimizer.
- Hints provide a mechanism to direct the optimizer to select a certain query execution plan based on the specific criteria.
  - For example, you might know that a certain index is more selective for certain queries. Based on this information, you might be able to select a more efficient execution plan than the plan that the optimizer recommends. In such a case, use hints to force the optimizer to use the optimal execution plan.
  - Example:

```
SELECT /*+ INDEX(e empfirstname_idx) skewed col */ *
FROM employees e
WHERE first_name='David'
```

- You use comments in a SQL statement to pass instructions to the optimizer.
- The plus sign (+) causes the system to interpret the comment as a list of hints. The plus sign must follow immediately after the comment delimiter. No space is permitted.
- When you use a hint, it is good practice to also add a comment about that hint.
- HINTS SHOULD ONLY BE USED AS A LAST RESORT.

Types of Hints

Single-table hints	Specified on one table or view
Multitable hints	Specify more than one table or view
Query block hints	Operate on a single query block
Statement hints	Apply to the entire SQL statement

Specifying Hints

- Hints apply to the optimization of only the block of the statement in which they appear.
- A statement block is:
  - A simple MERGE, SELECT, INSERT, UPDATE, or DELETE statement
  - A parent statement or a subquery of a complex statement
  - A part of a compound query using set operators (UNION, MINUS, INTERSECT)
- For example:
  - A compound query consisting of two component queries combined by the UNION operator has two blocks, one for each component query. For this reason, hints in the first component query apply only to its optimization, not to the optimization of the second component query.

Rules for Hints

- Place hints immediately after the first SQL keyword of a statement block.
- Each statement block can have only one hint comment, but it can contain multiple hints.
- Hints apply to only the statement block in which they appear.
- If a statement uses aliases, hints must reference the aliases rather than the table names.
- The optimizer ignores hints specified incorrectly without raising errors.
- Use hints carefully because they imply a high-maintenance load.
- Be aware of the performance impact of hard-coded hints when they become less valid.

Hint Categories

There are hints for:

- Optimization approaches and goals
- Access paths
- Query transformations
- Join orders
- Join operation
- Parallel execution
- Additional hints

```
UPDATE /*+ INDEX(p PRODUCTS_PROD_CAT_IX)*/
products p
SET   p.prod_min_price =
        (SELECT
         (pr.prod_list_price*.95)
FROM products pr
WHERE p.prod_id = pr.prod_id)
WHERE p.prod_category = 'Men'
AND   p.prod_status = 'available, on stock'
/
```

Optimization Goals and Approaches

ALL_ROWS	Selects a cost-based approach with a goal of best throughput - minimum total resource consumption
FIRST_ROWS(*n*)	Instructs the Oracle server to optimize an individual SQL statement for fast response

- If you specify either the ALL_ROWS or the FIRST_ROWS(n) hint in a SQL statement, and if the data dictionary does not have statistics about tables accessed by the statement, then the optimizer uses default statistical values to estimate the missing statistics.
- If you specify hints for access paths or join operations along with either the ALL_ROWS or FIRST_ROWS(n) hint, the optimizer gives precedence to the access paths and join operations specified by the hints.
- **Note**: The FIRST_ROWS hints are probably the most useful hints.

```
SELECT /*+ ALL_ROWS */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE employee_id = 7566;
```

```
SELECT /*+ FIRST_ROWS(10) */ employee_id, last_name, salary, job_id
  FROM employees
  WHERE department_id = 20;
```

Optimizer Hints

- If you alter the definition of a referenced object, dependent objects might not continue to work properly.
- Example: if the table definition is changed, the procedure might not continue to work without error.
- The Oracle server automatically records dependencies among objects.
- To manage dependencies, all schema objects have a status (valid or invalid) that is recorded in the Data Dictionary, and you can view the status in the USER_OBJECTS Data Dictionary view.

Hints for Join Orders

ORDERED	Causes the Oracle server to join tables in the order in which they appear in the `FROM` clause
LEADING	Uses the specified tables as the first table in the join order

```
SELECT  /*+ORDERED */ o.order_id, c.customer_id, l.unit_price * l.quantity
  FROM customers c, order_items l, orders o
  WHERE c.cust_last_name = :b1
    AND o.customer_id = c.customer_id
    AND o.order_id = l.order_id;
```

```
SELECT /*+ LEADING(e j) */ *
    FROM employees e, departments d, job_history j
    WHERE e.department_id = d.department_id
      AND e.hire_date = j.start_date;
```

PL/SQL Code Performance & Tuning

Oracle Academy Study Materials

Performance & Tuning

Standardizing Constants and Exceptions

- Constants and exceptions are typically implemented using a bodiless package (that is, a package specification).
- Standardizing helps to:
  - Develop programs that are consistent
  - Promote a higher degree of code reuse
  - Ease code maintenance
  - Implement company standards across entire applications
- Start with standardization of:
  - Exception names
  - Constant definitions

Standardizing Exceptions

Create a standardized error-handling package that includes all named and programmer-defined exceptions to be used in the application.

```
CREATE OR REPLACE PACKAGE error_pkg IS
  e_fk_err      EXCEPTION;
  e_seq_nbr_errEXCEPTION;
  PRAGMA EXCEPTION_INIT (e_fk_err, -2292);
  PRAGMA EXCEPTION_INIT (e_seq_nbr_err, -2277);
  ...
END error_pkg;
```

Standardizing Exception Handling

- Standardized exception handling can be implemented either as a stand-alone subprogram or a subprogram added to the package that defines the standard exceptions.
- Consider creating a package with:
  - Every named exception that is to be used in the application
  - All unnamed, programmer-defined exceptions that are used in the application. These are error numbers –20000 through –20999.
  - A program to call RAISE_APPLICATION_ERROR based on package exceptions
  - A program to display an error based on the values of SQLCODE and SQLERRM
  - Additional objects, such as error log tables, and programs to access the tables

Standardizing Exception Handling

- A common practice is to use parameters that identify the name of the procedure and the location in which the error has occurred. This enables you to keep track of run-time errors more easily.
- An alternative is to use the RAISE_APPLICATION_ERROR built-in procedure to keep a stack trace of exceptions that can be used to track the call sequence leading to the error. To do this, set the third optional argument to TRUE. For example:

```
RAISE_APPLICATION_ERROR(-20001, 'My first error', TRUE);
```

- This is meaningful when more than one exception is raised in this manner.

Standardizing Constants

For programs that use local variables whose values should not change:

- Convert the variables to constants to reduce maintenance and debugging
- Create one central package specification and place all constants in it

```
CREATE OR REPLACE PACKAGE constant_pkg IS
  c_order_received CONSTANT VARCHAR(2) := 'OR';
  c_order_shipped  CONSTANT VARCHAR(2) := 'OS';
  c_min_sal        CONSTANT NUMBER(3)  := 900;
END constant_pkg;
```

Local Subprograms

A local subprogram is a PROCEDURE or FUNCTION defined at the end of the declarative section.

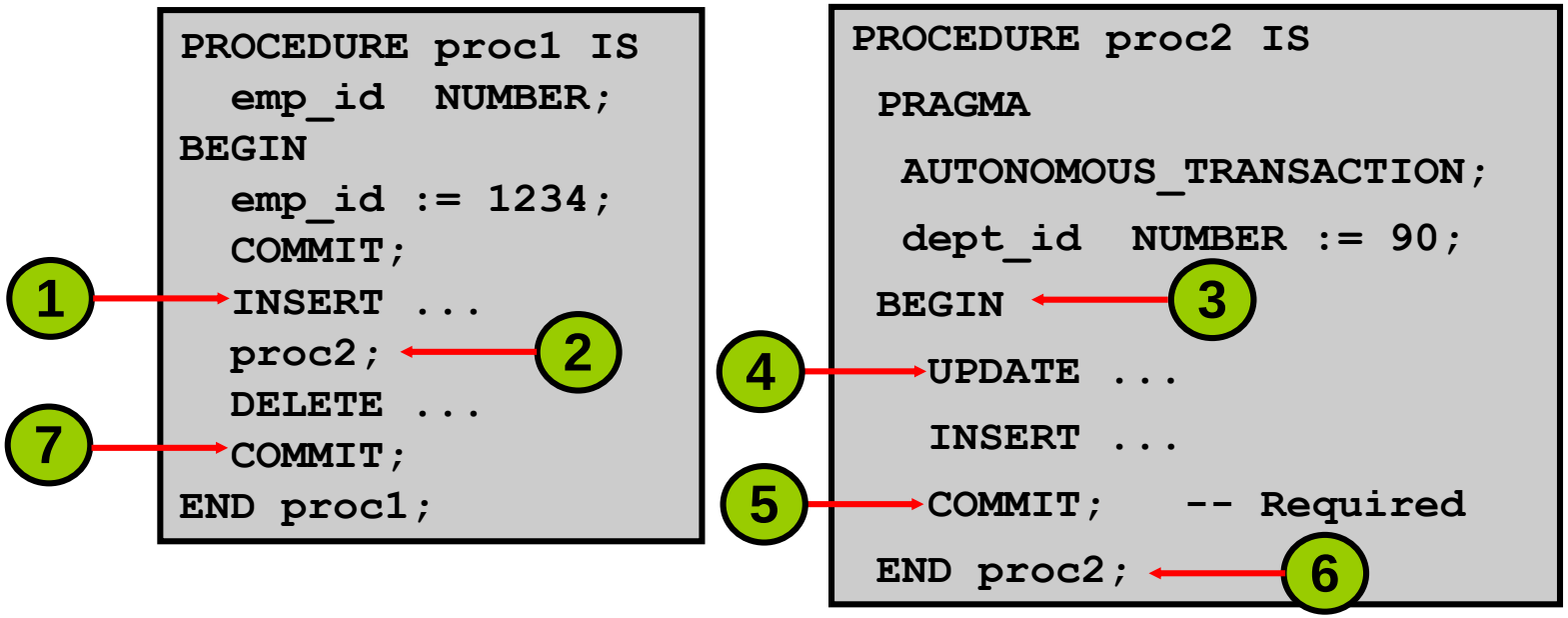
```
CREATE PROCEDURE employee_sal(p_id NUMBER) IS
   v_emp employees%ROWTYPE;
   FUNCTION tax(p_salary VARCHAR2) RETURN NUMBER IS
   BEGIN
     RETURN p_salary * 0.825;
   END tax;
BEGIN
   SELECT * INTO v_emp
   FROM EMPLOYEES WHERE employee_id = p_id;
   DBMS_OUTPUT.PUT_LINE('Tax: '|| tax(v_emp.salary));
END;
/
EXECUTE employee_sal(100)
```

```
PROCEDURE employee_sal(p_id Compiled.
anonymous block completed
Tax: 19800
```

Autonomous Transactions

- A transaction is a series of statements doing a logical unit of work that completes or fails as an integrated unit.
- Often, one transaction starts another that may need to operate outside the scope of the transaction that started it – in an existing transaction, a required independent transaction may need to commit or roll back changes without affecting the outcome of the starting transaction.
- An **autonomous transaction** (AT) is an independent transaction started by another main transaction (MT).
- Are specified with PRAGMA AUTONOMOUS_TRANSACTION

Autonomous Transactions



1. The main transaction begins.
2. A proc2 procedure is called to start the autonomous transaction.
3. The main transaction is suspended.
4. The autonomous transactional operation begins.
5. The autonomous transaction ends with a commit or roll back operation.
6. The main transaction is resumed.
7. The main transaction ends.

Features of Autonomous Transactions

- Are independent of the main transaction.
- Are not nested transactions.
- Suspend the calling transaction until the autonomous transactions are completed
- Do not roll back if the main transaction rolls back
- Changes made by an autonomous transaction become visible to other transactions when the autonomous transaction commits.
- There are no limits other than resource limits on how many autonomous transactions can be recursively called.
- Autonomous transactions must be explicitly committed or rolled back; otherwise, an error is returned when attempting to return from the autonomous block.
- You cannot use PRAGMA to mark all subprograms in a package as autonomous. Only individual routines can be marked autonomous.
- You cannot mark a nested or anonymous PL/SQL block as autonomous.

Using Autonomous Transactions: Example

```
PROCEDURE bank_trans(p_cardnbr NUMBER, p_loc NUMBER) IS
BEGIN
  log_usage(p_cardnbr, p_loc);
  INSERT INTO txn VALUES (9001, 1000,...);
END bank_trans;
```

```
PROCEDURE log_usage (p_card_id NUMBER, p_loc NUMBER)
IS
  PRAGMA AUTONOMOUS_TRANSACTION;
BEGIN
  INSERT INTO usage       -- usage is an existing table
  VALUES (p_card_id, p_loc);
  COMMIT;
END log_usage;
```

Using the PARALLEL_ENABLE Hint

- Can be used in functions as an optimization hint
- Indicates that a function can be used in a parallelized query or parallelized DML statement.
- Parallel execution feature divides the work of executing a SQL statement across multiple processes.
- Functions called from a SQL statement that is run in parallel can have a separate copy run in each of these processes, with each copy called for only the subset of rows that are handled by that proces.

```
CREATE OR REPLACE FUNCTION f2 (p_p1 NUMBER)
  RETURN NUMBER PARALLEL_ENABLE IS
BEGIN
  RETURN p_p1 * 2;
END f2;
```

```
SELECT /*+ PARALLEL(orders,4,1) */
       COUNT(*)
FROM orders;
```

SQL Query Result Cache

- Definition:
  - Cache the results of the current query or query fragment in memory, and then use the cached results in future executions of the query or query fragments.
  - Cached results reside in the result cache memory portion of the SGA.
- Benefits:
  - Improved performance

SQL Query Result Cache

- Scenario:
  - You need to find the greatest average value of credit limit grouped by state over the whole population.
  - The query results in a large number of rows being analyzed to yield a few or one row.
  - In your query, the data changes fairly slowly (say every hour) but the query is repeated fairly often (say every second).
- Solution:
  - Use the new optimizer hint /*+ result_cache */ in your query:

```
SELECT /*+ result_cache */
  AVG(cust_credit_limit), cust_state_province
FROM sh.customers
GROUP BY cust_state_province;
```

Using the Cross-Session PL/SQL Function Result Cache

- Each time a result-cached PL/SQL function is called with different parameter values, those parameters and their results are stored in cache.
- The function result cache is stored in a shared global area (SGA), making it available to any session that runs your application.
- Subsequent calls to the same function with the same parameters uses the result from cache.
- Performance and scalability are improved.
- This feature is used with functions that are called frequently and dependent on information that changes infrequently.
- To make a function result-cached:
  1. Include the RESULT_CACHE clause in the function declaration / definition
  2. Include an optional RELIES_ON clause to specify any tables or views on which the function results depend.

Using the Cross-Session PL/SQL Function Result Cache

```
CREATE OR REPLACE FUNCTION emp_hire_date (p_emp_id
    NUMBER, fmt VARCHAR) RETURN VARCHAR
 RESULT_CACHE RELIES_ON (employees) IS
  v_date_hired DATE;
BEGIN
  SELECT hire_date INTO v_date_hired
  FROM HR.Employees
  WHERE Employee_ID = p_emp_ID;
  RETURN to_char(v_date_hired, fmt);
END;
```

Using the DETERMINISTIC Clause with Functions

- Specify DETERMINISTIC to indicate that the function returns the same result value whenever it is called with the same values for its arguments.
- This helps the optimizer avoid redundant function calls.
- If a function was called previously with the same arguments, the optimizer can elect to use the previous result.
- Do not specify DETERMINISTIC for a function whose result depends on the state of session variables or schema objects.
- Do not specify this clause to define a function that uses package variables or that accesses the database in any way that might affect the return result of the function. The results of doing so will not be captured if Oracle Database chooses not to reexecute the function.

```
CREATE OR REPLACE FUNCTION text_length(a CLOB)
   RETURN NUMBER DETERMINISTIC IS
BEGIN
  RETURN DBMS_LOB.GETLENGTH(a);
END;
```

Comparing SQL with PL/SQL

– Some simple set processing is markedly faster than the equivalent PL/SQL.

```
BEGIN
  INSERT INTO inventories2
    SELECT product_id, warehouse_id
    FROM main_inventories;
END;
```

– Avoid using procedural code when it may be better to use SQL.

```
...FOR I IN 1..5600 LOOP
     counter := counter + 1;
     SELECT product_id, warehouse_id
       INTO v_p_id, v_wh_id
       FROM big_inventories WHERE v_p_id = counter;
     INSERT INTO inventories2 VALUES(v_p_id, v_wh_id);
   END LOOP;...
```

Comparing SQL with PL/SQL

- However, there are occasions when you will get better performance from PL/SQL, even when the process could be written in SQL.
- Correlated updates are slow. With correlated updates, a better method is to access only correct rows by using PL/SQL.
- The following PL/SQL loop is faster than the equivalent correlated update SQL statement:

```
DECLARE
  CURSOR cv_raise IS
    SELECT deptno, increase
    FROM emp_raise;
BEGIN
    FOR dept IN cv_raise LOOP
      UPDATE big_emp
        SET sal = sal * dept.increase
        WHERE deptno = dept.deptno;
    END LOOP;
...
```

Comparing SQL with PL/SQL

- From a design standpoint, do not embed your SQL statements directly within the application code.
- It is better if you write procedures to perform your SQL statements.
- Pros:
  - If you design your application so that all programs that perform an insert on a specific table use the same INSERT statement, your application will run faster because of less parsing and reduced demands on the System Global Area (SGA) memory.
  - Your program will also handle data manipulation language (DML) errors consistently.
- Cons:
  - You may need to write more procedural code.
  - You may need to write several variations of update or insert procedures to handle the combinations of columns that you are updating or inserting into.

Comparing SQL with PL/SQL

– Instead of:

```
...
INSERT INTO order_items
  (order_id, line_item_id, product_id,
   unit_price, quantity)
VALUES (...
```

– Create a stand-alone procedure:

```
insert_order_item (  2458, 6, 3515, 2.00, 4);
```

– Or a packaged procedure:

```
orderitems.ins (  2458, 6, 3515, 2.00, 4);
```