Best Practices: SQL

- ✓ **Strictly avoid "Select * ... "** , define exactly the fields you retrieve
- ✓ Use the **table name prefix for every field** you retrieve, ej. <u>TblEntRev.CveEntRev</u>
- ✓ **Calculate** the exact **cost** of execution (must be < 2, if not double check it)
    - o <u>EXPLAIN PLAN FOR SELECT * FROM tblentrev</u>
    - o <u>SELECT * FROM plan_table</u>
    - o Write the cost as a comment inside the code.
- ✓ **Avoid SORT operations** whenever possible. Operations requiring sorts:
    - o ORDER BY
    - o GROUP BY
    - o DISTINCT
    - o UNION, INTERSECT, MINUS

  If it is necessary think about INDEXES (see INDEXES section)
- ✓ **Order of the tables in Joins:** If you specify 2 or more tables in the FROM clause of a SELECT statement, then Oracle parser will process the tables from right to left, so the table name you specify last will be processed first. In this case you have to choose one table as driving table. Always choose the table with less number of records as the driving table.
- ✓ **Use Bind Variables:** It is also better to use bind variables in queries. That way the query becomes generic and therefore re-usable. For example, instead of writing a query like:
    - ▪ SELECT ename, sal
    - ▪ FROM emp
    - ▪ WHERE deptno = 20;

  Change it to:
    - ▪ SELECT ename, sal
    - ▪ FROM emp
    - ▪ WHERE deptno = :deptno;

  The first query can be re-used for deptno number 20 only, whereas the second query can be reused for any other deptno also.
- ✓ **SQL Writing Convention:** It is a good practice to use a standard syntax for wiring SQL queries. I will recommend following standards to use while writing SQL queries. Write all standard SQL TEXT in upper case. For example:
    - ▪ SELECT ename, sal
    - ▪ FROM emp
    - ▪ WHERE deptno = 20;

  Write all non standard SQL TEXT (Table name, Column name etc) in lower case. For example:
  Note: It is important to write similar SQL statement in same case. For example: **Oracle will reparse following queries as they are not written in the same case**
    - ▪ Select * from EMP;
    - ▪ Select * from emp;
- ✓ **Use EXISTS instead of DISTINCT:** Use EXISTS in place of DISTINCT if you want the resultset to contain distinct values while joining tables. For example:
    - ▪ SELECT DISTINCT d.deptno, d.dname

  - FROM dept d, emp e
  - WHERE d.deptno = e.deptno;

The following SQL statement is a better alternative.

  - SELECT d.deptno, d.dname
  - FROM dept d
  - WHERE EXISTS (SELECT e.deptno
  - FROM emp e
  - WHERE d.deptno = e.deptno);

- ✓ **Use of expressions and indexes:** The optimizer fully evaluates expressions whenever possible and translates certain syntactic constructs into equivalent constructs. This is done either because Oracle can more quickly evaluate the resulting expression than the original expression or because the original expression is merely a syntactic equivalent of the resulting expression. Any computation of constants is performed only once when the statement is optimized rather than each time the statement is executed. Consider these conditions that test for salaries greater than $2000.
  - sal > 24000/12
  - sal > 2000
  - sal*12 > 24000

  If a SQL statement contains the first condition, the optimizer simplifies it into the second condition. Please note that optimizer does not simplify expressions across comparison operators. The optimizer does not simplify the third expression into the second. For this reason, we should write conditions that compare columns with constants whenever possible, rather than conditions with expressions involving columns. The Optimizer does not use index for the following statement:
  - SELECT *
  - FROM emp
  - WHERE sal*12 > 24000 ;

  Instead of this use the following statement:
  - SELECT *
  - FROM emp
  - WHERE sal > 24000/12 ;
- ✓ **Use of NOT operator on indexed columns:** Never use NOT operator on an indexed column. Whenever Oracle encounters a NOT on an index column, it will perform full-table scan. For Example:
  - SELECT *
  - FROM emp
  - WHERE NOT deptno = 0;

  Instead use the following:
  - SELECT *
  - FROM emp
  - WHERE deptno > 0;
- ✓ **Avoid Transformed Columns in the WHERE Clause:** Use untransformed column values. For example, use:
  - WHERE a.order_no = b.order_no

  Rather than
  - WHERE TO_NUMBER (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
  - = TO_NUMBER (SUBSTR(a.order_no, INSTR(b.order_no, '.') - 1))
- ✓ **Combine Multiples Scans with CASE Statements:** Often, it is necessary to calculate different aggregates on various sets of tables. Usually, this is done with multiple scans on the table, but it is easy to calculate all the aggregates with one single scan. Eliminating n-1 scans can greatly improve performance. Combining multiple scans into one scan can be done by

moving the WHERE condition of each scan into a CASE statement, which filters the data for the aggregation. For each aggregation, there could be another column that retrieves the data. The following example has count of all employees who earn less than 2000, between 2000 and 4000, and more than 4000 each month. This can be done with three separate queries.

- SELECT COUNT (*)
- FROM emp
- WHERE sal < 2000;

- SELECT COUNT (*)
- FROM emp
- WHERE sal BETWEEN 2000 AND 4000;

- SELECT COUNT (*)
- FROM emp
- WHERE sal>4000;

However, it is more efficient to run the entire query in a single statement. Each number is calculated as one column. The count uses a filter with the CASE statement to count only the nrows where the condition is valid. For example:

- SELECT COUNT (CASE WHEN sal < 2000
- THEN 1 ELSE null END) count1,
- COUNT (CASE WHEN sal BETWEEN 2001 AND 4000
- THEN 1 ELSE null END) count2,
- COUNT (CASE WHEN sal > 4000
- THEN 1 ELSE null END) count3
- FROM emp;

✓ **Running a query in a loop is a BAD idea**. If you are executing the same query with different data, consider building a query string using UNION and executing it at the end of the loop, so you can execute multiple queries with only one trip across the network to the database.

✓ **Limit the use of correlated subqueries**; often they can be replaced with a JOIN.

Database Index Tips

- ✓ Create indexes on columns referenced in the ORDER BY clause.
- ✓ Put the most unique data element first in the index, the element that has the biggest variety of values. The index will find the correct page faster.
- ✓ Keep indexes small. It's better to have an index on just zip code or postal code, rather than postal code & country. The smaller the index, the better the response time.
- ✓ For high frequency functions (thousands of times per day) it can be wise to have a very large index, so the system does not even need the table for the read function.
- ✓ For small tables an index is disadvantageous. For ***any function*** the system would be better off by scanning the whole table. An index would only slow down.
- ✓ Index note:
- ✓ An index slows down additions, modifications and deletes. It's not just the table that needs an update, but the index as well. So, preferably, add an index for values that are often used for a search, but that do not change much. An index on bank account number is better than one on balance.
- ✓ Some databases extend the power of indexing by allowing indexes to be created on functions or expressions. For example, an index could be created on upper(last_name), which would only store the uppercase versions of the last_name field in the index
- ✓ It is possible to retrieve a set of row identifiers using only the first indexed column. However, it is not possible or efficient (on most databases) to retrieve the set of row identifiers using only the second or greater indexed column.