SQL Performance Tuning

Driving Table and Join Order

1. The DRIVING TABLE is the first table from which other tables in a query can be joined. You have to start by getting your first set of rows from some table in your query so where you start is the DRIVING TABLE Two things make the driving table important:

   It determines which tables can be accessed next during query execution, effectively deciding on a road-map for the query.

   It determines the initial number of rows that the query will start with and thus drives the amount of work the query will have to do in subsequent steps.

2. JOIN ORDER determines ongoing workload for the query.

   The key to correct join order for our purposes, is to keep the intermediary row-set sizes as small as possible throughout query execution.

   Pick a join sequence that is likely to remove the highest percentage of rows from consideration as soon as possible. This is what FILTERED ROWS PERCENTAGE method will attempt to do (LEAST PERCENTAGE OF ROWS).

3. FILTERED ROWS PERCENTAGE technique

   a.) Filtered Rows Percentage Method (FRP) helps us understand several crucial aspects about a query:

   - Determine Driving Table and Join Order.
   - Determine basic query style (PRECISION QUERY or WAREHOUSE QUERY)
   - Determine if the query got off to a good start (Stats and Cardinality Accuracy)

   b.)The process of FILTERED ROWS PERCENTAGE METHOD is as follows:

   1. Format the PROBLEM QUERY
   2. Familiarize yourself with the problem query
   3. Create a spreadsheet
   4. List tables from the query in the spreadsheet
   5. Build COUNT QUERIES and note the row count for each table
   6. Build and run FILTER QUERIES for each table
   7. Note the filtered row counts
   8. Compute FILTERED ROWS PERCENTAGE for each table
      - ROUND ((1-Y/X)*100, 0).
   9. Determine PREFERED JOIN ORDER
   10. Construct a QUERY DIAGRAM

       To construct a query diagram, follow these steps:

       - List tables in a straight line across the page, with even space between them, in the top down order as seen in the FROM clause
       - Examine the WHERE clause for joins between tables and draw lines to connect the tables that are joined
       - If desired, examine the WHERE clause for constant tests against tables and draw a vertical line from a filtered table down and label it with (c)
       - If desired, note outer-joins using (+) syntax
       - If desired, annotate the diagram with spreadsheet data (PREFERRED JOIN ORDER, ROWS, FILTERED ROWS, FILTERED ROWS%, and whatever

makes sense to you)
- ▪ Use the alias for a table name on the query diagram if you need to save space or to eliminate ambiguity.

4. Determine INITIAL JOIN ORDER
   a) Build and run RECONSTRUCTION QUERIES
      - ORDERED hint is used to ensure Oracle accesses tables in the order we want so that we can explicitly determine DRIVING TABLE and JOIN ORDER. In 11g we could also use LEADING hint
      - COUNT (*) is used to get the row count after the join to the new table and thus gives us an indication of workload for the new table. The row count will be useful in learning if a better join order might exist than the INITIAL JOIN ORDER we are currently using.
      - Example: SELECT /*+ ORDERED */ Count(*) ROWCOUNT FROM att_emp_org, cbe_emp
   b) Note reconstruction row counts
      - Two additional columns are added to the FRP spreadsheet: JOIN ROWS and ELAPSED SECONDS.
   b) Use CARDINALITY FEEDBACK to adjust join order
      - CARDINALITY FEEDBACK is the use of the count of rows retained after a join (our JOIN ROWS column) to determine if a query plan should be changed. For our purposes in FRP, we are interested in knowing if row counts go down as we progress, and if so, if there is a different join order that would allow us to get to the reduced number of rows faster.
5. Repeat (4) if join order changed
6. Determine if further action is necessary

Ways to use a Query Explain Plan

Some things you can learn from a QEP:
- Identifying where the CBO thinks it will be spending most of its time and resources
- Locating mistakes in Cardinality Estimates

a. perform table count:query: select count(*) from the_table;

b. perform table filter query from Predicate Information section: select count(*) from the_table where …..;

c. skewed data: select count(*)/count(distinct(column_value)) average_rows_per_value from the_table;

Correct using histograms
- ○ Observing how Oracle has Modified your Query
- ○ Evaluating Efficiency in Fetching Rows
- ○ Evaluating Efficiency in Joining Rows
- ○ Comparing Estimates to Actuals

d. run the query using the GATHER_PLAN_STATISTICS hint.
- Viewing text of REMOTE queries in distributed transactions
- Learning about Database Features from the OUTLINE

e. How to do an Explain Plan
- Explain Plan for Select ….. from …....
- select * from table(dbms_xplan.display('PLAN_TABLE',NULL,'ADVANCED'));
- select * from table(dbms_xplan.display('PLAN_TABLE',NULL,'ADVANCED

-projection -outline -alias'));

- Use hint: select /*+ gather_plan_statistics */ ….. from …..;
  - select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));

Best Indexes for a Query

1. Poor use of Data Types
   - use proper data types. Avoid conversion(ex. Varchar2 to date, or varchar2 to number)
2. Effect of Functions on Indexing
   - function based indexes are only good for the expressions they contain.
3. Where clause organization

The three reasons a column belongs in an index given a specific query being tuned:

- ACCESS means to fetch entries from an index.
- FILTER (sometimes referred to as pre-table filtering) means to throw away index entries after we fetch them because we now realize we do not want them.
- COVERAGE means to get everything our query needs from only the index so that we do not have to visit the underlying table.

Consider that:

- Equality predicates facilitate ACCESS to index entries.
- Only one inequality predicate can in general be used for ACCESS to index entries.
- NON-ACCESS predicates, including remaining inequality predicates, and columns modified by functions, can provide columns suitable for indexes for purposes of FILTER.

Creating Indexes for Join Queries

- Joins need new handling when defining indexes for a specific query. Since a join is between two tables, one of these tables will feed values into a join, and the other table will be the target of some type of lookup operation. The table supplying values is the OUTER table of the join and the table into which we look using these values is the INNER table of the join. A table's role as either OUTER or INNER in the join determines the usefulness of columns for indexing purposes on that table.
- Columns from JOIN PREDICATES cannot be used for ACCESS or FILTER on the outer table of a join because the values for these columns are unknown at the time an index is used on the OUTER table. But for the INNER table of a join, the columns from JOIN PREDICATES are known because rows from the outer table are providing values for these columns at the time we are accessing the index on the inner table.
- CONSTANT TEST PREDICATES dictate the first set of columns in an index and the JOIN PREDICATES dictate the later columns.

So far we have seen that columns can be added to indexes either:

- to support ACCESS in finding a set of index entries
- to support FILTER after index entries are retrieved but before the base table is accessed

Additionally we have noted that:

- equality predicates support ACCESS to the index

- inequality predicates can support ACCESS to the index BUT… we are allowed only one inequality predicate applied to the index
- all columns in an index after an inequality predicate is applied CANNOT support ACCESS to the index even if the predicate used is equality though they may support FILTER
- knowing the choice of OUTER table and INNER table in the join is essential for building the right set of indexes for a query because it affects which indexes can use the join columns
- with respect to a specific query, unused columns in an index stop ACCESS during an index lookup
- with respect to a specific query, columns after an unused column in an index will at best be used for FILTER (pre-table filtering)

Covering Indexes

- A covering index is an index which contains all the columns needed by a query on the indexed table.

Limitations of Covering Indexes:

- For a COVERING INDEX to work, I/O savings must be significant. I/O savings will usually come from scanning fewer index blocks as compared to table blocks in cases where the covering index replaces the table in a scan operation, or by avoiding the ROWID access to table blocks that comes at the end of an index lookup.
- Space: COVERING INDEX must be smaller than the table it covers. As a general rule it should be at most one third the size and preferably much smaller. The greater the difference between the size of the covering index and the size of the associated table the more benefit we get because the whole idea of COVERAGE is to skip I/O related to table visits.
- Usability. COVERAGE is a rather specialized indexing practice. A covering index is very sensitive to the columns listed in the query. If you were to modify the query and add an additional column, you would lose the benefit provided by the additional columns at the end of the index because the existence of just one column in the query but not in the index means you have to go back to the table and that means that the covering index stops being a covering index and starts being a regular index with lots of extra useless baggage on the end.
- Benefit: Normally when we do tuning we are looking for an order of magnitude change or better. COVERING INDEXES cannot perform this kind of feat. Instead they will give you only incremental improvements. You will generally see benefit up to about 50% reduction in runtime. Things going twice as fast are nice but not really game changing.
- 3 scenarios in which Covering Indexes are useful:
  - You have an OLTP application where online users are waiting for results and there is a high desire to reduce the wait time as much as possible. In this situation COVERING INDEXES can make it possible to cut most of the time out of an online application. But you generally need to use a lot of memory and fix these indexes in a KEEP POOL somewhere.
  - You have a query doing a FULL TABLE SCAN on a very wide table and only a few columns are needed and the table scan comprises a relatively large portion of the overall query time, then you can substitute a COVERING INDEX for the table and do an INDEX FAST FULL SCAN instead in order to remove most of this time.

  - You have a query which you have tuned and removed most of the waste and now you are down to the point where accessing tables is taking the most time.

Three reasons for putting a column in an index:

- To support predicates doing ACCESS to index entries on the DRIVING TABLE of a query or INNER TABLE OF A JOIN.
- To support FILTER predicates for pre-table filtering of retrieved index entries before we visit the underlying table.
- To support COVERAGE by creating a COVERING INDEX so that we can skip going to the underlying table all together.

Create Index Syntax:

DRIVING TABLE SYNTAX

create index <INDEX_NAME>
on <TABLE_NAME>
(

1. For ACCESS: columns on the DRIVING TABLE seen in equality predicates against constant values ,
2. For ACCESS: column (only one) on the DRIVING TABLE seen in an inequality predicate against a constant (>,<,between,like) ,
3. For FILTER: all other columns on the DRIVING TABLE from any predicate against constant values
4. For COVERAGE: all other columns on the DRIVING TABLE seen anywhere in the query);

INNER TABLE OF A JOIN SYNTAX

create index <INDEX_NAME>
on <TABLE_NAME>
(

1. For ACCESS: columns on the INNER TABLE of a join seen in equality predicates against constant values ,
2. For ACCESS: columns on the INNER TABLE of a join seen in equality predicates against columns from the OUTER TABLE of the join ,
3. For ACCESS: column (only one) on the INNER TABLE of a join, seen in an inequality predicate (>, <, between, like) ,
4. For FILTER: columns on the INNER TABLE of a join from any other predicate not used for ACCESS ,
5. For COVERAGE: all other columns on the INNER TABLE of a join seen anywhere in the query);

The Indexing Process

Process of building indexes for a specific query scenario.

1. Determine the DRIVING TABLE and JOIN ORDER for the query.

For each table in the query:

2. Reduce the query down to only those elements related to the table. We call this the REDUCED QUERY.
3. Select the appropriate variation of our own CREATE INDEX command syntax (DRIVING TABLE syntax or INNER TABLE OF A JOIN syntax). See create index

syntax.

4. Walk the REDUCED QUERY for each rule in the selected command syntax and pop columns into the index as they satisfy a rule.

POST-TABLE FILTERING

Every time we create an index we will inevitably use the concept of Access, Filtering or Post-table filtering.

- ACCESS is most efficient as it avoids forms of waste that occur where we fetch index entries and rows we do not want.

- FILTER is less efficient than ACCESS but is more efficient than POST-TABLE FILTERING. If we see FILTER after an index usage, this may indicate that the index usage is less selective than might be possible for the query's associated predicates. Adding additional columns to the index might help, along with possible function based indexes or column re-ordering. The result should be to remove the FILTER operation and have only ACCESS utilizing all related predicates in the query. Less ideal but often more realistic would be the changing of some though not all FILTER predicates into ACCESS predicates.

- POST-TABLE FILTERING is the worst of the bunch. It means we had to fetch index entries we did not want and then also fetch rows we did not want. This is wasted effort. There may be a possibility through index manipulation or query changes, to remove POST-TABLE FILTERING and turn it into either FILTER on the index or ACCESS on the index.

Joins

Important Join Points

- The 2% RULE. If your query is fetching more than 2% of the rows in a table, you should seriously be considering a Full Table Scan to get these rows and not an Index Lookup.
- There are only two types of queries, PRECISION style queries, (that uses NESTED LOOPS JOIN supported by INDEX LOOKUP to do its work) and WAREHOUSE style queries that uses HASH JOIN supported by TABLE SCAN as the way it does its work.
- Each type of join has a set of conditions in which it performs best.
- HASH JOINS are greedy for memory which means when tuning HASH JOINS, PGA_AGGREGATE_TARGET is your enemy since you will want (and may have to take) more memory than you are allowed when sharing memory with other queries.
- The single most common problem with HASH JOINS is that we did NOT use one when we should have. Bad CARDINALITY estimates often result in a NESTED LOOPS JOIN when a HASH JOIN should have been chosen.
- HASH JOIN is tuned by reducing NUMBER OF PASSES which means reducing its use of TEMP SPACE. But if your HASH JOIN is already OPTIMAL or ONE-PASS then there is not much left to tune so do not expect massive improvements.
- PARALLEL QUERY and PARTITIONING can help a HASH JOIN ultimately because they break large joins down into smaller joins which reduces the memory footprint of individual join pieces. But when not pre-planned, these are tuning methods of

desperation and may not be so easy to deploy and may offer only marginal benefit (2X-3X faster) not game changing improvements (10X faster).

Join Characteristics:

- Setup Cost (any work needed before the first joined row is produced)
- Join Cost (cost of the specific join algorithm used to create a joined row)
- Resource Requirements (need for memory or temp space or additional data structures)
- Sweet Spot (what conditions are ideal for each join type)

Join Conditions that provide best performance:

- For SORT MERGE JOIN there are two sweet spots.
    - The first is when we are doing a NON-EQUI-JOIN on a large number of rows and there is no supporting index on the destination side of the join (aka. inner table of the join). In this scenario, a HASH JOIN is not possible since this only works with EQUI-JOIN predicates, and a NESTED LOOPS JOIN would have to do a table scan on the inner table each time through the loop which would make NESTED LOOPS JOIN expensive.
    - The second sweet spot is when one or both of the sort steps can be skipped which can happen either as a fluke of previous steps in the query plan which happen to present the data in sorted order, or when an index can be used that pulls data in the order that the merge step needs and Oracle can rearrange the query plan to exploit this index at the right time.
- For NESTED LOOPS JOIN the sweet spot is when the outer table of the join is sending a "small" number of rows into the join operation and the inner table of the join has a supporting index which is an index where the join columns are part of the leading columns in the index. Small is an ambiguous term. Let me go out on a limb and suggest to you the 2% RULE. If the driving table of the join is feeding less than 2% of its rows into the join, then assuming an even distribution of the data, it is reasonable to believe we will be fetching less than 2% of the rows from the inner table in the join.
- For HASH JOIN, the sweet spot is when we have an EQUI-JOIN (most joins are an EQUI-JOIN) and we are joining a "large" number of rows on both sides of the join thus making use of an index and the associated single block fetching of rows that comes with index lookups, an expensive idea. If we use the 2% rule then large is whenever the outer table is feeding in more than 2% of its rows into the join.

Reading Query Explain Plan

- NESTED LOOPS JOIN : When looking at a NESTED LOOPS JOIN, the first table to be referenced under the NESTED LOOPS key word is the outer table and the second is the inner table of the join.
    - The sweet spot for the NESTED LOOPS JOIN we said was when the OUTER table in the join supplies a small number of rows (<2% of its total row count if we are using the 2% RULE) to the join.
    - This description of the sweet spot for NESTED LOOPS JOIN requires that two things be true.

        - First, that the number of rows provided by the OUTER table in the join be small.
        - Second that the lookup into the INNER table is supported by an index.
        - It follows easily then that a failure in one of these two needs results in a NESTED LOOPS JOIN that goes horribly wrong.
    - So look for these two scenarios when you see NESTED LOOPS JOIN in a slow query. Look for cardinality estimates that substantially under report the number of rows coming from the OUTER table of the NESTED LOOPS JOIN (one (1) is very common and often wrong). Look for non-selective indexes used to support access to the INNER table of the NESTED LOOPS JOIN. Incomplete set of leading columns is common, as is poor statistics which produce low cardinality estimates and thus possibly wrong index choice.

- HASH JOIN: When looking at a HASH JOIN, the first table referenced after the HASH JOIN step is the inner table and the second is the outer table of the join.
    - The two situations where HASH JOIN is not meant to be used are
        - 1) if you are doing a NON-EQUI-JOIN or
        - 2) if you are executing a true OLTP query that requires sub-second response time.
    - HASH JOIN by its nature requires an EQUI-JOIN. An EQUI-JOIN is of course an equals across columns (ex. AND T2.FK = T1.PK)
    - For now, true OLTP and sub-second response times remain the domain of NESTED LOOPS JOIN.
    - Two Most Common Hash Join Failures
        - First is a low cardinality estimate that causes a NESTED LOOPS JOIN to be used instead of a HASH JOIN
        - Second is a partial join as we saw in NESTED LOOPS JOIN. With hash join it involves mixed operations of EQUI-JOIN and NON-EQUI-JOIN to the same table which results in the same problem, a partial join, just without the index. (ex. where t1.c1 = t2.c1 and t1.event_date between t2.start_date and t2.end_date;)
            - Three ways to fix a hash join across a partial key.
                - Change in Join Method
                    - try to get the optimizer to do a SORT MERGE JOIN and hope it uses the inequality which is why SORT MERGE was created.. (ex select /*+ use_merge (t1,t2) */ ...)
                - Nearest Neighbor Lookup
                - Database Design Change

Hash Join Memory Management

- Avoiding the use of TEMPORARY STORAGE is the key to performance of a HASH JOIN and this is done using memory. The way a HASH JOIN works is that it takes one of the two tables it is joining (usually the smaller one) and hashes it into memory.

- What happens when the INNER table of a HASH JOIN does not fit into memory?
  - The first important lesson here is that when the INNER table fits into memory we have an OPTIMAL HASH JOIN which requires the minimum amount of work to do the HASH JOIN. But when the INNER table does not fit into memory, the workload of the HASH JOIN increases. Thus the most obvious way we have of tuning a HASH JOIN is to do our best to make sure we can fit as much of the INNER table as possible into memory. Normally the only thing we can do is allocate more memory to the HASH JOIN
- When a Hash Join hashes a table the memory used is called a WORKAREA and these work ares are visible in GV$SQL_WORKAREA_ACTIVE. The view also shows the TEMPSPACE used when table does not fit into memory.

Ways to make a hash join go faster:

- Exploit the DUNSEL JOIN (It is a join which serves no useful purpose) feature and drop the join altogether.
- Reduce the amount of data hashed in memory.
- Increase memory allocated for hash joins.
- Use Partitioning and Parallel Query to turn big hash joins into lots of small hash joins.
- Use techniques to eliminate DISTRIBUTION in a parallel hash join.

Types of Hash Joins

- FULL Partition Wise Hash Join
  - Criteria for Full Partition Wise Hash Join.
    - First both tables being joined must already be partitioned using the same partitioning criteria. This means each table you are trying to join must be broken down into the same number of partitions with the same range of values in each partition. This is called EQUI-PARTITIONING. If partitioning is not identical, you are toast.
    - Second this partitioning criteria must be based on the columns of your hash join. This means that the partitioning must be based on the join columns in your query.
    - Third, column data types must be compatible across the join. In general this mean same base data type (number, string, date).
  - Recap of partition wise hash join.
    - DYNAMIC SAMPLING can change query plans in unexpected ways. Sometimes it can cause you to get or not get a query plan with the features you are trying to exploit. This is not necessarily bad, as the purpose for dynamic sampling is to allow the database to do this kind of thinking for us. But it can cause confusion during testing or troubleshooting.
    - Joins that are compatible with the partitioning scheme of both tables, and compatible data type definitions are required in order to get a full partition wise hash join.
    - A full partition wise hash join can be identified by the placement of the PARTITION step. It will appear above the hash join step, not below it.

- PARTIAL Partition Wise Hash Join
  - A PARTIAL partition wise hash join is similar to a FULL partition wise hash join except that only one of the two tables is partitioned on the join columns. At query execution time the join process will dynamically partition the table that does not have the needed partitioning. This table can be an un-partitioned table, or a table partitioned using a partitioning that does not match the join columns.
  - Thus the savings gained by a PARTIAL partition wise hash join comes from not having to write/read ONE of the tables of the join in order to obtain the segment pairing needed to do the join.
- There are five ways to speed up a hash join:
  - Do not do the join at all. Use constraints to enable DUNSEL JOIN removal.
  - Reduce the size of the join. Remove unneeded columns from the query.
  - Improve the efficiency of the join. Increase memory available to the join.
  - Eliminate pre-work in the join. Use Partitioning and Partition Wise Hash Joins.
  - Throw more money at it. Use Parallel Query.

HINTS

The three hints we need depending upon database version:

- CARDINALITY/ OPT_ESTIMATE

  a. Cardinality (<table_name | alias> <rowcount>). ex. SELECT /*+ cardinality(clm 76183) cardinality(calc 78680) cardinality(ps 143) cardinality (ch 76642) */. Add correct cardinality numbers to see if Query Plan changes.
  b. OPT_ESTIMATE hint is the replacement for Cardinality. Optimizer will notice table constraints when using opt_estimate hint.
  - /*+ OPT_ESTIMATE([query block] operation_type identifier adjustment) */
  - OPT_ESTIMATE (TABLE <table> rows=<filtered rowcount>) OPT_ESTIMATE (JOIN (<table>,<table>) rows=<filtered rowcount>)
    - TABLE – lets us tell the optimizer the filtered row count from table fetch
    - JOIN - lets us tell the optimizer how many rows will return from a join
- ORDERED/ LEADING
  - ▪ LEADING hint determines which tables will be accessed first in a query and in what order they will be joined, but it does not require that all tables be named.. ex. select /*+ LEADING (b) */ * from a,b,c;
    - ex. select /*+ LEADING (C B)*/ * from a,b,c;
    - Leading hint can be used to specify a DRIVING TABLE.

  - ▪ ORDERED (before 10g) hint tells the optimizer to join row sources in the order in which they appear in the FROM clause. ex. select /*+ ORDERED */ * from b,c,a;
  - ▪ When the tables are rotated in the FROM clause, the JOIN ORDER changes because of the ORDERED hint.
- NO_INDEX.

    - If it is suspected that an index was used when it shouldn't have, we can check things by validating cardinalities in plan steps and induce FULL TABLE SCANS using the NO_INDEX hint
    - Common reasons why the wrong access method and join method may have been selected.
        - Statistics Issues (stale, out-of-bounds, dependence)
        - Predicate Issues (columns modified with a function, implicit data type conversion)
        - Index Issues (column ordering, wrong columns, missing columns, poor selectivity)
    - /*+ NO_INDEX (<table>) */
        - ex. select /*+ NO_INDEX (a) */ * from a;
- Additional valuable hints.
    - NO_PARALLEL
        - Turns off parallel query for the entire query. The SERIAL version of a query plan is much easier to read than the PARALLEL version of the same plan.
    - OPTIZIMZER_FEATURES_ENABLE
        - Very useful for migrations. It lets us specify the version of the database we want to emulate when the optimizer generates query plans. This hint will allow us to check to see if the change in query performance was possibly caused by a change in optimizer features.
    - GATHER_PLAN_STATISTICS (from 10g on)
        - It requires a two-step process. The problem query must first be executed using the hint.
            - select /*+ GATHER_PLAN_STATISTICS */ <your query here>;
            - select * from table(dbms_xplan.display_cursor(null,null,'ALLSTATS LAST'));
        - Creates additional overhead which may be significant. It gives the actual cardinality of plan steps.
        - It does not work that well with PARALLEL QUERY so you will likely want to turn off PARALLEL QUERY for the query.
    - DYNAMIC_SAMPLING
        - Two variations. One at the query level and one at the table level
            - /*+ DYNAMIC_SAMPLING (<level>) */
            - /*+ DYNAMIC_SAMPLING (<table> <level) */
                - Level 0 Do not user dynamic sampling 0
                - Level 1 (1 * default (32)) 32
                - Level 2 (2 * level 1) 64
                - Level 3 (2 * level 2) 128
                - Level 10   ALL
            - TABLE LEVEL sampling: ex. /*+dynamic_sampling (emp 2) dynamic_sampling (dept 4)*/ . This query tells the optimizer via hints that it must dynamically sample table EMP looking at 64 blocks of the table and that it must sample table DEPT looking at 256 blocks of the table.

        - Use this hint to increase the number of blocks sampled in situations where dynamic sampling is not getting a sufficiently satisfactory sample. How do you know that dynamic sampling is not getting a good sample? You don't.
        - Use this hint to force 100% sampling of table.
        - Turn off dynamic sampling bu using Level=0
    - QUERY LEVEL sampling:
        - ALTER SYSTEM SET OPTIMIZER_DYNAMIC_SAMPLING=2;
        - ALTER SESSION SET OPTIMIZER_DYNAMIC_SAMPLING=3;
        - SELECT /*+ DYNAMIC_SAMPLING(4) */ * FROM EMP;
        - Levels in QUERY LEVEL hint map to list of conditions and not just blocks.
            - Level 0 do not use dynamic sampling 0
            - Level 1 32 blocks
            - Level 2 - Apply dynamic sampling to all unanalyzed tables - 64 blocks (level 2 is the default)
            - Level 3
                - Sample any table for which:
                    - The table meets Level 2 criteria.
                    - Or selectivity estimation used a guess for one or more predicates that are a potential dynamic sampling predicate.
                    - For analyzed tables the number of sampled blocks is the default(32). For unanalyzed tables the number of sampled blocks is two times the default(ie. 64).