Performance tuning: practical tips & hints - common mistakes & pitfalls

Joakim Treugut

Chief Oracle Investigator

Synergy International Ltd

Abstract

Oracle's Cost-Based Optimizer (CBO) gets more and more intelligent and complex. How does CBO find the optimal execution plan? Did you think that the final execution plan was determined during the parse phase? What features are new in 9i? What are the common mistakes I see when I visit customers to optimise their systems? By understanding how CBO works, what affects CBO, and how it makes its decisions, we can with simple changes, gain huge improvements.

Introduction

This workshop is about common mistakes that I often see when I visit customers, and what developers and DBAs can do about this. But also what Oracle's Cost-Based Optimizer, CBO, does to cover up. We will also look at how CBO works and make its decisions.

What do I want you to remember

- You can write the SQL in many different ways and still get the same result – try it!
- Make it easier for CBO: use NOT NULL, CHECK, FOREIGN KEY and UNIQUE constraint.
- Rebuild tables and sort the rows on the most used indexed column, to keep the clustering factor low.
- Rebuild indexes (specially those that are targets for updates), to decrease the number of branch levels and leaf blocks.
- Identify indexes that are not used and drop them.
- Compress large indexes that are not unique.
- Analyse a table after major changes have been applied. Don't waste time and resources to analyse tables that don't change. Use the table monitoring feature to collect modification information.
- Create histograms on columns with skew distribution and that are used in the WHERE clause (even if the column is not indexed).
- Use the ALTER SESSION SET command to let different users/jobs have different parameter settings.
- Don't do a lot of work that is later filtered away. Either by accessing heaps of ROWIDs from an index that are filtered away when the table row is read, or building a big row source that is greatly reduced when joined with the next row source. Try to filter away unneeded rows as soon as possible.
- If CBO doesn't choose the plan that you think is best, it is because your plan has a higher cost than the chosen one – investigate why.

The SQL language

SQL is a simple and powerful language. It is called a non-procedural language because you tell it what to do, but no how to do it. The less semantic and abstract a programming language is, the more of the optimisation is left for the developer. When you write code in assembler you can't hope that someone else will fill in the missing pieces. You are the one who needs to write what should be done. SQL is a very high semantic language. The language was called in the beginning SEQUEL and was an abbreviation for Structured English QUEry Language. SQL is easy to learn, so you don't need to be an educated developer to fetch 2 gigabyte of data from a join with missing join predicates.

You can write a query that returns the same result in many different ways. IN or EXISTS? NOT IN or NOT EXISTS (are not always the same thing)? Join or subquery?

So the developer or user decides **what** to get, and the DBA/sysadm decides **where** it is.

Common mistakes

In this section I will show common mistakes, which Oracle features exist to compensate for these mistakes and what do you need to do manually.

Column is not indexed or the index is disabled

If the column is not indexed, then only a full-table scan is available:

```
SELECT * FROM emp WHERE sal = 5000;
```

CBO often chooses to access a table with a *full-table scan*, if the WHERE clause doesn't exist. But if all the columns in the column list are declared NOT NULL and are included in one index, then CBO also calculates what it costs to access the data with an *index full scan* and an *index fast full scan*. If all the columns are declared NOT NULL but are included in many indexes, then CBO calculates what an *index join* will cost. If the column is declared as nullable and is indexed, then CBO calculates what it will cost to access the table via an *index full scan* on other indexes that has at least one column declared NOT NULL. But these costs are always higher than a full-table scan cost.

```
SELECT ename FROM emp;
```

If ENAME is declared as nullable and indexed, then the same result (CBO calculates the cost between full-table scan, index full scan and index fast full scan) can be achieved by using the IS NOT NULL condition:

```
SELECT ename FROM emp WHERE ename IS NOT NULL;
```

When Oracle finds that the user tries to compare columns or values of different data types, Oracle does an implicit data type conversion (it prefers to compare NUMBERS and then DATE, so a character column will always get a TO_DATE or TO_NUMBER function applied). Comparing a NUMBER column with a string of numbers is no problem, and any existing index can still be used. But when we compare strings with numbers, then Oracle implicitly applies TO_NUMBER on the character column, and by doing that disables any existing index on that column.

```
WHERE char_col = 100;          =>     WHERE TO_NUMBER( char_col ) = 100;
```

The next one is a tricky one. Oracle will still choose a full-table scan, but if you use AUTOTRACE or EXPLAIN PLAN (directly in SQL*Plus or from tkprof), Oracle will show that an index lookup was done. This is because EXPLAIN PLAN only parses the query (it doesn't take the real execution plan, as can be seen in the STAT lines in a raw trace file) and assumes that the bind variable has the correct data type. *Reference: Problems with bind variables – how to find the datatype of a bind variable, Joakim Treugut, Synergy International Limited.*

```
WHERE char_col = :bind_var_declared_as_number;
```

Sometimes I encounter this problem in customers' systems. A join between two tables and the two join columns have different data types. In this case, Oracle cannot read table S first and then with nested loops look up the rows in R with the index on CHAR_COL. Oracle is forced to either read R first and then S, or read the CHAR_COL index with an *index fast full scan* and do a hash join (or sort/merge join).

WHERE r.char_col = s.num_col;

For a statement like this we can now use Function-Based Index:

WHERE num_col + 0 = 100;

WHERE UPPER(ename) = 'KING'

This statement, WHERE TRUNC(hiredate) = TRUNC(SYSDATE), can also use a Function-Based Index or be rewritten like:

WHERE hiredate >= TRUNC(SYSDATE) AND hiredate < TRUNC(SYSDATE+1)

When you have a function on a column, you disable CBO's chance to estimate the selectivity, cardinality and cost. CBO will apply the default selectivity to the predicate, 1%, and from then on, calculate cardinality and cost. So if you have a huge history table with ten years of data, and on average you save 100 rows per day, then the selectivity and cardinality for TRUNC(hiredate) will be 0.01 (default) and 3650 (10*365*100/100), when the real cardinality is 100.

If the column in the WHERE clause is not the leading column in the index, then that index was disabled in earlier releases, but in 9.0.1, Oracle introduced *index skip scan*. Here we have an index on (country_id, cust_city). If I had created the index with cust_city (most selective column of them two) first, then CBO chooses a full-table scan, because CBO calculates the cost to be number of branch levels in the index (blevel) + number of distinct values for the skipped/leading column (num_distinct). So the cost for an index skip scan is lower if the less selective column is the leading one. For a long time it has been said that if you have a composite index on two columns, and both columns are used equally often, then you should choose to have the most selective column first. Oracle's features "index skip scan" prefer to have the least selective column first.

SELECT * FROM sh.customers WHERE cust_city = 'Wellington';

Execution plan

SELECT STATEMENT Optimizer=CHOOSE (Cost=67 Card=81 Bytes=11583)
  TABLE ACCESS (BY INDEX ROWID) OF 'CUSTOMERS' (Cost=67 Card=81 Bytes=11583)
    INDEX **(SKIP SCAN)** OF 'IX_COUNTRY_CITY' (NON-UNIQUE) (Cost=21 Card=81)

Late filtering – index is not restrictive enough

There are many good reasons to filter away rows as early as possible: the row source becomes smaller, no wasted job is done, and joins become more efficient.

This SELECT accesses the index with the predicate CID = 102 and fetches 43715 ROWIDs, it then looks up the row in the table and checks if "OP = 'PARSE'. For 43714 rows this was false and for one row it was true. Something like this is doubly meaningless: first to read 43714 rows that we don't need and then filter them away. An index on both columns would in this case be much better.

```
SELECT *
  FROM tt$10046_op
 WHERE cid = 102
   AND op = 'PARSE';

COUNT OPERATION
    1 TABLE ACCESS (TT$10046_CURSOR) BY INDEX ROWID
43715   INDEX RANGE SCAN (IX_TEST)
```

If the wrong table is accessed first, then a much bigger row source is built, which is later filtered away

```
SELECT /*+ ORDERED */ COUNT( * )
  FROM parent, p
 WHERE p.pk = parent.pk;
```

```
COUNT OPERATION
    1 SORT( AGGREGATE )
    6   NESTED LOOPS
 1000     INDEX( FAST FULL SCAN ) OF 'PK_PARENT'
    6     INDEX( UNIQUE SCAN ) OF 'PK_P'
```

If the tables are accessed in the reverse order (with the table first that creates the smallest row source),

```
SELECT /*+ ORDERED */ COUNT( * )
  FROM p, parent
 WHERE parent.pk = p.pk;

COUNT OPERATION
    1 SORT( AGGREGATE )
    6   NESTED LOOPS
    6     INDEX( FAST FULL SCAN ) OF 'PK_PARENT'
    6     INDEX( UNIQUE SCAN ) OF 'PK_P'
```

Reference: Late Throwaway, Martin Berg, Oracle

Statistics are missing or not accurate

In earlier releases, CBO was very dependent on accurate statistics, the settings on some of the parameters, and the syntax. Nowadays, the developers don't need to think so much about the syntax they use (if they choose IN or EXISTS, NOT IN or NOT EXISTS, join or subquery, as long as the DBA does a good job with NOT NULL, UNIQUE and the other constraints), Now CBO is smart enough to handle most of the differences. But CBO still needs accurate statistics, and here Oracle has done a lot.

- Dynamic sampling of statistics (9.2). When optimizer_features_enable is set to 9.2.0, optimizer_dynamic_sampling gets the value 1.Which means (taken from the Tuning 9.2 guide): *Sample all unanalyzed tables if the following criteria are met: (1) there is at least 1 unanalyzed table in the query; (2) this unanalyzed table is joined to another table or appears in a subquery or non-mergeable view; (3) this unanalyzed table has no indexes; (4) this unanalyzed table has more blocks than the number of blocks that would be used for dynamic sampling of this table. The number of blocks sampled is the default number of dynamic sampling blocks (32).* In a raw trace file you will see this recursive SELECT appears:
  ```
  SELECT /*+ ALL_ROWS IGNORE_WHERE_CLAUSE */
         NVL(SUM(C1),0), NVL(SUM(C2),0), COUNT(DISTINCT C3)
    FROM (SELECT /*+ NOPARALLEL("C") */
                 1 AS C1, 1 AS C2, "C"."UNN" AS C3
            FROM "C" SAMPLE BLOCK (3.283898) "C"
         ) SAMPLESUB
  ```
- Auto gather stale statistics (table monitoring). I think this feature came in 8.1.7. When a table has the option monitoring turned on, SMON updates MON_MODS$ (externalised by dba_tab_modifications) every second hour (and at clean shutdown) with statistics about number of inserts, updates, deletes and truncates. After that you can call DBMS_STATS.GATHER_SCHEMA_STATS with the OPTION=>'GATHER STALE', and Oracle will then only gather statistics for tables that have more than 10% (I haven't really test this) changes on the table since the last analysed. But a DBA can also use this information to fine-tune how (s)he wants to call DBMS_STATS, when and with which options.

- Reading the extent map in the segment header to get the high-water mark (and from that value calculate better selectivity and cardinality). Instead of assuming that every table is 100 blocks big and that the average row length is 100 bytes, CBO now reads the extent map to find out how many blocks are below high-water mark, and then estimate the number of rows to be:
  ROUND(<blocks below HWM> * (db_block_size – 24) / <avg_row_len=100 >)

In one case, a developer couldn't understand why a specific index was not chosen for a table (the fourth table in the join order of five), the problem was that the second table had bad statistics on number of rows and number of distinct values for the column, so the problem started much earlier than the developer could see by just looking at how a specific table was accessed.

The column needs histograms

Histograms is something we should use when a column has skew distribution, is used in the WHERE clause, and is not compared with a bind variable. When CBO looks at a predicate without histogram, like WHERE job='PRESIDENT', then CBO assumes that the selectivity for that predicate will be 1 / num_distinct * (num_rows-num_nulls) / num_rows (if a table has 1000 rows, and the column has no null values and ten distinct values, then the selectivity becomes 0.1). The computed cardinality for this predicate will be 100 rows (num_rows * selectivity). The value 100 is far away from the correct value, 1. CBO needs histograms to compute more realistic selectivity for a predicate. If the selectivity gets a non-realistic value, it will have a snowball effect on cardinality, cost and maybe also on which access method, join method and join order to choose.

With no histogram on JOB (emp consists of 680 rows and job has 7 distinct values and no nulls):

```
SQL> SELECT * FROM emp WHERE job = 'WIZARD';

665 rows selected.

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=97 Bytes=3104)
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=2 Card=97 Bytes=3104)
    INDEX (RANGE SCAN) OF 'IX_EMP_JOB' (NON-UNIQUE) (Cost=1 Card=97)


SQL> SELECT * FROM emp WHERE job = 'PRESIDENT';

1 row selected.

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=97 Bytes=3104)
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=2 Card=97 Bytes=3104)
    INDEX (RANGE SCAN) OF 'IX_EMP_JOB' (NON-UNIQUE) (Cost=1 Card=97)
```

CBO returns the same cardinality for WIZARD as for PRESIDENT, and for these queries CBO chose to do an index range scan, but when EMP contains some more rows, it will switch to a full-table scan for both queries.

When there is a histogram on JOB:

```
SQL> SELECT * FROM emp WHERE job = 'WIZARD';

665 rows selected.

Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=4 Card=665 Bytes=21280)
  TABLE ACCESS (FULL) OF 'EMP' (Cost=4 Card=665 Bytes=21280)


SQL> SELECT * FROM emp WHERE job = 'PRESIDENT';

1 row selected.
```

```
Execution Plan
----------------------------------------------------------
SELECT STATEMENT Optimizer=CHOOSE (Cost=2 Card=1 Bytes=32)
  TABLE ACCESS (BY INDEX ROWID) OF 'EMP' (Cost=2 Card=1 Bytes=32)
    INDEX (RANGE SCAN) OF 'IX_EMP_JOB' (NON-UNIQUE) (Cost=1 Card=1)
```

Here CBO chooses a full-table scan when we select 665 rows from a total of 680, and an index range scan when only one row is selected.

When we join tables, it is even more important that CBO gets accurate column statistics. Otherwise CBO can choose to read the bigger table first, determine the wrong join order, and choose a non-optimal join method and access method. In the PRESIDENT example, if the query joins EMP and DEPT with a third table, and CBO chooses to join EMP and DEPT first, without histogram, CBO estimates that 97 rows will come out from the join, so when the third table gets joined with the estimated 97 rows row source, CBO may choose a inefficient hash join where a nested loops would be more efficient. But if CBO estimates that a nested loops with an index lookup should be used, when joining the row source with the third table, if we then are looking for WIZARD, the result will be that CBO repeats that index lookup 665 times, and we will get a very inefficient nested loops, but probably a good instance buffer cache hit ratio.

Oracle offers the functionality to let DBMS_STATS decides which columns to create histograms for. When _column_tracking_level is greater than zero (default is 1), COL_USAGE$ is updated with statistics about how the column is used in the WHERE clauses. Then you can call DBMS_STATS.GATHER_TABLE_STATS with method_opt set to AUTO (Oracle looks both at the column distribution and the workload of the column) or SKEWONLY (Oracle only looks at the column distribution).

Non-optimal settings of the initialisation parameters

There are many parameters that affect CBO's decisions. The most common are:

- optimizer_features_enable
- optimizer_index_caching
- optimizer_index_cost_adj
- db_file_multiblock_read_count
- hash_join_enabled
- hash_area_size
- sort_area_size
- query_rewrite_enabled
- _complex_view_merging
- _push_join_predicate
- _unnest_subquery

Not many companies let different sessions get different settings on these parameters.

Many DBAs have been struggling with finding the optimal values for the *_area_size parameters, in 9.0, Oracle introduced SQL working memory management, so you don't need to consider about the settings of these parameters.

Indexes that contain unique values are not declared as unique

When CBO finds an index that is created with the UNIQUE option, CBO knows that a query that uses this index can only return zero or one row. That's why CBO chooses to have tables with unique condition as early as possible in the join order. I often found a lot of indexes where every key in it, is unique, but the index has not the UNIQUE option. CBO first checks if there exists a PRIMARY KEY or UNIQUE constraint on the column, and after that it checks if the index was created with the CREATE UNIQUE option, so there is a performance benefit by creating the unique index through a PRIMARY KEY/UNIQUE constraint.

In one case, a SELECT chose a horrible execution plan, and took many minutes to complete, without returning any rows. When the column had the normal non-unique index changed to a UNIQUE constraint, CBO chose to have this table first in the join order, it found that no row satisfied the column condition, and then stopped the rest of the execution (all this was done in less than one second).

Columns with no NULL values are not declared NOT NULL

There are many reasons why a column that never contains NULL values should be declared NOT NULL.

1. If the reality is that we do not store NULL values, then the data dictionary should mirror the reality as well as possible.

2. It gives CBO better understanding of the data model, and helps it to make more clever decisions. More access paths become available (depending on the query, CBO will calculate the cost by using index full scan and index fast full scan). Oracle may also introduce new features that are dependent and only available for columns with NOT NULL constraints.

3. If the query contains an ORDER BY clause on a NOT NULL indexed column, then CBO compares the original plan which uses a SORT operation with what it would cost to access the table with an index full scan (because the index is already sorted and an index full scan reads the index block in order, the SORT operation can be skipped). SELECT ename FROM emp ORDER BY empno;

4. CBO cannot unnest NOT IN subqueries and use the ANTI-JOIN driver, unless both join columns are declared NOT NULL or both are using the IS NOT NULL condition.

SELECT COUNT(ix_char_200_bytes_col) FROM big_table;

If ix_col is declared NULL then only a full-table scan can be used, but if the column is declared NOT NULL, then Oracle can choose between every index for that table that has at least one column (and it doesn't even need to be ix_col) declared NOT NULL, and choose the one with the cheapest cost (which should be the smallest index). So in the example above, CBO probably uses the primary key index that is created on a NUMBER column.

SELECT COUNT(not_indexed_and_not_null_column) FROM big_table; Instead of doing a full-table scan, CBO can choose any index that includes at least one NOT NULL column, and do an index fast full scan.

How the application uses SQL

COST = AMOUNT * PRICE. When you buy something the total cost is dependent on how much you buy and the price, the same applies to using resources in a database system. You don't get anything for free. Even when the SQL statement is already found in the library cache and all the data blocks are already in memory, some resources are still used. So everything has a price. Tuning is very often focused on decreasing the price – make it cheaper (=faster). Optimisation also focuses on AMOUNT. How often are we doing it? Why are we doing it? If it doesn't need to be done at all, then the AMOUNT becomes 0, and even if the PRICE is mega huge, the COST will still be zero. There are many jobs that run more frequently than needed, and some jobs don't need to run at all.

I have seen examples where a customer had a very expensive DELETE statement inside the loop, while it could be outside the loop. I have seen customers having triggers that for every execution select information from v$database and repeatedly executes SELECT USER FROM dual, instead of cache the information in a PL/SQL variable.

Transitivity on join columns

If we have a WHERE condition like E.DEPTNO = 10 AND D.DEPTNO = E.DEPTNO, then CBO is smart enough to understand that also D.DEPTNO = 10. This gives CBO more alternatives on which table to access first, the join order, the join method and the access method. The more alternatives CBO has, the better chance that it chooses the optimal one.

CBO will not add transitivity to join columns. So if C.X = B.X AND B.X = A.X, then CBO doesn't add C.X = A.X (except for star transformations).

The FILTER operation

Be suspicious when you see this operation. The second (inner) query is repeated for every row that the first (outer) returns.

The order of evaluation of subqueries

CBO executes subqueries that couldn't be unnested in the order they appear in the WHERE clause. So if the second subquery filters away many more rows that the first one, then you should change the order between them.

The Cost-Based Optimizer

SQL is a very simple language. Its semantic is close to a real language. So we need someone more to help us do the job. When you are developing code in less semantic language, I said that you must to do more of the job. You need to know which files to open, which records to read, then you must allocate memory and build your hash table, join the records and finally sort the output. Only the code to sort the output will probably be much longer than if you could do the same in SQL. The easier it is for the user/developer to write the code, the more job it is for the optimizer to figure out how to do it.

Now we have come to the point where I will introduce this presentation's special guest. The SQL decides **what**, the DBA **where**, and now we need someone to decide **how**. Let me welcome Oracle's Cost-Based Optimizer.

- Query processing (PARSE)
  - View replacement
- Query Transformation
  - Simple
  - Heuristic
    - Transitive predicate generation (are not generated on join predicates)
      - Only CBO
    - Simple view merging
    - Complex view merging
      - Turned off in 8.1.5 – 8.1.7
      - Only CBO
    - Subquery flattening
    - Common subexpression elimination
    - Predicate pushdown and pullup
      - Turned off in 8.1.5 – 8.1.7
    - Group pruning for "CUBE" queries
    - Outer-join to inner-join conversion

- Only CBO

- Cost-based
  - Materialized view rewrite
  - OR-expansion
  - Star transformation
  - Predicate pushdown for outer-joined views

- Estimation
  - Selectivity
  - Cardinality
  - Cost
- Plan generation
  - Starting table (lowest computed cardinality)
  - Join order (sorted after computed cardinality)
    - Join selectivity
    - Join cardinality
  - Join method
    - Hash Join (Inner/Outer/Anti/Semi)
    - Merge Join (Inner/Outer/Anti/Semi/Cartesian)
    - Nested Loops (Inner/Outer/Anti/Semi)
  - Access method
- Dynamic runtime optimisation

Other good news in Oracle9i

- Identifying unused indexes (9.0)
- System statistics & CPU costing (9.0)
- Segment-Level Statistics (9.2)
- Runtime Row Source Statistics (9.2)
- Peek at the values of user-defined bind variables (9.0)
- tkprof shows the wait lines from a 10046, level 8 (or 12) trace (9.0)
- CONNECT BY PUMP (9.0)
- FIRST_ROWS_n optimisation (9.0)
- Bitmap join indexes (9.?)
- Shared latches???

Conclusions:

- Tips to make it easier for CBO and improve the performance.
- What have I been talking about?
- Do you still remember?

About the Author

Joakim Treugut is Synergy's chief Oracle investigator and problem solver, and a member of the Oaktable Network. He has inside knowledge of how Oracle works behind the scenes - thanks to 18 years of development and database experience. He started to work as a developer and DBA for the Stockholm Stock Exchange where he stayed for 13 years. After that he worked for Oracle Support in Sweden for 5 years. During his first year he was selected as an "Architect of Success". He soon became the specialist on performance problems and Oracle internals. He was the team leader of the Server kernel team, responsible for database performance and ORA-600/7445 errors. He was also a member of the European SQL tuning team.