A LOOK UNDER THE HOOD OF CBO: THE 10053 EVENT

Wolfgang Breitling, Centrex Consulting Corporation

Have you ever been faced with a SQL statement where you know that if only Oracle would use an index it would perform so much better? Or not use an index, or use a different driving table, or use a hash join, or, or, or … If you are a DBA then this is a rhetorical question. Generally, it is just a matter of using the appropriate hint to solve the problem. However, increasingly DBAs are faced with SQL statements that are generated dynamically by applications, where the source is inaccessible, or which can not be changed because of licensing or support restrictions. This paper will shed some light on the decisions the cost based optimizer (CBO) makes when parsing a SQL statement and choosing an access plan. It will show how the CBO calculates the cost of a plan and some of the rules and factors that go into these calculations. By extrapolation, if you know how the CBO assigned the cost of a plan, you may be able to make changes that lead the optimizer to choose a different plan – without the need to make changes to the SQL statement. While you can not compare the costs of different explain plans, the optimizer does compare the costs of different plans, and chooses the one with the lowest cost, in its current parse tree! You just don't see any of the plans which "lost out". Unless you activate the 10053 event trace, that is. There you see all the access plans the CBO evaluated and the costs assigned to them. So, lastly, this paper will act as a guide should you decide to venture into the jungle of a 10053 event trace yourself.

THE EVENT 10053

Event 10053 details the choices made by the CBO in evaluating the execution path for a query. It externalizes most of the information that the optimizer uses in generating a plan for a query.

DISCLAIMER

Oracle does not provide any documentation on the output of the 10053 event. This presentation is built on my interpretation of the results of many traces; with some jump-start help from ev10053.txt at *www.evdbt.com/library.htm*. There is no guarantee that the interpretation is accurate and there are certainly gaps in my knowledge and understanding of the trace.

HOW TO SET EVENT 10053

for your own session

A. on:
   ALTER SESSION SET EVENTS '10053 trace name context forever[, level {1|2}]'
B. off:
   ALTER SESSION SET EVENTS '10053 trace name context off'

for another session

A. on:
   SYS.DBMS_SYSTEM.SET_EV (<sid>, <serial#>, 10053, {1|2}, '')
B. off:
   SYS.DBMS_SYSTEM.SET_EV (<sid>, <serial#>, 10053, 0, '')

Unlike other events, where higher levels mean more detail, the 10053 event trace at level 2 produces less detail than the trace at level 1. Like the SQL_TRACE (a.k.a. 10046 event trace), the 10053 event trace is written to user_dump_dest. The trace is only generated if it the query is parsed by the CBO. Note that this entails two conditions: the query must be (hard) parsed and it must be parsed by the CBO. If the session, for which the 10053 trace has been enabled, is executing only SQL that is already parsed and is being reused, no trace is produced. Likewise, if the SQL statement is parsed by the rule based optimizer (RBO), the trace output will consist of the SQL query only, but none of the other information.

Q When, aside from a rule hint and optimizer_goal = rule, is a statement parsed by the RBO rather than the CBO?

A When none of the tables in the query is analyzed.
Contrary to a popular myth, the absence of statistics on a table does not mean that the RBO is being used. As long as at least one table in the query has statistics, the CBO will parse the query. We will see what it does for the missing statistics of unanalyzed tables.

Q When, despite optimizer_goal = rule or the absence of any statistics for all tables in the query, is a statement parsed by the CBO rather than the RBO?

A When a hint is used in the query. The presence of a hint – other than rule – will always force the use of the CBO.

Q How do you guarantee that a SQL statement gets parsed in order to generate a 10053 trace but avoid that it actually gets executed – which could take hours; after all that is why you were called to tune it.

A There are different techniques. One suggestion is to add an "and 0=1" predicate to the SQL. I personally do not like that. First, it does not guarantee that the statement is parsed and secondly, I would never be sure that the added predicate does not change the access plan. I simply explain the statement.

TRACE CONTENTS

The trace consists of 6 sections:

- Query
- Parameters used by the optimizer
- Base Statistical Information
- Base Table Access Cost
- General Plans
- Recosting for special features

This paper and presentation covers the first four and some information on the fifth (Join Order and Method Computations). We will use the trace for a simple query to look at each of the sections:

```
select dname, ename from emp, dept
where emp.deptno = dept.deptno
and ename = :b1
```

A trace of even a moderately complex query can easily go into megabytes in size.

QUERY

This is the most easily understood of the sections. It is simply the SQL that is being parsed. If the statement is parsed by the rule base optimizer then the trace does not extent beyond the query section. Remember, there are 3 reasons why a SQL may be parsed by the RBO:

1 optimizer_mode or optimizer_goal are set to rule
2 the statement contains a "rule" hint
3 None of the tables in the statement is analyzed *and* the statement does not contain any hints.

```
Oracle8i Enterprise Edition Release 8.1.7.0.0 – Production[1]
…
*** SESSION ID:(10.372) 2001-11-27 21:06:39.781
QUERY
explain plan set statement_id='d' for
select dname, ename from emp, dept
where emp.deptno = dept.deptno
and ename = :b1
```

PARAMETERS USED BY THE OPTIMIZER

In this section of the trace the optimizer lists all the init.ora parameters that have an influence on the access plan. The list changes from Oracle version to version. This here is the list for 8.1.7:

```
************************************
PARAMETERS USED BY THE OPTIMIZER
********************************
OPTIMIZER_FEATURES_ENABLE = 8.1.6
OPTIMIZER_MODE/GOAL = Choose
OPTIMIZER_PERCENT_PARALLEL = 0
HASH_AREA_SIZE = 131072
HASH_JOIN_ENABLED = TRUE
HASH_MULTIBLOCK_IO_COUNT = 0
OPTIMIZER_SEARCH_LIMIT = 5
PARTITION_VIEW_ENABLED = FALSE
_ALWAYS_STAR_TRANSFORMATION = FALSE
_B_TREE_BITMAP_PLANS = FALSE
STAR_TRANSFORMATION_ENABLED = FALSE
_COMPLEX_VIEW_MERGING = FALSE
_PUSH_JOIN_PREDICATE = FALSE
PARALLEL_BROADCAST_ENABLED = FALSE
OPTIMIZER_MAX_PERMUTATIONS = 80000
OPTIMIZER_INDEX_CACHING = 0
OPTIMIZER_INDEX_COST_ADJ = 100
QUERY_REWRITE_ENABLED = TRUE
QUERY_REWRITE_INTEGRITY = ENFORCED
_INDEX_JOIN_ENABLED = FALSE
_SORT_ELIMINATION_COST_RATIO = 0
_OR_EXPAND_NVL_PREDICATE = FALSE
_NEW_INITIAL_JOIN_ORDERS = FALSE
_OPTIMIZER_MODE_FORCE = TRUE
_OPTIMIZER_UNDO_CHANGES = FALSE
_UNNEST_SUBQUERY = FALSE
_PUSH_JOIN_UNION_VIEW = FALSE
_FAST_FULL_SCAN_ENABLED = TRUE
_OPTIM_ENHANCE_NNULL_DETECTION = TRUE
_ORDERED_NESTED_LOOP = FALSE
_NESTED_LOOP_FUDGE = 100
_NO_OR_EXPANSION = FALSE
_QUERY_COST_REWRITE = TRUE
QUERY_REWRITE_EXPRESSION = TRUE
_IMPROVED_ROW_LENGTH_ENABLED = TRUE
_USE_NOSEGMENT_INDEXES = FALSE
_ENABLE_TYPE_DEP_SELECTIVITY = TRUE
_IMPROVED_OUTERJOIN_CARD = TRUE
_OPTIMIZER_ADJUST_FOR_NULLS = TRUE
_OPTIMIZER_CHOOSE_PERMUTATION = 0
_USE_COLUMN_STATS_FOR_FUNCTION = FALSE
_SUBQUERY_PRUNING_ENABLED = TRUE
_SUBQUERY_PRUNING_REDUCTION_FACTOR = 50
_SUBQUERY_PRUNING_COST_FACTOR = 20
_LIKE_WITH_BIND_AS_EQUALITY = FALSE
_TABLE_SCAN_COST_PLUS_ONE = FALSE
_SORTMERGE_INEQUALITY_JOIN_OFF = FALSE
_DEFAULT_NON_EQUALITY_SEL_CHECK = TRUE
_ONESIDE_COLSTAT_FOR_EQUIJOINS = TRUE
DB_FILE_MULTIBLOCK_READ_COUNT = 32
SORT_AREA_SIZE = 131072
```

BASE STATISTICAL INFORMATION

Next the trace lists the base statistics for all tables in the query and their indexes. The base statistics consist of:

FOR TABLES:

Trace label	dba_tables column	
CDN	NUM_ROWS	The cardinality = number of rows of the table
NBLKS	BLOCKS	The number of blocks below the high water mark
TABLE_SCAN_CST		The estimated cost in I/O to full-table-scan the table
AVG_ROW_LEN	AVG_ROW_LEN	The average length of a row

[1] Text like this, surrounded by a box, is copied from an actual 10053 trace.

FOR INDEXES:

Trace label	dba_indexes column	
Index#, col#		The object# of the index and the column_id of the columns. Oracle 9 brings an improvement by using the index name rather than index#
LVLS	BLEVEL	The height of the index b-tree
#LB	LEAF_BLOCKS	The number of leaf blocks
#DK	DISTINCT_KEYS	The number of distinct keys of the index
LB/K	AVG_LEAF_BLOCKS_PER_KEY	The average number of leaf blocks per key
DB/K	AVG_DATA_BLOCKS_PER_KEY	The average number of data blocks per key
CLUF	CLUSTERING_FACTOR	The clustering factor of the index

```
***************************************
BASE STATISTICAL INFORMATION
***********************
Table stats    Table: DEPT   Alias: DEPT
  TOTAL ::  CDN: 16  NBLKS:  1  TABLE_SCAN_CST: 1  AVG_ROW_LEN:  20
-- Index stats
  INDEX#: 23577  COL#: 1
    TOTAL ::  LVLS: 0   #LB: 1  #DK: 16  LB/K: 1  DB/K: 1  CLUF: 1
***********************
Table stats    Table: EMP   Alias: EMP
  TOTAL ::  CDN: 7213  NBLKS:  85  TABLE_SCAN_CST: 6  AVG_ROW_LEN:  36
-- Index stats
  INDEX#: 23574  COL#: 1
    TOTAL ::  LVLS: 1   #LB: 35  #DK: 7213  LB/K: 1  DB/K: 1  CLUF: 4125
  INDEX#: 23575  COL#: 2
    TOTAL ::  LVLS: 1   #LB: 48  #DK: 42  LB/K: 1  DB/K: 36  CLUF: 1534
  INDEX#: 23576  COL#: 8
    TOTAL ::  LVLS: 1   #LB: 46  #DK: 12  LB/K: 3  DB/K: 34  CLUF: 418
***************************************
```

BASE TABLE ACCESS COST

All the information in the trace that we looked at so far it has been just a regurgitation of facts that are available from the static and dynamic (v$parameter) dictionary views. Now the optimizer is using this information to evaluate access plans. First the CBO looks at the different possibilities and costs to access each of the tables in the SQL by itself, taking into consideration all applicable predicates except join predicates.

BASE ACCESS PLANS

This is a list of the basic plans to access a single table or, more accurately, a row source. I have practically only seen plans 2, 3, 4, 5, and 23

0 parallel hint
1 no access path spec
2 table scan
3 index unique
4 index range
5 index and-equal
6 order by using an index
7 open cluster
8 hash cluster
9 rowid lookup
10 range scan backwards
11 rowid range scan
12 driving_site hint
14 cache hint
15 nocache hint
16 partitions hint
17 nopartitions hint
18 anti-join
19 index rowid range scan
20 bitmap index
21 parallel_index hint
22 noparallel_index hint
23 index fast full scan
24 swap inputs to join
25 fact table
26 not a fact table
27 merge of this view
28 don't push join predicate into this view
29 push join predicate into this view
30 no_merge of this view
31 semi-join

SINGLE TABLE ACCESS PATH

```
SINGLE TABLE ACCESS PATH ..... 1
Column:      ENAME  Col#: 2      Table: EMP   Alias: EMP ..... 2
    NDV: 42        NULLS: 0        DENS: 2.3810e-002 ..... 3
  TABLE: EMP     ORIG CDN: 7213  CMPTD CDN: 172 ..... 4
  Access path: tsc  Resc:  6  Resp:  6 ..... 5
  Access path: index (equal) ..... 6
      INDEX#: 23575  TABLE: EMP ..... 7
      CST: 39  IXSEL:  0.0000e+000  TBSEL:  2.3810e-002 ..... 8
  BEST_CST: 6.00  PATH: 2  Degree:  1 ..... 9
```

This part of the trace requires some explanation. It is the optimizer's evaluation of how to possibly access the EMP table. Line numbers have been put on the right margin for reference. First the CBO lists statistical information for column ename of table EMP (lines 2 and 3). This data is straight from dba_tab_columns. Why was it not printed in the section "Base Statistical Information"? This is because only relevant column statistics, statistics for columns that appear in a predicate for the table, are shown. If the same table is used multiple times in a query, with different predicates for each incidence, each occurrence will get its own SINGLE TABLE ACCESS PATH evaluation with a tailored list of column statistics.

The column statistics and their corresponding column names in dba_tab_columns are

Trace label	*dba_tables column*	
NDV	NUM_DISTINCT	Number of distinct values for the column
NULLS	NUM_NULLS	Number of rows with a null "value" for the column
DENS	DENSITY	"Density" of the column. Without histogram this is = 1/NDV
LO	LOW_VALUE	The lowest value for the column (only for numeric columns)
HI	HIGH_VALUE	The highest value for the column (only for numeric columns)

Line 4 restates the table's cardinality (number of rows) ORIG CDN and the CMPTD CDN (computed cardinality). The computed cardinality is calculated as

CMPTD CDN = ORIG CDN * FF

where FF is the "Filter Factor" of all predicates. We'll look at what a filter factor is and how it is calculated in a moment.

Line 5 states the cost of accessing the table with a full table scan. The cost, here 6[2], is a function of the table's blocks below the high water mark (NBLKS) and the init.ora parameter db_file_multi_block_read_count.

We will deal with the index access evaluation and cost calculation (lines 6 to 8) shortly.

Line 9 sums up the access path evaluation for this table showing the best cost (6.0) for a path 2 (Table Scan).

TABLE SCAN COST

Let's look at the table scan cost (TSC) in a bit more detail. Here are the base statistics for two other, larger tables:

```
TOTAL ::  CDN: 115630  NBLKS:  4339  TABLE_SCAN_CST: 265  AVG_ROW_LEN:  272
TOTAL :: CDN: 454503 NBLKS: 8975 TABLE_SCAN_CST: 548 AVG_ROW_LEN: 151
```

You often read that the cost of a table scan is blocks / db_file_multi_block_read_count. It seems to make sense since Oracle can read db_file_multi_block_read_count blocks with each I/O. However, a quick calculation with the above values shows that

NBLKS / TABLE_SCAN_CST = 4339 / 265 = 16.373 ≠ db_file_multi_block_read_count[3] (which was 32, see page 3)

For the other table we have

NBLKS / TABLE_SCAN_CST = 8975 / 548 = 16.377

---

[2] Resc is the cost for serial access, Resp the cost using the parallel query option.

[3] For an explanation why Oracle sometimes reads fewer than db_file_multi_block_read_count blocks see "Why are Oracle's Read Events "Named Backwards"?" by Jeff Holt (*www.hotsos.com*)

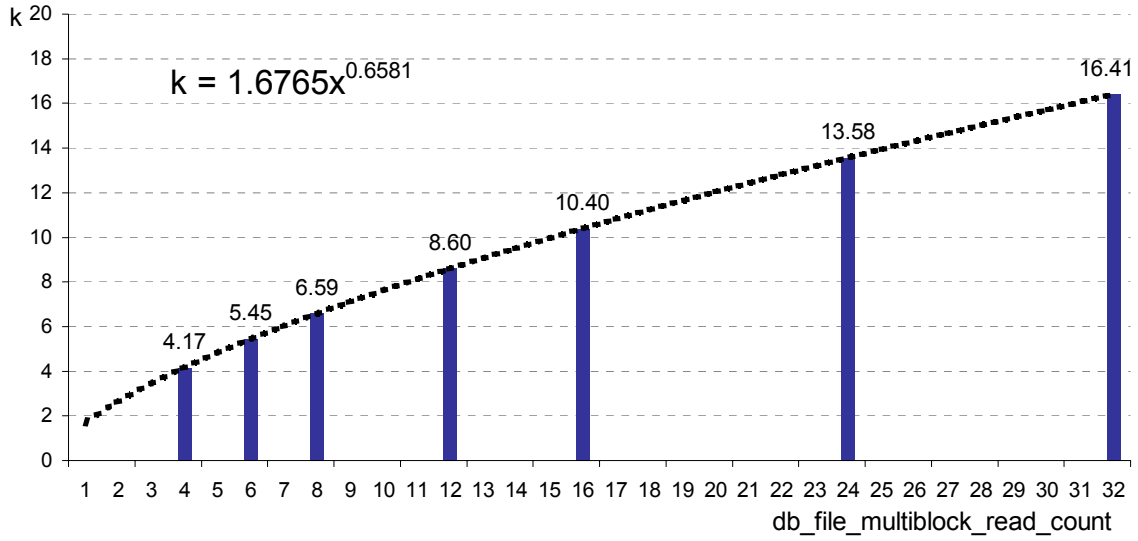
TABLE SCAN COST AND MULTI_BLOCK_READ_COUNT

The CBO's cost estimate for a full table scan is directly related to NBLKS, the number of blocks below the high water mark, and inversely related to db_file_multiblock_read_count (D_F_M_R_C). However, for the reason explained in Jeff Holt's paper, the CBO uses a value discounted from the full D_F_M_R_C:

TABLE_SCAN_CST = NBLKS / k

For anyone interested, I have attempted to find the rule linking k and D_F_M_R_C by running a number of tests for different size tables with different values for D_F_M_R_C – easy since you can change the latter with an "alter session" command. I calculated the k values for D_F_M_R_C values of 4, 8, 16, and 32, drew a column chart and fount a fitting trend curve. The k values for the D_F_M_R_C values of 6, 12, and 24 fell right on the trend curve, confirming the trend curve and formula.

It is evident from this process that the formula for k below is an empirical one. But it yields the correct values to "explain" the optimizer's TABLE_SCAN_CST estimates.



Tablescan cost factor k and db_file_multiblock_read_count

Keep in mind that the thus established k factor is only used in the CBO's estimate for the cost of a full table scan – or fast index scan. The actual cost in I/O of a full table scan depends on other factors besides D_F_M_R_C: proper extent planning and management and whether data blocks of the table are already present in the buffer pool and how many.

PREDICATES AND FILTER FACTORS

In order to understand the index access cost calculations it is necessary to discuss filter factors and their relationship to the query's predicates. A filter factor is a number between 0 and 1 and, in a nutshell, is a measure for the selectivity of a predicate, or, in mathematical terms, the probability that a particular row will match a predicate or set of predicates. If a column has 10 distinct values in a table and a query is looking for all rows where the column is equal to one of the values, you intuitively expect that it will return 1/10 of all rows, presuming an equal distribution. That is exactly the filter factor of a single column for an equal predicate:

FF = 1/NDV = density

Both statistics, NDV (a.k.a. num_distinct) and density, are in dba_tab_columns but the optimizer uses the value of density in most its calculations. This has ramifications as we will see.

Here is the relationship between predicates and the resulting filter factor:

WITHOUT BIND VARIABLES:

predicate	Filter factor
`c1 = value`	`c1.density`[4]
`c1 like value`	`c1.density`
`c1 > value`	`(Hi - value) / (Hi - Lo)`
`c1 >= value`	`(Hi - value) / (Hi - Lo) + 1/c1.num_distinct`
`c1 < value`	`(value - Lo) / (Hi - Lo)`
`c1 <= value`	`(value - Lo) / (Hi - Lo) + 1/c1.num_distinct`
`c1 between val1 and val2`	`(val2 – val1) / (Hi - Lo) + 2 * 1/c1.num_distinct`

These filter factor formulas for the predicates were observed on an Oracle system where all columns were NOT NULL. As Jonathan Lewis (jonathan@jlcomp.demon.co.uk) and Joakim Treugut (Joakim.Treugut@synergy.co.nz) have pointed out, in general there is an adjustment factor for nulls and the equality filter factor, for example, becomes

```
c1.density * ( 1 - num_nulls/num_rows)
```

It further appears that in the case of an equality predicate, the estimated cardinality is increased by 1 if there are nulls.

WHEN USING BIND VARIABLES:

predicate	Filter factor
`c1 = :b1`	`c1.density`
`c1 {like | > | >= | < | <=} :b1`	`{5.0000e-02 | c1.density }`[5]
`c1 between :b1 and :b2`	`5.0000e-02 * 5.0000e-02`

COMBINING PREDICATES / FILTER FACTORS:

predicate	Filter factor
`predicate 1 and predicate 2`	`FF1 * FF2`
`predicate 1 or predicate 2`	`FF1 + FF2 – FF1 * FF2`

The rules on how filter factors are calculated when predicates are combined with "and" or "or" are straight out of probability theory. Given this fact, it should be noted that probability theory stipulates that these formulas for combining probabilities are only valid if "the predicates are independent".

Just as the basic column densities presume a uniform distribution of the distinct column values, the CBO cost calculations for a plan presume independence of the predicates. But while there is some remedy in the form of histograms if the data distribution is not uniform, there is no remedy for cases where the predicates are not independent.

For the calculation of the filter factors for ranges of string literals, the value of the literal is the weighted sum of the ASCII values of its characters. Strings of different lengths are "right padded" with zeros:

```
ADAMS = 65*256^4   + 68*256^3   + 65*256^2   + 77*256^1   + 83*256^0   = 2.8032e+11
ward  = 119*256^4  + 97*256^3   + 114*256^2  + 100*256^1  + 0*256^0    = 5.1274e+11
James = 74*256^4   + 97*256^3   + 109*256^2  + 101*256^1  + 115*256^0  = 3.1946e+11
```

[4] Unless a histogram was collected on the column. Then the calculation of the filter factor is more complicated.

[5] Depending on whether the undocumented init.ora parameter _LIKE_WITH_BIND_AS_EQUALITY is false (the default) or true.

COLUMN STATS WITH HISTOGRAM

In the absence of histogram data, all filter factors are derived from the density of the columns involved, or from fixed values if no statistics have been gathered at all. Histograms complicate the filter factor calculations and go beyond the scope of this paper. In spite of that, we'll take a brief look at what changes when histograms are calculated on a column and correct the record on a couple of myths on the way. Histograms are gathered when the option

"for [all [indexed]] columns [size {n|75}] [col1 [size {n|75}] [, col2…]]" ANALYZE

"for [all [{indexed | hidden}]] columns size n [col1 {[, col2…]]" DBMS_STATS.GATHER_TABLE_STATS

is used with the analyze command or the DBMS_STATS.GATHER_TABLE_STATS procedure. Note the subtle difference in the phrasing between the two methods.

There are two types of histograms:

Value-Based Histograms[6]

The number of buckets is equal to the nr of distinct values and the "endpoint" of each bucket is the number of rows in the table with that value. Oracle builds a value based histogram if the size in the analyze for a column is larger than the number of distinct values for the column.

For a query without a bind variable in the predicate the optimizer uses the selectivity given directly by the histogram if the value exists in the histogram. If not, it uses the column's density for the selectivity[7] in the case of equality. For < or <= it uses the value count from the next lower histogram entry.

Height-Based Histograms[8]

Height-based histograms place approximately the same number of values into each bucket, so that the endpoints of the bucket are determined by how many values are in that bucket. Oracle builds a height based histogram if the size in the analyze for a column is smaller than the number of distinct values for the column.

A commonly held belief is that histograms are useless, i.e. have no effect on the access plan, if bind variables are used since the value is not known at parse time and the CBO – histograms are only ever used by the CBO – can not determine from the histogram if it should use an available index or not. While the latter is true, the gathering of histograms still *can* change the access plan. Why and how?

Because

a) the optimizer uses the density in its filter factor calculation, not NDV.

b) the density is calculated differently for columns with histograms, not simply as 1/NDV

If the density changes, the costs of plan segments and the cardinality estimates of row sources change and hence the entire plan may change. I have successfully exploited that aspect of histograms in tuning.

Another popular myth is that there is no point in gathering histograms on non-indexed columns, likely born from the assumption that the only role of a histogram is to let the optimizer decide between a full table scan and an index access. However, the CBO uses filter factors, derived from column densities, to calculate the costs of different access plans, and ultimately choose an access plan; and filter factors are used in two places in this calculation of access plan costs:

First In the calculation of index access costs. (see below, page 9)

Second In the calculation of row source cardinalities. (refer back to page 5)

In the latter calculation, the filter factors of predicates on non-indexed columns do get used. What is more, the row source cardinality has ultimately the more decisive effect as it guides the composition of the overall access plan. In my experience, the cause for a poor access plan is more often the incorrect estimate of the cardinality of a row source than the incorrect estimate of an index access cost.

[6] In the literature commonly referred to as "equi-width" histograms.

[7] It generally uses the density rather than 0 when it can't find what it is looking for in the histogram, e.g. in the case of < min-value.

[8] In the literature commonly referred to as "equi-depth" histograms

INDEX ACCESS COSTS

Having discussed filter factors, we are now ready to look at the other part of the single table access path evaluation – the calculation of the cost of accessing the needed rows via an index. Recall from page 5:

```
SINGLE TABLE ACCESS PATH ..... 1
Column:      ENAME  Col#: 2      Table: EMP   Alias: EMP ..... 2
    NDV: 42        NULLS: 0         DENS: 2.3810e-002 ..... 3
  TABLE: EMP     ORIG CDN: 7213  CMPTD CDN: 172 ..... 4
  Access path: tsc  Resc:  6  Resp:  6 ..... 5
  Access path: index (equal) ..... 6
      INDEX#: 23575  TABLE: EMP ..... 7
      CST: 39  IXSEL:  0.0000e+000  TBSEL:  2.3810e-002 ..... 8
  BEST_CST: 6.00  PATH: 2  Degree:  1 ..... 9
```

We have already dealt with the table access path. Now we look at the index access cost calculation (lines 6-8). Line 6 lists the kind of index access – an index (equal) scan[9].

In order to better follow the cost calculation, the statistics for the index are repeated below (from page 4)

```
  INDEX#: 23575  COL#: 2
    TOTAL ::  LVLS: 1   #LB: 48  #DK: 42  LB/K: 1  DB/K: 36  CLUF: 1534
```

How does the CBO arrive at the cost of 39 (line 8)? The formula goes as follows

```
blevel + FF*leaf_blocks + FF*clustering_factor
1 + 2.3810e-2*48 + 2.3810e-2*1534 = 1 + 1.1429 + 36.5245 = 38.6674
```

where FF is the filter factor, or selectivity, of the predicate. We have just established that for this predicate – ename = :b1 – the filter factor is equal to the density of the column ename. Trace 10053 lists the FFs used for the calculation of the index access cost in the IXSEL and TBSEL values. The IXSEL value is only independently used and non-zero for the cost evaluations of index-range scans. In this case, an "index (equal)" access, all leaf blocks for the predicate value qualify; there is not further selection and therefore no filter factor to be applied at the leaf level and the index selectivity is shown as 0.

The formula essentially equates to the count of blocks that Oracle has to traverse in order to get all qualifying rows:

From the root page follow the tree down to the leaf page	blevel blocks
Access all qualifying leaf blocks	leaf blocks * filter factor
Access all qualifying data blocks	clustering factor * filter factor

The cost calculations for other index access methods are:

Unique scan	blevel+1	
Fast full scan	leaf_blocks / k	($k = 1.6765x^{0.6581}$)
Index-only	blevel + FF*leaf_blocks	

Again, the cost formulas follow from the count of blocks to be accessed.

Conclusions that can be drawn from these cost formulas:

- leaf_blocks contributes to all but the unique index access cost. Index compression, where appropriate, reduces the number of leaf blocks, lowering the index access costs and can therefore result in the CBO using an index where before it did not.
- Except for a unique index access, the height of the index (blevel) contributes negligibly to the cost.
- The clustering factor affects only an index range scan, but then heavily, given the fact the it is orders of magnitude bigger than LEAF_BLOCKS.

[9] An index (equal) access – not to be confused with an "and equal" access plan – is a special case of an index range scan where the predicate(s) equal the column(s) of the index, which is the case here.

Let me demonstrate the index access cost calculation using another example, one with multiple composite indexes. The query is paraphrased as

```
select … from tbl a
where a.col#1 = :b1
  and a.col#12 = :b2
  and a.col#8 = :b3
```

And the index and column statistics are

INDEX#	COL#	LVLS	#LB	#DK	LB/K	DB/K	CLUF
8417	27, 1	1	13100	66500	1	22	1469200
8418	1, 12, 7	2	19000	74700	1	15	1176500
8419	3, 1, 4, 2	2	31000	49700	1	2	118000
15755	1, 12, 8	1	12600	18800	1	30	1890275
8416	1, 2, 33, 4, 5, 6	2	25800	1890300	1	1	83900

```
Col#: 1    NDV: 10     NULLS: 0        DENS: 1.0000e-1
Col#: 12   NDV: 8      NULLS: 0        DENS: 1.2500e-1
Col#: 8    NDV: 33     NULLS: 0        DENS: 3.0303e-1
```

```
Access path: index (scan) ..... 1
  INDEX#: 8418 CST: 14947  IXSEL:  1.2500e-002  TBSEL:  1.2500e-002 ..... 2
Access path: index (equal) ..... 3
  INDEX#: 15755  CST: 7209  IXSEL:  0.0000e+000  TBSEL:  3.7879e-003 ..... 4
Access path: index (scan) ..... 5
  INDEX#: 8416 CST: 10972  IXSEL:  1.0000e-001  TBSEL:  1.0000e-001 ..... 6
```

Of the 5 indexes, the first and third (#8417 and #8419) are not considered at all because their leading column is not among the predicates.

You have probably read that many times and the 10053 trace proves it:

> an index is not being used, is not even considered and entered into any plan cost calculation if its leading column is not among the predicates !

INDEX# 8418

Only two of the three index columns are predicates and are thus used to calculate the filter factor:

$FF = 1.0000e^{-1} * 1.2500e^{-1} = 1.2500e^{-2}$

$cost = lvl + FF*\#LB + FF*clustering\ factor$

$= 2 + 19{,}000*1.2500e^{-2} + 1176500*1.2500e^{-2}$

$= 2 + 237.5 + 14706.25 = 14945.75$

INDEX# 15755

All three of the three index columns are predicates and are thus used to calculate the filter factor:

$FF = 1.0000e^{-1} * 1.2500e^{-1} * 3.0303e^{-1} = 3.7879e^{-3}$

$cost = lvl + FF*\#LB + FF*clustering\ factor$

$= 1 + 12{,}600*3.7879e^{-3} + 1{,}890{,}275*3.7879e^{-3}$

$= 2 + 47.73 + 7160.13 = 7208.86$

INDEX# 8416

Only one of the three index columns are predicates and are thus used to calculate the filter factor:

$FF = 1.0000e^{-1}$

$cost = lvl + FF*\#LB + FF*clustering\ factor$

$= 2 + 25{,}800*1.0000e^{-1} + 83{,}900*1.0000e^{-1}$

$= 2 + 2580 + 8390 = 10972$

Even though index 8416 has only one column in the predicates of the query, its access cost is lower than that of index 8418 with two predicate columns, solely due to its much better clustering factor.

DEFAULT TABLE, INDEX, AND COLUMN STATISTICS

Remember that the rule based optimizer parses statements where none of the tables have statistics. What if only one, or a few but not all, tables, indexes, or columns have no statistics? There are different claims for what Oracle does in that case. The most popular is that Oracle uses the rule based optimizer for that table. But parsing is not a mix and match exercise – a statement is either parsed entirely by CBO or entirely by RBO. If at least one table in the query has statistics (and optimizer goal is not rule) then the cost base optimizer does parse the query.

Another claim is that Oracle will dynamically, at runtime, estimate statistics on the objects without statistics. I have not seen any evidence of that.

Let us examine what the 10053 trace shows if the statistics on the EMP table are deleted:

```
**************************************....1
BASE STATISTICAL INFORMATION....2
***********************....3
 Table stats    Table: EMP   Alias: EMP....4
  TOTAL :: (NOT ANALYZED) CDN: 3462 NBLKS: 85 TABLE_SCAN_CST: 6 AVG_ROW_LEN: 100....5
-- Index stats....6
  INDEX#: 23574  COL#: 1 ....7
    TOTAL ::  LVLS: 1   #LB: 25  #DK: 100  LB/K: 1  DB/K: 1  CLUF: 800....8
  INDEX#: 23575  COL#: 2 ....9
    TOTAL ::  LVLS: 1   #LB: 25  #DK: 100  LB/K: 1  DB/K: 1  CLUF: 800....10
  INDEX#: 23576  COL#: 8 ....11
    TOTAL ::  LVLS: 1   #LB: 25  #DK: 100  LB/K: 1  DB/K: 1  CLUF: 800....12
**************************************....13
```

Note the "(NOT ANALYZED)" label preceding the table and index statistics (line 5).

Comparing lines 5, 8, 10, and 12 with the corresponding lines from the "BASE STATISTICAL INFORMATION" of the analyzed table on page 4, we note the following:

AVG_ROW_LEN for "NOT ANALYZED" tables defaults to 100

NBLKS and hence TABLE_SCAN_COST are identical to those of the analyzed table. How is that possible?
The answer is actually quite simple and has interesting ramifications:

The NBLKS statistic for an unanalyzed table is taken from the table's segment header and is therefore more accurate than the NBLKS statistic of an analyzed table which may be stale.

The access plan of a query on analyzed tables does not change as long as the statistics and the init.ora parameters do not change (i.e. as long as the tables are not re-analyzed), even if the tables grow or change in other significant ways. This is no longer true if one if the tables is "unanalyzed". A change in its size (NBLKS) is immediately reflected not only in its TSC but also, as will be demonstrated shortly, in the defaults for the table's cardinality and column densities and can therefore result in a change of access plan.
As a corollary, if you are using DBMS_STATS to transport the statistics from a production database to a test database in order to do your SQL analysis and tuning there, watch out for tables without statistics. Since the CBO uses the actual number of blocks from the segment header, in this case from the test database rather than production, you can easily get different access plans.

The cardinality is a function of a mixture of actual (NBLKS) and default (AVG_ROW_LEN) values:

CDN = NBLKS * (db_block_size – block overhead) / AVG_ROW_LEN
CDN = 85 * (4096 – 24) / 100 = 346120 / 100 = 3462

The statistics for unanalyzed indexes default to

LVLS	#LB	#DK	LB/K	DB/K	CLUF
1	25	100	1	1	800

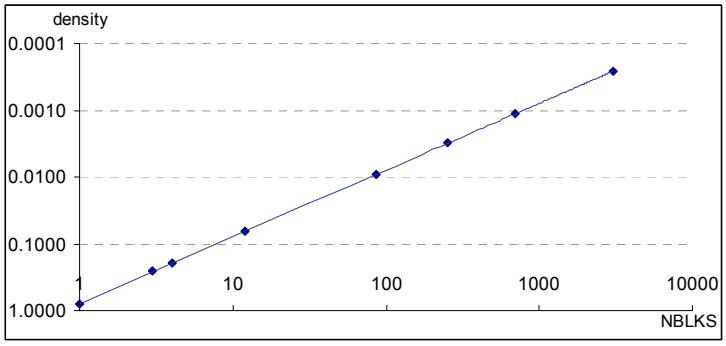
To find the default column statistics, remember that column statistics are not listed under "BASE STATISTICAL INFORMATION" but under "SINGLE TABLE ACCESS PATH":

```
Table stats    Table: EMP   Alias: EMP.....
  TOTAL :: (NOT ANALYZED) CDN: 3543  NBLKS: 87  TABLE_SCAN_CST: 6  AVG_ROW_LEN: 100

SINGLE TABLE ACCESS PATH ..... 1
Column:      ENAME  Col#: 2      Table: EMP   Alias: EMP..... 2
    NO STATISTICS  (using defaults) .....3
    NDV: 111       NULLS: 0         DENS: 9.0319e-003 .....4
```

The defaults for NDV and DENS do not look like nice round defaults like the ones for index statistics. Note also that the density is not 1/NDV[10]. Checking the default column statistics for differently sized tables confirms the suspicion that the column statistics defaults, like the table statistics defaults, are not static, but are derived from the NBLKS value. Examining and plotting the default column density of tables of different sizes in a scatter diagram against the tables' number of blocks shows not only a correlation but clear functional dependency:

NBLKS	NDV	DENS
3	4	2.6016E-01
4	5	1.9632E-01
12	15	6.5440E-02
87	111	9.0319E-03
254	323	3.0939E-03
697	887	1.1275E-03
3052	3884	2.5749E-04



correlation between default column density and NBLKS

The trend line in the double-logarithmic scatter chart between nblks and density has the equation:

`density = 0.7821*nblks^-0.9992  or practically  density = 0.7821 / nblks`

Note that again this is an empirically derived formula. For small values of NBLKS it does not yield the exact same densities as observed in the 10053 trace. Note also that the equation of the correlation function between density and NBLKS is different for different db_block_size values. The actual formula is not really important, but the fact that there is a dependency of the default column density on NBLKS and db_block_size is interesting.

As Joakim Treugut found, the formula is actually very simple:

`density = 1 / (CDN / 32 )  and  NDV = round ( 1 / density , 0)`

Similar to filter factors for range predicates with bind variables (page 7), the optimizer uses defaults for missing/unknown statistics of "not analyzed" tables, indexes, or columns. These defaults range from plain static values (index statistics and avg_row_size) to actual values (NBLKS) and, as we have seen, some values (CDN and column densities) derived from NBLKS using complicated formulas. The analysis of these default values was done on Oracle 8.1.7. It should surprise nobody if some of the default values or calculations have changed, and will continue to change, from Oracle release to release.

GENERAL PLANS

This concludes the single table costing part of the 10053 CBO trace. The next section in the 10053 event trace starts with the heading "GENERAL PLANS". For all but the simplest SQL statements, this section makes up the largest part of the trace. This is where the CBO looks at the costs of all different orders and ways of joining the individual tables and come up with the best access plan.

The CBO has three join methods in its arsenal. These are the three join methods and their costing formulas:

[10] That is actually the consequence of rounding as NDV is calculated as 1/density and then rounded to a whole number.

NL - NESTED LOOP JOIN
join cost = cost of accessing outer table
+ (cardinality of outer table * cost of accessing inner table)

SM – SORT MERGE JOIN
join cost = (cost of accessing outer table + outer sort cost)
+ (cost of accessing inner table + inner sort cost)

HA – HASH JOIN
join cost = (cost of accessing outer table)
+ (cost of building hash table)
+ (cost of accessing inner table)

We will look at the join costing of each method for our simple query.

JOIN ORDER [N]

In the GENERAL PLANS section, the optimizer evaluates every possible permutation of the tables in the query. An exception are tables in a correlated subquery where it is semantically impossible to access the table(s) in the subquery before any tables of the outer query and thus not all permutations constitute a valid plan. Apart from this situation, each permutation evaluation in the trace is given a number and is then followed by the list of the tables – with their aliases to distinguish them if a table occurs multiple times in the query.
The initial join order is chosen by ordering the tables in order of increasing computed cardinality.

In this simple case, the CBO is examining the cost of accessing first the DEPT table and then joining the EMP table. "Now Joining:" is always the beginning of the next batch of join evaluations, introducing the table joining the fray – no pun intended.

```
Join order[1]: DEPT [DEPT] EMP [EMP]
Now joining: EMP [EMP] *******
```

JOINS – NL

```
NL Join ..... 1
  Outer table: cost: 1  cdn: 16  rcz: 13  resp:  1 ..... 2
  Inner table: EMP ..... 3
    Access path: tsc  Resc: 6 ..... 4
    Join resc:  97  Resp:  97 ..... 5
  Access path: index (join stp) ..... 6
      INDEX#: 23575  TABLE: EMP ..... 7
      CST: 39  IXSEL:  0.0000e+000  TBSEL:  2.3810e-002 ..... 8
    Join resc:  625   resp:625 ..... 9
  Access path: index (join index) ..... 10
      INDEX#: 23576  TABLE: EMP ..... 11
      CST: 37  IXSEL:  0.0000e+000  TBSEL:  8.3333e-002 ..... 12
    Join resc:  593   resp:593 ..... 13
    Access path: and-equal ..... 14
      CST: 19 ..... 15
    Join resc:  305   resp:305 ..... 16
Join cardinality:  172 = outer (16) * inner (172) * sel (6.2500e-002)  [flag=0] .... 17
  Best NL cost: 97  resp: 97 ..... 18
```

Line1 lists the join method being evaluated.

Line 2 lists the access cost, estimated cardinality and estimated row size for accessing the outer table, which can be a real table, as in this case, or a row source, i.e. itself the result of a prior join. Please recall from page 4 that the (FTS) cost for DEPT was 1 and its cardinality 16, which is also the computed cardinality as there is no predicate for the DEPT table other than the join predicate. The average row size was 20 there while the row size for the outer table here is given as 13. This is because DEPT contains 3 columns (DEPTNO, DEPT, and LOC) but only DEPTNO and DEPT are required for the query or join, resulting in a lower row size.

Next – lines 3 through 16 – using the above mentioned formula, the cost of a nested loop join is calculated for different methods of accessing the inner table EMP:

Tablescan of EMP at a cost of 6:
cost = cost of outer + cardinality of outer * cost of inner = 1 + 16 * 6 = 97 lines 3 to 5

Scan of index 23575 on ENAME at a cost of 39:
cost = 1 + 16 * 39 = 625 lines 6 to 9

Scan of index 23576 on DEPTNO at a cost of 37:
cost = 1 + 16 * 37 = 593 lines 10 to 13

An "and-equal" access at a cost of 19:
cost = 1 + 16 * 19 = 305 lines 14 to 16

Then, in line 17, the optimizer estimates the cardinality of this join result, which may become a row source to the next join, as the product of the estimated cardinalities of the two sources and the join selectivity, which is

join selectivity = 1/max[NDV(t1.c1), NDV(t2.c2)]
* [(card t1 - # t1.c1 NULLs) / card t1]
* [(card t2 - # t2.c2 NULLs) / card t2]

The join cardinality is only listed in the nested loop join subsection but is the cardinality of the row source in subsequent steps irrespective of the method ultimately chosen for this join.

Finally, on line 18, the CBO lists the lowest cost of the possible ways to do this nested loop join. It does not indicate the method for accessing the inner table. You have to find that yourself among the evaluated methods, using the best NL cost as criterion.

JOINS - SM

```
SM Join
  Outer table:
    resc: 1  cdn: 16  rcz: 13  deg: 1  resp: 1
  Inner table: EMP
    resc: 6  cdn: 172  rcz: 9  deg:  1  resp: 6
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           25 Rows:         16
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           20 Rows:        172
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
  Merge join  Cost:  10  Resp:  10
```

```
SM Join (with index on outer)
  Access path: index (no sta/stp keys)
      INDEX#: 23577  TABLE: DEPT
      CST: 2  IXSEL:  1.0000e+000  TBSEL:  1.0000e+000
  Outer table:
    resc: 2  cdn: 16  rcz: 13  deg: 1  resp: 2
  Inner table: EMP
    resc: 6  cdn: 172  rcz: 9  deg:  1  resp: 6
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           20 Rows:        172
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
  Merge join  Cost:  10  Resp:  10
```

Gaining insight yet into how the CBO calculates the cost of a sort will require more investigation. Note that the SM cost calculation in this example is off by 1. It should be cost of outer + cost of inner + sort cost for outer + sort cost

for inner = 1 + 6 + 2 + 2 = 11. But CBO says it's 10. For the second SM join where the sort on the outer is avoided by accessing it through a sorted index, the formula and CBO do agree: 2 + 6 + 0 (no sort on outer) + 2 = 10.

Joakim Treugut again found the explanation for the apparent discrepancies: The sort costs in this case are actually 1.5. When reported individually, they get rounded to 2, but for the final merge join cost, only the total is rounded:

```
cost = 1 + 6 + 1.5 + 1.5 = 10   and   cost = 2 + 6 + 0 + 1.5 = 9.5 rounded to 10
```

JOINS – HA

```
HA Join
  Outer table:
    resc: 1  cdn: 16  rcz: 13  deg: 1  resp: 1
  Inner table: EMP
    resc: 6  cdn: 172  rcz: 9  deg:  1  resp: 6
  Hash join one ptn:  1   Deg:  1
      hash_area:  32   buildfrag:  33   probefrag:   1              ppasses:    2
  Hash join   Resc: 8   Resp: 8
Join result: cost: 8  cdn: 172  rcz: 22
```

Just as for the sort costs, more investigation will be required to find the rules behind the CBO'S cost calculations of the HA join and how the row sizes, the hash area size and other possible factors influence the cost.

For completeness and as contrast, here is the extract from the trace for the other of the two possible permutations of joining EMP and DEPT:

```
Join order[2]: EMP [EMP] DEPT [DEPT]
Now joining: DEPT [DEPT] *******
NL Join
  Outer table: cost: 6  cdn: 172  rcz: 9  resp:  6..... 4
  Inner table: DEPT
    Access path: tsc  Resc: 1
    Join resc:  178  Resp:  178..... 7
  Access path: index (unique)
      INDEX#: 23577  TABLE: DEPT
      CST: 1  IXSEL:  6.2500e-002  TBSEL:  6.2500e-002
    Join resc:  178   resp:178.....11
  Access path: index (eq-unique)
      INDEX#: 23577  TABLE: DEPT
      CST: 1  IXSEL:  0.0000e+000  TBSEL:  0.0000e+000
    Join resc:  178   resp:178.....15
Join cardinality:  172 = outer (172) * inner (16) * sel (6.2500e-002)  [flag=0]
  Best NL cost: 178  resp: 178
```

The cost for the nested loop join is, as before:
cost of outer table + cardinality of outer table * cost of inner table = 6 + 172 * 1 = 178 (lines 7, 11, and 15).

```
SM Join
  Outer table:
    resc: 6  cdn: 172  rcz: 9  deg: 1  resp: 6
  Inner table: DEPT
    resc: 1  cdn: 16  rcz: 13  deg:  1  resp: 1
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           20 Rows:        172
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           25 Rows:         16
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
  Merge join  Cost:  10  Resp:  10
```

The merge join cost without index access for the outer table, i.e. with sorts on both tables, is the same as in the previous permutation. This should not come as a surprise as this kind of SM join is symmetrical and independent of the join order. Both tables are accessed by their individually cheapest method and sorted.

```
SM Join (with index on outer)
  Access path: index (no sta/stp keys)
      INDEX#: 23576  TABLE: EMP
      CST: 448  IXSEL:  1.0000e+000  TBSEL:  1.0000e+000
  Outer table:
    resc: 448  cdn: 172  rcz: 9  deg: 1  resp: 448
  Inner table: DEPT
    resc: 1  cdn: 16  rcz: 13  deg:  1  resp: 1
    SORT resource      Sort statistics
      Sort width:             3 Area size:       43008   Degree: 1
      Blocks to Sort:         1 Row size:           25 Rows:         16
      Initial runs:           1 Merge passes:        1 Cost / pass:          2
      Total sort cost: 2
  Merge join  Cost:  451  Resp:  451
```

```
HA Join
  Outer table:
    resc: 6  cdn: 172  rcz: 9  deg: 1  resp: 6
  Inner table: DEPT
    resc: 1  cdn: 16  rcz: 13  deg:  1  resp: 1
  Hash join one ptn:  1   Deg:  1
      hash_area:  32   buildfrag:  33   probefrag:   1               ppasses:    2
  Hash join   Resc: 8   Resp: 8
```

For the hash join, too, the cost calculation and result are the same as in the other permutation.

MULTI-TABLE JOINS

The simple example of the EMP and DEPT table join which has been used to exemplify the different cost calculations by the CBO is not well suited to demonstrate how the optimizer evaluates all possible permutations of the base tables. The following is real-life SQL, not a made up example:

```
SELECT A.ACCOUNT, SUM(A.POSTED_TOTAL_AMT)
from PS_PCR_LEDSUM_OP A
  , PSTREESELECT06 L1
  , PSTREESELECT10 L
where A.LEDGER='XXXXXXX'
  and A.FISCAL_YEAR=1998
  and A.ACCOUNTING_PERIOD BETWEEN 1 and 12
  and A.CURRENCY_CD IN (' ', 'CAD')
  and A.PCR_TREENODE_DEPT='YYYYYYYY'
  and A.STATISTICS_CODE=' '
  and L1.SELECTOR_NUM= 7423
  and A.ACCOUNT=L1.RANGE_FROM_06
  and (L1.TREE_NODE_NUM BETWEEN 1968278454 and 1968301256
    OR L1.TREE_NODE_NUM BETWEEN 1968301263 and 1968301270
    OR L1.TREE_NODE_NUM BETWEEN 1968867729 and 196888696
    OR L1.TREE_NODE_NUM BETWEEN 1969156312 and 1969207615)
  and L.SELECTOR_NUM= 7432
  and A.DEPTID=L.RANGE_FROM_10
  and L.TREE_NODE_NUM BETWEEN 1692307684 and 1794871785
group by A.ACCOUNT
```

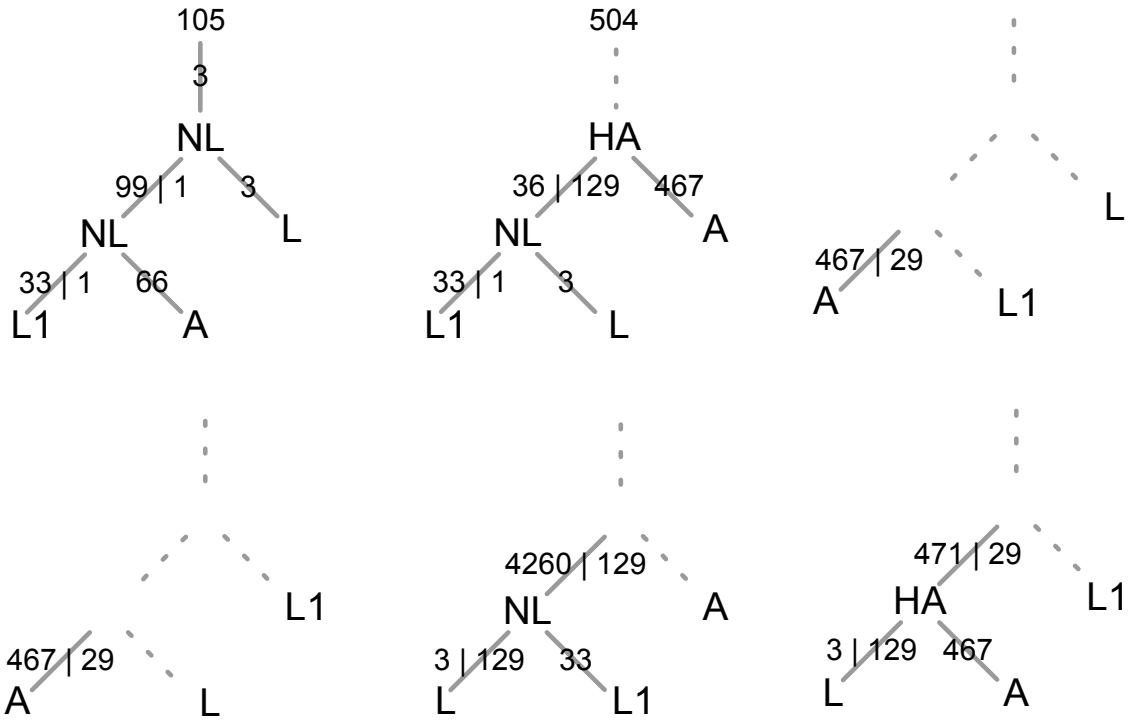
It joins 3 tables, enough to have some permutations of join orders to consider (6), but not so many that one gets lost following the trail. With 4 tables, there would be 24 permutations, with 5 tables 120 permutations, …. In general, there are n! (n faculty) possible permutations to join n tables, so the number of permutations – and the cost and time

to evaluate them – rises dramatically as the number of tables in the SQL increases. The init.ora parameter "optimizer_max_permutations" can be used to limit the number of permutations the CBO will evaluate.

There are also plenty of predicates to give rise to many different base table access considerations. But we are not going to go into those details. I only want to demonstrate the join permutations. Here are the best cost plans for each base table:

```
TABLE: PS_PCR_LEDSUM_OP     ORIG CDN: 683620  CMPTD CDN: 29
  BEST_CST: 467.00  PATH: 4  Degree:  1
TABLE: PSTREESELECT06     ORIG CDN: 125263  CMPTD CDN: 1
  BEST_CST: 33.00  PATH: 2  Degree:  1
TABLE: PSTREESELECT10     ORIG CDN: 238504  CMPTD CDN: 129
   BEST_CST: 3.00  PATH: 4  Degree:  1
```

Here are the schematics and costs of the join permutations:



Tree diagrams of 3-table join permutations

Only the top left plan is costed out completely. All the others were abandoned once their costs exceeded 105, the best cost so far. Here is a quick run down for each plan:

Top left: L1, with a base cost of 33 and a cardinality of 1, is joined with A (base cost 66) in a NL join, The cost of that row source is 33 + 1*66 = 99 with an expected cardinality of 1. This is joined to L, again in a NL join, for a cost of 99 + 1*3 = 102. The final cost of 3 for a total of 105 is for the group by sort. Why is the cost of accessing A – 66 – so much lower than the 467 shown previously and used in all the other plans? Because that is using an index on a join column, which is only applicable in this situation.

Top middle L1 is joined with L at a cost of 33 + 1*3 = 36 and a cardinality of 129. This row source is then joined to A in a HA join for a cost of 36 + 467 + the cost of the hash join operation. Since 36 + 467 is already > 105, the plan is abandoned.
The join index used in the prior plan is available here as well, but the NL join would have a cost of 36 + 129*66 = 8550, so the HA join was cheaper.

Top right	Just the base access of A is already costlier than the best plan so far.
Bottom left	Just the base access of A is already costlier than the best plan so far.
Bottom midle	L is joined with L1 in an NL join for a cost of 3 + 129*33 = 4260 and an estimated cardinality of 129. Plan abandoned.
Bottom right	L is joined with A in a HA join at a cost of 3 + 467 + 1 (hash join cost) = 471. Plan abandoned.

And this is the explain plan for the statement. Compare the operations, estimated cardinalities and cost accumulation with the plan tree above.

```
 cost   card operation
----- ------ -------------------------------------------------------------
  105      1 SELECT STATEMENT
  105      1   SORT GROUP BY
  102      1     NESTED LOOPS
   99      1       NESTED LOOPS
   33      1         TABLE ACCESS FULL PSTREESELECT06
   66     29         TABLE ACCESS BY LOCAL INDEX ROWID PS_PCR_LEDSUM_OP:6-6
    2     29           INDEX RANGE SCAN PS_PCR_LEDSUM_OP_ACC:6-6
    3    129       INDEX RANGE SCAN PSAPSTREESELECT10
```

CONCLUSION

If you take one piece of advice from this paper then this:

Pay close attention to the cardinality values in an explain plan

Wrong estimates by the CBO for the cardinalities can have a devastating effect on the choice of access plan and the performance of the SQL statement. Assuming for the moment that the estimate of cardinality 1 for table PSTREESELECT06 (line5) in the plan above is incorrect and too low, then that would invalidate the entire plan costs and perhaps make the optimizer choose a different, and better, plan. Armed with the knowledge of how the CBO arrived at this number you know what to change in order to help the optimizer make a better assumption: the filter factor of the combined predicates on this table, i.e. ultimately the densities and NDVs of the predicate columns. Here are some means of doing that:

- using histograms on some columns and experiment with the sizes (number of buckets). Even if the histogram data itself is never used, the density for a column with a histogram changes. Generally, collecting a value-based histogram for a column reduces its density by orders of magnitude, whereas collecting a height-based histogram increases the density as long as the number of buckets is less than NDV/2 of the column. If the number of buckets is greater than NDV/2 then the density decreases.
- Deleting the statistics of a column – now possible with the DBMS_STATS procedure – an index or an entire table and let the optimizer use the default densities.
- "Seeding" a table with rows that artificially either increase the cardinality of a column more than the cardinality of the table and thus lower its density, or increase the cardinality of the table without changing the cardinality of the column and thus raise the column's density.
- Using brute force and setting the density. This is possible as of Oracle 8.0 with the advent of DBMS_STATS.SET_xxx_STATS. I recommend using EXPORT_TABLE_STATS to export the statistics into a STATTAB table, modify the value(s) there, and then import the statistics back into the dictionary. Of course you'll make two exports – one into a "work" STATID and one into a "backup" STATID so that you can restore the original statistics in case the change does not have the desired effect. Another possibility is to use export with rows=no. It generates the DDL for the table including DBMS_STATS.SET_xxx_STATS statements to restore the statistics on import. You can then modify the statements to achieve the desired change of statistics. As long as you keep the original you also have a way to backout the changes if necessary. If you do not get an ANALSTATS set of SQL statements in the export dump then the NLS_LANG setting did not match that of the server, or the table had no statistics.

METALINK NOTES

If you want to do some research for yourself or need answers or clarification for questions raised in this paper, I recommend the following notes:

40656.1	Supposedly a note about event 10053. Not externally available (yet).
75713.1	Important Customer Information about numeric EVENTS
35934.1	Cost Based Optimizer - Common Misconceptions and Issues
212809.1	Limitations of the Oracle Cost Based Optimizer.
66030.1	Relationship between optimizer_max_permutations and optimizer_search_limit
32895.1	SQL Parsing Flow Diagram
68992.1	Predicate Selectivity
104817.1	Discussion on Oracle Joins - Costs - Algorithms & Hints
67522.1	Why is my index not used?
62364.1	Hints and Subqueries
46234.1	Interpreting Explain plan
33089.1	Troubleshooting Guide: SQL Tuning
1031826.6	Histograms: An Overview
72539.1	Interpreting Histogram Information
77228.1	How to Tell if a Table has been analyzed
70075.1	Use of bind variables in queries
31412.1	Select to show Optimizer Statistics for CBO
43214.1	Autotrace Option in 7.3

RESOURCES

Oracle University - Course ID: 65340
Oracle8i: Everything You Always Wanted to Know about the Optimizer

www.evdbt.com/library.htm	(Tim Gorman)	look for ev10053.txt
asktom.oracle.com	(Thomas Kyte)	
www.hotsos.com	(Cary Millsap)	
www.ixora.com.au	(Steve Adams)	
www.jlcomp.demon.co.uk	(Jonathan Lewis)	
www.orapub.com	(Craig Shallahamer)	
www.oraperf.com	(Anjo Kolk)	