WHAT IS NEW IN THE ORACLE 9*i* CBO

Wolfgang Breitling, Centrex Consulting Corporation

This paper looks at the differences between Oracle 8*i* and Oracle 9*i* from the perspective of the cost based optimizer (CBO). It discusses the major differences in parameters and features related to the CBO. Although some comparison has been done with Oracle 9*i* Release 1 (9.0.1), all the testing of the effect of new features has been done with Oracle 9*i* Release 2 (9.2.0) on Windows 2000 and Linux Redhat 7.2.

INIT.ORA PARAMETER CHANGES

In the following comparison of the "Parameters Used By The Optimizer" between 8.1.7 and 9.0 and 9.2 respectively, parameters that did not change have been omitted to accentuate the differences. Where the parameter is new in Oracle 9*i*, the parameter name has been highlighted, where the parameter existed in 8*i* but its default changed, the new default is highlighted.

BETWEEN 8.1.7 AND 9.0.1

OPTIMIZER_FEATURES_ENABLE	= 8.1.7	**OPTIMIZER_FEATURES_ENABLE**	= **9.0.1**
ALWAYS_ANTI_JOIN	= **NESTED_LOOPS**	ALWAYS_ANTI_JOIN	= **CHOOSE**
ALWAYS_SEMI_JOIN	= **STANDARD**	ALWAYS_SEMI_JOIN	= **CHOOSE**
OPTIMIZER_MAX_PERMUTATIONS	= 80000	OPTIMIZER_MAX_PERMUTATIONS	= **2000**
OPTIMIZER_PERCENT_PARALLEL	= 0		
_B_TREE_BITMAP_PLANS	= FALSE	_B_TREE_BITMAP_PLANS	= **TRUE**
_COMPLEX_VIEW_MERGING	= FALSE	_COMPLEX_VIEW_MERGING	= **TRUE**
		_CPU_TO_IO	= 0
		_DYN_SEL_EST_NUM_BLOCKS	= 30
		_DYN_SEL_EST_ON	= FALSE
		_GSETS_ALWAYS_USE_TEMPTABLES	= FALSE
		_GS_ANTI_SEMI_JOIN_ALLOWED	= TRUE
_INDEX_JOIN_ENABLED	= FALSE	_INDEX_JOIN_ENABLED	= **TRUE**
_NEW_INITIAL_JOIN_ORDERS	= FALSE	_NEW_INITIAL_JOIN_ORDERS	= **TRUE**
		_NEW_SORT_COST_ESTIMATE	= TRUE
		_OPTIMIZER_COST_MODEL	= CHOOSE
		_OPTIMIZER_PERCENT_PARALLEL	= **101**
_ORDERED_NESTED_LOOP	= FALSE	_ORDERED_NESTED_LOOP	= **TRUE**
		_PRED_MOVE_AROUND	= TRUE
_PUSH_JOIN_PREDICATE	= FALSE	_PUSH_JOIN_PREDICATE	= **TRUE**
_PUSH_JOIN_UNION_VIEW	= FALSE	_PUSH_JOIN_UNION_VIEW	= **TRUE**
		_SYSTEM_INDEX_CACHING	= 0
_UNNEST_SUBQUERY	= FALSE	_UNNEST_SUBQUERY	= **TRUE**

BETWEEN 8.1.7 AND 9.2.0

OPTIMIZER_FEATURES_ENABLE	= 8.1.7	OPTIMIZER_FEATURES_ENABLE	= 9.2.0
		ALWAYS_ANTI_JOIN	= CHOOSE
		ALWAYS_SEMI_JOIN	= CHOOSE
		OPTIMIZER_DYNAMIC_SAMPLING	= 1
OPTIMIZER_MAX_PERMUTATIONS	= 80000	OPTIMIZER_MAX_PERMUTATIONS	= 2000
OPTIMIZER_PERCENT_PARALLEL	= 0		
PARALLEL_BROADCAST_ENABLED	= FALSE	PARALLEL_BROADCAST_ENABLED	= TRUE
_B_TREE_BITMAP_PLANS	= FALSE	_B_TREE_BITMAP_PLANS	= TRUE
_COMPLEX_VIEW_MERGING	= FALSE	_COMPLEX_VIEW_MERGING	= TRUE
		_CPU_TO_IO	= 0
		_GS_ANTI_SEMI_JOIN_ALLOWED	= TRUE
		_GSETS_ALWAYS_USE_TEMPTABLES	= FALSE
_INDEX_JOIN_ENABLED	= FALSE	_INDEX_JOIN_ENABLED	= TRUE
_NEW_INITIAL_JOIN_ORDERS	= FALSE	_NEW_INITIAL_JOIN_ORDERS	= TRUE
		_NEW_SORT_COST_ESTIMATE	= TRUE
		_OPTIMIZER_COST_MODEL	= CHOOSE
		_OPTIMIZER_DYN_SMP_BLKS	= 32
		_OPTIMIZER_PERCENT_PARALLEL	= 101
_ORDERED_NESTED_LOOP	= FALSE	_ORDERED_NESTED_LOOP	= TRUE
		_PRED_MOVE_AROUND	= TRUE
_PUSH_JOIN_PREDICATE	= FALSE	_PUSH_JOIN_PREDICATE	= TRUE
_PUSH_JOIN_UNION_VIEW	= FALSE	_PUSH_JOIN_UNION_VIEW	= TRUE
		_SYSTEM_INDEX_CACHING	= 0
_TABLE_SCAN_COST_PLUS_ONE	= FALSE	_TABLE_SCAN_COST_PLUS_ONE	= TRUE
_UNNEST_SUBQUERY	= FALSE	_UNNEST_SUBQUERY	= TRUE

CHANGED INIT.ORA PARAMETER DEFAULTS

This is another view of the changed defaults of existing parameters:

	8.1.7	9.0	9.2
OPTIMIZER_MAX_PERMUTATIONS	80000	2000	2000
_B_TREE_BITMAP_PLANS	false	true	true
_COMPLEX_VIEW_MERGING	false	true	true
_INDEX_JOIN_ENABLED	false	true	true
_NEW_INITIAL_JOIN_ORDERS	false	true	true
_OR_EXPAND_NVL_PREDICATE	false	true	true
_ORDERED_NESTED_LOOP	false	true	true
_PUSH_JOIN_PREDICATE	false	true	true
_PUSH_JOIN_UNION_VIEW	false	true	true
_TABLE_SCAN_COST_PLUS_ONE	false	false	true
_UNNEST_SUBQUERY	false	true	true
_USE_COLUMN_STATS_FOR_FUNCTION	false	true	true

NEW INIT.ORA PARAMETERS

And here is a comparison of the new parameters and their values between the two Oracle 9*i* Releases. The only difference is in the parameters related to dynamic sampling.

	9.0	9.2
ALWAYS_ANTI_JOIN	choose	choose
ALWAYS_SEMI_JOIN	choose	choose
_SYSTEM_INDEX_CACHING	0	0
_OPTIMIZER_COST_MODEL	choose	choose

```
_GSETS_ALWAYS_USE_TEMPTABLES                false    false
_NEW_SORT_COST_ESTIMATE                     true     true
_GS_ANTI_SEMI_JOIN_ALLOWED                  true     true
_CPU_TO_IO                                  0        0
_PRED_MOVE_AROUND                           true     true
_DYN_SEL_EST_ON                             false
OPTIMIZER_DYNAMIC_SAMPLING                           1
_DYN_SEL_EST_NUM_BLOCKS                     30
_OPTIMIZER_DYN_SMP_BLKS                              32
```

As indicated at the beginning, the list of new and changed init.ora parameters is taken from a comparison of the "Parameters Used By The Optimizer" sections of 10053 event traces. There are, of course, many more – mostly hidden and undocumented – new init.ora parameters, some of which presumable are used by the optimizer; e.g. _OPTIM_PEEK_USER_BINDS (see below).

OPTIMIZER_FEATURES_ENABLE

This is of course not a new parameter but it is an appropriate place to list the optimizer features that are enabled depending on the setting for this parameter. The list is straight from Note 62337.1: "Some settings which this parameter controls in Oracle 9.2 are shown below. Most of these parameters are hidden (underscore) parameters and should not be explicitly set by customers unless specifically asked to do so by Oracle Support Services. They give a finer degree of control over the exact optimizer features which are enabled at a given release".[1]

```
fast_full_scan_enabled                  (true if = 8.0.4)

b_tree_bitmap_plans                     (true if = 8.0.4 and < 8.1.0 or = 9.0.0)
complex_view_merging                    (true if = 8.0.4 and < 8.1.0 or = 9.0.0)
push_join_predicate                     (true if = 8.0.4 and < 8.1.0 or = 9.0.0)
push_join_union_view                    (true if = 8.0.4 and < 8.1.0 or = 9.0.0)
ordered_nested_loops                    (true if = 8.0.4 and < 8.1.0 or = 9.0.0)

improved_outerjoin_card                 (true if = 8.0.6)

optim_enhance_nnull_detection           (true if = 8.1.5)

left_nested_loops_random                (true if = 8.1.6)
enable_type_dep_selectivity             (true if = 8.1.6)
optimizer_mode_force                    (true if = 8.1.6)
improved_row_length_enabled             (true if = 8.1.6)
subquery_pruning_enabled                (true if = 8.1.6)

eliminate_common_subexpr                (true if = 8.1.7)
or_expand_nvl_predicate                 (true if = 8.1.7)
use_column_stats_for_function           (true if = 8.1.7)
minimal_stats_aggregation               (true if = 8.1.7)

optim_peek_user_binds                   (true if = 9.0.0)
optimizer_new_join_card_computation     (true if = 9.0.0)
optim_new_default_join_sel              (true if = 9.0.0)
new_initial_join_orders                 (true if = 9.0.0)
new_sort_cost_estimate                  (true if = 9.0.0)
index_join_enabled                      (true if = 9.0.0)
unnest_subquery                         (true if = 9.0.0)
pred_move_around                        (true if = 9.0.0)
gs_anti_semi_join_allowed               (true if = 9.0.0)
always_semi_join                        (CHOOSE if = 9.0.0, OFF otherwise)
always_anti_join                        (CHOOSE if = 9.0.0, OFF otherwise)
unnest_notexists_sq                     (SINGLE if = 9.0.0, OFF otherwise)
optimizer_max_permutations              (2000 if = 9.0.0, 80000 otherwise)
```

```
optimizer_dynamic_sampling              (1 if = 9.2.0, 0 otherwise)
optim_adjust_for_part_skews             (true if = 9.2.0)
subquery_pruning_mv_enabled             (true if == OPTIM_FEATURES_SPECIAL_VAL)
```

CHANGES TO EXPLAIN PLAN

With every new version, or even release, Oracle seems to make changes to the PLAN_TABLE, and Oracle 9*i* is no exception.

NEW PLAN_TABLE COLUMNS

Below is an overview of the added columns:

CPU_COST	Estimated CPU cost of the operation. The value is proportional to the number of machine cycles required for the operation.
IO_COST	Estimated I/O cost of the operation. The value is proportional to the number of data blocks read by the operation.
TEMP_SPACE	Estimated temporary space, in bytes, used by the operation.
ACCESS_PREDICATES	Predicates used to locate rows in an access structure; for example, start or stop predicates for an index range scan
FILTER_PREDICATES	Predicates used to filter rows before producing them

The columns ACCESS_PREDICATES and FILTER_PREDICATES provide information as to which predicates the optimizer used to reduce the number of rows initially fetched (access predicate) as opposed to predicates used to reduce the result set from fetched rows (filter predicate). Prior to Oracle 9*i*, this information was available only through a 10060 event trace.

Remember to create a new plan table after upgrading (?/rdbms/admin/utlxplan.sql)

NEW EXPLAIN PACKAGE

Rather than updating your explain plan formatting script to include the new information, you can now use the DBMS_XPLAN.DISPLAY procedure to format an explain plan.

```
select * from table  (DBMS_XPLAN.DISPLAY(table, statement_id, format));
```

where

Table	table name where the plan is stored. (default PLAN_TABLE)
Statement_id	statement_id of the plan to be displayed. (default NULL - meaning the most recent one)
Format	the level of detail for the plan.
BASIC	Displays the minimum information in the plan - the operation ID, the object name, and the operation option.
TYPICAL	This is the default. Displays the most relevant information in the plan. Partition pruning, parallelism, and predicates are displayed only when available.
ALL	Maximum level. Includes information displayed with the TYPICAL level and adds the SQL statements generated for parallel execution servers (only if parallel).
SERIAL	Like TYPICAL except that the parallel information is not displayed, even if the plan executes in parallel.

Note: the procedure formats only the last of multiple plans with identical statement_ids – unless they were explained within a very short time of each other. Most often that is desirable, but sometimes it may not be. Just be aware of it.

Below is an example of an explain with format TYPICAL:

```
PLAN_TABLE_OUTPUT
--------------------------------------------------------------------------------

-------------------------------------------------------------------------
| Id  | Operation                 |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT          |             |    63 |  1890 |    20 |
|*  1 |  HASH JOIN                |             |    63 |  1890 |    20 |
```

```
|   2 |   TABLE ACCESS BY INDEX ROWID| T2          |     3 |    45 |     4 |
|*  3 |    INDEX RANGE SCAN          | T2X         |     3 |       |     1 |
|   4 |   TABLE ACCESS FULL          | T1          |   625 |  9375 |    15 |
--------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   1 - access("T2"."FK1"="T1"."PK1")
   3 - access("T2"."D2"=499)

Note: cpu costing is off
```

Note the "warning" that cpu costing is off. You get that if you have not collected system statistics, nor set them manually; more on that later. There is also a warning if the plan_table has the pre 9*i* structure, i.e. does not have the new columns.

NEW BASE ACCESS PATHS

Two new base access paths become available in Oracle 9*i*.

INDEX JOIN

Technically this is not a new access path as it existed in 8i, but there it was disabled by default (_INDEX_JOIN_ENABLED = false). The only way to get this access path "legally", i.e. without changing the undocumented parameter, was to use the INDEX_JOIN hint. In 9i it is enabled (_INDEX_JOIN_ENABLED = true) so the optimizer is now free to choose it where appropriate rather than where instructed to by a hint. For an INDEX_JOIN to be chosen, a sufficiently small number of indexes must exist that contain all the columns required to resolve the query. Based on my tests, the conditions for an index_join seem rare, but your mileage may vary.

For an example, consider the following table and indexes:

Name	Null?	Type	table	index	column	NDV	density	#LB	lvl
N1		NUMBER	A	A1		2,500		267	2
N2		NUMBER			N3	50	2.0000E-02		
N3		NUMBER			N1	2,500	4.0000E-04		
N4		NUMBER			C1	1	1.0000E+00		
N5		NUMBER							
C1		VARCHAR2(30)		A2		2,500		267	2
C2		VARCHAR2(30)			N4	20	5.0000E-02		
					N2	1,250	8.0000E-04		
					C2	1	1.0000E+00		

And this query

"`select a.c1, a.c2 from a where a.n3=10 and a.n4=10`"

All the columns in the query are available from the indexes, but only from both indexes together – exactly what the index join is for:

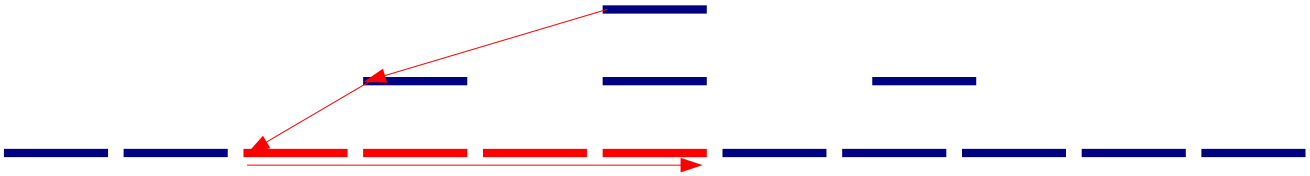
```
PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
| Id  | Operation            |  Name             | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |                   |    20 |  1520 |    31 |
|*  1 |  VIEW                | index$_join$_001  |    20 |  1520 |    31 |
|*  2 |   HASH JOIN          |                   |    20 |  1520 |       |
|*  3 |    INDEX RANGE SCAN  | A1                |    20 |  1520 |    45 |
|*  4 |    INDEX RANGE SCAN  | A2                |    20 |  1520 |    45 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
```

```
   1 - filter("A"."N3"=10 AND "A"."N4"=10)
   2 - access("indexjoin$_alias$_003".ROWID="indexjoin$_alias$_002".ROWID)
   3 - access("indexjoin$_alias$_002"."N3"=10)
   4 - access("indexjoin$_alias$_003"."N4"=10)
```

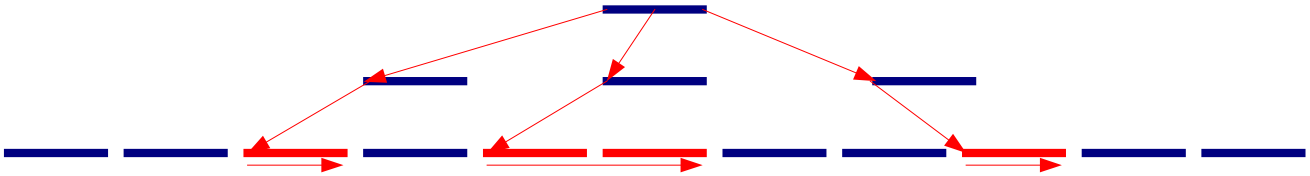
INDEX SKIP SCAN

This is a truly new access path in Oracle 9*i*. During a skip scan, the composite index is accessed once for each distinct value of the leading column(s). For each distinct value, the index is searched to find the query's target values.

In a range scan, the index structure is used to position on the starting index leaf block and the leaf blocks are then scanned in order until the stop condition is reached.



Skip Scans are initiated by probing the index for distinct values of the prefix column(s). Each of these distinct values is then used as a starting point for a regular index search. The result is several separate searches of a single index that, when combined, eliminate the effect of the prefix column.



Example of an index skip scan:

```
table  index  column    NDV        CLUF    CLUF   #LB lvl  #LB/K  #DB/K
------ ------ ------ ------ ----------- ------ ----- --- ------ ------
SS_A   SS_AI           746                996    60   1      1      1
              N1         1  1.0000E+00
              N2        12  8.3333E-02
              N3        24  4.1667E-02

select ch from ss_a where n3=6;

PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
| Id  | Operation                    |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    42 | 31794 |    75 |
|   1 |  TABLE ACCESS BY INDEX ROWID| SS_A        |    42 | 31794 |    75 |
|*  2 |   INDEX SKIP SCAN            | SS_AI       |     2 |       |    33 |
-------------------------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------

   2 - access("SS_A"."N3"=6)
       filter("SS_A"."N3"=6)
```

Note that contrary to popular belief, the optimizer may choose an index skip scan even if the skipped prefix consists of more than one column. It all depends on the estimated costs of the available choices.

In the 10053 trace, an index skip scan does not have its own access path number but goes as path 4 (index range scan).

Index skip scans provide two main benefits:

- improve the performance of certain queries, since queries which previously required table scans may now be able to take advantage of an existing index.
- allow an application to meet its performance goals with fewer indexes; fewer indexes will require less storage space and may improve the performance of DML and maintenance operations.

NEW COST MODEL

The Oracle 9*i* optimizer cost model recognizes that "CPU utilization is as important as I/O"[4] It furthermore takes into account that a multi block I/O incurs a longer wait than a single block I/O:

cost = (#srds * sreadtm + #mrds * mreadtm + #cpucycles / cpuspeed) / sreadtm

or

cost = #srds + [#mrds * mreadtm/sreadtm] + [#cpucycles / cpuspeed / sreadtm]

where

#srds	number of single block reads
sreadtm	single block read time
#mrds	number of multi block reads
mreadtm	multi block read time
#CPUCycles	number of CPU Cycles
CPUspeed	CPU cycles per second

CPUCycles includes CPU cost of query processing ("pure" CPU cost) and CPU cost of data retrieval (CPU cost of the buffer cache get). For parallel execution, necessary adjustments are made while computing estimates for #SRDS, #MRDS, and #CPUCycles.

The old costing model was based solely on the number of single block accesses, equivalent to #SRDS in the new cost model.

SYSTEM STATISTICS

The new cost model is dependent on system statistics – that values for sreadtm, mreadtm, mbrc, and cpuspeed are available. Without them, the cost of every part of an access path is reduced to #SRDS♦, the estimated number of single block reads, i.e. the old cost model.

Besides including an estimate of the cpu resource requirements, the new cost model decouples the cost of a full scan from the DB_FILE_MULTIBLOCK_READ_COUNT setting because the CBO will now use the MBRC system statistic to calculate the cost of a full scan: "The optimizer then has the information to better judge if a full table scan or an index access is more appropriate. Setting the DB_FILE_MULTIBLOCK_READ_COUNT parameter to a high value will thus not result in more full table scans".[3] Before 9iR2 one had to be careful in setting DB_FILE_MULTIBLOCK_READ_COUNT to a very high value in order to make FTS and FFS as efficient as possible because it lowered the cost of all FTS and FFS and risked unwanted access plan changes.[2] DB_FILE_MULTIBLOCK_READ_COUNT was used both as an operational parameter and by the optimizer to derive the cost of a full scan.

Without system statistics, under the old cost model, the cost of a full scan (FTS or FFS) is dependent on many variables – system db_blocksize, tablespace db_blocksize, db_file_multiblock_read_count and some internal parameters.

With system statistics and the new cost model, the cost simply becomes:

tsc = ceiling((nblks/mbrc) * (mreadtim/sreadtim)) + 1•

Note that if system statistics are explicitly set using DBMS_STATS.SET_SYSTEM_STATS:

♦ No distinction is then made between single-block and multi-block reads and the estimated #MRDS are counted as #SRDS.

• +1 only if _table_scan_cost_plus_one is true – the default

- cpuspeed and sreadtm must be set for cpu_costing to become effective.
- mbrc, mreadtm and sreadtm must be set for access path costing of multiblock paths to use the system statistics. Also, sreadtm must be smaller than mreadtm.

GATHERING, USING AND VIEWING SYSTEM STATISTICS

System statistics are designed to give the optimizer information beyond the confines of the currently parsed SQL. Besides gathering somewhat static data such as cpu speed, system statistics also gather data that are workload specific and may change during the course of a day, i.e. the ratio of multi-block to single-block read I/O speed and the average number of blocks read in a multiblock read. These statistics can be very different between time of predominantly OLTP processing versus times of predominantly batch processing. You can gather system statistics for different distinct workloads and load and activate them as appropriate. However, unlike when table, index, or column statistics get updated, the Oracle server does not invalidate already-parsed SQL statements when system statistics get updated. Only the new parsed SQL statements will use the newly activated statistics unless the shared pool is flushed when activating different system statistics.

System statistics are stored in table SYS.AUX_STATS$:

```
select * from sys.aux_stats$;
```

SNAME	PNAME	PVAL1	PVAL2
SYSSTATS_INFO	STATUS		COMPLETED
SYSSTATS_INFO	DSTART		12-25-2002 13:26
SYSSTATS_INFO	DSTOP		12-25-2002 13:29
SYSSTATS_INFO	FLAGS	0	
SYSSTATS_MAIN	SREADTIM	1.532	
SYSSTATS_MAIN	MREADTIM	4.31	
SYSSTATS_MAIN	CPUSPEED	344	
SYSSTATS_MAIN	MBRC	24	
SYSSTATS_MAIN	MAXTHR	1600512	
SYSSTATS_MAIN	SLAVETHR	-1	

To gather, set, export or import system statistics the respective DBMS_STATS procedures must be used.

PEEKING OF USER-DEFINED BIND VARIABLES

Prior to Oracle 9*i*, gathering histograms on columns with skewed data is of somewhat limited use if the SQL are using bind variables[1] since the optimizer can not use the value frequency information of the histogram because it is oblivious of the bind values. Oracle 9*i* tries to correct that by "peeking" at the bind value at parse time:

"The CBO peeks at the values of user-defined bind variables on the first invocation of a cursor. This feature lets the optimizer determine the selectivity of any WHERE clause condition, as well as if literals have been used instead of bind variables. On subsequent invocations of the cursor, no peeking takes place, and the cursor is shared, based on the standard cursor-sharing criteria, even if subsequent invocations use different bind values".[4]

In order to demonstrate this, let us create a table with a column with 21 different values from 0 to 20 with frequencies that are according to the Gauss' normal distribution (see column frequency chart in appendix C). In total the table has 10,000 rows and a frequency histogram for the table has been gathered.

The cursor is first opened with a bind value of 0. From the histogram, the CBO obtains a cardinality of 6 and chooses to use an index access on column n1. For subsequent executions of the same SQL but with different bind values, the plan does not change.

[1] Not useless, however. See [5]. and [6].

var b1 number	card	operation	rows	elapsed
exec :b1 := 0;		SELECT STATEMENT		
	1	SORT AGGREGATE	1	0.42
select sum(n2) from hist3 where n1 = :b1;	6	TABLE ACCESS BY INDEX ROWID HIST3	6	0.37
	6	INDEX RANGE SCAN HIST3_IX	6	0.18
exec :b1 := 11;		SELECT STATEMENT		
	1	SORT AGGREGATE	1	25.44
select sum(n2) from hist3 where n1 = :b1;	6	TABLE ACCESS BY INDEX ROWID HIST3	1,258	22.49
	6	INDEX RANGE SCAN HIST3_IX	1,258	3.67

After flushing the shared pool we begin with a different bind variable value – 11, the 2nd most frequently occurring value. This time the CBO chooses a full scan of the table and again the plan does not change for subsequent executions with different bind values.

var b1 number	card	operation	rows	elapsed
exec :b1 := 11;		SELECT STATEMENT		
	1	SORT AGGREGATE	1	58.16
select sum(n2) from hist3 where n1 = :b1;	1,258	TABLE ACCESS FULL HIST3	1,258	55.56
exec :b1 := 0;		SELECT STATEMENT		
	1	SORT AGGREGATE	1	47.34
select sum(n2) from hist3 where n1 = :b1;	1,258	TABLE ACCESS FULL HIST3	6	47.20

"When bind variables are used in a statement, it is assumed that cursor sharing is intended and that different invocations are supposed to use the same execution plan. If different invocations of the cursor would significantly benefit from different execution plans, then bind variables may have been used inappropriately in the SQL statement".[4]

EXPLAIN PLAN DILEMMA

This exposes an explain plan dilemma – "explain plan" may not settle on the same access path as the CBO chooses when the SQL is actually executed. In the case of bind variables on columns with histograms this is aggravated by the fact that explain plan does not peek at the bind variable, so you can end up with 3 different access plans (potentially even more) for the same SQL – one as a result of parsing by "explain plan", one as a result of parsing with a "popular" (i.e. frequently occurring) bind value, and one as a result of parsing with an infrequently occurring bind value.

Consider the following example of a table PEEK (see Appendix D) with a frequency histogram on column N2 with value 1 occuring 521 times and each of the values 2-25 occuring 20 times.

1. Access path when using "explain plan":

```
explain plan set statement_id = 'peek-21' for
select n1, dt, length(ch) from peek where n2 = :n2 and n3 > 5   PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
| Id  | Operation                    |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |    32 | 31552 |    29 |
|   1 |  TABLE ACCESS BY INDEX ROWID| PEEK        |    32 | 31552 |    29 |
|*  2 |   INDEX RANGE SCAN           | PEEK_2      |   792 |       |     5 |
-------------------------------------------------------------------------------
```

Access path with frequently occurring bind value:

```
exec :n2 := 1;
select n1, dt, length(ch) from peek where n2 = :n2 and n3 > 5   PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------
| Id  | Operation            |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------
|   0 | SELECT STATEMENT     |             |       |       |    50 |
|*  1 |  TABLE ACCESS FULL   | PEEK        |   412 |   396K|    50 |
-------------------------------------------------------------------------
```

Access path with infrequently occurring bind value:

```
exec :n2 := 21;
select n1, dt, length(ch) from peek where n2 = :n2 and n3 > 5   PLAN_TABLE_OUTPUT
-------------------------------------------------------------------------------
| Id  | Operation                    |  Name       | Rows  | Bytes | Cost  |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT             |             |       |       |    16 |
|*  1 |  TABLE ACCESS BY INDEX ROWID| PEEK        |    16 | 15776 |    16 |
|*  2 |   INDEX RANGE SCAN           | PEEK_1      |    20 |       |     2 |
-------------------------------------------------------------------------------
```

Execution Plan Information

It is only fitting, given the just demonstrated very real possibility that the explain plan and the execution plan differ, that Oracle 9*i* now provides more feedback from the actual execution of a SQL statements. V$SQL and V$SQLAREA have new columns, most notably elapsed time, so you can now scan them directly for slow SQL rather than using buffer_gets or disk reads as in indirect measure of inefficient SQL.

V$SQLAREA

Oracle 8*i*		Oracle 9*i*	
SQL_TEXT	VARCHAR2(1000)	SQL_TEXT	VARCHAR2(1000)
SHARABLE_MEM	NUMBER	SHARABLE_MEM	NUMBER
PERSISTENT_MEM	NUMBER	PERSISTENT_MEM	NUMBER
RUNTIME_MEM	NUMBER	RUNTIME_MEM	NUMBER
SORTS	NUMBER	SORTS	NUMBER
VERSION_COUNT	NUMBER	VERSION_COUNT	NUMBER
LOADED_VERSIONS	NUMBER	LOADED_VERSIONS	NUMBER
OPEN_VERSIONS	NUMBER	OPEN_VERSIONS	NUMBER
USERS_OPENING	NUMBER	USERS_OPENING	NUMBER
		FETCHES	NUMBER
EXECUTIONS	NUMBER	EXECUTIONS	NUMBER
USERS_EXECUTING	NUMBER	USERS_EXECUTING	NUMBER
LOADS	NUMBER	LOADS	NUMBER
FIRST_LOAD_TIME	VARCHAR2(19)	FIRST_LOAD_TIME	VARCHAR2(19)
INVALIDATIONS	NUMBER	INVALIDATIONS	NUMBER
PARSE_CALLS	NUMBER	PARSE_CALLS	NUMBER
DISK_READS	NUMBER	DISK_READS	NUMBER
BUFFER_GETS	NUMBER	BUFFER_GETS	NUMBER
ROWS_PROCESSED	NUMBER	ROWS_PROCESSED	NUMBER
COMMAND_TYPE	NUMBER	COMMAND_TYPE	NUMBER
OPTIMIZER_MODE	VARCHAR2(25)	OPTIMIZER_MODE	VARCHAR2(25)
PARSING_USER_ID	NUMBER	PARSING_USER_ID	NUMBER
PARSING_SCHEMA_ID	NUMBER	PARSING_SCHEMA_ID	NUMBER
KEPT_VERSIONS	NUMBER	KEPT_VERSIONS	NUMBER
ADDRESS	RAW(4)	ADDRESS	RAW(4)
HASH_VALUE	NUMBER	HASH_VALUE	NUMBER
MODULE	VARCHAR2(64)	MODULE	VARCHAR2(64)

MODULE_HASH	NUMBER	MODULE_HASH	NUMBER
ACTION	VARCHAR2(64)	ACTION	VARCHAR2(64)
ACTION_HASH	NUMBER	ACTION_HASH	NUMBER
SERIALIZABLE_ABORTS	NUMBER	SERIALIZABLE_ABORTS	NUMBER
		CPU_TIME	NUMBER
		ELAPSED_TIME	NUMBER
		IS_OBSOLETE	VARCHAR2(1)
		CHILD_LATCH	NUMBER

V$SQL

Some of the new columns are the same as those for V$SQLAREA. One particularly useful new column for execution plan analysis is PLAN_HASH_VALUE. Rather than comparing two plans line by line, comparing one PLAN_HASH_VALUE to another easily identifies whether or not two plans are the same. Of course, by nature of the hash, identical hash values do not guarantee identical plans. Different hash values, however do guarantee that the plans are different.

Oracle 8*i*		**Oracle 9*i***	
SQL_TEXT	VARCHAR2(1000)	SQL_TEXT	VARCHAR2(1000)
SHARABLE_MEM	NUMBER	SHARABLE_MEM	NUMBER
PERSISTENT_MEM	NUMBER	PERSISTENT_MEM	NUMBER
RUNTIME_MEM	NUMBER	RUNTIME_MEM	NUMBER
SORTS	NUMBER	SORTS	NUMBER
LOADED_VERSIONS	NUMBER	LOADED_VERSIONS	NUMBER
OPEN_VERSIONS	NUMBER	OPEN_VERSIONS	NUMBER
USERS_OPENING	NUMBER	USERS_OPENING	NUMBER
		FETCHES	NUMBER
EXECUTIONS	NUMBER	EXECUTIONS	NUMBER
USERS_EXECUTING	NUMBER	USERS_EXECUTING	NUMBER
LOADS	NUMBER	LOADS	NUMBER
FIRST_LOAD_TIME	VARCHAR2(19)	FIRST_LOAD_TIME	VARCHAR2(19)
INVALIDATIONS	NUMBER	INVALIDATIONS	NUMBER
PARSE_CALLS	NUMBER	PARSE_CALLS	NUMBER
DISK_READS	NUMBER	DISK_READS	NUMBER
BUFFER_GETS	NUMBER	BUFFER_GETS	NUMBER
ROWS_PROCESSED	NUMBER	ROWS_PROCESSED	NUMBER
COMMAND_TYPE	NUMBER	COMMAND_TYPE	NUMBER
OPTIMIZER_MODE	VARCHAR2(10)	OPTIMIZER_MODE	VARCHAR2(10)
OPTIMIZER_COST	NUMBER	OPTIMIZER_COST	NUMBER
PARSING_USER_ID	NUMBER	PARSING_USER_ID	NUMBER
PARSING_SCHEMA_ID	NUMBER	PARSING_SCHEMA_ID	NUMBER
KEPT_VERSIONS	NUMBER	KEPT_VERSIONS	NUMBER
ADDRESS	RAW(4)	ADDRESS	RAW(4)
TYPE_CHK_HEAP	RAW(4)	TYPE_CHK_HEAP	RAW(4)
HASH_VALUE	NUMBER	HASH_VALUE	NUMBER
		PLAN_HASH_VALUE	NUMBER
CHILD_NUMBER	NUMBER	CHILD_NUMBER	NUMBER
MODULE	VARCHAR2(64)	MODULE	VARCHAR2(64)
MODULE_HASH	NUMBER	MODULE_HASH	NUMBER
ACTION	VARCHAR2(64)	ACTION	VARCHAR2(64)
ACTION_HASH	NUMBER	ACTION_HASH	NUMBER
SERIALIZABLE_ABORTS	NUMBER	SERIALIZABLE_ABORTS	NUMBER
OUTLINE_CATEGORY	VARCHAR2(64)	OUTLINE_CATEGORY	VARCHAR2(64)
		CPU_TIME	NUMBER
		ELAPSED_TIME	NUMBER
		OUTLINE_SID	NUMBER
		CHILD_ADDRESS	RAW(4)
		SQLTYPE	NUMBER

REMOTE	VARCHAR2(1)
OBJECT_STATUS	VARCHAR2(19)
LITERAL_HASH_VALUE	NUMBER
LAST_LOAD_TIME	VARCHAR2(19)
IS_OBSOLETE	VARCHAR2(1)
CHILD_LATCH	NUMBER

V$SQL_PLAN

The cost based optimizer is more and more taking current conditions into account when parsing a SQL statement and deciding on an access plan. Examples of such current conditions are system statistics, bind variable peeking and dynamic sampling. An explain plan without the exact same conditions in effect can easily result in a different access plan – as has been shown with the bind variable peeking. As it may be difficult, if not impossible, to know all the conditions in effect at execution time, let alone reproduce them in an explain session, "explain plan …" will become increasingly unreliable, if not useless, for the purpose of showing the access plan that was, or will be, used. In order to know the actual access plan used when executing a SQL statement, one will need to enable a SQL trace or query the new V$SQL_PLAN table.

V$SQL_PLAN has essentially the same columns as plan_table. The differences are highlighted:

V$sql_plan		Plan_table	
ADDRESS	RAW(4)	**STATEMENT_ID**	VARCHAR2(30)
HASH_VALUE	NUMBER	**TIMESTAMP**	DATE
CHILD_NUMBER	NUMBER	**REMARKS**	VARCHAR2(80)
OPERATION	VARCHAR2(30)	OPERATION	VARCHAR2(30)
OPTIONS	VARCHAR2(30)	OPTIONS	VARCHAR2(255)
OBJECT_NODE	VARCHAR2(10)	OBJECT_NODE	VARCHAR2(128)
OBJECT#	NUMBER		
OBJECT_OWNER	VARCHAR2(30)	OBJECT_OWNER	VARCHAR2(30)
OBJECT_NAME	VARCHAR2(64)	OBJECT_NAME	VARCHAR2(30)
		OBJECT_INSTANCE	NUMBER
		OBJECT_TYPE	VARCHAR2(30)
OPTIMIZER	VARCHAR2(20)	OPTIMIZER	VARCHAR2(255)
ID	NUMBER	ID	NUMBER
PARENT_ID	NUMBER	PARENT_ID	NUMBER
DEPTH	NUMBER		
POSITION	NUMBER	POSITION	NUMBER
SEARCH_COLUMNS	NUMBER	SEARCH_COLUMNS	NUMBER
COST	NUMBER	COST	NUMBER
CARDINALITY	NUMBER	CARDINALITY	NUMBER
BYTES	NUMBER	BYTES	NUMBER
OTHER_TAG	VARCHAR2(35)	OTHER_TAG	VARCHAR2(255)
PARTITION_START	VARCHAR2(5)	PARTITION_START	VARCHAR2(255)
PARTITION_STOP	VARCHAR2(5)	PARTITION_STOP	VARCHAR2(255)
PARTITION_ID	NUMBER	PARTITION_ID	NUMBER
OTHER	VARCHAR2(4000)	OTHER	LONG
DISTRIBUTION	VARCHAR2(20)	DISTRIBUTION	VARCHAR2(30)
CPU_COST	NUMBER	CPU_COST	NUMBER
IO_COST	NUMBER	IO_COST	NUMBER
TEMP_SPACE	NUMBER	TEMP_SPACE	NUMBER
ACCESS_PREDICATES	VARCHAR2(4000)	ACCESS_PREDICATES	VARCHAR2(4000)
FILTER_PREDICATES	VARCHAR2(4000)	FILTER_PREDICATES	VARCHAR2(4000)

Since V$SQL_PLAN has much the same columns as PLAN_TABLE you can practically use your explain plan formatting script. – just replace statement_id with the HASH_VALUE-CHILD_NUMBER combination as identifier. In the example below some information, IO and CPU cost for example – has been omitted so that the lines fit without wrapping. See Appendix A for the SQL.

```
Statement_id  cost   card  operation                          predicates
3572258809-0   255        SELECT STATEMENT
               255    250   TABLE ACCESS BY INDEX ROWID PS_JOB2
                 4    250     INDEX RANGE SCAN PSBJOB2         Access: "COMPANY"='CCC'
                                                                  AND "PAYGROUP"='FGH'
```

You can not, however, use the new DBMS_XPLAN.DISPLAY procedure to format plan in V$SQL_PLAN unless you create a view on top of it. An example of such a view is attached in Appendix B.

V$SQL_PLAN_STATISTICS

provides execution statistics at the row source level, both for the most recent execution and cumulatively.

Name	Type
ADDRESS	RAW(4)
HASH_VALUE	NUMBER
CHILD_NUMBER	NUMBER
OPERATION_ID	NUMBER
EXECUTIONS	NUMBER
LAST_STARTS	NUMBER
STARTS	NUMBER
LAST_OUTPUT_ROWS	NUMBER
OUTPUT_ROWS	NUMBER
LAST_CR_BUFFER_GETS	NUMBER
CR_BUFFER_GETS	NUMBER
LAST_CU_BUFFER_GETS	NUMBER
CU_BUFFER_GETS	NUMBER
LAST_DISK_READS	NUMBER
DISK_READS	NUMBER
LAST_DISK_WRITES	NUMBER
DISK_WRITES	NUMBER
LAST_ELAPSED_TIME	NUMBER
ELAPSED_TIME	NUMBER

In order to get plan statistics collected for this view, the initialization parameter STATISTICS_LEVEL must be set to ALL. This can be done dynamically for a session and even the system. STATSPACK.SNAP for example does that for snap levels above 5

By joining V$SQL_PLAN and V$SQL_PLAN_STATISTICS execution details are available down to the row source level:

```
statement_id  card  operation                          rows  cr gets  cu gets  reads  elapsed
1983399701-0        SELECT STATEMENT
                1   SORT GROUP BY                       183    7,641        0  2,337     9.42
                1     TABLE ACCESS BY INDEX ROWID PS_PAY_CHECK5 1,206  7,641  0  2,337  9.40
               43       NESTED LOOPS                   6,459    1,730        0    513     1.90
               50         TABLE ACCESS BY INDEX ROWID PS_JOB5  530  535  0  466  1.73
               50           INDEX RANGE SCAN PSBJOB5      530      9        0      9     0.03
                1         INDEX RANGE SCAN PS_PAY_CHECK5  5,928    1,195        0     47     0.12
```

Comparing the optimizer's cardinality estimates with the actual row counts can aid in tuning. After collecting histograms the optimizer chose the plan below. Prior to Oracle9*i* that analysis would have required to re-run the SQL with SQL trace enabled and using tkprof to get the rows source cardinalities and elapsed times. Note that the statement id (hash_value-child_number) is the same since the tuning action changed the statistics, not the SQL.

```
statement_id  card  operation                          rows  cr gets  cu gets  reads  elapsed
1983399701-0        SELECT STATEMENT
                1   SORT GROUP BY NOSORT                183    3,810        0  1,841     0.92
                2     TABLE ACCESS BY INDEX ROWID PS_JOB5 1,206  3,810  0  1,841  0.91
               85       NESTED LOOPS                   1,621    2,627        0  1,519     0.68
               40         TABLE ACCESS BY INDEX ROWID PS_PAY_CHECK5  414  2,174  0  1,519  0.64
```

```
   200          INDEX RANGE SCAN PS_PAY_CHECK5              2,120       59        0        0    0.01
     2        INDEX RANGE SCAN PSBJOB5                      1,206      453        0        0    0.02
```

NEW TREATMENT OF TABLES WITHOUT STATISTICS

In Oracle 8 and prior, the optimizer would use default values if statistics where missing for some tables in a query. Oracle 9*i* Release 2 introduces the option of dynamic sampling of tables at parse time to gather missing statistics or to improve the selectivity and cardinality estimates for predicates where existing statistics do not provide an accurate estimate. Oracle 9.0.1 also has a dynamic sampling feature, but there it must be activated by changing an undocumented parameter and it does not have the fine grained control of different levels as 9.2 provides. The following is taken mostly from [4] and applies to Oracle 9.2 only.

DYNAMIC SAMPLING

The purpose of dynamic sampling is to improve server performance by determining more accurate selectivity and cardinality estimates. More accurate selectivity and cardinality estimates allow the optimizer to produce better performing plans.

For a query that normally completes quickly (in less than a few seconds), you will not want to incur the cost of dynamic sampling. However, dynamic sampling can be beneficial under any of the following conditions:

- A better plan can be found using dynamic sampling.
- The sampling time is a small fraction of total execution time for the query.
- The query will be executed many times.

Dynamic sampling can be applied to a subset of a single table's predicates and combined with standard selectivity estimates of predicates for which dynamic sampling is not done.[4]. Dynamic sampling is controlled with the parameter OPTIMIZER_DYNAMIC_SAMPLING, or with the DYNAMIC_SAMPLING hint, with values between 0 and 10. Setting OPTIMIZER_FEATURES_ENABLE at the same time to a version prior to 9.0.2 turns off dynamic sampling, however.

Meaning of the dynamic sampling values:

0 No dynamic sampling will be done.

1 Dynamic sampling will be performed if

There is more than one table in the query.

Some table has not been analyzed, has no indexes, and a relatively expensive table scan would be required for this unanalyzed table.

2 Apply dynamic sampling to all unanalyzed tables.
The number of blocks sampled is the default number (32) of dynamic sampling blocks.

3 Same as 2 plus all tables for which standard selectivity estimation used a guess for some predicate that is a potential dynamic sampling predicate.
The number of blocks sampled is the default number of dynamic sampling blocks.

4 Same as 3 plus all tables that have single-table predicates that reference 2 or more columns.
The number of blocks sampled is the default number of dynamic sampling blocks.

5-9 Same as 4 but with increasing multiples of the default number of sampling blocks sampled: 2, 4, 8, 32, and 128.

10 Same as 4 but with a full scan rather than a sample.

If dynamic sampling 5 through 9 results in a number of blocks to be sampled that is half the total blocks of the table, or more, Oracle performs a full scan rather than a sample.

Dynamic sampling is repeatable if no rows have been inserted, deleted, or updated in the table being sampled.

APPENDIXES

APPENDIX A

SQL to join plans in V$SQL_PLAN with execution statistics in V$SQL_PLAN_STATISTICS

```
select p.hash_value||'-'||p.child_number "statement_id"
 , p.cost "cost"
 , p.io_cost "io cost"
 , p.cpu_cost "cpu cost"
 , p.cardinality "card"
 , lpad(' ',2*(level-1))||p.operation||' '||
   p.options||' '||
   p.object_name||
   decode(p.partition_start,null,' ',':')||
   translate(p.partition_start,'(nrumbe','(nr')||
   decode(p.partition_stop,null,' ','-')||
   translate(p.partition_stop,'(nrumbe','(nr') "operation"
 , p.position "pos"
 , (select s.last_output_rows from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id) "rows"
 , (select s.last_cr_buffer_gets from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id)  "cr gets"
 , (select s.last_cu_buffer_gets from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id)  "cu gets"
 , (select s.last_disk_reads from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id)  "reads"
 , (select s.last_disk_writes from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id)  "writes"
 , (select round(s.last_elapsed_time/1000,2) from v$sql_plan_statistics s
        where s.address=p.address and s.operation_id=p.id)  "elapsed
from v$sql_plan p
where p.address in (select address from v$sql_plan
        where id=0 and cost is not null)
  and p.address in (select address from v$sql_plan
        where object_owner = sys_context('USERENV','CURRENT_SCHEMA'))
start with p.id=0
connect by prior p.id = p.parent_id
 and prior p.address = p.address
```

APPENDIX B

View on V$SQL_PLAN which can be used with DBMS_XPLAN.DISPLAY:

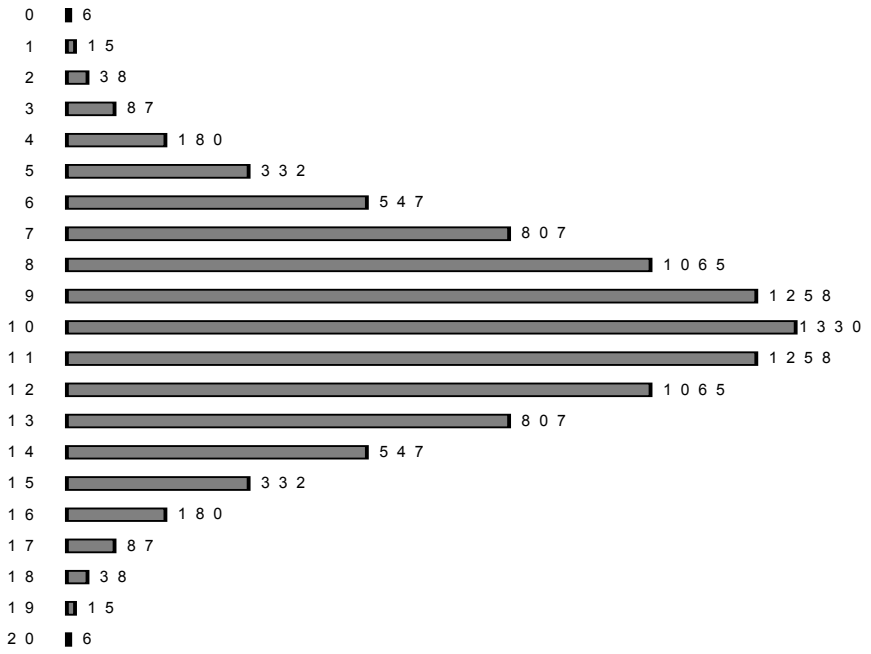
```
create view v$plan_table (          as select
statement_id                        hash_value||'-'||child_number
  , timestamp                         , trunc(sysdate, 'MI')
  , remarks                           , null
  , operation                         , operation
  , options                           , options
  , object_node                       , object_node
  , object_owner                      , object_owner
  , object_name                       , object_name
  , object_instance                   , null
  , object_type                       , null
  , optimizer                         , optimizer
  , search_columns                    , search_columns
  , id                                , id
  , parent_id                         , parent_id
  , position                          , position
  , cost                              , cost
  , cardinality                       , cardinality
  , bytes                             , bytes
  , other_tag                         , other_tag
  , partition_start                   , partition_start
  , partition_stop                    , partition_stop
  , partition_id                      , partition_id
  , other                             , other
  , distribution                      , distribution
  , cpu_cost                          , cpu_cost
  , io_cost                           , io_cost
  , temp_space                        , temp_space
  , access_predicates                 , access_predicates
  , filter_predicates )               , filter_predicates
                                    from v$sql_plan
```

APPENDIX C

Value distribution of table used in bind value peek example.

```
select n1, count(0)
from hist3 group by n1;
```

N1	COUNT(0)
0	6
1	15
2	38
3	87
4	180
5	332
6	547
7	807
8	1065
9	1258
10	1330
11	1258
12	1065
13	807
14	547
15	332
16	180
17	87
18	38
19	15
20	6



APPENDIX D

Describe peek:

Name	Null?	Type	table	column	NDV	density	lo	hi	bkts
N1	NOT NULL	NUMBER	PEEK	N1	1,001	9.9900E-04	1	1020	1
N2	NOT NULL	NUMBER	PEEK	N2	25	4.9950E-04	1	25	24
N3	NOT NULL	NUMBER	PEEK	N3	25	4.0000E-02	0	24	1
DT	NOT NULL	DATE	PEEK	DT	25	4.0000E-02	AD 2003-10-22	AD 2003-11-15	1
CH	NOT NULL	VARCHAR2(2000)	PEEK	CH	61	1.6393E-02	0**********	Z**********	1

table	% free	% used	rows	blks	empty	avg row
PEEK	10	40	1,001	500	0	995

table	index	column	NDV	CLUF	#LB	lvl	#LB/K	#DB/K
PEEK	PEEK_1		1,001	700	6	1	1	1
		N2	25					
		N1	1,001					
	PEEK_2		350	750	5	1	1	2
		N3	25					
		N2	25					

table	column	EP	value
PEEK	N2	521	1
PEEK	N2	541	2
PEEK	N2	561	3
…			
PEEK	N2	981	24
PEEK	N2	1001	25

value 1 occurs 521 times and each of the values 2-25 occurs 20 times

REFERENCES

1. Note:62337.1: *Reference Note for Init.Ora Parameter "Optimizer_Features_Enable"*. metalink.oracle.com
2. Note:149560.1: *Collect and Display System Statistics (Cpu and Io) for Cbo Usage*. metalink.oracle.com
3. Note:153761.1: *Scaling the System to Improve Cbo Optimizer*. metalink.oracle.com
4. *Oracle 9i Database Performance Tuning Guide and Reference*. 2002: Oracle Corporation.
5. Wolfgang Breitling: *Fallacies of the Cost Based Optimizer*; proceedings of the 2003 Hotsos Symposium on Oracle® System Performance, 2003. Dallas, TX. http://www.centrexcc.com
6. Wolfgang Breitling: *A Look under the Hood of Cbo - the 10053 Event*; proceedings of the 2003 Hotsos Symposium on Oracle® System Performance, 2003. Dallas, TX. http://www.centrexcc.com