TUNING BY CARDINALITY FEEDBACK
METHOD AND EXAMPLES

Wolfgang Breitling[i]
Centrex Consulting Corporation

The presentation introduces a method of tuning which is based on the premise that whenever the CBO chooses a bad plan it can be traced back to an error in the estimation of the cardinality of one or more row sources.

THE EMPIRICAL BASIS FOR THE METHOD

As the caption[1] implies, the method grew over time out of observations and the need to tune SQL without being able to change it. At the 2004 Hotsos Symposium I presented a testcase to demonstrate the danger of gathering statistics indiscriminately:[2]

A. Baseline

B. Insert 1 row into 40,000 row table

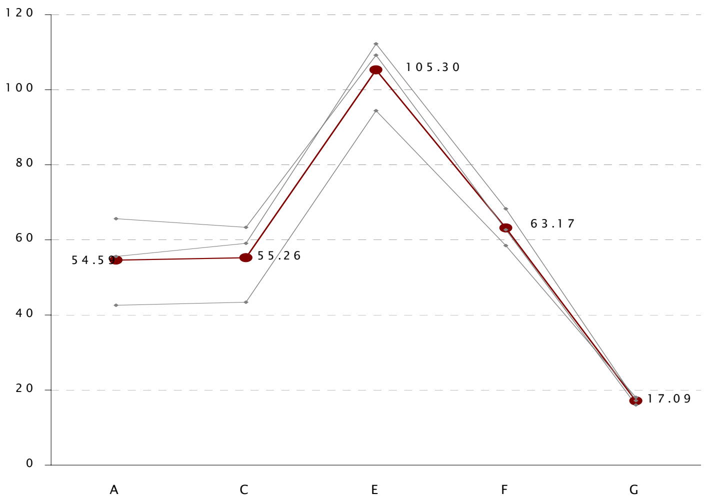
C. Re-execute SQL

D. Analyze 40,000 row table

E. Re-execute SQL

F. Execute SQL with OICA=25, OIC=90[3]

G. Execute SQL after TCF tuning



The SQL at all five execution points is identical. All that changes between executions are the table statistics, or, at point F, optimizer parameters.

As execution points A, C, E, and eventually F show, the change in performance was not due to the change in data volume, but entirely due to changes in statistics. The testcase also clearly demonstrates the power of statistics and how important it is to have the **right** statistics. Note that right is not synonymous to fresh or up-to-date.

OBSERVATION

IF AN ACCESS PLAN IS NOT OPTIMAL IT IS BECAUSE THE CARDINALITY ESTIMATE FOR ONE OR MORE OF THE ROW SOURCES IS GROSSLY INCORRECT.

Just recently I learned that this observation is corroborated by members of Oracle's optimizer development team[1]:

"● The most common reason for poor execution plans with perceived "good" statistics is inaccurate row count estimates
 – This leads to bad access path selection
 – This leads to bad join method selection
 – This leads to bad join order selection

● In summary one bad row count estimate can cascade into a very poor plan"

The usual suggestion – if not to say knee-jerk reaction – is to re-analyze all tables, possibly with a higher sampling percentage. However,

a) Remember Dave Ensor's paradox: "It is only safe to gather statistics when to do so will make no difference" viz the performance chart above for time points B-C-D-E.

[1] em•pir•i•cal adj. "originating in or based on observation or experience." (http://www.m-w.com/dictionary/empirical)

[2] The chart show the times captured for three separate runs and their average times, which are highlighted and labeled.

[3] This step was only included to muffle the many proponents of this kind of "silver bullet".

b) Outdated statistics on base tables and columns are only **one** possible reason for inaccurate cardinality estimates, especially of intermediate join results

c) There are many reasons – i.e. assumptions – in the optimizer's logic which lead to over- or under-inflated estimates when violated, despite accurate statistics on the base tables and columns.

d) Violation of the predicate independence assumption[2] in particular, and its cousin, the join uniformity assumption, lead to cardinality estimates which are orders of magnitude too low, resulting in disastrous NL join plans. And setting optimizer_index_cost_adj to a value < 100 does nothing in those cases except entrench the NL choice even more firmly.

Note that the inverse of the above observation

- bad plan => cardinality estimate is wrong

do **not** follow from logic:

- cardinality estimate is wrong => bad plan
- cardinality estimate is correct => good plan

However, I do formulate the

CONJECTURE

THE CBO DOES AN EXCELLENT JOB OF FINDING THE BEST ACCESS PLAN FOR A GIVEN SQL PROVIDED IT IS ABLE TO ACCURATELY ESTIMATE THE CARDINALITIES OF THE ROW SOURCES IN THE PLAN

Unlike other tuning methodologies and practices which often attempt to coerce the optimizer into a particular access plan, tuning by cardinality feedback (TCF) looks for discrepancies between estimated and real row source cardinalities of an execution plan and strives to find what caused the CBO to err in calculating the estimates and choose a (presumably) sub-optimal access plan. Once the answer to that question is found, the next goal is to find a way to remedy the reason for the miscalculation, but ultimately get out of the way and let the CBO do its job again, trusting it to find a better plan itself based on the corrected, more accurate estimates.

The methodology is thus not dissimilar to that of profiles generated by DBMS_SQLTUNE. Profiles give the CBO adjustment factors (see page 11) to correct the row source cardinality estimates while TCF aims to give the CBO information such that the row source cardinality estimates become more accurate in the first place.

TUNING BY CARDINALITY FEEDBACK

THE METHOD

The SQL for it – and all the other examples – can be found in the appendix.

❶ List the explain plan with the cardinality projections
– from explain or, preferably, from v$sql_plan

❷ Get the actual row counts
– from a SQL trace or from v$sql_plan_statistics.
Make sure the actual plan is identical to the explain plan!
Something that is automatically the case if you use v$sql_plan and v$sql_plan_statistics.

❸ Look for the first (innermost) row source where the ratio of actual/estimated cardinality is orders of magnitude – usually at least in the 100s

```
   Ratio      Rows      card  operation
                           2  SELECT STATEMENT
                  2        2    SORT GROUP BY
              6,274               FILTER
   504.6     13,120       26        HASH JOIN
   534.9    208,620      390          HASH JOIN
     1.0         15       15            TABLE ACCESS FULL PS_RETROPAYPGM_TBL
   858.1     44,621       52            NESTED LOOPS
   353.3     14,131       40              HASH JOIN
     1.0          5        5                TABLE ACCESS FULL PS_PAY_CALENDAR
     3.0     40,000   13,334                TABLE ACCESS FULL WB_JOB
     1.6     44,621   27,456              TABLE ACCESS BY INDEX ROWID WB_RETROPAY_EARNS
     2.7     74,101   27,456                INDEX RANGE SCAN WB0RETROPAY_EARNS
     1.0     13,679   13,679          TABLE ACCESS FULL PS_RETROPAY_RQST
              9,860        1        SORT AGGREGATE
              4,930        1          FIRST ROW
              4,930        1            INDEX RANGE SCAN (MIN/MAX) WB_JOB
             20,022        1        SORT AGGREGATE
              7,750        1          FIRST ROW
             10,011        1            INDEX RANGE SCAN (MIN/MAX) WB_JOB
```

❹ Find the predicates in the SQL for the tables that contribute to the row source with the miscalculated cardinality and look for violated assumptions:

- Uniform distribution
- Predicate independence
- Join uniformity

The innermost row source with the largest ratio of discrepancy is the hash join of PS_PAY_CALENDAR and WB_JOB. So we are checking the column statistics for the two tables

```
table              column                        NDV     density lo          hi          bkts
PS_PAY_CALENDAR    COMPANY                        11  9.0909E-02 ACE         TES            1
                   PAYGROUP                       15  6.6667E-02 ACA         TEP            1
                   PAY_END_DT                    160  6.2500E-03 1998-01-18  2004-02-22     1
                   RUN_ID                        240  4.1667E-03             PP2            1
                   PAY_OFF_CYCLE_CAL               2  5.0000E-01 N           Y              1
                   PAY_CONFIRM_RUN                 2  5.0000E-01 N           Y              1


WB_JOB             EMPLID                     26,167  3.8216E-05 000036      041530         1
                   EMPL_RCD#                       1  1.0000E+00 0           0              1
                   EFFDT                          10  1.0000E-01 1995-01-01  2004-02-01     1
                   EFFSEQ                          3  3.3333E-01 1           3              1
                   COMPANY                        10  1.0000E-01 ACE         TES            1
                   PAYGROUP                       14  7.1429E-02 ACA         TEP            1
```

If emplid, effdt and effseq, which form a unique key on wb_job[4], **were** independent then wb_job would need to have 26,167 * 10 * 3 = 785,010 rows. To counteract the estimate miscalculation due to the violated predicate independence assumption, we neutralize the effdt and effseq cardinalities by setting them to 1.

While this ultimately is probably an accurate root cause analysis, the mechanics of how that results in the incorrect row source cardinality estimate is more complicated and may contain an optimizer deficiency/bug. If it were a simple case of predicate dependency then using optimizer_dynamic_sampling=4 (or higher) should detect it, which it does not.

In this case there is virtually no danger of side effects as it is extremely unlikely that EFFDT, much less EFFSEQ, will be used by themselves as predicates in a query.

The optimizer uses sometimes density, sometimes NDV – and other times yet other statistics – when estimating predicate selectivity.[3]

[4] Technically, there is a 4th column in the unique key, but that has the same value for all rows in this instance

When modifying column statistics I try to leave the real NDV in place if possible and only change the density. This makes it “obvious” that the statistics were modified after gathering.

Note: set_column_stats(distcnt=>) not only changes num_distinct, but also density to 1/distcnt, while
set_column_stats(density=>) only changes density, leaving num_distinct unchanged.

Adjust the column statistics to counteract the violated assumption(s):

```
DBMS_STATS.SET_COLUMN_STATS('SCOTT','WB_JOB','EFFDT',density => 1);
DBMS_STATS.SET_COLUMN_STATS('SCOTT','WB_JOB','EFFSEQ',density => 1);
```

```
table             column                          NDV     density lo          hi           bkts
WB_JOB            EMPLID                       26,167  3.8216E-05 000036      041530          1
                  EMPL_RCD#                         1  1.0000E+00 0           0               1
                  EFFDT                            10  1.0000E+00 1995-01-01  2004-02-01      1
                  EFFSEQ                            3  1.0000E+00 1           3               1
                  COMPANY                          10  1.0000E-01 ACE         TES             1
                  PAYGROUP                         14  7.1429E-02 ACA         TEP             1
```

The plan with row source cardinality estimates, actuals and ratios after modifying the statistics:

```
   Ratio      Rows      card operation
                           2 SELECT STATEMENT
                 2         2   SORT GROUP BY
             6,274               FILTER
    17.5    13,120       750       HASH JOIN
     1.0        15        15         TABLE ACCESS FULL PS_RETROPAYPGM_TBL
    28.1    42,054     1,499         HASH JOIN
    29.8    44,621     1,499           HASH JOIN
     9.9    14,130     1,429             HASH JOIN
     1.0         5         5               TABLE ACCESS FULL PS_PAY_CALENDAR
     1.0    40,000    40,000               TABLE ACCESS FULL WB_JOB
     4.5   122,813    27,456             TABLE ACCESS FULL WB_RETROPAY_EARNS
     1.0    13,679    13,679           TABLE ACCESS FULL PS_RETROPAY_RQST
            11,212         1       SORT AGGREGATE
             5,606         1         FIRST ROW
             5,606         1           INDEX RANGE SCAN (MIN/MAX) WB_JOB
            17,374         1       SORT AGGREGATE
             6,418         2         FIRST ROW
             8,687         2           INDEX RANGE SCAN (MIN/MAX) WB_JOB
```

Comments on the plan after adjusting the column statistics:

1. The ratios of actual/estimated are much smaller
2. The cardinalities of 4 of the 5 base table row sources are accurately estimated
3. Even though the estimates for the cardinalities of PS_PAY_CALENDAR and WB_JOB were correct, the CBO underestimated the cardinality of their join by a factor of 10, suggesting a violation of the join uniformity assumption.

It may appear that TCF is only about adjusting statistics. This is not strictly true. It is about addressing the cause for the cardinality estimate miscalculation. Virtually always this is caused by missing information. The additional information may come from an additional index, or from a histogram (which in turn changes the statistics). In this case, the information about the predicate dependency can not be given directly to the CBO. By faking the column statistics, two wrongs make a right – in this particular case.

REAL-LIFE EXAMPLES USING TCF

Unlike the demo case, which was an artificial test case – albeit based on a real-life case – the following examples are actual, recent tuning cases. For the record, examples 1-3 are from a system running Oracle 9.2.0.6 SE[5] with system statistics:

SREADTIM	1.971
MREADTIM	2.606
CPUSPEED	618
MBRC	8
MAXTHR	-1
SLAVETHR	-1

Example 4 is from an Oracle 9.2.0.6 EE system without system statistics.

The explain plan and execution numbers are from v$sql_plan and v$sql_plan_statistics. The operation detail with the gross cardinality mismatch is highlighted. Note that on some platforms, all that I have worked with, you need to set statistics_level=all in order to get execution details from v$sql_plan_statistics. Contrary to my general advice not to pay attention to the cost, I **did** include the plan cost in the displays for reasons which will become clear later.

In the three examples there are two (E1 and E3) where an actual row count is 0. This is another indicator to watch for – of course it can be viewed as a special case of the ratio indicator since any ratio with 0 as the denominator is out of bounds. Provided the row source cardinality is **consistently** 0, then either that part of the SQL and plan is obsolete, or the plan could benefit from executing that part early and reduce or eliminate subsequent work.

EXAMPLE 1

The first comparison of the optimizer's cardinality estimates and the actual row source counts come from the dynamic performance views V$SQL_PLAN and V$SQL_PLAN_STATISTICS (see script v$xplain in the appendix). In order to get the execution statistics it is necessary to set STATISTICS_LEVEL=ALL. The second row source operation counts is taken from the event 10046 trace file (the Cary trace ☺) processed with tkprof. As with the prior method, in order to get the individual row source execution data (the data in the brackets behind the operation), STATISTICS_LEVEL=ALL needs to be set. The trace file **does** contain the "Rows" counts though, even with STATISTICS_LEVEL=TYPICAL or BASIC, provided the cursor is closed before the trace is.

Getting the comparison data from the dynamic performance views has several advantages over the tkprof alternative:

- No doubt whether the explained plan and the actual execution plan are the same.
- Both, the cardinality estimates and the cardinality actuals are in the same output. No need to assemble and align them.
- No need to start a trace, find the trace file on the OS, format it – and then find that for one reason or another it does not contain the execution plan details (the STAT lines for the cursor).

Of course, the dynamic view way has its own pitfalls. Unless you have the system to yourself, you have to act fast. On a busy system the plan and plan_statistics details can age out of the shared pool quickly. It is not uncommon to still find the SQL in V$SQL, but the associated plan is no longer in V$SQL_PLAN.

[5] Therefore no parallel execution statistics (maxthr and slavethr)

V$SQL_PLAN and V$SQL_PLAN_STATISTICS[6]:

```
  COST        CARD  operation                                ELAPSED     ROWS    CR_GETS

   220              SELECT STATEMENT
   220      12,516    HASH JOIN                                0.060        0      2,362
     3          15      TABLE ACCESS FULL PS_PRCSRECUR         0.000       15          3
   214      34,057      TABLE ACCESS FULL PSPRCSQUE            0.060        0      2,359
```

TKPROF:

```
   Rows  Row Source Operation

      0  HASH JOIN  (cr=2362 r=0 w=0 time=59799 us)
     15   TABLE ACCESS FULL PS_PRCSRECUR (cr=3 r=0 w=0 time=171 us)
      0   TABLE ACCESS FULL PSPRCSQUE (cr=2359 r=0 w=0 time=58218 us)
```

Tuning of Example 1

The following actions were taken when trying to tune the SQL in example 1.

❶ The SQL is using a highly selective value for PSPRCSQUE. PRCSJOBSEQ in the predicate but there is no usable index on it. Create an index on psprcsque

create index uc_psprcsque_ix1 on psprcsque(prcsjobseq,recurname)

However, that did not change the plan. Of course, with access to the SQL one could force the use of the index. But that is not possible in this case.

❷ Create a (frequency) histogram on prcsjobseq

That did change the cardinality estimates, but not enough to change the plan:

```
  COST   CARD  operation                               ELAPSED    ROWS   CR_GETS

   209         SELECT STATEMENT
   209    236    HASH JOIN                               0.040       0     2,362
     3     15      TABLE ACCESS FULL PS_PRCSRECUR        0.000      15         3
   206    641      TABLE ACCESS FULL PSPRCSQUE           0.040       0     2,359
```

❸ Modifying the PSPRCSQUE.PRCSJOBSEQ statistics finally did - in conjunction with the index

DBMS_STATS.SET_COLUMN_STATS(USER,'PSPRCSQUE','PRCSJOBSEQ',DISTCNT=>250);

```
  COST  CARD  operation                                ELAPSED    ROWS   CR_GETS

    40        SELECT STATEMENT
    40     3    HASH JOIN                                0.010       0       112
    37     9      TABLE ACCESS BY INDEX ROWID PSPRCSQUE                      112
     3   109        INDEX RANGE SCAN UC_PSPRCSQUE_IX1                  112        49
     3    15      TABLE ACCESS FULL PS_PRCSRECUR         0.000       0         0
```

❹ Changing the DISTCNT of PSPRCSQUE.PRCSJOBSEQ even more results in a still better plan[7]

DBMS_STATS.SET_COLUMN_STATS(USER,'PSPRCSQUE','PRCSJOBSEQ',DISTCNT=>1000);

```
  COST  CARD  OPERATION                                ELAPSED    ROWS   CR_GETS

    14        SELECT STATEMENT
    14     1   NESTED LOOPS                              0.010       0       112
    12     2     TABLE ACCESS BY INDEX ROWID PSPRCSQUE                       112
     3    27       INDEX RANGE SCAN UC_PSPRCSQUE_IX1                   112        49
     2     1     TABLE ACCESS BY INDEX ROWID PS_PRCSRECUR  0.000       0         0
     1     1       INDEX UNIQUE SCAN PS_PRCSRECUR                                0
```

[6] The SQL for formatting v$sql_plan and v$sql_plan_statistics into the report below can be found in the appendix.

[7] From the report below alone it is not obvious, or even apparent, why this is a better plan. All the execution statistics seem identical. However, a close look at more detailed execution statistics warrants this assertion.

Example 2

```
   COST          CARD  operation                                      ELAPSED    ROWS    CR_GETS
    546                SELECT STATEMENT
    546         1,065    HASH JOIN                                      0.240       1      6,922
    201         1,101      TABLE ACCESS FULL PSPRCSQUE                  0.030       1      2,359
    341        36,284      TABLE ACCESS FULL PSPRCSPARMS                0.080  38,539      4,563
```

```
   Rows  Row Source Operation
      1  HASH JOIN  (cr=6922 r=0 w=0 time=236770 us)
      1   TABLE ACCESS FULL PSPRCSQUE   (cr=2359 r=0 w=0 time=30341 us)
  38539   TABLE ACCESS FULL PSPRCSPARMS (cr=4563 r=0 w=0 time=77536 us)
```

Tuning of Example 2

❶ As with example 1, the predicate value for PSPRCSQUE.RUNSTATUS is highly selective and this time there is a, not ideal but reasonably usable, index on PSPRCSQUE. However, the optimizer is not using it

❷ Create a (frequency) histogram on PSPRCSQUE.RUNSTATUS

With the changed column statistics, the optimizer did use the index

```
Table          column          NDV     density  nulls   lo   hi  av lg   bkts
PSPRCSQUE      RUNSTATUS        11  1.3764E-05      0    1    9      3     10
```

```
  COST  CARD  operation                                ELAPSED    ROWS   CR_GETS
   133        SELECT STATEMENT
   133     1    NESTED LOOPS                             0.020       1        16
   132     1      TABLE ACCESS BY INDEX ROWID PSPRCSQUE                       13
   131     1        INDEX SKIP SCAN PSAPSPRCSQUE                              12
     2     1      TABLE ACCESS BY INDEX ROWID PSPRCSPARMS  0.000       1         3
     1     1        INDEX UNIQUE SCAN PS_PSPRCSPARMS                           2
```

Example 3

```
   COST          CARD  operation                                      ELAPSED    ROWS    CR_GETS
    221                SELECT STATEMENT
    221           108    SORT ORDER BY                                  0.040       0      2,363
                           FILTER                                                          2,363
    220           108        HASH JOIN                                                     2,363
      5             8          MERGE JOIN CARTESIAN                     0.000       7          4
      3             1            TABLE ACCESS BY INDEX ROWID PS_SERVERCATEGORY      1          2
      2             1              INDEX RANGE SCAN PS_SERVERCATEGORY                          1
      2             8            BUFFER SORT                                        7          2
      3             8              TABLE ACCESS BY INDEX ROWID PS_SERVERCLASS         2
      2             8                INDEX RANGE SCAN PS_SERVERCLASS                           1
    215           108          TABLE ACCESS FULL PSPRCSQUE              0.040       0      2,359
                         FILTER                                         0.000       0          0
      3             1      TABLE ACCESS BY INDEX ROWID PSPRCSQUE                      0
      2             5        INDEX RANGE SCAN PSDPSPRCSQUE                                     0
   Rows  Row Source Operation
-------  ---------------------------------------------------
      0  SORT ORDER BY     (cr=2363 r=0 w=0 time=39950 us)
      0   FILTER    (cr=2363 r=0 w=0 time=39930 us)
      0    HASH JOIN       (cr=2363 r=0 w=0 time=39926 us)
      7     MERGE JOIN CARTESIAN  (cr=4 r=0 w=0 time=213 us)
      1      TABLE ACCESS BY INDEX ROWID PS_SERVERCATEGORY  (cr=2 r=0 w=0 time=70 us)
      1       INDEX RANGE SCAN PS_SERVERCATEGORY     (cr=1 r=0 w=0 time=40 us)
      7      BUFFER SORT   (cr=2 r=0 w=0 time=97 us)
      7       TABLE ACCESS BY INDEX ROWID PS_SERVERCLASS    (cr=2 r=0 w=0 time=50 us)
      7        INDEX RANGE SCAN PS_SERVERCLASS (cr=1 r=0 w=0 time=25 us)
      0     TABLE ACCESS FULL PSPRCSQUE (cr=2359 r=0 w=0 time=39040 us)
      0    FILTER
      0     TABLE ACCESS BY INDEX ROWID PSPRCSQUE
      0      INDEX RANGE SCAN PSDPSPRCSQUE
```

TUNING OF EXAMPLE 3

❶ By the time we got to tune this SQL, there was nothing left to do.

The SQL had gotten tuned as well by the actions to tune the other two. That is one of the big advantages of tuning by adjusting statistics over tuning with hints, which by extension, includes stored outlines and profiles.

Statistics changes which are beneficial for one SQL often are beneficial for other, related, SQL as well while hints always affect **only** the SQL they are placed in.

Of course, this can just as easily become a disadvantage if the statistics change(s) have a negative effect on other SQL while the effect of hints, outlines, and profiles is naturally isolated and limited to the tuned SQL.

```
 COST   CARD  operation                                         ELAPSED     ROWS    CR_GETS
-----  -----  ------------------------------------------------  ---------  -------  --------
  160         SELECT STATEMENT
  160      5    SORT ORDER BY                                       0.010        0         2
                  FILTER                                                                    2
  159      5        HASH JOIN                                                               2
  154      5          TABLE ACCESS BY INDEX ROWID PSPRCSQUE                                 2
  156      5            NESTED LOOPS                                               2         2
    3      1              TABLE ACCESS BY INDEX ROWID PS_SERVERCATEGORY 0.000      1         2
    2      1                INDEX RANGE SCAN PS_SERVERCATEGORY                               1
                          INLIST ITERATOR                                          0         0
  142    103                INDEX RANGE SCAN PSAPSPRCSQUE                                    0
    3      8          TABLE ACCESS BY INDEX ROWID PS_SERVERCLASS                             0
    2      1            INDEX RANGE SCAN PS_SERVERCLASS                                      0
                FILTER                                                                       0
    3      1      TABLE ACCESS BY INDEX ROWID PSPRCSQUE                                      0
    2      5        INDEX RANGE SCAN PSDPSPRCSQUE                                            0
```

Example 4

```
 COST  CARD  operation                                               ROWS ELAPSED    CR_GETS
  144       SELECT STATEMENT
  144     1 VIEW                                                       87  21.490 81,889,486
             FILTER                                                    87  21.490 81,889,486
  137     1   SORT GROUP BY                                        12,565  21.480 81,889,486
               FILTER                                              24,437 606.170 81,889,486
  135     1     NESTED LOOPS                                    3,244,217 600.030 81,851,299
  134     1      NESTED LOOPS                                   3,244,217 565.460 75,362,863
  132     1       NESTED LOOPS                                 13,519,270 376.740 53,001,492
  131     1        NESTED LOOPS                                13,519,270 259.290 39,482,220
  122     3         HASH JOIN                                  12,985,742  23.110      1,239
    3     2          TABLE ACCESS FULL PS_RT_RATE_TBL                 975   0.020         22
  118     9          TABLE ACCESS FULL PS_CUST_CREDIT              34,676   0.390      1,217
    3     1         TABLE ACCESS BY INDEX ROWID PS_CUSTOMER    13,519,270 194.890 39,480,981
    2     1          INDEX RANGE SCAN PSBCUSTOMER              13,831,838 106.360 26,075,609
    1     1        TABLE ACCESS BY INDEX ROWID PS_RT_INDEX_TBL 13,519,270  84.440 13,519,272
          1         INDEX UNIQUE SCAN PS_RT_INDEX_TBL          13,519,270  26.830          2
    2     1       TABLE ACCESS BY INDEX ROWID PS_CUST_DATA      3,244,217 152.400 22,361,371
    1     1        INDEX RANGE SCAN PSACUST_DATA                9,206,235  98.990 13,627,522
    1     1      TABLE ACCESS BY INDEX ROWID PS_CUSTOMER        3,244,217  25.700  6,488,436
          1       INDEX UNIQUE SCAN PS_CUSTOMER                 3,244,217  13.270  3,244,219
          1     SORT AGGREGATE                                     12,631   1.290     38,160
    4     1      TABLE ACCESS BY INDEX ROWID PS_CUST_CREDIT        12,767   1.190     38,160
    3     1       INDEX RANGE SCAN PS_CUST_CREDIT                  12,771   0.340     25,378
          1    SORT AGGREGATE                                           4   0.040         27
    3     2     NESTED LOOPS                                          749   0.040         27
    2     1      TABLE ACCESS FULL PS_RT_INDEX_TBL                      4   0.000         12
    1     2      INDEX RANGE SCAN PS_RT_RATE_TBL                      749   0.040         15
```

Tuning of Example 4

❶ Adjust the density of the EFFDT column of tables PS_RT_RATE_TBL and PS_CUST_CREDIT:

```
@SET_COL_DENSITY PS_CUST_CREDIT EFFDT 1
@SET_COL_DENSITY PS_RT_RATE_TBL EFFDT 1
```

the rational is the extraordinarily high discrepancy of the hash join cardinality estimate (12,985,742 : 3) after the – comparably – modest divergence in the estimates for the constituent tables. This quite obviously is a case of join uniformity violation. Note that, not so coincidentally, both tables involved in the hash join with the catastrophic cardinality assessment have an associated effective-date subquery. The effective-date subqueries instill a false impression of uniqueness, or close-to unique, on the CBO. As with the demo query, forcing the EFFDT density to 1 neutralizes this false appearance of uniqueness.

```
 COST     CARD operation                                             ROWS ELAPSED    CR_GETS
53825          SELECT STATEMENT
53825    6,488  VIEW                                                   87  17.720     49,408
                 FILTER                                                87  17.720     49,408
 8409    6,488    SORT GROUP BY                                    12,565  17.710     49,408
                   FILTER                                          24,437  17.430     49,408
  756  129,744      HASH JOIN                                   3,244,217  13.610      6,807
    2        1       TABLE ACCESS FULL PS_RT_INDEX_TBL                  1   0.000          3
  748  129,744        HASH JOIN                                 3,244,217   7.570      6,804
    3      975         TABLE ACCESS FULL PS_RT_RATE_TBL               975   0.030         22
  742    3,726          HASH JOIN                                  24,578   2.800      6,782
  570    2,651           HASH JOIN                                 36,097   1.980      5,212
  492    2,651            HASH JOIN                                36,097   1.210      4,724
  338    5,267             TABLE ACCESS FULL PS_CUSTOMER           40,715   0.170      3,507
  118   39,207             TABLE ACCESS FULL PS_CUST_CREDIT        34,676   0.210      1,217
   48   42,133            INDEX FAST FULL SCAN PS0CUSTOMER         42,133   0.070        488
  152   29,605           TABLE ACCESS FULL PS_CUST_DATA            29,609   0.040      1,570
                 1        SORT AGGREGATE                           14,091   0.890     42,574
    4        1         TABLE ACCESS BY INDEX ROWID PS_CUST_CREDIT  14,229   0.860     42,574
    3        1          INDEX RANGE SCAN PS_CUST_CREDIT            14,233   0.220     28,326
```

```
           1         SORT AGGREGATE                                4    0.020        27
   3       2          NESTED LOOPS                               749    0.020        27
   2       1           TABLE ACCESS FULL PS_RT_INDEX_TBL           4    0.000        12
   1       2            INDEX RANGE SCAN PS_RT_RATE_TBL          749    0.020        15
```

Comparing TCF to Hints and Profiles

While preparing the plan listings for examples 1-3 I noticed something which I wanted to explore further[8]. I had become accustomed to the apparent fact – mostly from postings in newsgroups – that a tuned plan has a higher cost than the original plan[9]. Just as I was to include a remark to that effect in the presentation I noticed that that was not the case here – the tuned plans all had a lower cost[10]. I thus got curious about what the cost would be with other tuning methods, namely hints and profiles[11].

Original Plan and Cost

```
   COST       CARD  operation                                ELAPSED     ROWS    CR_GETS
   220              SELECT STATEMENT
   220     12,516     HASH JOIN                                0.060        0      2,362
     3         15       TABLE ACCESS FULL PS_PRCSRECUR         0.000       15          3
   214     34,057       TABLE ACCESS FULL PSPRCSQUE            0.060        0      2,359
```

After new Index and frequency Histogram

```
   COST       CARD  operation                                ELAPSED     ROWS    CR_GETS
   209              SELECT STATEMENT
   209        236     HASH JOIN                                0.040        0      2,362
     3         15       TABLE ACCESS FULL PS_PRCSRECUR         0.000       15          3
   206        641       TABLE ACCESS FULL PSPRCSQUE            0.040        0      2,359
```

After setting PSPRCSQUE.PRCSJOBSEQ to 250

```
   COST       CARD  operation                                ELAPSED     ROWS    CR_GETS
    40              SELECT STATEMENT
    40          3     HASH JOIN                                0.010        0        112
    37          9       TABLE ACCESS BY INDEX ROWID PSPRCSQUE                        112
     3        109         INDEX RANGE SCAN UC_PSPRCSQUE_IX1                 112         49
     3         15       TABLE ACCESS FULL PS_PRCSRECUR         0.000        0          0
```

After setting PSPRCSQUE.PRCSJOBSEQ to 1000

```
   COST       CARD  OPERATION                                ELAPSED     ROWS    CR_GETS
    14              SELECT STATEMENT
    14          1    NESTED LOOPS                              0.010        0        112
    12          2      TABLE ACCESS BY INDEX ROWID PSPRCSQUE                         112
     3         27        INDEX RANGE SCAN UC_PSPRCSQUE_IX1                  112         49
     2          1      TABLE ACCESS BY INDEX ROWID PS_PRCSRECUR 0.000      0         0
     1          1        INDEX UNIQUE SCAN PS_PRCSRECUR                                  0
```

Note how the cost goes down with progressive tuning by adjusting the statistics.

[8] Isn't that how it often goes: trying to find the answer to one question only raises more questions to investigate.

[9] Of course it could be that one just does not hear of all the other, normal occurrences, only of the ostensible anomalies.

[10] That was before I added example 4

[11] You may notice an inconsistency here. All the examples are from Oracle 9.2 system – which has no profiles!? The answer is that I exported the tables involved to a 10g system in order to test DBMS_SQLTUNE and prodiles.

WITH INDEX HINT

/*+ INDEX(R, UC_PSPRCSQUE_IX1) INDEX(S, PS_PRCSRECUR) USE_NL(S,R) */

```
 COST   CARD  operation                                   ELAPSED    ROWS    CR_GETS
  672         SELECT STATEMENT
  672     54    NESTED LOOPS                                0.010       0        111
  529    142     TABLE ACCESS BY INDEX ROWID PSPRCSQUE      0.010       0        111
    9  1,703      INDEX RANGE SCAN UC_PSPRCSQUE_IX1         0.010     112         48
    2      1     TABLE ACCESS BY INDEX ROWID PS_PRCSRECUR   0.000       0          0
    1      1      INDEX UNIQUE SCAN PS_PRCSRECUR            0.000       0          0
```

Despite being the same plan, the one resulting from hints has a much higher cost than both the untuned, slower plan and especially the TCF tuned plan. That should not come as a surprise. If the cost would be lower the CBO would have chosen the hinted plan in the first place. The high cost is entirely fuelled by the incorrect cardinality estimates.

Clearly the correlation between cost and SQL performance is broken when tuning with hints – the cost went up, but the performance improved.

FINDINGS SECTION (2 FINDINGS)

1- SQL Profile Finding (see explain plans section below)
A potentially better execution plan was found for this statement.
 Recommendation (estimated benefit: 89.1%)
 Consider accepting the recommended SQL profile.

```
2- Using SQL Profile
------------------------------------------------------------------------------------------
| Id  | Operation                     | Name          | Rows  | Bytes | Cost (%CPU)| Time     |
------------------------------------------------------------------------------------------
|   0 | SELECT STATEMENT              |               |     1 |    92 |    47   (0)| 00:00:01 |
|   1 |  NESTED LOOPS                 |               |     1 |    92 |    47   (0)| 00:00:01 |
|   2 |   TABLE ACCESS BY INDEX ROWID| PSPRCSQUE     |     1 |    73 |    46   (0)| 00:00:01 |
|   3 |    INDEX SKIP SCAN            | PSAPSPRCSQUE  |     1 |       |    45   (0)| 00:00:01 |
|   4 |   TABLE ACCESS BY INDEX ROWID| PS_PRCSRECUR  |     1 |    19 |     1   (0)| 00:00:01 |
|   5 |    INDEX UNIQUE SCAN          | PS_PRCSRECUR  |     1 |       |     0   (0)| 00:00:01 |
------------------------------------------------------------------------------------------
```

```
  ATTR  ATTR_VALUE
     1  OPT_ESTIMATE(@"SEL$1", TABLE, "R"@"SEL$1", SCALE_ROWS=0.00664262176)
     2  OPT_ESTIMATE(@"SEL$1", INDEX_FILTER, "R"@"SEL$1", PSAPSPRCSQUE,
        SCALE_ROWS=0.0001556864475)
     3  OPT_ESTIMATE(@"SEL$1", INDEX_SKIP_SCAN, "R"@"SEL$1", PSBPSPRCSQUE,
        SCALE_ROWS=2.784763486)
     4  OPT_ESTIMATE(@"SEL$1", INDEX_FILTER, "R"@"SEL$1", UC_PSPRCSQUE_IX1,
        SCALE_ROWS=6.021536625)
     5  OPT_ESTIMATE(@"SEL$1", INDEX_FILTER, "R"@"SEL$1", PSEPSPRCSQUE,
        SCALE_ROWS=6.919397666e-005)
     6  OPT_ESTIMATE(@"SEL$1", INDEX_SKIP_SCAN, "R"@"SEL$1", UC_PSPRCSQUE_IX1,
        SCALE_ROWS=4.516152469)
     7  OPT_ESTIMATE(@"SEL$1", INDEX_SKIP_SCAN, "R"@"SEL$1", PSEPSPRCSQUE,
        SCALE_ROWS=5.18954825e-005)
```

The DBMS_SQLTUNE exercise came up with a very similar plan to the one derived at by adding the histogram. This plan uses an index skip scan on an existing index instead of the range scan on the custom index which, as can be seen in the profile hints, had been created before running DBMS_SQLTUNE. Note that the profile contains cardinality scale factors for every base access path of the table. Therefore, even with the profile in place, at parse time the CBO can still choose between different access paths depending on other factors, e.g. predicate values.
Interestingly enough, the recommendations did not include one for a histogram.

Note that the profile essentially does the same what the TCF method attempts to achieve – correct the cardinality estimates. Profiles do that by applying a scale factor to individual row sources rather than adjusting the base statistics. The result is obviously a more targeted and precise adjustment and one that has fewer possible side-effects.

The tale behind the tuning exercise (of Examples 1-3) or What I found when I visited a user

Told in the presentation

References

1. Holdsworth, A., et al. *A Practical Approach to Optimizer Statistics in 10g*. in *Oracle Open World*. September 17-22, 2005. San Francisco.
2. Breitling, W. *Fallacies of the Cost Based Optimizer*. in *Hotsos Symposium on Oracle Performance*. 2003. Dallas, Texas.
3. Lewis, J., *Cost-based Oracle: Fundamentals*. 2006: Apress. ISBN 1590596366.

Appendix

Demo SQL

```
SELECT A.COMPANY, A.PAYGROUP, E.OFF_CYCLE, E.SEPCHK_FLAG, E.TAX_METHOD
  , E.TAX_PERIODS, C.RETROPAY_ERNCD, sum(C.AMOUNT_DIFF)
from PS_PAY_CALENDAR A
   , WB_JOB B
   , WB_RETROPAY_EARNS C
   , PS_RETROPAY_RQST D
   , PS_RETROPAYPGM_TBL E
where A.RUN_ID = 'PD2'
  and A.PAY_CONFIRM_RUN = 'N'
  and B.COMPANY = A.COMPANY
  and B.PAYGROUP = A.PAYGROUP
  and B.EFFDT = (SELECT MAX(F.EFFDT) from WB_JOB F
      where F.EMPLID = B.EMPLID
        and F.EMPL_RCD# = B.EMPL_RCD#
        and F.EFFDT <= A.PAY_END_DT)
  and B.EFFSEQ = (SELECT MAX(G.EFFSEQ) from WB_JOB G
      where G.EMPLID = B.EMPLID
        and G.EMPL_RCD# = B.EMPL_RCD#
        and G.EFFDT = B.EFFDT)
  and C.EMPLID = B.EMPLID
  and C.EMPL_RCD# = B.EMPL_RCD#
  and C.RETROPAY_PRCS_FLAG = 'C'
  and C.RETROPAY_LOAD_SW = 'Y'
  and D.RETROPAY_SEQ_NO = C.RETROPAY_SEQ_NO
  and E.RETROPAY_PGM_ID = D.RETROPAY_PGM_ID
  and E.OFF_CYCLE = A.PAY_OFF_CYCLE_CAL
group by A.COMPANY, A.PAYGROUP, E.OFF_CYCLE, E.SEPCHK_FLAG, E.TAX_METHOD
  , E.TAX_PERIODS, C.RETROPAY_ERNCD
```

The WB_ prefixed tables are scaled back versions of the originals solely for performance reason. With the original sizes the query never finished (or, rather, was not given the time to finish). All the data in the demo tables are made up, but the tables, except for the scaled back WB_ tables, have sizes and other statistics to match the originals. The real PS_RETROPAY_EARNS table was more than 10 times the size of its demo sibling, ~ 1.5 million rows.

Example 1 SQL

```
SELECT R.PRCSINSTANCE ,R.ORIGPRCSINSTANCE ,R.RECURORIGPRCSINST,
  R.MAINJOBINSTANCE ,R.PRCSJOBSEQ ,R.PRCSJOBNAME ,R.PRCSNAME,
  R.PRCSTYPE, R.RECURNAME
FROM PSPRCSQUE R ,PS_PRCSRECUR S
```

```
WHERE ((R.RUNSTATUS IN (:"SYS_B_00", :"SYS_B_01") AND S.INITIATEWHEN =
:"SYS_B_02")
 OR (R.RUNSTATUS IN (:"SYS_B_03", :"SYS_B_04", :"SYS_B_05", :"SYS_B_06",
:"SYS_B_07",:"SYS_B_08", :"SYS_B_09") AND S.INITIATEWHEN = :"SYS_B_10"))
 AND R.INITIATEDNEXT = :"SYS_B_11"
 AND R.OPSYS = :1
 AND R.RUNLOCATION = :"SYS_B_12"
 AND R.RECURNAME <>  :"SYS_B_13"
 AND R.PRCSJOBSEQ = :"SYS_B_14"
 AND R.SERVERNAMERUN = :2
 AND R.RECURNAME = S.RECURNAME
```

Obviously, these example came from a database running with CURSOR_SHARING=force.

Example 2 SQL

```
SELECT Q.PRCSINSTANCE, Q.JOBINSTANCE, Q.MAINJOBINSTANCE, Q.SESSIONIDNUM
, Q.OPRID, Q.OUTDESTTYPE, Q.GENPRCSTYPE, Q.PRCSTYPE
, P.PRCSOUTPUTDIR FROM PSPRCSQUE Q
   , PSPRCSPARMS P
WHERE Q.RUNSTATUS = :1
  AND Q.SERVERNAMERUN = :2
  AND Q.RUNLOCATION = :"SYS_B_0"
  AND Q.PRCSINSTANCE = P.PRCSINSTANCE
```

Example 3 SQL

```
SELECT R.PRCSINSTANCE, R.ORIGPRCSINSTANCE, R.JOBINSTANCE, R.MAINJOBINSTANCE
, R.MAINJOBNAME, R.PRCSITEMLEVEL, R.PRCSJOBSEQ, R.PRCSJOBNAME, R.PRCSTYPE
, R.PRCSNAME, R.PRCSPRTY, TO_CHAR(R.RUNDTTM,:"SYS_B_00"), R.GENPRCSTYPE
, R.OUTDESTTYPE, R.RETRYCOUNT, R.RESTARTENABLED, R.SERVERNAMERQST, R.OPSYS
, R.SCHEDULENAME, R.PRCSCATEGORY, R.P_PRCSINSTANCE, C.PRCSPRIORITY
, S.PRCSPRIORITY, R.PRCSWINPOP, R.MCFREN_URL_ID
FROM PSPRCSQUE R, PS_SERVERCLASS S, PS_SERVERCATEGORY C
WHERE R.RUNDTTM <= SYSDATE
  AND R.OPSYS = :1 AND R.RUNSTATUS = :2
  AND (R.SERVERNAMERQST = :3 OR R.SERVERNAMERQST = :"SYS_B_01")
  AND S.SERVERNAME = :4 AND R.PRCSTYPE = S.PRCSTYPE
  AND R.PRCSCATEGORY = C.PRCSCATEGORY AND S.SERVERNAME = C.SERVERNAME
  AND ((R.PRCSJOBSEQ = :"SYS_B_02" AND R.PRCSTYPE <> :"SYS_B_03")
   OR (R.PRCSJOBSEQ > :"SYS_B_04" AND R.MAINJOBINSTANCE IN (
     SELECT A.MAINJOBINSTANCE  FROM PSPRCSQUE A WHERE A.MAINJOBINSTANCE >
:"SYS_B_05"
       AND A.PRCSTYPE=:"SYS_B_06" AND A.RUNSTATUS=:"SYS_B_07"
       AND A.PRCSJOBSEQ = :"SYS_B_08"
       AND (A.SERVERNAMERUN = :"SYS_B_09" OR A.SERVERNAMERUN = :5))))
  AND C.MAXCONCURRENT > :"SYS_B_10"
ORDER BY C.PRCSPRIORITY DESC, R.PRCSPRTY DESC, S.PRCSPRIORITY DESC, R.RUNDTTM ASC
```

Example 4 SQL

```
SELECT A.SETID, A.CUST_ID, A.NAME1, A.BAL_AMT, A.CR_LIMIT, (A.CR_LIMIT_REV_DT)
, A.CUSTCR_PCT_OVR, A.CR_LIMIT_RANGE, A.CR_LIMIT_CORP_DT, A.XX_FOLLOWUP_DATE
, A.CURRENCY_CD
, (A.BAL_AMT - (A.CR_LIMIT + ((A.CR_LIMIT * A.CUSTCR_PCT_OVR) / 100) ) )
, 'CRLMT'
FROM PS_XX_CR_LMT_VW A
WHERE A.BAL_AMT > (A.CR_LIMIT + ((A.CR_LIMIT * A.CUSTCR_PCT_OVR) / 100) )
  AND (A.XX_FOLLOWUP_DATE <= sysdate OR A.XX_FOLLOWUP_DATE IS NULL)
  AND A.CR_LIMIT > 0
ORDER BY XX_FOLLOWUP_DATE
```

With the view defined as

```
CREATE OR REPLACE VIEW PS_XX_CR_LMT_VW
 (SETID, CUST_ID, NAME1, BAL_AMT, CR_LIMIT, CR_LIMIT_REV_DT, CUSTCR_PCT_OVR
  , CR_LIMIT_RANGE, CR_LIMIT_CORP_DT, TT_FOLLOWUP_DATE, CURRENCY_CD )
AS
SELECT C.REMIT_FROM_SETID, C.REMIT_FROM_CUST_ID, N.NAME1
, SUM(D.BAL_AMT * R.CUR_EXCHNG_RT), O.CR_LIMIT_CORP, O.CR_LIMIT_REV_DT
, O.CORPCR_PCT_OVR, O.CR_LIM_CORP_RANGE, O.CR_LIMIT_CORP_DT, O.TT_FOLLOWUP_DATE
, O.CURRENCY_CD
FROM PS_CUSTOMER C
, PS_CUST_DATA D
, PS_CUST_CREDIT O
, PS_CUSTOMER N
, PS_CUR_RT_TBL R
WHERE C.CUST_STATUS = 'A'
  AND C.BILL_TO_FLG = 'Y'
  AND C.CUST_LEVEL <> 'P'
  AND C.SETID = 'TTS'
  AND C.CUST_ID = D.CUST_ID
  AND O.SETID = C.REMIT_FROM_SETID
  AND O.CUST_ID= C.REMIT_FROM_CUST_ID
  AND O.EFFDT = (
    SELECT MAX(EFFDT) FROM PS_CUST_CREDIT OO
    WHERE OO.SETID = O.SETID
      AND OO.CUST_ID = O.CUST_ID
      AND OO.EFFDT <= TO_DATE(TO_CHAR(SYSDATE,'YYYY-MM-DD'),'YYYY-MM-DD')
      AND OO.EFF_STATUS = 'A' )
  AND N.SETID = C.REMIT_FROM_SETID
  AND N.CUST_ID = C.REMIT_FROM_CUST_ID
  AND R.FROM_CUR = D.CURRENCY_CD
  AND R.TO_CUR = O.CURRENCY_CD
  AND R.CUR_RT_TYPE = O.RT_TYPE
  AND R.EFFDT = (
    SELECT MAX(EFFDT) FROM PS_CUR_RT_TBL RR
    WHERE RR.FROM_CUR =R.FROM_CUR
      AND RR.TO_CUR = R.TO_CUR
      AND RR.CUR_RT_TYPE =R.CUR_RT_TYPE
      AND RR.EFFDT <= TO_DATE(TO_CHAR(SYSDATE,'YYYY-MM-DD'),'YYYY-MM-DD')
      AND RR.EFF_STATUS = 'A' )
GROUP BY C.REMIT_FROM_SETID, C.REMIT_FROM_CUST_ID, N.NAME1
, C.ROLEUSER, O.CR_LIMIT_CORP, O.CR_LIMIT_REV_DT, O.CORPCR_PCT_OVR
, O.CR_LIM_CORP_RANGE, O.CR_LIMIT_CORP_DT, O.TT_FOLLOWUP_DATE, O.CURRENCY_CD
```

TABLE AND INDEX STATISTICS FOR THE PSPRCSQUE TABLE (EXAMPLES 1-3)

The psprcsque table was fingered in all three examples as the one with the cardinality estimate problem:

```
TABLE_NAME                 rows          blks   empty  avg_row
PSPRCSQUE                38,539         2,326       0      204
```

```
table       index          column                     NDV         DENS  #LB lvl #LB/K #LB/K    CLUF
PSPRCSQUE   PSAPSPRCSQUE                               43                257   2     5   116   4,998
                           SERVERNAMERQST               3   3.3333E-01    0         0     0       0
                           SERVERNAMERUN                3   3.3333E-01    0         0     0       0
                           OPSYS                        2   5.0000E-01    0         0     0       0
                           RUNSTATUS                   11   9.0909E-02    0         0     0       0
table       index          column                     NDV         DENS  #LB lvl #LB/K #LB/K    CLUF
PSPRCSQUE   PSBPSPRCSQUE                           38,539                242   2     1     1   3,735
                           SERVERNAMERUN                3   3.3333E-01    0         0     0       0
                           PRCSINSTANCE            38,539   2.5948E-05    0         0     0       0
```

```
table         index           column                   NDV       DENS   #LB lvl #LB/K #LB/K    CLUF
PSPRCSQUE     PSCPSPRCSQUE                           38,539               315   1     1     1   2,658
                              PRCSINSTANCE           38,539  2.5948E-05     0         0     0       0
                              SESSIONIDNUM            9,015  1.1093E-04     0         0     0       0
                              OPRID                     139  7.1942E-03     0         0     0       0

              PSDPSPRCSQUE                            7,249               174   1     1     1   4,395
                              MAINJOBINSTANCE         7,249  1.3795E-04     0         0     0       0

              PSEPSPRCSQUE                           10,735               221   2     1     1   3,121
                              RECURORIGPRCSINST      10,731  9.3188E-05     0         0     0       0
                              RECURNAME                   4  2.5000E-01     0         0     0       0
                              INITIATEDNEXT               2  5.0000E-01     0         0     0       0

              PS_PSPRCSQUE    U                      38,539               166   1     1     1   2,658
                              PRCSINSTANCE           38,539  2.5948E-05     0         0     0       0
```

Table and Index Statistics for Table in Example 4

```
TABLE_NAME           rows        blks    empty   avg_row
PS_RT_RATE_TBL        975          19        0        48

PS_CUST_CREDIT     39,593       1,214        0       109
```

```
table           index           column                  NDV         DENS   #LB  lvl #LB/K #LB/K    CLUF
PS_RT_RATE_TBL  PSART_RATE_TBL                          975   8.1485E+01    13    1     1     1     196
                                EFFDT                   466   2.1459E-03
                                FROM_CUR                  2   5.0000E-01
                                TO_CUR                    2   5.0000E-01
                                TERM                      1   1.0000E+00
                                RT_TYPE                   7   1.4286E-01
                                RT_RATE_INDEX             1   1.0000E+00

                PSBRT_RATE_TBL                           16   9.0272E+01     7    1     1     7     112
                                SYNCID                   16   6.2500E-02

                PS_RT_RATE_TBL  U                       975   7.7197E+01    14    1     1     1     237
                                RT_RATE_INDEX             1   1.0000E+00
                                TERM                      1   1.0000E+00
                                FROM_CUR                  2   5.0000E-01
                                TO_CUR                    2   5.0000E-01
                                RT_TYPE                   7   1.4286E-01
                                EFFDT                   466   2.1459E-03

PS_CUST_CREDIT  PS_CUST_CREDIT  U                    39,593   9.6475E+01   333    2     1     1   2,567
                                SETID                     1   1.0000E+00
                                CUST_ID              39,309   2.5439E-05
                                EFFDT                 4,430   2.2573E-04
```

SQL Script to Show Execution Plan with Row Source Statistics (if Available)

```
-------------------------------------------------------------------------------
--
-- Script:  v$xplain.sql
-- Purpose: format the plan and execution statistics from the dynamic
--          performance views v$sql_plan and v$sql_plan_statistics
--
-- Copyright:     (c)1996-2006 Centrex Consulting Corporation
-- Author:  Wolfgang Breitling
--
-- Usage    One parameter: sql_hash_value
--
-------------------------------------------------------------------------------

set define '~'
define hv=~1

set verify off echo off feed off
set linesize 300 pagesize 3000
```

```
col hv head 'hv' noprint
col "cn" for 90 print
col "card" for 999,999,990
col "ROWS" for 999,999,990
col "ELAPSED" for 99,990.999
col "CPU" for 99,990.999
col CR_GETS for 99,999,990
col CU_GETS for 99,999,990
col READS for 9,999,990
col WRITES for 99,990

break on hv skip 0 on "cn" skip 0

SELECT P.HASH_VALUE hv
 , P.CHILD_NUMBER "cn"
 , to_char(p.id,'990')||decode(access_predicates,null,null,'A')
   ||decode(filter_predicates,null,null,'F') id
 , P.COST "cost"
 , P.CARDINALITY "card"
 , LPAD(' ',depth)||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$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "ROWS"
 , ( SELECT ROUND(S.LAST_ELAPSED_TIME/1000000,2)
     FROM V$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "ELAPSED"
 , (SELECT S.LAST_CR_BUFFER_GETS FROM V$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "CR_GETS"
 , (SELECT S.LAST_CU_BUFFER_GETS FROM V$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "CU_GETS"
 , (SELECT S.LAST_DISK_READS FROM V$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "READS"
 , (SELECT S.LAST_DISK_WRITES FROM V$SQL_PLAN_STATISTICS S
     WHERE S.ADDRESS=P.ADDRESS and s.hash_value=p.hash_value
       and s.child_number=p.child_number AND S.OPERATION_ID=P.ID)  "WRITES"
FROM V$SQL_PLAN P
where p.hash_value = ~hv
order by P.CHILD_NUMBER, p.id
/
```

[i] Wolfgang Breitling had been a systems programmer for IMS and later DB2 databases on IBM mainframes for several years before, in 1993, he joined a project to implement Peoplesoft on Oracle. In 1996 he became an independent consultant specializing in administering and tuning Peoplesoft on Oracle. The particular challenges in tuning Peoplesoft, with often no access to the SQL, motivated him to explore Oracle's cost-based optimizer in an effort to better understand how it works and use that knowledge in tuning. He has shared the findings from this research in papers and presentations at IOUG, UKOUG, local Oracle user groups, and other conferences and newsgroups dedicated to Oracle performance topics.