Using DBMS_STATS in Access Path Optimization

Wolfgang Breitling, Centrex Consulting Corporation

Following a brief introduction to the procedures of the DBMS_STATS package in section 1, section 2 looks at the differences of GATHER_*XXX*_STATS between Oracle 8 and 9. It will further show ways of transferring statistics between databases with caveats on what to watch for when using production statistics in a test database. Finally, in section 3 we will show two scenarios where the deliberate use of SET_COLUMN_STATS to change column statistics helps the optimizer choose a better, faster and more scalable access plan. The goal is to promote the notion that the statistics are a means to give the CBO information about the data in the database in order to enable it to make the best possible access path decisions and that the statistics are not a "sacred cow" but that it is OK to alter them in order to give the optimizer better information.

The findings presented are based on experience and tests with Oracle 8*i* (8.1.7) on Windows 2000, Linux Redhat 7.2, HP-UX 11.0, and Compaq Tru64 5.1. Comments on changes in Oracle 9*i* are based on Oracle 9.2.0 on Windows 2000 and Linux Redhat 7.2.

The DBMS_STATS Package

With Oracle 8 Oracle introduced the DBMS_STATS package for statistics collection and maintenance: "With DBMS_STATS you can view and modify optimizer statistics gathered for database objects. The statistics can reside in the dictionary or in a table created in the user's schema for this purpose." (A76936-01, 1999, A96612-01, 2002)

The DBMS_STATS subprograms perform the following general functions:

- Gather optimizer statistics
- Transfer statistics
- Set or get statistics

When a DBMS_STATS subprogram modifies or deletes the statistics for an object, all the dependent cursors are invalidated by default and corresponding statements are subject to recompilation next time so that the new statistics have immediate effects. This behavior can be altered with the NO_INVALIDATE argument.[1]
Oracle 9 expands the package to include new procedures and options.

Gather Optimizer Statistics

There are four procedures to gather statistics on objects in the database:

- GATHER_DATABASE_STATS, GATHER_SCHEMA_STATS
  gathers statistics for all objects in the database or in a schema. There are many options for special processing, e.g. finding all objects without, or with stale statistics.
- GATHER_TABLE_STATS
  gathers table and column statistics. It can gather statistics for individual partitions; the default is to gather global table statistics and individual statistics for all partitions. GATHER_TABLE_STATS is also the procedure to gather histograms. Unlike ANALYZE, GATHER_TABLE_STATS can do much of the statistics gathering in parallel, but there are limitations. Consult the manual. After gathering table and column stats, GATHER_TABLE_STATS can also gather statistics on all indexes of the table, but unlike ANALYZE does not do so by default (cascade => { true | <u>false</u> })

[1] New in Oracle 9

- GATHER_INDEX_STATS
  gathers index statistics. It does not execute in parallel. As with GATHER_TABLE_STATS, unless a partition name is specified, global statistics and statistics for all individual partitions are gathered for a partitioned index.

All gathering procedures have the option to save the current statistics in a user statistics table before overwriting them. There are corresponding DELETE_*xxx*_STATS procedures plus an additional DELETE_COLUMN_STATS procedure. With DBMS_STATS it is possible to delete the statistics for individual columns.

Additions to DBMS_STATS in Oracle 9*i*:

- GATHER_SYSTEM_STATS
  gathers system statistics. The current values can be retrieved with GET_SYSTEM_STATS or viewed by querying sys.aux_stats$

sreadtim	average time to read single block (random read), in milliseconds
mreadtim	average time to read an mbrc block at once (sequential read), in milliseconds
cpuspeed	average number of CPU cycles per second, in millions
mbrc	average multiblock read count for sequential read, in blocks
maxthr	maximum I/O system throughput, in bytes/sec
slavethr	average slave I/O throughput, in bytes/sec

Unlike the database object statistics gathering, gathering system statistics does not invalidate any SQL in the SGA and cause it to be reparsed. However, Oracle9*i* now has a parameter in all the object statistics gathering procedures to not invalidate SQL either (no_invalidate => { true | <u>false</u> })

- ALTER_TAB_SCHEMA_MONITORING, ALTER_DATABASE_TAB_MONITORING
  Shortcut ways to enable monitoring for all tables in a schema or in the entire database, optionally including catalog tables.
- FLUSH_DATABASE_MONITORING_INFO
  To request a flush of the monitoring data from the SGA to the sys.mon_mods$ catalog table. The monitoring data is also flushed at regular intervals by SMON or when the database is closed.

For details on the DBMS_STATS package and its sub procedures please refer to the appropriate documentation. (A76936-01, 1999, A96612-01, 2002)

Differences between ANALYZE and DBMS_STATS

In Oracle 8*i* there is actually not much difference between DBMS_STATS and ANALYZE, provided equivalent requests are used, since DBMS_STATS is using ANALYZE under the covers for index statistics and histograms.

Recall that
gather_table_stats('owner', 'table_name', NULL, estimate_pct)
is <u>not</u> equivalent to
analyze table owner.table_name {compute | estimate} statistics

The former does not gather statistics on indexes since CASCADE defaults to false.
Oracle 9*i* does not use ANALYZE anymore as we will see next.

DBMS_STATS Differences between Oracle 8 and Oracle 9

Following we look in more detail at the internal processing of GATHER_TABLE_STATS.

Tracing a GATHER_TABLE_STATS call with the 10046 event reveals the dependent SQL executed on behalf of the request. In this paper only 1st level dependent SQL that is not attributed to SYS are shown.

Using the Standard / Default Method_Opt Setting

```
DBMS_STATS.GATHER_TABLE_STATS (
  ownname => NULL
, tabname => 'T1'[2]
[, estimate_percent => null]
[, method_opt => 'FOR ALL COLUMNS SIZE 1']
[, degree => NULL]
[, granularity => 'DEFAULT']
, cascade => TRUE
);
```

Oracle 8*i*

```
select /*+[3] */ count(*)
, count("PK1"), count(distinct "PK1"), sum(vsize("PK1")), min("PK1"), max("PK1")
, count("PK2"), count(distinct "PK2"), sum(vsize("PK2")), min("PK2"), max("PK2")
, count("D1"), count(distinct "D1"), 8, min("D1"), max("D1")
, count("D2"), count(distinct "D2"), sum(vsize("D2")), min("D2"), max("D2")
, count("D3"), count(distinct "D3"), sum(vsize("D3")), min(substrb("D3",1,32)),
max(substrb("D3",1,32))
from "SCOTT"."T1" t
analyze index "SCOTT"."T1P" COMPUTE statistics
```

As we can see, the first SQL gathers the information for the table and column statistics. Average column and row size can be calculated from the sum of the column sizes and the row count. The number of nulls for each column can be deduced from the difference of row and column counts and the apparently missing table block count is taken from the table's segment header. The second SQL reveals that Oracle 8 resorts to the ANALYZE command in order to get the index statistics.

Oracle 9*i*

The same GATHER_TABLE_STATS request in Oracle 9*i* is processed as follows:

```
select  /*+ */  count(*)
, count("PK1"), count(distinct "PK1"), sum(vsize("PK1")),
       substrb(dump(min("PK1"),16,0,32),1,120), substrb(dump(max("PK1"),16,0,32),1,120)
, count("PK2"), count(distinct "PK2"), sum(vsize("PK2")),
       substrb(dump(min("PK2"),16,0,32),1,120), substrb(dump(max("PK2"),16,0,32),1,120)
, count("D1"), count(distinct "D1"),
       substrb(dump(min("D1"),16,0,32),1,120), substrb(dump(max("D1"),16,0,32),1,120)
, count("D2"), count(distinct "D2"), sum(vsize("D2")),
       substrb(dump(min("D2"),16,0,32),1,120), substrb(dump(max("D2"),16,0,32),1,120)
, count("D3"), count(distinct "D3"), sum(vsize("D3")),
       substrb(dump(min(substrb("D3",1,32)),16,0,32),1,120),
       substrb(dump(max(substrb("D3",1,32)),16,0,32),1,120)
from "SCOTT"."T1" t
select /*+ */ count(*) as nrw
, count(distinct sys_op_lbid(177913,'L',t.rowid)) as nlb
, count(distinct hextoraw(sys_op_descend("PK1")||sys_op_descend("PK2"))) as ndk
, sys_op_countchg(substrb(t.rowid,1,15),1) as clf
from "SCOTT"."T1"  t
where "PK1" is not null
   or "PK2" is not null
```

[2] See the appendix for the table definition.

[3] The list of, partly undocumented, hints has been omitted for brevity.

The first SQL which gets the table and column statistics is very much like the one in Oracle 8 except for the different way of determining the column min and max values. Obviously, the 2nd SQL replaces the "analyze index" command of Oracle 8. Note the undocumented internal functions.

Using Non-Default Method_Opt Settings

I have never had any trouble with DBMS_STATS – aside from an early problem with partitioned tables (just needed to gather statistics explicitly for each partition) in 8.1.6. But I see complaints about DBMS_STATS on metalink and in newsgroups by others. Maybe it is because I always use the default Method_Opt setting, except for a few select histograms. Let us look at the generated SQL for two requests posted in newsgroups:

```
DBMS_STATS.GATHER_TABLE_STATS (
  ownname => NULL
, tabname => 't1'
, estimate_percent => 10
, method_opt => 'FOR ALL COLUMNS'
, cascade => TRUE
);
```

Oracle 8*i*

```
analyze table "SCOTT"."T1" ESTIMATE statistics sample 10 percent
FOR TABLE
FOR ALL INDEXES
FOR ALL COLUMNS
```

I am not certain if the requester realizes and intended what he is asking for. It is the creation of histograms for all columns of the table with the default bucket size of 75. Note that Oracle 8 in that case retorts completely to ANALYZE, which is one more reason why I do not understand the claims of different results from GATHER_TABLE_STATS as opposed to ANALYZE.

Oracle 9*i*

Oracle 9*i* does not use ANALYZE anymore but watch what it generates from this request. Again, almost all hints have been omitted for brevity and the SQL have been numbered to allow for easier reference:

```
1.  create global temporary table sys.ora_temp_1_ds_269
      on commit preserve rows cache
      as
      select  /*+ */ "PK1","PK2","D1","D2","D3"
      from "SCOTT"."T1" sample (10) t  where 1 = 2
2.  insert /*+ append */ into sys.ora_temp_1_ds_269
      select  /*+ */ "PK1","PK2","D1","D2","D3"
      from "SCOTT"."T1" sample (10) t
3.  select  /*+ */  count(*), count("PK1"),sum(vsize("PK1")), count("PK2"), sum(vsize("PK2")), count("D1"),
      count("D2"), sum(vsize("D2")), count("D3"), sum(vsize("D3"))
      from sys.ora_temp_1_ds_269 t
4.  select min(minbkt), maxbkt, substrb(dump(min(val),16,0,32),1,120) minval,
     substrb(dump(max(val),16,0,32),1,120) maxval, sum(rep) sumrep, sum(repsq) sumrepsq, max(rep) maxrep,
     count(*) bktndv
     from ( select val, min(bkt) minbkt, max(bkt) maxbkt, count(val) rep, count(val)*count(val) repsq
            from (  select  /*+ */ "PK1" val, ntile(75) over (order by "PK1") bkt
                    from sys.ora_temp_1_ds_269  t
                    where "PK1" is not null)
            group by val)
     group by maxbkt
     order by maxbkt
5.  select substrb(dump(val,16,0,32),1,120) ep, cnt
     from ( select  /*+ */ "PK1" val, count(*) cnt
            from sys.ora_temp_1_ds_269  t
            where "PK1" is not null
            group by "PK1"
            order by 1)
```

```
6.  select min(minbkt), maxbkt, substrb(dump(min(val),16,0,32),1,120) minval,
     substrb(dump(max(val),16,0,32),1,120) maxval, sum(rep) sumrep, sum(repsq) sumrepsq, max(rep) maxrep,
     count(*) bktndv
     from ( select val, min(bkt) minbkt, max(bkt) maxbkt, count(val) rep, count(val)*count(val) repsq
            from (  select  /*+ */ "PK2" val, ntile(75) over (order by "PK2") bkt
                  from sys.ora_temp_1_ds_269  t
                  where "PK2" is not null)
            group by val)
     group by maxbkt
     order by maxbkt
7.  select substrb(dump(val,16,0,32),1,120) ep, cnt
     from ( select  /*+ */ "PK2" val, count(*) cnt
            from sys.ora_temp_1_ds_269  t
            where "PK2" is not null
            group by "PK2"
            order by 1)
8.  select min(minbkt), maxbkt, substrb(dump(min(val),16,0,32),1,120) minval,
     substrb(dump(max(val),16,0,32),1,120) maxval, sum(rep) sumrep, sum(repsq) sumrepsq, max(rep) maxrep,
     count(*) bktndv
     from ( select val, min(bkt) minbkt, max(bkt) maxbkt, count(val) rep, count(val)*count(val) repsq
            from (  select  /*+ */ "D1" val, ntile(75) over (order by "D1") bkt
                  from sys.ora_temp_1_ds_269  t
                  where "D1" is not null)
            group by val)
     group by maxbkt
     order by maxbkt
9.  select min(minbkt), maxbkt, substrb(dump(min(val),16,0,32),1,120) minval,
     substrb(dump(max(val),16,0,32),1,120) maxval, sum(rep) sumrep, sum(repsq) sumrepsq, max(rep) maxrep,
     count(*) bktndv
     from (select val, min(bkt) minbkt, max(bkt) maxbkt, count(val) rep, count(val)*count(val) repsq  from (
            select  /*+ */ "D2" val, ntile(75) over (order by "D2") bkt
                    from sys.ora_temp_1_ds_269  t
                    where "D2" is not null)
            group by val)
     group by maxbkt
     order by maxbkt
10. select min(minbkt), maxbkt, substrb(dump(min(val),16,0,32),1,120) minval,
     substrb(dump(max(val),16,0,32),1,120) maxval, sum(rep) sumrep, sum(repsq) sumrepsq, max(rep) maxrep,
     count(*) bktndv
     from ( select val, min(bkt) minbkt, max(bkt) maxbkt, count(val) rep, count(val)*count(val) repsq
            from (  select  /*+ */ substrb("D3",1,32) val
                    , ntile(75) over (order by substrb("D3",1,32)) bkt
                    from sys.ora_temp_1_ds_269  t
                    where substrb("D3",1,32) is not null)
            group by val)
     group by maxbkt
     order by maxbkt
11. select substrb(dump(val,16,0,32),1,120) ep, cnt
     from ( select  /*+ */ substrb("D3",1,32) val, count(*) cnt
            from sys.ora_temp_1_ds_269  t
            where substrb("D3",1,32) is not null
            group by substrb("D3",1,32)
            order by 1)
```

Interesting to note the use of a temporary table when using estimate_pct < 50.

Statements 4-11 perform the histogram collection. In the statement pairs 4-5, 6-7, and 10-11 Oracle first collects data for a height-balanced histogram using the `NTILE` analytic function (4, 6, 10). After recognizing that there are more buckets than distinct values, Oracle then re-scans the table for a frequency histogram (5,7,11). Columns D1 and D2 have more distinct values than the 75 buckets and the height balanced histogram is retained (statements 8 and 9).
If there are no pre-existing statistics on the column(s), Oracle 9i will first gather a height-balanced histogram using the `NTILE` analytic function and when it turns out that the requested buckets are sufficient (if `NDV` is between 1.4 and 1.6 times buckets) it will rescan for a frequency histogram If there are already statistics, it will use these to make the determination which kind of histogram to collect. Presumable it will still rescan for a frequency histogram in that case as well if the first choice of HB histogram proves not to be right anymore.

This table had only five columns. Now imagine a table with dozens of columns. Maybe that is the reason that some complain that GATHER_TABLE_STATS got slower in Oracle 9. It all depends on what you are asking it to do. It is actually not as bad as it looks initially. Note that Oracle does not repeatedly sample the real table, but collects a sample into a temporary table and from then on uses it for the histogram data gathering. Through this trick GATHER_TABLE_STATS achieves several things:

- All data collected is from the same sample. Any correlation between values of different columns is preserved. Something that might be problematic if each histogram was based on a separate random sample of the original table.
- Using a global temporary table, there is a chance that the table remains in the PGA during the entire process and does not need to be externalized (to the temp tablespace).

As an aside: even if intended, collecting histograms on all columns is a waste of time and resources to collect them and of space to store them. A histogram on a column can only be of value if the column is used as a predicate in a SQL[4]. Even using the option "for all indexed columns" is too wasteful, yet too limiting at the same time. Too wasteful because – even if we assume that indexed columns are used in predicates, which is by no means a given – not every predicate column requires, nor should have, a histogram, only those with a skewed data distribution. And too limiting because, contrary to popular belief, the purpose of a histogram is not to let the optimizer choose between an index access and a full table scan. The purpose is to give the CBO better information about the selectivity of the column, given the particular predicates of the SQL being parsed, when the distribution of the values of the column deviates significantly from the base assumption that every value is equally likely to occur – the uniform distribution assumption. The choice of index access vs. full scan is only one byproduct of the more accurate selectivity and cardinality estimate.
Histograms are like drugs – An overdose [of histograms] can kill [performance].

Automatisms

Increasingly Oracle attempts to aid the DBA with maintenance and tuning tasks or even become self-maintaining and self-tuning. Next we look at some of the features in DBMS_STATS that fall into that category.

The "List Stale" and "Gather Stale" Options

Requesting a list of tables with "stale" statistics – statistics which are not up-to-date – requires that
a) monitoring is enabled: (ALTER TABLE xxx MONITORING), and
b) the table is analyzed.
1.
The order of the two operations does not matter. Oracle then accumulates counts of rows inserted, updated, or deleted on the table in memory, flushing the counts to the catalog table MON_MODS$ at regular intervals or when the database is shut down. The procedures GATHER_SCHEMA_STATS and GATHER_DATABASE_STATS refer to that table when invoked with the options "LIST STALE", or "GATHER STALE" to determine if the existing statistics are stale and should be refreshed:

```
declare
  a dbms_stats.objecttab;
begin
  dbms_stats.gather_schema_stats(OWNNAME=>'scott',OPTIONS=>'LIST STALE',OBJLIST=>a);
end;
SELECT[5] /*+ ordered full(t) use_hash(t) use_nl(o) use_nl(u)  */
U.NAME OWN,O.NAME TAB, NULL  PART, NULL  SPART
FROM SYS.MON_MODS$ M,SYS.TAB$ T,SYS.OBJ$ O,SYS.USER$ U
WHERE M.OBJ# = T.OBJ#
```

[4] That is actually true of any statistic for a column. However, the standard statistics – NUM_DISTINCT, nulls, min, and max are collected for "free" with the table statistics. It would be more cumbersome to exclude or remove them than to keep them.

[5] The actual SQL contains unions with the equivalent selects for table partitions and subpartitions which have been omitted for brevity.

```
  AND BITAND(T.FLAGS,16) = 16
  AND ((BITAND(M.FLAGS,1) = 1 )
   OR ((M.INSERTS + M.UPDATES  + M.DELETES ) > (.1 * T.ROWCNT ) ))
  AND T.OBJ# = O.OBJ# AND O.OWNER# = U.USER#
ORDER BY 1,2,3,4
```

It is evident that Oracle considers statistics stale when the combined DML activity touched 10% of the rows of the table. I am not aware of a – documented or undocumented – parameter to change that percentage.

New in Oracle 9i

Method_Opt => 'FOR COLUMNS SIZE AUTO'

With this option Oracle determines the columns for which to collect histograms not only based on their data distribution – `Method_Opt => SKEWONLY` does that – but also SQL workload. To do that it uses table `SYS.COL_USAGE$` which is maintained by CBO during parsing:

```
dbms_stats.gather_table_stats(NULL, 't1', method_opt => 'FOR ALL COLUMNS SIZE AUTO');
select  … , cu.timestamp cu_time, cu.equality_preds cu_ep
, cu.equijoin_preds cu_ejp, cu.range_preds cu_rp, cu.like_preds cu_lp
from user$ u, obj$ o, col$ c, col_usage$ cu, hist_head$ h
where u.name = :b1 and o.name = :b2
  and c.obj# = cu.obj# (+) and c.intcol# = cu.intcol# (+) …
```

In the COL_USAGE$ table Oracle tracks which columns are used in predicates and in what kinds of predicates. That information may be interesting for DBAs in its own right, not just as input to `GATHER_TABLE_STATS`.

Estimate_Percent => DBMS_STATS.SAMPLE_AUTO_SIZE

With this option Oracle will dynamically determine the sample size necessary to collect accurate statistics starting with a very coarse 1/100000th % and increasing the sampling rate by a factor of 100 until it is satisfied. For tables with rather uniform data distribution it may be able to terminate sooner than for tables with very skewed distribution.

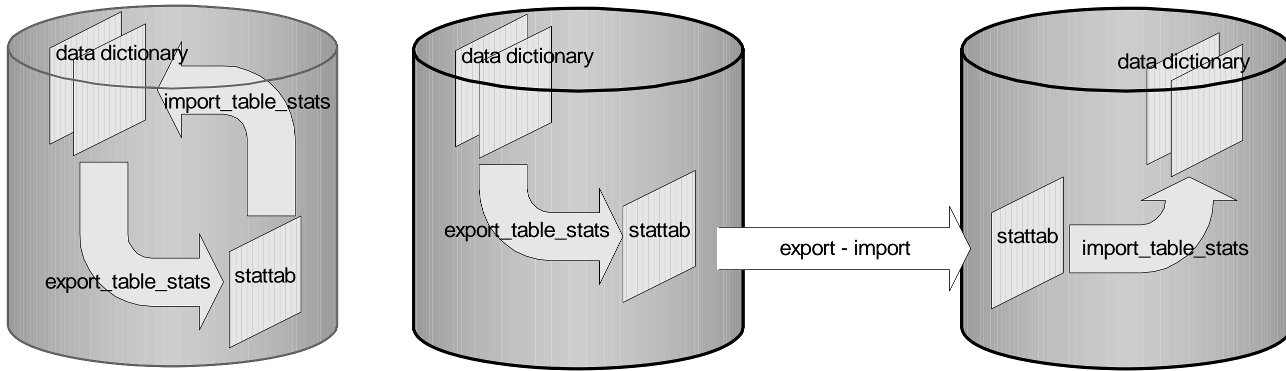
```
select /*+ … */ count(*) from "SCOTT"."BIG_TABLE_16K" sample block (.00001);
select /*+ … */ count(*) from "SCOTT"."BIG_TABLE_16K" sample block (.001);
select /*+ … */ count(*) from "SCOTT"."BIG_TABLE_16K" sample block (.1);
select /*+ … */ count(*) from "SCOTT"."BIG_TABLE_16K" sample block (10);
```

Options=>'GATHER AUTO'

Effectively combines all the automatic features – "gather empty", "gather stale", "size auto", and auto_sample_size" into one option where Oracle determines which tables to analyze with what sampling precision and for which columns to gather histograms.

Backup and Transfer of Statistics

Most of the DBMS_STATS procedures include the three parameters STATOWN, STATTAB, and STATID. They allow for statistics to be stored outside of the dictionary, where they do not affect the optimizer.
The STATTAB parameter specifies the name of a table in which to hold statistics. Unless the STATOWN parameter is also specified it is assumed that it resides in the same schema as the object for which statistics are collected. Different sets of statistics can be maintained within a single STATTAB by using the STATID parameter. For the SET and GET procedures, if STATTAB is not provided (that is, NULL), then the operation works directly on the dictionary statistics. If STATTAB is NOT NULL, then the SET or GET operation works on the specified user statistics table, and not the dictionary.



The DBMS_STATS package further offers the ability to copy statistics between the dictionary and the user statistics table: EXPORT_*_STATS copies the statistics from the dictionary to the user table and IMPORT_*_STATS does the reverse.
The user statistics table and the EXPORT_*_STATS and IMPORT_*_STATS procedures make it possible to backup statistics; and restore them if new statistics have undesirable effects on the optimizer's plan choices. Since the user statistics table is an ordinary table, it can be exported and imported into another database and then IMPORTed into that database's dictionary to make it "look" to the optimizer like the original database. SQL statement plans can thus be analyzed in a test database with less – or even no – data.
The biggest caveat when trying to use statistics from one database in another and expecting the CBO to make the same decisions are missing statistics.

- Index statistics: CBO substitutes fixed defaults – LVLS=1, #LB=100, LB/K=1, DB/K=1, CLUF=800
  so that is not a problem.
- Table statistics: CBO gets the ACTUAL number of blocks from the segment header, uses a default avg_row_len = 100 and derives the cardinality from #blocks * block_size / 100
  that can lead to significant differences if the actual table sizes (hwm) are different or the block size is different.
  Aside from that, blocksize has surprisingly little effect on the CBO's decisions
- Column statistics: CBO derives density from num_rows established in table statistics and NDV is round(1/density,0) (Joakim Treugut)

Missing index statistics clearly do not pose a problem since the substituted defaults are the same regardless of the actual index size. Table and column statistics, however, are derived from the actual size of the table and if that differs between the databases the optimizer will likely come to different conclusions. Even if all the statistics are present and transferred, other parameters may be different: block size, sort area size, hash area size, multiblock read counts, etc. leading to different execution plans. And then in 9 there could be dynamic sampling, system statistics and dynamic area sizing with `PGA_AGGREGATE_TARGET`.

Another way to backup or transfer statistics is via the export and import utilities. Export creates DBMS_STATS.SET_XXX_STATS statements in the dmp file which, when executed by import restore the statistics as they existed in the source database at the time of the export. The import utility will load the exported statistics
even if

- the table already exists (ignore=Y), or
- no data is imported (rows=N), unless
- analyze = N, or
- recalculate statistics = Y

Unfortunately the Oracle 8 export utility has several, IMHO severe, restrictions that preclude the export of statistics:

"In some cases, Export will place the precomputed statistics in the export file as well as the ANALYZE commands to regenerate the statistics. However, the precomputed optimizer statistics will not be used at export time if:

- A table has indexes with system-generated names (including LOB indexes)
- A table has columns with system-generated names
- There were row errors while exporting
- The client character set or NCHARSET does not match the server character set or NCHARSET
- You have specified a QUERY clause
- Only certain partitions or subpartitions are to be exported
- Tables have indexes based upon constraints that have been analyzed (check, unique, and primary key constraints)
- Tables have indexes with system-generated names that have been analyzed (IOTs, nested tables, type tables that have specialized constraint indexes)

" (A76955-01, 1999)

Fortunately, Oracle 9 lifts all except the second limitation – tables having columns with system generated names. In some of the cases which prevented the Oracle 8 export utility from using the precomputed statistics, the Oracle 9 export utility will issue a warning "EXP-00091: Exporting questionable statistics." (A96652-01, 2002)

The "STATTAB" Table

The user statistics table must be built with the CREATE_STAT_TABLE procedure. There is also a DROP_STAT_TABLE procedure which may be useful in a stored procedure, otherwise the table may just be dropped with a "drop table …" statement.
These are the columns of the stattab table:

Name	Type
STATID	VARCHAR2(30)
TYPE	CHAR(1)
VERSION	NUMBER
FLAGS	NUMBER
D1	DATE
CH1	VARCHAR2(1000)
C1	VARCHAR2(30)
C2	VARCHAR2(30)
C3	VARCHAR2(30)
C4	VARCHAR2(30)
C5	VARCHAR2(30)

Name	Type
N1	NUMBER
N2	NUMBER
N3	NUMBER
N4	NUMBER
N5	NUMBER
N6	NUMBER
N7	NUMBER
N8	NUMBER
N9	NUMBER
N10	NUMBER
N11	NUMBER
N12	NUMBER
R1	RAW(32)
R2	RAW(32)

In order to be able to use the stattab table to modify statistics, it is necessary to understand what the columns mean. The following mapping is based on observation. Oracle does not document it: "The columns and types that compose this table are not relevant as it should be accessed solely through the procedures in this package".
STATID is the only documented column and is the user settable identifier. CH1 is currently unused and VERSION is always 4, even for Oracle 9*i*. The meaning of the FLAGS bits is unknown, but bitand(flags,2) seems to correspond to GLOBAL_STATS and bitand(flags,1) to USER_STATS. The latter also seems to be the default if FLAGS is not set in a SET_xxx_STATS procedure call.

Except for the new system statistics, C5 is the owner and D1 is the LAST_ANALYZED date. TYPE identifies the type of object and the meaning of the other columns depends on the TYPE:

TYPE 'T'		TYPE 'I'		TYPE 'C'		TYPE 'C' histogram	
C5	OWNER	C5	OWNER	C5	OWNER	C5	OWNER
C1	TABLE_NAME	C1	INDEX_NAME	C1	TABLE_NAME	C1	TABLE_NAME
C2	PARTITION_NAME	C2	PARTITION_NAME	C2	PARTITION_NAME	C2	PARTITION_NAME
C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME	C3	SUBPARTITION_NAME
				C4	COLUMN_NAME	C4	COLUMN_NAME
N1	NUM_ROWS	N1	NUM_ROWS				
N2	BLOCKS	N2	LEAF_BLOCKS	N1	NUM_DISTINCT	N10	ENDPOINT_NUMBER
N3	AVG_ROW_LEN	N3	DISTINCT_KEYS	N2	DENSITY	N11	ENDPOINT_VALUE
N4	SAMPLE_SIZE	N4	LEAF_BLOCKS_PER_KEY	N4	SAMPLE_SIZE		
		N5	DATA_BLOCKS_PER_KEY	N5	NUM_NULLS		
		N6	CLUSTERING_FACTOR	N6	LO_VALUE		
		N7	BLEVEL	N7	HI_VALUE		
		N8	SAMPLE_SIZE	N8	AVG_COL_LEN		

If you want to actively work with the statistics in the STATTAB table it is best to create views that translate the terse, generic column names, which have different meaning depending on TYPE, into names corresponding to those of the dictionary views.

SET or GET Statistics

Another possibility is to actively manipulate the statistics to affect the optimizer's access plans using the SET_XXX_STATS procedures. We will see uses of that technique in section 3.

Dave Ensor's DBMS_STATS Paradox

Having discussed the procedures to gather statistics, the question of course is when and how often. Whenever someone asks for advice on how to deal with a poorly performing SQL, be it on a news forum or on metalink, one of the first questions invariably is "Are the statistics up-to-date". And that is a valid question when faced with a performance problem. Many DBAs have therefore implemented a scheduled statistics refresh process, usually during off hours, overnight or on a weekend, to keep the statistics up-to-date. We already looked at the "GATHER STALE" option which attempts to automate that by deciding which tables have changed enough to need to be re-analyzed. But do they? There is a reason for the sayings "never change a winning team" and "if it ain't broke, don't fix it"
In his presentation at the UKOUG conference Dec 9th 2003 Dave Ensor suggested to analyze once and then leave the [production]database alone, coining the term "DBMS_STATS Paradox":

> It is only safe to gather statistics when to do so will make no difference.

While a bit oversimplified, I fully agree with Dave's assertion. After the conference, Mogens Norgaard posted the question on the Oracle-L list "Should we stop analyzing?", referring to Dave's presentation. Here are some quotes from the thread:

Contra:

- It doesn't make sense that this would create statistics that would be detrimental to performance, unless the data at the time the statistics are gathered is substantially different than at the time of usage.
- If my data changes, and I analyze it, CBO should still find reasonable execution paths for the current data.

- However, there are cases where you really do need to get some statistics up to date - particularly for columns like timestamps or sequences that are always increasing in value.
- But I don't see a problem with gathering stale many times a day, every hour say[6]. If your tables aren't subject to much DML activity then they won't be analysed anyway.
- CBO is broke if fresh statistics result in poor performance

Pro:
- I recall a particular PS site where every time we analyzed, we got into trouble... That was with 8.0 and there was nothing we could do other than stop analyzing. Which we did and the problems went away (on that particular table).
- No, not at all. I'm really against this "tune-every-minute" approach. (Nuno Souto)
- Is analyzing over and over again not one of the symptoms of CTD? I would not analyze weekly, or because x % of the data has changed. I would analyze when response times degrade. Then you can search for the cause (trace the SQL involved) and do some analyzing, when appeared necessary. (Carel-Jan Engel)
- What the CBO thinks is the best path based on estimated cardinalities can be way off. By accident, an inefficient execution plan (as seen by the CBO) might actually be more efficient than the CBO's optimal choice. Analyzing can change these plans even if nothing is broken. (Henry Poras)

Diplomatic:
- Too many people do it too aggressively, too often and waste their time and the machine resources doing it for very little benefit. But if you have the time and resources, then it doesn't often do too much damage. (Jonathan Lewis)

Using DBMS_STATS in SQL Optimization

Means to Change an Access Plan

There are a number of ways to make the cost based optimizer choose a different access plan. Listed by increasing global impact:
- Change the statement
- Use a hint

These two have only local impact, isolated to the changed statement. Obviously, both require the ability to change the statement.
- Use stored outlines
  This too is isolated to the affected statement. Unlike the first two it can be used without having access to the source, even in cases where there is no source, where the SQL is generated "on the fly". However it requires that the statement's hash value is known and constant.
- Change statistics
  all SQL that use or reference the component whose statistics are changed may be impacted. The big question, of course, is: "How does one know which other SQL may be affected"? A possible technique is "Explain Plan Analysis"[7]
- Create or drop an index
  all SQL that use the table may be impacted. Again, how does one know what those are? The same "Explain Plan Analysis" addresses that and there are tools on the market which do that.
- Change initialization parameters
  This is the most global change since all SQL may be impacted. Unless the parameter can be changed for just a session, e.g. a batch job. Again, the "Explain Plan Analysis" aims at finding the SQL that are affected.

---

[6] Gathering statistics that frequently can play havoc with the parsing rates. [comment by the author]

7 An idea by the author waiting to be realized.

As the risk of changing the access paths of other SQL statements increases, so does the potential reward – rather than tuning SQL by SQL, many statements could be improved with a single change.
We will next look at an example that uses option 3 "change statistics" to improve the performance of a SQL statement without modifying the statement itself.

Using DBMS_STATS to Change an Access Plan

Statistics Affecting the CBO

In order to effectively use statistics in SQL tuning, one needs to know which statistics affect the CBO and how. The following are the statistics used by the cost based optimizer in deciding on an access plan.

Table - num_rows, blocks, avg_row_len
Column - num_distinct, density, num_nulls, low_value, high_value with histograms: buckets, endpoint_number, endpoint_value
Index - blevel, leaf_blocks, distinct_keys, avg_leaf_blocks_per_key, avg_data_blocks_per_key, clustering_factor

Rather than setting statistics directly, which would be possible with the DBMS_STATS.SET_xxx_STATS procedures, one could export the statistics, change the value(s) in the stattab table and then re-import the changed statistics back into the dictionary.

Overcome a Deficiency of Analyze and DBMS_STATS in Oracle 8

In Oracle 8 gathering histogram statistics for a column of a partitioned table does not create a global histogram, one for the column over the entire table, only for each partition individually. As we already established, Oracle 8 uses the analyze command to build the histogram, no matter what the granularity.
Table TP1 is range partitioned on column N1 with three partitions (`< 1000, < 2000, < MAXVALUE`).

```
GATHER_TABLE_STATS (NULL,'TP1',method_opt=>'FOR COLUMNS N2 SIZE
   99',granularity=>'DEFAULT');
ANALYZE TABLE "SCOTT"."TP1" COMPUTE STATISTICS FOR TABLE FOR COLUMNS N2 SIZE 99
```

The following column statistics show that the bucket size is still 1 and the density = 1/NDV on the global level after the analyze while on the partition level the bucket size is 2 (one less than NDV) and the density is orders of magnitude smaller than 1/NDV, typical for a frequency histogram.

table	column	NDV	density	nulls	lo	hi	av lg	bkts	G	U
TP1	N1	10,000	1.0000E-04	0	0	9999	3	1	N	N
	N2	3	3.3333E-01	0	0	999	2	1	N	N
	N3	1,000	1.0000E-03	0	0	999	3	1	N	N
table	column	NDV	density	nulls	lo	hi	av lg	bkts	G	U
TP1.P1	N1	1,000	1.0000E-03	0	0	999	3	1	N	N
	N2	3	4.9373E-05	0	0	999	2	2	N	N
	N3	1,000	1.0000E-03	0	0	999	3	1	N	N

The data from dba_histograms and dba_tab_histograms shows the extreme skewness of the data distribution of column n2 at the partition level while on the global table level only minimum and maximum were gathered and stored.

table	column	EP	value
TP1	N2	0	0
TP1	N2	1	999

table	column	EP	value
TP1.P1	N2	10106	0
TP1.P1	N2	10118	998
TP1.P1	N2	10127	999

Single Target Partition Known at Compile Time

If the optimizer can, at parse time, use partition pruning to narrow the plan to a single partition it uses the histogram data for that partition in the access path evaluation and choice:

```
select a.n1, b.n1, a.n3, a.d1, b.d1 from tp1 a, tp1 b
where a.n1 between 100 and 900 and a.n2 = 999
and a.n1 = b.n1 and a.n3 = b.n3

Rows     Row Source Operation
      9  NESTED LOOPS
     10   TABLE ACCESS BY LOCAL INDEX ROWID TP1 PARTITION: START=1 STOP=1
     10    INDEX RANGE SCAN PARTITION: START=1 STOP=1 (object id 101102)
      9   TABLE ACCESS BY LOCAL INDEX ROWID TP1 PARTITION: START=1 STOP=1
     90    INDEX RANGE SCAN PARTITION: START=1 STOP=1 (object id 101102)
```

The CBO realizes that the result is limited to rows from partition P1 and, using the histogram on TP1.P1.N2, the high selectivity of the predicate "n2 = 9999" and chooses an index access. A good choice as the execution statistics confirm:

call	count	cpu	elapsed	disk	query	current	rows
Parse	10	0.00	0.00	0	0	0	0
Execute	11	0.00	0.00	0	0	0	0
Fetch	20	0.13	0.13	0	2090	0	90
total	41	0.13	0.13	0	2090	0	90

No Single Target Partition Known at Compile Time

Shifting the range predicate on the partitioning column n1 such that it crosses the partition boundary (n1 < 1000), the optimizer now uses the global statistics and, in the absence of a global histogram, does not recognize the high selectivity of the predicate "n2 = 9999" and chooses a hash join with full table scans.

```
select a.n1, b.n1, a.n3, a.d1, b.d1from tp1 a, tp1 b
where a.n1 between 600 and 1400 and a.n2 = 999
and a.n1 = b.n1 and a.n3 = b.n3

Rows     Row Source Operation
     10  PARTITION RANGE ITERATOR PARTITION: START=1 STOP=2
     10   HASH JOIN
     10    TABLE ACCESS FULL TP1 PARTITION: START=1 STOP=2
   4017    TABLE ACCESS FULL TP1 PARTITION: START=1 STOP=2
```

call	count	cpu	elapsed	disk	query	current	rows
Parse	10	0.00	0.00	0	0	0	0
Execute	10	0.00	0.00	0	0	0	0
Fetch	20	4.68	4.83	96260	100160	1390	100
total	40	4.68	4.83	96260	100160	1390	100

We will use SET_COLUMN_STATS to build the missing global histogram, but

Before modifying statistics – and that includes analyze! - Always, always make a backup.

```
DBMS_STATS.EXPORT_TABLE_STATS (
  ownname => NULL, tabname => 'TP1',
  stattab => 'stats_table', statid => 'bkup');
```

We determine the global data distribution with a SQL and use the results as input to the helper procedure PREPARE_COLUMN_VALUES. The density value we simply "borrow" from one of the partition statistics:

```
select n2, count(0)
from tp1 group by n2;
      N2   COUNT(0)
       0      99943
     998         33
     999         25
```

```
DECLARE
   SREC DBMS_STATS.STATREC;
   NOVALS DBMS_STATS.NUMARRAY;
BEGIN
   SREC.EAVS := 0;
   SREC.CHVALS := NULL;
   SREC.EPC := 3;
   NOVALS := DBMS_STATS.NUMARRAY(0,998,999);
   SREC.BKVALS := DBMS_STATS.NUMARRAY(99943,33,25);
   DBMS_STATS.PREPARE_COLUMN_VALUES (SREC,NOVALS);
   DBMS_STATS.SET_COLUMN_STATS(NULL, 'TP1', 'N2', NULL,
      NULL, NULL, 3, .000049373, 0, SREC, 2, 2);
   END;
```

Afterwards the global column statistics show the presence of the frequency histogram and, of course, the modified density.

table	column	NDV	density	nulls	lo	hi	av lg	bkts	G	U
TP1	N1	10,000	1.0000E-04	0	0	9999	3	1	N	N
	N2	3	4.9373E-05	0	0	999	2	2	Y	N
	N3	1,000	1.0000E-03	0	0	999	3	1	N	N

And we have a global frequency histogram.

table	column	EP	value
TP1	N2	99943	0
TP1	N2	99976	998
TP1	N2	100001	999

But, just like every tuning effort, it only matters if it makes a difference in the optimizer's access path choice and its performance. With the improved information about the selectivity of "n2 = 999" on the global level the optimizer chooses a plan similar to the one when only one partition was involved, except now it has to iterate over two partitions.

```
Rows     Row Source Operation
     10  PARTITION RANGE ITERATOR PARTITION: START=1 STOP=2
     10   NESTED LOOPS
     12    TABLE ACCESS BY LOCAL INDEX ROWID TP1 PARTITION: START=1 STOP=2
     12     INDEX RANGE SCAN PARTITION: START=1 STOP=2 (object id 101102)
     10    TABLE ACCESS BY LOCAL INDEX ROWID TP1 PARTITION: START=1 STOP=2
    110     INDEX RANGE SCAN PARTITION: START=1 STOP=2 (object id 101102)
```

And the performance is comparable except that it takes twice as long to iterate over two partitions compared to the single partition access path.

call	count	cpu	elapsed	disk	query	current	rows
Parse	10	0.02	0.02	0	0	0	0
Execute	11	0.00	0.00	0	0	0	0
Fetch	20	0.15	0.24	65	2270	0	100
total	41	0.17	0.26	65	2270	0	100

TUNE A SQL WHERE THE SOURCE IS INACCESSIBLE

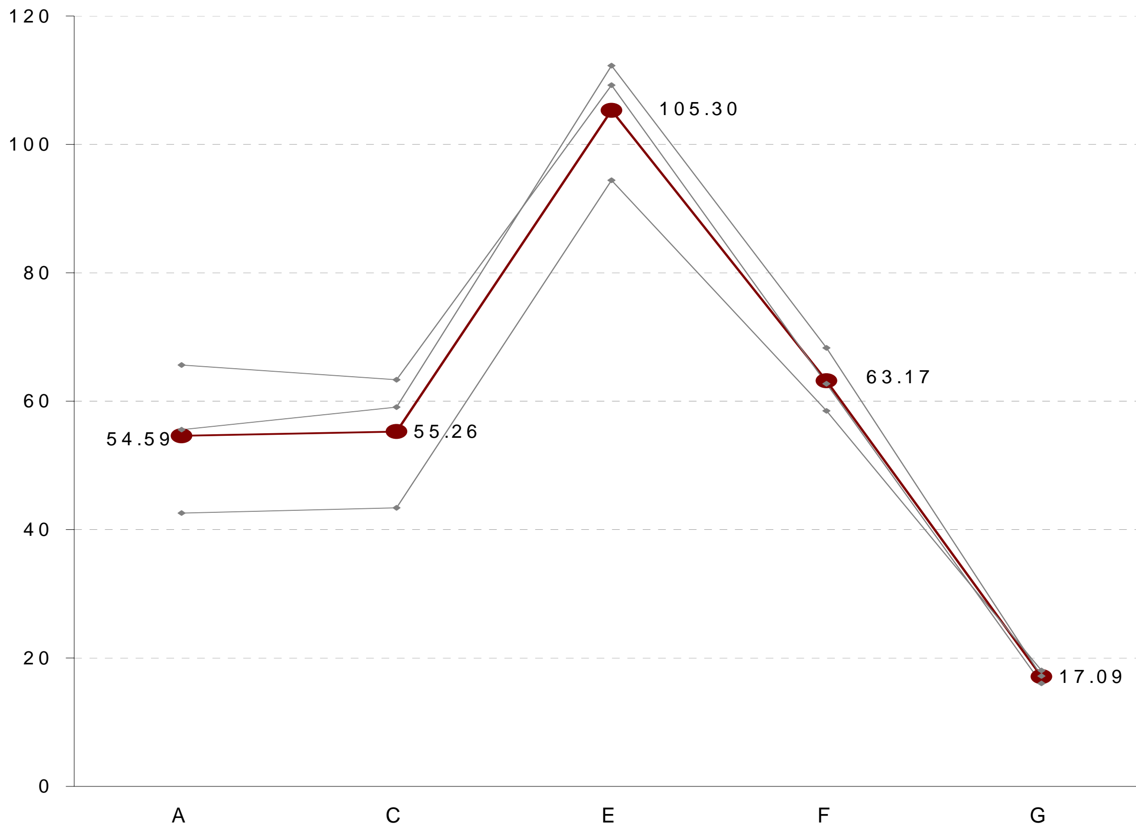
After the thread on the Oracle-L list, inspired by Dave Ensor's remark in his presentation at UKOUG, I decided to create a demonstration for my presentation at the Hotsos symposium based on one of my tuning cases, actually the first where I employed the "Tuning by Cardinality Feedback" method. I wanted the demo to satisfy four conditions:

- Demonstrate the danger of scheduled statistics refreshes by showing how the performance of a query deteriorates not because of the DML activity, but because of the analyze.
- Show that it is possible to improve the performance of a SQL statement by targeted modification of statistics without touching the SQL itself.
- In addition to those two main goals, the testcase should be "immune" to often touted setting of the initialization parameters OPTIMIZER_INDEX_COST_ADJ and OPTIMIZER_INDEX_CACHING.
- Last but not least, the testcase – five executions of the SQL – had to finish during my presentation.

This last condition proved to be the most challenging to attain. I had to scale back two of the tables to find that balance where the demo would exhibit all the traits I wanted it to, but still finish in a reasonable time. Of course the differences between the plans are now much less pronounced than in the real case. In a batch job who would really care that it takes a minute longer, but there it did not finish at all – or more correctly we did not have the time and patience to let it finish. The same testcase produces very similar results under Oracle 9i (9.2.0.4) where it is even impervious to dynamic sampling which is supposed to be a solution to the predicate dependence problem. Just the run times and the exact "trigger points" where a statistics change causes a plan change are different.
The case is based on a Peoplesoft HR system. Due to the sensitivity of the original content all data in the testcase tables is random generated such that the resulting statistics are identical to the original statistics except for the two scaled back tables. The WB_ prefixed tables are the trimmed down tables. The real PS_RETROPAY_EARNS tables was more than 10 times the size (~ 1.5 million rows). The SQL and the plans in this demo are slightly different from the ones in reality, but the point I want to make and the tuning solution by cardinality feedback are identical.
Refer to appendix B for the table, column, and index statistics of the tables and the SQL itself.
The testcase consists of the following steps:

1. Execute the SQL.
2. Make a change to the data. The DML is the following
3. `INSERT INTO WB_JOB (EMPLID, EMPL_RCD#, EFFDT, EFFSEQ, JOBCODE, POSITION_NBR, EMPL_STATUS, COMPANY, PAYGROUP, DEPTID) values('005792', 0, trunc(sysdate), 1, '001DAD', '00005839', 'A', 'ECC', 'ECA', '0000000339');`
   simulating a change to the employee effective today. It could be a change to any or all of the attributes JOBCODE through DEPTID.
4. Rerun the SQL to verify that the performance does not change as a result of this change.
5. Analyze the WB_JOB table
6. Rerun the SQL and note the performance degradation.
7. Attempt to tune the sql by setting optimizer_index_cost_adj = 25 and optimizer_index_caching = 90.
8. The performance returns to match that of steps A and C (almost), but the plan differs and becomes less scalable. Furthermore, it would deteriorate rapidly and drastically if the slimmed tabled were to be scaled back up.
9. Use "Tuning by Cardinality Feedback" and achieve the best performance of all four plans.

Below is a chart of the elapsed times of the query. They are an average from three separate runs of the demo to minimize the effect of random variations in the times. The labels correspond to steps above.



One may assume that it is the low cardinalities of WB_JOB.EFFDT and WB_JOB.EFFSEQ that make the plan so vulnerable to change from just small variations in the statistics. Truth is that in the original case there were > 20,000 different EFFDT and 11 EFFSEQ. Even with this scaled down example, the SQL continues to vacillate between plans as the cardinalities of EFFDT and EFFSEQ change (increase).

A further plus of my proposed (and implemented) tuning solution is the fact that it stabilizes the access path and the performance. Although it is still subject to gradual deterioration as the size of the tables grows, that deterioration is linear in relation to the table growth. I suppose any solution that includes a "ban" on table re-analyze will freeze the access path (ever heard of plan stability?!)

Tuning by Cardinality Feedback

Observation

If an access plan is not optimal it is because the cardinality estimate for one or more of the row sources is off the mark.

1. List the explain plan with the cardinality projections
2. Get the actual row counts from a SQL trace or from V$SQL_PLAN. Make sure the actual plan is identical to the explain plan. This is the feedback part.
3. Look for the first (innermost) row source where the ratio of actual/estimated cardinality is orders of magnitude – usually at least in the 100s
4. 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

In the table below, on the right (in black) the output of an explain plan of the SQL statement with the statistics as of "E" (after the analyze of WB_JOB). To the left of it (in blue) are the actual cardinalities from the SQL trace (tkprof) of run "E" and at the far left the ratios of actual / estimated cardinality. Highlighted in red and bold is the operation where the estimate and actual differ significantly for the first time in the sequence of operations.

```
           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
```

Next we look at what tables contribute to the row source. In this case , a join, there are two tables, PS_PAY_CALENDAR and WB_JOB. We therefore scrutinize the statistics of all columns used in a predicate in the SQL at hand. The estimate for table PS_PAY_CALENDAR matches the actual rowcount so we focus on WB_JOB, although the problem could– and to a certain degree does – lie in the join cardinality estimation.

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	**9**	**1.1111E-01**	1995-01-01	2003-01-01	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

Looking at the cardinality estimate for WB_JOB it is off by a factor of 3, which, curiously, is the cardinality of EFFSEQ. If we look at EFFDT and EFFSEQ closely we realize that they are not independent of EMPLID, or in other words, not every employee record occurs 27 times with 9 different effective dates and 3 different sequence numbers. If they were we would have 27 * 26,167 = 706,509 rows in WB_JOB[8] instead of 40,000.

[8] In the real case, with >20,000 EFFDTs and 11 EFFSEQs the predicate independence violation was far more marked.

We will counteract, even overcompensate, that assumption by setting the density[9] of both columns to 1.

```
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	9	**1.0000E+00**	1995-01-01	2003-01-01	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

And this is the plan of the SQL in step G:

```
          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
```

Observations

- The ratios of actual/estimated are much smaller
- The cardinalities of 4 of the 5 tables are estimated accurately
- 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.

Before you begin to question my sanity consider that both IBM and Microsoft have automated and implemented feedback in the optimizers of DB2 and SQLServer respectively. (Bruno and Chaudhuri, 2002, Markl and Lohman, 2002, Stillger, et al., 2001)
Below is a comparison of a partial latch and statistics profile of the 4 access plans[10] based on Tom Kyte's test harness. Only latches and statistics with significant differences between the 4 plans are included here. Except for the sort statistics to show that the sort work is the identical for all.

NAME	A	C	E	F	G

[9] That proves enough in this case. Depending on circumstances modifying NDV may be required too.

[10] Plans A and C are of course identical.

LATCH.cache buffers chains	353,482	353,952	371,852	547,254	128,987
LATCH.multiblock read objects	948	952	948	850	1,232
LATCH.sort extent pool	18	19	61	6	25
STAT...CPU used by this session	267	252	617	320	188
STAT...buffer is not pinned count	110,977	110,977	110,977	233,992	10,326
STAT...buffer is pinned count	121,211	121,211	121,118	166,025	33,847
STAT...consistent gets	197,895	197,897	197,714	283,380	78,089
STAT...db block gets	83	88	83	72	170
STAT...physical reads	7,418	7,698	18,751	8,488	6,466
STAT...physical reads direct	1,604	1,604	12,678	17	2,770
STAT...session logical reads	197,978	197,985	197,797	283,452	78,259
STAT...sorts (disk)	1	1	1	1	1
STAT...sorts (memory)	11	11	11	11	11
STAT...sorts (rows)	14,124	14,124	14,124	14,124	14,124
STAT...table fetch by rowid	88,160	88,160	88,160	149,864	0
STAT...table scan blocks gotten	3,786	3,786	3,786	3,399	10,324
STAT...table scan rows gotten	55,974	55,974	55,974	40,015	220,707
STAT...table scans (long tables)	2	2	2	1	3
STAT...table scans (short tables)	2	2	2	1	2

A Few Notes:

- Plan F (with OPTIMIZER_INDEX_COST_ADJ = 25) shows a marked increase in consistent gets (and thus logical reads), accompanied by an increase in "cache buffer chains" latch requests making it vulnerable to multi-user and scaling issues.
- Plan G proves once more that full table scans are not evil.
- Plan E and G have the lowest BCHR of the four plans - 90.52% and 91.74% respectively – while having the worst and best performance, proving yet again the inadequacy of that measure as a performance indicator.

Appendix

```
Name                          Null?     Type
PK1                                     NUMBER
PK2                                     NUMBER
D1                                      DATE
D2                                      NUMBER
D3                                      VARCHAR2(2000)
```

SQL and Object Statistics of the DEMO Test Case

The 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
```

Table Statistics

TABLE_NAME	free	used	rows	blks	empty	avg row
PS_RETROPAYPGM_TBL	8	65	15	1	6	65
PS_PAY_CALENDAR	8	65	2,280	68	3	94
PS_RETROPAY_RQST	10	40	13,679	319	0	78
WB_JOB	10	40	40,000	3,398	3	282
WB_RETROPAY_EARNS	8	65	164,733	6,538	3	138

Column Statistics

table	column	NDV	density	lo	hi	av lg
PS_RETROPAYPGM_TBL	RETROPAY_PGM_ID	15	6.6667E-02	001	015	4
	EFFDT	7	1.4286E-01	1901-01-01	2003-05-15	8
	EFF_STATUS	1	1.0000E+00	A	A	2
	OFF_CYCLE	1	1.0000E+00	N	N	2
PS_PAY_CALENDAR	COMPANY	11	9.0909E-02	ACE	TES	4
	PAYGROUP	15	6.6667E-02	ACA	TEP	4
	PAY_END_DT	160	6.2500E-03	1998-01-18	2004-02-22	8
	RUN_ID	240	4.1667E-03		PP2	4
	PAY_OFF_CYCLE_CAL	2	5.0000E-01	N	Y	2
	PAY_PERIOD	1	1.0000E+00			2
PS_RETROPAY_RQST	RETROPAY_SEQ_NO	13,679	7.3105E-05	2061843	2075521	8
	EMPLID	8,304	1.2042E-04	0000302	0012670	8
	EMPL_RCD#	1	1.0000E+00	0	0	2
	RETROPAY_PGM_ID	14	7.1429E-02	001	014	4
table	column	NDV	density	lo	hi	av lg
WB_JOB	EMPLID	26,167	3.8216E-05	000036	041530	7
	EMPL_RCD#	1	1.0000E+00	0	0	2
	EFFDT	9	1.1111E-01	1995-01-01	2003-01-01	8
	EFFSEQ	3	3.3333E-01	1	3	3
	COMPANY	10	1.0000E-01	ACE	TES	4
	PAYGROUP	14	7.1429E-02	ACA	TEP	4
WB_RETROPAY_EARNS	RETROPAY_SEQ_NO	14,532	6.8814E-05	2061843	2076374	8
	COMPANY	5	2.0000E-01	ACE	TES	4
	PAYGROUP	7	1.4286E-01	ACA	TEP	4
	PAY_END_DT	56	1.7857E-02	1999-08-15	2001-09-23	8
	OFF_CYCLE	2	5.0000E-01	N	Y	2
	EMPLID	38,926	2.5690E-05	000036	041531	7
	EMPL_RCD#	1	1.0000E+00	0	0	2
	RETROPAY_PRCS_FLAG	3	3.3333E-01	C	X	2
	RETROPAY_LOAD_SW	2	5.0000E-01	N	Y	2

Index Statistics

Table	index	column	NDV	DENS	#LB	LVL	CLUF	
PS_RETROPAYPGM_TBL	PS#RETROPAYPGM_TBL					15		10
	1							
		RETROPAY_PGM_ID	15	6.67E-02				
		DESCR	15	6.67E-02				
	PS0RETROPAYPGM_TBL				15		1	01
		DESCR	15	6.67E-02				
		RETROPAY_PGM_ID	15	6.67E-02				
		EFFDT	7	1.43E-01				
	PS_RETROPAYPGM_TBL		U		15		1	01
		RETROPAY_PGM_ID	15	6.67E-02				
		EFFDT	7	1.43E-01				

Table	index	column	NDV	DENS	#LB	LVL	CLUF	
PS_PAY_CALENDAR	PS#PAY_CALENDAR		2,280			24	1	877
		COMPANY	11	9.09E-02				
		PAYGROUP	15	6.67E-02				
		PAY_END_DT	160	6.25E-03				
		RUN_ID	240	4.17E-03				
		PAY_SHEETS_RUN	1	1.00E+00				
		PAY_CALC_RUN	1	1.00E+00				
		PAY_CONFIRM_RUN	2	5.00E-01				
	PS0PAY_CALENDAR		2,280			22	1	2,247
		RUN_ID	240	4.17E-03				
		PAY_CONFIRM_RUN	2	5.00E-01				
		COMPANY	11	9.09E-02				
		PAYGROUP	15	6.67E-02				
		PAY_END_DT	160	6.25E-03				
	PS_PAY_CALENDAR	U	2,280			17	1	877
		COMPANY	11	9.09E-02				
		PAYGROUP	15	6.67E-02				
		PAY_END_DT	160	6.25E-03				

Table	index	column	NDV	DENS	#LB	LVL	CLUF	
PS_RETROPAY_RQST			PS#RETROPAY_RQST		13,254		96	113,641
			EMPLID	8,304	1.20E-04			
			MASS_RETRO_RQST_ID			147.14E-02		
			RETROPAY_PRCS_FLAG			33.33E-01		
	PS0RETROPAY_RQST				13,679		158	113,625
			MASS_RETRO_RQST_ID			147.14E-02		
			RETROPAY_SEQ_NO	13,679	7.31E-05			
			EMPLID	8,304	1.20E-04			
			RETROPAY_EFFDT	285	3.51E-03			
			RETROPAY_PRCS_FLAG			33.33E-01		
	PS1RETROPAY_RQST				13,679		158	113,635
			RETROPAY_PRCS_FLAG			33.33E-01		
			RETROPAY_SEQ_NO	13,679	7.31E-05			
			EMPLID	8,304	1.20E-04			
			RETROPAY_EFFDT	285	3.51E-03			
			MASS_RETRO_RQST_ID			147.14E-02		
	PS_RETROPAY_RQST			U	13,679		131	113,637
			RETROPAY_SEQ_NO	13,679	7.31E-05			
			EMPLID	8,304	1.20E-04			
			RETROPAY_EFFDT	285	3.51E-03			

Table	index	column	NDV	DENS	#LB	LVL	CLUF	
WB_JOB	WBBJOB			14		356	2	27,443
			COMPANY	10	1.00E-01			
			PAYGROUP	14	7.14E-02			
	WB_JOB		U	40,000		434	2	39,989
			EMPLID	26,167	3.82E-05			
			EMPL_RCD#	1	1.00E+00			
			EFFDT	9	1.11E-01			
			EFFSEQ	3	3.33E-01			

Table	index	column	NDV	DENS	#LB	LVL	CLUF
WB_RETROPAY_EARNS	WB0RETROPAY_EARNS		164,733		3,231	2	164,665
		EMPLID	38,926	2.57E-05			
		EMPL_RCD#	1	1.00E+00			
		RETROPAY_SEQ_NO	14,532	6.88E-05			
		COMPANY	5	2.00E-01			
		PAYGROUP	7	1.43E-01			
		PAY_END_DT	56	1.79E-02			
		OFF_CYCLE	2	5.00E-01			
		PAGE#	1	1.00E+00			
		LINE#	1	1.00E+00			
		ADDL#	1	1.00E+00			
		EARNS_TYPE	7	1.43E-01			
		ERNCD	1	1.00E+00			
		SEQ_NO	1	1.00E+00			
		RETROPAY_EFFDT	56	1.79E-02			
	WBBRETROPAY_EARNS		164,733	0.00E+00	2,657	2	164,733
		COMPANY	5	2.00E-01			
		PAYGROUP	7	1.43E-01			
		PAY_END_DT	56	1.79E-02			
		OFF_CYCLE	2	5.00E-01			
		PAGE#	1	1.00E+00			
		LINE#	1	1.00E+00			
		ADDL#	1	1.00E+00			
		EARNS_TYPE	7	1.43E-01			
		ERNCD	1	1.00E+00			
		SEQ_NO	1	1.00E+00			
		RETROPAY_PRCS_FLAG	3	3.33E-01			
		RETROPAY_LOAD_SW	2	5.00E-01			
		RETROPAY_SEQ_NO	14,532	6.88E-05			
	WB_RETROPAY_EARNS U		164,733	1.00E+02	2,423	2	6,538
		RETROPAY_SEQ_NO	14,532	6.88E-05			
		COMPANY	5	2.00E-01			
		PAYGROUP	7	1.43E-01			
		PAY_END_DT	56	1.79E-02			
		OFF_CYCLE	2	5.00E-01			
		PAGE#	1	1.00E+00			
		LINE#	1	1.00E+00			
		ADDL#	1	1.00E+00			
		EARNS_TYPE	7	1.43E-01			
		ERNCD	1	1.00E+00			
		SEQ_NO	1	1.00E+00			

Metalink Notes

114671.1 Gathering Statistics for the Cost Based Optimizer
130899.1 How to Set User-Defined Statistics Instead of RDBMS Statistics
122009.1 How to Retrieve Statistics Generated by ANALYZE SQL Statement

130688.1	Report Statistics for a Table, it's columns and it's indexes with DBMS_STATS
130911.1	How to Determine if Dictionary Statistics are RDBMS-Generated or User-Defined
102334.1	How to automate ANALYZE TABLE when changes occur on tables
1074354.6	DBMS_STATS.CREATE_STAT_TABLE: What Do Table Columns Mean?
117203.1	How to Use DBMS_STATS to Move Statistics to a Different Database
149560.1	Collect and Display System Statistics (CPU and IO) for CBO usage
153761.1	Scaling the system to improve CBO optimizer

Bibliography

Christodoulakis, S. (1984). *Implications of Certain Assumptions in Database Performance Evaluation*. ACM Transactions on Database Systems (TODS), 9(2).

Breitling, Wolfgang. *Fallacies of the Cost Based Optimizer*. presented at the Hotsos Symposium on Oracle Performance, Dallas, Texas, February 10-12 2003.

Breitling, Wolfgang. *A Look under the Hood of CBO: The 10053 Event*. presented at the Hotsos Symposium on Oracle Performance, Dallas, Texas, February 10-12 2003.

References

A76936-01. *Oracle8i - Supplied PL/SQL Packages* Oracle Corporation, 1999. Available from http://otn.oracle.com/pls/tahiti/tahiti.docindex.

A76955-01. *Oracle8i - Utilities* Oracle Corporation, 1999. Available from http://otn.oracle.com/pls/tahiti/tahiti.docindex.

A96612-01. *Oracle9i - Supplied PL/SQL Packages and Types Reference* Oracle Corporation, 2002. Available from http://otn.oracle.com/pls/db92/db92.homepage?remark=tahiti.

A96652-01. *Oracle9i - Utilities* Oracle Corporation, 2002. Available from http://otn.oracle.com/pls/db92/db92.homepage?remark=tahiti.

Bruno, Nicolas, and Surajit Chaudhuri. *Exploiting Statistics on Query Expressions for Optimization.* presented at the ACM SIGMOD international conference on Management of data, Madison, Wisconsin 2002.

Markl, Volker, and Guy Lohman. *Learning Table Access Cardinalities with LEO.* presented at the ACM SIGMOD international conference on Management of data, Madison, Wisconsin, June 4-6 2002.

Stillger, Michael, Guy Lohman, Volker Markl, and Mokhtar Kandil. *LEO – DB2's Learning Optimizer.* presented at the 27th International Conference on Very Large Data Bases (VLDB), Rome, Italy 2001.

About the Author

Wolfgang Breitling has over 25 years experience working in the IT industry, and of those more than 20 years working with databases. After a university degree in mathematics he joined IBM Germany in the QA department where he was involved in the development of an OS kernel for /370 hardware architecture testing and a test harness to measure the performance of /370 operation codes. After leaving IBM, Mr. Breitling worked in Switzerland and Canada with many database systems (DL/1, IMS, Adabas, SQL/DS, and DB2) and since 1993 Oracle. In 1996 he founded Centrex Consulting Corporation (www.centrexcc.com) which specializes in Oracle and Peoplesoft administration and optimization. In this capacity he has contracted among others to Anderson Consulting (now Accenture), Oracle and IBM. In recent years, Mr. Breitling has given presentations at IOUG conferences, at the 2003 Hotsos Symposium on Oracle Performance, and at the 2003 AOTC spring conference.