HISTOGRAMS - MYTHS AND FACTS

Wolfgang Breitling, Centrex Consulting Corporation[i]

This paper looks at some common misconceptions about histograms. The hope is that the paper and the facts on histograms presented and demonstrated (with repeatable test scripts) will help prevent the misconceptions from becoming myths.

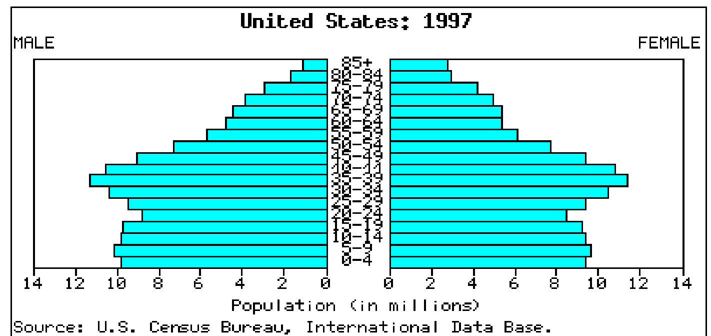
The findings presented are based on experience and tests with Oracle 9*i* (9.2.0.4 through and 9.2.0.7) on Windows 2000, Linux Redhat ES 3, Solaris 8, and AIX 5.2. Some facts have been verified on Oracle 10g (10.1.0.3 and 10.1.0.4) on Windows 2000 and Linux Redhat ES 3.

TWO TYPES OF HISTOGRAMS

First a word, well, a few words, about histograms in general. Oracle uses two types of histograms.

Frequency or equi-width histogram

This is what most people associate with the term histogram. Prior to Oracle9*i* it was called a "value based histogram" in Oracle manuals. The term "equi-width" is used in papers and conference proceedings about optimizer theory and describes how the histogram buckets are defined: the range of values of the column (dba_tab_columns.num_distinct) is divided into n buckets – where n is the number of buckets requested – with each bucket, except perhaps the last, assigned a range of "ceiling(num_distinct/n)" values. One of the best known frequency histograms is the "age pyramid", actually two histograms – one for the male population and a one for the female. Each bucket/bar, except the last, has a width of 5, i.e. covers 5 ages. Note that Oracle will only create equi-width histograms with a width of 1, i.e. each bucket contains the count of rows for only 1 column value: number of buckets = number of distinct values for that column[α]. Because the column values are pre-assigned to a particular bucket, the histogram collection consists – conceptually at least – simply of reading every row, examining the value of the histogram column and incrementing the row count of the corresponding bucket. Just like counting ballots and incrementing the votes for each candidate. Note that equi-width histograms for multiple columns could be collected with one pass through the table's rows. Note also that Oracle does not do that. It does a separate scan of the table for each histogram requested.

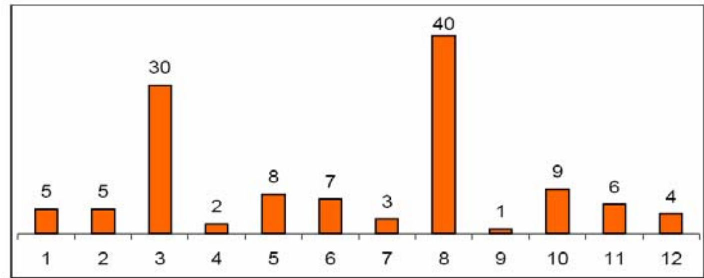


In the Oracle case, single value buckets, a frequency histogram is simply the count of rows for each value. However, Oracle does not store the values and the individual counts, but rather the running total in DBA_HISTOGRAMS. See SQL scripts in Appendix

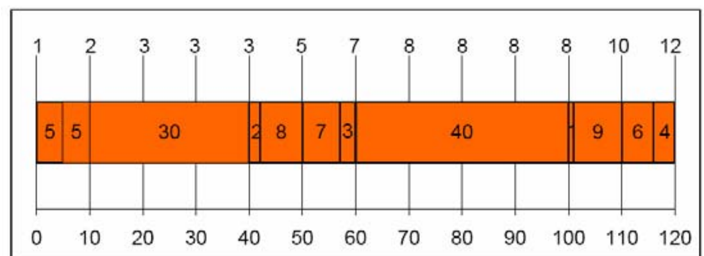
[α] Note, however, that the reverse is not true – requesting a histogram with "size NUM_DISTINCT" will not necessarily create a frequency histogram when using Oracle 8i, 9i and 10.1 DBMS_STATS. In fact for these versions it will almost certainly not.

Height-balanced or equi-depth histogram

While for equi-width histograms the number of column values per bucket is fixed and pre-determined, and the number of rows per bucket depends on the distribution of column's values, it is the other way around for equi-depth histograms. Here the number of rows per bucket is fixed and predetermined as "ceiling(num_rows/n)" – where n is the number of buckets – and the number of distinct values in each bucket depends on the distribution of the column's values. The two graphs represent the same value distribution (1 through12) for a 120 row table with 12 buckets – once as a frequency histogram and once as a height-balanced histogram.



In order to gather a height-balanced histogram, the rows must be sorted by the column's value. This immediately rules out the possibility to collect a histogram for more than one column at a time. Then – conceptually – the rows are "filled" into the buckets, "lapping" into the next bucket as each one fills. Thus a bucket can represent rows for many column values, or the rows for one column value can spill into several buckets. The latter ones (3 and 8 in the example) are referred to as "popular values". In reality, Oracle does not "fill the buckets", it is merely interested in the column value of each m-th row of the sorted rowset (where m is the determined size/capacity of each bucket) plus the column values of the first and last row. Height-balanced histograms are very vulnerable to where the bucket boundaries fall in the sorted rowset, based on the number and thus the size of the buckets. It is not uncommon that a most frequently occurring value is missed being recognized as a popular value, but a value in 2nd or 3rd place in the rank of column frequencies does show up as a popular value. Small changes in the distribution of data values can easily change which values are found to be "popular".



Histograms and Bind Variables

Histograms and bind variables have a somewhat strained relationship. They do not really go well together. A common misconception, however, if not to say myth, is that histograms are ignored when the parsed SQL uses bind variables. There are two corrections to that misconception depending on the Oracle version.

Prior to Oracle 9i

In many of its cost calculations, the CBO uses the column statistics "density" rather that the value of 1/num_distinct for the selectivity of a (equality) predicate in order to estimate the cardinality of a row source in the plan. For columns without a histogram density equals 1/num_distinct, so the cardinality estimates are the same, regardless. However, as we'll explore later in this paper, once histograms are involved, the density calculation is different, the row source cardinality estimate can be different, and therefore, ultimately, the optimizer may very well choose a different path, solely by virtue of the presence of the histogram. See [1] for an example.

Oracle 9*i* and 10g

With Oracle9*i* Oracle introduced "bind variable peeking". Whenever the optimizer is called upon to parse a SQL statement and compile an access plan (hard parse), it uses the value for the bind variable for that

execute request as if it had been coded as a literal in the SQL. Thus it can make full use of any available histograms. However, all future executes of this SQL will use this same plan, regardless of any difference in bind variable values, as long as the SQL remains valid in the shared pool. If that first hard parse uses an uncharacteristic bind value from the rest of the "mainstream" SQL, then this bind variable peeking can backfire badly. There are a few ways to deal with that:

a) Execute the SQL in a startup trigger with bind values corresponding to the typical use – and then lock the SQL in the shared pool with the DBMS_SHARED_POOL.KEEP procedure.

b) Create a stored outline – or profile in Oracle10g – for the SQL when used with bind values corresponding to the typical use and set "USE_STORED_OUTLINES" for those sessions interested.

c) Disable bind variable peeking by setting "_OPTIM_PEEK_USER_BINDS=false". With this the optimizer returns to the pre-Oracle9*i* calculation of selectivities of columns with histograms. Since this is an undocumented (and unsupported) initialization parameter, it would be safer to just delete the histograms
   - unless there are also SQL that use literals and for which the histogram(s) make a difference

   or
   - unless, of course, you rely on the side-effect of a histogram on the selectivity as described above. However, that can just as well be achieved by changing just the density (with the DBMS_STATS.SET_COLUMN_STATS procedure) without going through the trouble to collect a histogram.

The stored outline / profile approach appears to be the most promising.

HISTOGRAMS ON NON-INDEXED COLUMNS

A widespread misconception about histograms is that they are only useful and thus needed on indexed columns. The origin of this misconception may lie in the fact that there are two basic access methods for a table, a full scan and an index access. If the column is indexed then the histogram can tell the optimizer that for a low-cardinality value the index access will be more efficient (read "faster") while for a high-cardinality value a full table scan will be more efficient. If the column is not indexed, so the argument goes, then the only possible access path is a full table scan – unless, of course, other, indexed predicates allow for an index access – so what possible difference can a histogram make. The fact that Oracle offers a "for all indexed columns" option in both, the old analyze as well as the new gather_table_stats statistics gathering methods appears to lend credence to this argument.

What the argument misses, however, is that column value cardinality information is not only used for evaluating the cost of base table access paths, but also, and more importantly, for evaluating the cost of different join methods and orders. If the cardinality information from a non-indexed column with a histogram helps the optimizer to estimate a low cardinality for that table, it can make sense to use that table as the outer table in a NL (nested loop) join early on in the access path instead of in a SM (sort merge) or HA (hash) join later in the access path, leading to smaller intermediate rowsets and overall faster execution.

An example will best illustrate the usefulness of a histogram on a non-indexed column. To disprove[1] the misconception we create 3 tables with identical structure, unique (primary key) indexes and some other indexes (See appendix A for the script to reproduce the test). Our focus is on column t1.d3 which is deliberately not indexed to make the point. The tables are loaded in such a way that the distribution of the data in

---

[1] It is often easier to disprove something – provided of course it is not true – because all one needs is a counter example.

column d3 of table t1 is extremely skewed with all but 2 of the 62,500 values being 1 whereas the same columns in tables t2 and t3 have uniformly distributed data.

```
select d3, count(0)
from t1 group by d3;

        D3  COUNT(0)
---------- ---------
         0         2
         1     62498
```

```
select d3, count(0)
from t2 group by d3;

        D3  COUNT(0)
---------- ---------
         1      3087
         2      3162
         3      3119
         4      3223
         5      3175
         6      3114
         7      3059
         8      3092
         9      3207
        10      3104
        11      3122
        12      3182
        13      3117
        14      3107
        15      3088
        16      3161
        17      3131
        18      3043
        19      3120
        20      3087
```

```
select d3, count(0)
from t3 group by d3;

        D3  COUNT(0)
---------- ---------
         1      3121
         2      3124
         3      3068
         4      3097
         5      3114
         6      3108
         7      3098
         8      3108
         9      3166
        10      3042
        11      3200
        12      3128
        13      3137
        14      3172
        15      3160
        16      3172
        17      3197
        18      3201
        19      3033
        20      3054
```

Since neither t2.d3 nor t3.d3 are going to be used in the query, their data distribution does not even matter, really. The SQL we are going to execute is the following:

```
select sum(t1.d2*t2.d3*t3.d3)
from t1, t2, t3
where t1.fk1 = t2.pk1
  and t3.pk1 = t2.fk1
  and t3.d2 = 35
  and t1.d3 = 0;
```

The sum in the select is there only to reduce the resultset to a single row so that we don't have to watch 100s or 1000s of rows scroll by and that therefore the performance effect to be demonstrated is not being thwarted by the time taken to render the result. Below is the tkprof output of the sql trace. As you can see, the "sort aggregate" operation to produce the sum adds a negligible 1264 microseconds (0.014%) to the runtime:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.00	0.01	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	1.57	8.97	44109	47534	0	1
total	4	1.57	8.99	44109	47534	0	1

```
Rows      Row Source Operation
      1   SORT AGGREGATE (cr=47534 r=44109 w=0 time=8976818 us)
   1392    HASH JOIN  (cr=47534 r=44109 w=0 time=8975554 us)
      2     TABLE ACCESS FULL T1 (cr=23498 r=22165 w=0 time=4159263 us)
 175296     HASH JOIN  (cr=24036 r=21944 w=0 time=4625555 us)
    627      TABLE ACCESS BY INDEX ROWID T3 (cr=632 r=0 w=0 time=3727 us)
    627       INDEX RANGE SCAN T3X (cr=7 r=0 w=0 time=583 us)(object id 226985)
  62500      TABLE ACCESS FULL T2 (cr=23404 r=21944 w=0 time=4430244 us)
```

Let us see if and how things change when we create a histogram on the non-indexed(!) column t1.d3:

```
exec dbms_stats.gather_table_stats(null, 't1',
  method_opt=>'for columns size skewonly d3');
```

This creates a frequency histogram for t1.d3. The new sql_trace – collecting the histogram invalidated any parsed plan dependent on t1 – shows the changed plan:

```
call       count       cpu    elapsed       disk      query    current        rows
Parse          1      0.00       0.08          0          0          0           0
Execute        1      0.00       0.00          0          0          0           0
Fetch          2      0.68       4.72      22161      24313          0           1
total          4      0.68       4.81      22161      24313          0           1

Rows      Row Source Operation
      1   SORT AGGREGATE (cr=24313 r=22161 w=0 time=4722357 us)
   1392    HASH JOIN  (cr=24313 r=22161 w=0 time=4720858 us)
    627     TABLE ACCESS BY INDEX ROWID T3 (cr=632 r=0 w=0 time=589957 us)
    627      INDEX RANGE SCAN T3X (cr=7 r=0 w=0 time=9110 us)(object id 226985)
    500     TABLE ACCESS BY INDEX ROWID T2 (cr=23681 r=22161 w=0 time=4109511 us)
    503      NESTED LOOPS  (cr=23504 r=21984 w=0 time=4045239 us)
      2       TABLE ACCESS FULL T1 (cr=23498 r=21980 w=0 time=4020665 us)
    500       INDEX RANGE SCAN T2P (cr=6 r=4 w=0 time=23835 us)(object id 226986)
```

Obviously, not only did the plan change as a result of the histogram, it changed for the better. The sql executed more than twice as fast and used less than half (43%) of cpu time[2]. This clearly disproves the notion that a histogram on a non-indexed column is pointless.

Clearly, the testcase was constructed with the purpose of demonstrating the benefit of a histogram on a non-indexed column. But what if this was a real case? How would we "know" that a histogram would be the tuning answer[3]? In my presentation at the 2004 Hotsos Symposium[2], I introduced the method of "Tuning by Cardinality Feedback"[3] based on the notion and experience that whenever the optimizer picks a bad plan, one or more of the cardinality estimates in the plan are off by orders of magnitude and, vice versa, the plan is all but optimal if the cardinality estimates are correct. This testcase is a perfect example of that tuning method.

```
    Card    Rows  Row Source Operation
       1       1  SORT AGGREGATE              (cr=47534 r=44109 w=0 time=8976818 us)
21229620    1392   HASH JOIN                  (cr=47534 r=44109 w=0 time=8975554 us)
```

---

2 Of course, depending on your hardware, the OS and the exact Oracle release, your results may vary somewhat, but the case with the histogram should always be the faster.

3 As an aside, an index on t1.d3 does not change the plan and therefore does not make the SQL perform better. Try it out. The histogram is the tuning answer here.

```
  31250       2     TABLE ACCESS FULL T1            (cr=23498 r=22165 w=0 time=4159263 us)
 169837  175296     HASH JOIN                       (cr=24036 r=21944 w=0 time=4625555 us)
    625     627      TABLE ACCESS BY INDEX ROWID T3 (cr=632 r=0 w=0 time=3727 us)
    625     627       INDEX RANGE SCAN T3X          (cr=7 r=0 w=0 time=583 us)
  62500   62500      TABLE ACCESS FULL T2           (cr=23404 r=21944 w=0 time=4430244 us)
```

It is evident that the optimizer's cardinality estimates and the actual row counts are extremely close until it comes to the cardinality estimate for table t1. Without the histogram on the predicate column d3, the optimizer assumes that the two column values, 0 and 1, are uniformly distributed and that therefore the cardinality estimate for t1 is 0.5 (the selectivity of the predicate d3=0) times 62500 (the cardinality, i.e. row count of t1) = 31250. But, of course that is wrong because of the skew and only 2 rows qualify for d3=0.

Following the "tuning by cardinality feedback method, we need to investigate the discrepancy in the estimated cardinality vs. actual row count by examining the predicate(s) on table t1 for violations of the optimizer's assumptions (see [1]). Here there is only one predicate, "t1.d3 = 0" which will lead us straight to the violation of the "uniform distribution" assumption and its remedy, a histogram.

HISTOGRAMS ON ALL INDEXED COLUMNS

Ok, so we have seen that a histogram on a non-indexed column can be beneficial for the performance of a SQL. But what about the reverse? Can a histogram on a column, indexed or not, be detrimental to the performance of a SQL?

I have always been convinced that the answer to that is "Yes" and that I could create a testcase to show it. Somehow I never got around to it. But "Good Things Come to Those Who Wait." Just such an example fell into my lap one day at a client site and in a far more dramatic manner than I would ever have dared to concoct: The performance of a 90 second job had suddenly dropped and it was canceled after several hours[α]. The SQL itself[β] is not really important and not very meaningful without the actual data, which of course I can not divulge because of client confidentiality. However, for reference, it is included in appendix B.

The analysis of the cause for the performance degradation was made rather easy by the fact that the job still performed in the customary 90 seconds in an older environment, but not in production nor in a very recently cloned test database. The first thing I always do when confronted with a poor performing SQL is to run a script that shows me the indexes on the tables with their columns and statistics. From there it was immediately obvious, if you know what to look for in any case, that the databases with the poor performance had histograms on the indexed columns while the older environment, with the still good performance, did not have histograms. What remained was to verify that this was really the cause by dropping the histograms and confirm that the job performance returned to the accustomed level. It did[δ].

This is the tkprof "performance profile" of the sql without the histograms "on all indexed columns":

[α] At the rate it was going I estimated that it would have taken over two days to finish the daily (!) job which had previously taken just 90 seconds.

[β] Mildly changed to obscure the identity of the client. It should still be obvious for those in the know that it is from a Peoplesoft job.

[δ] To be exact, because I could only work with the still "good" database, the proof consisted of collecting histograms "on all indexed columns" and see the performance drop – and return to normal again after re-analyzing with "for all columns size 1".

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	0.01	0.00	0	68	0	2
total	4	0.02	0.01	0	68	0	2

And this is it with the histograms:

call	count	cpu	elapsed	disk	query	current	rows
Parse	1	0.01	0.00	0	0	0	0
Execute	1	0.00	0.00	0	0	0	0
Fetch	2	80.11	78.23	0	770842	0	2
total	4	80.12	78.24	0	770842	0	2

What aggravated the situation was that this same sql was executed for every setid-emplid combination in the run control set. With 0.01 seconds per sql, 9,000 combinations could be processed in 90 seconds. At 78.24 seconds per sql those same 9,000 combinations would take 8 days and 3:36 hours.
What had happened was that the DBAs had decided to change the statistics gathering routine from the default 'for all columns size 1' to 'for all indexed columns size skewonly'. The result was devastating, at least for this sql. Needless to say that the "for all indexed columns size skewonly" statistics gathering option was quickly and unceremoniously dumped.

To be fair, this disaster was not the "fault" of the CBO, but due to the blunder of its sidekick, the statistics gathering procedure and particularly its "size skewonly" option (see below).

Histograms and "density"

Before looking at some of the options for gathering histograms with the DBMS_STATS package, let us clarify first how gathering histograms affects the calculation of one of the column statistics – density.

There are three different calculation for the density statistic:

- Without a histogram density = 1/NDV[?]
- With a height-balanced histogram density = Σ cnt^2 / (num_rows˜ * Σ cnt)[ε]
- With a frequency histogram density =1/(2 * num_rows˜)

The formula for the density of a height-balanced histograms requires some explanation. In words it would spell out as "the sum of the squared frequencies of all non-popular values divided by the sum of the frequencies of all non-popular values times the count of rows with not null values of the histogram column". I am not confident that that explains it much better. It is best shown with a series of examples[4].

Histograms and "size skewonly"

SKEWONLY - Oracle determines the columns to collect histograms based on the data distribution of the columns. [4]

The first chapter showed that the option "for all indexed columns size xx" is not good enough because there can be non-indexed columns that benefit from having a histogram. That could be overcome by

[?] Number of Distinct Values = DBA_TAB_COLUMN.NUM_DISTINCT

[ε] num_rows˜ means "number of rows with not null values of the histogram column

[4] They would take up too much room in this paper and require too much verbiage to explain them. They do form a good part of the presentation.

collecting histograms on all columns. However, as the second chapter showed, an inappropriate histogram on a column, indexed or not, can be detrimental, even disastrous.
So, what is a DBA to do? Oracle9*i* introduced an option which seems to be just what the doctor ordered: "size skewonly". According to the documentation[5]

SKEWONLY - Oracle determines the columns to collect histograms based on the data distribution of the columns.

That is exactly what we want. Gather statistics with method_opt=>'for all columns size skewonly' and let Oracle figure out what columns have a skewed data distribution and should have a histogram so that the optimizer has better statistics for its cardinality estimates.

Ever heard the warning "If it sounds too good to be true – it probably is" ? Well, as I am going to show, "skewonly" unfortunately falls into that category, at least for the time being[6].

Case 1 – column with perfectly uniform data distribution

```
create table test (n1 number not null);
insert into test (n1) select mod(rownum,5)+1 from dba_objects where rownum <= 100;
```

Column n1 has 5 distinct values, 1..5, each occurring exactly 20 times. Data distribution cannot get more uniform than that. Next we gather normal statistics.

```
exec DBMS_STATS.GATHER_TABLE_STATS (null, 'test', method_opt => 'FOR ALL COLUMNS SIZE 1');
```

These statistics describe column n1 perfectly since the data distribution is in complete harmony with the optimizer's assumption of uniform data distribution:

table	column	NDV	density	nulls	lo	hi
TEST	N1	5	2.0000E-01	0	1	5

The CBO's cardinality estimate for an equality predicate yields an accurate row count prediction:

card = base_cardinality * selectivity = num_rows * density = $100 * 2.0e^{-1} = 20$

The 1 bucket "histogram" has just two entries for the min (lo) and max (hi) values of the column:

table	column	EP	value
TEST	N1	0	1
TEST	N1	1	5

Now let's see what changes when we change the statistics collection to use "size skewonly". First we delete the current statistics to make sure there is no interference or residue from the prior statistics:

```
exec DBMS_STATS.DELETE_TABLE_STATS (null, 'test');
exec DBMS_STATS.GATHER_TABLE_STATS (null, 'test', method_opt => 'FOR COLUMNS N1 SIZE SKEWONLY');
```

table	column	NDV	density	nulls	lo	hi
TEST	N1	5	5.0000E-03	0	1	5

Most of the statistics have not changed, but one has – the density. And now the CBO's cardinality estimate for an equality predicate is wrong:

card = base_cardinality * selectivity = num_rows * density = $100 * 5.0e^{-3} = 0.5$ rounded to 1

---

[5] Oracle9*i* Rel 2 "Supplied PL/SQL Packages and Types Reference" Part No. A96612-01

[6] which as of this writing includes Oracle 10.2.0.1.

Querying the histogram table confirms that there really is a histogram confirming the perfectly uniform data distribution.

table	column	EP	value
TEST	N1	20	1
TEST	N1	40	2
TEST	N1	60	3
TEST	N1	80	4
TEST	N1	100	5

Case 2 – single value column

```
create table test (n1 number not null);
insert into test (n1) select 1 from dba_objects where rownum <= 100;
```

Column n1 has only 1 value: 1. I don't know whether I should classify that as extremely skewed, or as uniform? In any case, the standard statistics again are adequate to describe the data distribution perfectly:

```
exec DBMS_STATS.GATHER_TABLE_STATS (null, 'test', method_opt => 'FOR ALL COLUMNS SIZE 1');
```

table	column	NDV	density	nulls	lo	hi
TEST	N1	1	1.0000E-00	0	1	1

Again, the CBO's cardinality estimate for an equality predicate yields an accurate row count prediction:

card = base_cardinality * selectivity = num_rows * density = 100 * $1.0e^{-0}$ = 100

And, as before, the 1 bucket "histogram" has the two entries for the min (lo) and max (hi) value(s):

table	column	EP	value
TEST	N1	0	1
TEST	N1	1	1

What, if anything, changes when we change the statistics collection to use "size skewonly":

```
exec DBMS_STATS.DELETE_TABLE_STATS (null, 'test');
exec DBMS_STATS.GATHER_TABLE_STATS (null, 'test', method_opt => 'FOR COLUMNS N1 SIZE SKEWONLY');
```

table	column	NDV	density	nulls	lo	hi
TEST	N1	1	5.0000E-03	0	1	1

Once more, the only statistics value that changed is the density – to the same value as in case 1! And therefore CBO's cardinality estimate becomes

card = base_cardinality * selectivity = num_rows * density = 100 * $5.0e^{-3}$ = 0.5 rounded to 1

That is an error of one order of magnitude. And because of the way the density of a frequency histogram is calculated (ref page 7), the error can easily be several orders of magnitude, depending solely on the number of rows in the table with non null values of that column. This error in cardinality estimate can then lead to wrong plan choices[3, 5]. For example, based on the, erroneous as we know, cardinality estimate of 1, the CBO may use this table as the driving table in an NL join. But instead of the estimated one time, the inner rowsource will be access for every row in the table!. If that inner row source is itself the result of a complex and costly subplan, the result of the repetitive execution can be devastating. Something like that was responsible for the dramatic performance degradation of the example on page 6.

The histogram shows an interesting content: only one row:

table	column	EP	value
TEST	N1	100	1

HISTOGRAMS AND "SIZE AUTO"

AUTO - Oracle determines the columns to collect histograms based on data distribution and the workload of the columns. [4]

Using "size auto" is identical in its outcome to "size skewonly". The difference is that the list of columns for which to collect a histogram is compared to the columns which have been used in predicates. Unless a column has been used in a predicate before, gather_table_stats with method_opt '… size auto' will not gather a histogram. Otherwise, the results of gathering with "size auto" are the same as gathering with "size skewonly".

HISTOGRAMS AND "SIZE REPEAT"

REPEAT - Collects histograms only on the columns that already have histograms.[4]

The problem I am going to discuss is not so much one of "size repeat" but of histograms and gathering with less than a full compute. It therefore applies equally to gathering histograms 'for all columns size skewonly' - or auto. I discovered it when a SQL I had tuned by gathering a histogram on one of the predicate columns reverted back to the "bad" plan after the weekly statistics refresh, which tried to be sensible and used 'for all columns size repeat'. It turned out that the statistics were gathered with estimate, not compute while the histogram I had gathered had been done with compute. The setup of a testcase to illustrate it is in appendix C

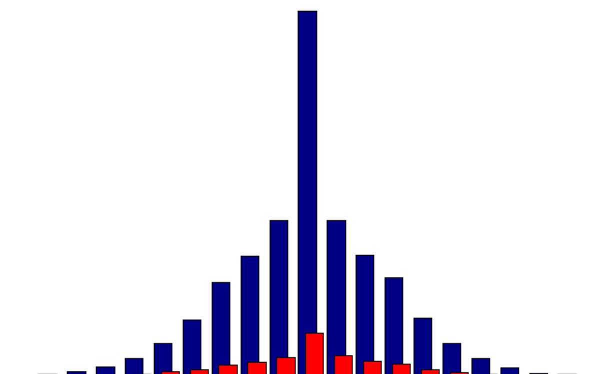
1. Gather table statistics using estimate_percent=>dbms_stats.auto_sample_size
2. Gather a histogram on columns n1, n3, n3 with estimate_percent=>100, method_opt=>'size skewonly'
3. Gather table statistics using estimate_percent=>dbms_stats.auto_sample_size, method_opt=>'size repeat'

Comparing the histogram for n3, the column with the extremely skewed distribution – only 8 out of 40000 rows have a different value – shows the problem:

after gathering with estimate_percent=>100:				after gathering with auto_sample_size			
Table	column	EP	value	Table	column	EP	value
TEST	N3	8	0	TEST	N3	2	0
TEST	N3	40000	1	TEST	N3	4973	1

Given that auto_sample_size sampled approximately 5000 of the 40,000 rows, it did capture the relative distribution of the n3 values rather well, but the absolute values are way off. They should have been pro-rated to the full table size – the number of rows was actually estimated rather accurately. The value of 2 for "n3=0" instead of 8 is not so much in danger of causing plan problems, but 4,971 instead of 39,992 gets into the danger zone for possibly using an index..

The data values for columns n1 and n2 were generated to follow a normal distribution and a graphic representation of the histograms after step 2 (blue) and step 3 (red) shows again the "flattened" histogram with the real danger that even second or third most predicate values will get accessed using an index, which may turn out to be totally inappropriate



Aside from the failure of dbms_stats not pro-rating the histogram obtained through sampling, it does not make sense to gather hostograms with sampling in the

first place. After all, histograms are only gathered – or ought to – for columns with skewed data distribution. Sampling is liable to miss infrequently occurring values and therefore skew the resulting histogram.

APPENDIX A

SCRIPT TO DEMONSTRATE USEFULNESS OF A HISTOGRAM ON NON-INDEXED COLUMN

```
prompt example of benefit of histogram on non-indexed columns

prompt Create the tables

drop table t1;
create table t1 (
 pk1 number,
 pk2 number,
 fk1 number,
 fk2 number,
 d1 date,
 d2 number,
 d3 number,
 d4 varchar2(2000),
 primary key (pk1, pk2)
 );

create index t1x
 on t1(d2);

drop table t2;
create table t2 (
 pk1 number,
 pk2 number,
 fk1 number,
 fk2 number,
 d1 date,
 d2 number,
 d3 number,
 d4 varchar2(2000),
 primary key (pk1, pk2)
 );

create index t2x
 on t2(d2);

drop table t3;
create table t3 (
 pk1 number not null,
 pk2 number not null,
 fk1 number,
 fk2 number,
 d1 date,
 d2 number,
 d3 number,
 d4 varchar2(2000),
 primary key (pk1, pk2)
 );

create index t3x
 on t3(d2);

prompt Load the tables with data
begin
  dbms_random.seed(67108863);
  for i in 1..250 loop

    insert into t1
    select i, rownum j
      , mod(trunc(100000*dbms_random.value),10)+1
      , mod(trunc(100000*dbms_random.value),10)+1
      , trunc(sysdate)+trunc(100*dbms_random.value)
      , mod(trunc(100000*dbms_random.value),100)+1
      , decode(mod(trunc(100000*dbms_random.value), 65535),0,0,1)
      , dbms_random.string('A',trunc(abs(2000*dbms_random.value)))
      from dba_objects where rownum <= 250;

    insert into t2
    select i, rownum j
      , mod(trunc(100000*dbms_random.value),10)+1
      , mod(trunc(100000*dbms_random.value),10)+1
      , trunc(sysdate)+trunc(100*dbms_random.value)
      , mod(trunc(100000*dbms_random.value),100)+1
      , mod(trunc(100000*dbms_random.value),20)+1
      , dbms_random.string('A',trunc(abs(2000*dbms_random.value)))
      from dba_objects where rownum <= 250;
```

```
    insert into t3
    select i, rownum j
      , mod(trunc(100000*dbms_random.value),10)+1
      , mod(trunc(100000*dbms_random.value),10)+1
      , trunc(sysdate)+trunc(100*dbms_random.value)
      , mod(trunc(100000*dbms_random.value),100)+1
      , mod(trunc(100000*dbms_random.value),20)+1
      , dbms_random.string('A',trunc(abs(2000*dbms_random.value)))
      from dba_objects where rownum <= 250;

    commit;
  end loop;
end;
/

BEGIN
  DBMS_STATS.GATHER_TABLE_STATS(null,'t1');
  DBMS_STATS.GATHER_TABLE_STATS(null,'t2');
  DBMS_STATS.GATHER_TABLE_STATS(null,'t3');
END;
/
select d3, count(0) from t1 group by d3;
alter session set sql_trace true;

select /* 1 */ sum(t1.d2*t2.d3*t3.d3)
from t1, t2, t3
where t1.fk1 = t2.pk1
  and t3.pk1 = t2.fk1
  and t3.d2 = 35
  and t1.d3 = 0;

alter session set sql_trace false;
dbms_stats.gather_table_stats(user,'t1' method_opt=>'for columns size 75 d3')

alter session set sql_trace true;

select sum(t1.d2*t2.d3*t3.d3)
from t1, t2, t3
where t1.fk1 = t2.pk1
  and t3.pk1 = t2.fk1
  and t3.d2 = 35
  and t1.d3 = 0;

alter session set sql_trace false;
spool off
```

APPENDIX B

```
SELECT B1.SETID, B3.EMPLID , B3.EMPL_RCD, B3.EFFSEQ, B3.EFFDT b3_effdt,
B3.CURRENCY_CD,
...
 From PS_GRP_DTL B1
 , PS_JOB B3
 , PS_JOBCODE_TBL B4
WHERE B1.SETID = rtrim(:b1,' ')
  AND B3.EMPLID = rtrim(:b2, ' ')
  AND B1.SETID = B4.SETID
  AND B1.BUSINESS_UNIT IN (B3.BUSINESS_UNIT, ' ')
  AND B3.BUSINESS_UNIT IN ('BU001', 'BU007', 'BU017', 'BU018', 'BU502', 'BU101', ' ')
  AND B1.DEPTID IN (B3.DEPTID, ' ')
  AND B1.JOB_FAMILY IN (B4.JOB_FAMILY, ' ')
  AND B1.LOCATION IN (B3.LOCATION, ' ')
  AND B3.JOBCODE = B4.JOBCODE
  AND B4.EFF_STATUS = 'A'
  AND B1.EFFDT = (SELECT MAX(A1.EFFDT) FROM PS_GRP_DTL A1
    WHERE B1.SETID = A1.SETID AND B1.JOB_FAMILY = A1.JOB_FAMILY
    AND B1.LOCATION = A1.LOCATION AND B1.DEPTID = A1.DEPTID
    AND A1.EFFDT <= to_date(:b3, 'yyyy-mm-dd'))
  AND B3.EFFDT = (SELECT MAX(A2.EFFDT) FROM PS_JOB A2
    WHERE B3.EMPLID = A2.EMPLID AND B3.EMPL_RCD = A2.EMPL_RCD
    AND A2.EFFDT <= to_date(:b3, 'yyyy-mm-dd'))
  AND B3.EFFSEQ = (SELECT MAX(A3.EFFSEQ) FROM PS_JOB A3
    WHERE B3.EMPLID = A3.EMPLID
      AND B3.EMPL_RCD = A3.EMPL_RCD AND B3.EFFDT = A3.EFFDT)
  AND B4.EFFDT = (SELECT MAX(A4.EFFDT) FROM PS_JOBCODE_TBL A4
    WHERE B4.JOBCODE = A4.JOBCODE AND A4.EFFDT <= to_date(:b3, 'yyyy-mm-dd'))
ORDER BY B1.SETID desc, B3.EMPLID desc, B1.BUSINESS_UNIT desc
    , B1.DEPTID desc, B1.JOB_FAMILY desc, B1.LOCATION desc
```

APPENDIX C

```
create table test( n1 number not null, n2 number not null, n3 number not null,
filler varchar2(4000));

exec dbms_random.seed(134217727);
insert into test
select 100+trunc(60*dbms_random.normal),
100+trunc(20*dbms_random.normal),
decode(mod(trunc(10000*dbms_random.normal),16383),0,0,1),
dbms_random.string('a',2000)
from dba_objects
where rownum <= 5000;

insert into test select * from test;
insert into test select * from test;
insert into test select * from test;
```

BIBLIOGRAPHY

Jonathan Lewis, *Cost-based Oracle: Fundamentals*. 2006: Apress. ISBN 1590596366.

Metalink Note 114671.1 *Gathering Statistics for the Cost Based Optimizer*

Metalink Note 44961.1 *Gathering statistics frequency and strategy guidelines*

REFERENCES

1. Wolfgang Breitling A Look Under the Hood of CBO: The 10053 Event
2. Wolfgang Breitling Using DBMS_STATS in Access Path Optimization
3. Wolfgang Breitling Tuning by Cardinality Feedback - Method and Examples
4. *PL/SQL Packages and Types Reference 10g Release 2 (10.2)*. 2005, B14258-01.
5. Andrew Holdsworth, et al. *A Practical Approach to Optimizer Statistics in 10g*. in *Oracle Open World*. September 17-22, 2005. San Francisco.

---

[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.