JOINS, SKEW AND HISTOGRAMS

Wolfgang Breitling, Centrex Consulting Corporation

In my paper "Fallacies of the Cost Based Optimizer" [1]I touched on one example (example 3 on page 11) where the CBO underestimates the cardinality of a join due to skew in the data distribution of the join columns. This paper explores if and how the presence of histograms on join columns can help the CBO improve its cardinality estimates. To limit the complexity this paper limits the joins studied to equijoins on columns of type number and focuses on the effect of histograms on different data distributions of the join columns. The tests were conducted on Oracle 9i (9.2.0.5, 9.2.0.7 and 9.2.0.8), 10gR1 (10.1.0.4 and 10.1.0.5) and 10gR2 (10.2.0.1, 10.2.0.2 and 10.2.0.3). Except for 10.1.0.5 and 10.2.0.1 the results were the same. I'll cover the reason for the different results for these two releases in chapter II.4.

I - JOINS AND CARDINALITY ESTIMATES

Let's briefly revisit the basic join cardinality estimate by the CBO[2] and how it holds up to various data distributions of the join columns.

join cardinality[1] = $card_A$ * $card_B$ * join selectivity

where join selectivity[2] = $1/\max(ndv_A, ndv_B)$

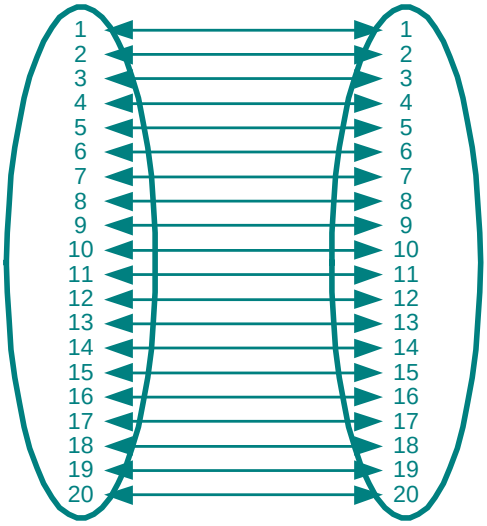
The underlying assumption for this formula is the "principle of inclusion": each value of the smaller domain has a match in the larger domain. This is obviously true for joins between foreign keys and primary keys.

The simplest case to illustrate that is one where there is a one-to-one match of the join values (script I page 17):

```
PLAN_TABLE_OUTPUT
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    20 |
|*  1 |  HASH JOIN         |      |    20 |
|   2 |   TABLE ACCESS FULL| J1   |    20 |
|   3 |   TABLE ACCESS FULL| J2   |    20 |
-----------------------------------------

C1    C2
----- -----
A.1   B.1
A.2   B.2
A.3   B.3
…
A.18  B.18
A.19  B.19
A.20  B.20

20 rows selected.
```



In this case the CBO's cardinality estimate is on the mark. That remains to be true if we extend the test to one-to-many (script IIa) or the case that one domain is a true subset of the other (script IIIa) as well as many-to-many (script IV) as long as the tenet of the principle of inclusion holds **and** the distribution of the values is uniform. We'll cover violations of both of these conditions next.

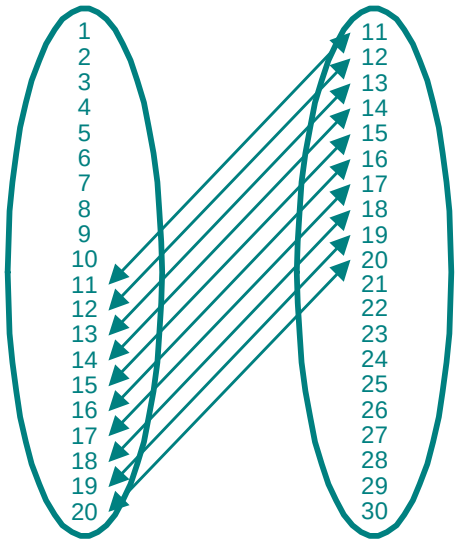
[1] the cardinalities exclude rows where the join columns are NULL
[2] ndv = num_distinct; number of distinct values

Once the principle of inclusion is violated the optimizer's join cardinality estimates begin to differ from the actual join cardinality. See the results of script V

```
PLAN_TABLE_OUTPUT
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    20 |
|*  1 |  HASH JOIN         |      |    20 |
|   2 |   TABLE ACCESS FULL| J1   |    20 |
|   3 |   TABLE ACCESS FULL| J2   |    20 |
-----------------------------------------

C1    C1
----- -----
A.11  B.11
A.12  B.12
A.13  B.13
A.14  B.14
A.15  B.15
A.16  B.16
A.17  B.17
A.18  B.18
A.19  B.19
A.20  B.20

10 rows selected.
```



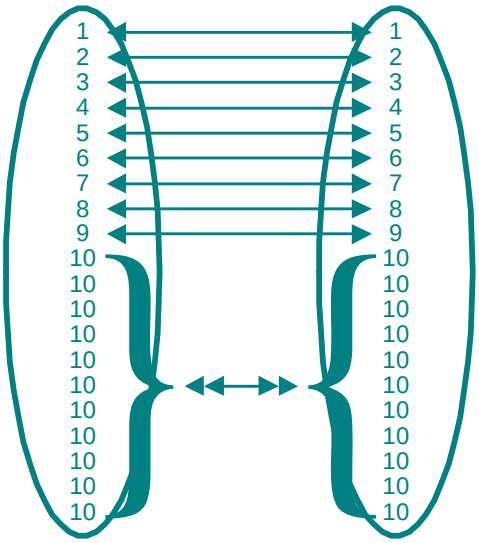
Since there are values in either table without a match in the other table the principle of inclusion is violated and the CBO overestimates the join cardinality.

Another case of cardinality miscalculation is shown with script VI. Here the optimizer underestimates the join cardinality even though the principle of inclusion is not violated. This is the same scenario as the aforementioned example 3 in [1]

```
PLAN_TABLE_OUTPUT
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    40 |
|*  1 |  HASH JOIN         |      |    40 |
|   2 |   TABLE ACCESS FULL| J1   |    20 |
|   3 |   TABLE ACCESS FULL| J2   |    20 |
-----------------------------------------


C1    C1
----- -----
A.1   B.1
A.2   B.2
A.3   B.3
A.4   B.4
A.5   B.5
A.6   B.6
…
A.6   B.6
A.6   B.6
A.6   B.6

130 rows selected.
```



Now it is the skew in the value distribution of the join columns which leads to an underestimation of the join cardinality. That is the difference between this many-to-many case and the one of script IV where the value distributions of the join columns are uniform.

In the following chapters we will investigate if and under what circumstances histograms on the join columns can help the optimizer improve the cardinality estimate of the join. As [3] and [4] point out, accurate estimates by the optimizer are a prerequisite for good execution plans.

II – HISTOGRAMS AND JOIN CARDINALITY ESTIMATES

We have seen two circumstances where the basic join cardinality formula fails to yield the correct join cardinality estimate – violation of the principle of inclusion and skew in the value distribution of the join columns. We will now explore if and how the presence of histograms improves the join cardinality estimate. We will also probe if there are negative side effects from the existence of histograms.

First we will use the same scripts as in chapter one, but will focus on the two cases where the CBO estimated the join cardinality wrong and gather histograms – 'for all columns size 254' – instead of just the base statistics. We will also check if and where " size {skewonly | auto} " make a difference.

II.1 – PRINCIPLE OF INCLUSION MAINTAINED

First the control check on the simple cases where the join cardinality estimate without histograms is correct.

CASE 1 – ONE-TO-ONE (SCRIPT I) [1]

With histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    20 |
|*  1 |  HASH JOIN         |      |    20 |
-----------------------------------------
```

error [2] 0%

Without histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    20 |
|*  1 |  HASH JOIN         |      |    20 |
-----------------------------------------
```

error 0%

actual: **20**

No difference in the cardinality estimate, with or without a histogram. But then, this is a very simple case.

CASE 2 – ONE-TO-MANY (SCRIPT IIa)

With histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    21 |
|*  1 |  HASH JOIN         |      |    21 |
-----------------------------------------
```

error 45%

Without histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    40 |
|*  1 |  HASH JOIN         |      |    40 |
-----------------------------------------
```

error 0%

actual: **40**

Note that with a histogram the cardinality estimate is half the actual – plus one – while without a histogram the estimate matches the actual. The reason for that is explained in Alberto Dell'Era's formula (see page 14) for cardinality estimates with histograms ("mystery of halving"). If the histograms were collected using size skewonly or auto then the cardinality estimate would be the same as without histograms. This is because skewonly, and by extension auto, do not collect a histogram on the parent column where each value occurs only once, i.e. the column is unique[3], and the CBO reverts to the base formula if only one predicate column has a histogram

However, this is a perfect example that one has to be careful not to generalize the observations from a single testcase too quickly. There are several simplifications embedded in this case:

- The number of distinct values of both join columns is so small that both histograms are frequency histograms
- Every parent row has child rows
- Each parent row has the same number of child rows.

---

[1] For clarity and to save space, the dbms_xplan.display output has been truncated after the join rowsource

[2] error is calculated as abs(estimate – actual) / actual

[3] Of course, this too is an oversimplification. There are circumstances where skewonly collects a histogram on a unique column, one being a unique varchar2 column where there is some distinction, non-uniqueness, in the first 6 characters. For another see page 5. This is the danger of using skewonly and auto – they can change their "behaviour" based on changes of the underlying data.

Despite these simplifications, this is such an important and frequently occurring scenario– think lookup tables – that I want to spend some time to explore this further, keeping only the first constraint: the number of distinct values is small enough that we end up with frequency histograms. At the same time I'll point out important differences between " size n ", " size skewonly " and " size auto "

We create a lookup table, J1, with 150 distinct unique values – so that even 9i can still create a frequency histogram. We even create a primary key constraint for it. Then we create a "fact" table, J2, with a skewed value distribution (script VIII). Note the gap of values in the middle

ID	COUNT(0)
1	9229
2	577
3	114
4	37
5	15
6	8
7	4
8	3
9	2
10	1
141	1
142	1
143	1
144	1
145	1
146	1
147	1
148	1
149	1
150	1

Without histograms — actual

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

10000

With " size 254 " histograms — actual

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |  4995 |
|*  1 |  HASH JOIN         |      |  4995 |
-----------------------------------------
```

10000

With " size skewonly " histograms — actual

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

10000

With " size auto " histograms — actual

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

10000

Why are the cardinality estimates with the "skewonly" and "auto" histograms better than the "254" histograms? The simple reason is that there are no histogram**s** (plural!). There is only one histogram – on the "fact" table. Skewonly and auto do not gather histograms on columns with unique values, where each value occurs only once. And unless **both** join columns have a histogram the optimizer reverts to the standard cardinality estimate.

Next we delete the rows from the lookup table whose values do not occur in the "fact" table, values 11 to 140. There are (at least) two ways to do that:

```
delete from j1 where id not in (select distinct id from j2)
```

```
delete from j1 where id between 11 and 140
```

Using delete SQL #1 and re-gathering the histograms results in the following cardinality estimates.

Without histograms

```
|  Id | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

With " size 254 " histograms — actual

```
|  Id | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |  4996 |
|*  1 |  HASH JOIN         |      |  4996 |
-----------------------------------------
```

10000

With " size skewonly " histograms

```
|  Id | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |  4996 |
|*  1 |  HASH JOIN         |      |  4996 |
-----------------------------------------
```

With " size auto " histograms — actual

```
|  Id | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

10000

Now what is going on? I had said that skewonly would not create a histogram for unique columns and that has not changed. J1.id is still unique. However, instead of one consecutive range of values it now has two narrow ranges separated by a rather large gap. And for that distribution skewonly **does** create a histogram[1]. And rightfully so. The information the histogram provides can be vital for range predicates. But why not "size auto"?

The answer lies in what happens if we use delete SQL #2:

Without histograms

```
| Id | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      | 10000 |
|* 1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

With " size 254 " histograms

```
| Id | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |  4996 |
|* 1 |  HASH JOIN         |      |  4996 |
-----------------------------------------
```

actual **10000**

With " size skewonly " histograms

```
| Id | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |  4996 |
|* 1 |  HASH JOIN         |      |  4996 |
-----------------------------------------
```

With " size auto " histograms

```
| Id | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |  4996 |
|* 1 |  HASH JOIN         |      |  4996 |
-----------------------------------------
```

actual **10000**

Now "size auto" **too** created a histogram. So what is different. Obviously it must be the delete SQL since that is the only thing we changed. SQL #2 uses a range predicate to identify the rows to be deleted while SQL #1 did not. That predicate usage is recorded in sys.col_usage$

```
TABLE    COLUMN     EQUALITY   EQUIJOIN NONEQUIJOIN      RANGE       LIKE       NULL
-------- -------- ---------- ---------- ----------- ---------- ---------- ----------
J1       ID                0          2           0          1          0          0
```

As mentioned above, while a histogram on this kind of data distribution is of little or no use for equi and equi-join predicates – because every column value occurs equally often: once – it can be vital for a better cardinality estimate of range predicates. Gather_table_stats(…"size auto") consults sys.col_usage$, finds that the ID column has been used in a range predicate and uses skewonly to gather the data for a histogram. We had seen previously that the value distribution is such that skewonly does actually gather a histogram. With the first SQL there was no range predicate and the gather_table_stats(… "size auto") procedure skipped histogram generation entirely[2].

CASE 3 – ONE-TO-MANY

After this excursion into the effects of different ways of gathering histograms we return to checking what, if any, effect histograms have on the cardinality estimates of the cases where the standard formula was sufficient. Unless stated otherwise histograms were gathered using size 254.
One-to-many with more than 254 distinct parents and children (500 each, script IIb), i.e. with height-balanced histograms .

With histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |  1001 |
|* 1 |  HASH JOIN         |      |  1001 |
-----------------------------------------
     error                          0.1%
```

Without histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |  1000 |
|* 1 |  HASH JOIN         |      |  1000 |
-----------------------------------------
     error                            0%
```

actual **1000**

Both join columns now have height-balanced histograms and the halving is gone. Except for the (negligible) additional one the estimate is again the same as without histograms and matches the actual count.

[1] Credit goes to Christian Antognini who pointed that out in response to an Oracle-L post of mine (http://www.freelists.org/archives/oracle-l/12-2006/msg00395.html)

[2] It is likely a bit more complicated than that. "Size auto" would not "know" until after gathering the data that a histogram would be useful or not based on past usage recorded in sys.col_usage$

CASE 4 – ONE-TO-MANY

with more than 254 distinct parents but fewer than 254 distinct children; i.e. no longer has every parent row child rows. The testcase has 500 parent rows but only the first 100 parents have 100 children each (script IIc).

With histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10101 |
|*  1 |  HASH JOIN         |      | 10101 |
|*  1 |  HASH JOIN         |      | 10001 |
-----------------------------------------
```

error 1%

Without histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

error 0%

actual: 10000

With a height-balanced histogram on the parent join predicate and a frequency histogram on the child predicate there is again a discrepancy in the join cardinality estimate There is an additional component which corresponds to the cardinality of the children – 100 each in this case. This is due to the "special cardinality" in the formula (page 14). What makes matters even worse, or at least more confusing, is the fact that it changes as you go through Oracle upgrades. The above is true for Oracle 9i[1] and early releases of 10.1. When you upgrade to 10.1.0.5, however, things change. The cardinality estimate then is shown in italics above. Though relatively small, it has the potential to change access plans, especially when you get into higher child cardinality numbers. Then, when you upgrade to 10gR2, more precisely 10.2.0.2 or higher, the join cardinality changes once more, back to what it was in 9i.

This testcase is also ideal to demonstrate a problem with histograms in different Oracle releases that I have already alluded to. If we change the parameters such that 200 parents have 50 children each – same total number of child rows but num_distinct of the join column in the child table is now 200 instead of 100 (script IId):

With histograms

```
| Id  | Operation              | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT       |      |       |
|*  1 |  HASH JOIN (9.2.x)     |      |  6624 |
|*  1 |  HASH JOIN (10.1.0.5)  |      | 10001 |
|*  1 |  HASH JOIN (10.2.0.2+) |      | 10051|
-----------------------------------------
```

error 1%

Without histograms

```
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
|*  1 |  HASH JOIN         |      | 10000 |
-----------------------------------------
```

error 0%

actual: 10000

This discrepancy results from the fact that 9i dbms_stats fails to gather a frequency histogram as soon as the number of buckets reaches or exceeds the number of distinct values of the column. It depends on the exact data distribution but size typically has to be 35% higher than num_distinct before dbms_stats switches to a frequency histogram, which in this case would be ~ 270. Since we requested "only" 254 buckets dbms_stats gathered a height-balanced histogram in 9i and frequency histograms in 10g. And then, of course, we have the difference between 10.1.0.5 and 10.2.0.2, same reason as above.

CASE 5 – ONE-TO-MANY (SCRIPT IV B)

This ought to be the case that most resembles real world scenarios: more than 254 rows in the parent table, most, but maybe not all, parent rows have child rows and different parent rows have different numbers of child rows.

[1] tested on 9.2.0.5, 9.2.0.7, 9.2.0.8 and 10.1.0.4 except that 9.2.0.5 and 10.1.0.4 do not have the additional 1

With histograms[2] | Without histograms | actual

```
| Id  | Operation          | Name |  Rows |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 86857 |
|*  1 |  HASH JOIN         |      | 86857 |
-----------------------------------------
```

```
| Id  | Operation          | Name |   Rows |
-----------------------------------------
|   0 | SELECT STATEMENT   |      | 100000 |
|*  1 |  HASH JOIN         |      | 100000 |
-----------------------------------------
```

error 13.1%	error 0%	100000

We knew going in that in all these cases the standard formula was sufficient for accurate join cardinalities. Gathering histograms produced some surprise join cardinality estimates. In general the differences are likely not severe enough to cause problems, but in specific circumstances they might. Caveat emptor.

II.2 – Principle of Inclusion violated

Next we examine closer the cases where the base join cardinality formula provided incorrect estimates

Case 1 – Partial Overlap of Join Column Ranges

First the partial overlap of join values (script V). Without histograms, the optimizer overestimated the join cardinality, the more so the smaller the overlap of values – until the ranges of the join columns become disjoint. The CBO recognizes that from the fact that the low value of one range is greater than the high value of the other range and estimates a join cardinality of 1[2]

With histograms | Without histograms | actual

```
| Id | Operation          | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |   20 |
|* 1 |  HASH JOIN         |      |   20 |
-----------------------------------------
```

```
| Id | Operation          | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |   20 |
|* 1 |  HASH JOIN         |      |   20 |
-----------------------------------------
```

error 100%	error 100%	10

For the partial range overlap, the presence of the (frequency) histogram does not make a difference to the join cardinality estimate.

Case 2 – No Overlap of Join Column Ranges

However, in the case of completely disjoint ranges the CBO surprisingly looses the recognition that the ranges are disjoint (see also [5] pages 279-280). This is because with histograms the optimizer uses a different set of rules to calculate the join cardinality than when histograms are present (see Appendix A).

With histograms | Without histograms | actual

```
| Id | Operation          | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |   20 |
|* 1 |  HASH JOIN         |      |   20 |
-----------------------------------------
```

```
| Id | Operation          | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT   |      |    1 |
|* 1 |  HASH JOIN         |      |    1 |
-----------------------------------------
```

error ∞[3]	error ∞	0

Using 'size skewonly' or 'size auto' may save the day if they "decide" not to create a histogram. But that is not something one can rely on and it can change from one analyze to the next based on changes to the column content.

[2] The exact join cardinality estimate depends on the Oracle version and release level. The differences are generally marginal but could in extreme cases be the cause for a plan change.

[2] the optimizer never estimates a cardinality of 0; maybe because the statistics could be out-of-date, but more likely to prevent the risk of dividing by 0.

[3] Since actual is zero, our error calculation breaks down.

II.3 – SKEW IN DATA DISTRIBUTION

As we have seen, if there is skew in the data distribution of the join column, the optimizer underestimates the join cardinality. The estimation error is "proportional" to the severity of the data skew. Therefore we'll explore next if histograms can help in this case using three scenarios

- "Symmetrical" Skew – frequently occurring values in one join column are also frequently occurring values in the other. In addition we'll look at two subcases. One where the frequency decreases with increasing column value and one where the frequency increases with increasing column values.
- "Anti-symmetrical" Skew – frequently occurring values in one join column are infrequently occurring in the other and vice-versa.
- "Random" Skew – the frequently occurring values in each column are independently randomly scattered over each range.

CASE 1 – "SYMMETRICAL" SKEW

Using script VI but collecting frequency histograms we get:

With histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |  123 |
|* 1 |  HASH JOIN           |      |  123 |
-----------------------------------------
      error                          5.4%
```

Without histograms

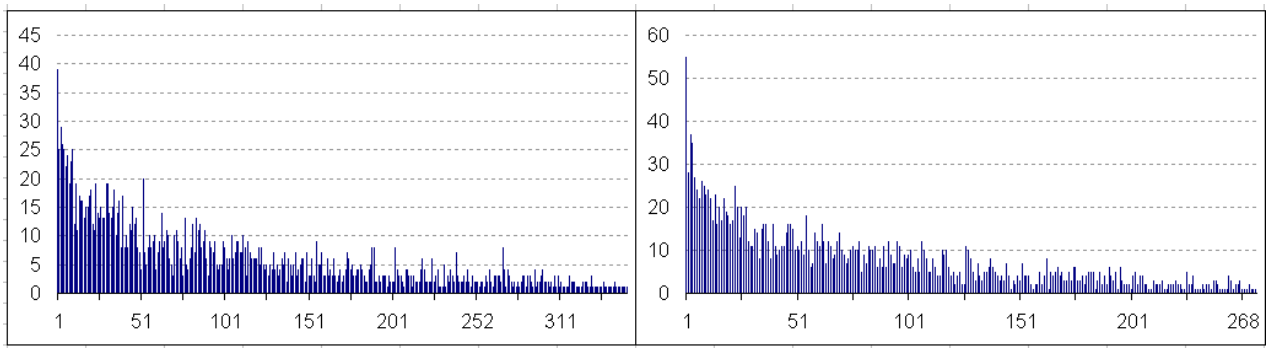
```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |   40 |
|* 1 |  HASH JOIN           |      |   40 |
-----------------------------------------
      error                         69.2%
```

actual: **130**

Clearly much closer to the actual join cardinality of 130.

Next we extend the example such that we can no longer collect a frequency histogram, i.e. num_distinct > 254, using dbms_random to create the skewed data distribution (script VII). By using different seeds we create similar but not identical data distributions. Due to the manner in which the data is generated, using dbms_random, the principle of inclusion may be violated, i.e. there may be values in each domain without a match in the other domain.

First with frequencies descending as the column values ascend:



With histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |21350 |
|* 1 |  HASH JOIN           |      |21350 |
-----------------------------------------
      error                          9.4%
```

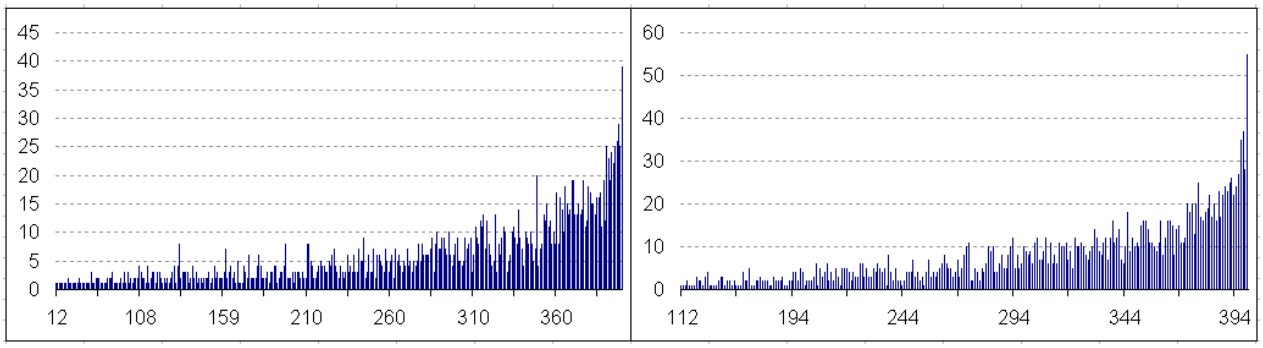
Without histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |11730 |
|* 1 |  HASH JOIN           |      |11730 |
-----------------------------------------
      error                         50.2%
```

actual: **23568**

Even with a height-balanced histogram, the estimate is better than without histograms.

Next with frequencies ascending as the column values ascend (script VIIb):



With histograms		Without histograms		actual
Id	Operation / Name / Rows	Id	Operation / Name / Rows	
0	SELECT STATEMENT / / 23824	0	SELECT STATEMENT / / 11730	
* 1	HASH JOIN / / **23824**	* 1	HASH JOIN / / **11730**	**23568**
	error 1.1%		error 50.2%	

This time the estimate is even better. But that is (probably) a statistical fluke. But why the emphasis on the two seemingly identical cases? Because calculations for the cardinality estimate in the presence of a histogram are not symmetrical. Case in point: just replacing the column values with their negative values (see also chapter II.4) – and re-analyze! – results in the following estimates:

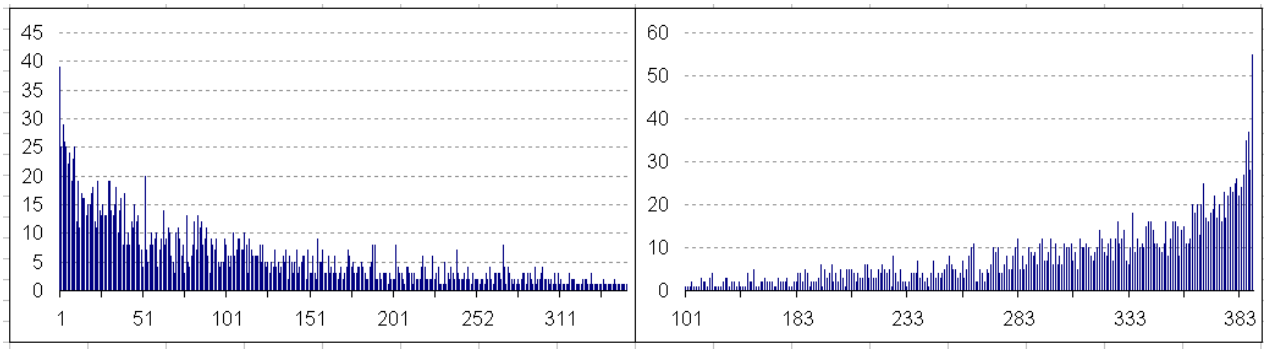
- for subcase 1 – now ascending frequencies: 18852 (instead of 21350)
- for subcase 2 – now descending frequencies: 21350 (instead of 23824)

In both cases the estimated cardinality is different after the inversion on zero. The fact that both estimates are now lower should not immediately be used to draw any conclusions without much more testing. However, given the root for the asymmetry in the cardinality formula "let CJH_PLUS_2 be the CJH plus the next two buckets following the LHV" (page 15) it is plausible that that is the general case.

Case 2 – "Non-Symmetrical" Skew in the Join Columns

An extreme case of non-symmetry is created by reversing the frequencies in the values: frequently occurring values in one join column occur infrequently in the other and vice versa (script VIIc). As the second domain is smaller several subcases come to mind:

- both domains have the same high but different low boundaries This the one examined below
- both domains have same low but different high boundaries
- smaller domain has a lower high and a higher low boundary
- both domains have same high and low boundary but the smaller domain is sparser.



With histograms

Id	Operation	Name	Rows
0	SELECT STATEMENT		10381
* 1	HASH JOIN		**10381**

error 282.8%

Without histograms

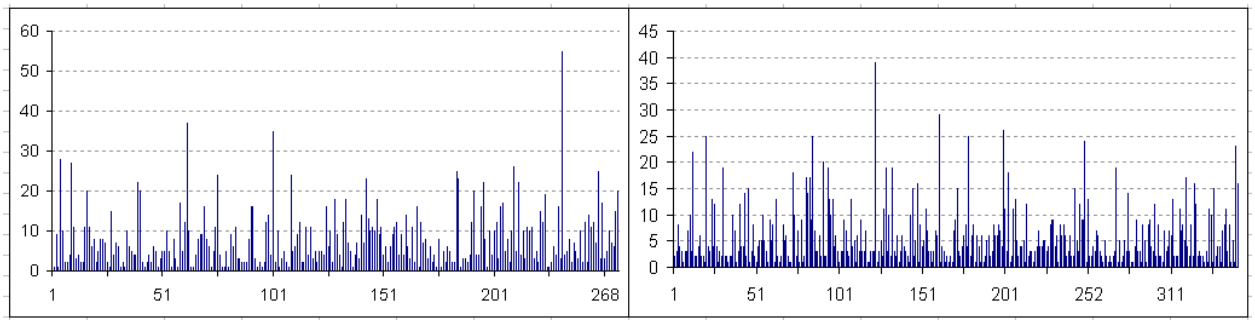
Id	Operation	Name	Rows
0	SELECT STATEMENT		11730
* 1	HASH JOIN		**11730**

error 332.5%

actual: **2712**

In this extreme case the estimate with histograms is hardly any better than the estimate without. Both grossly overestimate the actual cardinality.

Case 3 – "Random" Skew in the Join Columns



With histograms

Id	Operation	Name	Rows
0	SELECT STATEMENT		15270
* 1	HASH JOIN		**15270**

error 35%

Without histograms

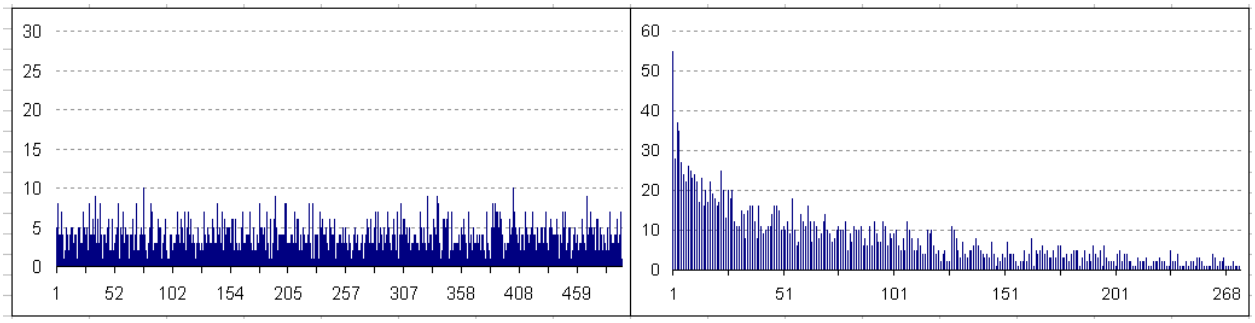
Id	Operation	Name	Rows
0	SELECT STATEMENT		11730
* 1	HASH JOIN		**11730**

error 3.5%

actual: **11328**

In this case the estimate error with histogram is even higher than without. Again, without much further testing no premature conclusion should be drawn as to the general applicability of this observation. The only conclusion that can be drawn is that the histograms do not improve the cardinality estimate in this case.

Case 4 – Skew in one join column, "uniform randomness" in the other



With histograms

Id	Operation	Name	Rows
0	SELECT STATEMENT		7966
* 1	HASH JOIN		**7966**

error 2.5%

Without histograms

Id	Operation	Name	Rows
0	SELECT STATEMENT		8163
* 1	HASH JOIN		**8163**

error .09%

actual: **8170**

Once more, the histograms exhibit no advantage over no histograms regarding the cardinality estimate. It is worth noting that size SKEWONLY does not create a histogram for the "uniformly random" column and therefore results in the same estimate as the no histogram case.

This, for example, is one case where the fact that I restricted the examples to number types could matter. The same testcase with varchar2 join column may "behave" differently.

CASE 5 – "RANDOM" SKEW IN BOTH JOIN COLUMNS

With this last case (script XI) I want to show a main disadvantage of histograms, especially height balanced histograms – they are very vulnerable to even small variations in the frequency of data values due to DML activity.

Both join columns are loaded using dbms_random.value. After collecting what turns out to be HB histograms, the join cardinality estimate is:

With histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |64516 |
|* 1 |  HASH JOIN           |      |64516 |
-----------------------------------------
```

Without histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |64516 |
|* 1 |  HASH JOIN           |      |64516 |
-----------------------------------------
```

actual: **65224**

Notice that there is no difference in the estimate despite the effort and expense of collecting the histograms. Next I update 1 value of the smaller domain table and re-analyze the table:

With histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |61703 |
|* 1 |  HASH JOIN           |      |61703 |
-----------------------------------------
```

Without histograms

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |64516 |
|* 1 |  HASH JOIN           |      |64516 |
-----------------------------------------
```

actual: **65223**

That small update caused the HB histogram to have a popular value and the cardinality estimate changes – further away from the actual value. In most cases this small discrepancy will not cause a change in access path, but it can. Remember Murphy's Law "Anything that can go wrong, will". I should note that once again, collecting the histograms using size SKEWONLY avoids some of this volatility. Skewonly does not collect histograms for marginal skew like this[1].

II.4 – ODDITIES (BUGS?)

INVERSION ABOUT ZERO

I already mentioned this on page 9. Alberto Dell'Era found this with a very compelling testcase (script IX). Unlike the other diagram pairs, here both sides have a – frequency – histogram. The difference is the inversion about zero.

Negative values

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      | 5001 |
|* 1 |  HASH JOIN           |      | 5001 |
-----------------------------------------
```

Positive values

```
| Id | Operation            | Name | Rows |
-----------------------------------------
|  0 | SELECT STATEMENT     |      |10001 |
|* 1 |  HASH JOIN           |      |10001 |
-----------------------------------------
```

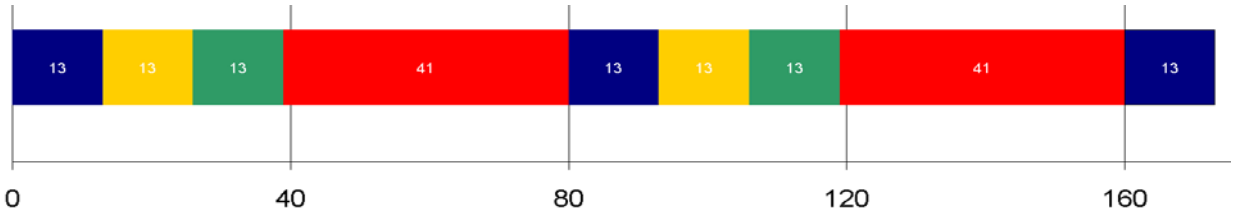
actual: **10000**

HB HISTOGRAM CHANGES IN 10.1.0.5 AND 10.2.0.1

Script X reveals a change in height-balanced histogram collection in Oracle 10.1.0.5 and 10.2.0.1. Conceptually, a HB histogram of size "n" – i.e. for n buckets – is created by sorting the column values and then recording every (num_rows/n)th value in addition to the first and last (low and high values). What appears to be happening in those two releases, however, is that in addition to the first and last values every (num_rows/n+1)th value of the sorted list is recorded. The script is carefully designed such that every 4th value (4,8,12,…) ends exactly at a bucket boundary and such that each 4th value occurs (at least) bucket-

[1] Not verbatim correct. Skewonly **does** collect the histogram but then after analysis throws it away and especially does not store it in the dictionary.

size+1 times (= num_rows / num_buckets +1 = 10160 / 254 +1 = 41). That is the minimum necessary for a value to appear at the end of two consecutive endpoints and thus be regarded a popular value. But since 10.1.0.5 records the first value in each bucket instead of the last, it misses all of the formerly popular values. Of course, one can just as easily construct a distribution where 10.1.0.5 detects popular values that weren't there in 9.2. I should mention that "size skewonly" adds an even more hideous twist to this situation: it does not collect a histogram for this distribution in 9.2[1] but does in 10.1.0.5 – even though the latter has no popular value.



SUMMARY

You never know what you will find when you start digging. I don't know or remember what I expected when I started to explore the effect of histograms on join cardinality estimates but whatever it was, some of the results surprised me. In particular, I gained a new respect for the SKEWONLY - and by extension AUTO - size option of dbms_stats.gather_table_stats.

To summarize the findings:

- There appears to be no benefit from having histograms on join columns as long as the join is between tables in a foreign key relationship and the value distribution of the child join column is uniform. There the base join cardinality formula cannot be improved upon by adding histograms. On the contrary, histograms may lead to over or under estimation depending on the exact data distribution – and even the Oracle version and release level.
  That statement relates only to the join itself. If the foreign key column has a skewed value distribution and is also used as a non-join predicate then a histogram on the foreign key column may be crucial to good estimates.
- Beyond foreign key joins, frequency histograms (FH) provide the best input to the CBO for improved join cardinality estimates. The only exception is the case where one of the join columns is unique – the "halving mystery". However, using size skewonly or auto generally avoids this by not creating a histogram for the unique column.
  Of course, the big drawback of FH is that they are limited to 254 distinct values in the underlying domain[2]. They can switch to a height-balanced histogram when the number of distinct values exceeds this threshold – and you re-gather statistics.
  Tip – keep a close eye on any FH histograms if their NUM_DISTINCT is above 200.
- Height-balanced histograms (HbH) turned out to be of rather limited, if not to say dubious, value. There is only one scenario in all the test cases – the "symmetrical skew" distribution – where they provided clearly better, more accurate join cardinality estimates compared to the no histogram control case.
- Most of the negative effects of histograms can be avoided using size SKEWONLY – or by extension AUTO. It is not because they create better histograms but because they do not store a histogram in the dictionary if there is no pronounced skew in the distribution of the column values.

[1] unless we increase the cardinality of the popular values to >= 53
[2] Even fewer than that for 9i and 10gR1

- The volatility of histograms, especially HbH and especially on join columns, can be a problem. Small, seemingly insignificant changes in column values can make popular values appear or disappear with the consequence of changing the join cardinality estimate. However, this volatility can be taken as a hint that the histogram should not be collected in the first place. Again, SKEWONLY may help avoid that.
- Histograms tend to be "high maintenance", i.e. they need to be re-gathered more frequently than other statistics. This together with the aforementioned volatility does not contribute to plan stability. However, **not** refreshing the histogram may keep plans the same but no longer optimal if the histogram no longer reflects the column's changed data distribution.

APPENDIX A

The following are the set of rules which match the optimizer's join cardinality calculations – most of the time; there are still oddities which could be gaps in "our" understanding of the rules or potentially bugs in the implementation. I put "our" in quotes because it was Alberto Dell'Era (ref also [6]) who came up with the rules. I have only formatted his text

DEFINITIONS

** Popularity
A value in a histogram, whether frequency or height-balanced, is defined "popular" if
delta(EP) = EP - previous (EP) > 1
Let popularity (EP) = 1 if delta(EP) > 1, 0 otherwise

** Counts
Let counts(EP) = num_rows * delta(EP) / max(EP) = num_rows * delta(EP) / num_buckets
Note that for FH, max(EP) = num_rows so counts = delta(EP).

** definition of Join Histogram and Join Value Popularity.
Given the histograms on the join columns:

H1_HIST

EP	Value	Counts	Popularity
0	0	0	0
1	9	10	0
2	10	10	0
4	20	20	1
5	40	10	0
6	50	10	0

H2_HIST

EP	Value	Counts	Popularity
0	0	0	0
1	9	10	0
2	10	10	0
4	20	20	1
5	30	10	0
7	50	20	1

The Join Histogram (JH) is

Value	H1_Counts	H2_Counts	H1_Popularity	H2_Popularity	Join_Popularity
0	0	0	0	0	0
9	10	10	0	0	0
10	10	10	0	0	0
20	20	20	1	1	2
30	0	10	0	0	0
40	10	0	0	0	0
50	10	20	0	1	1

i.e. : consider the set union of H1_HIST.value and H2_HIST.value. Copy H1_HIST.counts (H2_HIST.counts) in H1_COUNTS (H2_COUNTS); if the value is not contained in the H1_HIST(H2_HIST) histogram, let H1_COUNTS=0 (H2_COUNTS=0). Same for H1_HIST.POPULARITY and H1_POPULARITY (H2_POPULARITY). Let JOIN_POPULARITY = H1_POPULARITY + H2_POPULARITY.

** Chopped Join Histogram (CJH)
Let
LLV (Lowest Low Value) = min(min(t1.c), min(t2.c))
HLV (Highest Low Value) = max(min(t1.c), min(t2.c))
LHV (Lowest High Value) = min(max(t1.c), max(t2.c))
HHV (Highest High Value) = max(max(t1.c), max(t2.c))
LMV (Lowest Matching Value) = min({c | t1.c = t2.c})
HMV (Highest Matching Value) = max({c | t1.c = t2.c})
Note: could be equivalently defined in terms of H1_HIST, H2_HIST, JH.
The CJH is simply JH restricted to
LMV <= JH.value <= LHV
Also, to explain a probable bug, let CJH_PLUS_2 be the CJH plus the two buckets immediately following the LHV.

CARDINALITY FORMULA

The cardinality is the sum of the following 4 contributions:

1. Contribution of buckets with JOIN_POPULARITY = 2 ("Popular Matching Popular")
sum of
H1_COUNTS * H2_COUNTS
of rows in the CJH with JOIN_POPULARITY = 2

2. Contribution of buckets with JOIN_POPULARITY = 1 ("Popular not Matching Popular")
sum of (
decode (h1_popularity, 1, h1_counts, num_rows(t1) * density(t1))
*
decode (h2_popularity, 1, h2_counts, num_rows(t2) * density(t2))
)
of rows in the CJH with JOIN_POPULARITY = 1

That is "popular values contribute with their counts, unpopular values with num_rows * density"
Note that for FH, density = 0.5 / num_rows, hence num_rows * density = 0.5;
here's the source of the mysterious halving.

3. Contribution of buckets with POPULARITY = 0 ("All Unpopulars")
Let
num_rows_unpopular (table.column)
= sum of counts of rows in the histogram with POPULARITY = 0
that are represented in CJH_PLUS_2 and VALUE > LMV
(why CJH_PLUS_2 and not CHJ ? probably a bug)
Note that it does not matter whether the value matches or not in the other histogram, or, a fortiori, if it matches with a popular value or not.

Contribution is
if num_rows_unpopular (t1.c) > 0 and num_rows_unpopular (t2.c) > 0
num_rows_unpopular (t1.c) * num_rows_unpopular (t2.c)
* min (density (t1.c) , density (t2.c))
elsif num_rows_unpopular (t1.c) = 0
num_rows_unpopular (t2.c) / num_buckets (t1.c)
elsif num_rows_unpopular (t2.c) = 0
num_rows_unpopular (t1.c) / num_buckets (t2.c)
end if

The first branch could be thought as the standard formula, rewritten in terms of densities, applied to residual values (not popular at all).
Note : this contribution is always rounded up and, if zero, is returned as 1 - this is the source of the "ubiquitous one".

4. Contribution of special cardinality (Probably a bug)
   In this special case, the highest value of one of the two tables is considered a "popular not matching a popular" value:

```
if HMV = LHV and LHV < HHV
    if max (t1.c) = HHV
      h2_counts (value = LHV) * num_rows(t1) * density (t1)
    else /* max (t2.c) = HHV */
      h1_counts (value = LHV) * num_rows(t2) * density (t2)
    end if;
else
    0
end if
```

FALLBACK TO STANDARD FORMULA

Prologue: we know that the CBO uses the standard formula

num_rows (t1) * num_rows (t2) * 1 / max (num_distinct (t1.c) , num_distinct (t2.c))

when no histogram is collected.

There are two variants of this formula (check [5], page 278) that differ only in their ability to detect no-overlaps over the min/max range of the join columns.

Oracle8i does not check the no-overlap condition and always blindly applies the formula

Oracle9i/10g return 1 when the min/max ranges do not overlap.

This formula is also used when histograms were collected and

(a) if any table has num_rows <= 1
    => back to standard formula of 9i/10g

let HPV (Highest Popular Value) = max value in CJH where JOIN_POPULARITY >= 1
if HMV is null (no matching value)
or HPV is null (no popular value)
or LMV > HPV (all popular values < lowest matching value)
=> back to standard formula of 8i
Could be rephrased as "back to standard formula of 8i if all popular values are below the highest matching value, or of course, if no popular or matching value exist."

*** caveat due to Wolfgang's bug
We know that swapping the LHS[1] and the RHS[2] of the = join predicate sometimes changes the CBO join cardinality. Since we can't say which one is the correct cardinality, and which one is the bugged one, by definition we can't say that we have "discovered the formula in full". Could be that the bugged one almost always show up, and sometimes the correct one surfaces.

[1] LHS (Left Hand Side)
[2] RHS (Right Hand Side)

APPENDIX B

TEST CASE SCRIPTS

The sql for the following test cases is the same; only the way the two tables are populated differs.

- @gts j1 100
  @gts j2 100
- explain plan set statement_id='01' for
  select 'A.'||a.id c1, 'B.'||b.id c1
  from j1 A, j2 B
  where a.id = b.id;
- @xplain 01
- col c1 for a5
  col c2 for a5
  select 'A.'||a.id c1, 'B.'||b.id c1
  from j1 A, j2 B
  where a.id = b.id;

I – ONE-TO-ONE JOIN

- create table j1 as select rownum id from dual connect by level <=20;
  create table j2 as select rownum id from dual connect by level <=20;

II – ONE-TO-MANY JOIN

a) create table j1 as select rownum id from dual connect by level <=20;
   create table j2 as select mod(rownum-1,10)+1 id from dual connect by level <=40;

b) create table j1 as select rownum id from dual connect by level <=500;
   create table j2 as select mod(rownum-1,500)+1 id from dual connect by level <=1000;

c) create table j1 as select rownum id from dual connect by level <=500;
   create table j2 as select mod(rownum-1,100)+1 id from dual connect by level <=1000;

d) create table j1 as select rownum id from dual connect by level <=500;
   create table j2 as select mod(rownum-1,200)+1 id from dual connect by level <=1000;

III – ONE-TO-MANY JOIN WITH"HOLES" – SOME PARENTS WITHOUT CHILDREN

a) all parents with same number of children - except those with no children
   create table j1 as select rownum id from dual connect by level <=20;
   create table j2 as select 2*(mod(rownum-1,10)+1) id from dual connect by level <=20;

b) parents with variable number of children
   create table j1 as select rownum id from dual connect by level <=500;
   exec dbms_random.seed(13);
   create table j2 as select mod(rownum-1,trunc(dbms_random.value(1,401)))+1 id from dual connect by level <=100000;

IV – MANY-TO-MANY JOIN

- create table j1 as select mod(rownum-1,20)+1 id from dual connect by level <=40;
  create table j2 as select mod(rownum-1,20)+1 id from dual connect by level <=40;

V – PRINCIPLE OF INCLUSION VIOLATED

- create table j1 as select rownum id from dual connect by level <=20;
  create table j2 as select rownum+10 id from dual connect by level <=20;

VI – Skew in Join Columns

- create table j1 as select case when rownum < 11 then rownum else 10 end id
  from dual connect by level <=20;
  create table j2 as select case when rownum < 11 then rownum else 10 end id
  from dual connect by level <=20;

VII – Symmetrical Skew in Join Columns

a) exec dbms_random.seed(11);
create table j1 as select mod(rownum-1,trunc(dbms_random.value(1,400)))+1 id
from dual connect by level <=5000;
exec dbms_random.seed(17);
create table j2 as select mod(rownum-1,trunc(dbms_random.value(1,400)))+1 id
from dual connect by level <=5000;

b) exec dbms_random.seed(11);
create table j1 as select 400-mod(rownum-1,trunc(dbms_random.value(1,400))) id
from dual connect by level <=5000;
exec dbms_random.seed(17);
create table j1 as select 400-mod(rownum-1,trunc(dbms_random.value(1,400))) id
from dual connect by level <=5000;

VII – Non-symmetrical Skew in Join Columns

- exec dbms_random.seed(11);
  create table j1 as select mod(rownum-1,trunc(dbms_random.value(1,400)))+1 id
  from dual connect by level <=5000;
  col maxval new_value _max
  select max(id) maxval from j1;
  exec dbms_random.seed(17);
  create table j1 as select &_max-mod(rownum-1,trunc(dbms_random.value(1,420))) id
  from dual connect by level <=5000;

VIII – Skew in Join Columns

- create table j1 as select rownum id, to_char(rownum,'fm0000') cat from dual connect by level <= 150;
  create unique index j1_pk on j1(id);
  alter table j1 add constraint j1_pk primary key (id) using index j1_pk;

  create table j2 as select 1 id from dual connect by level <=9229
  union all select 2 id from dual connect by level <=577
  union all select 3 id from dual connect by level <=114
  union all select 4 id from dual connect by level <=37
  union all select 5 id from dual connect by level <=15
  union all select 6 id from dual connect by level <=8
  union all select 7 id from dual connect by level <=4
  union all select 8 id from dual connect by level <=3
  union all select 9 id from dual connect by level <=2
  union all select 10 id from dual connect by level <=1
  union all select 140+rownum id from dual connect by level <=10;

IX – ASYMMETRY ON ZERO

- Join on columns with frequency histograms collected
  -- The CBO estimated cardinality changes when the values of the join columns change sign
  --
  -- Alberto Dell'Era
  -- 10.2.0.2, 9.2.0.6

  ```
  drop table t1;
  drop table t2;

  create table t1 as select 0 x from dual connect by level <= 10000;

  create table t2 as select 0 x from dual connect by level <= 1
  union all
  select 4242 x from dual connect by level <= 42; -- actual value and cardinality do not matter

  begin
    dbms_stats.gather_table_stats(user,'t1',method_opt=>'for all columns size 254');
    dbms_stats.gather_table_stats(user,'t2',method_opt=>'for all columns size 254');
  end;
  /

  explain plan set statement_id='A'
  select t1.x, t2.x from t1, t2 where t1.x = t2.x;
  @xplain A
  ```

- this should not change the estimated cardinality

  ```
  update t1 set x = -x;
  update t2 set x = -x;
  commit;

  begin
    dbms_stats.gather_table_stats(user,'t1',method_opt=>'for all columns size 254');
    dbms_stats.gather_table_stats(user,'t2',method_opt=>'for all columns size 254');
  end;
  /

  explain plan set statement_id='B'
  select t1.x, t2.x from t1, t2 where t1.x = t2.x;
  @xplain B
  ```

X – HEIGHT-BALANCED HISTOGRAM DIFFERENCE IN 10.1.0.5 AND 10.2.0.1

- create table j1 as select rownum id, mod(rownum-1,508)+1 n1 from dual connect by level <= 10160;
  update j1 set n1=n1+3 where mod(n1,4)=1 and id <= 3556; -- = 7 * 508
  update j1 set n1=n1+2 where mod(n1,4)=2 and id <= 3556;
  update j1 set n1=n1+1 where mod(n1,4)=3 and id <= 3556;
  @gts j1 100
  @gcs j1 100 id 254

XI – HEIGHT-BALANCED HISTOGRAM VOLATILITY

- exec dbms_random.seed(31);
  create table j1 as select trunc(dbms_random.value(1,401)) id from dual connect by level <=5080;
  exec dbms_random.seed(11);
  create table j2 as select trunc(dbms_random.value(1,351)) id from dual connect by level <=5080;
  @gts j1 100
  @gcs j1 100 id 254

  col rowid new_value _rowid;

  For 9.2.x and 10.2.0.2+

  ```
  select rowid from j2 where id=342;
  update j2 set id=334 where rowid='~_rowid';
  ```

  For 10.1.0.5 and 10.2.0.1

  ```
  select rowid from j2 where id=235;
  update j2 set id=247 where rowid='~_rowid';
  ```

  @gts j1 100
  @gcs j1 100 id 254

UTILITY SCRIPTS

XPLAIN

```
set define '&'
set ver off echo off
define stmt_id=&1
set lines 180
select plan_table_output from table(dbms_xplan.display('plan_table','&stmt_id','typical'));
```

GTS

```
set define '&'
set ver off echo off
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
     ownname => user
    , tabname => '&1'
    , estimate_percent => &2
    , method_opt => 'FOR ALL COLUMNS SIZE 1'
  );
END;
/
```

GCS

```
set define '&'
set ver off echo off
BEGIN
  DBMS_STATS.GATHER_TABLE_STATS (
     ownname => user
    , tabname => '&1'
    , estimate_percent => &2
    , method_opt => 'FOR COLUMNS size &4 &3'
END;
/
```

REFERENCES

1. Breitling, W. *Fallacies of the Cost Based Optimizer*. in *Hotsos Symposium on Oracle Performance*. 2003. Dallas, Texas.
2. *Predicate Selectivity* ref metalink.oracle.com 68992.1
3. *Tuning by Cardinality Feedback - Method and Examples* ref www.centrexcc.com/papers.html
4. Holdsworth, A., et al. *A Practical Approach to Optimizer Statistics in 10g*. in *Oracle Open World*. 2005. San Francisco.
5. Lewis, J., *Cost Based Oracle - Fundamentals*. 2006: Apress. 520.
6. *Join Cardinality Cracked* ref http://www.jlcomp.demon.co.uk/cbo_book/ch_10.html

---

Wolfgang Breitling is 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 Hotsos Symposiums, IOUG, UKOUG, local Oracle user groups, and other conferences and newsgroups dedicated to Oracle performance topics.

The author wishes to thank Alberto Dell'Era for his contributions. Not only for the formula of how the CBO calculates the join cardinality in the presence of histograms but also for proofreading this paper and his valuable comments.