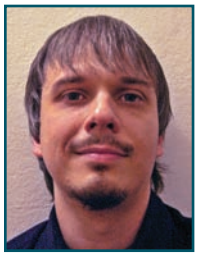
Join Cardinality Estimation with Histograms Explained



By Alberto Dell'Era

This article explains how the Cost Based Optimizer estimates the cardinality of a join when histograms are in place. The goal is to explain how it works in intuitive terms, referring the interested readers to my paper "Join over Histograms"[1] where everything is discussed in full detail, and completely demonstrated by repeatable scripts.

New Look and Feel

Computing a histogram on a column can usually change the plan of statements referencing the column dramatically, and not necessarily for the better. So it is very important to carefully pick the columns on which to put a histogram (and in general the statistics gathering strategy) during the design phase, and then test the resulting application performance before going live.

The more you know about how the CBO uses histograms and how it works, the better your designing and troubleshooting experience will be. A lot of material has been produced over the years about histograms, and [1] is, as far as I know, the first paper that has investigated the (equi) join-with-histograms scenario in full: This article distills the results of that paper into a quick grasp of the essentials. (To learn more about Histograms, refer Wolfgang Breitling's excellent article titled "Histograms – Myths and Facts" in the Q3 2006 issue of *SELECT Journal* as well as John Kanagaraj's article titled "Judicious Use of Histograms in SQL Tuning" in the Q2 2005 issue.)

Versions Scope

The algorithm in [1] has been exhaustively checked on Oracle9*i* Database Release 9.2.0.4 and Oracle Database 10*g* Release 10.2.0.3, but it applies to other 9*i* and 10*g* versions as well. I have also checked it in Oracle Database 11*g* (11.1.0.6), although not exhaustively and it does not seem to have changed, even though 11.1.0.6 uses a new formula for density. (Oracle Database 11*g* computes a "NewDensity" internally that overrides the density stored in the data dictionary that may obviously change the numerical results sometimes.)

All the examples in this article have been run in 10.2.0.3, and can be reproduced using the scripts provided at the end of the article.

Note: For simplicity, we are going to ignore NULL values (corrections to handle them are easy and intuitive anyway).

Histograms Distilled

Height-Balanced Histograms

A Height-Balanced histogram (referred to as HB from now on) with N buckets (`'FOR COLUMN X SIZE N'`) is computed by first sorting the values in the column, and then sampling N sorted values on an uniform grid, then adding the first value (Listing 1 illustrates this).

```
sorted values: 1,2,3,4,5,6,7,8,9,10,11,12
N=2 1,2,3,4,5,6,7,8,9,10,11,12
N=3 1,2,3,4,5,6,7,8,9,10,11,12
N=4 1,2,3,4,5,6,7,8,9,10,11,12
Listing 1
```

Values that get sampled more than once are named *popular* values; for example, in Listing 2 the only popular value is 5 for N=4.

```
sorted values: 1,3,3,4,5,5,5,5,5,5,11,12
N=2 1,3,3,4,5,5,5,5,5,5,11,12  no popular value
N=3 1,3,3,4,5,5,5,5,5,5,11,12  no popular value
N=4 1,3,3,4,5,5,5,5,5,11,12  popular values: 5
Listing 2
```

Spotting popular values is crucial for the CBO. In fact, while estimating the cardinality, the CBO can use only the information contained in the histograms; and, the sampling of course makes the non-sampled values disappear from the CBO's eyes (in technical speak, it reduces the information available). The information reduction is quite severe when no popular value is present. For example, here is what the CBO sees for the second case:

```
N=3 1,?,?,4,?,?,?,5,?,?,?,12  no popular value
```

Only a fraction of the original information is kept, nothing can be said about the non-sampled values. But since the values are sorted before being sampled, when a popular value shows up the CBO is able to keep seeing much more:

```
N=4 1,?,3,?,?,5,5,5,5,?,?,12  popular values: 5
```

Much better! We know with absolute certainty that the popular value 5 occurs at least four times in the column, we have less "?" blocking our vision—much less information reduction.

And we can say some more—since the following two extreme scenarios, where we sampled value 5 exactly at the point where it was changing, are both definitely unlikely:

```
N=4 1,?,3,3,4,5,5,5,5,9,9,12  (5 occurs 4 times)
N=4 1,?,3,5,5,5,5,5,5,5,5,12  (5 occurs 8 times)
```

We can estimate that the most probable number of occurrences of our popular value is halfway between the two extremes—that is, the number of times the popular value has been sampled (2) divided by the number of buckets (4) times the number of rows (12), which is (2/4) * 12 = 6; and

continued on page 14

this is what the CBO exactly does (and stores in the conceptual COUNTS we are going to see below).

The same reasoning cannot be done for unpopular values; all that could be said with certainty is that they occur at least one time in the column, which is not very informative. In fact, we will see that popular values enjoy special consideration in the core of the join cardinality formula, and the role of unpopular ones is quite limited (they are almost ignored).

Frequency Histograms

Frequency histograms (referred as FH from now on) simply contain the number of actual values for each value (the output of the query `SELECT X, COUNT(*) FROM T GROUP BY X`). No sampling is performed here (or, if you prefer, all values are sampled), hence a FH perfectly represents the actual data distribution, with no loss of information whatsoever—so that, if FHs are available for both joined columns, the CBO could in theory make an exact estimate of the join cardinality. (This assumes of course that the tables have not been modified since the statistics collection—an obvious, but silent assumption in this article).

Surprisingly, the concept of popularity extends to FH also: The values whose COUNT (*) > 1 are considered popular, those with COUNT (*) = 1 are considered unpopular, and the latter are essentially ignored as well in the core of the formula. This is quite strange at first sight, since for a FH sampling does not occur, and so values with COUNT (*) =1 are *guaranteed* to be present in the table *exactly* one time, values with COUNT (*) >1 are guaranteed to be present in the table *exactly* COUNT (*) times; there's no apparent reason to distinguish between the two cases!

The reason happens to be purely an implementation one, as we are going to see in the next section; we will see in the last section of this article how this implementation decision may affect the precision of the estimate, with potentially harmful results. Also check [1], chapter "Frequency Histograms and the Mystery of Halving" for more information.

The Unified In-memory Histogram

Nowhere in the data dictionary (until 10.2.0.3 at least) is the histogram type stored. In fact even the HISTOGRAM column in the view DBA_TAB_COLUMNS is an heuristic based on the statistics NUM_ROWS, NUM_BUCKETS, NUM_DISTINCT, and DENSITY, and not a simple decode of a SYS table column as one would expect. This apparently important information is not recorded seemingly because it is not used by the CBO; when loading the histogram data from the SYS tables, the CBO (at least conceptually) transforms it into a common data structure, and the transformation does not need to know the histogram type.

This is probably done to simplify the implementation, and uniformly manage the three possible cases in a join condition: Both joined columns have HBs collected, both have a FH, and one a HB and the other a FH.

The common in-memory histogram (again conceptually) is an array containing

- VALUE: column ENDPOINT_VALUE from DBA_HISTOGRAMS
- COUNTS: the estimate of the number of occurrences of VALUE in the column
- POP: a flag to indicate whether the value is popular or not.

How the transformation is performed is best seen by example; this is of course described in detail in [1].

Transformation for Height-Balanced Histograms

The last example in Listing 2 (that for N=4) is stored this way in the data dictionary (Listing 3; VALUE is ENDPOINT_VALUE and EP is ENDPOINT_NUMBER, both from DBA_HISTOGRAMS):

```
sorted values: 1,3,3,4,5,5,5,5,5,5,11,12
N=4 1,3,3,4,5,5,5,5,5,11,12  popular values: 5

     VALUE         EP
---------- ----------
         1          0
         3          1
         5          3
        12          4
Listing 3
```

A popular value is identified by a "gap" in EP: In this case, for VALUE=5 we have EP=3, for the previous value (VALUE=3) we have EP=1, and since the EP difference (2) is > 1, the value is flagged as popular.

We have already seen before how the CBO estimates COUNTS for HBs; since the number of buckets is equal to max (EP), that can be (and is) computed as

```
num_rows * EP difference / max (EP)
```

So, here is how the in-memory histogram looks like for our HB:

```
     VALUE     COUNTS        POP
---------- ---------- ----------
         1          0          0
         3          3          0
         5          6          1
        12          3          0
```

Transformation for Frequency Histograms

For a FH, the transformation is performed using the *very same algorithm* (here is why the CBO does not need to know the histogram type). So, let's use the same data as in Listing 3, this time with a FH collected:

```
sorted values: 1,3,3,4,5,5,5,5,5,5,11,12
     VALUE         EP
---------- ----------
         1          1
         3          3
         4          4
         5         10
        11         11
        12         12
Listing 4
```

So, the only two gaps in EP are for VALUE=3 and VALUE=5, so we have only two popular values. Their COUNTS are computed by

```
num_rows * EP difference / max (EP)
  VALUE=3 COUNTS = 12 * ( 3-1) / 12 = 2
  VALUE=5 COUNTS = 12 * (10-4) / 12 = 6
```

This happens to be the exact number of occurrences in the column (as you can verify by checking the data above).

The in-memory histogram is

```
     VALUE     COUNTS        POP
---------- ---------- ----------
         1          1          0
         3          2          1
         4          1          0
         5          6          1
        11          1          0
        12          1          0
```

Note in passing that COUNTS is exact also for unpopular values, but the CBO chooses not to use this perfect estimate. As a preview of what we will see in more detail later, the computations it makes are nearly equivalent to using 0.5 as an estimate, instead of using COUNTS=1 – a result of being always num_rows*density = 0.5 for FH. For this reason, I call this strange behavior "Mystery of Halving" in [1] and in the last section of this article.

The Join Cardinality Formula

The formula uses the information contained in the two in-memory histograms of the two joined columns, plus (as we will see) the statistic *density* of both columns and of course *num_rows* of both tables.

The join cardinality estimate is the sum of three contributors that get added together to produce the final estimate:

1. Populars matching populars
2. Populars not matching populars
3. Not-populars subtable.

As the names imply, the first two contributors mine for popular values in the in-memory histograms—and completely ignore not-popular values.

The third caters for the remaining not-popular values by applying a variation of the standard formula (the one used when no histogram is present) to the subtables obtained by removing (conceptually) the popular values.

Note: There is a fourth contributor, which I call "Special cardinality" in [1], but since it is quite likely a bug, I am not going to discuss it here. It should be negligible in most (yet not all) cases.

We will illustrate the contributors using the histogram examples we saw before (one an HB, the other a FH), whose in-memory histograms we repeat here (Listing 5) for your convenience:

```
table T1:
sorted values: 1,3,3,4,5,5,5,5,5,5,11,12
     VALUE     COUNTS        POP
---------- ---------- ----------
         1          0          0
         3          3          0
         5          6          1
        12          3          0
```

```
table T2:
sorted values: 1,3,3,4,5,5,5,5,5,5,11,12
     VALUE     COUNTS        POP
---------- ---------- ----------
         1          1          0
         3          2          1
         4          1          0
         5          6          1
        11          1          0
        12          1          0
```

Listing 5

So we are going to join two tables (T1 and T2) on two columns both containing exactly the same values (just for ease of discussion), but with different histograms (different SIZE) collected, hence, different in-memory histograms (and different density).

Populars Matching Populars (Contributor 1)

The CBO searches for popular VALUEs matching in the two in-memory histograms; it multiplies the COUNT figures for every matching pair, and then adds the results together.

In our example, the only popular matching pair is for VALUE=5, so the result is 6*6 = 36.

The rationale is simple: We have seen before that COUNTS is the best estimate (actually exact for FHs) that the CBO can make about the number of occurrences of VALUE in the tables. So as far as the CBO can infer, VALUE occurs 6 times in T1 and 6 times in T2, hence 36 rows are going to be produced by the join execution. This is extraordinarily sound and intuitive!

Populars not Matching Populars (Contributor 2)

What about popular values that do not match in the other table? If we knew their (estimated) number of occurrences in the other table we could apply the same reasoning as in the first contributor, but unfortunately, as we have seen before, COUNTS is not a reliable estimate for unpopular values, not to mention that our popular VALUE might even not have been recorded in the other histogram (but, still be present in the other table).

So the CBO has to produce the estimate by factoring-in other information, and it does so by observing that a good estimate is the same it produces for an equality predicate on VALUE ("SELECT X FROM <OTHER TABLE> WHERE X = <VALUE>"). Conceptually, we may think that it "probes" the other table with the popular VALUE, without of course actually performing the SELECT, but by simply re-using the algorithm for cardinality estimation of equality filter predicates.

And for an equality predicate, if VALUE is not a popular value in the other table (as this is precisely the case here), the estimate is num_rows*density. So the CBO simply multiplies COUNTS for num_rows*density of the other column.

In our example, the only popular value not matching a popular is VALUE=3 in table T2 with COUNTS=2. T1 has 12 rows, density for the joined column in T1 is 0.111111111, and hence the contributor is 2 * (12 * 0.111111111) = 2.66666666 (rounded up to 3).

Not-populars Subtable (Contributor 3)

First, a step back—when no histogram is collected, the CBO uses the fundamental standard *formula* to estimate the join cardinality:

continued on page 16

```
num_rows (T1) * num_rows (T2) / max (num_distinct (T1), num_distinct (T2))
```

Since density is set to (1 / num_distinct) when no histogram is collected, the standard formula can be rewritten as

```
num_rows (T1) * num_rows (T2) * min (density (T1), density (T2))
```

You can find some information in [1] (especially Appendix A) about the standard formula, but for the purpose of this discussion it is enough to say that it is the formula the CBO uses when it has no idea about the value distribution of the values in the columns (an "average scenario" so to speak, yet able to provide excellent results in many cases).

Now, let's get back to the scenario with histograms. Contributors 1 and 2 have accounted for all popular values, so we must cater for the remaining not-popular ones, and we are done.

Conceptually, the CBO first builds the not-populars subtables by removing all popular values from both tables, and joins them together. It has no idea about the value distribution of these tables, so it resorts to using the standard formula; naming *num_rows_np* the number of values in the not-popular subtables, and density_np the density, this equation is

```
num_rows_np(T1) * num_rows_np(T2) * min (density_np (T1), density_np (T2))
```

Num_rows_np for both subtables is computed by summing COUNTS for the unpopular values, since even if COUNTS is unreliable to estimate the number of occurrences of VALUE, it is a perfect estimate of the number of values covered by the *bucket* whose higher extreme is VALUE. For our example, we have (0+3+3) = 6 for table T1, and (1+1+1+1) = 4 for table T2.

Density_np is assumed equal to the table density; in [1], section "num_rows * density" I explain why this does not seem very sound to me, and in [1], "Overlaps, and improvements," I test an improvement suggestion. In my opinion, it should be density_np = 1 / num_distinct_np.

For our example (10.2.0.3), we calculate that the third contributor is 6 * 4 * min (0.111111111, .041666667) = 1.0000000.

Note on Density

You can skip this section since it is not necessary to know the density formula for the purpose of this article. However, for the curious, letting no (v) equal to the number of occurrences of value v in the not-popular subtable, density for HBs is:

```
sum ( no2(v) ) / [ sum ( no(v) ) * num_rows ], over all v
```

Note that it gives more "weight" to the values that occur more frequently because of the squaring.

Note also that in Oracle Database 11.1.0.6, density is ignored for filter predicates and overridden by a new "NewDensity" computed internally, that for HBs is (num_rows_np / num_rows) * (1 / num_distinct_np), which is definitely aligned with intuition (num_rows*NewDensity = num_rows_np / num_distinct_np), and a strong hint that the square-based formula above is now considered for replacement. The "NewDensity" is used for the second and third contributor as well.

For FHs, density is simply hard coded to 0.5 / num_rows in 9*i*/10g. In 11.1.0.6, "NewDensity" is set to the same value as well, but sometimes the formula for HB is used, which can be shown to be equivalent to setting density to 1.0 / num_rows if a FH is in place.

Final Check

The CBO estimates the join cardinality as being 40:

```
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |    40 |
|*  1 |  HASH JOIN         |      |    40 |
|   2 |   TABLE ACCESS FULL| T1   |    12 |
|   3 |   TABLE ACCESS FULL| T2   |    12 |
-----------------------------------------
Listing 6
```

And the sum of the three contributors is 36 + 3 + 1 = 40, as required.

Chopped In-memory Histograms, and Other

When the two in-memory histograms are not defined on the same interval (their min and max VALUE do not match), as I have silently assumed above for ease of discussion, the CBO "essentially" chops them on the common range—that is, say the histograms range is [1-100] for T1 and [70-172] for T2: The values outside the common range [70-100] are removed from the in-memory histograms, and then the calculations we have shown above are performed.

I say "essentially" in double-quotes since the truncation follows very strange rules (covered in detail in [1] of course) that I personally consider bugs, but the intention seems to ignore values outside the common range, which is sound and intuitive. See [1] for a detailed discussion and some examples illustrating the problems that these bugs (bugs, again, in my humble opinion) may cause.

Also, I have not discussed the rules that the CBO uses to decide whether or not to fall-back to the standard formula even if histograms are collected on both columns. In-depth details can be found in [1], but the main rule is to fall-back if at least one popular value cannot be found above the lowest matching value in the in-memory histograms.

Here is a bonus: The script join_over_histograms.sql provided in [1] implements the CBO formula (the full version, including chopped histograms, the fourth contributor, fall-back to the standard formula and so on) in PL/SQL, reading from the data dictionary views. It is very handy for diagnosing the reasons for a bad join cardinality estimate.

The Mystery of Halving Solved

Now that we have understood in theory how the CBO estimates the join cardinality, it is time to see the formula in practice, on a real-life scenario. We are going to illustrate a strange behavior of the CBO that could be very harmful, perfectly explain it using the knowledge we have gained, and discuss how to prevent it from occurring, and/or how to diagnose it when troubleshooting.

This discussion is going to show yet another example that the more you know about how things work, the more you know about how and where to search

for pitfalls: After all, isn't it counterintuitive that collecting a histogram can make for a worse cardinality estimate, even if you are giving more information to the CBO?

The Scenario: Join over a Foreign Key

The scenario I have chosen is both real-life and frequently occurring: a join over a foreign key, that is, between a child table and a parent table. I have put a foreign key and a primary key constraint on the tables, but this is done just to reproduce the most probable real-life scenario. The presence of the constraints does not change the CBO estimate for our example (it can in some special cases; for example, if you are selecting no column from the parent table, but that is because the CBO *Query Transformation Engine* rewrites the statement as a child-only select, thus removing the join altogether).

So consider a lookup table that translates an internal id into a more descriptive string, maybe to be displayed on the user screen or in a report:

```
SELECT * FROM LOOKUP ORDER BY STATUS_ID;

 STATUS_ID SCREEN_NAME
---------- -----------
         0 PROCESSING
         1 SHIPPED
        99 ARCHIVED
```

This lookup table decodes the status of, say, orders, so say we have an ORDERS table and we want to display each order code and its user-friendly status by performing this join:

```
SELECT ORDERS.CODE, LOOKUP.SCREEN_NAME
  FROM ORDERS, LOOKUP
 WHERE ORDERS.STATUS_ID = LOOKUP.STATUS_ID;
```

Table ORDERS has this value distribution (most orders are ARCHIVED, definitely not uniform):

```
SELECT STATUS_ID, COUNT(*) FROM ORDERS GROUP BY STATUS_ID ORDER BY STATUS_ID;

 STATUS_ID   COUNT(*)
---------- ----------
         0        100
         1        100
        99        800
```

So, we have 1,000 child rows, and obviously the real join cardinality is going to be 1,000 as well.

Intuition Gone Bad

Now, let's show how a seemingly perfectly sound and correct train of thoughts turns out to be a mistake. Pretend we have only a high-level knowledge about histograms, and we did read somewhere that the CBO assumes uniform distribution when no histogram is collected, so we are concerned that the estimate of the join cardinality is going to be horrible for our definitely skewed distribution. Also, knowing that we need histograms on both sides of the join to have the CBO abandon the standard formula and use the alternative estimate of the join cardinality we discussed, we decide to collect histograms on both ORDERS.STATUS_ID and LOOKUP. STATUS_ID, use Frequency Histograms to provide the CBO with the perfect image of the data, and since the reasoning made so far seems sound, obvious, and intuitive, we do not investigate further and call the job done.

Here is the plan for our join (Oracle Database 10.2.0.3):

```
-----------------------------------------------------
| Id  | Operation                    | Name      | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |           |   501 |
|   1 |  MERGE JOIN                  |           |   501 | <= 501 ?
|   2 |   TABLE ACCESS BY INDEX ROWID| LOOKUP    |     3 |
|   3 |    INDEX FULL SCAN           | LOOKUP_PK |     3 |
|*  4 |   SORT JOIN                  |           |  1000 |
|   5 |    TABLE ACCESS FULL         | ORDERS    |  1000 |
-----------------------------------------------------
Listing 7
```

How can it be that the CBO estimates 501, *half* the real cardinality (1,000) of our join?

The mystery is easily solved by doing the math—all values of table ORDERS are popular, all values of table LOOKUP are unpopular, so the first contributor (populars matching populars) is zero, the second (populars not matching populars) is the COUNTS of each value of ORDERS (100,100 and 800) times num_rows*density of LOOKUP (always 0.5 for Frequency Histograms) for a total of 500, and the third contributor (not-populars subtable) accounts for the remaining 1.

This is valid in general of course: Every time you join this way over a child/parent relationship (whether or not a foreign key constraint is enforced) with frequency histograms on both joined columns, you get a halving. Strange as it sounds, that is the way things are implemented.

Note: In Oracle Database 11.1.0.6, this does not happen since in this particular example the CBO uses "NewDensity" that it sets to 1.0 / num_rows, not the traditional 0.5 / num_rows it uses in Oracle Database 9.2.0.4 and Oracle Database 10.2.0.3. This is probably always the case for histograms on parents in Oracle Database 11.1.0.6.

Standard Formula Revisited

Now that we have seen that our intuitive reasoning does not work, what can we do? It seems that we are doomed, since the formula for join over histograms provides a bad estimate, and we "know" that the standard formula assumes uniform distributions, and our data distribution for ORDERS is wildly skewed. But let's give the standard formula a chance and hence remove the histogram on the LOOKUP table:

```
-----------------------------------------------------
| Id  | Operation                    | Name      | Rows  |
-----------------------------------------------------
|   0 | SELECT STATEMENT             |           |  1000 |
|   1 |  MERGE JOIN                  |           |  1000 |
|   2 |   TABLE ACCESS BY INDEX ROWID| LOOKUP    |     3 |
|   3 |    INDEX FULL SCAN           | LOOKUP_PK |     3 |
|*  4 |   SORT JOIN                  |           |  1000 |
|   5 |    TABLE ACCESS FULL         | ORDERS    |  1000 |
-----------------------------------------------------
Listing 8
```

Here the CBO *perfectly* estimates that we are going to retrieve 1,000 rows! How can the standard formula work not only reasonably well, but *perfectly*,

continued on page 18

on skewed data? The reason is simple: The standard formula does not need a uniform distribution on both tables. You can find a short discussion and the mathematical proof in [1], Appendix A, but it is easy to verify that it *always* provides the perfect estimate for the child/parent. The standard formula is

```
num_rows (parent) * num_rows (child) / max (num_distinct (parent), num_
distinct (child))
```

But num_distinct (parent) >= num_distinct (child), since all values in the child need to match in the parent, but some values in the parent may not match a value in the child. So we have

```
num_rows (parent) * num_rows (child) / num_rows (parent) = num_rows (child)
```

which is always the exact number of rows returned.

So to get the exact cardinality, one option is to avoid collecting a histogram on the parent table. However, having a histogram or not on the child makes no difference. One might consider collecting the histogram perhaps to help other statements (such as ones containing "WHERE STATUS_ID = 99"). It is not, however, used by the standard formula for join cardinality estimation.

In passing, as noted by Wolfgang Breitling in [3], if you use SIZE AUTO when collecting histograms, dbms_stats will not collect histograms for uniform distributions such as the one on LOOKUP. This information is definitely worth knowing when designing your statistics gathering strategy.

Moral of the Story

The point I have tried to make with this example is that one has to really know how things work to be able to work effectively. Most certainly, my goal was not to leave you with yet another "best practice," such as "never collect histograms on primary keys," since our industry is already cursed by way too many "best practices" that, when blindly followed, lead to way too many disasters.

Sure, so-called best practices may be used (and are used by seasoned professionals) as a list of things to watch for, as a rough map to design a system, as a collection of things that can go wrong and so on, but seasoned professionals always use their knowledge about how Oracle works when making decisions (not to mention that a professional always tests, makes scripted prototypes, and so on).

Just think of a slightly more complex scenario: Say we need a histogram on the primary key to help a range predicate ("WHERE STATUS_ID <= ...", "WHERE STATUS_ID BETWEEN ...") of another query. What could we do in this case? We certainly cannot answer without knowing how things work.

I hope that this article's explanation of "how things work" for join with histograms will improve your conceptual model of Oracle's CBO, and this will help you in your future work experience.

Acknowledgements

I am grateful to Wolfgang Breitling for his help in writing [1], on which this article is based. As well, I am grateful to Jonathan Lewis for the same reason as well as for writing his book "Cost Based Oracle"[4], the most famous book about the CBO that really needs no introduction. Their contributions are further credited in [1].

References

[1] Alberto Dell'Era, "Join over Histograms", www.adellera.it/investigations/join_over_histograms

Further Reading, and Bibliography

Detailed description of Histograms:
[2] Wolfgang Breitling, "Histograms - Myths and Facts, www.centrexcc.com
[3] Wolfgang Breitling, "Joins, Skew and Histograms", www.centrexcc.com

On the CBO in general (and of course, on Histograms and joins as well):
[4] Jonathan Lewis, "Cost Based Oracle: Fundamentals", Apress, 2006, ISBN 978-1590596364.

New material about joins with filter predicates, and multi-column joins can be found here
[5] Alberto Dell'Era, "Select without replacement", www.adellera.it/investigations/select_without_replacement

How to Reproduce the Examples

Listings 1-6

All histograms are computed using this command (N is the only parameter):

```
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'T', METHOD_OPT=>'FOR COLUMNS X SIZE
N', ESTIMATE_PERCENT=>100);
```

Listing 99 can be used to calculate the in-memory histogram:

```
CREATE OR REPLACE VIEW FORMATTED_HIST_T AS
WITH HIST1 AS (
  SELECT ENDPOINT_NUMBER EP, ENDPOINT_VALUE VALUE
    FROM USER_HISTOGRAMS
   WHERE TABLE_NAME  = 'T'
     AND COLUMN_NAME = 'X'
), HIST2 AS (
  SELECT EP, VALUE,
         LAG (EP) OVER (ORDER BY EP) PREV_EP,
         MAX (EP) OVER ()            MAX_EP
    FROM HIST1
)
SELECT VALUE, EP,
       (SELECT NUM_ROWS FROM USER_TABLES WHERE TABLE_NAME  = 'T')
       * (EP - NVL (PREV_EP, 0))
       / MAX_EP AS COUNTS,
       DECODE (EP - NVL (PREV_EP, 0), 0, 0, 1, 0, 1) AS POP
 FROM HIST2
ORDER BY EP;
Listing 99
```

For Listing 1:

```
CREATE TABLE T AS SELECT ROWNUM X FROM DUAL CONNECT BY LEVEL <= 12;
```

compute stats with N=2,3,4

For Listing 2 and 3:

```
CREATE TABLE T AS SELECT ROWNUM X FROM DUAL CONNECT BY LEVEL <= 12;
UPDATE T SET X = 3 WHERE X = 2;
UPDATE T SET X = 5 WHERE X BETWEEN 6 AND 10;
```

compute stats with N=2,3,4 (Listing 3 is for N=4)

For Listing 4.

same data in Listing 3, then compute stats with N=254

For Listing 6:

```
CREATE TABLE T1 AS SELECT ROWNUM X FROM DUAL CONNECT BY LEVEL <= 12;
UPDATE T1 SET X = 3 WHERE X = 2;
UPDATE T1 SET X = 5 WHERE X BETWEEN 6 AND 10;
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'T1', METHOD_OPT=>'FOR COLUMNS X
SIZE 4', ESTIMATE_PERCENT=>100);
CREATE TABLE T2 AS SELECT ROWNUM X FROM DUAL CONNECT BY LEVEL <= 12;
UPDATE T2 SET X = 3 WHERE X = 2;
UPDATE T2 SET X = 5 WHERE X BETWEEN 6 AND 10;
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'T2', METHOD_OPT=>'FOR COLUMNS X
SIZE 254', ESTIMATE_PERCENT=>100);
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT *
  FROM T1, T2
 WHERE T1.X = T2.X;
SET AUTOTRACE OFF
```

Listings 7-8 (Mystery of Halving)

Set up:

```
CREATE TABLE LOOKUP (STATUS_ID INT CONSTRAINT LOOKUP_PK PRIMARY KEY, SCREEN_
NAME VARCHAR2(10 CHAR));
INSERT INTO LOOKUP (STATUS_ID, SCREEN_NAME) VALUES ( 0, 'PROCESSING');
INSERT INTO LOOKUP (STATUS_ID, SCREEN_NAME) VALUES ( 1, 'SHIPPED');
INSERT INTO LOOKUP (STATUS_ID, SCREEN_NAME) VALUES (99, 'ARCHIVED');
COL SCREEN_NAME FORM A11
SELECT * FROM LOOKUP ORDER BY STATUS_ID;
CREATE TABLE ORDERS (CODE VARCHAR2 (10 CHAR), STATUS_ID INT CONSTRAINT ORDERS_
FK REFERENCES LOOKUP(STATUS_ID));
INSERT INTO ORDERS(CODE, STATUS_ID) SELECT 'ORD'||ROWNUM, 99 FROM DUAL CONNECT
BY LEVEL <= 1000;
UPDATE ORDERS SET STATUS_ID = 0 WHERE STATUS_ID = 99 AND ROWNUM <= 100;
UPDATE ORDERS SET STATUS_ID = 1 WHERE STATUS_ID = 99 AND ROWNUM <= 100;
SELECT STATUS_ID, COUNT(*) FROM ORDERS GROUP BY STATUS_ID ORDER BY STATUS_ID;
SELECT COUNT(*)
  FROM ORDERS, LOOKUP
 WHERE ORDERS.STATUS_ID = LOOKUP.STATUS_ID;
```

For Listing 7:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'LOOKUP', CASCADE=>TRUE, METHOD_
OPT=>'FOR ALL COLUMNS SIZE 254', ESTIMATE_PERCENT=>100);
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'ORDERS', CASCADE=>TRUE, METHOD_
OPT=>'FOR ALL COLUMNS SIZE 254', ESTIMATE_PERCENT=>100);
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT ORDERS.CODE, LOOKUP.SCREEN_NAME
  FROM ORDERS, LOOKUP
 WHERE ORDERS.STATUS_ID = LOOKUP.STATUS_ID;
SET AUTOTRACE OFF
```

For Listing 8:

```
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'LOOKUP', CASCADE=>TRUE, METHOD_
OPT=>'FOR ALL COLUMNS SIZE 1', ESTIMATE_PERCENT=>100);
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'ORDERS', CASCADE=>TRUE, METHOD_
OPT=>'FOR ALL COLUMNS SIZE 1', ESTIMATE_PERCENT=>100);
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT ORDERS.CODE, LOOKUP.SCREEN_NAME
  FROM ORDERS, LOOKUP
 WHERE ORDERS.STATUS_ID = LOOKUP.STATUS_ID;
SET AUTOTRACE OFF
EXEC DBMS_STATS.GATHER_TABLE_STATS (USER, 'ORDERS', CASCADE=>TRUE, METHOD_
OPT=>'FOR ALL COLUMNS SIZE 254', ESTIMATE_PERCENT=>100);
SET AUTOTRACE TRACEONLY EXPLAIN
SELECT ORDERS.CODE, LOOKUP.SCREEN_NAME
  FROM ORDERS, LOOKUP
 WHERE ORDERS.STATUS_ID = LOOKUP.STATUS_ID;
SET AUTOTRACE OFF
```


About the Author

Alberto Dell'Era has spent all his professional life working within the Italian Telecommunications sector since 1996, specializing in Oracle full-time since 1999. He currently works for Etnoteam S.p.A. (a Value Team S.p.A. company, a consultancy of more than 2,600 employees), where he is mainly responsible for all Oracle-related developments for the flagship customer Web portal of one of Italy's largest mobile operators. He is a member of the OakTable Network, the famous organization of Oracle professionals distinguished by the use of the Scientific Method (and related ethics) for all their activities. He holds a degree in Electronics Engineering and can be contacted at **alberto.dellera@gmail.com**.