JOIN OVER HISTOGRAMS

Alberto Dell'Era
alberto.dellera@gmail.com

ABSTRACT

We will present and illustrate the formula used by the Oracle™ Cost Based Optimizer to estimate the cardinality of an equijoin, when both the columns referenced in the join predicate have histograms collected.

This paper is divided in three parts - the first, "Essentials of the Formula", has been designed to give a quick, intuitive grasp of the structure and meaning of the formula, providing some general context at the same time. It will appeal to readers that are interested in quickly absorbing the general results.

The following chapter, "The Formula in Full", exposes the formula in all its glory and technical details, and serves as a precise, mathematical definition of the formula for the keen Performance Tuner.

The final chapter, "The Formula in Action", shows the formula applied on real-case scenarios; people that need a practical use to be motivated in reading a paper can start peeking there.

To help reproducibility, the formula has been implemented as a PL/SQL program, and to build confidence in its correctness, it has been validated by checking it over all possible scenarios using an automatic test engine, that checks the formula output against the CBO output. Both are provided in the paper's supporting code.

Ackowledgements

This paper is the result of a collaboration between me and Wolfgang Breitling, who has shared and discussed with me a lot of his data and reasonings (many have been published as papers on his web site, www.centrexcc.com, and presented at various Oracle Events) and played the fundamental role of Devil's Advocate, and agreed to the task of reviewing the final drafts of the paper. Without Wolfgang's continued support, the investigation, whose results are presented in this paper, would have never been finished or published. Of course, the responsibility for any mistake is exclusively mine.

Many thanks also to Jonathan Lewis who prompted me to add and improve some important points of the final draft.

Introduction

The number of rows (or *actual cardinality*) returned by a straight equijoin between two tables

```
 select ...
   from t1, t2
 where t1.value = t2.value; -- (equi) join predicate
```

can be anything between zero (when no value matches over the join predicate) and num_rows[i] (t1) * num_rows(t2) (when a unique, matching, value is contained in both columns) - a very huge range.

Estimating where the actual figure will lie between these two physical bounds at execution time is absolutely *critical* for the CBO, since a bad estimation would easily make for a very bad plan, especially if this simple equijoin is just a fragment of an (even slightly) more complex statement, as it is almost always the case.

Note: in this paper we will consider only this case of a single-column equi-join between two tables, without filter predicates and other SQL clauses commonly found in real-life. This is of course done to keep the paper as simple and focused as possible.

[i] unless otherwise noted, in this paper we will assume that the tables have not changed since the last statistic collection, and so num_rows(t) is equal to the actual number of rows in table t.

To illustrate the problem, appreciate the difficulty of the task, and allow me to introduce some terminology informally, consider these scenarios for two tables with num_rows(t1)=num_rows(t2) =9, num_distinct(t1)[ii] = num_distinct(t2) =3:

Scenario **A**	Scenario **B**	Scenario **C**
t1 t2 1 1 2 2 3 3 3 3 3 3 3 3 3 3 3 3 3 3	t1 t2 1 1 1 1 1 1 2 2 2 2 2 2 3 3 3 3 3 3	t1 t2 1 1 2 1 3 1 3 1 3 1 3 1 3 1 3 2 3 3
cardinality = **1*1** + **1*1** + **7*7** = 51	cardinality = **3*3** + **3*3** + **3*3**= 27	cardinality = **1*7** + **1*1** + **7*1** = 15

Definitely the cardinality depends heavily not only on the "shape" of the data distribution (B.t1 and B.t2 are cases of *uniform*[iii] distributions, all the others of *skewed* distributions) but also on how the data match (note that the "shapes" of the distributions in scenario A and C are exactly the same, the only difference is in the matching value pairs - and the resulting cardinality is dramatically different).

So if you desire an exact calculation, you have to know, for both tables, the map between the list of values (1,2,3) and their counts [e.g. for A.t2, the map is counts(1) = 1, counts(2) = 2, counts(3) = 7]. If you know this map for both tables, you simply add the products of the counts for the matching value pairs to get the exact join cardinality (as I have done at the bottom of each scenario).

Oracle can do this calculation automatically: all you have to do is to tell it to calculate the map[iv] for each table and then store it in the data dictionary as a *Frequency Histogram* statistic[v]. Since you provide perfect information, the CBO estimation could be perfect as well - I say "could" and not "will" because the CBO calculation, surprisingly, is not exact, as we will see. But in principle, since the map is all it takes to make a perfect calculation, the CBO could calculate the join cardinality exactly.

Unfortunately the perfect map could be really huge (think about a table with millions of distinct values), and since the query compilation (PARSE in Oracle terminology) happens online, when the user is waiting, the CBO limits the calculation elapsed time by simply limiting the amount of information it has to process - by allowing you to collect Frequency Histograms (maps) with at most 254 values. This of course saves space in the data dictionary as well.

Since we do not need an exact calculation after all, but only one that is accurate enough to identify the optimal plan (and we can even be happy with a slightly suboptimal one in most cases), another option to limit the query compilation elapsed time is to store only *some* of the information contained in the perfect map - only the amount that can be processed quickly enough - and accept the unavoidable reduction in the estimate quality.

One method Oracle provides to reduce the information in the map is to sample (deterministically) the values in a particular way and put them in an *Height-Balanced* histogram, whose max size is constrained by 254 as well. We'll recap how the information is reduced in some detail, since it is necessary to understand the meaning of the formula.

Side note: it is worth stressing how difficult it is to pick the information to keep. Assume for example the simplistic (but sound at first sight) approach to put a limit on the number of values to keep in the map, and keep only those with the greatest counts. Say you want to keep only two values: in scenario C, for table t1 you definitely want to keep value =3, then you need to choose between value=1 and value=2 - say you end up ignoring the value=1 for t1, and say the same happens for value=3 in t2. It may seem at first sight that we have kept almost all of the "information" in each map, but the resulting estimate would be down from the exact 15 to the very bad 1. So one could think to look at the values in the other table and keep only the ones that match "the most" - until one realizes that a table could be potentially joined to every other table in the database, and discards this approach as impractical. And so on - whatever bit you ignore, it could be the most important bit, and there is no easy recipe to tell which the most important bit is. A really hard problem indeed.

[ii] I should technically write num_distinct (t1.*value*), since num_distinct is a column statistic, not a table one - but since our tables will always have one single column, the notation is unambiguous and makes for slightly less visual clutter.
[iii] aka "even".
[iv] Obtained by performing, as the reader has probably guessed, a simple *select value, count(*) from t group by value*.
[v] Since in this paper we generally assume that the tables have not changed since the last statistics collection, these figures could be called *measures*, not simply *statistics*.

You do not necessarily need the map to perform an exact estimation. In fact if the distribution of each table[vi] is perfectly uniform [counts(value(i)) = constant = num_rows / num_distinct] and *all the values in the table with the smaller num_distinct match in the other table*[vii] ("principle of inclusion", see [Breitling, JSH]), you don't need to know the map, since it could be easily[viii] shown mathematically that the exact cardinality is given by the *standard formula*[ix]:

num_rows(t1) * num_rows (t2) / max (num_distinct (t1), num_distinct (t2))

For example, Scenario B has both a perfectly uniform distribution and all matching values, and in fact the exact cardinality is 9*9 / max (3, 3) = 27.

This elegant formula is the one used by Oracle if no histogram is collected[x]; in this case the CBO assumes an uniform distribution and that the principle of inclusion holds, so applies the standard formula. We will see that a variant of it is used also as one of the contributors to the formula we are discussing in this paper.

Side note: if the distribution is perfectly uniform, the map can be rebuilt knowing only num_rows and num_distinct - so one can conceptually say that the map is reduced to two simple numbers, without any loss of information whatsoever, even if the map contains billions of distinct values. Collecting an histogram can only degrade or at most match this perfect representation of the map. So if you know that your data is perfectly (or reasonably) uniform, and that the principle of inclusion (reasonably) holds, you can easily get hugely better cardinality estimations by collecting no histogram - since you are effectively collecting the perfect maps, and the standard formula is making the exact calculation based on the two perfect maps (this is the cornerstone of the proof in Appendix A).

Side note**: the standard formula gives the exact cardinality when a FK exists between the two columns (for single-column FKs only**[xi] - I am restricting the scope of this paper to single-column joins unless otherwise noted), even if the child table distribution is not uniform (as it is generally the case) - in fact the two assumptions stated above can be relaxed[xii], they are not the most general possible. It is not difficult to check the validity in this scenario: the cardinality in this case is equal to num_rows (child table), since every row in the child has one and only one match in the parent. Some values of the parent may not match in the child, so num_distinct (parent table) >= num_distinct (child table); the uniqueness of parent values implies num_rows (parent table) = num_distinct (parent table), hence the formula becomes num_rows (child table) * num_rows (parent table) / num_rows (parent table) = num_rows (child table). So if you only join across single-column FKs (as it almost always, if not always, happens in OLTP) - histograms are useless or dangerous to improve the join cardinality estimation - but you may want them for other reasons; for example, because you also put a filter predicate on one or both the join columns (in the join statement or another, maybe single-table, statement), or because one of the tables is joined to other tables that are not related with a nice FK constraint. As always, you really need to know your data, and how they are used, to design or optimize a system or a statistics gathering strategy - and histograms are no exception.

[vi] This is the way it is usually presented, but this requirement can be relaxed into requiring that only one table (the one with the greater num_distinct) has a perfect uniform distribution (see Appendix A and the following side note about FKs).

[vii] If this is not the case, even if both tables have a uniform distribution you may want to collect (frequency) histograms to tell the CBO which values match, and which do not.

[viii] Easily but not trivially - so I have provided the derivation in Appendix A.

[ix] See [Lewis, CBO, page 265] for more details (also Metalink note 68992.1) - I have simplified the formula assuming that the tables contain no null values, a silent assumption in this paper. Adjusting for null values is trivial anyway.

[x] Technically, if *at least one table* lacks an histogram.

[xi] For multi-column (aka composite) FKs, the formula holds as well, but since the number of distinct values of the composite key is not *generally* known, the CBO cannot use it - I have said "generally" since in 10g the CBO may know, and use, that information from other sources (distinct_keys of PK-supporting composite indexes) and it does seem to compute the lower and upper bound of the multi-column standard formula and use it as a sanity check (see [Lewis, CBO, page 273]). Some of this functionality is reported (I have not investigated extensively) to have been backported in 9.2.0.6 as well.

[xii] See Appendix A.

Essentials of the Formula

We will first recap the traits of Frequency and Height-Balanced histograms that play an essential role in the formula (for an extensive dissertation, see [Lewis, CBO, page 151]), spend some words about the very important concept of popularity, discuss the meaning of the fundamental num_rows*density derived statistic, and finally present the formula, which is composed of four factors or contributors. We will introduce on our way most of the terminology and observations we are going to use when we finally present the formula.

Frequency histograms (FH)

In order to tell Oracle to build a Frequency Histogram (FH), you simply collect your column statistics[xiii] using SIZE N[xiv], using N >=[xv] num_distinct. Since N must currently (10.2.0.3) be <= 254, a FH can be collected only if the column has num_distinct <= 254.

1
1
2
3
3
3
3

The FH of the table on the left[xvi] is:

VALUE	EP	COUNTS
1	2	**2**
2	3	**1**
3	7	**4**

VALUE is dba_histograms.endpoint_value[xvii], and is the actual value found in the table, the entry point in the map.
EP is dba_histograms.endpoint_number, and in the case of FHs is the number of rows whose value is less than or equal than VALUE.
COUNTS is not contained in dba_histograms, it is derived by simply computing the difference between the current VALUE and the previous one, so in this case
counts(1) = 2-0 = 2
counts(2) = 3-2 = 1
counts(3) = 7-3 = 4
and of course is the number of counts of this value in the map (as you can check visually).

Height-balanced histograms (HB)

In order to collect an Height-Balanced histograms (HB), you use SIZE N with N <[xviii] num_distinct and of course N != 1 (since SIZE 1 is the way to order Oracle to compute no histogram). N is constrained to be <= 254 currently (10.2.0.3), to put a limit on the resources used by the histogram, and especially to limit the query optimization elapsed time.

Oracle will compute the histogram by sampling the values - precisely, by ordering the values and then performing a uniform sampling, with the sampling interval equal to num_rows / N[xix], and then adding the min value[xx]. This is best

[xiii] dbms_stats.gather_table_stats (..., method_opt=>'for column value size N',..), and variants.
[xiv] According to Wolfgang Breitling ([Breitling, HMF, slide 34]), SIZE SKEWONLY computes the histogram using SIZE N with N equal to a version-dependent internal formula (always 254 in 10g, between 75 and 151 depending on the size of the table in 9i), then discards the histogram if it decides it is not worth keeping. SIZE AUTO adds knowledge about the usage of the column in where clauses on top of it. Anyway, the resulting histogram, if any, is no different from the ones you may manually collect using SIZE N, and the method used by dbms_stats (SKEWONLY, AUTO or SIZE N) for collection is not known to the optimizer.
[xv] Because of a bug, in some versions N must be about 35% bigger than num_distinct to collect a FH. See [Breitling, JSH], [Lewis, CBO, page 164] for details. The documentation (e.g. Metalink note 72539.1) says that N bust be "greater than or equal" num_distinct, and of course the documentation dictates the expected behavior.
[xvi] This example is contained in fh_hb_simple_examples.sql.
[xvii] The actual value may be stored in dba_histograms.endpoint_**actual**_value if the column is not a number (i.e. a varchar2) and the first six bytes of some values are the same. We consider only numbers in this paper for simplicity.
[xviii] But see the previous note for FH about the "35%" bug.
[xix] Actually ceiling (num_rows / N), since the sampling interval has to be an integer (so the last bucket may be smaller than the others). I will strive with num_rows / N = integer for simplicity whenever possible.

seen with an example[xxi] - I have put an arrow next to the sampled values, and matched them by color in the resulting HB histogram (N=3):

1 <= 2 3 <= 4 5 6 <= 7 8 9 <=	The HB of the table on the left (N=3): VALUE / EP / COUNTS 1 / 0 / **0** 3 / 1 / **3** 6 / 2 / **3** 9 / 3 / **3**

EP and VALUE are from dba_histograms (see the previous section). VALUE has the same meaning; EP is zero for the first value sampled (the min value) and indicates that this is the EP-th sample. Letting max_ep = max(EP), diff_ep(i) = current (EP) - previous (EP) [and defining previous(EP)=0 for the first value], COUNTS(i) is given by

num_rows * diff_ep(i) / max_ep

in words, it is approximately the number of rows contained between the current sampled value and the previous one (note that EP=0 has COUNTS=0).

Notice what happens when the same value gets sampled twice[xxii]:

1 <= 2 3 <= 4 9 9 <= 9 9 9 <=	The HB of the table on the left (N=3): VALUE / EP / COUNTS 1 / 0 / **0** 3 / 1 / **3** 9 / 3 / **6**

Even though VALUE=9 is sampled twice, it is reported only once in the histogram, but EP increases by 2, not by one (of course, if it were sampled k times, EP would increase by k)[xxiii]. COUNTS increases accordingly.

NB: this definition of COUNTS as "num_rows * diff_ep(i) / max_ep" can be applied to FHs and give the same results we have seen in the previous section; mathematically they are the same.

Side note: *an Height Balanced histogram is the Frequency Histogram of the sampled values.*
This will become apparent later in the paper, but it's worth performing this mental experiment: take the sampled values (minus the first one, the min value) and put them in a table, compute a FH on this table, then add the min value as the first endpoint to record the (very important) min value if not already represented in the histogram: the resulting histogram is indistinguishable from the HB one on the original table. Demo hb_as_fh_of_sampled_values.sql illustrates this for the table above (the sampled values are the ones marked with an arrow of course).
One could also conceptually say that the only difference between HBs and FHs is the first sampling step, which is a real (uniform) sampling for HBs and a "sample all" for FHs, followed in both cases by the computation of a Frequency Histogram as defined in Mathematics.

Values that are sampled more than once (and thus whose diff_ep > 1) play a very important role in the formula, and are called **popular values**. The concept of popularity is so important, in fact, to merit a full section all by itself.

[xx] The sampling is not perfectly uniform, of course, when num_rows/distinct is not a natural number, and there are also version-dependent variations (see [Breitling, JSH]), but the actual sampling is always an excellent approximation of the provided description, which is adapted from the one in [Breitling, HMF, page 2].
[xxi] This example is contained in fh_hb_simple_examples.sql as well.
[xxii] It's in the same script as the previous example, fh_hb_simple_examples.sql.
[xxiii] This is officially called *compressing* the histogram (see the description of dbms_stats.prepare_columns_stats in the *PL/SQL Packages and Types Reference)*.

Popularity

For HBs, **popular** (those with diff_ep > 1) values have statistical properties quite different from **unpopular** (those with ep_diff = 1) ones, since the actual number of occurrences in the table for popular values is reasonably approximated by COUNTS, but the same does not apply for unpopular ones.

Consider the following data distributions that produce the same HB histogram[xxiv] after sampling, and notice the unpopular value equal to **6**:

1 <=	1 <=	1 <=	1 <=	1 <=	1 <=
2	2	2	2	2	2
3 <=	3 <=	3 <=	3 <=	3 <=	3 <=
4	4	4	4	4	6
5	5	5	6	6	6
6 <=	6 <=	6 <=	6 <=	6 <=	6 <=
7	6	6	6	6	6
8	8	6	8	6	6
9 <=	9 <=	9 <=	9 <=	9 <=	9 <=

The HB of the tables on the left (N=3):

VALUE	EP	COUNTS
1	0	0
3	1	3
6	2	**3**
9	3	3

diff_ep (VALUE=6) = 1 => unpopular

The number of actual occurrences in the table for an unpopular value can be anything between 1 and 2 * T - 1, where T is the sampling interval (num_rows / N). Notice that COUNTS (equal to T for HBs) does not give a reliable estimate of the actual number of occurrences of the unpopular value, even if it is the midpoint between the physical minimum (1) and maximum (2 * T - 1): it can be wrong by T * 100% (first case on the left), and especially, the distributions on the left (1 or few occurrences per value) occur more frequently in practice than the ones on the right, and for them, COUNTS=T is a very bad estimate, unless T is close to a few units - but unfortunately T cannot be less than the (current) maximum sampling resolution of num_rows / 254. To overcome this limitation, we will see that the CBO ignores COUNTS for unpopular values and cleverly uses another statistic instead: num_rows * density.

The scenario is radically different for popular values[xxv]:

1 <=	1 <=	1 <=	1 <=	1 <=	1 <=
2	2	2	2	2	2
3 <=	3 <=	3 <=	3 <=	3 <=	3 <=
4	4	4	4	4	9
5	5	5	9	9	9
9 <=	9 <=	9 <=	9 <=	9 <=	9 <=
9	9	9	9	9	9
9	9	9	9	9	9
9 <=	9 <=	9 <=	9 <=	9 <=	9 <=
10	9	9	9	9	9
11	11	9	11	9	9
12 <=	12 <=	12 <=	12 <=	12 <=	12 <=

The HB of the tables on the left (N=4):

VALUE	EP	COUNTS
1	0	0
3	1	3
9	3	**6**
12	4	3

diff_ep (VALUE=9) = 2 => popular

Since the sampling is performed after ordering by value, the actual values between the sampling points of each popular value are equal to the popular value itself - hence the number of actual occurrences in the table is at least (diff_ep-1) * T + 1 = COUNTS - T +1, at most (diff_ep+1) * T - 1 = COUNTS + T - 1, and so the midpoint between "least" and "most" is exactly COUNTS, can be wrong by at most T-1, and the error/estimate ratio is T/COUNTS. So COUNTS, for popular values, is a reasonable estimate of the actual number of their occurrences in the table.

Notice also that if you make a very small "infinitesimal" modification to the table (add or remove a row, update a value) an unpopular value can very easily disappear from the histogram as the values shift under the sampling grid (it is "very unstable"), but a popular value is much more likely to stay there - it can have its "popularity degree", diff_ep, and so COUNTS, changed (could be also downgraded to unpopular) but it is comparatively "much more stable" then unpopular values. This difference is accounted for in the formula, as we will see.

[xxiv] Demo in the ubiquitous fh_hb_simple_examples.sql.
[xxv] Demo, again, in fh_hb_simple_examples.sql.

What about FHs ? The above discussion does not apply for them, since both COUNTS and VALUE are not obtained by sampling, and COUNTS is exactly the actual number of occurrences of the corresponding VALUE, with no uncertainty whatsoever. But surprisingly, the CBO ignores this fact, extends the definition of popularity (diff_ep > 1, which means COUNTS > 1) to them, and consider unpopular values (COUNTS=1) less reliable. In fact we will see that, despite them looking very different at first sight, *the formula does not distinguish between FHs and HBs*, they both look the same once "loaded" in memory to compute the join cardinality estimate.

*num_rows * density*

The derived statistic *num_rows * density*[xxvi] is such a cornerstone of the formula, that it is really worth understanding its meaning, and play a bit with it.

When we will look at the formula, we will see that the most probable meaning of *num_rows * density* is an approximation of the "average number of occurrences of each value not represented in the histogram as a popular value". That is, build a "not-populars subtable" by removing the popular values; "select count(value) / select count (distinct value) from subtable" should be very close to the original table's num_rows * density.

This is evident when we have no histogram, and so no popular value - in this case, dbms_stats sets density = 1 / num_distinct, hence num_rows * density = num_rows / num_distinct.

Let's see this for a very important scenario (first example in the num_rows_times_density.sql demo) - a table where all values are distinct, with the exception of a value (here, 99) that occurs often, and so gets recorded as a popular value:

1 <=
2
3 <=
4
5
6 <=
99
99
99 <=
99
99
99 <=

The HB of the table on the left (N=4):

VALUE	EP	COUNTS	
1	0	0	<= unpopular
3	1	3	<= unpopular
6	**2**	3	<= unpopular
99	**4**	6	<= **popular**

num_rows * density = **1**; num_rows/num_distinct = 1.71428571

```
SQL> select count(value) / count (distinct value) from t1  where value != 99;
        1
SQL> select count(value) / count (distinct value) from t1;
        1.71428571
```

In this case, **num_rows * density** matches exactly the output of the first query, which measures the number of occurrences per value in the not-populars subtable; in general, it matches the output perfectly when the distribution of not-popular values is perfectly uniform (check the second example in the demo).

There is more - the CBO uses num_rows * density also to estimate the cardinality of queries like "where value = constant" (see [Lewis, CBO, page 43] for details) if constant is not a popular value (NB: regardless of whether constant matches an unpopular value in the histogram, matches a not-popular value not in the histogram, or does not match any value in the table), and of course, physically, the expected cardinality for this kind of selection is the average number of occurrences for each value in the not-populars subtable.
The demo checks this on the table presented above:

```
SQL> select * from t1 where value = 2; -- unpopular value in the histogram

-----------------------------------------
| Id  | Operation         | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |
|*  1 |  TABLE ACCESS FULL| T1   |     1 |
-----------------------------------------
```

[xxvi] density, of course, of the column we are considering - and I am assuming we have no null in the column, otherwise we should correct num_rows by subtracting the number of null values, user_tab_columns.num_nulls.

```
SQL> select * from t1 where value = 2.3; -- value not in the table
----------------------------------------
| Id  | Operation         | Name | Rows  |
----------------------------------------
|   0 | SELECT STATEMENT  |      |     1 |
|*  1 |  TABLE ACCESS FULL| T1   |     1 |
----------------------------------------
```

Multiply density by **11**:

```
SQL> exec set_density[xxvii] ('t1', 'value', 11 * .0833333333333333);
density of t1.value changed from .0833333333333333 to .9166666666666663

SQL> select * from t1 where value = 2; -- unpopular value in the histogram

----------------------------------------
| Id  | Operation         | Name | Rows  |
----------------------------------------
|   0 | SELECT STATEMENT  |      |    11 |
|*  1 |  TABLE ACCESS FULL| T1   |    11 |
----------------------------------------

SQL> select * from t1 where value = 2.3; -- value not in the table

Execution Plan
----------------------------------------
Plan hash value: 3617692013

----------------------------------------
| Id  | Operation         | Name | Rows  |
----------------------------------------
|   0 | SELECT STATEMENT  |      |    11 |
|*  1 |  TABLE ACCESS FULL| T1   |    11 |
----------------------------------------
```

Side note: it is not surprising after all that there is a link between this and the join formula - since a join can be thought conceptually as a loop on one table, and for each row retrieved, a selection on the second using "where value = <value retrieved>".

Side note: the same applies for FHs. In fact, when constant="unpopular value in the histogram", the cardinality is really num_rows * density = 0.5 rounded to 1, as you can check by manually setting (increasing) density.

Moreover - adapting from [Lewis, CBO, page 172][xxviii] , density for HBs is computed as

function (not popular subtable) / (**num_rows** * num_rows_unpopular)

we know that for FHs, density is set to

0.5 / **num_rows**

which makes for a strong hint that density is meant to be multiplied by **num_rows.**

[xxvii] set_density is a routine of mine to change "density" while preserving the histogram and other statistics. You can't simply use dbms_stats.set_column_stats (..., density => <new value>, ...) since this destroys the histogram also, as any call to dbms_stats.set_column_stats() without an histogram in input will do.
In fact the fastest way to erase an histogram without re-visiting the table to re-collect the other statistics is to call dbms_stats.set_column_stats(..., distinct => <current num_distinct>, ...) - a technique invented by Wolfgang Breitling in an answer on the Oracle-L mailing list (http://www.freelists.org/archives/oracle-l/03-2007/msg00575.html)

[xxviii] Exact quotation: "In purely descriptive term, the density [of height-balanced histograms] is as follows:
sum of the square of the frequency of the nonpopular values /
(number of nonnull rows * number of nonpopular nonnull rows)"

Side note: another observation seems to contradict the above reasoning. The standard formula

num_rows(t1) * num_rows (t2) / max (num_distinct (t1), num_distinct (t2)) =
num_rows(t1) * num_rows (t2) * min (1 / num_distinct (t1), 1 / num_distinct (t2))

when no histogram is collected, can be rewritten, since 1 / num_distinct = density as

num_rows(t1) * num_rows (t2) * min (density (t1), density (t2))

and we will see that one of the contributors of the formula is

num_rows_unpopulars (t1) * num_rows_unpopulars (t2) * min (density (t1), density (t2))

that is, the standard formula applied to the "not-populars subtables" - but unfortunately if num_rows * density is meant to give the "average number of occurrences for each value in the not-populars subtable", num_distinct should be

"number of rows" / "average number of occurrences for each value" =
num_rows_unpopulars / (num_rows * density)

so the equivalent of 1 / num_distinct for the not-populars subtable should not be density, but instead

(num_rows / num_rows_unpopulars) * density

and the rewrite of the standard formula should be

num_rows_unpopulars (t1) * num_rows_unpopulars (t2) * min (
(**num_rows (t1) / num_rows_unpopulars (t1)**) * density (t1),
(**num_rows (t2) / num_rows_unpopulars (t2)**) * density (t2)
)

Would this modification make for a better estimation of the cardinality ? I have found some scenarios where this is true:

a) unpopulars_subtable_corrected.sql shows the CBO underestimating the cardinality by 20% - once corrected, the estimate gives the *exact* result instead;
b) see the "overlaps, and improvements" section.

In my opinion, the current implementation is a (small) mistake, but it could also be that the current version gives more accurate results "in general", even if it is not really accurate at least on these two scenarios (deterministically uniform and statistically uniform distributions respectively) - I am thinking at "biased estimators" perhaps.

You will not probably see much difference in practice; in fact, this contributor is normally small compared to the other contributors of the formula, and in many cases the corrected formula gives almost the same result: e.g., in the extreme case where you have no popular value, num_rows_unpopulars = num_rows so the difference disappears, and in the opposite extreme case where you have num_rows_unpopulars = 0 for at least one table, the difference does not matter since num_rows_unpopulars (t1) * num_rows_unpopulars (t2) = 0. So only in the intermediate cases (popular values covering, say, 50% of the rows) the correction may matter.

The Formula in a Nutshell

In order to estimate the cardinality, the CBO loads in memory the histograms for both tables and then performs a series of calculations, most of which require searching for VALUE matches. I will base the illustration on the concept of the Join Histogram, which makes it easy to see the matching pairs; my implementation of the formula in PL/SQL and SQL also explicitly computes the Join Histogram (a full outer join between the histograms) as its first step.

Whether the histogram is a FH or an HB is not factored in in the formula - and in fact, the histogram type is not stored in the data dictionary[xxix], even if dbms_stats knows this information, since it would be useless to the CBO. Maybe, this was done to simplify the algorithm, and make transparent the scenario of a join between two tables whose histograms are of a different type.

The formula itself is composed of 4 components added together. Let's illustrate them using an example[xxx] that captures the essentials, without touching any of the "border" cases.

The join histogram

Given these tables with their respective histograms (the left one an HB, the right one a FH):

10 <=
10 <=
10
10 <=
10.5
20 <=
30
30 <=
30
30 <=
30.5
40 <=
40.5
50 <=
50.5
60 <=
70
70 <=
70
70 <=

The HB of table t1 on the left (N=10):

VALUE	EP	COUNTS
10	2	4
20	3	2
30	5	4
40	6	2
50	7	2
60	8	2
70	10	4

num_rows (t1) = 20
density (t1) = 0.05
num_rows(t1) *density(t1) = 1

The FH of table t2 on the right:

VALUE	EP	COUNTS
10	2	2
20	3	1
50	6	3
60	7	1
70	11	4

num_rows (t2) = 11
density (t2) = 0.045454545
num_rows(t2) *density(t2) = 0.5
.

10
10
20
50
50
50
60
70
70
70
70

The Join Histogram is

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)
POP	4	10	2	POP
UN	2	20	1	UN
POP	4	30	-	-
UN	2	40	-	-
UN	2	50	3	POP
UN	2	60	1	UN
POP	4	70	4	POP

[xxix] In fact the 10g dba_tab_columns.histogram is an heuristic based on num_rows, num_buckets, num_distinct and density. It is also frequently wrong for FHs, since it contains the factor density*num_buckets <= 0.5 that is very prone to rounding errors (e.g. density*num_buckets = 0.5000001 instead of 0.5)
Also, the type of histogram reported in the 10053 event is for information only, probably based on the same formula used for dba_tab_columns.histogram in 10g, and with the same weaknesses.
[xxx] Demo in join_histogram_essentials.sql.

The Join Histogram is simply the two histograms matched side-by-side by value (a full outer join) together with the only relevant information for the formula: VALUE, COUNTS and a flag to indicate whether the value is popular or not. Note that values 30 and 40 do not match in T2.

Contribution 1: populars matching populars

This contribution is the sum of the products of the counts of all the popular values that match a popular value in the other table:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)
POP	**4**	**10**	**2**	**POP**
UN	2	20	1	UN
POP	4	30	-	-
UN	2	40	-	-
UN	2	50	3	POP
UN	2	60	1	UN
POP	**4**	**70**	**4**	**POP**

So it is **4*2** + **4*4** = **24**

This is intuitively sound since we have seen that COUNTS for popular values is a good approximation (actually exact for FHs) of the actual number of occurrences in the table for the value, and VALUE itself is very stable, since it is unlikely to disappear from the table (and from the histogram) for small variations of the table rows. So, simply multiplying COUNTS will get a nice estimate of the number of rows in the join resultset for this value.

Contribution 2: populars not matching populars

This is the sum of the products of counts of the popular values that do not match a popular value in the other table (NB: it does not matter whether they match an unpopular value or do not match a value at all) times num_rows*density of the other table:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)
POP	4	10	2	POP
UN	2	20	1	UN
POP	**4**	**30**	-	**-**
UN	2	40	-	-
UN	**2**	**50**	**3**	**POP**
UN	2	60	1	UN
POP	4	70	4	POP

So it is **4 * num_rows(t2) *density(t2)** + **num_rows(t1) *density(t1) * 3** = **4 * 0.5** + **1 * 3** = **5**

Notice that the **counts for t1.value=50** are ignored. This is not surprising since we have seen that COUNTS for an unpopular value is an unreliable approximation of the actual number of occurrences (which is true for HB but false for FH, since for the latter it is actually the exact number), so the formula uses, instead, the more precise "average number of rows for each unpopular value" given by num_rows * density, and multiplies it with the high-quality COUNTS for the popular value.

Notice also that the CBO does not consider a match with an unpopular value (t1.value=50) any different from a non-match (t2.value=30), because VALUE is considered unstable (again, true for HB but not for FH).

Side note: the CBO makes here an assumption it makes very often, that is, that the popular value will find at least a match in the other table. This is the same reasoning made for "where value = constant"; the CBO cannot know a priori whether the client is after some rows or if it is just checking that no row exists, but since the former is generally the case, it assumes that "constant" matches a value and so estimates it will get back the average number of rows per value - which is num_rows / num_distinct for tables with no histogram, and num_rows * density for columns with histogram and "constant" not a popular value. In the case of this contribution, it applies the same reasoning, since it knows very accurately that the match in the other table is not a popular value.

Contribution 3: not-populars subtable

Here the CBO applies a variation of the standard formula to the two subtables obtained by removing the popular values, since the previous two contributions already accounted for matches "driven" by the popular values.

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)
POP	4	10	2	POP
UN	**2**	20	**1**	**UN**
POP	4	30	-	-
UN	**2**	40	-	-
UN	**2**	50	3	POP
UN	**2**	60	**1**	**UN**
POP	4	70	4	POP

num_rows_unpopulars (t1) = **2+2+2+2** = **8**
num_rows_unpopulars (t2) = **1+1** = **2**

Notice that even if COUNTS is not useful to estimate the number of occurrences for VALUE, it gives exactly the number of occurrences of unpopular values contained between the current (inclusive) and the previous (exclusive) endpoint, so num_rows_unpopulars is actually a very good estimation (you can check it on the original tables given above) of the actual number of not-populars value in the table.

Then, the CBO uses this formal rewrite[xxxi] of the standard formula

num_rows(a) * num_rows (b) / max (num_distinct(a), num_distinct(b)) =
num_rows(a) * num_rows (b) * min (1 / num_distinct(a), 1 / num_distinct(b)) =
num_rows(a) * num_rows (b) * min (density(a), density (b))

that applied to the two unpopular subtables gives

num_rows_unpopulars (t1) * num_rows_unpopulars (t2) * min (density (t1), density (t2))=
= **8** * **2** * min (**0.05**, **0.045454545**) = **0.72727272**

Contribution 4: special cardinality

This is a very strange contribution that it is much likely simply a bug. We will discuss it in the details section only; for cases such as the one presented (tables matching on their max value), the special cardinality is always zero.

Check against the CBO estimate

The CBO cardinality estimate for the example presented is 30:

```
SQL> select count(*)
  2    from t1, t2
  3   where t1.value = t2.value;

-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |     6 |     9  (12)| 00:00:01 |
|   1 |  SORT AGGREGATE     |      |     1 |     6 |            |          |
|*  2 |   HASH JOIN         |      |    30 |   180 |     9  (12)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T2   |    11 |    33 |     4   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T1   |    20 |    60 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

The sum of the four contributions is, in fact, 24 + 5 + 0.72727272 + 0 = 29.72727272, rounded to 30, as required.

[xxxi] See the "num_rows * density" section for further discussions on this formula.

The Formula In Full

In "Essentials of the formula" we have considered the case of perfectly overlapping ranges, with matching values on the min and max value of both tables. In the general case, the formula considers, fundamentally, the histogram values on the "common range" only and applies the same reasoning we have seen - but the selection of the "common range" is made in a very peculiar way, following some not very intuitive rules (some of them may be bugs). So, let's revisit the formula and show its complete and precise definition using a complete example[xxxii].

The join histograms: full, chopped and chopped_plus_2

Given these tables with their respective histograms (the left one an HB, the right one a FH):

20 <=
20 <=
20.5
40 <=
40.5
50 <=
50.5
60 <=
70
70 <=
70
70 <=

The HB of table t1 on the left (N=6):

VALUE	EP	COUNTS
20	1	2
40	2	2
50	3	2
60	4	2
70	6	4

num_rows (t1) = 12
density (t1) = .104166667
num_rows(t1) *density(t1) = 1.25

The FH of table t2 on the right:

VALUE	EP	COUNTS
10	1	1
30	3	2
50	4	1
60	8	4
70	10	2
80	12	2
90	13	1
99	14	1

num_rows (t2) = 14
density (t2) = .035714286
num_rows(t2) *density(t2) = 0.5
.

10
30
30
50
60
60
60
60
70
70
80
80
90
99

The full Join Histogram is, as defined in the "Essentials" chapter:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	10	1	UN			
UN	2	20	-	-			
-	-	30	2	POP			
UN	2	40	-	-			
UN	**2**	**50**	**1**	**UN**	**minMV**	CJH	CJH_PLUS_2
UN	**2**	**60**	**4**	**POP**		CJH	CJH_PLUS_2
POP	**4**	**70**	**2**	**POP**	**minmax**, maxMV	CJH	CJH_PLUS_2
-	**-**	**80**	**2**	**POP**			CJH_PLUS_2
-	**-**	**90**	**1**	**UN**			CJH_PLUS_2
-	-	99	1	UN	maxmax		

Letting
minmax (min of the max values) = min (max (t1.value), max (t2.value)) = 70
maxmax (max of the max values) = max (max (t1.value), max (t2.value)) = 99
minMV (min Matching Value) = min ({value | t1.value = t2.value}) = 50
maxMV (max Matching Value) = max ({value | t1.value = t2.value}) = 70

The Chopped Join Histogram (CJH), that is, the only values that are considered by most of the contributors, is given by the subjoinhistogram defined by minMV <= value <= minmax, highlighted in **red**.

[xxxii] Demo in join_histogram_complete.sql.

Notice the strange asymmetry: the low extreme of the range is defined by the first value that matches, the high extreme by the last value that the tables have in common.

Side note: I find this rather curious; I do not see any reason to treat "low" values differently from "high" values, since the join resultset does not depend on the ordering of the values, only on their matching or not in the other table (and in fact the values themselves, and thus their ordering, are frequently "random" ones for columns meant to be used in join predicates - think "synthetic keys"). The "Detecting peaks" example below also shows how the definition of the low extreme can make for less accurate, and more unstable, estimates.

The Chopped Join Histogram Plus 2 (CJH_PLUS_2) contains the values that are considered by one contributor, and is defined by CHJ plus the two rows which follow immediately (highlighted in **blue**). Since there is nothing magic in the value 2, this is most likely a bug - probably a pointer or a loop overshooting by 2.

All other values are ignored.

Let's revisit the four contributors, and complete their definition. Please check the comments in the "Essentials of the formula" chapter if you are interested in my interpretation of the "physical meaning" of the contributors.

Contribution 1: populars matching populars

This contribution is the sum of the products of the counts of all the popular values in the CJH that match a popular value in the other table:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	10	1	UN			
UN	2	20	-	-			
-	-	30	2	POP			
UN	2	40	-	-			
UN	2	50	1	UN	minMV	**CJH**	CJH_PLUS_2
UN	2	60	4	POP		**CJH**	CJH_PLUS_2
POP	**4**	**70**	**2**	**POP**	minmax, maxMV	**CJH**	CJH_PLUS_2
-	-	80	2	POP			CJH_PLUS_2
-	-	90	1	UN			CJH_PLUS_2
-	-	99	1	UN	maxmax		

So here it is simply **4*2** = **8**.

Contribution 2: populars not matching populars

This is the sum of the products of the counts of the popular values in the CJH that do not match a popular value in the other table (NB: it does not matter whether they match an unpopular value or do not match a value at all) times num_rows*density of the other table:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	10	1	UN			
UN	2	20	-	-			
-	-	30	2	POP			
UN	2	40	-	-			
UN	2	50	1	UN	minMV	**CJH**	CJH_PLUS_2
UN	2	**60**	**4**	**POP**		**CJH**	CJH_PLUS_2
POP	4	70	2	POP	minmax, maxMV	**CJH**	CJH_PLUS_2
-	-	80	2	POP			CJH_PLUS_2
-	-	90	1	UN			CJH_PLUS_2
-	-	99	1	UN	maxmax		

Here is **num_rows(t1) *density(t1) * 4 = 1.25 * 4 = 5**.

Contribution 3: not-populars subtable

Here the CBO applies the formal rewrite of the standard formula we have seen in the "Essentials" chapter to the two subtables obtained by removing the popular values from **CJH_PLUS_2** and *that are not equal to minMV*:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	10	1	UN			
UN	2	20	-	-			
-	-	30	2	POP			
UN	2	40	-	-			
UN	*2*	*50*	*1*	*UN*	minMV	CJH	**CJH_PLUS_2**
UN	**2**	60	4	POP		CJH	**CJH_PLUS_2**
POP	4	70	2	POP	minmax, maxMV	CJH	**CJH_PLUS_2**
-	-	80	2	POP			**CJH_PLUS_2**
-	-	90	**1**	**UN**			**CJH_PLUS_2**
-	-	99	1	UN	maxmax		

num_rows_unpopulars (t1) = **2**
num_rows_unpopulars (t2) = **1**

The variation of the formula is slightly more complicated than the one I presented in the "Essentials" chapter, since it provides a default value in case some table has an empty not-populars subtable:

```
   decode ( num_rows_unpopular (t1), 0, num_rows(t1) / max_ep(t1), num_rows_unpopular (t1) )
 * decode ( num_rows_unpopular (t2), 0, num_rows(t2) / max_ep(t2), num_rows_unpopular (t2) )
 * min ( density (t1) , density (t2) ) =
```

2 * 1 * min (0.104166667, 0.035714286) = 2 * 1 * 0.035714286 = 0.071428572

Contribution 4: special cardinality

I frankly think that this contribution is a mere bug, since it seems to me to have not any "physical meaning" and especially since it overcounts a popular value whose counts have already contributed to other contributors.

It is different from zero only if maxMV = minmax and minmax < maxmax, that is, if the max matching value is the last in one table and is popular, and there are other values in the other table beyond the common max matching value.

In pseudocode:

```
if maxMV = minmax and minmax < maxmax then

   if max (t1) = maxmax
     decode (h2_popularity (value = minmax),
           1, h2_counts  (value = minmax) * num_rows(t1) * density (t1),
           0)
   else /* max (t2) = maxmax */
     decode (h1_popularity (value = minmax),
           1, h1_counts  (value = minmax) * num_rows(t2) * density (t2),
           0)
   end if;
else
   0
end if;
```

That is, the max value for the "shorter" table is considered a "popular not matching popular", and it does not matter whether the value that matches in the other table is popular or not.

In our example:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	10	1	UN			
UN	2	20	-	-			
-	-	30	2	POP			
UN	2	40	-	-			
UN	2	50	1	UN	minMV	CJH	CJH_PLUS_2
UN	2	60	4	POP		CJH	CJH_PLUS_2
POP	**4**	70	2	POP	**minmax, maxMV**	CJH	CJH_PLUS_2
-	-	80	2	POP			CJH_PLUS_2
-	-	90	1	UN			CJH_PLUS_2
-	-	99	1	UN	**maxmax**		

We have that minmax = maxMV = 70 and minmax < maxmax; max (t2) = maxmax; the value in t1 for value=70 is indeed popular; so the contribution is **4 * num_rows(t2) * density (t2)** = **4 * 0.5** = **2**.

Notice that this is *in addition* to the "popular matching popular" contribution of value=70 in table t1 (so the value gets counted more the one time), and that it can be far from being negligible.

Check against the CBO estimate

The CBO cardinality estimate for the example presented is 16:

```
SQL> select count(*)
  2    from t1, t2
  3   where t1.value = t2.value;

-------------------------------------------------------------------------------
| Id  | Operation           | Name | Rows  | Bytes | Cost (%CPU)| Time     |
-------------------------------------------------------------------------------
|   0 | SELECT STATEMENT    |      |     1 |     6 |     9  (12)| 00:00:01 |
|   1 |  SORT AGGREGATE     |      |     1 |     6 |            |          |
|*  2 |   HASH JOIN         |      |    16 |    96 |     9  (12)| 00:00:01 |
|   3 |    TABLE ACCESS FULL| T1   |    12 |    36 |     4   (0)| 00:00:01 |
|   4 |    TABLE ACCESS FULL| T2   |    14 |    42 |     4   (0)| 00:00:01 |
-------------------------------------------------------------------------------
```

The sum of the four contributions is, in fact, 8 + 5 + 0.071428572 + 2 = 15.071428572, rounded to 16, as required.

Fallback to the standard formula

Even when histograms are in place on both tables, the CBO can ignore them and revert to the standard formula; in essence, it does this when the histograms, loosely speaking, do not "match enough".

There are actually two incarnations of the formula (check [Lewis, CBO, page 278][xxxiii]) that differ only in their ability to detect overlapping ranges: the "8i" variation, that simply returns num_rows(t1) * num_rows (t2) / max (num_distinct (t1), num_distinct (t2)) even when the two ranges do not overlap at all, and the "9i/10g" one that detects completely nonoverlapping ranges (but not partially overlapping ranges) and returns 0 (rounded to 1) in this case.

These are the sufficient conditions for the fallback (most of them were discovered by Wolfgang Breitling as I recall):

[xxxiii] By the way, the paper's formula explains the experiments Jonathan performed starting from page 280; see the section "overlaps, and improvements".

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

(b) let maxPV (max Popular Value) =
max value in CJH where at least one value is popular

if maxMV is null (no matching value)
or maxPV is null (no popular value)[xxxiv]
or maxPV < minMV (all popular values < min 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 lowest matching value, or of course, if no popular or matching value exist."

Additionally, if the formula produces a result of exactly zero (something you may observe only if you set density=0 manually probably) a sanity check kicks in and the standard formula of 9i/10g is used instead.
If the formula result is close to zero but not zero, normal rounding to 1 applies.

[xxxiv] This was observed by Jonathan Lewis in [Lewis, CBO, page 281-283] : "... and the critical feature was that one of [the histograms] ... did show a popular value."

The Formula in Action

Let's see the formula at work in real-life (yet simplified to make them manageable) scenarios, and how knowing the formula enables you to prevent, or at least diagnose, possible pitfalls.

Frequency Histograms and the "mystery of halving"

If you build a Frequency Histogram on a column with all distinct values (each value occurring exactly once), the CBO estimate of the join cardinality will be systematically[xxxv] **half** the real cardinality.

The fh_and_mystery_of_halving.sql demo builds table t2 with all distinct values, table t1 with some matching values, compute histograms, and join. The real cardinality is 500, the CBO estimated one is 251. Here's why:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
POP	**73**	10	**1**	**UN**	minMV	CJH	CJH_PLUS_2
POP	**61**	20	**1**	**UN**		CJH	CJH_PLUS_2
POP	**97**	30	**1**	**UN**		CJH	CJH_PLUS_2
POP	**82**	40	**1**	**UN**		CJH	CJH_PLUS_2
POP	**91**	50	**1**	**UN**		CJH	CJH_PLUS_2
-	-	60	**1**	**UN**		CJH	CJH_PLUS_2
POP	**96**	70	**1**	**UN**	minmax,maxmax, maxMV	CJH	CJH_PLUS_2
							CJH_PLUS_2

populars matching populars	0	
populars not matching populars	**250**	**73*0.5 + 61*0.5 + 97*0.5 + 82*0.5 + 91*0.5 + 96*0.5**
not-populars subtable	**0.01**	
special cardinality	0	
TOTAL (rounded)	250.01 (**251**)	

This is because the CBO considers values in FHs with COUNTS=1 as unpopular, so they are credited against the "populars not matching populars" contributor with num_rows*density = 0.5 instead of their exact cardinality (1).

This is wildly counterintuitive since when we have FHs on both columns, the histograms contain all the information needed to compute an exact estimation (they perfectly represent the "map" we spoke about in the introduction), and so we may expect the CBO to leverage it - but instead, it makes a systematic 50% underestimation. It would be simple to correct it - all it would take is the CBO considering all values in FHs as popular, regardless of their counts - which is statistically sound since, as we saw, the statistical properties of all values in FHs are the same as popular values in HBs.

So, if you have columns with unique values (or many values that occur only once), watch out for this issue; computing a FH on them may be dangerous, not simply useless as one would expect intuitively.

It is interesting to note that if you have a FK between the two tables (so a unique/primary key constraint on the table with unique values), the scenario is the one discussed in Appendix A, and so the standard formula will give you the **exact** cardinality - and the CBO will use the standard formula if you remove the histogram on the parent (but you can keep the one on the child if needed for other reasons). This is also checked in the demo.

Most probable scenario in practice: you have an application that uses lookup tables (aka decoding tables) extensively to map internal identifiers to external ones (e.g. status_id to "new", "accepted", "rejected" to be displayed on screen, etc). In this case, you will get a halving for every lookup table you join (for three lookup tables = 1/2 * 1/2 * 1/2 = 1/8, you miss 7/8th of the real cardinality). Check the demo for the two tables scenario.

[xxxv] I have assumed a FH on the other table as well, which should be the most occurring scenario in this case; if the histogram on the other table is an HB, the correct statement should be "each popular value will contribute with half its counts to the total cardinality".

A way to dramatically lessen the probability of computing a FH on the key column of lookup tables by mistake is to use SIZE SKEWONLY and SIZE AUTO instead of the usual SIZE 254 when computing a FH. Wolfgang Breitling has explored this extensively in his paper "*Joins, Skew and Histograms*" [Breitling, JSH] so I point you there for further informations - to summarize his findings, SIZE SKEWONLY will not compute histograms on unique columns (unless there are very large gaps in the range of values, something that is quite unlikely to happen in practice) and SIZE AUTO, additionally, will not compute histograms unless the column has been used in a range predicate, something that is unlikely to have happened for the key column of lookup tables.

Detecting peaks

One reason to use histograms is to tell the CBO about the location of "peaks", i.e. frequently occurring values that make an otherwise (almost) uniform distribution skewed, thus violating the default assumption by the CBO of data uniformity. It works reasonably well for predicates such as "where value=constant" (provided that the peaks are wide enough to be detected by the sampling of course), and looking at the join formula, it seems that the "not-popular subtable" contributor has been designed exactly for that purpose. Does it work (reasonably) well in practice ? The answer is yes, but with a pitfall.

The demo detecting_peaks.sql builds two tables that have unique values in the range 0..79, and a peak of 20 values at the end (value=9998 for t1 and 9999 for t2). To model the case of many distinct values, thus exceeding the max value of 254 for SIZE N, I have collected histograms with SIZE 13 for t1 and SIZE 15 for t2, thus making most of the values in histograms not matching, as it is almost always the case in practice for HBs.

The real cardinality is, of course, 80, and without histograms, the CBO estimates a cardinality of 123. With the HBs in place, the CBO does detect the peaks, and makes a much better estimate - let's see the Join Histogram:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
UN	0	0	0	UN	minMV	CJH	CJH_PLUS_2
-	-	6	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	7	-	-		CJH	CJH_PLUS_2
-	-	13	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	15	-	-		CJH	CJH_PLUS_2
-	-	20	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	23	-	-		CJH	CJH_PLUS_2
-	-	27	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	31	-	-		CJH	CJH_PLUS_2
-	-	34	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	39	-	-		CJH	CJH_PLUS_2
-	-	41	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	47	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	54	**6.67**	**UN**	maxMV	CJH	CJH_PLUS_2
-	-	60	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	62	-	-		CJH	CJH_PLUS_2
-	-	67	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	69	-	-		CJH	CJH_PLUS_2
-	-	73	**6.67**	**UN**		CJH	CJH_PLUS_2
UN	**7.69**	77	-	-		CJH	CJH_PLUS_2
POP	**23.08**	9998	-	-	minmax	CJH	CJH_PLUS_2
-	-	9999	26.67	POP	maxmax		CJH_PLUS_2

populars matching populars	0	
populars not matching populars	**23.08**	**23.08 * num_rows(t2)*density(t2)**
not-populars subtable	**56.41**	**(7.69*10) * (6.67*11) * min (0.01, 0.01)**
special cardinality	0	
TOTAL (rounded)	**79.49(80)**	

It works very well, actually the real cardinality (80) is matched perfectly is this case.

But now, a seemingly innocent modification, a delete of a single row (the first, say it was archived), ruins it all:

```
delete from t1 where value = 0;
```

The Join Histogram becomes:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
-	-	0	0	UN			
UN	.00	1	-	-			
-	-	6	6.67	UN			
UN	7.62	8	-	-			
-	-	13	6.67	UN			
UN	7.62	16	-	-			
-	-	20	6.67	UN			
UN	7.62	24	-	-			
-	-	27	6.67	UN			
UN	7.62	32	-	-			
-	-	34	6.67	UN			
UN	7.62	39	-	-			
-	-	41	6.67	UN			
UN	7.62	47	6.67	UN	minMV	CJH	CJH_PLUS_2
UN	**7.62**	54	**6.67**	UN	maxMV	CJH	CJH_PLUS_2
-	**-**	60	**6.67**	UN		CJH	CJH_PLUS_2
UN	**7.62**	62	**-**	-		CJH	CJH_PLUS_2
-	**-**	67	**6.67**	UN		CJH	CJH_PLUS_2
UN	**7.62**	69	**-**	-		CJH	CJH_PLUS_2
-	**-**	73	**6.67**	UN		CJH	CJH_PLUS_2
UN	**7.62**	77	-	-		CJH	CJH_PLUS_2
POP	**22.85**	9998	-	-	minmax	CJH	CJH_PLUS_2
-	**-**	9999	26.67	POP	maxmax		CJH_PLUS_2

populars matching populars	0	
populars not matching populars	**22.85**	**22.85 * num_rows(t2)*density(t2)**
not-populars subtable	**8.12**	**(7.62*4) * (6.67*4) * min (.01010101, 0.01)**
special cardinality	0	
TOTAL (rounded)	**30.97 (31)**[xxxvi]	

So from an excellent exact match, we are down to 31 against a real cardinality of 79.

This is due to the peculiar definition of the lower bound of the CJH, that is, the first matching value; it used to be 0, now it is 47, so losing a lot of values. Had the definition been the same as the upper bound (the intuitive max of the min values), "not-populars subtable" would have been (7.62*10) * (6.67*11) * min (.01010101, 0.01) = 55.91, for a total of 78.75 - exact.

This issue is very likely to occur in practice, since in general, of course, even if the tables are almost identical, it is very unlikely that unpopular values will match exactly, so the first matching value is likely to be well inside the Join Histogram (if it exists at all). Sure, the first value in the table, which is always represented in the histogram, has an higher probability of matching but all it takes is to have tables that does not match in the first value, or whose statistics have been collected at slightly different times, to make it different as well. There is not a real solution to this, unless you manually edit the histograms and add a first common value (but you must do this for all tables joined to a given table...) - short of modifying the CBO code of course.

[xxxvi] Actually the CBO estimates 32 - probably it rounds the two cardinalities separately, then adds.

The Perils of the Special Cardinality

The strange "Special Cardinality" contributor can make the CBO overestimate the last values in the tables.

The perils_of_special_cardinality.sql demo builds two tables with the following data layout:

```
SQL> select value, count(*) from t1 group by value order by value;

VALUE   COUNT(*)
----- ----------
   10          2
   20        100

SQL> select value, count(*) from t2 group by value order by value;

VALUE   COUNT(*)
----- ----------
    0        100
   20          2
```

The real cardinality is 200, and without histograms the estimated cardinality is 5202, so it makes sense to collect (frequency) histograms to tell the CBO both the location of the peak for t1.value=20 and that the peak for t2.value=0 is out of the common range. This makes for an excellent (off by only one) cardinality estimate:

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
- POP	- 2	0 10	100 -	POP -			
POP	**100**	20	**2**	**POP**	minMV,minmax, maxmax, maxMV	CJH	CJH_PLUS_2
							CJH_PLUS_2

populars matching populars	**200**
populars not matching populars	0
not-populars subtable	**0.0049**
special cardinality	0
TOTAL (rounded)	**200.0049 (201)**

Now, if you insert a single row above the max value, a modification that does not change the real cardinality of course:

```
insert into t2(value) values (99);
```

POP(T1)	COUNTS(T1)	VALUE	COUNTS(T2)	POP(T2)			
- POP	- 2	0 10	100 -	POP -			
POP	**100**	20	**2**	**POP**	minMV,minmax, maxMV	CJH	CJH_PLUS_2
-	-	99	1	**UN**	maxmax		CJH_PLUS_2

populars matching populars	**200**	
populars not matching populars	0	
not-populars subtable	**0.0048**	
special cardinality	**50.00**	**counts(t1, value=20) * num_rows(t2)*density(t2)= 100*0.5**
TOTAL (rounded)	**250.0048 (251)**	

The special cardinality kicks in and ruins the perfect estimation we saw before. In this case the last value of table t1 (20) gets overcounted by "only" 50% since we have a FH on t2 and so num_rows(t2)*density(t2)=0.5; in general of course num_rows(t2)*density(t2) could be much greater than one if you collect an HB.

Overlaps, and improvements

Jonathan Lewis investigated in his book "Cost Based Oracle" (see [Lewis, CBO, pages 278-283]) what happens to the CBO join cardinality estimate when the two joined columns do not overlap perfectly on their low/high value range.

Jonathan builds one table containing the values 0..99, with a random quasi-uniform distribution (or if you prefer, statistically uniform distribution) and another table with the values 0+offset .. 99+offset and the same distribution, and then compares the CBO cardinality estimate versus the real cardinality, for different values of the offset.

Let's recap his findings (we will discuss the 10g case only for simplicity - results ought to be similar for 9i), explain them using the knowledge about the join cardinality formula we presented in this paper, and show how the formula, once improved, could give exceptional results.

With no histogram, the CBO essentially ignores the amount of overlapping and always applies the standard formula (it returns 1 only when the two ranges are completely disjunct). This is reasonably sound: the CBO does not know the value distribution inside the range, so it cannot know whether the common subrange contains few, some, most or all the values, and assumes that all values match.

Jonathan then investigates using Histograms to tell the CBO about the value distribution inside the two ranges (and so, over the common subrange), using different bucket size (SIZE N) for the two tables, and both HB and FH.

I have repeated his experiments[xxxvii] (organizing them differently but on the same data sets), and measured the statistical properties of the percentual absolute estimate error, defined as

error = 100 * abs (estimated_cardinality - real_cardinality) / real_cardinality

With offset in (50,60,70,90) and SIZE in 75..90 (for both tables, so Height-Balanced), 10.2.0.3:

avg	stddev	max
363%	303%	929%

The reason for the bad estimate is that the CBO almost always (1232 times out of 1280) fallbacks to the standard formula, and so produces an estimate of 1,000,000, very distant from the real cardinality.

In fact Jonathan reports that the fallback happens unless "at least one histogram shows a popular value". Since the data distribution has E[counts(v)] = constant = 100, and a standard deviation of approx 10, the probability to have a popular value is very low (e.g. with SIZE 90 only values with counts > approx 111 (10,000/90) can be detected as popular), and even then only if at least one popular value is contained in the common range (same range of the Chopped Join Histogram for this scenario) the CBO will not fallback. Almost impossible, as the results show.

The fallback is unfortunate, since even without popular values, the formula "not-populars subtable" contributor seems perfect to estimate the cardinality with great accuracy - it is in essence the standard formula applied to the common subrange, and on the common subrange our data distribution does fit statistically very well the requirements that would give an exact result in a deterministic scenario (see Appendix A: uniform distributions, principle of inclusion, etc).

The beauty of having implemented the formula in PL/SQL is that I can change it to explore new territories easily. So, in join_over_histograms_improved.sql I removed the fallback (and also, even if the change is negligible on this data[xxxviii], I have made the Chopped Join Histogram low extreme definition consistent with the high one, and got rid of both the special cardinality and the Chopped Join Histogram Plus 2); on the same data:

[xxxvii] Demo overlaps.sql.
[xxxviii] But not negligible in the general case of partial overlaps, especially if the HB histogram values do not match, which is very likely as discussed in the "Detecting peaks" section.

avg	stddev	max
69.5%	14.4%	91.1%

Definitely much better, but why the residual error of 69.5% ? I would have expected much finer accuracy over such "very uniform" distributions. So, I decided to check my reasonings over the meaning of the "not-populars subtable" contributor (they are presented at the bottom of the "num_rows * density" section above) and investigated changing it from

num_rows_unpopulars (t1) * num_rows_unpopulars (t2) * min (density (t1), density (t2))

to

num_rows_unpopulars (t1) * num_rows_unpopulars (t2) * min (
(**num_rows (t1) / num_rows_unpopulars (t1)**) * density (t1),
(**num_rows (t2) / num_rows_unpopulars (t2)**) * density (t2)
)

with this change, we get

avg	stddev	max
2.33%	1.92%	10.2%

which makes for an exceptional accuracy.

Side note: I have also investigated Jonathan's main scenario, a FH histogram on one table (the lhs) and an HB on the other (also in the demo). The results show a slight improvement in accuracy:

	avg	stddev	max
CBO	6.72%	4.35%	15.3%
all changes minus "not popular subtables"	2.44%	1.37%	4.55%
all changes	2.48%	1.40%	4.58%

In these test runs the CBO has never fallen back (since all FH histogram values are popular), the improvements are due mainly to the removal of the "special cardinality".
In the perfect case of FH on both tables[xxxix], we start from an already excellent accuracy; the changes made it even better, that makes no practical difference but confirms that the proposed changes are sound:

	avg	stddev	max	abs($card_e$ - $card_r$)
CBO	0.0234%	0.0151%	0.0610%	60
all changes minus "not popular subtables"	2.28E-08%	1.45E-08%	5.29E-08%	.00005
all changes	0.000228%	0.000145%	0.000529%	.5

the percentual error is so small that I have added the last column, that shows
max (abs (estimated_cardinality - real_cardinality))
to appreciate the differences from another perspective.

Note: of course here all FH values are popular (counts > 1); beware the "mystery of halving" for FHs that have many or all unpopular values (counts = 1).

[xxxix] To reproduce, uncomment "for repeat in 1..50 loop" in the demo.

Supporting Code

All the examples presented in the paper have been produced using repeatable scripts, named in the discussion, and of course they are available as supporting code. In addition, I am sharing also the "tools" I have developed and used to help my investigation efforts (all are -I hope- properly documented inline):

cbo_cardinality.sql	Retrieves the CBO estimate for the join from v$sql_plan.
set_histo.sql	Sets histograms on columns.
join_over_histograms.sql	Implements the paper formula as a PL/SQL package, reading statistics from the data dictionary views. Very useful when diagnosing a bad cardinality estimate.
exhaustive_topological.sql	Calculates the difference between the output of the formula (from join_over_histograms.sql) and the CBO output (from cbo_cardinality.sql) over the set of "all" possible scenarios.

The last tool has never produced an error greater than 1.5 on the tens of thousands of scenarios I have checked; this means that the formula is exact, when you consider that the CBO rounds *intermediate* calculations[xl] and the formula does not.

JoinOverHistograms.txt is a leaflet summarizing the formula.

Bibliography

[Breitling, HMF]	Wolfgang Breitling, *Histograms - Myths and Facts*, Hotsos Conference 2005; available on www.centrexcc.com
[Breitling, JSH]	Wolfgang Breitling, *Joins, Skew and Histograms*, Hotsos Conference 2007; available on www.centrexcc.com.
[Lewis, CBO]	Jonathan Lewis, *Cost Based Oracle: Fundamentals*, Apress, 2006, ISBN 978-1590596364.

Appendices

Appendix A - derivation of the standard formula

It is usually stated as

If the distribution of each table is perfectly uniform [counts(value(i)) = constant = num_rows / num_distinct] and all the values in the table with the smaller num_distinct match in the other table ("principle of inclusion", see [Breitling, JSH]), the exact cardinality is given by the *standard formula*[xli]:

num_rows(t1) * num_rows (t2) / max (num_distinct (t1), num_distinct (t2))

[xl] My guess is that the "not-populars subtable" contribution is always rounded up, and then added with the rounded sum of all the others - but I have not digged too much since the rounding strategy is not really important, it changes the result only slightly.

[xli] I have simplified the standard formula assuming that the tables contain no null values.

But we will prove and illustrate the formula using this less-restrictive statement:

If the distribution *of the table with the smaller num_distinct* is perfectly uniform [counts(value(i)) = constant = num_rows / num_distinct] and all the values in the table with the smaller num_distinct match in the other table ("principle of inclusion", see [Breitling, JSH]), the exact cardinality is given by the *standard formula*[xlii]:

num_rows(t1) * num_rows (t2) / max (num_distinct (t1), num_distinct (t2))

i (bucket #)	counts(t1,i)	table t1	table t2	counts(t2,i)
1	1	1	1 1	2
2	4	2 2 2 2	2 2	2
3	2	3 3	3 3	2
4			99 99	2

num_distinct (t1) = 3
num_distinct (t2) = 4

Table t1 is the one with the smaller num_distinct.

All values in t1 match in t2.

Let k = counts (t2,i) = constant = num_rows(t2) / num_distinct (t2)

$$\text{cardinality} = \sum_i \text{counts}(t2,i) * \text{counts}(t1,i) = k * \sum_i \text{counts}(t1,i) = k * \text{num\_rows}(t1)$$

From num_distinct (t2) >= num_distinct (t1), follows num_distinct (t2) = max (num_distinct (t2) , num_distinct (t1)), so k = num_rows(t2) / max (num_distinct (t2) , num_distinct (t1)), hence the formula.

Observation: the hypotheses both hold if t2 has a unique or PK constraint, and a FK exists from t2 to t1 (in this case, k=1). So the standard formula always gives the exact result when a FK exists between the two joined columns.

Paper version: 1.3, 2007-04-25

[xlii] I have simplified the standard formula assuming that the tables contain no null values.