SELECT WITHOUT REPLACEMENT

Alberto Dell'Era
alberto.dellera@gmail.com

ABSTRACT

We will show how the Oracle™ Cost Based Optimizer uses pervasively a fundamental statistical formula regarding "selection without replacement" for cardinality estimation, and we will concentrate especially on the algorithm for multicolumn joins.

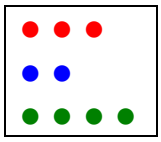
Acknowledgements

I wish to thank Jonathan Lewis for his book "Cost Based Oracle", a book that has immensely improved my understanding of the CBO and that has provided some intriguing puzzles whose investigation has eventually led to this paper.

Introduction

Selection from an urn without replacement

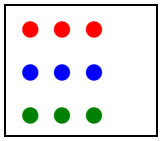
Consider the following urn (or box, bag), that contains three red balls, two blue and four green ones, for a total of 9 balls and three distinct (different) colors:



Now pick "s" balls at random without replacement (that is, you discard each ball as you pick it - the alternative would be to put it again inside the urn and it is called *selection with re-placement*); how many distinct (different) colors can you expect to find on average in your selection of s balls ?

To illustrate, say s=3. In one run you may get all of the three red balls (1 distinct color), in another two blue and one green (2 distinct colors), and in yet another one red, one blue and one green (3 distinct colors); in this case you would say that you got (1 + 2 + 3) / 3 = 2.0 distinct colors on average. What would be the average over all possible ways of drawing three balls, i.e., as a statistician would put it, the expected value? It is 2.22619 with six-digit precision.

A very important special case (that happens to be the one considered by the Cost Based Optimizer) is an urn containing the same number of balls for each color (a perfectly uniform distribution):



In this case the formula becomes much simpler and quicker to calculate, and it is the only one you can usually assume if you know only the total number of balls and the number of distinct colors, as it is often the case in practice. If you allow for some colors to have one more ball (I call this a weakly uniform distribution) the formula is slightly more complex but can cover the case of a non integer ratio of number of balls / number of distinct colors.

For the formulae (with statistical proofs and brute-force validations) see [Dell'Era, 2005]; [Yao, 1977] considers only the perfectly uniform case. The CBO uses its own formula which seems to be very close to Yao's one and also very accurately approximating the formula I derived for the weakly uniform distribution (that gives the exact statistical answer).

Select from a table without replacement

Consider the following SQL statement:

```
select x
  from t
 where filter = 42;
```

The statement filters some rows from table T using a filter predicate (I've used an equality predicate on the FILTER column for illustration purposes, but the predicate could be anything) and then returns all the values of another column X from the filtered result sets.

What is the expected number of distinct values of the final result set, a figure of paramount importance for the CBO calculations aimed at finding the best plan for any statement that contains some filtering operations?

If one assumes (as the CBO does in general) that column X and FILTER are non-correlated, this statistical problem is exactly the same as a selection from an urn without replacement. You can think to have num_rows (T) balls with num_distinct (X) different colors in the urn (table), each with a label attached that shows a number (FILTER), each label attached previously by a girl at random without caring for the ball colors (this is the non-correlation[1] assumption) - now you are asked to get all the s balls whose label shows "42". You can spare the girl her work and fetch yourself s rows at random instead and you will get the same statistical answer.

So the filter operator takes a row set as input (those contained in table T in this case, but T could be the output of another operator in general, for example an in-line view) and produces a *filtered* row set as output; the input has a certain cardinality and the output another, usually named *filtered cardinality*. We can extend this concept and terminology and say as well that each input column has a distinct cardinality and each output column another *filtered distinct cardinality*.

In this paper, we will use the "f_" prefix to distinguish between the input and output statistics. So if table T has a cardinality of num_rows(T) and column X a distinct cardinality of num_distinct (T.X), the output has a filtered cardinality of f_num_rows(T) and column X a filtered distinct cardinality of f_num_distinct(T.X).

In formulae

```
f_num_rows(T)       = selectivity (filter operator) * num_rows(T)
f_num_distinct(T.X) = SWR ( distribution of distinct values of (T.X), f_num_rows(T) )
```

Where SWR() is the general formula for a Selection Without Replacement. If we consider (as the CBO does) only the case of a (weakly) uniform distribution, we can describe completely the distribution by using the two numbers num_rows(T) and num_distinct (T.X), and use the simpler formula SWRU[2]:

```
output num of distinct values = SWRU ( input num of distinct values,
                                       num values selected,
                                       input num values)

f_num_distinct(T.X) = SWRU ( num_distinct(T.X), f_num_rows(T), num_rows(T) )
```

Appendix A plots and discusses briefly the layout of the SWRU formula.

Distinct and group by

Let's see the formula in action and start with the simplest (and very frequent in practice) case possible:

```
select distinct x          select x, count(*), ...
  from t                     from t
 where filter = ...;        where filter = ...
                            group by x;
```

Both the DISTINCT and GROUP BY operators take as input the output of the filtering operation we saw in the previous chapter, and of course the estimated cardinality of their output has to be f_num_distinct (T.X), so these statements enable us to open up the CBO box and use explain plan to look directly at how the CBO calculates f_num_distinct (T.X). We need to consider only the DISTINCT case since the CBO considers it a special case of the GROUP BY operator, hence the computations are the same for both.

Script distinct_example_single_column.sql builds table T with num_rows(T) = 10000, num_distinct(T.X) = 70; the CBO computes[3] the following plan:

[1] actually independence which is a stronger requirement than non-correlation, but non-correlation is enough to claim the equivalence of the two statistical problems. The difference doesn't matter much in practice by the way.
[2] Both SWR and SWRU are implemented in PL/SQL in script DistinctBallsPlSql.sql, borrowed from [Dell'Era, 2005]. Let me stress that these formulae give the *exact* statistical answer, they are not approximations (beside numerical errors for arithmetic on the NUMBER datatype, of course, but I have tested those in [Dell'Era, 2005] for IEEE 754 arithmetic and they seem to be negligibly small).
[3] Tested in both 10.2.0.3 and 9.2.0.8 as all of the scripts of this paper, unless otherwise noted.

```
-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |    56 |
|   1 |  HASH UNIQUE       |      |    56 |
|*  2 |   TABLE ACCESS FULL| T    |   112 |
-------------------------------------------
```

so f_num_rows (T) = 112 and f_num_distinct (T.X)=56. In fact

SWRU (70, 112, 10000) = 56.1555023, that rounded to 56 gives the observed figure.

Moreover in the 10053 trace we find:

```
SINGLE TABLE ACCESS PATH
...
Card: Original: 10000  Rounded: 112  Computed: 112.00  Non Adjusted: 112.00
...
Grouping column cardinality [         X]    56
...
GROUP BY cardinality:  56.00, TABLE cardinality:  112.00
```

(In passing, note that the trace says "GROUP BY cardinality" even if we have used DISTINCT).

To be sure to have correctly and accurately understood the CBO algorithm, I have made an exhaustive check in script distinct_exhaustive_single_column.sql. This script builds automatically a set of value distributions, fetches the CBO estimated cardinality from v$sql_plan, then puts the output and the test parameters in table RESULTS. For num_rows(T) = 250, and both num_distinct(T.X) and f_num_rows(T) ranging between 1 and num_rows(T) (a total of 62,500 different test cases), 10.2.0.3 and 9.2.0.8, Yao's formula (rounded and then set to 1 when zero) gives at most an absolute error of 1 in 100% of the cases and perfect match in 94.34% of the cases. Using the exact formula, we get an absolute error of 6 at most, less than 1 in 74.60% of the cases and less than 4 in 94.36% of the cases, for an average of 1.05 and a standard deviation of 1.54. The percentage of error has an average of 0.94% and a standard deviation of 1.31%. This is fantastic accuracy, and shows that Oracle approximates very well the true SWRU formula[4].

But does the adjustment of the number of distinct values made by the SWRU formula make for a significant increase in the cardinality estimation accuracy? Most definitely, of course provided the statistical assumptions of non-correlation hold and the filtered cardinality estimation is good. Script distinct_swru_relevancy_example.sql builds such a test case with num_rows(T)=100,000, num_distinct (T.X) = 6000 - the actual real cardinality of our SQL statement is 3428 and the plan is

```
-------------------------------------------
| Id  | Operation          | Name | Rows  |
-------------------------------------------
|   0 | SELECT STATEMENT   |      |  3477 |
|   1 |  HASH UNIQUE       |      |  3477 |
|*  2 |   TABLE ACCESS FULL| T    |  5066 |
-------------------------------------------

SWRU ( 6000, 5066, 100000) = 3476.81841
```

The percentage of error is a mere (3477 - 3428) / 3428 = 1.42% which makes for fantastic accuracy, and note that the estimate of 3477 is significantly distant from both num_distinct (T.X) = 6000 and the estimated filtered cardinality of 5066.

Using the test case in distinct_example_single_column.sql I have also checked whether the CBO could use other useful information contained in the data dictionary.

Since I have always put correlated data in the table T by design in script distinct_exhaustive_single_column.sql (this does not invalidate the previous results since I'm checking the SWRU formula against the CBO output, not against the actual cardinality of the actual result set), I could try for free putting indices on T(X, FILTER) and T(FILTER, X), to see whether the CBO could use the distinct_keys figure to estimate the coefficient of correlation. This made no difference, the CBO (until 10.2.0.3 at least) always assumes non-correlated X and FILTER columns.

[4] I would be very interested, for curiosity's sake, to take a peek at the numerical algorithm that can give such an excellent approximation and be fast at the same time. My email is on the front page.

I have also tried putting a frequency histogram and then an height-based histogram on T.X to check whether the CBO could use the general formula, but again the CBO assumes (weakly) uniform distributions and does not mine the histogram data.

Now let's consider the case of two columns (script distinct_exhaustive_mult_columns_2.sql):

```
select distinct x, y        select x, y, count(*), ...
  from t                      from t
 where filter = ...;         where filter = ...
                             group by x, y;
```

The formula (algorithm) becomes

```
f_num_distinct (T.X) = SWRU (num_distinct (T.X), f_num_rows(T), num_rows(T));
f_num_distinct (T.Y) = SWRU (num_distinct (T.Y), f_num_rows(T), num_rows(T));
f_num_distinct (T.X, T.Y) = max ( (1/sqrt(2) )  * f_num_distinct (T.X) * f_num_distinct (T.Y),
                                  f_num_rows(T)
                                )
```

An example of computation is given in script distinct_example_mult_columns_2.sql.

max (...,f_num_rows(T)) is a constraint to the physical upper bound, since of course the number of distinct values cannot be greater than the number of values (another incarnation of this concept is used for multi-column joins in 10g as a sanity check, as we will see).

The product of f_num_distinct (T.X) and f_num_distinct (T.Y) can be explained, of course, by the CBO assuming that X and Y are non-correlated; in this case all the possible combinations of distinct values are present, and the number of combinations is the product of the number of the distinct values in each column. Here the CBO first filters, and then combine the filtered distinct values.

Side note: it seems strange that the CBO does not consider also

```
SWRU ( max ( num_distinct (T.X) * num_distinct (T.Y), num_rows(T) ), f_num_rows(T), num_rows(T))
```

that is, first combine and then filter. This way it may use the distinct_keys of an index on X,Y to get a much more accurate figure instead of simply assuming non-correlation. But again, putting indices on (X,Y) or (Y,X) doesn't change the CBO output (as I've tested in 10.2.0.3 using the exhaustive check script below).

I'm at a loss at explaining the statistical meaning of the factor (1 / sqrt (2)) - it probably factors in some amount of correlation between the two columns, but why exactly (1 / sqrt (2)) - is it simply an heuristic, or has it any statistical justification ? If someone knows the answer, please let me know (my email is on the front page).

The results of the script's exhaustive check are comparable to those of the single-column one - for num_rows(T)=50 (125,000 different test cases) and using the exact formula, we get an absolute error of less than 1 in 99.87% of the cases, average of absolute error of 0.04 (standard deviation of 0.21) and an average of the percentage of error[5] of 0.43% (standard deviation of 2.97%). The slight increase in accuracy is due to the physical upper bound constraint, since the product of the filtered num_distincts tends to overshoot the filtered cardinality very often, thus cancelling rounding errors.

I've checked also the cases where num_columns > 2 using other scripts (not included in the supporting code) and the formula holds when naturally extended. The factor (1/sqrt(2)) is in general power(1/sqrt(2), num_columns-1), as already noted in [Lewis, CBO, page 391].

[5] Defined with the CBO cardinality as a denominator, since here we are checking the fitness of the algorithm against the CBO output, not the fitness against the exact formula as we did for the single-column case. Anyway in both cases, swapping the denominator changes the results only slightly, since the numerator is so small and the figures to be compared so close.

Single-column equijoin

Let's now consider the join of the results of two filtering operations, and see that the SWRU formula is, indeed, used in this case also. We'll start with a single-column equijoin since it's simpler but also because the multiple-column case behaves quite differently, with very surprising results, and so needs a dedicated chapter to illustrate.

The single-columns equijoin SQL template is

```
select t1.*, t2.*
  from t1, t2
 where t1.x = t2.x
   and t1.filter = ...
   and t2.filter = ...;
```

The formula used by the CBO is the natural extension of the *standard formula* (so named in [Lewis, CBO, page 265]), simply replacing num_distinct with f_num_distinct (that is, using SWRU):

```
f_num_distinct (T1.X) = SWRU ( num_distinct (T1.X), f_num_rows(T1), num_rows(T1) )
f_num_distinct (T2.X) = SWRU ( num_distinct (T2.X), f_num_rows(T2), num_rows(T2) )
join_sel = 1 / ceil ( max (f_num_distinct (T1.X), f_num_distinct (T2.X) )
join_card = round ( f_num_rows(T1) * f_num_rows(T2) * join_sel )
```

In passing, note that the interval of the columns values [low_value, high_value] doesn't play any role in the formula[6] (that is not the case for multi-column join predicates, as we will see), unless the intervals do not overlap at all, in which case the join cardinality is set to 1 (probably zero rounded to 1) as noted in [Lewis, CBO, page 278].

Note: there are some special cases that trigger when one or both of num_distinct equal one that I have excluded from all the paper investigations for simplicity.

As an example, let's use Jonathan Lewis' test case in chapter "Biased Joins" [Lewis, CBO, page 269], an intriguing example I investigated back in 2005 and that made me discover the SWRU formula, ultimately leading to this paper. The statement is slightly different from our template since it has a filter only on the T2 table, but this is a nice opportunity to show that the formula applies in general:

```
select t1.*, t2.*
  from t1, t2
 where t1.x = t2.x
   and t2.filter = 1;
```

Script join_example_single_column_jpl.sql (Jonathan's join_card_01a.sql adapted) builds a table with num_rows(T1) = num_rows(T2) = 10000, num_distinct(T1.X) = 30 and num_distinct(T2.X) = 40; the plan is

```
-------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost  |
-------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |   333 | 10323 |     9 |
|*  1 |  HASH JOIN         |      |   333 | 10323 |     9 |
|*  2 |   TABLE ACCESS FULL| T2   |    10 |   170 |     4 |
|   3 |   TABLE ACCESS FULL| T1   |  1000 | 14000 |     4 |
-------------------------------------------------------------
```

so we have f_num_rows(T1) = 1000 (equal to num_rows(T1) since no filter predicate is applied to T1) and f_num_rows(T2) = 10. Applying the formula:

```
f_num_distinct (T1.X) = SWRU (num_distinct (T1.X), f_num_rows (T1), num_rows (T1)) =
                        SWRU ( 30, 1000, 1000 ) = 30; (unchanged)
f_num_distinct (T2.X) = SWRU (num_distinct (T2.X), f_num_rows (T2), num_rows (T2)) =
                        SWRU ( 40,   10, 1000 ) = 8.98285634; (down from 40)
join_sel  = 1 / ceil ( max (f_num_distinct (T1.X), f_num_distinct (T2.X) ) =
          = 1 / ceil ( max ( 30, 8.98285634 ) ) = 1 / 30;
join_card = round (f_num_rows (T1) * f_num_rows (T2) * join_sel ) =
          = round ( 1000*10 * 1/30 ) = round ( 333.333333 ) = 333 (as required)
```

[6] tested in join_exhaustive_single_column.sql.

In essence, the filtering predicate on T2 has filtered out so many rows that the filtered num_distinct has dropped way below the one on the other end of the join predicate (30), and so the selectivity has picked the latter; if the CBO had not used the SWRU formula to filter the distinct values, it would have picked the former and the selectivity would have been 1 / 40.

The usual exhaustive check (join_exhaustive_single_column.sql)[7], 72,600 different test cases, using Yao's formula, gives an absolute error less than 1 in 98.72% of the cases, and perfect match in 96.40% of the cases.

As in the DISTINCT/GROUP BY case, putting indexes on (X,FILTER) and (FILTER,X) has no effect.

Collecting histograms on the joined columns does change the cardinality, but that's because the CBO completely changes the algorithm (that I have investigated with great detail in *Join Over Histograms* [Dell'Era, 2007] with significant help from Wolfgang Breitling); the standard formula corrected by SWRU is not used anymore.

Multiple-column equijoin

And now let's make the big leap to the multiple-column equijoin, and its two main surprises: that changing the order of the join predicate can change the estimated cardinality, and that the interval of values of the columns (the minimum and max value, recorded in the column statistics low_value and high_value) influences the cardinality calculation.

Of course the SQL template for a two-column equijoin is

```
select t1.*, t2.*
  from t1, t2
 where t1.x = t2.x
   and t1.y = t2.y
   and t1.filter = ...
   and t2.filter = ...;
```

again I've used an equality filter predicate just for illustration purposes, it can be anything since the formula considers only the resulting f_num_rows statistic.

Each of the two join predicates (t1.x = t2.x and t1.y = t2.y) produces a selectivity (join_sel_x and join_sel_y) and the resulting join selectivity (join_sel) is simply the product of the two, as already noted in [Lewis, CBO, page 271]:

```
join_sel = join_sel_x * join_sel_y
```

The surprise is in the way these selectivities are computed.

The CBO will choose one of the join predicates as the first join predicate and apply it; the corresponding selectivity calculation is no different from the single-column case we already saw. In my test cases the predicate chosen as first has always been simply the first listed in the SQL statement, but of course it may be just the first in an internal structure that happens to be populated by chance almost always in the same order found in the SQL statement, and then there's always the Query Transformation preprocessor that might change the order for more complex statements.

So let's say that t1.x = t2.x is chosen as first; its contributing selectivity is

```
f_num_distinct (T1.X) = SWRU ( num_distinct(T1.X), f_num_rows(T1), num_rows(T1) )
f_num_distinct (T2.X) = SWRU ( num_distinct(T2.X), f_num_rows(T2), num_rows(T2) )

join_sel_x = 1 / ceil ( max ( f_num_distinct(T1.X), f_num_distinct(T2.X) )
```

Then, the other (second) predicate is applied, but the first predicate acts as an additional "filter" predicate that will apply two additional selectivities (first_pred_sel(T1.Y) and first_pred_sel_t2_y) that will reduce the f_num_rows "seen" by the second predicate selectivity formula, and so the filtered num_distinct figure:

[7] For my reference: num_rows(T1)=num_rows(T2)=100, num_distinct step=trunc(num_rows/10),f_num_rows(T1) step=trunc(num_rows/5), f_num_rows(T2) step=1

```
f_num_distinct (T1.Y) = SWRU (num_distinct(T1.Y),
                              first_pred_sel (T1.Y) * f_num_rows(T1),
                              num_rows(T1)
                             )
f_num_distinct (T2.Y) = SWRU (num_distinct(T2.Y),
                              first_pred_sel (T2.Y) * f_num_rows(T2),
                              num_rows(T2)
                             )

join_sel_y = 1 / ceil ( max ( f_num_distinct (T1.Y), f_num_distinct (T2.Y) ) )
```

The additional selectivities use the interval values (the column statistics low_value and high_value) of the columns referenced in the first predicate in their calculation. The rationale is most likely that if we have a join predicate such as t1.x = t2.x, all the values outside the overlapping interval can't match in the tables, hence they will be filtered out by the first join predicate. So each additional selectivity is set proportionally to how much of the overlapping interval covers the column interval (much like a BETWEEN clause); precisely, to the ratio of the width of the intersection with the overlapping interval and its interval width[8]. For example:

```
t1.x interval = [0, 100],  t2.x interval = [98, 142] => overlapping interval = [98, 100]
first_pred_sel (T1.Y) = (100 - 98) / (100 -  0) = 2 / 100
first_pred_sel (T2.Y) = (100 - 98) / (142 - 98) = 2 / 44
```

Note that the effect of these additional selectivities is to *increase* the cardinality, since they decrease f_num_distinct(T1.Y) and f_num_distinct(T2.Y) thus increasing the selectivity join_sel_y (which is proportional to 1 / f_num_distinct) and so the cardinality, since the cardinality is the product of the two f_num_rows times the product of the join selectivity and *the calculation of f_num_rows doesn't factor in the additional selectivities*. We'll explore this further below.

The familiar exhaustive check (join_exhaustive_mult_columns_2.sql)[9], 175,561 different test cases, using Yao's formula, multicolumn join sanity checks disabled[10], gives an absolute error less than 1 in 99.72 % of the cases, and perfect match in 99.08 % of the cases.

Note: If we do not disable the sanity checks, the formula changes - we will discuss the sanity check formula in a dedicated chapter, anyway the script checks it as well and in this case the absolute error is less than 1 in 100.00% of the cases, and perfect match in 99.86% of the cases.

As in the previous scenarios, creating indexes on (X,FILTER), (FILTER,X) and (X,Y),(Y,X) for both tables has no effect, with either the sanity checks enabled or disabled (again checked using join_exhaustive_mult_columns_2.sql[11]).

Collecting histograms on X and Y, again, changes the algorithm when the sanity checks are disabled. I have made no exhaustive investigation for this case, but script join_mult_histogram_influence.sql strongly suggests that the join selectivity is still the product of the selectivities of the single join predicates (t1.x=t2.x and t1.y=t2.y), and that the single join predicate selectivity calculation when histograms are collected does not factor in the intervals of the columns referenced in the other join predicate (that is, histograms trigger their own formula, identical to the single-column join scenario investigated in [Dell'Era, 2007]). So that the intervals influence the join selectivity only when the second single join predicate has no histogram collected - in all the two other cases (both predicates with histograms, then first without histograms and second with histograms) the intervals have no influence, as tested in the script.

When the sanity checks are enabled, collecting histograms does not change the join selectivity at all (this strongly suggests that the new 10g sanity checks always override the algorithm here described, as discussed below) - again checked with join_exhaustive_mult_columns_2.sql.

[8] Script interval_helpers.sql conveniently implements a function to calculate these selectivities.

[9] For my reference: num_rows(T1)=num_rows(T2)=11 and then 100, num_distinct(T1.X)=2+step(8), num_distinct(T1.Y)=2+step(8), num_distinct(T2.X)=2+step(8), num_distinct(T2.Y)=2+step(8),f_num_distinct(T1)=1+step(8), f_num_distinct(T2)=1+step(8).

[10] Hidden parameter "_optimizer_join_sel_sanity_check" set to false.

[11] See commented code - uncomment to reproduce.

Multiple-column equijoin: join predicate order influences the cardinality

From the previous discussion is evident that the algorithm is not symmetric, and that which join predicate is applied first matters, potentially a lot. Let's see in action, also as an example of calculation.

Script join_example_mult_order_changes_card.sql builds tables T1 and T2 with the following statistics (and runs with the multicolumn join sanity checks disabled[12] to factor out their effect):

```
num_rows(T1) = num_rows(T2) = 10000
num_distinct(T1.X) = 20 , num_distinct(T1.Y) = 20
num_distinct(T2.X) = 100, num_distinct(T2.Y) = 390
f_num_rows(T1) = 10000 (no filter on table T1)
f_num_rows(T2) = 100

low_value (T1.X) = low_value (T1.Y) = low_value (T2.X) = low_value (T2.Y) = 0
high_value(T1.X) = 19, high_value(T1.Y) = 19, high_value(T2.X) = 99, high_value(T2.Y) = 389
```

```
select t1.v1, t2.v1
  from t1, t2
 where t2.x = t1.x
   and t2.y = t1.y
   and t2.filter = 10;
```

```
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |   781 |
|*  1 |  HASH JOIN         |      |   781 |
|*  2 |   TABLE ACCESS FULL| T2   |   100 |
|   3 |   TABLE ACCESS FULL| T1   | 10000 |
-----------------------------------------
```

The first join predicate is t2.x = t1.x:

```
f_num_distinct (t1.x) = SWRU (   20, 10000, 10000) = 20
f_num_distinct (t2.x) = SWRU (  100,   100, 10000) = 63.5805485
join_sel_x = 1 / ceil (greatest (20, 63.5805485) = 1 / 64
```

The second join predicate is t2.y = t1.y:

```
first_pred_sel ( t1.y ) = 1 (interval of t1.x completely contained in t2.x)
first_pred_sel ( t2.y ) = (19-0) / (99-0) = 19 / 99

f_num_distinct (t1.y) = SWRU (  20, 10000 * 1      , 10000) =  20
f_num_distinct (t2.y) = SWRU ( 390,   100 * 19 / 99, 10000) =  18.7673466
join_sel_y = 1 / ceil (greatest (20, 18.7673466) = 1 / 20
```

Hence

```
join_card = round ( 100*10000 * (1 / 64) * (1 / 20) ) =  round ( 781.25 ) = 781 (as required)
```

Now let's simply swap the order of the join predicates:

```
select t1.v1, t2.v1
  from t1, t2
 where t2.y = t1.y
   and t2.x = t1.x
   and t2.filter = 10;
```

```
-----------------------------------------
| Id  | Operation          | Name | Rows  |
-----------------------------------------
|   0 | SELECT STATEMENT   |      |   562 |
|*  1 |  HASH JOIN         |      |   562 |
|*  2 |   TABLE ACCESS FULL| T2   |   100 |
|   3 |   TABLE ACCESS FULL| T1   | 10000 |
-----------------------------------------
```

[12] Hidden parameter "_optimizer_join_sel_sanity_check" set to false.

The first join predicate is now t2.y = t1.y:

```
f_num_distinct (t1.y) = SWRU (   20, 10000, 10000) = 20
f_num_distinct (t2.y) = SWRU (  390,   100, 10000) = 88.6905667
join_sel_y = 1 / ceil (greatest (20, 88.6905667) = 1 / 89
```

The second join predicate is now t2.x = t1.x:

```
first_pred_sel ( t1.x ) = 1 (interval of t1.y completely contained in t2.y)
first_pred_sel ( t2.x ) = (19-0) / (389-0) = 19 / 389
f_num_distinct (t1.x) = SWRU (  20, 10000 * 1       , 10000) =  20
f_num_distinct (t2.x) = SWRU ( 100,   100 * 19 / 389, 10000) =   4.79127781
join_sel_x = 1 / ceil (greatest (20, 4.79127781) = 1 / 20
```

Hence again the correct cardinality:

```
join_card = round ( 100*10000 * (1 / 89) * (1 / 20) ) = round ( 561.797753 ) = 562 (as required)
```

A way to fix this dependency on the predicate order would be simply to extend the calculation of the additional selectivities to the first join predicate; that would make the algorithm symmetric (and self-consistent). After all, if filtering based on the value intervals is deemed an improvement, I can see no reason why the first predicate should be excluded from this refinement.

Multiple-column equijoin: less is more

Let's examine the effect of the intervals for multi-column joins. Script join_mult_overlap_dependency.sql builds a table with all intervals equal to [0, 100], and then varies the T1.X interval. It checks internal intersection (one interval containing another) by setting T1.X=[0, overlap%] and then external intersection by setting T1.X=[100- overlap%, 200-overlap%]- the latter experiment is analogous to the one performed by Jonathan Lewis in [Lewis, CBO, page 279] for the single-column join scenario. The resulting CBO cardinalities are (with the multicolumn join sanity checks disabled[13]; column card_sanity reports the cardinality with the sanity checks enabled):

Internal intersection

overlap%	low	high	card	sel_y	card_sanity
100	0	100	25	0.1	25
80	0	80	28	0.1	25
50	0	50	31	0.13	25
20	0	20	50	0.2	25
2	0	2	125	0.5	25
0	0	0	-	-	-

External intersection

overlap%	low	high	card	sel_y	card_sanity
100	0	100	25	0.1	25
80	20	120	28	0.1	25
50	50	150	31	0.13	25
20	80	180	50	0.2	25
2	98	198	125	0.5	25
0	100	200	250	1	25
n/a	101	201	1	-	1

So counter-intuitively, the less overlap you have (so the fewer rows are going to match over the join predicate) the more the estimated cardinality will be. This is because the CBO calculates the filtered cardinalities first (in the "SINGLE TABLE ACCESS PATH" section at the beginning at the 10053 trace), looking only at the filter predicates and ignoring any join predicate, then calculates the join selectivity, and then combines the two, as apparent from the 10053 trace:

Join Card: 250.00 = outer (100.00) * inner (25.00) * sel (0.1)

Only the join selectivity algorithm considers the additional filtering performed by the join predicates, the filtered cardinality one not. The two algorithms are not consistent.

So as the overlap shrinks, the selectivity formula will "see" less and less rows, so less and less filtered distinct values, and the less distinct values you have in a join, the more rows are going to match, hence the join_sel_y selectivity will go up.

Notice in particular that when the overlap eventually reaches zero, only one distinct value survives the filtering, so join_sel_y = 1 (a Cartesian product). At this stage only a few rows survive, but the filtered cardinality formula fails to notice this.

If the filtered cardinality formula considered the additional filtering, the final cardinality would nicely degrade instead of strangely increase (probably, the best strategy would be to consider the additional interval-based selectivities *only* in the filtered cardinality formula). Instead, 10g and the latest patches of 9i (including 9.2.0.8) choose to deploy the

[13] Hidden parameter "_optimizer_join_sel_sanity_check" set to false.

hatchet of the multicolumn join sanity checks, that cuts the cardinality at 25, essentially cancelling the effect of the additional join selectivities altogether (in this case, I haven't checked whether this is always the case). We'll explore them in the next chapter.

Note also that when the intervals fail to overlap at all (T1.X = [101,201]), the cardinality falls abruptly from 250 to 1 (which is zero rounded[14] to 1), as expected.

Multiple-column equijoin: sanity check or a superior new algorithm?

Let's revisit (briefly and qualitatively) the statistical justification that leads to the usual formula for join cardinality estimates, and see how that can lead to a diversion that explains the new *10g multicolumn join selectivity sanity check* - then briefly discuss the probable reasons that made the kernel developers adopt the latter instead of the former for now and the future.

A possible way to look at a join (actually, the *definition* of a join in relational algebra) is to generate the Cartesian product of the tables (possibly after applying the filtering predicates if any exists), then apply the join predicates (delete from the Cartesian product the rows that doesn't satisfy them).

But there are two ways to apply the join predicates: the first (let's call it the *traditional* way) is to make a first pass applying the first join predicate, and then make another pass on the surviving rows[15] with the second join predicate. That could be visualized this way:

```
   where (t1.x = t2.x)   <-- first pass
     and (t1.y = t2.y)   <-- second pass
```

If the two tables are "big" enough[16] and no constraints are enforced (two hypotheses of *paramount importance* as we will see shortly after), so that a simple probabilistic analysis can be made, the first predicate in isolation has a selectivity of sel_x, and the second of sel_y, and if we can assume that the columns are non-correlated, that leads naturally to the usual formula join_sel = sel_x * sel_y. That becomes, if we can assume that the preconditions for the standard formula apply, the familiar formula (let nd = num_distinct, possibly filtered, for brevity):

```
join_sel = [ 1 / (max ( nd(T1.X) , nd (T2.X) ) ] * [ 1 / max ( nd(T1.Y) , nd (T2.Y) ) ]
```

The other way to apply the join predicates (let's call it the *join key* way) is to make a single pass filtering out the rows whose pairs (T1.X, T1.Y) and (T2.X, T2.Y) do not match:

```
   where (t1.x, t1.y) = (t2.x, t2.y)   <-- single pass
```

This gives obviously the same exact result set as before, and again, if the preconditions for the standard formula apply, that leads to the formula:

```
join_sel = 1 / max ( nd(T1.X,T1.Y) , nd(T2.X,T2.Y) )
```

We need to know the combined (concatenated) number of distinct values of the columns - but *for "big" unconstrained tables* and non-correlated columns, that would be nd (col1, col2) = nd (col1) * nd (col2). Thus:

```
join_sel = 1 / max ( nd(T1.X) * nd(T1.Y) , nd(T2.X) * nd(T2.Y) )
```

The two approaches seem equivalent at a first glance, but actually they are not. Precisely, they are equivalent in these two cases (since they both "get the max value" from only one of the tables):

```
nd(T1.X) <= nd(T2.X)
nd(T1.Y) <= nd(T2.Y)
```

```
nd(T1.X) >= nd(T2.X)
nd(T1.Y) >= nd(T2.Y)
```

[14] From the 10053 trace:
Join Card: 0.00 = outer (100.00) * inner (100.00) * sel (0.0000e+000)
Join Card - Rounded: 1 Computed: 0.00

[15] So the first join predicate acts as an additional "filtering" predicate - that's probably the train of thoughts followed by the designers of the algorithm we discussed before.

[16] I should say "with an infinite number of rows and a finite number of distinct values".

But they are not in these:

```
nd(T1.X) <= nd(T2.X)
nd(T1.Y) >  nd(T2.Y)
```

```
nd(T1.X) >= nd(T2.X)
nd(T1.Y) <  nd(T2.Y)
```

This could be checked even deterministically[17] in this scenario, and note that the traditional way estimates the selectivity exactly:

table T1	
X	Y
1	1
2	2
3	1
1	2
2	1
3	2
nd=3	nd=2

table T2	
X	Y
1	1
2	2
1	3
2	1
1	2
2	3
nd=2	nd=3

join_sel = 4 / (6*6) = 1 / 9

[1/ (max(3,2)] * [1/max(3,2)] = 1 / 9

1 / max (2*3, 3*2) = 1 / 6

The reason for the difference is that one of standard formula preconditions (check [Dell'Era, 2007, Appendix A] for the proof) is the principle of inclusion: "all the values in the table with the smaller num_distinct must have a match in the other table". This is certainly true for the single columns (T1.X against T2.X, T2.Y against T1.Y) but not for the (T1.X,T1.Y) and (T2.X, T2.Y) pairs - two pairs for each table does not match in the other table.

This can be also seen very convincingly using script join_card_04.sql from [Lewis, CBO, page 272], that creates two big unconstrained tables whose statistical distribution shows the "zig-zag" pattern for the num_distinct figures. The actual real cardinality for the test case is 50,262, and the CBO plan on page 272 (that uses the "traditional" formula) gives the statistically correct answer (50,000); the CBO plan on page 273 instead, that uses the "join key" way, gives the wrong estimate (62,500). If you remove the zig-zag pattern, both give the same (statistically correct) result.

Hence for "big" unconstrained tables, the traditional way is superior.

An *improved* version of the "join key" way is used in 10g (and backported to the latest patchsets of 9i, I have only tested 9.2.0.8) and is activated by the hidden parameter "_optimizer_join_sel_sanity_check", which defaults to true. It is described in [Lewis, CBO, page 274] and overrides the "traditional" algorithm setting

```
join_selectivity = 1 / max ( mjkc (T1), mjkc (T2) )
```

where mjkc is the "Multi-column Join Key Card" (name from the 10053 trace):

```
mjkc (table) = min ( nd (col1) * nd (col2), num_rows (table) )
```

and the improvement consists in not using the plain product of the distinct values (nd (col1) * nd (col2)) but to truncate it at the obvious upper value of num_rows (you can't have more distinct values then the number of values) which is far from being a secondary refinement - as we will see shortly after, it changes considerably the output and is the key of the new method. For "big" unconstrained tables, we can assume nd (col1) * nd (col2) << num_rows (table) and so mjkc (table) = nd (col1, col2), and so, in this case this is the same as the plain join key method.

I've said *overrides the "traditional" algorithm* because even if I've tried hard[18], I haven't found a single test case where the traditional formula is used, so as far as I can tell this is not really a "sanity check" but an algorithm substitution. I'd like to hear from anyone that knows otherwise.

So, is the new algorithm a step backward since it performs worst for big unconstrained tables? Not at all, and actually the opposite - once we consider the most common type of joins that occur in real life applications.

Joins in OLTP applications occur almost always (if not always) over FK relations; even many-to-many relations are usually implemented using an intermediate relation table and two FKs to the related tables. Joins in DWH/DSS applications either use star joins (which show again FKs from the fact tables to the dimension tables), or simply are

[17] Just the repeat the six-row pattern ad libitum to have a "big" table, the answer doesn't change.
[18] generating thousands of test cases and checking them, using histograms, even faking density, ...

plain old OLTP schemas just made very big. For enforced constraints, if the PK is supported by a UNIQUE index[19], the CBO has a special code path (that shows as "using concatenated index card" in the 10053 trace) that sets join_sel = 1/distinct_keys (pk unique index), which is the same as 1/num_rows(parent) - this essentially cancels num_rows(parent) from the join cardinality formula that hence becomes join_card = num_rows(child) * selectivity (child filter predicates) * selectivity (parent filter predicates), which is statistically sound. So for the most important type of joins, the cardinality estimation is nicely solved.

But the world is cursed by "database-independent" applications that refrain from explicitly declaring and enforcing FKs (PKs) in the database and manage them themselves, and usually FKs (less commonly PKs) in huge DWH applications are not enforced[20] due to the prohibitive cost of managing them - yet they are still "logically present" if the application (or data load) manages data correctly - and what remains if we strip the constraints keeping the rest? We have tables with unique values on columns meant to be joined ("logical PKs") that act as "logical parents" in join operations with columns in other tables that act as "logical children" - that is, all the rows in the logical child match one and only one row in the logical parent. The DDL SQL constraints may not be there, still the tables are statistically constrained.

And for this kind of joins over "logical" FKs, the new method (the *10g multicolumn join selectivity sanity check)* always produces *better* cardinality estimates than the traditional way, and with a vast potential to estimate the cardinality *exactly*. Let's see why.

Since nd(parent.x) >= nd (child.x) and nd (parent.y) >= nd (child.y), the traditional join selectivity is

```
join_sel_traditional = 1 / ( nd(parent.x) * nd (parent.y) )
```

The exact cardinality is num_rows(child), so the exact cardinality is obtained by factoring out num_rows(parent) from the join cardinality formula - hence the exact join selectivity is

```
join_sel_exact = 1 / num_rows (parent)
```

Thus join_sel_traditional is obviously wrong with the exception of the unlikely case that nd(parent.x) * nd (parent.y) = num_rows (parent), so in general the traditional selectivity is a severe underestimation of the exact one.

The plain join key method produces the same wrong answer, since from (parent.x) >= nd (child.x) and nd (parent.y) >= nd (child.y) follows that the zig-zag pattern cannot occur and so the estimate is the same as the traditional one. In passing, that's because putting the logical uniqueness and matching constraints invalidates some of the prerequisites[21] of the statistical analysis above.

So the CBO makes the smart and critical correction of using mjkc instead, which is the join key cardinality capped by num_rows(table), that is, as we have seen

```
mjkc (table) = min ( nd (col1) * nd (col2), num_rows (table) )
```

This is of paramount importance since now for the parent we have

```
mjkc(parent) = min (nd(parent.x) * nd(parent.y) , num_rows (parent) ) = num_rows (parent)
```

Hence the join key selectivity becomes

```
join_sel_join_key = 1 / max ( mjkc (child), num_rows (parent) )
                  = 1 / max ( min ( nd(child.x)*nd(child(y), num_rows(child)), num_rows(parent))
```

it can be shown that

```
join_sel_traditional <= join_sel_join_key <= join_sel_exact
```

that is, the new method selectivity is always closer or equal to the exact one than the traditional, and in general, not worse.

[19] So if the PK is not deferrable, if another not unique index with the same leading columns didn't exist at PK creation time, etcetera.
[20] I'm ignoring RELY constraints here to keep the discussion short.
[21] E.g. you cannot have a parent table with both an infinite number of rows and a finite number of distinct values.

But let's see how it is also usually much better - that is, it is able to produce the exact cardinality in many important real-life scenarios - keeping in mind that the traditional one is fixed at the wrong selectivity we saw above.

We get the exact cardinality when

```
join_sel_join_key = join_sel_exact
max ( mjkc (child) , num_rows (parent) ) = num_rows(parent)
min ( nd(child.x) * nd(child.y), num_rows(child) ) <= num_rows(parent)
```

For example, we get the exact cardinality when we have num_rows(child) <= num_rows(parent), which is far from being an infrequent case - consider an ORDERS fact table partitioned by month, and a join of the last month partition to the CUSTOMERS dimension; if only a fraction of your customers has placed one or few orders last month, which is probably the norm[22], you'll have num_rows(child) << num_rows(parent), so the exact cardinality. The traditional method would have estimated the selectivity as 1 / (nd(parent.x) * nd (parent.y)) << 1 / num_rows (parent) instead, so anything ranging from a possibly severe underestimation to an unlikely exact estimation (in the lucky case that nd(parent.x) * nd (parent.y) = num_rows (parent)).

When num_rows(child) > num_rows(parent), you'll get the exact selectivity again when nd(child.x) * nd(child.y) <= num_rows(parent), and then the wrong selectivity - but that is going to be, as we have seen, *closer to the exact one* or equal than the traditional estimate 1 / (nd(parent.x) * nd (parent.y)), and usually much better.

So in short, recent versions of the CBO are "biased" towards the most common join type (over enforced or logical FKs) and will perform definitely better for this type of ubiquitous join - at the cost of possibly performing worse for some of the other, less-frequent (or improbable in real-life) scenarios.

As a final note, if the join key is really the way for the future, the discussion in this paper easy leads to a forecast - that one day the CBO will consider a suitable index (not necessarily unique) covering the N columns in an N-column join and use the following formula to compute the filtered num_distinct:

```
f_num_distinct ( t.C1,...,t.CN ) = SWRU ( distinct_keys(index), f_num_rows(T), num_rows(T) )
```

thus removing the assumption of non-correlation and leading to a (much?) better estimate of the number of distinct values to be fed to the standard formula.

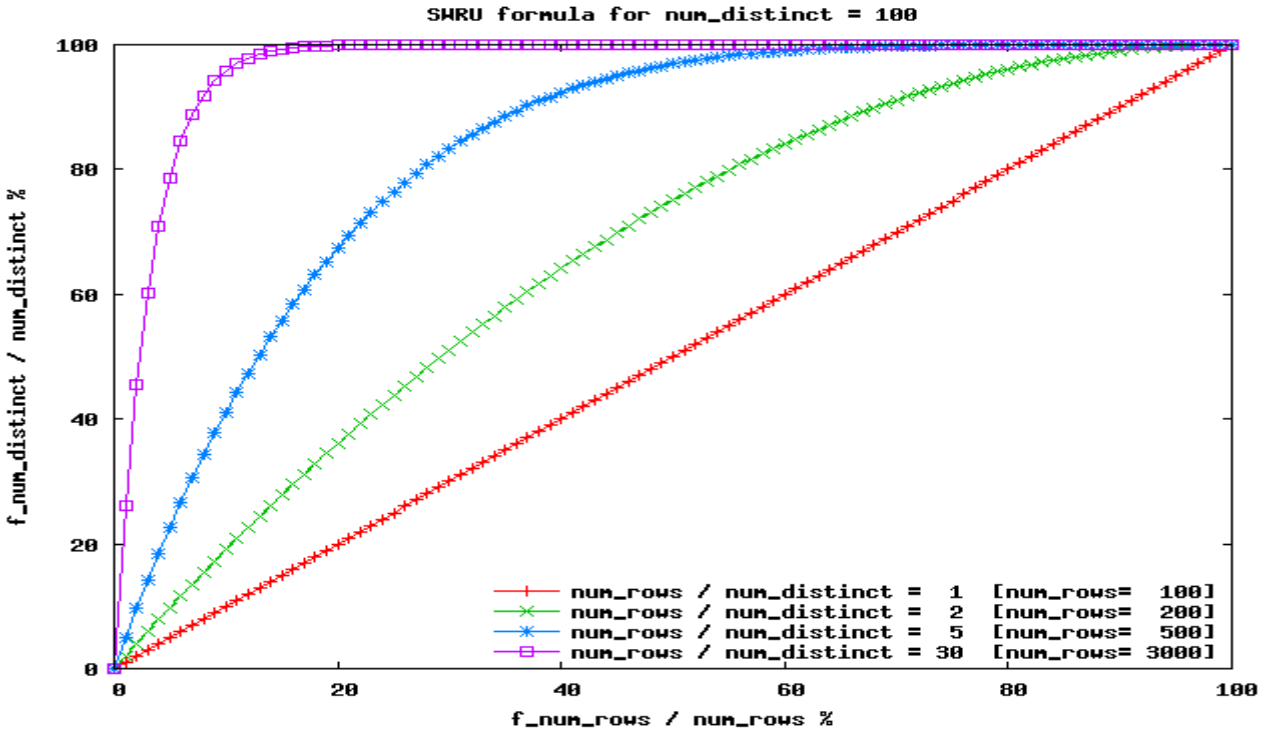
Conclusion

We have seen that the SWRU formula is used in many fundamental formulae by the CBO: for distinct, group by and join cardinality estimates, for both single-column and multiple-column scenarios.

We have also discussed the peculiar (and surprising at times) traditional algorithm for multicolumn joins, probably now superceded by a new algorithm designed to handle better the most common join scenarios.

[22] I place an order on Amazon about every three weeks - yet the vast majorities of people I know make an order every six month, or once every Christmas, or have placed a single order in their lifetime.

Appendix A - the SWRU formula



The graph[23] diagrams the percentage of filtered distinct values / distinct values (that is, the fraction of distinct values that survive the filtering) against the percentage of rows filtered from the table.
The more distant the y-value is from 100%, the higher is the amount of distinct values filtering, so the less we can ignore the SWRU formula and approximate it with num_distinct.

If we are selecting 100% of the rows from the table (right end of the diagram), obviously we are not filtering at all, so all the distinct values survive and we have exactly f_num_distinct = num_distinct. As we filter more, thus moving toward the left end of the diagram, obviously some distinct value may be left behind, and the SWRU correction starts to kick in.

Now, consider the upper (violet) curve, where we have 30 values per distinct value: if we have a lot of values (rows) per distinct value, and we select a lot of values (right end), it is very likely that all the values we select contain all the distinct values, hence the curve will tend to stay very near to 100%. But if we are selecting only a few rows (left end), sure the values will be different, but they cannot be more than the number of values we are picking - hence the curve will be very near the straight line f_num_distinct = f_num_rows. The two effects will combine for the points in the middle, producing the (quite rapid) transition from the two curves we observe.

As we decrease the number of distinct values per value (green line, where we have only 2 values per distinct value), it is more likely that the same distinct value is picked twice (thus not increasing f_num_distinct), thus the curve will be lower and only if we pick almost all the rows we can be sure to get all the distinct values.

The bottom (red) line is for the case num_distinct = num_rows (all distinct values); here any value not picked means exactly one distinct value less, hence we get a straight line between (0,0) and (100%,100%).

So, only if you are selecting almost all rows, or have a very high ratio of num_rows / num_distinct, the SWRU formula can be ignored (approximated by num_distinct) - provided you are not selecting only a few rows.

If you are selecting only a small amount of rows, or have a low ratio of num_rows / num_distinct, the SWRU formula has to be factored in the calculation.

[23] Script swru_graph_data_plot.sh, that calls swru_graph_data_plot.sql.

Bibliography

[Lewis, CBO]	Jonathan Lewis, *Cost Based Oracle: Fundamentals*, Apress, 2006, ISBN 978-1590596364.
[Yao, 1977]	S.B. Yao, *Approximating Block Accesses in Database Organizations*. Communications of the ACM, April 1977, Volume 20, Number 4.
[Dell'Era, 2005]	Alberto Dell'Era, *Expected distinct values when selecting from a bag without replacement*. Available on www.adellera.it/investigations/distinct_balls
[Dell'Era, 2007]	Alberto Dell'Era, *Join Over Histograms*. Available on www.adellera.it/investigations/join_over_histograms

Paper version: 1.1, 2007-09-07
Converted into pdf format by PrimoPdf, configuration "_prepress.ini"

