Join Selectivity

Jonathan Lewis
JL Computer Consultancy
London, UK

Keywords:

Selectivity, Cardinality, Statistics, Joins, Cost-based Optimizer

Introduction

In this note I will be describing the basic mechanism that Oracle uses in calculating join selectivity. We will start with the simplest case of a two-table join using a single column with equality, and then extend upwards to multi-column joins and the strategy for handling more than two tables. Given the limits on space and time I will not cover details of how the arithmetic changes when there are histograms on the join columns, but I will emphasise the importance of two "sanity checks" that the optimizer makes to adjust the basic calculation method.

As before, in the presentation and note on single table selectivity, I shall not be looking at cases involving histograms.

Getting Started

Before we can handle the calculation of join selectivity we need to cover two more aspects of the calculation of selectivity on a single table: the method that the optimizer uses to calculate selectivity when comparing two columns in the same table, and the method the optimizer uses to calculate the number of distinct values of a column after selection of a subset of a table. For determining the strategy that the optimizer uses for the latter we owe thanks to Albero Dell'Era who has published a significant number of articles on this topic at: `http://www.adellera.it/investigations`

Comparing columns (equality)

The mechanism the optimizer uses for the basic and most common comparison of columns is very simple. Consider the following table and query:

```
create table t1
as
select
        rownum                                  id,
        mod(rownum-1,200)                       mod_200,
        trunc(dbms_random.value(0,300))         rand_300
from
        {very large rowsource}
where
        rownum <= 1e6
;
```

```
select * from t1 where mod_200 = rand_300;
```

In cases like this the optimizer effectively decides whether it should treat the calculation as "mod_200 = {unknown_value}" or whether to treat it as "rand_300 = {unknown_value}", picking the calculation that produces the lower selectivity. In this case 1/300 is smaller than 1/200, so the optimizer chooses the former and the execution plans predicts 3,333 rows as the number of rows to be returned.

```
-------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  3333 | 39996 |   348  (15)|
|*  1 |  TABLE ACCESS FULL| T1   |  3333 | 39996 |   348  (15)|
-------------------------------------------------------------

Predicate Information (identified by operation id):
---------------------------------------------------
   1 - filter("MOD_200"="RAND_300")
```

An important point to note is that for this type of example Oracle doesn't even try to consider the effects of how well the columns overlap. Intuitively we can look at the data and spot that at least one third of the data can't possibly be returned by the query because the rand_300 value is greater the 199 – the highest value for mod_200. We can exaggerate the effect by changing the definition for rand_300 in the table generation code to add 250 to the value (so that ALL the values exceed the maximum value for mod_200) and still the optimizer predicts a join cardinality of 3,333 rows.

Comparing columns (rages)

Joins with range-based predicates are probably much rarer than joins with equality predicates, nevertheless they are used. The optimizer's approach to the arithmetic approaches a very human viewpoint of simply counting the possibilities (which means that in this case it WILL consider the effects of degree of overlap of the values in the columns).

In effect, the optimizer considers a simplified (uniformly distributed) model of all possible combinations of the two columns, based on their low_value, high_value and num_distinct, and then counts the number of combinations that match the predicate. Of course this is actually done through a formula that caters for 4 possible ways in which the ranges could overlap (and the degenerate case where they don't overlap).

To demonstrate the concept, consider the following idealised data set and suitable query:

```
select
              rownum                        id,
              20 + 4 * mod(rownum,6)        col_x,
              11 + 3 * mod(rownum,8)        col_y
from
              all_objects
where
              rownum <= 48
;
```

```
select
                *
from            t1
where           col_x > col_y
;
```

I have created an idealised data set for the conditions low_value = 20, high_value = 40, num_distinct = 6 for col_x, and low_value = 11, high_value = 32, and num_distinct = 8. There are 48 possible combinations, so I have created 48 rows to capture them all. Oracle "visualises" the join as shown in figure 1:

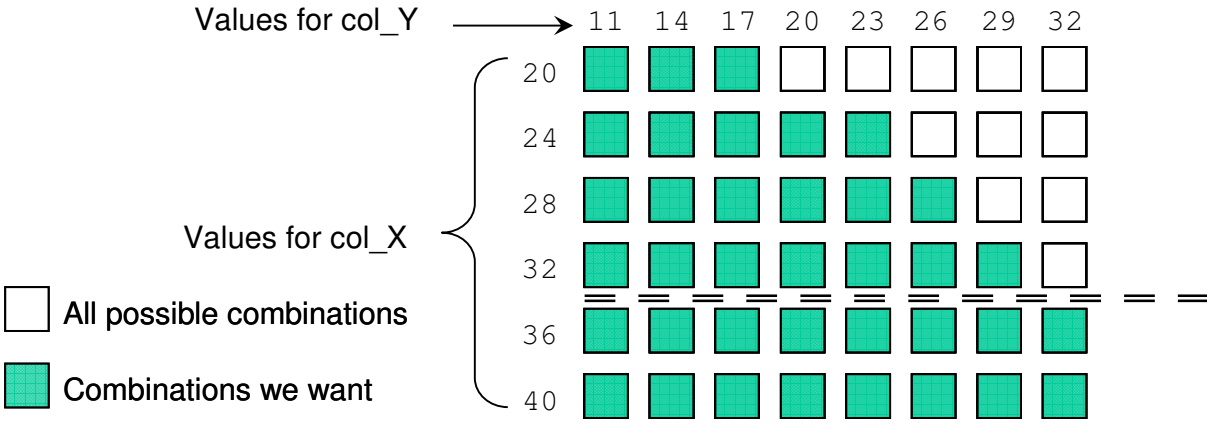

Figure. 1: Visualising the join

In effect we turn the arithmetic into geometry. We have a (possibly truncated) triangle where the values for col_x and col_y overlap and a (possibly empty) rectangle where the values for one column fall outside the range of values for the other. The formula becomes messy when you try to cater for the combinations of overlap – but the most significant factor is that when the two sets of data are "close matches" the selectivity will be close to 0.5.

Distinct

Returning to the data set created for the session on singletable selectivity, you may decide that you don't write queries like the following very often:

```
select distinct mod_300 from t1 where date_1000 = {constant};
```

The optimizer, on the other hand, needs to have an algorithm for getting a good estimate for the number of rows that a query like this will return because it's an important part of calculating join selectivity.

I know that mod_300 has been defined to hold 300 distinct values, but I also know that date_1000 holds 1,000 distinct values, so my query above will return 1,000 rows from the million rows in the table. How many of those 300 distinct values am I likely to find in a "randomly" selected 1,000 rows?

Clearly if I pick just 300 random rows I'd have to be very lucky to get one row for each value; equally obviously if I pick 999,000 rows I'd probably get at least one row of each value. The probably number of distinct values that I will get is dependent on the number of rows I pick.

In his work investigating how Oracle handles this question, Alberto Dell'Era deduced that it was applying a result of probability theory called "selection without replacement", for which we can simply quote a formula.

t1 has 1,000,000 rows (number of rows)	call this nr
mod_300 has 300 distinct values (number of distinct)	call this nd
date_1000 = {constant} returns a sample of 1,000 rows	call this s

The expected number of distinct values for mod_300 in the sample will be:

nd * (1 - power(1 - s/nr, nr/nd))

In our case,

300 * (1 - power(1 - 1000/1000000, 1000000/1000)) = 289.3156

Basic Joins

We are now equipped to handle the calculation of join cardinality and join selectivity (at least, for the simple cases). Oracle even gives us a statement (in MoS document 68992.1) of how the calculation works for queries of the following form:

```
select  *
from    t1, t2
where   t1.c1 = t2.c2
and     {filter predicate on t1}
and     {filter predicate on t2}
```

Paraphrasing, and cosmetically editing, the statement we have the following:

The join cardinality is given by

join selectivity * filtered cardinality(t1) * filtered cardinality(t2)

The join selectivity is given by the (still rather messy)

((num_rows(t1) – num_nulls(t1.c1)) / num_rows(t1)) *
((num_rows(t2) – num_nulls(t2.c2)) / num_rows(t2)) /
greater(num_distinct(t1.c1), num_distinct(t2.c2))

Putting this into words – we apply the (single table) filter predicates to each table in turn to calculate the number of rows selected from each table, then do a Cartesian merge join of those two result sets. We apply the join selectivity to this Cartesian product.

But at this point the join selectivity is about a single table – the Cartesian product – and is derived in the way we derived the selectivity when comparing two columns for equality in a single table: it's 1/num_distinct for whichever column has the larger number of distinct values.

The slightly messy bits of the calculation "(num_rows – num_nulls)/num_rows" appearing twice is the effect of factoring out the fraction of each table that can't be returned by the join because comparisons with null don't return true.

There is a very important detail to the calculation, though. The num_distinct() that appears twice in the formula is NOT the value we get from user_tab_columns, it's the value we get from the "selection with replacement" formula after allowing for filter predicate on the related table. Let's demonstrate this with a worked example – we'll start with the table t1 from the Single Table session, and create a copy of it called t2 and then join the two table with the following query:

```
select  *
from
        t1,  t2
where
        t1.date_1000 = {constant}
and     t2.mod_200   = t1.rand_300
```

The optimizer produced the following plan for the query:

```
-------------------------------------------------------------------
| Id  | Operation          | Name | Rows  | Bytes | Cost (%CPU)|
-------------------------------------------------------------------
|   0 | SELECT STATEMENT   |      |  3448K|   368M|  2466  (15)|
|*  1 |  HASH JOIN         |      |  3448K|   368M|  2466  (15)|
|*  2 |   TABLE ACCESS FULL| T1   |  1000 | 56000 |  1269  (17)|
|   3 |   TABLE ACCESS FULL| T2   |  1000K|    53M|  1127   (7)|
-------------------------------------------------------------------
```

Our target is to derive the same 3,448K as the join cardinality.

Filtered cardinality of t1 = 1,000
Filtered cardinality of t2 = 1,000,000 (as there are no filter predicates)
Num_distinct(t2.mod_200) = 200 (as there are no filter predicates on t2)
Num_distinct(t1.rand_300) = 289.3156 (as derived earlier in the section on "Distinct"

So the larger value of num_distinct is 289.3156, there are no nulls to worry about, and the join cardinality is 1,000 * 1,000,000. The join cardinality is therefore: 1e9/289.3156 = 3,456,433 – which is close enough, especially given the K notation used by Oracle.

Extending the algorithm

Once you've learned the basic steps of the calculations anything more complex is simply a case of repeating the basics. Consider the following query:

```
select        t1.v1, t2.v1, t3.v1
from
```

```
                t1, t2, t3
where
                t2.join1 = t1.join1
and             t2.join2 = t1.join2
/*                              */
and             t3.join2 = t2.join2
and             t3.join3 = t2.join3
/*                              */
and             t3.join4 = t1.join4
;
```

The query joins three tables, and the joins into t2 and t3 are multi-column joins. How do you handle a three table join – one step at a time: we calculate for a two-table join between t1 and t2, then calculate for a two table join between (t1,t2) and t3. We already know how to handle multi-column joins – because a join condition becomes a single table condition after we've thought through the Cartesian join stage, and we know (from the previous presentation) how to combine single table predicates.

If we take the join into t3 as a demonstration, I'm going to simplify the situation by assuming we have no nulls involved so that I can remove all the "(num_rows – num_nulls) / num_rows" expressions for the equation. We simply multiply together the three components describing the t3 join – that's on columns that I've named join2, join3, and join4 – so we get:

```
greater( num_distinct(t3.join2),  num_distinct(t2.join2)) *
greater( num_distinct(t3.join3),  num_distinct(t2.join3) ) *
greater( num_distinct(t3.join4),  num_distinct(t1.join4) )
```

Conveniently I don't have any filter predicates on my tables – but where we see num_distinct() in the above we would have to remember to identify the relevant filtering predicates and adjust the basic value by using the "selection without replacement" formula.

Sanity Checks

I'm not going to construct a long and tedious example of all the working – I only want give you an outline of the mechanism for calculating join selectivity as a basis for a couple of suggestions and warnings.

The first warning, inevitably, is about the effect of multiplying individual selectivities – if your predicates are not independent the optimizer is making a bad assumption and the arithmetic can produce some very bad numbers. Fortunately you can help the optimizer by creating (with due caution) extended statistics of the "column group" type to give the optimizer better information. A warning here, though: make sure you check the predicate section of any execution plans – thanks to features like transitive closure predicates can appear to move from one table to another, so the predicates you see in your query may not be the predicates you need to create column groups for.

The second warning is about one of the "sanity checks" the optimizer does. Although it can work out all the individual selectivities and multiply them based purely which of each pair of columns has the greater num_distinct, it also checks for multiple predicates joining the same pair of tables and modifies the selection of the individual predicates to "group" them by table. In the example above, for instance, the optimised will either use num_distinct(t3.join2) * num_distinct(t3.join3) or num_distinct(t2.join3) * num_distinct(t2.join2) – it won't pick one value from t3 and one value from t2. This is known as the "multi-column" sanity check.

This can have side effects when you edit the query – even with a simple cosmetic edit. If you check the query you will see that there's a predicate t3.join2 = t2.join2; but (thanks to predicate closure) I might have used the predicate t3.join2 = t1.join2. If I had done that then my multi-column sanity check between t2 and t3 would disappear and I'd be looking at a multi-column sanity check between t3 and t1 – so the numbers would change, and with a change in the selectivity we might get a change in plan "for no obvious reason".

The second sanity check is the index sanity check. Again it will only be relevant if you have joins involving multiple columns on a given table. Again our example shows (join2, join3) as a pair of columns joining t2 and t3. If either of these tables has a unique index on exactly these columns the optimizer will use the number of distinct keys of an index rather than multiplying up two separate column selectivities. This re-introduces the warning about cosmetic edits – in a query that visibly joined t3 to t2 on column join2 you might be using index stats from t2, rewrite the query to join t3 to t1 on join2 and you change the optimizer's choice of numbers to use in the calculation.

Fortunately the index sanity check applies only to unique indexes – which tend to exist for very good reasons, and are unlikely to be dropped. But a side effect of the index sanity check is that if you do drop an index an execution plan that is not "using" that index might change its execution plan because the distinct_keys value for the index was being used in the calculation even though the index itself didn't appear in the plan. Although we don't often drop unique indexes, though, there are cases when we might think about changing a unique index into a non-unique index so that (for example) we can make a unique constraint deferrable – if you make that change and some execution plans may change "for no obvious reason".

Conclusion

Without chasing into the very fine detail I've given you an outline of the type of work the optimizer does to calculate join selectivity. I've highlighted a couple of anomalies and inconsistencies regarding the use of low and high values, and the occasions when index statistics are used. I believe that (generally) it is the impact of the variations that causes the most surprises; so if you are aware of the strategies the optimizer is using you will be able to identify more easily why you are getting an unexpected plan (or change in plan). With this information you will be better equipped to decide the best way to fix an errant plan – this may be by creating column group statistics, modifying the way an SQL statement expresses its join lists, or sometimes (by necessity) creating an SQL Profile to correct to optimizer's estimates or creating an SQL Baseline (or an outline, or a set of in-line hints) because that's the only way to get the optimizer to follow the most efficient path.

Contact address:

Jonathan Lewis
JL Computer Consultancy
1, Saxonbury Gardens
Long Ditton
Surrey KT6 5HF
UK

Phone: +44(0)7973 188785
Fax: n/a
Email jonathan@jlcomp.demon.co.uk
Internet: jonathanlewis.wordpress.com