Basic Selectivity

Jonathan Lewis
JL Computer Consultancy
London, UK

Keywords:

Selectivity, Cardinality, Statistics, Cost-based Optimizer

Introduction

In this note I will be describing some of the ways in which Oracle calculates "single table selectivity"; but given the length of the note, and the brevity of the presentation it supports, the content will have to be strictly limited; in particular I will not be considering any cases involving histograms, or the effects of partitioned objects. This note should be read in conjunction with the presentation slides.

Why ?

It is important to have some idea of how the optimizer calculates selectivity because its cardinality ("number of rows") estimates are based on the selectivity estimates, and cardinality is one of the key inputs to the optimizer's choice of execution plan. (The other key inputs being the pattern of data scatter – as indicated by the clustering_factor of indexes – and the capability of the machine as indicated by the "system stats" – as captured in sys.aux_stats$.

When the optimizer picks a bad execution plan your knowledge of how it calculates selectivity may help you understand why a particular line in the execution plan has produced a bad cardinality that is responsible for driving the optimizer down the wrong path; once you've identified the problem you can then decide how to address the problem – perhaps by creating some "extended stats", perhaps by writing code to use the dbms_stats package to create a better model of the data in a particular table's stats, perhaps by hinting the code (explicitly, or implicitly through an outline or SQL Baselines).

Warning

Before saying anything else it is important to supply a warning, which I will accompany by an example: Oracle is constantly refining the optimizer code, and this means that changes in calculations (usually changes for the better) can appear in every point release of the product. Even the calculation for the following basic example has evolved across recent versions of the optimizer:

```
create table t1
as
with generator as (
        select          --+ materialize
                rownum id
        from dual
        connect by
                level <= 1e4
)
select
```

```
        rownum                     id,
        mod(rownum-1,200)          mod_200,
        mod(rownum-1,10000)        mod_10000,
        lpad(rownum,50)            padding
from
        generator        v1,
        generator        v2
where
        rownum <= 1e6
;

begin
        dbms_stats.gather_table_stats(
                ownname      => user,
                tabname      =>'T1',
                method_opt  => 'for all columns size 1'
        );
end;
/

create index t1_c2 on t1(mod_200, mod_10000) compress;

select
              *
from          t1
where         mod_200 = 100
and           mod_10000 = 100
;
```

The following execution plans come from 10.2.0.5 and 11.2.0.4 (the change in cardinality actually appeared in 11.1.0.7):

```
Execution Plan 10.2.0.5
--------------------------------------------------------------------
| Id | Operation                     | Name  | Rows   | Bytes | Cost  |
--------------------------------------------------------------------
|  0 | SELECT STATEMENT              |       |      1 |    63 |   103 |
|  1 |  TABLE ACCESS BY INDEX ROWID| T1    |      1 |    63 |   103 |
|  2 |   INDEX RANGE SCAN           | T1_C2 |    100 |       |     3 |
--------------------------------------------------------------------

Execution plan 11.2.0.4
--------------------------------------------------------------------
| Id | Operation                     | Name  | Rows   | Bytes | Cost  |
--------------------------------------------------------------------
|  0 | SELECT STATEMENT              |       |    100 |  6300 |   103 |
|  1 |  TABLE ACCESS BY INDEX ROWID| T1    |    100 |  6300 |   103 |
|  2 |   INDEX RANGE SCAN           | T1_C2 |    100 |       |     3 |
--------------------------------------------------------------------
```

Although the costs of the two plans are the same, the estimated cardinality has gone from badly wrong to correct. In fact, earlier versions of Oracle would have done an even worse job, with an estimated cardinality of the index range scan also showing the value one. You can imagine that in a more complex query 10g might prefer to execute this "expensive" part of the query at the very start of the plan because it should return only one row for further processing; but 11g might prefer to access table t1 at a later point in the query when a cheaper option for accessing the data became available.

Basics

We'll start with a definition: The ***selectivity*** of a predicate (or set of predicates) is the fraction of a data set identified by that predicate; the value will necessarily be between 0 and 1.

Internally the optimizer uses ***selectivity*** to do its arithmetic, externally it displays the ***cardinality***, which is the ***number*** of rows of a data set identified by the predicate. So cardinality = {input number of rows} * selectivity. In many of the example I show, I will use be using the Rows (cardinality) column of execution plans to make comments about selectivity more comprehensible.

There are many cases where the method the optimizer uses to calculate single table selectivity (hence cardinality) is something that seems obvious and reasonable; difficulties start to appear where the optimizer has to start making guesses (or estimates) to cater for a mismatch between the information it has available and the requirement of the query. Let's start with the simplest cases, and then ask a few questions about the slightly harder cases.

There are some basic statistics that the optimizer uses in its calculation – mostly from the view user_tab_cols (and its relatives). The relevant columns are:

num_distinct	number of distinct (non-null) values that appear in the column
density	fraction of rows that match the predicate "column = constant"
num_nulls	number of rows where the column is null
low_value	(representation of) the lowest value seen in the column
high_value	(representation of) the highest value seen in the column

Based on its definition, you will appreciate that the density appears to be closely related to num_distinct: if there are 100 distinct values in a column then "column = constant" should identify 1/100 of the data provided the values are evenly distributed: density = 1/num_distinct. This is, indeed, the case provided the data is evenly distributed and Oracle has not created a histogram showing a skewed distribution in the data. If there is a histogram in place then Oracle tries to "factor out" the popular values and calculate a density that represents the non-popular values, at which point the relationship between density and num_distinct no long holds.

There is one other statistic that Oracle can use to calculate selectivity, and that is the distinct_keys column from user_indexes (and its relatives). Finally, of course, we can mention the num_rows column of user_tables (etc.) that allows Oracle to convert from selectivity to cardinality.

If we wanted to examine histogram information, we would then look at use_tab_histograms (and the related views) to check the columns endpoint_number and endpoint_value (with, optionally endpoint_actual_value); and in 12c with the new, improved, types of histogram endpoint_repeat_count. But histograms are outside the scope of this presentation and paper.

Simple formulae then:

We have already seen that (in the absence of histograms) we can give the selectivity of "column = {constant}" as user_tab_columns.density = 1/user_tab_columns.num_distinct.

If we have a range-based predicate e.g. "column < {constant}", "column between {k1} and {k2}" the basic intent of the formula is: "range you want / total possible range" – where the total possible range is based on user_tab_columns. high_value – user_tab_columns.low_value. In most cases I find that this very informal statement is sufficient when I'm doing a quick estimate, but technically the optimizer does have some special adjustments at about boundaries depending on whether the predicate uses open (>, <) or closed (>=, <=) ranges. So, for example:

Selectivity (col1 < X)	(X - low_value) / (high_value - low_value)
Selectivity (col1 >= Y)	(high_value - Y) / (high_value - low_value) + 1/num_distinct
Selectivity (col1 between X and Y)	(X - Y)/(high_value – low_value) + 2/num_distinct

Note, particularly, that "between" is equivalent to "col1 >= X and col1 <= Y"

Basic questions

The ideas I've summarised above cover just a single column behaving very nicely – but very few queries are that nice. So here are a few questions:

What happens if your predicate involves an "unknown" values ?
What happens if your predicate uses "function(column)" ?
How do you calculate "high_value – low_value" for character types ?
What happens if your predicate uses multiple columns ?
What happens if your predicate is OUTSIDE the low/high range ?

The optimizer has various strategies and tools for dealing with these questions, and the strategies may change over time. In the case of "unknown" values the optimizer will often just "guess" – i.e. use a hard-coded values, commonly 1%, 5%, OR 0.25%, although for indexes it will also use 0.9% and 0.45%; however we can enable dynamic sampling at level 3 or above to eliminate some of these guesses; and we may find a few special cases in later versions of Oracle where the optimizer "pre-computes" parts of a query while optimising it so that it can use constants instead of guesses.

There's a similar issue if you're using predicates involving function(column). Oracle has to guess – again with typical values being 1% or 5% - and again dynamic sampling at level 3 or above may help. In this case, though, we can (in 11g) create virtual columns based on the function declaration and collect stats on that virtual column.

Doing arithmetic on character strings is easy – once you realise that a string is stored as a sequence of bytes; all you have to do is pretend that the bytes represent a number. Unfortunately Oracle does this with only the first 15 bytes of a character column, then rounds the number to 15 (decimal) significant digits before storing it as a statistic. This means that statistically the optimizer know about roughly the first 6 characters of a string – if you're using a single-byte character set; this isn't good if you're trying to store URLs in the database, once Oracle's got through the http:// there's no room for information about the rest of the string.

Predicates on multiple columns start to get interesting because they introduce an intractable problem. Oracle assumes that predicates are independent and uses the formulae of basic probability theory to produce combined selectivity so:

Selectivity(predicateX **AND** predicateY) = selectivity(predicateX) * selectivity(predicateY)

Selectivity(predicateX **OR** predicateY) = selectivity(predicateX) + selectivity(predicateY) – selectivity (predicateX AND predicateY)

= selectivity(predicateX) + selectivity(predicateY) – selectivity(predicateX) * selectivity(predicateY)

But we often construct data and write SQL for which this assumption of independence is false, leading the optimizer to dramatic over-estimates or under-estimates of selectivity. Again we may find that dynamic sampling helps, this time at level 4 or above; however Oracle also gives us the option to create "extended stats" in 11g, most specifically "column group" statistics so that we can at least give the optimizer some information about the number of distinct combinations of columns. At present the "column group" option is limited to no more than 20 sets of stats for a table, though, and there are some odd limitations and bugs (check metalink carefully) with using them.

Out of range

The final question I posed is an interesting one because it can introduce a randomly catastrophic effect on performance. Why might you query for data that (apparently) doesn't exist? There are two key reasons.

First, time passes and data values move on but we don't always refresh statistics in a timely fashion – so, for example, if you have an orders table the order date is going to keep increasing; and a common query might be "tell me about orders placed in the last three days". If you don't gather stats on the orders table at least once every three days that query is eventually going to be asking for data that (apparently) doesn't exist.

Secondly, most systems still use a sample when gathering stats – if you're unlucky a sample may fail to pick up rows which represent the boundary values of the data. If you're on 11g you should (almost certainly) be using the auto_sample_size option for gathering stats with the "approximate NDV" mechanism enabled to bypass this particular problem.

To deal with out-of-range values, Oracle uses a "linear decay" strategy. Informally it looks like the optimizer saying: "there's probably some data there but less than the volume that's in range, and the further out of range you go the less data there should be". Consider the graph in illustration 1:

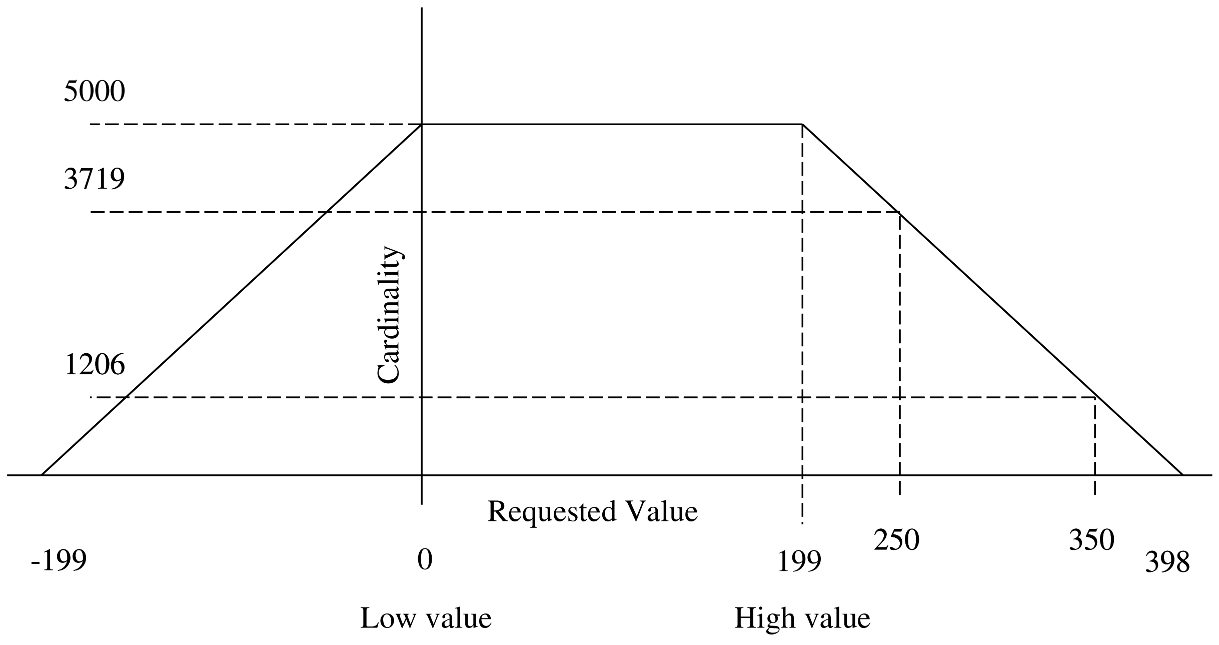

Illustration. 1: modelling out of range predicates

This represents a data set of 1,000,000 rows with a column holding 200 distinct values uniformly distributed between zero and 199, so each value corresponds to 5,000 rows.

To address queries like "columnX = 250" which requests data outside the known range Oracle will calculate the size of the known range and extend the range by that amount to either side, allowing the estimated cardinality to decay in a straight line to zero across that range.

In our example, the known range is (0,199), which Oracle extends to add (199,398) and (-199,0). Oracle then assumes that the cardinality estimates should decrease linearly from 5,000 to 0 across the extended ranges. So, for our predicate "columnX = 250" we can then read off the graph (or do the arithmetic) that the estimated cardinality is 3,719. (Informally, and approximately, 250 is roughly quarter of the way from 199 to 398, so the estimated cardinality should drop roughly by a quarter of the original 5,000.)

In the short term this strategy probably works query well for equality predicates; unfortunately for range-based predicates the optimizer doesn't behave in a way that is consistent with the equality strategy. The predicate "columnX >= 250" uses 1/num_distinct as the selectivity. Apart from the inconsistency you can see how this might easily produced catastrophic execution plans – imagine an order-processing system that gets 5 orders per second. Inside the low/high range a query of the form "how many orders were place in the last 15 minutes" would get a (fairly accurate) estimate of 4,500 but outside the range the estimate would be 5.

In general, monotonic increasing columns (i.e. sequence based or time-based) which are queried with range-based predicates can very easily produce bad cardinality estimates because the statistics go out of date the moment they have been gathered. If you recognise such a situation one solution is write code that calls dbms_stats.set_column_stats() to ensure that the low and high values for the column stats ensure that the expected predicates don't fall outside the range.

Conclusion
In this note I have given you an idea of the basic strategies the optimizer uses to calculate selectivity, and highlighted a few of the weaknesses in the models it is using. I have also made a few suggestions of how you help (or work around) the optimizer's failing once you've spotted the root cause of the problem.

We can recognise that predicates on character-based columns can cause problems because Oracle handles (at best) just the first six characters when doing the basic arithmetic. There's often little that we can do about this beyond forcing execution plans by hinting in some way (perhaps through SQL Profile mechanism).

If we've used functions on columns in predicates we should try to change the code to eliminate the use of the function, but it may be that in some cases dynamic sampling at level 3 will bypass the problem. In some cases we can deal with dependency between columns by using dynamic sampling at level 4, but we also have the "column group" version of extended stats available to use in 11g.

If we run into problems with out of range predicates then we may need to "fake" some statistics to ensure that the predicates never look out of range. This is quite easy to code using the various calls in the dbms_stats package to "set_XXX_stats", but timing may be critical.

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