61.1. Row Estimation Examples

The examples shown below use tables in the PostgreSQL regression test database. The outputs shown are taken from version 8.3. The behavior of earlier (or later) versions might vary. Note also that since `ANALYZE` uses random sampling while producing statistics, the results will change slightly after any new `ANALYZE`.

Let's start with a very simple query:

```
EXPLAIN SELECT * FROM tenk1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..458.00 rows=10000 width=244)
```

How the planner determines the cardinality of `tenk1` is covered in Section 14.2, but is repeated here for completeness. The number of pages and rows is looked up in `pg_class`:

```
SELECT relpages, reltuples FROM pg_class WHERE relname = 'tenk1';

 relpages | reltuples
----------+-----------
      358 |     10000
```

These numbers are current as of the last `VACUUM` or `ANALYZE` on the table. The planner then fetches the actual current number of pages in the table (this is a cheap operation, not requiring a table scan). If that is different from `relpages` then `reltuples` is scaled accordingly to arrive at a current number-of-rows estimate. In the example above, the value of `relpages` is up-to-date so the rows estimate is the same as `reltuples`.

Let's move on to an example with a range condition in its `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000;

                                 QUERY PLAN
-----------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=24.06..394.64 rows=1007 width=244)
   Recheck Cond: (unique1 < 1000)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

The planner examines the `WHERE` clause condition and looks up the selectivity function for the operator < in `pg_operator`. This is held in the column `oprrest`, and the entry in this case is `scalarltsel`. The `scalarltsel` function retrieves the histogram for `unique1` from `pg_statistics`. For manual queries it is

more convenient to look in the simpler `pg_stats` view:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='unique1';

                    histogram_bounds
------------------------------------------------------
 {0,993,1997,3050,4040,5036,5957,7057,8029,9016,9995}
```

Next the fraction of the histogram occupied by "< 1000" is worked out. This is the selectivity. The histogram divides the range into equal frequency buckets, so all we have to do is locate the bucket that our value is in and count part of it and all of the ones before. The value 1000 is clearly in the second bucket (993-1997). Assuming a linear distribution of values inside each bucket, we can calculate the selectivity as:

```
selectivity = (1 + (1000 - bucket[2].min)/(bucket[2].max - bucket[2].min))/num_buckets
            = (1 + (1000 - 993)/(1997 - 993))/10
            = 0.100697
```

that is, one whole bucket plus a linear fraction of the second, divided by the number of buckets. The estimated number of rows can now be calculated as the product of the selectivity and the cardinality of `tenk1`:

```
rows = rel_cardinality * selectivity
     = 10000 * 0.100697
     = 1007  (rounding off)
```

Next let's consider an example with an equality condition in its `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'CRAAAA';

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=30 width=244)
   Filter: (stringu1 = 'CRAAAA'::name)
```

Again the planner examines the `WHERE` clause condition and looks up the selectivity function for `=`, which is `eqsel`. For equality estimation the histogram is not useful; instead the list of *most common values* (MCVs) is used to determine the selectivity. Let's have a look at the MCVs, with some additional columns that will be useful later:

```
SELECT null_frac, n_distinct, most_common_vals, most_common_freqs FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

null_frac         | 0
n_distinct        | 676
most_common_vals  | {EJAAAA,BBAAAA,CRAAAA,FCAAAA,FEAAAA,GSAAAA,JOAAAA,MCAAAA,NAAAAA,WGAAAA}
most_common_freqs | {0.00333333,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003,0.003}
```

Since `CRAAAA` appears in the list of MCVs, the selectivity is merely the corresponding entry in the list of most common frequencies (MCFs):

```
selectivity = mcf[3]
            = 0.003
```

As before, the estimated number of rows is just the product of this with the cardinality of `tenk1`:

```
rows = 10000 * 0.003
     = 30
```

Now consider the same query, but with a constant that is not in the MCV list:

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 = 'xxx';

                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=15 width=244)
   Filter: (stringu1 = 'xxx'::name)
```

This is quite a different problem: how to estimate the selectivity when the value is not in the MCV list. The approach is to use the fact that the value is not in the list, combined with the knowledge of the frequencies for all of the MCVs:

```
selectivity = (1 - sum(mvf))/(num_distinct - num_mcv)
            = (1 - (0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003 +
                    0.003 + 0.003 + 0.003 + 0.003))/(676 - 10)
            = 0.0014559
```

That is, add up all the frequencies for the MCVs and subtract them from one, then divide by the number of other distinct values. This amounts to assuming that the fraction of the column that is not any of the MCVs is evenly distributed among all the other distinct values. Notice that there are no null values so we don't have to worry about those (otherwise we'd subtract the null fraction from the numerator as well). The estimated number of rows is then calculated as usual:

```
rows = 10000 * 0.0014559
     = 15  (rounding off)
```

The previous example with `unique1 < 1000` was an oversimplification of what `scalarltsel` really does; now that we have seen an example of the use of MCVs, we can fill in some more detail. The example was correct as far as it went, because since `unique1` is a unique column it has no MCVs (obviously, no value is any more common than any other value). For a non-unique column, there will normally be both a histogram and an MCV list, and the histogram does not include the portion of the column population represented by the MCVs. We do things this way because it allows more precise estimation. In this situation `scalarltsel` directly applies the condition (e.g., "< 1000") to each value of the MCV list, and adds up the frequencies of the MCVs for which the condition is true. This gives an exact estimate of the selectivity within the portion of the table that is MCVs. The histogram is then used in the same way as above to estimate the selectivity in the portion of the table that is not MCVs, and then the two numbers are combined to estimate the overall selectivity. For example, consider

```
EXPLAIN SELECT * FROM tenk1 WHERE stringu1 < 'IAAAAA';

                         QUERY PLAN
------------------------------------------------------------
 Seq Scan on tenk1  (cost=0.00..483.00 rows=3077 width=244)
   Filter: (stringu1 < 'IAAAAA'::name)
```

We already saw the MCV information for `stringu1`, and here is its histogram:

```
SELECT histogram_bounds FROM pg_stats
WHERE tablename='tenk1' AND attname='stringu1';

                                histogram_bounds
--------------------------------------------------------------------------------
 {AAAAAA,CQAAAA,FRAAAA,IBAAAA,KRAAAA,NFAAAA,PSAAAA,SGAAAA,VAAAAA,XLAAAA,ZZAAAA}
```

Checking the MCV list, we find that the condition `stringu1 < 'IAAAAA'` is satisfied by the first six entries and not the last four, so the selectivity within the MCV part of the population is

```
selectivity = sum(relevant mvfs)
            = 0.00333333 + 0.003 + 0.003 + 0.003 + 0.003 + 0.003
```

```
            = 0.01833333
```

Summing all the MCFs also tells us that the total fraction of the population represented by MCVs is 0.03033333, and therefore the fraction represented by the histogram is 0.96966667 (again, there are no nulls, else we'd have to exclude them here). We can see that the value `IAAAAA` falls nearly at the end of the third histogram bucket. Using some rather cheesy assumptions about the frequency of different characters, the planner arrives at the estimate 0.298387 for the portion of the histogram population that is less than `IAAAAA`. We then combine the estimates for the MCV and non-MCV populations:

```
selectivity = mcv_selectivity + histogram_selectivity * histogram_fraction
            = 0.01833333 + 0.298387 * 0.96966667
            = 0.307669

rows        = 10000 * 0.307669
            = 3077  (rounding off)
```

In this particular example, the correction from the MCV list is fairly small, because the column distribution is actually quite flat (the statistics showing these particular values as being more common than others are mostly due to sampling error). In a more typical case where some values are significantly more common than others, this complicated process gives a useful improvement in accuracy because the selectivity for the most common values is found exactly.

Now let's consider a case with more than one condition in the `WHERE` clause:

```
EXPLAIN SELECT * FROM tenk1 WHERE unique1 < 1000 AND stringu1 = 'xxx';

                                     QUERY PLAN
-------------------------------------------------------------------------------
 Bitmap Heap Scan on tenk1  (cost=23.80..396.91 rows=1 width=244)
   Recheck Cond: (unique1 < 1000)
   Filter: (stringu1 = 'xxx'::name)
   ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..23.80 rows=1007 width=0)
         Index Cond: (unique1 < 1000)
```

The planner assumes that the two conditions are independent, so that the individual selectivities of the clauses can be multiplied together:

```
selectivity = selectivity(unique1 < 1000) * selectivity(stringu1 = 'xxx')
            = 0.100697 * 0.0014559
            = 0.0001466

rows        = 10000 * 0.0001466
            = 1  (rounding off)
```

Notice that the number of rows estimated to be returned from the bitmap index scan reflects only the condition used with the index; this is important since it affects the cost estimate for the subsequent heap fetches.

Finally we will examine a query that involves a join:

```
EXPLAIN SELECT * FROM tenk1 t1, tenk2 t2
WHERE t1.unique1 < 50 AND t1.unique2 = t2.unique2;

                                      QUERY PLAN
--------------------------------------------------------------------------------
 Nested Loop  (cost=4.64..456.23 rows=50 width=488)
   ->  Bitmap Heap Scan on tenk1 t1  (cost=4.64..142.17 rows=50 width=244)
         Recheck Cond: (unique1 < 50)
         ->  Bitmap Index Scan on tenk1_unique1  (cost=0.00..4.63 rows=50 width=0)
```

```
               Index Cond: (unique1 < 50)
   ->  Index Scan using tenk2_unique2 on tenk2 t2  (cost=0.00..6.27 rows=1 width=244)
         Index Cond: (unique2 = t1.unique2)
```

The restriction on `tenk1`, `unique1 < 50`, is evaluated before the nested-loop join. This is handled analogously to the previous range example. This time the value 50 falls into the first bucket of the `unique1` histogram:

```
selectivity = (0 + (50 - bucket[1].min)/(bucket[1].max - bucket[1].min))/num_buckets
            = (0 + (50 - 0)/(993 - 0))/10
            = 0.005035

rows        = 10000 * 0.005035
            = 50  (rounding off)
```

The restriction for the join is `t2.unique2 = t1.unique2`. The operator is just our familiar `=`, however the selectivity function is obtained from the `oprjoin` column of `pg_operator`, and is `eqjoinsel`. `eqjoinsel` looks up the statistical information for both `tenk2` and `tenk1`:

```
SELECT tablename, null_frac,n_distinct, most_common_vals FROM pg_stats
WHERE tablename IN ('tenk1', 'tenk2') AND attname='unique2';

tablename  | null_frac | n_distinct | most_common_vals
-----------+-----------+------------+------------------
 tenk1     |         0 |         -1 |
 tenk2     |         0 |         -1 |
```

In this case there is no MCV information for `unique2` because all the values appear to be unique, so we use an algorithm that relies only on the number of distinct values for both relations together with their null fractions:

```
selectivity = (1 - null_frac1) * (1 - null_frac2) * min(1/num_distinct1, 1/num_distinct2)
            = (1 - 0) * (1 - 0) / max(10000, 10000)
            = 0.0001
```

This is, subtract the null fraction from one for each of the relations, and divide by the maximum of the numbers of distinct values. The number of rows that the join is likely to emit is calculated as the cardinality of the Cartesian product of the two inputs, multiplied by the selectivity:

```
rows = (outer_cardinality * inner_cardinality) * selectivity
     = (50 * 10000) * 0.0001
     = 50
```

Had there been MCV lists for the two columns, `eqjoinsel` would have used direct comparison of the MCV lists to determine the join selectivity within the part of the column populations represented by the MCVs. The estimate for the remainder of the populations follows the same approach shown here.

Notice that we showed `inner_cardinality` as 10000, that is, the unmodified size of `tenk2`. It might appear from inspection of the `EXPLAIN` output that the estimate of join rows comes from 50 * 1, that is, the number of outer rows times the estimated number of rows obtained by each inner index scan on `tenk2`. But this is not the case: the join relation size is estimated before any particular join plan has been considered. If everything is working well then the two ways of estimating the join size will produce about the same answer, but due to round-off error and other factors they sometimes diverge significantly.

For those interested in further details, estimation of the size of a table (before any `WHERE` clauses) is done in `src/backend/optimizer/util/plancat.c`. The generic logic for clause selectivities is in `src/backend/optimizer/path/clausesel.c`. The operator-specific selectivity functions are mostly found in `src/backend/utils/adt/selfuncs.c`.

Prev How the Planner Uses Statistics
Home
Up
Next Appendixes

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.