Database System Architecture (INFO-H417) - Statistics and estimation for range types

December 2021

Chun Han Li (000511879)
Niccolò Morabito (000541608)
Kainaat Amjid (000510449)
Vlada Kylynnyk (000511146)

Introduction to the problem

The purpose of this project is to improve overall statistical collection and cardinality estimation for range types. We need to improve rangetype_typeanalyze.c using statistics, in a way that it improves the already implemented selectivity estimation for strictly left(<<) and overlaps(&&). Moreover, there is no implementation for cardinality estimation of overlap operator, so we need to do that. Lastly, we need to benchmark our implementations to check the performance of our code.

Range types

Range types are data types that contain a range of values. They represent many elements in a single range value. Postgres has following built- in range types Int4range, int8range, numrange, tsrange, tstzrange and daterange. New kinds of range types can also be defined in Postgres.

Every non-empty range has two bounds, lower bound and upper bound. Here we defined the length of a range data as its upper bound minus lower bound.

In inclusive bound, the boundary of bound is included. For inclusive lower and upper bound it is represented by "[" and "]" respectively. While the exclusive boundary of bound is not included. For exclusive lower and upper bound it is represented by "(" and ")" respectively.

In infinite or unbounded ranges upper or lower bounds are omitted or both of the bounds can be omitted. It can be inclusive or exclusive. Omitting lower bound means all the values lower than upper bound are included in range e.g. (, upper-bound] or (, upper-bound) or[, upper-bound] or [, upper-bound). Omitting upper bound means all the values greater than the lower bound are included in range e.g. (lower-bound ,] or (lower-bound ,) or [lower-bound ,) or [lower-bound ,] .

Bound values can be written within double quotes. It is necessary when ranges contain commas, brackets, quotes or backslashes.

Statistics used by the planner

The planner needs to estimate the number of rows retrieved by a query in order to make a good choice of query plans. Pg_class table stores the total number of rows in each table, index, histogram bound array and disk space for each table. This information can be checked by performing a query on the pg_class table. Pg_class contains columns, reltuples and relpages, which contain information about the number of rows and occupied number of pages on disk respectively. These tables usually don't have updated information. They are updated by VACUUM ANALYZE and some DDL commands. VACUUM doesn't scan the whole table at once. Whatever part it scans it updates the pg_table with an approximate value. Planner scales the values it finds in pg_class to approximate the results to the actual table size.

For queries that do not need to access all rows, the planner estimates their selectivity to further optimize the query to generate the cheapest plan tree.

Pg_stats is another table that can be used to examine the statistics manually. It can be accessed by all users. While pg_statistic can only be accessed by super users.

The default limit of histogram bins is 100. This can be changed to make the estimation more accurate at the expense of occupying more space and more calculation time while estimating the selectivity.

Row estimation

“Analyze” uses random sampling, every time it produces statistics, so results will change slightly after running a new analysis. When any query is performed, the number of rows and number of pages are looked up in the pg_class table. Then the planner extracts the actual current number of pages that the table occupies without scanning the whole table. If the number of pages is different from the relpages column value, then the planner scales the reltuples (which contains the estimated number of rows of the table) accordingly.

The histogram contains buckets with equal frequency. To estimate the selectivity of operator < operator it locates the bucket where value is in and counts part of it and all the number of buckets that are before it. And divide by the number of buckets.

To estimate the number of rows it multiplies the selectivity by relative cardinality (number of rows reltuples column of pg_class table contains).

No. of rows = selectivity * relative cardinality

When two conditions are present in a query for example << operator and &&, the planner will assume both conditions are independent of each other. The planner will calculate the individual selectivity of both and multiply them together to get a final estimation of selectivity.

Planner statistics and security

Only superusers can access pg_statistics to restrict ordinary users from accessing the contents of pg_statistics. In order to perform a query, the user either has “select” privilege on the table or the involved columns or the operator should be leak proof. If not then the selectivity estimator will behave as if no statistics are available, and the planner will proceed with default or fall-back assumptions. This restriction applies only to cases where the planner would need to execute a user-defined operator on one or more values from pg_statistic. Thus the planner is permitted to use generic statistical information, such as the fraction of null values or the number of distinct values in a column, regardless of access privileges.(c)

Upper bound and lower bound approach

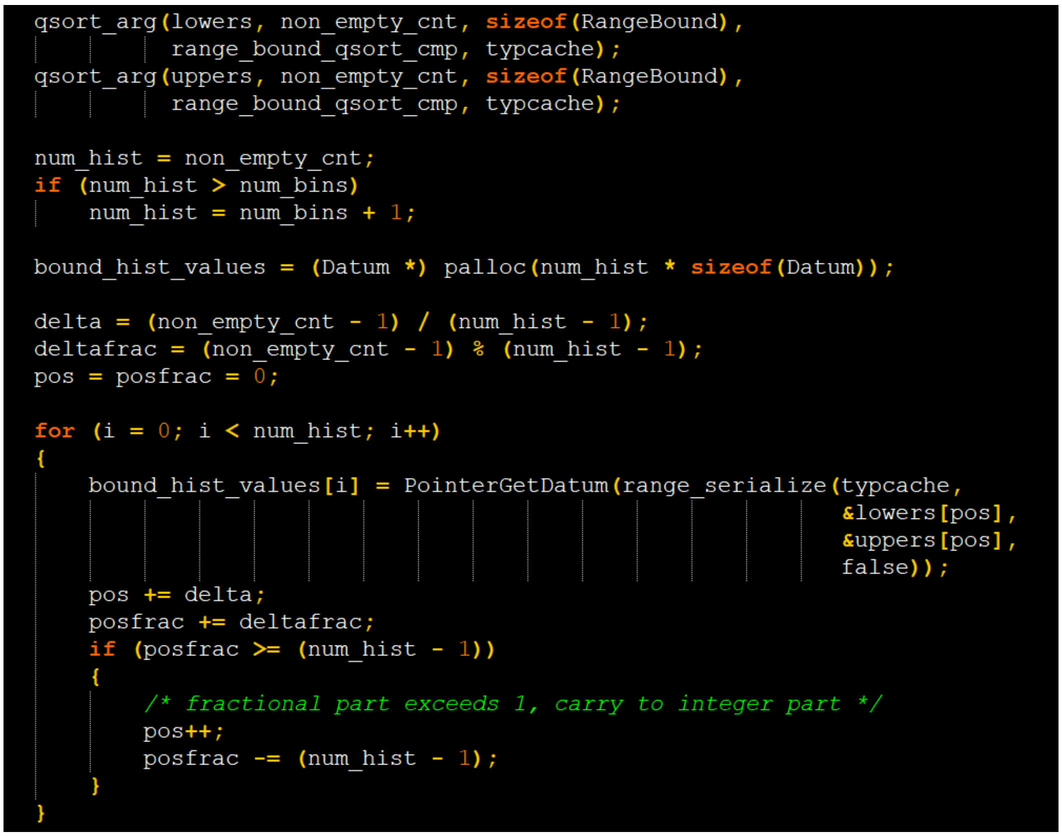
Range typanalyze

Theory

Firstly, it scans the table according to the number of sample rows. Deserialize the range values into upper and lower values. Take the difference between upper and lower value and store it in a length float8 type array. It counts the number of non-empty rows. After that, it stores upper and lower values into upper and lower rangebound type arrays.

Then if the non-empty rows are greater than zero, it calculates null_frac and width statistics. After checking non-empty count greater than 2, it creates bound histogram slots. It sorts the upper and lower arrays. Define the number of bins, default value of the number of buckets in the histogram is 100. If the non-empty count is less than the default number of bins then the number of bins will be equal to non_empty rows. It calculates histogram bins boundaries by calculating the delta = (non_empty_cnt - 1) / (num_hist - 1). After that, it takes values of lower and upper histograms at positions that start from zero. And adds delta to get the next position. It keeps adding delta in position variables until it loops equal to the total number of histogram bins. Then it serialises the lower and upper values. In the end, it stores lower and upper ranges using stats.

Implementation



```
qsort_arg(lowers, non_empty_cnt, sizeof(RangeBound),
          range_bound_qsort_cmp, typcache);
qsort_arg(uppers, non_empty_cnt, sizeof(RangeBound),
          range_bound_qsort_cmp, typcache);

num_hist = non_empty_cnt;
if (num_hist > num_bins)
    num_hist = num_bins + 1;

bound_hist_values = (Datum *) palloc(num_hist * sizeof(Datum));

delta = (non_empty_cnt - 1) / (num_hist - 1);
deltafrac = (non_empty_cnt - 1) % (num_hist - 1);
pos = posfrac = 0;

for (i = 0; i < num_hist; i++)
{
    bound_hist_values[i] = PointerGetDatum(range_serialize(typcache,
                                                           &lowers[pos],
                                                           &uppers[pos],
                                                           false));
    pos += delta;
    posfrac += deltafrac;
    if (posfrac >= (num_hist - 1))
    {
        /* fractional part exceeds 1, carry to integer part */
        pos++;
        posfrac -= (num_hist - 1);
    }
}
```

<< operator selectivity estimation

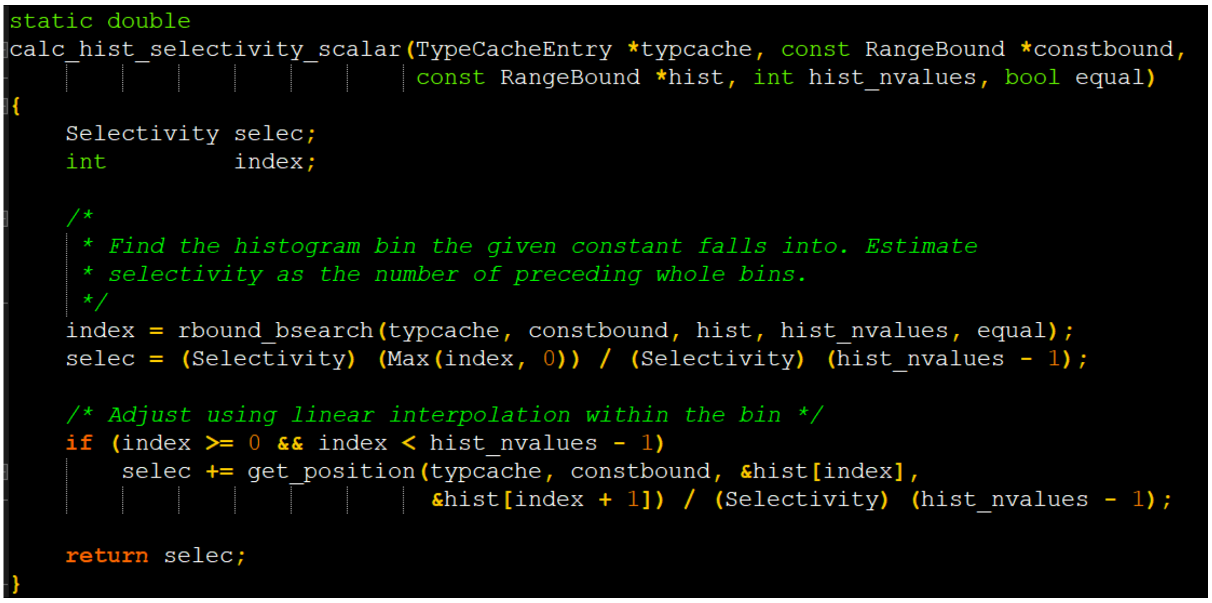
Theory

In rangetype_selfuncs.c, postgres is calculating selectivity of the strictly left operator and overlap operator with a constant range value and a table histograms.

To calculate selectivity estimation of a strictly left operator, it is using lower bound histogram and lower value of constant range only. Firstly, the function does binary search on histogram buckets to find in which bucket the lower value of the constant range falls in. It will return all the buckets before the selected bin and the relative portion of the selected bucket depending upon where the constant range value falls in, then divide it by the total number of buckets to estimate the selectivity.

Implementation

The function calc_hist_selectivity_scalar receives a constant value and a histogram and returns an estimation of selectivity. It first used b-search to find the number of bins lower than the constant value, then used linear interpolation to calculate the selectivity which is lower than the constant value in the bin it located.



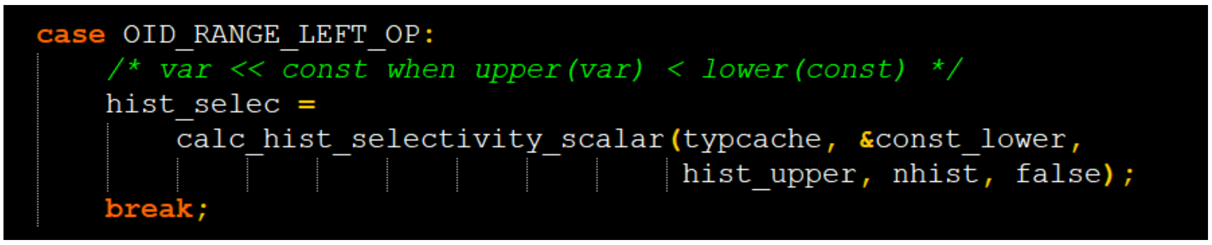
```c
static double
calc_hist_selectivity_scalar(TypeCacheEntry *typcache, const RangeBound *constbound,
                             const RangeBound *hist, int hist_nvalues, bool equal)
{
    Selectivity selec;
    int         index;

    /*
     * Find the histogram bin the given constant falls into. Estimate
     * selectivity as the number of preceding whole bins.
     */
    index = rbound_bsearch(typcache, constbound, hist, hist_nvalues, equal);
    selec = (Selectivity) (Max(index, 0)) / (Selectivity) (hist_nvalues - 1);

    /* Adjust using linear interpolation within the bin */
    if (index >= 0 && index < hist_nvalues - 1)
        selec += get_position(typcache, constbound, &hist[index],
                              &hist[index + 1]) / (Selectivity) (hist_nvalues - 1);

    return selec;
}
```

Therefore, << operator selectivity is just a function call.



```c
  case OID_RANGE_LEFT_OP:
      /* var << const when upper(var) < lower(const) */
      hist_selec =
          calc_hist_selectivity_scalar(typcache, &const_lower,
                                       hist_upper, nhist, false);
      break;
```

&& operator selectivity estimation

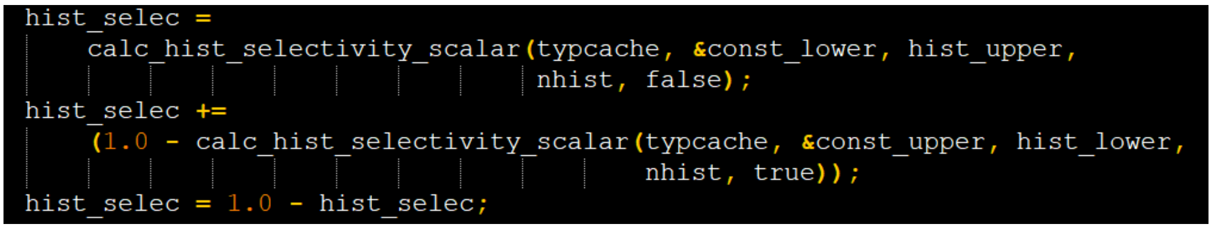
Theory

To calculate selectivity estimation of overlap of a table with single constant range value, it is using following formula A && B <=> NOT (A << B OR A >> B)

It implements the left logic, which is equivalent to the right side. Firstly, it calculates the selectivity estimation of a strictly left operator with constant lower bound value and upper bound histogram of table(A.lower << B table’s_upper_histogram). Then it calculates the selectivity estimation of strictly left operator with constant upper bound value and lower bound histogram of table(A.upper << B table’s_lower_histogram) and minus it from one to get selectivity estimation of strictly right operator (A.upper >> B table’s_lower_histogram). After that, it adds both estimations (A << B OR A >> B) and minus the estimation from one to get an estimation of overlap “NOT (A << B OR A >> B)” which is equivalent to A && B

Implementation

The first one calculated upper histogram << lower value, the second one calculated (1 - lower histogram << upper value)= lower histogram >> upper value. (1 - the first one - the second one)=NOT (upper histogram << lower value OR lower histogram >> upper value)=&& operator.



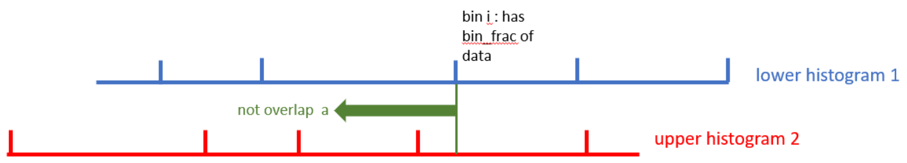
```
hist_selec =
    calc_hist_selectivity_scalar(typcache, &const_lower, hist_upper,
                                 nhist, false);
hist_selec +=
    (1.0 - calc_hist_selectivity_scalar(typcache, &const_upper, hist_lower,
                                        nhist, true));
hist_selec = 1.0 - hist_selec;
```

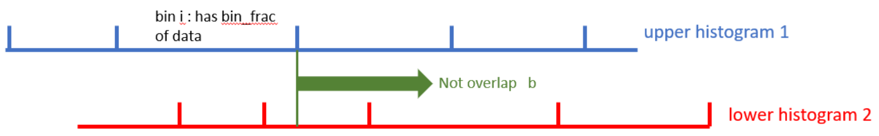
&& operator cardinality estimation

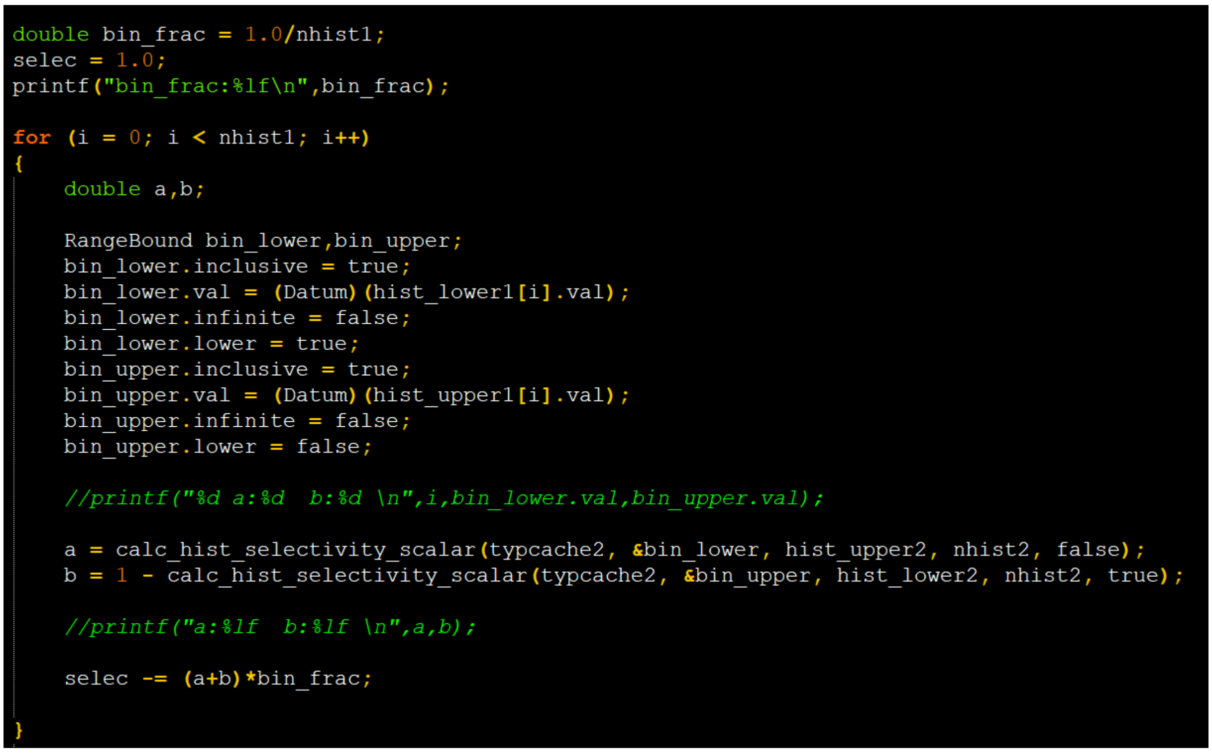
Theory

Since we have made selective estimates of overlap, we extend this concept from one value to multiple values. For every bin of histogram1, calculate the number proportion of one bin over the whole data. Comparing the lower bound of that bin in the lower histogram of the first table with the upper histogram of the second table, we get the proportion of data that this single bin would not overlap, which is the data that is strictly left to all the data within that bin. On the other hand, using the same method, we can get the proportion of data that this single bin would not overlap for strictly right to all the data within that bin. After accumulating the proportion of data in all bins that would not be overlapped, we got the selectivity value.

Implementation







```
double bin_frac = 1.0/nhist1;
selec = 1.0;
printf("bin_frac:%lf\n",bin_frac);

for (i = 0; i < nhist1; i++)
{
    double a,b;

    RangeBound bin_lower,bin_upper;
    bin_lower.inclusive = true;
    bin_lower.val = (Datum)(hist_lower1[i].val);
    bin_lower.infinite = false;
    bin_lower.lower = true;
    bin_upper.inclusive = true;
    bin_upper.val = (Datum)(hist_upper1[i].val);
    bin_upper.infinite = false;
    bin_upper.lower = false;

    //printf("%d a:%d  b:%d \n",i,bin_lower.val,bin_upper.val);

    a = calc_hist_selectivity_scalar(typcache2, &bin_lower, hist_upper2, nhist2, false);
    b = 1 - calc_hist_selectivity_scalar(typcache2, &bin_upper, hist_lower2, nhist2, true);

    //printf("a:%lf  b:%lf \n",a,b);

    selec -= (a+b)*bin_frac;

}
```

Benchmark

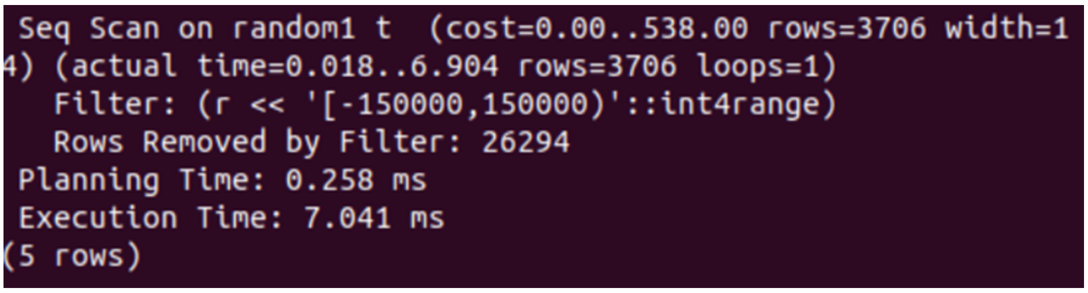
Theory - Data Design

All data is between ±500000, with 30000 rows. We designed five different tables.

- Random table: Randomly selected two bounds.
- Short length(upper bound minus lower bound) table: The length of all range data is less than 10000, which is roughly equal to the width of a bin because the default number of bins is 100.
- Long length table: The length of all data is bigger than 100000, which is roughly equal to ten bins width.
- Small table: The same as the random table but only 300 rows, each bin contains about 3 data.
- Big table: The same as the random table but with 300000 rows, each bin contains about 3000 data.

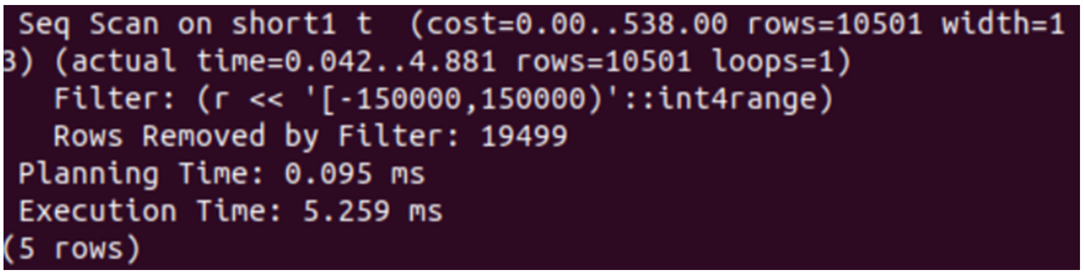
Selectivity estimation for strictly left — table<<(-150000,150000)

Random table — error percentage = 0%



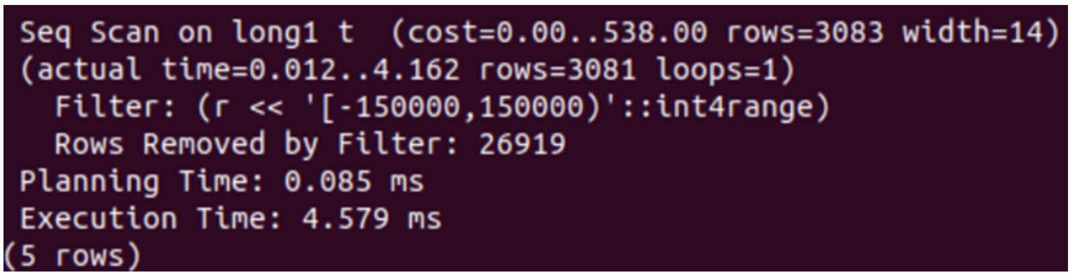
```
 Seq Scan on random1 t  (cost=0.00..538.00 rows=3706 width=1
4) (actual time=0.018..6.904 rows=3706 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 26294
 Planning Time: 0.258 ms
 Execution Time: 7.041 ms
(5 rows)
```

Short length table — error percentage = 0%



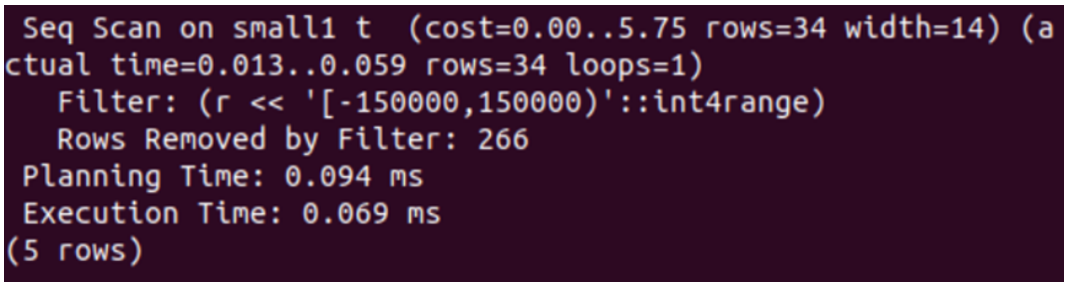
```
 Seq Scan on short1 t  (cost=0.00..538.00 rows=10501 width=1
3) (actual time=0.042..4.881 rows=10501 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 19499
 Planning Time: 0.095 ms
 Execution Time: 5.259 ms
(5 rows)
```

Long length table — error percentage = (3083-3081)/30000 = 0.01%



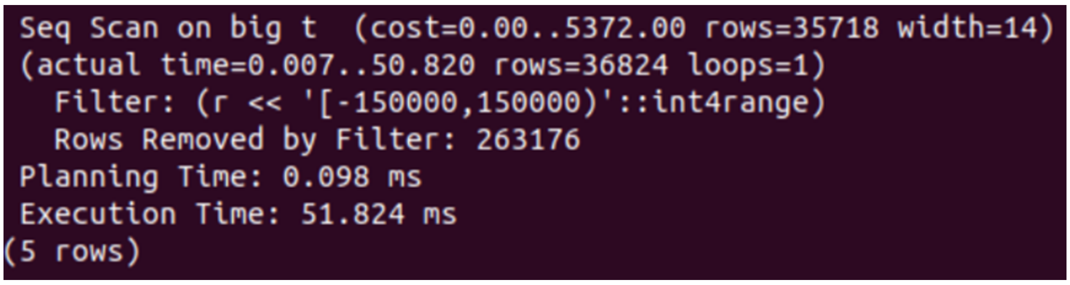
```
 Seq Scan on long1 t  (cost=0.00..538.00 rows=3083 width=14)
 (actual time=0.012..4.162 rows=3081 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 26919
 Planning Time: 0.085 ms
 Execution Time: 4.579 ms
(5 rows)
```

Small table — error percentage = 0%



```
 Seq Scan on small1 t  (cost=0.00..5.75 rows=34 width=14) (a
ctual time=0.013..0.059 rows=34 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 266
 Planning Time: 0.094 ms
 Execution Time: 0.069 ms
(5 rows)
```

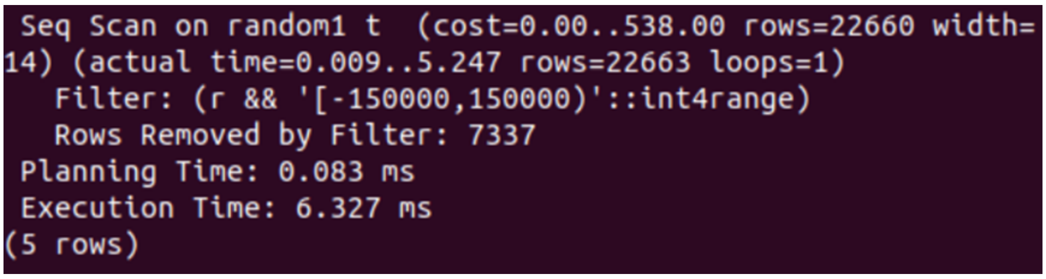
Big table — error percentage = (36824-35718)/300000 = 0.36%



```
 Seq Scan on big t  (cost=0.00..5372.00 rows=35718 width=14)
 (actual time=0.007..50.820 rows=36824 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 263176
 Planning Time: 0.098 ms
 Execution Time: 51.824 ms
(5 rows)
```

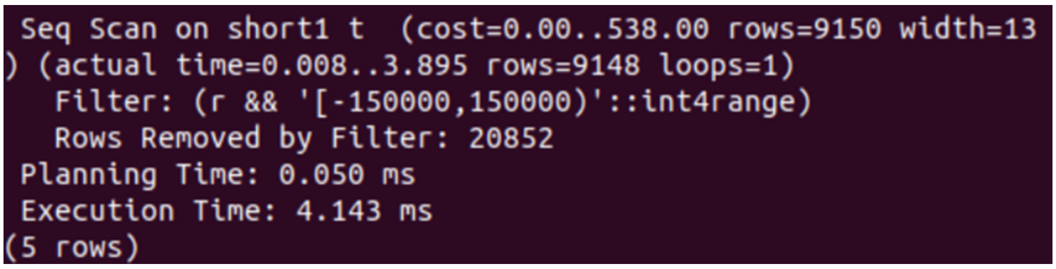
Selectivity estimation for overlap — table&&(-150000,150000)

Random table — error percentage = (22663-22660)/30000=0.01%



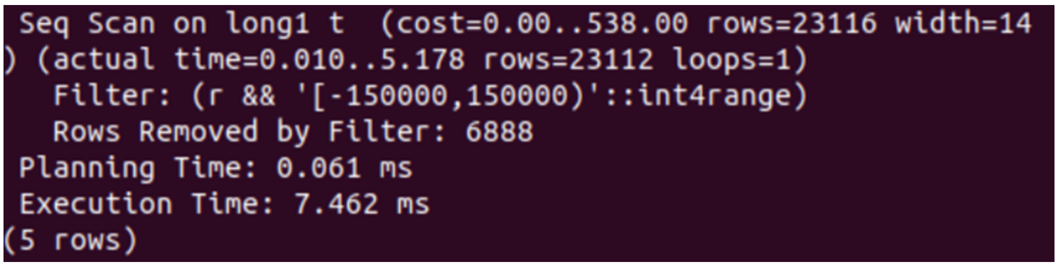
```
 Seq Scan on random1 t  (cost=0.00..538.00 rows=22660 width=
14) (actual time=0.009..5.247 rows=22663 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 7337
 Planning Time: 0.083 ms
 Execution Time: 6.327 ms
(5 rows)
```

Short length table — error percentage = (9150-9148)/30000= 0.007%



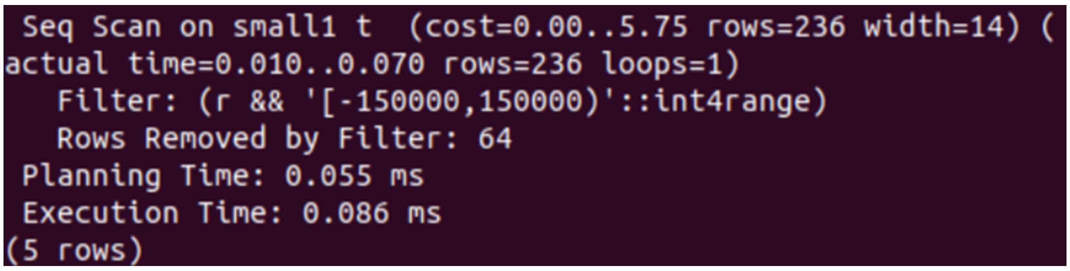
```
 Seq Scan on short1 t  (cost=0.00..538.00 rows=9150 width=13
) (actual time=0.008..3.895 rows=9148 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 20852
 Planning Time: 0.050 ms
 Execution Time: 4.143 ms
(5 rows)
```

Long length table — error percentage = (23116-23112)/30000=0.013%



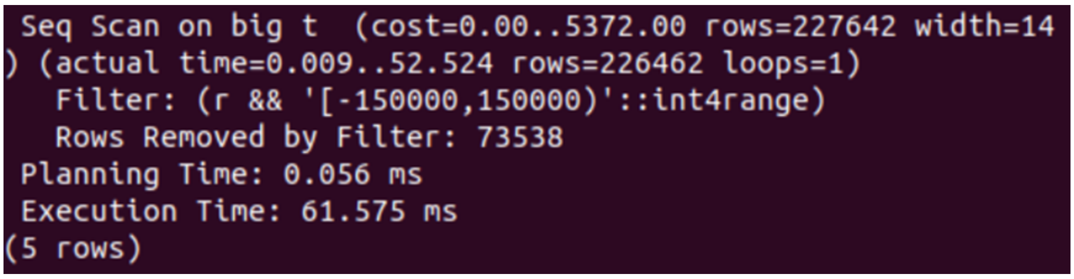
```
 Seq Scan on long1 t  (cost=0.00..538.00 rows=23116 width=14
) (actual time=0.010..5.178 rows=23112 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 6888
 Planning Time: 0.061 ms
 Execution Time: 7.462 ms
(5 rows)
```

Small table — error percentage = 0%



```
 Seq Scan on small1 t  (cost=0.00..5.75 rows=236 width=14) (
actual time=0.010..0.070 rows=236 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 64
 Planning Time: 0.055 ms
 Execution Time: 0.086 ms
(5 rows)
```

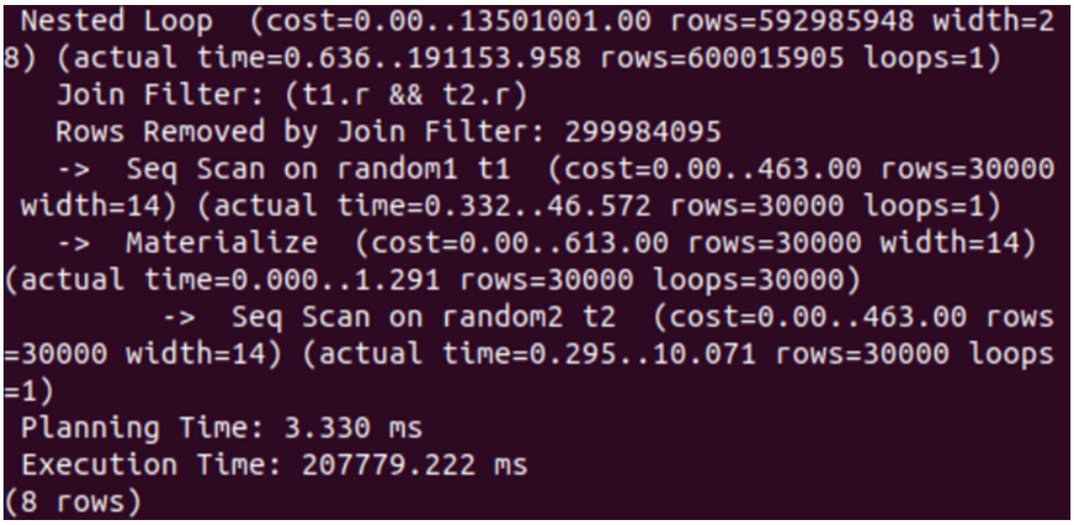
Big table — error percentage = (227642-226462)/300000=0.39%



```
 Seq Scan on big t  (cost=0.00..5372.00 rows=227642 width=14
) (actual time=0.009..52.524 rows=226462 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 73538
 Planning Time: 0.056 ms
 Execution Time: 61.575 ms
(5 rows)
```

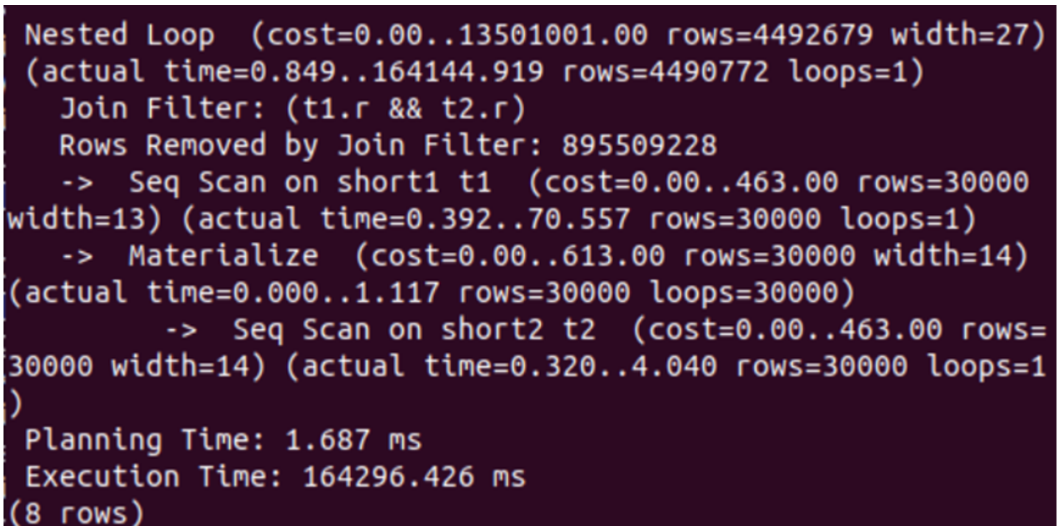
Join cardinality estimation for overlap — table1&&table2

Random table && Random table —
error percentage = (600015905-592985948)/(30000*30000)=0.78%



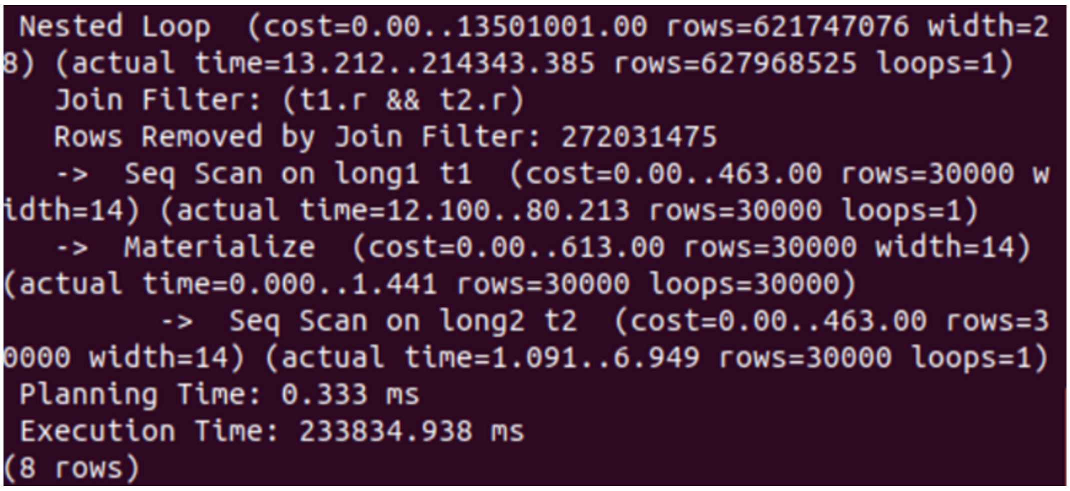
```
 Nested Loop  (cost=0.00..13501001.00 rows=592985948 width=2
8) (actual time=0.636..191153.958 rows=600015905 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 299984095
   ->  Seq Scan on random1 t1  (cost=0.00..463.00 rows=30000
 width=14) (actual time=0.332..46.572 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14)
(actual time=0.000..1.291 rows=30000 loops=30000)
         ->  Seq Scan on random2 t2  (cost=0.00..463.00 rows
=30000 width=14) (actual time=0.295..10.071 rows=30000 loops
=1)
 Planning Time: 3.330 ms
 Execution Time: 207779.222 ms
(8 rows)
```

Short length table && Short length table —
error percentage = 1-(4492679-4490772)/(30000*30000)=0.0002%



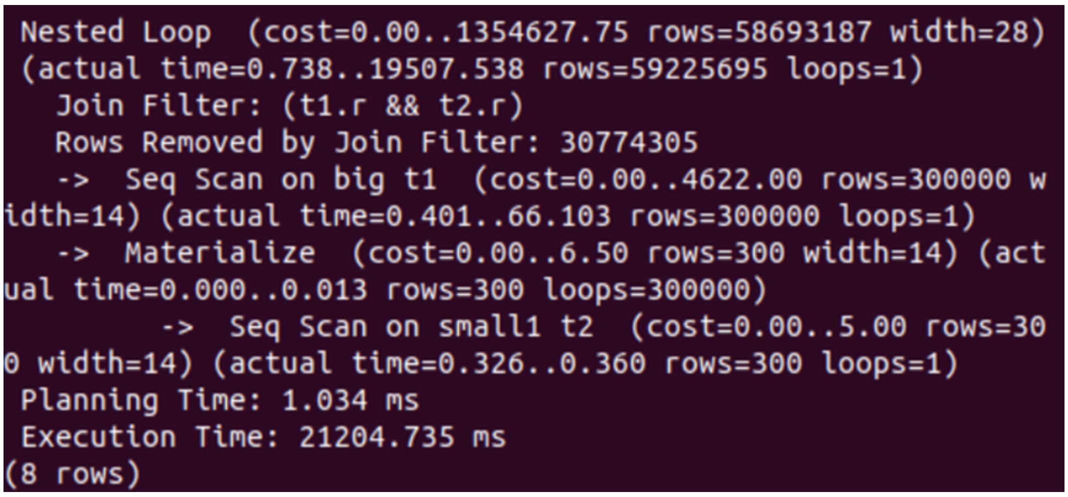
```
 Nested Loop  (cost=0.00..13501001.00 rows=4492679 width=27)
 (actual time=0.849..164144.919 rows=4490772 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 895509228
   ->  Seq Scan on short1 t1  (cost=0.00..463.00 rows=30000
width=13) (actual time=0.392..70.557 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14)
(actual time=0.000..1.117 rows=30000 loops=30000)
         ->  Seq Scan on short2 t2  (cost=0.00..463.00 rows=
30000 width=14) (actual time=0.320..4.040 rows=30000 loops=1
)
 Planning Time: 1.687 ms
 Execution Time: 164296.426 ms
(8 rows)
```

Long length table && Long length table —
error percentage = (627968525-621747076)/(30000*30000)=0.7%



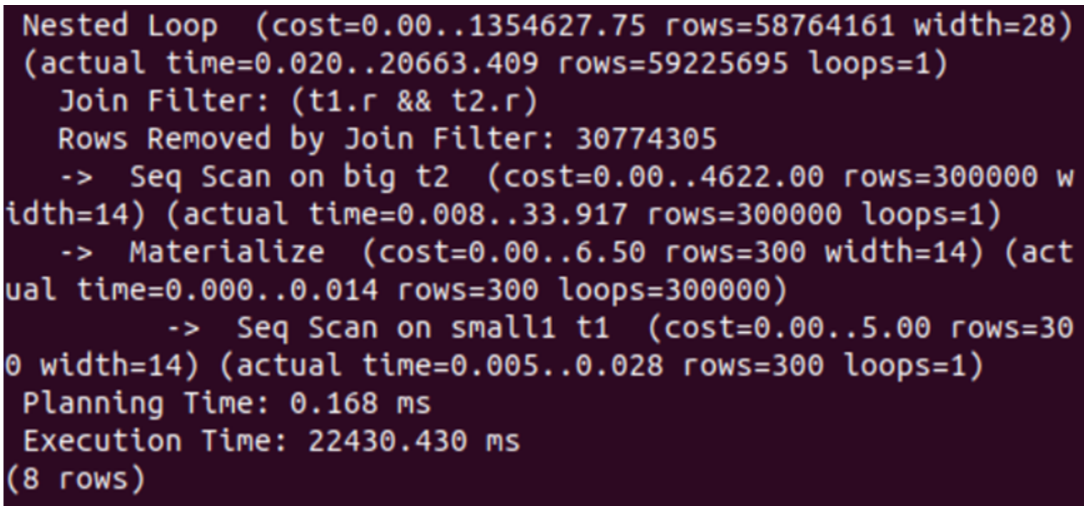
```
 Nested Loop  (cost=0.00..13501001.00 rows=621747076 width=2
8) (actual time=13.212..214343.385 rows=627968525 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 272031475
   ->  Seq Scan on long1 t1  (cost=0.00..463.00 rows=30000 w
idth=14) (actual time=12.100..80.213 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14)
(actual time=0.000..1.441 rows=30000 loops=30000)
         ->  Seq Scan on long2 t2  (cost=0.00..463.00 rows=3
0000 width=14) (actual time=1.091..6.949 rows=30000 loops=1)
 Planning Time: 0.333 ms
 Execution Time: 233834.938 ms
(8 rows)
```

Big table && Small table — error percentage = (59225695-58693187)/(300000*300)=0.59%



```
 Nested Loop  (cost=0.00..1354627.75 rows=58693187 width=28)
 (actual time=0.738..19507.538 rows=59225695 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 30774305
   ->  Seq Scan on big t1  (cost=0.00..4622.00 rows=300000 w
idth=14) (actual time=0.401..66.103 rows=300000 loops=1)
   ->  Materialize  (cost=0.00..6.50 rows=300 width=14) (act
ual time=0.000..0.013 rows=300 loops=300000)
         ->  Seq Scan on small1 t2  (cost=0.00..5.00 rows=30
0 width=14) (actual time=0.326..0.360 rows=300 loops=1)
 Planning Time: 1.034 ms
 Execution Time: 21204.735 ms
(8 rows)
```

Small table && Big table — error percentage = (59225695-58764161)/(300*300000)=0.51%



```
 Nested Loop  (cost=0.00..1354627.75 rows=58764161 width=28)
 (actual time=0.020..20663.409 rows=59225695 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 30774305
   ->  Seq Scan on big t2  (cost=0.00..4622.00 rows=300000 w
idth=14) (actual time=0.008..33.917 rows=300000 loops=1)
   ->  Materialize  (cost=0.00..6.50 rows=300 width=14) (act
ual time=0.000..0.014 rows=300 loops=300000)
         ->  Seq Scan on small1 t1  (cost=0.00..5.00 rows=30
0 width=14) (actual time=0.005..0.028 rows=300 loops=1)
 Planning Time: 0.168 ms
 Execution Time: 22430.430 ms
(8 rows)
```

Benchmark conclusion

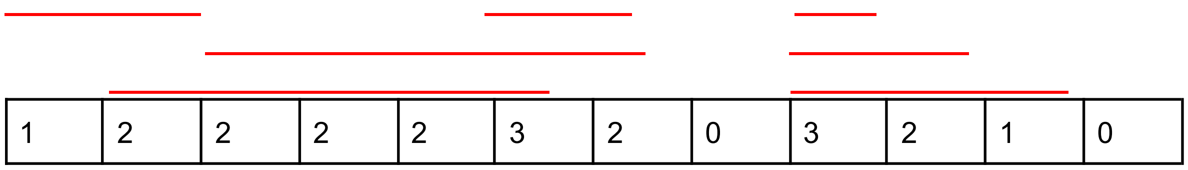
- The error percentage of all results is smaller than 1%, and all the planning time is less than 3.5 milliseconds.
- The only error using these histograms comes from the estimating error when using linear interpolation to calculate the selectivity within a single bin. In order to use linear interpolation, we assumed the data distribution of a single bin follows a uniform distribution, which obviously won't be the case.
- The planning times are just a few milliseconds. On the other hand, all planning times are less than 0.002% of the entire execution time. Therefore, users won't feel that there is any cost to this estimation.

Frequency approach

Theory

This approach is based on the frequency of ranges, meaning that the collected statistics are composed of an equi-width histogram where each bin contains the number of ranges overlapping that bin.
For instance, let's consider two relations R1 and R2, each of them having a range-type column. The first relation is composed of 7 ranges, each of them represented as a red line.

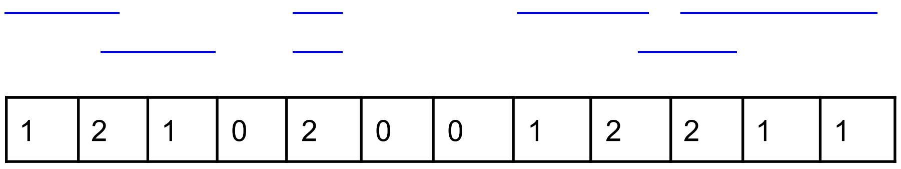
R1



1	2	2	2	2	3	2	0	3	2	1	0

Similarly, the second relation has a range-type column with 7 tuples represented as blue lines.

R2



1	2	1	0	2	0	0	1	2	2	1	1

The starting point of the histogram is the minimum lowest bound between all the ranges and the ending point is the biggest upper bound. The whole space is then split into equal-width bins. In the example above, both the histograms have 12 bins. However, the bin width will be different because the extent is smaller. The next step is to count how many ranges intersect every cell, and then the statistics are ready to be used for estimating selectivity and join cardinality.

The selectivity for the strictly left operator with a range [a, b] can be computed by counting the frequencies in all the bins [bin_start, bin_end] where bin_end < a, because all the ranges that intersect with these bins are necessary before the range [a, b]. However, some ranges could satisfy the condition even in the following bin, and linear interpolation could be used to improve the estimation. In any case, the number obtained with this counting is taking into account the same ranges more times (each of them contributes to the frequency of all the bins that overlap), therefore it must be normalized dividing for the sum of the frequency of all the bins. The resulting value should represent an estimation of the number of selected rows.

For the strictly right operator, the approach is analogous. After computing the selectivity for the two operators, it is possible to obtain also the selectivity for the overlap operator.

For the join cardinality estimation, on the other hand, the approach is slightly more complex. There are two frequency histograms, one for each relation, and we need to estimate the number of ranges between the two relations that overlap. After aligning the two histograms (because they could have different starting points), the estimation corresponds to the sum of the products between the frequencies of two overlapping bins. In the example below, we consider the same histograms as before supposing that the starting point is the same for simplicity.

1	2	2	2	2	3	2	0	3	2	1	0

1	2	1	0	2	0	0	1	2	2	1	1

In this case, the total number of overlapping ranges is :
1+2+4+2+2+4+4+3+4+4+3+3 = 36 tuples
Of course, also in this case this number must be normalized dividing it for the product between the sum of the frequencies of all the bins.

Implementation

The implementation part can be split into three parts, corresponding to the three files that have been changed:

- Creation of a frequency histogram (in `rangetypes_typanalyze.c`)
- Selectivity estimation functions using the new frequency histogram (in `rangetypes_selfuncs.c`)
- Join estimation function using the new frequency histogram (in `geo_selfuncs.c`)

Creation of histogram

Since the frequency histogram described above is equi-width, it is necessary to find its starting and its ending values and then compute the bin width as the difference between these two values divided by the number of bins we want to use.
The starting point of the histogram is the minimum lower bound of the ranges in the tables; similarly, the ending point is the maximum upper bound of the ranges.

After that, for each bin the histogram is divided into, a frequency value is computed by counting all the ranges in the column that are overlapping the bin:

```
start_hist = min(lower_bounds);
end_hist = max(upper_bounds);
bin_width = (end_hist - start_hist) / num_hist;

histogram <- array
for (i = 0; i < num_hist; i++) {
   bin_start = start_hist + bin_width*i
   bin_end = bin_start + bin_width
   frequency = 0
```

```
    for (range in column) {
        // if overlaps
        if (range.lower_bound <= bin_end && range.upper_bound >= bin_start) {
            frequency++
        }
    }
    histogram[i] = frequency
}
```

As this pseudocode shows, the frequency number set in position `i` of the histogram, where the `i`-th bin is from `a` to `b=a+bin_width`, is equal to the number of ranges that overlap `[a, b]`. Obviously, the implementation of this simple algorithm could be optimized reducing the number of total iterations. The complexity of the pseudocode above is O(n*b), where n is the number of rows in the column and b is the number of bins; however, since the number of bins is generally a little constant integer number, it could be considered linear.
The resulting histogram is then stored in memory for further use, and it is now possible to run the `vacuum tablename` command to save the updated statistics in the `pg_stats` table.

For the second and the third point, i.e. for computing some selectivity estimation from the frequency histogram, it is nevertheless necessary to store other values too. In particular, it is important to be able to retrieve the boundaries of each bin. In order to do so, we saved the starting and the ending values of the frequency histogram taking into account that from these values it is possible to extract the bin width again as explained before.
Therefore, another simple statistic is stored in memory: a one-element array containing the range `[histogram_start, histogram_end].`

Moreover, another number could be important as well: the sum of all the frequencies in the histogram. This is going to be used to normalize the result obtained by summing the frequencies for estimation, since each range can be counted more times, depending on its length (the larger it is, the more times the range is counted). This number can be easily obtained from the histogram itself before estimating selectivity. Therefore, the only reason for storing it is for efficiency reasons: the statistics are collected less frequently (whenever the `vacuum` command is run or the statistics are updated automatically), while the code for selectivity or join estimation is run for every query. However, we should also consider that the complexity of the approach is not changing since it is generally necessary to iterate on the whole frequency histogram in any case and since the number of bins of the histogram is very low.

Selectivity estimation functions

The assignment requires implementing the selectivity estimation function for two operators: strictly left (<<) and overlap (`&&`). Since the overlap estimation is directly correlated with the strictly left and strictly right operators, the implemented code computes a double for these two cases, while the overlap selectivity can be easily obtained in the following way.

Since `A && B ⇔ NOT (A<<B or A>>B)` and since `A<<B` and `A>>B` are mutually exclusive (i.e. we can sum their probabilities to find the probability of `A<<B or A>>B`), we can say that:

```
  selectivity(A&&B) = 1 - [selectivity(A<<B) + selectivity(A>>B)]
```

For strictly left and strictly right operators, the selectivity is estimated by counting the frequencies in the bins that are respectively before or after the lower or the upper bound of the range we are selecting on (`constvalue`). In particular, the pseudocode for the strictly left selectivity estimation is the following:

```
index, count = 0

while (hist_start + bin_width*(index+1) < constval.lower_bound && index<nbins) {
   count += histogram[index]
   index += 1
}
```

while the cycle changes a bit in case the selectivity estimation is based on the strictly right operator:

```
index, count= 0

while (hist_start + bin_width*(nbins-index) > constval.upper_bound && index<nbins){
   count += histogram[nbins-index-1]
   index += 1
}
```

Of course, the resulting counter does not represent the number of rows we expect the query to select. This basically depends on what has been explained before: each range can be counted in more bins, and this simple sum should be normalized by dividing it by the total frequencies number (the sum of the frequencies for all the bins). In this way, the resulting value must be in [0, 1] and represents the percentage of the rows that could be selected. E.g., if `constvalue = [a, b]` and `a` is greater than the histogram ending point (the bigger upper bound of the column), the strictly right estimation on `constvalue` will return `1`. As `a` becomes closer to the histogram starting point, the resulting selectivity value will become closer to `0`.

In order to be more accurate, it is possible to consider also the linear interpolation of the bin which contains `constvalue.lower_bound` (or `constvalue.upper_bound` for strictly right) and add a percentage of the corresponding frequency value (according to the intersection point between the bin bounds and the value).

Join estimation function

Finally, the last code to implement must compute a selectivity estimation value for join between two tables on a range column using the overlap operator. After retrieving the frequency histogram and its boundaries from each table, the algorithm should compare the two histograms (taking into account that each of them could have different starting and ending points and different bin widths) and sum out the product of the frequencies for each intersection.

First of all, we need to ignore all those bins at the beginning of the histogram with the smaller starting point that are not overlapping with any of the other histogram's bins. This is done by the first two ifs in the pseudocode below. In case the two histograms do not overlap, the cycle ends with `counter=0`.

On the other hand, in case there is an overlap we need to compute the product between the two frequencies for each overlapping bin and we distinguish three cases according to the ending points of the bins. If the `i1`-th bin of the first histogram overlaps with the `i2`-th bin of the second histogram:

- `bin_end1 <  bin_end2`
  Then also the (i1+1)-th bin of histogram1 is going to overlap the i2-th bin of histogram2, therefore the following bin of histogram1 should be taken into account next;
- `bin_end1 > bin_end2`
  Then the (i2+1)-th bin of histogram2 is going to overlap the i1-th bin of histogram1, therefore the following bin of histogram2 should be taken into account next;
- `bin_end1 = bin_end2`
  In this case, neither the (i1+1)-th bin of histogram1 can overlap the i2-th bin of histogram2 nor the (i2+1)-th bin of histogram2 can overlap the i1-th bin of histogram1, therefore both the indices should be incremented to take into account the next bins.

```
hist_start1, hist_start2 <- starting point of the two histograms
bin_width1, bin_width2 <- bin width of the two histograms

count, i1, i2 = 0
bin_start1 = hist_start1
bin_start2 = hist_start2

while (i1<nbins1 && i2<nbins2) {
   bin_end2 = bin_start2 + bin_width2
   if (bin_end2<=bin_start1) {
       bin_start2 = bin_end2
       i2 += 1
       continue
   }

   bin_end1 = bin_start1 + bin_width1;
   if (bin_end1 <= bin_start2) {
       bin_start1 = bin_end1
       i1 += 1
       continue
   }
   count += histogram1[i1] * histogram2[i2];

   if (bin_end1 < bin_end2) {
       // increment in the first histogram
       bin_start1 = bin_end1
       i1 += 1
   } else if (bin_end1 == bin_end2) {
       // increment in both the histograms
       bin_start1 = bin_end1
       i1 += 1
       bin_start2 = bin_end2
       i2 += 1
   } else {
       // increment in the second histogram
       bin_start2 = bin_end2
       i2 += 1
   }
}
```

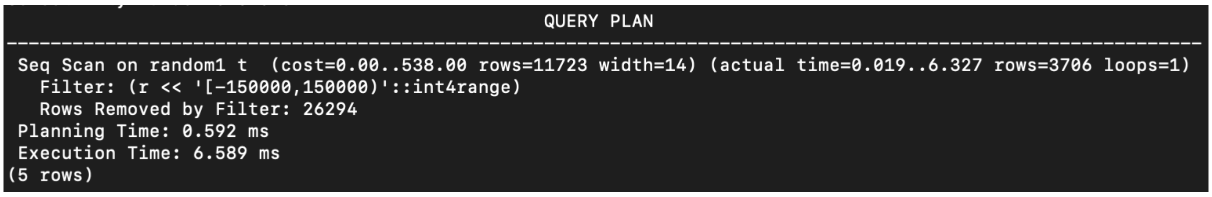
Benchmark

The data and the queries used to test this implementation are the same used and described for the other approach.

Selectivity estimation for strictly left — table<<(-150000,150000)

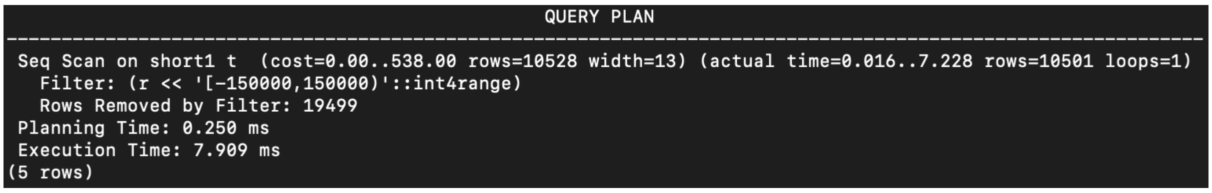
abs(true-estimated)/true

Random table — error-percentage = (11723-3706)/30000 = 26.72%



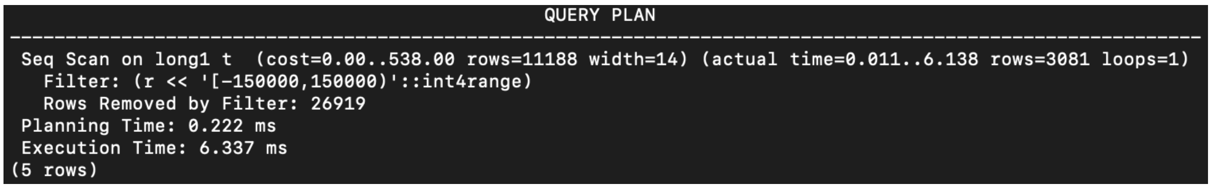
```
                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on random1 t  (cost=0.00..538.00 rows=11723 width=14) (actual time=0.019..6.327 rows=3706 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 26294
 Planning Time: 0.592 ms
 Execution Time: 6.589 ms
(5 rows)
```

Short length table — error-percentage = (10528-10501)/30000 = 0.09%



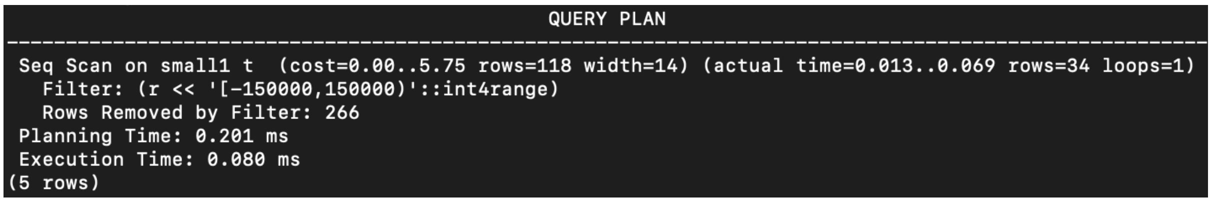
```
                                                    QUERY PLAN
---------------------------------------------------------------------------------------------------------------
 Seq Scan on short1 t  (cost=0.00..538.00 rows=10528 width=13) (actual time=0.016..7.228 rows=10501 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 19499
 Planning Time: 0.250 ms
 Execution Time: 7.909 ms
(5 rows)
```

Long length table — error-percentage = (11188-3081)/30000 = 27.02%



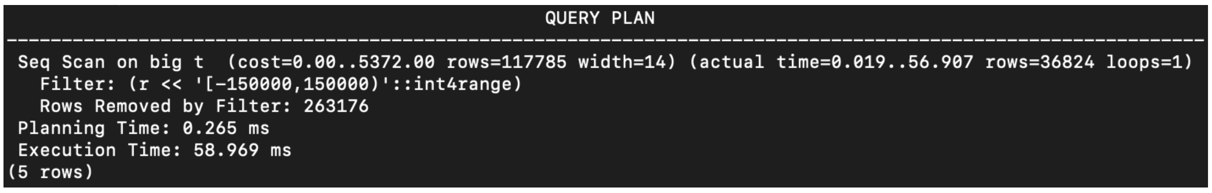
```
                                                QUERY PLAN
----------------------------------------------------------------------------------------------------------
 Seq Scan on long1 t  (cost=0.00..538.00 rows=11188 width=14) (actual time=0.011..6.138 rows=3081 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 26919
 Planning Time: 0.222 ms
 Execution Time: 6.337 ms
(5 rows)
```

Small table — error-percentage = (118-34)/300 = 28%



```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------
 Seq Scan on small1 t  (cost=0.00..5.75 rows=118 width=14) (actual time=0.013..0.069 rows=34 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 266
 Planning Time: 0.201 ms
 Execution Time: 0.080 ms
(5 rows)
```

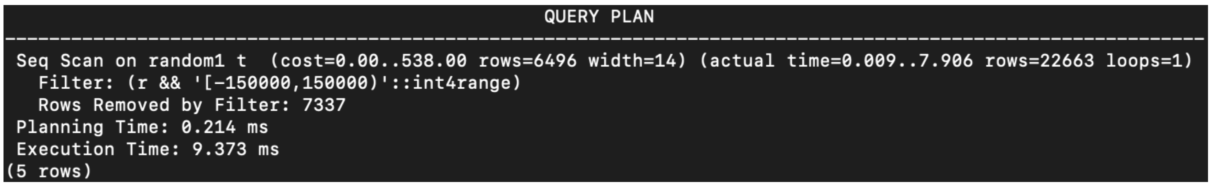
Big table — error-percentage = (117785-36824)/300000 = 27%



```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on big t  (cost=0.00..5372.00 rows=117785 width=14) (actual time=0.019..56.907 rows=36824 loops=1)
   Filter: (r << '[-150000,150000)'::int4range)
   Rows Removed by Filter: 263176
 Planning Time: 0.265 ms
 Execution Time: 58.969 ms
(5 rows)
```

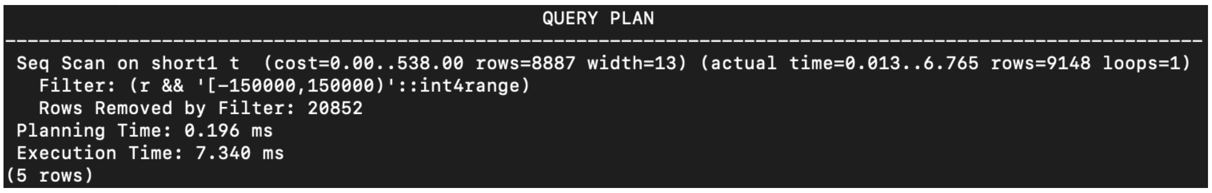
Selectivity estimation for overlap — table&&(-150000,150000)

Random table — error-percentage = (22663-6496)/30000 = 53.89%



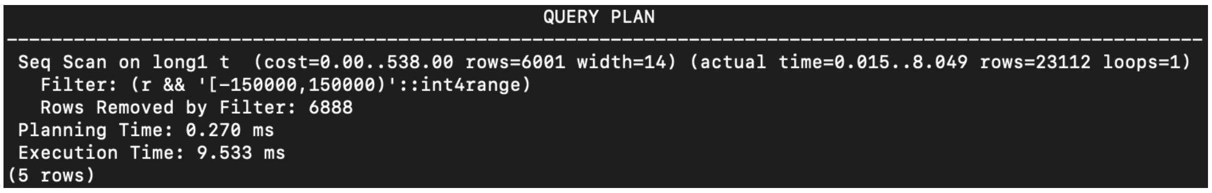
```
                                                QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on random1 t  (cost=0.00..538.00 rows=6496 width=14) (actual time=0.009..7.906 rows=22663 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 7337
 Planning Time: 0.214 ms
 Execution Time: 9.373 ms
(5 rows)
```

Short length table — error-percentage = (9148-8887)/30000 = 0.87%



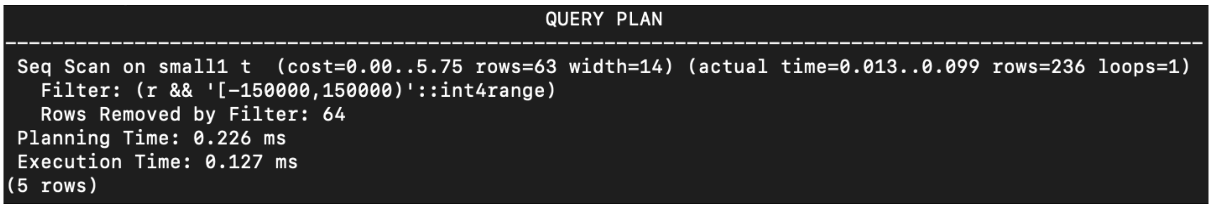
```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on short1 t  (cost=0.00..538.00 rows=8887 width=13) (actual time=0.013..6.765 rows=9148 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 20852
 Planning Time: 0.196 ms
 Execution Time: 7.340 ms
(5 rows)
```

Long length table — error-percentage = (23112-6001)/30000 = 57.04%



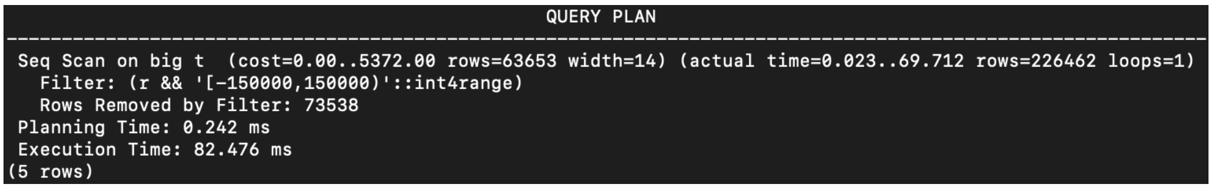
```
                                                QUERY PLAN
-----------------------------------------------------------------------------------------------------------
 Seq Scan on long1 t  (cost=0.00..538.00 rows=6001 width=14) (actual time=0.015..8.049 rows=23112 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 6888
 Planning Time: 0.270 ms
 Execution Time: 9.533 ms
(5 rows)
```

Small table — error-percentage = (236-63)/300 = 57.67%



```
                                                  QUERY PLAN
---------------------------------------------------------------------------------------------------------
 Seq Scan on small1 t  (cost=0.00..5.75 rows=63 width=14) (actual time=0.013..0.099 rows=236 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 64
 Planning Time: 0.226 ms
 Execution Time: 0.127 ms
(5 rows)
```

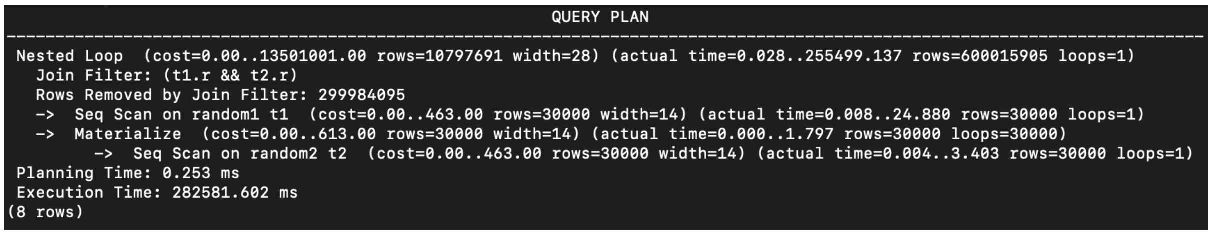
Big table — error-percentage = (226462-63653)/300000 = 54.27%



```
                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
 Seq Scan on big t  (cost=0.00..5372.00 rows=63653 width=14) (actual time=0.023..69.712 rows=226462 loops=1)
   Filter: (r && '[-150000,150000)'::int4range)
   Rows Removed by Filter: 73538
 Planning Time: 0.242 ms
 Execution Time: 82.476 ms
(5 rows)
```

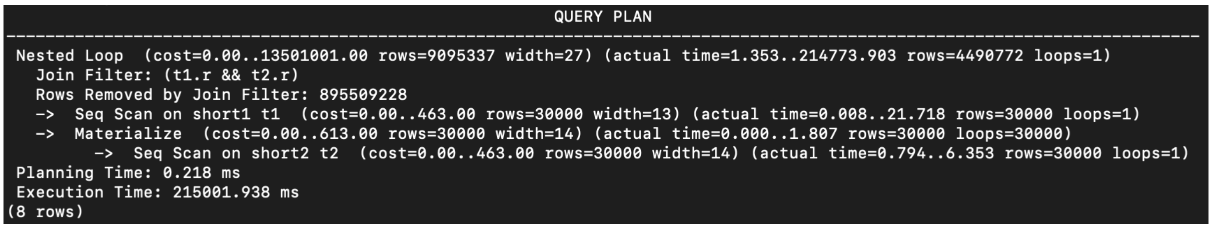
Join cardinality estimation for overlap — table1&&table2

Random table && Random table — error-percentage = (600015905-10797691)/(30000*30000) = 65.47%



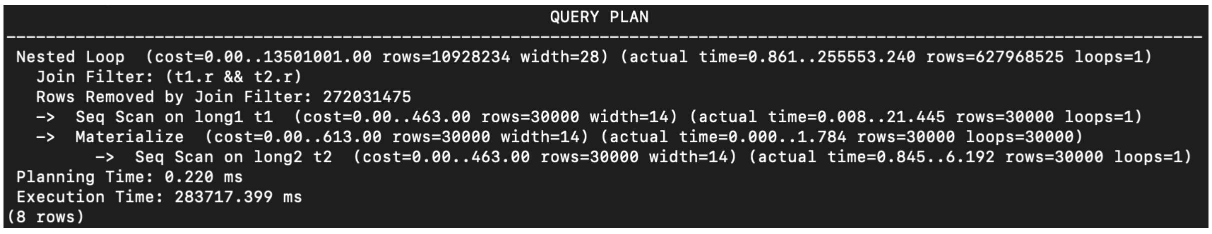
```
                                                           QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..13501001.00 rows=10797691 width=28) (actual time=0.028..255499.137 rows=600015905 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 299984095
   ->  Seq Scan on random1 t1  (cost=0.00..463.00 rows=30000 width=14) (actual time=0.008..24.880 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14) (actual time=0.000..1.797 rows=30000 loops=30000)
         ->  Seq Scan on random2 t2  (cost=0.00..463.00 rows=30000 width=14) (actual time=0.004..3.403 rows=30000 loops=1)
 Planning Time: 0.253 ms
 Execution Time: 282581.602 ms
(8 rows)
```

Short length table && Short length table — error-percentage = (9095337-4490772)/(30000*30000) = 0.51%



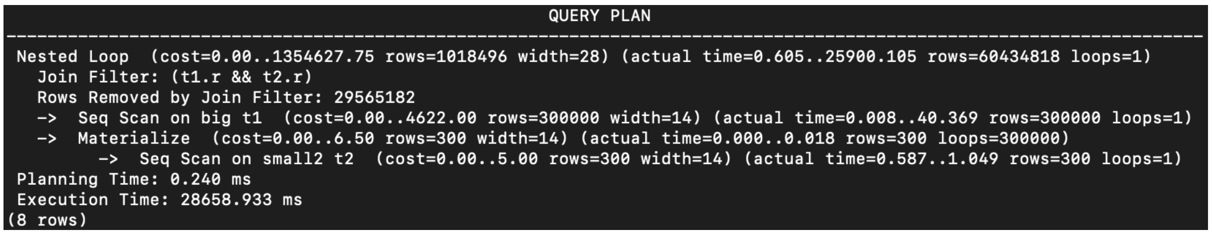
```
                                                           QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..13501001.00 rows=9095337 width=27) (actual time=1.353..214773.903 rows=4490772 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 895509228
   ->  Seq Scan on short1 t1  (cost=0.00..463.00 rows=30000 width=13) (actual time=0.008..21.718 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14) (actual time=0.000..1.807 rows=30000 loops=30000)
         ->  Seq Scan on short2 t2  (cost=0.00..463.00 rows=30000 width=14) (actual time=0.794..6.353 rows=30000 loops=1)
 Planning Time: 0.218 ms
 Execution Time: 215001.938 ms
(8 rows)
```

Long length table && Long length table — error-percentage = (627968525-10928234)/(30000*30000) = 68.56%



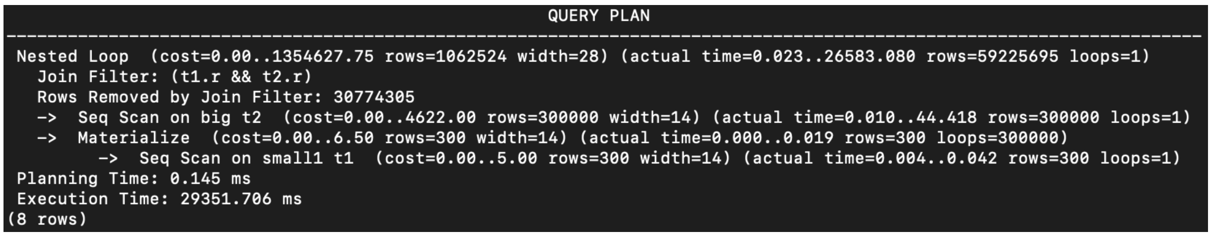
```
                                                           QUERY PLAN
--------------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..13501001.00 rows=10928234 width=28) (actual time=0.861..255553.240 rows=627968525 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 272031475
   ->  Seq Scan on long1 t1  (cost=0.00..463.00 rows=30000 width=14) (actual time=0.008..21.445 rows=30000 loops=1)
   ->  Materialize  (cost=0.00..613.00 rows=30000 width=14) (actual time=0.000..1.784 rows=30000 loops=30000)
         ->  Seq Scan on long2 t2  (cost=0.00..463.00 rows=30000 width=14) (actual time=0.845..6.192 rows=30000 loops=1)
 Planning Time: 0.220 ms
 Execution Time: 283717.399 ms
(8 rows)
```

Big table && Small table — error-percentage = (60434818-1018496)(300000*300) = 66%



```
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..1354627.75 rows=1018496 width=28) (actual time=0.605..25900.105 rows=60434818 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 29565182
   ->  Seq Scan on big t1  (cost=0.00..4622.00 rows=300000 width=14) (actual time=0.008..40.369 rows=300000 loops=1)
   ->  Materialize  (cost=0.00..6.50 rows=300 width=14) (actual time=0.000..0.018 rows=300 loops=300000)
         ->  Seq Scan on small2 t2  (cost=0.00..5.00 rows=300 width=14) (actual time=0.587..1.049 rows=300 loops=1)
 Planning Time: 0.240 ms
 Execution Time: 28658.933 ms
(8 rows)
```

Small table && Big table — error-percentage = (59225695-1062524)/(300*300000) = 64.6%



```
                                                 QUERY PLAN
------------------------------------------------------------------------------------------------------------------------
 Nested Loop  (cost=0.00..1354627.75 rows=1062524 width=28) (actual time=0.023..26583.080 rows=59225695 loops=1)
   Join Filter: (t1.r && t2.r)
   Rows Removed by Join Filter: 30774305
   ->  Seq Scan on big t2  (cost=0.00..4622.00 rows=300000 width=14) (actual time=0.010..44.418 rows=300000 loops=1)
   ->  Materialize  (cost=0.00..6.50 rows=300 width=14) (actual time=0.000..0.019 rows=300 loops=300000)
         ->  Seq Scan on small1 t1  (cost=0.00..5.00 rows=300 width=14) (actual time=0.004..0.042 rows=300 loops=1)
 Planning Time: 0.145 ms
 Execution Time: 29351.706 ms
(8 rows)
```

Benchmark conclusion

Looking at the execution results for the different queries run on different tables and types, you can notice that the general error percentage of the strictly left selectivity function is around 27% (except for one query which showed almost a perfect result), while this percentage increases drastically when the selectivity is performed on the overlap operator.

The first difference is that the general tendency for the strictly left selectivity estimation function is to overestimate the number of rows that are going to be selected, while the overlap join estimation has an opposite behaviour, tending to underestimate. Considering that the join estimation is computed inversely from the strictly left and right functions, as already explained before, and considering that also the strictly right selectivity is overestimated, the explanation for this behaviour is evident. In addition to this, it is important to see how adding interpolation (similarly to what is currently done in the PostgreSQL code) would not improve the results, since it would increase the number of estimated rows for both the operators, worsening the error on both the selectivity functions that are being tested.

On the other hand, the overlap join estimation shows weak results, grossly underestimating the number of rows (the error percentage is more than 65% for almost all the queries).

In general, the benchmark lead to much less accurate results than the upper and lower bounds approach for all the queries, and these results should not surprise taking into account that the frequency approach has an important flaw: the collected statistics (i.e. the frequency histogram) counts the same range more times according to its length. This approach, in addition to worsening the general accuracy (even if the results are normalized, it is clear how it cannot really represent the overall composition and distribution of the data), makes the benchmark really depending on the data which it is performed on. In fact, it is possible to notice that all queries on the tables with short ranges had very good results, showing that the approach could perform very well with data with a particular distribution.

However, for the purpose of this project, the data has been generated randomly to reduce the presence of bias (so that table should be considered only randomness) and the results have shown that only the upper and lower bounds approach is able to perform very well on all the kinds of data and queries, and this should be obviously preferred on the frequency approach.

With regards to the planning time, it is possible to notice that the time required to compute the estimation is negligible: even in this case, most of the queries require less than 1 millisecond, and the ratio between planning and execution time is close to 0. More complex and accurate tests should be done to understand if this approach is faster than the one based on upper and lower bounds. However, even a potential improvement in planning time would not justify such a performance loss, therefore the first approach should be preferred anyway.

Work Cited

c, postgres. “Documentation: 13: 71.3. Planner Statistics and Security.” *PostgreSQL*, https://www.postgresql.org/docs/13/planner-stats-security.html. Accessed 12 December 2021.