PostgreSQL Range Type Query Planning: Theory and Implementation of Equal Width Histograms with Polynomial Regression

INFO-H417 – Database Systems Architecture

Students

Adam Broniewski | abro0001
Victor Diví | vdiv0001
Sergio Postigo | spos0001
Filip Sotiroski | fsot0001

Professor

Mahmoud Sakr

ULB UNIVERSITÉ LIBRE DE BRUXELLES

December 2021

Table of Contents

List of Tables

List of Figures

1 Introduction

PostgreSQL is an open source Database Management System (DBMS), which provides the opportunity for the community of developers to continuously contribute to enhance existing functionalities and add new capabilities. While this DBMS is already robust and versatile for many types of applications, they are also many areas that can be improved. Without doubts, one of the most important functions that deserves the community's attention is the query planner.

The main goal of the query planner is to generate an optimal execution plan for a specific query. That is, decide on one way to execute a query among many possible options giving the same result. To do this, the planner runs estimations about the computational cost of each possible solution and takes the optimal one. For this task, the planner relies on statistics collected about the columns of the tables. These statistics will be of different kind according to the data type of the columns. For example, tables including columns with range type attributes will have statistics containing, equi-depth histograms about its ranges' bounds.

Range types in PostgreSQL are used to contain intervals of values with different subtypes. While the query planner performs acceptable estimations with the statistics collected for range types, this only happens for queries statements with some clauses and operands. There is here an interesting opportunity of improvement to address.

This report will focus on analyzing and improving the way PostgreSQL performs the cost estimations based on statistics for queries involving range datatypes. Specifically, it will study the *selectivity* estimation for queries with filtering clauses and the operands "overlap" (&&) and "strictly left of" (<<). Furthermore, the report will also address the *cardinality* estimation in queries with inner join clauses and the operand "overlap" (&&). The objective is to improve the accuracy of estimations of these types of queries. This will be done by developing a new theoretical estimation model, implementing it in PostgreSQL source code, and benchmarking query estimation cost against algorithms.

The report will start by providing a background about PostgreSQL query performance on range type data focusing on cost estimations, *selectivity* and *cardinality* concepts, and the improvement opportunities. Next, the proposals and implementations to challenge the previously identified issues will be described. And finally, a benchmark will be completed to compare the performance of the new implementation against the existing method.

2 Background

2.1 PostgreSQL Performance

Query performance is a measure of how quickly and efficiently a SQL instruction performs a specific function. It is one of the most important aspects to considerate when choosing a DBMS for specific applications. In PostgreSQL, query performance can be affected by many factors, which can be gathered into two general categories: those that can be controlled by the user like query optimization, and those that depend on the design of the system [1]. Inside the last group is the query planner.

2.1.1 Query Plan (or Execution Plan)

It is a list of actions that the DBMS needs to follow to execute a query on the data. It may be represented by a relational algebra statement. There can be many possible query plans for a single query and thus, PostgreSQL has to choose which one to use. This is done by the query planner.

2.1.2 Query planner

It is a component of the DBMS that chooses the most optimal query plan for each query. In simple words, the query planner will select the query plan that demands the less computational cost, and thus, the one that takes the less time. In order to achieve this, the query planner has to perform an estimation about the query plan's cost. That estimation will be based on statistics about the tables that the query is addressing.

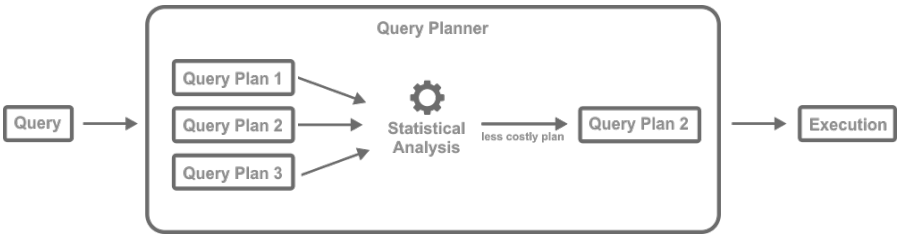


Figure 2.1: PostgreSQL Query planner Operation Schema

PostgreSQL provides the user a way to get information about the query plan that the planner has selected for any specific query. This functionality is invoked with the command EXPLAIN.

2.1.3 Information About the Plan: The EXPLAIN Command

EXPLAIN displays a visualization of the query plan that the planner selects for any query statement. Like it was mentioned before, the query plan includes a list of actions, among which we can find [2]:

1) Scans: How the table(s) in the query will be scanned (sequential scan, index scan, index only scan or bitmap index scan)
2) Joins: If multiple tables are referenced in the query, what methods will be used to get the required rows from each of them (nested loop, hash join or merge join).
3) Sorts: How will a sorting be performed (external merge sort, quick sort, heap sort, etc.)

Actions are organized in nodes. Each node corresponds to one action. There can be nodes nested inside other nodes, which means actions that depend on other actions. This last gives the idea of a tree [3].

The Figure 2.2 shows the output of the EXPLAIN command for a simple query:

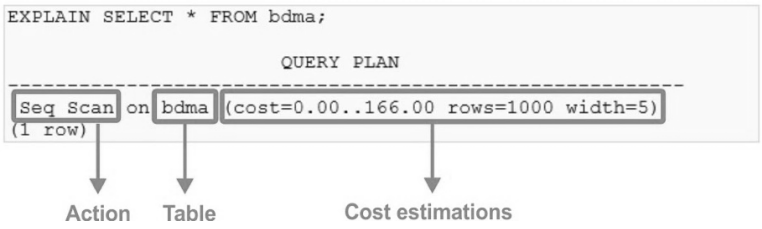


Figure 2.2: EXPLAIN command output details

As it can be seen, the plan's cost estimations include four magnitudes, which will be detailed in the next subsection.

2.1.4 Cost Estimations

The query planner uses the Cost Estimations to decide which Query Plan to execute. As it was seen, the EXPLAIN command details this estimations including four magnitudes [3]:

- cost:
  - *Estimated start-up cost*: the cost before the node starts executing and the cost between the start of the execution and the writing of the first row of the output (0.00 in Figure 2.2).
  - *Estimated total cost:* the cost from when the node starts executing till it finishes writing all output's rows (166.00 in Figure 2.2).
- *rows:* Estimated number of rows from the node's output (1000 in Figure 2.2). Also known as cardinality.
- *width:* Estimated average size in bytes of an output's row (5 in Figure 2.2).

Further information can be obtained by including the option ANALYZE after the EXPLAIN command. This option will execute the planned query to get additional information, including the query output's row count. The accuracy of the planner's estimations can be measured by comparing the estimated output's row count versus the actual row count.

Additional information is also the actual query execution time and the planning time, which is the time in which the planner computes the estimations.

As stated earlier, the query planner generates all its estimates based on pre-calculated statistics available for each table that the query is addressing.

2.1.5 Statistics

PostgreSQL includes functionalities to collect statistics about the tables of a database. These statistics are consumed by the query planner to produce estimations about execution costs. The statistics are stored in system tables (Figure 2.3).

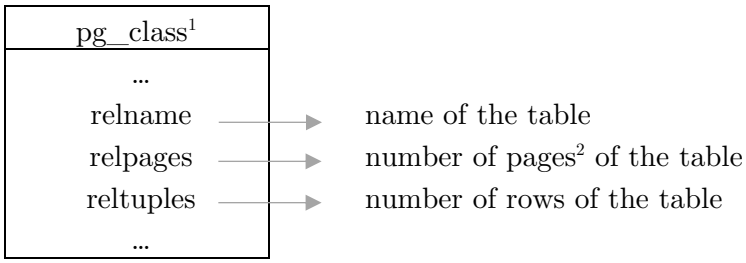


Figure 2.3: Storing general PostgreSQL statistics in pg_class.

Each row in pg_class corresponds to another system table, and it contains general information about it. Relevant for the estimations are the columns *relpages* and *reltuples*, which are detailed in the above schema.

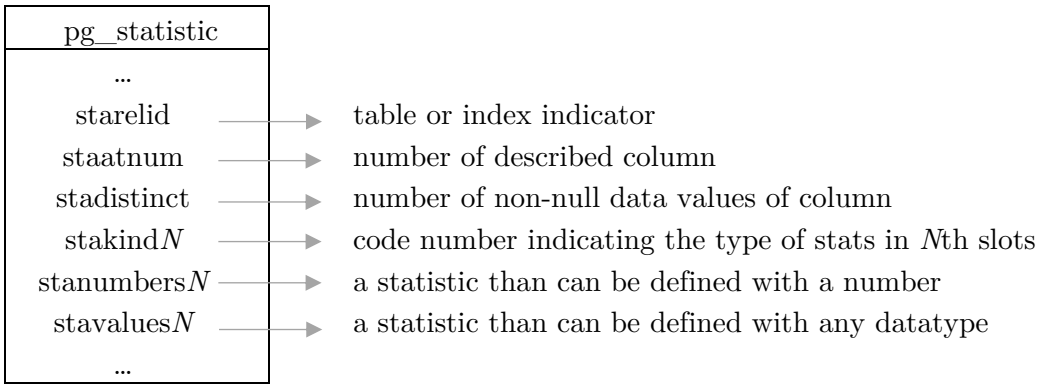


Figure 2.4: Storing table specific information in pg_statistic.

Each row in pg_statistic (Figure 2.4) corresponds to one column of one table in the database. Since different columns in a table may have different datatypes and ranges of values, it makes sense to collect different statistics for each of them. Because of this pg_statistic has free "slots" which are no more than customizable columns able to contain custom statistics for each row. All the columns with the same *"N"* denomination are features of the same collected statistic. There can be up to 5 groups of "slots" ($N = 1, 2, 3, 4, 5$) [4].

[1] While it isn't a purely statistics table, we included it here because it contains relevant information for estimations.
[2] Block of 8kb of data from a table.

This table is generated by running the command VACCUM ANALYZE followed by the table name.

Since the table pg_statistic may contain data that is considered business sensitive[3], read access requires special permissions. However, there is a public view of this table generated called pg_stats (Figure 2.5), which mimics the read permissions of the rest of the database. In essence it contains the same information as pg_statistic, but it is presented in a more readable way and omits the columns that may be considered sensitive [4].

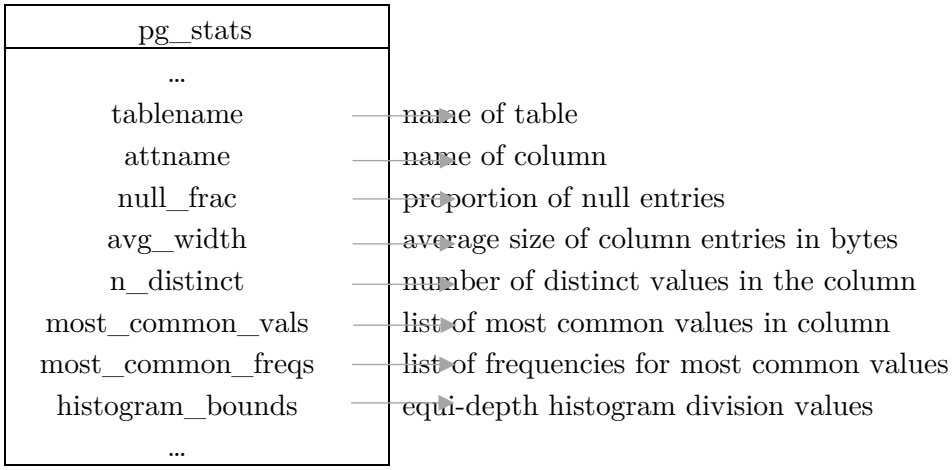


Figure 2.5: Example of information include in pg_stats table.

Each row in pg_statistics and pg_stats contain statistics which may be of different type depending on the data type of the addressing table's column. So, the statistical analysis performed by the query planner will be different depending on the column's datatypes that the query focuses on. In the next section we target tables with columns of type “range”.

2.2 Range Types

PosgreSQL includes some built-in data subtypes for representing ranges of values of specific types [5]. These are detailed as follows:

- *int4range:* Range of *integer* values
- *int8range:* Range of *bigint* values
- *numrange:* Range of *numeric* values
- *tsrange:* Range of *timestamp* values without timezone
- *tstzrange:* Range of *timestamp* values with timezone
- *daterange:* Range of *date* values

As it is common to all data types in PostgreSQL, the range data types have some associated operators that can be applied over them.

[3] For example, summarized salary data for a company.

2.2.1 Operators

Whether the user wants to compare two ranges or check if an element is present in a range, PostgreSQL offers a list of operators applicable to the built-in range data types. Relevant for our research are the *overlap* (&&) and the *strictly left of* (<<) operators.

The *overlap* (&&) operator is applied to compare two ranges and find out is there are any common points between them — it checks whether they are intersected. Table 2-1 shows examples and what the operator would return.

int8range(1,10) && int8range(100,110)	False
int4range(1,10) && int4range(9,100)	True

Table 2-1: Overlap (&&) usage examples

The *strictly left of* (<<) operator is applied to compare two ranges and find out if the first one is completely at the left side of the second one. In other words, it looks if the upper bound of the first one is less than the lower bound of the second one (Table 2-2).

int8range(1,10) << int8range(100,110)	True
int4range(1,10) << int4range(9,100)	False

Table 2-2: Strictly Left of (<<) usage examples

When using these operators in a query addressing columns with range datatypes, some specific statistics are consumed by the query planner to perform the cost estimations.

2.2.2 Statistics

The types of statistics, and the functions used by PostgreSQL to complete the collection of statistics for each column in a table varies depending on the data type of the columns. In the case of range data type, the statistics are collected with the script range_typanalyze.c[4] and stored in the pg_statistic table [6]. The most relevant statistics for this data type are shown in Figure 2.6.

[4] postgres/src/backend/utils/adt/rangetypes_typanalyze.c

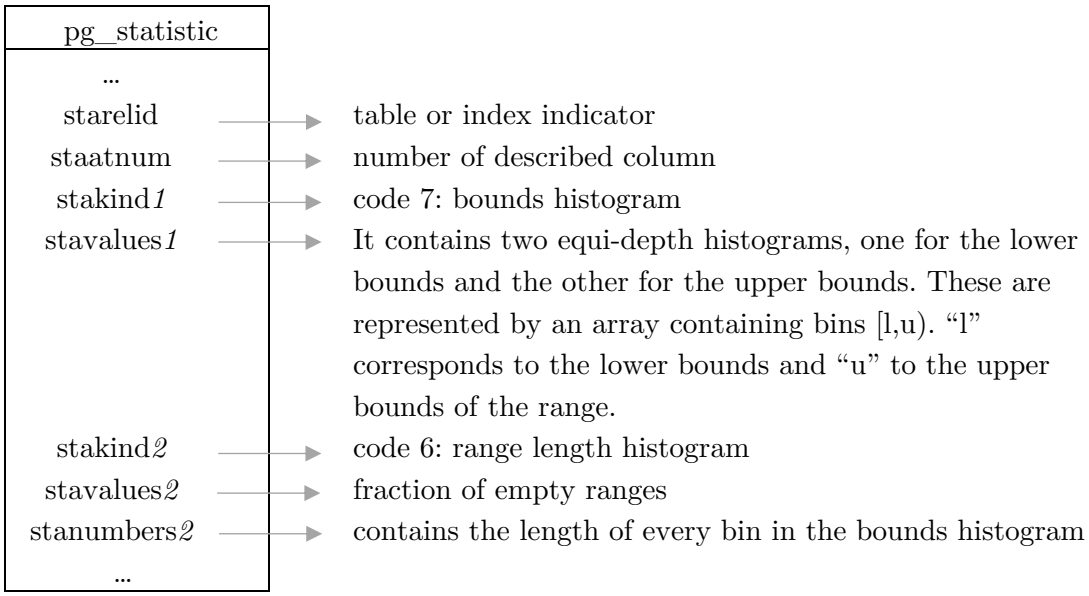


Figure 2.6: Relevant statistics collected in pg_statistic table for query estimation on range types.

A key concept to have in mind is that the histograms are generated by taking a sample of the column's values. The sample can take as maximum 30,000 values from the column [7].

The use of the pg_statistic is necessary to provide the cost estimation for queries involving range data types. With them, the query planner should be able to have an accurate prediction of how many rows a node or a nested group of nodes will have.

2.2.3 Cardinality and Selectivity

The measure of how many rows a table (in this case the output table of a node) will have is called *cardinality*. The way the planner performs the *cardinality* estimation for queries involving range types depends on the clauses and operators used in the statement. Three examples are presented below[5].

2.2.3.1 Example 1 – Filter Strictly Left of (<<) Ranges

```
SELECT *
FROM table1
WHERE range_attr1 << int8range(18,24)
```

Here, the planner looks inside the file *rangetypes_selfuncs.c* and runs the function *calc_hist_selectivity*. The two histograms are generated as follows:

Assume that stavalues1[6] = {(0, 0), (10,16), (17, 20), (25, 34), (30, 40)}

[5] Being table1 and table2 tables with attributes of range data type (range_attr1 and range_attr2 respectively) and 1000 rows respectively. We assume that the VACCUM ANALYZE function was performed for table1 and table 2. So, we have the statistics for these tables.
[6] Statistic from range_attr1

Two histograms are generated (Figure 2.7).

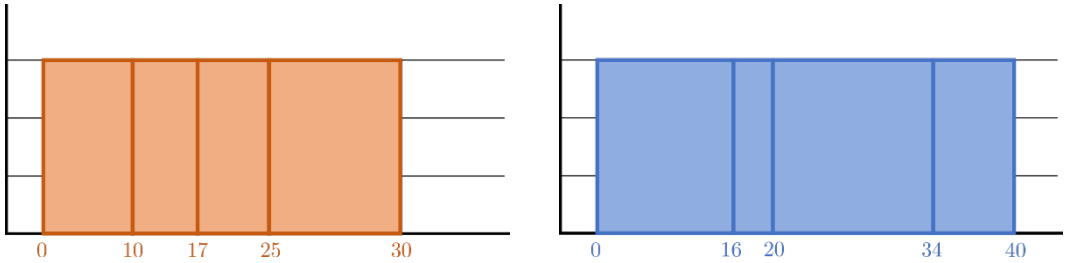


Figure 2.7: Histograms showing strictly left of (<<) lower bounds (left) and upper bounds (right)

In order to satisfy the condition range_attr1 << int8range(18,24), the upper bounds from column range_attr1 should be less than 18 (Figure 2.8).

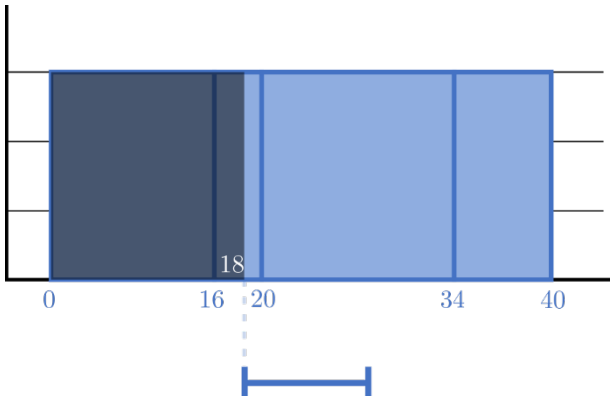


Figure 2.8: Values that comply the condition < 18 in the upper bound histogram (dark shade on left side)

From the above graph, the 18 value falls into the second bin (16-20). The fraction of rows that comply the condition is called *selectivity*. It is calculated as follows:

$$selectivity = \frac{(number\ of\ bins - bin[2].index) + \frac{18 - bin[2].min}{bin[2].\max - bin[2].min}}{number\ of\ bins}$$

$$= \frac{1 + \frac{18-16}{20-16}}{4}$$

$$= 0.375$$

The *selectivity* is used to calculate the *cardinality* with:
cardinality = reltuples[7] x selectivity *cardinality* = 1000 x 0.375 = 375

[7] As seen before, reltuples is the total number of rows from the table and it's retrieved from pg_class table

The planner estimates that the query will output 375 rows.

2.2.3.2 Example 2 – Filter Overlapping (&&) Ranges

```
SELECT *
FROM table1
WHERE range_attr1 && int8range(18,24)
```

Here, the planner looks inside the file *rangetypes_selfuncs.c* and runs the function *calc_hist_selectivity*. The two histograms are generated as follows:

Assume that stavalues1[8] = {(0, 0), (10,16), (17, 20), (25, 34), (30, 40)}

The same two histograms are generated (Figure 2.9).

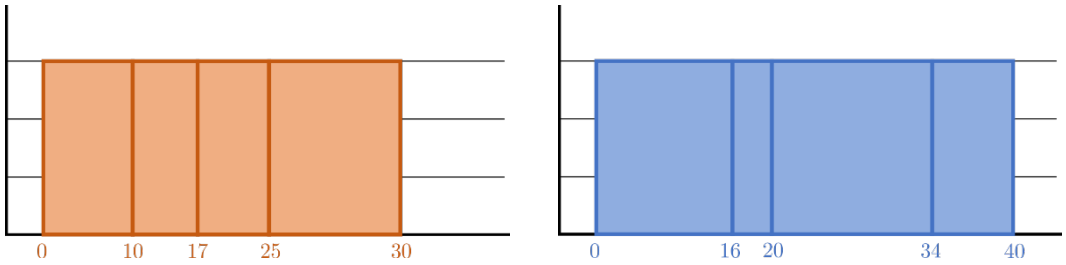


Figure 2.9: Histograms showing overlap (&&) lower bounds (left) and upper bounds (right)

In order to satisfy the condition range_attr1 && int8range(18,24), at least one of the conditions range_attr1 << int8range(18,30) and range_attr1 >> int8range(18,24) should NOT be satisfied[9].

In order to satisfy the condition range_attr1 << int8range(18,24), the upper bounds from column range_attr1 should be less than 18. From previous example we calculated that the *selectivity* of range_attr1 << int8range(18,24) was 0.375.

In order to satisfy the condition range_attr1 >> int8range(18,24), the lower bounds from column range_attr1 should be more than 24 (Figure 2.10).

[8] Statistic from range_attr1
[9] A && B ⇔ NOT (A << B OR A>>B)

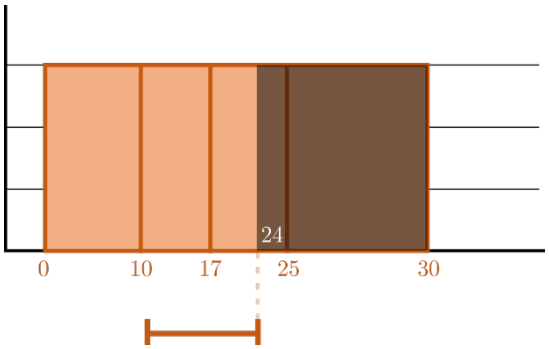


Figure 2.10: Values that comply the condition > 24

From the above graph, it is determined that the 18 value falls into the third bin (18-25). The *selectivity* for this condition is calculated below:

$$selectivity = \frac{(number\ of\ bins - bin[3].index) + \frac{bin[3].\max - 24}{bin[3].\max - bin[2].min}}{number\ of\ bins}$$

$$= \frac{1 + \frac{25-24}{25-18}}{4}$$

$$= 0.285$$

$cardinality = 1000 \times 0.285 = 285$

The *selectivity* of rows that are strictly at the left of (18,24) is 0.375 and the one from rows strictly at the right is 0.285. Thus, it can be concluded that the *cardinality* of the overlapped ranges is 1 – 0.375 – 0.285 = 0.340. The *cardinality* for this query is then 1000 x 0.340 = 340.

The planner estimates that the query will output 340 rows.

2.2.3.3 Example 3 – Join Overlapping (&&) ranges

```
SELECT *
FROM table1
INNER JOIN table2
ON range_attr1 && range_attr2
```

Here, the planner looks inside the file *geo_selfuncs.c* and runs the function *areajoinsel*. This function provides a default *selectivity* value of 0.005. Then the planner multiplies the row counts of both tables and finally multiplies the result with the default *selectivity* to calculate the *cardinality*. In the example:

$cardinality = 1000 \times 1000 \times 0.005 = 5000$

The planner estimates that the query will output 5000 rows.

From this example, it is apparent that *cardinality* estimation for queries that involve joins may have a big inaccuracy. In examples 1 and 2, the *selectivity* is calculated for every single case using the statistics, while in example 3, the *selectivity* is a default value no matter what the joining tables' values are and so, the planner is not using the statistics for its calculation. Moreover, the calculation of the *cardinality* uses an ambiguous *selectivity* multiplying it by the product of the joining tables' counts giving inaccurate estimations.

The next section will focus on proposing and implementing a statistical method to calculate the *cardinality* estimation for queries including the inner join clause and "overlap" operands, seen in example 3. Additionally, it will introduce some improvements to the calculation of the *selectivity* in queries with filtering clauses and the operands "strictly left of" or "overlap", as seen in examples 1 and 2.

3 Proposals and Implementation

As it was explained in the previous section, the *selectivity* estimation performed by the query planner in queries involving range data types, filtering clauses and the operands "strictly left of" or "overlap" relies on calculations based on the statistics collected about the tables. Specifically, the estimations are based on the two equal depth (or equi-depth) histograms collected for the range data type column addressed in the query.

While this approach gives acceptable results for such queries, it can be limited in terms of its application in statements with other clauses. It was observed that in queries with inner join clauses and "overlap" operands, the planner does not even use the statistics for estimating the *selectivity*. Instead, it takes a default *selectivity* value defined for this specific type of query and multiplies it with the product of the row counts of the two tables being joined.

In this section, a proposal to collect a new statistic is presented. Its consumption will create a new way of calculating the *selectivity*, which can be applied to all the mentioned query types. Furthermore, it will be explained how the *cardinality* estimation in queries with inner join clauses and "overlap" operands can be improved with this new approach.

3.1 A New Statistic: the Equi-Width Histogram

An equi-width histogram splits data into a fixed number of ranges that all have the same width. The height of each range is the count of values falling inside the range. This new approach will be implemented with the collection of such statistic occurring when the function VACUUM ANALYZE is run over tables with range type data. The implementation is completed within PostgreSQL's range_datatype.c file, which gathers the functions to collect the statistics for range data type columns.

3.1.1 Equi-Width Concept Overview

Suppose we have a table with a column of range type data like that in Figure 3.1.

range_attr1
[0, 12)
[4, 16)
[15, 34)
[14, 40)
[14, 25)
[16, 24)

Figure 3.1: Example of column with range type data.

Now, we can take the maximum and minimum value from all the ranges and create four equi-width ranges (or bins)[10] between them as below. Using four ranges is arbitrary and only being used for illustrative purposes (Figure 3.2).

[0, 10), [10, 20), [20, 30), [20 40)

Figure 3.2: Equi-width bins generated for illustrative purposes.

With ranges as defined above, a histogram is created with every row's data as follows: From bin [0, 10), count the number of rows whose ranges overlap it. In this case, both the rows with ranges [0, 12) and [4, 16) overlap this bin, so the count in the histogram will be 2. This is repeated for each of the equi-width bins in the histogram. The result is shown in Figure 3.3.

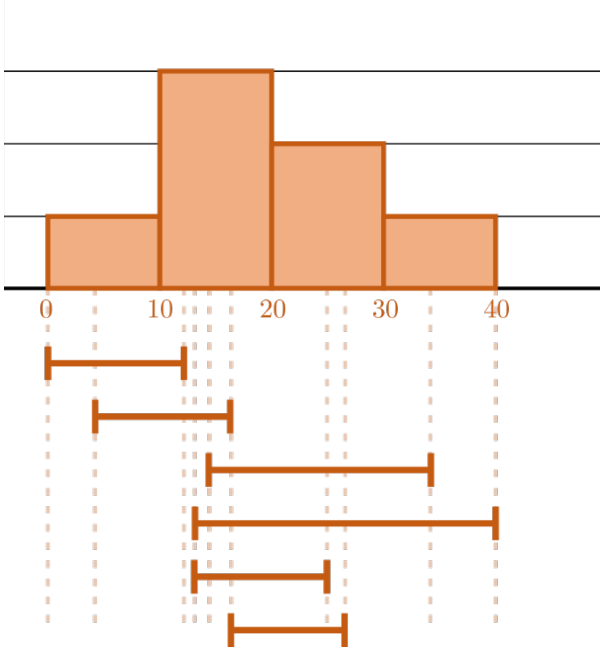


Figure 3.3: Illustrative equi-width histogram built from range type data.

The histogram frequency will be then defined by {2, 6, 4, 2}. This statistic would then be stored in pg_statistics. In this example, there are four bins, and each bin has a non-zero entry. In the actual implementation, a default number of 100 equi-width ranges will be defined regardless of the number of data points. While a larger number of equi-width ranges would provide more accurate estimations, 100 is selected according to the default_statistics_target value, which is a parameter that controls the amount of statistics collected [8]. Additional useful information to collect is the width of the histogram bins, the width of the whole histogram (max – min) and the histogram bounds (min, max). These values will be stored in pg_statistic as shown in Figure 3.4.

[10] The terms "equi-width range", "bin", and "equal width range" can all be used interchangeably.

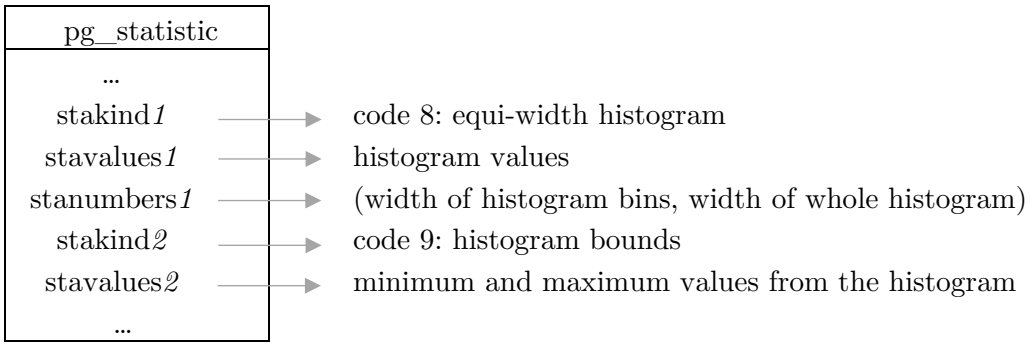


Figure 3.4: pg_statistic histogram value storage.

The statistics that were previously on slots 1 and 2 are displaced to 3 and 4 respectively. The implementation to achieve this is explained in the next section.

3.1.2 Implementation

The follow implementation is completed in the rangetype_analyze.c file.

Figure 3.5 shows the code used to obtain the maximum and minimum values from the equi-width histogram.

```
int64* equiWidthHistogram;

// Get histogram absolute min and max values (edges of histogram)
RangeBound minLower = lowers[0];
RangeBound maxUpper = uppers[0];
// Initialize bounds to first non-infinite bound
for (int i = 0; i < non_empty_cnt && minLower.infinite; ++i) minLower = lowers[i];
for (int i = 0; i < non_empty_cnt && maxUpper.infinite; ++i) maxUpper = uppers[i];


// Find min lower and max upper bounds (ignoring infinite bounds)
for (int i = 0; i < non_empty_cnt; ++i) {
    if (!lowers[i].infinite && range_cmp_bounds(typcache, &lowers[i], &minLower) < 0) {
        minLower = lowers[i];
    }
    if (!uppers[i].infinite && range_cmp_bounds(typcache, &uppers[i], &maxUpper) > 0) {
        maxUpper = uppers[i];
    }
}
```

Figure 3.5: Minimum and maximum value of the histogram

Next, the code to get determine the equi-width histogram values is executed as shown in Figure 3.6.

```
// Initialize histogram
equiWidthHistogram = (int64*)palloc(equiWidthBins * sizeof(int64));
memset(equiWidthHistogram, 0, equiWidthBins * sizeof(int64));
//Fill histogram
int sumBinsPerRange = 0;
for (int i = 0; i < non_empty_cnt; ++i) {
    RangeBound  lower = lowers[i], upper = uppers[i];
    int startBin;
    if (lower.infinite) startBin = 0; // First bin if lower is -infinite
    else {
        float8 lowerDiff = DatumGetFloat8(FunctionCall2Coll(&typcache->rng_subdiff_finfo,
            typcache->rng_collation,
            lower.val, minValue));
        startBin = (int)lowerDiff / binWidth;
    }
    int endBin;
    if (upper.infinite) endBin = equiWidthBins - 1; // Last bin if upper is infinite
    else {
        float8 upperDiff = DatumGetFloat8(FunctionCall2Coll(&typcache->rng_subdiff_finfo,
            typcache->rng_collation,
            upper.val, minValue));
        endBin = (int)upperDiff / binWidth;
        if (endBin >= equiWidthBins) endBin = equiWidthBins - 1; // Edge case when upper == maxUpper
    }
    // Get the number of bins that this range is in (to calculate the average afterwards)
    int numBins = endBin - startBin + 1;
    sumBinsPerRange += numBins;
    // Increase the bins this range is in
    for (int bin = startBin; bin <= endBin; ++bin) equiWidthHistogram[bin]++;
}
```

Figure 3.6: Determining the equi-width histogram bin boundary values

The histogram values are stored in stavalues*1* and the minimum and maximum values from the histogram are stored in stavalues*2*. Additionally, histogram metadata such as the width of histogram bins or the width of whole histogram are stored in stanumbers*1*. This is done with the code shown in Figure 3.7.

```
// Save histogram datatype metadata
stats->stakind[slot_idx] = STATISTIC_KIND_EQUIWIDTH_RANGE_HISTOGRAM;
stats->staop[slot_idx] = Int8LessOperator;
stats->stacoll[slot_idx] = InvalidOid;
stats->statypid[slot_idx] = INT8OID;
stats->statyplen[slot_idx] = sizeof(int64);
stats->statypbyval[slot_idx] = USE_FLOAT8_BYVAL;
stats->statypalign[slot_idx] = TYPALIGN_DOUBLE;

// Save histogram data
stats->stavalues[slot_idx] = equiWidthHistogram;
stats->numvalues[slot_idx] = equiWidthBins;

// Save histogram metadata
stats->numnumbers[slot_idx] = 4;
float4* binInfo = (float4*)palloc(sizeof(float4) * 4);
binInfo[0] = (float4)binWidth;
binInfo[1] = avgBinsPerRange;
binInfo[2] = (float4)histogramWidth;
binInfo[3] = (float4)sumBinsPerRange;
stats->stanumbers[slot_idx] = binInfo;

slot_idx++;

// To save the histogram bounds we need to use another statistic slot
// because the bounds are of unknown type and need to be saved as Datum
stats->stakind[slot_idx] = STATISTIC_KIND_EQUIWIDTH_RANGE_HISTOGRAM_BOUNDS;
stats->stavalues[slot_idx] = histogramBounds;
stats->numvalues[slot_idx] = 1;

slot_idx++;
```

Figure 3.7: Capturing histogram statistics and metadata.

3.2 Selectivity Estimation with a Single Histogram

As seen in the background section, the query planner calculates the *selectivity* based on two histograms, one for the lower and one for the upper bounds. However, this approach is only applicable to some query statements and requires taking statistics for both lower and upper bounds. In contrast to this, the new statistic proposed in the previous section requires less space, since it only takes the count of the ranges overlapping the histogram equi-width bins. Additionally, the equi-width histogram is useful to calculate the *selectivity* for more types of queries involving ranges.

3.2.1 Selectivity Concept Overview

To better explain this approach, we will build on the example queries from the background section and apply the new *selectivity* calculation method.

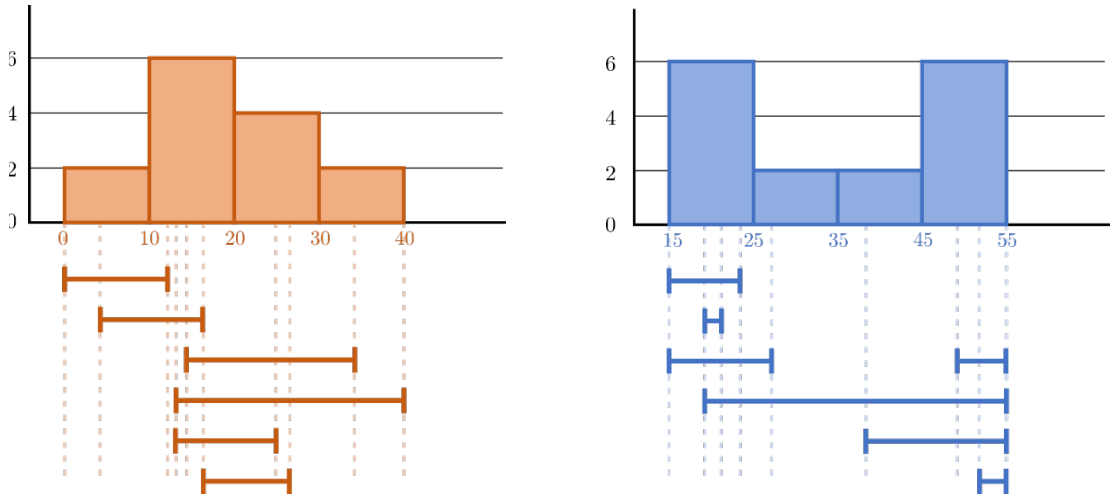
Suppose two tables named table1 and table2 contain the range type column values shown in Figure 3.8.

range_attr1
[0, 12)
[4, 16)
[15, 34)
[14, 40)
[14, 25)
[23, 24)

range_attr2
[15, 22)
[19, 21)
[15, 27)
[19, 55)
[49, 55)
[38, 55)
[53, 55)

Figure 3.8: Range type column for table 1 (left) and table 2 (right)

Two equi-width histograms are generated[11] ().



stavalues1 = {2, 6, 4, 2}
stanumbers1 = {10, 40}
stavalues2 = {0, 40}

stavalues1 = {6, 2, 2, 6}
stanumbers1 = {10, 40}
stavalues2 = {15, 55}

Figure 3.9: Equi-width histogram example with range_attr1 (left) and range_attr2 (right).

We can now examine the same examples shown previously and how they will be represented with the new equi-width histogram.

3.2.1.1 Example 1 – Filter Strictly Left of (<<) Ranges

```
SELECT *
FROM table1
WHERE range_attr1 << int8range(18,24)
```

Here, the planner will consume the equi-width histogram from the statistics collected for table1. Then, it will create points taking the middle point of the histogram bins and the respective frequency as shown in Figure 3.10.

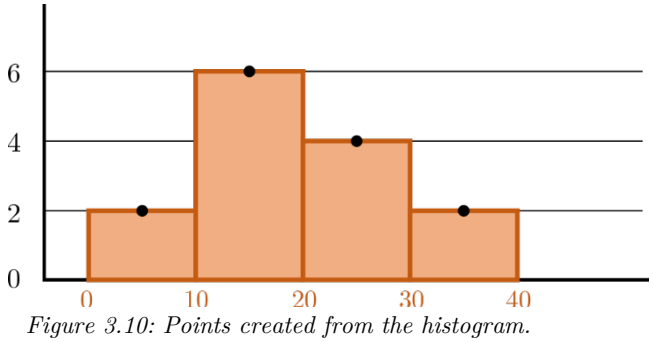


Figure 3.10: Points created from the histogram.

[11] The actual equi-width histogram has 100 equi-width ranges. For illustrative purposes, only four ranges are shown in this proposal for simplicity.

A 5^{th} order polynomial regression function $f(x)$ is estimated according to the points taken from the previous step (Figure 3.11).

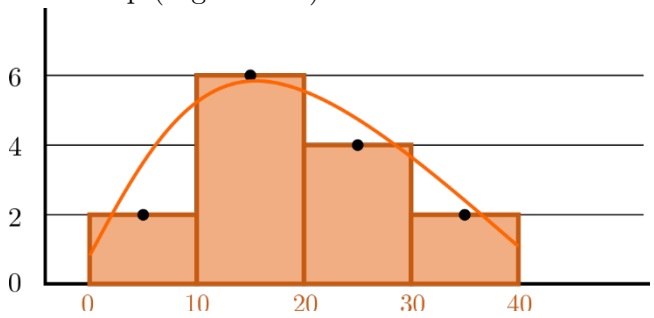


Figure 3.11: 5^{th} grade polynomial curve

Next, the area under the curve bounded by the histogram maximum and minimum values is calculated (Figure 3.12).

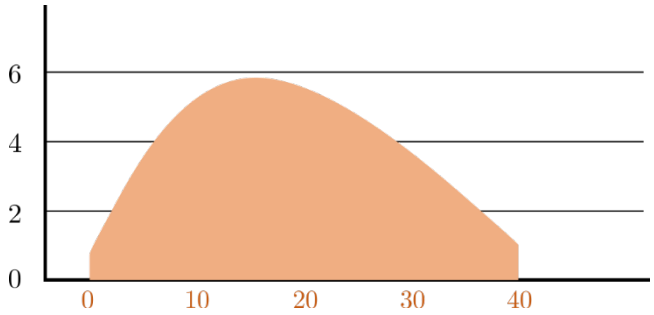


Figure 3.12: Area under the curve limited by the histogram extreme bounds

The area under the curve is $\int_0^{40} f(x)dx = 135$. A key idea here is to understand that this value is representative to the total number of rows of table1.

Next, the filtering process proceeds as follows. In order to satisfy the condition range_attr1 << int8range(18,24), the portion of the area below 18 will be taken (Figure 3.13).

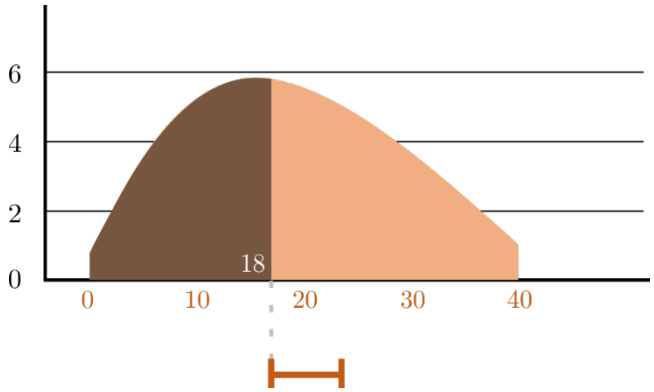


Figure 3.13: Portion of the area that complies the condition

The shaded area represents the proportion of the rows that comply with the condition "strictly left of". The area under the curve in that section is

$$\int_0^{18} f(x)dx = 61.344.$$

The *selectivity* is now calculated as follows:

$$selectivity = \frac{\int_0^{18} f(x)dx}{\int_0^{40} f(x)dx} = \frac{61.344}{135} = 0.454$$

Where $\int_0^{18} f(x)dx$ is the area below the curve representative to the rows that comply with the condition and $\int_0^{40} f(x)dx$ is the area below the curve representative of the total number of rows.

3.2.1.2 Example 2 – Filter Overlapping (&&) Ranges

```
SELECT *
FROM table1
WHERE range_attr1 && int8range(18,24)
```

Here again, the planner will consume the equi-width histogram from the statistics collected for table1. Then, it will create points taking the middle point of the histogram bins and the respective height as follows (Figure 3.14).

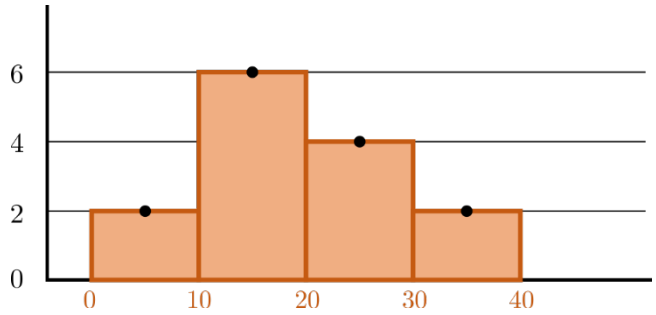


Figure 3.14: Mid points used from the center of each histogram bin.

Now, a 5th order polynomial regression function $f(x)$ is estimated according to the points taken from the previous step (Figure 3.15).

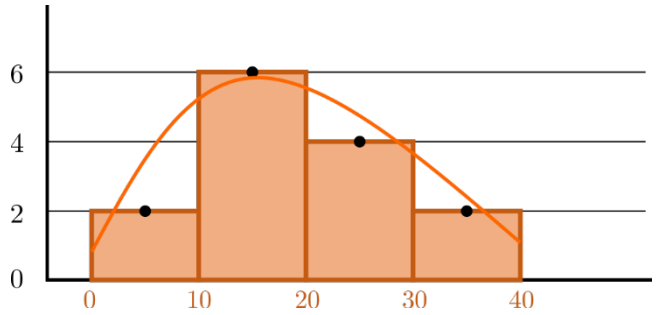


Figure 3.15: 5th grade polynomial regression curve generated from histogram bins.

Next, the area under the curve bounded by the histogram maximum and minimum values is calculated (Figure 3.16).

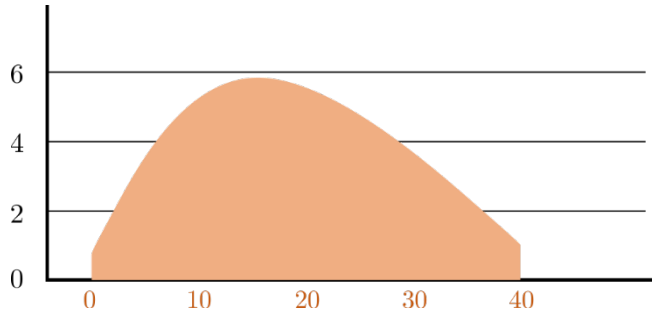


Figure 3.16: Area under the curve limited by the histogram maximum and minimum.

The area under the curve is $\int_0^{40} f(x)dx = 135$. Again, consider that this value is representative to the total number of rows of table1.

In order to satisfy the condition range_attr1 && int8range(18,24), the area under the curve will be taken with the following boundaries:

- Integral lower bound: The lower bound of the histogram or the lower bound of the range in the query. Choose the greater one.
- Integral upper bound: The upper bound of the histogram or the upper bound of the range in the query. Choose the lower one.

In this case, the range is contained completely within the histogram minimum and maximum bounds and thus, the integral is defined from the minimum to the maximum bound in the range (Figure 3.17).

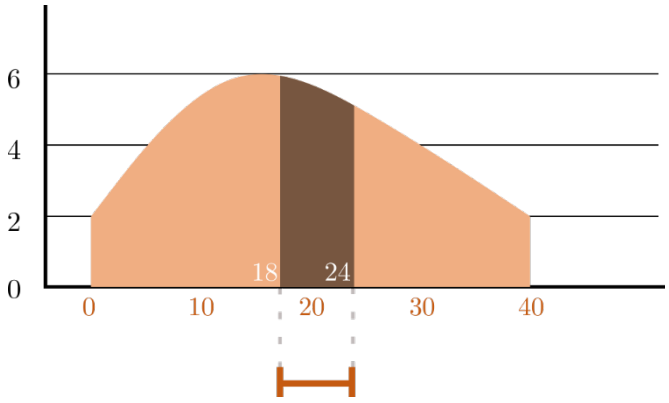


Figure 3.17: Area under the curve over 18 and below 24

The area is:

$$\int_{18}^{24} f(x)dx = 43.056$$

Finally, the *selectivity* value is calculated as follows:

$$selectivity = \frac{\int_{18}^{24} f(x)dx}{\int_{0}^{40} f(x)dx} = \frac{30.6}{135} = 0.226$$

Here $\int_{18}^{24} f(x)dx$ is the area below the curve that is representative of the rows over 18 and below 24 and $\int_{0}^{40} f(x)dx$ is the area below the curve representative of the total number of rows.

3.2.1.3 Example 3 – Join Overlapping (&&) ranges

```
SELECT *
FROM table1
INNER JOIN table2
ON range_attr1 && range_attr2
```

Here, the planner will consume the equi-width histograms from the statistics collected for table1 and table2. Then, it will create the 5th grade polynomial curve for each one the same way as in the previous examples (Figure 3.18, Figure 3.19, Figure 3.20, and Figure 3.21).

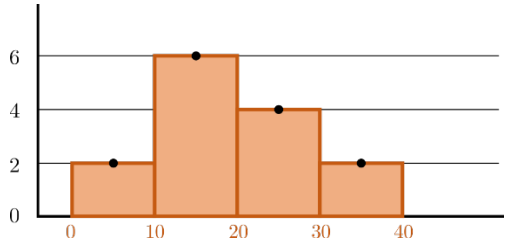


Figure 3.18: Bin mid-points of table1

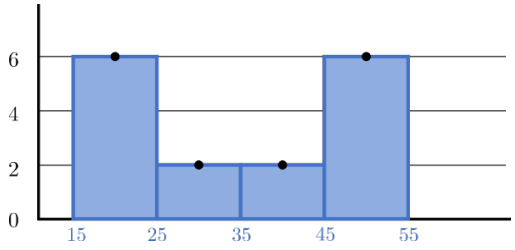


Figure 3.19: Bin mid-points of table2

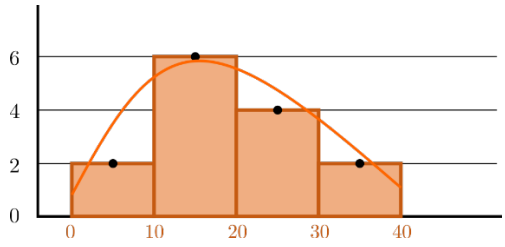


Figure 3.20: Polynomial curve f(x) for table1

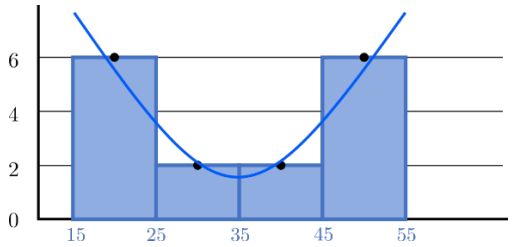


Figure 3.21: Polynomial curve g(x) for table2

Next, the areas below the curves are taken as shown in Figure 3.22 and Figure 3.23.

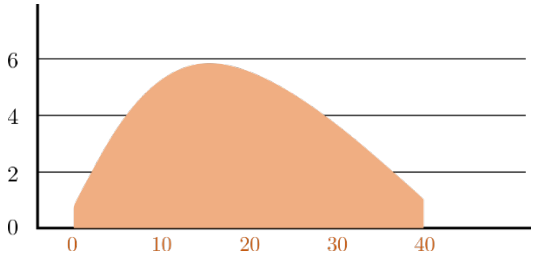


Figure 3.22: Area below the curve for table1

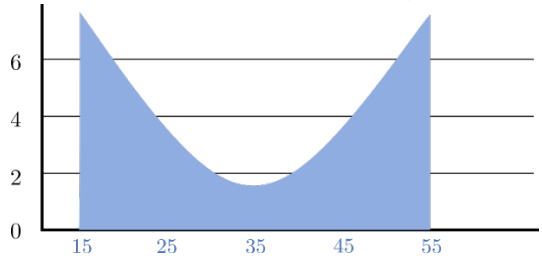


Figure 3.23: Area below the curve for table2

$$\int_0^{40} f(x)dx = 135$$

$$\int_{15}^{55} g(x)dx = 166.67$$

In order to calculate the selectivity for the inner join, an intersection between the two areas will be generated as in Figure 3.24.

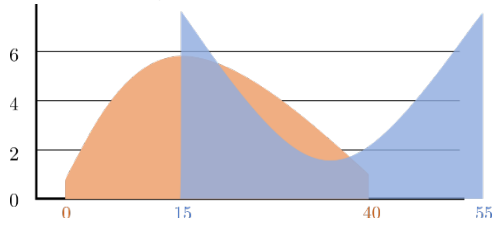


Figure 3.24: Frequency integral areas shown overlapping for inner join.

Now, we can take the portion of the areas that falls into the intersection range (Figure 3.25).

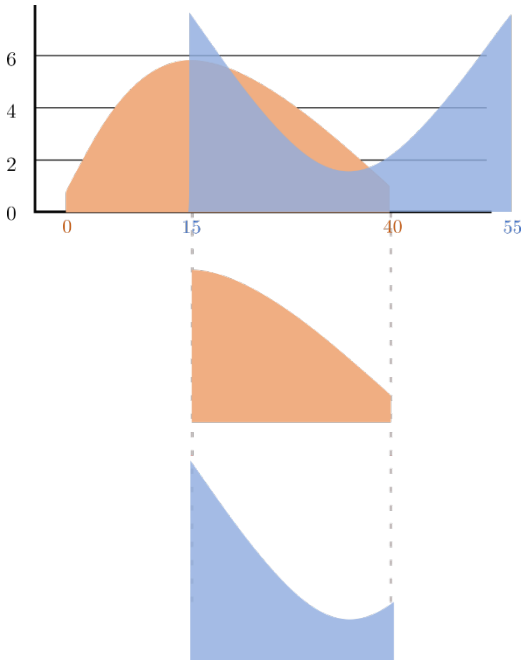


Figure 3.25: Overlapping areas bounded by highest max and lowest min of the two integrals.

For table1:

$$\int_{15}^{40} f(x)dx = 91.406$$

For table2:

$$\int_{15}^{40} g(x)dx = 91.667$$

Now, similarly to previous examples, get the proportion that each extracted area represents from its respective original area.

For table1:

$$\frac{\int_{15}^{40} f(x)dx}{\int_{0}^{40} f(x)dx} = \frac{91.406}{135} = 0.677$$

For table2:

$$\frac{\int_{15}^{40} g(x)dx}{\int_{15}^{55} g(x)dx} = \frac{91.667}{166.67} = 0.549$$

Each magnitude represents the proportion of rows of the respective table that should be included in the inner join result or in other words, the *selectivity* for each table. Since they do not represent the final *selectivity* of the query, we will call each a *partial selectivity*.

Finally, the two *partial selectivities* are used to calculate the final *selectivity* of the query.

$$selectivity = partial\ selectivity\ 1\ \times partial\ selectivity\ 2$$

$$selectivity = 0.677\ \times 0.549 = 0.371$$

Multiplying the two partial selectivities is a simple and naïve way to combine the results. It was observer during implementation and benchmarking that the approach works will with relatively short ranges and accuracy decreases with relatively wide ranges. This would be the main area of focus for future improvement.

As it was seen in the background section, in queries with inner joins and overlap operands, the query planner took a default value as the *selectivity* of the query and then multiplied it with the product of the two tables' rows counts to calculate the *cardinality*. With this new approach, the planner now takes a custom value for each combination of tables.

The implementation of the above methods for the *selectivity* calculations are explained in the following subsection.

3.2.2 Implementation

In this subsection the coding implementation of the proposal presented previously will be performed. It includes the changes for the *selectivity* estimation in queries with filtering clauses and the operands "overlap" and "strictly left of", and the new way of calculating the *selectivity* for queries with inner join clauses and the operand "overlap". This is done in PostgreSQL source code.

To calculate the selectivities, some functions taken from other people's work have been used. These two functions are as follows:

- Polyfit function calculates the coefficients of the polynomial approximation of the histogram, given the X and Y axes values, the number of elements, the order of the polynomial to create (in our case, always 5), and a pointer to an array where these coefficients will be saved. This function is taken from natedomin's polyfit repository (polyfit.c file)[12] (Figure 3.26).

```
int polyfit(
    const double* const dependentValues,
    const double* const independentValues,
    unsigned int        countOfElements,
    unsigned int        order,
    double*             coefficients)
```

Figure 3.26 Polyfit function header from natedomin's repository.

- Trapezoidal function calculates the area under a polynomial over a given range [a,b] using the trapezoid method. Although this function hasn't been taken literally from anywhere, it was written based on the examples available here[13] (Figure 3.27).

```
double trapezoidal(
    double* coefficients,
    int     order,
    double  a,
    double  b)
```

Figure 3.27 Trapezoidal function header

3.2.2.1 Filtering clauses

The implementation for filtering clauses is coded inside *rangetype_selfuncs.c*. Here the statistics are retrieved as shown in Figure 3.28.

[12] https://github.com/natedomin/polyfit
[13] https://rosettacode.org/wiki/Numerical_integration#C

```
hist_min = (RangeBound*)palloc(sizeof(RangeBound));
hist_max = (RangeBound*)palloc(sizeof(RangeBound));
hist_vals = (int64*)hvslot.values;
num_bins = hvslot.nvalues;

range_deserialize(typcache, DatumGetRangeTypeP(hbslot.values[0]),
                  hist_min, hist_max, &hist_empty);

lowerDistance = DatumGetFloat8(FunctionCall2Coll(&typcache->rng_subdiff_finfo,
                  typcache->rng_collation,
                  const_lower.val, hist_min->val));
upperDistance = DatumGetFloat8(FunctionCall2Coll(&typcache->rng_subdiff_finfo,
                  typcache->rng_collation,
                  const_upper.val, hist_min->val));

binWidth = hnslot.numbers[0];
totalBinSum = hnslot.numbers[3];
```

Figure 3.28: Retrieving statistics needed for selectivity estimation.

Then, these statistics are consumed differently depending on the operand of the query statement. Both “strictly left of” and “overlap” cases will be analyzed below.

Strictly left of

Instead of calling the function *calc_hist_selectivity* when this operand is used, a new function *calc_equi_hist_selectivity_left* is invoked (Figure 3.29).

```
case OID_RANGE_LEFT_OP:
    /* var << const when upper(var) < lower(const) */
    hist_selec = calc_equi_hist_selectivity_left(
        typcache, lowerDistance, hist_vals, num_bins, totalBinSum, binWidth
    );
    // hist_selec =
    //  calc_hist_selectivity_scalar(typcache, &const_lower,
    //                               hist_upper, nhist, false);
    break;
```

Figure 3.29: Calling the new calc_equi_hist_selectivity_left function for strictly left of (<<).

The function *calc_equi_hist_selectivity_left* is implemented with the following code (Figure 3.30).

```
static double
calc_equi_hist_selectivity_left(TypeCacheEntry *typcache, const float lowerDistance,
                                const int64 *hist, const int num_bins,
                                const float totalBinSum, const float binWidth)
{
    Selectivity selec;

    float8* histogramX = (float8*)palloc(num_bins * sizeof(float8));
    float8* histogramY = (float8*)palloc(num_bins * sizeof(float8));
    for (int i = 0; i < num_bins; ++i) {
        histogramX[i] = binWidth * i;
        histogramY[i] = (float8)hist[i];
    }
    selec = calculate_range_left_of_fraction(
        histogramX, histogramY, 0.0, binWidth * num_bins, num_bins, lowerDistance
    );
    return selec;
}
```

Figure 3.30: calc_equi_hist_selectivity_left function

It is important to highlight that the x-axis of the histogram contains the relative values of the bins rather than the absolute values. For example, if the subtype of the range type in the

query is a date, the x-axis labels in the histogram wouldn't be the dates themselves but the difference in days from the first date (lower bound of the histogram). A consequence of this is that the histogram always starts at 0.

Inside *calc_equi_hist_selectivity_left*, another function is called to calculate the final *selectivity* using the polynomial approximation and the area analysis explained previously. The function is called *calculate_range_left_of_fraction* (Figure 3.31).

```
double calculate_range_left_of_fraction(
    double* xa,
    double* ya,
    double min_a,
    double max_a,
    unsigned int countOfElements,
    double const_lower
) {
    const unsigned int order = 5;
    double coefficients_a[6]; // resulting array of coefs

    // Don't waste time in trivial cases
    if (const_lower < min_a) return 0.0;
    else if (const_lower > max_a) return 1.0;

    // Get polynomial aproximation
    int a_function = polyfit(xa, ya, countOfElements, order, coefficients_a);

    // Compute histogram areas
    double integ_a = trapezoidal(coefficients_a, order, min_a, max_a);
    double intersect_a = trapezoidal(coefficients_a, order, min_a, const_lower);

    return intersect_a / integ_a;
}
```

Figure 3.31: calculate_range_left_of_fraction function used to calculate polynomial approximation.

Overlap

Similarly, instead of calling the function *calc_hist_selectivity* when this operand is used, the new function *calc_equi_hist_selectivity_overlap* is called (Figure 3.32).

```
case OID_RANGE_OVERLAP_OP:
    hist_selec = calc_equi_hist_selectivity_overlap(
        typcache, lowerDistance, upperDistance, hist_vals, num_bins, totalBinSum, binWidth
    );
    break;
```

Figure 3.32: New calc_equi_hist_selectivity_overlap function called for overlap (&&).

The function *calc_equi_hist_selectivity_overlap* is implemented in the code shown in Figure 3.33.

```c
static double
calc_equi_hist_selectivity_overlap(TypeCacheEntry *typcache, const float lowerDistance,
                                   const float upperDistance, const int64 *hist,
                                   const int num_bins, const float totalBinSum, const float binWidth)
{
    Selectivity selec;

    float8* histogramX = (float8*)palloc(num_bins * sizeof(float8));
    float8* histogramY = (float8*)palloc(num_bins * sizeof(float8));
    for (int i = 0; i < num_bins; ++i) {
        histogramX[i] = binWidth * i;
        histogramY[i] = (float8)hist[i];
    }
    selec = calculate_range_overlap_fraction(
        histogramX, histogramY, 0.0, binWidth * num_bins, num_bins, lowerDistance, upperDistance
    );
    return selec;
}
```

Figure 3.33: calc_equi_hist_selectivity_overlap function implementation.

Inside *calc_equi_hist_selectivity_overlap*, another function is called to calculate the final *selectivity* using the polynomial approximation and the area analysis explained in section 3.2.1. The function is called *calculate_range_overlap_fraction* (Figure 3.34).

```c
double calculate_range_overlap_fraction(
    double* xa,
    double* ya,
    double min_a,
    double max_a,
    unsigned int countOfElements,
    double const_lower,
    double const_upper
) {
    const unsigned int order = 5;
    double coefficients_a[6]; // resulting array of coefs

    // Don't waste time in trivial cases
    if (const_upper < min_a) return 0.0;
    else if (const_lower > max_a) return 0.0;
    else if (const_lower < min_a && const_upper > max_a) return 1.0;

    // Calculate bounds of intersection
    double lower = Max(min_a, const_lower);
    double upper = Min(max_a, const_upper);

    // Get polynomial aproximation
    int a_function = polyfit(xa, ya, countOfElements, order, coefficients_a);

    // Compute histogram areas
    double integ_a = trapezoidal(coefficients_a, order, min_a, max_a);
    double intersect = trapezoidal(coefficients_a, order, lower, upper);

    return intersect / integ_a;
}
```

Figure 3.34: calculate_range_overlap_fraction function

3.2.2.2 Inner join clauses

As seen before, PostgreSQL used a default value for the *selectivity* in queries with inner join clauses and the “overlap” operator. This was defined in *geo_selfuncs.c*. The new approach will make a calculation of the *selectivity* in the same file.

First, the statistics about the tables are retrieved and information about the two histograms is obtained. The x-axis values of the histograms are calculated relative to the minimum lowest bound (Figure 3.35). This means that one of the histograms (the one with the lowest lower bound) will start at 0, while the other one will start at the difference between the

lower bounds (e.g. given two histograms *a* and *b*, having *low(a)<low(b)*, histogram *a*'s X-axis will start at 0, while *b*'s will start at *low(b)-low(a)*) (Figure 3.35).

```
RangeBound* minBound = histogram1LowerBound->val < histogram2LowerBound->val ?
                       histogram1LowerBound :
                       histogram2LowerBound;
float8 lowerDiff1 = DatumGetFloat8(FunctionCall2Coll(&typcache1->rng_subdiff_finfo,
    typcache1->rng_collation,
    histogram1LowerBound->val, minBound->val));
float8 lowerDiff2 = DatumGetFloat8(FunctionCall2Coll(&typcache1->rng_subdiff_finfo,
    typcache1->rng_collation,
    histogram2LowerBound->val, minBound->val));
float8 baseValue1 = lowerDiff1 + binWidth1 / 2;
float8 baseValue2 = lowerDiff2 + binWidth2 / 2;
for (int i = 0; i < numberOfBins1; ++i) {
    histogram1x[i] = baseValue1 + binWidth1 * i;
    histogram1y[i] = (float8)histSlot1.values[i];
}
for (int i = 0; i < numberOfBins2; ++i) {
    histogram2x[i] = baseValue2 + binWidth2 * i;
    histogram2y[i] = (float8)histSlot2.values[i];
}
```

Figure 3.35: Retrieving statistics for overlap (&&) and building the histograms.

Then, a new function *calculate_range_join_overlap_fraction* is called with the statistics magnitudes retrieved (Figure 3.36).

```
double result = calculate_range_join_overlap_fraction(
    histogram1x,
    histogram1y,
    lowerDiff1,
    lowerDiff1 + histogramWidth1,
    histogram2x,
    histogram2y,
    lowerDiff2,
    lowerDiff2 + histogramWidth2,
    numberOfBins1
);
```

Figure 3.36: Calling new function calculate_range_join_overlap_fraction with statistics retrieved.

This last function is implemented in the next code (Figure 3.37). Here the *partial selectivity* as well as the total *selectivity* is estimated using the 5^{th} grade polynomial approximation and the area analysis described previously.

```
double calculate_range_join_overlap_fraction(
    double* xa,     // X Axis of Histogram 1
    double* ya,     // Y Axis of Histogram 1
    double min_a,   // Lower bound of Histogram 1
    double max_a,   // Upper bound of Histogram 1
    double* xb,     // X Axis of Histogram 2
    double* yb,     // Y Axis of Histogram 2
    double min_b,   // Lower bound of Histogram 2
    double max_b,   // Upper bound of Histogram 2
    unsigned int countOfElements
) {
    const unsigned int order = 5;       //Degree of the polynomial aproximation
    double coefficients_a[6]; // resulting array of coefs
    double coefficients_b[6]; // resulting array of coefs

    // Calculate polynomial aproximation for both histograms
    int a_function = polyfit(xa, ya, countOfElements, order, coefficients_a);
    int b_function = polyfit(xb, yb, countOfElements, order, coefficients_b);

    // Calculate the integral of both polynomials
    double integ_a = trapezoidal(coefficients_a, order, min_a, max_a);
    double integ_b = trapezoidal(coefficients_b, order, min_b, max_b);

    // Find start and end point of intersection
    double start = Max(min_b, min_a);
    double end = Min(max_b, max_a);

    if (end > start) {
        // Histograms don't intersect
        return 0;
    }

    // Calculate partial selectivity of each histogram
    double intersect_a = trapezoidal(coefficients_a, order, start, end) / integ_a;
    double intersect_b = trapezoidal(coefficients_b, order, start, end) / integ_b;

    // Calculate the final result
    double final_result = intersect_a * intersect_b;

    return final_result;
}
```

Figure 3.37: Calculating partial and total selectivity with calculate_range_join_overlap_fraction function.

3.3 Influence in the Cardinality

The *cardinality* estimation consumes the *selectivity* values from the examples shown previously to get the final number of rows that a query would have in its result. In the case of the queries with inner join clauses and "overlap" operands, the *cardinality* will now be calculated by the planner considering the new *selectivity* estimation method proposed in the previous section instead of using a default value.

4 Benchmark

To test the performance of the proposals and implementations developed in the previous section, a benchmark was completed. The benchmark compares the accuracy in the estimation of the number of rows in given queries using the original and the new estimation method.

4.1 Methodology

The benchmark will require some queries with specific features addressing tables with range types. As the goal is to compare the accuracy of the cost estimations and specifically the output's number of rows, the command EXPLAIN ANALYZE will be used over each query to get the necessary information. This will be performed once using the original cost estimation method and once using the new proposed method.

A total of 30 queries were benchmarked against the data described below.

4.2 Data

The data used for the benchmark is composed of 10 tables with a range column of int4range subtype. Of these 10 tables, 3 are used for the select queries (queries over a single table that filters rows with the "<<" or "&&" operands), 4 for the join queries (queries joining two tables with the "&&" operand), and the last 3 for the triple join query (query joining the three tables with the "&&" operand). All tables use the same range type (int4range), but the actual type should have no impact in neither the planning time, nor the estimation results, since the values used for the histograms are relative values.

To avoid estimation biasing, the table values follow different distributions: normal, uniform, and bimodal. The tables' names and its corresponding distribution are shown in Table 4-1.

Table 4-1: Benchmark tables and the statistical range distribution.

Table	Distribution
select_normal	Normal
select_uniform	Uniform
select_binormal	Bimodal: 2 normals added
join_normal	Normal
join_uniform	Uniform
join_uniform_right	Uniform
join_uniform_far_right	Uniform
triple_join_left	Uniform
triple_join_middle	Uniform
triple_join_right	Uniform

4.3 Queries

The queries used for the benchmark include statements with filtering clauses and operands "overlap" and "strictly left of", and others with inner join clauses with "overlap" operands.

The structure of the queries with filtering clauses are as follows:

```
SELECT *
FROM table_name
WHERE val operator range_value::int4range;
```

The values for table_name, operator, and range_value are shown in Table 4-2 and are used to define all the queries.

Table 4-2: Queries description for filtering using "overlap" and "strictly left of" operands.

Table name	Operator	Range value	Query identifier
select_uniform	<<	[-45, 23]	1
		[12345, 13456]	2
		[2525, 4240]	3
		[4525, 5240]	4
		[-23, 1025]	5
select_normal		[-45, 23]	6
		[12345, 13456]	7
		[2525, 4240]	8
		[4525, 5240]	9
		[-23, 1025]	10
select_binormal		[-45, 23]	11
		[12345, 13456]	12
		[2525, 4240]	13
		[4525, 5240]	14
		[-23, 1025]	15
select_uniform	&&	[-45, 23]	16
		[12345, 13456]	17
		[2525, 4240]	18
		[4525, 5240]	19
		[-23, 1025]	20
select_normal		[-45, 23]	21
		[12345, 13456]	22
		[2525, 4240]	23
		[4525, 5240]	24
		[-23, 1025]	25
select_binormal		[-45, 23]	26
		[12345, 13456]	27
		[2525, 4240]	28
		[4525, 5240]	29
		[-23, 1025]	30

The structure of the queries with inner join clauses will be the following:

```
SELECT *
FROM table_name_1 l, table_name_2 r
WHERE l.val && r.val
```

The tables' names used in the inner-join queries and their corresponding distributions are shown in Table 4-3.

Table 4-3: Queries used for inner-join clauses.

Query	Query identifier
uniform non overlapping	31
uniform overlapping	32
uniform normal	33
uniform normal bigger overlap	34
self join normal	35
self join uniform	36

An extra query performing a join between three tables was also run using the following structure. The query completed is shown in Table 4-4.

```
SELECT *
FROM triple_join_left l,
    triple_join_middle m,
    triple_join_right r
WHERE l.val && m.val
  AND l.val && r.val;
```

Table 4-4: Query used for triple inner-join.

Query	Query identifier
triple join	37

4.4 Cost estimation analysis

To get the information to test the accuracy of the original versus the new estimation method, the command EXPLAIN ANALYZE will be used. As seen in 2.1.4, this command shows the number of rows that the planner estimates and the real number of rows that the execution of the query will output. The analysis was completed twice for each query, once with the existing PostgreSQL estimation method and once with the new estimation implementation. The results of queries with filtering clauses using “overlap” and “strictly left of” operands are shown in Table 4-5. Figure 4-1 shows the relative results in graphical format.

Table 4-5: Estimate-to-real ratio comparison for filtering clauses using "overlap" and "strictly left of" operands.

Query	Real # Rows	Original estimation		New estimation		Δ Estimate-to-Real Ratio
		# Rows	Estimate-to-Real Ratio	# Rows	Estimate-to-Real Ratio	
1	0	1	99.00%	1	99.00%	0.00%
2	1000000	1000000	100.00%	1000000	100.00%	0.00%
3	247607	242727	98.03%	248831	100.49%	2.47%
4	447340	443265	99.09%	449391	100.46%	1.37%
5	0	1	99.00%	1	99.00%	0.00%
6	20	1	5.00%	1	5.00%	0.00%
7	1000000	1000000	100.00%	1000000	100.00%	0.00%
8	21599	22627	104.76%	1856	8.59%	-96.17%
9	338206	339375	100.35%	376643	111.36%	11.02%
10	23	1	4.35%	1	4.35%	0.00%
11	10	2557	25570.00%	1	10.00%	-25560.00%
12	1000000	1000000	100.00%	1000000	100.00%	0.00%
13	121248	120400	99.30%	185037	152.61%	53.31%
14	487801	483652	99.15%	444537	91.13%	-8.02%
15	12	2262	18850.00%	1	8.33%	-18841.67%
16	2366	2286	96.62%	2205	93.20%	-3.42%
17	0	1	99.00%	1	99.00%	0.00%
18	176253	175900	99.80%	172382	97.80%	-2.00%
19	76433	76735	100.40%	70810	92.64%	-7.75%
20	102306	101176	98.90%	98085	95.87%	-3.02%
21	13	1	7.69%	1	7.69%	0.00%
22	0	1	99.00%	1	99.00%	0.00%
23	251109	249873	99.51%	299470	119.26%	19.75%
24	238386	240019	100.69%	196328	82.36%	-18.33%
25	707	4845	685.29%	7504	1061.39%	376.10%
26	12	433	3608.33%	1	8.33%	-3600.00%
27	0	1	99.00%	1	99.00%	0.00%
28	354237	351636	99.27%	224921	63.49%	-35.77%
29	16236	18468	113.75%	88454	544.80%	431.05%
30	2124	5253	247.32%	10000	470.81%	223.49%

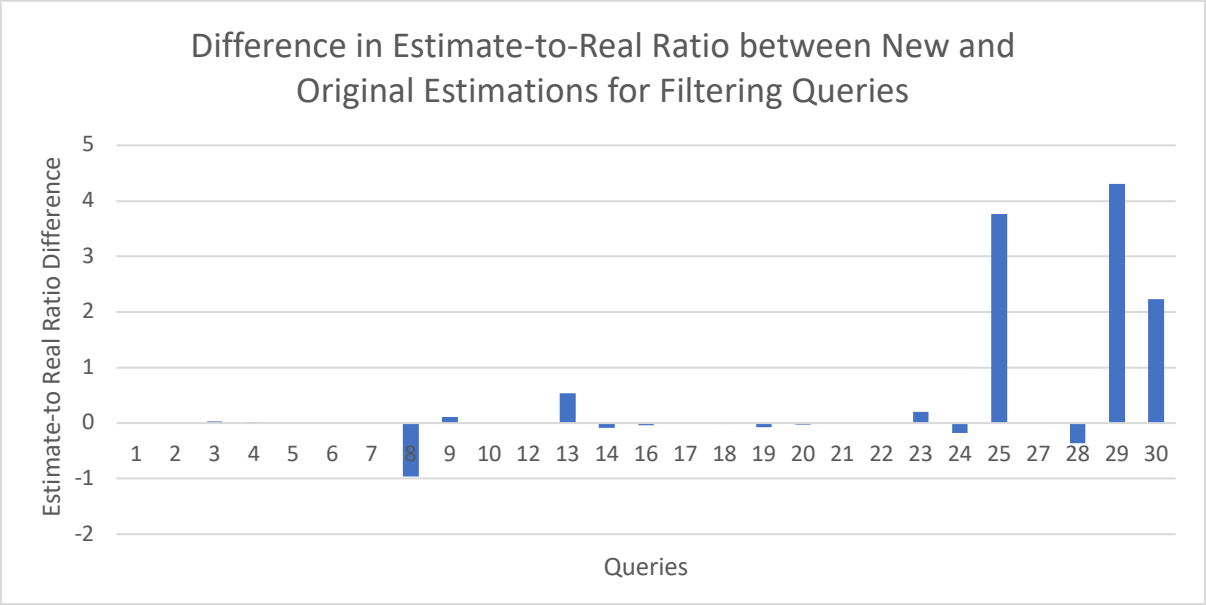


Figure 4-1: Comparison of filter query estimation estimate-to-real ratio for new and original statistics. Positive values indicate improved performance of new implementation.

From Figure 4-1 it can be remarked that in general, the new method does not improve the accuracy in the estimation of the number of rows. Additionally, the planning time was obtained for every query using each estimation method. To summarize the results, the minimum and the maximum planning times were taken among all the queries (Table 4-6).

Table 4-6: Comparison of planning times of original and the new implementation for filtering clauses

	Minimum plan time (s)	Maximum plan time (s)
Original	0.046	0.093
New	0.061	0.144

From the previous table in can be concluded that the new method is in general slower than the original one for queries with filtering clauses.

Table 4-7 and Figure 4-2 show results of the inner join queries with “overlap” operand.

Table 4-7: Estimate-to-real ratio comparison for inner join clauses "overlap" operands.

Query	Real # rows	Original estimation		New estimation		Δ Estimate-to-Real Ratio
		# rows	Estimate-to-Real Ratio	# rows	Estimate-to-Real Ratio	
31	0	50000000	0.00%	1	0.00%	0.00%
32	198033886	50000000	25.25%	605780618	305.90%	280.65%
33	486823100	50000000	10.27%	643822165	132.25%	121.98%
34	994284247	50000000	5.03%	1764235641	177.44%	172.41%
35	8870058330	50000000	0.56%	10000000000	112.74%	112.18%
36	963556002	50000000	5.19%	10000000000	1037.82%	1032.63%

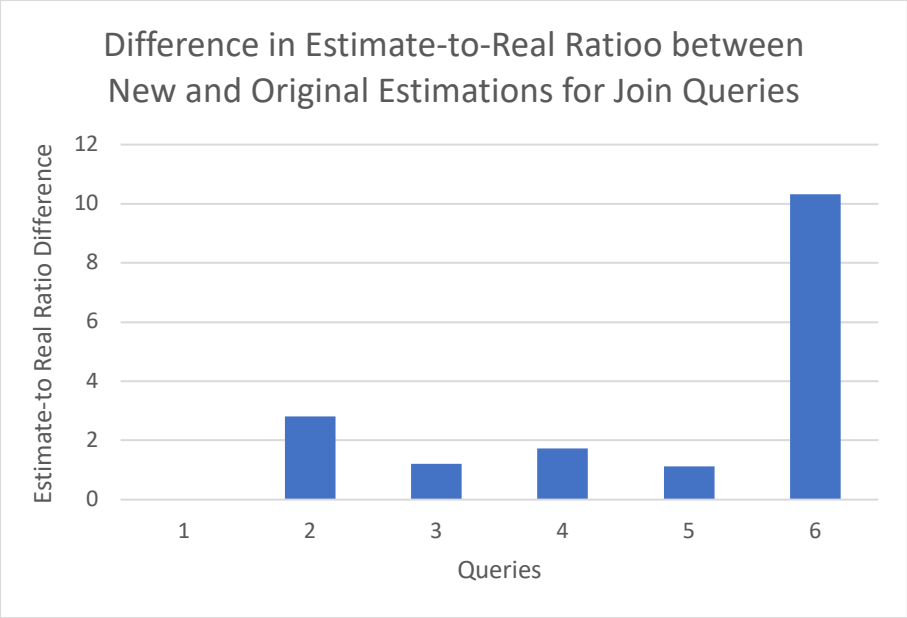


Figure 4-2: Comparison of join query estimation estimate-to-real ratio for new and original statistics. Positive values indicate improved performance of new implementation.

We can see in Table 4-7 how the current PostgreSQL estimation just gives a constant fraction of the cartesian product, while the new method gives a more accurate estimation of the number of rows (although it is now usually an overestimation).

The first row shows the edge case for when there is no overlap between the tables. We can see that the existing estimation returns the same constant as always, but the new method is able to see that there is no overlap.

This can greatly affect the performance of a more complex query, like query 37 (triple join). This query joins three tables, two big ones (1 million rows each) with no overlap, with one smaller (100 thousand rows) that overlaps with both. Since the existing estimation just

returns a constant fraction, the query planner selects a query that joins one of the big tables with the smaller one, and then tries to join the result with the other big one.

Instead, with the new estimation, the query planner can see that joining the two big tables first is better, since it can see that they have virtually no overlap (estimated rows are 1). This causes the query planner to choose a query in which the two big tables are joined, and the result is joined with the smaller table. In practice, this last join is not even performed.

Table 4-8 shows the impact the new estimation has in the triple join query. Although the new planning takes more than ten times more time, we end up with a query that takes less than 2 seconds to run, while when using the existing estimation, the query had to be manually stopped after 50 minutes of execution, without results (Table 4-8).

Table 4-8: Query performance on triple inner-join using (&&) operand.

	Number of rows	Estimated Rows	Planning Time	Execution time
Existing	0	2.5 x 10^{12}	0.032ms	Manually stopped after 50 mins
New	0	1	0.518ms	1153.168ms

4.5 Future Work: Alternative approaches

4.5.1 Combining Join Partial Selectivity

The next step in this project work would be to improve on the naïve way in which partial selectivity of 2 tables is combined in a join operation. Benchmarking and implementation results showed the current method favours certain data distributions, most notably data that has relatively short ranges.

One attempt to improve results was to alter the number of equal-width bins that existed within the area of range overlaps. It was found that as the number of bins increased, the accuracy became worse. This would be explained by the increase in the number of ranges that get “double counted” in cardinality as they span across multiple bins.

4.5.2 Non-Linear Regression and Skewness

An aspect of the implemented model that could be ported to the current PostgreSQL approach for additional refinement is the use of non-linear regression to determine how many rows would be returned in a single bin. For the currently implemented equal depth histogram used in PostgreSQL, accuracy for the number of rows returned is superior when an entire bin is returned, however there is no consideration given to the frequency of data points within a bin that needs to have some portion of the bin returned.

One way this could be improved is to calculate the mean, median, and standard deviation of values within each bin to determine the frequency skewness, and then apply a "skewness factor" to the linear regression within the bin. Issues that arise with this approach is an assumption of parametric distribution[14] of frequency data, which may or may not be the case. It could also be possible to use different cases and estimation models when choosing how to calculate the proportion of a bin to return. This could, for example, look for the presence of multi-modal distributions to determine whether to apply a skewness factor. Since each approach must take into consideration the computational overhead required for statistics calculations during the DBMS vacuum analyze function, benchmarking of planning times would be an important data point for comparison.

4.5.3 Generating Sample Data

Another area of further assessment to improve query performance is the methodology used the generate sample data used to calculate statistics in estimation. Currently, the sample data is pulled randomly from the existing data. Depending on the size and variation within the dataset, alternative methods could improve estimations by using different sampling methods on the data including algorithmic or parametric [9].

4.5.4 Implementation of Existing Theoretical Approaches

There are many theoretical discussions of different sampling methods that can be applied for query estimation, as can be seen in the "Efficient Sampling Strategies for Relational Databases research paper [10]. With additional resources, a literature review can enumerate different existing theoretical models, implement the models and benchmark performance with real world data cases as well as statistically generated data. This effort would be aided by more formally developed and procedural benchmarking processes that test different implementations of statistics for accuracy.

[14] Meaning the number of values is normally distributed, with most values existing in the center of the bin

5 Conclusion

The cost estimation process used by PostgreSQL for range data types was investigated. For queries involving inner joins and "overlap" operands, the *selectivity* is not calculated but is always a default value regardless of the values of the tables targeted by the statements. This has a direct influence in the estimation of the *cardinality* for such queries since query planning takes the value of *selectivity* in its calculation.

An alternative method to generate query estimations using equi-width histograms and the integrals of a 5^{th} order polynomial was proposed, implemented, and benchmarked. The method includes the collection of new statistics for range types to generate the equi-width histogram. These statistics are consumed in a new way to calculate *selectivity*, which has a direct influence in the *cardinality* estimation — the estimated output's row count. The same statistics are used to generate cost estimations for both filtering and join queries that use the "overlap" and "strictly left of" operands. The proposed implementation improves the existing estimation method by including *selectivity* calculation for inner join queries does use the statistics to calculate the *selectivity*.

The new method's estimations applied in queries with filtering clauses showed that it is more accurate when addressing tables with normal distributions, in comparison with uniform and binormal distributions. However, in general the new method did not provide better results than the original in terms of estimating the number of rows. The planning time was also found to be slower than the original.

Regarding queries with inner join clauses, the new implementation usually overestimates the actual number of rows that will be returned, but it is still much closer to actual returns. Simply having an estimation that is related to the true values avoids problems seen in the original implementation seen in the triple-join query.

6 References

[1] The PostgreSQL Global Development Group, "Performance Tips," *Documentation - PostgreSQL 14*. [Online]. Available: https://bit.ly/3po9AP5. [Accessed: 12-Mar-2021].

[2] J. Harvey, "The basic understanding of EXPLAIN ANALYZE," *PostgresConf South Africa 2019*, 2019.

[3] The PostgreSQL Global Development Group, "EXPLAIN," *Documentation - PostgreSQL 14*. [Online]. Available: https://bit.ly/3G8zUU3.

[4] The PostgreSQL Global Development Group, "pg_statistic," *Documentation - PostgreSQL 14*. [Online]. Available: https://bit.ly/31r2uRq.

[5] The PostgreSQL Global Development Group, "Range Functions and Operators," *Documentation - PostgreSQL 14*. [Online]. Available: https://bit.ly/3xTQBPW.

[6] The PostgreSQL Global Development Group, "pg_statistic.h," *PostgreSQL Source Code*. [Online]. Available: https://bit.ly/3lxo6Te.

[7] The PostgreSQL Global Development Group, "rangetypes_typanalyze.c," *PostgreSQL Source Code*. [Online]. Available: https://bit.ly/3xVboCR.

[8] The PostgreSQL Global Development Group, "Query Planning," *PostgreSQL Source Code*. [Online]. Available: https://bit.ly/3lLd2Cd.

[9] R. J. Lipton, J. F. Naughton, D. A. Schneider, and S. Seshadri, "Efficient sampling strategies for relational database operations," *Theor. Comput. Sci.*, vol. 116, no. 1, pp. 195–226, 1993.

[10] J. Frost, "Nonparametric Tests vs. Parametric Tests," *Statistics By Jim*, 2021. [Online]. Available: https://bit.ly/3DHj64L.