Range Types: Statistics, Selectivity and Join Estimations

INFO-H417-Database System Architecture

Project Report

Tejaswini Dhupad
Mohammad Zain Abbas
Himanshu Choudhary
Mohammad Ismail Tirmizi

UNIVERSITÉ LIBRE DE BRUXELLES

December, 2021

1. Introduction

Range types are used to represent a range, or all the values between two discrete values, of different (basic or custom) types of data. The type being used to represent range in this case is called *range's subtype*, which can be int, float, date, time or anything else. E.g. A range type of int4range would have an entry like: [23, 40]. Both values being inclusive in the range. A case of int4range can be used to figure how much the pay range (of employees in different companies) overlaps with each other. Here the subtype is int and it must have a total order so that it is well-defined whether element values are within, before, or after a range of values.

Range types are useful because they represent many element values in a single range value, and because concepts such as overlapping ranges can be expressed clearly. The use of time and date ranges for scheduling purposes is the clearest example; but price ranges, measurement ranges from an instrument, and so forth can also be useful.

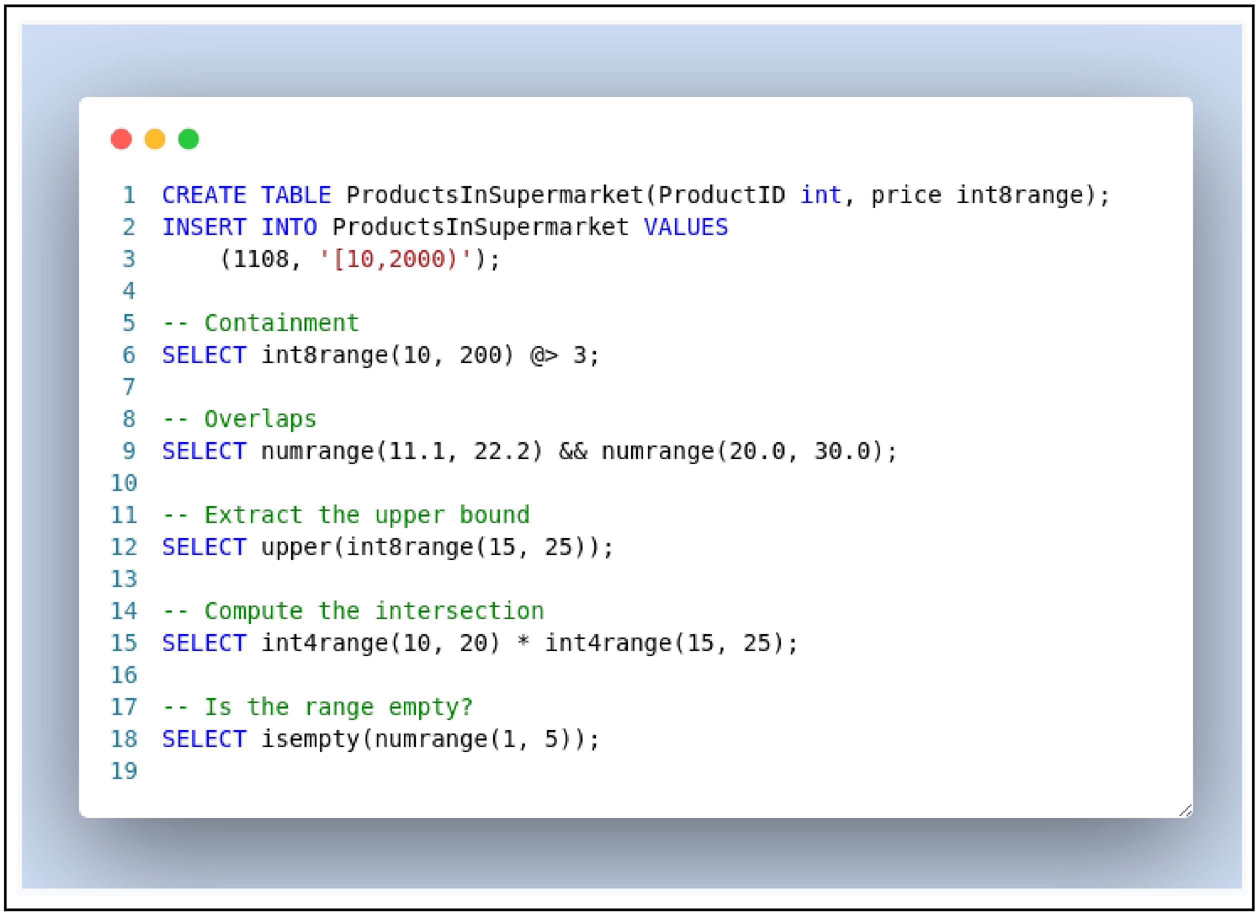
Every range type has a corresponding multi-range type. A multirange is an ordered list of non-contiguous, non-empty, non-null ranges. Most range operators also work on multi-ranges, and they have a few functions of their own.[1]

1.1 Built-in Range and Multirange Types

PostgreSQL comes with the following built-in range types:

- int4range — Range of integer, int4multirange — corresponding Multirange
- int8range — Range of bigint, int8multirange — corresponding Multirange
- numrange — Range of numeric, nummultirange — corresponding Multirange
- tsrange — Range of timestamp without time zone, tsmultirange — corresponding Multirange
- tstzrange — Range of timestamp with time zone, tstzmultirange — corresponding Multirange
- daterange — Range of date, datemultirange — corresponding Multirange

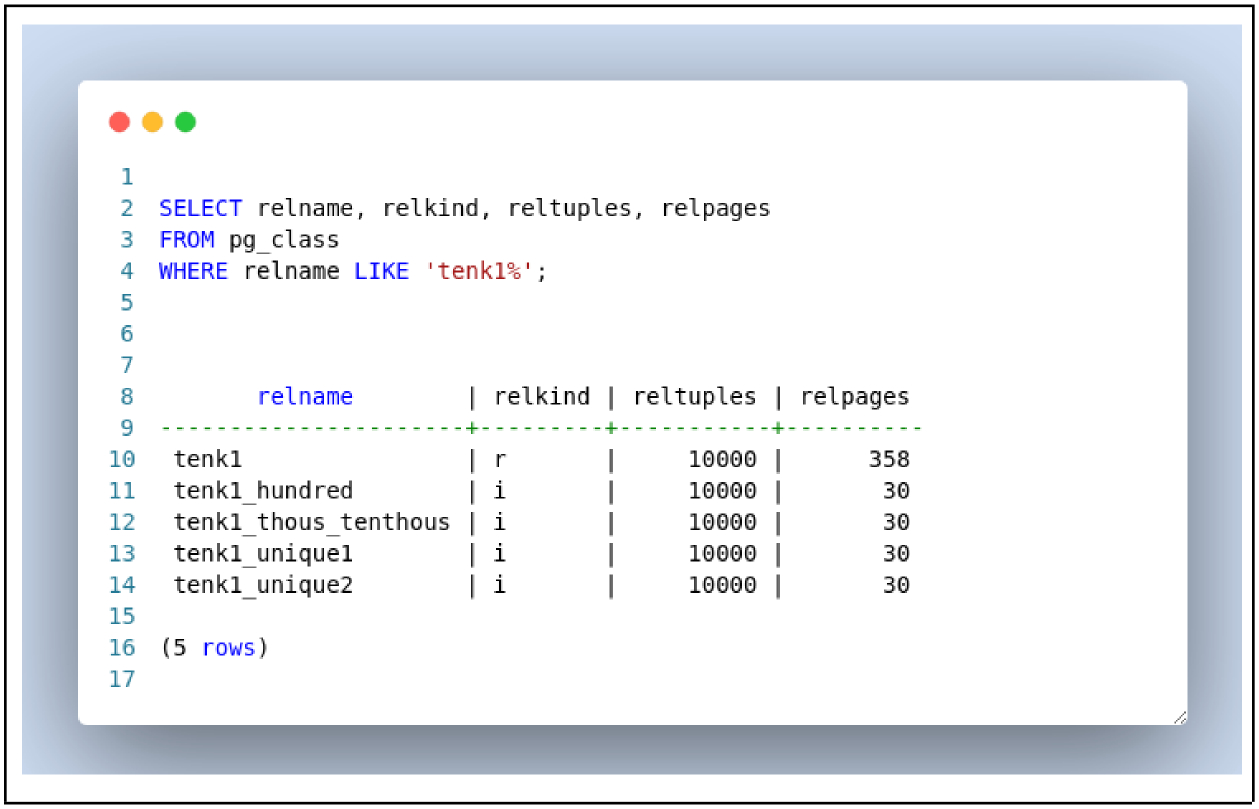
In addition, you can define your own range types; see CREATE TYPE for more information.[1]



```sql
CREATE TABLE ProductsInSupermarket(ProductID int, price int8range);
INSERT INTO ProductsInSupermarket VALUES
    (1108, '[10,2000)');

-- Containment
SELECT int8range(10, 200) @> 3;

-- Overlaps
SELECT numrange(11.1, 22.2) && numrange(20.0, 30.0);

-- Extract the upper bound
SELECT upper(int8range(15, 25));

-- Compute the intersection
SELECT int4range(10, 20) * int4range(15, 25);

-- Is the range empty?
SELECT isempty(numrange(1, 5));

```

1.2 Statistics

In Postgresql in order to select an optimal plan, the query planner uses various table level and column level statistics. For tables it stores the total number of entries in each table and index, as well as the number of disk blocks occupied by each table and index. These values are updated by VACUUM ANALYZE, and a few DDL commands such as CREATE INDEX. A VACUUM or ANALYZE operation that does not scan the entire table. Though these values are approximate, the planner will scale the values it finds in pg_class to match the current physical table size, thus obtaining a closer approximation.[1]



```

SELECT relname, relkind, reltuples, relpages
FROM pg_class
WHERE relname LIKE 'tenk1%';



       relname        | relkind | reltuples | relpages
----------------------+---------+-----------+----------
 tenk1                | r       |     10000 |      358
 tenk1_hundred        | i       |     10000 |       30
 tenk1_thous_tenthous | i       |     10000 |       30
 tenk1_unique1        | i       |     10000 |       30
 tenk1_unique2        | i       |     10000 |       30

(5 rows)

```

Another part of query planning is to estimate the selectivity of the WHERE clause. This is because many queries return only a fraction of the rows in the table, due to filtering done by WHERE clause. As part of query planning this information can be found in the pg_statistic system catalog**.** Entries in pg_statistic are updated by the ANALYZE and VACUUM ANALYZE commands. But for the purposes of understanding the statistics and viewing them use pg_stats which is a view on top of pg_statistics. The amount of information stored in pg_statistic by ANALYZE, in particular the maximum number of entries in the most_common_vals and histogram_bounds arrays for each column, can be set on a column-by-column basis using the ALTER TABLE SET STATISTICS command, or globally by setting the default_statistics_target configuration variable. The default limit is presently 100 entries. Raising the limit might allow more accurate planner estimates to be made, particularly for columns with irregular data distributions, at the price of consuming more space in pg_statistic and slightly more time to compute the estimates. Conversely, a lower limit might be sufficient for columns with simple data distributions.[1]

Columns in pg_stats :

Name	Type	References	Description
schemaname	name	pg_namespace.nspname	Name of schema containing table
tablename	name	pg_class.relname	Name of table
attname	name	pg_attribute.attname	Name of the column described by this row
null_frac	real		Fraction of column entries that are null
avg_width	integer		Average width in bytes of column's entries
n_distinct	real		If greater than zero, the estimated number of distinct values in the column. If less than zero, the negative of the number of distinct values divided by the number of rows. (The negated form is used when ANALYZE believes that the number of distinct values is likely to increase as the table grows; the positive form is used when the column seems to have a fixed number of possible values.) For example, -1 indicates a unique column in which the number of distinct values is the same as the number of rows

most_common_vals	anyarray		A list of the most common values in the column. (NULL if no values seem to be more common than any others.) For some datatypes such as tsvector, this is a list of the most common element values rather than values of the type itself.
most_common_freqs	real[]		A list of the frequencies of the most common values or elements, i.e., number of occurrences of each divided by total number of rows. (NULL when most_common_vals is.) For some datatypes such as tsvector, it can also store some additional information, making it longer than the most_common_vals array.
histogram_bounds	anyarray		A list of values that divide the column's values into groups of approximately equal population. The values in most_common_vals, if present, are omitted from this histogram calculation. (This column is NULL if the column data type does not have a < operator or if the most_common_vals list accounts for the entire population.)

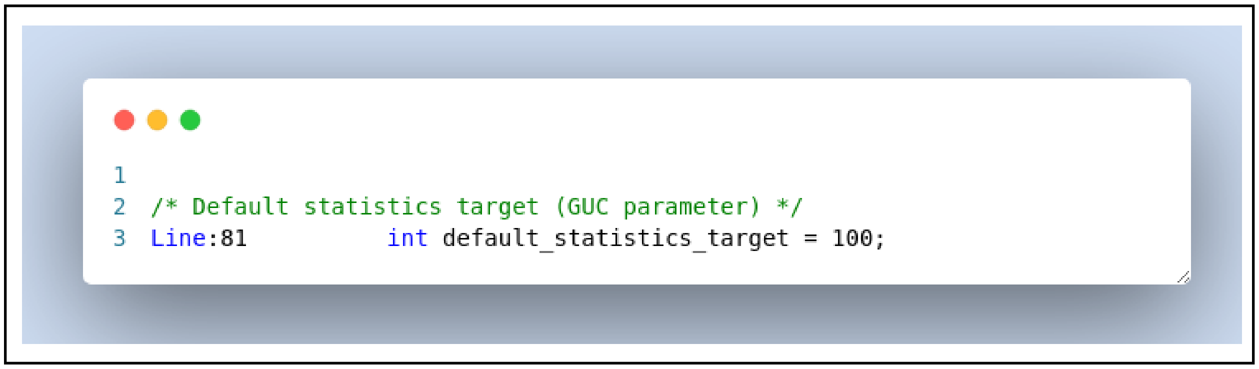
correlation	real		Statistical correlation between physical row ordering and logical ordering of the column values. This ranges from -1 to +1. When the value is near -1 or +1, an index scan on the column will be estimated to be cheaper than when it is near zero, due to reduction of random access to the disk. (This column is NULL if the column data type does not have a < operator.)

1.3 Statistics in Range Types

The statistics used by range type columns are different from those used for columns of other data types. For a range type column, statistics stored are:

- Equi-depth histograms of lower and upper bounds
- Equi-depth histogram of lengths
- Fraction of NULLs
- Fraction of empty ranges

As mentioned the number of maximum bins are fixed for the histograms and they are fixed by default[1]. In analyze.c it is fixed by:



```

/* Default statistics target (GUC parameter) */
Line:81         int default_statistics_target = 100;
```

2. Range Type analyze New Statistics

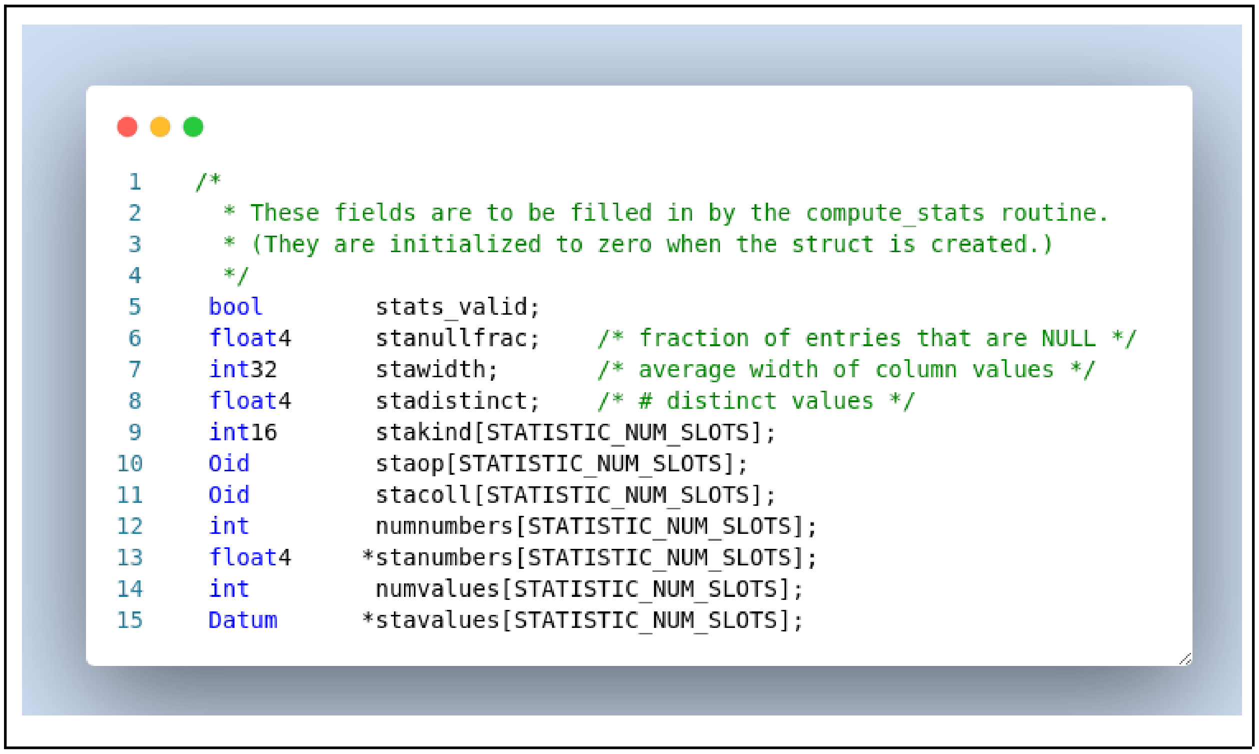
In this section we explain how we implemented a new data structure in PostgreSQL statistics. This new data structure or statistics uses original unsorted samples to construct an histogram-like array, which has higher values for regions where there are more overlaps of ranges.

In the PostgreSQL source code the location of this procedure is at - **src/backend/utils/adt/rangetypes_typanalyze.c**

Rangetypes_typanalyze.c - is a procedure in PostgreSQL for gathering statistics from range columns.

It is executed when we run the below commands -
VACUUM ANALYZE <table_name>;

The statistics analysed by this procedure are persisted in pg_statistics and in code the - VacAttrStats structure is used to hold these values. In our case Rangetypes_typanalyze.c will populate the following fields of VacAttrStats.



```
/*
  * These fields are to be filled in by the compute_stats routine.
  * (They are initialized to zero when the struct is created.)
  */
 bool        stats_valid;
 float4      stanullfrac;    /* fraction of entries that are NULL */
 int32       stawidth;       /* average width of column values */
 float4      stadistinct;    /* # distinct values */
 int16       stakind[STATISTIC_NUM_SLOTS];
 Oid         staop[STATISTIC_NUM_SLOTS];
 Oid         stacoll[STATISTIC_NUM_SLOTS];
 int         numnumbers[STATISTIC_NUM_SLOTS];
 float4     *stanumbers[STATISTIC_NUM_SLOTS];
 int         numvalues[STATISTIC_NUM_SLOTS];
 Datum      *stavalues[STATISTIC_NUM_SLOTS];
```

For a range type column, histograms of lower and upper bounds , and fraction of NULL and fraction of empty ranges are collected.

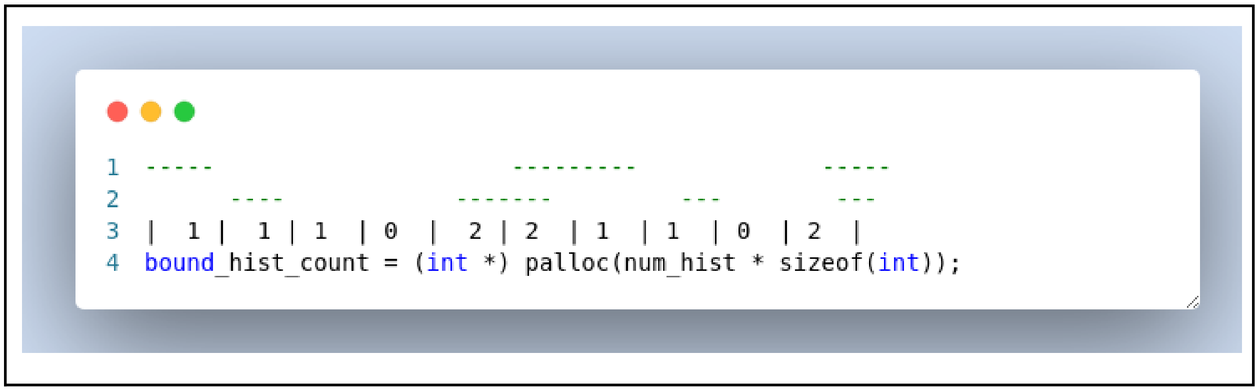
Both histograms have the same length, and they are combined into a single array of ranges. This has the same shape as the histogram that std_typanalyze would collect, but the values are different. Each range in the array is a valid range, even though the lower and upper bounds come from different tuples.

The task is about implementing a new data structure for improving the current statistics. These statistics are used to calculate rangetype selectivity and join cardinality estimates. The data-structure that we tried to implement is based on the concepts taught by Prof. Mahmoud SAKR in the Advanced Database System Architecture class, as well as in the exercise lab sessions. In this we create an array which can maintain or represent the position of ranges in comparison to all the other ranges in a simple way. The weakness of the current statistics structure is that it loses the original link between the sampled ranges. It stores all the lower & upper bounds but it sorts them, thus losing the spatial awareness of the ranges. Our data structure solves this problem.

The solution works as follows :- A float array is defined, which conceptually represents the minimum value to maximum value from the sampled ranges.

Next we make fixed bins out of this array. In this case it becomes an equi-width histogram. In PostgreSQL the number of bins is fixed by default so we will also be following the same convention and use bins specified by num_bins = stats->attr->attstattarget. Finally we compare the sampled ranges with the bins (each bin represents a mini range) in the array, to find how many ranges overlap with the bin and put that value in the index corresponding to that conceptual range.

The diagram 2.1 below showcases the visual representation of the new structure and how it's calculated.

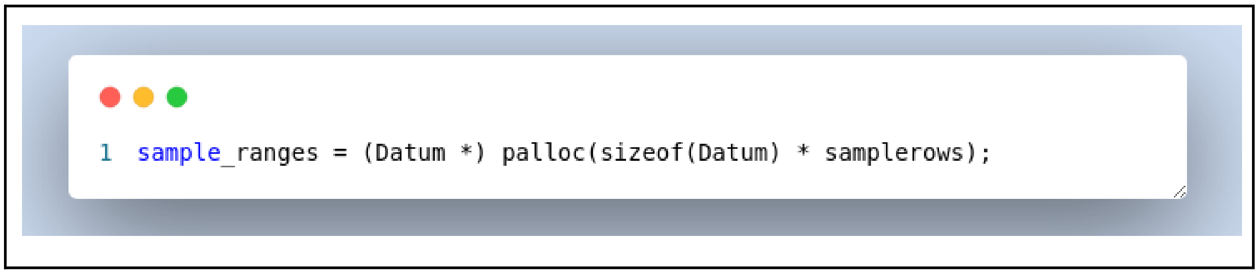


```
-----                     ---------             -----
      ----            -------         ---        ---
|  1 |  1 | 1  | 0  |  2 | 2  | 1  | 1  | 0  | 2  |
bound_hist_count = (int *) palloc(num_hist * sizeof(int));
```

Diagram 2.1 Visual representation of the new structure and its calculations

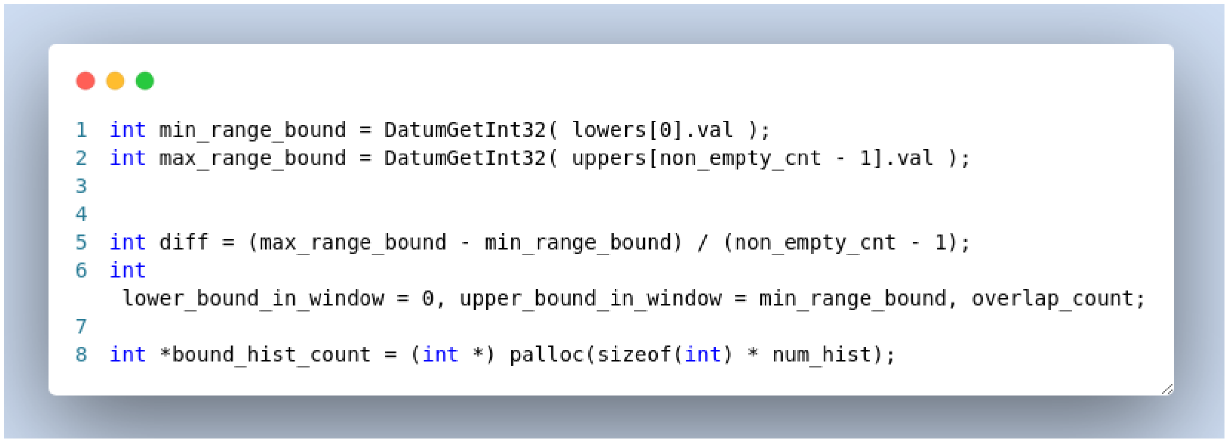
Implementation Details

Sample_ranges is an array used to maintain a copy of sample ranges, because the lower & upper ranges used by bound_hist_values statistics are sorted.



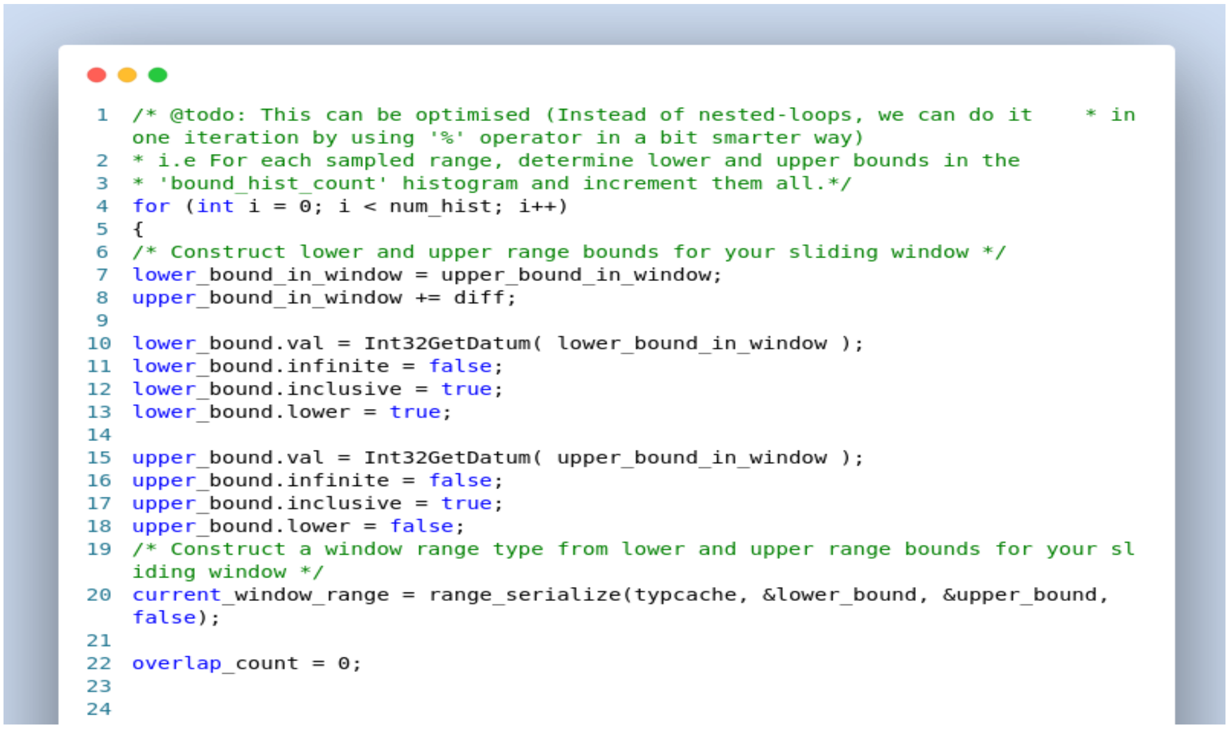
```
sample_ranges = (Datum *) palloc(sizeof(Datum) * samplerows);
```

Initialize and/ or set variables for bound_hist_count calculation:

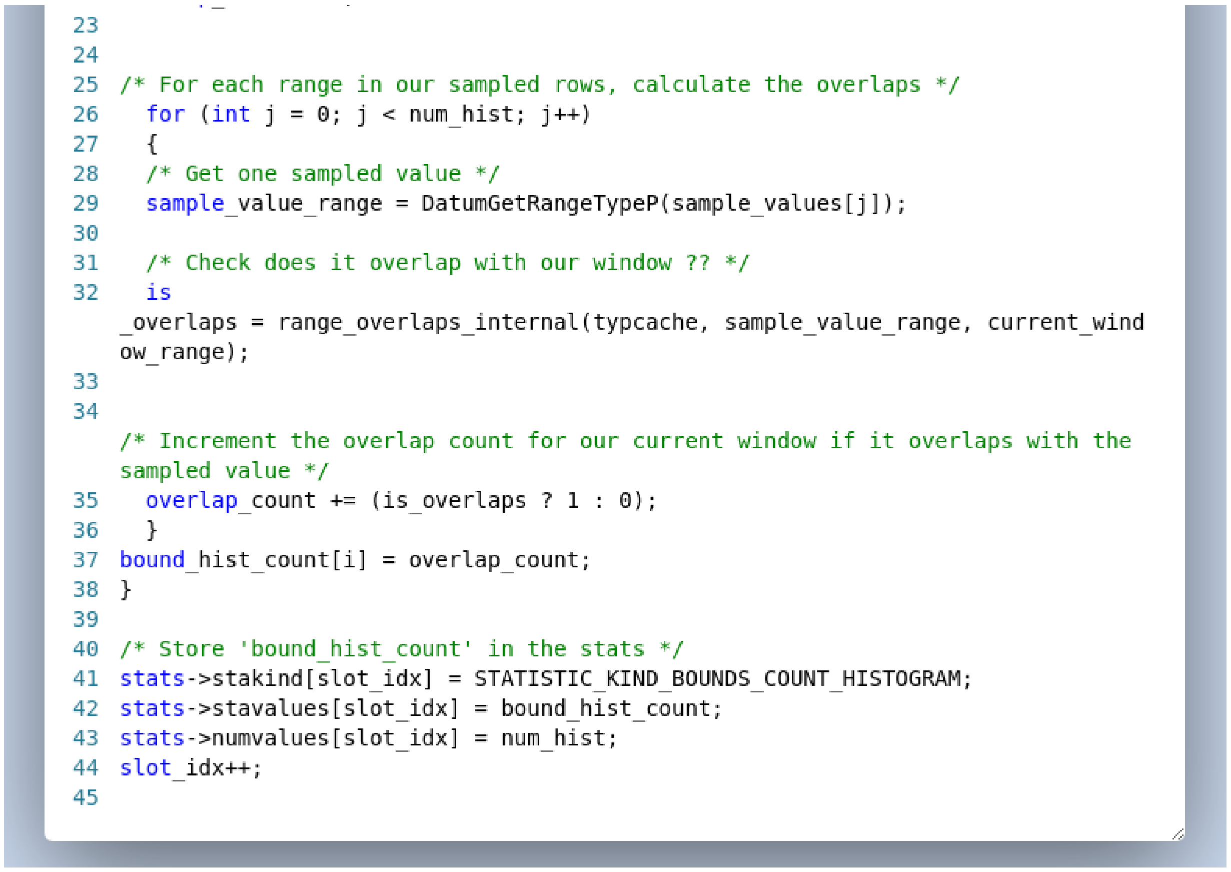


```c
int min_range_bound = DatumGetInt32( lowers[0].val );
int max_range_bound = DatumGetInt32( uppers[non_empty_cnt - 1].val );


int diff = (max_range_bound - min_range_bound) / (non_empty_cnt - 1);
int lower_bound_in_window = 0, upper_bound_in_window = min_range_bound, overlap_count;

int *bound_hist_count = (int *) palloc(sizeof(int) * num_hist);
```

Constructing a lower and upper range bounds for the sliding window and a window range type :



```c
/* @todo: This can be optimised (Instead of nested-loops, we can do it    * in one iteration by using '%' operator in a bit smarter way)
* i.e For each sampled range, determine lower and upper bounds in the
* 'bound_hist_count' histogram and increment them all.*/
for (int i = 0; i < num_hist; i++)
{
/* Construct lower and upper range bounds for your sliding window */
lower_bound_in_window = upper_bound_in_window;
upper_bound_in_window += diff;

lower_bound.val = Int32GetDatum( lower_bound_in_window );
lower_bound.infinite = false;
lower_bound.inclusive = true;
lower_bound.lower = true;

upper_bound.val = Int32GetDatum( upper_bound_in_window );
upper_bound.infinite = false;
upper_bound.inclusive = true;
upper_bound.lower = false;
/* Construct a window range type from lower and upper range bounds for your sliding window */
current_window_range = range_serialize(typcache, &lower_bound, &upper_bound, false);

overlap_count = 0;


```



```c


/* For each range in our sampled rows, calculate the overlaps */
  for (int j = 0; j < num_hist; j++)
  {
  /* Get one sampled value */
  sample_value_range = DatumGetRangeTypeP(sample_values[j]);

  /* Check does it overlap with our window ?? */
  is_overlaps = range_overlaps_internal(typcache, sample_value_range, current_window_range);


/* Increment the overlap count for our current window if it overlaps with the sampled value */
  overlap_count += (is_overlaps ? 1 : 0);
  }
bound_hist_count[i] = overlap_count;
}

/* Store 'bound_hist_count' in the stats */
stats->stakind[slot_idx] = STATISTIC_KIND_BOUNDS_COUNT_HISTOGRAM;
stats->stavalues[slot_idx] = bound_hist_count;
stats->numvalues[slot_idx] = num_hist;
slot_idx++;

```

3. Selectivity Estimation Function

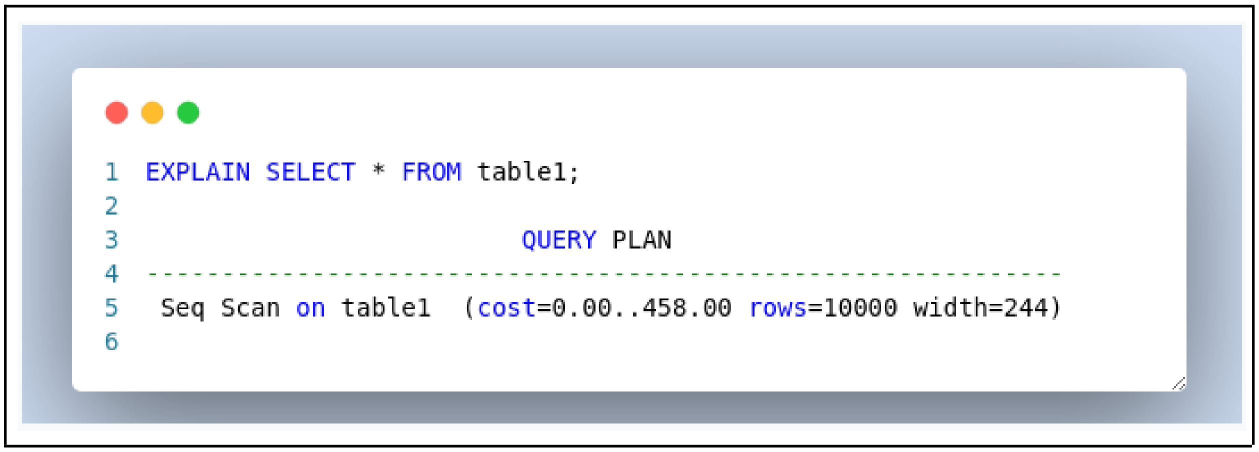
PostgreSQL devises a query plan for each query it receives. Choosing the right plan to match the query structure and the properties of the data is absolutely critical for good performance, so the system includes a complex planner that tries to choose good plans. You can use the EXPLAIN command to see what query plan the planner creates for any query.[1]

Hereby, we just explain the selectivity estimation function working in PostgreSQL as our ingenious solution used existing data-structures and so we weren't needed to implement this functionality as it was already implemented.

3.1 EXPLAIN Basics

The structure of a query plan is a tree of plan nodes. Nodes at the bottom level of the tree are scan nodes: they return raw rows from a table. There are different types of scan nodes for different table access methods: sequential scans, index scans, and bitmap index scans. There are also non-table row sources, such as VALUES clauses and set-returning functions in FROM, which have their own scan node types. If the query requires joining, aggregation, sorting, or other operations on the raw rows, then there will be additional nodes above the scan nodes to perform these operations. Again, there is usually more than one possible way to do these operations, so different node types can appear here too. The output of EXPLAIN has one line for each node in the plan tree, showing the basic node type plus the cost estimates that the planner made for the execution of that plan node. The very first line (the summary line for the topmost node) has the estimated total execution cost for the plan; it is this number that the planner seeks to minimize.[1]

Here is a trivial example, just to show what the output looks like:



```
EXPLAIN SELECT * FROM table1;

                         QUERY PLAN
-------------------------------------------------------------
 Seq Scan on table1  (cost=0.00..458.00 rows=10000 width=244)

```

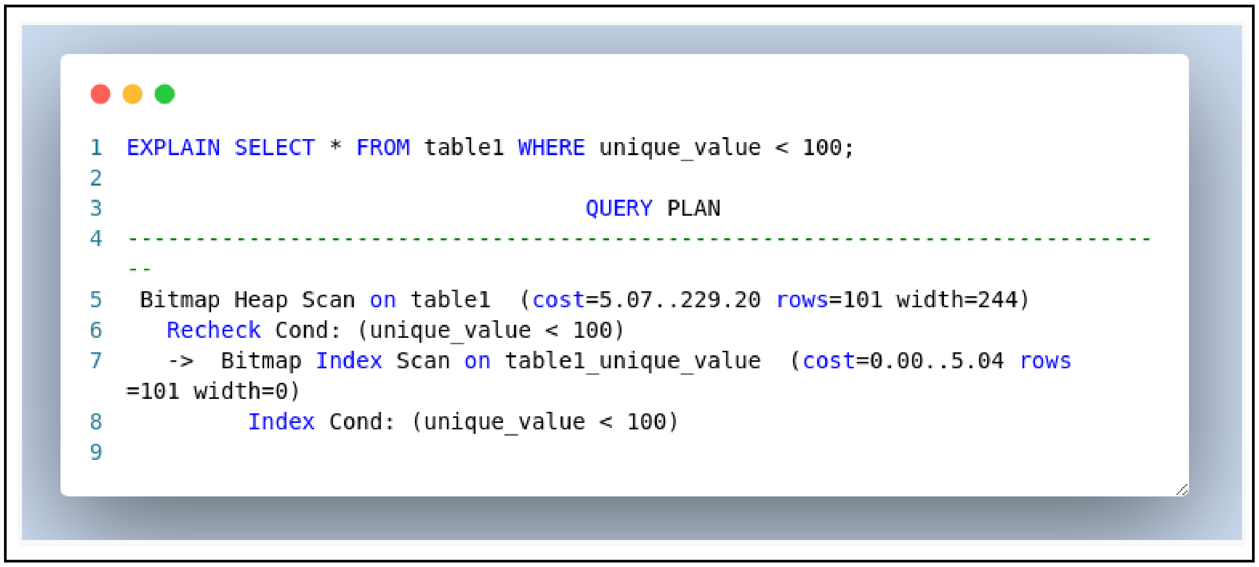
Since this query has no WHERE clause, it must scan all the rows of the table, so the planner has chosen to use a simple sequential scan plan. The numbers that are quoted in parentheses are (left to right):

- Estimated start-up cost. This is the time expended before the output phase can begin, e.g., time to do the sorting in a sort node.
- Estimated total cost. This is stated on the assumption that the plan node is run to completion, i.e., all available rows are retrieved. In practice a node's parent node might stop short of reading all available rows (see the LIMIT example below).
- Estimated number of rows output by this plan node. Again, the node is assumed to be run to completion.
- Estimated average width of rows output by this plan node (in bytes).

The costs are measured in arbitrary units determined by the planner's cost parameters. Traditional practice is to measure the costs in units of disk page fetches; that is, seq_page_cost is conventionally set to 1.0 and the other cost parameters are set relative to that.
It's important to understand that the cost of an upper-level node includes the cost of all its child nodes. It's also important to realize that the cost only reflects things that the planner cares about. In particular, the cost does not consider the time spent transmitting result rows to the client, which could be an important factor in the real elapsed time; but the planner ignores it because it cannot change it by altering the plan.
The rows value is a little tricky because it is not the number of rows processed or scanned by the plan node, but rather the number emitted by the node. This is often less than the

number scanned, as a result of filtering by any WHERE-clause conditions that are being applied at the node. Ideally the top-level rows estimate will approximate the number of rows actually returned, updated, or deleted by the query.[1]

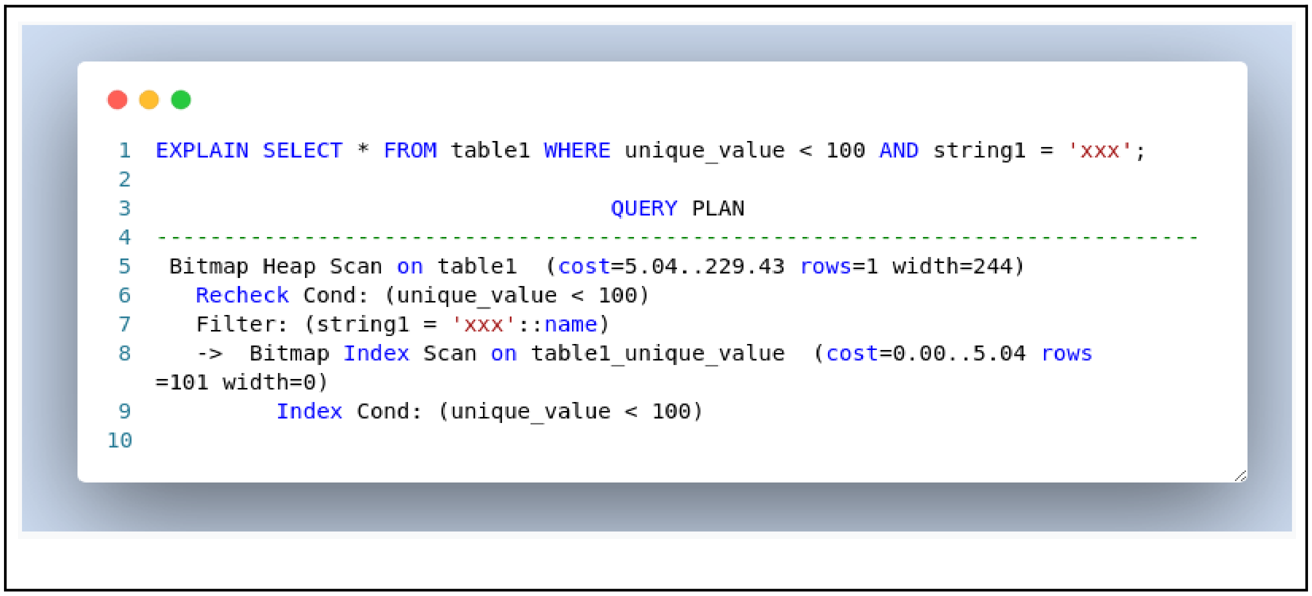
Now, let's make the condition more restrictive:



```
EXPLAIN SELECT * FROM table1 WHERE unique_value < 100;

                                  QUERY PLAN
----------------------------------------------------------------------------
--
 Bitmap Heap Scan on table1  (cost=5.07..229.20 rows=101 width=244)
   Recheck Cond: (unique_value < 100)
   ->  Bitmap Index Scan on table1_unique_value  (cost=0.00..5.04 rows
=101 width=0)
         Index Cond: (unique_value < 100)

```

Here the planner has decided to use a two-step plan: the child plan node visits an index to find the locations of rows matching the index condition, and then the upper plan node actually fetches those rows from the table itself. Fetching rows separately is much more expensive than reading them sequentially, but because not all the pages of the table have to be visited, this is still cheaper than a sequential scan. (The reason for using two plan levels is that the upper plan node sorts the row locations identified by the index into physical order before reading them, to minimize the cost of separate fetches.[1]

Now let's add another condition to the WHERE clause:



```
EXPLAIN SELECT * FROM table1 WHERE unique_value < 100 AND string1 = 'xxx';

                                  QUERY PLAN
------------------------------------------------------------------------------
 Bitmap Heap Scan on table1  (cost=5.04..229.43 rows=1 width=244)
   Recheck Cond: (unique_value < 100)
   Filter: (string1 = 'xxx'::name)
   ->  Bitmap Index Scan on table1_unique_value  (cost=0.00..5.04 rows
=101 width=0)
         Index Cond: (unique_value < 100)
```

The added condition string1 = 'xxx' reduces the output row count estimate, but not the cost because we still have to visit the same set of rows. Notice that the string1 clause cannot be applied as an index condition, since this index is only on the unique_value column. Instead it is applied as a filter on the rows retrieved by the index. Thus the cost has actually gone up slightly to reflect this extra checking.[1]

4. Join Cardinality Estimation Function

Inorder to do perform join cardinality estimation we are using the existing statistics. Note that we are not leveraging the new statistic for this implementation.

In the Postgresql source code the location of this procedure is at -
src/backend/utils/adt/geo_selfuncs.c

geo_selfuncs.c - is a procedure in Postgresql for selectivity routines registered in the operator catalog in the “oprrest” and “oprjoin” attributes.

It is executed when we run the below commands -
VACUUM ANALYZE <table_name>;

Implementation Details

The task is about implementing the function to calculate the actual overlaps between the given rangetypes. The solution that we tried to implement is as shown in the below figure:

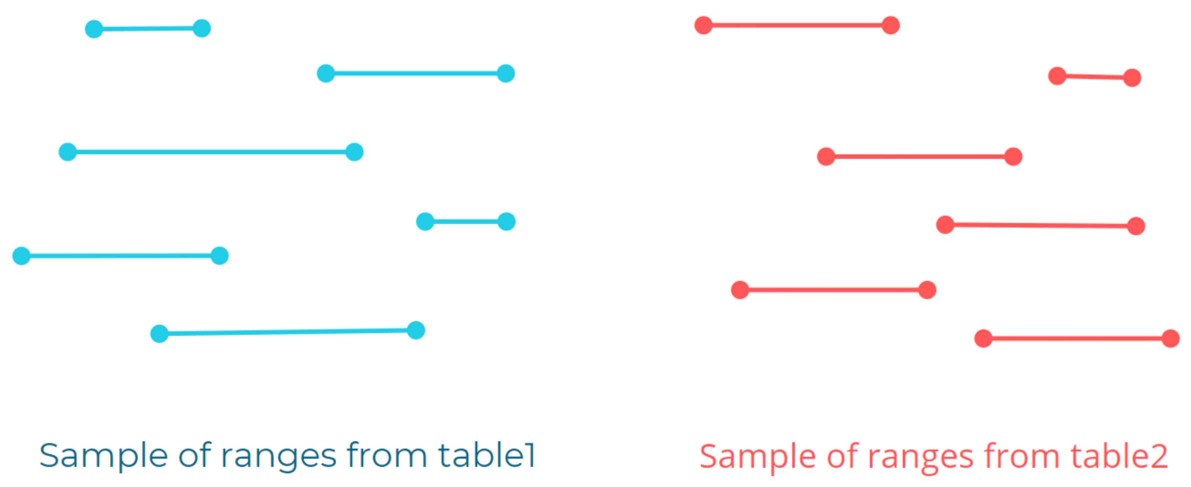


Figure 3.1 View of sample values from both tables

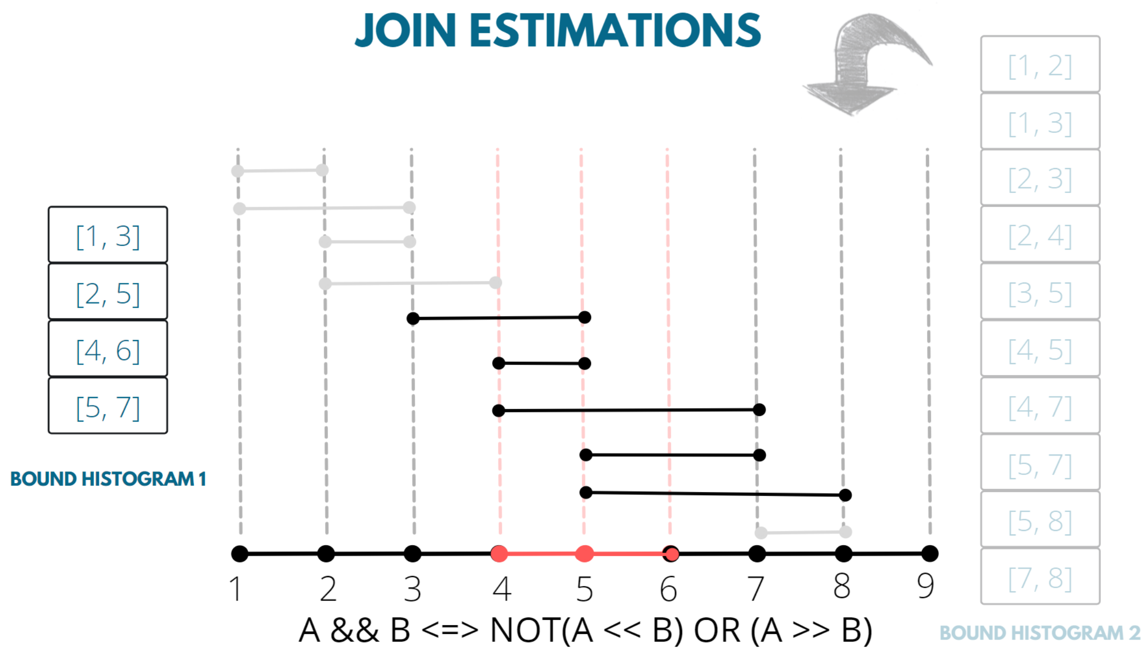


Figure 3.2 Join cardinality estimation function i.e. *overlap*

Here A represents ranges from the 1st table, B represents the ranges from the 2nd table. For calculating the actual overlap that is happening, we mainly perform two types of selectivity - left and right. In selectivity function, we calculate the overlapping points based on the below formula :

A && B <=> NOT (A >> B OR A << B)

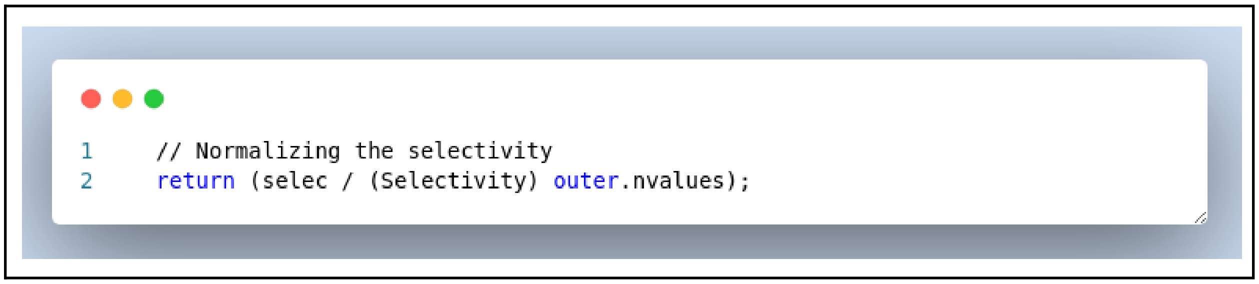
<u>Higher level idea of the solution</u>

The ingenious idea that we had was, in order to calculate the estimate of overlap between two tables, we calculate the overlap between the two ***bound_hist_values*** stats of both the tables. The idea is that bound_hist_values though not maintaining the original ranges from the table, would still be somewhat of a presentation of ranges inside the tables. So a good assumption can be that tables and their bound_hist_value overlaps with similar fractions.

Some low level details of implementation

Normally the algorithm to compare all the ranges in, say, table “A” with all the ranges in table “B” is O(n^2). But we have tried to optimize this a lot and reduced the complexity to O (n* log(n)). The first N comes from iterating one table’s “bound_hist_values”, the Log(n) comes from comparing a single range from 1st table’s bound_hist_value with 2nd table’s bound_hist_value, using the . Some key implementational details are as follows:

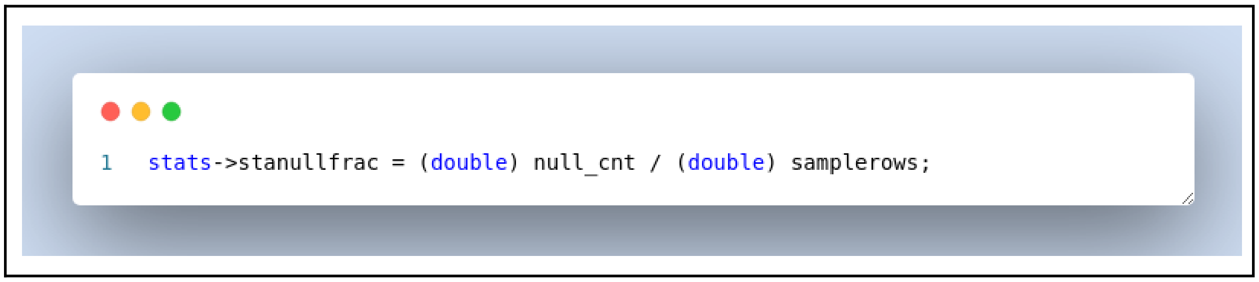
1. First we determine with table has the less values in its bound_hist_value. A in this case.
2. We iterate over A’s bound_hist_value, this becomes our outer loop.
3. We split the bound_hist_values of the other table, B’s in this case into range-bounds of “lowers” and “uppers”. This is because we are trying to use the existing function “calculate_hist_selectivity” to calculate the overlapping fractions.
4. Next the loop runs as follows:
    a. In the start of the for loop we take one value from stats (bound_hist_value) of A.
    b. The lower-bound of A is compared with the “uppers” (upper-bonds) array of B, and percentage of selectivity is calculated. Next upper-bound of A is compared with the “lowers” (lower-bounds) array of B and selectivity of this is added to the previous selectivity. Finally this is subtracted from 1 to get the selectivity of overlap. This formula sums of the idea used here:
    A && B <=> NOT (A >> B OR A << B)
    c. To compensate the values in selectivity we normalize the values by the number of values in A (the outer loop) as:



```
// Normalizing the selectivity
return (selec / (Selectivity) outer.nvalues);
```

5. For handling the duplicates in the calculation of overlap fraction, we fetch the next values of Lower & Upper from “A”, we compare them with last values and check for duplicates, if yes, we don’t check these new values for selectivity.

6. For catering for NULLs values in the tables, we are using the stuanullfrac from stats:

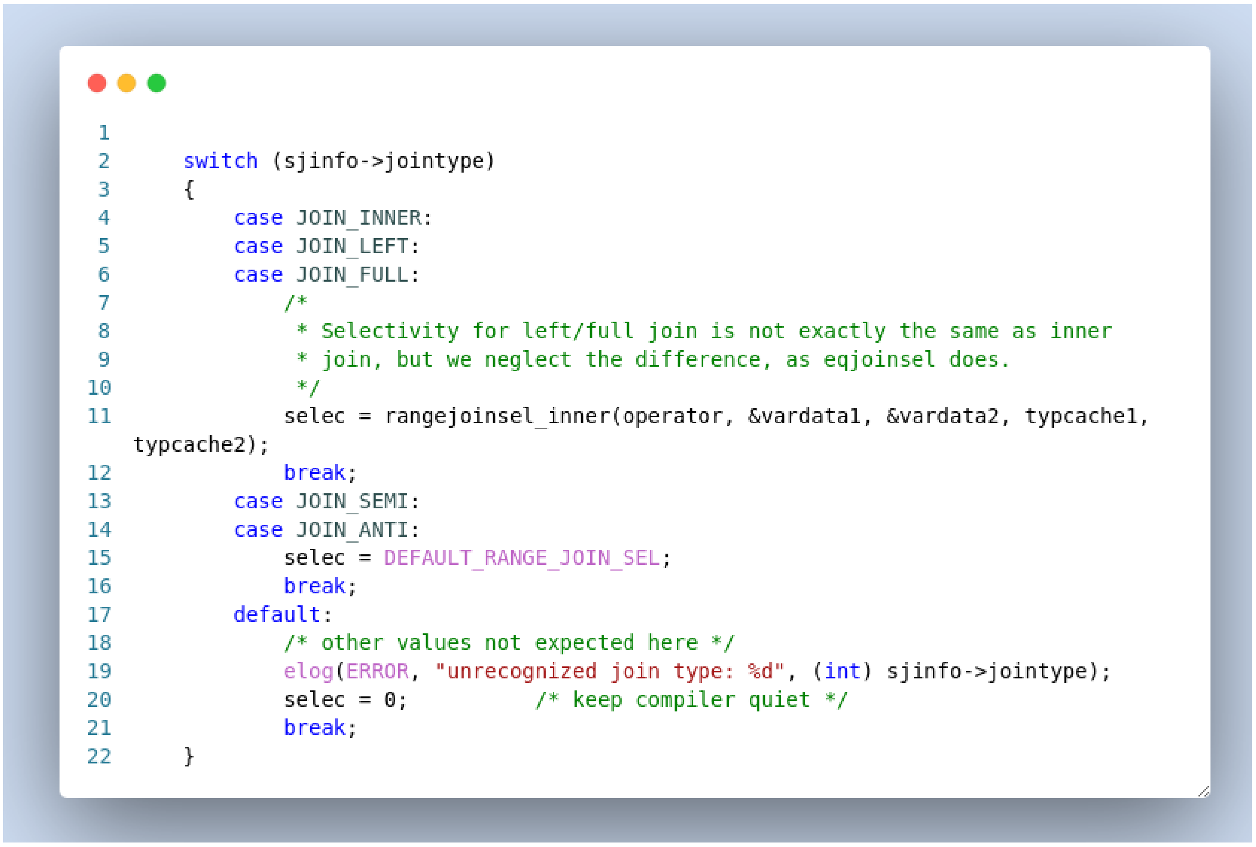


```
stats->stanullfrac = (double) null_cnt / (double) samplerows;
```

To tell us what fraction of values are NULL, next we subtract this estimation from the final final estimation as well. Also, we are using the for-loop to perform this operation recursively on all the range-type values in the sample.

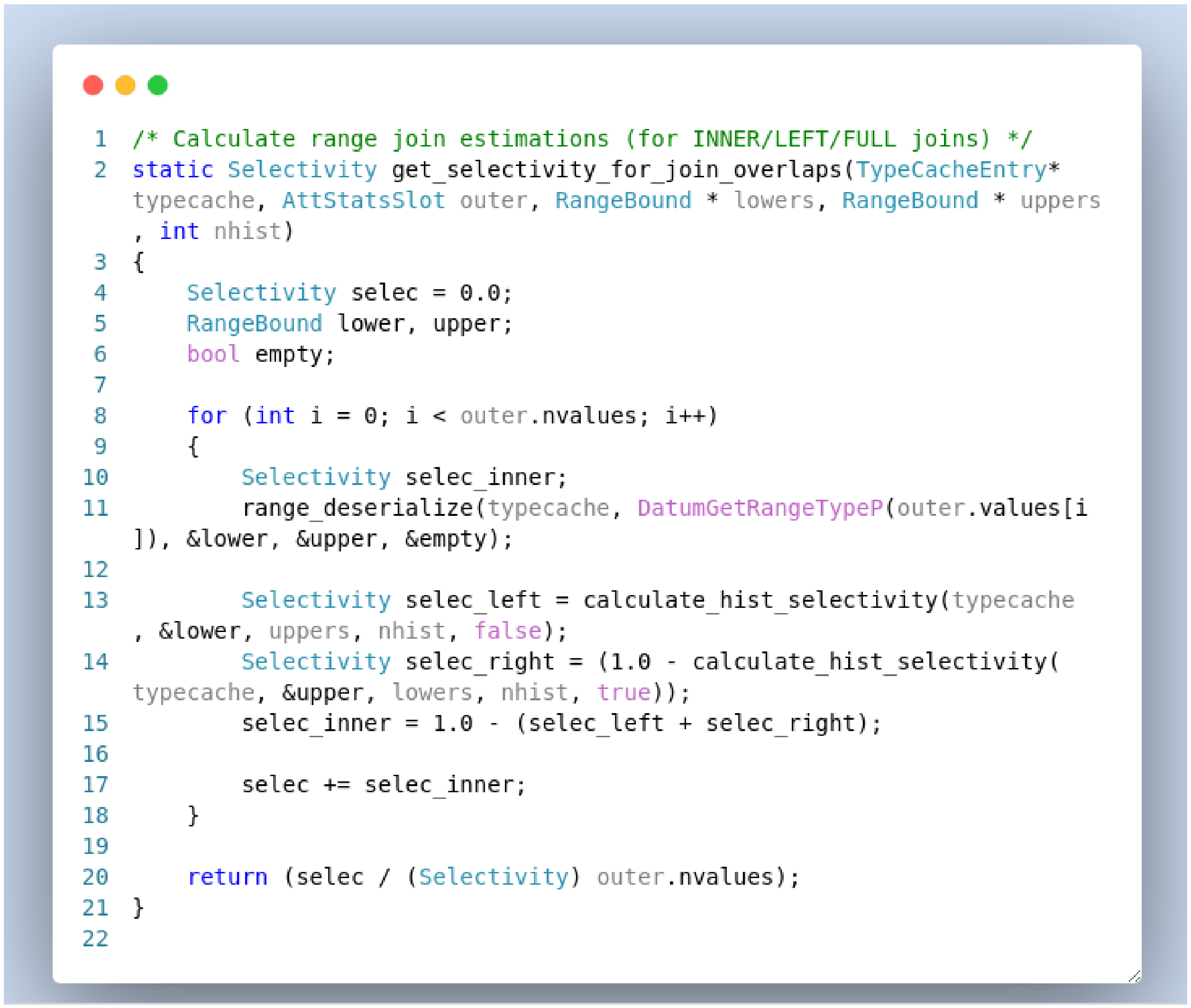
Implementation Steps

I. Calling major function (rangejoinsel_inner) based on join type



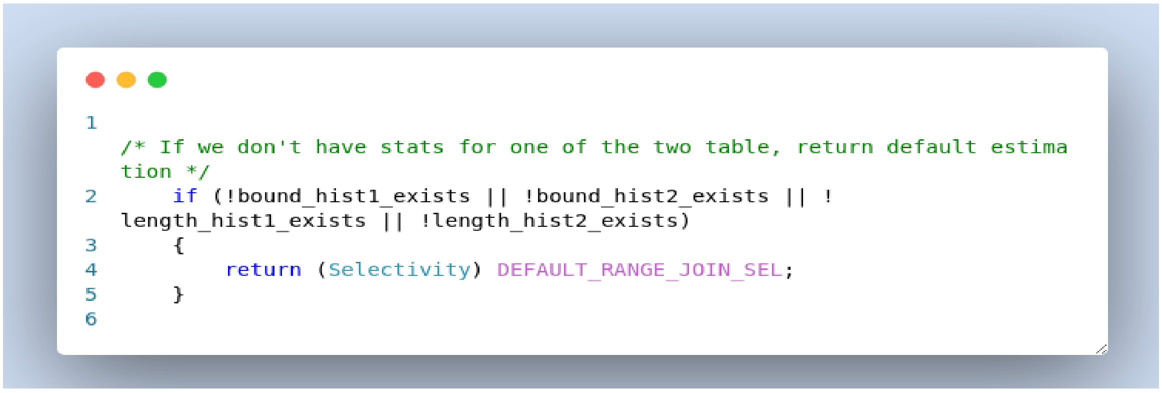
```

    switch (sjinfo->jointype)
    {
        case JOIN_INNER:
        case JOIN_LEFT:
        case JOIN_FULL:
            /*
             * Selectivity for left/full join is not exactly the same as inner
             * join, but we neglect the difference, as eqjoinsel does.
             */
            selec = rangejoinsel_inner(operator, &vardata1, &vardata2, typcache1, typcache2);
            break;
        case JOIN_SEMI:
        case JOIN_ANTI:
            selec = DEFAULT_RANGE_JOIN_SEL;
            break;
        default:
            /* other values not expected here */
            elog(ERROR, "unrecognized join type: %d", (int) sjinfo->jointype);
            selec = 0;          /* keep compiler quiet */
            break;
    }
```

II. Higher level function within rangejoinsel_inner to collect statistics and normalize selectivity



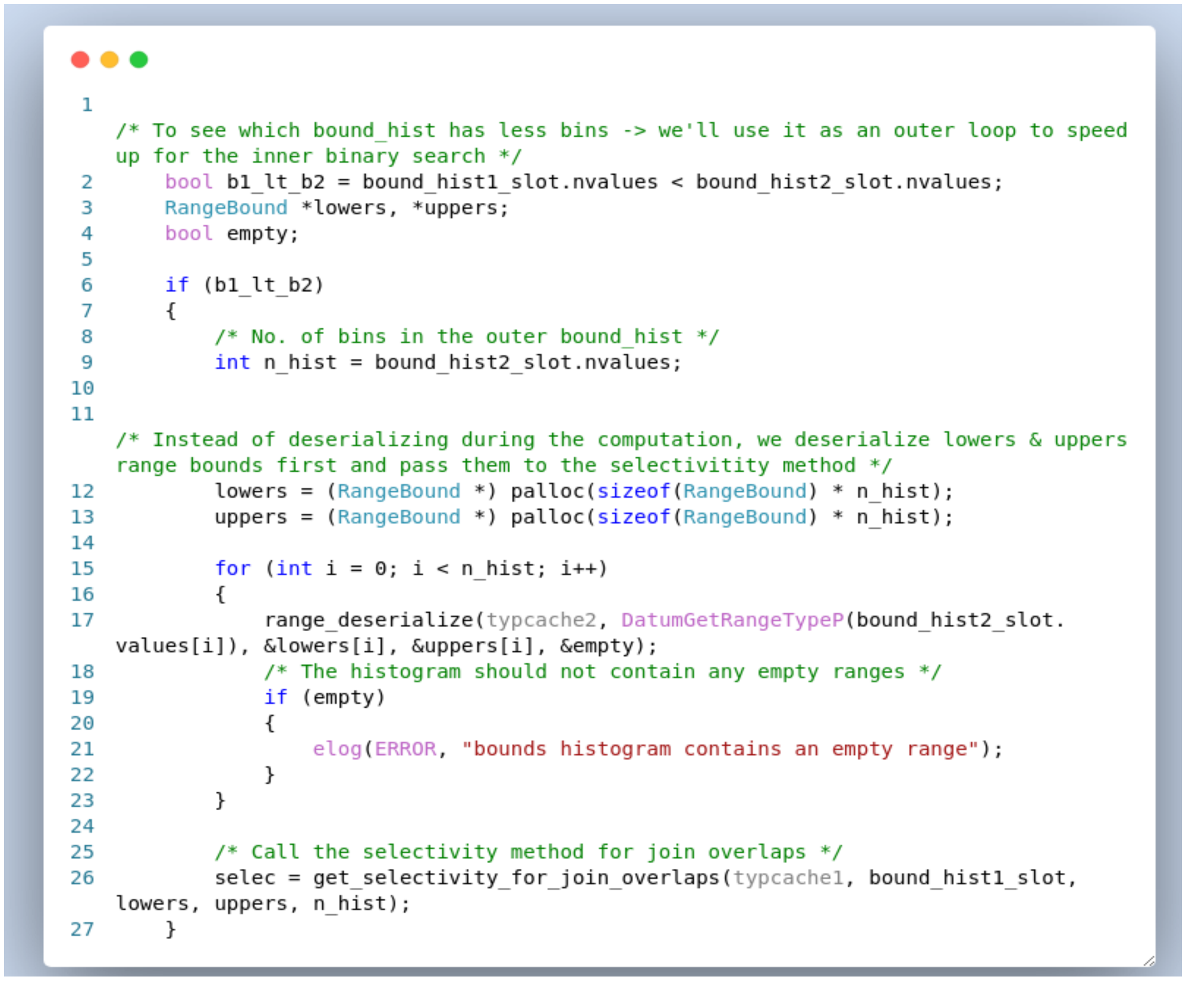
```c
/* Calculate range join estimations (for INNER/LEFT/FULL joins) */
static Selectivity get_selectivity_for_join_overlaps(TypeCacheEntry* typecache, AttStatsSlot outer, RangeBound * lowers, RangeBound * uppers, int nhist)
{
    Selectivity selec = 0.0;
    RangeBound lower, upper;
    bool empty;

    for (int i = 0; i < outer.nvalues; i++)
    {
        Selectivity selec_inner;
        range_deserialize(typecache, DatumGetRangeTypeP(outer.values[i]), &lower, &upper, &empty);

        Selectivity selec_left = calculate_hist_selectivity(typecache, &lower, uppers, nhist, false);
        Selectivity selec_right = (1.0 - calculate_hist_selectivity(typecache, &upper, lowers, nhist, true));
        selec_inner = 1.0 - (selec_left + selec_right);

        selec += selec_inner;
    }

    return (selec / (Selectivity) outer.nvalues);
}

```

III. Checking if Bound histogram doesn't exist return default value



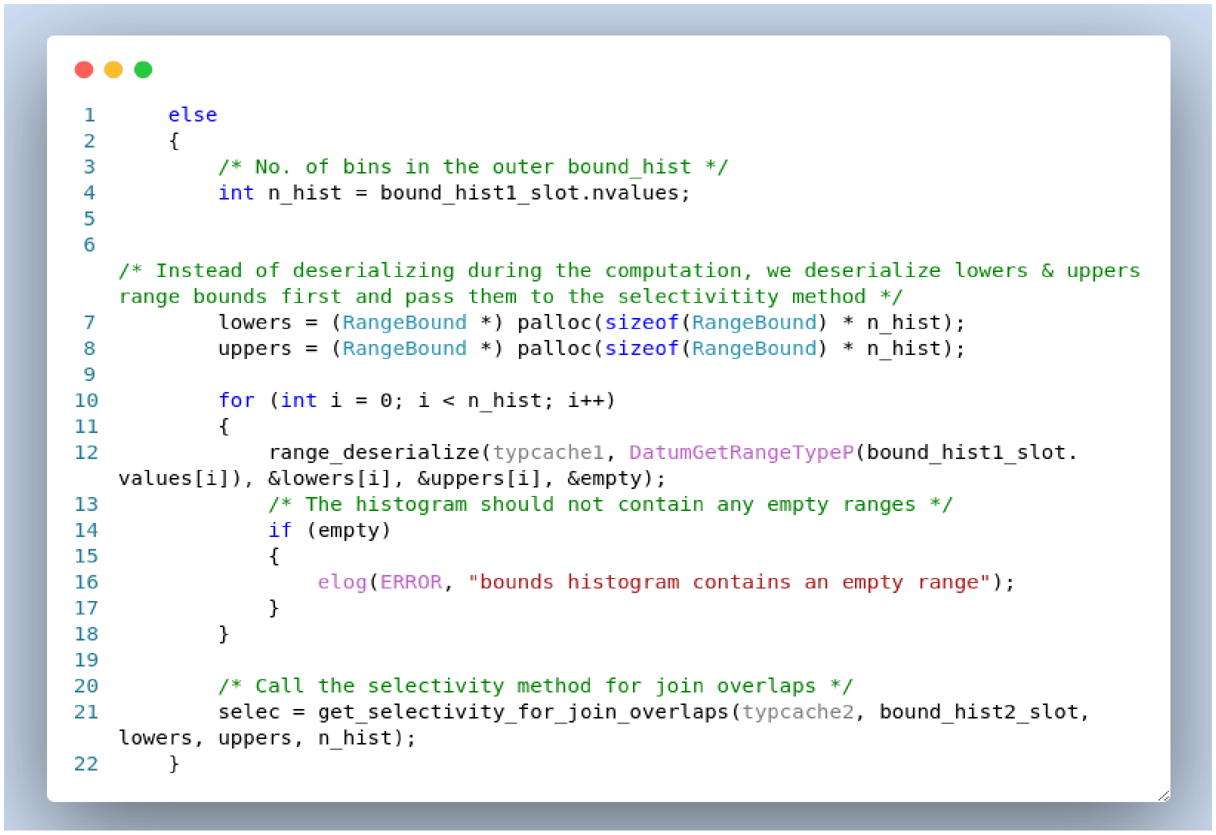
```c
/* If we don't have stats for one of the two table, return default estimation */
    if (!bound_hist1_exists || !bound_hist2_exists || !length_hist1_exists || !length_hist2_exists)
    {
        return (Selectivity) DEFAULT_RANGE_JOIN_SEL;
    }

```

IV. Based on the length of hist bins, If bin1 length is smaller perform binary search on bin2 histogram



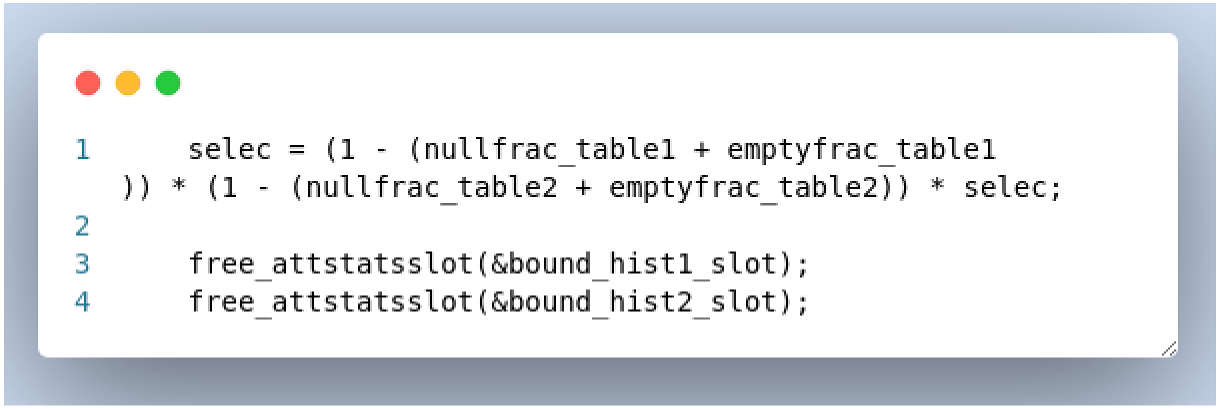
```c
/* To see which bound_hist has less bins -> we'll use it as an outer loop to speed up for the inner binary search */
    bool b1_lt_b2 = bound_hist1_slot.nvalues < bound_hist2_slot.nvalues;
    RangeBound *lowers, *uppers;
    bool empty;

    if (b1_lt_b2)
    {
        /* No. of bins in the outer bound_hist */
        int n_hist = bound_hist2_slot.nvalues;


/* Instead of deserializing during the computation, we deserialize lowers & uppers range bounds first and pass them to the selectivity method */
        lowers = (RangeBound *) palloc(sizeof(RangeBound) * n_hist);
        uppers = (RangeBound *) palloc(sizeof(RangeBound) * n_hist);

        for (int i = 0; i < n_hist; i++)
        {
            range_deserialize(typcache2, DatumGetRangeTypeP(bound_hist2_slot.values[i]), &lowers[i], &uppers[i], &empty);
            /* The histogram should not contain any empty ranges */
            if (empty)
            {
                elog(ERROR, "bounds histogram contains an empty range");
            }
        }

        /* Call the selectivity method for join overlaps */
        selec = get_selectivity_for_join_overlaps(typcache1, bound_hist1_slot, lowers, uppers, n_hist);
    }
```

V. Based on the length of hist bins, If bin2 length is smaller perform binary search on bin1 histogram



```c
    else
    {
        /* No. of bins in the outer bound_hist */
        int n_hist = bound_hist1_slot.nvalues;


/* Instead of deserializing during the computation, we deserialize lowers & uppers range bounds first and pass them to the selectivitity method */
        lowers = (RangeBound *) palloc(sizeof(RangeBound) * n_hist);
        uppers = (RangeBound *) palloc(sizeof(RangeBound) * n_hist);

        for (int i = 0; i < n_hist; i++)
        {
            range_deserialize(typcache1, DatumGetRangeTypeP(bound_hist1_slot.values[i]), &lowers[i], &uppers[i], &empty);
            /* The histogram should not contain any empty ranges */
            if (empty)
            {
                elog(ERROR, "bounds histogram contains an empty range");
            }
        }

        /* Call the selectivity method for join overlaps */
        selec = get_selectivity_for_join_overlaps(typcache2, bound_hist2_slot, lowers, uppers, n_hist);
    }
```

VI. Removing null and empty fractions from the final selectivity estimation



```c
    selec = (1 - (nullfrac_table1 + emptyfrac_table1)) * (1 - (nullfrac_table2 + emptyfrac_table2)) * selec;

    free_attstatsslot(&bound_hist1_slot);
    free_attstatsslot(&bound_hist2_slot);
```

Time Complexity

Time complexity estimates how an algorithm performs regardless of the kind of machine it runs on. We can get the time complexity by counting the number of operations performed by your code. This time complexity is defined as a function of the input size n using Big-O notation. n indicates the input size, while O is the worst-case scenario growth rate function.[2]

In the solution that we have implemented, we are using a binary search hence O(log n). Plus, we are sorting the elements in the array with a for-loop. Hence, the total time complexity of our algorithm is :

Time Complexity = O *(n log n)*

Where **n** is Total number of bins created (Default=100)

5. Benchmarking

All tables (relations) here are created with extreme randomization, to ensure that most cases are covered and we get an accurate comparison.

Explanation of headings in the tables:

- "Proposed Method row prediction v.1" refers to the implementation in which we did not account for duplicate ranges in the samples.
- "Proposed Method row prediction v.2" refers to the implementation in which we account for duplicate ranges in the samples.

As for "Vacuum Analyze" it is run each time for different tables, before and after the implementation of the proposed solution. Since time in case of solution-v1 & solution-v2 is very similar as we are using the original statistics only for the join cardinality estimation.

5.1 Int4range

Int4range contains the range of Integer values (A discrete data type)

1. Vacuum Time Analysis

Original Code (POSTGRES) (ms)	**PROPOSED METHOD (ms)**
12.280	11.867

2. Row Estimations Analysis

Queries	Actual rows in the table	Original Code (Postgres) row estimation	Proposed Method row prediction v.1	Proposed Method row prediction v.2	Error from original method (Postgres)	Error from proposed method v.1	Error from proposed method v.2	Method Accuracy
1	6888	50	6951	6951	0.9927	0.0090	0.0090	0.9909
2	2564	5000	2704	2704	0.4872	0.0517	0.0517	0.94822
3	66807998	500000	66516217	66516217	0.9925	0.0043	0.0043	0.9956

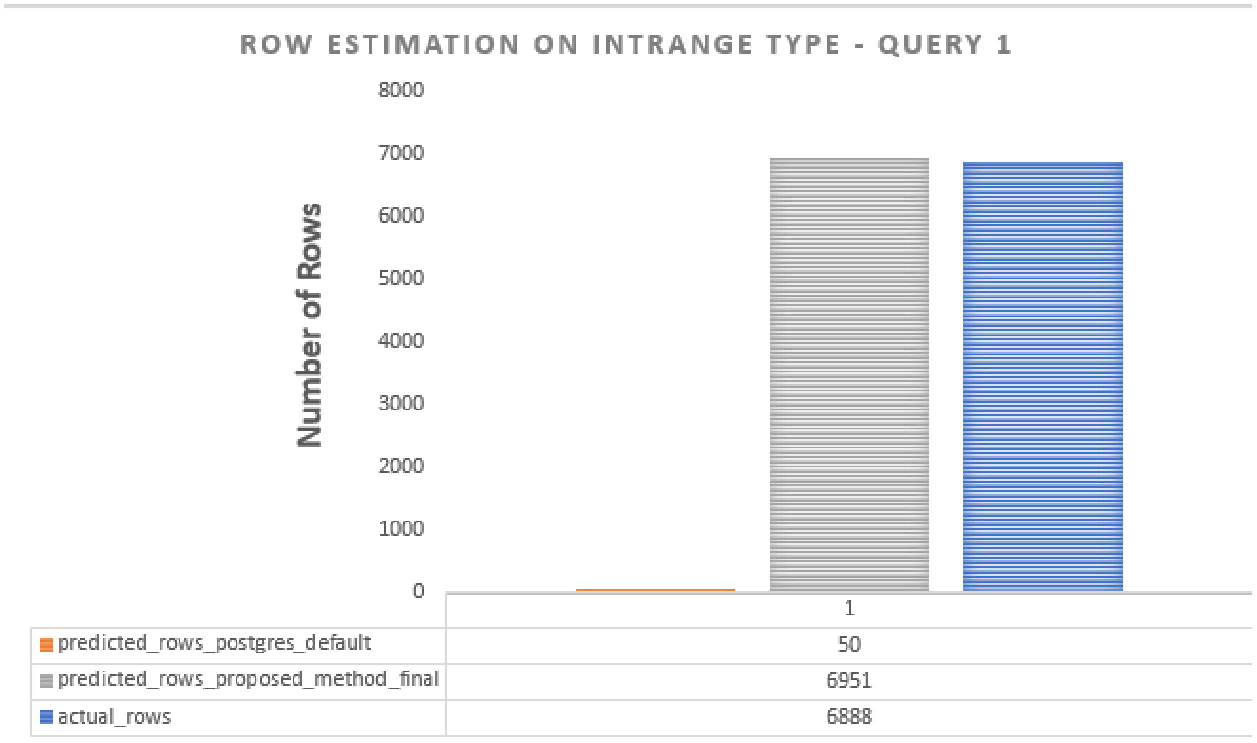
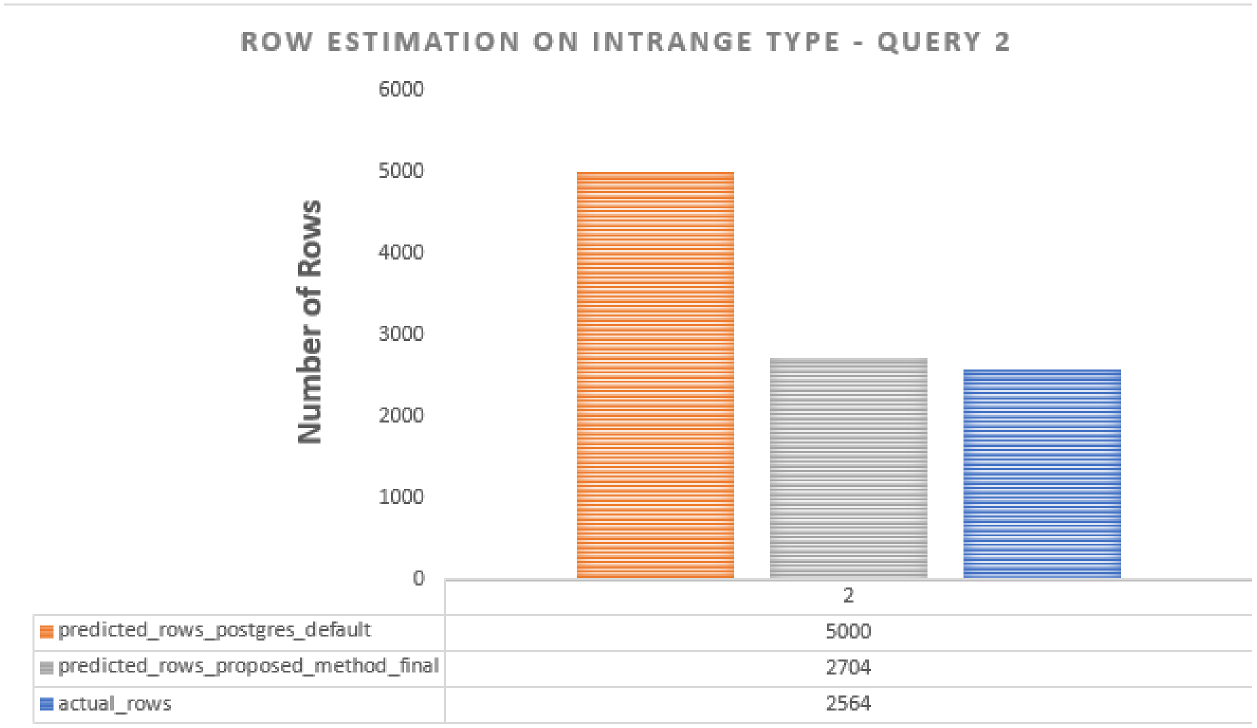
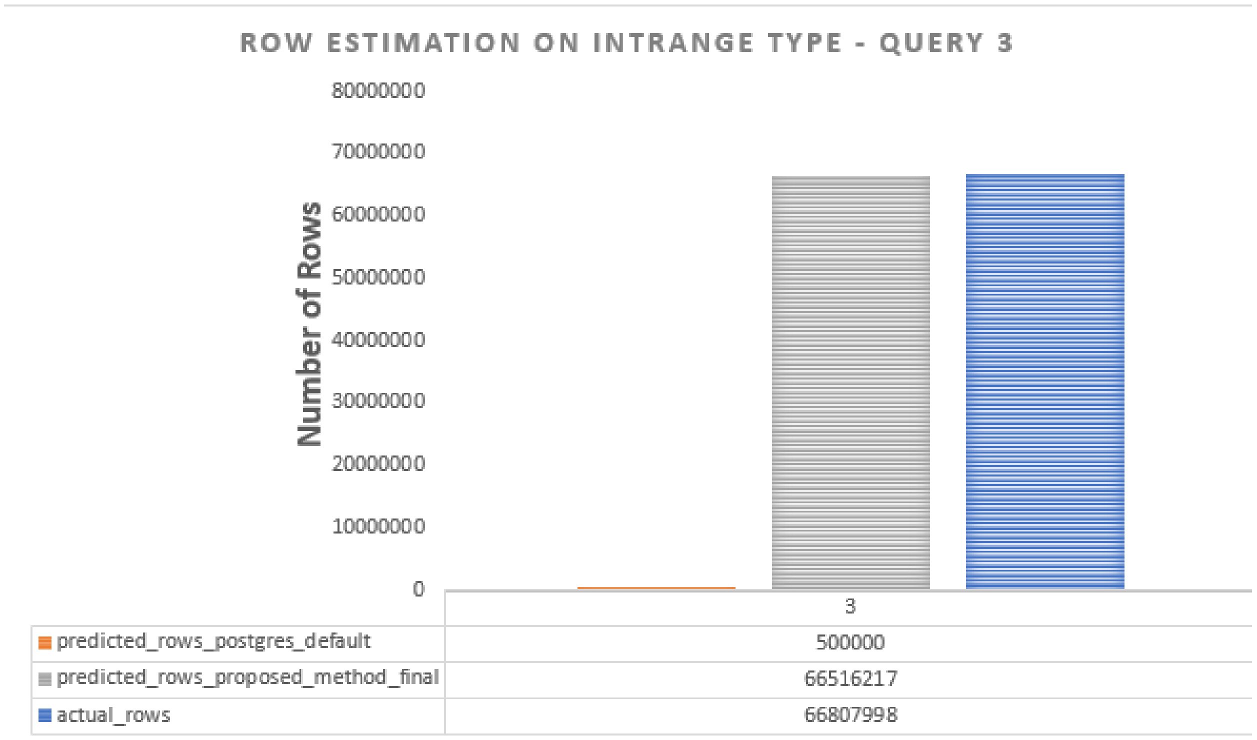


Fig 5.1 Row Estimation on IntRange Type - Query 1



	2
🟧 predicted_rows_postgres_default	5000
▨ predicted_rows_proposed_method_final	2704
🟦 actual_rows	2564

Fig 5.2 Row Estimation on IntRange Type - Query 2



	3
🟧 predicted_rows_postgres_default	500000
▨ predicted_rows_proposed_method_final	66516217
🟦 actual_rows	66807998

Fig 5.3 Row Estimation on IntRange Type - Query 3

3. Query Time analysis

Queries	Original method (planning time, Postgres)	Proposed method planning time with v.1	Proposed method planning time v.2	Query time Postgres	Query time proposed method
1	0.052	0.083	0.083	1.446	1.486
2	0.029	0.058	0.062	85.202	83.696
3	0.043	0.045	0.052	11377.364	11502.662

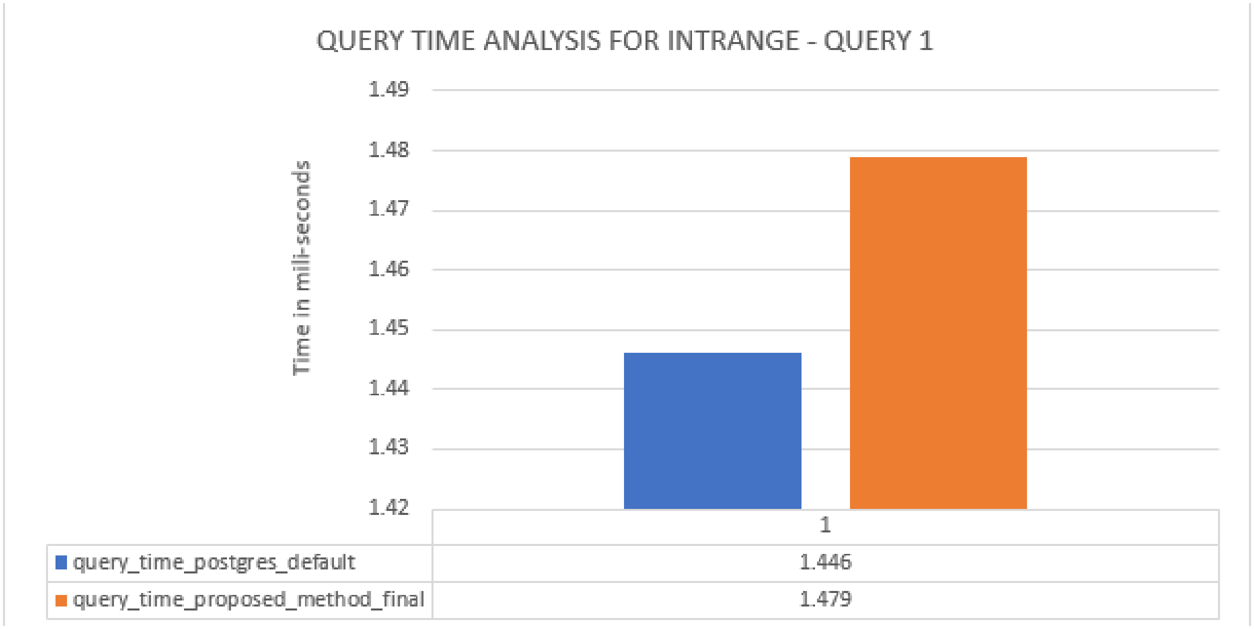
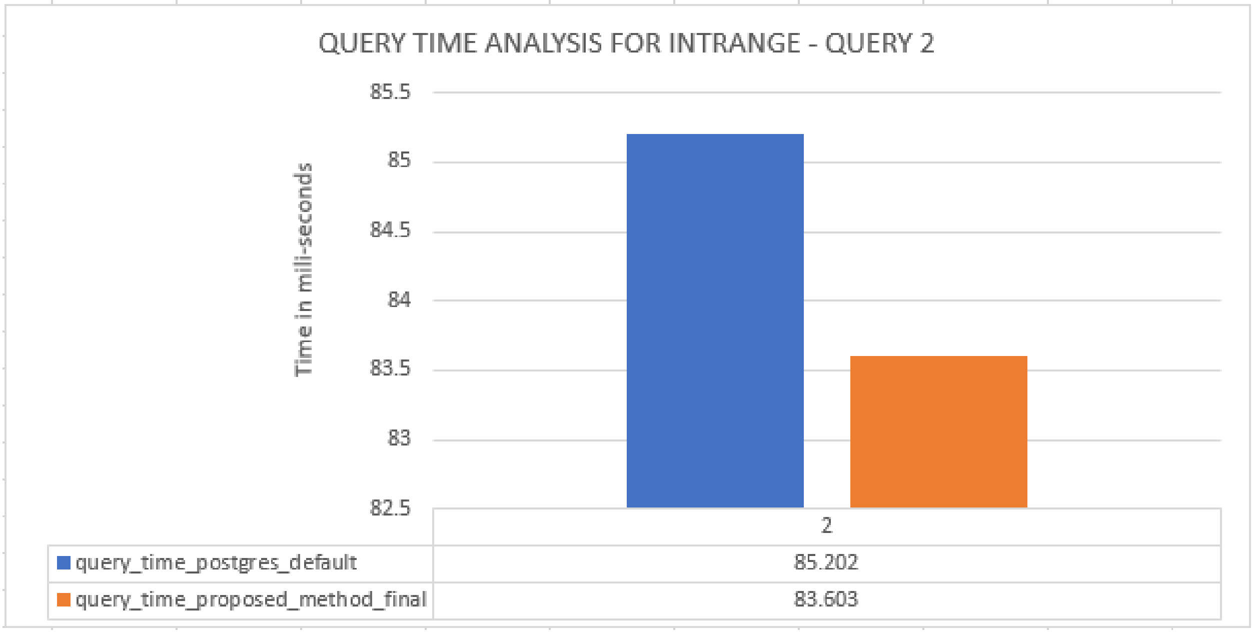
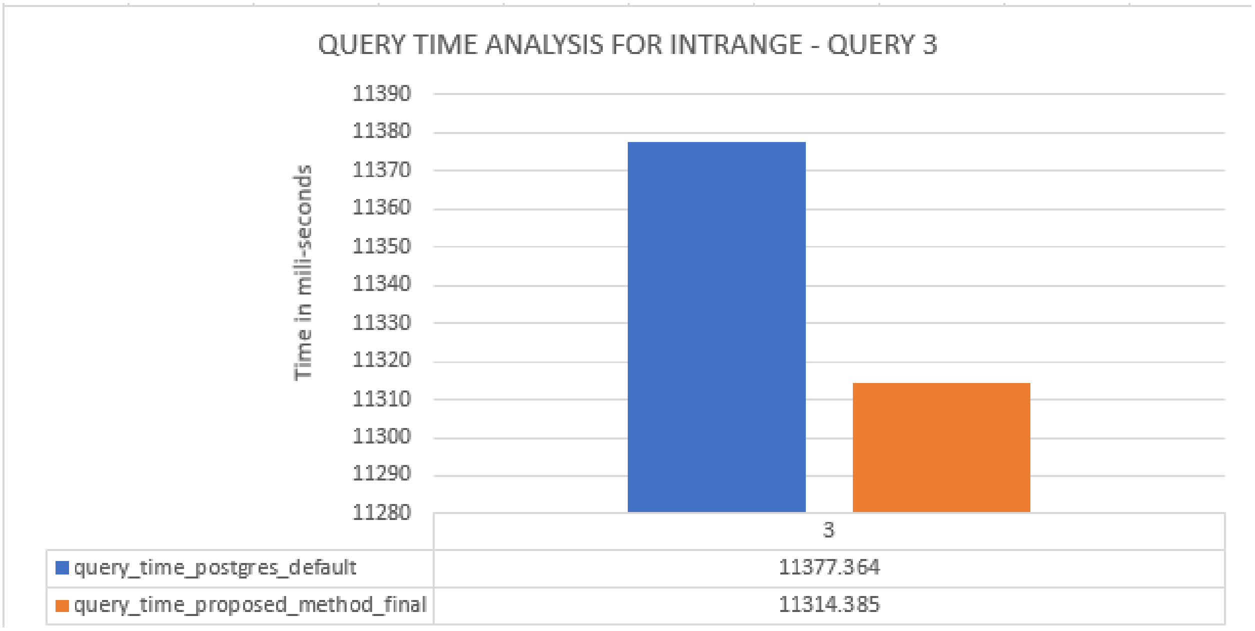


Fig 5.4 Query Time Analysis For IntRange Type - Query 1



	2
🟦 query_time_postgres_default	85.202
🟧 query_time_proposed_method_final	83.603

Fig 5.5 Query Time Analysis For IntRange Type - Query 2



	3
🟦 query_time_postgres_default	11377.364
🟧 query_time_proposed_method_final	11314.385

Fig 5.6 Query Time Analysis For IntRange Type - Query 3

5.2 FloatRange

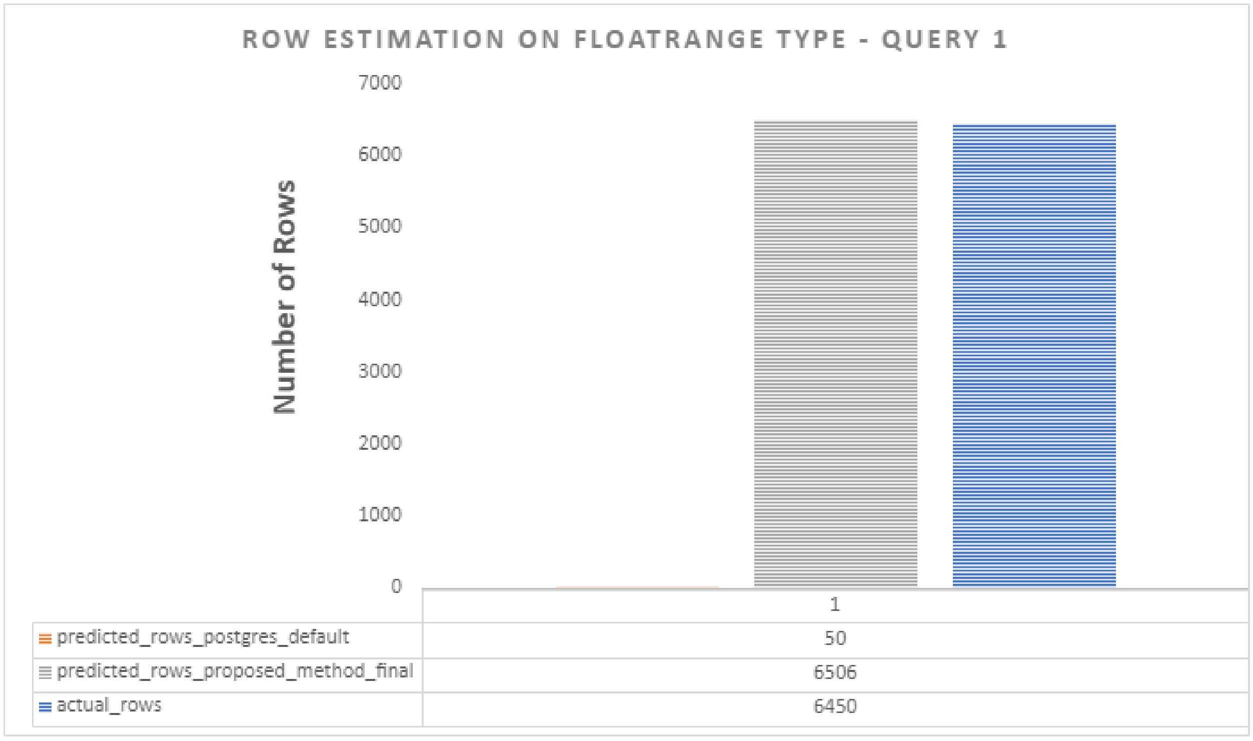
Float range contains the range of numeric values (A discrete data type)

1. Vacuum Time Analysis

Original Code (POSTGRES)	PROPOSED METHOD
12.451	12.051

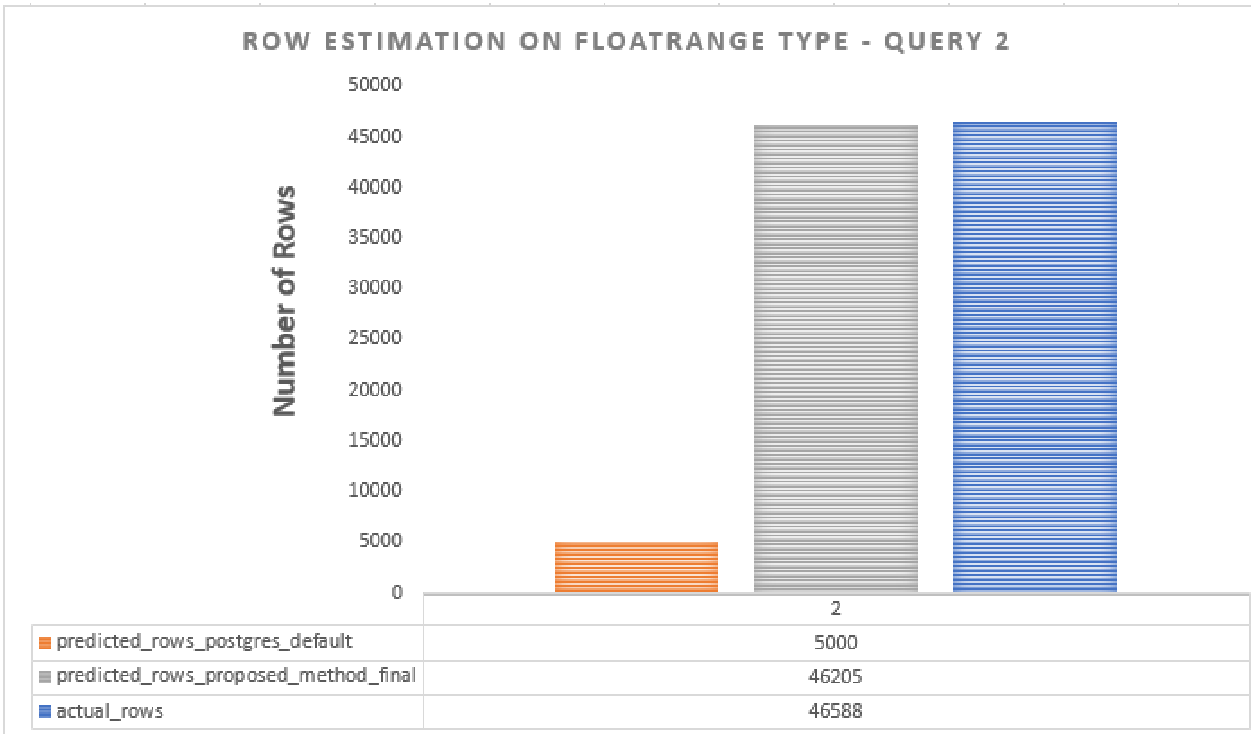
2. Row Estimations Analysis

Queries	Actual rows in the table	Original Code (Postgres) row estimation	Proposed Method row prediction with v.1	Proposed Method row prediction v.2	Error from original method (Postgres)	Error from proposed method v.1	Error from proposed method v.2	Method Accuracy
1	6450	50	6506	6506	0.9922	0.0086	0.0086	0.9913
2	46588	5000	46205	46205	0.8926	0.0082	0.0082	0.9917
3	66817044	500000	66021021	66021021	0.9925	0.0119	0.0119	0.9880



	1
🟧 predicted_rows_postgres_default	50
⬜ predicted_rows_proposed_method_final	6506
🟦 actual_rows	6450

Fig 5.7 Row Estimation on FloatRange Type - Query 1



	2
🟧 predicted_rows_postgres_default	5000
⬜ predicted_rows_proposed_method_final	46205
🟦 actual_rows	46588

Fig 5.8 Row Estimation on FloatRange Type - Query 2

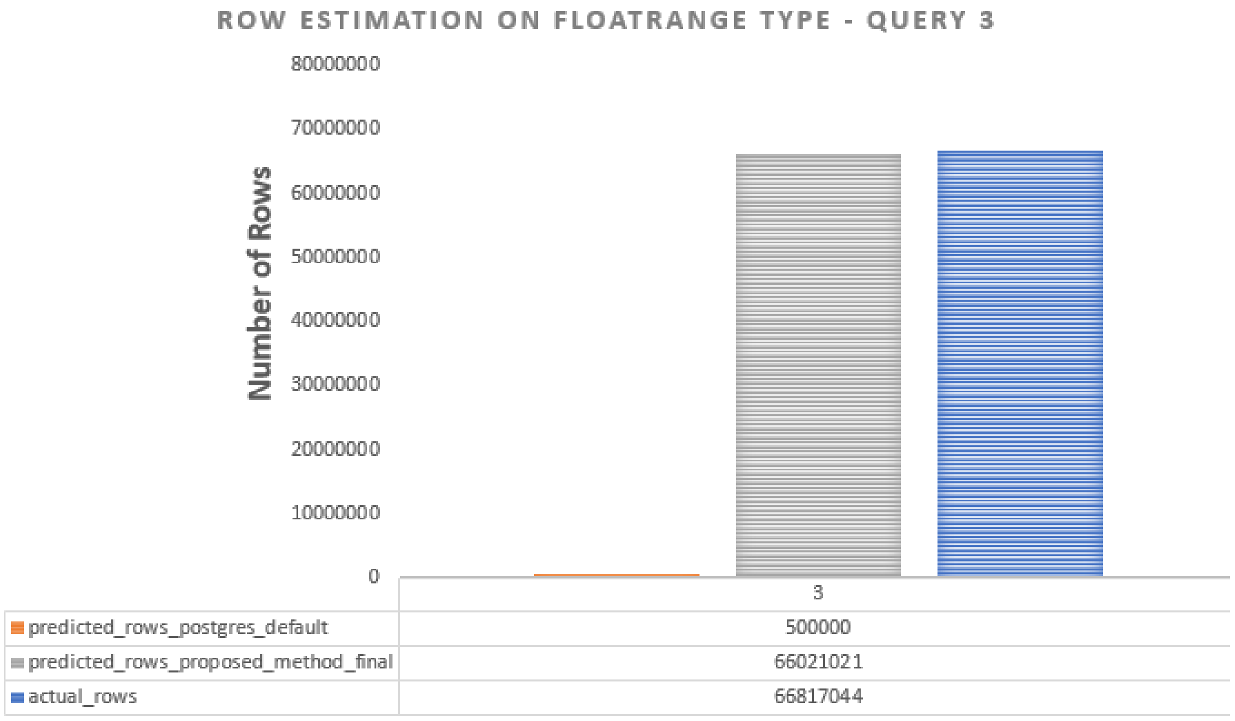


Fig 5.9 Row Estimation on FloatRange Type - Query 3

3. Query Time analysis

Queries	Original method (planning time, postgres)	Proposed method planning time with v.1	Proposed method planning time v.2	Query time postgres	Query time proposed method
1	0.062	0.155	0.159	1.915	2.039
2	0.037	0.13	0.137	154.144	154.64
3	0.044	0.134	0.141	17865.989	18140.431

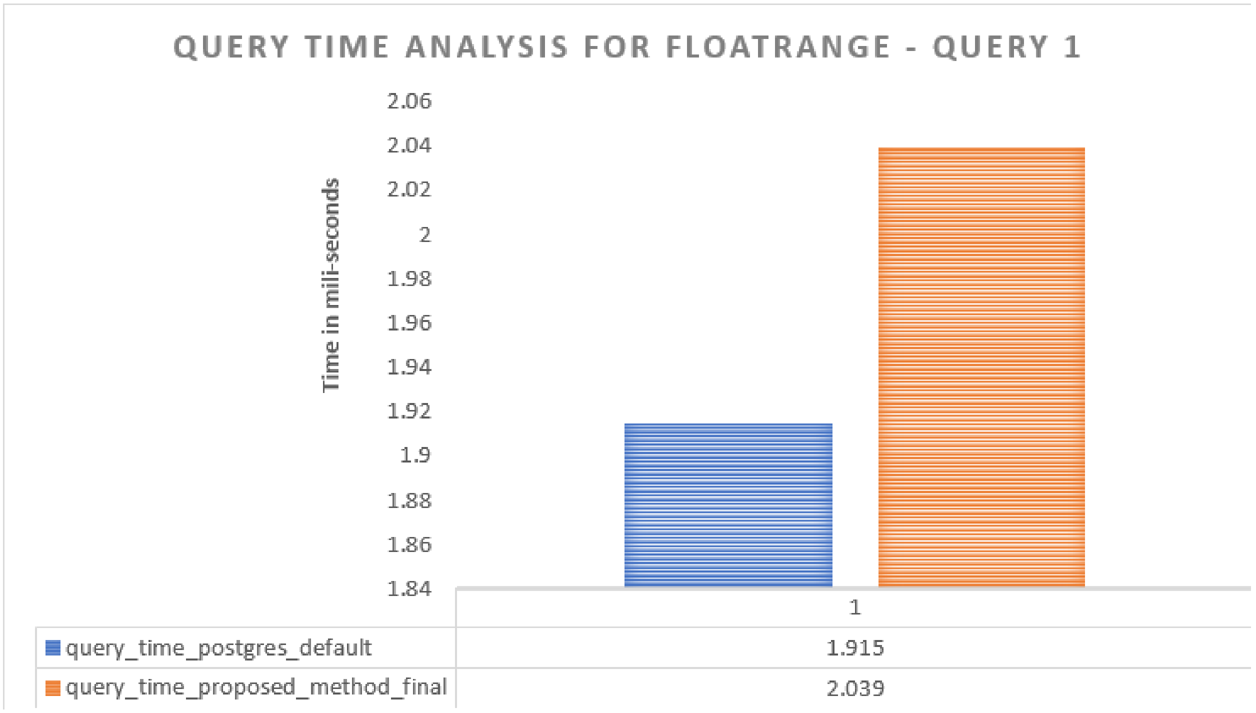


Fig 5.10 Query Time Analysis For FloatRange Type - Query 1

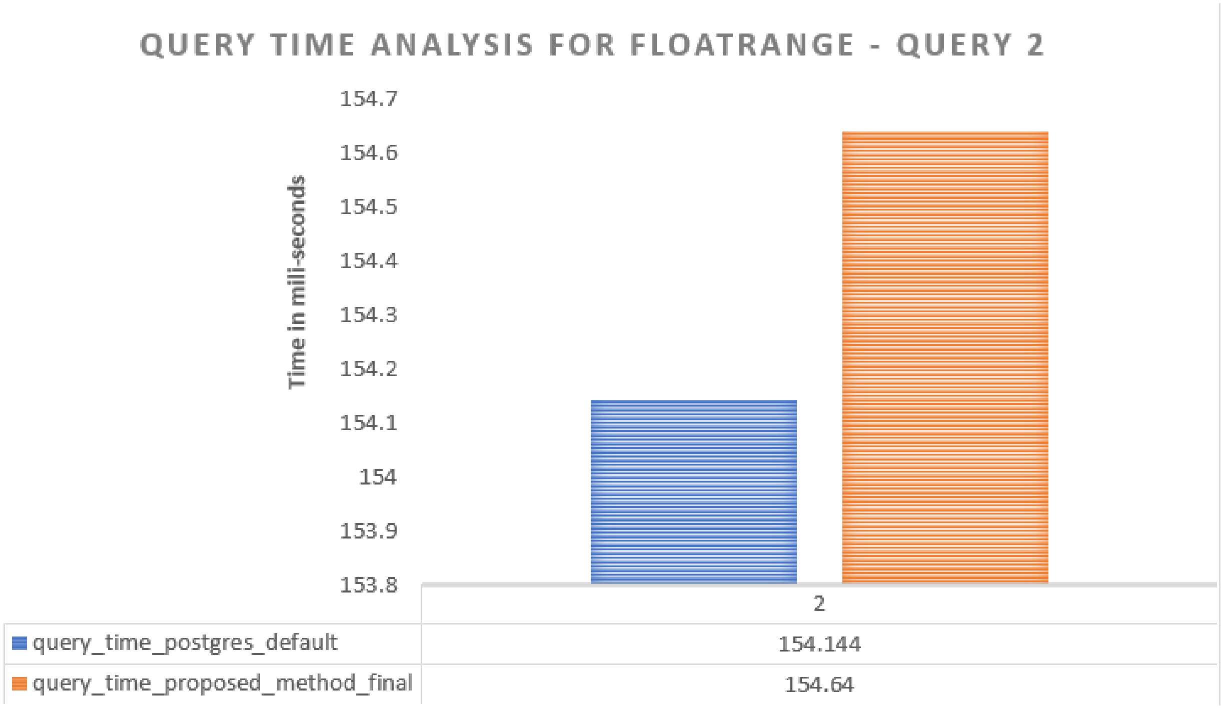


Fig 5.11 Query Time Analysis For FloatRange Type - Query 2

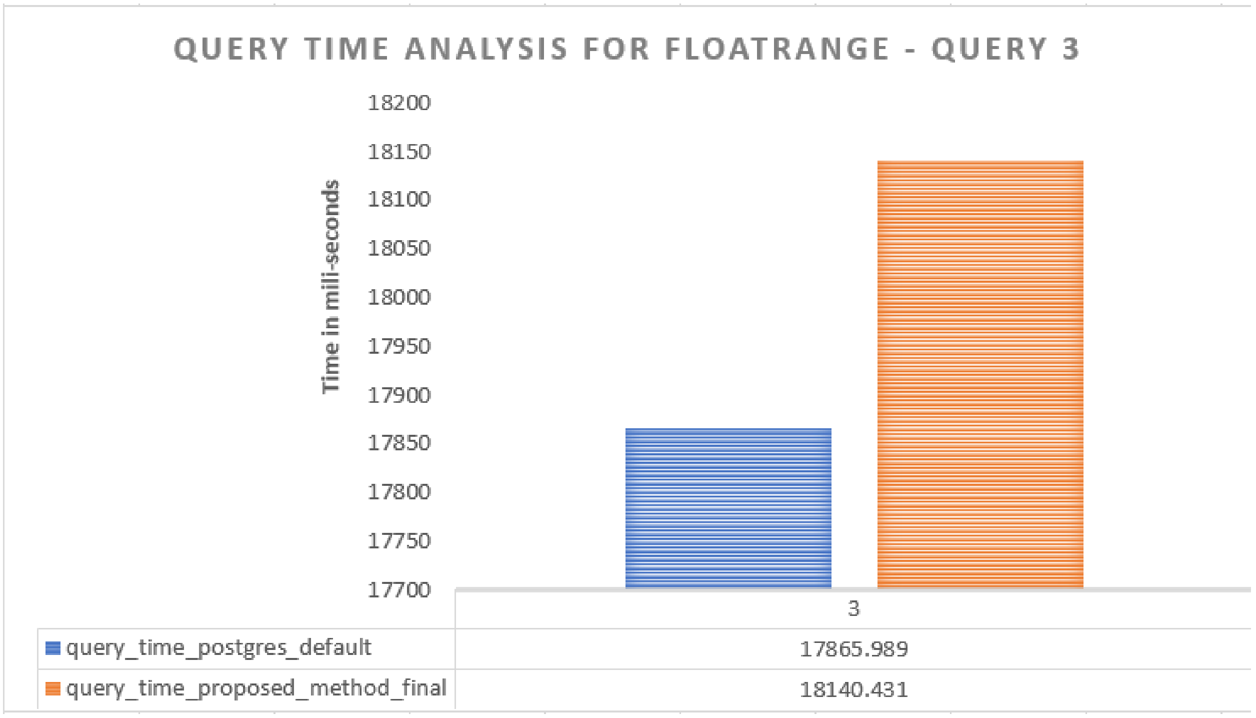


Fig 5.12 Query Time Analysis For FloatRange Type - Query 3

5.3 TsRange

TsRange contains the range of timestamp without time zone (A continuous Data Type)

1. Vacuum Time Analysis

Original Code (POSTGRES)	**PROPOSED METHOD**
11.999	11.943

2. Row Estimations Analysis (Benchmarking)

Queries	**Actual rows in the table**	**Original Code (Postgres) row estimation**	**Proposed Method row prediction with v.1**	**Proposed Method row prediction v.2**	**Error from original method (Postgres)**	**Error from proposed method v.1**	**Error from proposed method v.2**	**Method Accuracy**
1	6021	50	6075	6075	0.9916	0.0088	0.0088	0.9911
2	433487	5000	432925	432925	0.9884	0.0012	0.0012	0.9987
3	62107563	500000	61328338	61328338	0.9919	0.01254	0.01254	0.9874

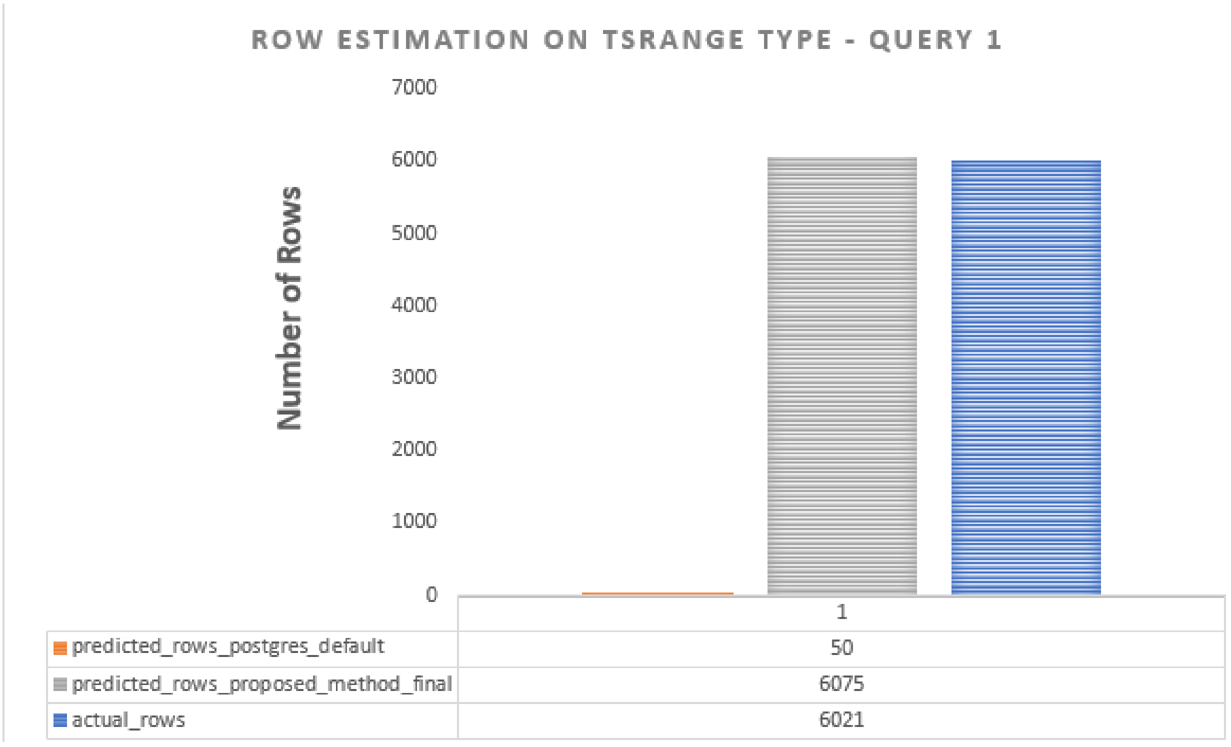
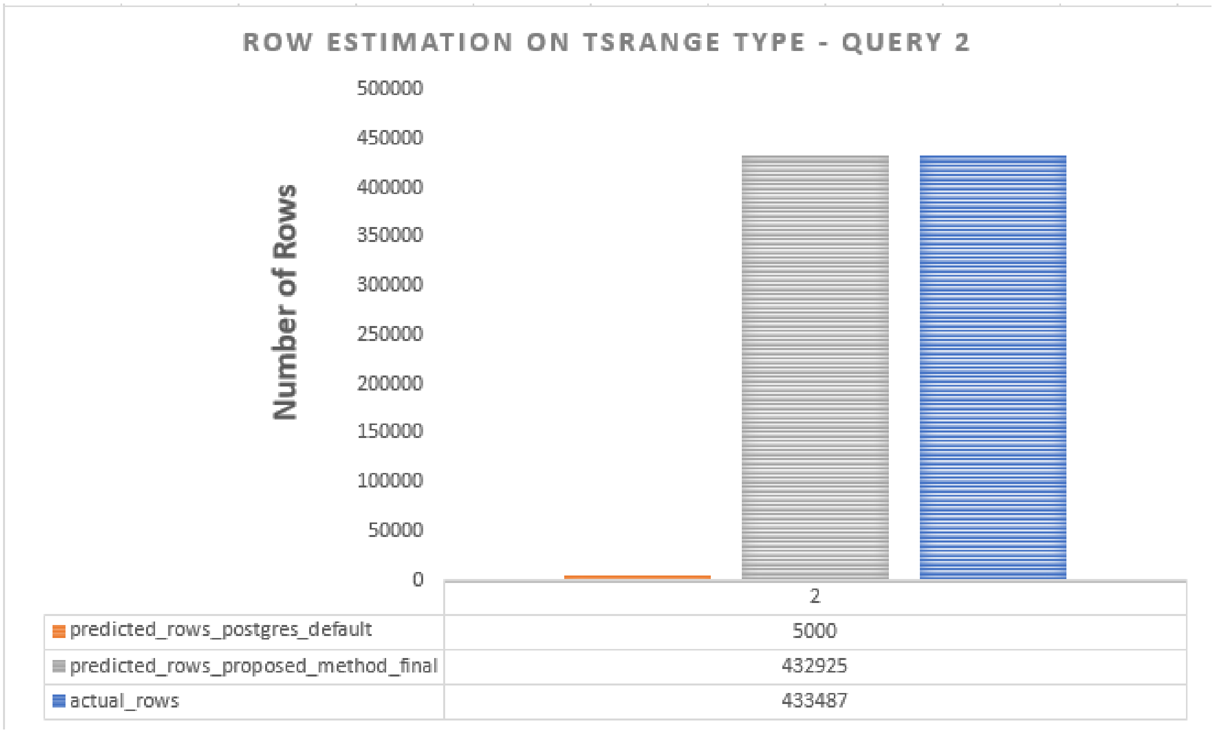
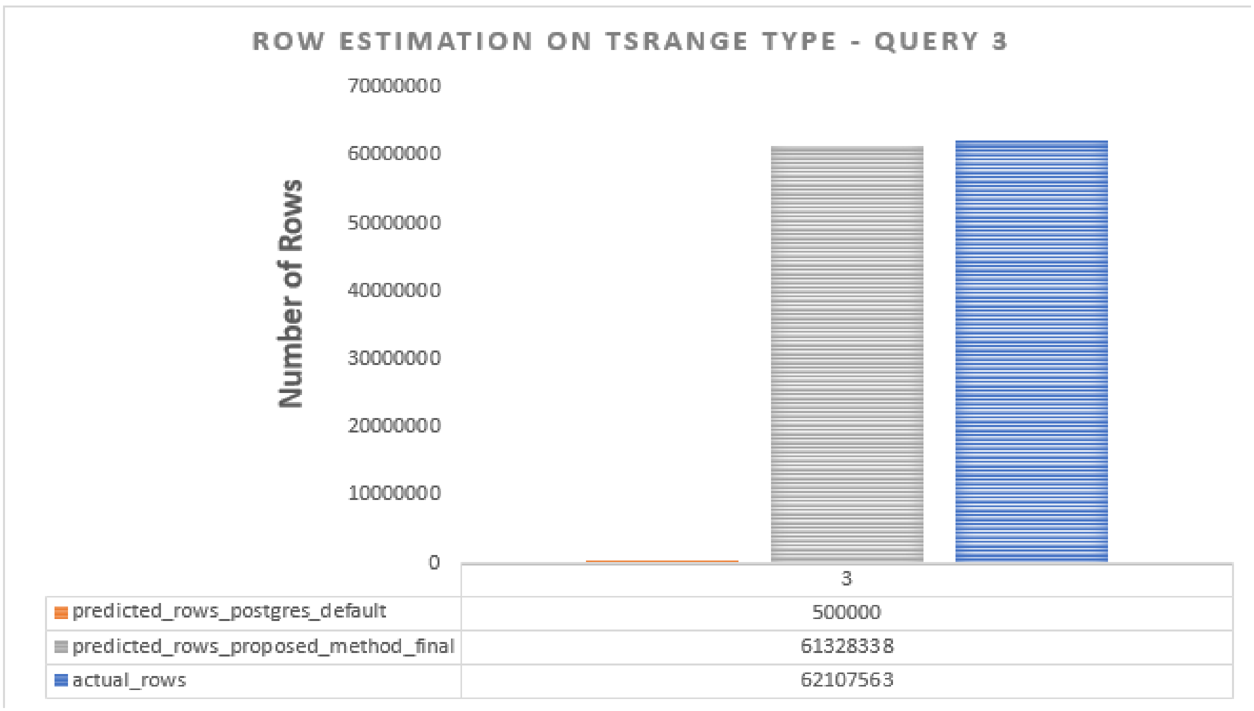


Fig 5.13 Row Estimation on TsRange Type - Query 1



	2
🟧 predicted_rows_postgres_default	5000
▤ predicted_rows_proposed_method_final	432925
🟦 actual_rows	433487

Fig 5.14 Row Estimation on TsRange Type - Query 2



	3
🟧 predicted_rows_postgres_default	500000
▤ predicted_rows_proposed_method_final	61328338
🟦 actual_rows	62107563

Fig 5.15 Row Estimation on TsRange Type - Query 3

3. Query Time analysis

Queries	Original method (planning time, postgres)	Proposed method planning time with v.1	Proposed method planning time v.2	Query time postgres	Query time proposed method
1	0.052	0.083	0.083	1.446	1.486
2	0.029	0.058	0.062	85.202	83.696
3	0.043	0.045	0.052	11377.364	11502.662

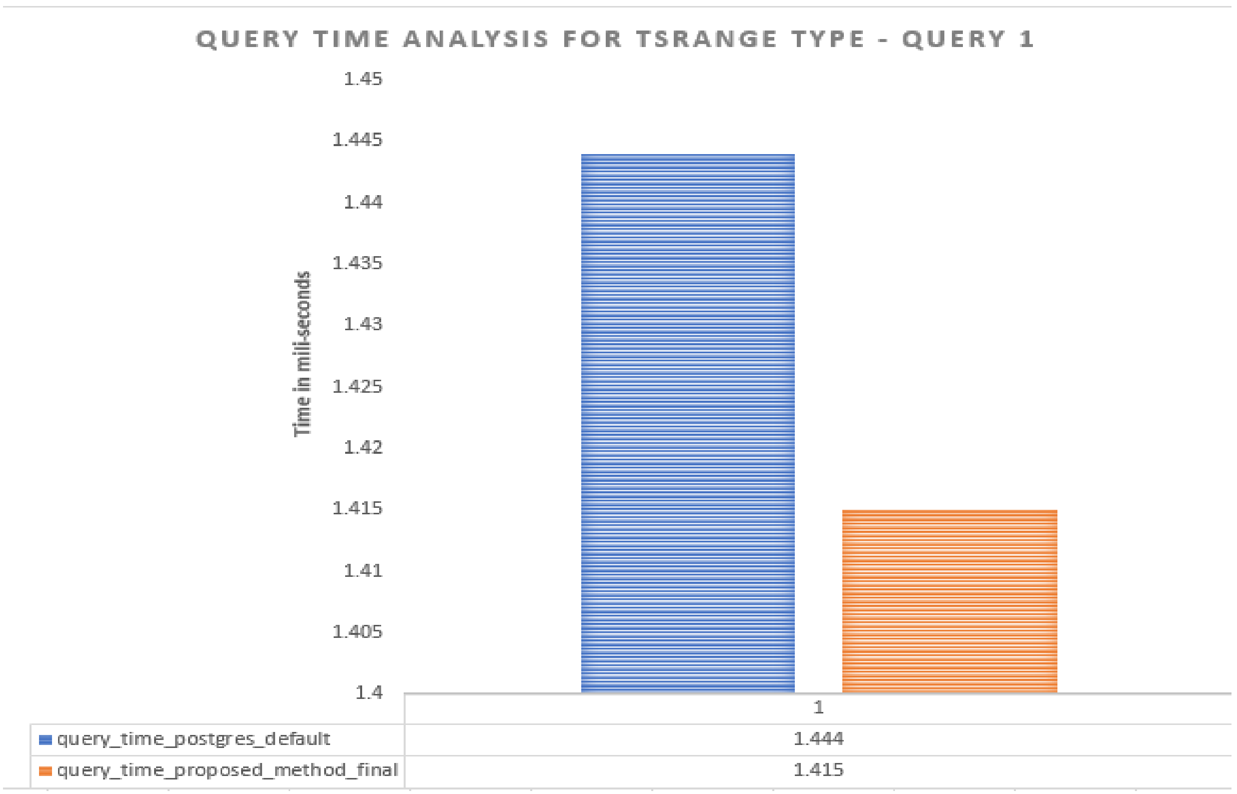


Fig 5.16 Query Time Analysis For TsRange Type - Query 1

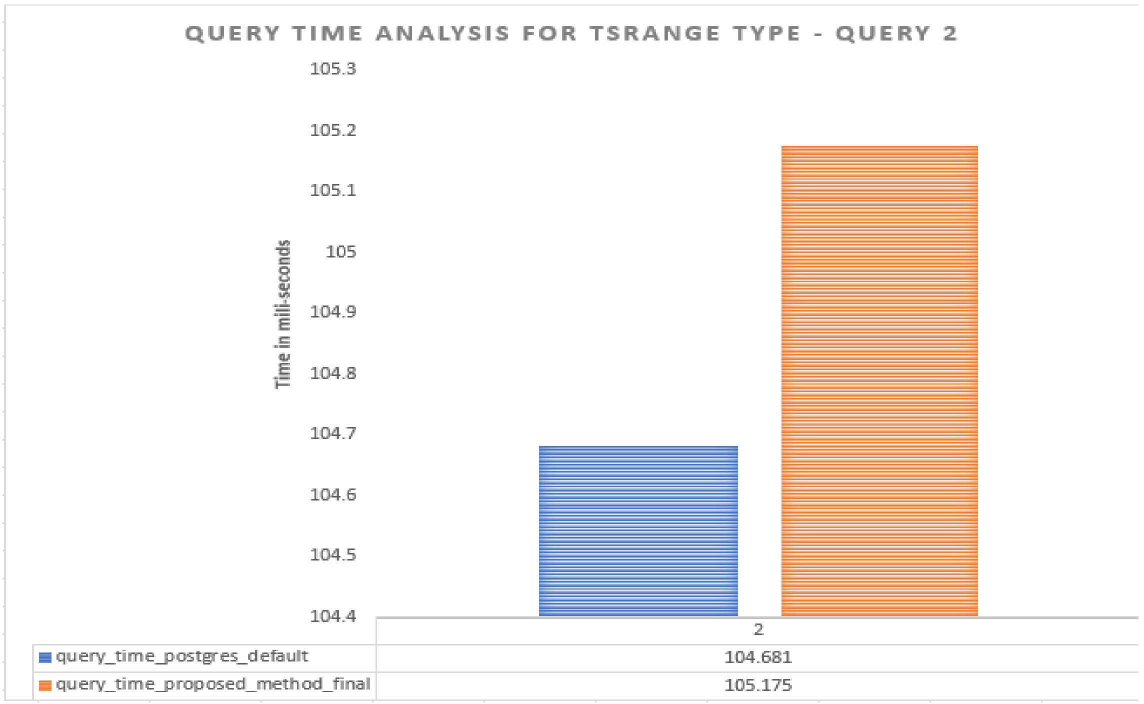


Fig 5.17 Query Time Analysis For TsRange Type - Query 2

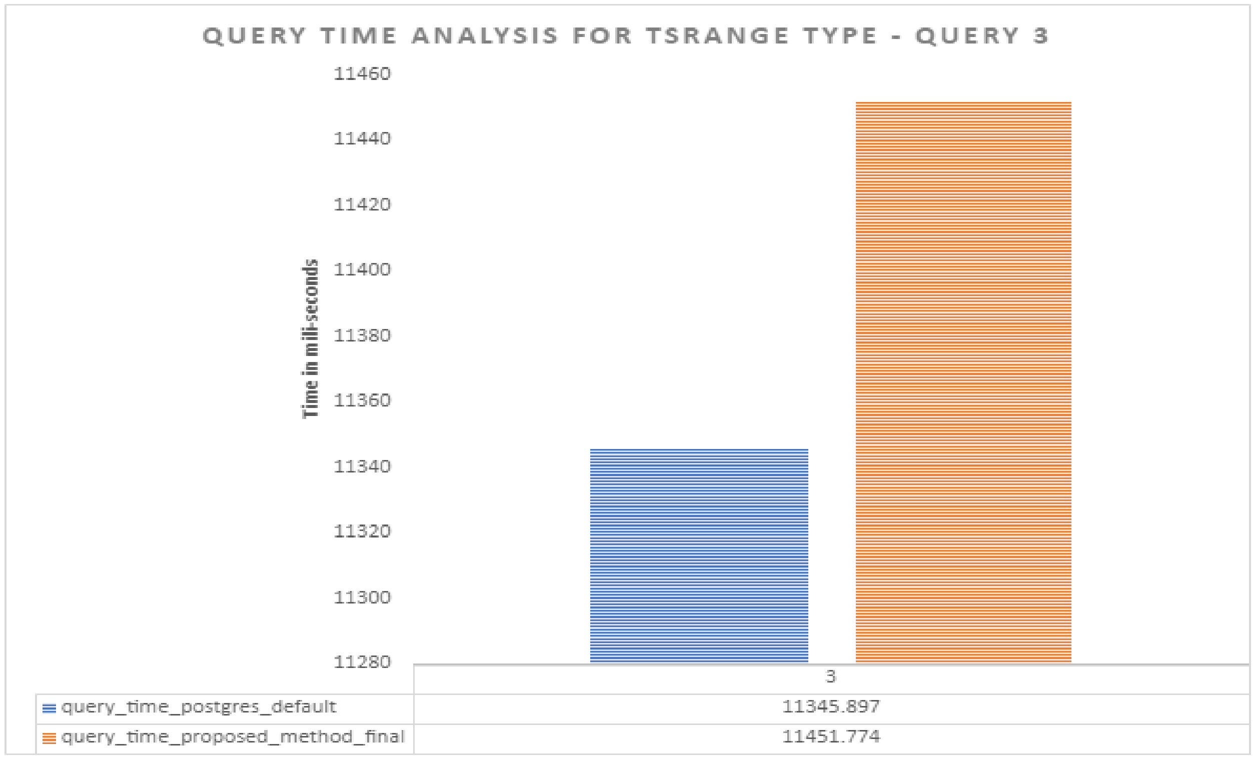


Fig 5.18 Query Time Analysis For TsRange Type - Query 3

5.4 DateRange

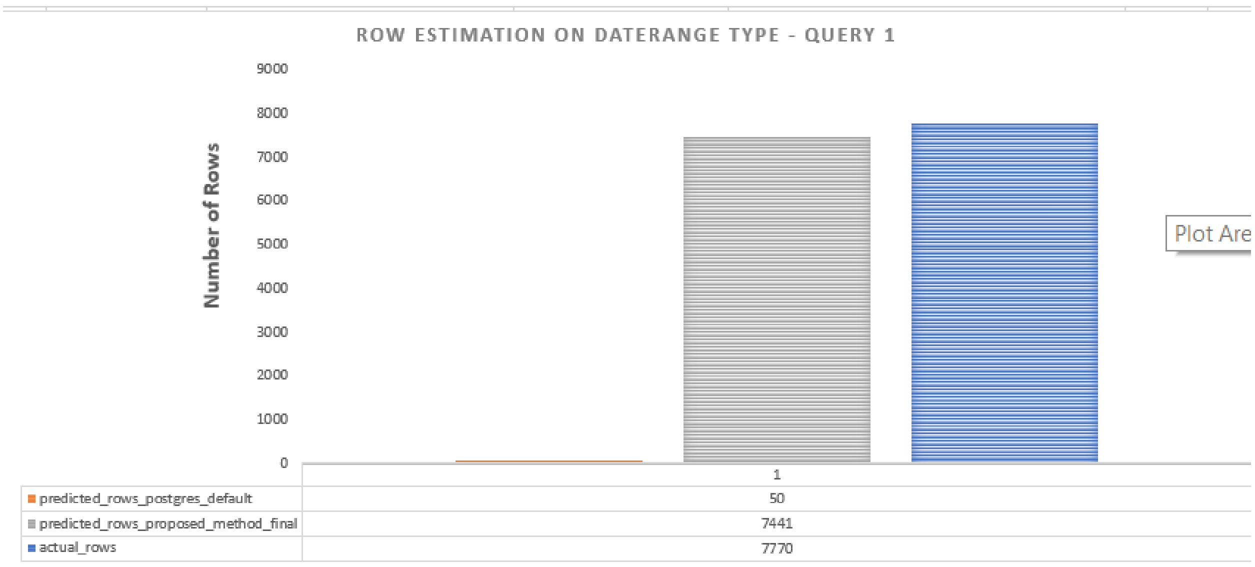
DateRange contains the element type (subtype) Dates (A Discrete Data Type)

1. Vacuum Time Analysis

Original Code (POSTGRES)	**PROPOSED METHOD**
12.951	14.464

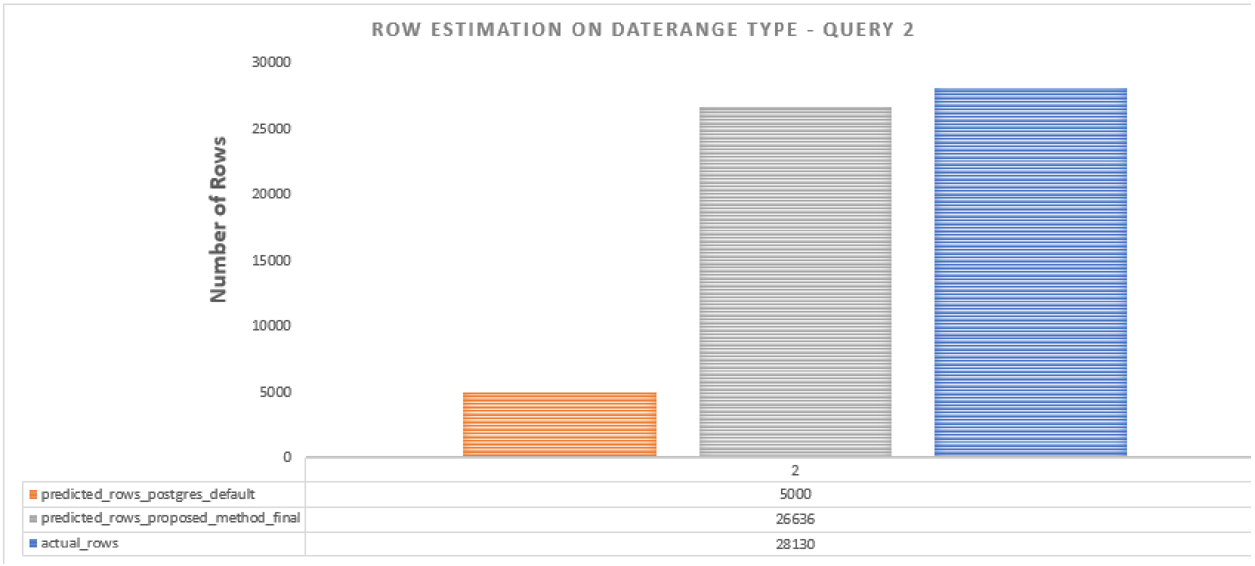
2. Row Estimations Analysis

Queries	**Actual rows in the table**	**Original Code (Postgres) row estimation**	**Proposed Method row prediction with v.1**	**Proposed Method row prediction v.2**	**Error from original method (Postgres)**	**Error from proposed method with dup**	**Error from proposed method final**	**Method Accuracy**
1	7770	50	7828	7441	0.9935	0.0074	0.0423	0.9576
2	28130	5000	28371	26636	0.8222	0.0084	0.0531	0.94688
3	77993373	500000	77439134	77439134	0.9935	0.0071	0.0071	0.9928



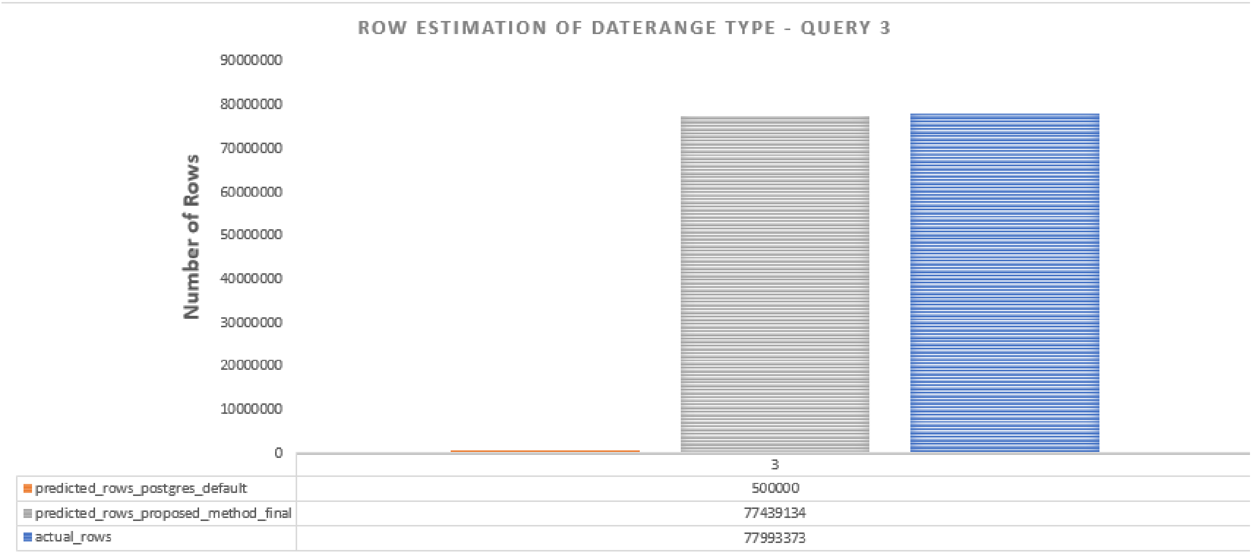
	1
■ predicted_rows_postgres_default	50
■ predicted_rows_proposed_method_final	7441
■ actual_rows	7770

Fig 5.19 Row Estimation on DateRange Type - Query 1



	2
■ predicted_rows_postgres_default	5000
■ predicted_rows_proposed_method_final	26636
■ actual_rows	28130

Fig 5.20 Row Estimation on DateRange Type - Query 2



	3
🟧 predicted_rows_postgres_default	500000
▨ predicted_rows_proposed_method_final	77439134
🟦 actual_rows	77993373

Fig 5.21 Row Estimation on DateRange Type - Query 3

3. Query Time analysis

Queries	Original method (planning time, postgres)	Proposed method planning time with v.1	Proposed method planning time v.2	Query time postgres	Query time proposed method
1	0.099	1.057	0.528	2.691	3.996
2	0.05	0.096	0.108	102.915	104.403
3	0.05	0.085	0.088	11506.824	11530.335

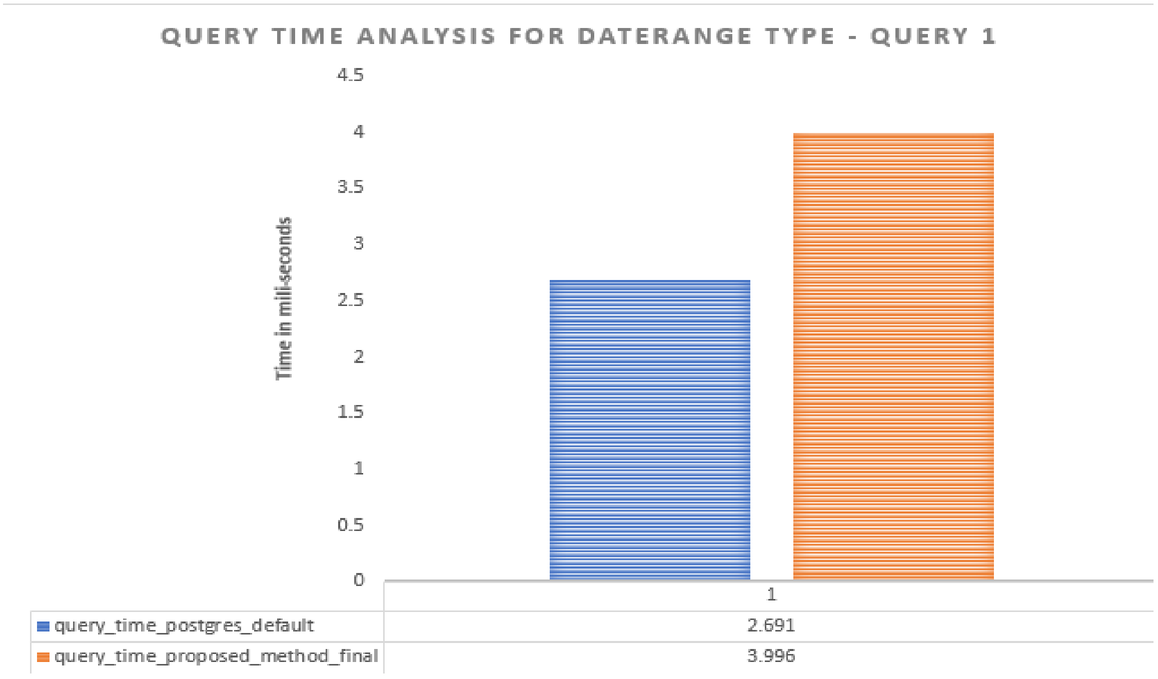


Fig 5.22 Query Time Analysis For DateRange Type - Query 1

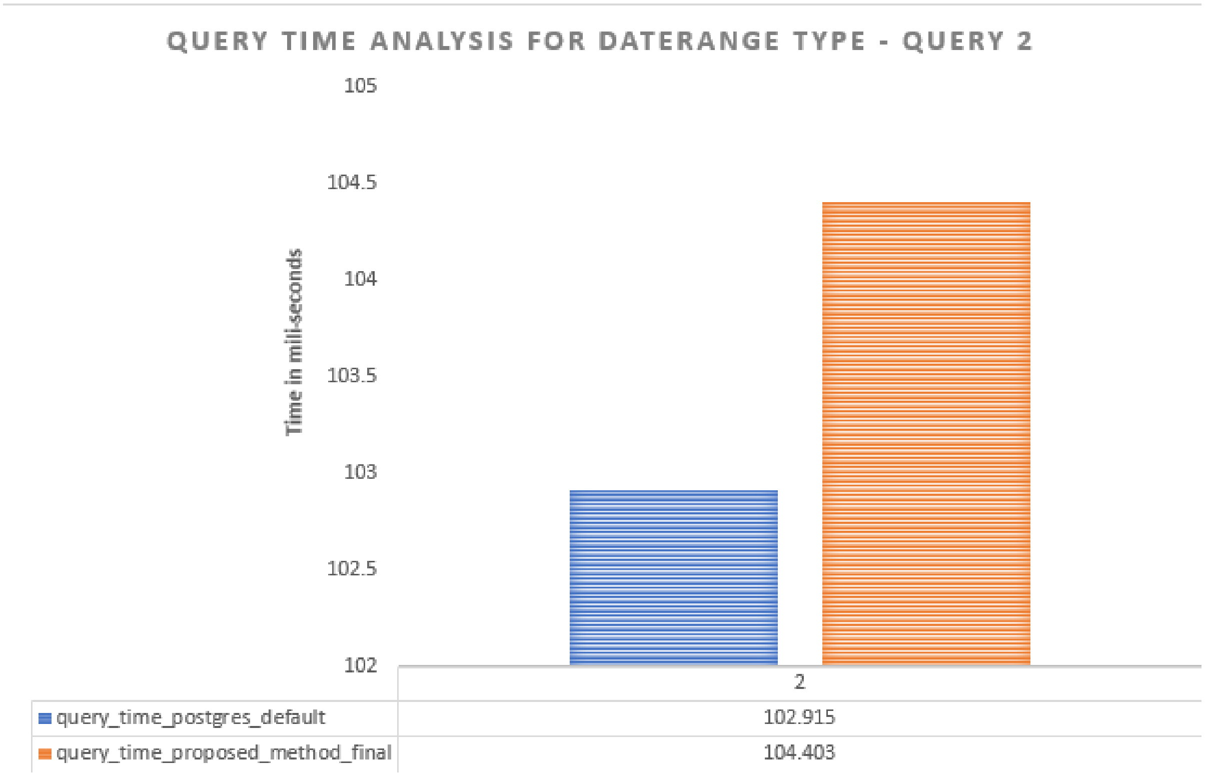
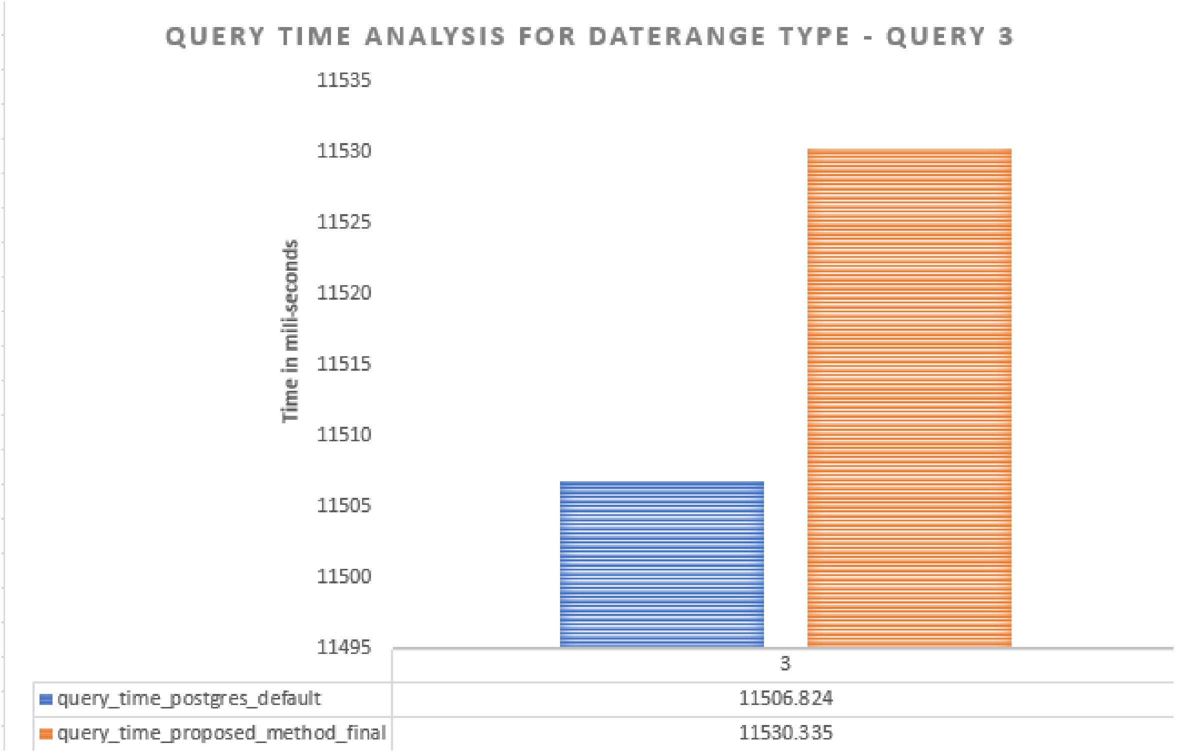


Fig 5.23 Query Time Analysis For DateRange Type - Query 2



	3
🟦 query_time_postgres_default	11506.824
🟧 query_time_proposed_method_final	11530.335

Fig 5.24 Query Time Analysis For DateRange Type - Query 3

Conclusion

To conclude, we built an algorithm which can use the existing statistics present in Postgres for range type. The algorithm was optimized in such a way that we got the state of the art results for join-cardinality estimation without impacting the inbuilt selectivity estimation by Postgres. Our rigorous experiments on each range type show significant improvements for the join-cardinality estimation. Since the current implementation is a hardcoded (default) value, it always returns a fixed fraction of the total number of rows in the table. Our solution improves this by analysing the sample ranges and gives an actual approximation of the join-cardinality. The proposed method accuracy reaches up to 99% for join-cardinality estimation without impacting much on query execution time.

We also implemented a new statistic which has the information that where there are more overlaps ranges and where there are less. This new statistic gives an approximation of where there is a higher density of ranges. It should be noted that this statistic is not utilised for the implementation of the join cardinality estimation.

The proposed solution also opens up the gate for the research of cardinalities involving both join and selectivity operations for range type. Irrespective of the previous default returned value, the new algorithm gives correct estimation for join cardinality which helps the query optimizer to choose the most optimal plan to execute the query. So, the proposed solution can be very helpful, not only in join cardinality estimation but also in optimizing all the queries involving selectivity with join overlaps.

References

[1] https://www.postgresql.org/docs/13/using-explain.html

[2] https://github.com/postgres/postgres

[3] https://www.postgresql.org/docs/9.3/rangetypes.html

[4] https://www.postgresql.org/docs/9.3/functions-range.html#RANGE-FUNCTIONS-TABLE

GitHub Repository Link :
https://github.com/mohammadzainabbas/database-system-architecture-project