Calculating Spatial Distance Histograms For 3D Points

Luis Quezada
University of South Florida, Tampa
luis4@usf.edu

Mushfiq Mahmud
University of South Florida, Tampa
mushfiq@usf.edu

Tommy Truong
University of South Florida, Tampa
ttruong@usf.edu

***Abstract*—Complex aggregates are functions offered by database management systems to do rigorous calculations without redundancy in code. Database management systems such as PostgreSQL offers users the ability to implement and integrate additional complex aggregates. The project goal was to compute the spatial histogram in 3D space. We used 3D spatial distance histogram implementation and generated a quadtree index of the points. Result of the SDH query was stored into a table.**

I. Introduction

Aggregation is the formation of a number of things into a cluster such as adding all values of a column to find the sum. Database management systems (DBMS) such as PostgreSQL supports such complex aggregates to help the user perform complex analytical task with their SQL server. PostgreSQL currently supports various aggregates such as SUM, MAX, MIN, AVG, and COUNT. If a aggregate is not supported, PostgreSQL provides the ability for users to implement their own aggregate function in a compatible language such as C and integrate it into the codebase.

A. Problem Statement

A spatial distance histogram (SDH) is the histogram of distances between all pairs of particles in the system [2]. While the current codebase handles SDH queries for a set of 2D points, our task is to extend the complex aggregates in another language to handle a set of 3D points and integrate it into the codebase.

II. Methodology

In this section, we will discuss the algorithm behind SDH, how to integrate with PostgreSQL and how quadtrees can be used for indexing.

A. Spatial Distance Histogram Calculation

For our project, we went with the brute force method [5],[1] as shown in Algorithm 1.

Algorithm 1: Spatial Distance Histogram

```
Input: Set of 3D points
Output: Set of distances of 3D points between
        buckets in a histogram
1  arr = Set of 3D points
2  bucketWidth = width of each histogram bucket
3  histogram = list of buckets
4  for $i \in numSamples$ do
5      for $j \in i + 1$ do
6          $x_1, y_1, z_1 = arr[i]$
7          $x_2, y_2, z_2 = arr[j]$
8          dist = euclidean distance between
             $(x_1, y_1, z_1)\ and\ (x_2, y_2, z_2)$
9          $h_pos = \lfloor dist/bucketWidth \rfloor$
10         histogram[$h_pos$].distCount++
11     end for
12 end for
13 return histogram
```

B. PostgreSQL Integration

PostgreSQL allows users to extend functionality by creating custom attributes to complement the database server such as custom types, aggregate functions and general purpose functions. Moreover, while the PostgreSQL source code is written in the C language, we have the freedom of using any external language that can be integrated into the source code including the procedural language provided by PostgreSQL themselves called `plpgsql`. We have made use of these to create external functions in both `C` as well as `plpgsql`.

At a high level, the custom source code is dynamically loaded via shared library object files which are OS specific. As long as the absolute path to these `.so` files are provided or they are available in the dynamic library path, these files can be referenced inside a database session in PostgreSQL using SQL functions. PostgreSQL provides some much needed abstractions in the form of C macros to aid in retrieving user data such as arguments to a function or data inside of SQL tables.

We start by creating a table to store all our 3D points. The table has 3 columns to store each of our 3 dimensions. We can then insert our points into this table one at a time to store the dimensional data.

In order to generate the SDH from our points, we need to be able to perform a nested loop over the entire table to be able to calculate the point-to-point distance of each point to every other point.

In order to generate the SDH from our points, we need to be able to perform a nested loop over the entire table to be able to calculate the point-to-point distance of each point to every other point.

Our inner loop is implemented using the function `one_to_all_following`. This loop needs to be able to traverse a point and calculate the distance from it to all other points that come after it. It is important to only calculate distance to points *after* the given point because otherwise we would be calculating redundant distances which points previous to the given point might have already calculated. As long as we follow a specific order, we should be able to just get by using calculations done between a certain point and all points after it giving us an $n!$ runtime for calculating distance where $n-1$ is the number of input points since the first point will have to calculate $n-1$ distances to each of the $n-1$ points after it, the second will have to calculate $n-2$ distances and so on.

Our inner loop is designed by using a sql query execution. We make use of a PostgreSQL window function [4] called `nth_value` which takes in a column name and a row number. The column name will be traversed and every row value will be compared with the value at the row number given for that column. This way every row can be compared with a specific row value. This is exactly what we need. One important thing to conssider is that window functions need an order to be supplied using the `OVER` clause which makes sure the row number supplied always points to the same row since the ordering is specified. For our purposes, we use `ORDER BY x, y, z` to keep things consistent. And lastly, we also perform an `OFFSET` to denote that we only want values after the current nth row.

Overall, our query returns a table of point-to-point distances of the given row number to every other points that come after it (in the previously mentioned order).

Our inner loop function also takes a bukcet width which is used to calculate the SDH indices by using the calculation

```
index = floor( dist / bucketWidth)
```

This will return a table of indices for the histogram which we can use to build the histogram.

The outer loop can now perform the inner loop over all rows in the table. Our outer loop is designed as a custom aggregate called `cagg` which gets passed around to each row of any input relation.

It does this by taking in a state transition function [4], `stfunc` in our case, to determine what values should be passed through to the next row and what to do with these values every iteration.

We create a custom type to hold our intermediary state values, aptly named `state` in our case. This has a numeric array `res[]` which is used to hold all the values returned and an integer `n` to store the count of points calculated thus far. When using the aggregate we provide the initial conditions by using `initcond` [4] property and initialize an empty array and the count to `0`. At every iteration, we call the inner loop function which returns a table of histogram indices. We convert this table into an array and append all the values into `res` and pass `res` over to the next iteration. We also increment the value of `n`.

This aggregate also optionally takes in a final function [4], `ffunc` in our case, which is often used to dress the output since we are passing and returning multiple parameters that we might not necessarily want to return at the very end. We use this function to return `res` at the very end of our aggregate to output the final array of histogram indices.

At this point, we have a numeric array of histogram indices that we can use to calculate the histogram. We do this by one final custom function `calc_histogram`.

This function takes in takes in the numeric array and does a loop over each of these indices. A temporary array `s` is declared at the start of the function and initialized to all zeroes. Whenever an index value is encountered, the corresponding index of `s` is incremented. The return value of this function is the final SDH which holds the information of how many distances fall into which bucket of the histogram according to our algorithm (Algorithm 1).

C. *QuadTrees/Octrees*

QuadTree is a tree like data structure which has exactly four children. For our purposes QuadTree is used to store 2D points that can be traversed efficiently in logarithmic time [3]. QuadTrees are commonly used to index geographical data by partitioning two-dimensional space into four quadrants and recursively subdividing it until all points are mapped. Octrees are an extension of QuadTrees, Octrees have eight children and are often used to partition three-dimensional spaces and store 3D points, such as images. For our uses, since we are just calculating the distance of a spatial histogram, a QuadTree is more than enough to store the data.

III. Results

For the calculation of the SDH query, we were able to create a list of 3d points and stream them into the table such as shown on Fig. 1.

Once the 3d point table was populated, it was passed into the SDH custom aggregate that goes row by row to calculate the point-to-point distances as shown on Fig. 3 and updated the histogram distribution count as per Algorithm 1. The resulting SDH is shown on Fig. 2.

As we can see, the SDH table provides an insight into the distribution of point to point distances between a set of 3d points. The distribution appear to be a normal distribution as most of the distances are localized in the middle buckets of the histogram with the distribution tapering off at the extremes.

IV. Future Work

Since the scope of the project was small, we had to implement the ideas that were feasible while discarding the rest for a later time. Some of the future work that we could do is has the 3D point accept any datatype, on top that we would also like to be able to pass in a whole set of 3D points at once rather than one at a time. From the SDH standpoint, another future work is to implement the faster method [2] of SDH instead of the brute force method. The next improvement we could do is do a full scale implementation of Quadtree and octree to utilize the spatial distance between points close in proximity.

In order for our custom data type of 3D points to create a QuadTree index we would need to define a default operator class for our data type to generate our index. By having a QuadTree index our retrieval of point data will be reduced by logN.

While researching the SDH problem, we found an external library call PostGIS which makes solving the SDH problem easier for future integration. PostGIS is a spatial database extender for PostgreSQL object-relational database. It adds support for geographic objects allowing location queries to be run in SQL. What made PostGis useful is that it had an implementation of SDH that was more robust and adaptable to different scenario.

V. Conclusion

Overall this project was nourishing in what it set for us to do and how to accomplish it. In the pursuit of implementing a complex aggregates, we learned the inner workings of PostgreSQL codebase. Since the codebase was highly modularized, it was simple to see how to plug in our complex aggregate implementation. We learned how to dynamically link a function with the source code and we learned a little bit about the complexity of indexing in a DBMS. Some of the problems however were mostly in project requirement. Since the project description wasn't very clear, there was some confusion as to what a spatial distance histogram was since we never learned it in class, how the SDH query gets tested, and what are the expected results. Another issue we had is that the description said there was an initial SDH codebase that handled 2D points but that wasn't provided beforehand nor was it in PostgreSQL codebase so we had to do some investigation to find it. However once those were cleared up, we had no issue doing the project.

References

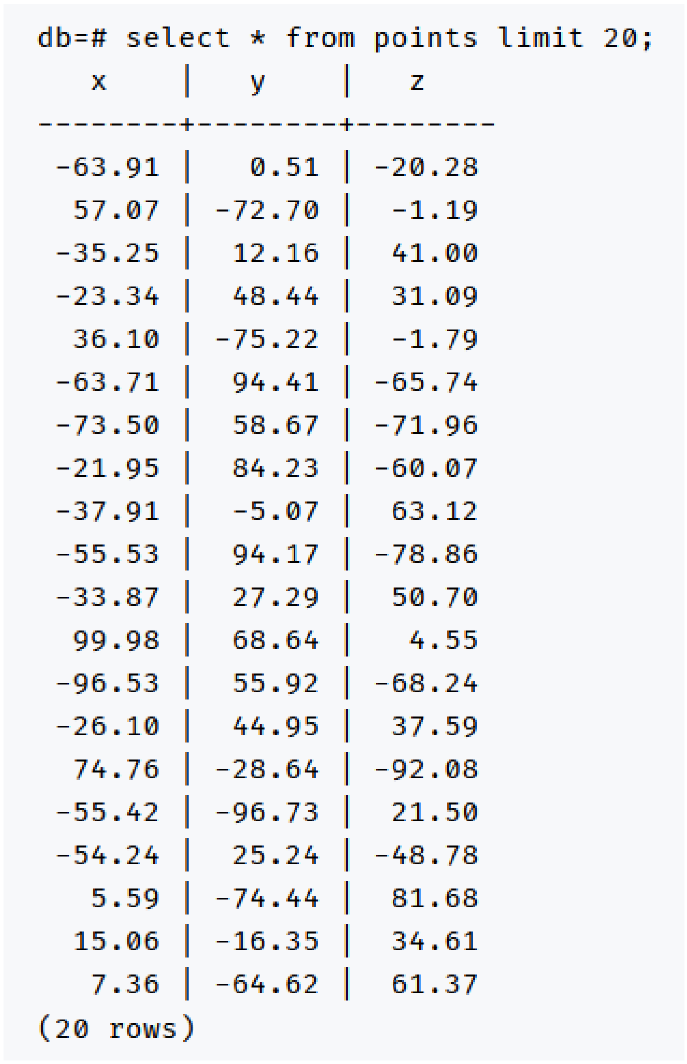
[1] Michael Ankerst, Gabi Kastenmuller, Hans-Peter Kriegel, and Thomas Seidl. 3d shape histograms for similarity search and classification in spatial databases. July 1999.

[2] Anand Kumar, Vladmir Grupcev, Yongke Yuan, Jin Huang, Yi-Cheng Tu, and Gang Shen. Computing spatial distance histograms for large scientific datasets on-the-fly. October 2014.

[3] Chengceng Mou. A comparative study of dual-tree algorithms for computing spatial distance histogram, November 2015.

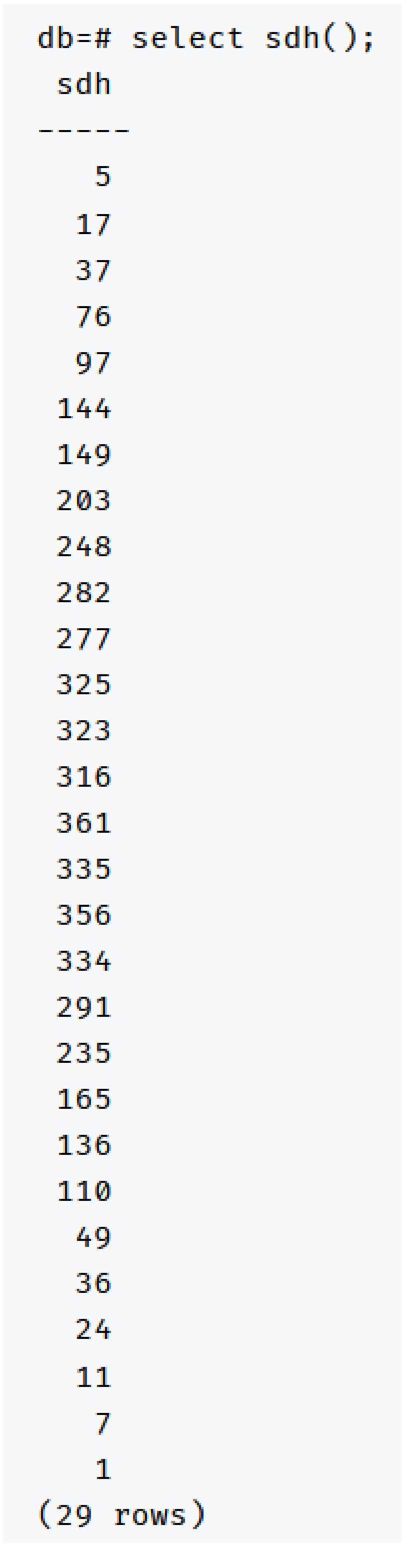
[4] PostgreSQL. 37.12. user-defined aggregates. https://www.postgresql.org/docs/12/xaggr.html, 2021.

[5] Yi-Cheng Tu, Shaoping Chen, and Sagar Pandit. Computing spatial distance histograms efficiently in scientific databases. pages 796–807, November 2009.



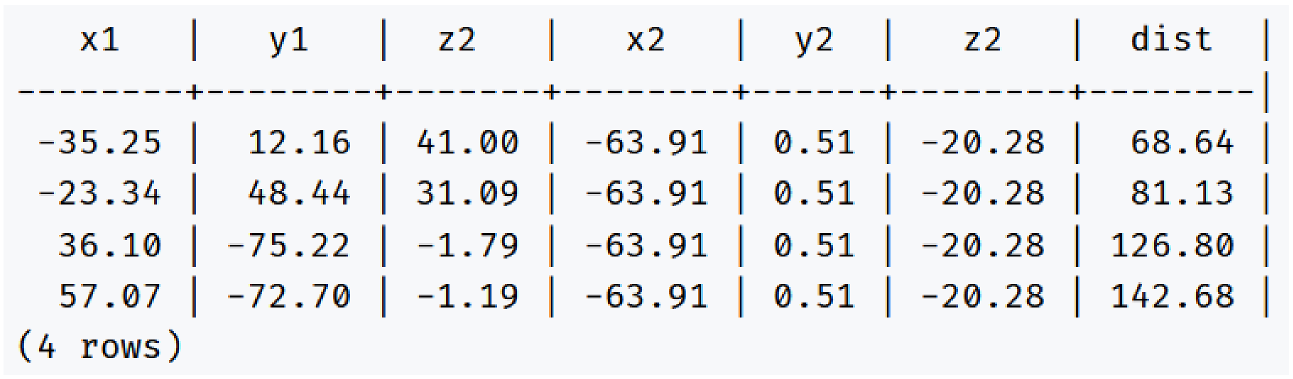
```
db=# select * from points limit 20;
   x     |    y     |    z
---------+---------+--------
 -63.91 |    0.51 | -20.28
  57.07 | -72.70 |  -1.19
 -35.25 |  12.16 |  41.00
 -23.34 |  48.44 |  31.09
  36.10 | -75.22 |  -1.79
 -63.71 |  94.41 | -65.74
 -73.50 |  58.67 | -71.96
 -21.95 |  84.23 | -60.07
 -37.91 |  -5.07 |  63.12
 -55.53 |  94.17 | -78.86
 -33.87 |  27.29 |  50.70
  99.98 |  68.64 |   4.55
 -96.53 |  55.92 | -68.24
 -26.10 |  44.95 |  37.59
  74.76 | -28.64 | -92.08
 -55.42 | -96.73 |  21.50
 -54.24 |  25.24 | -48.78
   5.59 | -74.44 |  81.68
  15.06 | -16.35 |  34.61
   7.36 | -64.62 |  61.37
(20 rows)
```

Fig. 1. An excerpt from our `points` table storing 3d points



```
db=# select sdh();
 sdh
-----
   5
  17
  37
  76
  97
 144
 149
 203
 248
 282
 277
 325
 323
 316
 361
 335
 356
 334
 291
 235
 165
 136
 110
  49
  36
  24
  11
   7
   1
(29 rows)
```

Fig. 2. Table showing the output of the SDH for 100 sample 3D points with a bucket width of 100



```
   x1    |   y1    |  z2    |   x2    |  y2   |   z2    |  dist   |
---------+---------+--------+---------+-------+---------+---------|
 -35.25  |  12.16  | 41.00  | -63.91  | 0.51  | -20.28  |  68.64  |
 -23.34  |  48.44  | 31.09  | -63.91  | 0.51  | -20.28  |  81.13  |
  36.10  | -75.22  | -1.79  | -63.91  | 0.51  | -20.28  | 126.80  |
  57.07  | -72.70  | -1.19  | -63.91  | 0.51  | -20.28  | 142.68  |
(4 rows)
```

Fig. 3. Table showing point to point distances