Fast Exploratory Analysis with Spatio-temporal Aggregation over Polygonal Regions

Catherine Higgins and Suprio Ray
Faculty of Computer Science, University of New Brunswick, Fredericton, Canada
Email: {c.higgins, sray}@unb.ca

***Abstract*—Exploratory data analysis, which is at the heart of data science workflows, is becoming important due to the rapid rise in spatio-temporal data volume, and popularity of Web and mobile mapping applications. Such exploratory data analysis often involves the user selecting an arbitrary polygon region to perform a statistical computation on the selected region. Existing approaches for spatio-temporal data aggregation support rectangular query regions only, and not arbitrary polygons. A recently proposed system called GeoBlocks supports polygonal queries, but GeoBlocks was designed for spatial data, not spatio-temporal data. Another aspect of exploratory data analysis is that the users often repeatedly perform similar statistical analyses over the same selected query region. Although the reuse of already computed answers can improve the response time, existing approaches do not support this reuse for advanced statistical analysis. Data Canopy is a recently proposed approach that supports statistics synthesis by reusing basic aggregates, however, it does not support spatial or spatio-temporal analysis.**

To address the mentioned challenges, we introduce *ScanCube*, an exploratory statistical analysis system over any arbitrary polygonal query region for any time interval. ScanCube also supports statistics synthesis by reusing a small set of basic aggregates that are computed and stored a priori. We introduce two techniques, ScanX1 and ScanX2, for providing a grid-based polygonal approximation, which offers distance-based bounded error. Experimental evaluation suggests that ScanCube significantly outperforms GeoBlocks.

I. Introduction

Spatial data generates significant value due to the billions of GPS-enabled devices, which gave rise to many spatial enabled applications [1]. Such applications can include interactive mapping applications such as Uber's Movement platform [2], where a user can select a static polygon region of interest. Static regions are limiting, however, for exploring a dataset. To support ad hoc polygon regions, applications require real-time response time in the milliseconds [3], but, there are associated challenges of querying enormous amounts of spatial data [4]. Summarizing large volumes of data into aggregates can significantly reduce query workload and provide fast response time. The storing, indexing and reuse of basic spatio-temporal aggregates is advantageous as exploratory analysis is often repetitive. In addition, spatial queries that use a polygonal boundary is computationally heavy and usually involve a *filter* step and a *refine* step. The filter uses a geometric approximation to filter out objects quickly and the refine step uses an expensive geometric algorithm to determine if objects lie within the polygonal boundary [5]. Therefore, using a geometric approximation for data analysis can significantly reduce query workload. Two common approaches (Fig. 2) are the Minimum Bounding Rectangle (MBR) [5], which is the smallest rectangle that encloses a geometric object and a grid-based approach, where a polygon region is converted into a set of unit grid cells overlap the polygon region.

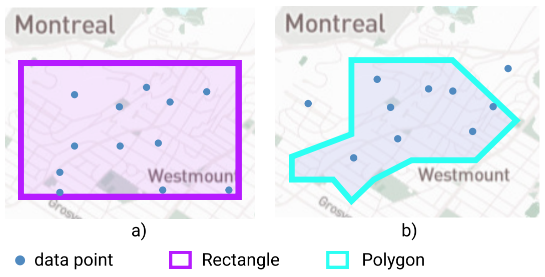

Fig. 1: Querying a: (a) rectangular region and (b) arbitrary polygonal region in a exploratory spatial enabled application.

Existing approaches for spatial and spatio-temporal aggregation query processing are designed for querying rectangular regions. These approaches include aR-tree [6], aRB-tree [7], Nanocubes [8], and Hashedcubes [9]. Figure 1 illustrates the difference between querying a rectangular region and a polygonal region with an example query over the Westmount suburb in Montreal. The polygonal query window can be much more specific in selecting a region of interest than the rectangular window in this example. GeoBlocks [4], a recently proposed system, employs a distance-based error bound polygonal approximation by using a grid-based approach, which consists of a hierarchical quadtree decomposition method. However, this grid-based approach suffers from an expensive intersection operation. Further, to our understanding, the creators of GeoBlocks in their paper did not include the overhead associated with this grid approximation in their overall query processing evaluation. A polygon rasterization approach by [10] also offers a distance-based error bound but suffers from an expensive rasterization process and only supports count and aggregate sum functions. A scanline method, inspired by a polygon filling algorithm [11] in the computer graphics community, supports a grid-based approach on vector polygon regions with raster dataset by avoiding rasterization or an expensive point-in-polygon algorithm [12]. To the best of our knowledge, a scanline based approach has not been considered for spatial or spatial-temporal aggregation over ad hoc arbitrary polygons for vector-only data.

Most of the existing spatial data systems lack efficient storage, index or reuse of basic spatio-temporal aggregates

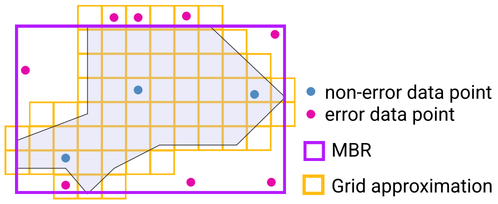

Fig. 2: Two types of spatial approximations: (a) Minimum Bounding Rectangle (purple) and (b) grid-based approximation (orange).

to support descriptive and advanced statistical operations for processing large volumes of data. In addition, many of the existing in-memory or disk-based approaches only support a very limited number of functions or consider spatial data only. GeoBlocks is focused on spatial aggregation, and does not directly support spatio-temporal aggregation in their paper. Supporting spatio-temporal data is important, as the temporal dimension is integral to being able to measure and understand change over time. Nanocubes, SmartCube and Hashedcubes support spatio-temporal aggregates but only support count aggregates. The aRB-tree and ST-Cube [13] are disk-based approaches that support spatio-temporal aggregates, but ST-Cube is also limited to a count aggregate. A rich ecosystem of libraries in R and Python supports a variety of libraries; however, many of these libraries do not support processing of large volumes of data and must be used in conjunction with database systems by means of database connectors. Even then, statistical functions are computed from scratch each time, resulting in lost opportunities to speed up query processing [14]. Many statistical equations are composed of basic aggregates or statistical functions (see Table I). Hence, there is an opportunity to pre-compute a set of basic aggregates, called pre-aggregates, and reuse them to synthesize complex statistics, which can speed up exploratory data analysis significantly. Data Canopy [14] uses temporal aggregates as building blocks to support complex descriptive statistics and machine learning algorithms such as linear regression. Data Canopy, however, does not support statistical composition with spatial or spatio-temporal aggregates. Note that although GeoBlocks supports aggregation queries over polygonal regions, it does not support complex statistical synthesis and it only supports basic statistics (*min*, *max*, *sum*, *count* and *mean*).

We propose ScanCube, a novel pre-aggregation based approach that supports real-time exploratory data analysis with spatio-temporal aggregation queries over arbitrary ad hoc polygonal regions for vector data. Similar to GeoBlocks, our approach guarantees an error bounded answer. On the other hand, In our experimental evaluation of ScanCube, we report the end-to-end query latency by including the cost of polygonal approximation. GeoBlocks does not include the cost of polygonal approximation in their evaluation. Our approach consists of four main features. **First**, we develop an efficient way to approximate the polygonal region by introducing two novel polygon approximation algorithms: ScanX1 and ScanX2. We demonstrate that these algorithms are significantly faster than the algorithm utilized in GeoBlocks. **Second**, we pre-compute and store basic aggregates to accelerate spatio-temporal queries involving descriptive and dependence statistics. Our pre-aggregation system is implemented using a grid index in which the geographic space is decomposed into unit cells for which pre-aggregates are computed and stored. The user can determine the granularity of the cell. **Third**, we can support the synthesis of a rich set of statistical functions by reusing a set of basic aggregates for each cell. As shown in Table I, these include basic distributive statistics, standard descriptive statistics (e.g. *standard deviation* and *sample covariance*), geographic statistics (e.g. *mean center* and *standard distance*), and advanced statistics (e.g. *sample correlation*, *simple linear regression*). Moreover, unlike Data Canopy, we support such statistics composition over arbitrary query polygons.

We conducted a systematic experimental evaluation of our system ScanCube with real world datasets, NYC Taxi and USA Tweets. Our results reveal that our approach achieves significantly better performance than GeoBlocks, which claims to be the first system to support ad hoc polygonal queries. In the best case ScanCube performs 16$\times$ faster compared to the GeoBlocks approach with end-to-end spatial polygonal queries.

II. Contributions

The main contributions of this thesis are as follows:

- A novel approach called ScanCube that supports spatio-temporal aggregation query on ad hoc arbitrary polygon regions with precision guarantee.
- Two new polygon approximation algorithms, ScanX1 and ScanX2, for efficient polygonal approximations that significantly accelerate query run-time.
- The synthesis of advanced statistical measures by storing, indexing and reusing basic aggregates.

III. Related Work

In this section, we present previous works that are related to our research.

A. Spatial and Spatio-temporal Point Index

Many data structures have been proposed to support indexing of spatial and spatio-temporal data. Index structures, such as the R-tree [15], build a hierarchy of MBRs that organize the spatial objects such as points at the leaf level. Another category of spatial index structures partitions the spatial domain such that the spatial objects are distributed among the partitions based on a specific criterion. Examples of such indexes are the quadtree [16] and k-d. Grid-based indexes [17] and space-filling curve indexes also leverage spatial partitioning.

Several spatio-temporal indexes have been proposed, owing to the recent popularity of location-based services (LBS). Approaches such as the STR-tree [18] and MV3R-tree [19] are based on the R-tree, whereas, BB^x index [20] and ST2B-tree [21] are B-tree based techniques.

TABLE I: Overview of functions that ScanCube can support and the role of aggregates or functions as a building block to synthesize statistics.

Name	Equation	Basic Aggregates						
		$\sum x$	$\sum y$	$\sum xy$	$\sum x^2$	$\sum y^2$	x_{min}	x_{max}
Basic Distributive Statistics								
Sum	$\sum x$	✓						
Min	x_{min}						✓	
Max	x_{max}							✓
Mean (avg)	$\frac{\sum x}{n}$	✓						
Geographic Statistics								
Mean Center (mc)	$\frac{\sum x_i}{n}, \frac{\sum y_i}{n}$	✓	✓					
Standard Distance (sd)	$\sqrt{\frac{\sum x^2 - n \cdot mc(x)^2}{n} + \frac{\sum y^2 - n \cdot mc(y)^2}{n}}$	✓	✓		✓	✓		
Standard Descriptive Statistics								
Root Mean Square (rms)	$\sqrt{\frac{\sum x^2}{n}}$				✓			
Variance (var)	$\frac{\sum x_i^2 - n \cdot avg(x)^2}{n}$	✓	✓					
Standard Deviation (std)	$\sqrt{\frac{\sum x_i^2 - n \cdot avg(x)^2}{n}}$	✓	✓					
Sample Covariance (cov)	$\frac{\sum x_i \cdot y_i}{n} - \frac{\sum x_i \cdot \sum y_i}{n^2}$	✓		✓		✓		
Advanced Statistics								
Simple Linear Regression (slr)	$\frac{cov(x,y)}{var(x)}, avg(x), avg(y)$	✓	✓	✓		✓		
Sample Correlation (cor)	$\frac{n \sum x_i y_i - \sum x_i \sum yi}{\sqrt{n \sum x_i^2 - (\sum x_i)^2} \cdot \sqrt{n \sum y_i^2 - (\sum y_i)^2}}$	✓	✓	✓	✓	✓		

B. Spatial and Spatio-temporal Index for Aggregation

Grouping and summarizing data into aggregates are basic data analysis operations. Spatial index structures, such as the R-tree, quadtree and grid-based indexes, are the foundation of building and indexing spatial aggregates. The aggregate R-tree (aR-tree) [6] is built by aggregating values at each node of an R-tree. The aR-tree, however, stores all records at the leaf nodes and is not suitable for an in-memory approach for large volumes of data. The aggregate Point-tree (aP-tree) [22] converts data points into key-time intervals, which are stored in a multi-version B-tree (MVB-tree) along with aggregate information. GeoBlocks [4] uses Hilbert space-filling curve to compute and store pre-aggregate data at different resolutions.

For spatio-temporal aggregation, only a few indexing approaches have been proposed. The aggregate RB-tree (aRB-tree) [7] is such an index that is an extension of the aR-tree. It stores additional B-trees at each node of the R-tree for more exact precision on the temporal dimension. The aggregate multi-version RB-tree (aMVRB-tree) [7] adopts the multi-version R-tree (MVR-tree) as a host index.

C. Polygonal Approximation

Spatial approximation is a term used to describe the abstraction (or simplication) of geometric objects including data points and complex geometric shapes such as polygons. For complex geometric shapes like polygons, the MBR is commonly used in the *filter* and *refine* approach [5]. The MBR, however, does not provide a meaningful approximation as points that lie within the MBR may in fact be quite far away [10]. The MBR is an example of a rectangular region. Many existing approaches, such as Nanocubes, Smartcube, aR-tree, Hashedcubes, and ST-Cube only support rectangular query windows.

GeoBlocks' authors claim their approach to be the first approach that supports arbitrary polygonal query regions in the context of vector data by using a unit grid-like approximation. GeoBlocks uses a hierarchical quadtree decomposition of a geographic space to retrieve a set of grid cells that overlap a polygon region. This polygon approximation method suffers from computation overhead in that every grid cell must be checked for containment. To our understanding, the overhead associated with GeoBlocks' polygonal approximation is not included in GeoBlocks paper for the overall query execution latency, and the main contribution of GeoBlocks is a caching system to support quick retrieval of aggregates. Therefore, the evaluation presented in GeoBlocks paper cannot be considered to be an end-to-end query evaluation. Polygonal approximation is also utilized in the zonal statistics problem where the underlying data is of raster form and the polygonal boundary is of vector form. In such cases, where two representations of data exist, a conversion of one data type to the other is needed. A spatial system that supports vector data, requires vectorizing the raster data into many points and suffers from needing to perform a point-in-polygon check [12]. Similarly, a raster system requires vectorizing the polygon into many pixels, which can be quite expensive at higher resolutions [12]. ScanCube, similar to [12], also avoids any expensive geometric containment operations in our polygonal approximation by using two new algorithms ScanX1 and ScanX2. Note that the focus of the work in [12] was to address the zonal statistics problem, which is specific to raster data, whereas our focus is spatio-temporal aggregation on ad hoc polygonal regions and complex statistical synthesis on vector-only data.

D. Data Systems for Statistical Analysis

Statistical analysis is central to data analysis processes. Several libraries and frameworks have been developed that support such statistical analysis. For instance, sophisticated libraries are supported by R and Python. An issue with these systems is that they do not support the reuse and composition of statistics from already calculated statistics (or pre-aggregates). As a result, every time a statistical function is evaluated, it must be computed from scratch. This can be time-consuming and hence not suitable for exploratory data analysis. To address these challenges, Data Canopy [14] was introduced, which enables computing and caching several basic aggregates a priori. Data Canopy only supports temporal aggregates, however, and does not consider spatial or spatial-temporal aggregation. Our system, ScanCube, is the first system that enables composition of statistics over an ad hoc arbitrary polygonal query region, by reusing a set of basic aggregates that are computed ahead in advance of ad hoc query processing.

IV. Problem statement

We define spatio-temporal polygonal aggregation query.

Definition 1:

R is a spatio-temporal relation and each record $r \in R$ is a tuple $(g, t, a_0, a_1, ...a_k)$, where a_i are the set of attributes, t is the timestamp of the tuple and g is the geometry specification of the record. The geometry specifies the set of m vertices i.e. $g = \{v_j | j = 1, 2, ..m\}$. When g is a point object, m=1 and it consists of a single vertex $v_1 = (x_1, y_1)$ that corresponds to the location of the point.

Given an arbitrary polygonal query region Q_P, a time interval Q_T, and an aggregate function F, where Q_P specifies a set of vertices $\{v_{q1}, v_{q2}, ..., v_{qm}\}$ and Q_T denotes a query time interval $[t_s, t_e]$, a *spatio-temporal polygonal aggregation query* reports an aggregated measure $Agg(Q_P, Q_T, F, R.a_i)$ on attribute $R.a_i$ for the geometric objects inside a region that intersects Q_P during Q_T. Formally,

$$Agg(Q_P, Q_T, F, R.a_i) = F\{r.a_i | r.g \ intersects \ Q_P \ and \ r.t \in Q_T\}, \forall r \in R \tag{1}$$

Additional filter conditions can be incorporated in the above definition that would filter out records based on a non-spatio-temporal predicate. The spatio-temporal polygonal aggregate query can be specified using a SQL syntax as follows:

```
select F(R.a_i) from R
where R.g intersects Q_P and R.t in Q_T
[ and additional_filter_condition]
```

V. ScanCube

We introduce ScanCube that enables fast exploratory analysis through spatio-temporal querying on large volumes of data by supporting the storage, indexing and reuse of basic spatio-temporal aggregates in main-memory. In the following sections, we discuss the overall system organization of ScanCube by describing the spatio-temporal index, and the construction and storage of spatio-temporal aggregates. Next, we present our polygonal approximation algorithms (Section V-C). We then discuss the query processing by first describing the grid-based polygonal approximation method and then retrieval of aggregates to compute basic aggregate functions (Section V-D). Lastly, we discuss the synthesis of descriptive statistics (Section V-E).

A. Spatio-temporal Aggregates

Spatio-temporal aggregates are aggregations that share a spatial and temporal extent [23]. To build spatio-temporal aggregates, we must partition data by both spatial and temporal dimensions. For spatial partitioning, we use a 2-dimensional grid index, which divides a space into equi-width square cells. Temporal partitions are also equal in size in that they have the same time span. Both spatial and temporal partitioning are configurable. For example, a user can adjust the level of precision of the approximation result by configuring the cell edge length. A user can also configure the granularity of the time span by hours, days, weeks, months or years. ScanCube is flexible in that users determine the level of granularity both for spatial and temporal dimensions. Users can also choose to build one or multiple cubes at a time to allow for drill-down and roll-up operations. To store the spatio-temporal aggregates that share both spatial and temporal extent, we use a 3-dimensional cube. The cube is a common multi-dimensional structure that allows grouping of data with one or more dimensions together. Figure 3 (c) shows the cube, where one face of the cube represents the spatial partitions and the other the temporal partitions. The build and retrieval of spatio-temporal aggregates depend on the spatio-temporal index, which is discussed next.

B. Spatio-temporal Index and Aggregate Construction

The spatio-temporal index is responsible for building spatio-temporal aggregates and indexing the 3-dimensional cube used for storing pre-aggregates. To build the spatio-temporal aggregates, we first configure the cell edge length of our 2-dimensional spatial grid indexes and determine the time span for the temporal partitions. We read data in chronological order, and build one spatial grid index for each temporal partition. We use a grid mapping function to map data points to a grid cell and for each grid cell, we collect a number of basic aggregates on various measures (Fig. 3 (c)). For any new data points being mapped to a cell, we update the previous basic aggregates to include the new data point. To index the 3-dimensional storage cube, we again use the grid index to access or map specific cells and use a segment tree to index the temporal partition intervals. Figure 3 (b) illustrates a segment tree. A node in the segment tree consists of the start and end time of a time span along with the corresponding indices in the cube's temporal dimension. A segment tree is an ideal structure for indexing time intervals, as to query a segment tree, one only needs to find the corresponding start and end node of the query interval and then include all temporal partitions that lie in between. The use of spatio-index to query processing is discussed in Section V-D.

C. Polygonal Approximation

We propose two new algorithms called *ScanX1* and *ScanX2* for polygon approximation that enable spatio-temporal polygonal aggregation querying. Our polygon approximation algorithms are inspired by a scanline polygon filling algorithm used in computer graphics to render objects [11]. We first describe the key aspects of the baseline scanline algorithm and then describe our new algorithms *ScanX1* and *ScanX2*. The baseline algorithm that we use has the following steps, which are summarized in Figure 4. In Step 1, we prepare an edge table for a polygon and sort the edges by their maximum *y* value as a pre-processing step. The slope is pre-computed and recorded in the edge table along with the minimum and maximum *y* values of the polygon during this step. In Step 2, using our spatial grid index, we determine the maximum and minimum row by using our grid mapping function with minimum and maximum *y* values of our polygon (Fig. 4(a)). In Step 3, we begin by processing each row in a top to bottom manner by casting a horizontal ray (i.e. scanline) from the centre of each

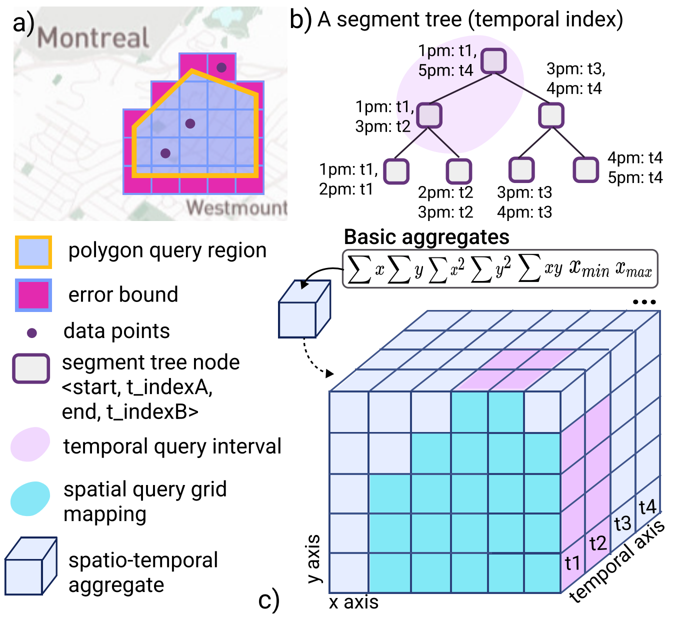

Fig. 3: A overview of the organization, index and query processing of ScanCube. (a) A query begins by querying an ad hoc polygonal region and applying a polygonal grid-based approximation. (b) The system utilizes the temporal index (segment tree) to retrieve the range of temporal slices in the 3-dimensional cube that satisfy the query time window criterion. In this example, the query time window is 1pm to 3pm. (c) Lastly, we use the polygonal approximation to map and retrieve specific pre-aggregates per spatio-temporal aggregate, illustrated as a unit cube.

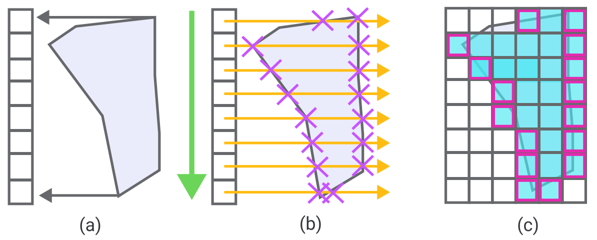

Fig. 4: Steps of the baseline polygon approximation algorithm (a) First, find maximum and minimum y values. (b) Second, find all x-intercepts of each horizontal ray cast (i.e. scanline) in a top to bottom manner. (c) Third, retrieve boundary cells (in pink) and cells that lie between boundary pairs.

grid cell starting from the maximum row as illustrated by a orange horizontal line in Figure 4 (b). For each row, we find the polygon edges that intersects a scanline by iterating over all edges whose maximum y value is greater or equal to the y value of the scanline. In Figure 4 (b), these are marked by a purple cross (x) to indicate the points at which the line intersects an edge. The x-intercept of an intersecting edge is calculated and added to an active edge table. Once all edges for a scanline are found, we sort the active edge table by the x-intercept value per row in ascending order. Before each new iteration of a scanline, we check the active edge table

Algorithm 1: ScanX1 Algorithm

```
Input  : A 2d array edges, a Grid grid, numbers
         ymax, ymin, startIdx, endIdx and minEdge,
         and 4d array storage.
Function ScanX1(edges, grid, ymin, startIdx,
  endIdx, minEdge, storage) : double
    max_row ← get_rmax(ymax, grid)
    min_row ← get_rmin(ymin, grid)
    numrows ← max_row- min_row
    active_edges ← [numrows +1][]
    sortchecker ← [numrows +1]
    I ← minEdge
    cur_row ← 0
    read_edges_clockwise (i, edges.length- 1,
      active_edges, sortchecker, cur_row)
    read_edges_clockwise (0, i, edges,
      active_edges, sortchecker, cur_row)
    return active_edges
Function read_edges_clockwise (start, end,
  edges,active_edges, sortchecker, cur_row)
    while start <= end do
        scanline ← get_scanline(cur_row)
        if edges [start ].y2 > edges [start ].y1 then
            while edges [i].y2 >= scanline do
                if is_within_range(edges [start
                 ]) then add edge
                    x ← get_x_intercept(edge
                     [start ], scanline, start)
                    add_edge(x, edges [start ],
                     active_edges, sortchecker)
                cur_row ← cur_row + 1
                scanline ←
                 get_scanline(cur_row)
        else
            while edges [i].y2 <= scanline do
                if is_within_range(edges [start
                 ]) then add edge
                    x ← get_x_intercept(edges
                     [start ], scanline, start)
                    add_edge(x, edges [start ],
                     active_edges, sortchecker)
                cur_row ← cur_row- 1
                scanline ←
                 get_scanline(cur_row)
        start ← start + 1
```

for polygon edges that are valid for the new iteration and add the edges with a new x-intercept value to the new row being processed. In Step 3 (Figure 4(c)), we use pairings of x-intercept values to determine boundary query cells using a grid mapping function. In Figure 4 (c), boundary cells are outlined in pink. Every aggregate value from cells that lie between two boundary cells is retrieved along with the aggregate values from boundary query cells.

ScanCube improves the scanline algorithm for the purpose of our application by skipping the initial sorting of edges of a polygon.

1) ScanX1: ScanX1 avoids the baseline algorithm's pre-processing step of sorting edges by their maximum y and iterates over edges in a clock-wise fashion independent of the maximum y value. Like the baseline, ScanX1 includes a pre-processing step, which consists of building an edge table, pre-computing the slope and storing the maximum and minimum y values. ScanX1 further stores the position of the edge with the smallest starting *y* value, as this edge determines the order in which to begin reading the polygon edges. ScanX1 (Algorithm 1) begins by first retrieving the minimum and maximum rows similar to the baseline Scanline Algorithm 1 (lines 3-4). ScanX1 also initializes a 2-d active edge table to store all active edges for every row within the maximum and minimum row range (line 6) and a 1-d array sort-checker to flag rows that need to sort active edges (line 7). ScanX1 differs from the baseline in that instead of adopting a top-down approach to cast new scanlines per row, we read the polygon edges in a clock-wise manner and scanlines are generated based on the direction of the edge (lines 9-10). If an edge is going upwards, we find all *x*-intercepts of rows that the edge occupies by generating scanlines in a bottom-to-top manner for each row (lines 15-21). A similar idea is applied if the edge is going downwards (lines 23- 28). Before any *x*-intercept is added to the respective active edge row, we check to determine if the previous *x*-intercept value is smaller. If not, we flag the sort-checker for that row to indicate a sort will be needed.

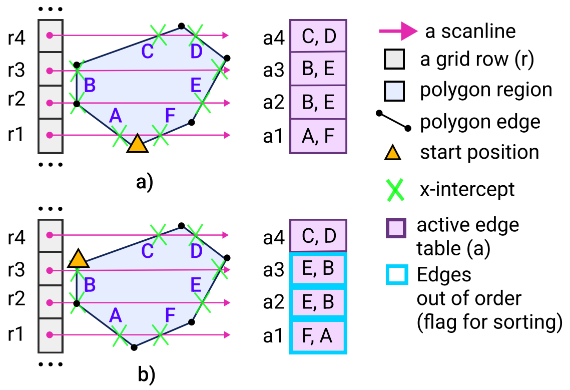

Fig. 5: ScanCube polygon approximation algorithms: a) ScanX1 b) ScanX2. ScanX1 begins to read the edge with the smallest starting *y* value, while ScanX2 begins with any arbitrary edge.

By reading the polygon in a clock-wise manner, we avoid the pre-processing sort. We illustrate this in Figure 5 (a). ScanX1 begins to read edge *A* and generates a scanline for *r1*. As the scanline intersects edge *A*, we find the *x*-intercept and add it to the active edge row *a1*. Since edge *A* is going in an upward direction, we then move up a row and generate a scanline for *r2*. As edge *A* does not intersect the scanline from *r2*, we iterate to the next edge, edge *B* and follow a similar procedure. Edge *B* does intersect *r2*. We then add edge *B* to *a2*. Since edge *B* is going upward, we generate a scanline for *r3* and see that *B* also intersects this scanline. We add edge *B* to the active edge row *a3*. The process repeats for edge *C* and then proceeds back in a downward direction for edges *D*, *E* and *F*.

2) ScanX2: ScanX2 differs from ScanX1 and the baseline algorithms in that there is no pre-processing step as polygon edges are processed on-the-fly. Like ScanX1, ScanX2 iterates over the polygon edges in a clock-wise manner. However, ScanX2 does not start by reading the edge with the smallest starting *y* value, but rather at any arbitrary edge of the polygon. ScanX2 also initializes a 2-d array of active edges table and a 1-d sort-checker. The ScanX2 algorithm for retrieving all *x*-intercepts is identical to ScanX1 (line 9) with the exception of keeping track of the maximum and minimum rows processed, as this was not determined in a pre-processing step and is required later to map out the grid approximation to the cube.

As shown in Figure 5 (b), ScanX2 begins to read the polygon at edge *C* and follows a similar approach previously described for Figure 5 (a).

3) Distance-based Bounded Error: ScanCube's polygonal approximation offers a distance-based error bound. Distance-based error bound means that the error is determined by the distance of two geometric objects usually measured by the Hausdorff distance [10]. In an approach that supports only rectangular query windows, a polygonal query region has to be converted into an MBR. However, with an MBR based approximation, there is no way to determine whether points lie outside or inside of a geometric boundary without a point-in-polygon test and consequently no way to determine which portion of data points contribute to an error. A grid-based approximation, on the other hand, provides us with some information on which data points contribute to exact results (i.e. inner grid cells) and that potentially contribute to error (i.e. boundary grid cells) [10]. Moreover, with such an approach, we can ensure that points that contribute to errors are within a certain distance by setting the edge length of grid cells. ScanCube use a grid-based approach, where the cell edge length determines the error-bound. We can determine for any boundary cell that the error is within a distance of $\sqrt{\varepsilon_1^2 + \varepsilon_2^2}$ where ε_1 and ε_2 are cell edge length. A smaller edge length contributes to a better precision for boundary cells while inner grid cells can be any size as they do not contribute to any error.

D. Query Processing

The steps for our spatio-temporal polygonal aggregate processing are described in Algorithm 2 and are illustrated in Figure 3. Such a query begins by processing an ad hoc polygonal region (Fig. 3(a)), followed by applying our polygonal approximation methods (line 4), which populate our active edge table with all *x*-intercepts that are used to find boundary cells. The active edge table is later used to map boundary edges to the spatial dimensions of Figure 3(c). Before retrieving the corresponding pre-aggregates, the temporal index or segment tree, is utilized to obtain the corresponding cube indices for the start and end time points of the query interval (line 5) (see Fig. 3(b)). In Figure 3, we queried 1pm to 3pm, which returns indices representing *t1* and *t2* respectively. To process and combine results from the basic aggregates, we first initialize accumulators, which act as storage for intermediate results. Essentially, an accumulator stores the $\sum_i agg_i$ of a primitive basic aggregate with the exception of x_{min} and x_{max}, which does not require a sum. In line 6, we call our function to process the active edges row by row and pass in our initialized accumulators. For each row, we determine if a sort is needed by checking the sort-checker (line 11). To process a row we

Algorithm 2: Spatio-temporal Polygonal Aggregation Query

```
Input : A 2d array edges, numbers start, end and
        querytype as functiontype, a Grid grid, 3d
        array storage and a segment tree tree.
1  Function spatio-temporal_aggregate
   (edges, start, end, grid, storage, functiontype,
   tree) : double
2      accumulators ← init_accumulators
         (functiontype)
3      results, startIdx, endIdx, cur_row, numrows ← 0
4      active_edges ← get_polygonal_approx
         (edges, querytype, cur_row, numrows)
5      probe_segment_tree (start, end, startIdx,
         endIdx, tree)
6      get_all_cell_aggs (cur_row, active_edges,
         , sortchecker, numrows, startIdx, endIdx,
         accumlators)
7      results ← process_function (accumlators,
         functiontype)
8      return results
9  Function get_all_cell_aggs (cur_row,
   active_edges, grid, sortchecker, numrows,
   startIdx, endIdx, accumlators) : double
10     for cur_row <= numrows + 1 do
11         if sortchecker [cur_row] == true then sort
12             sort_active_by_x (active_edges)
13         get_cell_aggs (max_row, active_edges,
             storage, startIdx, endIdx, accumlators)
14     return accumlators
15 Function get_cell_aggs (active_edges, storage,
   startIdx, endIdx, grid, accumlators) : double
16     $i$ ← 0
17     previous ← -1
18     while $i$ < active_edges do
19         col1, col2 ← get_column (grid, $i$,
             active_edges)
20         if col1 == previous then increment col1
21             col1 ← col1 + 1
22         for col1 <= col2 do
23             for startIdx <= endIdx do
24                 update_accumulator (storage
                     [startIdx ][max_row][col1],
                     accumlators)
25         previous ← col2
26         $i$ ← i + 2
27     return
```

call our function *get_cell_aggs* in line (13). This function processes every pair of *x*-intercepts by finding the boundary columns with our grid mapping function and retrieves the aggregates for the boundary cells and cells that lie between boundary cells, which are then processed by one or more accumulators.

E. Synthesizing Statistics

Many statistical formulas can be synthesized from a set of basic statistical functions or basic aggregates. For example, to compute the spatial statistic *mean centre* [24], we use the $\sum x$ and $\sum y$ coordinates and divide each by the count n, as shown in Table I. In addition, many statistical equations are the building blocks for other statistics. For example, the *mean centre* is used to compute the *standard distance*, which is the spatial equivalent of the *standard deviation* [24]. We illustrate in Figure 6 the general process of statistical synthesis. To begin, as mentioned previously, we pre-compute and store a set of basic aggregates for data that share a spatial and temporal extent. During the query processing, **a)** we process each spatio-aggregate cube (unit-cube) one at a time and extract specific aggregate values depending on the function to be passed to their respective accumulator. **b)** Each accumulator acts as a combiner to store intermediate results. Once all aggregate unit cubes are processed, we then collect all values from the accumulators, which are passed to a function processor. **c)** The function processor acts as the statistical equation constructor. In other words, the accumulator values are the ingredients needed for the function processor to compute the final results. The function processor only processes functions that are commutative or associative, that is, *count*, *sum*, *min*, *max* and product functions [14]. Further, the function processor may include one or more intermediate steps when dealing with statistics, which are built on the output of other statistics like the *standard distance*. Essentially, a function processor output can be the input to a new function processor.

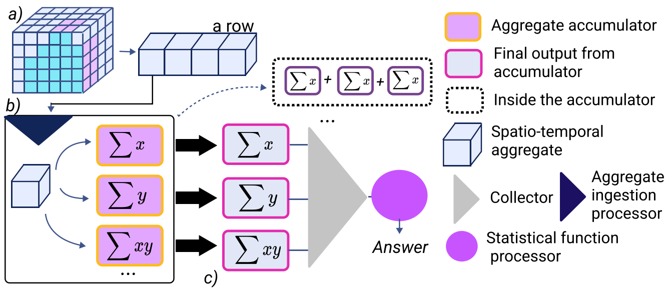
a)
a row
b)
c)
$\sum x$
$\sum y$
$\sum xy$
...
Answer
Aggregate accumulator
Final output from accumulator
Inside the accumulator
Spatio-temporal aggregate
Collector
Aggregate ingestion processor
Statistical function processor

Fig. 6: Steps in synthesizing statistics from basic aggregates. (a) Rows are processed one at a time. (b) Spatio-temporal aggregates are then processed one at a time by the aggregate ingestion processor. Basic aggregates are extracted and passed to their respective accumulator. (c) Final outputs of accumulators are collected and passed to a statistical function processor for statistic composition.

TABLE II: Experimental setup - approaches and their labels.

Approach [Label used in charts]	**Query window**	**Query type**
PH-tree [phtree]	Rectangle	Spatial
Aggregate R-tree [artree]	Rectangle	Spatial
GeoBlocks without caching [GBlock]	Polygon	Spatial
ScanCube with scanline [Base]	Polygon	Spatial/ Spatio-temporal
ScanCube with ScanX1 [SC1]	Polygon	Spatial/ Spatio-temporal
ScanCube with ScanX2 [SC2]	Polygon	Spatial/ Spatio-temporal

VI. Experimental Evaluation

The following sections describe our experimental setup, dataset, query set, pre-processing step, and performance evaluation of the index construction and query execution.

TABLE III: ScanCube and GeoBlocks corresponding grid resolution represented by a grid cell edge length in metres.

ScanCube	GeoBlock
4000m	4000-5000m
267m	281-306m
133.5m	140-153m
67.7m	70-77m
33.4m	35-38m
16.5m	18-19m

A. Experimental Setup

1) Approaches Evaluated: In our evaluation, we compare ScanCube performance with GeoBlocks, as well as two systems that the authors of GeoBlocks *evaluated in their paper*: aR-tree and PH-tree. Since ScanCube supports two new polygon approximation algorithms, ScanX1 and ScanX2, we show results for both of them, when applicable, and include them in the overall spatio-temporal polygonal query processing, which we refer to as SC1 and SC2 respectively. Table II shows the approaches we evaluate, where the text inside the square brackets are the labels used in the charts to represent the corresponding approach. Table IV shows a summary of the experiments we conducted and their details.

2) Dataset: We use a vector dataset for our experiments, called **NYC Taxi**, which consists of point data representing taxi trips from New York [25]. A baseline size of 10 million records is used in all experiments. The dataset includes information such as pick-up and drop-off location, passenger count, tip and fare amount, and distance traveled.

We also use a Twitter dataset, called **USA Tweets**, which consists of 2 million tweets with geolocations distributed across the United States.

3) Query Set: To simulate real-time ad hoc polygonal query windows, we use the census tract neighborhoods of New York when querying the NYC Taxi dataset. We use the USA state boundaries when querying the USA Tweets dataset. [26]. Our primary dataset is the NYC Taxi, unless specified otherwise. Each set of queries was run three times, where the first run-time is considered the warm-up run and is excluded from our results.

4) Experimental Equipment: Experiments were conducted on a computer with a Intel Core i5 and 16 GB of RAM. All approaches are implemented in Java and executed within a Ubuntu 20.04 OS.

B. Experimental results

1) Index Construction and Memory Overhead: We evaluate the build time of all structures with 10 million data points, which include the build time for constructing the index, and building and storage of pre-aggregates (Figure 7) (a). Note that GeoBlocks is the only approach in our evaluation that includes a sorting phase. Overall, the best build time is from SC1, which is 13× faster than the total build time of GeoBlocks including the sorting time and 16× faster than the PH-tree. The aR-tree by far had the slowest build time.

We compare the memory overhead of each build in Figure 7 (b). Altogether, the lowest memory consumption was from

TABLE IV: Summary of experiments and their details.

Experiment	Dataset	Measures	Figure
Index construction	NYC Taxi	Build time	7(a)
Memory overhead	NYC Taxi	Pre-aggregates and index storage	7(b)
Query performance	NYC Taxi USA Tweets	End-to-end query processing	8
ScanX1 and ScanX2	NYC Taxi USA Tweets	End-to-end query processing	9
Scalability	NYC Taxi	End-to-end query processing	10
Number of temporal partitions	NYC Taxi	End-to-end query processing	11
Grid resolution and query performance	NYC Taxi	End-to-end query processing	12(a)
		Retrieval of pre-aggregates only	12(b)
Grid resolution and relative error	NYC Taxi	n/a	12(c)
Statistical synthesis	NYC Taxi	End-to-end query processing	13

SC1. SC1 memory usage was 1.5× smaller than GeoBlocks. GeoBlocks stores a 64 bit spatial key that can account for a slightly higher space consumption. Since the aR-tree stores all individual records at the leaf nodes, the storage requirement for the aR-tree is the highest. The memory consumption of the PH-tree is higher than GeoBlocks and ScanCube, as PH-tree uses prefix sharing and not pre-aggregates.

2) Query Processing Time: We evaluate the end-to-end execution time for spatial query processing in Figure 8 for all spatial approaches. Query processing in Figure 8 refers to the execution time of the polygonal approximation and retrieval of pre-aggregates combined. Overall, the best query execution time is from SC2, which uses ScanX2 polygonal approximation method. The absence of a pre-processing step explains the faster execution time of SC2 over SC1, which uses ScanX1. The second best performance was from SC1, which was 13× faster than GeoBlocks with the NYC Taxi dataset. Note that GeoBlocks uses an S2 function for their polygonal approximation and that this was not part of their evaluation in their paper. The aR-tree also had a relatively fast query execution time, but the aR-tree is limited to a rectangular query window. To query over polygonal regions, the aR-tree uses an MBR, which requires a simpler spatial approximation computation than a grid-based polygonal approximation method such as, SC1, SC2 or GeoBlocks. This may explain why the aR-tree outperforms GeoBlocks. Nonetheless, the SC1 and SC2 polygonal approximation approaches do not seem to hinder query performance at all with the NYC Taxi dataset. We can observe, however, that the aR-Tree outperforms SC1 and SC2 with the USA Tweets dataset. Again, note that the PH-tree and aR-tree support rectangular query windows only. The PH-tree had the slowest time, which can be explained by the lack of aggregation that reduces the number of objects to be processed.

3) ScanX Polygonal Approximation: To evaluate our polygon approximation methods, ScanX1 and ScanX2, we implement the baseline scanline (Base) that follows the form

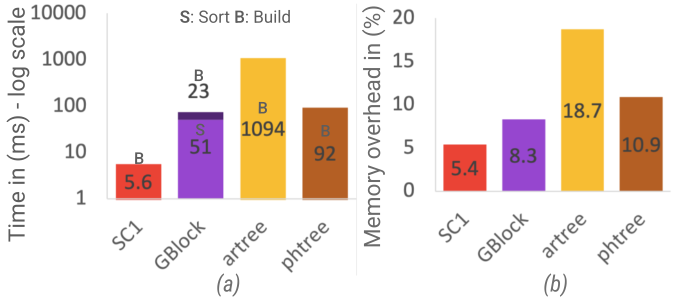

Fig. 7: Index construction time (a) and memory overhead (b) for 10 million data points.

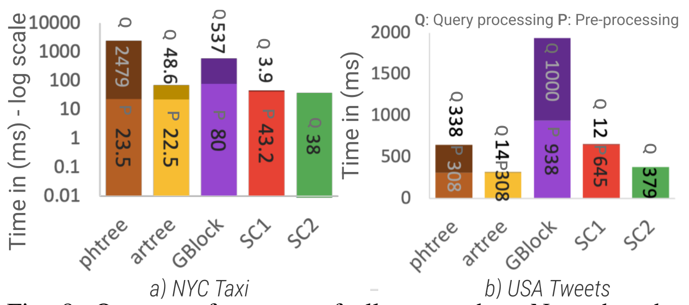

Fig. 8: Query performance of all approaches. Note that the PH-tree and aR-tree support rectangular query windows only.

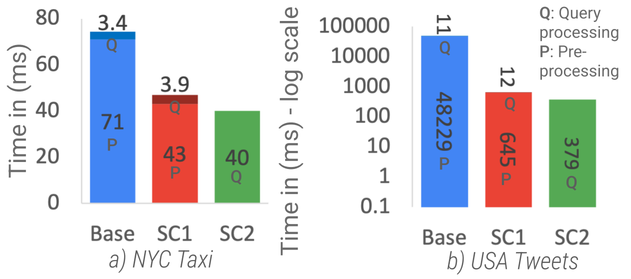

Fig. 9: Performance of ScanX1 (SC1) and ScanX2 (SC2) algorithms vs. basic scanline (Base) method.

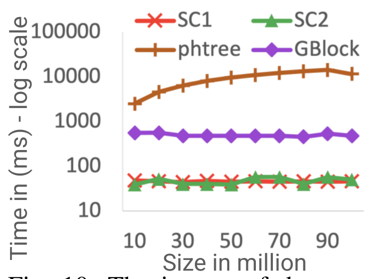

Fig. 10: The impact of data set size on **end-to-end query execution time**.

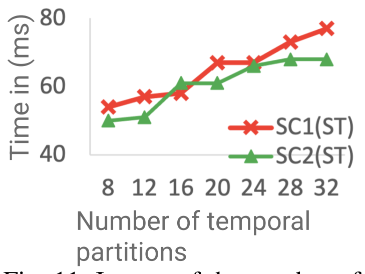

Fig. 11: Impact of the number of temporal partitions on execution time.

described in our approach section. The pre-processing step is included in our evaluation. As shown in Figure 9, the best performance was from SC2, which supports ScanX2 and was 1.9× faster than the Base with the NYC Taxi dataset and 127× faster than the Base with the USA Tweets. SC1 that supports ScanX1, was the second fastest and was 1.6× faster than the Base in (a) and 73.4× faster than the Base in (b). Note that the Base had a faster query processing time than SC1 but not by much. The real benefit of SC1 over the Base is in its pre-processing step. SC1 excludes a sorting step in its pre-processing step, which is included in the Base.

4) Scalability: We examine the scalability of all approaches, with the exception of the aR-tree. To study the scalability, we use NYC Taxi dataset and vary the size starting from 10 million up to 100 million, in 10 million increments We evaluate the scalability by examining the execution time for end-to-end querying (Fig. 10). We note that GeoBlocks and ScanCube both show consistent query times regardless of the size of the dataset for both Figure 10. Since both GeoBlocks and ScanCube use aggregates, the query performance depends on the number of pre-aggregates processed and not the total number of data points. In Figure 10, we observe a performance decrease of 4.6× for PH-tree from 10 to 100 million data points. The PH-tree does not use any aggregation method; therefore it requires processing a higher number of objects, which explains the increase in query latency.

5) Impact of Grid Resolution on Query Performance and Approximation Error: ScanCube and GeoBlocks both support distance-based bounded error, which are determined by the grid resolution (i.e. cell edge length), where a smaller grid resolution results in a higher precision guarantee. A smaller grid resolution, however, means that there are more pre-aggregates to retrieve. We investigate the end-to-end query execution (Fig. 12 (a)), retrieval of pre-aggregates execution time (Fig. 12 (b)) and relative error (Fig. 12 (c)) at various grid resolutions. Five different resolutions are measured. As we did not get an exact match of grid cell edge length between ScanCube and GeoBlocks, we will refer the reader to Table III for the corresponding grid cell edge settings.

In general, there is a decrease in performance for both ScanCube (SC1 and SC2) and GeoBlocks, as one might expect and is illustrated in Figure 12 (a) and (b). However, in Figure 12 (a), GeoBlocks has the biggest performance decrease for end-to-end query execution, 44×, while SC1 only decreases by 2.2×. In Figure 12 (b) GeoBlocks performance decreases by 35× and SC1 by 20× for retrieval of pre-aggregates only. Note that to our understanding, the authors of GeoBlocks in their paper evaluated the execution time for the retrieval of pre-aggregates only. We conclude that a higher number of pre-aggregates results in a decrease in performance overall, however, the GeoBlocks approach seems to suffer the most in Figure 12 (a) due to their polygonal approximation approach while in Figure 12 (b) results are much more comparable.

To measure the approximation error, we use the relative error. As shown in Figure 12 (c), as the cell edge length decreases, for both approaches we observe a lower relative error. However, a lower error for GeoBlocks comes at a greater cost of a performance decrease, which is not ideal for real-time analysis. The aR-tree and PH-tree are not included in Figure 12 because they only support rectangular query windows, and not

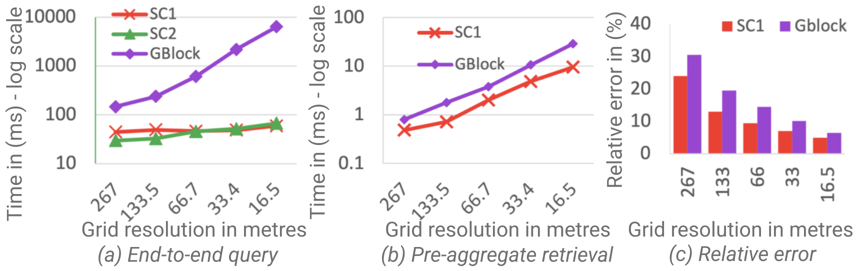

Fig. 12: Impact of grid resolution on end-to-end query execution time (a), retrieval of pre-aggregates only (b) and relative error (c).

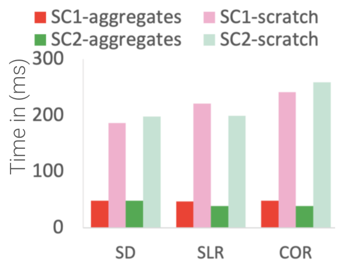

Fig. 13: Performance of statistics computed from scratch vs. statistical synthesis

polygonal query regions.

6) Statistical Synthesis: We examine the performance difference when computing statistical functions from scratch, in comparison to statistical synthesis by reusing basic aggregates. We adapt SC1 and SC2 to compute the statistics from scratch, in order to make the comparison fair. We evaluated 3 statistics: Sample Correlation (COR), Simple Linear Regression (SLR) and Standard Distance (SD). Overall, there was an improvement in all cases of statistics composition by reusing pre-aggregates (Fig 13). The biggest improvement was observed from Sample Correlation, with a speed-up of 5× for SC1 and 6.2× for SC2.

7) Temporal Partitions: To study ScanCube with spatio-temporal data, we investigate the impact of the number of temporal partitions in the overall query processing (Figure 11). Since GeoBlocks does not support spatio-temporal aggregation, we could not compare its performance in this experiment. We vary the number of temporal partitions starting from 8 up to 32. A partition here simply refers to an equal interval of time. Overall, both SC1 and SC2 performance decreases by 1.4× and 1.3× respectively.

VII. Conclusion

The ability to perform exploratory data analysis over ad hoc arbitrary polygonal regions has become a necessity. To support interactive sub-second response time during such analyses, pre-aggregation and statistical synthesis over pre-computed basic aggregates can be helpful. Existing approaches, except GeoBlocks, only support rectangular query regions.

We have presented ScanCube, that supports exploratory data analysis with spatio-temporal aggregation queries over arbitrary polygonal regions. It enables interactive response times with two fast polygon approximation algorithms and synthesizes advanced statistics by reusing a set of basic aggregates. Although Data Canopy supports statistical synthesis, it does not support spatial or spatio-temporal querying. GeoBlocks does not support statistical synthesis nor does it evaluate spatio-temporal querying. ScanCube introduces two new polygon approximation methods, ScanX1 and ScanX2. With extensive experimental analysis, we demonstrate that ScanCube significantly outperforms existing approaches evaluated in this paper.

References

[1] N. Henke, J. Bughin, M. Chui, J. Manyika, T. Saleh, and B. Wiseman, "The age of analytics: competing in a data-driven world," 2016.

[2] U. Movement, "Let's find smarter ways forward, together." [Online]. Available: https://movement.uber.com/?lang=en-CA

[3] Z. Liu and J. Heer, "The effects of interactive latency on exploratory visual analysis," *IEEE TVCG*, vol. 20, no. 12, pp. 2122–2131, 2014.

[4] C. Winter, A. Kipf, C. Anneser, E. T. Zacharatou, T. Neumann, and A. Kemper, "Geoblocks: A query-cache accelerated data structure for spatial aggregation over polygons," in *EDBT*, 2021, pp. 169–180.

[5] S. Ray, "High performance spatial and spatio-temporal data processing," Ph.D. dissertation, University of Toronto, 2015.

[6] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao, "Efficient olap operations in spatial data warehouses," in *SSTD*, 2001.

[7] Y. Tao and D. Papadias, "Historical spatio-temporal aggregation," vol. 23, no. 1, p. 61–102, 2005.

[8] L. Lins, J. T. Klosowski, and C. Scheidegger, "Nanocubes for real-time exploration of spatiotemporal datasets," *IEEE TVCG*, vol. 19, no. 12, pp. 2456–2465, 2013.

[9] C. A. L. Pahins, S. A. Stephens, C. E. Scheidegger, and J. L. D. Comba, "Hashedcubes: Simple, low memory, real-time visual exploration of big data," *IEEE TVCG*, vol. 23, pp. 671–680, 2017.

[10] E. T. Zacharatou, A. Kipf, I. Sabek, V. Pandey, H. Doraiswamy, and V. Markl, "The case for distance-bounded spatial approximations," *arXiv preprint arXiv:2010.12548*, 2020.

[11] C. Wylie, G. Romney, D. Evans, and A. Erdahl, "Half-tone perspective drawings by computer," in *Fall joint computer conference*, 1967.

[12] A. Eldawy, L. Niu, D. Haynes, and Z. Su, "Large scale analytics of vector+raster big spatial data," in *ACM SIGSPATIAL*, 2017.

[13] W. Choi, D. Kwon, and S. Lee, "Spatio-temporal data warehouses using an adaptive cell-based approach," *Data Knowl. Eng.*, vol. 59, no. 1, 2006.

[14] A. Wasay, X. Wei, N. Dayan, and S. Idreos, "Data canopy: Accelerating exploratory statistical analysis," in *SIGMOD*, 2017, p. 557–572.

[15] A. Guttman, "R-trees: A dynamic index structure for spatial searching," in *SIGMOD*, 1984, pp. 47–57.

[16] R. A. Finkel and J. L. Bentley, "Quad trees a data structure for retrieval on composite keys," *Acta informatica*, vol. 4, no. 1, pp. 1–9, 1974.

[17] P. van Oosterom, "Spatial access methods," *Geographical information systems*, vol. 1, pp. 385–400, 1999.

[18] D. Pfoser, C. S. Jensen, Y. Theodoridis *et al.*, "Novel approaches to the indexing of moving object trajectories." in *VLDB*, 2000, pp. 395–406.

[19] T. Yufei and D. Papadias, "Mv3r-tree: A spatio-temporal access method for timestamp and interval queries," *Techical report*, 2000.

[20] D. Lin, C. S. Jensen, B. C. Ooi, and S. Šaltenis, "Efficient indexing of the historical, present, and future positions of moving objects," in *MDM*, 2005, pp. 59–66.

[21] S. Chen *et al.*, "ST2B-tree: A Self-tunable Spatio-temporal B+-tree Index for Moving Objects," in *SIGMOD*, 2008.

[22] Y. Tao, D. Papadias, and J. Zhang, "Aggregate processing of planar points," in *EDBT*, vol. 2287. Springer, 2002, pp. 682–700.

[23] I. V. Lopez, R. T. Snodgrass, and B. Moon, "Spatiotemporal aggregate computation: A survey," *IEEE TKDE*, vol. 17, no. 2, pp. 271–286, 2005.

[24] J. E. Burt and G. M. Barber, *Elementary Statistics for Geographers*. Guilford Publications, 1996.

[25] N. Taxi and L. Commission, "Tlc trip record data." [Online]. Available: https://www1.nyc.gov/site/tlc/about/tlc-trip-record-data.page

[26] L. Chen, G. Cong, C. S. Jensen, and D. Wu, "Spatial keyword query processing: an experimental evaluation," in *PVLDB*, 2013.