GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons

Eleni Tzirita Zacharatou*‡, Harish Doraiswamy*†,
Anastasia Ailamaki‡, Cláudio T. Silva†, Juliana Freire†
‡ École Polytechnique Fédérale de Lausanne † New York University
{eleni.tziritazacharatou, anastasia.ailamaki}@epfl.ch {harishd, csilva, juliana.freire}@nyu.edu

ABSTRACT

Visual exploration of spatial data relies heavily on spatial aggregation queries that slice and summarize the data over different regions. These queries comprise computationally-intensive point-in-polygon tests that associate data points to polygonal regions, challenging the responsiveness of visualization tools. This challenge is compounded by the sheer amounts of data, requiring a large number of such tests to be performed. Traditional pre-aggregation approaches are unsuitable in this setting since they fix the query constraints and support only rectangular regions. On the other hand, query constraints are defined interactively in visual analytics systems, and polygons can be of arbitrary shapes. In this paper, we convert a spatial aggregation query into a set of drawing operations on a canvas and leverage the rendering pipeline of the graphics hardware (GPU) to enable interactive response times. Our technique trades-off accuracy for response time by adjusting the canvas resolution, and can even provide accurate results when combined with a polygon index. We evaluate our technique on two large real-world data sets, exhibiting superior performance compared to index-based approaches.

PVLDB Reference Format:
E. Tzirita Zacharatou, H. Doraiswamy, A. Ailamaki, C. T. Silva, and J. Freire. GPU Rasterization for Real-Time Spatial Aggregation over Arbitrary Polygons. *PVLDB*, 11(3): 352 - 365, 2017.
DOI: https://doi.org/10.14778/3157794.3157803

1. INTRODUCTION

The explosion in the number and size of spatio-temporal data sets from urban environments (e.g., [10,41,60]) and social sensors (e.g., [43,62]) creates new challenges for analyzing these data. The complexity and cost of evaluating queries over space and time for large volumes of data often limits analyses to well-defined questions, what Tukey described as *confirmatory data analysis* [61], typically accomplished through a batch-oriented pipeline. To support *exploratory analyses*, systems must provide interactive response times, since high latency reduces the rate at which users make observations, draw generalizations and generate hypotheses [34].

Not surprisingly, the problem of providing efficient support for visualization tools and interactive queries over large data has attracted substantial attention recently, predominantly for relational data [1,6,27,30,31,33,35,56,66]. While methods have also been proposed for speeding up selection queries over spatio-temporal data [17,70], these do not support interactive rates for aggregate queries, that slice and summarize the data in different ways, as required by visual analytics systems [4,20,44,51,58,67].

Motivating Application: Visual Exploration of Urban Data Sets. In an effort to enable urban planners and architects to make data-driven decisions, we developed Urbane, a visualization framework for the exploration of several urban data sets [20]. The framework allows the user to visualize a data set of interest at different resolutions and also enables the visual comparison of several data sets.

Figures 1(a) and 1(b) show the distribution of NYC taxi pickups (*data set*) in the month of June 2012 using a heat map over two resolutions: neighborhoods and census tracts. To build these heatmaps, aggregate queries are issued that count the number of pickups in each neighborhood and census tract. Through its visual interface, Urbane allows the user to change different parameters dynamically, including the time period, the distribution of interest (e.g., count of taxi pickups, average trip distance, etc.), and even the polygonal regions. Figure 1(c) shows multiple data sets being compared using a single visualization: a parallel coordinate chart [28]. In this chart, each data set (or dimension) is represented as a vertical axis, and each region (neighborhood) is mapped to a polyline that traverses across all of the axes, crossing each axis at a position proportional to its value for that dimension. Note that each point in an axis corresponds to a different aggregation for the selected time range for each neighborhood, e.g., Taxi reflects the number of pickups, while Price shows the average price of a square foot. This visual representation is effective for analyzing multivariate data, and can provide insights into the relationships between different indicators. For example, by filtering and varying crime rates, users can observe related patterns in property prices and noise levels over the different neighborhoods.

Motivating Application: Interactive Urban Planning. Policy makers frequently rezone different parts of the city, not only adjusting the zonal boundaries, but also changing the various laws (e.g., new construction rules, building policies for different building types). During this process, they are interested in viewing how the other aspects of the city (represented by urban data sets) vary with the new zoning. This operation typically consists of users changing polygonal boundaries, and inspecting the summary aggregation of the data sets until they are satisfied with a particular configuration.

In this process, urban planners may also place new resources (e.g., bus stops, police stations), and again inspect the coverage with respect to different urban data sets. The coverage is com-

* These authors contributed equally to this work.

Proceedings of the VLDB Endowment, Vol. 11, No. 3
Copyright 2017 VLDB Endowment 2150-8097/17/11... $ 10.00.
DOI: https://doi.org/10.14778/3157794.3157803

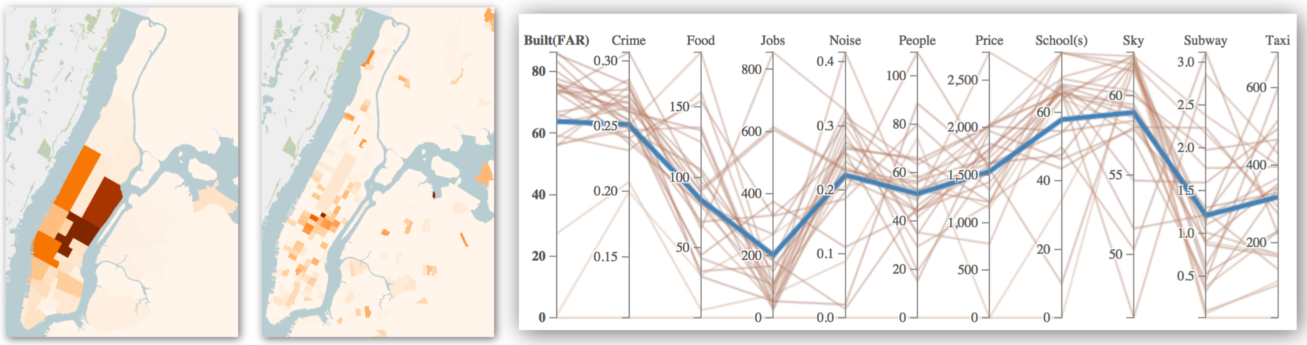


Figure 1: Exploring urban data sets using Urbane: (a) visualizing data distribution per neighborhood, (b) visualizing data distribution per census tract, (c) comparing data over different neighborhoods. The blue line denotes the NYC average for these data.

monly computed by using a restricted Voronoi diagram [7] to associate each resource with a polygonal region, and then aggregating the urban data over these polygons. To be effective, these summarizations must be executed in real-time as configurations change.

Problem Statement and Challenges. In this paper, we propose new approaches to speedup the execution of spatial aggregation queries, which, as illustrated in the examples above, are essential to explore and visualize spatio-temporal data. These queries can be translated into the following SQL-like query that computes an aggregate function over the result of a spatial join between two data sets, typically a set of points and a set of polygons.

```
SELECT AGG(a_i) FROM P, R
WHERE P.loc INSIDE R.geometry [AND filterCondition]*
GROUP BY R.id
```

Given a set of points of the form $P(loc, a_1, a_2, \ldots)$, where *loc* and a_i are the location and attributes of the point, and a set of regions $R(id, geometry)$, this query performs an aggregation (`AGG`) over the result of the join between P and R. Functions commonly used for `AGG` include the count of points and average of the specified attribute a_i. The `geometry` of a region can be any *arbitrary polygon*. The query can also have zero or more `filterConditions` on the attributes. In general, P and R can be either tables (representing data sets) or the results from a sub-query (or nested query).

The heat maps in the Figures 1(a) and 1(b) were generated by setting P as pickup locations of the taxi data; R as either neighborhood (a) or census tract (b) polygons; `AGG` as COUNT(*); and filtered on time (June 2012). On the other hand, to obtain the parallel coordinate visualization in Figure 1(c), multiple queries are required: *the above query has to be executed for each of the data sets of interest that contribute to the dimensions of the chart.*

Enabling fast response times to such queries is challenging for several reasons. First, the point-in-polygon (PIP) tests to find which polygons contain each point require time linear with respect to the size of the polygons. Real-world polygonal regions have complex shapes, often consisting of hundreds of vertices. This problem is compounded due to the fact that data sets can have hundreds of millions to several billion points. Second, as illustrated in the examples above, when using interactive visual analytics tools, users can dynamically change not only the filtering conditions and aggregation operations, but also the polygonal regions used in the query. Since *the query rate is very high* in these tools, delays in processing a query have a snowballing effect over the response times.

Existing spatial join techniques, common in database systems, are costly and often suitable only for batch-oriented computations. The join is first solved using approximations (e.g., bounding boxes) of the geometries. Then, false matches are removed by comparing the geometries (e.g., performing PIP tests), which is a computationally expensive task. This two stage evaluation strategy also introduces the overhead of materializing the results of the first stage. Finally, the aggregates are computed over the materialized join results and incur additional query processing costs. Data cube-based structures (e.g., [33]) can be used to maintain aggregate values. However, creating such structures requires costly pre-processing while the memory overhead can be prohibitively high. More importantly, these techniques *do not support queries over arbitrary polygonal regions*, and thus are unsuitable for our purposes.

Last but not least, while powerful servers might be accessible to some, many users have no alternative other than commodity hardware (e.g., business grade laptops, desktops). Having approaches to efficiently evaluate the above queries on commodity systems can help democratize large-scale visual analytics and make these techniques available to a wider community.

For visual analytics systems, approximate answers to queries are often sufficient as long as they do not alter the resulting visualizations. Moreover, the exploration is typically performed using the "level-of-detail" (LOD) paradigm: first look at the overview, and then zoom into the regions of interest for more details [53]. Thus, these systems can greatly benefit from an approach that trades-off accuracy for response times, and enables LOD exploration that improves accuracy when focusing on details.

Our Approach. By leveraging the massive parallelism provided by current generation graphics hardware (Graphics Processing Units or GPUs), we aim to support interactive response times for spatial aggregation over large data. However, accomplishing this is challenging. Since the memory capacity of a GPU is limited, data must be transferred between the CPU and GPU, and this introduces significant overhead when dealing with large data. In addition, to best utilize the available parallelism, GPU occupancy must be maximized. We propose rasterization-based methods that use the following key insights to overcome the above challenges:

- *Insight 1:* It is not necessary to explicitly materialize the result of the spatial join since the final output of the query is simply the aggregate value;
- *Insight 2:* A spatial join between two data sets can be considered as "drawing" the two data sets on the same canvas, and then examining their intersections; and
- *Insight 3:* When working with visualizations, small errors can be tolerated if they cannot be perceived by the user in the visual representation.

Insight 1 allows combining the aggregation operation with the actual join. The advantages of this are twofold: (i) no memory needs to be allocated for storing join results, allowing the GPU to process more input data, and thus computing the result in fewer passes; and

(ii) since no materialization (and corresponding data transfer overhead) is required, query times are improved. Insight 2 allows us to frame the problem of evaluating spatial aggregation as renderings, using operations that are highly optimized for the GPU. In particular, it allows us to exploit the *rasterization* operation, which converts a polygon into a collection of pixels. Rasterization is an important component of the graphics rendering pipeline and is natively supported by GPUs. As part of the driver provided by the hardware vendors, rasterization is optimized to make use of the underlying architecture and thus maximize occupancy of the GPU. By allowing approximate results, Insight 3 enables a mechanism to completely avoid the costly point-in-polygon tests, and use *only* the drawing operations, thus leading to a significant performance improvement over traditional techniques. Moreover, it allows an algorithmic design in which the input data is transferred *only once* to the GPU, further reducing the memory transfer overhead.

Even though our focus in this work is to enable seamless interaction on visual analysis tools, we would like to note that the spatial aggregation has utility in a variety of applications in several fields. For example, this type of query is commonly used to generate scalar functions for topological data analysis [11,16,37]. While these applications might not require interactivity per se, having fast response times would definitely improve analysis efficiency.

Contributions. Our contributions can be summarized as follows:

- Based on the observation that spatial databases rely on the same primitives (e.g., points, polygons) and operations (e.g., intersections) common in computer graphics rendering, we *develop spatial query operators that exploit GPUs and take advantage of their native support for rendering*.
- We *propose bounded raster join, an efficient approximate approach, that by eliminating the need for costly point-in-polygon tests provides close to accurate results in real-time*.
- We *develop an accurate variant of the bounded raster join that combines an index-based join with rasterization to efficiently evaluate spatial aggregation queries*.

To the best of our knowledge, this is the first work that efficiently evaluates spatial aggregation using rendering operations. The advantages of blending computer graphics techniques with database queries are amply clear from our comprehensive experimental evaluation using two real world data sets– NYC taxi data (~868 million points) and geo-tagged Twitter (~2.2 billion points). The results show that the bounded raster join obtains over two orders of magnitude speedup compared to an optimized CPU approach when the data fits in main memory (note that the data need not fit in GPU memory), and over 30X speedup otherwise. In fact, it can execute queries involving over 868 million points in only 1.1 second even on a current generation laptop. We also report the accuracy-efficiency as well as bound-error trade-offs of the bounded approach, and show that the errors incurred using even a very coarse bound do not impact the quality of the generated visual representations. This makes our approach extremely valuable for visualization-based exploratory analysis where interactivity is essential. Given the widespread availability of GPUs on desktops and laptops, our approach brings large-scale analytics to commodity hardware.

2. RELATED WORK

Spatial Aggregation. To support interactive response times for analytical queries in visualization systems, compact data structures such as Nanocubes [33] and Hashedcubes [45] have been designed to store and query the CUBE operator for spatio-temporal data sets. These techniques mainly rely on static pre-computation: they pre-aggregate records at various spatial resolutions and store this summarized information in a hierarchy of rectangular regions (maintained using a quadtree). To enable filtering and aggregation support over different attributes, these attributes must be known at build-time to be included as a dimension of the cube. Also, the granularity of the filtering depends on the number of discrete ranges the attribute is divided into. Thus, supporting filtering and aggregation over arbitrary attributes not only entails substantial pre-computation costs, but also *exponentially* increases the storage requirements, often making it impractical for real-world, large data sets. More importantly, since these structures maintain aggregate information over a hierarchy of rectangular regions, they have three key limitations: 1) the queries supported are constrained to only rectangular regions; 2) spatial aggregation has to be executed as a collection of queries, one for each region, which is inefficient for a large number of regions; and 3) the computed aggregates are approximate and the error cannot be dynamically bounded (since the accuracy depends on the quadtree resolution). Supporting arbitrary polygons and obtaining accurate results requires accessing the raw data (which might require additional spatial indexes) and defeats the purpose of maintaining a cube structure.

Several algorithms have also been proposed by the database community to evaluate spatial aggregate queries [59,63]. For instance, the aRtree [46] enhances the R-tree [24] structure by keeping aggregate information in intermediate nodes. These algorithms rely on annotated data structures and thus suffer from the aforementioned key limitations. Besides, they only support a spatial range selection predicate and do not support predicates on other attributes which makes them unsuitable for a dynamic setting. Closest to our approach are online aggregation techniques for spatial databases. However, prior work in the area [65] is also limited to range queries and does not provide support for join and group-by predicates.

Spatial Joins on CPUs. Our work is closely related to spatial join techniques since the join operation is the most expensive component of spatial aggregation queries. However, recall that explicit materialization of the join results is not required. Spatial joins typically involve two steps: filtering followed by refinement. The filtering step finds pairs of spatial elements whose approximations (minimum bounding rectangles - MBRs) intersect with each other, while the refinement step detects the intersection between the actual geometries. Past research on spatial join algorithms has largely focused on the filtering step [9,29,47,48]. To improve the processing of spatial queries over complex geometries, the Rasterization Filter [73] approximates polygons with rectangular tiles and serves as an additional filtering step that reduces the number of costly geometry-geometry comparisons. This approximation is calculated statically and stored in the database. In contrast, our approach exploits GPU rasterization to produce a fine-grained polygonal approximation on-the-fly and completely eliminates MBR-based tests. Apart from the aforementioned standalone solutions, several commercial and freely available DBMSs offer spatial extensions complying with the two-step evaluation process [14, 15, 19, 39, 50, 64]. While the filtering step is usually efficient, the refinement often degrades query performance since it involves costly computational geometry algorithms [55]. As a point of comparison, we performed a join between only 10 NYC neighborhood polygons and the taxi data using a commercial database. The query took over ten minutes to execute. This performance is not suitable for interactive visual analytics systems. More recently, distributed solutions such as Hadoop-GIS [3] and Simba [68] were proposed for spatial query processing. Both these solutions suffer from network bottlenecks which might affect interactivity, and also rely on the presence of powerful clusters for processing. Hadoop-based solutions are further constrained due to disk I/O. As we show in our experiments, our approach attains interactive speeds using GPUs that are ubiquitous in current generation desktops and laptops.

Spatial Query Processing on GPUs. Over the past decade, several research efforts have leveraged programmable GPUs to boost the performance of general, data-intensive operations [5,18,22,26,32]. Earlier techniques (e.g., [22]) employed the programmable rendering pipeline to execute these queries. Due to a fixed set of operations supported by the pipeline, it often resulted in overly complex implementations to work around the restrictions. With the advent of more flexible GPGPU interfaces, there have been several full fledged GPU-accelerated RDBMSs [8, 36]. MapD [36] accelerates SQL queries by compiling them to native GPU code and leveraging GPU parallelism. It is a relational database that currently does not support polygonal queries[1]. On the other hand, exploiting graphics processors for spatial databases is natural, as it involves primitive types (geometrical objects) and operations (spatial selections, containment tests) that are similar to the ones used in graphics. However, there has been limited amount of work in this area. Sun et al. [57] used GPU rasterization to improve the efficiency of spatial selections and joins. In the case of joins, they used rasterization as part of the join refinement phase to determine if two polygons do not intersect. However, this approach does not scale with an increasing number of polygons since the GPU is only used to perform pairwise comparisons. In contrast, the rasterization pipeline plays an integral part in our technique. By exploiting the capabilities of modern GPUs, we are able to perform more complex operations at faster speeds.

Closest to our work, Zhang et al. [69, 71] used GPUs to join points with polygons. They index the points with a Quadtree to achieve load balancing and enable batch processing. In the filtering step of the join, the polygons are approximated using MBRs. Zhang et al. [70] use the spatial join technique proposed in [69] to pre-compute spatial aggregations in a pre-defined hierarchy of spatial regions on the GPU. In contrast, we perform the aggregation on-the-fly, taking into account dynamic constraints. More recently, they extended their spatial join framework [72] to handle larger point data sets. As they materialize the join result, to optimize the memory usage, they make the limiting assumption that no two polygons intersect, thus ensuring the join size is at most the size of the input. Additionally, to improve efficiency, they truncate coordinates to 16-bit integers, thus resulting in approximate joins as well. Because we focus on analytical queries that do not require explicit materialization of the join result, we can use rasterization to better approximate the polygons as well as combine the join with the aggregation operation.

Aghajarian et al. [2] employ the GPU to join non-indexed polygonal data sets. The focus of our work, however, is aggregating points contained within polygonal regions. Doraiswamy et al. [17] proposed a customized kd-tree structure for GPUs that supports arbitrary polygonal queries. While the proposed index provides interactive response times for selection queries, the evaluation of the join requires one selection to be performed for each polygon, and is thus inefficient when the polygon data set is large.

3. BACKGROUND: GRAPHICS PIPELINE

The most common operation in graphics-intensive applications (e.g., games) is to render a collection of triangular and polygonal meshes that make up a scene. To achieve *interactivity*, such applications rely heavily on rasterization and approximate visual effects (e.g., shadows) to render the scenes. Modern GPUs exhibit impressive computational power (the latest Nvidia GTX 1080 Ti reaches 10.6 TFLOPS) and implement rasterization in hardware to speedup the rendering process. The key idea in our approach is to leverage the graphics hardware rendering pipeline for rasterization and the efficient execution of spatial aggregation queries.

[1]MapD currently has only one GIS function: `https://www.mapd.com/docs/latest/mapd-core-guide/dml/#geometric-function-support`

Rasterization-based Graphics Pipeline. Rendering a collection of triangles is accomplished in a series of processing stages that compose a graphics pipeline. First, the coordinates of all the vertices (of the triangles) that compose the scene are transformed into a common world coordinate system, and then projected onto the screen space. Next, triangles falling outside the screen (also called *viewport*) are discarded, while those partially outside are *clipped*. Parts of triangles within the viewport are then *rasterized*. *Rasterization* converts each triangle in the screen space into a collection of *fragments*. Here, a *fragment* can be considered as the data corresponding to a pixel. The fragment size therefore depends on the *resolution* (the number of pixels in the screen space). For example, a 800×600 rendering of a scene has fewer pixels (480k pixels) than a high resolution rendering (e.g., $1920 \times 1080 \approx 2$M pixels), and thus has a bigger fragment size. In the final step, each fragment is appropriately colored and displayed onto the screen. Figure 2 shows an example where a triangle is rasterized (pixels colored violet) at a given resolution.

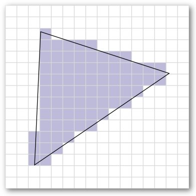


Figure 2: Rasterizing a triangle into pixels.

OpenGL [54], a cross platform rendering API, supports the ability to program parts of the rendering pipeline with a shading language (GLSL), thus allowing for custom functionality. In particular, the custom rendering pipeline, known as *shader programs*, commonly consists of a *vertex shader* and a *fragment shader*. The vertex shader allows modifying the first stage of the pipeline, namely, the transformation of the set of vertices to the screen space. The clipping and rasterization stages are handled by the GPU (driver). Finally, the fragment shader allows defining custom processing for each fragment that is generated. Both shaders are executed using a single program, multiple data (SPMD) paradigm.

Rasterization. Given the crucial part it plays in the graphics pipeline, parallel rasterization has had a long research history, as can be seen from these classical papers [38, 49]. Hardware vendors optimize parallel rasterization by directly mapping computational concepts to the internal layout of the GPU. While the details of the rasterization approaches used in current GPU hardware are beyond the scope of this work, we briefly describe the key ideas.

Hardware drivers typically use a variation of the algorithm proposed by Olano and Greer [42]. As a key optimization, they focus on the rasterization of triangles instead of general polygons. The triangle is the simplest convex polygon, and it is thus computationally efficient to test whether a pixel intersects with it. The intersection tests are typically performed by solving linear equations, called edge functions [42]. The rasterization algorithm allows to test whether pixels lie within a given triangle in parallel, and thus is amenable to hardware implementation.

Triangulation. Rendering polygons on the GPU is often accomplished by decomposing them into a set of triangles, an operation called *triangulation*. The problem of polygon triangulation has a rich history in the computational geometry domain. The two most common approaches for triangulation are the ear-clipping algorithm [7] and Delaunay triangulation, in particular, a constrained Delaunay triangulation [52]. Delaunay-based approaches have the advantage of providing theoretical guarantees regarding the quality of the generated triangles (such as minimum angle), and are often preferred for generating better triangle meshes. In this work, we employ constrained Delaunay polygon triangulation.

Frame buffer objects (FBO). Instead of directly displaying the rendered scene onto a physical screen (monitor), OpenGL also allows outputting the result into a "virtual" screen. The virtual screen is represented by a *frame buffer object* (FBO) having a resolution defined by the user. Even though the resolutions supported by existing monitors are limited, current graphics generation hardware supports resolutions as large as 32K × 32K. Each pixel of the FBO is represented by 4 32-bit values, $[r,g,b,a]$, corresponding to the red, blue, green, and alpha color channels. Users can also modify the FBO to store other quantities such as depth values. Since our goal is to compute the result of a spatial aggregation, we do not make use of any physical screen, but we make extensive use of FBOs to store intermediate results.

4. RASTER JOIN

Existing techniques execute spatial aggregation for a given set of points and polygons in two steps: (1) the spatial join is computed between the two data sets; and (2) the join results are aggregated. Such an approach has two shortcomings. The join operation is expensive, in particular, the PIP tests it requires – in the best-case scenario, one PIP test must be performed for every point-polygon pair that satisfies the join condition, and the complexity of each PIP test is linear with the size of the polygon. Even when the PIP tests are executed in parallel on the GPU, queries still require several seconds to execute even for a relatively small number of points (see Section 7 for details). To compute the aggregate as a second step, the join must be materialized. Consequently, given the limited memory on a GPU, the join has to be performed in batches, which incurs additional memory transfer between the CPU and GPU.

In this section, we first discuss how the rasterization operation can be applied to overcome the above shortcomings. We then propose two algorithms: bounded and accurate raster join, which produce approximate and exact results, respectively.

4.1 Core Approach

The design of raster join builds on two key observations:

1. A spatial join between a polygon and a point is essentially the intersection observed when the polygon and point are *drawn* on the same canvas.
2. Given that the goal of the query is to compute aggregates, if the join and aggregate operations are combined, there is no need to materialize the join results.

Intuitively, our approach *draws* the points on a canvas and keeps track of the intersections by maintaining *partial aggregates* in the canvas cells. It then *draws* the polygons on the same canvas, and computes the aggregate result from the partial aggregates of the cells that intersect with each polygon. The above operations are accomplished in two steps as described next. To illustrate our approach, we use the following example. We apply the query:

```
SELECT COUNT(*) FROM $D_{pt}$, $D_{poly}$
WHERE $D_{poly}$.region CONTAINS $D_{pt}$.location
GROUP BY $D_{poly}$.id
```

to the data sets shown in Figure 3, D_{poly} with 3 polygons, and D_{pt} with 33 points.

Step I. Draw points: The first step renders the points onto an FBO as shown in Procedure DrawPoints. We maintain an array A of size equal to the number of polygons, which is 3 for the example in Figure 3. This array is initially set to 0. When a point is processed, it is first transformed into the screen space, and then rasterization converts it into a fragment that is rendered onto an FBO. In this FBO, we use the color channels of a pixel for storing the count of points falling in that pixel. Instead of setting a color to that pixel, we *add* to the color of the pixel (e.g., the red channel of the pixel is incremented by 1). OpenGL only allows specifying colors for a fragment in the fragment shader. The way the specified color is combined with that in the FBO is controlled by a *blend function*. We set this function such that the specified color is added to the existing color in the FBO. This step results in F_{pt}, an FBO storing the count of points that fall into each of its pixels. The FBO for the input in Figure 3 is illustrated in Figure 4(a).

Procedure DrawPoints

```
Require: Points $D_{pt}$, Point FBO $F_{pt}$
1: Initialize array $A$ to 0
2: Clear point FBO $F_{pt}$
3: for each $p = (x,y) \in D_{pt}$ do
4:     $(x',y') = transform(p)$
5:     $F_{pt}(x',y')$ += 1    % can be any function, see Section 5
6: end for
7: return $A$, $F_{pt}$
```

Step II. Draw polygons: The second step renders all the polygons and incrementally updates the query result. As explained in Section 3, the polygons are first triangulated. All triangles corresponding to a polygon are assigned the same key (or ID) as that polygon. As before, the vertices of the polygons are transformed into the screen space and the rasterization converts the polygons into discrete fragments. The generated fragments are then processed in the fragment shader (Procedure DrawPolygons). When processing a fragment corresponding to a polygon with ID i, the count of points corresponding to this pixel (stored in F_{pt}) is added to the result $A[i]$ corresponding to polygon i. Figure 4(b) highlights the pixels that are counted with respect to one of the polygons. After all polygons are rendered, the array A stores the result of the query. As we discuss in Section 5, this approach can be extended to handle other aggregation and filtering conditions.

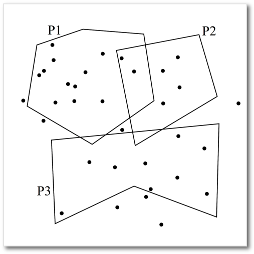


Figure 3: Example input.

Procedure DrawPolygons

```
Require: Polygon fragment $(x',y')$, Polygon ID $i$,
    Point FBO $F_{pt}$, Array $A$
1: $A[i] = A[i] + F_{pt}(x',y')$    % same function as in DrawPoints
2: return $A$
```

4.2 Bounded Raster Join

Raster join is an approximate technique that introduces some false positive and false negative points. In this section, we show that the number of these errors depends on the resolution and their 2D location can be bounded.

The introduction of *false negatives* is an artifact of the rasterization of the triangles that compose a polygon: a pixel is part of a triangle only when its center is inside the triangle. As a result, the points that fall in the intersection between a pixel and a triangle not containing the pixel's center are not aggregated. The pixels that intersect the polygon outline are considered to be part of the polygon and they introduce *false positives*, as all the points contained in those pixels are aggregated. In the example shown in Figure 4(b), P1 is approximated by the violet fragments and the false positive

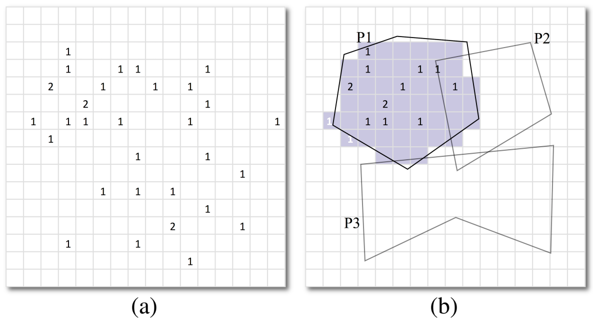


Figure 4: The raster join approach first renders all points onto an FBO storing the count of points in each pixel (a). In the second step, it aggregates the pixel values corresponding to fragments of each polygon (b).

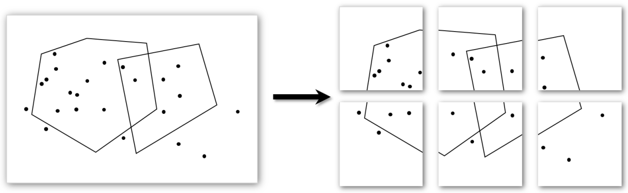


Figure 5: When the resolution required to satisfy the given ϵ-bound is greater than what is supported by the GPU, the canvas used for drawing the geometries is split into multiple small canvases, each having resolution within the GPU's limit.

counts are highlighted in white. By increasing the screen resolution, the pixel size decreases and thus pixels better approximate the polygon outline. As a result, the expected number of both false positives and false negatives decreases. Clearly, with an appropriately high resolution, we can converge to the actual aggregate result.

In real-world data, there is typically uncertainty associated with respect to the location of a point. Similarly, polygon boundaries (which often correspond to political boundaries) are fuzzy, in the sense that there is often some leeway as to their exact location. For example, the neighborhood boundaries of NYC fall on streets, and in most cases, the whole street surface (rather than a one-dimensional line) is considered to be the boundary. This means that when analyzing data over neighborhoods, it is often admissible to consider data points falling on boundary streets to be part of either of the two adjacent neighborhoods. In such cases, it is sufficient to compute the aggregate with respect to a polygon i' that closely approximates the given polygon i. Formally, a polygon i' ϵ-approximates the polygon i if the Hausdorff distance $d_H(i,i')$ between the polygons is at most ϵ, where

$$d_H(i,i') = \max\left\{\max_{p'\in i'}\min_{p\in i} d(p,p'), \max_{p\in i}\min_{p'\in i'} d(p',p)\right\}$$

Here, $d(p',p)$ denotes the Euclidean distance between two points.

Given ϵ, raster join can guarantee that $d_H(i,i') \le \epsilon$, by using a pixel side length equal to $\epsilon' = \frac{\epsilon}{\sqrt{2}}$ (i.e., the length of the diagonal of the pixel is ϵ). Intuitively, this ensures that any false positive (false negative) point that is present (absent) in the approximate polygon, and thus considered (or not) in the aggregation, is within a distance ϵ from the boundaries of polygon i. For example, the outline of the violet pixelated polygon in Figure 4(b) represents the approximation used corresponding to P1. In the example of NYC neighborhoods, a meaningful aggregate result is obtained by using a pixel size approximately equal to the average street width.

The required resolution onto which the points and polygons are rendered to guarantee the ϵ-bound is $w' \times h' = \frac{w}{\epsilon'} \times \frac{h}{\epsilon'}$, where $w \times h$ are the dimensions of the bounding box of the polygon data set. When ϵ becomes small, the required resolution $w' \times h'$ can be higher than the maximum resolution supported by the GPU. To handle such cases, the canvas is split into smaller rectangular canvases, and the raster join algorithm described in Section 4.1 is executed over each one of them. This multi-rendering process is illustrated in Figure 5. Recall that during the rendering process, the points or polygons that do not fall onto the canvas are *automatically* clipped by the graphics pipeline. This ensures that every point-polygon pair satisfying the join is correctly counted exactly once.

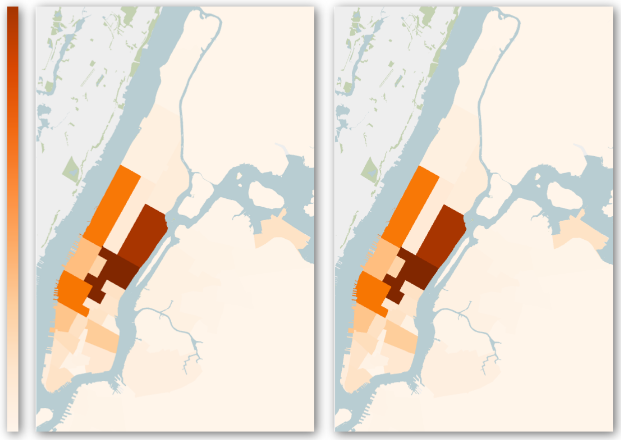


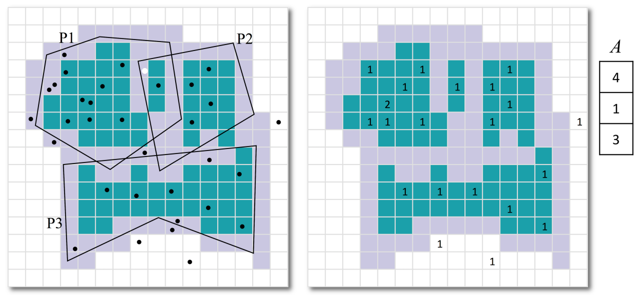
Figure 6: Visualizing the approximate (left) and accurate (right) results of the example query in Figure 1. The ϵ-bound was set to 20m. Note that the two visualizations are virtually indistinguishable from one another.

Typically, in a visualization scenario such as the motivating example in Section 1, it is perfectly acceptable to trade-off accuracy for interactivity, and increase the query rate by performing only a single rendering operation with a relatively low resolution. For example, Figure 6(left) shows the number of taxi pick-ups that happened in the month of June 2012 over the neighborhoods of NYC as obtained using the raster technique with a canvas resolution of approximately $4k \times 4k$ that corresponds to $\epsilon = 20$ meters. Note that this approximate result is almost indistinguishable from the visualization obtained from an accurate aggregation (right), but it can be computed at a fraction of the time. Also, if we fix a resolution as is common in visualization interfaces, when the user zooms into an area of interest, a smaller region is rendered with a larger number of pixels. Effectively, this is equivalent to computing the aggregation with a higher accuracy without any significant change in computation times (since the FBO resolution does not change). Thus, our approach is naturally suited for LOD rendering.

4.3 Accurate Raster Join

While Bounded Raster Join derives good (and bounded) approximations for spatial aggregation queries, some applications require accurate results. In this section, we describe a modification of the core raster approach that obtains exact results through the addition of a minimal number of PIP tests.

Consider the same point and polygon data sets described in the previous section, but as illustrated in Figure 7(a). Notice that certain fragments (pixels), colored green and white respectively, are either completely inside one of the polygons, or outside all polygons. Grid cells colored violet are on the boundary of one or more polygons. Recall that the errors from the raster approach are due only to the points that lie in these boundary pixels. This observation can be used to minimize the number of PIP tests: by performing tests just on these points, we can guarantee that no errors occur.



(a) (b)

Figure 7: Accurate raster join performs PIP tests only on points that fall on the violet cells in (a) that correspond to pixels forming the boundaries of the polygons. The other points are accumulated in the green pixels (b), which are then added to the polygons that are "drawn" over them.

This is accomplished in three steps.

1. Draw the outline of all the polygons: In this step, the boundaries of the set of polygons are rendered onto an FBO. In particular, the fragment shader assigns a predetermined color to the fragments corresponding to the boundaries of the polygons. The FBO is first cleared to have no color ($[0,0,0,0]$), thus ensuring that at the end of this step, only pixels on the boundary will have a color. The outline FBO for the example data will consist of an image having only the violet pixels from Figure 7.

2. Draw points: This step (Procedure AccuratePoints) builds on the core raster approach described above. As before, we maintain a result array A initialized to 0. When a point is processed, it is first transformed into the screen space. If the fragment corresponding to the point falls into a boundary pixel (which is determined by examining the pixel color in the Boundary FBO), the point is processed with Procedure JoinPoint. This procedure first uses an index over the polygons to identify candidate polygons that might contain the point, and then performs a PIP test for every candidate. Since our focus is on time efficiency, we use a grid index that stores in each grid cell the list of polygons intersecting it, thus allowing for constant $O(1)$ lookup time. If a point is inside the polygons with IDs $I = \{i_1, i_2, \ldots, i_l\}$, $l \leq k$, where k is the total number of polygons, then each of the array elements $A[i]$, $i \in I$, is incremented by 1.

If the fragment does not correspond to a boundary pixel, then this fragment is rendered onto a second FBO. In this FBO, as in the core approach, we use the color channels of a pixel to store the count of points falling in that pixel. This step results in two outputs: A, which stores the partial query result corresponding to data points that fall on the boundary of the polygons; and F_{pt}, an FBO storing the count of points that fall into each of its pixels (see Figure 7(b)).

Procedure AccuratePoints

```
Require: Polygon Index $Ind$, Points $D_{pt}$, Boundary FBO $F_b$
 1: Initialize array $A$ to 0
 2: Clear point FBO $F_{pt}$
 3: for each $p = (x,y) \in D_{pt}$ do
 4:     $(x',y') = transform(p)$
 5:     if $F_b(x',y')$ is a boundary then    % test pixel color in FBO
 6:         execute JoinPoint($Ind$, $p$, $A$)
 7:     else
 8:         $F_{pt}(x',y')$ += 1    % same function as in DrawPoints
 9:     end if
10: end for
11: return $A$, $F_{pt}$
```

3. Render polygons: The final step simply renders all the polygons and updates the query result when processing the polygon fragments in the fragment shader (Procedure AccuratePolygons).

Procedure JoinPoint

```
Require: Polygon Index $Ind$, Point $(x,y)$, Array $A$
 1: $P = Ind.\text{query}(x,y)$
 2: for each $r_i \in P$ do
 3:     if $r_i$ contains $p$ then
 4:         $A[i] = A[i] + 1$    % same function as in DrawPoints
 5:     end if
 6: end for
 7: return $A$
```

The only difference from the core approach in this procedure is checking if a fragment corresponding to a polygon with ID i falls on a boundary pixel. If the fragment is on a boundary pixel, then it is discarded since all points falling into that pixel have already been processed in the previous step. Otherwise, all points that fall into the pixel are *inside* this polygon. Thus, the count of points corresponding to the pixel (stored in F_{pt}) is added to the result $A[i]$ corresponding to polygon i. Note that when polygons intersect, fragments completely inside one polygon can be on the boundary of another polygon. The white point in Figure 7(a) is one such example: it lies inside P1, but on the boundary of P2. After all polygons are rendered, the array A stores the result of the query.

Procedure AccuratePolygons

```
Require: Polygon fragment $(x',y')$, Polygon ID $i$,
    Boundary FBO $F_b$, Point FBO $F_{pt}$, Array $A$
 1: if $F_b(x',y')$ is not a boundary then    % test FBO pixel color
 2:     $A[i] = A[i] + F_{pt}(x',y')$    % same function as in JoinPoint
 3: end if
 4: return $A$
```

5. RASTER JOIN EXTENSIONS

In this section, we discuss how our approach can be extended to handle different aggregations and filtering clauses, as well as data larger than GPU memory. We also describe how accurate ranges can be computed for the aggregate results. While the bounded approach provides guarantees with respect to the spatial region to take into account the uncertainties in the spatial data, providing bounds over the query result can be also useful for a more in-depth analysis.

Aggregates. Aggregate functions are categorized into distributive, algebraic and holistic [23]. *Distributive aggregates*, such as count, (weighted) sum, minimum and maximum, can be computed by dividing the input into disjoint sets, aggregating each set separately and then obtaining the final result by further aggregating the *partial aggregates*. *Algebraic aggregates* can be computed by combining a constant number of distributive aggregates, e.g., average is computed as $sum/count$. *Holistic aggregates*, such as median, cannot be computed by partitioning the input. In this paper we focus on *count* queries, while our current implementation also supports *sum* and *average*. However, note that our solutions apply to any distributive or algebraic (but not to holistic) aggregates in a straightforward manner. When computing the average, as with the count function, one of the color channels in the FBO (e.g., red) is used for counting the number of points while another channel (e.g., green) is used to sum the appropriate attribute. Similarly, instead of a single output array A, we use two arrays A_1 and A_2 to maintain the sum and count values when the polygons are processed (or when PIP tests are performed in the accurate variant). After all polygons are drawn in the final step of the algorithm, the query result is obtained by dividing the elements of the sum array A_1 by the elements of the count array A_2. Note that the data corresponding to the aggregated attribute is also transferred to the GPU.

Query Parameters. When query constraints are specified, they can also be tested on the GPU for each data point. The constraint test is performed in the vertex shader before transforming the point into screen space. The vertex shader discards the points that do not satisfy the constraint by positioning them outside the screen space so that they are clipped and they are not further processed in the fragment shader. We currently support the following constraints: $>, \geq, <, \leq$, and $=$. Note that the data corresponding to the attributes over which constraints are imposed is also transferred to the GPU.

Out-of-Core Processing. When the data points do not fit into GPU memory, they are split into batches that fit into the GPU. Then the query is executed on each of the batches and the results are combined. Thus, a given point data set has to be transferred to the GPU *exactly once*. Current generation GPUs have at least a few GB of memory that can easily fit several million polygons (depending on their size). Thus, we assume that the polygon data set fits into GPU memory and does not need to be transferred in batches.

Estimating the Result Range. We extend the bounded variant to compute a range for the aggregate result at each polygon. This is accomplished using the boundary pixels corresponding to the polygons as follows. Given a polygon i, let P_i^+ (P_i^-) be the set of pixels on its boundary that contain false positive (negative) results. Since only these pixels contribute to the approximation, counting the points contained in them provides loose bounds on the result range. In particular, the sums $\epsilon_i^+ = \sum_{(x,y)\in P_i^+} F_{pt}(x,y)$ and $\epsilon_i^- = \sum_{(x,y)\in P_i^-} F_{pt}(x,y)$ are used to compute the worst case lower and upper bounds respectively, resulting in the interval $[A[i]-\epsilon_i^+, A[i]+\epsilon_i^-]$ with 100% confidence.

Independent of the actual data distribution, since the region corresponding to a pixel covers a very small fraction of the spatial domain, we can reasonably assume that the spatial and value-domain distribution of the data points *within each pixel is uniform*. Under this assumption, we provide tighter expected intervals by computing the intersection between the boundary pixels and the polygons. In particular, let $f_i(x,y)$ denote the fraction area of the pixel (x,y) that intersects polygon i. Then the expected lower and upper bounds, respectively, are computed as before using:

$$\epsilon_i^+ = \sum_{(x,y)\in P_i^+} f_i(x,y) \times F_{pt}(x,y)$$

$$\epsilon_i^- = \sum_{(x,y)\in P_i^-} f_i(x,y) \times F_{pt}(x,y)$$

The corresponding intervals for *sum* and *average* can be computed in a similar fashion.

6. IMPLEMENTATION

In this section we first discuss the implementation of the raster join approaches using OpenGL[2]. We then briefly describe the GPU baseline used for the experiments.

6.1 OpenGL Implementation

We used C++ and OpenGL for the implementation. We make extensive use of the newer OpenGL features, such as *compute shaders* and *shader storage buffer objects* (SSBO). Compute shaders add the flexibility to perform general purpose computations (similar to cuda [40]) from within the OpenGL context while making the implementation (hardware) manufacturer independent. SSBOs enable shaders to write to external buffers (in addition to FBOs). For memory transfer between the CPU and GPU, we use the newly introduced *persistent mapping* that is part of OpenGL's techniques for Approaching Zero Driver Overhead.

[2]`https://github.com/vida-nyu/raster-join`

Polygon Triangulation. To triangulate polygons, we use the *clip2tri* library [12], which implements an efficient constrained Delaunay-based triangulation strategy. Triangulation is accomplished in parallel on the CPU and the set of triangles is transferred to the GPU during query execution.

Bounded Raster Join. Each of the two steps of the bounded approach, i.e., drawing points followed by drawing polygons, is composed of two shaders – a vertex shader and a fragment shader.

When drawing points, we transfer them to the GPU by copying them to a persistently mapped buffer that is used as a vertex buffer object (VBO). Each vertex shader instance takes a single data point from the VBO and transforms it into screen space (Line 4 in Procedure DrawPoints). The transformed point is processed in the fragment shader, which essentially updates the FBO at the given location (Line 5 in Procedure DrawPoints). Note that the memory for the FBO is allocated directly on the GPU.

When drawing polygons, the triangle coordinates are passed to the GPU as part of the VBO, and the vertex shader again transforms the endpoints to screen space. The rasterization is accomplished as part of the OpenGL pipeline, and each fragment resulting from this operation is processed in the fragment shader (Procedure DrawPolygons). Since the FBO from the previous step is already on the GPU, it is simply bound as a texture to the fragment shader, thus ensuring there is no unnecessary data transfer between the CPU and GPU. The result array A is maintained as an SSBO, and atomic operations are used when updating it. An advantage of SSBOs is that they allow processing intersecting polygons in a single pass thus avoiding unnecessary, additional processing.

Computing Result Ranges. Recall that the boundary pixels that contribute to both false positives and false negatives have to be identified to compute the result intervals. False positive pixels are identified by simply drawing the outline of a polygon. Identifying false negative pixels, however, is less straightforward. To accomplish this, we use *conservative rasterization* that allows rendering *all partially covered* pixels: the outline is drawn using conservative rasterization, and pixels that are not part of the regular rasterization form the false negative pixels. Conservative rasterization is supported via a custom OpenGL extension (*GL_NV_conservative_raster*) on Nvidia GPUs. On non-Nvidia GPUs, conservative rasterization can be accomplished by drawing a thicker outline and discarding pixels that do not intersect with the drawn polygon.

Deriving the tighter expected result interval requires computing the intersection between a pixel and its corresponding polygon. To do this efficiently, in the vertex shader, in addition to the transformed coordinates, we output the edge that is being drawn. We then use the Cohen-Sutherland line clipping algorithm [21] in the fragment shader to compute the fraction of the pixel that intersects with the polygon.

Accurate Raster Join. The first step of the accurate variant (drawing polygon boundaries) is again implemented as a vertex and fragment shader, where the boundaries are stored in an FBO. Conservative rasterization is used to ensure that no boundary pixels are missed. The second step (lines 4–9 in Procedure AccuratePoints) is implemented using compute shaders. As before, persistent mapped buffers are used for sharing data between the CPU and GPU. In addition to the data points, this step also requires a grid index on the query polygons. This index is created on-the-fly on the GPU as described next. The implementation of the third step (Procedure AccuratePolygons) is similar to that of the bounded raster join.

Polygon Index. We implemented a grid index that stores a polygon identifier in all the grid cells intersecting the bounding box of that polygon. The grid is represented as an array of linked lists, one for every grid cell. Each linked list stores all polygons that are assigned to a grid cell. We build the index on the GPU on-the-fly for every query in two passes. Given a grid resolution, the first pass computes the number of cells each polygon intersects with to estimate the size of the index. The second pass assigns the polygons to their corresponding cells. Since dynamic memory allocation is not supported on the GPU, we allocate the required memory directly on the GPU as a single contiguous region and implement a custom linked list. This memory is discarded after the query is executed.

Query Options. We chose to pass attributes as part of the VBO rather than regular buffers to allow for an efficient stratified access to the data when processing the vertex information in the vertex shader. However, this imposes the restriction that the size of each vertex must be fixed at compile time. As a result, in our implementation, we support constraints (which are conjunctions) over a maximum of 5 attributes. We can increase this constant up to the hardware limit in the shader code.

6.2 Baseline: Index Join Approach

As we show in the next section, using current GPU-based spatial join techniques [72] to execute the spatial aggregation is not very efficient mainly due to the materialization of the spatial join prior to the aggregation. To have a better baseline to compare our rasterization-based approaches, we extend existing index-based techniques to combine the spatial join operation with the aggregation so as not to explicitly materialize the join.

The key idea is to use an index on the polygon data to identify polygons on which to perform PIP tests. As with the accurate raster join, we use the grid index for this purpose. The query is executed as shown in Procedure IndexJoin. As with the raster join variants, the result array A is initialized to 0. Then the algorithm processes each point $p(x, y) \in D_{pt}$ using Procedure JoinPoint. Note that in the case of accurate raster join, this Procedure is executed only for a *subset* of points that are within a small distance from the polygon boundaries. After all points are processed, the array A contains the query result. Similar to the accurate raster join variant, the above query is implemented using a compute shader.

```
Procedure IndexJoin
Require: Polygon Index Ind, Points $D_{pt}$
 1: Initialize array A to 0
 2: for each $p = (x, y) \in D_{pt}$ do   % Can be run in parallel
 3:     execute JoinPoint(Ind, p)
 4: end for
 5: return A
```

7. EXPERIMENTAL EVALUATION

In this section, we first describe the experimental setup and then present a thorough experimental analysis that demonstrates the benefits of our raster join approach using two real-world data sets. Sections 7.3–7.5 discuss the scalability of the approaches when data fits in main memory. The goal of these experiments is threefold: 1) demonstrate the benefits of exploiting the parallelism of GPU; 2) verify the scalability of the approaches with increasing input sizes; and 3) show that the bounded variant outperforms all the other approaches in terms of query performance. Section 7.6 provides an in-depth analysis of the accuracy trade-offs of the bounded raster join approach. Finally, Section 7.7 presents experiments on data sizes larger than the main memory.

7.1 Experimental Setup

Hardware Configuration. The experiments were performed on a Windows *laptop* equipped with an Intel Core i7 Quad-Core processor, running at 2.8 GHz, 16 GB RAM and 1 TB SSD, and an NVIDIA GTX 1060 mobile GPU with 6 GB of GPU memory.

Table 1: Polygonal data sets and processing costs.

Region	Nr of polygons	Text file size	Triangulation	Index Creation		
				GPU	Multi-CPU	Single-CPU
NYC neighborhoods	260	877 KB	20 ms	10 ms	0.57 s	2.15 s
US counties	3945	20 MB	0.66 s	14 ms	23.34 s	37.05 s

Data Sets. We use two real-world data sets for our experiments: NYC yellow taxi and Twitter. The *NYC Taxi data set* contains details of over 868 million yellow taxi trips that occurred in NYC over a 5 year period (2009 - 2013). The data is available [60] as a collection of csv files, which when converted to binary occupy 72 GB. Each record corresponds to a taxi trip and consists of two spatial attributes (pickup and drop-off locations), two temporal attributes (pickup and drop-off times), as well as other attributes such as fare, tip, and number of passengers. The data is stored as columns on disk and the required columns (attributes) from the Taxi data set are loaded into main memory prior to performing any experiments. The *Twitter data set* was collected from Twitter's live public feed over a period of 5 years. It consists of over 2.29 billion geo-tagged tweets in the USA formatted as json records. Each tweet record has attributes corresponding to the location and time of the tweet, the actual text, and other information such as the favorite and retweet counts. When converted to binary, the data (excluding the text) occupies 69 GB on disk. Note that both data sets are *skewed*. Taxi trips are mostly concentrated in Lower Manhattan, Midtown, and airports, while there is a denser concentration of tweets around large cities. Sections 7.2–7.6 use the taxi data set. To perform the join queries, we also use two *polygon data sets*, summarized in Table 1. These data sets contain complex polygons commonly used by domain experts in their analyses.

Queries. For the experimental evaluation, we use *Count()* as the most representative aggregate function. Unless otherwise stated, no filtering is performed on additional attributes. To vary the input sizes, we first divided the data into roughly equal time intervals. The input size of a query was then increased by using data from increasing number of time intervals.

CPU Baseline: Index Join Approach. In addition to the GPU approaches, we also implemented the Index Join Approach on the CPU (described in Section 6). We further optimized the approach by assigning a polygon only to those grid cells that the actual geometry intersects. That is, we build the polygon index by first identifying all the cells intersecting with the MBR of the polygon, and then perform cell-polygon intersection tests. The algorithm was implemented in C++. We also implemented a parallel version with OpenMP, where we used *#pragma omp parallel for* to parallelize the PIP tests (Line 2 in Procedure IndexJoin). To avoid locking delays, each thread maintains the aggregates in a thread-local data structure, and all the aggregates are merged into a single result array in the end. The building of the polygon index was also parallelized (each polygon was processed independently).

Processing Polygon Data. Recall that both rasterization variants require the polygons to be triangulated (Section 3), while the accurate variant, and the CPU as well as GPU baselines require the creation of an index. For all the GPU approaches, our implementation computes the triangulation (in parallel on the CPU) and the indexes (on the GPU) on-the-fly for each query. On the other hand, since the CPU computation is much slower, the indexes were precomputed. Table 1 shows the time taken for each of these cases. To be consistent, we do not include the polygon processing time in the reported query execution time. However, note that even if

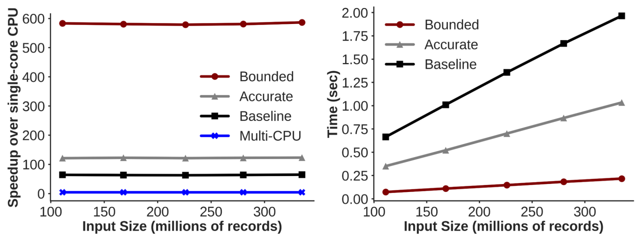


Figure 8: Scaling with increasing input sizes for Taxi ⋈ Neighborhood when the data fits in GPU memory. (Left) Speedup over single-CPU. (Right) Total query time. Bounded Raster Join has the best scalability as it eliminates all PIP tests. Accurate Raster Join performs fewer PIP tests than the Baseline.

these time were included, they would have a minimal effect on the performance of the GPU approaches.

Configuration Parameters. We limited the GPU memory usage to 3 GB, and the maximum FBO resolution to 8192×8192. Unless otherwise stated, the default ϵ-bound for NYC polygons is 10 m, and 1 km for US polygons. The resolution of the grid index for the neighborhood polygonal data set was set to 1024^2. For US counties, the GPU approaches use a grid index with 1024^2 cells, while the CPU baselines use 4096^2 cells. The index resolution for the GPU approaches was chosen based on the total time, including index creation time, since this was part of the query execution. The overall performance of using a 1024^2 index far outweighed that of using a 4096^2 index on the GPU. On the other hand, since we pre-computed the index for the CPU implementation, we chose the resolution that provided best query performance.

7.2 Choice of GPU Baseline

Table 2 compares our GPU index-based approach (Index Join) with state-of-the-art work on GPU join/aggregation[3] [72]. Our implementation performs 2-3× faster, mainly due to avoiding the materialization of the join result. We could not perform experiments with bigger input sizes as the provided code ran out of GPU memory. In the remaining experiments, given its clear advantages, we use our Index Join as the GPU baseline.

Table 2: Choice of GPU baseline.

Input Size (#points)	Zhang et al. [72] (total time - ms)	Index Join Baseline (total time - ms)
57,676,723	1060	344
111,659,661	1649	651
168,368,285	2129	999

7.3 Scalability with Points

In-Memory Performance. Figure 8 (left) plots the *speedup* of the parallel approaches (GPU and CPU) relative to our single-threaded CPU baseline when the point data sets fit in the GPU memory (i.e., the GPU memory holds the entire data set and data need not be transferred from the CPU to the GPU). Figure 8 (right) plots the *total time* against input size. The rasterization-based approaches are over two orders of magnitude faster than the single-core CPU implementation. Moreover, *the bounded variant is over 4 times faster than the accurate versions.* Given that our test system has a quad core processor (with a total of 8 threads), the multi-core CPU implementation provides a 5× speedup over the single-core CPU implementation. Thus, for the same laptop the GPU offers at least an order of magnitude more parallelism.

[3]Code made available by the authors at `http://geoteci.engr.ccny.cuny.edu/zs_supplement/zs_gbif.html`

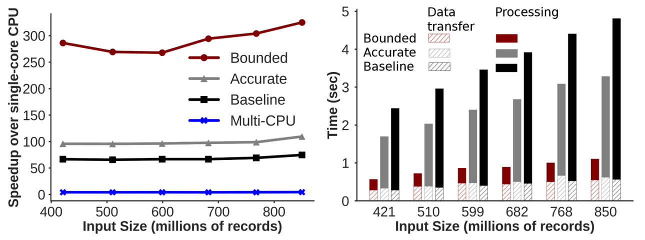


Figure 9: Scaling with increasing input sizes for Taxi ⋈ Neighborhood when the data does not fit in GPU memory. (Left) Speedup over single-CPU. (Right) Break down of the execution time. Note that the memory transfer between CPU and GPU dominates the execution time for the bounded approach.

Out-of-Core Performance. While significant speedups are obtained for in-memory queries, large speedups are achieved even when the data does not fit in GPU memory. As illustrated in Figure 9, the GPU approaches still obtain over an order of magnitude speedup over the CPU implementation while bounded raster join has a speedup of over two orders of magnitude. Note that, since the query times in this case are in milliseconds, the speedups are affected by even small fluctuations in the time (e.g., due to other windows background processes). The scalability observed for the different approaches is similar to that when data fits in GPU memory. By eliminating the costly polygon containment tests, the bounded approach significantly outperforms the other approaches. Even when the input size is around 868 million points, query execution takes only 1.1 seconds. The linear scaling also shows that the computation time is typically not affected by the number of additional passes required in the out-of-core scenario. To better understand the reduced speedup attained by the out-of-core technique compared to in-memory, we broke down the total execution time into the different components of query evaluation, i.e., processing and data transfer. The data transfer has a significant contribution in the overall time, especially in the case of bounded raster join where it dominates the execution time.

7.4 Scalability with Polygons

Generating Polygons. Since real-world polygonal data consists of a small number of polygons (100's to 1000's), we generated polygonal data to test the scalability of our approaches with the number of polygons. Our goal was to generate synthetic polygons with properties similar to the real ones. In particular, the generated data should contain a combination of simple as well as complex-shaped polygons (concave and arbitrary) with varying sizes. To accomplish this, we use the Voronoi diagram to generate a collection of convex polygons of varying sizes (based on the location of the points) and then ensure that concave and more complex shapes are generated by merging multiple adjacent convex polygons. More specifically, to generate n polygons, we first randomly generated $4n$ points within the rectangular extent of the data. We then computed the constrained Voronoi diagram over these points. This generates a collection of $4n$ convex polygons partitioning the rectangular region. Next, we randomly chose two neighboring polygons and merged them into a single polygon. We repeated this step until only n polygons remained.

Polygon Processing Costs. Figure 10 (left) shows the cost of processing the polygons (i.e., triangulation and index creation). As with the neighborhood data, we build a grid index with 1024^2 cells. Recall that the bounded variant requires only triangulation, the baseline only grid index creation, and the accurate variant both. As expected, triangulation time increases with an increasing number of polygons. When building the index, since the polygons partition

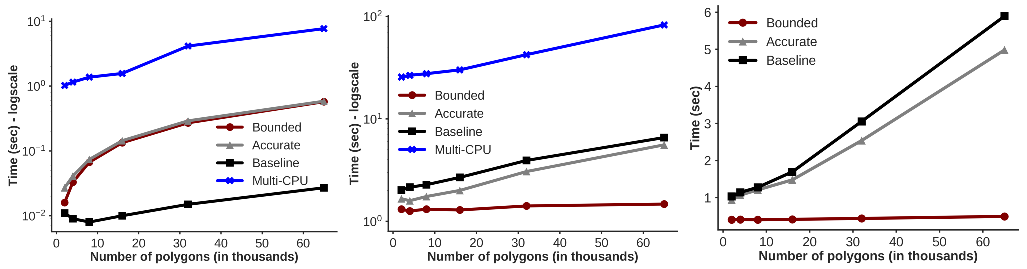


Figure 10: Scaling with polygons. (Left) Polygon processing costs. (Middle) Total query time when data does not fit in GPU memory. (Right) GPU processing time. Note that increasing the number of polygons has almost no effect on Bounded Raster Join.

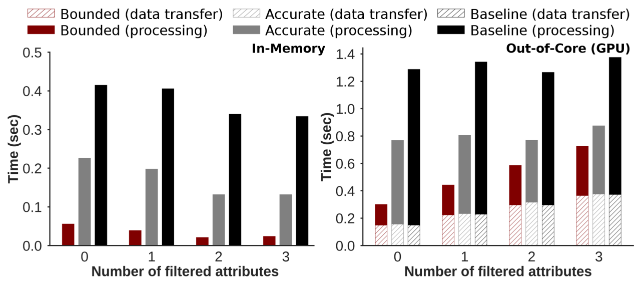


Figure 11: Scaling with number of attribute constraints.

the space, we touch all cells of the grid index one or more times depending on the polygons' structure. That explains the small drop at the beginning of the plot: even though the number of polygons increases, the sizes of their bounding boxes become smaller and each grid cell needs to be processed fewer times. As the number of polygons further increases, each grid cell intersects more polygon bounding boxes and needs to be processed more times thus increasing the building time. Note that even 64K polygons are processed in milliseconds. Thus, in dynamic settings where the polygons are not known a priori, they can be efficiently processed on-the-fly.

Performance. Figure 10 (middle) plots the total time when joining with 600 M points that do not fit in GPU memory. Figure 10 (right) focuses on the time spent on the GPU. The performance gap between the accurate variant and the baseline is much smaller in this scenario. Given the large number of polygons, the polygon outlines cover a significantly higher number of pixels, thus requiring more PIP tests to be performed. In the worst case, if the polygonal data set is very dense, the accurate variant degenerates into the baseline. Current generation GPUs can handle millions of polygons at fast frame rates. Since the bounded variant decouples the processing of points and polygons, increasing the number of polygons has almost no effect on the query time. The performance of the in-memory scenario is similar to that of Figure 10 (right), which shows the GPU processing times.

7.5 Adding Constraints

As mentioned in Section 1, users commonly change query parameters interactively as they explore a data set. To test the efficiency of our approach in this scenario, we incrementally apply constraints on different attributes of the taxi data. Figure 11 shows the total execution time of these queries for two input sizes, the first fitting in GPU memory (85 M points) and the second not (226 M points). The out-of-GPU-core breakdown shows that increasing the number of constraints increases the memory transfer time, as more data corresponding to the filtered attributes has to be transferred. However, the processing time is sometimes reduced with a higher number of constraints, because points that do not satisfy the constraints are discarded in the vertex shader, before performing any processing, thus reducing the amount of work done by the GPU.

7.6 Accuracy

Accuracy-Time Trade-Off. Figure 12(a) plots the trade-off between accuracy and query time for a query involving 600 M points (out-of-core). As the value of ϵ decreases, the number of rendering passes increases quadratically, thus the query time increases. After some point, the bounded variant becomes slower than the accurate. Analyzing this trade-off can help a query optimizer to automatically select a variant based on the value of ϵ.

Accuracy-ϵ-Bound Trade-Off. Figure 12(b) shows the effect of the specified ϵ-bound on the accuracy of the query results. The whiskers of the box plot represent the extent that is within 1.5 times the interquartile range of the 1st and 3rd quartiles, respectively. Decreasing the ϵ-bound decreases the error range converging towards the accurate values. The error range for the default value of $\epsilon = 10$ m is small, with a median of only about 0.15%. To show the actual differences in the aggregation results, we also plot the accurate vs. the approximate value for each of the polygons, using the coarsest bound ($\epsilon = 20$ m) in Figure 12(c). The fact that all the points lie very close to the diagonal indicates that even for a coarse bound a very good approximation is obtained. The bars in these plots denote the expected result interval that is computed (being very small, it is not clearly visible on the complete scatter plot). As seen in the highlighted region, our approach provides a tight interval even for a coarse ϵ value. The overhead of computing the intervals is negligible; computing them even for the costliest bound of $\epsilon = 1$ m required only an additional 140 ms. The accuracy trade-offs of the in-memory setup have a similar behavior.

Effect on Visualizations. Figure 6 shows side-by-side the visualizations computed through the bounded and accurate variations respectively. Note that the two visualizations are *perceptually* indistinguishable. The quality of the approximations can also be formally verified using *just-noticeable difference* (JND), a quantity used to measure how much a color should "increase" or "decrease" before humans can reliably detect the change [13]. In particular, sequential color maps used in the above visualizations can have a maximum of 9 perceivable color classes [25], resulting in a JND equal to $\frac{1}{9}$. A human can perceive the difference between the approximate and accurate visualizations, only when the difference between the corresponding normalized values is greater than $\frac{1}{9}$. However, the *maximum absolute error* between the normalized values even for the coarsest error bound ($\epsilon = 20$ m) is less than $0.002 \ll 0.11$, clearly showing that the difference from the visualization obtained using the bounded variation is not perceivable.

7.7 Performance on Disk-Resident Data

Figure 13 shows the performance of the different approaches when the data does not fit into the main memory of the system. We use the twitter data for this purpose, and aggregate over all counties in the USA. The increase in query time is primarily due to disk access times. Our implementation simply reads data from disk as

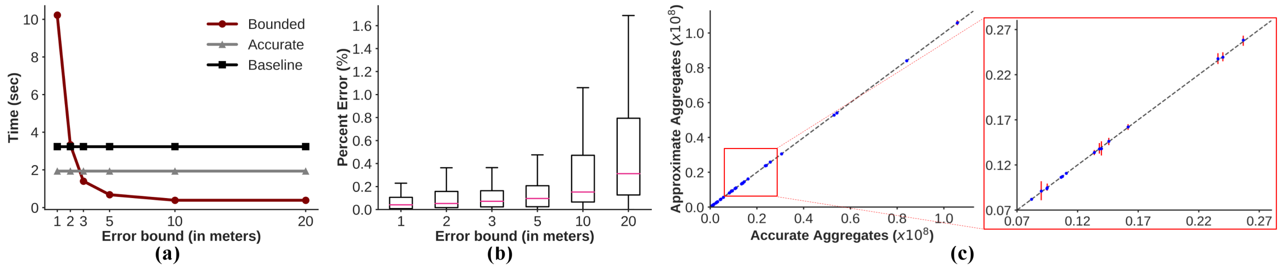


Figure 12: Accuracy analysis. (a) Accuracy-time trade-off. (b) Accuracy-ϵ-bound trade-off. The box plot shows the distribution of the percent error over the polygons for different ϵ-bounds. (c) The scatter plot shows, for each polygon, the accurate value against the approximate value for $\epsilon = 20$ m. The red error bars indicate the expected result intervals (see the enlarged highlighted region).

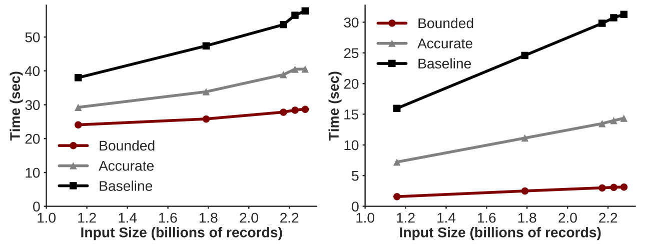


Figure 13: Scaling with points when data does not fit in main memory (Twitter ⋈ County). (Left) Total query time. (Right) Processing time excluding memory access time.

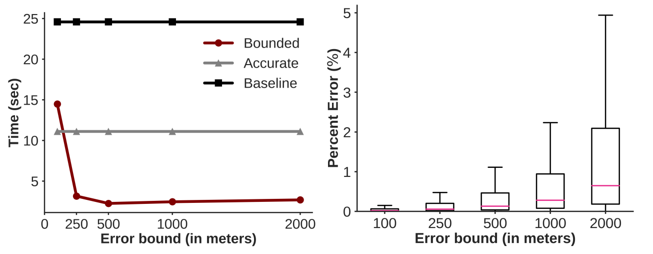


Figure 14: Accuracy-Time trade-off (left) and ϵ-bound trade-off (right) using the Twitter data.

and when required to transfer to the GPU, and does not apply any I/O optimizations such as parallel prefetching, which are beyond the scope of this work. Our focus was on the design of an efficient *operator* to perform spatial aggregation that can be integrated into any existing DBMS which efficiently handles such scenarios. In spite of this increase, the GPU approaches still provided over an order of magnitude speedup over the CPU baseline. When looking at only the processing times (time spent by the GPU), note that the timings are consistent with those when data fits in main memory. Even when executing a query with close to 2.3 billion points and over 3,900 polygons, the GPU processing time with Bounded is *less than 5 seconds*. Due to lack of space, we do not report other scalability results, but we note that the results were consistent with the main memory results with the addition of disk access time.

Since the counties data spans the whole USA, we chose $\epsilon = 1$ km for the above experiments. Figure 14 shows the accuracy-time as well as accuracy-ϵ-bound trade-off for 1.8 billion points. The scatter plot visualizing the accuracy of the query is similar to the taxi experiments, with the points falling close to the diagonal.

8. LIMITATIONS AND DISCUSSION

Worst-Case Scenario for the Accurate Approach. When the polygonal data set is very dense, every pixel of the FBO will fall at the boundary of some polygon and the accurate variant essentially becomes the baseline index-based approach. In fact, in this case, the accurate variant will take more time than the baseline, as it performs additional drawing (rendering) operations. This can also happen if the data is skewed such that all points fall close to the boundaries of the polygons.

Choice of Color Maps. We assume that continuous color maps are used for visualizations. In the case of categorical color maps, for values that fall around the boundary of two colors, even a minute error can completely change the color of the visualization.

Choosing Between the two Raster Variants. Setting a very small bound can result in the accurate variant becoming faster than the bounded variant of the raster join. This is because of the high number of renderings required to satisfy the input bound. We intend to add an estimate of the time required for the two variants, so that an optimizer can choose the best option based on the input query.

Performing Multiple Aggregates. Our current implementation performs only one aggregate per query. For multiple aggregates, multiple queries have to be issued. However, the implementation can be extended to support multiple aggregate functions by having multiple color attachments to the FBO. Similar to the multiple constraints scenario, this will increase the memory transfer time.

9. CONCLUSIONS AND FUTURE WORK

In this paper, we propose efficient algorithms to evaluate spatial aggregation queries over arbitrary polygons, which is an essential operation in visual analytics systems. By efficiently making use of the graphics rendering pipeline and trading-off accuracy for performance, our approach achieves real-time responses to these queries and takes into account dynamic updates to query parameters without requiring any memory-intensive pre-computation. In addition, the OpenGL implementation makes the technique portable and easy to incorporate as an operator in existing database systems.

By showcasing the utility of computer graphics techniques in the context of spatial data processing, we believe this work opens new opportunities to make use of advanced graphics techniques for database research, especially in the context of the ever increasing spatial/spatio-temporal data. For example, spatial joins between 3D data sets could greatly benefit from the use of ray casting and collision detection approaches. These approaches could also be applied to perform more complex spatio-temporal joins.

Acknowledgments. This work was supported in part by: the Moore-Sloan Data Science Environment at NYU; NASA; DOE; NSF awards CNS-1229185, CCF-1533564, CNS-1544753, CNS-1730396, and OAC 1640864; EU Horizon 2020, GA No 720270 (HBP SGA1); and the EU FP7 (ERC-2013-CoG), GA No 617508 (ViDa). J. Freire and C. T. Silva are partially supported by the DARPA MEMEX and D3M programs. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA.

10. REFERENCES

[1] S. Agarwal, B. Mozafari, A. Panda, H. Milner, S. Madden, and I. Stoica. Blinkdb: Queries with bounded errors and bounded response times on very large data. In *Proc. EuroSys*, pages 29–42, 2013.

[2] D. Aghajarian, S. Puri, and S. Prasad. Gcmf: An efficient end-to-end spatial join system over large polygonal datasets on gpgpu platform. In *Proc. SIGSPATIAL*, pages 18:1–18:10, 2016.

[3] A. Aji, F. Wang, H. Vo, R. Lee, Q. Liu, X. Zhang, and J. Saltz. Hadoop gis: A high performance spatial data warehousing system over mapreduce. *PVLDB*, 6(11):1009–1020, 2013.

[4] G. Andrienko, N. Andrienko, C. Hurter, S. Rinzivillo, and S. Wrobel. Scalable Analysis of Movement Data for Extracting and Exploring Significant Places. *IEEE TVCG*, 19(7):1078–1094, 2013.

[5] N. Ao, F. Zhang, D. Wu, D. S. Stones, G. Wang, X. Liu, J. Liu, and S. Lin. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *PVLDB*, 4(8):470–481, 2011.

[6] L. Battle, M. Stonebraker, and R. Chang. Dynamic reduction of query result sets for interactive visualizaton. In *Proc. IEEE Big Data*, pages 1–8, 2013.

[7] M. d. Berg, O. Cheong, M. v. Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 3rd edition, 2008.

[8] S. Breß, M. Heimel, N. Siegmund, L. Bellatreche, and G. Saake. Gpu-accelerated database systems: Survey and open challenges. In *Transactions on Large-Scale Data- and Knowledge-Centered Systems XV*, pages 1–35. Springer, 2014.

[9] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using r-trees. In *Proc. SIGMOD*, pages 237–246, 1993.

[10] Chicago Open Data. `https://data.cityofchicago.org/`.

[11] F. Chirigati, H. Doraiswamy, T. Damoulas, and J. Freire. Data polygamy: The many-many relationships among urban spatio-temporal data sets. In *Proc. SIGMOD*, pages 1011–1025, 2016.

[12] Using clipper and poly2tri together for robust triangulation. `https://github.com/raptor/clip2tri`, 2015.

[13] S. Coren, L. M. Ward, and J. T. Enns. *Sensation and Perception*. Wiley, 2003.

[14] O. Corporation. Oracle spatial and graph: Advanced data management. Technical report, Oracle, 2014.

[15] J. Davis. Ibms db2 spatial extender: Managing geo-spatial information with the dbms. *IBM White Paper*, 1998.

[16] H. Doraiswamy, N. Ferreira, T. Damoulas, J. Freire, and C. Silva. Using topological analysis to support event-guided exploration in urban data. *IEEE TVCG*, 20(12):2634–2643, 2014.

[17] H. Doraiswamy, H. T. Vo, C. T. Silva, and J. Freire. A gpu-based index to support interactive spatio-temporal queries over historical data. In *Proc. ICDE*, pages 1086–1097, May 2016.

[18] W. Fang, B. He, and Q. Luo. Database compression on graphics processors. *PVLDB*, 3(1-2):670–680, 2010.

[19] Y. Fang, M. Friedman, G. Nair, M. Rys, and A.-E. Schmid. Spatial indexing in microsoft sql server 2008. In *Proc. SIGMOD*, pages 1207–1216, 2008.

[20] N. Ferreira, M. Lage, H. Doraiswamy, H. Vo, L. Wilson, H. Werner, M. Park, and C. Silva. Urbane: A 3d framework to support data driven decision making in urban development. In *Proc. IEEE VAST*, pages 97–104, 2015.

[21] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1990.

[22] N. K. Govindaraju, B. Lloyd, W. Wang, M. Lin, and D. Manocha. Fast computation of database operations using graphics processors. In *Proc. SIGMOD*, pages 215–226, 2004.

[23] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–53, 1997.

[24] A. Guttman. R-trees: a dynamic index structure for spatial searching. *SIGMOD Rec.*, 14(2):47–57, June 1984.

[25] M. Harrower and C. A. Brewer. Colorbrewer.org: An online tool for selecting colour schemes for maps. *The Cartographic Journal*, 40(1):27–37, 2003.

[26] B. He, K. Yang, R. Fang, M. Lu, N. Govindaraju, Q. Luo, and P. Sander. Relational joins on graphics processors. In *Proc. SIGMOD*, pages 511–524, 2008.

[27] J.-F. Im, F. Villegas, and M. McGuffln. Visreduce: Fast and responsive incremental information visualization of large datasets. In *Proc. IEEE Big Data*, pages 25–32, 2013.

[28] A. Inselberg and B. Dimsdale. Parallel coordinates. In *Human-Machine Interactive Systems*, pages 199–233. Springer, 1991.

[29] E. H. Jacox and H. Samet. Spatial join techniques. *ACM Trans. Database Syst.*, 32(1), Mar. 2007.

[30] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *PVLDB*, 7(10):797–808, 2014.

[31] N. Kamat, P. Jayachandran, K. Tunga, and A. Nandi. Distributed and interactive cube exploration. In *Proc. ICDE*, pages 472–483, 2014.

[32] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. D. Nguyen, T. Kaldewey, V. W. Lee, S. A. Brandt, and P. Dubey. Fast: fast architecture sensitive tree search on modern cpus and gpus. In *Proc. SIGMOD*, pages 339–350, 2010.

[33] L. Lins, J. Klosowski, and C. Scheidegger. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE TVCG*, 19(12):2456–2465, Dec 2013.

[34] Z. Liu and J. Heer. The effects of interactive latency on exploratory visual analysis. *IEEE TVCG*, 20(12):2122–2131, 2014.

[35] Z. Liu, B. Jiang, and J. Heer. immens: Real-time visual querying of big data. *CGF*, 32, 2013.

[36] Mapd technology. `https://www.mapd.com/`.

[37] F. Miranda, H. Doraiswamy, M. Lage, K. Zhao, B. Gonalves, L. Wilson, M. Hsieh, , and C. Silva. Urban pulse: Capturing the rhythm of cities. *IEEE TVCG*, 23(1):791–800, 2017.

[38] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs. A sorting classification of parallel rendering. *IEEE CG&A*, 14(4):23–32, July 1994.

[39] *MySQL 5.0 Reference Manual (11.5 Extensions for Spatial Data)*, 2015.

[40] Nvidia. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[41] NYC Open Data. `http://data.ny.gov`.

[42] M. Olano and T. Greer. Triangle scan conversion using 2d homogeneous coordinates. In *Proc. ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS, pages 89–95, 1997.

[43] Yahoo labs. `https://webscope.sandbox.yahoo.com/`.

[44] T. Ortner, J. Sorger, H. Steinlechner, G. Hesina, H. Piringer, and E. Grller. Vis-a-ware: Integrating spatial and non-spatial visualization for visibility-aware urban planning. *IEEE TVCG*, 23(2):1139–1151, Feb 2017.

[45] C. Pahins, S. Stephens, C. Scheidegger, and J. Comba. Hashedcubes: Simple, low memory, real-time visual exploration of big data. *IEEE TVCG*, 23(1):671–680, 2017.

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

[47] J. Patel and D. DeWitt. Partition Based Spatial-Merge Join. In *Proc. SIGMOD*, pages 259–270, 1996.

[48] M. Pavlovic, T. Heinis, F. Tauheed, P. Karras, and A. Ailamaki. Transformers: Robust spatial joins on non-uniform data distributions. In *Proc. ICDE*, pages 673–684, May 2016.

[49] J. Pineda. A parallel algorithm for polygon rasterization. *SIGGRAPH Comput. Graph.*, 22(4):17–20, June 1988.

[50] PostGIS: Spatial and geographic objects for PostgreSQL. `http://postgis.net/`.

[51] R. Scheepens, N. Willems, H. van de Wetering, G. Andrienko, N. Andrienko, and J. van Wijk. Composite density maps for multivariate trajectories. *IEEE TVCG*, 17(12):2518–2527, 2011.

[52] J. R. Shewchuk. Delaunay refinement algorithms for triangular mesh generation. *Comput. Geom. Theory Appl.*, 22(1-3):21–74, May 2002.

[53] B. Shneiderman. The eyes have it: a task by data type taxonomy for information visualizations. In *Proc. IEEE Symposium on Visual Languages*, pages 336–343, 1996.

[54] D. Shreiner, G. Sellers, J. M. Kessenich, and B. M. Licea-Kane. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Version 4.3*. Addison-Wesley Professional, 8th edition, 2013.

[55] B. Simion, D. N. Ilha, A. D. Brown, and R. Johnson. The price of generality in spatial indexing. In *Proc. BigSpatial*, 2013.

[56] C. Stolte and P. Hanrahan. Polaris: A system for query, analysis and visualization of multi-dimensional relational databases. *IEEE Transactions on Visualization and Computer Graphics*, 8(1):52–65, 2002.

[57] C. Sun, D. Agrawal, and A. El Abbadi. Hardware acceleration for spatial selections and joins. In *Proc. SIGMOD*, pages 455–466, 2003.

[58] G.-D. Sun, Y.-C. Wu, R.-H. Liang, and S.-X. Liu. A Survey of Visual Analytics Techniques and Applications: State-of-the-Art Research and Future Challenges. *J. of Comp. Sci. and Tech.*, 28(5):852–867, 2013.

[59] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *Proc. EDBT*, 2002.

[60] TLC Trip Record Data. `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`, 2015.

[61] J. W. Tukey. *Exploratory Data Analysis*. Pearson, 1977.

[62] Twitter API. `https://dev.twitter.com/`.

[63] I. F. Vega Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Trans. on Knowl. and Data Eng.*, 17(2):271–286, Feb. 2005.

[64] M. Vermeij, W. Quak, M. Kersten, and N. Nes. Monetdb, a novel spatial columnstore dbms. In *FOSS4G*, 2008.

[65] L. Wang, R. Christensen, F. Li, and K. Yi. Spatial online sampling and aggregation. *PVLDB*, 9(3):84–95, 2015.

[66] H. Wickham. Bin-summarise-smooth: a framework for visualising large data. Technical report, had.co.nz, 2013.

[67] N. Willems, H. Van De Wetering, and J. J. Van Wijk. Visualization of vessel movements. *CGF*, 28(3):959–966, 2009.

[68] D. Xie, F. Li, B. Yao, G. Li, L. Zhou, and M. Guo. Simba: Efficient in-memory spatial analytics. In *Proc. SIGMOD*, pages 1071–1085, 2016.

[69] J. Zhang and S. You. Speeding up large-scale point-in-polygon test based spatial join on gpus. In *Proc. BigSpatial*, pages 23–32, 2012.

[70] J. Zhang, S. You, and L. Gruenwald. High-performance online spatial and temporal aggregations on multi-core cpus and many-core gpus. In *Proc. DOLAP*, pages 89–96, 2012.

[71] J. Zhang, S. You, and L. Gruenwald. High-performance spatial join processing on gpgpus with applications to large-scale taxi trip data. Technical report, The City College of New York, 2012.

[72] J. Zhang, S. You, and L. Gruenwald. Efficient parallel zonal statistics on large-scale global biodiversity data on gpus. In *Proc. BigSpatial*, pages 35–44, 2015.

[73] G. Zimbrao and J. M. d. Souza. A raster approximation for processing of spatial joins. In *Proc. VLDB*, pages 558–569, 1998.