Algorithms for Fundamental Spatial Aggregate Operations Over Regions

Mark McKenney & Brian Olsen
Southern Illinois University Edwardsville
Department of Computer Science
Edwardsville, IL USA
{marmcke, bolsen}@siue.edu

ABSTRACT

Aggregate operators are a useful class of operators in relational databases. In this paper, we examine spatial aggregate operators over regions. Spatial aggregates are defined to operate over a set of regions, and return a single region as a result. We systematically identify individual spatial aggregate operations by extending existing spatial operations into aggregate form. Semantic meaning for each operator is defined over a specified data model. Once defined, algorithms for computing spatial aggregates over regions are provided. We show that all proposed aggregates can be computed using a single algorithm. Furthermore, we provide serial and parallel algorithm constructions that can take advantage of vector co-processors, such as graphical processing units (GPUs), and that can be integrated into map/reduce queries to take advantage of big data-style clusters. Example queries and their results are provided.

Categories and Subject Descriptors

H.2.8 [**Information Systems Applications**]: Database Management Database Applications [Spatial databases and GIS]; I.3.5 [**Computing Methodologies**]: Computer Graphics Computational Geometry and Object Modeling [Geometric algorithms, languages, and systems]

General Terms

Algorithms, performance.

Keywords

Aggregate operations, spatial aggregates, vector algorithms.

1. INTRODUCTION

A wide range of scientific, business, and social applications require the management and analysis of data entities that contain a spatial component that may vary over time. The recognized value and utility of spatial and spatiotemporal data coupled with technologies that are encouraging the development of larger computational clusters and data centers, and an ever-growing number of small, location-enabled devices, has provided vast distributed data collection and analysis opportunities. Examples of spatial and spatiotemporal applications include land and resource management, weather monitoring and prediction, roadway traffic monitoring and forecasting, vehicle navigation systems, and disease pattern analysis.

Aggregate operations are a useful class of operations for conducting data analysis in databases. In relational databases, aggregate operators provide functionality to compute the sum, average, maximum value, minimum value, etc., of a set of single valued tuples of the appropriate data type [8, 9]. Traditionally, aggregate functions are defined such that a function is a mapping from a multiset of values to a single value; for example, the aggregation function *average* maps a multiset of numbers to a number representing their arithmetic mean. Recently, much attention has been given to spatial and spatiotemporal aggregates using a similar model (Section 2). Although there is significant activity in this area, much of the work focuses on either computing a numeric value from attributes attached to spatial data, or computing a numeric value from some spatial property of spatial data. An example of such a query is to compute the density of vegetation of a state by county, given a spatial databases containing county boundaries and regions indicating the density of various types of vegetation; this query is an interesting query that requires spatial computation to achieve, but eventually reduces to computing averages of vegetation density, which is a numerical attribute of a region, and then returning a number. We argue that this type of query is an aggregate query that requires spatial computation, as opposed to a *spatial aggregate query*; we propose the consideration of "pure" spatial aggregates, which operate on spatial data and return spatial data. In other words, a "pure" spatial aggregate does not transform into a traditional aggregate in order to return its final answer.

In this paper, we consider spatial aggregation operations that are fundamental in the sense that they are agnostic of descriptive attributes attached to spatial data objects, and provide aggregation computations based purely upon the spatial properties of objects. For example, given a set of regions representing hurricane paths, one may wish to know the region representing the area where most hurricanes have crossed. Such a query relies only on the spatial properties of the regions in the set, not their associated attributes. Another example might be to find the areas where only one disease commonly occurs in a database containing regions

ACM SIGSPATIAL GIS-BIGSPATIAL '13:,
November 05 - 08 2013, Orlando, FL, USA

http://dx.doi.org/10.1145/2534921.25349xx.

such that each region represents the area where a particular disease commonly occurs. Such a query could find pockets where a disease can be studied in isolation. Spatial systems exist that can achieve the results of the proposed queries, but rely on a collection of operations to be assembled by a knowledgeable user. We argue that each query can be represented in terms of a basic aggregate operation that can be easily integrated into SQL style syntax.

This paper provides four main contributions. (i) First, fundamental spatial aggregate operations for regions are systematically discovered based on traditional region operators. (ii) Formal definitions of the discovered aggregate operations are provided. (iii) Once defined, fundamental spatial aggregate operations can be implemented using a variety of algorithms. We provide an algorithm for implementing fundamental spatial aggregates that is suited to massive parallelism. The proposed algorithm utilizes a brute-force style line segment intersection algorithm, resulting in time complexity of $O((n+k)^2)$ for n input segments containing k intersections, but makes up for relative theoretical inefficiency by being suited to massive parallelization. The algorithm is suited to acceleration on graphic processing units or other massively parallel co-processors that can execute vectorized algorithms, and can be executed in a map/reduce framework over multiple nodes in order to take advantage of big data style systems. The single proposed algorithm computes all fundamental spatial aggregation operations identified in the first part of this paper. Finally, (iv) a prototype implementation of the proposed algorithm is used to compute spatial aggregate queries on a data set of Atlantic Hurricanes.

We focus our efforts on vectorized algorithms due to hardware trends that are favoring a resurgence in vector computations. The continuing trend of greater transistor density and more available transistors on each generation of chips has lead to multi-core processors becoming standard; indeed, even mobile phones have multi-core processors. High end desktop processors are now being shipped with on-chip graphics processing units (GPUs), allowing for vectorized co-processing of data on the CPU. Furthermore, standalone GPUs and other co-processors have revolutionized the use of vectorized algorithms for many applications. Thus, we provide vectorized algorithms to take advantage of this trend.

The paper is structured as follows. Section 2 reviews related literature relevant to spatial aggregates. Fundamental spatial aggregates are presented in Section 3 and formally defined in Section 4. Algorithms for computing spatial aggregates are presented in Section 5, and a prototype implementation is used to execute example queries in Section 6. In Section 7, we draw some conclusions.

2. RELATED WORK

Aggregate operations have a long history of use and study in databases (see the survey [11]). The development of spatial aggregates is more recent, but has similarly received much attention. A significant portion of the literature on spatial aggregates is devoted to mechanisms to support range queries, or box queries. Aggregate range queries perform some aggregate operation over spatial or spatiotemporal data that fall into a user specified area (the range or box), possibly over some specified time window [17, 10, 13]. Such aggregation mechanisms seem to stem from the support for range queries provided by spatial indexing methods such as R-trees and their variants [6, 1, 15], but much work has focused on developing data structures specific to aggregations. For these types of aggregates, the spatial portion of the aggregate tends to focus on discovering data objects that fall into the specified range, while the aggregate portion tends to be an aggregate operator over some attribute of the spatial objects that satisfy the range query. Thus, the aggregation portion of the operation behaves much like a traditional aggregate operator, and spatial operators are performed on spatial objects in isolation, rather than in aggregate, to compute an attribute value that is passed to the aggregate portion. Similarly, [11] provides a framework for spatial aggregation that uses spatial operators to compute regions of interest, then performs aggregate operations over the attribute values associated with those regions.

Our work differs from previous work in two respects. First, we seek to define spatial aggregates that aggregate on spatial data objects, and return an aggregated spatial data object, as opposed to aggregating over attribute values associated with spatial objects. For the scope of this paper, we will focus on aggregate operations over complex regions [14]. Second, we do not seek to develop new index-style structures for these queries, rather we focus on the development of algorithms that can scale to use parallelism. This is not to say that existing spatial indexing mechanisms cannot accelerate these queries, indeed they can be used effectively for this purpose; however, we provide parallel algorithms to be used on sets of regions that need exact computations for intersection tests, etc., once those sets have been identified through a spatial access method.

3. SPATIAL AGGREGATES

An aggregate operation is defined as a mapping from a multiset of values of a data type (possibly tuples) to a single value. Let $A = \{a_1, \ldots, a_n\}$, $B = \{b_1, \ldots, b_n\}$ be sets each respectively defining a type such that the sets contains all possible values of those types. Let $M = \{m \in A\}$ be a multiset of type A and $[M]$ be the set of a all valid multisets of type A. An aggregate function f is then a mapping from a member of $[M]$ to an element of B:

$$f : [M] \rightarrow B$$

The traditional aggregate functions implemented widely in databases utilize operations defined for the data type over which they are applied; for instance, *max* and *min* are typically defined using comparison operators applied to numbers and *sum* and *average* use arithmetic operators over numbers. These aggregates also return a type that is the same or similar to their input type (an average of integers typically returns a real number, so output types are not always identical as input types, but are generally similar). The exception to this trend is the *count* aggregate which simply sums the number of tuples it sees, effectively having an output data type that can be much different from its input data type; we denote this type of aggregate a *mixed class aggregate* as opposed to a *single class aggregate*. In Section 2, we noted that spatial aggregates in the literature have focused either on mixed class aggregates, where the input type is a spatial type and the output is a number, such as a count, or they have focused on applying spatial operations to spatial objects in order to generate a value of a traditional type, such as a number, and then a single class aggregate operation is performed over the numerical values. Although those types

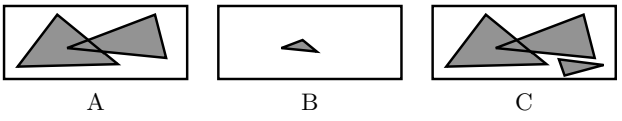

Figure 1: (A) two overlapping input regions. (B) the result of the intersection operator on the regions in (A). (C) The input regions from (A) with a third region.

of operations are interesting and important, in this paper, we focus on single class aggregates with spatial types.

Single class aggregates apply operators to a set of values that are defined over those values; for example, the sum aggregate applies the addition operator to numbers, the max operator applies a comparison operator to numbers, and the average operator combines the results of sum and count aggregates to compute an average. We approach the formalizations of spatial aggregates in much the same way by first examining spatial operators and then forming aggregates using them. In this paper, we focus on the spatial data type of complex regions [14] and define single class spatial aggregates over them.

Because our focus is single type aggregates over complex regions, we first look to spatial operators over regions that return regions. We use the notation A_R to indicate the type of spatial regions such that A_R is the set of all possible valid regions (including the empty region). Let $M_R = \{r \in A_R\}$ be a multiset of type A_R and $[M_R]$ be the set of all valid multisets of type A_R. We take operators from the Open Geospatial Consortium (OGC) simple feature specification [7] in an interest of generality with regards to spatial systems. The OGC specification lists the following spatial operations that take complex regions as input and return complex regions: the set operations of union, intersection, difference, and symmetric difference; buffer; and convex hull. We now consider each operation independently to determine if it may be used as a spatial aggregate.

The spatial set operators have been studied extensively in the literature and form the foundation of many spatial operators and analysis techniques. The *union* operator is an operator that naturally lends itself to an aggregate form with the function signature: $union : [M_R] \rightarrow R$. In fact, many spatial systems already provide a union aggregate operator. One reason the union aggregate is widely used is that it is lossless in nature; in other words, regardless of the topological relationship of input geometries, the output geometry will not be empty if any input geometry is not empty. In effect, the area covered by a region is preserved no matter what additional regions are added to the union.

The *intersection* operator does not lend itself to an aggregate operation in such a straightforward way as the union operator. In fact, the aggregate intersection of a set of regions does not immediately have a clear meaning. For example, if one defines an aggregate intersection of a set of regions as the region representing the area contained by *all* regions in the set, then one might expect the result in Figure 1B from the input in Figure 1A using the following definition:

DEFINITION 3.1. *Intersection aggregate:*
$intersect : [M_R] \rightarrow A_R$
$intersect(R) := \bigcap_{r \in R} r$

However, this definition does not contain the lossless property of union, or the property of traditional aggregates that non-empty output will be generated by a non-empty input. For example, the same definition of aggregate intersection applied to the set of regions in Figure 1C will return the empty region. The lack of a lossless property means that simply adding a single region to the aggregate can cause an empty output, a potentially drastic effect, but one that may be desired.

The lossy property of the intersect aggregate is potentially unwieldy leaving the opportunity for a lossless intersection aggregate. A candidate for such a lossless intersection is the map overlay operation [16], in which the boundaries of all input regions are maintained. Map overlay operations are typically more suited to management of thematic attributes of a set a regions, which trends towards mixed class aggregates. Furthermore, a map overlay is by definition a mixed class aggregate since it takes regions as input and produces either a single map object, or a collection of regions and keys to associate thematic information with each region. Spatial aggregates, as considered in this paper, require an operation similar to a map overlay, but without the need to manage attribute data. As we show in Section 5, the disregard for attribute values simplifies algorithms.

In order to achieve a lossless property in an intersection-oriented operator, we must effectively remove the ability of a single region to cause the output of an intersection operation to become empty; therefore, we introduce the concept of a *max intersection*. The max intersection computes the region representing the area covered by the most input regions. For example, the region shown in Figure 1B will be the result of the max intersection of either of the input region sets shown. Thus, the operator is lossless in the sense that once a region r is added as input to the aggregate, the subsequent addition of regions will not cause r to not contribute to the result.

The existence of a max intersection operator implies the existence of a min intersection operator, although a min intersection does not have a single intuitive definition. One way of interpreting the min intersection operator is to effectively reverse the definition of a max intersection: find the region representing area where the fewest number of input regions intersect. Clearly, the area where the fewest number of regions intersect is where no regions intersect, leading to an empty result unless regions exist in the input that occupy the entire embedding space for regions, a situation which is not always practical. Instead, we define the min intersection operator to return the area covered by x regions where is x is the smallest, non-zero number of intersecting regions in the input set. Furthermore, if we allow such aggregates, what stops us from wanting to know the region representing the area covered by exactly two input regions? Therefore, we define the *k-intersection* aggregate to find the region representing the area covered by exactly k input regions. We generalize the *k-intersection* aggregate further to the *k-range-intersection* aggregate, in which the k value is specified as a range. We have the following definitions:

DEFINITION 3.2. *Max intersection aggregate:*
$\text{max-intersection} : [M_R] \rightarrow A_R$
$\text{max-intersection}(R) := \bigcup_{s \in S} intersect(s)$
where $S = \{X \in W | \forall Y \in (W - X) : |X| \geq |Y|\}$
and $W = \{V \in 2^R | intersect(V) \neq \emptyset\}$

DEFINITION 3.3. *Min intersection aggregate:*
min-intersection : $[M_R] \rightarrow A_R$
min-intersection$(R) := \bigcup_{s \in S} intersect(s)$
where $S = \{X \in W | \forall Y \in (W - X) : |X| \leq |Y|\}$
and $W = \{V \in 2^R | intersect(V) \neq \emptyset\}$

DEFINITION 3.4. *K-range intersection aggregate:*
k-range-intersection : $[M_R] \times \mathbb{Z} \times \mathbb{Z} \rightarrow A_R$
k-range-intersection$(R, i, j) := \bigcup_{x \in X} x - \bigcup_{y \in Y} y$
where $X = \{intersect(S)$ *for* $S \in 2^R | i \leq |S| < j\}$
and $Y = \{intersect(T)$ *for* $T \in 2^R | |T| \geq j \wedge |T| < i\}$

The max intersection operation must first find the set W containing all multisets of regions in the power set of the input multiset R whose aggregate intersection is non-empty. We then assign the set S to contain the multisets from W with the largest cardinality. The aggregate intersection of each element of S is computed, and the union of the result of each aggregate intersection forms the final region of max intersection. The min intersection operation is defined similarly, but using a less than or equal operator.

The k-range intersection operation finds all multisets in the power set of the input multiset R whose cardinality falls into the specified range. For each of these multisets, the intersection aggregate is computed (possibly resulting in an empty set), and the results of the intersection aggregates are combined in a union operator. The same sequence of operations is performed on members of the power set of R with cardinalities outside the specified bounds. The difference of the two sets is computed to produce the final output set.

The symmetric difference operator can be conceptualized in aggregate form rather easily now that the k-intersection operator is defined. The symmetric difference of a set of regions consists of the union of the portions of those regions that do not intersect any other region. Effectively, the difference of a set of regions R is the k-intersection where $k = 1$:

DEFINITION 3.5. *Difference aggregate:*
difference : $[M_R] \rightarrow A_R$
difference(R) := k-range-intersection$(R, 1, 2)$

The traditional buffer operator takes a region and a distance d and returns a region representing the original region expanded by d. This operator does not readily translate to aggregate form since one must simply compute an aggregate union of a set of regions and then compute a traditional buffer operation on the result of the aggregate union; or, one must compute the buffer of the input regions and the compute an aggregate union of the buffer regions. In either case, the union is the aggregate operator, and the buffer operator only acts on single regions.

The convex hull operator, although not explicitly defined as an aggregate in [7], naturally lends itself to aggregate form in both concept and implementation. As we show later in this paper, the aggregates defined thus far can be computed using a common algorithmic framework. Convex hull algorithms are already well known and do not fit this framework, therefore, we will not consider them further in this paper.

4. DISCRETE DEFINITIONS

This section defines spatial aggregate operations in terms of a discrete data model that is suited to implementation. We begin by defining a data model, then show how the various operations can be computed using that data model.

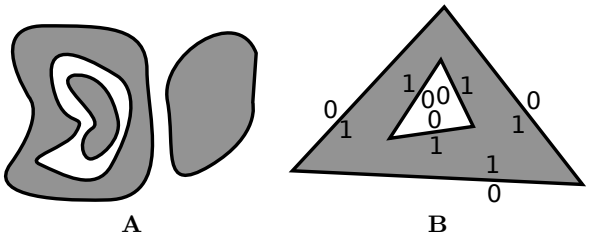

Figure 2: (A) an example of a complex region with multiple faces and a hole. (B) An example of a complex region defined using topo-segments. Each topo-segment has its associated *ia* and *ib* values respectively placed above and below.

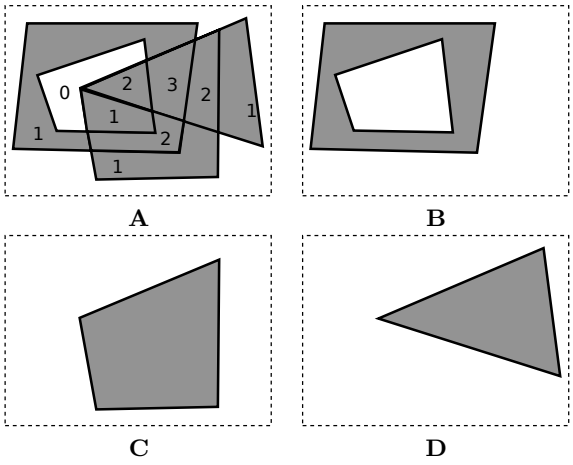

Figure 3: (A) an example of a count partition. Numbers indicate the amount of overlapping interiors of original regions used to build the partition. (B,C,D) The original regions used to compose the count partition in (A).

4.1 Discrete Data Model

As mentioned previously, we consider spatial aggregates over complex regions as defined in [14]. We assume that, for the purpose of implementation, a region is defined by its boundary, represented as a set of straight line segments that intersect only at endpoints, and that form cycles that satisfy the definition of complex regions (Figure 2). Since we are concerned with aggregate operations over sets of regions, it is useful to encode topology information about the region on each line segment. Specifically, we are concerned with the number of regions that a straight line segment bounds (recall the *k-range-intersection* operator which finds regions indicating where k region interiors intersect). Therefore, we define a *topo-segment* to be a straight line segment defined by two endpoints, p and q such that p is less than q, and two integers respectively representing the number of region interiors that lie above and below the segment (Figure 2B):

DEFINITION 4.1. *Topo-segment:*
topo-segment $= (p, q, ia, ib) : p = (x_1, y_1) \wedge q = (x_2, y_2) \wedge x_1, y_1, x_2, y_2 \in \mathbb{R} \wedge (x_1 < x_2 \vee (x_1 = x_2 \wedge y_1 < y_2)) \wedge ia, ib \in \mathbb{Z}$

We denote *ia* to be the *interior above* number and *ib* to be the *interior below* number corresponding to the number of region interiors respectively lying above and below the

segment, or to the left and right in the case of a vertical segment. For a single region, $ia, ib \in \{0, 1\}$ since a region has a single interior and exterior. Recall that the *k-range-intersection* operator cannot be computed using the classic intersection operation between regions, but must use an operations similar to a map overlay. *Spatial partitions* [5] are defined as partitions of the embedding euclidean space such that individual regions in the partition satisfy the definition of complex regions. Furthermore, the spatial partition induced in the plane by overlapping regions preserves the entire boundaries of all regions. For example, the spatial partition constructed from the regions in Figure 3B-D will have the structure shown Figure 3A. Spatial partitions are ideal for our situation since they preserve holes in regions. Effectively a spatial partition records which input regions overlap each region in the resulting partition. However, we are not concerned with keeping track of which regions overlap an area of the partition, rather we are interested in *how many* regions overlap each area in the partition. Thus, we will represent a collection of regions as a spatial partition defined by topo-segments such that the labels on the topo-segments will indicate the number of interiors of input regions that lie above and below each topo-segment in the partition. We will ignore information relating to which regions from the input set cover each area in the partition. We denote our variant of a spatial partition a *count partition*; a count partition is defined as a set of topo-segments that: (i) intersect only at endpoints, (ii) represent all boundaries of all original regions in the set of regions used to construct the partition, and (iii) are labeled to reflect the number of region interiors from the original set of regions used to construct the partition that each topo-segment bounds. Figure 3 depicts an example count partition.

4.2 Aggregate Operators Over Count Partitions

In Section 5, we provide an algorithm to construct a count partition from a set of input regions. For now, we assume the existence of the operator *count-partition* that takes an input consisting of a multiset of count partitions defined using topo-segments (recall a region is a special case of a count partition with all instances of $ia, ib \in \{0, 1\}$), and returns a count partition consisting of a set of topo-segments with appropriate labeling. Using the notation from Section 3, let A_C be the set of valid count partitions whose boundaries are defined by topo-segments, let $M_C = \{c \in A_C\}$ be a multiset of type A_C, and let $[M_C]$ be the set of all valid multisets of type A_C.

DEFINITION 4.2. count-partition: $[M_C] \rightarrow A_C$

The *k-range-intersection* can now be defined based upon the count-partition computed from $R \in [M_C]$. Recall that the k-range-intersection operator seeks to determine the region representing the area where x regions overlap such that $i <= x < j$ for input parameters i and j. Since the count-partition operator returns topo-segments with appropriate labels, the k-range-intersection reduces to a problem of keeping or discarding topo-segments with appropriate labels. The k-range-intersection operator must simply keep topo-segments that contain a single label that falls within the query range. If both labels of a segment fall within the query range, then the segment bounds two areas with the appropriate number of interiors, and does not form the boundary between the exterior and interior of the desired output region, but rather falls in the interior of the desired output region. For example, a k-range-intersection query for the regions shown in Figure 3B-D for a range indicating 2-3 interiors, we only keep segments with a label of 2 or 3 on one side, and any number other than 2 or 3 on the other. The segment with $ia = 3, ib = 2$ and the segment with $ib = 2, ia = 3$ will not be included. Let R be the count partition equivalents of the regions shown in Figure 3B-D: the result of *k-range-interection*$(R, 2, 4)$ is shown in Figure 4A.

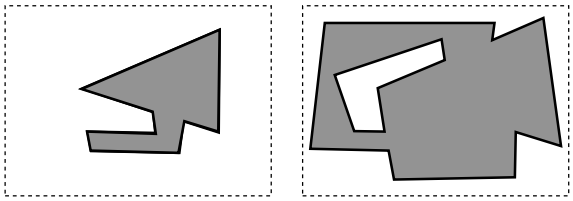

Figure 4: The result of the aggregate *k-range-operator* with $i = 2$ and $j = 4$ (A) and the result of the aggregate *union* operator (B) for an input set of regions containing the regions shown in Figure 3(B-D).

Formally, let $R \in [M_C]$:

DEFINITION 4.3. *k-range-intersection aggregate:*
k-range-intersection$(R, i, j) := \{s \in$ count-partition$(R) | s = (p, q, ia, ib) \wedge ((i \leq ia < j \wedge (ib < i \vee j \leq ib)) \vee (i \leq ib < j \wedge (ia < i \vee j \leq ia)))\}$

Similarly, the *max intersection* operator is computed as a *k-range-intersection* operator with the range identified as the maximum *ia* or *ib* number. The *min intersection* operator uses the minimum, non-zero *ia* or *ib* number as the range:

DEFINITION 4.4. *Max intersection aggregate:*
max-intersection$(R) :=$ k-range-intersection$(R, i, i+1) : i = max(\{ia\} \cup \{ib\} | (p, q, ia, ib) \in$ count-partition$(R))$

DEFINITION 4.5. *Min intersection aggregate:*
min-intersection$(R) :=$ k-range-intersection$(R, i, i+1) : i = min(\{ia\} \cup \{ib\} - \{0\} | (p, q, ia, ib) \in$ count-partition$(R))$

The *union* operator is computed from the result of a *count-partition* operator by keeping segments that form the outer boundary of a partition; ie., the topo-segments that have a label of 0 on one side of the topo-segment, and a non-zero label on the other. Figure 4B depicts the result of the aggregate union operator applied to the set of regions R consisting of the regions shown in Figure 3B-D.

DEFINITION 4.6. *Union aggregate:*
union$(R) := \{s \in$ count-partition$(R) | s = (p, q, ia, ib) \wedge ((ia = 0 \wedge ib \neq 0) \vee (ib = 0 \wedge ia \neq 0))\}$

The aggregate difference operator, as shown in Definition 3.5, is computed in terms of a *k-range-intersection* with $i = 1$ and $j = 2$.

5. ALGORITHMS

By defining aggregate operations in terms of collecting topo-segments with specific *ia* and *ib* values, we reduce the

problem of computing spatial aggregates to the problem of computing a count-partition consisting of topo-segments. In the following, we discuss methods to compute a count-partitions from an input set of regions using serial algorithms, vectorized parallel algorithms, and map/reduce algorithms. Regardless of the final algorithm used, the result of the algorithm must reflect the computation of two fundamental portions of the count-partition, structure and topology. The structural portion of the count-partition requires that topo-segments intersect only at endpoints, and that no duplicate topo-segments exist. Effectively, line segment intersection algorithms must be employed in some fashion. The topology portion of the count-partition refers to correct computation of the ia and ib labels. The structure and topology can be computed simultaneously or independently.

5.1 Serial Algorithms

Perhaps the most recognizable algorithmic tool for computing line segment intersections is the plane sweep algorithm [2, 3]. The plane sweep paradigm is applicable to many line segment intersection and topology problems, and can be effectively used to implement many spatial operations and topological predicates. The algorithm proceeds by constructing an imaginary *sweep line* that 'sweeps' over the embedding space in a direction perpendicular to its orientation. We will consider a vertical sweep line that sweeps across the plane in the x direction from left to right (ascending x values). In practice, the sweep line does not make a continuous sweep, rather its movement is defined by an ordering of the input line segments that the algorithm is operating over. Each input segment is represented twice, as two *half segments*, one used to order the segment by its left end point (the left halfsegment), and the other to order its segment by its right endpoint (the right halfsegment). In our case, halfsegments are created from topo-segments, preserving the ia and ib values. The halfsegments are then sorted such that the sweep line encounters them from left to right as it traverses the plane. When a left halfsegment is encountered, it is added to an *active list*: a list of halfsegments sorted vertically by their intersection point with the sweep line. When a right halfsegment is encountered, its associated left halfsegment is removed from the active list.

The first advantage of the plane sweep algorithm is that intersections between line segments are only computed between neighbors in the active list, leading to efficient time complexity, optimally $O(n \lg n + k)$ for n input line segments with k intersections. The second advantage is that when a halfsegment is added to the active list, it is possible to compute its topology values by examining the halfsegment that is ordered directly before (ie., below it) in the active list. Due to halfsegment ordering, when a halfsegment h is inserted into the active list, the halfsegment s directly below h will have its final topology values computed. Therefore, h's final topology values can be computed from s. For example, in Figure 5, when the sweep line (dashed) indicates that segment a should be added to the active list, segment b will be directly below it in the active list. The ia label of b will indicate that a single region interior lies above it; thus, a must lie inside that region interior. Since segment a has a single region interior (from its original input region) lying above it, and no region interiors lying below it, in the count partition it will have two region interiors lying above it, and one lying below it. This method generalizes to multiple input regions. For a halfsegment h newly added to the active list with halfsegment s directly below h in the active list, s's ia and ib values will reflect the total number of region interiors lying above and below it. Thus, the ia of s will indicate the number of region interiors that h lies in, and h's labels can be adjusted accordingly.

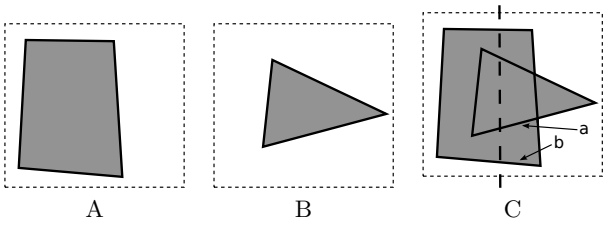

Figure 5: Two input regions (A and B), and both regions superimposed with a dashed sweep line (C).

The plane sweep algorithm suffers from two drawbacks. First, it is sensitive to numerical robustness errors due to rounding approximations used by floating point arithmetic and a heavy reliance on half segment ordering. Second, the plane sweep algorithm does not parallelize well. A more straightforward, although less computationally efficient, method of computing a count-partition is by using the brute-force approach. Essentially, each input line segment is tested for intersections against each other input line segment. We impose the following constraints on a brute-force line segment intersection computation. First, we require the input parameters to take a set of count partitions instead of a set of regions (a region is effectively a count partition containing a single region). Second, we expect the input to consist of correctly labeled topo-segments. Given these constraints, a brute force algorithm begins by first computing line segment intersections between all input line segments. If a topo-segment $s = (p, q, ia, ib)$ from one input count partition intersects a line segment from any other count partition at point k, s is split into $s_1 = (p, k, ia, ib), s_2 = (k, q, ia, ib)$ such that labeling is preserved. If a line segment from one input count partition overlaps a line segment from any other count partition, the overlapping portion is used to create two segments, each respectively preserving the labeling from a contributing count-partition. The result of this step of the algorithm is a set of correctly labeled count-partitions corresponding to the original input count partitions and with boundaries identical to the original input count-partitions, but possibly defined using more topo-segments (since some may have been broken at intersection points). Numerical robustness problems have a much smaller impact on this approach to line segment intersection than the plane sweep approach due to non-reliance on extensive use of comparison operators that the plane sweep algorithm requires for correctness; however, topology computations of segments are not inherently computed along with structure, as with the plane sweep.

Once line segment intersections between the input set of partitions are computed, we use a variant of the point-in polygon test to compute the topology of the final count-partition. The point-in-polygon test is a test that determines if a given point lies on the interior or exterior of a polygon. One method of computing a point-in-polygon test is known as the ray-shooting method: an imaginary ray is

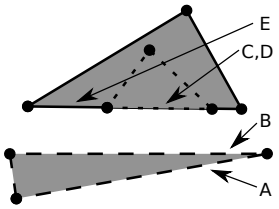

Figure 6: Three count partitions used as input to Algorithm 1. The lower boundary of the partition identified by a solid boundary is broken into three segments such two identical segments overlap at the lower boundary of the dotted triangle.

shot from a point and a count of the number of segments it crosses is maintained. If the ray crosses an odd number of segments, it must have originated within a polygon, otherwise it must have originated from outside of the polygon. Various special cases must be accounted for, namely: a point that lies on a segment, segments parallel to the ray, and rays passing through segment endpoints. We use a similar algorithm that we denote the *segment-in-region-count* operation. The segment-in-region-count operation takes advantage of topo-segment labels to determine the number of region interiors that a segment lies in. To perform a segment-in-region-count operation, we choose an endpoint p of a topo-segment s, and shoot an imaginary ray from p; for our example, we will shoot a ray downwards (negative y direction). Every other topo-segment is tested for intersection with the ray. For each topo-segment t discovered that (i) intersects the ray in t's interior, or on t's least-most endpoint, (ii) is not vertical, and (iii) does not belong to the same input region as s, we adjust the labels of s based on the labels of t. The labels of t will indicate the number of region interiors bounded on either side of t. Subtracting $ia - ib$ of t reveals the change in the number of region interiors that occurs when one crosses from below t to above t, we denote this change $\Delta_{interiors}$ of t. More formally:

$$\Delta_{interiors}(t) := ia - ib | t = (p, q, ia, ib)$$

Consider segment E (belonging to the solid boundary triangle) in Figure 6. When performing the segment-in-region-count test for E, segments A and B will be found that intersect the ray shot downwards from E's least end point in their interiors. The $\Delta_{interiors}$ of A will indicate a single region interior lies above A; therefore, we will add a 1 the labels of E. The $\Delta_{interiors}$ of B indicate the B is on the upper boundary of a region, effectively ending the extent of a region exterior that extends upwards from some topo-segment below it (in this case A); thus, the labels of E are subtracted by the amount of $\Delta_{interiors}$ of B. The result is that E's labels will indicate a single region lies above it.

One might notice that a problem arises when performing this algorithm on segment B. B will find A below it, and adjust its labels accordingly. Thus, B's labels will be $ia = 0 + \Delta_{interiors}(a) = 1$ and $ib = 1 + \Delta_{interiors}(a) = 2$, clearly incorrect. The problem is that B does not recognize that the interior A bounds is the same interior B is already labeled as bounding. This situation only occurs for segments who bound a region interior from above. Effectively, any line segment from a count partition with $\Delta_{interiors} <= 0$ bounds a region interior or region interiors from above. To remedy this situation, we simply reduce the labels for any input segment s with $\Delta_{interiors}(s) < 0$ by the $\Delta_{interiors}(s)$. For our example, segment B's labels are initialized to $ia = ia + ia - ib = -1$ and $ib = ib + ia - ib = 0$. When the segments that lie below and bound the same region interior as such a segment, in this case A, are encountered, the labels will be adjusted correctly. A second special case occurs when the ia and ib values are identical; in this case, both values are set to 0. If a segment has identical ia and ib values, it will form the upper boundary of some region, and thus, its ia and ib values will be adjusted appropriately by the segments that lie below it. Therefore the segment-in-region-count operation will initialize each input segment s using the following:

$$\begin{array}{l} initialize(s = (p, q, ia, ib)) : \\ \quad x \leftarrow ia - ib \\ \quad if(\ \Delta_{interiors}(s) < 0\ ) : \\ \quad\quad ia \leftarrow ia + x \\ \quad\quad ib \leftarrow ib + x \\ \quad else\ if(\ ia == ib\ ) : \\ \quad\quad ia \leftarrow ib \leftarrow 0 \end{array}$$

One special case must be handled in the algorithm described thus far. Recall that overlapping portions of input segments are represented in each partition in the input set, resulting the possibilty of equivalent segments arising in the segment-in-region-count operation. Assume that the input set of count-partitions consists of three partitions respectively representing the dashed triangle, the dotted triangle, and the solid boundary triangle whose lower segment overlaps the lower segment of the dotted triangle in Figure 6. At the start of the segment-in-region-count operation, segments C and D are identical, and are both contained in the input to the operation. Segments C and D can influence each others ia and ib values independently of each other. Thus, if segment $s = (p, q, ia_s, ib_s)$ encounters segment $t = (p, q, ia_t, ib_t)$ in the segment-in-region-count operation such that the segments are identical (although their labeling may not be), then $ia_s \leftarrow ia_t \leftarrow (ia_s + ia_t)$ and $ib_s \leftarrow ib_t \leftarrow (ib_s + ib_t)$.

Once all segments have been observed in relation to each input segment, the ia and ib labels for all segments will reflect the number of region interiors that lie on each side of each segment, thus, the final count-partition is complete. The algorithm is provided in Algorithm 1. Consider the scene in Figure 6 as an example. The scene depicts three count partitions differentiated by their boundaries: solid, dashed, and dotted. Assume all segment intersections are computed; thus, the lower boundary of the solid line triangle is broken into three segments, one of which is identical to the lower boundary segment of the dotted line triangle. Assume each boundary segment is represented as a topo-segment appropriately labeled for each original count partition, ie., the labels are all either 1 or 0. In line 1, topo-segments are initialized. When topo-segment C (belonging to the solid line triangle) is chosen in line 3 of the algorithm, then the inner loop (lines 4-12) identifies segments A and B that lie below it, and segment D (belonging to the dotted triangle) to be equal to it. Therefore, the labels belonging to $C = (p_c, q_c, ia_c, ib_c)$ are adjusted as follows:

$$\begin{array}{l} ia_c \leftarrow ia_c + \Delta_{interiors}(A) + \Delta_{interiors}(B) + ia_D \\ ia_c \leftarrow 1 + 1 + (-1) + 1 = 2 \\ ib_c \leftarrow ib_c + \Delta_{interiors}(A) + \Delta_{interiors}(B) + ib_D \\ ib_c \leftarrow 0 + 1 + (-1) + 0 = 0 \end{array}$$

Algorithm 1: Using the *segment-in-region-count* algorithm, compute the count-partition induced by set of input count-partitions.

Input: a vector T of topo-segments representing input count-partitions with correct initial labeling. Duplicate topo-segments may exist, but otherwise topo-segments only intersect at endpoints.

Output: A vector F of topo-segments representing the count-partition induced by the topo-segments in T

```
1  initialize(t ∈ T);
2  for i ∈ {0, ..., length(T) − 1} do
3  |  s = (p_s, q_s, ia_s, ib_s) ← T[i];
4  |  for j ∈ {i + 1, ..., length(T)} do
5  |  |  t = (p_t, q_t, ia_t, ib_t) ← T[j];
6  |  |  if s = t then
7  |  |  |  ia_s ← ia_s + ia_t;
8  |  |  |  ib_s ← ib_s + ib_t;
9  |  |  else if t is not vertical, t is below s, a ray shot
   |  |  downwards from p_s intersects t at some point
   |  |  other than q_t then
10 |  |  |  ia_s ← ia_s + Δ_interiors(t);
11 |  |  |  ib_s ← ib_s + Δ_interiors(t);
12 |  |  end
13 |  end
14 |  insert s into F;
15 end
```

The computation results in C reflecting the fact that two region interiors lie above it, and no region interiors lie directly below it.

5.2 Vectorized Parallel Algorithms

Plane sweep algorithms do not readily translate to vectorized parallel algorithms since the sweep line must visit segments in a particular order and makes assumptions based on this ordering that are needed for algorithm correctness. In contrast, the brute force algorithm has no ordering requirements. Algorithm 1 is defined in such a way the final labels for a segment are constructed by summing label values for segments, which can occur in any order. Because the input to the algorithm is a vector of segments, the algorithm is easily vectorized. In our implementation, we generate a thread for each input segment. Each one of those threads then examines the necessary other input segments, and adjusts labels accordingly. In effect, the outer loop of Algorithm 1 is replaced by thread generation, and the inner loop remains. This algorithm is ideally suited for execution of accelerators such as graphics cards.

Thus, the final vectorized algorithm has time complexity $O((n+k)^2)$ for n input segments containing k intersections. Because line segment intersections must first be computed in order for Algorithm 1 to execute correctly, Algorithm 1 computes a nested loop over $(n+k)$ line segments.

Although the vectorized algorithm remains polynomial in time complexity, various optimization measures can be employed to reduce the number of computations needed. For example, input segments can be sorted so that threads can end early and not examine all possible input segments, and spatial decomposition techniques can be used to segment the input space. Furthermore, the algorithm does not necessarily need to be run over the entire input. Spatial decomposition techniques based on bounding rectangle approximations, such as R-trees, can identify sets of input count partitions that are disjoint, thus they do not need to be computed against each other.

Algorithm 2: Map/reduce construction for computing count-partitions on a supporting file system.

Input: Each map function receives the file name of a file storing a count partition and the vector of topo-segments c stored in the file. Each reduce function takes a key emitted from a map function, and an array of count-partitions emitted from map functions with identical keys. (In this case, all emitted keys are 0.

Output: The reduce functions will result in a vector of topo-segments representing the final count-partition

```
1  map(fileName, c){
2    emit( 0, c);
3  }
4  reduce( key, partitions[ ] ){
5    find segment intersections and construct
     non-intersecting segments for all segments in
     partitions;
6    P ← the count-partition resulting from Algorithm 1;
7    return P;
8  }
```

5.3 Map/Reduce Construction

Because Algorithm 1 produces correct results for inputs consisting of count partitions, constructing a basic map/reduce algorithm to compute count partitions is trivial. Let R be a set of count partitions with appropriate labels stored in a file system supporting a map/reduce framework. A map function simply emits the partition. If the partition is not stored with labels, the map function can compute the correct labels and emit the partition. The reduce function will take a list of count partitions, combine them into a result partition, and emit the result (Algorithm 2).

Algorithm 2 contains various opportunities for optimization. Spatial approximation and decomposition techniques can be used in the reduce function to reduce the number of computations that must be computed. For example, bounding rectangle approximations can easily determine if two input count-partitions are disjoint in certain configurations, requiring no further computation. If a *k-range-intersection* is being computed, topo-segments with both labels greater than the upper end of the range can be excluded from further consideration since they lie on the interior of an area that contains more regions than is desired in the result (that area will maintain its outer boundary). For example, if the count-partition in Figure 3A arises for a *k-range-intersection* operation with range $i = 1, j = 2$, the topo-segments with both $ia, ib \geq 2$ do not have to be returned.

6. QUERIES

We have implemented a prototype library that computes aggregate *k-range-intersection*, *union*, and *difference* operations over an input set. In this section, we describe the

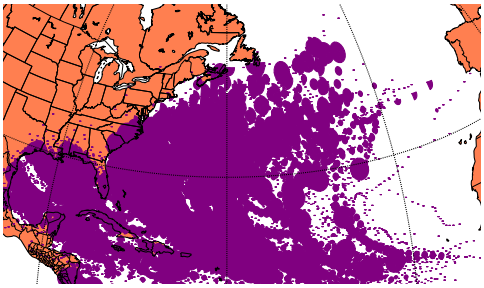

Figure 7: Snapshots of hurricanes from 1988-2009 shown at discrete time instants (purple).

implementation and provide example query results.

Our prototype is written in Python and computes the brute force style algorithm for input regions. As an illustration of the utility of spatial aggregates, and an exposition spatiotemporal data as an area where these operations provide interesting analysis opportunities, we will perform queries over a dataset consisting of hurricanes originating in the Atlantic ocean and Gulf of Mexico. The data set we use is the Tropical Cyclone Extended Best Track Dataset [4]. (EBTD). The EBTD contains snapshots in time of tropical storms at discrete time points. For each storm snapshot, the data set records information about the state of the storm at that particular time instant. The data set used in our example contains storms from 1988 until 2009. For our purposes, we use the storm's center, and the radius of hurricane force winds that extend out from the storm center in each of the quadrants: northeast, southeast, southwest, and northwest. Figure 7 shows the raw hurricane data plotted on a map; because each snapshot describes a hurricane at a discrete point in time, the faster moving hurricanes appear as trails of individual polygons, while slower moving hurricanes blend together. In total, 144 hurricanes exist in the data set.

The EBTD data set provides opportunities for interesting queries. For example:

1. A shipping company may want to know the portions of the Atlantic and Gulf of Mexico that experience the most hurricanes so ships can avoid these areas.
2. A person considering a move to a coastal area may want to know which areas have historically experienced few hurricanes.
3. A meteorologist might want to find the total area covered by all hurricanes over a time period.

These queries are problematic because they assume the connotative, spatiotemporal property of hurricanes that they move continuously over time, which is not reflected in the data. Figure 7 shows that hurricanes frequently move fast enough to leave areas between snapshots that the actual hurricane traveled over, but are not explicitly represented. Based on this data set, mistakingly thinking a shipping lane crosses no hurricanes when it simply travels between snapshots (and actually crosses a hurricane path) could be potentially hazardous. Therefore, our first step to completing these queries is to convert the data to a form that aligns more closely with reality. Using the algorithm in [12], we convert a hurricane represented as a series of regions at discrete points in time into a *moving region* with infinite temporal granularity. Essentially, the region snapshots are plotted in 3D space

Figure 8: The result of the *k-max-intersection* query. Result region boundaries are purple.

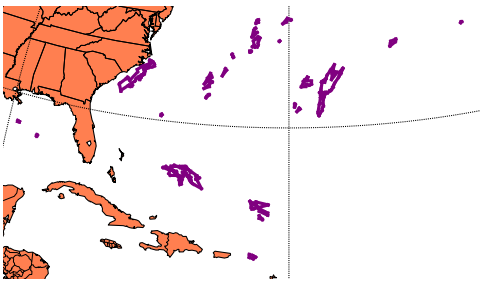

Figure 9: The result of the *k-range-intersection*$(8, 12)$ query. Result region boundaries are purple.

where the third dimension is time, and the region's movements between adjacent snapshots are interpolated; this process forms an approximation of the hurricane across a time interval. Once a 3D moving region is constructed, we must then compute the region representing the path that the hurricane has covered. The moving region is projected out of the time dimension and into 2D space, resulting in a region representing area covered by hurricane force winds for a particular hurricane. This process is completed for each hurricane, leaving a collection *coverage regions*.

Spatial aggregate operators can compute the queries above on the set of computed coverage regions. For example, a *k-max-intersection* query reveals that in our dataset, the areas covered by the highest number of hurricanes are covered by 11 hurricanes (the area outlined in purple in Figure 8). Interestingly, areas where many hurricanes overlap tend to be rather small, and politely remain over water as opposed to land. In fact, we must adjust a range query to include 8 to 11 overlapping hurricanes before any overlap occurs in the Gulf Mexico. Figure 9 depicts the *k-range-intersection*$(8, 12)$ query result, indicating a small area of coverage in the Gulf of Mexico and an area of coverage near North Carolina. These operators apply to both Query 1 and Query 2 above.

The person who is interested in Query 2 is interested in areas that have historically had very few hurricanes. The aggregate difference operator returns the areas that have only been covered by a single hurricane in the data set. Figure 10 shows a portion of the US Gulf Coast with the result of the difference operator over the hurricane data set. The purple lines form the boundaries of regions that are covered by only one hurricane in the data set. Areas north of this region have been covered by no hurricanes, areas south of the region have been covered by more than one hurricane. Although past events are no guarantee of future hurricane activity, the person posing Query 2 could use this result as part of their risk assessment.

Query 3 requires a *union* aggregate, which we implement as a *k-range-intersection*$(1, 12)$ query. No figure is provided

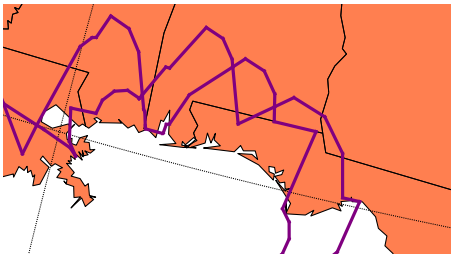

Figure 10: The result of the *difference* query.

for space considerations.

In addition to the prototype written in Python, we implemented the vectorized algorithm in Cuda, and a plane sweep version of the algorithm in C++ using an AVL tree library written in C. The GPU algorithm employs a simple spatial decomposition scheme in which input space is divided into vertical strips. The vectorized algorithm is effectively run over each strip. We collected running times for the implementations on two GPUs and two systems with differing CPUs. The faster GPU is a newer model, and the slower GPU is roughly three years old at the time of writing. Similarly, the faster CPU system is recent system, and slower system is roughly 5 years old. Times are computed with compiler optimizations turned on and are an average of three runs. The results are summarized in Table 1. The input data sets consist of the hurricane data set in the first data column, and data sets consisting of randomly generated regions in the remaining columns. The generated data sets contain large numbers of line segment intersections; such configurations were chosen to deliberately test the vectorized algorithm's performance in scenarios that favor the better theoretical time complexity of the the plane sweep algorithm. In every reported case, the vectorized algorithm is faster despite its inferior theoretical time complexity.

Table 1: Running times for the vectorized and serial algorithms. The number of input line segments and line segment intersections are reported in the first two lines. Running times are in seconds. GPU 1: GTX780. GPU2: GTX460. CPU 1: Core i7 2GHz. CPU 2: Core2 Duo 2.3GHz.

Input	7043	260924	328991	371651	381391
Intersections	9724	142118	170392	201066	188876
GPU 1	0.03	0.71	0.87	0.99	1.01
GPU 2	0.03	0.76	1.01	1.2	1.21
CPU 1	0.05	1.2	1.46	1.76	1.77
CPU 2	0.06	1.92	2.44	2.85	2.78

7. CONCLUSION

Spatial aggregate queries are a useful class of operations for spatial data analysis. In this paper, we identified and defined new spatial aggregates, and provided algorithms for their construction. Running times for algorithms were provided, showing the promise of vectorized spatial algorithms on GPUs. Furthermore, we demonstrated the utility of spatial aggregates through the creation of a prototype implementation and the execution of spatial aggregates over a data set consisting of Atlantic hurricanes. This example demonstrates the usefulness of spatial aggregates in spatiotemporal systems in addition to purely spatial systems.

8. REFERENCES

[1] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The r*-tree: an efficient and robust access method for points and rectangles. *SIGMOD Rec.*, 19(2):322–331, May 1990.

[2] J. Bentley and T. Ottmann. Algorithms for Reporting and Counting Geometric Intersections. *IEEE Trans. on Computers*, C-28:643–647, 1979.

[3] B. Chazelle and H. Edelsbrunner. An Optimal Algorithm for Intersecting Line Segments in the Plane. *J. ACM*, 39(1):1–54, 1992.

[4] J. Demuth, M. DeMaria, and J. A. Knaff. Improvement of advanced microwave sounder unit tropical cyclone intensity and size estimation algorithms. *J. Appl. Meteor.*, 45:1573–1581, 2006.

[5] M. Erwig and M. Schneider. Partition and Conquer. In *3rd Int. Conf. on Spatial Information Theory (COSIT)*, pages 389–408. Springer-Verlag, 1997.

[6] A. Guttman. R-trees: a Dynamic Index Structure for Spatial Searching. In *SIGMOD '84: Proceedings of the International Conference on Management of Data*, pages 47–57, New York, NY, USA, 1984. ACM.

[7] OpenGIS Implementation Standard for Geographic information - Simple feature access - Part 2: SQL option, 2010.

[8] ISO/IEC 9075:1992 - Database Language - SQL.

[9] A. Klug. Equivalence of relational algebra and relational calculus query languages having aggregate functions. *J. ACM*, 29(3):699–717, July 1982.

[10] I. Lazaridis and S. Mehrotra. Progressive approximate aggregate queries with a multi-resolution tree structure. *SIGMOD Rec.*, 30(2):401–412, May 2001.

[11] I. Lopez, R. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: a survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.

[12] M. McKenney and J. Webb. Extracting moving regions from spatial data. In *Proceedings of the 18th SIGSPATIAL Intl. Conf. on Advances in Geographic Information Systems*, pages 438–441, 2010.

[13] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient olap operations in spatial data warehouses. In *Proceedings of the International Symposium on Advances in Spatial and Temporal Databases*, 2001.

[14] M. Schneider and T. Behr. Topological Relationships between Complex Spatial Objects. *ACM Trans. on Database Systems (TODS)*, 31(1):39–81, 2006.

[15] T. Sellis, N. Roussopoulos, and C. Faloutsos. The r+-tree: A dynamic index for multi-dimensional objects. In *Proceedings of the International Conference on Very Large Data Bases*, 1987.

[16] C. D. Tomlin. *Geographic Information Systems and Cartographic Modelling*. Prentice-Hall, 1990.

[17] D. Zhang and V. J. Tsotras. Improving min/max aggregation over spatial objects. In *Proceedings of the 9th ACM international symposium on Advances in geographic information systems*, GIS '01, 2001.