Noname manuscript No.
(will be inserted by the editor)

Sequenced Spatiotemporal Aggregation for Coarse Query Granularities

Igor Timko · Michael Böhlen · Johann Gamper

Received: date / Accepted: date

Abstract Sequenced spatiotemporal aggregation (SSTA) is an important query for many applications of spatiotemporal databases, such as traffic analysis. Conceptually, an SSTA query returns one aggregate value for each individual spatiotemporal granule. While the data is typically recorded at a fine granularity, at query time a coarser granularity is common. This calls for efficient evaluation strategies that are granularity aware.

In this paper we formally define an SSTA operator that includes a data-to-query granularity conversion. Based on a discrete time model and a discrete 1.5 dimensional space model, we generalize the concept of time constant intervals to constant rectangles, which represent maximal rectangles in the spatiotemporal domain over which an aggregation result is constant. We propose an efficient evaluation algorithm for SSTA queries that takes advantage of a coarse query granularity. The algorithm is based on the plane sweep paradigm, and we propose a granularity aware event point schedule, termed *gaEPS*, and a granularity aware sweep line status, termed *gaSLS*. These data structures store space and time points from the input relation in a compressed form using a minimal set of counters. In extensive experiments we show that for coarse query granularities gaEPS significantly outperforms a basic EPS that is based on an extension of previous work, both in terms of memory usage and runtime.

Igor Timko
the Free University of Bozen-Bolzano (Italy)
E-mail: timko@inf.unibz.it

Michael Böhlen
the University of Zürich (Switzerland)
E-mail: boehlen@ifi.uzh.ch

Johann Gamper
the Free University of Bozen-Bolzano (Italy)
E-mail: gamper@inf.unibz.it

1 Introduction

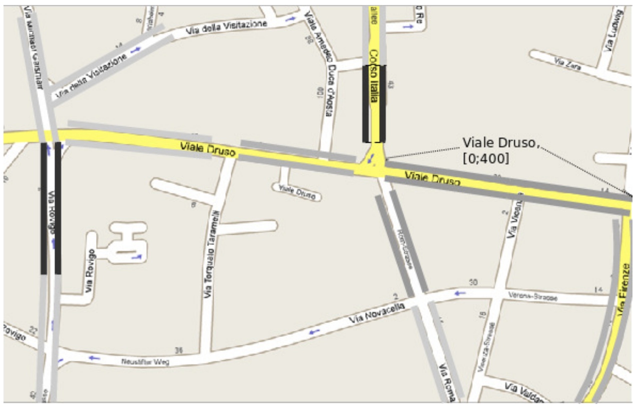
Spatiotemporal databases are becoming more and more popular in various application domains, including traffic data analysis. The proliferation of the GPS (Global Positioning System) technology facilitates the tracking of car positions, and huge amounts of traffic data are collected. Cars are equipped with a GPS receiver and periodically send their current position to a central server [13]. We assume that the server stores GPS data using a discrete time and space model. The time is a finite sequence of time granules (e.g., seconds), and the space is a finite sequence of space granules (e.g., 1-meter road segments). Spatiotemporal granules are obtained by combining time and space granules.

In such an application scenario, *sequenced spatiotemporal aggregation* (SSTA) can be used to obtain a summary of the traffic density in a city/region. Consider the following example query:

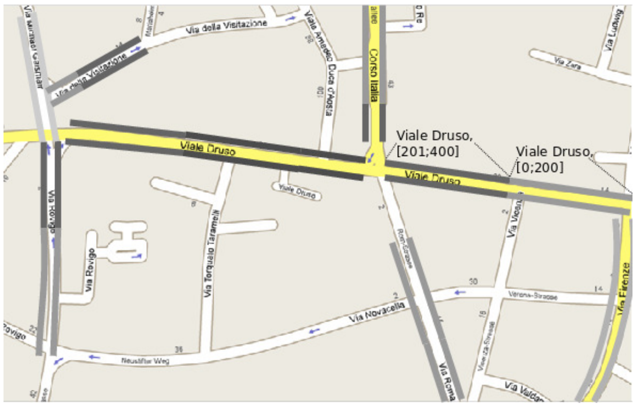
Q1: *"For each 10-second time period, what is the number of cars in each 100-meter road segment?"*

Conceptually, an SSTA query computes one aggregate value for each query granule. Typically, the query granularity, the granularity at which the user wants to get the result, is much coarser than the data granularity, the granularity at which the server records the car positions. Figure 1 illustrates a typical result for Q1, evaluated on data from the city of Bolzano-Bozen between 5:00PM and 5:15PM. The lines in different shades of gray along road segments indicate the traffic density on that segment, varying from very low traffic (no line) to moderate traffic (gray line) and jammed traffic (black line).

Past research on spatiotemporal aggregation focused mainly on box aggregation, and no efficient algorithm for SSTA exists [15]. This paper is the first to formally define SSTA and to present an efficient evaluation algorithm for



(a) 5:00PM–5:10PM



(b) 5:11PM–5:15PM

Fig. 1 Traffic Analysis Using SSTA.

the COUNT, SUM, AVG, MIN, and MAX aggregation functions. The algorithm relies on the efficient computation of *constant rectangles*, which are 2D generalizations of constant intervals as used in temporal aggregation [17]. A constant rectangle is defined as a maximal rectangle spanned by a space and a time interval, such that the aggregate value is constant at all space-time points in the rectangle. In Fig. 1(a), each line segment with a different shade represents a constant rectangle, each having the same time interval [5:00,5:11) but a different space interval. For instance, the rectangle marked as “Viale Druso, [0,401)” represents a segment of the street “Viale Druso”, stretching 400 meters from the beginning of the road. Figure 1(b) shows the traffic situation over the time interval [5:11,5:16). The constant rectangles changed, e.g., “Viale Druso, [0,401)” is now replaced by “Viale Druso, [0,201)” and “Viale Druso, [201,401)”, which have a different traffic density.

To efficiently compute SST aggregates, we reduce SSTA to the problem of rectangle intersection, for which efficient plane sweep solutions exist [20]. Besides determining intersecting rectangles we have to compute aggregate functions over these rectangles and leverage granularity conversion. We do so by designing a granularity aware event point schedule (gaEPS) and a granularity aware sweep line status (gaSLS). gaEPS efficiently handles duplicate time and space points, which occur frequently for coarse query granularities. (Many cars send their position updates within a 60-second period and on the same 100-meter road segment.) gaEPS maintains a summary of the input tuples. We propose two different implementations, namely gaEPS^T and gaEPS^H, which use, respectively, trees and hashmaps to store the summary. gaSLS extends the Balanced Tree [17] with a minimal set of counters.

We implemented the new SSTA framework on top of the Secondo DBMS [7], and we conducted extensive experiments. The results show that for queries with a coarse granularity gaEPS is significantly more efficient than an EPS that is based on an extension of previous work, both in terms of runtime and memory consumption. The results of the experiments also show that gaEPS^T is generally faster, while gaEPS^H uses less memory.

The main technical contributions of this paper can be summarized as follows:

– We provide a formal definition of SSTA with data-to-query granularity conversion.
– We propose an effective and efficient reduction of SSTA to the plane sweep framework.
– We propose gaEPS and gaSLS: two granularity aware data structures that are designed for coarse query granularities with many duplicates.
– We generalize the 1-dimensional incremental computation of aggregates to two dimensions and we extend it to MIN/MAX.
– We implement our algorithm in the Secondo DBMS and conduct experiments that confirm that our solution takes advantage of coarse granularity queries.

The rest of the paper is structured as follows. Section 2 introduces preliminary concepts, followed by a discussion of related work in Section 3. In Section 4, we define the SSTA operator with granularity conversion. In Section 5, we describe how to process SSTA queries using a plane sweep strategy, and we introduce efficient EPS and SLS data structures, termed gaEPS and gaSLS, respectively. Section 6 presents an evaluation algorithm that is based on the two new data structures. Two implementation variants of gaEPS, namely gaEPS^T and gaEPS^H, are discussed. The results of our experimental study are described in Section 7. Section 8 concludes the paper and points to future work.

2 Preliminaries

We use a discrete time model. The *time line*, Δ^T, is a finite sequence of atomic *chronons*. A *time granularity*, Γ^T, is a partitioning of the time line into convex sets of chronons,

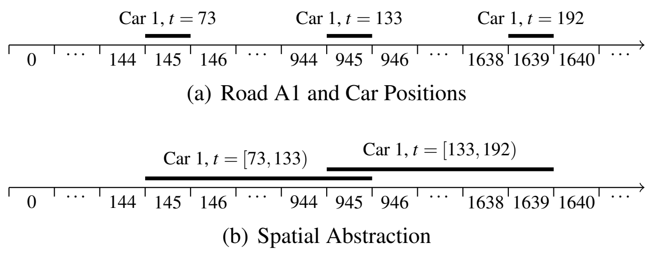


Fig. 2 Space Model.

called *time granules*. For a time granule, $g_t \in \Gamma^T$, the predecessor and successor granules are $g_t - 1$ and $g_t + 1$, respectively. A *time interval* is a convex set of time granules, and it is represented as $\bar{t} = [t_s, t_f)$, where t_s is its inclusive start time granule and t_f its exclusive finish time granule.

We use a discrete 1.5-dimensional space model. The space consists of a finite set of connected space lines that represent roads. Each *space line*, Δ^S, is a finite sequence of atomic *hodons*. A *space granularity*, Γ^S, is a partitioning of a space line into convex sets of hodons, called *space granules*. For a space granule, $g_s \in \Gamma^S$, the predecessor and successor granules are $g_s - 1$ and $g_s + 1$, respectively. A *space interval* is a convex set of space granules, and it is represented as $\bar{s} = [s_b, s_e)$, where s_b is its inclusive begin space granule and s_e its exclusive end space granule.

Whenever it is clear from the context, we use the term *corner points* both to refer to the end points, t_s and t_f, of a time interval as well as to the end points of a space interval, s_b and s_e.

A *spatiotemporal granularity*, $\Gamma = \Gamma^T \times \Gamma^S$, is a partitioning of the spatiotemporal domain, $\Delta^T \times \Delta^S$, into convex sets of spatiotemporal granules. A *spatiotemporal granule* is a pair $(g_t, g_s) \in \Gamma^T \times \Gamma^S$.

We assume an application scenario with GPS-based tracking of car positions [13]. Each car is equipped with a GPS receiver and periodically sends the actual position to a central server (e.g., every 60 seconds). Since a car's position is unknown between measurement points, we assume that at any time between two consecutive measurements the car is somewhere between the two reported space granules, including the two positions. This abstraction introduces spatiotemporal uncertainty. Figure 2(a) illustrates a road A1 and three measurements about the position of Car 1. The road is subdivided into granules of 1 meter each (i.e., $\Gamma_d^T = \{0, 1, \dots\}$), whereas the time granularity is 1 second (i.e., $\Gamma_d^S = \{0, 1, \dots\}$). The three measurements are indicated by a short line and report that car 1 was at space granule 145 at time 73, at space granule 945 at time 133, and at space granule 1639 at time 192. Figure 2(b) shows the abstracted measurements. Notice that position 945 is included in two space intervals, since the car may also stop at this position for a while.

A history of car movements is stored in a *spatiotemporal relation* (ST relation), R, with schema $(A_1, \dots, A_k, T, S)$, where $A_1, \dots, A_k$ are explicit attributes, $T = [T_s, T_f)$ is the timestamp attribute with domain Γ_d^T, and $S = [S_b, S_e)$ is the spacestamp attribute with domain Γ_d^S. The product $T \times S$ is termed (spatiotemporal) *validity rectangle* of a tuple. The fact represented by an ST tuple is valid at each spatiotemporal granule $g \in T \times S$. A tuple is valid at time granule g_t iff $g_t \in T$ and is valid at space granule g_s iff $g_s \in S$.

Example 1 Figure 3 shows the ST relation, `Cars`, that stores a history of car movements. *RID* (road ID) and *CID* (car ID) are explicit attributes, and T and S are the time- and spacestamp attributes, respectively. For instance, tuple r_1 represents that car 1 is on road A1 in the validity rectangle $[73; 133) \times [145; 946)$. We use the `Cars` relation as a running example throughout the paper.

	CID	*RID*	*T*	*S*
r_1	1	A1	[73,133)	[145,946)
r_2	1	A1	[133,193)	[945,1640)
r_3	2	A1	[75,135)	[143,902)
r_4	2	A1	[135,195)	[901,1652)
r_5	3	A1	[78,138)	[140,973)
r_6	3	A1	[138,198)	[972,1609)
r_7	4	A1	[5,65)	[1001,1701)
r_8	4	A1	[65,125)	[710,1002)
r_9	5	A1	[6, 66)	[145, 910)
r_{10}	5	A1	[66, 126)	[909, 920)

Fig. 3 ST Relation `Cars` Storing Car Movements.

Table 1 summarizes the most important notation used in this paper.

3 Related Work

Aggregation is an important operator in many application scenarios, and it has been studied for different data types and settings. Specifically, OLAP applications heavily rely on complex aggregation queries, and a lot of research has been dedicated to provide efficient solutions for aggregation [9, 12]. Most of these techniques consider data to be static. The intrinsic properties of space and time make aggregation of temporal and spatial data more involved (e.g., the computation of constant intervals/rectangles), and different techniques are required. In this section we provide a comparative analysis of the relevant work on spatial, temporal, and spatiotemporal aggregation.

Notation	Description
Δ^T, Δ^S	time line, space line
$\Gamma^T, \Gamma^S, \Gamma = \Gamma^T \times \Gamma^S$	time, space, and spatiotemporal granularity
Γ_d, Γ_q	data and query granularity
$g_t, g_s, (g_t, g_s)$	temporal, spatial, spatiotemporal granule
g, $g+1$, $g-1$	granule, its successor, and predecessor
g_d, g_q	data granule, query granule
$R(A_1, \ldots, A_k, T, S)$	ST relation schema
$T = [T_s, T_f)$	timestamp attribute
$S = [S_b, S_e)$	spacestamp attribute
$T \times S$	(spatiotemporal) validity rectangle
$\bar{t} = [t_s, t_f)$	time interval
$\bar{s} = [s_b, s_e)$	space interval
γ_t, γ_s, γ	time, space, and spatiotemporal granularity converter
$F = \{f_1/C_1, \ldots, f_k/C_k\}$	set of aggregate functions
$\mathscr{G}^{SSTA}[F, \gamma]R$	SSTA operator (with coalescing)
$CTI(R)$	set of constant temporal intervals of R
$CSI(R, \bar{t})$	set of constant spatial intervals of R
$CR(R)$	set of constant rectangles of R
CTP, CSP	distinct time and space end points of R

Table 1 Notation.

3.1 Spatial Aggregation

Previous research work on spatial aggregation [18,23] focuses mainly on range (or box) aggregation in the two-dimensional space, which computes an aggregate function over all objects that fall into the query region.

The *aR-tree* [18] is based on the R-tree [11] and maintains for each R-tree bounding box the total number of objects (for COUNT aggregation) that fall into that box. This speeds up query processing, because one does not need to descend the nodes that are totally enclosed by the query region. The main disadvantage of the aR-tree is that the query cost depends on the size of the query region: the larger the query region, the more bounding boxes overlap with it.

The *aP-tree* [23] avoids this disadvantage at the expense of extra memory usage. It transforms spatial objects into objects in the (key, time) plane and computes the aggregation results by using the MVB-tree [2].

Zhang and Tsotras [28] present the *MR-tree* together with a number of different optimization techniques to improve the query performance for the MIN and MAX aggregate functions. In [29] the efficient computation of the COUNT, SUM, and AVG functions, and specialized aggregate indexes that incrementally maintain aggregates are proposed.

There is no work on sequenced spatial aggregation.

3.2 Temporal Aggregation

In contrast to the research activities on spatial aggregation, which largely ignored sequenced aggregation, past work on temporal aggregation investigated both box (range) temporal aggregation and sequenced temporal aggregation, which conceptually assigns one aggregate to each time granule. A number of methods for one-dimensional temporal aggregation have been developed [4,14,17]. The main problem tackled by these methods is the efficient computation of time constant intervals (i.e., the maximal time intervals over which the aggregate value remains constant). There is no work on two-dimensional temporal aggregation.

The *Aggregation tree* (A-tree) [14] algorithm works in two steps. First, while scanning the input relation a tree is built in memory. Each node in the tree represents a time interval and an associated partial aggregate value. Each level of the tree partitions the entire timeline. The intervals at the leaf level represent the constant intervals, while the intervals higher up in the tree partition the timeline at a coarser level. Second, the tree is traversed in depth-first order. The partial aggregate values along a path from the root to a leaf node are accumulated to produce the aggregation result which is associated with that leaf node's constant interval. The A-tree has two drawbacks. First, if the tuples are sorted by timestamp, the tree degenerates to a linked list. Second, the tree is large, because all constant intervals are stored in the leave nodes.

The *Balanced Tree* [17] avoids the pitfalls of the A-tree. While scanning the input tuples, the corner points of a tuple's timestamp are sorted by inserting them into a self-balancing binary search tree. Each node of the tree stores a time point and two counters that record the number of tuples that start and finish at this time point, respectively. Once the tree has been built, it is traversed in-order to identify the constant intervals. Two consecutive time points define a constant interval. Compared to the A-tree, the Balanced Tree has always logarithmic insertion time and it is generally smaller, because each endpoint of a constant interval is stored only once.

The *TMDA operator* [4] improves over the A-tree and Balanced Tree methods by consuming less memory on average and still providing the same runtime. The TMDA operator computes constant intervals based on the following observation: if the input relation is scanned in chronological order (by the tuple's start time), at any time point, t, the result tuples that end before t can be computed. Hence, as the argument relation is being scanned, result tuples are produced and old tuples are removed from main memory; only tuples that are valid at time t are kept in memory.

The above frameworks pursue memory-based solutions of temporal aggregation. The *SB-tree* [24,25] is a disk-based index structure for temporal aggregation that supports 1D, sequenced and cumulative temporal aggregation. The tree maintains a hierarchy of time intervals, each one being associated with a partial aggregation result. The tree is traversed in a depth-first order to compute the sequenced aggregation. The *MVSB-tree* [26,27] extends the SB-tree and supports temporal aggregation combined with a key-range predicate over one key dimension. The MVSB-tree is logically a series of SB-trees, one per time point. The MVSB-tree efficiently processes dominance-sum queries. A box query in the (key, time) plane can be reduced to four dominance-sum queries.

3.3 Spatiotemporal Aggregation

Past work on spatiotemporal aggregation [19,21,22] assumes a 2D space model and concentrates on spatiotemporal box aggregation, which is a generalization of spatial box aggregation. Given a spatial region and a time interval, an aggregation is computed over all spatiotemporal objects that are present in that region during that time interval. There is no work on SSTA.

The *aRB-tree* [19] extends the aR-tree with a time dimension. In an aRB-tree, 2D spatial regions are indexed by an R-tree. For each bounding box of this R-tree, the time-varying number of objects that fall into the box is kept in a B-tree [1]. Similarly to the aR-tree, the aRB-tree speeds up aggregation by storing the number of objects for the bounding intervals of a B-tree. This eliminates the need to traverse the subtree of nodes that are totally enclosed by the query region. The aRB-tree does not prevent the well-known double counting problem. Double counting means that the same object is counted twice if it stays in the query region during two time granules of the query time interval.

The *sketch index* [22] avoids double counting. This index modifies the aRB-tree: instead of recording the number of objects, a sketch (compressed representation) of the object IDs is kept for each B-tree's bounding interval. As a result, the sketch index is generally larger than the aRB-tree. Moreover, the sketch index only answers queries approximately.

The *Adaptive Multi-Dimensional Histogram* [21] is another method for approximate box query processing. The 2D space is divided into a (large) number of cells. A counter for a number of objects is associated with each cell. To speed up processing, a histogram is built over the space. Cells with similar counter values are put into the same bucket. Thus, each bucket of this histogram holds a spatial (2D) region and a counter for the number of objects in this region. The spatial regions do not overlap. As the counter values of the cells change, the buckets are reorganized.

3.4 Rectangle Intersection with Plane Sweep

The rectangle intersection problem is solved efficiently with a plane sweep algorithm, which works in two steps. The first step creates the *Event Point Schedule* (EPS), which is a sorted list of events, where each event consists of a corner point in one dimension (say the time dimension) together with the two corner points in the other dimension (say the space dimension) of a rectangle. The second step sweeps a line through the EPS and halts at each event. The active rectangles from the EPS are collected in the *Sweep Line Status* (SLS). To obtain the intersection of the rectangles, the one-dimensional space intervals in SLS are intersected for each new event.

The EPS is usually organized as a sorted list of all event (time) points [20]. The SLS is implemented as a segment tree [3,20], interval tree [8,20], or priority search tree. The segment tree [3,20] has the disadvantage that for inserting active space intervals we frequently need to descend to low levels in the tree. The interval tree [8,20] overcomes this weakness by introducing additional links and nodes. The priority search tree [16,20] improves over the segment and interval tree by storing the corner space points only for the active space intervals rather than for all space intervals.

The plane sweep paradigm leads generally to asymptotic worst-case optimal solutions in runtime and memory usage [20]. In this paper, we present data structures that are designed to achieve good average-case complexity. We present an efficient EPS that takes advantage of coarse query granularities and groups all events with the same time point into a single event, which leads to a significant reduction of the EPS size.

4 SSTA with Data-to-Query Granularity Conversion

SSTA groups the input tuples by spatiotemporal granules, one group per granule, and applies one or more aggregation functions to each group. This section provides a formal definition of SSTA that includes data-to-query granularity conversion and coalescing of result tuples.

4.1 Data-to-Query Granularity Conversion

Since the granularity at which data is measured is typically much finer than the query granularity, we convert the validity rectangles of the input tuples from the data granularity to the query granularity.

Definition 1 (Granularity Converter) Let Γ_d and Γ_q be data and query (time or space) granularities, respectively, and assume that for each data granule, $g_d \in \Gamma_d$, there exists a query granule, $g_q \in \Gamma_q$, such that $g_d \subset g_q$ and $g_d \cap g'_q = \emptyset$ for all $g'_q \in \Gamma_q$, $g'_q \neq g_q$. The *granularity converter* is a function, $\gamma : \Gamma_d \mapsto \Gamma_q$, that is defined as $\gamma(g_d) = g_q$.

The granularity converter maps the finer data granularity, Γ_d, to the coarser query granularity, Γ_q, by assigning to each data granule in Γ_d the unique corresponding query granule in Γ_q. We use the granularity converter to transform the validity rectangle of spatiotemporal tuples to query granularity.

Definition 2 (Validity Rectangle Converter) Let R be an ST relation with schema $(A_1, \dots, A_k, T, S)$ and Γ_d^T, Γ_d^S, Γ_q^T, and Γ_q^S be, respectively, data time and space and query time and space granularity with converter functions $\gamma_t : \Gamma_d^T \mapsto \Gamma_q^T$ and $\gamma_s : \Gamma_d^S \mapsto \Gamma_q^S$. For a tuple, $r \in R$, the *validity rectangle converter*, γ, is defined as

$$\gamma(r.T, r.S) = \{(\gamma_t(g_t), \gamma_s(g_s)) \mid (g_t, g_s) \in r.T \times r.S\}.$$

Function γ converts each individual data granule of the validity rectangle determined by the two arguments to the query granularity. The following lemma yields a more efficient granularity conversion strategy that converts only the corner points of a validity rectangle.

Lemma 1 Let R, Γ_d^T, Γ_d^S, Γ_q^T, Γ_q^S, γ_t, and γ_s be as in Def. 2. For a tuple, $r \in R$, with validity rectangle $r.T \times r.S$ the following holds:

$$\gamma(r.T, r.S) = [\gamma_t(r.T_s), \gamma_t(r.T_f - 1) + 1) \\ \times [\gamma_s(r.S_b), \gamma_s(r.S_e - 1) + 1).$$

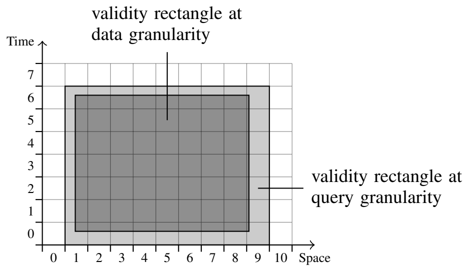
Proof Follows from the observation that two adjacent granules may only be converted to either the same granule or to two adjacent granules. □

Example 2 Consider Query Q1. The data is stored at the granularity 1 sec × 1 m, whereas the query specifies the granularity 10 sec × 100 m. Therefore, for each input tuple the validity rectangle is transformed to the query granularity using the following converter functions for time and space granularities, respectively: $\gamma_t(g_t) = \lfloor g_t/10 \rfloor$ and $\gamma_s(g_s) = \lfloor g_s/100 \rfloor$. Figure 4 illustrates the conversion of the validity rectangles for the Cars relation. Figure 4(a) shows the validity rectangle of tuple r_9 together with the converted rectangle. The conversion of the entire relation is shown in Fig. 4(b) in tabular representation and in Fig. 4(c) in a graphical representation.

Notice that many distinct validity rectangles at the data granularity collapse into identical validity rectangles at the query granularity. For instance, tuples r_1, r_3, and r_5 have different data validity rectangles, but at the query granularity their validity rectangles are identical (cf. Fig. 4(c)).

In addition to the granularity converters for time and space points, we need data-to-query granularity converters for the explicit attributes. For an attribute A, we assume a converter, $\gamma_a : \Gamma_d^A \mapsto \Gamma_q^A$, where Γ_d^A is the domain of attribute A and Γ_q^A is a partitioning of this domain into larger granules. For instance, the speed of cars is usually measured at the granularity of 1 km/h (e.g., 63 km/h), while in a query we might only be interested in larger ranges, e.g., what is the number of cars that drive at a speed in the ranges 50–59 km/h, 60–69 km/h, etc.

Whenever it is clear from the context, in the rest of the paper we will use γ as converter for time points, space points, and explicit attributes.

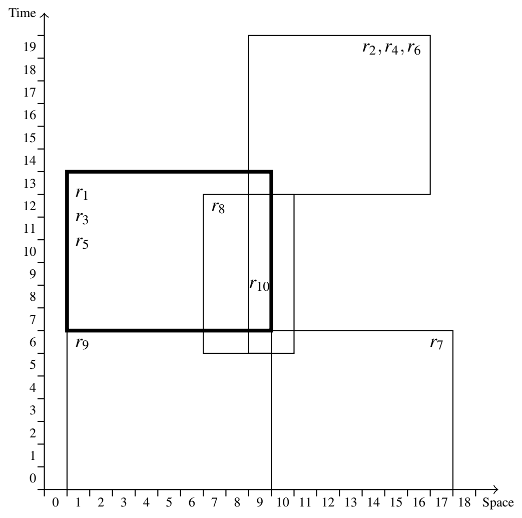


validity rectangle at data granularity
Time
7
6
5
4
3
2
1
0
validity rectangle at query granularity
0 1 2 3 4 5 6 7 8 9 10 Space

(a) Validity Rectangle of Tuple r_9.

TID	T	S	$\gamma_t(T)$	$\gamma_s(S)$
r_1	[73,133)	[145,946)	[7,14)	[1,10)
r_2	[133,193)	[945,1640)	[13,20)	[9,17)
r_3	[75,135)	[143,902)	[7,14)	[1,10)
r_4	[135,195)	[901,1652)	[13,20)	[9,17)
r_5	[78,138)	[140,973)	[7,14)	[1,10)
r_6	[138,198)	[972,1609)	[13,20)	[9,17)
r_7	[5,65)	[1001,1701)	[0,7)	[10,18)
r_8	[65,125)	[710,1002)	[6,13)	[7,11)
r_9	[6,66)	[145,910)	[0,7)	[1,10)
r_{10}	[66,126)	[909,920)	[6,13)	[9,10)

(b) Conversion of Validity Rectangles.



(c) Validity Rectangles at Query Granularity.

Fig. 4 Conversion of Validity Rectangles for Relation Cars.

4.2 Definition of SST Aggregation

We begin with an informal conceptual definition of sequenced spatiotemporal aggregation and proceed with coalescing to obtain a more compact result relation.

Conceptually, SST aggregation returns one aggregate value for each individual spatiotemporal granule. Let R be an ST relation, F be a set of aggregate functions, and $\Gamma_q^T \times \Gamma_q^S$ be the query granularity. The SSTA operator determines for each query granule, g, the group of all tuples in R that are valid at g. The aggregate functions in F are evaluated over each group and reported as result tuple with g as validity rectangle. For example, Query Q1 can be expressed as an SSTA query with $F = \{\text{COUNT}(*)\}$. Figure 5 illustrates the query result, where each cell with an aggregate value represents a single result tuple. Notice the granularity conversion from data granularity to query granularity.

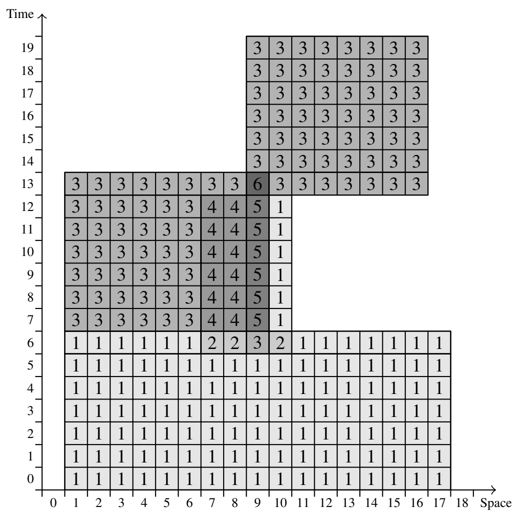


Fig. 5 Result of SST Aggregation (without Coalescing of Result Tuples).

The conceptual definition of SST aggregation defines a result tuple for each spatiotemporal granule for which data is available, which leads to unnecessarily large result relations. The size of the aggregation result can be reduced by *coalescing* result tuples over adjacent granules that have equal aggregation values. This is a two-step process:

1. The SSTA result tuples are coalesced along the time dimension, yielding maximal timestamps over which the input tuples are constant.
2. The intermediate result of step 1 is coalesced along the space dimension, yielding maximal spacestamps over which the aggregation results are constant.

The resulting validity rectangles are termed constant rectangles and are formalized in the following definition.

Definition 3 (Constant Rectangles) Let R be an ST relation with schema $(A_1, \ldots, A_k, T, S)$, $F = \{f_1/C_1, \ldots, f_k/C_k\}$ be a set of aggregate functions, and $R[g_t] = \{r \mid r \in R \wedge g_t \in \gamma_t(r.T)\}$ be the set of all tuples of R that are valid at query time granule g_t. The set of *constant temporal intervals* is defined as

$$\begin{aligned} CTI(R) = \{\bar{t} \mid \bar{t} = [t_s, t_f) \wedge \\ R[t_s - 1] \neq R[t_s] \wedge \\ R[t_f] \neq R[t_s] \wedge \\ \forall g_t \in \bar{t}(R[g_t] = R[t_s])\} \end{aligned}$$

Let $R[g_s, \bar{t}] = \{r \mid r \in R \wedge g_s \in \gamma_s(r.S) \wedge \bar{t} \cap \gamma_t(r.T) \neq \emptyset\}$ be the set of all tuples of R that are valid at space time granule g_s and constant interval $\bar{t} \in CTI(R)$. The set of *constant spatial intervals* of R over time interval $\bar{t}$ is defined as

$$\begin{aligned} CSI(R, \bar{t}) = \{\bar{s} \mid \bar{s} = [s_b, s_e) \wedge \\ F(R[s_b - 1, \bar{t}]) \neq F(R[s_b, \bar{t}]) \wedge \\ F(R[s_e, \bar{t}]) \neq F(R[s_b, \bar{t}]) \wedge \\ \forall g_s \in \bar{s}(F(R[g_s, \bar{t}]) = F(R[s_b, \bar{t}])\}. \end{aligned}$$

Finally, the set of *constant rectangles*, $CR(R)$, of R is defined as $CR(R) = \{\bar{t} \times \bar{s} \mid \bar{t} \in CTI(R) \wedge \bar{s} \in CSI(R, \bar{t})\}$.

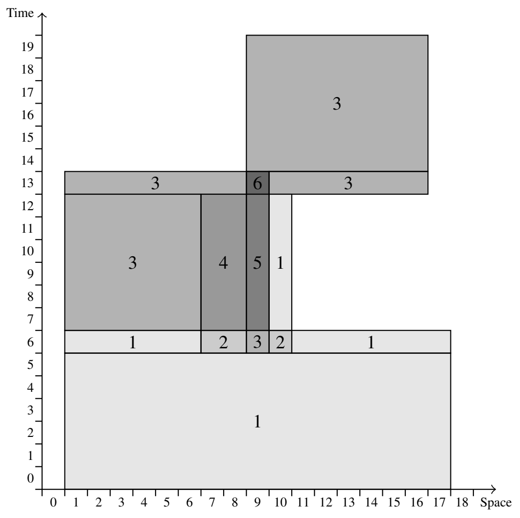
A constant temporal interval, $\bar{t} = [t_s, t_f)$, is a maximal time interval such that the set of tuples that are valid at the query time granules in $\bar{t}$ is constant throughout $\bar{t}$ and different at $t_s - 1$ and t_f. The second step performs coalescing along the spatial dimension. For each constant temporal interval, a set of constant spatial intervals is computed. The coalescing along the space dimension involves the aggregation functions. A constant spatial interval, $\bar{s} = [s_b, s_e)$, over a constant temporal interval, $\bar{t}$, is a maximal space interval such that the aggregation results evaluated over all tuples in R that temporally overlap $\bar{t}$ are constant. Finally, a constant rectangle, $\bar{t} \times \bar{s}$, is a maximal spatiotemporal rectangle, where $\bar{t}$ is a constant temporal interval and $\bar{s}$ is a constant spatial interval of $\bar{t}$, and the value of the aggregate functions is constant at each granule in $\bar{t} \times \bar{s}$ and changes in at least one adjacent granule.

Example 3 Consider the converted validity rectangles of the `Cars` relation in Fig. 4. The set of constant temporal intervals is $CTI(\texttt{Cars}) = \{[0,6)), [6,7), [7,13), [13,14), [14,20)\}$. For instance, for the constant interval $[7,13)$ we have $R[6] = \{r_7, r_8, r_9, r_{10}\}$, $R[7] = \{r_1, r_3, r_5, r_8, r_{10}\}$, and $R[13] = \{r_1, r_2, r_4, r_5, r_6\}$. Hence, $R[6] \neq R[7]$, $R[13] \neq R[7]$, and $\forall g_t \in [7,13)(R[g_t] = R[7])$. For $\bar{t} = [0,6)$, the set of constant spatial intervals is $CSI(\texttt{Cars}, \bar{t}) = \{[1,18)\}$. The interval $[1,18)$ is maximal, since $R[0,\bar{t}] = \emptyset$, $R[1,\bar{t}] = \{r_9\}$, and $R[18,\bar{t}] = \emptyset$, hence $\text{COUNT}(R[0,\bar{t}]) \neq \text{COUNT}(R[1,\bar{t}])$, $\text{COUNT}(R[18,\bar{t}]) \neq \text{COUNT}(R[1,\bar{t}])$, and $\forall g_s \in [1,18)(\text{COUNT}(R[g_s,\bar{t}]) = \text{COUNT}(R[1,\bar{t}]))$ (cf.

Fig. 5). Notice that the aggregation value of 1 is produced by two different input tuples, namely r_9 over the space interval $[1,10)$ and r_7 over the space interval $[10,18)$. The next constant time interval is $[6,7)$, for which five different constant space intervals are produced, i.e., $CSI(\texttt{Cars},[6,7)) = \{[1,7),[7,9),[9,10),[10,11),[11,18)\}$.

	T	S	Cnt
z_1	1	[0,6)	[1,18)
z_2	1	[6,7)	[1,7)
z_3	2	[6,7)	[7,9)
z_4	3	[6,7)	[9,10)
z_5	2	[6,7)	[10,11)
z_6	1	[6,7)	[11,18)
z_7	3	[7,13)	[1,7)
z_8	4	[7,13)	[7,9)
z_9	5	[7,13)	[9,10)
z_{10}	1	[7,13)	[10,11)
z_{11}	3	[13,14)	[1,9)
z_{12}	6	[13,14)	[9,10)
z_{13}	3	[13,14)	[10,17)
z_{14}	3	[14,20)	[9,17)

(a) Tabular Representation



(b) Graphical Representation

Fig. 6 Coalesced Result of SST Aggregation.

Definition 4 (SST Aggregation) Let $F = \{f_1/C_1,\dots,f_k/C_k\}$ be a set of aggregate functions and $R[g] = \{r \mid r \in R \wedge g \in \gamma(r.T,r.S)\}$ be the set of all tuples of R that are valid at g. The *sequenced spatiotemporal aggregation (SSTA)* operator, $\mathscr{G}^{SSTA}[F,\gamma]R$, is defined as

$$\mathscr{G}^{SSTA}[F,\gamma]R = \{x \mid \bar{t} \times \bar{s} \in CR(R) \wedge \exists g(g \in \bar{t} \times \bar{s} \wedge x = (f_1(R[g]),\dots,f_k(R[g]),\bar{t},\bar{s}))\}.$$

The result relation has schema $(C_1,\dots,C_k,T,S)$.

SSTA (with coalescing) defines result tuples over constant rectangles. Each $f_i/C_i \in F$ is some aggregate function that takes an ST relation as argument and applies aggregation to one of the attributes. The aggregation result is stored as the value of an attribute C_i. For each constant rectangle, $\bar{t} \times \bar{s}$, the SSTA operator produces one result tuple by evaluating the aggregate functions, F, over the set of all tuples that cover the constant rectangle. Notice the granularity conversion from data granularity to query granularity in $R[g]$.

Example 4 Query Q1 of our running example can be expressed as $\mathscr{G}^{SSTA}[\text{COUNT}(*)/Cnt,\gamma]\texttt{Cars}$, where γ is the rectangle converter that uses γ_t and γ_s from Example 2. Figure 6 illustrates the coalesced query result. Each result tuple is drawn as a box with the aggregate result inside.

5 SSTA Query Processing

We look at the evaluation of SST aggregation queries from a geometrical perspective and reduce it to the problem of rectangle intersection, which can be efficiently solved with a plane sweep algorithm [20]. In addition to the rectangle intersection we need to determine (maximal) constant rectangles at query granularity and for each such rectangle we compute the aggregate functions over all input tuples that fall into that rectangle. We begin with a baseline plane sweep algorithm and proceed by improving it to efficiently support SSTA queries with a coarse query granularity. To simplify the notation, we assume a single aggregate function over an attribute A, i.e., $F = \{f(A)\}$. It is straightforward to extend all definitions and algorithms for the more general case for $F = \{f_1,\dots,f_k\}$.

5.1 A Baseline Algorithm

Algorithm 1 shows a basic plane sweep algorithm for SSTA queries. The algorithm sweeps the 2D plane along the time axis by jumping from event to event in the *event point schedule* (EPS). The EPS records in chronological order all corner time points of the input tuples together with a summary of the tuple consisting of the spacestamp and optionally some explicit attribute values, which is required to compute the constant rectangles and the aggregation values. For instance, an event $(0,start,[10,18))$ records that tuple r_7 starts at time 0 and has a spacestamp $[10,18)$. The sweep line stops whenever it enters or exits one or more validity rectangles. As the sweep line advances, the *sweep line status* (SLS) consumes the events from the EPS and stores the summaries of those tuples that overlap the sweep line. When the sweep line moves to the next event, two actions are performed. First, a new set of constant rectangles together with the aggregate values is derived from the SLS. Since the time interval of these rectangles is determined by the previous and current positions of the sweep line, the constant rectangles can be determined efficiently by intersecting only the one-dimensional space intervals in the SLS. Second, the SLS is updated with the new event. That is, if the sweep line enters (exits) a tuple's validity rectangle, its summary is inserted to (deleted from) the SLS.

Algorithm 1: Basic Plane Sweep Algorithm

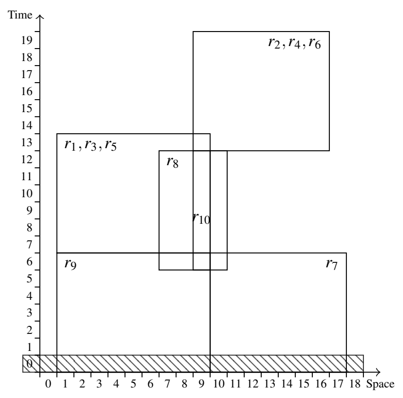
```
Construct EPS;
Initialize empty SLS;
while EPS is not empty do
    Move sweep line to the next event in EPS;
    if event > first event then
        Compute result tuples using SLS;
    Update SLS with event;
```

Example 5 Figure 7 illustrates a few steps of the baseline algorithm for Query Q1. The event time points are 0, 6, 7, 13, 14, and 20. In Fig. 7(a) the sweep line is at time point 0 (hatched line), where it enters the validity rectangles of r_7 and r_9, yielding $SLS = \{[1,10),[10,18)\}$; only the spacestamps need to be recorded for the COUNT aggregate. No result tuples are produced after reading the first event. Figure 7(b) shows the sweep line at the next event at time 6. A single result tuple $(1,[0,6)\times[1,18))$ is produced with the aggregate value 1 (gray rectangle). Notice that the two input tuples in SLS are coalesced. The SLS is updated with tuples r_8 and r_{10} that both start at time 6, yielding $SLS = \{[1,10)$, $[7,11)$, $[9,10)$, $[10,18)\}$. Next, the sweep line moves to time point 7 as shown in Fig. 7(c). Five new result tuples with constant rectangles that all have the same time interval, $[6,7)$, are derived.

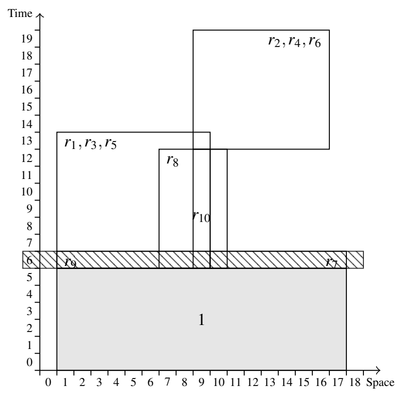
Next, we define a basic EPS structure that is an extension of previous work [20], and we show that it cannot take advantage of a coarse query granularity.

Definition 5 (Basic EPS) The *basic event point schedule* for an ST relation R, aggregate function $f(A)$, and granularity converter γ is a sorted list of events, $\{e_1,\ldots,e_{2|R|}\}$. Each *event* is a quadruple $e = (t, type, a, [s_b, s_e))$, where $t \in \Gamma_q^T$, $type \in \{start, finish\}$, $a = \Gamma_q^A$, and $s_b, s_e \in \Gamma_q^S$. For each tuple, $r \in R$, the EPS contains two events, $(\gamma(r.T_s), start, \gamma(r.A), \gamma(r.S))$ and $(\gamma(r.T_f - 1) + 1, finish, \gamma(r.A), \gamma(r.S))$.

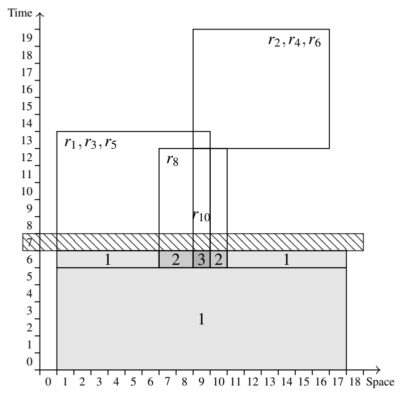
The basic EPS contains two events for each input tuple r: the start event with $t = \gamma(r.T_s)$ when the sweep line enters the tuple's validity rectangle and the finish event with time point $t = \gamma(r.T_f - 1) + 1$ when the sweep line exits the rectangle. Both events store the aggregation attribute A and the spacestamp of r. There may be several events at a time point t. Ties are broken in an arbitrary way.



(a) Sweep Line at Time Point 0



(b) Sweep Line at Time Point 6



(c) Sweep Line at Time Point 7

Fig. 7 Plane Sweep Algorithm to Compute SSTA.

Example 6 Figure 8 shows the basic EPS for Query Q1. The events are sorted bottom up. Many events have the same time point, and each input tuple produces exactly two events. For instance, events e_2 and e_9 are derived from tuple r_9 with validity rectangle $[0,7)\times[1,10)$. No explicit attributes are needed for the COUNT aggregate function.

Lemma 2 For an ST relation R, the number of events in the basic EPS is independent of the query granularity and it is twice the cardinality of R, i.e., $|\text{Basic EPS}| = 2|R|$.

	t	*type*	$\bar{s}$
e_{20}	20,	*finish*,	$[9,17)$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
e_9	7,	*finish*,	$[1,10)$
e_8	7,	*finish*,	$[10,18)$
e_7	7,	*start*,	$[1,10)$
e_6	7,	*start*,	$[1,10)$
e_5	7,	*start*,	$[1,10)$
e_4	6,	*start*,	$[9,10)$
e_3	6,	*start*,	$[7,11)$
e_2	0,	*start*,	$[1,10)$
e_1	0,	*start*,	$[10,18)$

Fig. 8 Basic EPS.

Proof The proof follows immediately from Def. 5. □

The size of the basic EPS depends on the number of input tuples, hence the algorithm cannot take advantage of a coarse query granularity. We now develop an efficient, granularity aware EPS that takes advantage of coarse query granularities.

5.2 gaEPS — an EPS for Coarse Query Granularity

We start from the basic EPS structure and introduce three different optimizations to obtain a granularity aware EPS that takes advantage of a coarse query granularity to reduce the number of distinct time and space points.

To facilitate the discussion, for an ST relation, R, and a granularity converter, γ, we define the following two sets: $CTP = \{t \mid \exists r \in R(t = \gamma(r.T_s) \vee t = \gamma(r.T_f - 1) + 1)\}$ is the set of *distinct corner time points* that are obtained from all tuples in R after granularity conversion and $CSP(t) = \{s \mid \exists r \in R(t \in r.T \wedge (s = \gamma(r.S_b) \vee s = \gamma(r.S_e - 1) + 1))\}$ is the set of *distinct corner space points* that are obtained from those tuples in R that are valid at time t after granularity conversion.

5.2.1 Optimization 1: Eliminating Duplicate Time Points

The first optimization is to merge all events in the basic EPS that have the same time point, t, into a single event of the form (t, Xs, Xf), where

$$Xs(t) = \{(\gamma(r.A), \gamma(r.S)) \mid r \in R \wedge \gamma(r.T_s) = t\}$$
$$Xf(t) = \{(\gamma(r.A), \gamma(r.S)) \mid r \in R \wedge \gamma(r.T_f) = t\}.$$

The bag Xs (Xf) contains the converted spacestamps and the converted aggregation attribute of all input tuples for which time t is a corner point.

Example 7 Figure 9 shows the EPS after applying optimization 1. The 20 events in the basic EPS have been reduced to six events, each having a different time point. For instance, the event with $t = 0$ records that two tuples with space intervals $[10,18)$ and $[1,10)$ start at time 0; no tuples finish at time 0.

	t	$Xs(t)$	$Xf(t)$
e_6	20,	$\{\}$,	$\{[9,17), [9,17), [9,17)\}$
e_5	14,	$\{\}$,	$\{[1,10), [1,10), [1,10)\}$
e_4	13,	$\{[9,17), [9,17), [9,17)\}$,	$\{[7,11), [9,10)\}$
e_3	7	$\{[1,10), [1,10), [1,10)\}$,	$\{[10,18), [1,10)\}$
e_2	6,	$\{[7,11), [9,10)\}$,	$\{\}$
e_1	0,	$\{[10,18), [1,10)\}$,	$\{\}$

Fig. 9 EPS after Optimization 1.

5.2.2 Optimization 2: Transforming Space Intervals into Points

The second optimization transforms space intervals to their corner points. Instead of storing tuples with spacestamps in the events, we store the distinct corner points, s, of the spacestamps together with a summary of the tuples that have s as corner point. For instance, in Fig. 9 the event with time point 7 stores three identical space intervals. This can be avoided by storing the corner points only once together with the information that three tuples begin and end at 1 and 10, respectively.

The following definition specifies four groups of tuples that form the basis for this transformation.

Definition 6 (Tuple groups) Let R be an ST relation, γ be a granularity converter, and $t \in \Gamma_q^T$ and $s \in \Gamma_q^S$ be a query time and space granule, respectively. We define the following four groups of tuples:

$$R_{sb}[t,s] = \{r \in R \mid \gamma(r.T_s) = t \wedge \gamma(r.S_b) = s\},$$
$$R_{se}[t,s] = \{r \in R \mid \gamma(r.T_s) = t \wedge \gamma(r.S_e - 1) + 1 = s\},$$
$$R_{fb}[t,s] = \{r \in R \mid \gamma(r.T_f - 1) + 1 = t \wedge \gamma(r.S_b) = s\},$$
$$R_{fe}[t,s] = \{r \in R \mid \gamma(r.T_f - 1) + 1 = t \wedge \gamma(r.S_e - 1) + 1 = s\}.$$

Each group contains all tuples in R that have granule (t,s) as one of its corner points. This is graphically illustrated in Fig. 10. The black box represents the query granule (t,s). The four large validity rectangles are representative for all tuples that are collected in the corresponding group. For instance, the rectangle with solid lines represents the group, $R_{sb}[t,s]$, of all tuples for which the granule (t,s) forms the lower-left corner of the validity rectangle.

Example 8 In our example, the four tuple groups for granule $(7,10)$ are as follows: $\texttt{Cars}_{sb}[7,10] = \{\}$, $\texttt{Cars}_{se}[7,10] = \{r_1, r_3, r_5\}$, $\texttt{Cars}_{fb}[7,10] = \{r_7\}$, $\texttt{Cars}_{fe}[7,10] = \{r_9\}$.

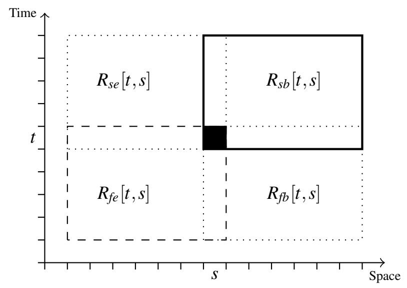


Fig. 10 Graphical Illustration of the Groups of Tuples.

With these tuple groups in place, we replace the bag of tuples Xs and Xf by a set of space points with associated tuple groups. Instead of storing each individual input tuple, we map the input tuples to their corner space points and store for each distinct corner point the tuple groups introduced above. Each event in the EPS with time point t has now the form (t, Xs', Xf'), where

$$Xs'(t) = \{(s, R_{sb}[t,s], R_{se}[t,s]) \mid s \in CSP(t)\} \text{ and}$$
$$Xf'(t) = \{(s, R_{fb}[t,s], R_{fe}[t,s]) \mid s \in CSP(t)\}.$$

Xs' records (among all the tuples that start at time t) the set of tuples for which s is a corner space point. Similar, Xf' records (among all the tuples that finish at time t) the set of tuples for which s is a corner space point.

Next, we reduce the size of the tuple groups by computing a summary of the tuple groups. For an event (t, Xs', Xf') and an aggregate function $f(A)$, we get

$$Xs''(t) = \{(s, \tilde{f}(R_{sb}[t,s]), \tilde{f}(R_{se}[t,s])) \mid s \in CSP(t)\} \text{ and}$$
$$Xf''(t) = \{(s, \tilde{f}(R_{sb}[t,s]), \tilde{f}(R_{se}[t,s])) \mid s \in CSP(t)\},$$

where

$$\tilde{f} = \begin{cases} \mathscr{G}_{\text{COUNT}(*)} & \text{if } f = \text{COUNT}, \\ \mathscr{G}_{\text{SUM}(A)} & \text{if } f = \text{SUM}, \\ {}_A\mathscr{G}_{\text{COUNT}(*)} & \text{if } f = \text{MIN} \lor f = \text{MAX}. \end{cases}$$

For COUNT and SUM, $\tilde{f}$ computes the aggregate function over the tuple groups. For MIN and MAX, we need to count for each different attribute value of A the number of tuples with this value. This information is needed to determine the next MIN or MAX value when we leave the validity rectangle of the tuple that determines the current MIN or MAX value.

Example 9 Figure 11 shows the EPS after applying optimization 2. Notice the difference to Fig. 9, where for example the event at time 7 records three identical spacestamps for the tuples r_1, r_3, r_5. Instead, Fig. 11 stores the corner points, 1 and 10, together with two counters representing the number of tuples that begin and end at this space point, respectively. Hence, $Xs''(7) = \{(1, \{3\}, \{0\}), (10, \{0\}, \{3\})\}$ means that three space intervals begin at space granule 1 and three end at space granule 10.

5.2.3 Optimization 3: Computing the Difference of the Summaries

The last optimization towards a granularity aware EPS is to replace the four summaries by a single summary, which is achieved by computing a difference between the individual sets.

We use the following sum and difference operators, $\oplus$ and $\ominus$, between two sets that contain attribute-value pairs, (a, v):

$$\begin{aligned} X \oplus Y = \{(a,v) \mid & \\ & \exists v', v''((a,v') \in X \land (a,v'') \notin Y \land v = v' + v'') \lor \\ & (a,v) \in X \land \forall v'((a,v') \notin Y) \lor \\ & (a,v) \in Y \land \forall v'((a,v') \notin X)\}, \end{aligned}$$

$$\begin{aligned} X \ominus Y = \{(a,v) \mid & \\ & \exists v', v''((a,v') \in X \land (a,v'') \notin Y \land v = v' - v'') \lor \\ & (a,v) \in X \land \forall v'((a,v') \notin Y) \lor \\ & \exists v''((a,v'') \in Y \land v = -v'' \land \forall v'((a,v') \notin X))\}. \end{aligned}$$

For pairs with equal attribute value, a, the operators apply, respectively, the $+$ and $-$ operation on v. If a appears only in one of the two sets, the result for $\oplus$ is v. For the $\ominus$ operator, the negative value, $-v$, is returned if a appears only in Y.

Definition 7 (gaEPS) Let R be an ST relation, γ be a granularity converter, $f(A)$ be an aggregate function, CTP be the set of corner time points, and $CSP(t)$ be the set of corner space points at time t. The *granularity aware event point schedule (gaEPS)*, is a sorted list of events,

$$gaEPS = \{(t_1, X(t_1)), \ldots, (t_k, X(t_k))\},$$

where $t_i \in CTP$ is a corner time point and $X(t_i) = \{(s_1, Y(s_1)), \ldots, (s_l, Y(s_l))\}$ is a set of corner space points, $s_j \in CSP(t_i)$, with associated summaries that are defined as

$$\begin{aligned} Y(s_j) = \{ & (\tilde{f}(R_{sb}[t_i,s_j]) \ominus \tilde{f}(R_{fb}[t_i,s_j])) \ominus \\ & (\tilde{f}(R_{se}[t_i,s_j]) \ominus \tilde{f}(R_{fe}[t_i,s_j]))\}. \end{aligned}$$

gaEPS records an event for each corner time point after granularity conversion. An event with time t_i stores a set, $X(t_i)$, of distinct corner space point, s_j, together with an associated set $Y(s_j)$ of summary information that is computed over the corresponding tuple groups. For instance, for the COUNT aggregate function the summary is determined as follows: First, for the tuples that begin at space point s_j, the difference between the number of tuples that start and the number of tuples that finish at time t_i is built. Second, the same difference is built for the tuples that end at space s_j. Finally, the difference between these two intermediate results is computed. Notice that the counters might have negative values.

	t	$Xs''(t)$	$Xf''(t)$
e_6	20,	{},	{ (9, 3, 0), (17, 0, 3) }
e_5	14,	{},	{ (1, 3, 0), (10, 0,3) }
e_4	13,	{ (9, 3, 0), (17, 0, 3) },	{ (7, 1, 0), (9, 1, 0), (10, 0, 1), (11, 0, 1) }
e_3	7,	{ (1, 3, 0), (10, 0, 3) },	{ (1, 1, 0), (10, 1, 1), (18, 0, 1) }
e_2	6,	{ (7, 1, 0), (9, 1, 0), (10, 0, 1), (11, 0, 1) },	{}
e_1	0,	{ (1, 1, 0), (10, 1, 1), (18, 0, 1) },	{}

Fig. 11 EPS after Optimization 2.

Example 10 Figure 12 illustrates gaEPS for our running example. Each event represents a distinct corner time point, t, and a list of distinct space points together with a counter. For example, the event with time point 7 stores a set X with three space points and associated counters. The pair $(10,\{-3\})$, for instance, states that the difference between the tuples that begin and end at space point 10 is -3.

	t	$X(t)$
e_6	20,	{ (9, −3), (17, 3) }
e_5	14,	{ (1, −3), (10, 3) }
e_4	13,	{ (7, −1), (9, 2), (10, 1), (11, 1), (17, −3) }
e_3	7,	{ (1, 2), (10, −3), (18, −1) }
e_2	6,	{ (7, 1), (9, 1), (10, −1), (11, −1) }
e_1	0,	{ (1, 1), (10, 0), (18, −1) }

Fig. 12 gaEPS — EPS after Optimization 3.

5.3 gaSLS — an SLS for Coarse Query Granularity

When the sweep line advances along the time, the SLS consumes the events from the EPS and maintains the summary about all current tuples, i.e., tuples with a validity rectangle that overlaps the sweep line. From this information the result tuples are derived.

Definition 8 (gaSLS) Let R be an ST relation, γ be a granularity converter, $f(A)$ be an aggregate function, CTP be the set of distinct time corner points, and $CSP(t)$ be the set of distinct space corner points at time t. The *granularity aware sweep line status (gaSLS)* at time $t_i \in CTP$ is a sorted list of pairs, $gaSLS = \{(s_1, Z(s_1)), \ldots, (s_l, Z(s_l))\}$, where $s_j \in CSP(t_i)$ is a corner space point and $Z(s_j)$ is defined as

$$Z(s_i) = \tilde{f}(\{r \in R \mid \gamma(r.S_b) = s \wedge t_i \in \gamma(r.T)\}) \ominus \\ \tilde{f}(\{r \in R \mid \gamma(r.S_e - 1) + 1 = s \wedge t_i \in \gamma(r.T)\}).$$

gaSLS records information for a time slice at time t_i. An item, $(s_j, Z(s_j))$, represents a distinct corner point, s_{i_j}, with the associated summary, $Z(s_j)$, that is computed over all tuples that are valid at t_i and s_j (whereas in events only tuples for which t_i and s_j are corner points are considered for the summary). The summary is a single counter (sum) for the COUNT (SUM) function, and is a set of (attribute value, counter) pairs for the MIN and MAX functions.

t	$gaSLS$
7	$\{(1,3),(7,1),(9,1),(10,-4),(11,-1)\}$
6	$\{(1,1),(7,1),(9,1),(10,-1),(11,-1),(18,-1)\}$
0	$\{(1,1),(10,0),(18,-1)\}$

Fig. 13 gaSLS at Different Time Points.

Example 11 Figure 13 shows gaSLS at three consecutive events with time points 0, 6, and 7, respectively (see also Fig. 7). At time 0, the sweep line overlaps the validity rectangles of tuples r_7 and r_9. Since at space point 10 tuple r_9 finishes and tuple r_7 starts, the counter has the value 0. At time 6 the sweep line enters two new tuples, r_8 and r_{10}, while r_7 and r_9 are still valid. Next, at time 7 the sweep line enters three new tuples, r_1, r_3, and r_5, which all have the same space interval $[1,10)$, and it exits from two validity rectangles, $[0,7) \times [10,18)$ and $[0,7) \times [1,10)$.

The gaSLS contains sufficient information to compute result tuples over all input tuples that actually overlap the sweep line, and it can be incrementally updated by using the information in gaEPS. Thus, as the sweep line traverses the EPS and reads a new event from EPS, two steps are performed: (1) new result tuples are derived from the current gaSLS, and (2) gaSLS is updated with the information from the current event.

First, consider the computation of new result tuples from *gaSLS* for an aggregate function $f(A)$. This can be done incrementally by traversing *gaSLS* in sequential order. Let t_i and t_{i+1} be two consecutive events, where *gaSLS* has been updated with the event at t_i, and t_{i+1} is the current event. We initialize an accumulator variable, $\bar{Z} = \emptyset$. Each pair of consecutive items, $(s_j, Z(s_j))$ and $(s_{j+1}, Z(s_{j+1}))$ in *gaEPS*, produces a result tuple $(\bar{f}(\bar{Z}), [t, t_{i+1}) \times [s_j, s_{j+1}))$, where

$$\bar{f}(\bar{Z}) = \begin{cases} v & \text{if } f = \text{COUNT} \wedge \bar{Z} = \{v\} \\ v & \text{if } f = \text{SUM} \wedge \bar{Z} = \{v\} \\ a & \text{if } f = \text{MIN} \wedge a = \min\{a' \mid (a', .) \in \bar{Z}\} \\ a & \text{if } f = \text{MAX} \wedge a = \max\{a' \mid (a', .) \in \bar{Z}\} \end{cases}$$

For the COUNT and SUM function, the accumulator $\bar{Z}$ contains exactly one value, which is the result of the aggregate function. For the MIN and MAX functions, we take the smallest respective the largest attribute value a in $\bar{Z}$. After producing a result tuple, the accumulator variable is updated to $\bar{Z} = \bar{Z} \oplus Z(s_{j+1})$.

Example 12 Consider the *gaSLS* at time point 6 shown in Fig. 13. The traversal of gaSLS produces the result tuples z_2 to z_6, e.g., $z_2 = (1, [6,7), [1,7))$ and $z_3 = (2, [6,7), [7,9))$ (see also Fig. 6(a)).

The second step is to update gaSLS with the information from the current event, $e_i = (t_i, X(t_i))$. For each $(s_j, Y(s_j)) \in X(t_i)$, the *gaSLS* is updated as follows:

$$gaSLS = \begin{cases} gaSLS \cup \{(s_j, Y(s_j))\} & \nexists (s_j, Z(s_j)) \in gaSLS \\ (gaSLS \setminus \{(s_j, Z(s_j))\}) & \\ \cup \{(s_j, Y(s_j) \oplus Z(s_j))\} & \exists (s_j, Z(s_j)) \in gaSLS \end{cases}$$

New space points not yet in *gaSLS* are added together with the associated summaries, whereas for existing space points the respective summaries from the event are added to the summary in *gaSLS*.

Example 13 Consider the *gaSLS* at time 0 in Fig. 13. When the sweep line moves to time point 6, the *gaSLS* is updated with event e_2 from Fig. 12. A new entry in *gaSLS* is created for the space points 7, 9, and 11. The counter for the space point 10 is updated to -1.

6 Aggregation Algorithm

In this section, we describe an efficient plane sweep algorithm for computing SST aggregation queries that uses gaEPS and gaSLS.

6.1 Top-Level Algorithm

Algorithm 2 presents the main algorithm, SSTAGG, which takes an ST relation, R, and a validity rectangle converter, γ, as input and outputs the aggregation result as an ST relation, *Res*. The algorithm iterates through all road IDs of the input relation and applies a sweep line strategy to compute the aggregates for each road. For each road, *rid*, two steps are performed: (1) gaEPS is constructed by retrieving all input tuples that match *rid*, converting them to the query granularity, and inserting them into the list. (2) By traversing gaEPS, the constant rectangles and the aggregation results are computed.

Algorithm 2: SSTAGG(R, γ, F)

```
Input: ST relation $R$, rectangle converter $\gamma$, and aggregate
       functions $F$;
Output: ST relation $Res$;
$Res \leftarrow \emptyset$;
foreach road-ID $rid \in \pi[RID]R$ do
    $gaEPS \leftarrow$ LOADEPS$(\sigma[RID = rid]R, \gamma, F)$;
    $Res_{rid} \leftarrow \{rid\} \times$ TRAVERSEEPS$(gaEPS, F)$;
    $Res \leftarrow Res \cup Res_{rid}$;
return $Res$;
```

6.2 Loading gaEPS

Algorithm 3 presents the algorithm LOADEPS for constructing a gaEPS for an input relation R and a granularity converter γ. After initializing *gaEPS* to an empty list, the algorithm iterates through the input relation. For each input tuple, $r \in R$, the corner points of the validity rectangle, $[r.T_s, r.T_f) \times [r.S_b, r.S_e)$, are converted to the query granularity, yielding t_s, t_f, s_b, and s_e, respectively. Then *gaEPS* is updated. First, the start time, t_s, is processed. If an event, (t, X), for the start time t_s already exists in *gaEPS*, the summaries for the corner points s_b and s_e are updated with r; a new entry is added to X, if no entry exists for a corner point. In the case that no event for time point t_s exists in *gaEPS*, a new event is added which contains the two corner points, s_b and s_e, with the associated summary. Second, the rectangles's finish time, t_f, is processed in a similar way. The only difference is that the summary for s_b is subtracted, whereas the summary for s_e is added.

Example 14 Figure 14 depicts *gaEPS* after processing tuple r_1, which at the query granularity is valid in $[7, 14) \times [1, 10)$ (cf. Fig. 4(b)). It contains two events that represent r_1's corner time points. Each of the two events stores a set with two space points and associated counters (summary). For instance, the event at time 7 records that at time 7 one tuple starts with a space interval that begins at space point 1 and ends at space point 10. The event at time 14 records that r_1 finishes. Notice that the counters associated to the space points changed sign, which reflects that the sweep line exits the constant rectangle.

t	X
14	{(1, −1), (10, 1)}
7	{(1, 1), (10, −1)}

Fig. 14 *gaEPS* after Inserting Tuple r_1.

6.3 Traversing gaEPS

Algorithm 4 shows the algorithm TRAVERSEEPS that takes as input a gaEPS and returns the aggregation result. The

Algorithm 3: LOADEPS(R,γ,F)

```
Input: ST relation $R$, rectangle converter $\gamma$, and aggregate
       function F = {f(A)};
Output: $gaEPS$;
$gaEPS \leftarrow \emptyset$;
foreach $r \in R$ do
    /* Convert $r'$s validity rectangle */
    $t_s \leftarrow \gamma(r.T_s)$;
    $t_f \leftarrow \gamma(r.T_f)$;
    $s_b \leftarrow \gamma(r.S_b)$;
    $s_e \leftarrow \gamma(r.S_e)$;
    /* Compute summary of $r$ */
    $v \leftarrow \tilde{f}(\{r\})$;
    /* Process $r$'s start time point */
    if $\exists(t,X) \in gaEPS$ with $t = t_s$ then
        if $\exists(s,Y) \in X$ with $s = s_b$ then $Y \leftarrow Y \oplus v$;
        else $X \leftarrow X \cup \{(s_b,v)\}$;
        if $\exists(s,Y) \in X$ with $s = s_e$ then $Y \leftarrow Y \ominus v$;
        else $X \leftarrow X \cup \{(s_e,\emptyset \ominus v)\}$;
    else
        $gaEPS \leftarrow gaEPS \cup \{(t_s,\{(s_b,v),(s_e,\emptyset \ominus v)\}\}$;
    /* Process $r$'s finish time point */
    if $\exists(t,X) \in gaEPS$ with $t = t_f$ then
        if $\exists(s,Y) \in X$ with $s = s_b$ then $Y \leftarrow Y \ominus v$;
        else $X \leftarrow X \cup \{(s_b,\emptyset \ominus v)\}$;
        if $\exists(s,Y) \in X$ with $s = s_e$ then $Y \leftarrow Y \oplus v$;
        else $X \leftarrow X \cup \{(s_e,v)\}$;
    else
        $gaEPS \leftarrow gaEPS \cup \{(t_s,\{(s_b,\emptyset \ominus v),(s_e,v)\}\}$;
return $gaEPS$;
```

Algorithm 4: TRAVERSEEPS$(gaEPS,F)$

```
Input: granularity aware end point structure $gaEPS$;
Output: ST relation $Res$;
$Res \leftarrow \emptyset$;
$t_{prev} \leftarrow -1$;
$gaSLS \leftarrow \emptyset$;
foreach $(t,X) \in gaEPS$ do
    /* Traverse SLS and produce result */
    if $t_{prev} > -1$ then
        $s_{prev} \leftarrow -1$;
        $\bar{Z} \leftarrow \emptyset$;
        foreach $(s,Z) \in gaSLS$ do
            if $s_{prev} > -1$ then
                $Res \leftarrow Res \cup \{(\bar{f}(\bar{Z}),[t_{prev},t),[s_{prev},s))\}$;
            $\bar{Z} \leftarrow \bar{Z} \oplus Z$;
            $s_{prev} \leftarrow s$;
        $t_{prev} \leftarrow t$;
    /* Update SLS */
    foreach $(s,Y) \in X$ do
        if $\exists(s,Z) \in gaSLS$ then $Z \leftarrow Z \oplus Y$;
        else $gaSLS \leftarrow gaSLS \cup \{(s,Y)\}$;
        if $Z = \emptyset$ then $Z \leftarrow Z \setminus \{(s,Z)\}$;
return $Res$;
```

main loop processes the events in *gaEPS* in chronological order. For each event, (t,X), two actions are performed. First, *gaSLS* is traversed and new result tuples are produced. For that, an accumulator result variable, $\bar{Z}$, is initialized to the empty set. Then *gaSLS* is traversed, and for each item, $(s,Z) \in gaSLS$, a result tuple is produced with time interval $[t-1,t)$ and space interval $[s-1,s)$ and aggregate value that is derived from $\bar{Z}$. Second, *gaSLS* is updated with the space points and associated summaries, (s,Y), in X. If for a space point, s, an entry in *gaSLS* already exists, the summary from the current event is added to the entry; otherwise, a new entry, (s,Y), is inserted into *gaSLS*.

Example 15 Figure 15 depicts the evolution of *gaSLS* together with the produced result tuples during the processing of the first three events in *gaEPS*. After processing the first event at time 0, *gaSLS* contains two items with space points 1 and 18, which have been inserted as new items. The space point 10 has been (inserted and then) deleted, since its counter is zero. No result tuples are produced. Next, TRAVERSEEPS enters the main loop and reads the event with time point 6. The traversal of gaSLS produces one result tuple, where the time interval of the validity rectangle is determined by the time of the previous and current event. Then *gaSLS* is updated with the three space points with associated counters of the current event. All three space points are not yet in *gaSLS*, and hence are inserted. The next event at time 7 produces five new result tuples, each with time interval $[6,7)$. The update of *gaSLS* produces no new items, but deletes the item with space point 18.

gaEPS

t, X
20, $\{(9,-3),(17,3)\}$
14, $\{(1,-3),(10,3)\}$
13, $\{(7,-1),(9,2),(10,1),(11,1),(17,-3)\}$
7, $\{(1,2),(10,-3),(18,-1)\}$
6, $\{(7,1),(9,1),(10,-1),(11,-1)\}$
~~0, $\{(1,1),(10,0),(18,-1)\}$~~

gaSLS

s, Z
18, -1
1, 1

Result tuples

Cnt	T	S

t, X
20, $\{(9,-3),(17,3)\}$
14, $\{(1,-3),(10,3)\}$
13, $\{(7,-1),(9,2),(10,1),(11,1),(17,-3)\}$
7, $\{(1,2),(10,-3),(18,-1)\}$
~~6, $\{(7,1),(9,1),(10,-1),(11,-1)\}$~~
~~0, $\{(1,1),(10,0),(18,-1)\}$~~

s, Z
18, -1
11, -1
10, -1
9, 1
7, 1
1, 1

Cnt	T	S
1	$[0,6)$	$[1,18)$

t, X
20, $\{(9,-3),(17,3)\}$
14, $\{(1,-3),(10,3)\}$
13, $\{(7,-1),(9,2),(10,1),(11,1),(17,-3)\}$
~~7, $\{(1,2),(10,-3),(18,-1)\}$~~
~~6, $\{(7,1),(9,1),(10,-1),(11,-1)\}$~~
~~0, $\{(1,1),(10,0),(18,-1)\}$~~

s, Z
11, -1
10, -4
9, 1
7, 1
1, 3

Cnt	T	S
1	$[0,6)$	$[1,18)$
1	$[6,7)$	$[1,7)$
2	$[6,7)$	$[7,9)$
3	$[6,7)$	$[9,10)$
2	$[6,7)$	$[10,11)$
1	$[6,7)$	$[11,18)$

Fig. 15 Evolution of gaSLS and Result Tuples.

6.4 Complexity Analysis

Since gaEPS needs to be accessed in chronological order, we use a self-balancing binary search tree, which has logarithmic seek and insertion time. Similar, we store the space points with the associated summaries, $(s,Y) \in X$, in a self-balancing binary search tree. The runtime complexity of LOADEPS is then $O(N * \log n * \log m)$, where $N = |R|$ is the number of input tuples, $n = |CTP|$ is the number of distinct corner time points in R, and $m = |CSP(t)|$ is the number of distinct corner space points over all input tuples that are valid at time t. That is, for each input tuple, $r \in R$, we have to update two events in gaEPS, and for each of the two events we have to update the summaries of the two corner points. The set X needs not to be sorted, but an efficient search for identical space points is important. By storing X in a hashmap instead in a tree, the runtime complexity of LOADEPS becomes $O(N * \log n)$. Our experiments, however, showed that the resize overhead of hashmaps can be quite high.

For gaSLS, we use a self-balancing binary search tree, since it needs to be traversed in sort order to produce the aggregation results. For each event in gaEPS, the traversal of gaSLS takes $O(m)$ time. The update with the current event is done in $O(\log m)$ time, which yields an overall runtime complexity of $O(n * m \log m)$ for TRAVERSEEPS.

The space complexity of gaEPS is $O(n * m)$. For each distinct time corner point, t, we need to store an entry for each distinct space corner point over all tuples that are valid at t. The size of gaSLS is $O(m)$.

The size of n and m depends on the query granularity. Let t_{min} and t_{max} (s_{min} and s_{max}) be the minimum and maximum time corner points (space corner points) in R, respectively, and let $|g_q^T|$ ($|g_q^S|$) be the size of a query time (space) granule. Then n and m are upper-bounded by

$$n \leq (t_{max} - t_{min})/|g_q^T|$$
$$m \leq (s_{max} - s_{min})/|g_q^S|.$$

Thus, in the worst case we have a distinct space end point at each time granule and a distinct space end point at each space granule. With respect to the size N of the input relation, we get

$$n \leq N * |g_d^T|/|g_q^T|$$
$$m \leq N * |g_d^S|/|g_q^S|.$$

Summarizing, n and m are inverse proportional to the granularity, which means that SSTA benefits from a coarse query granularity. Also note that m is limited by the length of a road and does not grow along with the number of the streets (i.e., the size of the database).

6.5 Implementation Details

6.5.1 Time Points in gaEPS

In order to efficiently support the loading of gaEPS, we organize it as a self-balancing binary search tree [6]. Each event, (t,X), is represented as a tree node with t as its key. Such an implementation guarantees $O(\log n)$ loading time for each tuple, where n is the number of distinct corner time points, i.e., $n = CTP$. The traversal of gaEPS is an in-order traversal of the tree, which can be done in $O(n)$ time.

Example 16 Figure 16 depicts a self-balancing binary search tree that implements gaEPS from Fig. 12.

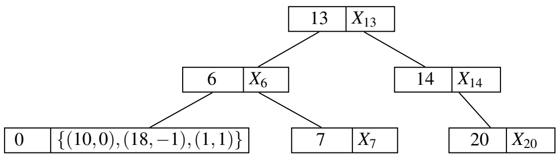


Fig. 16 gaEPS as Binary Tree.

6.5.2 Space Points in gaEPS

Given an event (e,X), we propose two different techniques to efficiently implement X.

gaEPS with Tree-Based Items (gaEPS^T). Each set X of an event (t,X) is stored as a self-balancing binary search tree. A node in the spacestamp tree represents a pair (s,Y), where the space point s is the node's key.

Example 17 Figure 17 depicts a gaEPS^T that implements gaEPS from Fig. 12. For each item, a number denotes its time point and a string indicates its spacestamp tree pointer. In the figure, we can see the spacestamp tree $\mathscr{T}_0$ of the item with time point 0.

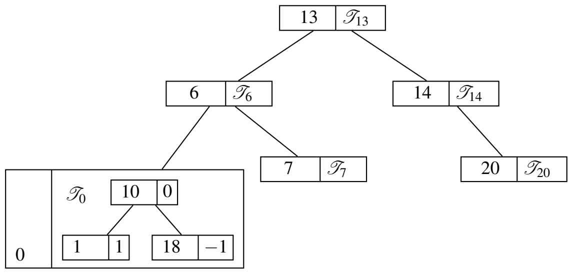


Fig. 17 gaEPS^T.

gaEPS with Hashmap-Based Items (gaEPS^H). An important observation about gaEPS is that the space points and associated summaries in X are not directly used for the computation of the aggregate functions (they are first inserted into gaSLS), hence the order is irrelevant. The only crucial part is to efficiently group identical space points during the construction of gaEPS. Therefore, as an alternative we propose to use hashmaps which have a constant insertion time. An entry of the hashmap is given as (s, Y), where s is the entry's key.

Example 18 Figure 18 depicts gaEPS^H that implements gaEPS from Fig. 12. Each item contains a hashmap to store the corner space points and counters.

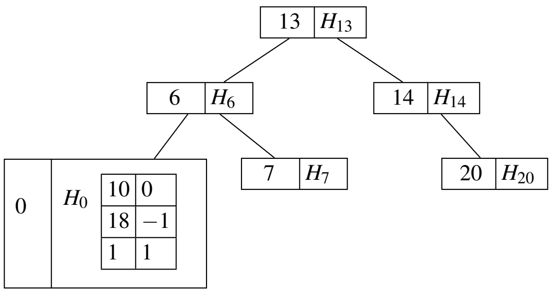


Fig. 18 gaEPS^H.

6.5.3 gaSLS

We implement gaSLS as a self-balancing binary search tree, which guarantees logarithmic insertion and deletion times as well as linear traversal time. Each item, (s, cnt), is implemented as a tree node with s as the key. Thus, our implementation of gaSLS is similar to the Balanced Tree [17] with the following differences. First, without loosing information, we reduce the number of counters per node from two to one. Second, in addition to the Balanced Tree functionality, we need to be able to decrement counters, which may lead to node deletions (see Section 5.3).

Example 19 Figure 19 depicts the implementation of gaSLS at time 7 from Fig. 13.

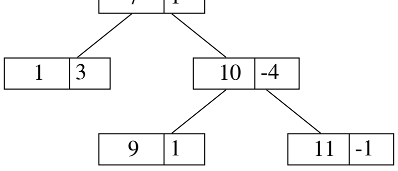


Fig. 19 gaSLS as Binary Tree.

7 Experiments

In this section, we experimentally compare gaEPS and the basic EPS. We present experimental results for the COUNT and MAX aggregation functions. The performance of SUM and MIN is identical to that of COUNT and MAX, respectively. The performance of AVG is very similar to that of COUNT in terms of runtime and memory usage, respectively.

7.1 Setup and Data

We implemented the plane sweep solution for the evaluation of SSTA as algebra operators (user-defined functions) in the Secondo DBMS [7]. The car movement data are stored as ST relations on disk, whereas the EPS and SLS are constructed in memory while retrieving the data from disk. The experiments were run on a machine with an Intel 1.66 GHz CPU and 1 GB RAM under Ubuntu Linux 9.04 (Jaunty).

For our experiments, we use six different ST relations. Each relation has the following schema: $(CID, Speed, RID, T, S)$, where $Speed$ is used as an aggregation attribute for MAX aggregations. The relations are generated with Brinkhoff's generator for moving objects [5]. Specifically, we simulate GPS logs of cars in the road network of the city of Oldenburg, Germany. The number of roads in the network is around 7K. The number of cars per relation varies from 5K to 30K in steps of 5K. For each relation, the number of distinct corner time points is fixed at 3000, while the number of distinct corner space points varies and is proportional to the number of tuples. The data granularity is $1\ sec \times 0.5\ m$. The validity time intervals of all tuples are 10 granules long, since each car reports its position every 10 seconds; the length of the validity space interval varies depending on the car's speed.

7.2 gaEPS vs. Basic EPS

In this experiment, we compare gaEPS and the basic EPS for both runtime and memory usage (using the gaEPS^T and the dense gaEPS^H variant for COUNT and MAX aggregations, respectively; the two variants demonstrate a very similar performance, see Section 7.3). The input relation with 30K cars or 6.5M tuples is used.

7.2.1 Varying the Query Space Granularity

First, we analyze runtime and memory usage when the query space granularity varies between 0 and 500 meters. The query time granularity is fixed at 10 seconds.

Runtime. Figure 20(a) shows the runtime for creating and loading the EPS. For the COUNT function, gaEPS outperforms the basic EPS for all query granularities. This is because gaEPS sorts the events incrementally using a binary tree, and duplicate time points are eliminated, which keeps the list small (max. 3000 events). The basic EPS puts all events (13M) in a very large array and sorts this array. For the MAX function the situation is similar. gaEPS outperforms the basic EPS for coarse query granularities, though the difference is smaller. The reason is that for the MAX function more complex operations are required to construct gaEPS, which is not required for the basic EPS. The relative advantage of the granularity aware gaEPS over the basic EPS increases logarithmically with the query granularity. This is because the number of distinct space points after granularity conversion is inversely proportional to the query space granularity, whereas the size of the basic EPS is constant.

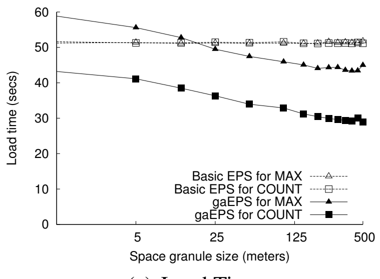
Figure 20(b) shows the time to traverse the EPS and to compute the aggregation results. Again, gaEPS outperforms the basic EPS, except for the MAX function in combination with a small query granularity. Each event in gaEPS requires only one update of gaSLS for each distinct space end point, whereas the basic EPS requires two updates for each tuple associated with the event. The traversal time for both, gaEPS and the basic EPS, depends logarithmically on the query space granularity, since the size of the SLS is inversely proportional to the space granule size. The relative advantage of gaEPS with respect to the basic EPS increases with a coarser space granularity.

Figure 20(c) shows the total runtime of SSTAGG. For COUNT, gaEPS clearly outperforms the basic EPS, and the relative advantage of gaEPS increases when the space granularity becomes coarser. The picture is very similar for the MAX function, except for small query granularities that are close to the data granularity.

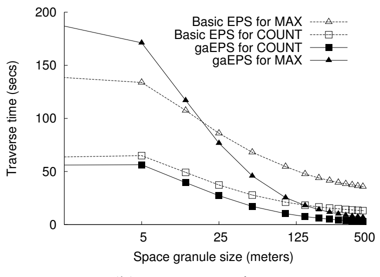
Memory. Figure 21 compares the main memory requirements of the EPSs. We measure the memory usage for each road and take the maximum. Recall that the memory usage of gaEPS is inversely proportional to the space granularity, while the memory usage of the basic EPS is constant. Only for very small space granularities (less than 12.5 meters for the COUNT and 25 meters for the MAX function) the basic EPS consumes less space, while for all larger granularities gaEPS requires significantly less memory. For space granules of one hundred meters or more (which seems realistic in many applications), the space consumption of gaEPS is a small fraction of the space required by the basic EPS.

7.2.2 Varying the Query Time Granularity

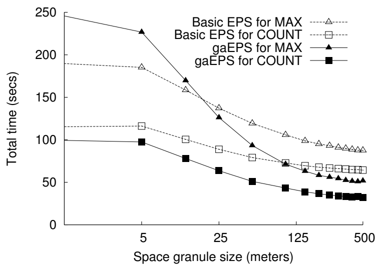
Next, we analyze the dependency of runtime and memory usage from the query time granularity. The query space granularity is fixed at 500 meters, whereas the time granularity varies from 1 to 120 seconds.



(a) Load Time



(b) Traverse Time



(c) Total Time (Load + Traverse)

Fig. 20 Runtime of SSTAGG by Varying the Space Granularity.

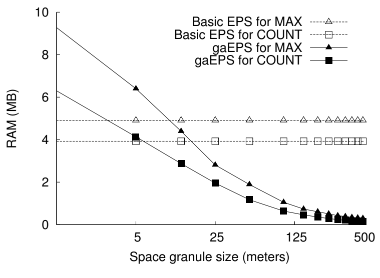
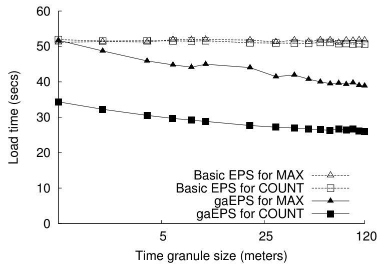


Fig. 21 Memory Usage by Varying the Space Granularity.

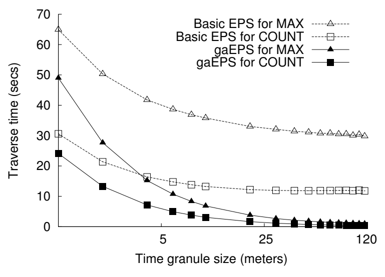
Runtime. Figure 22(a) shows the time it takes to create the EPS structure. gaEPS outperforms the basic EPS regardless of the query time granularity. The relative advantage of gaEPS over the basic EPS increases logarithmically when the query time granularity increases. The reasons for this result are the same as before: gaEPS keeps the list small by storing all tuples with the same time point in one event.

Figure 22(b) shows the time to traverse the EPS and to compute the result tuples. Again, we can see that regardless of the time query granularity gaEPS outperforms the basic EPS. The traversal time of both gaEPS and the basic EPS depend logarithmically on the time query granule size, because the size of the SLS is inversely proportional to the space granule size. However, the relative advantage of gaEPS over the basic EPS increases when the time query granularity size increases. The reason for this is that the total number of inserts/deletes from the SLS is directly proportional to the total number of begin and end space points in the EPS, which is inversely proportional to the time granule size for gaEPS (the fewer time points in gaEPS, the more begin and end space points from the input relation fall under the same element of the spacestamp hashmap (tree)) and constant for the basic EPS.

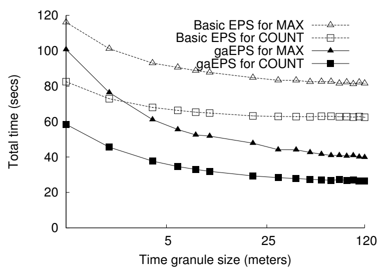
Figure 22(c) shows the total time taken by the SSTA algorithm. As we can see, gaEPS always outperforms the basic EPS and the relative advantage of gaEPS over the basic EPS increases when the time query granule size increases.



(a) Load Time



(b) Traverse Time



(c) Total Time (Load + Traverse)

Fig. 22 Runtime of SSTAGG by Varying the Time Granularity.

Memory. Figure 23 compares the main memory usage of the EPS. Given an input relation, we measure the memory usage for each road and then compute the maximum. The memory consumption of gaEPS is inversely proportional to the time granule size, while the memory consumption of the basic EPS is constant. The reason for that is that the number of corner space points in gaEPS is inversely proportional to the time granule size (the fewer time points in gaEPS, the more corner space points from the input relation are grouped in the same element of the spacestamp hashmap/tree), while the number of corner space points in the basic EPS is constant. gaEPS consumes only a small fraction of the memory that is needed by the basic EPS.

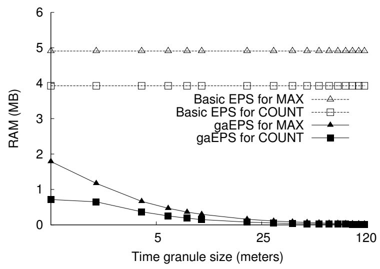
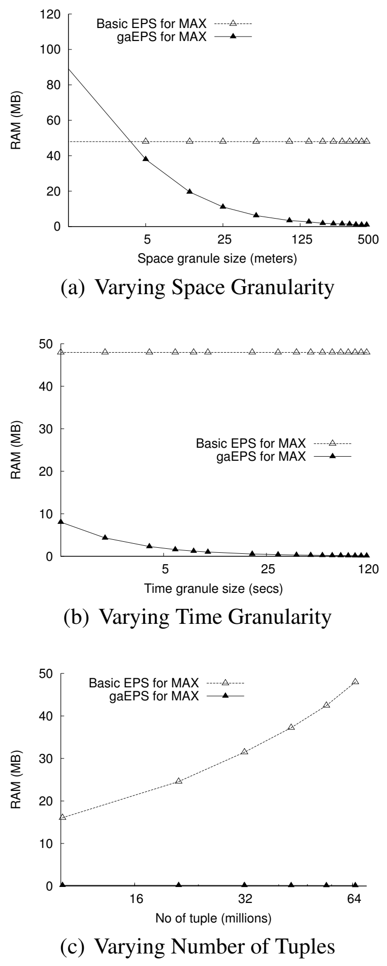


Fig. 23 Memory Usage by Varying the Time Granularity.

7.2.3 Memory Usage for Larger Datasets

In this experiment, we further study the memory-related advantage of gaEPS over the basic EPS, using larger ST relations. Each relation contains data for cars moving for 30 K seconds (i.e., 30 K distinct corner points); the other characteristics are the same as in Sec. 7.1. Again, we measure the memory usage for each road and take the maximum. We show only the MAX aggregation function, since the results for the COUNT function are analogous.

In Fig. 24(a), the space query granularity varies, while the time query granularity is fixed at 10 seconds. In Fig. 24(b), the time query granularity varies, while the space query granularity is fixed at 500 meters. In both figures, the input relation contains 30K cars (i.e., approx. 65 million tuples). For large data sets with significant memory requirements, the basic EPS may become impractically large, and coarsening the query granularity does not help. In contrast, by taking advantage of coarse query granularities gaEPS remains small and scales well for large data sets. In Fig. 24(c), the number of cars varies, while the time and space query granularity is fixed at 120 seconds and 500 meters, respectively. The memory requirements of the basic EPS increase significantly as the size of the input relation grows. At the same time, gaEPS remains extremely small.

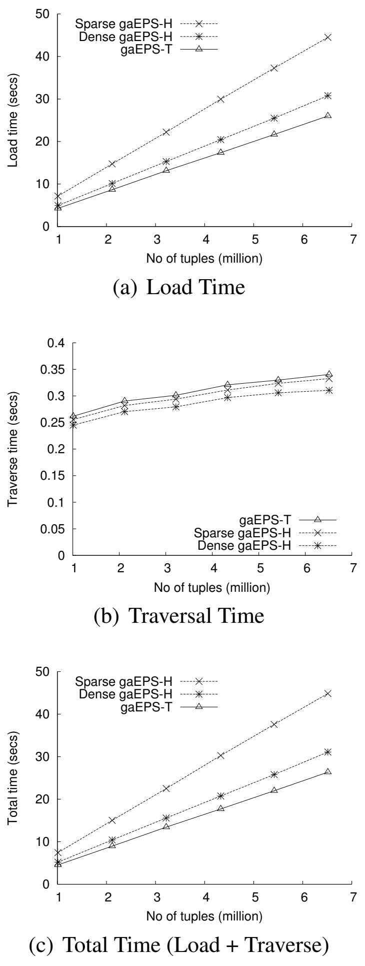


(a) Varying Space Granularity

(b) Varying Time Granularity

(c) Varying Number of Tuples

Fig. 24 Memory Usage for Larger Datasets.



(a) Load Time

(b) Traversal Time

(c) Total Time (Load + Traverse)

Fig. 25 Runtime of Different gaEPS Implementations.

7.3 Different gaEPS Implementations

Figures 25 and 26 analyze the use of different variants of gaEPS (gaEPS^T and gaEPS^H). On the x-axis we vary the size of the input relation. The time and space granularities are fixed at 120 seconds and 500 meters, respectively. We discuss the experimental results only for the COUNT function, since the implementation differences of gaEPS^T and gaEPS^H do not depend on the aggregation function.

We compare two different implementations of gaEPS^H: the first one uses Google's *sparse hashmap* and the second one Google's *dense hashmap*. The sparse hashmap is optimized for memory (an empty bucket occupies almost no space), while the dense hashmap is optimized for speed (at the expense of memory, because the "empty key" is stored in each empty bucket). Both types of hashmap use internal probing (i.e., only one entry, (key, value), per bucket). For details, see [10].

Runtime. Figure 25(a) compares the loading time of gaEPS. It depends linearly on the number of tuples. Since hashmaps have constant insert/lookup time, one could expect that gaEPS^H is faster than gaEPS^T. Surprisingly, gaEPS^H performs worse than gaEPS^T. The reason is the overhead of Google's hashmap implementation: the hashmap is resized (doubled in size) when half of its buckets are full. This is implemented by copying the data into a new hashmap. The sparse hashmap has additional overhead related to memory management.

Figure 25(b) compares the time used for computing constant rectangles from a loaded gaEPS. This is done by updating and traversing gaSLS while traversing gaEPS. The traversal time depends linearly on the number of tuples. All the implementations have approximately the same speed.

Figure 25(c) shows the total time for loading gaEPS and computing constant rectangles by traversing gaEPS.

Memory. Figure 26 compares the memory used by gaEPS, where we take the maximum memory over all roads. Notice that all implementations are (almost) independent from the number of tuples, and they use only a tiny fraction of

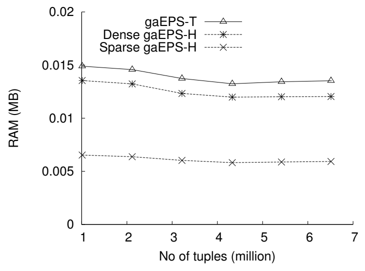


Fig. 26 Memory Usage by Different gaEPS Implementations.

the memory that is consumed by the basic EPS (less than 0.015 MB versus approx. 4 MB).

gaEPS^T is the least efficient, because of a node pointer overhead in spacestamp trees. For each space point, it uses 20 Bytes: 8 Bytes for the data (the key pointer and the count) and 12 Bytes for the item pointers (left, right, and parent). Thus, the constant overhead per space point is 12 Bytes.

The dense hashmap-based gaEPS^H is in terms of memory more efficient than gaEPS^T. The spacestamp hashmaps use 8 Bytes for a filled bucket (the key pointer and the count) and 4 Bytes for an empty bucket (the "empty" key). Thus, the average overhead per space point is $4 * \frac{n_e}{n_f}$ Bytes, where n_e is the number of empty buckets and n_f is the number of filled buckets. The maximum overhead is when the hashmap has just resized and 75% of buckets are empty. Thus, the maximum overhead per space point reaches the overhead of gaEPS^T (i.e., 12 Bytes), but most of the time it is smaller.

The sparse hashmap-based gaEPS^H is the most efficient implementation. It uses 8 Bytes for a filled bucket and only 2 Bytes for an empty bucket, yielding almost no overhead. On average, the sparse hashmap-based gaEPS^H uses approximately 45% of the memory that is needed by gaEPS^T.

Summarizing, from Fig. 25 and 26 we conclude that both gaEPS^T and gaEPS^H scale well in terms of runtime (linear) and very well in terms of memory usage (constant).

7.4 Summary of the Experimental Results

We draw the following conclusions from our experimental results. gaEPS takes advantage of coarse query granularities and clearly outperforms the basic EPS both in terms of runtime and memory usage. For the memory requirements the improvements are very significant. The memory consumption of gaEPS is independent of the database size and consumes only a tiny fraction of the memory that is required by the basic EPS (which depends on the number of input tuples). As for the different gaEPS implementations, there is a trade-off between speed and memory usage. The gaEPS^T implementation is less memory efficient, but it is the fastest. The sparse hashmap-based gaEPS^H consumes least memory, but it is the slowest implementation. The dense hashmap-based gaEPS^H is somewhere in the middle: it is slightly slower than gaEPS^T, but requires a bit less memory. The insertion time for hashmaps is not constant: the resize overhead is quite significant and linear to the hashmap size.

8 Conclusions and Future Work

Many applications of spatiotemporal databases (e.g., traffic data analysis, land management, and weather monitoring) require support for sequenced spatiotemporal aggregation (SSTA). Conceptually, an SSTA query returns one aggregate value for each spatiotemporal granule at the query granularity. Typically, the query granularity is much coarser than the data granularity, because the purpose of such queries is to produce compact summaries. Thus, a query evaluation algorithm must convert from the data granularity to the query granularity. This paper proposes a formal definition of SSTA that includes a data-to-query granularity conversion. Based on a discrete time model and a discrete, 1.5 dimensional space model that represents a road network, we generalize the concept of (time) constant intervals towards constant rectangles that represent maximal rectangles in the spatiotemporal domain over which the aggregation result is constant. We propose an efficient algorithm to compute SSTA queries for the COUNT, SUM, AVG, MIN, and MAX aggregation functions. The algorithm is based on the plain sweep paradigm, which requires two data structures: an Event Point Schedule (EPS) and a Sweep Line Status (SLS). We propose an efficient EPS (termed gaEPS) and an SLS (termed gaSLS), which maintain the corner points of the input tuples together with a compact summary. Our experiments show that the proposed solution takes advantage of coarse query granularities and clearly outperforms a baseline solution both in terms of memory usage and runtime.

Future work points in several directions. First, the current method performs precise aggregation, that is, a set of spatiotemporal granules is coalesced only if we have exactly the same aggregate value for each granule in the set. We plan to extend our method for approximate aggregation, where a set of spatiotemporal granules is coalesced if the aggregate value of the granule in a set fall into some range. Second, the order in which dimensions are processed is fixed: we sweep along the time dimension. It would be interesting to develop a cost model that helps to determine along which dimension to sweep. Third, currently each query is processed from scratch, without reusing results of previous queries. We want to develop techniques for the incremental maintenance of query results. Fourth, it would be interesting to run experiments on real-world GPS logs and provide a disk-based implementation of our method. Finally, the current method does double-counting (i.e., the same data is sometimes counted (or summed, depending on the used aggregation function) several times). It would be interesting to develop solutions that avoid double-counting.

9 Acknowledgments

This work was funded in part by the Swiss National Science Foundation (SNSF) through the Tameus project (proposal no 200021_135361).

References

1. R. Bayer. Binary b-trees for virtual memory. In *ACM SIGFIDET*, pages 219–235, 1971.
2. B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An asymptotically optimal multiversion b-tree. *VLDB J.*, 5(4):264–275, 1996.
3. J. L. Bentley. *Algorithms for Klee's rectangle problems*. Computer Science Department, Carnegie-Mellon University, Pittsburgh, 1977.
4. M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.
5. T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.
6. T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. The MIT Press, September 2001.
7. S. Dieker and R. H. Güting. Plug and play with query algebras: SECONDO – a generic DBMS development environment. In *IDEAS*, pages 380–392, 2000.
8. H. Edelsbrunner. *Dynamic Rectangle Intersection Searching*. Institute for Information Processing Rept. 47, Technical University of Graz, Austria, 1980.
9. S. Geffner, D. Agrawal, A. El Abbadi, and T. Smith. Relative prefix sums: an efficient approach for querying dynamic olap data cubes. In *Proc. of the ICDE-99*, pages 328–335, Mar. 1999.
10. Google. Google's sparsehash project. http://code.google.com/p/google-sparsehash/. Current as of December 12, 2008.
11. A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.
12. J. M. Hellerstein, P. J. Haas, and H. J. Wang. Online aggregation. In *Proc. of SIGMOD-97*, pages 171–182, 1997.
13. C. S. Jensen, K.-J. Lee, S. Pakalnis, and S. Šaltenis. Advanced tracking of vehicles. In *European Congress and Exhibition on ITS*, page 12 pages, 2005.
14. N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, 1995.
15. I. F. V. Lopez, R. T. Snodgrass, and B. Moon. Spatiotemporal aggregate computation: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 17(2):271–286, 2005.
16. E. M. McCreight. Priority search trees. *SIAM J. Comput.*, 14(2):257–276, 1985.
17. B. Moon, I. F. V. López, and V. Immanuel. Efficient algorithms for large-scale temporal aggregation. *IEEE Trans. Knowl. Data Eng.*, 15(3):744–759, 2003.
18. D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.
19. D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.
20. H. Samet. Hierarchical representations of collections of small rectangles. *ACM Comput. Surv.*, 20(4):271–309, 1988.
21. J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the past, the present, and the future in spatio-temporal databases. In *ICDE*, pages 202–213, 2004.
22. Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–226, 2004.
23. Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *EDBT*, pages 682–700, 2002.
24. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *ICDE*, pages 51–60, 2001.
25. J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. *The VLDB Journal*, 12:262–283, 2003.
26. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. Efficient computation of temporal aggregates with range predicates. In *PODS*, 2001.
27. D. Zhang, A. Markowetz, V. J. Tsotras, D. Gunopulos, and B. Seeger. On computing temporal aggregates with range predicates. *ACM Trans. Database Syst.*, 33(2), 2008.
28. 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, pages 88–93, 2001.
29. D. Zhang, V. J. Tsotras, and D. Gunopulos. Efficient aggregation over objects with extent. In *PODS*, pages 121–132, 2002.