Sequenced Spatio-Temporal Aggregation in Road Networks

Igor Timko   Michael H. Böhlen   Johann Gamper
Faculty of Computer Science, Free University of Bozen-Bolzano, Italy
{timko, boehlen, gamper}@inf.unibz.it

ABSTRACT

Many applications of spatio-temporal databases require support for sequenced spatio-temporal (SST) aggregation, e.g., when analyzing traffic density in a city. Conceptually, an SST aggregation produces one aggregate value for each point in time and space.

This paper is the first to propose a method to efficiently evaluate SST aggregation queries for the COUNT, SUM, and AVG aggregation functions. Based on a discrete time model and a discrete, 1.5 dimensional space model that represents a road network, we generalize the concept of (temporal) constant intervals towards constant rectangles that represent maximal rectangles in the space-time domain over which the aggregation result is constant. We propose a new data structure, termed SST-tree, which extends the Balanced Tree for one-dimensional temporal aggregation towards the support for two-dimensional, spatio-temporal aggregation. The main feature of the Balanced Tree to store constant intervals in a compact way by using two counters is extended towards a compact representation of constant rectangles in the space-time domain. We propose and evaluate two variants of the SST-tree. The SST^T-tree and SST^H-tree use trees and hashmaps to manage spacestamps, respectively. Our experiments show that both solutions outperform a brute force approach in terms of memory and time. The SST^H-tree is more efficient in terms of memory, whereas the SST^T-tree is more efficient in terms of time.

1. INTRODUCTION

Spatio-temporal databases are becoming more and more popular in various application domains, including traffic data analysis. The proliferation of Global Positioning System (GPS) technology facilitates the tracking of car positions, and huge amounts of traffic data is collected. Commercially available tracking systems operate according to the following scenario [12]. Each car is equipped with a GPS receiver and periodically sends its position and current timestamp to a central server.

In such an application scenario, *sequenced spatio-temporal (SST) aggregation* can be used to obtain a summary of the traffic density in a city/region. Conceptually, an SST aggregation produces one aggregate value for each point in time and space. Con-

(a) 5:00PM – 5:10PM

(b) 5:11PM – 5:15PM

Figure 1: Road Traffic Analysis: "For each road, what is the number of cars at each point in time (between 5:00PM and 5:15PM) and space (position on road network)?"

sider the following example query: "For each road, what is the number of cars at each point in time and space?". Figure 1[1] illustrates a typical result of this query, 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 density of the traffic on that segment, varying from very low traffic (no line) to moderate traffic (gray line) and jammed traffic (black line).

This paper is the first to formally define SST aggregation and to present an efficient evaluation algorithm to evaluate SST aggre-

---

[1]In order to create this figure, we used a PNG image file produced by Google Maps.

gation queries for the COUNT, SUM, and AVG aggregation functions. The algorithm relies on the efficient computation of so-called *constant rectangles*, which are the 2D generalization of constant intervals as used in temporal aggregation [17]. A constant rectangle is defined as a pair of a spatial and a temporal interval such that an aggregate value is constant at all space-time points inside the rectangle spanned by the two intervals. For example, in Figure 1(a) each line segment with a different shade of gray represents a constant rectangle. All of them have the same temporal interval [5:00;5:10], but different spatial intervals. One of these rectangle's spatial interval is marked as "Corso Italia, [0;100]" and represents a segment on the street "Corso Italia", stretching 100 meters from the beginning of the road. Figure 1(b) shows the traffic density over the same part of the road network, but during the time period [5:11;5:15]. Notice that also the spatial extension of the constant rectangles changed.

To efficiently compute SST aggregates, we propose a data structure called *Sequenced Spatio-Temporal Tree* (SST-tree). The SST-tree is a two-dimensional extension of the Balanced Tree [17]. The main motivation for choosing the Balanced Tree as the starting point for the new data structure is its efficient handling of duplicate (time) points, which due to the use of a discrete space and time model are common in GPS logs. Many cars send their position updates within the same minute, and many cars might be on the same 100-meter long road segment. The SST-tree allows to store spatio-temporal information in a compact way using a minimal set of counters and supporting an efficient evaluation of aggregate functions. We propose two different variants of the SST-tree: the SST^T-tree, which uses trees to store the spatial information, and the SST^H-tree, which uses hashmaps to store the spatial information.

The evaluation algorithm works in two steps. First, the input tuples are scanned and an SST-tree is constructed. Second, the constant rectangles are computed by traversing the SST-tree. Thereby, a so-called *dynamic tree* is used to collect and compute the spatial component of constant rectangles from the partial information stored in the SST-tree.

We implemented the new framework for spatio-temporal aggregation on top of the Secondo DBMS [6], and we conducted several experiments, mainly to compare the different variants of the SST-tree. The results of the experiments show that the SST^T-tree is generally faster, while the SST^H-tree uses less memory. Another, interesting observation is that hashmaps do not give constant insert/lookup time due to the resize overhead of hashmaps. Our experiments also shows that both SST-tree implementations outperform a brute force approach, both in terms of memory and time usage.

The rest of the paper is structured as follows. Section 2 discusses preliminaries, including the time and space models, spatio-temporal uncertainty, and the adopted spatio-temporal data model. Section 3 discusses related work. In Section 4, we introduce and formally define the SST aggregation operator. Section 5 presents basic ideas of query processing, which in Section 6 are progressed towards the SST-tree as the main data structure for SST query evaluation. In Section 7, we present a query evaluation algorithm based on the SST-tree. Section 8 presents the results of a first experimental evaluation of our framework. Finally, Section 9 concludes the paper and points to future work.

2. PRELIMINARIES

We use a *discrete time model*. The time line is composed of a finite sequence of atomic *time granules*, denoted as Δ^T. A timestamp (or time interval) is a convex set of time granules and is represented as $T = [T_s, T_f)$, where T_s is its inclusive start point and T_f its exclusive finish point. Such a model is well suited for and widely used in temporal database research [21].

We use a *discrete, 1.5-dimensional space model*. The space consists of a road network, i.e., a finite set of roads. Each road is a 1D line and is divided into a finite sequence of atomic line segments, termed *space granules*. The set of all space granules is referred to as Δ^S. A spacestamp (or space interval) is a convex set of space granules and is represented as $S = [S_b, S_e)$, where S_b is its inclusive beginning position and S_e its exclusive ending position. A specific *position* in our space model is represented as a pair (rid, g), where rid is a road ID and g is a granule number. A 1.5-dimensional space model is frequently used in location-based services, as content is typically positioned with respect to a transportation infrastructure [10, 18].

We assume an application scenario with GPS-based tracking of car positions [12]. Each car is equipped with a GPS receiver. It periodically reads its actual position from the receiver and sends a message that contains the position data and the current time point to a central server. Typically, a time-based update policy is used, requiring that each car sends the updates at regular time intervals (e.g., every 3 seconds).

Example 1. Figure 2(a) illustrates a road with road-ID 1101. The road is subdivided in 12 space granules, and there are three measurements for a specific car, reporting that the car was in space granule 1, space granule 6, and space granule 10. These car positions are represented as $(1101, 1)$, $(1101, 6)$, and $(1101, 10)$ and are indicated by solid lines over the corresponding granules. □

(a) Road and Car Positions

(b) Spatial Interpolation
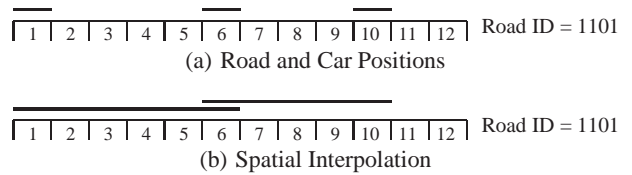
Figure 2: Space Model: a Road and Car Positions.

The car messages sent to the server are the source of *spatio-temporal uncertainty*, that is, we do not know how the position of a car is changing between the measurement points [12]. Therefore, two consecutive car messages are interpreted as follows: at any time between the two measurements the car is somewhere between the two space granules that have been reported (including the two positions), assuming that a car cannot turn back. In other words, we do a spatial interpolation and represent the possible car positions as spatial intervals. Figure 2(b) illustrates this interpolation for the measurements in Figure 2(a). Notice that position 6 is included in two spatial intervals, because the car may also stop at this position for a while.

Given the above interpretation of car messages, we represent a history of car movements in a relational context by a *Spatio-Temporal Data Model* (STDM), which has an explicit timestamp and spacestamp attribute, respectively. An STDM relation schema is given as $(A_1, \ldots, A_k, T, S)$, where $A_1, \ldots, A_k$ are explicit (non-spatial and non-temporal) attributes, $T = [T_s, T_f)$ is the valid timestamp attribute, and $S = [S_b, S_e)$ is the spacestamp attribute. The product of a space and time interval, $S \times T$, is called *spatio-temporal rectangle*. The fact represented by an STDM tuple is valid during each spatio-temporal granule in the corresponding spatio-temporal rectangle.

	CID	RID	T	S
r1	1	1101	[1;4)	[1;7)
r2	1	1101	[4;7)	[6;11)
r3	2	1101	[3;6)	[3;8)
r4	2	1101	[6;9)	[7;11)
r5	3	1101	[3;6)	[6;9)
r6	3	1101	[6;9)	[8;11)

(a) Tabular Representation
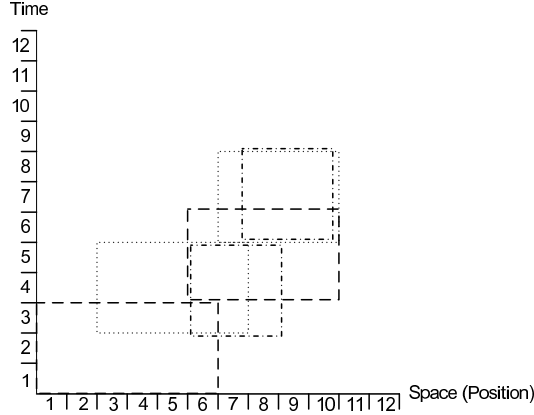
(b) Graphical Representation

Figure 3: STDM Relation, CARS, Storing Car Movements.

Example 2. Figure 3(a) shows the STDM relation, CARS, that stores a history of car movements. *RID* (road ID) and *CID* (car ID) are the two explicit attributes, and *T* and *S* are the time- and space-stamp attributes, respectively. The first tuple, $r1$, represents that the car with ID 1 is on the road with ID 1101 in the spatio-temporal rectangle $[1;4) \times [1;7)$. The second tuple, $r2$, stores the position of the same car in a later time period. Figure 3(b) shows a graphical representation, where the spatio-temporal rectangles are drawn as boxes. Spatio-temporal rectangles that represent the same car have the same border style. We use the CARS relation as a running example throughout the rest of the paper. □

3. RELATED WORK

Conceptually, a sequenced, spatio-temporal (SST) aggregation produces one aggregate per spatio-temporal granule. In the following discussion, if not stated otherwise, we assume that our 1.5D space consists of one space line. Thus in total, we have one space line and one time line (i.e., a 2D plane). This assumption does not limit the generality, because an SST aggregation query is processed independently for each space line.

3.1 Spatial Aggregation

Previous research work on spatial aggregation [19, 25] concentrates on range (or box) aggregation in a two-dimensional space, which computes an aggregate function over all spatial objects that fall into the query region. The *aR-tree* [19] is based on the R-tree [9] and maintains for each R-tree bounding box the total number of objects (for the COUNT aggregation function) 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* [25] avoids this disadvantage at the expense of extra memory usage. It

transforms spatial objects into objects in the (key, time) plane and computes then the aggregation results by using the MVB-tree [2]. 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 temporal granule. A number of methods for one-dimensional temporal aggregation have been developed [3, 13, 17]. The main problem tackled by these methods is the efficient computation of temporal 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) [13] 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. Thereby, 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 into 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 start and finish time points of the tuple's timestamp intervals are sorted by inserting them into a balanced binary search tree. Each node of the tree stores a time point and two counters, namely 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 only the constant intervals are stored.

The *TMDA operator* [3] improves over the A-tree and Balanced Tree methods in that it consumes less memory on average and still provides the same running time. This 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.

While the above frameworks pursue memory-based solutions of temporal aggregation, disk-based index structures for temporal aggregation have also been investigated. The *SB-tree* [28] supports 1D, sequenced and cumulative temporal aggregation. The tree maintains a hierarchy of temporal 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* [29] 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 Spatio-Temporal Aggregation

Past work on spatio-temporal aggregation [20, 23, 24] assumes a 2D space model and concentrates on spatio-temporal box aggregation, which is a generalization of spatial box aggregation. Given a spatial region and a time interval, an aggregation is computed over all spatio-temporal objects that are present in that region during that time interval. There is no work on SST aggregation.

The *aRB-tree* [20] extends the aR-tree with a temporal 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 bounding intervals of the B-tree. This eliminates the need to traverse the subtree of nodes that are totally enclosed by the query region. A disadvantage of the aRB-tree is that it allows double counting. 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* [24] avoids double counting. This index modifies the aRB-tree: instead of the number of objects, the sketch (compressed representation) of the objects' IDs is kept for each B-tree's bounding interval. However, the sketch index is generally larger than the aRB-tree. Moreover, the sketch index only answers queries approximately.

The *Adaptive Multi-Dimensional Histogram* (AMH) [23] 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. For speeding 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 Other Issues

Most previous proposals for spatial, temporal, and spatio-temporal aggregation are not implemented in a DBMS. No commercial DBMS supports spatio-temporal aggregation. As for the research prototypes, they provide only limited support. *Domino* [26] focuses on spatio-temporal range queries rather than on aggregation. To the best of our knowledge, *Secondo* [6] is the most versatile spatio-temporal DBMS, because it implements a great number of algorithms for spatio-temporal (moving) objects [15]. However, Secondo does not have an implementation of an *efficient* algorithm for SST aggregation. We implement our data structures and algorithms as a module of Secondo, but our implementation can be easily ported to any other relational DBMS that supports integer attribute types.

Many past works on spatio-temporal queries for road networks [8, 10, 11, 22, 27] adopt the 1.5-dimensional space model, which is also used in this paper. The general idea is as follows: if data points (e.g., cars) are never outside a road network, then users are not interested in the space outside the network; consequently, we can prune the search space by modeling data and performing computations in the 1.5D space model instead of the 2D space model. Recently, the 1.5D space model has been implemented by a commercial DBMS [14].

4. DEFINING SST AGGREGATION

Informally, *sequenced spatio-temporal (SST) aggregation* is defined as follows [16]: group the input tuples by spatio-temporal granules, one group per granule, and apply one or more aggregation functions to each group. Next, we provide a formal definition

	Cnt	T	S
1	1	[1;3)	[1;7)
2	1	[3;4)	[1;3)
3	2	[3;4)	[3;6)
4	3	[3;4)	[6;7)
5	2	[3;4)	[7;8)
6	1	[3;4)	[8;9)
7	1	[4;6)	[3;6)
8	3	[4;6)	[6;8)
9	2	[4;6)	[8;9)
10	1	[4;6)	[9;11)
11	1	[6;7)	[6;7)
12	2	[6;7)	[7;8)
13	3	[6;7)	[8;11)
14	1	[7;9)	[7;8)
15	2	[7;9)	[8;11)

(a) Tabular Representation
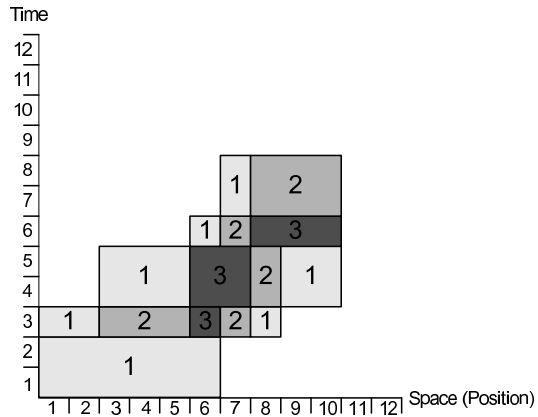
(b) Graphical Representation

Figure 4: Result of SST Aggregation.

of this type of aggregation.

Definition 1. [SST Aggregation] Let R be a STDM relation with schema $(A_1, \ldots, A_k, T, S)$, where $A_1, \ldots, A_k$ are explicit attributes, T is the timestamp attribute, and S is the spacestamp attribute, and let $F = \{f_1/C_1, \ldots, f_k/C_k\}$ be a set of aggregate functions. Further, let $g \in \Delta^T \times \Delta^S$ be a spatio-temporal granule and $R_g = \{r \mid r \in R \wedge g \in r.T \times r.S\}$ be the aggregation group of g that contains all tuples of R whose spatio-temporal rectangle contains g. Then the *SST aggregation* operator, $\mathcal{G}^{SST}[F]R$, is defined as

$$\mathcal{G}^{SST}[F]R = \{x \mid g \in \Delta^T \times \Delta^S \wedge R_g \neq \emptyset \wedge$$
$$x = (f_1(R_g), \ldots, f_i(R_g), g.T, g.S)\}$$

The result relation has the schema $(C_1, \ldots, C_k, T, S)$. □

For each spatio-temporal granule, g, the SST aggregation operator evaluates the aggregate functions, F, over the set of all tuples that are valid at g (i.e., the tuples that have a spatio-temporal rectangle that contains g). Each $f_i/C_i \in F$ is some aggregate function that takes an STDM relation as argument and applies aggregation to one of the relation's attributes; the aggregation result is stored as the value of an attribute C_i.

To obtain a more compact representation, tuples with adjacent granules and equal aggregate results are coalesced into maximal spatio-temporal rectangles, termed *constant rectangles*. This is similar to the concept of constant intervals in 1D temporal aggregation [4], the major difference being that coalescing in 2D is not
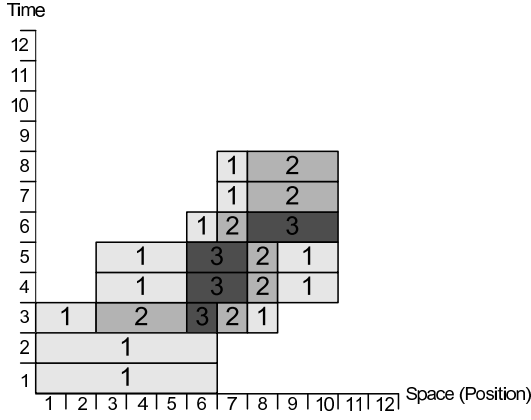
Figure 5: Result of the Brute Force Approach.

(a)

(b)

Figure 6: (a) $\pi[CID, RID, T]$CARS and (b) its Balanced Tree

unique. That is, the size and the shape of the constant rectangle depends on the order in which the two dimensions are coalesced. Though the coalescing step is not included in the above definition of SST aggregation, throughout the paper we assume coalescing, first along the spatial dimension and then along the temporal dimension.

Thoughout the rest of the paper we assume the COUNT aggregation function. SUM is a straightforward generalization of COUNT, and AVG can be computed from SUM and COUNT.

Example 3. Consider the STDM relation in Figure 3 and the following SST query: *"At each point in time, what is the number of cars at each point on the road with road ID 1101?"*. This query can be expressed as $\mathcal{G}^{SST}[count(*)/Cnt]$CARS. Its result after coalescing adjacent tuples with identical aggregation results is shown in Figure 4(a) as an STDM relation. Each tuple represents a constant rectangle and the number of cars in that rectangle. For example, the first tuple says that in the rectangle $[1;3) \times [1;7)$ the number of cars in each spatio-temporal granule is equal to 1. Figure 4(b) shows a graphical illustration of the query result. The spatio-temporal rectangles of the result tuples are drawn as boxes with the aggregate result inside. If temporal coalescing would have been applied before spatial coalescing, the first result tuple starting in the lower left corner would be $(1, [1, 4), [1, 3))$. □

5. IDEAS OF SST AGGREGATION QUERY PROCESSING

5.1 Brute Force

While the evaluation of sequenced one-dimensional aggregation has been studied quite extensively in the past, to the best of our knowledge there is no work on the evaluation of sequenced two-dimensional aggregation. A *brute force approach* to evaluate SST aggregation could be the following: (1) for each temporal granule, determine the group of all tuples that are valid during that granule, and (2) for each such group of tuples perform sequenced one-dimensional aggregation using one of the known spatial or temporal aggregation frameworks. This brute force approach is obviously not very efficient, because it runs an aggregation procedure once for each point in time. Moreover, the result is only coalesced along the spatial dimension and not along the temporal dimension. Figure 5 shows the result of applying the brute force approach in our running example. Notice the difference to the intended result as illustrated in Figure 4.
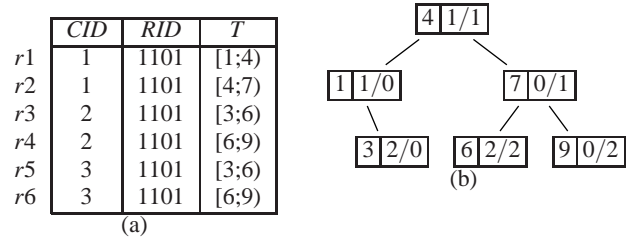
5.2 Towards Efficient Query Processing

The basic idea of our approach to process SST aggregation queries is to conceptually separate the temporal and spatial dimension, which are orthogonal and can be handled in the same way. First, we ignore the spatial part of the input tuples and do the temporal sequenced aggregation, computing in this way temporal constant intervals. Second, for each constant temporal interval we determine the group of all tuples that are valid during that interval. Third, for each group of tuples we ignore the temporal part and do the spatial sequenced aggregation.

5.2.1 Balanced Tree for Constant Temporal Intervals

The computation of aggregates over constant temporal intervals has been studied in the past, and we adopt the Balanced Tree algorithm [17] which works in two steps (the following description assumes the computation of the COUNT aggregation). First, as the input tuples are scanned, their start and finish times are extracted and stored in a Balanced Tree together with two counters: 1) a start counter that stores the number of tuples that *start* at this time point and 2) a finish counter that stores the number of tuples that *finish* at this time point. When all input tuples have been processed, the tree contains all start and finish points of the tuples, which represent also the start and finish points of the constant intervals. Second, by performing an in-order traversal of the tree, the values of the counters are combined and the temporal aggregation results are produced. Whenever a node, v, is visited, the aggregate value is incremented by the value of the start counter and decremented by the value of the finish counter, yielding a result tuple over the constant interval that is formed by the time point of v and the time point of its successor.

Example 4. Figure 6(b) shows the Balanced Tree for the CARS relation in Figure 3 after removing the spacestamp attribute, i.e., for $\pi[CID, RID, T]$CARS in Figure 6(a). The set of distinct starting and finishing time points extracted from the input tuples is $\{1, 3, 4, 6, 7, 9\}$. Each of these time points is stored in a separate node together with the associated start- and finish counter. For instance, two tuples start and two tuples finish at time 6. The in-order traversal starts at the node with time 1 (the smallest time point in the tree) and an initial aggregate value of 0. The aggregate value is now incremented by 1 (value of start counter) and decremented by 0 (value of finish counter), yielding the aggregate value 1. Thus, the first result tuple is $(1, [1, 3))$. Next, the node with time 3 is visited. Incrementing the aggregate value by 2 and decrementing by 0, yields the second result tuple $(3, [3, 4))$. Overall, the following constant temporal intervals are produced: $[1; 3)$, $[3; 4)$, $[4; 6)$, $[6; 7)$, and $[7; 9)$ with the associated aggregate values. □

5.2.2 Naively Extended Balanced Tree

In a first, naive extension of the Balanced Tree for 2D spatio-temporal aggregation we substitute the counters in the nodes by the set of all tuples that are valid at the node's time point (and consequently are valid throughout the constant interval that starts at the node's time point). In fact, for the count aggregation as in the running example, it is sufficient to store the spatial intervals of these tuples. When a node, v, is visited during the traversal of the tree, instead of accumulating the counters, the sequenced spatial aggregation over all tuples that are stored in v is computed. Obviously, the same Balanced Tree algorithm can be applied for the spatial aggregation as well.

Example 5. The Naively Extended Balanced Tree of the CARS relation is shown in Figure 7. Each node stores a time point and the set of spatial intervals that are extracted from all input tuples that are valid at the node's time point. For example, the spacestamps $[1;7)$, $[3;8)$, and $[6;9)$ are extracted from the tuples $r1, r3,$ and $r5$ and associated with time 3. The tree traversal begins at the node with time 1, which has associated a single spatial interval, $[1,7)$. The spatial aggregation over this interval yields the spatial result tuple $(1,[1;7))$, which is then combined with the constant temporal interval, $[1,3)$, to produce the first result tuple, $(1,[1;3),[1;7))$. Next, the node with time 3 is visited. The processing of the associated space intervals produces five constant space intervals with aggregate values, which in combination with the constant temporal interval $[3,4)$ produce the result tuples 2–6 in Figure 4. □
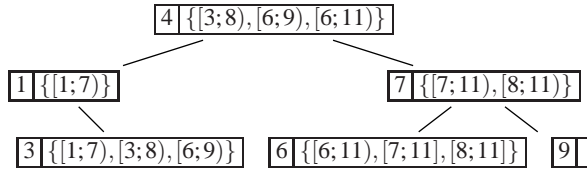

Figure 7: The Naively Extended Balanced Tree.

5.2.3 Optimizations

Obviously, the Naively Extended Balanced Tree is rather inefficient, and we can think of several optimizations:

1. First, each aggregation group over a constant time interval shares some tuples with the aggregation group of the preceding and/or next constant interval. For instance, in Figure 7 the nodes with time point 3 and 4 share the two space intervals $[3;8)$ and $[6;9)$. Therefore, an incremental approach can be applied to store only those tuples that start and finish at the node's time point (e.g., only the space intervals $[6;11)$ and $[1;7)$ need to be stored at the node with time 4).

2. Second, we can adopt the same idea as in the Balanced Tree to obtain a more compact representation, namely to store space points instead of space intervals and to store each distinct space position only once together with four counters. Two counters indicate how many space intervals begin and end at this position for the tuples that start at this time point. The other two counters record the same information for the tuples that finish at this time point. For example, in Figure 7 at the node with time point 6, compared to the previous node (time point 4), two new space intervals appear, $[7;11)$ and $[8;11)$. These intervals store the finish position, 11, two times. This can be avoided by storing it only once together

with the information that two tuples are finishing at that position. In this paper we will advance this idea even further by reducing the two begin counters to only one counter that stores the difference between the original counters. The same reduction is possible for the two end counters.

In Section 6, we explore these optimizations and introduce a data structure that supports a more efficient evaluation of SST aggregation.

6. THE SST-TREE

In this section, we present a data structure, called the SST-tree, that supports an efficient evaluation of SST aggregation queries. In Section 6.1, we define a generic version of the SST-tree. In Sections 6.2 and 6.3, we describe two variants of the SST-tree, which differ in how tree nodes are implemented.

6.1 General Definition

The *Sequenced Spatio-Temporal Tree* (SST-tree) extends the Balanced Tree and applies the two optimizations mentioned in Section 5.2.3. The main difference is that instead of storing time points with counters, the SST-tree stores both time points and space points, and the space points are associated with counters. By combining these pieces of information the constant spatio-temporal rectangles and the associated aggregation value can be computed.

The first optimization concerns the removal of redundant spatial intervals in the tree nodes. Instead of storing at a node with time t the spatial intervals of all tuples that are valid at time t, we store only the spatial intervals of those tuples that start and finish at time t. Thereby, it is important to distinguish these two sets of tuples.

Example 6. Figure 8 shows the naively extended balanced tree after applying the first optimization step. For instance, the spatial interval $[1,7)$ produced by tuple $r1$ can be removed from the node with time 3, since $r1$ starts at time 1 and finishes at time 4. Thus, the node becomes $(3,\{[3;8),[6;9)\}/\{\})$, stating that two tuples with spatial intervals $[3;8)$ and $[6;9)$ start at time 3 and that no tuple finishes. Similar, the node at time 4 can be reduced to $(4,\{[6;11)\}/\{[1;7)\})$.
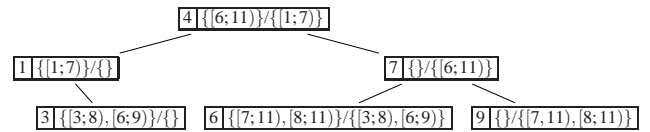

Figure 8: The Naively Extended Balanced Tree after Optimization 1.

To recover the removed space intervals for the computation of aggregate values, the traversal of the tree requires a dynamic data structure to keep track of the tuples that started before and to remove them when they finish. This leads to an incremental computation of the result tuples in chronological order.

The second optimization is to replace the spatial intervals in the nodes of the Naively Extended Balanced Tree by the begin and end points of the intervals together with counters, similar to what the Balanced Tree is doing for the one-dimensional case. Thus, we extend the idea of the Balanced Tree for the two-dimensional case, where we have to compute constant (spatio-temporal) rectangles instead of constant intervals. Recall that a node with time t in the 1D Balanced Tree stores two counters that store the cardinality of
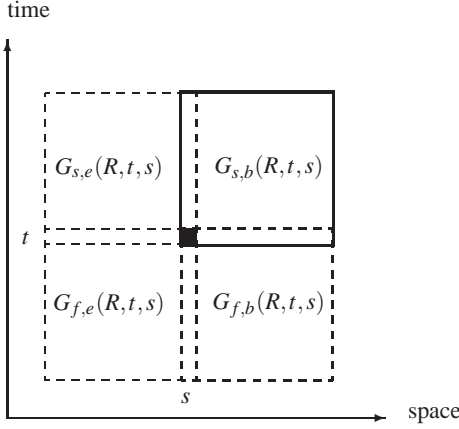
Figure 9: Graphical Illustration of the Groups of Tuples.

the following two sets:

$$G_s(R,t) = \{r \mid r \in R \wedge r.T_s = t\}$$
$$G_f(R,t) = \{r \mid r \in R \wedge r.T_f = t\}$$

That is, the set of all tuples that start at t (the start counter) and the set of all tuples that finish at t (the finish counter).

Computing constant rectangles in 2D can conceptually be seen as computing constant intervals along both dimensions and then combining corresponding intervals. Therefore, we need a counter for each combination of the two dimensions, i.e.,

$$\begin{Bmatrix} T_s \\ T_f \end{Bmatrix} \times \begin{Bmatrix} S_b \\ S_e \end{Bmatrix}$$

The following Definition 2 specifies the corresponding groups of tuples.

Definition 2. [Tuple groups] Let R be an STDM relation with schema $(A_1,\dots,A_k,T,S)$, where $A_1,\dots,A_k$ are the explicit (non-temporal and non-spatial) attributes, T is the timestamp attribute, and S is the spacestamp attribute. Further, let t be a temporal granule and s be a spatial granule. Then we define the following groups of tuples for t and s:

$$G_{s,b}(R,t,s) = \{r \mid r \in R \wedge r.T_s = t \wedge r.S_b = s\}$$
$$G_{s,e}(R,t,s) = \{r \mid r \in R \wedge r.T_s = t \wedge r.S_e = s\}$$
$$G_{f,b}(R,t,s) = \{r \mid r \in R \wedge r.T_f = t \wedge r.S_b = s\}$$
$$G_{f,e}(R,t,s) = \{r \mid r \in R \wedge r.T_f = t \wedge r.S_e = s\}$$

□

The four groups correspond to the four corners of a spatio-temporal rectangle. This is graphically illustrated in Figure 9. The small black rectangle represents the spatio-temporal granule (t,s). The large boxes indicate the spatio-temporal rectangles that are collected in the four groups. For instance, the box with solid lines represents the first group, $G_{s,b}(R,t,s)$, that contains all tuples from R that start at time point t and begin at space point s, that is, for which the spatio-temporal granule (t,s) forms the lower-left corner in the spatio-temporal rectangle.

Example 7. Consider the CARS relation in Figure 3. The four groups for the time granule 3 and space granule 6 are computed as follows: $G_{s,b}(\text{CARS},3,6) = \{r5\}$ and the other three groups are

empty. For time granule 6 and space granule 8 we get the following groups: $G_{s,b}(\text{CARS},6,8) = \{r6\}$, $G_{f,e}(\text{CARS},6,8) = \{r3\}$, while the other two groups are empty. □

The four groups introduced above form the basis for the incremental computation of the aggregate values. For each spatio-temporal granule in the SST-tree, the cardinality of these groups is stored as a counter, which during the traversal of the tree are combined to compute the result tuples.

A further reduction of the data that needs to be stored with each node is possible. Instead of storing all four counters, we pairwise combine these counters and store only the difference. This reduction is applied in the following definition of an SST-tree.

Definition 3. [SST-tree] Let R be an STDM relation with schema $(A_1,\dots,A_k,T,S)$, where $A_1,\dots,A_k$ are the explicit (non-temporal and non-spatial) attributes, T is the timestamp attribute, and S is a spacestamp attribute. Further, let $P_T = \{t \mid r \in R \wedge (t = r.T_s \vee t = r.T_f)\}$ be the set of all start and finish time points in R and $P_S = \{s \mid r \in R \wedge (s = r.S_b \vee s = r.S_e)\}$ be the set of all begin and end space points in R. Then, an *SST-tree* for relation R is a balanced binary search tree. A node in the tree stores a pair (t,S_t), where $t \in P_T$ is the node's key and represents a start/finish time point and S_t is a set of space positions together with two counters and is defined as

$$S_t = \{(s,cnt_b,cnt_e) \mid s \in P_S \wedge$$
$$cnt_b = |G_{s,b}(R,t,s)| - |G_{f,b}(R,t,s)| \wedge$$
$$cnt_e = |G_{s,e}(R,t,s)| - |G_{f,e}(R,t,s)|\}$$

□

Thus, an SST-tree has a node for each distinct (start or finish) time point in relation R. Each node that represents a specific time point, t, stores a set, S_t, of triples of the form (s,cnt_b,cnt_e), where cnt_b and cnt_e are two counters. cnt_b is the difference between two numbers: (1) the number of tuples that *begin* at position s and *start* at time t and (2) the number of tuples that *begin* at position s and *finish* at time t. cnt_e is also the difference between two numbers: (1) the number of tuples that *end* at position s and *start* at time t and (2) the number of tuples that *end* at position s and *finish* at time t. Different from the one-dimensional Balanced Tree, here the counters might have negative values.

Example 8. Figure 10 illustrates the SST-tree for the CARS relation. Notice the difference to the tree in Figure 8. The spatial intervals are replaced by a space position, representing the beginning and ending positions of the intervals, and two counters. For instance, the root node with time point 4 stores a set of four space points and associated counters. The pair $(1,-1/0)$ states that 1) the difference between the number of tuples that begin at position 1 and start at time 4 and the number of tuples that begin at position 1 and finish at time 4 is -1 and 2) the difference between the number of tuples that end at position 1 and start at time 4 and the number of tuples that end at position 1 and finish at time 4 is 0. □
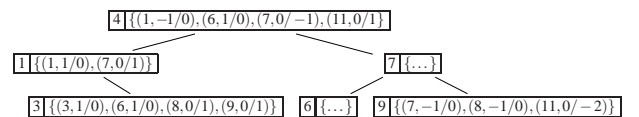

Figure 10: SST-tree.

In the following, we introduce two different techniques to efficiently implement the space points and associated counters in an SST-tree.

6.2 SST-tree with Tree-Based Nodes

In a SST-tree with Tree-Based Nodes (SST^T-tree), the information about tuples that start or finish at a node is stored in this node's *spacestamp tree*. Thus, a node contains one time point and one tree.

Definition 4. [SST^T-tree] Let R be an STDM relation with schema $(A_1, \ldots, A_k, T, S)$. An *SST^T-tree* for relation R is defined as an SST-tree, where each node, (t, S_t), uses a balanced binary search tree, termed *spacestamp tree*, to store the set, S_t, of space positions and counters. A node of the spacestamp tree is given as (s, cnt_b, cnt_e), where s is the node's key. □

Example 9. Figure 11 depicts an SST^T-tree that contains information about the relation, CARS, from Figure 3. For each node, a number denotes its time point and a string indicates its spacestamp tree pointer. In the figure, we can see the spacestamp trees, $T3$ and $T9$, of the nodes with time points 3 and 9, respectively. This SST^T-tree corresponds to the SST-tree from Figure 10. □
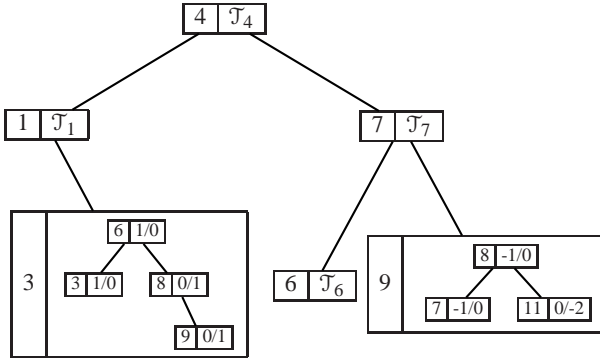

Figure 11: SST^T-tree.

6.3 SST-Tree with Hashmap-Based Nodes

An important characteristics of the SST-tree is that the space positions and associated counters are not directly used for the computation of the aggregate functions, but are inserted into a dynamic tree during the traversal of the SST-tree. The crucial part during the construction of the SST-tree is to efficiently group identical space positions and to update the associated counters. The order in which the space positions and counters are stored is irrelevant. Therefore, an interesting alternative to the SST^T-tree with a logarithmic insertion time, is to use hashmaps with a constant insertion time.

Definition 5. [SST^H-tree] Let R be an STDM relation with schema $(A_1, \ldots, A_k, T, S)$. An *SST^H-tree* for relation R is defined as an SST-tree, where each node, (t, S_t), uses a hashmap, termed *spacestamp hashmap*, to store the set, S_t, of space positions and counters. An entry of the hashmap is given as (s, cnt_b, cnt_e), where s is the entry's key. □

Example 10. Figure 12 depicts the SST^H-tree for our running example. Each node contains a hashmap to store the spatial points and counters. □

7. AGGREGATION ALGORITHM

In this section, we describe our algorithm for sequenced spatio-temporal aggregation that uses the SST-tree defined in Section 6.
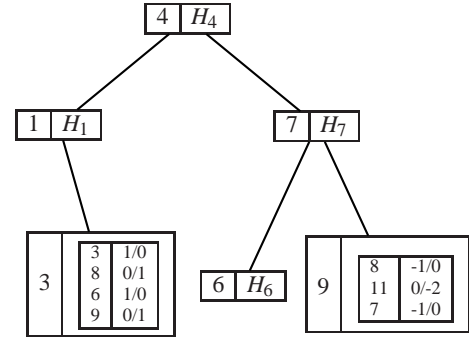

Figure 12: SST^H-tree.

7.1 Top-Level Algorithm

The overview of our aggregation procedure is presented in Algorithm 1. The procedure takes an STDM relation, R, as input and outputs the aggregation result as an STDM relation, Z. The algorithm iterates through the set of road IDs of the input relation ("for each" loop in lines 2–5). For each road, rid, it does two main steps. During the first step it scans the set of tuples of the input relation that contain rid and builds an SST-tree, T. For each tuple, its timestamp and spacestamp is extracted and inserted into the SST-tree. During the second step it computes the constant rectangles and the aggregation results.

Input: STDM relation R
Output: STDM relation Z
1 $Z \leftarrow \emptyset$;
2 **foreach** *road-ID* $rid \in \pi[RID]R$ **do**
3 $\quad T \leftarrow \text{LOADTREE}(\sigma[RID = rid]R)$;
4 $\quad Z_{rid} \leftarrow \{rid\} \times \text{COMPUTECONSTRECT}(T)$;
5 $\quad Z \leftarrow Z \cup Z_{rid}$;
6 **return** Z;

Algorithm 1: Algorithm SSTAGG.

Input: STDM relation R
Output: SST^T-tree T
1 $T \leftarrow$ empty SST^T-tree;
2 **foreach** $r \in R$ **do**
3 $\quad tn \leftarrow \text{GETNODE}(T, r.T_s)$;
4 $\quad sn \leftarrow \text{GETNODE}(tn.T, r.S_b)$;
5 $\quad sn.cnt_b ++$;
6 $\quad sn \leftarrow \text{GETNODE}(tn.T, r.S_e)$;
7 $\quad sn.cnt_e ++$;
8 $\quad tn \leftarrow \text{GETNODE}(T, r.T_f)$;
9 $\quad sn \leftarrow \text{GETNODE}(tn.T, r.S_b)$;
10 $\quad sn.cnt_b --$;
11 $\quad sn \leftarrow \text{GETNODE}(tn.T, r.S_e)$;
12 $\quad sn.cnt_e --$;
13 **return** T;

Algorithm 2: LOADTREE

7.2 Loading an SST-Tree

Algorithm 2 presents the algorithm LOADTREE for constructing an SST^T-tree, T, from an STDM input relation, R. Here we assume the use of a tree to store the spatial part, however, the algorithm can

easily be adopted for the SST^H-tree that instead uses hashmaps for the spatial part.

After initializing an empty SST^T-tree, the algorithm iterates through the tuples of the input relation, R. For each tuple, $r \in R$, we extract the start/finish time point and the begin/end positions and update the tree accordingly. First, we process the tuple's start time, $r.T_s$. The function GETNODE() retrieves from $\mathcal{T}$ the node, tn, with key value equal to $r.T_s$; if no such node exists, a new node, $(r.T_s, \mathcal{T}_\emptyset)$, with an empty spatial tree is created and inserted into $\mathcal{T}$. Then the spatial tree of node tn is updated. More specifically, the node with key value equal to the begin position, $r.S_b$, and the node with key value equal to the end position, $r.S_e$, are retrieved and the two counters are incremented (lines 4–7). If the nodes are not yet in the tree, the function GETNODE creates and inserts a new node, $(r.S_b, 0, 0)$, where the two counters are initialized to 0. Second, the tuple's finish time, $r.T_f$, is processed in a similar way. The only difference is that now the counters are decremented to represent the fact that the tuple is finishing.

Example 11. Figure 13 depicts the SST^T-tree after processing the first tuple, $r1$, of Figure 3, which is valid during the spatio-temporal rectangle $[1; 4) \times [1; 7)$. The SST^T-tree has two nodes, representing $r1$'s start and finish point, respectively. Each of these nodes stores a tree with space points and counters. In $\mathcal{T}_4$ there are two nodes, with keys 1 and 7, because the spatial interval of $r1$ begins at 1 and ends at 7. The begin and end counts of the node with key 7 are 0 and -1 respectively, because at time 4 no tuples beginning with 7 are added, but one tuple finishing with 7 is removed. □

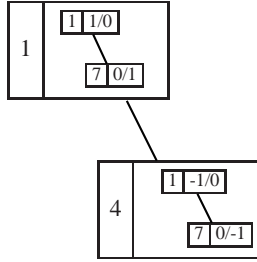**Figure 13: SST^T-tree after Inserting Tuple $r1$.**

7.3 Computing Constant Rectangles

Algorithm 3 shows the algorithm COMPUTECONSTRECT that takes as input an SST^T-tree and returns the final aggregation result. To compute the constant rectangles of the result tuples, a dynamic tree is used and continuosly updated as the algorithm traverses the SST^T-tree.

Basically, we use the dynamic tree for computing the spatial components of constant rectangles (the details are given shortly). We implement the dynamic tree as a Balanced Tree extended with a node deletion algorithm. Specifically, a dynamic tree node contains a space point, a begin count, and an end count. The information in the node tells how many tuples begin and end at the node's space point.

The main loop of the algorithm uses an in-order tree traversal to process the nodes in chronological order. For each node, tn, two steps are performed. First, the dynamic tree, $\mathcal{T}_{dyn}$, is updated with the information from the spacestamp tree, $tn.\mathcal{T}$. That is, for each node in $tn.\mathcal{T}$ the corresponding node in $\mathcal{T}_{dyn}$ is retrieved and the two counters are updated. If there is no node with space position $sn.s$ in

Input: SST^T-tree $\mathcal{T}$
Output: STDM relation Z
1 $Z \leftarrow \emptyset$;
2 $\mathcal{T}_{dyn} \leftarrow$ empty tree;
3 **foreach** *pair of consecutive nodes* $(tn, tn') \in \mathcal{T}$ *in in-order* **do**
  /* Update the dynamic tree   */
4   **foreach** $sn \in tn.\mathcal{T}$ **do**
5      $dn \leftarrow$ GETNODE$(\mathcal{T}_{dyn}, sn.s)$;
6      $dn.cnt_b = dn.cnt_b + sn.cnt_b$;
7      $dn.cnt_e = dn.cnt_e + sn.cnt_e$;
8   **if** $dn.cnt_b = 0 \wedge dn.dnt_e = 0$ **then**
9   Delete dn from $\mathcal{T}_{dyn}$;

  /* Traverse the dynamic tree   */
10    $cnt \leftarrow 0$;
11   **foreach** *pair of consecutive nodes* $(dn, dn') \in \mathcal{T}_{dyn}$ *in in-order* **do**
12      $cnt \leftarrow cnt + dn.cnt_b - dn.cnt_e$;
13      $Z \leftarrow Z \cup \{(cnt, [tn.t, tn'.t), [dn.s, dn'.s))\}$;

14 **return** Z;

Algorithm 3: COMPUTECONSTRECT

the dynamic tree, a new node, $(sn.s, 0, 0)$, with the counters initialized to 0 is inserted. Since the counters of sn might contain negative values, the counters in the dynamic tree node might become 0. If both counters of a node in $\mathcal{T}_{dyn}$ are equal to 0, the node can be deleted, since there is no new constant rectangle at this point. The second step traverses the dynamic tree in-order and computes the result tuples. The temporal part of the constant rectangle is fixed and determined by the time points of the two consecutive nodes, tn and tn'. The spatial part of the constant intervals as well as the aggregate results are determined while traversing the dynamic tree.

Example 12. Figure 14 depicts the evolution of the dynamic tree together with the produced result tuples. New nodes or updated nodes as well as new result tuples are displayed with a gray background. Figure 14(a) illustrates the situation after processing the first node of the SST^T-tree, which represents time point 1. The dynamic tree contains two nodes with space points 1 and 7, which have been inserted as new nodes. Next, the dynamic tree is traversed, producing a single result tuple. Figure 14(b) depicts the situation after processing the second node of the SST^T-tree, which represents time point 3. The associated spacestamp tree contains four nodes with space positions 3, 6, 8, and 9, respectively. Since neither of these nodes is yet in the dynamic tree, new nodes with these positions as key are inserted. Traversing the dynamic tree now produces five new result tuples. Figure 14(c) depicts the dynamic tree after the third iteration of the loop, when the node with time point 4 is processed. The begin counter for space position 6 is incremented, the counts for the position 1 and 7 is decremented, and a new node with position 11 is inserted. Since now the counts for 1 and 7 are equal to 0, these two nodes are removed from the dynamic tree. Traversing now the dynamic tree generates four new result tuples. □

The discussion above assumed that the SST-tree is implemented as the SST^T-tree. If we have the SST^H-tree instead, we do the same, only using spacestamp hashmaps instead of spacestamp trees.

8. EXPERIMENTS

In this section, we experimentally compare the SST^T-tree defined in Section 6.2, the SST^H-tree defined in Section 6.3, and the brute
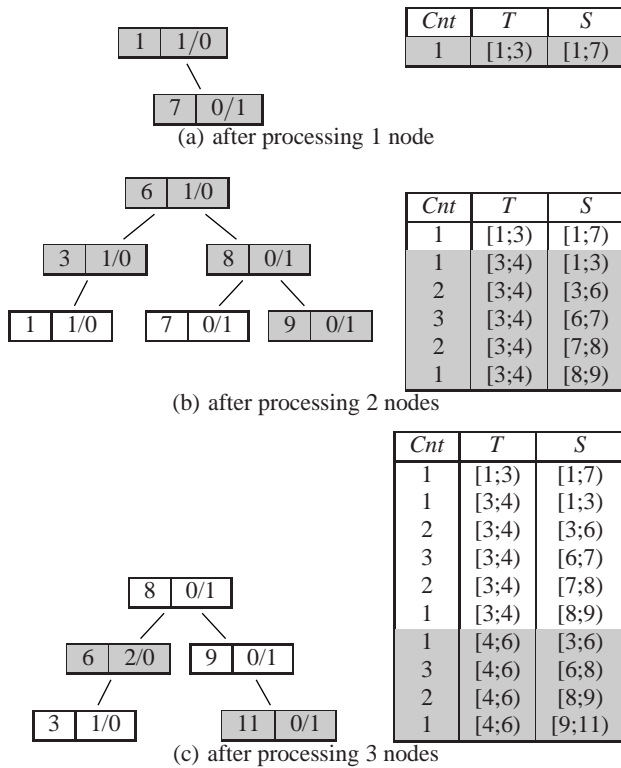
Cnt	T	S
1	[1;3)	[1;7)

(a) after processing 1 node

Cnt	T	S
1	[1;3)	[1;7)
1	[3;4)	[1;3)
2	[3;4)	[3;6)
3	[3;4)	[6;7)
2	[3;4)	[7;8)
1	[3;4)	[8;9)

(b) after processing 2 nodes

Cnt	T	S
1	[1;3)	[1;7)
1	[3;4)	[1;3)
2	[3;4)	[3;6)
3	[3;4)	[6;7)
2	[3;4)	[7;8)
1	[3;4)	[8;9)
1	[4;6)	[3;6)
3	[4;6)	[6;8)
2	[4;6)	[8;9)
1	[4;6)	[9;11)

(c) after processing 3 nodes

Figure 14: Evolution of the Dynamic Tree and Result Tuples.

Figure 15: Load time (Algorithm 2)

Figure 16: Traversal time (Algorithm 3)

force approach described in Section 5.1 (implemented as a set of Balanced Trees, one per time granule).

We compare two implementations of the SST^Htree; the first one uses Google's *sparse hashmap* and the second one uses Google's *dense hashmap*. In short, the sparse hashmap is optimized for memory (an empty bucket occupies almost no space), while the dense hashmap is optimized for speed (a key is stored in each empty bucket). Both types of hashmap use *internal probing* (i.e., only one entry per bucket). For details, see [7].

We compare main memory based versions of our data structures. The data structures are loaded with data from STDM relations stored on disk.

We ran our experiments on a machine with Intel 1.66 GHz CPU and 1 GB RAM, under Ubuntu Linux 8.04. We implement our algorithms in C++ as algebra operators (i.e., user-defined functions) of the Secondo spatio-temporal database management system [6].

For our experiments, we use 10 different STDM relations. The schema of the relations is the schema of the example input relation from Figure 3 (i.e., *(CID:int, RID:int, T:int, S:int))*. The relations are derived from the data generated with Brinkhoff's generator of moving objects [5]. Specifically, we simulate GPS logs of cars in the road network of Oldenburg, Germany. The number of roads in the network is around 7K. The number of cars per relation varies from 3K to 30K, with a step of 3K. Each relation contains the fixed number of distinct time points (100), while the number of distinct space points varies and is proportional to the number of tuples. Each tuple's temporal validity interval is 3 granules long. This relatively short interval is a *good case* for the brute force approach, because this approach considers *each* granule from the interval.

Figures 15–18 show results for Algorithm 1, issued to the 10 input relations. The marks on the X axis indicate the number of tuples in each relation.
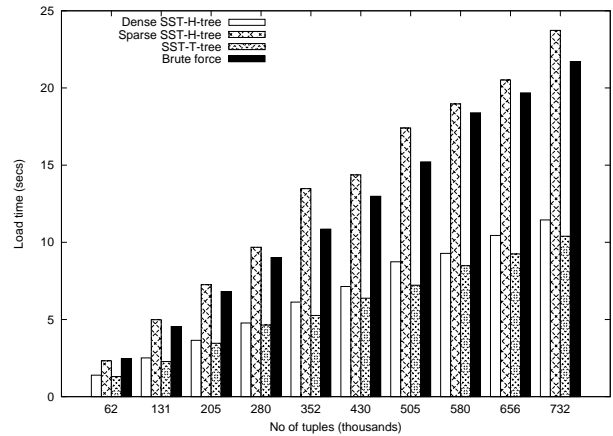
8.1 Time

8.1.1 Loading Time
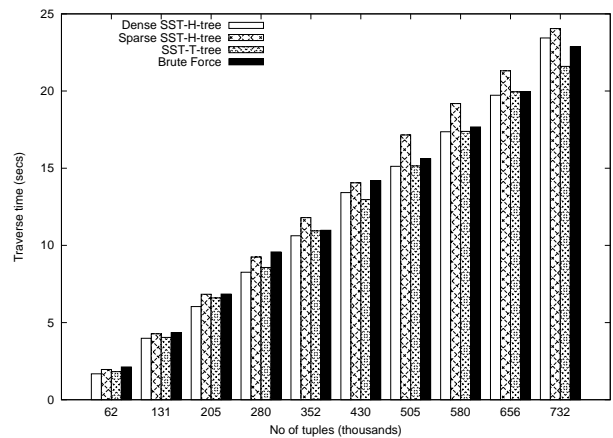
Figure 15 compares the time used for loading an SST-tree (Algorithm 2) or Balanced Trees for the brute force approach.

As expected, the brute force approach is slow, because per tuple we do $2*n$ space point lookups and up to $2*n$ space point inserts, where n is the length of the tuple's temporal validity interval. As mentioned above, in our experiments, for each tuple, $n = 3$, so $2*n = 6$. Other approaches do 4 space point lookups and up to 4 space point inserts per tuple. This number is *independent* from the temporal validity interval's length. Thus, for larger values of n, the brute force approach will become even slower, while the performance of the other approaches will not be affected.

Next, one could expect that the SST^Htree is faster than the SST^Ttree since hashmaps have constant insert/lookup time. Surprisingly, the SST^Htree performs worse than the SST^Ttree (e.g., by 11% on average for the dense hashmap-based SST^Htree). The reason for this is an overhead of Google's hashmap implementation: because of the internal probing, a Google's hashmap is resized (doubled in size) when half of its buckets are full. This is implemented by copying the data into a new, larger hashmap. The sparse hashmap has additional overhead related to memory-management.
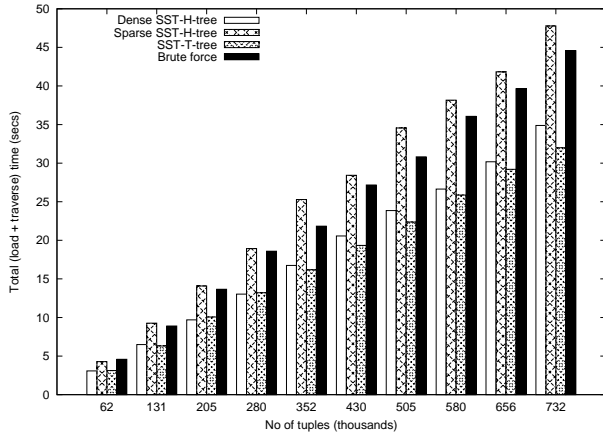
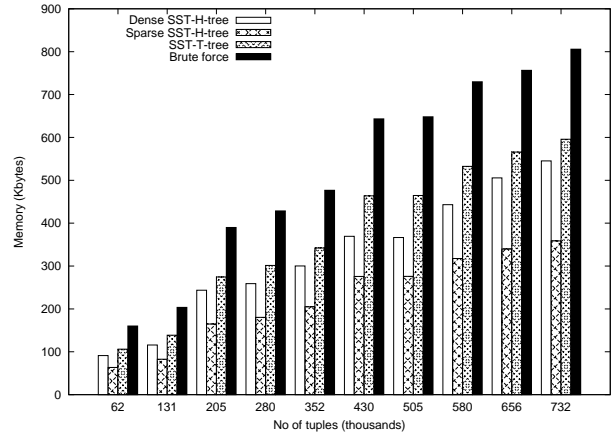**Figure 17: Total (load + traversal) time (Algorithm 1)**

Figure 18: RAM

8.1.2 Traversal Time

Figure 16 compares the time used for computing constant rectangles from a loaded SST-tree (Algorithm 3) or from a set of loaded Balance Trees for the brute force approach. Basically, this is done by traversing the data structures.

All the methods have approximately the same speed. The SST^H-tree is a little bit slower that the SST^T-tree, because the Google's hashmap iterators are relatively slow.

Surprisingly, the brute force approach demonstrates the same speed as the other methods. The reason for this is that the brute force approach does not insert space points into the dynamic tree, but simply traverses each of its Balanced Trees. However, this simplified algorithm produces a larger, uncoalesced output relation (cf. Figure 5). For example, the output relation for the leftmost input relation in Figure 16 contains 305K and 255K tuples, for the brute force and SST-tree method, respectively. Moreover, as discussed before, in our experiments, tuples have relatively short temporal validity intervals, which is a good case for the brute force approach. We expect that for longer intervals the performance of the brute force approach deteriorates. Other approaches do not depend on the length of the temporal validity interval.

8.1.3 Total Time

Figure 17 shows the total time, used for loading an SST-tree or a set of Balanced Trees first and then computing constant rectangles from it (Algorithm 1). Thus, the times in Figure 17 are obtained by adding together the corresponding times from Figure 15 and Figure 16.

8.2 Memory

Figure 18 compares the RAM used by the SST-tree or by the set of Balanced Trees for the brute force approach. Specifically, given an input relation, we measure the memory usage for each group of tuples (each road) and then compute the maximum.

The brute force approach is the most inefficient, because for each tuple from the input relation it stores the tuple's space points n times, where n is the length of this tuple's temporal validity interval. In addition, space points are stored in tree nodes, which have significant node pointer overhead. Specifically, for each space point, we need 28 bytes: 16 bytes for the data (key pointer, key value (space point), begin count, and end count) and 12 bytes for node pointers (left, right, and parent).

The SST^T-tree is more efficient. However, there is still the node

pointer overhead. For each space point, it uses 28 Bytes: 16 Bytes for the data (key pointer, key value (space point), begin count, and end count) and 12 Bytes for node pointers (left, right, and parent). Thus, the constant overhead per space point is 12 Bytes.

A dense hashmap-based SST^H-tree node is more efficient in terms of memory than the SST^T-tree. It uses 16 Bytes for a filled bucket (key pointer, key value (space point), begin count, and end count) and 4 Bytes for an empty bucket ("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 a hashmap has just resized and 75% of buckets are empty. Thus, the maximum overhead per space point reaches the overhead of the SST^T-tree (i.e., 12 Bytes), but most of the time it is smaller.

A sparse hashmap-based SST^H-tree node is the most efficient in terms of memory. It uses 16 Bytes per filled bucket and only 2 bits per empty bucket. Thus, there is almost no overhead. For every input relation, the sparse hashmap-based SST-tree uses less than 50% of memory needed by the brute force approach.

From Figure 18, we can see that both SST^T-tree and SST^H-tree scale well in terms of main memory consumption. The brute force approach is much less scalable, because it depends on the length of the tuples' validity interval.

8.3 Conclusions

We draw the following conclusions from our experimental results. The brute force approach consumes a lot of memory and is not so fast, even when the length of the tuples' temporal validity intervals is short (i.e., even in a good case). For these reasons, it should not be used. As for the other methods, there is a trade-off between the speed and the main memory usage. The SST^T-tree is relatively inefficient in terms of memory, but it is the fastest. The sparse hashmap-based SST^H-tree is very memory efficient, but it is the slowest. The dense hashmap-based SST^H-tree is somewhere in the middle: it is a little bit slower than the SST^T-tree, but at the same time it uses a little bit less memory.

Note that the insert time of hashmaps is not constant. The hashmap resize overhead is quite significant and linear to the size of a hashmap.

9. CONCLUSIONS AND FUTURE WORK

To the best of our knowledge, this paper is the first to propose an efficient method for the evaluation of *sequenced, spatio-temporal*

aggregation queries. It is based on a new data structure, called SST-tree, that efficiently stores information about a spatio-temporal input relation, which then allows to compute the aggregation results including the constant rectangles of the result tuples by an in-order traversal of the tree. We consider two variants of SST-tree implementation. Each variant has its strengths: our experiments show that the SST^T-tree is generally faster, while the SST^H-tree uses less memory. We provide an efficient algorithm that implements this two-step method: first it scans the input relation and builds the SST-tree, followed by a tree traversal to compute the result relation. The experiments show that the new method is more memory-efficient and (almost always) more time-efficient than a brute force approach. The current implementation uses the Secondo DBMS, but it is general enough to be ported to any relational DBMS.

Future work includes the following aspects. The current method works for COUNT, SUM, and AVG aggregation functions. We will extend our method for other aggregation functions (e.g., MIN and MAX). The current method performs precise aggregation (i.e., a set of spatio-temporal granules is coalesced only if we have exactly the same aggregate value for each granule from the set). We will extend our method for approximate aggregation (i.e., a set of spatio-temporal granules is coalesced if the aggregate value of each granule from the set falls into some range). Then, the order in which dimensions are processed is now fixed: the time points are always loaded into the first level of the SST-tree and the space points - into the second level. We would like to come up with a cost model that would tell which dimension should come first. In addition, we will run experiments on real-world GPS logs and provide a disk-based implementation of the method.

10. 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] M. H. Böhlen, J. Gamper, and C. S. Jensen. Multi-dimensional aggregation for temporal data. In *EDBT*, pages 257–275, 2006.

[4] M. H. Böhlen, R. T. Snodgrass, and M. D. Soo. Coalescing in Temporal Databases. In *VLDB*, pages 180–191, 1996.

[5] T. Brinkhoff. A framework for generating network-based moving objects. *GeoInformatica*, 6(2):153–180, 2002.

[6] S. Dieker and R. H. Güting. Plug and play with query algebras: SECONDO – a generic DBMS development environment. In *IDEAS*, pages 380–392, 2000.

[7] Google. Google's sparsehash project. http://code.google.com/p/google-sparsehash/. Current as of December 12, 2008.

[8] R. H. Güting, V. T. de Almeida, and Z. Ding. Modeling and querying moving objects in networks. *VLDB J.*, 15(2):165–190, 2006.

[9] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *SIGMOD Conference*, pages 47–57, 1984.

[10] C. Hage, C. S. Jensen, T. B. Pedersen, L. Speicys, and I. Timko. Integrated data management for mobile services in the real world. In *VLDB*, pages 1019–1030, 2003.

[11] C. S. Jensen, J. Kolárvr, T. B. Pedersen, and I. Timko. Nearest neighbor queries in road networks. In *GIS*, pages 1–8, 2003.

[12] 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.

[13] N. Kline and R. T. Snodgrass. Computing temporal aggregates. In *ICDE*, pages 222–231, 1995.

[14] R. Kothuri, A. Godfrind, and E. Beinat. *Pro Oracle Spatial*. Apress, 2004.

[15] J. A. C. Lema, L. Forlizzi, R. H. Güting, E. Nardelli, and M. Schneider. Algorithms for moving objects databases. *Comput. J.*, 46(6):680–712, 2003.

[16] 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.

[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] NCHRP. *A Generic Data Model for Linear Referencing Systems*. Transportation Research Board, Washington, DC, USA, 1997.

[19] D. Papadias, P. Kalnis, J. Zhang, and Y. Tao. Efficient OLAP operations in spatial data warehouses. In *SSTD*, pages 443–459, 2001.

[20] D. Papadias, Y. Tao, P. Kalnis, and J. Zhang. Indexing spatio-temporal data warehouses. In *ICDE*, pages 166–175, 2002.

[21] R. T. Snodgrass, editor. *The TSQL2 Temporal Query Language*. Kluwer, 1995.

[22] L. Speicys and C. S. Jensen. Enabling location-based services - multi-graph representation of transportation networks. *GeoInformatica*, 12(2):219–253, 2008.

[23] 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.

[24] Y. Tao, G. Kollios, J. Considine, F. Li, and D. Papadias. Spatio-temporal aggregation using sketches. In *ICDE*, pages 214–226, 2004.

[25] Y. Tao, D. Papadias, and J. Zhang. Aggregate processing of planar points. In *EDBT*, pages 682–700, 2002.

[26] O. Wolfson, H. Cao, H. Lin, G. Trajcevski, F. Zhang, and N. Rishe. Management of dynamic location information in domino. In *EDBT*, pages 769–771, 2002.

[27] O. Wolfson, S. Chamberlain, S. Dao, L. Jiang, and G. Mendez. Cost and imprecision in modeling the position of moving objects. In *ICDE*, pages 588–596, 1998.

[28] J. Yang and J. Widom. Incremental computation and maintenance of temporal aggregates. In *ICDE*, pages 51–60, 2001.

[29] 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.