# RDBMS Support for Efficient Indexing of Historical Spatio-Temporal Point Data

Daniel Mallett, Mario A. Nascimento, Viorica Botea and Joerg Sander

November 1, 2005

TR-84

A TIMECENTER Technical Report

| | |
|---|---|
| Title | RDBMS Support for Efficient Indexing of Historical Spatio-Temporal Point Data |
| | Copyright © 2005 Daniel Mallett, Mario A. Nascimento, Viorica Botea and Joerg Sander. All rights reserved. |
| Author(s) | Daniel Mallett, Mario A. Nascimento, Viorica Botea and Joerg Sander |
| Publication History | November 2005. A TIMECENTER Technical Report. |

**Abstract**

Despite pressing need, current RDBMS support for spatiotemporal data is limited and inadequate, and most existing spatiotemporal access methods cannot be readily integrated into an RDBMS. This paper proposes SPIT, an adaptive technique for spatiotemporal storage, indexing and query support that can be fully integrated within any off-the-shelf RDBMS. We initially propose a cost model that assumes a uniform data distribution for determining an optimal partitioning of the data space in terms of query processing time. We then use this model as a basis for a heuristic method for partitioning the data space without making any assumption about the data distribution. Using Oracle as our implementation platform with both real and synthetic datasets, we show that SPIT is robust and significantly outperforms other RDBMS-based options for managing historical spatiotemporal data.

## 1 Introduction

The need for spatiotemporal access methods (STAMs) integrated within a relational database management system (RDBMS) has become increasingly apparent. A prolific number of GPS, wireless computing, and mobile phone devices are capable of accurately reporting their position, and applications that can take advantage of this information, e.g., traffic control, data mining, fleet monitoring, and location-aware services, are in high demand. Managing large datasets of such data demands the convenience, reliability, and data storage capabilities that a traditional RDBMS affords. However, little work exists on how to provide spatiotemporal data support, a STAM in particular, inside a RDBMS, [1] being an exception. Our work fills this crucial need by proposing a spatiotemporal access method which can be fully integrated within any RDBMS. We identify two chief alternatives for providing RDBMS support for spatiotemporal data: loosely coupling a STAM to the RDBMS, and tightly coupling a STAM inside the RDBMS via a relational mapping. Our approach falls under the latter, i.e., we design a STAM that leverages existing RDBMS functionality. The chief advantage is that our method can be readily integrated into any RDBMS.

There are two main types of spatiotemporal databases [2], those that manage historical information and those that manage current information for current/predictive query purposes. This paper focuses on the first category, i.e., we assume that the database stores the complete history of moving objects through time and must answer queries about any time in the history of objects. We assume that records about object's movements are tracked and sent (possibly via regular updates) to the RDBMS. Each record has the attributes $\langle oid, x, y, t_s, t_e\rangle$, where $oid$ identifies an object, $\langle x, y\rangle$ are spatial coordinates, and $\langle t_s, t_e\rangle$ indicate the interval during which an object remained at position $\langle x, y\rangle$. A typical domain where such a model fits is mobile device tracking, e.g., of GPS, PDA, or wireless phone devices. Unlike the trajectory model [3], our data model does not assume anything about the movement of objects between records. The model reflects a real-world application[1] constraint where assuming an object follows a linear trajectory between data points may lead to incorrect assumptions. For example, in security/monitoring applications, a person could be mistakingly assumed to have entered a restricted area because his/her movement was interpolated. Our model can be viewed as a step-wise interpolation instead of a linear interpolation, i.e., objects are assumed to remain at the given $\langle x, y\rangle$ position for the given $\langle t_s, t_e\rangle$ time interval.

In this work a a spatiotemporal range query $Q$ takes the form $Q = \langle \mathcal{R}, \mathcal{T}\rangle$ where $\mathcal{R}$ is a spatial region and $\mathcal{T}$ is a time range. $Q$ returns the unique $oid$'s of records where $\langle x, y\rangle$ is inside $\mathcal{R}$ and $\langle t_s, t_e\rangle$ intersects with $\mathcal{T}$. An example of such a query would be "find all objects that were in the West Edmonton Mall at some point between noon and 1 p.m. yesterday".

In this paper we propose an efficient spatiotemporal indexing technique fully integrated within a RDBMS via a relational mapping. Our approach is based on a partitioning of the data space, which can be done at logical or physical level at the underlying RDBMS. The general idea of our approach is similar to SETI [4] in that we have primary partitions the of two-dimensional space, and independent temporal index structures in each of the spatial partitions.

The new contributions of this paper are the following: (1) We develop a cost model to analytically determine the number of primary partitions to use. The model suggests the number of primary partitions so that the expected number disk accesses is minimized, assuming a uniform data distribution and an average expected query size. (2) Based on this cost model, we also propose a heuristic method for partitioning the data space for arbitrary data distributions, which yields very good performance in practice. (3) We design a relational mapping of our

[1] Details of which cannot be disclosed due to confidentiality reasons.

proposed STAM, which can be used to easily deploy this STAM using any RDBMS. (4) We show in a comprehensive experimental comparison that our proposed technique dramatically outperforms other RDBMS-supported spatiotemporal indexing alternatives.

This remainder of this paper is structured as follows, Section 2 reviews related work and also provides background on what options for RDBMS support of spatiotemporal data exist. Section 3 details our proposed approach, and the associated cost model. In Section 4 we describe how our approach can be implemented using a particular RDBMS. In Section 5 we confirm the reliability of the model and compare our approach to several other methods for indexing spatiotemporal data inside a RDBMS. Section 6 concludes the paper.

## 2 Related Work

A thorough overview of work on STAMs for historical and current/predictive spatiotemporal support can be found in [2]. Predictive STAMs support queries that predict a moving object's location at a given time based on the current velocity of the object. Historical STAMs support queries that can be classified as coordinate-based (the case we are interested in) or trajectory-based [3].

The current state-of-the-art for predictive STAMs is the B$^x$-tree [5]. Built on top of a B$^+$-tree and using a space filling curve underneath it, it allows, like in our case, the index to be used within an existing DBMS. Another recent access structure of interest is the TPR$^*$-Tree [6], which is an a version of the TPR-tree TPR-tree [7] with improved construction algorithms based on a performance model.

Many historical STAMs have been proposed [3, 8, 4, 9, 10] the majority of which are based on the R-tree [11], [4] being a notable exception. The 3-D R-Tree [10] treats time as a third dimension and indexes spatiotemporal data using a 3-dimensional R-tree. The Historical R-tree (HR-tree) [9], an overlapping and multi-version structure, adapts the R-tree for historical spatiotemporal data. The MV3R-tree [12] improved upon these providing more efficient support for interval queries. The Trajectory Bundle Tree (TB R-tree) [3] proposes a trajectory-oriented access method that can (under certain conditions) answer trajectory-oriented queries faster than the R-tree. The 2-3TR-tree [8] suggests the use of two R-tree indexes, a two-dimensional point index representing current data, and a three-dimensional historical index.

The work presented in [4] is the one closest to ours. The authors propose a grid-based spatiotemporal indexing technique which they call SETI. SETI partitions the spatial dimension into static, non-overlapping partitions, and within each partition uses a "sparse" temporal index –which the paper describes as a 1-dimensional R-tree over the temporal interval of all the object records stored in a single data page. An in-memory "front-line" structure keeps track of the last position of each moving object. However, unlike our proposal, no cost model or heuristic is presented to guide the partitioning of the data space.

## 3 SPIT: Space Partitioning with Indexes on Time

SPIT partitions the data according to its spatial location and then creates temporal indexes over each partition. The data is partitioned into a fixed number of cells, each cell corresponding to a different partition in the RDBMS. The key advantage of spatial partitioning is that of partition elimination at query time. Cells that do not intersect the spatial component of the query window can be eliminated from consideration. For spatiotemporal data this works extremely effectively because we can further apply a temporal filter within all intersecting cells. The spatial discrimination is achieved at next to no cost and the local temporal index benefits from having to manage only a (small) subset of the data. As in [4], query processing proceeds according to four stages: (1) coarse *spatial filtering* based on the grid location of tuples, (2) *temporal filtering* using the per grid temporal indexes, (3) fine *spatial refinement* based the on actual spatial location of tuples, and (4) *duplicate elimination*.

As shown in Figure 1, for the case of a regular grid, the cells could be numbered using a horizontal sweep space-filling curve, giving each cell a unique identifier $pid$ (shown in the upper right corner of each cell in the figure). The length $l$ refers to the length of a grid cell in each spatial dimension. A local temporal index on $\langle t_s, t_e \rangle$ is created over the domain of tuples within each partition. For that we use a combined B-tree index on $\langle t_s, t_e \rangle$. We considered the use of a 1-dimensional R-tree to index the temporal dimension. However, we abandoned the idea because preliminary experiments showed that a large degree of overlap among the temporal intervals of objects occurs. In such a situation the performance (and index creation times) of the 1-D R-tree approach is prohibitively expensive. It is important to note that not only can the combined B-tree on $\langle t_s, t_e \rangle$ be readily supported in any

RDBMS, but the B-tree index also has a performance advantage of being able to perform an index range scan. For similar performance reasons we do not consider the use of techniques for RDBMS-support of temporal data, i.e., the temporal RI-tree [13]. Finally, assuming tuples are sorted by time, at query time a sequential scan is performed on disk over the range where tuples intersect the temporal query interval.

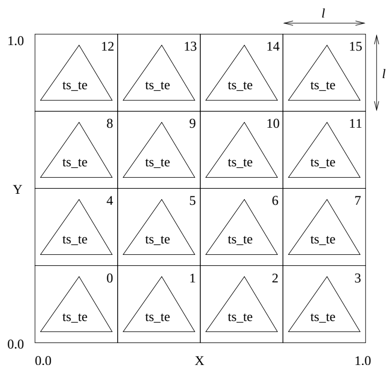

Figure 1: SPIT's approach for a 4 $\times$ 4 regular grid.

**Algorithm 1** $st\_query()$ function.

**INPUT:** $\langle \mathcal{R}, \mathcal{T} \rangle$
**OUTPUT:** list of $oid$'s

```
 1: $pid\_list := p\_intersect(\mathcal{R})$
 2: for all $pid$ in $pid\_list$ do
 3:     $oid\_list := oid\_list \ \cup$
 4:               SELECT $oid$
 5:               FROM $pid$
 6:               WHERE $t_s$ BETWEEN $\mathcal{T}.t_{min} -$ MAX_TI AND $\mathcal{T}.t_{max}$
 7:               AND $t_e$ BETWEEN $\mathcal{T}.t_{min}$ AND $\mathcal{T}.t_{max} +$ MAX_TI
 8:               AND $x$ between $\mathcal{R}.x_{min}$ and $\mathcal{R}.x_{max}$
 9:               AND $y$ between $\mathcal{R}.y_{min}$ and $\mathcal{R}.y_{max}$
10: end for
11: sort $oid\_list$ and remove duplicates
12: return $oid\_list$
```

Algorithm 1 provides the pseudo-code for the function $st\_query()$ which processes queries using the SPIT model. The algorithm assumes that the function ***p_intersect()*** exist. Its task is to simply return the identifiers ***pid*** of the grid cells that intersect the query's spatial component. Note that lines 4–9 assume the existence of a SQL interface in order to retrieve matching tuples from partitions in the RDBMS. Thus, the filtering occurs in a pipelined fashion – at each stage of query processing only those tuples satisfying the previous stage are further examined.

As was done in [14], Algorithm 1 uses the fact that the the largest temporal interval is known (and denoted as *MAX_TI*). This is a reasonable and practical assumption, e.g., in fleet monitoring, it can be safe to assume that vehicles do not remain stationary for more than 2 or 3 days, which serves to further restrict the temporal range that needs to be inspected at query time, hence improving query processing time. In cases where a minority of objects may occasionally exceed the *MAX_TI*, the offending records can be split into two or more records that adhere to the assumption.

## 3.1 SPIT's Cost Model and Partitioning

We propose the following cost model to choose an optimal cell size for use within SPIT assuming a fixed regular grid. We assume the average query size, on both the temporal and spatial dimensions, are known –the robustness of

| Symbol | Meaning |
|---|---|
| $N$ | number of tuples in the database |
| $DA$ | number of disk I/Os to answer a query |
| $GA$ | average number of grid cell accessed |
| $DA_g$ | number of data (disk) I/Os per grid cell accessed |
| $IA_g$ | number of index (disk) I/Os per grid cell accessed |
| $f$ | fanout of a B-tree index |
| $BS$ | block size (the number of tuples that fit in one block on disk) |
| $q_s$ | average size (percentage-wise) of the query in each spatial dimension with respect to the modeled space |
| $q_t$ | average size (percentage-wise) of the temporal aspect of the query with respect to the number of observed timestamps |
| $l$ | length of a grid cell in each dimension |
| $l^*$ | optimal length of a grid cell in each dimension |
| $N_g$ | total number of cells in the grid = $(1/l)^2$ |
| $N_g^*$ | optimal total number of cells in the grid = $(1/l^*)^2$ |
| $N_p^*$ | optimal number of partitions (grid cells) in one dimension |

Table 1: Notation used.

SPIT with respect to such an assumption is discussed in the experimental Section 5.3. We also assume the spatial domain to be the unit square and that the temporal domain is formed by the total set of recorded timestamps. Note that this means the temporal domain is therefore bounded and a finite number of observations exist. Table 1 lists the notation we will use throughout the reminder of the paper.

The total number of disk accesses to answer a query can be calculated by the average number of grid cells (partitions) that need to be accessed and the number of I/O's performed inside each accessed grid cell – which is the combination of reads to the data and reads to the temporal index structure inside each grid cell, i.e.: $DA = GA \times (DA_g + IA_g)$.

As per [15], the average number of cells that will be scanned is the total number of cells multiplied by the average space the spatial component of a query covers extended by $l$: $GA = N_g(l + q_s)^2$.

Assuming a uniform data distribution, there are on average $N/N_g$ tuples per grid cell which take up $\frac{N/N_g}{BS}$ blocks on disk to store. Because the index on $\langle t_s, t_e \rangle$ will point to the range of tuples in the query answer set, we only need to scan those blocks that are within the temporal range of our query $q_t$, i.e.: $DA_g = \frac{N/N_g}{BS} \times q_t$.

We assume a B-tree on the combined key of $\langle t_s, t_e \rangle$ and (as in the worst case) that none of the index pages are located in buffer, the number of index accesses can be described in terms of the fanout $f$ and $N/N_g$ using: $IA_g = log_f N/N_g$. If we simplify the index access cost to $IA_g = 3$, which is typical for indexes with $f \approx 100$ and $N$ in the millions of tuples [16], we obtain $DA = (l + q_s)^2(\frac{N \times q_t}{BS} + \frac{3}{l^2})$.

One immediate observation is that the index performance it more sensitive to the size of the spatial component than to the temporal component. This is due to the fact that increasing the query's area requires traversing more partitions and the indexes within them. On the other hand, increasing the query's temporal range requires only a larger scan on the indexes, which can be done efficiently.

After some algebraic manipulation it is easy to see that the grid size $l^*$ that will minimize disk accesses is given by $l^* = \sqrt[3]{\frac{6q_s \times BS}{2N \times q_t}}$ which can be shown to be a unique solution using the second derivative. Finally, the optimal number of grid cells ($N_g^*$) can be represented in terms of $l^*$ using

$$N_g^* = \frac{1}{(l^*)^2} = (\frac{N \times q_t}{3q_s \times BS})^{2/3}. \quad (1)$$

Thus, we can set $N_p^* = \lceil \sqrt{N_g^*} \rceil$ in order to obtain a *regular* partitioning of the data space that minimizes the number of disk access per query given an average query size.

It should be noted that partitioning the data space using the criteria just presented is optimal given the assumption of a uniform data distribution. While in real life scenarios data is seldom truly uniformly distributed if one considers the whole modeled data space, it is often the case that for some regions of the data space such an assumption can be made. For instance in a map, it is much more reasonable to assume that objects are uniformly distributed inside the boundaries of a city than that they are collectively uniformly distributed over the whole map. In what follows we use this reasoning and the cost model above in order to provide a partitioning heuristic for an arbitrary data distribution.

The idea is to recursively divide the space into four subspaces, as in a Quad-tree [17], until all obtained subspaces satisfy a uniform distribution criteria. The obtained cells are then partitioned using the criteria yielded by the cost model. The uniformity of the data distribution can be checked using Pearson's Chi-Square test [18]. The test partitions the data into $K$ equally sized cells (categories) and computes the sum ($S^2$) of squared differences between the actual number of objects inside each cell and the expected number of objects under the uniformity assumption (i.e., $N/K$, where $N$ is the total number of objects). If the value of $S^2$ is smaller than $\chi^2_{K-1}(\alpha)$ then the uniformity assumption is accepted, otherwise it is rejected. Algorithm 2 states this procedure using pseudo-code.

**Algorithm 2** *Partition()* recursive algorithm.

```
INPUT: An MBR containing data points
OUTPUT: A set of MBRs (each corresponding to a grid cell) and respective partitionings
1: Assume a uniform distribution of the data points and perform Pearson's test on MBR (using the grid granularity suggested by the cost model)
2: if Pearson's test is successful, i.e., the data distribution within the MBR is considered uniform, then
3:    Partition the MBR, in a regular manner, as suggested by the cost model
4:    Store the resulting MBRs coordinates into table Partitions
5: else
6:    Partition the MBR, splitting each dimension in half, obtaining MBR$_i$, $i = 1, 2, 3, 4$
7:    for $i$=1 to 4 do
8:       Partition(MBR$_i$)
9:    end for
10: end if
```

It should be clear that if the data is truly uniformly distributed, the heuristic presented above yields an optimal grid partitioning (as per the cost model assumptions). Indeed, in such a case the uniformity test would be immediately successful and the algorithm would not recurse at all, yielding exactly what the cost model would have suggested in the first place.

It may appear that in the worst case the partitioning above can result in a very large of partitions with very few objects in each of them. This obviously is not a good idea since there is an overhead cost to access a partition, and there is a point where access less data in more partitions is more expensive than accessing more data within less partitions. Fortunately, the heuristic partitioning above identifies such situation and stops the partitioning accordingly. We discuss this in the following.

Recall that, during the partitioning, $q_s^2$ is the query size with respect to the current MBR, and $N$ is the number of objects inside the current MBR. Initially the current MBR is the whole unit square, but as the partitioning goes, i.e., the MBRs are subdivided and the current MBRs become smaller, as a consequence, $q_s$ becomes relatively larger. On the other hand, the number $N$ of objects per MBR becomes likely smaller as the MBRs are subdivided.

Let us consider the case the when the query size becomes equal to the current MBR, i.e., $q_s = 1$. From Equation 1 one can see that if $q_s = 1$ and $BS$ and $q_t$ are constants, then $N < \frac{3BS}{q_t}$ yields $N_g^* = 1$, i.e., no further partioning is needed. This agrees with the intuition that as the partitioning progresses, it eventually leads to the situation where accessing less data in more partitions becomes more likely and is more expensive than accessing more data within a single partition, therefore triggering the partitioning process to stop automatically.

Although only optimal for the case of uniformly distributed data, the resulting overall performance by SPIT is typically very good. Indeed, as we shall see in the experimental section it is never worse than the best ad-hoc partitioning, i.e., the best partitioning one could obtain by trial-and-error. More importantly, however, SPIT is able to find very good partitions of the data space autonomously, not relying on any information but the dataset itself and an expected query size. Naturally, the better the user can estimate the query size (which should happen with time) the better the partitioning and therefore the query performance.

```
create table ST_SPIT (
   oid       integer,
   x         number,
   y         number,
   t_s       number,
   t_e       number,
   pid      integer
) partition by range (pid) (
   partition p01 values less than (1),
   partition p02 values less than (2),
   partition p03 values less than (3),
   partition p04 values less than (4)
)
```

(a)

```
1: SELECT UNIQUE oid
2: FROM   ST_SPIT
3: WHERE  pid IN (0,1,4,5)
4: AND    t_s BETWEEN (0.5-MAX_TI)
          AND 0.6
5: AND    t_e BETWEEN 0.5 AND (0.6
          +MAX_TI)
6: AND    x BETWEEN 0.1 AND 0.3
7: AND    y BETWEEN 0.2 AND 0.4
```

(b)

Figure 2: Creating and querying the `ST_SPIT` table for a 2×2 grid.

## 4 SPIT's Implementation

The SPIT grid is implemented using Oracle's built-in table partitioning support –each grid cell determined by our heuristic algorithm *Partition()* corresponds to a single Oracle table partition. An unique partition id (`pid`) along with its MBR is stored in a table called `Partitions`. The `ST_SPIT` table (whose DDL for an example 2×2 grid is provided in Figure 2(a)) stores records along with the additional $pid$ attribute. When inserting an object into table `ST_SPIT` its coordinates are checked against the `Partitions` table to determine in which partition it should be inserted. Oracle range partitioning is used to automatically map the spatial grid to unique table partitions on disk. Note that Oracle's partitioning facility is not a requirement for SPIT to work. An RDBMS which does not provide such facility can be used by simply creating a physical table for each grid cell.

Given the sample query "find the objects that were within the area enclosed by the MBR determined by vertices (0.1,0.3) and (0.2,0.4) during the time interval [0.5,0.6]", Figure 2(b) provides the SQL query that would be issued against the `ST_SPIT` table created in Figure 2(a).

Line 3 of the sample query corresponds to the *spatial filtering* stage of SPIT's query processing. The clause forces Oracle to scan only table partitions corresponding to cells (0,1,4,5) –The list is computed by performing a lookup on table `Partitions`. Only 4 out of 16 partitions need be scanned, which, even for such a trivial example, is a significant reduction in I/O cost.

Lines 4–5 correspond to the *temporal filtering* stage of SPIT's query processing. Within each partition, the combined B-tree index on $\langle t_s, t_e \rangle$ will be taken advantage of as Oracle will perform a local index range scan of the data. The clustering of data according to $\langle t_s, t_e \rangle$ speeds up this phase of query processing.

Lines 6–7 correspond to the *spatial refinement* stage of SPIT's query processing. All tuples whose spatial coordinates are not inside of the spatial query range are removed from the query result. Finally, line 1 performs the *duplicate elimination* stage of SPIT's query processing.

We defined a PL/SQL function that generates dynamic SQL queries of the form provided in Figure 2(b) given a query spatial and temporal range. We choose to implement the algorithms using PL/SQL because of the ease of integration between PL/SQL and SQL queries in ORACLE, however, any language capable of interacting with the RDBMS, e.g., using embedded SQL, could be used.

## 5 Experimental Results

In order to test our proposal we used both synthetic and real datasets. One of the synthetic data set, denoted as UNIFORM, has the objects uniformly distributed in the space and moving freely throughout the whole space. This satisfies the assumptions for SPIT's cost model (v. Section 3.1). The second synthetic dataset was generated using the GSTD tool[2] [19] and shows a scenario where the objects have an initial gaussian distribution in the center of the data space and then migrate towards the north-east corner of the same. A sample instance of this dataset, denoted as GSTD, is illustrated in Figure 3(a), where all observation positions for a sample of 100 objects are shown. This

[2] `http://db.cs.ualberta.ca:8080/gstd/`

| Parameter | Values (default in **bold**) |
|---|---|
| Average Query Spatial Range ($q_s$) [% of data space] | 0.25%, **1%** and 4% |
| Average Query Temporal Length ($q_t$) [timestamps] | 5, **10** and 20 |
| Dataset size ($N$) [millions of observations] | 1, **2.5** and 5 |

Table 2: Parameter and values investigated.

dataset is more realistic that the previous one, e.g., it could depict a scenario where animals are migrating from one area to another in a park. It also will serve to show how well the heuristic partitioning approach we proposed adapts for a truly non-uniform data distribution. The final dataset, denoted as INFATI, contains real GPS positions of 20 cars roaming across the municipality of Aalborg, Denmark [20]. Each car's positions have been sampled every second, except when they were parked, for about 6 continuous weeks over a period of 3 months. The dataset contains approximately 1.9 million observations and is illustrated in Figure 3(b) where all observations are plotted –one can clearly see the notion of actual roads in this case.

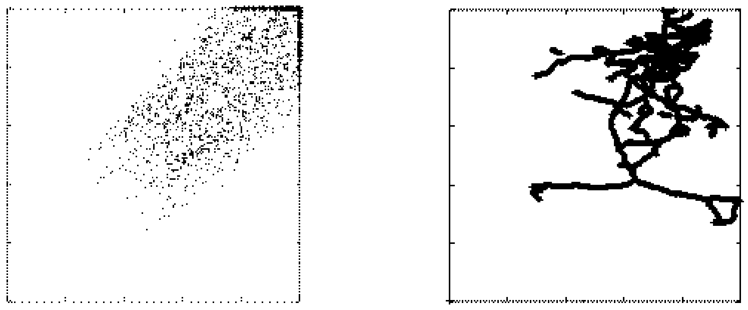
(a) GSTD dataset (b) INFATI dataset

Figure 3: Data distribution for the GSTD and INFATI datasets.

For each of the synthetic datasets we have three different cardinalities, namely 1, 2.5 and 5 million observation data points. Given how the data is generated it means that each dataset has about 10, 25 and 50 thousand objects of interest, respectively. We assume a unit two-dimensional dataspace and for query sizes we have used 0.25%, 1% and 4% of the dataspace. Note that a query of 4% of the unit space has selectivity of about 20% in each dimension, i.e., it is not a small query. We experimented using the temporal query component equal to 5%, 10% and 20% of all observed timestamps. Table 2 summarizes the parameters used for the experiments. Unless otherwise mentioned whenever one parameter is being investigated, e.g., the robustness with respect to dataset size, all other parameters are kept constant at their default values.

To investigate the average cost per query we issued 100 random queries following the same distribution of the dataset, and measured the average number of disk I/Os (physical accesses) perquery using the system's own internal tools. All tests were carried out on a desktop using Oracle 10g Enterprise for Windows Edition. Before executing each query the DBMS's buffers were forced clear to avoid any influence on query performance.

We compare SPIT's performance to two other approaches that could be implemented on top of Oracle. (Recall that our main goal is to have an indexing scheme that can be deployed upon off-the-shelf RDBMS.) The first method uses an R-tree for the spatial component along with a B-tree for the temporal component. We adapt the LRS spatio-temporal indexing approach suggested by Oracle [1] to our data model by creating a 2-dimensional R-tree over point objects consisting of the $\langle x, y \rangle$ of records and a B-tree index on $t_s$ and on $t_e$. In what follows we refer to this scheme as "R-tree+B-tree" The second approach is a simple *Linear Scan* which should provide the lower bound for expected performance.

We also used a scheme based on Z-values for of each tuple based on its $\langle x, y \rangle$, and another B-tree for the temporal component. For each dataset, we calculated Z-values using the same number of cells in each dimension ($N_p^*$) that SPIT employs. Although feasible and actually simple to implement, our preliminary experiments have shown that this technique does not yield competitive performance and therefore we did not consider it further. Details on how to implement both the R-tree+B-tree and the Z-order based schemes can be found in [21].

## 5.1 Evaluating SPIT's Partitioning Heuristic

We initially confirm the reliability of our cost model by comparing the analytical optimal number of grid cells to the number of disk accesses reported by Oracle when using the Uniform data distribution (for which the model gives an optimal partitioning). In this case the only alternative for comparing performance is an ad-hoc partitioning where the user chooses a grid size manually.

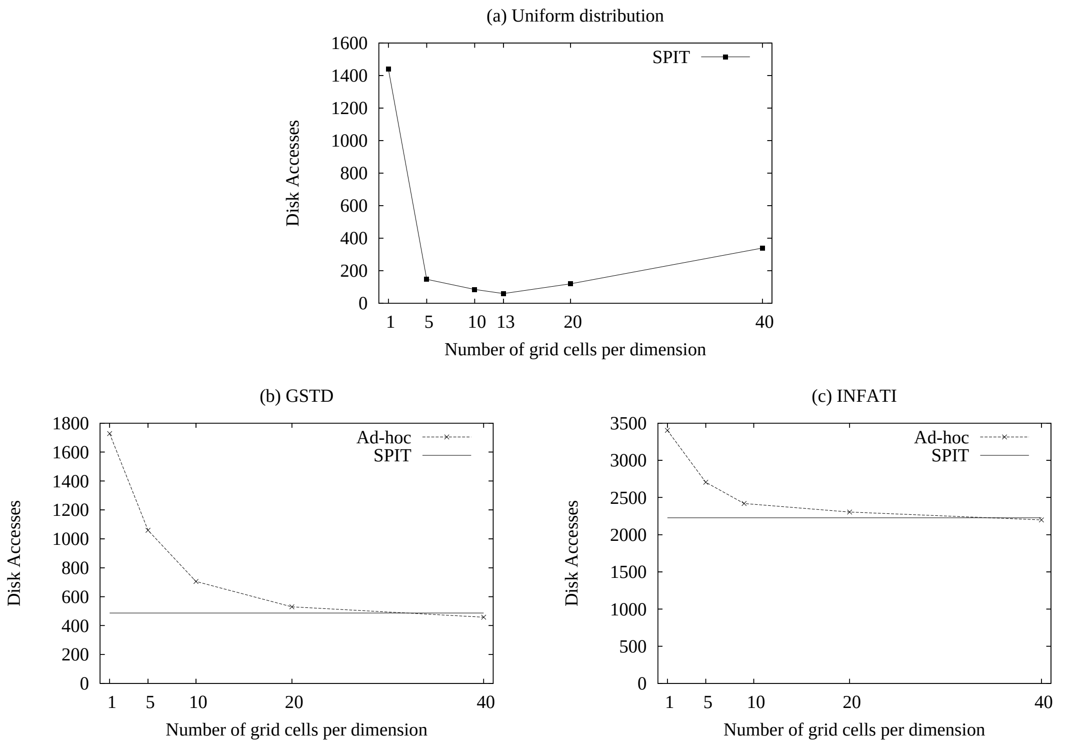

Figure 4: Comparing I/O performance yielded by SPIT's partitioning against the use of ad-hoc regular grids.

When using all experimental default values and a block size of 8192 bytes our cost model determines a $13 \times 13$ grid, which indeed is the best option when compared to several other choices for a regular partitioning of the data space as shown in Figure 4(a).

It is interesting to note that as the number of partitions increases beyond the optimum, there is an increasing overhead due to the cost of accessing more partitions. Even though not shown here, this is even more clear for larger query sizes, which cover a larger number of partitions. When the number of partitions is smaller than the optimum then the overhead is due to reading more data per partition than it would be necessary in the optimal case.

As discussed earlier, for non-uniform distributions SPIT uses the cost model to obtain a non-regular partition of the dataspace. Again we compare to the ad-hoc alternative of having the user trying several different regular grids. As can be seen in Figures 4(b) and (c), for both non-uniform distributions the grid partition determined by SPIT provides performance at least as good to the best ad-hoc partitioning. (Since the resulting grid is non-uniform it does not make sense to plot performance as a function of the number of grid cells as in the case of Uniform data distribution, hence the flat line for the SPIT performance.)

Again, the additional cost of underpartitioning is clear, but unlike in the case for Uniform data, overpartitioning seems to be not as prejudicial. In fact, it is clear that the partitioning obtained via SPIT is not optimal (though it was not meant to be in the first place), as the finer partitioning yields performance slightly superior to SPIT's. Nevertheless, giving the trend in the figures it reasonable to expect that over partitioning would eventually lead to too much overhead and deteriorate performance as well.

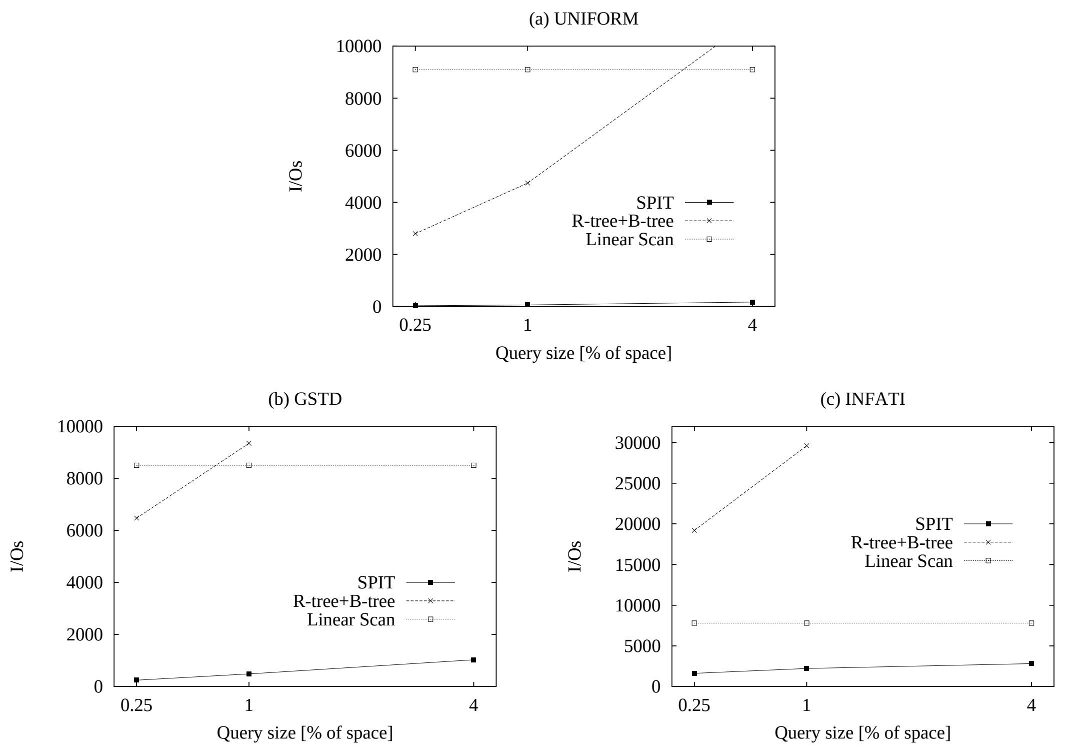

Figure 5: Comparing I/O performance as a function of the size of the spatial component of the query.

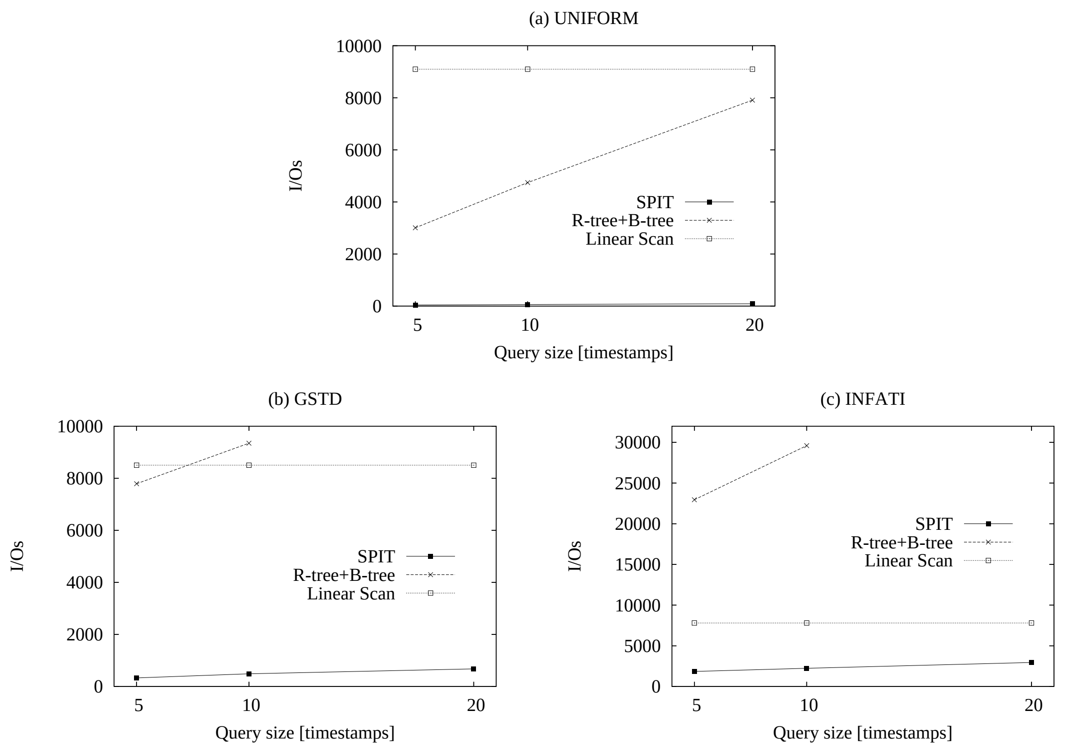

Figure 6: Comparing I/O performance as a function of the length of the temporal component of the query.

## 5.2 Query Performance Evaluation

Next we compare the performance of SPIT against the R-tree+B-tree approach and a linear scan of the data, i.e., no index support. All approaches make use of the assumption that *MAX_TI* is known at query time.

Figure 5 shows query performance as a function of the size of the spatial component of the query, while Figure 6 shows the performance when varying the length of the temporal compoment. As expected, in both cases the performance of the linear scan is constant, as it depends only on the cardinality of the dataset. In all figures it is easy to see that the performance of the R-tree+B-tree approaches degrades rather quickly, unlike for the other approaches. In the case of the UNIFORM dataset is the only one where the R-tree+B-tree remains competitive with the linear scan for up to medium sized queries. For GSTD data the R-tree+B-tree is not competitve at all. This happens because for the former the density of the data in the occupied portion of the space is higher, causing the undelying R-tree to have more overlaps and, consequently require more tree traversals. The case for the INFATI dataset is even more extreme. In this case the linear scan performs relatively much better and is only about 2.65 times slower than SPIT for larger queries. This in contrast for the UNIFORM and GSTD dataset where, for the same size of queries, the linear scan was up to 104 and 13, respectively, times slower. This loss of relative advantage for SPIT seems to be related to the scan on the B-tree leaves. Recall that we make use of the *MAX_TI* information. If this value is large a large portion of the leaves in the B-tree would need to be scanned, rendering the B-tree itself of not much use, i.e., degenerating to a linear scan of the dataset inside the investigated partitions. In our future work we plan to investigate how to split larger time intervals into smaller one at the expense of increasing the number of indexed objects in order to further improve SPIT's relative performance.

SPIT consistently provides the best performance, being up to 100 faster than the other approaches. More importantly however, it is very robust with the increase of the query size for all distributions. This suggests that the grid partitioning heuristic is able to cope well with variations in this parameter.

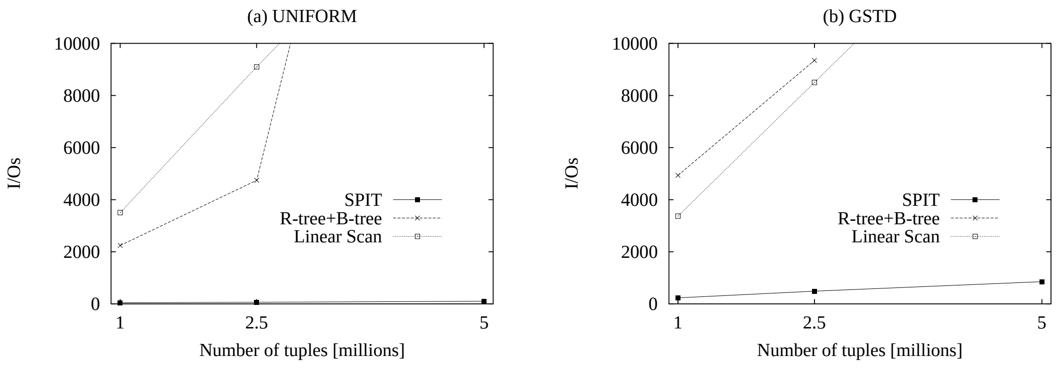

Figure 7: Comparing I/O performance as a function of the dataset cardinality.

SPIT is also very robust with respect to the increase in the dataset size as can be seen in Figure 7. (Note that the INFATI data set was not used here as the dataset cardinality is fixed and an intrinsic part of the dataset features.) The linear scan, as one would expect, does not scale well with the size of the dataset, and again the R-tree+-B-tree approach is also a poor choice. SPIT's performance is up at least two orders of magnitude faster than the other approaches, and more importantly, quite stable with respect to the dataset size. This is due to very effective filtering of heavily heavily populated partitions that do not contribute to the query's answer, leading to I/O efficient query processing.

## 5.3 Robustness

As we detailed earlier the cost model depends on an assumed query size, both on the temporal and in the spatial component. Next we discuss how the performance is affected when the user estimates one query size but the actual posed queries have a different size. Ideally, one would want the performance to be robust, i.e., to not degrade much more with reasonable variances between the assumed and actual query sizes. In all forthcoming tables the values

| | $S=0.05,T=0.1$ | $S=0.1,T=0.1$ | $S=0.2,T=0.1$ |
|---|---|---|---|
| $S=0.05,T=0.1$ | **29.32** | 34.77 | 43.08 |
| $S=0.1,T=0.1$ | 60.37 | **60.27** | 73.65 |
| $S=0.2,T=0.1$ | 169.42 | **147.94** | 156.29 |

(a) UNIFORM dataset

| | $S=0.05,T=0.1$ | $S=0.1,T=0.1$ | $S=0.2,T=0.1$ |
|---|---|---|---|
| $S=0.05,T=0.1$ | **244.98** | 288.32 | 303.98 |
| $S=0.1,T=0.1$ | **445.69** | 486.56 | 505.63 |
| $S=0.2,T=0.1$ | 1062.74 | 1053.73 | **1026.65** |

(b) GSTD dataset

| | $S=0.05,T=0.1$ | $S=0.1,T=0.1$ | $S=0.2,T=0.1$ |
|---|---|---|---|
| $S=0.05,T=0.1$ | **1624.96** | 1647.83 | 1716.12 |
| $S=0.1,T=0.1$ | **2216.70** | 2228.41 | 2364.74 |
| $S=0.2,T=0.1$ | **2780.47** | 2782.19 | 28.24.00 |

(c) INFATI dataset

Table 3: I/O robustness of SPIT for all three datasets with respect to spatial query size (temporal query size is fixed).

in the first row represent the query sizes assumed at index contruction time, while the values on the first column are the sizes of the issued queries.

In Tables 3(a), (b) and (c) we can see the performance obtained when the spatial component of the query varies, and the temporal range is fixed for all three datasets. Tables 4(a), (b) and (c), on the other hand, show the performance when the temporal range varies and the spatial query remains fixed.

It is interesting to note that the smallest number does not always appear in the diagonal of the tables as one would have liked. The main reason for this is that even in the case of the UNIFORM dataset, the data cannot be said to fit perfectly to the the cost model assumptions, thus some variation is not surprising. Another reason, that applies to the GSTD and INFATI datasets is that the data partitioning follows an heuristic and is therefore expected to be sub-optimal. Nevertheless, the most important property, that is having performance not varying too much if one builds the dataset assuming a wrong, within reasonable limits, average query is observed throughout the table.

Overall, these results serve to show that SPIT is indeed a robust approach with respect to the assumed query size. That is to say, that even if the user estimated query size, for building the indices, is off by a factor of two or four in either the spatial or temporal dimension, SPIT is still able to deliver good performance.

## 5.4 Index Creation

Our final remarks on the experiments deal with time required to create and index the database. In this regard, the Linear Scan is obviously the most efficient since there is no overhead associated to it. This comes at the expense of inefficient query performance as detailed above. The times reported were obtained on a PC with an AMD Athlon XP 3200+ running at 2.19GHz and with 1.00GB of RAM, and using the GSTD dataset with 2.5 million objects.

There are two main tasks that need be performed within SPIT. First, the partitioning must be obtained using the heuristic algorithm presented in Section refmodel. After that the objects need to be inserted in the correct partitions and, finally, the local B-trees (one per partition are created). SPIT required about 865 sec to build the partitions and inserting the data objects onto those, with an additional 33 sec needed to created the local B-trees, for a total of 898 sec. The R-tree+B-tree approach, on the other hand needed only 200 sec to insert the data on the (single) table but needed 784 sec to build the the indices, for a total of 984 sec. It should be noted that both approaches made use of SQL*Loader facility available at typical Oracle installations.

Even though SPIT is overall about 10% faster we could observe that the partitions look-up pose most of the overhead at data insertion time. We did considered the idea of building an index to speed up the partition lookup,

|  | $S = 0.1, T = 0.05$ | $S = 0.1, T = 0.1$ | $S = 0.1, T = 0.2$ |
|---|---|---|---|
| $S = 0.1, T = 0.05$ | **38.16** | 42.29 | 42.09 |
| $S = 0.1, T = 0.1$ | 62.91 | **60.27** | 60.86 |
| $S = 0.1, T = 0.2$ | 113.58 | 101.26 | **87.42** |

(a) UNIFORM dataset

|  | $S = 0.1, T = 0.05$ | $S = 0.1, T = 0.1$ | $S = 0.1, T = 0.2$ |
|---|---|---|---|
| $S = 0.1, T = 0.05$ | 328.80 | 302.27 | **298.34** |
| $S = 0.1, T = 0.1$ | 518.99 | 486.56 | **446.05** |
| $S = 0.1, T = 0.2$ | 816.80 | 770.39 | **675.40** |

(b) GSTD dataset

|  | $S = 0.1, T = 0.05$ | $S = 0.1, T = 0.1$ | $S = 0.1, T = 0.2$ |
|---|---|---|---|
| $S = 0.1, T = 0.05$ | **1838.15** | 1843.23 | 1955.60 |
| $S = 0.1, T = 0.1$ | 2364.74 | 2228.41 | **2215.77** |
| $S = 0.1, T = 0.2$ | 3159.20 | 2974.37 | **2949.53** |

(c) INFATI dataset

Table 4: I/O robustness of SPIT for all three datasets with respect to temporal query size (spatial query size is fixed).

but the number of partitions was fairly low (close to 100) and it would not benefit from an index, as compared to a simple linear scan of the partitions table.

Recall that SPIT was designed to handle historical spatio-temporal data, as such handling updates should not be a concern. SPIT is a feasible alternative for a scenario where data is collected and at one point (later) in time is to be queried. Even for large datasets index creation should be such that one can start querying data not much longer after it was collected.

## 6 Conclusions and Future Work

The Space-Partitioning with Indexes on Time (SPIT) approach leverages existing RDBMS technology by providing support for "out-of-the-box" RDBMS based management of historical spatio-temporal point data. SPIT is based on a cost model aiming at optimizing query cost (I/O). For the case of a uniform data distribution the cost model provides an optimal partitioning of the dataset. For arbitrary data distributions, the cost model is used to guide a heuristic partitioning which leads to very good query performance. SPIT has been shown to outperform other alternatives by a large margin for spatio-temporal data management, also to be robust with respect to the query size assumed at index construction time. While the use of spatial partitioning to index the spatial component of the data is a well known method, applying this strategy in the spatio-temporal domain while providing automatic and tightly integrated RDBMS support has not been done before.

We are considering ways in which SPIT's partitioning could be periodically re-adjusted as the database size increases. Even though some preliminary experimental results suggest that SPIT is resilient to modest increases in database size, rebuilding the index is bound to be necessary after some point in time. It would be useful to develop a technique to automatically determine such point(s) in the database lifetime. We also plan to investigate whether a self-adaptation scheme, where the RDBMS would re-configure partitions by itself without having to rebuild the whole index, can be developed. This would help make SPIT even more scalable and adaptable for very large databases.

Finally, we are currently investigating how to extend the proposed SPIT approach in order to handle trajectories, obtained, for instance, by interpolating two subsequent observations. A query of interest in such a case would be find trajectories that intersect a given spatial range within a determined time window, where possibly no observed data point actually falls within the queried range/time.

## Acknowledgments

Research partially supported by CFI, iCORE Alberta, NSERC Canada, and Profilium, Inc.

## References

[1] Kothuri, R., Ravada, S.: Spatio-Temporal Indexing in Oracle: Issues and Challenges. IEEE TCDE Bulletin **25** (2002) 56–60

[2] Mokbel, M., Ghanem, T., Aref, W.: Spatio-Temporal Access Methods. IEEE TCDE Bulletin **26** (2003) 40–49

[3] Pfoser, D., Jensen, C., Theodoridis, Y.: Novel Approaches in Query Processing for Moving Object Trajectories. In: Proc. of VLDB. (2000) 395–406

[4] Chakka, V., et al.: Indexing Large Trajectory Data Sets With SETI . In: Online Proc. of CIDR. (2003) [http://www-db.cs.wisc.edu/cidr/program/p15.pdf].

[5] Jensen, C., Lin, D., Ooi, B.C.: Query and Update Efficient B$^+$-Tree Based Indexing of Moving Objects. In: Proc. of VLDB. (2004) 768–779

[6] Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In: Proc. of VLDB. (2003) 790–801

[7] Saltenis, S., et al.: Indexing the Positions of Continuously Moving Objects. In: Proc. of ACM SIGMOD. (2000) 331–342

[8] Abdelguerfi, M., et al.: The 2-3TR-tree, a Trajectory-Oriented Index Structure for Fully Evolving Valid-Time Spatio-Temporal Datasets. In: Proc. of ACM GIS. (2002) 29–34

[9] Nascimento, M., Silva, J.: Towards historical R-trees. In: Proc. ACM SAC. (1998) 235–240

[10] Theodoridis, Y., Vazirgiannis, M., Sellis, T.: Spatio-Temporal Indexing for Large Multimedia Applications. In: Proc. of IEEE ICMCS. (1996) 441–448

[11] Guttman, A.: R-trees: a dynamic index structure for spatial searching. In: Proc. of ACM SIGMOD. (1984) 47–57

[12] Tao, Y., Papadias, D.: MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In: Proc. of VLDB. (2001) 431–440

[13] Kriegel, H.P., Pötke, M., Seidl, T.: Managing Intervals Efficiently in Object-Relational Databases. In: Proc. of VLDB. (2000) 407–418

[14] Nascimento, M., Dunham, M.: Indexing valid time databases via B+ -trees – the MAP21 approach. IEEE TKDE **11** (1999) 1–19

[15] Theodoridis, Y., Sellis, T.: A Model for the Prediction of R-tree Performance. In: Proc. of PODS. (1996) 161–171

[16] Lewis, P., A.B., Kifer, M.: Database and Transaction Processing. Addison-Wesley (2002)

[17] Samet, H.: The Quadtree and Related Hierarchical Data Structures. ACM Comput. Surveys **16** (1984) 187–260

[18] Ross, S.: Introductory Statistics. McGraw-Hill (1996)

[19] Theodoridis, Y., Silva, J.R.O., Nascimento, M.A.: On the Generation of Spatiotemporal Datasets. In: Proc. of SSD. (1999) 147–164

[20] Jensen, C., et al.: The INFATI data. Technical Report TR-79, TimeCenter (2004) [http://arxiv.org/abs/cs.DB/0410001].

[21] Mallett, D.: Relational database support for spatio-temporal data. Technical Report TR04-21 (M.Sc. Thesis), Dept. of Computing Science, Univ. of Alberta (2004) [http://www.cs.ualberta.ca/TechReports/2004/TR04-21/TR04-21.pdf].