Geoinformatica (2008) 12:143–168
DOI 10.1007/s10707-007-0030-3

PIST: An Efficient and Practical Indexing Technique for Historical Spatio-Temporal Point Data

Viorica Botea · Daniel Mallett · Mario A. Nascimento · Jörg Sander

Received: 5 October 2006 / Revised: 13 April 2007 /
Accepted: 20 April 2007 / Published online: 17 August 2007

Abstract Despite pressing need, current relational database management systems (RDBMS) support for spatio-temporal data is limited and inadequate, and most existing spatio-temporal indices cannot be readily integrated into existing RDBMSs. This paper proposes a practical index for spatio-temporal (PIST) data, an indexing technique, rather than a new indexing structure, for historical spatio-temporal data points that can be fully integrated within existing RDBMSs. PIST separates the spatial and temporal components of the data. For the spatial component, we develop a formal cost model and a partitioning strategy that leads to an optimal space partitioning for uniformly distributed data and an efficient heuristic partitioning for arbitrary data distributions. For the temporal component of the data a B^+-tree is used. We show that this layer's performance can be maximized if an optimal maximal temporal range is enforced, and we present a procedure to determine such an optimal value. Being fully mapped onto a RDBMS, desirable and important properties, such as concurrency control, are immediately inherited by PIST. Using ORACLE as our implementation platform we perform extensive experiments with both real and synthetic datasets comparing its performance against other RDBMS-based options, as well as the MV3R-tree. PIST outperforms the former by at least one order of

V. Botea · D. Mallett · M. A. Nascimento (✉) · J. Sander
Department of Computing Science, University of Alberta,
Edmonton, Alberta, Canada
e-mail: mn@cs.ualberta.ca

V. Botea
e-mail: viorica@cs.ualberta.ca

D. Mallett
e-mail: mallett@cs.ualberta.ca

J. Sander
e-mail: joerg@cs.ualberta.ca

magnitude, and is competitive or better with respect to the latter, with the unarguable advantage that it can readily used on top of virtually any existing RDBMS.

Keywords relational database management system · spatio-temporal data · rdbms · indexing

1 Introduction

The need for spatio-temporal 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 collections of such data demands the convenience, reliability, and data storage capabilities that a traditional RDBMS affords. However, little work has been done on providing spatio-temporal data support, a STAM in particular, inside a RDBMS, [7] being an exception. Our work fills this crucial need by proposing a spatio-temporal access technique which can be fully integrated within any RDBMS.

There are two main types of spatio-temporal databases, those that manage historical information and those that manage current information for current/predictive query purposes [11]. 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 each database record has the format $\langle oid, x, y, t_s, t_e \rangle$, where *oid* identifies an object, (x, y) are spatial coordinates, and $[t_s, t_e)$ indicate the open-ended non-null interval during which an object remained at position (x, y). 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 [13], 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, and unacceptable, assumptions. For example, in security/monitoring applications, a person could be mistakenly assumed to have entered a restricted area, instead of gone around it, because his/her movement was interpolated. Our model can be viewed as a step-wise interpolation instead of a linear interpolation, i.e., for a given object o_k, two consecutive observations have the form $\langle o_k, x', y', t'_s, t'_e \rangle$, and $\langle o_k, x'', y'', t''_s, t''_e \rangle$, where $t''_s = t'_e$ and $(x', y') \neq (x'', y'')$. That is, as long as the object's position is not updated in the database it is assumed to remain stationary in its last observed position.

In this work a spatio-temporal range query Q takes the form $Q = \langle \sigma, \tau \rangle$ where σ is a spatial region and τ is a time range. Q returns the unique *oid*'s of records where (x, y) is inside σ and $[t_s, t_e)$ intersects with τ. An example of such a query would be "find all people who were inside Louvre's Salle des Etats at some point between noon and 1 p.m. yesterday".

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

In this paper we propose an efficient spatio-temporal indexing technique fully integrated within a RDBMS via a relational mapping. Our approach is based on a two-layer approach. First the data space is partitioned into a grid. This can be done at the logical or physical level of the underlying RDBMS. At query time only those grid cells that actually intersect the query's spatial region need be inspected. Typically this intersection represents a relatively small portion of the total space and hence of the database. Within each grid cell a temporal index is built in order to speedup finding the objects that also satisfy the query's temporal predicates.

The general framework of the practical index for spatio-temporal (PIST) data is similar to scalable and efficient trajectory index (SETI) [3]. PIST, however, introduces several new and important features, namely: (1) We develop a cost model to determine the optimal number of grid partitions to use. The model suggests the number of primary partitions so that the expected number of disk accesses is minimized, assuming a uniform data distribution and an average expected query size. There are no guidelines on how to partition the space within SETI's approach. (2) Based on this cost model, we propose a heuristic for partitioning the data space for arbitrary data distributions, which yields very good performance in practice. (3) We rely only on the availability of B^+-trees, unlike SETI which uses a one dimensional "sparse" R-tree for indexing temporal ranges.[2] (4) We propose an optimal splitting of temporal ranges that can speed up considerably the processing time of the query's temporal predicate. In SETI's original paper there is no similar concern. (5) We present a relational mapping which can be used to deploy PIST using any RDBMS without requiring any modifications to the same. SETI and its associated indices were implemented by adapting, rather than simply reusing, SHORE [2], a prototype object storage engine.

We show in a comprehensive experimental comparison that our proposed technique dramatically outperforms other practical alternatives for spatio-temporal indexing. Finally, we show that PIST's performance is at least comparable to the MV3R-tree [17], which is a specialized, and unlike SETI, parameter-free efficient index for historical spatio-temporal data but which is not practical in the sense that it cannot be easily implemented on top of a RDBMS. In summary, we propose, investigate and evaluate an efficient and ready-to-use effective solution for the problem of indexing historical spatio-temporal data.

The remainder of this paper is structured as follows. The next section reviews related work. Section 3 details our proposed approach, and the associated cost model. In Section 4 we describe how our approach can be implemented using ORACLE. In Section 5 we confirm the reliability of the model and compare our approach to other methods for indexing spatio-temporal data. Section 6 concludes the paper and offers directions for further research.

2 Related work

A thorough overview of work on STAMs for historical and current/predictive spatio-temporal support can be found in [11]. Predictive STAMs support queries that

[2]While some RBDMS do offer R-trees as a native index those are not nearly as widely available as B^+-trees.

predict a moving object's location at a future point in 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 [13].

The current state-of-the-art for predictive STAMs is the B^x-tree [6]. 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 [18]. It improves on the TPR-tree construction algorithms [15], which were based on the classical R*-tree, i.e., for static data, in order to achieve near-optimal performance.

Many historical STAMs have been proposed [1], [3], [12], [13], [21] the majority of which are based on the R-tree [3], [4] being a notable exception. The 3-D R-Tree [21] treats time as a third dimension and indexes spatio-temporal data using a 3-dimensional R-tree. The historical R-tree [12], an overlapping and multi-version structure, adapts the R-tree for historical spatio-temporal data. The MV3R-tree [17] improves upon these providing more efficient support for interval queries. It uses a Multi-Version R-tree with 3D R-trees built on its leaf nodes. The MV3R-tree, like we do, assumes step-wise interpolation of the observed positions. The trajectory bundle tree (TB R-tree) [13] proposes a trajectory-oriented access method that can (under certain conditions) answer trajectory-oriented queries faster than the R-tree. The 2-3TR-tree [1] suggests the use of two R-tree indices, a two-dimensional point index representing current data, and a three-dimensional historical index.

SETI [3] is the work closest to ours, where the authors propose a grid-based spatio-temporal indexing technique. It partitions the spatial dimension into static, non-overlapping partitions, and within each partition it uses a sparse temporal index—which the paper describes as a one-dimensional sparse R-trees. An in-memory "front-line" structure keeps track of the last position of each moving object. As detailed in Section 1, there are several aspects upon which our technique improves on SETI's, e.g., the use of cost models to guide the partitioning of the space, the indexing of time ranges and the relying on only Structured Query Language (SQL)-based mappings onto a RDBMS.

3 PIST: An indexing technique for Spatio-Temporal data

In what follows we present, PIST, a practical index for spatio-temporal (historical) data. A technique rather than a new access structure, PIST partitions the data space according to the spatial location of the indexed objects and then creates temporal indices over each partition. The historical data is partitioned into a fixed number of cells, each cell corresponding to a different partition in the RDBMS. Recall that we are only concerned with indexing and querying historical observations, which are completely available at index creation time. Therefore, we do not consider updates at this point, constructing a static partitioning. 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 spatio-temporal 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

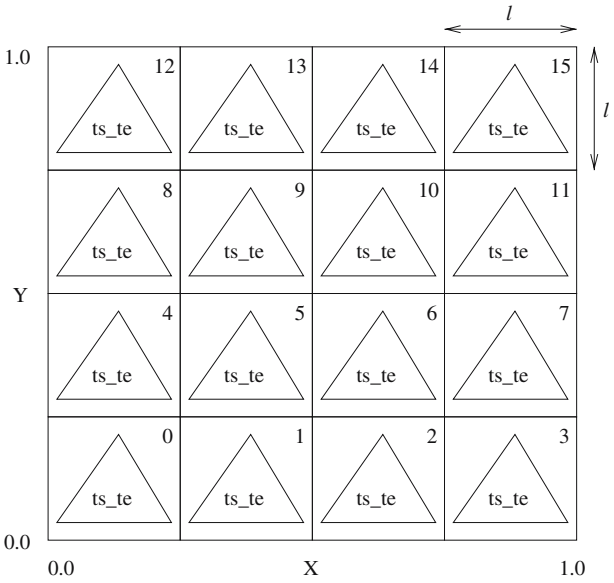


Fig. 1 PIST's approach for a 4 × 4 regular grid

only a (small) subset of the data. As in [3], 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 indices,
3. fine *spatial refinement* based the on actual spatial location of tuples, and
4. *duplicate elimination*.

As shown in Fig. 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, and for that we use a B$^+$-tree. 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) can be prohibitively expensive. In fact, preliminary results [10] have shown this to be the case for the temporal RI-tree [8] as well. It is important to note that not only can the 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 very efficiently.

We also assume that the spatio-temporal data is stored in an index-organized table. An index-organized table is one table which is embedded within an index, i.e., the index contains the tuples themselves in the leaf nodes of the index, e.g., a B$^+$-tree. Although index-organized tables are supported by existing DBMSs, e.g., ORACLE, their use is not mandatory for our proposal. It could be implemented using a regular index and tables, the only difference would be some extra I/Os

Algorithm 1 *st_query*() function.

INPUT: $\langle \sigma, \tau \rangle$
OUTPUT: list of *oid*'s

```
1:  pid_list := p_intersect(σ)
2:  for all pid in pid_list do
3:    oid_list := oid_list ∪
4:          SELECT oid
5:          FROM pid
6:          WHERE t_s BETWEEN τ.t_min − 𝒯 AND τ.t_max
7:          AND t_e BETWEEN τ.t_min AND τ.t_max + 𝒯
8:          AND x between σ.x_min and σ.x_max
9:          AND y between σ.y_min and σ.y_max
10: end for
11: sort oid_list and remove duplicates
12: return oid_list
```

required for obtaining the actual tuples from a table after traversing the index. Finally, if the index-organized table is built on the temporal attributes, at query time an efficient sequential range scan of the index leaves can be performed on disk over the range where tuples intersect the temporal query interval.

Algorithm 1 provides the pseudo-code for the function *st_query*(), which processes queries using the PIST model. The algorithm assumes that the function *p_intersect*() exists. 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 the query processing only those tuples satisfying the previous stage are further examined.

Algorithm 1 uses the fact that the largest temporal interval, denoted as $\mathcal{T}$, is known. This is a reasonable and practical assumption, e.g., in fleet monitoring it can be safe to assume that no vehicle remains stationary for more than 2 or 3 days. Knowledge of a dataset's $\mathcal{T}$ serves to further restrict the temporal range that needs to be inspected at query time, hence improving query processing time. In fact, given one value of $\mathcal{T}$ one can easily split all records whose temporal range length exceeds $\mathcal{T}$ into two or more records that adhere to the assumption. As can be easily seen in Algorithm 1, the smaller the value of $\mathcal{T}$, the smaller the range scan on the temporal index. There is, however, a trade-off in splitting the dataset's temporal ranges. It increases the number of indexed temporal ranges and hence the number of records in the database. We explore this issue in Section 3.3 in order to obtain an optimal value of $\mathcal{T}$ for a given distribution of temporal ranges.

PIST was designed to handle historical spatio-temporal data, as such handling updates, i.e., new observations is not a concern. PIST is a feasible alternative for a scenario where data is collected to be queried at a later point in time. Given an indexed (historical) dataset, a new dataset can be merged with the current one using the existing partitioning or a new index could be built altogether for the newly combined database. The former may yield sub-optimal performance, depending on the size and spatial distribution of the new dataset. From this perspective the latter

Table 1 Notation used

Notation	Meaning
N	Number of tuples, i.e., observations, in the dataset
DA	Number of disk I/Os to answer a query
GA	Average number of accessed grid cells
DA_g	Number of data I/Os per accessed grid cell
IA_g	Number of index I/Os per accessed grid cell
BS	Block size (number of tuples) per data page
q_s	Average size (%) of the query in each spatial dimension wrt the modeled space
q_t	Average size (%) of the temporal range of the query wrt the number of observed timestamps
l (l^*)	Length (optimal length) of a grid cell per dimension
N_g (N_g^*)	Total (optimal) number of cells in the grid
$\mathcal{T}$ ($\mathcal{T}^*$)	Maximal (optimal maximal) length (%) of the indexed temporal ranges wrt the number of observed timestamps

is a better option, and, as our experiments will show, index building times are quite reasonable for most practical purposes. Another possibility could be to create several indices for different time periods, e.g., one per week. In this case queries would have to be re-written for handling the case where they span over several such indices.

3.1 Optimal partitioning of the space

We assume the average query size, on both the temporal and spatial dimensions, are known–the robustness of PIST with respect to such an assumption is discussed in the experimental Section 5.4. We also assume the spatial domain to be the unit square and that the temporal domain is formed by the set of recorded timestamps. Note that this means the temporal domain of time points is bounded by the finite number of observations that exist in the dataset. Table 1 lists the notation we will use for the cost model presentation.

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/Os performed inside each accessed grid cell, i.e., the combination of reads to the data and reads to the temporal index structure inside each grid cell. This can be formalized as:

$$DA = GA \times (DA_g + IA_g) \tag{1}$$

As per [19], 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, thus:

$$GA = N_g(l + q_s)^2 \tag{2}$$

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 extended by $\mathcal{T}$ (recall that we assume the maximal recorded temporal length is known), i.e.:

$$DA_g = \frac{N/N_g}{BS} \times (q_t + \mathcal{T}) \tag{3}$$

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: $IA_g = log_f N/N_g$. We simplify the index access cost to $IA_g = 3$, which is typical for indices with $f \approx 100$ and N in the millions of tuples [9], obtaining:

$$DA = (l + q_s)^2 \left(\frac{N \times (q_t + \mathcal{T})}{BS} + \frac{3}{l^2} \right) \tag{4}$$

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 indices within them. On the other hand, increasing the query's temporal range requires only a larger scan on the indices, which can be done efficiently.

After some algebraic manipulation it can be shown that the grid size l^* that will minimize disk accesses is given by

$$l^* = \sqrt[3]{\frac{3q_s \times BS}{N \times (q_t + \mathcal{T})}} \tag{5}$$

Finally, the optimal number of grid cells N_g^* can be represented in terms of l^* using

$$N_g^* = \frac{1}{(l^*)^2} = \left(\frac{N \times (q_t + \mathcal{T})}{3q_s \times BS} \right)^{2/3}. \tag{6}$$

Thus, in order to minimize the number of disk access per query given an average query size, we must create a *regular* partitioning of the data space by creating $\left\lceil \sqrt{N_g^*} \right\rceil$ equally sized partitions along each dimension. In addition note that $\mathcal{T}$ in Eq. 6 is the only parameter one could fine-tune, the others are query or system dependent. In Section 3.3 we discuss how one can explore this in order to further improve performance.

3.2 Heuristic partioning of arbitrary distributions

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, 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 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 [16], until all obtained subspaces satisfy a uniform distribution criteria. The obtained cells are then partitioned using the cost model developed above. The uniformity of

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 in the current MBR, and partition the MBR optimally using the cost model. Using the resulting grid cells as categories, perform Pearson's Chi-Square test on the current MBR.
2: **if** the Chi-Square test is successful, i.e., the data distribution within the MBR can be considered uniform **then**
3: Store the (coordinates of the) grid cells of the current MBR as partitions in the table `Partitions`
4: **else**
5: Split each dimension of the current MBR in half, obtaining MBR_i, $i = 1, 2, 3, 4$
6: **for** i=1 to 4 **do**
7: *Partition*(MBR_i)
8: **end for**
9: **end if**

a data distribution can be determined using Pearson's Chi-Square test [14]. 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. 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.

It should be clear that if the data is truly uniformly distributed, the heuristic presented above yields an optimal regular grid partitioning (under the cost model assumptions). In such a case the uniformity test would be immediately successful and the algorithm would not recurse.

It may appear at first that the partitioning strategy may result in many small cells with very few objects in each of them. This obviously would not be a good idea since there is an overhead cost to access a partition, and there is a point where accessing less data in more partitions is more expensive than accessing more data within less partitions. Fortunately, the heuristic above identifies such situation and stops the partitioning accordingly. Recall that, during the partitioning, q_s^2 is the query size with respect to the current Minimum Bounding Rectangle (MBR), and similarly N is the number of objects inside the current modeled space, i.e., the current MBR. Initially the current MBR is the whole unit square, but as the partitioning progresses, the MBRs are subdivided and the current MBRs become smaller. As an obvious consequence, q_s becomes larger with respect to the current MBR. 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 Eq. 6 one can see that if $q_s = 1$ and BS and q_t are constants, then $N < \frac{3BS}{(q_t+\mathcal{T})}$ yields $N_g^* = 1$, i.e., no further partioning is needed. This agrees with the intuition that as the partitioning progresses, there is a point where accessing less data in more partitions becomes more likely and more expensive than accessing more data within a single partition. At that point the partitioning process stops automatically.

Although only optimal for the case of uniformly distributed data, the resulting overall performance by PIST 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, PIST 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.

We note that since PIST is designed to handle historical data, changes in the data distribution is not an issue of concern. Nevertheless, it remains an interesting open problem how to be able to detect whether there has been sufficient change in the distribution which would warrant a complete (or partial)re-partioning of the data.

3.3 Optimizing $\mathcal{T}$

As mentioned earlier the size of the range scan on the B$^+$-tree indexing the temporal ranges depends on the length of the largest indexed range $\mathcal{T}$. There is nothing however, that prevents one to setting $\mathcal{T}$ "artificially" in order to optimize the index performance. As one decreases the value of $\mathcal{T}$ from its so-called intrinsic value, i.e., that inherent to the dataset, to zero, the number of temporal ranges that have to be split increases. As a consequence the range scan on the index will become shorter, and the number of indexed objects will increase. The former has potential positive impact on the performance of the temporal index while the latter has a negative effect.

In the following we show how to balance those two effects and obtain an optimal value for $\mathcal{T}$, denoted as $\mathcal{T}^*$, taking advantage of the fact that we are dealing with historical data, i.e., the distribution of the temporal ranges is known at index construction time.

Let $C(l_i)$ be the count of the number of temporal ranges with length equal to l_i, i.e., $\langle C(l_1), C(l_2), ..., C(l_M)\rangle$ is the histogram of the distribution of the temporal range lengths. Note that M is finite as long as one assumes a discrete time space, otherwise it can be made so to the user's discretion, with no loss of generality of the argumentation that follows.

Let $l_k \in \{l_1, l_2, ...l_{M-1}\}$ be one given length that is going to be set as the maximal length. (The case where $l_k = l_M$ induces no splits and therefore is not of interest.) When splitting all ranges larger than l_k the current number of indexed ranges, originally N, will now become $N_k = N + \sum_{p=k+1}^{M}[C(l_p) \times (\lceil l_p/l_k \rceil - 1)]$. Returning to Eq. 6, we want to minimize $N \times (q_t + \mathcal{T})$, hence we set $\mathcal{T}^* = l_k$ where $k = \arg min_k\{N_k \times (q_t + l_k)\}$.

The more skewed the distribution of the temporal range lengths is towards shorter ranges, the more potential for savings exist, i.e., splitting a few long ranges has the effect of substantially decreasing the value of $\mathcal{T}$ without increasing the number of indexed ranges N noticeably.

Note that in the case a user is given or wishes to impose a storage budget that can be used for optimizing performance, e.g., the database can grow to up to N^{max} tuples due to the splitting, the problem can be solved similarly. In this case the (potentially sub-optimal) solution, is found by simply finding k such that $N_k \times (q_t + l_k)$ is minimized subject to $N_k - N \leq N^{max}$.

```
create table ST_PIST (
   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 p04 values less than (16)
)
```

a A sample PIST table

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

b Querying PIST's data. (In this piece of SQL code `T` plays the role of the optimal range length $\mathcal{T}^*$.)

Fig. 2 PIST's mapping onto a RBDMS

Finally, finding the optimal $\mathcal{T}^*$ has linear complexity on the number of distinct indexed lengths, which can be arbitrarily discretized.

4 PIST's implementation

The PIST 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. A unique partition id (`pid`) along with its MBR is stored in a table called `Partitions`. The `ST_PIST` table (whose Data Definition Language (DDL) for an example 4×4 grid is sketched in Fig. 2(a)) stores records along with the additional *pid* attribute. When inserting an object into table `ST_PIST` 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 PIST to work. A RDBMS which does not provide such a facility can still 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]", Fig. 2b provides the SQL query that would be issued against the `ST_PIST` table created in Fig. 2a.

Line 3 of the sample query corresponds to the *spatial filtering* stage of PIST'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 and 5 correspond to the *temporal filtering* stage of PIST'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 and 7 correspond to the *spatial refinement* stage of PIST'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 PIST's query processing.

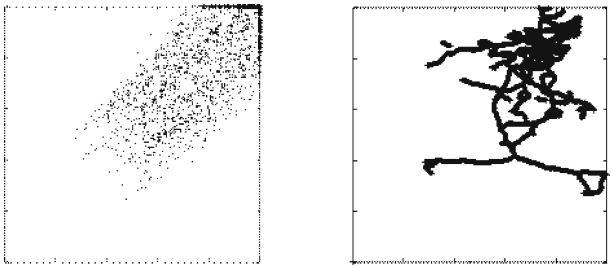
We defined a PL/SQL function that generates dynamic SQL queries of the form provided in Fig. 2b given a query spatial and temporal range. We choose to implement the algorithms using PL/SQL because of the ease of integration between PL/SQL and SQL queries in ORACLE, however, any language capable of interacting 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 sets, denoted as UNIFORM, has the objects uniformly distributed in the space and moving freely throughout the whole space. This satisfies the assumptions for PIST's cost model (v. Section 3.1). The second synthetic dataset was generated using the GSTD tool[3] [20] 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 Fig. 3a, where all observed positions for a sample of 100 objects are shown. This dataset is more realistic, 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 [5]. 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 Fig. 3b where all observations are plotted—one can clearly see the notion of actual roads in this case.

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 investigated unit 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. For the temporal query component we performed experiments using the query range equal to 5%, 10% and 20% of all observed timestamps. Table 2 summarizes the parameters used for the experiments. Unless otherwise mentioned

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



a GSTD dataset **b** INFATI dataset

Fig. 3 Data distribution for the GSTD and INFATI datasets

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) per query 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 PIST'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 an off-the-shelf RDBMS.) The first approach is a simple *Linear Scan* which should provide the lower bound for expected performance. The second method uses an R-tree for the spatial component along with a B^+-tree for the temporal component. We adapt the Linear Referencing System (LRS) spatio-temporal indexing approach suggested by ORACLE [7] 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".

Note that in R-tree+B^+-tree scheme, just like within PIST, the temporal ranges are indexed using a B^+-tree, meaning that it can potentially benefit from the knowledge of $\mathcal{T}$ as well. However, finding an optimal value of $\mathcal{T}$ for this case, as opposed for PIST (Section 3.3) is not trivial. The reason being that splitting an observation into two or more now also affects the R-tree as the split objects modulo timestamps are, in a sense, "replicated." Our intuition suggests that if the number of split objects is not very large, e.g., as in the presence of relatively few long temporal ranges, the effects on the R-tree are not very large. On the directory level a large number of splits would be needed to cause relevant changes in the structure. On the leaf level the number of I/Os will increase proportionally to the increase in the number of indexed objects, which we assume to be not very high. In addition this effect is mitigated

Table 2 Parameters and respective values investigated

Parameter	Values (default in bold)
Average q_s [% of data space]	0.25%, **1%** and 4%
Average q_t [% of timestamps]	5%, **10%** and 20%
N [millions of observations]	1, **2.5** and 5

by the efficiency of the underlying range scan of the B^+-tree. Considering all these factors, we decided to use within the R-tree+B^+-tree the same value of $\mathcal{T}^*$ derived for PIST. Nonetheless, as we shall see shortly the difference in performance between PIST and R-tree+B^+-tree is so large that finding the true optimal value for $\mathcal{T}$ for the latter would very unlikely improve its performance by a factor large enough to make it a competitive approach.

We also used a scheme that indexes Z-values of each tuples' spatial coordinates using a B^+-tree, with an additional B^+-tree for the temporal component. For each dataset, we calculated Z-values using the same number of cells in each dimension that PIST 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.

Finally, we also compare PIST to the MV3R-tree [17] using the source code kindly made available by its authors.[4] Even though the MV3R-tree is not an index that can be easily mapped onto an RDBMS, and therefore lacks the practical aspect that PIST promotes, it is a well known index for historical spatio-temporal data which makes the same assumption about discrete object movement as we do, and it has been shown to outperform a simple 3D R-tree, which would have been another competitor for PIST.

5.1 Partitioning effectiveness

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.

When using all experimental default values and a block size of 8,192 bytes our cost model determines a 13×13 grid, which indeed is the best option when compared to several other choices for a regular partitioning of the data space as shown in Fig. 4a.

It is interesting to note that when the number of partitions is smaller than the optimum there is increasingly sharp overhead due to reading more data per partition than it would be necessary in the optimal case. Similarly, but not as severe, 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. This behavior was also verified in [3].

As discussed earlier, for non-uniform distributions PIST 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 Fig. 4b,c, for both non-uniform distributions the grid partition determined automatically by PIST 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 PIST performance.) Again, the additional cost

[4] http://www.cs.cityu.edu.hk/~taoyf/codes/mvr.zip

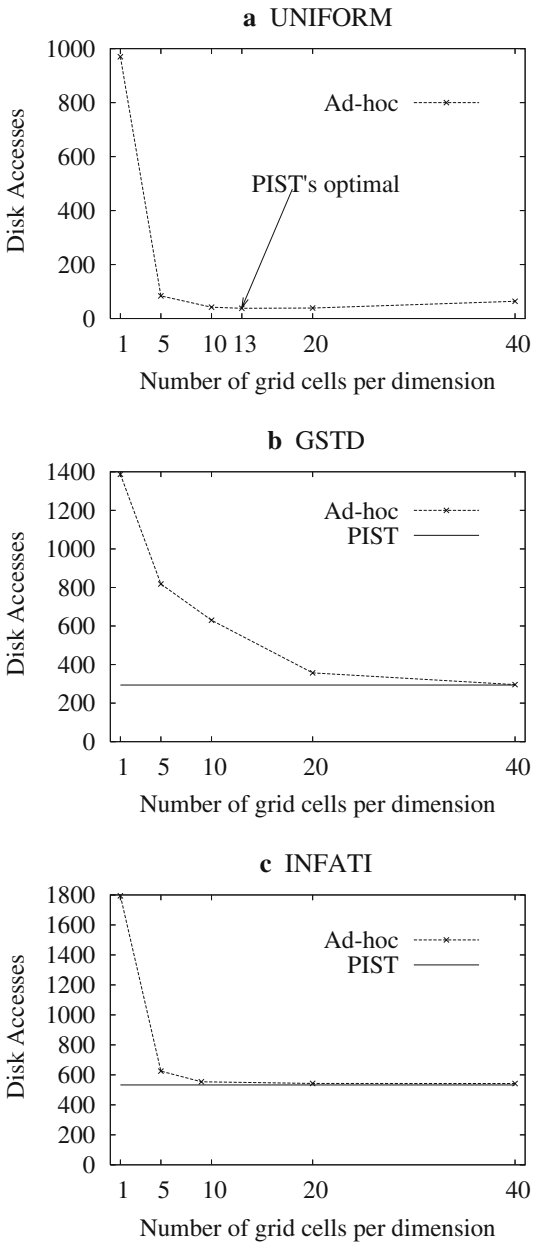


Fig. 4 Comparing I/O performance yielded by PIST's partitioning against the use of ad-hoc regular grids

of underpartitioning is clear, but unlike in the case for Uniform data, overpartitioning seems to be not as prejudicial.

5.2 Optimizing $\mathcal{T}$

As discussed earlier, adjusting the value of $\mathcal{T}$ yields a trade-off between improving query performance and enlarging the database. Our argument is that the more skewed the temporal ranges are the more worthwhile is enforcing the optimal

Table 3 Performance improvement and storage overhead due to $\mathcal{T}$

	I/Os w/ $\mathcal{T}$	I/Os w/ $\mathcal{T}^*$	Perf. gain (%)	Storage overhead (%)
UNIF	78.4	52.9	32	80
UNIF+exp(0.5)	73.5	39.5	46	16
UNIF+exp(4)	62.3	20.9	67	4
INFATI	2,267.8	533.1	77	0.5

value $\mathcal{T}^*$. In order to investigate this, we used four datasets. Three datasets are synthetic datasets having a uniform spatial distribution (since the optimization is on the temporal level, the spatial distribution is irrelevant), with varying degrees of skewedness on the temporal ranges length. One is simply uniformly distributed and the other have the range lengths following an Exponential distribution, with parameters λ equal 0.5 and 4 (the larger λ the more skewed the distribution). Those are denoted by UNIF+exp(0.5) and UNIF+exp(4) respectively. Finally, as the fourth dataset we used INFATIs' as a representative of a realistic distribution.

Table 3 shows the obtained performance when using both the dataset's intrinsic $\mathcal{T}$ and the optimal $\mathcal{T}^*$ obtained as described in Section 3.3, as well as the yielded storage overhead. The dataset sizes as well as the query sizes used were the default values in Table 2.

Clearly, the more skewed the distribution of the lengths of the temporal ranges towards shorter ranges, the better the improvement in query processing time and the smaller the storage overhead. The skewedness of the temporal ranges is in fact a realist assumption, as evidenced by INFATI's distribution, which not coincidentally yielded the largest improvement with the smallest overhead. Even though figures are not presented here, the gains are even larger when the temporal query range is smaller. This is due to the fact the range scan on the B^+-tree leaves has a length of $q_t + \mathcal{T}$ (Eq. 6), the smaller q_t the more important $\mathcal{T}$ becomes, and thus the more important it is to optimize it accordingly. Hence, this optimization is used as an integral part of the PIST technique in the remainder of the experiments.

5.3 Query performance

Next we compare the performance of PIST against the R-tree+B^+-tree approach and a linear scan of the data. All approaches make use of the assumption that $\mathcal{T}$ is known at query time.

Figure 5 shows query performance as a function of the size of the spatial component of the query, while Fig. 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 approach degrades rather quickly, unlike for the other approaches. 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 the GSTD and INFATI datasets the R-tree+B^+-tree is not competitive at all. This happens because for the GSTD dataset the density of the data in the occupied portion of the space is higher, causing the underlying R-tree to have more overlaps and, consequently require more tree traversals. The case for the INFATI dataset is even more extreme, as even a simple linear scan performs

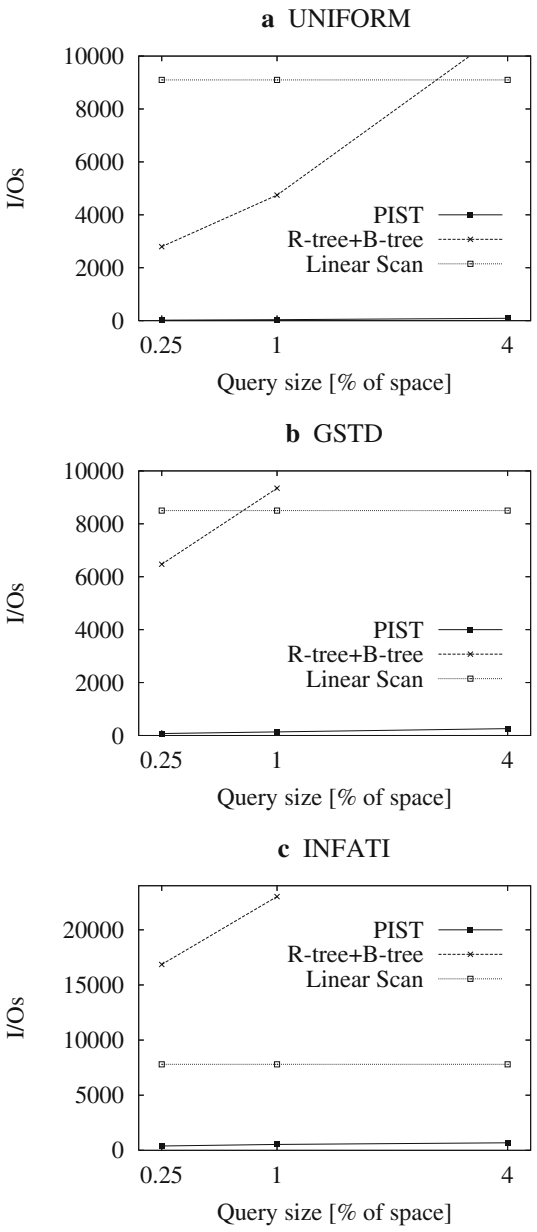


Fig. 5 Comparing I/O performance as a function of the size of the spatial component of the query

relatively much better. For all datasets and query sizes PIST deliver unquestionably much better performance.

PIST 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 confirms that the proposed grid partitioning is able to cope well with variations in this parameter.

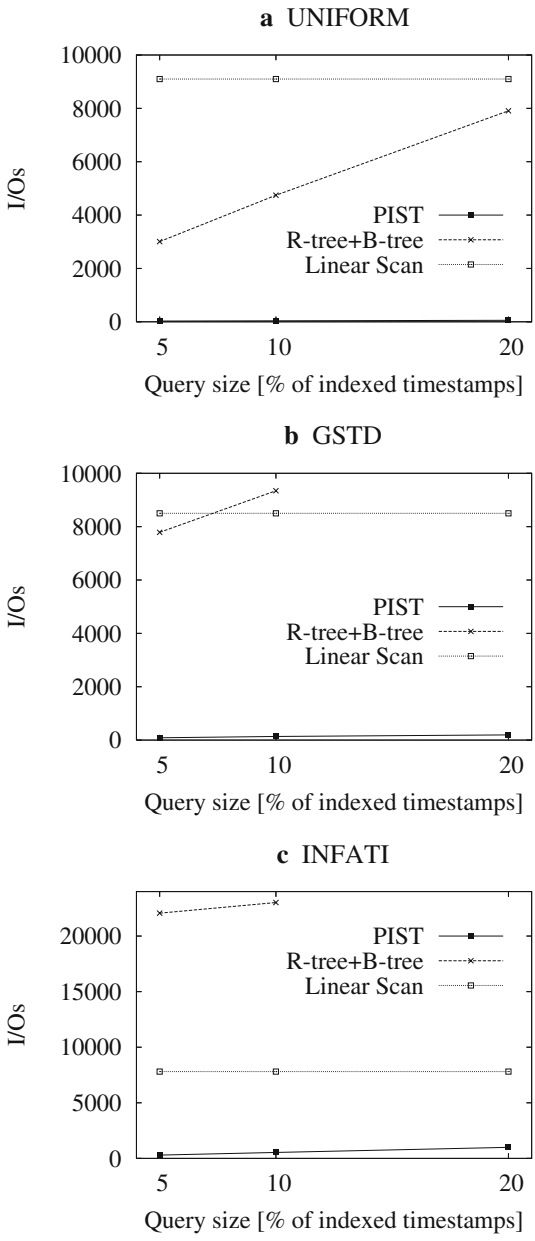


Fig. 6 Comparing I/O performance as a function of the length of the temporal component of the query

PIST is also very robust with respect to the increase in the dataset size as can be seen in Fig. 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.

In summary, PIST's performance is often two or more orders of magnitude faster than the other approaches, while being quite robust with respect to all parameters

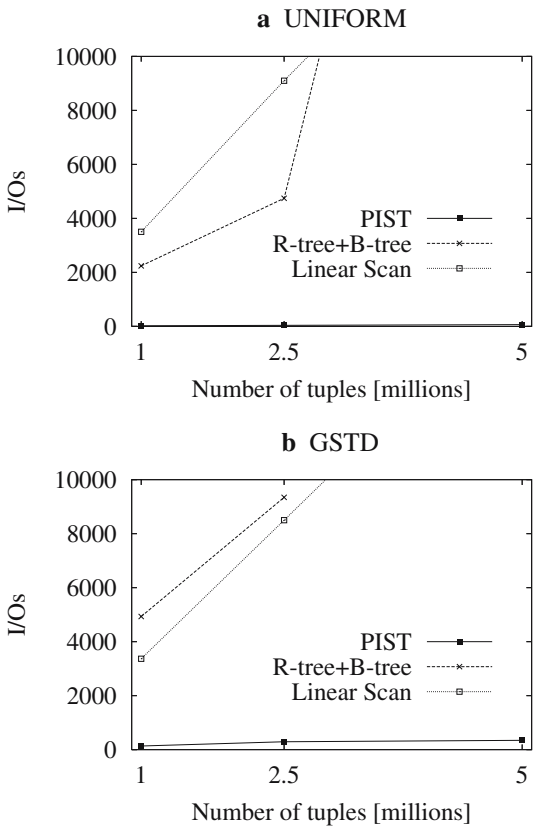


Fig. 7 Comparing I/O performance as a function of the dataset cardinality

investigated. This is due to very effective filtering of heavily populated partitions that do not contribute to the query's answer, leading to highly efficient query processing.

5.4 Robustness

As we discussed earlier the cost model depends on an assumed query size, both for the temporal and the spatial component. In the next set of experiments we show 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 with reasonable variances between the assumed and actual query sizes. In all forthcoming tables the values in the first row represent the query sizes assumed at index construction time, with S being the size percentage-wise with respect to data space, and T being the size of the query as the number of indexed timestamps. Following a similar notation the values on the first column are the sizes of the issued queries. Hence, in the ideal case, the minimum values should appear in the diagonal of the tables.

In Table 4a,b,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. Table 5a,b,c, on the other hand, show the performance when the temporal range varies and the spatial query remains fixed.

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

	S=0.25% $T = 10$	S=1% $T = 10$	S=4% $T = 10$
a UNIFORM dataset			
$S = 0.05$, $T = 0.1$	18.82	21.22	25.58
$S = 0.1$, $T = 0.1$	39.51	37.72	42.23
$S = 0.2$, $T = 0.1$	95.25	89.38	91.03
b GSTD dataset			
$S = 0.05$, $T = 0.1$	141.85	168.80	197.02
$S = 0.1$, $T = 0.1$	278.39	294.10	329.49
$S = 0.2$, $T = 0.1$	682.95	670.52	672.52
c INFATI dataset			
S=0.25%, $T = 10$	390.53	388.75	399.94
S=1%, $T = 10$	545.22	533.11	552.37
S=4%, $T = 10$	704.64	679.09	682.77

It is interesting to note that the smallest number does not always appear in the diagonal of the tables as in the ideal case. One reason for this is that the cost model often suggests a non-integer number of grid cells (Eq. 6), which is obviously not practical and has to be approximated to an integer. Another reason is the partioning procedure is not guaranteed to deliver optimal results in the case of non-uniform spatial distributions, which is the case of the GSTD and INFATI datasets. Nevertheless, even in such cases, the difference between the actual minimum and expected minimum is very small.

Overall, performance does not vary too much if one builds the dataset assuming a "wrong" (within reasonable limits) average query as can be seen throughout the tables. These results serve to show that PIST 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, PIST is still able to deliver good performance.

Table 5 I/O robustness of PIST for all three datasets with respect to temporal query size (spatial query size is fixed)

	S=1% $T = 5$	S=1% $T = 10$	S=1% $T = 20$
a UNIFORM dataset			
S=1%, $T = 5$	25.45	27.63	27.35
S=1%, $T = 10$	43.73	37.72	37.93
S=1%, $T = 20$	69.01	61.82	55.80
b GSTD dataset			
S=1%, $T = 5$	204.82	209.87	227.94
S=1%, $T = 10$	322.17	294.10	281.11
S=1%, $T = 20$	485.95	438.41	416.83
c INFATI dataset			
S=1%, $T = 5$	298.91	296.21	312.28
S=1%, $T = 10$	553.78	533.11	544.01
S=1%, $T = 20$	1,043.51	991.90	995.35

5.5 Comparing with the MV3R-tree

Even though our main aim in this paper is to contribute a practical technique rather than a novel data structure for indexing spatio-temporal data, we compare its performance to the MV3R-tree [17], which, despite not being feasible to be implemented on top of existing RDBMSs, is arguably a good representative of special purposed indices for spatio-temporal data. Our goal in the set of experiments discussed next is to show that PIST can indeed offer performance at least comparable to a leading and specialized index structure.

Figures 8 and 9 show the performance of PIST and the MV3R-tree when varying the size of the spatial and temporal component of the queries. While all other parameters remain at their default value as before, the GSTD dataset had to be downsized to 1 million observations as the source code we obtained for the MV3R-tree was somehow unable to cope with larger datasets.

As one can see for smaller queries both structures deliver nearly the same performance, while for non-uniform data and larger queries there is a slight advantage for PIST. Unfortunately, there seems to be an upper limit of indexing 30,000 distinct timestamps on the MV3R-tree which prevented us to index the INFATI data set.

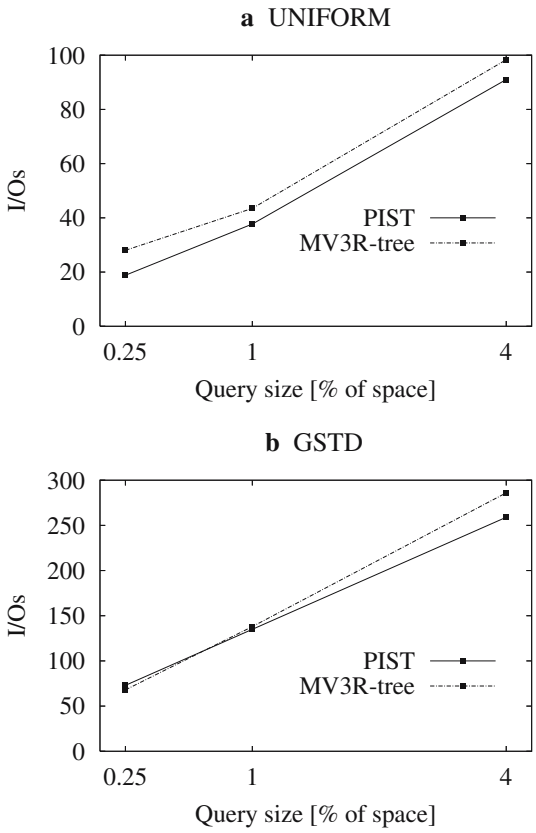


Fig. 8 Performance of PIST vs. MV3R-tree when varying size of the spatial component. **a** UNIFORM. **b** GSTD

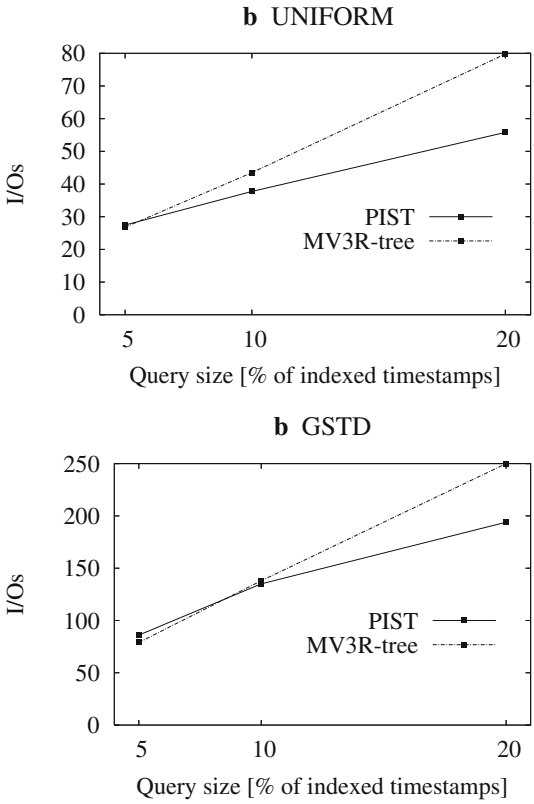


Fig. 9 Performance of PIST vs. MV3R-tree when varying the length of the temporal component of the query. **a** UNIFORM. **b** GSTD

In terms of scalability the inability of handling large datasets in the case of the MV3R-tree makes a fair comparison not possible. This is because for very small datasets, say in the order of up to a few hundred thousands of observations there is an inherent overhead within PIST due to the underlying DBMS which is not present within the MV3R-tree. One should note though that it is well known that for very small datasets a trivial linear scan is often the most efficient solution.

5.6 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 shown above.

The times reported for index creation were obtained on a PC with an AMD Athlon XP 3200+ running at 2.19 GHz and with 1.00 GB of RAM, and using the GSTD dataset with 2.5 million objects. The partitioning was determined using the default query sizes on the spatial and temporal domains. The results using other datasets follow the same trend.

There are two main tasks that need to be performed within PIST. First, the partitioning must be obtained using the heuristic algorithm presented in Section 3.1. After that, the objects need to be inserted into the correct partitions, i.e., the index-organized tables. The first parts took 76 sec. while the second required 843 sec. for a total of 919 sec. It should be 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 indices, for a total of 984 sec. It should be noted that both approaches made use of the SQL*Loader facility available in typical ORACLE installations.

Even though PIST is overall about 7% faster we could observe that the partitions lookup, i.e., finding in which partition an object should be inserted, poses most of the overhead at data insertion time. We did consider the idea of using an index, e.g., an R-tree, for the grid partitions themselves in order to speed up the partition lookup process. However, the number of partitions was fairly low (in the order of hundreds) for all experiments and it would not benefit from an index, as compared to a simple linear scan of the partitions table.

6 Conclusions and future work

PIST leverages existing RDBMS technology by providing support for "out-of-the-box" RDBMS-based management of historical spatio-temporal point data. PIST 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 in practice. In addition PIST offers the possibility of pre-processing the data, splitting the temporal ranges of some observations, in order to further improve performance. Using both real and synthetic datasets, PIST has been shown to outperform other alternatives for spatio-temporal data management by a large margin. We have also shown that PIST is robust with respect to the query size assumed at index construction time.

As for directions for future work the following questions are worth considering:

- How could PIST's partitioning be adjusted as the database size increases? Even though some preliminary experimental results suggest that PIST 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. Along the same line, it remains to be investigated how resilient a partitioning is with respect to changes in the original distribution. A variation of this theme would be investigating whether the RDBMS could re-configure partitions by itself, "on-the-fly", without having to rebuild the whole index.
- How to extend the proposed PIST approach in order to handle trajectories, obtained, for instance, by linearly 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. The research question to be investigated in this case is how to obtain a cost model to guide an optimal partitioning given a set of trajectories.

- Finally, how to make PIST capable of indexing not only historical but current spatio-temporal data? The topic of having indexing structures, the B^+-tree included, able to sustain very high update ratios, e.g., several millions of updates per second, is still an open problem. Nevertheless, perhaps an approach similar to the one used within SETI, where a memory-resident, "front index" is used to alleviate the problem, could be adapted for use within PIST.

References

1. M. Abdelguerfi et al. "The 2-3TR-tree, a trajectory-oriented index structure for fully evolving valid-time spatio-temporal datasets," in *Proc. of ACM GIS*, pp. 29–34, 2002.
2. M.J. Carey et al. "Shoring up persistent applications," in *Proc. of the ACM SIGMOD Conf.*, pp. 383–394, 1994.
3. V.P. Chakka et al. "Indexing large trajectory data sets with SETI," in *Online Proc. of CIDR*, 2003. http://www-db.cs.wisc.edu/cidr/cidr2003/program/p15.pdf
4. A. Guttman. "R-trees: a dynamic index structure for spatial searching," in *Proc. of the ACM SIGMOD Conf.*, pp. 47–57, 1984.
5. C.S. Jensen et al. "The INFATI data," Technical Report TR-79, TimeCenter, 2004. http://arxiv.org/abs/cs.DB/0410001.
6. C.S. Jensen, D. Lin, and B.-C. Ooi. "Query and update efficient B^+-Tree based indexing of moving objects," in *Proc. of VLDB*, pp. 768–779, 2004.
7. R.V. Kothuri and S. Ravada. "Spatio-temporal indexing in oracle: issues and challenges," *IEEE TCDE Bulletin*, Vol. 25(2):56–60, 2002.
8. H.-P. Kriegel, M. Pötke, and T. Seidl. "Managing intervals efficiently in object-relational databases," in *Proc. of VLDB*, pp. 407–418, 2000.
9. P.M. Lewis, A.B., and M. Kifer. Database and transaction processing. Addison-Wesley, 2002.
10. D. Mallett. "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.
11. M.F. Mokbel, T.M. Ghanem, and W.G. Aref. "Spatio-temporal access methods," *IEEE TCDE Bulletin*, Vol. 26(2):40–49, 2003.
12. M.A. Nascimento and J.R.O. Silva. "Towards historical R-trees," in *Proc. ACM SAC*, pp. 235–240, 1998.
13. D. Pfoser, C.S. Jensen, and Y. Theodoridis. "Novel approaches in query processing for moving object trajectories," in *Proc. of VLDB*, pp. 395–406, 2000.
14. S.M. Ross. Introductory statistics. McGraw-Hill, 1996.
15. S. Saltenis et al. "Indexing the positions of continuously moving objects," in *Proc. of the ACM SIGMOD Conf.*, pp. 331–342, 2000.
16. H. Samet. "The quadtree and related hierarchical data structures," *ACM Comput. Surveys*, Vol. 16(2):187–260, 1984.
17. Y. Tao and D. Papadias. "MV3R-Tree: a spatio-temporal access method for timestamp and interval queries," in *Proc. of VLDB*, pp. 431–440, 2001.
18. Y. Tao, D. Papadias, and J. Sun. "The TPR*-Tree: an optimized spatio-temporal access method for predictive queries," in *Proc. of VLDB*, pp. 790–801, 2003.
19. Y. Theodoridis and T. Sellis. "A model for the prediction of R-tree rerformance," in *Proc. of PODS*, pp. 161–171, 1996.
20. Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. "On the generation of spatiotemporal datasets," in *Proc. of SSD*, pp. 147–164, 1999.
21. Y. Theodoridis, M. Vazirgiannis, and T.K. Sellis. "Spatio-temporal indexing for large multimedia applications," in *Proc. of IEEE ICMCS*, pp. 441–448, 1996.

Viorica Botea obtained her B.Sc. degree in Computer Science from the University of Bucharest, Romania in 1998 and a M.Sc. degree in Computing Science from the University of Alberta, Canada in 2006. She is currently a research programmer with the Logic and Computation Program of National ICT Australia (NICTA).

Daniel Mallett obtained his BSc (2002) and MSc (2004) in Computing Science from the University of Alberta. He is currently working in industry as a Database Administrator for Divestco Inc, an oil and gas software, data, and services company based in Alberta. His main area of research interest is spatial and temporal databases.

Mario A. Nascimento obtained his Ph.D. degree in Computer Science at Southern Methodist Univ.'s School of Engineering in 1996. Between 1989 and 1999 he was a researcher with the Brazilian Agency for Agricultural Research (Information Technology Center) and, between 1997 and 1999, he was also associated with the Institute of Computing of the State Univ. of Campinas (Brazil). Since then Mario has been with the Department of Computing Science of the Univ. of Alberta, where he currently is an Associate Professor. In addition he has also been an IITA Invited Professor at Chung-Ang Univ. in Korea and Visiting Professor at the National Univ. of Singapore (Associate) and Aalborg Univ., Denmark. His main research interests lie in the areas of Indexing/Access Structures for (Spatio/Temporal/Image) Databases and Sensor Networks. He is a member of ACM, SIGMOD and IEEE Computer Society. (Current information can be found at http://www.cs.ualberta.ca/~mn.)

Jörg Sander is currently an Associate Professor at the University of Alberta, Canada. He received his MS in Computer Science in 1996 and his PhD in Computer Science in 1998, both from the University of Munich, Germany, under the supervision of Prof. Hans-Peter Kriegel. He worked one year as a post-doctoral fellow at the University of British Columbia, Canada, under the supervision of Prof. Raymond Ng. His research interests include spatial and spatio-temporal databases, as well as Knowledge Discovery in Databases, especially clustering and data mining in spatial and biological data sets. (Current information can be found at http://www.cs.ualberta.ca/~joerg.)