University of Alberta

Relational Database Support for Spatio-Temporal Data

by

Daniel James Mallett

Technical Report TR 04-21
September 2004

DEPARTMENT OF COMPUTING SCIENCE
University of Alberta
Edmonton, Alberta, Canada

Relational Database Support for Spatio-Temporal Data

Daniel James Mallett

September 26, 2004

Abstract

As computing technology evolves, the need to manage very large collections of complex data objects continues to arise. Database systems have been continually adapting to support new data-types, for instance, support for spatial data management is now standard in most Relational Database Management Systems (RDBMSs). With the proliferation of mobile devices capable of accurately reporting their position in time and space, there exists an urgent need to efficiently manage very large databases of moving object data inside a RDBMS. Currently, RDBMS support for such spatio-temporal data is limited and inadequate for non-trivial datasets, and most existing spatio-temporal access methods cannot be readily integrated into a RDBMS. This technical report proposes an approach for spatio-temporal storage, indexing and query support that can be fully integrated within any off-the-shelf RDBMS. A cost model which allows one to fine tune the proposed approach in order to minimize the number of disk accesses at query time is also presented. An implementation of the approach in Oracle 9i is compared with other alternatives to map the problem into a RDBMS. Extensive experiments show that the cost model is reliable and that the proposed approach significantly outperforms other options for managing spatio-temporal data inside a RDBMS.

1 Introduction

For decades, Relational Database Management Systems (RDBMSs) have provided users with advanced data management capabilities. A RDBMS offers a host of convenient and useful features to their users, e.g., a central repository for data storage, concurrency control, backup and recovery mechanisms, support for multiple users and transactions, and tools to manage very large datasets. The basic building blocks of a database remains relational tables whose columns consist of simple data-types such as integers, floating point numbers, and text strings. RDBMS developers have integrated within their systems index support for these basic data-types, however the data-types and index support is oftentimes insufficient for more complex application domains. Inevitably, as the need to manage more complex data-types emerges, indexing support for these advanced data types is integrated into the RDBMS. For example, most off-the-shelf RDBMS systems have storage, indexing, and retrieval support for geometric or spatial data used in Geographic Information System (GIS) applications [3, 27, 35]. A specialized spatial object type and multi-dimensional index structure can dramatically enhance retrieval performance for large databases versus the alternative of having to map a spatial object into the simple data-types and index structures that would normally be available. Similarly, specialized index structures integrated within the RDBMS for temporal data [19, 30] have been proposed.

Given emerging application domains involving moving object data, there now exists an urgent need to support spatio-temporal data inside a RDBMS. Anything that changes position through time can be classified as spatio-temporal data. A prolific number of GPS, wireless computing, and mobile phone devices are capable of accurately reporting their position [16], and ubiquitous applications that can take advantage of this information are in high demand, e.g., tracking and fleet management [20], traffic and re-routing applications [9], spatio-temporal data-mining [40, 29], and location-aware services [16]. There are an estimated 500 million users of mobile-phones worldwide [37], many of which can be tracked. US law [10] has mandated that all wireless phones in that country be traceable in order to provide enhanced location information to 911 dispatchers during emergencies. The overwhelming task of managing large datasets of such data demands the convenience, reliability, and data storage capabilities that a traditional RDBMS affords.

Although ample research on Spatio-Temporal Access Methods (STAMs) has been performed (an overview can be found in [25]), very little work exists on how to provide a STAM inside a RDBMS, [17] being a notable exception. This work fills this crucial need by proposing a spatio-temporal access method which can be fully integrated within any RDBMS.

1.1 Problem Statement

There are two main types of spatio-temporal databases [25], those that manage historical information and those that manage current information for current/predictive query purposes. This report 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 temporal interval during which an object remained at position $\langle x, y \rangle$.

A typical domain where such a model fits is mobile device tracking [50], e.g., of GPS users, wireless computing devices, or cell phones. Unlike the trajectory model [39], where the movement of objects between data points is interpolated, our data model does not assume anything about the movement of objects between records. The model reflects real-world applications 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. The temporal interval implicitly performs a form of data compression, when an object's location does not change (either because the object does not move or does not move beyond the accuracy of the positioning device) only the temporal interval will grow and a single tuple will be added to the database.

A spatio-temporal 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 (1) $\langle x, y \rangle$ is inside $\mathcal{R}$ and (2) $\langle t_s, t_e \rangle$ 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*".

The problem we investigate in this technical report is to define a practical storage and indexing model fully integrated inside a RDBMS that can store a very large database of the given records, handle regular inserts into the database and efficiently answer spatio-temporal range queries.

1.2 Technical Report Scope and Organization

This technical report provides:

- An efficient spatio-temporal indexing technique fully integrated within a RDBMS,
- A cost model that fine tunes the proposed technique for optimal performance, and
- An experimental study demonstrating the reliability of the cost model and both the efficiency and effectiveness of our proposed method.

This report is structured as follows: Section 2 reviews related work in the spatio-temporal domain, including a discussion of spatio-temporal data generation and modeling, the types of spatio-temporal queries that exist, and an overview of several spatio-temporal access methods that have been proposed. Section 3 provides background on what options for RDBMS support of spatio-temporal data exist. Therein we identify three general alternatives for providing RDBMS support for spatio-temporal data:

1. Recasting the problem to employ existing RDBMS support for spatial data,
2. Loosely coupling a STAM to the RDBMS, and
3. Tightly coupling a STAM inside the RDBMS via a relational mapping.

Our approach falls under the third alternative, i.e., we design a STAM that leverages existing RDBMS functionality. The chief advantage thereof being that our method can be readily integrated into any relational database management system. In Section 3, we also provide important background about the other methods that we use for experimental comparison to our approach, specifically a spatial index based approach and a space-filling curve approach. Section 4 details our proposed approach, and the associated cost model. We call our method the Space-Partitioning with Indexes on Time (SPIT) approach. SPIT is a grid-based access method with a relational mapping. The SPIT cost model provides a means to minimize the number of disk accesses at query time with respect to grid size. In Section 5 we describe how our approach can be implemented using a RDBMS, Oracle 9i in particular. Here we also describe relevant implementation details of the approaches we will use for experimental comparison to SPIT, namely, the spatial index based approach and the space-filling curve approach. In Section 6 we confirm the reliability of the cost model and compare our approach to the other methods for indexing spatio-temporal data inside a RDBMS. We show that SPIT is extremely efficient – outperforming all approaches for the dataset and queries we investigated. Section 7 concludes the report and provides ideas for extending the work to handle additional spatio-temporal data models and queries, and to further enhance query performance.

2 Related Work

A RDBMS organizes data into tabular files (tables) that can be related to each other by common fields. The columns of each table represent the data fields and have a predetermined data-type, i.e., integer, floating point number, or string. Table rows, often called tuples, store the actual data. In order to enhance query performance a RDBMS provides means to index table columns. An index speeds query processing by providing a separate data structure with pointers to the rows, identified by unique row identifiers, of entries in the table that satisfy query criteria.

The basic index structures provided inside a RDBMS are hash-based and tree-based indexes. A hash-based index uses a function to map records to pages on disk. To answer queries on a specific value efficiently, the hash function on that value is applied which returns the page where any records with that value are located. Assuming an effective hash function, it is often the case that only one disk access is necessary to retrieve a matching tuple at query time. Typically, the B+tree [4] is the tree-based index structure of choice inside the RDBMS. Given a column of table data already in sorted order, a B+tree is constructed by adding each entry along with a pointer (the unique row identifier of that tuple) into nodes at the leaf-level of the index. Each node corresponds to a page on disk. As shown in Figure 1, parent nodes are constructed so as to contain values which bound the minimum and maximum range of values that their children nodes contain. The process is repeated recursively to construct the upper levels of the tree. For every node (except the root) a minimum m and maximum M number of entries per node is specified so as to ensure that the tree has a balance structured. At query time, the root node and internal level nodes direct the search. For a point query, i.e., a query against a single value, a single path down the tree is followed. For range queries as well, only a single path down the tree must be followed – once the minimum value of the range is found, an index range scan can proceed by following pointers between the leaf-level nodes. For 1-dimensional data, i.e., a column of characters in a database table, the B+tree index offers extremely good performance and scalability. Even given millions of tuples (database rows), the B+tree structure typically requires only three levels. Assuming that root and intermediate level nodes are cached, only 1 disk access (I/O) is necessary to answer a point query because leaf-level nodes contain the actual column value stored in its associated table tuple.

Although the basic data modeling and index support provided in an off-the-shelf RDBMS is sufficient for many real-world applications, it has long been acknowledged that more complex data types require extensions to basic RDBMS support. Via object-relational extensions to the database, a column can be made to store a complex object type. The challenge is to provide built-in index support for these complex object types.

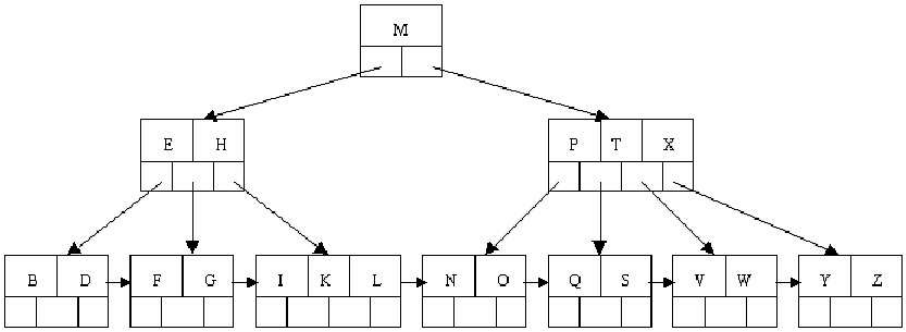


Figure 1: B+tree

Support for spatial data types and indexes is provided in most RDBMS. The R-tree [14] is the classic structure that most RDBMS vendors base their spatial index support on. A R-tree builds a Minimum Bounding Rectangle (MBR) approximation of every spatial object in the database and inserts each MBR in the leaf level nodes of the R-tree. Figure 2 illustrates a three-dimensional R-tree, boxes A–F represent the MBRs of the actual 3-dimensional spatial objects on disk. The parents nodes, R1 and R2, represent the MBRs of a group of object MBRs. At insertion time, a cost-based algorithm is used to decide which node a new object should be inserted in based on the criterion of limiting the amount of overlap between nodes and the amount of dead-space in the tree. For example, in Figure 2, grouping objects A, E, and F together into R1 creates a smaller MBR than if A, E, and C were grouped together instead. As with the B+tree, a minimum and maximum number of entries per node is enforced so as to ensure that the tree remains balanced. At query time, the tree is traversed, beginning at the root, by visiting each node wherein the query window intersects a MBR. At the leaf-level, only those object MBRs that intersect the query MBR need to be retrieved from disk. Recall that with a B+tree, only a single path through the tree need be traversed. With the R-tree, however, it may be necessary to follow several paths along the tree, because the query window may intersect several MBRs in each node.

Another type of spatial index structure supported in many RDBMSs is the Quad-tree [42]. The Quad-tree is a partition-based index in that objects are recursively divided into quadrants. A spatial object is represented by the quadrant in which it belongs. Figure 3 illustrates a Quad-tree index structure. The root node represents the entire data-space, the second level represents the division of the data-space into quadrants, and subsequent levels divide each of the quadrants into smaller sub-quadrants. Leaf nodes point to the actual spatial objects within

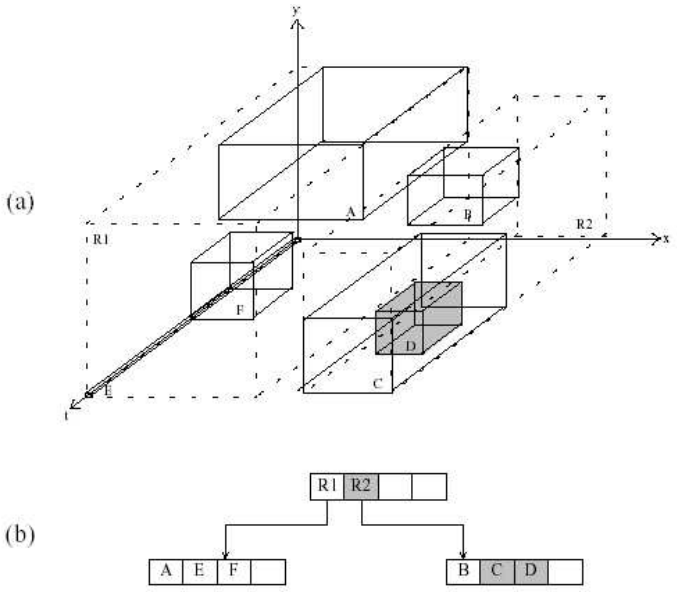


Figure 2: Three-dimensional MBRs stored in a R-tree structure (from [45])

that quadrant. At query time, the tree is navigated by following those nodes where the query window intersects a quadrant until the leaf level pointers are reached. For general usage in spatial applications, it is reported that the R-tree tends to be more efficient than the Quad-tree [18]. This behavior is at least partially due to the R-tree's ability to adapt to the data it represents, i.e., the index is based on the actual spatial objects in the database, not on a division of space. For certain applications, especially in the domain of image-based applications [49], the Quad-tree offers better performance because data exists throughout the space.

As with spatial data, there exists a need to incorporate support for temporal data-types and index structures within a RDBMS. A standard database only supports a view of the data at a single point in time, i.e., can only answer queries of the form "what is employee X's salary?". In many domains, however, it is important to store historical information or to have information become valid at a future point in time. For example, queries such as "what was employee X's salary last year?". It is of interest to support indexing in such domains in order to efficiently answer temporal queries. One approach is the Relational-Interval Tree (RI-Tree) [19], which integrates temporal index into the RDBMS by modifying the R-tree so as to support efficient indexing of 1-dimensional temporal intervals. The approach can be inte-

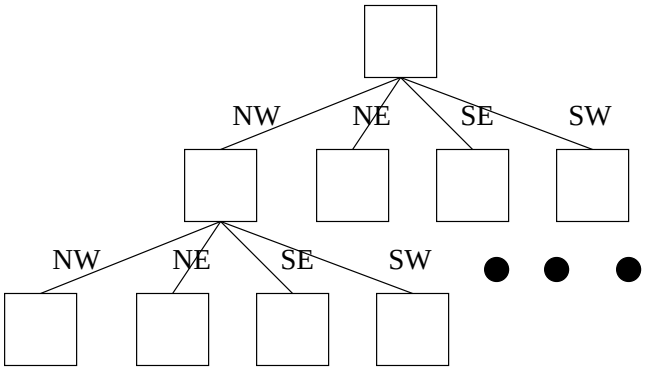


Figure 3: Quad-tree

grated into an Oracle RDBMS using database extender technology, which provides a mapping between the API of the index and query structure, and the underlying database tables and index structures. A related technique for indexing temporal data inside a RDBMS is the MAP21 approach [30]. With MAP21, a maximum temporal interval assumption is used so as to map the start and end points of a temporal interval into a one dimensional value that can be readily indexed using a B+tree.

The need to provide RDBMS support for spatio-temporal data is a natural extension of the stance taken for purely spatial or purely temporal database support. In the most general sense, spatio-temporal databases deal with geometries changing over time [11]. The geometry represents a moving object (whose geometry may or may not change over time), and the object is assumed to move continuously. If only the position of an object in space is of interest, for example when tracking vehicles, then the data is essentially that of moving points. If the object geometries (spatial extents) are of importance, for example when dealing with the movement of weather patterns, then the data consists of a moving region that can shrink or grow over time. In the rest of this section we discuss issues related to the acquisition and generation of spatio-temporal data, the modeling of spatio-temporal data, spatio-temporal query types, and existing methods, in particular STAMs, to support spatio-temporal data and queries.

2.1 Spatio-Temporal Data

Spatio-temporal data comes from various sources, for example the movements of people, animals, vehicles, airplanes, boats, extra-terrestrial objects, military ob-

jects, weather (areas of high/low pressure, and storms), the changing borders of countries or cities [11] – the list is endless because anything that changes position through time can be classified as spatio-temporal data. An important classification of moving objects is to define any constraints on movement that may exist. In general, three movement scenarios exist – "unconstrained movement (vessels at sea), constrained movement (cars, pedestrians), or movement in networks (trains and, in some cases, cars)" [37].

Important uses for spatio-temporal information include tracking inventory, personal, shipments, fleet, spatio-temporal data-mining, e.g., finding patterns in large datasets of changing data or grouping users based on similar behavior patterns. Of enormous interest in cities with traffic congestion problems is traffic and re-routing applications, e.g., directing users where to go, helping a user when they are lost, or suggesting alternative routes when a road is congested or closed [9].

Various technologies can be used to actually collect the data. If a device includes a GPS (Global Positioning System) module, the user's location can be defined very accurately (within 2-20 meters) [16]. GPS-based devices do not work well indoors, and require that the user device sends location updates to the service provider on a regular basis. Besides advanced positioning and mapping applications, GPS modules can be used to provide location-aware services, fleet monitoring support, and for historical tracking purposes.

The location of a cell phone in the mobile network can be extrapolated by telecommunication operators without the need for a GPS module. Based on the cell in which the phone is in or by measuring the distance between overlapping cells, it is possible to pinpoint the location of a cell phone within a radius of 50 meters (in urban areas) and a few kilometers in rural settings [15]. The advantage of such an approach is that existing mobile phone technology can be used for the purposes of providing location-aware services or tracking, without the need for extra technological infrastructure. US law [10] has mandated that all wireless phones in that country be traceable in order to provide enhanced location information to 911 dispatchers during emergencies, and wireless carriers are beginning to take advantage of this capability to offer location-based services to users. Location-based services can be defined as "services that are related as such or by their information contents to certain places or locations" [16].

Wireless computing devices can also generate spatio-temporal data [16]. Depending on the density of the network, the accuracy of such an approach can be extremely high (within 2 meters). Typically, the required wireless network infrastructure means that data can only be generated within restricted areas such as an office building or a university campus.

Not surprisingly, due to the private and proprietary nature of data generated using any of the above technologies, it can be difficult to obtain real datasets of

spatio-temporal data for experimental purposes. Some animal tracking, hurricane, and public bus datasets are publicly available [31]; however, the number of records (in the hundreds) is insuffi cient for large scale database benchmarking. Due to the lack of readily available real datasets and the need to generate data in a controlled fashion for experimental purposes, synthetic spatio-temporal data generators have been developed. The most notable synthetic generators include the Generate SpatioTemporal Data (GSTD) Tool [46], the Network-based Generator of Moving Objects [7], a Generator for Time-Evolving Regional Data (G-TERD) [48], and the City Simulator by IBM.

The GSTD [1] [46] generates data according to prescribed statistical distributions. Currently uniform, Gaussian, and Skewed distributions are supported. Both point and moving region (rectangle) spatio-temporal data can be generated. The cardinality of the dataset can be easily adjusted in order to perform large scale experiments. For point data, GSTD requires an initial, time, and center distribution. The initial distribution defi nes where objects begin within the unit space that GSTD assumes. The time distribution controls the timestamps when object locations are updated. The center distribution controls the movement of points in space. Advanced features supported by the GSTD include support for a framework, i.e., obstacles that objects cannot enter, support for multiple datasets with different properties, and an on-line dataset visualizer. GSTD also supports three approaches for handling points that leave the data-space – a radar approach where objects exiting the space are allowed, an adjustment approach where objects are forced to remain in the space and a toroid approaches where objects "wrap-around" so as to remain in the space. Figure 4 shows three snapshots taken from the GSTD visualizer tool for a generated dataset consisting of points moving with a toroid approach, i.e., wrap-around. A point represents an object and boxes represent the set of obstacles, a framework, that restrict an object's movement.

Even with the use of a framework, the objects generated by the GSTD can move anywhere and along any path within the unit space where obstacles do not exist. On the other hand, the Network-based Generator of Moving Objects [7] can generate a dataset where objects follow a given network, e.g., roads. In such a scenario the maximum allowable speed and capacity of a network is important, as well as the interaction of moving objects. Objects are assumed to have a starting location and a destination. A key advantage of the Network-based generator is that synthetic data that follows the real topology of a network, e.g., the roads of a city, can be generated. Objects can also belong to a class, e.g., cars vs pedestrians, with different properties such as maximum velocity. A representation of the data generated by the Network-based Generator is shown in Figure 5. The dots repre-

[1] http://db.cs.ualberta.ca:8080/gstd/index.html

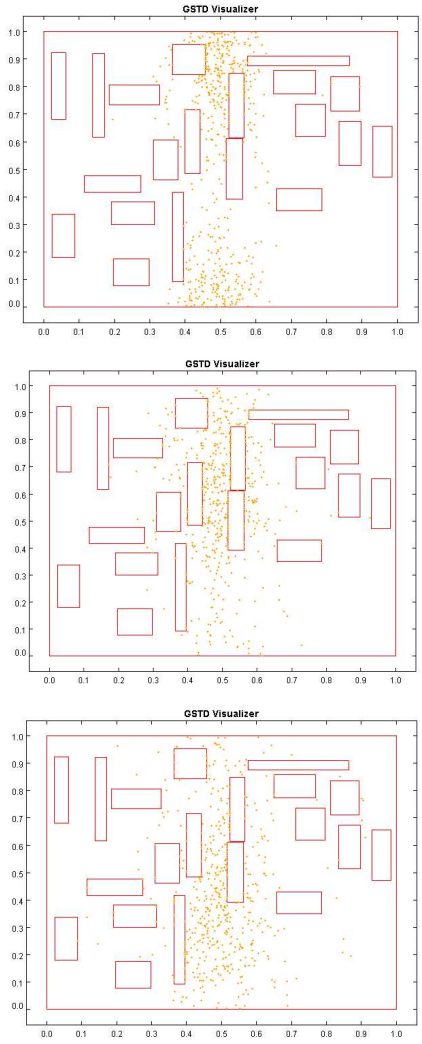


Figure 4: Snapshots of a framework-based GSTD Dataset

sent different types of moving objects and the lines the infrastructure that objects navigate.

G-TERD [48] generates a sequence of raster images rather than points as with the GSTD. With G-TERD, objects have associated colors that can change so as to represent additional properties (for example, temperature) of an object beyond the spatio-temporal. Data generation requires a large number of parameters. G-

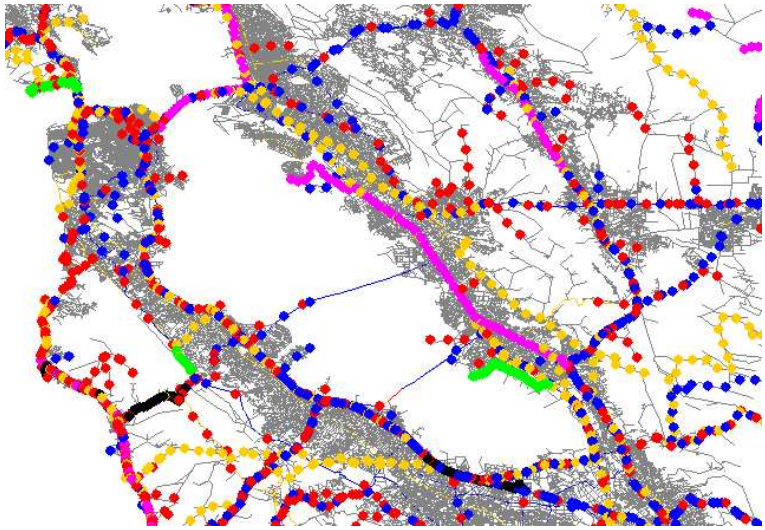


Figure 5: Snapshot of a Network-based Generator Dataset

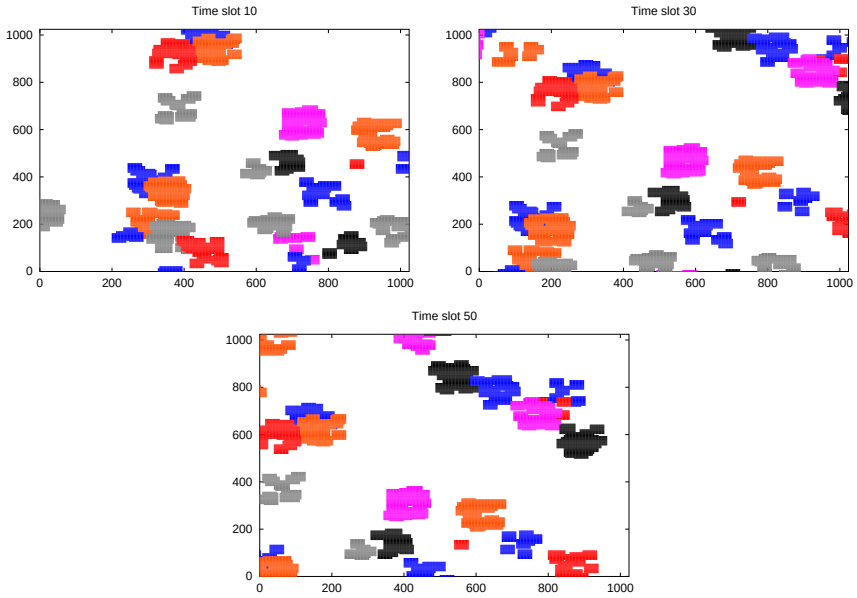


Figure 6: Snapshots of a G-TERD Dataset

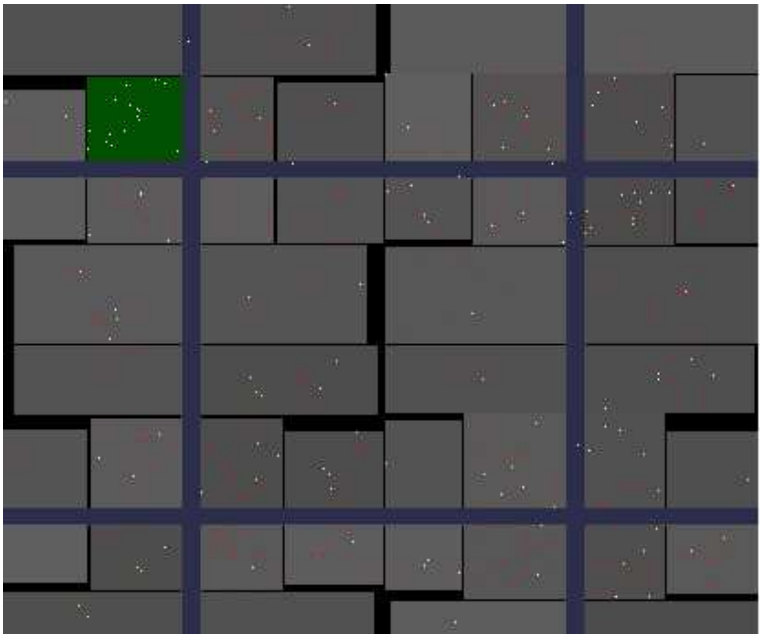


Figure 7: Snapshot of an IBM City Simulator Dataset

TERD also supports frameworks, or obstacles to movement. Three snapshots of a G-TERD dataset[2] are shown in Figure 6. The moving objects are the regions themselves.

IBM's City Simulator[3] is a Java based spatio-temporal data generator that creates spatio-temporal data representing the movement of people in a fully 3-dimensional virtual city. One can generate a dataset of up to 1 million people moving through the infrastructure of a city, i.e., buildings, parks, roads, and intersections. Most generators only allow for up to 2-dimensional movement in the spatial plane; the city simulator is unique in that it allows movement between the floors of buildings – elevation is taken into account. A snapshot of a city simulator dataset is shown in Figure 7. The points represent people in the city. The darker gray regions represent the road structure of the map, with lighter gray sections corresponding to buildings in the city. In the top-left corner one sees a park where several people have congregated.

[2] from http://delab.csd.auth.gr/stdbs/g-terd.html

[3] http://alphaworks.ibm.com/tech/citysimulator

2.2 Modeling Spatio-Temporal Data

As identified in [2], both real and synthetic spatio-temporal data can be modeled in various ways depending on the semantics of the application and the types of queries that must be supported. The data can be modeled as 2 or 3 dimensional spatial objects with time as another dimension. For many applications, the spatial extents of objects are not of interest, therefore the data consists of points in a 3 or 4 dimensional space, for example as shown in Figure 8. Although the movement of objects through time is continuous, the data typically consists of discrete sample points of the object position through time. For example, a taxi cab might report its position to a central server every 10 minutes.

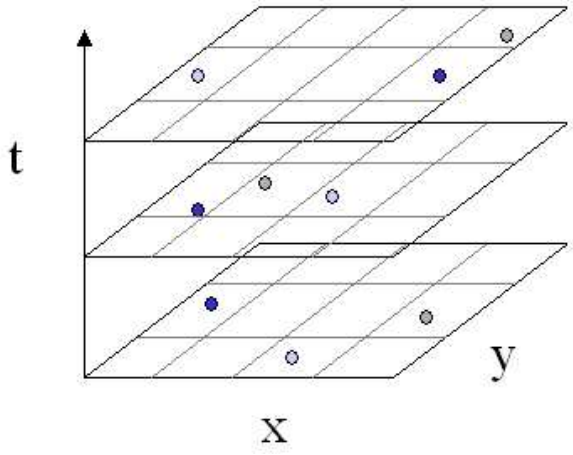


Figure 8: Modeling Spatio-Temporal Data as Points

An abstraction that can be used on these points is that of the trajectory model. In such a case a linear trajectory is assumed between data points. As Figure 9 exemplifies, this model assumes that an object traveled in a straight line at a constant velocity between actual recorded data points. The actual data to be indexed consists of a set of polylines – one for each moving object. The trajectory model can be an especially useful abstraction in domains where the position of objects is updated relatively infrequently and queries may fall between time update intervals.

A third alternative for data modeling, used more often when the current and future position of moving objects is of interest, is a parametric model [43]. Instead of recording actual data points, the spatio-temporal database stores a parameterized velocity function for each moving object in the system as per Figure 10. Although

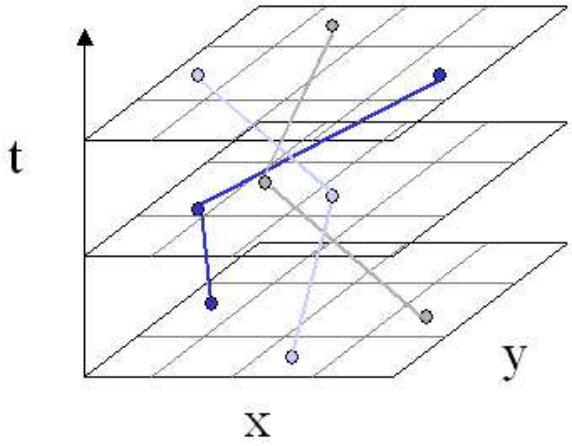


Figure 9: Modeling Spatio-Temporal Data as Trajectories

this approach means reduced storage overhead, the accuracy of the information is uncertain because the model assumes that an object follows a projected path. When the parameters of the stored object function changes beyond a set threshold, the function parameters (and any associated database indexes) must be updated. This can occur via polling (a pull-model) or a push-model where the object itself is responsible to inform the database of its changes. Under a parametric approach, the history of object movement is generally discarded – only the current velocity vector is of interest, however a separate mechanism could be used to store historical information using this model.

Another alternative data model approach is to take advantage of an underlying network [9] or constraints in the space if they exist [37]. For instance, data can be modeled as points along the 1-dimensional line of a road, or as a relative position along a path in a constrained data space.

The model we use is slightly different than the above in that we assume point geometries with a temporal interval, e.g., objects remain in a given position for an interval of time as shown in Figure 11. A step-wise linear interpolation between data points is assumed. The model is more realistic for spatio-temporal data tracking, i.e., for surveillance purposes. In such a domain one cannot assume anything about the movement of object's between data points.

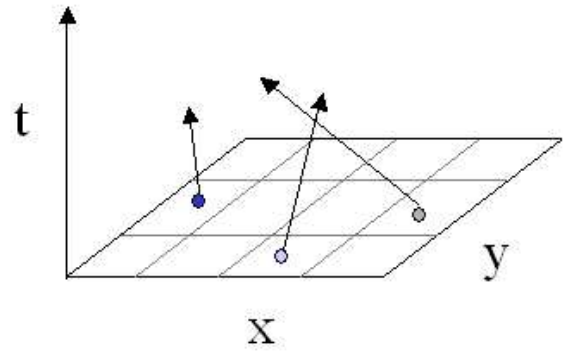


Figure 10: Modeling Spatio-Temporal Data as Parametric Functions

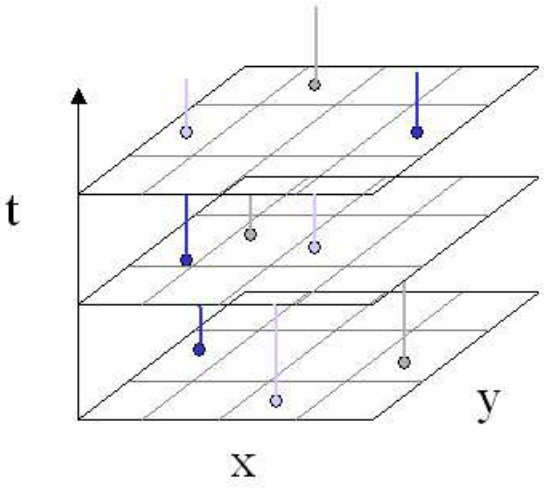


Figure 11: Modeling Spatio-Temporal Data as Step-Wise Interpolations

2.3 Spatio-Temporal Queries

The types of spatio-temporal queries depend on the spatio-temporal database of interest – historical or current. Current/predictive queries look to answer queries regarding the current or projected path of objects, for example, “which taxi cabs will be near the hockey arena at 7 p.m.?”. Historical queries focus on reporting information about the past movement of objects. Answering both types of queries effi ciently is very important for spatio-temporal based applications.

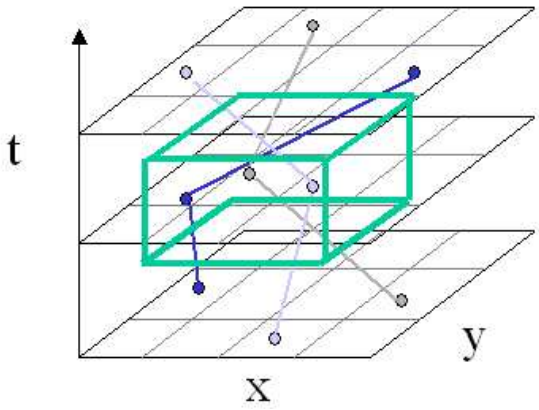


Figure 12: Spatio-Temporal Range Query

Current/Predictive STAMs support queries that predict a moving object's location at a given time based on the current velocity of the object. Much of the challenge of predictive queries is that the answer to a query is tentative in that it is based on what is currently known about objects in the real world – the motion vectors of objects can change at any time and thus (possibly) change the query answer set. Continuous queries offer an alternative solution to this problem, in the sense that they provide a query answer that is continuously updated as the state of the database changes, i.e., updates about object velocities are received.

Historical STAMs support queries that can be classified as coordinate-based or trajectory-based [39]. Coordinate-based queries can come in many forms, including range, nearest neighbor, k-nearest neighbor, or reverse nearest neighbor [9]. Range queries, which is exactly the case we are interested in, focus on retrieving objects within a prescribed area at a given time period or time slice. The cuboid in Figure 12 represents an example range query. Given the coordinates of a moving object, a nearest neighbor query finds the the object within the closest proximity at a given time. A k-nearest neighbor (where k is a user-specified constant) finds the k objects closest to the coordinates of the query object's coordinates. Reverse nearest neighbor queries return the objects that have a query object as their closest object [6]. Trajectory-based queries focus on topological or navigational information [36]. A topological query asks whether trajectories enter, leave, cross, stay within, or bypass a given spatio-temporal range [39]. A navigational query considers derived information such as speed (e.g. top or average), heading, traveled distance, covered area, etc. Combined queries (using both coordinated-based and trajectory-based information), can also be considered.

2.4 Spatio-Temporal Access Methods

Here we provide an overview of work on STAMs for current/predictive and historical spatio-temporal support. Another comprehensive overview can be found in [25].

Current/Future Position STAMs

The class of STAMs that answer queries regarding the current position of objects and predicted future position include many R-tree and Quad-tree based variations.

The 2+3 Trajectory R-tree (2+3 TR-tree) [28] actually indexes both current and past information. Two separate R-trees are used, one for the current two-dimensional points of objects and one for the historical three-dimensional trajectories (two spatial dimensions and one temporal dimension). At query time, both trees may need to be scanned depending on the query time interval. The 2-3 Trajectory R-tree (2-3 TR-tree) [1], a close cousin of the previous approach, also uses two separate R-tree indexes for current and past information. The 3-dimensional R-tree, however, is not trajectory-based.

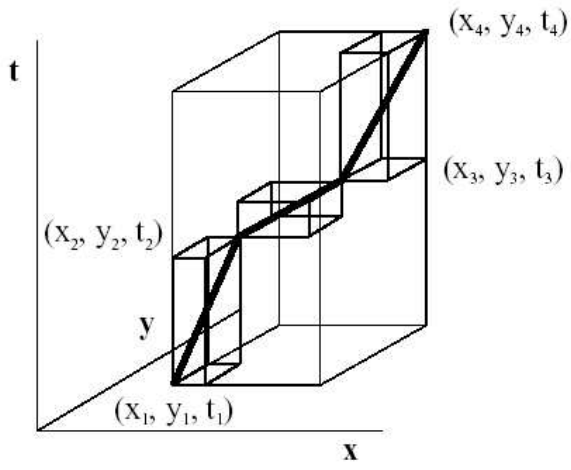


Figure 13: The Problem with Indexing Lines With Boxes

One of the main problems with using R-tree based approaches for indexing spatio-temporal data is the degradation of index performance due to overlap and dead-space among moving objects, especially when using a trajectory model. As shown in Figure 13, using Minimum Bounding Rectangles (MBRs) to approximate a trajectory leads to much wasted dead-space inside the approximation that will be

indexed by the R-tree, especially when one considers the MBR of the complete trajectory polyline. The inaccurate approximation means that the query window will intersect several MBRs in the R-tree, many of which being false positives in the sense that the query window intersects the MBR but does not intersect the actual object trajectory. Filtering out false positives will reduce query performance.

In [51], the authors offer a Quad-tree [42] based approach to help deal with this problem. A combined Quad-tree and R-tree, the Q+RTree, is suggested. The basis of the Q+RTree is to distinguish between fast moving objects and "quasi-static objects", defined as objects that move slowly (if at all). The authors argue that fast-moving objects are what lead to performance degradation in the R-tree. Quasi-static objects are indexed using an R*-tree (a modified version of the R-tree, c.f. [5]) , and fast-moving objects are indexed in a Quad-tree. At query time both indexes may need to be scanned; however the performance of the R-tree over the quasi-static objects will (hopefully) be good, and the fast-moving objects (of which the authors assume only a small percentage are) will benefit from the Quad-tree's use of a spatial partitioning.

The main STAM of interest for predictive queries is the TPR-tree [41], a parametric spatial access method based heavily on the R*-tree that uses bounding boxes extended by the velocity vector of moving objects. The velocity of an object is modeled as a parametric function that is updated whenever objects deviate from their stored model. The TPR*-Tree, a version of the TPR-tree with improved construction algorithms based on a performance model, has recently been proposed [43].

In [9], the authors present a grid-based approach for current/future position support. In this model, space is only 1-dimensional because objects are assumed to follow a road network. Of particular interest with this approach is that moving objects impact the movement of other moving objects. For example, when too many objects are located within the same grid cell, i.e., due to traffic congestion, the speed of objects decreases via a "ripple effect".

Historical STAMs

For historical STAMs (our focus), data is typically modeled in terms of trajectories as per Figure 9. Assuming a two-dimensional spatial region, time is introduced as a third dimension and the movement of objects between actual records is modeled in terms of line segments. Many historical STAMs have been proposed [1, 8, 32, 39, 47] the majority of which are based on the R-tree [14], [8] being a notable exception.

The 3-D R-Tree [47] is perhaps one of the simplest R-tree variants; it deals with the temporal dimension simply by treating time as a regular "spatial" dimension.

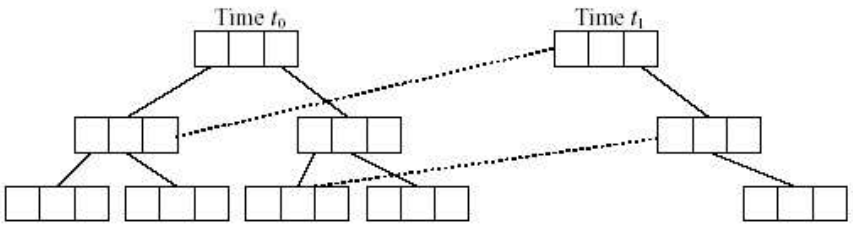


Figure 14: HR-tree (from [45])

Spatio-temporal data is indexed using a standard 3-dimensional R-tree structure.

The Historical R-tree (HR-tree) [32] is an example of an overlapping and multi-version structure which also adapts the R-tree for historical spatio-temporal data. The HR-tree extends the idea of multi-version B+-trees [21] into the spatial-temporal domain. A "virtual" spatial R-tree for each time-stamp (sample point) in the dataset is created. As shown in Figure 14, to avoid the storage overhead of storing a separate R-tree index at each time-stamp, the nodes pointing to nodes that do not change are linked back to the original structure. For a time slice query, the approach is extremely efficient because a R-tree at each time stamp "virtually" exists. For window queries, however, the approach is not as effective. A similar approach using Quad-trees is suggested in [49].

The Trajectory Bundle Tree (TB R-tree) [39] proposes a trajectory-oriented access methods that can (under certain conditions) answer trajectory-oriented queries faster than the R-tree. As shown in Figure 15, the TB R-tree calls for a modified R-tree structure wherein leaf nodes that contain information from the same trajectory are linked together. This can lead to reduced query time when retrieving the complete trajectory of an object. In [37], the authors suggest an R-tree based approach to handle on-line mobile objects with infrastructure (lakes, etc.) which uses the TB R-tree as the underlying STAM. In order to improve query performance, the authors propose a method to segment spatio-temporal queries. The work of Pfoser in [38] is noteworthy in that it deals with networked constrained historical data using the TB R-tree. A technique to reduce the dimensionality of the dataset is employed in order to take advantage of the underlying network.

In [8], the authors propose a grid based spatio-temporal indexing technique which they call SETI. The idea of using a grid or partitioning scheme to index data dates back to the work on grid files [12, 33]. The Quad-tree [42, 49] in essence is also a spatial partitioning approach. SETI partitions the spatial dimension into static, non-overlapping partitions, and within each partition uses a "sparse" tempo-

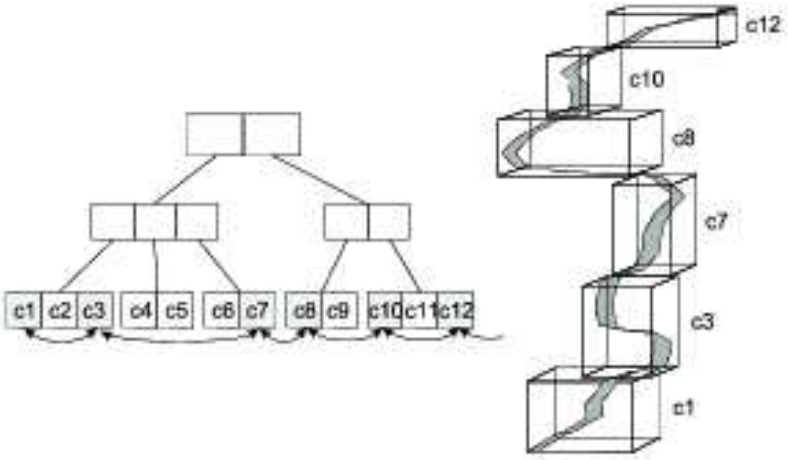


Figure 15: TB R-tree (from [36])

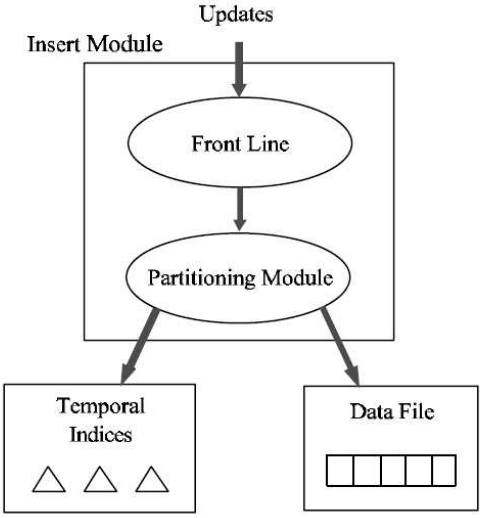


Figure 16: SETI (from [8])

ral 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. Figure 16

provides a conceptual representation of the SETI model. The in-memory front line structure aims to process inserts and updates in an expeditious manner. The partitioning module manages a separate temporal index and data file for each partition.

Our approach builds on the framework SETI uses, i.e., a temporal index inside of spatial partitions, in addition, our work provides (1) a cost model to analytically (instead of experimentally as SETI does) determine the optimal number of partitions to use with respect to minimizing disk access, (2) a relational mapping of our proposed STAM to any RDBMS and (3) a comprehensive experimental comparison of our proposed technique against several other RDBMS-supported spatio-temporal indexing alternatives.

3 RDBMS Support for Spatio-Temporal Data

There are several lines along which the problem of indexing spatio-temporal data inside a RDBMS can be classified. In this section, we provide an implementation-oriented classification in the sense that we focus on the various options for integrating the internal mechanisms of support inside the RDBMS. This section also provides necessary theoretical background on the techniques that will be used for experimental comparison later in this technical report. We distinguish three main alternatives for providing RDBMS support for spatio-temporal data:

1. Recasting the problem in a way such that existing RDBMS facilities for spatial data can be used.

2. Designing a new STAM, and loosely coupling it with a RDBMS via wrapper functions that can be called from within SQL queries, and

3. Designing a new STAM, and tightly coupling it within the RDBMS using native RDBMS facilities, i.e., relational tables and existing index support.

Next we discuss each of these in turn.

3.1 Recasting the problem

Most RDBMSs, including DB2 [3], MySQL [27], and Oracle [35], provide support for spatial data, e.g., R-tree and Quad-tree indexes. There are several ways to recast the spatio-temporal data indexing problem to take advantage of RDBMS-support for spatial data. Perhaps the most straightforward recourse is to model time as an additional "spatial" dimension so that existing support for spatial indexes can be re-used. By combining the 2-dimensional spatial domain and the 1-dimensional temporal domain, a 3-dimensional R-tree (c.f., Figure 2) can be adapted to support spatio-temporal indexing using this approach. Such an approach often results in several snags – performance is poor because the spatial index is unaware of the distinctive nature of the temporal dimension. Typically, R-tree construction requires that each dimension be bounded – a constraint that time does not observe. Also, certain spatio-temporal data models, such as the parametric and the step-wise model we employ, cannot be readily integrated into a 3-dimensional R-tree structure. It should also be mentioned that typically, the spatial index structures provided by the RDBMS are tuned to support 2-dimensional data [35]. For example, a 3-dimensional R-tree in Oracle supports only a small subset of the query functionality and optimization provided were the data 2-dimensional. Although the strategy of treating time as a third dimension benefits from the ease of re-using

existing RDBMS spatial support, because of the discussed drawbacks the approach is unsuitable to solve the spatio-temporal indexing problem.

3.1.1 Linear Referencing System (LRS) Approach

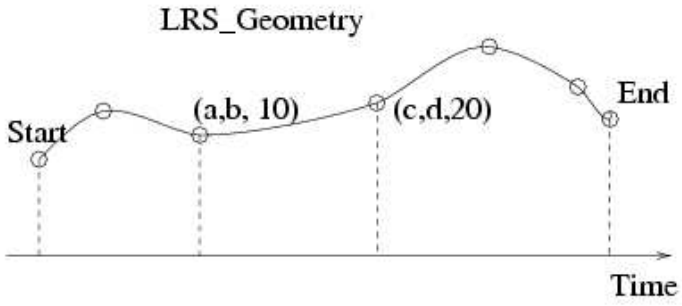


Figure 17: Example of a Spatio-Temporal Geometry Modeled using a LRS_geometry (from [17])

To overcome some of the above limitations, in [17], the authors suggest the use of Oracle's Linear Referencing System (LRS) to index the third (temporal) dimension of spatial-temporal objects. The advantage of using LRS is that the temporal dimension can be indexed separately from the spatial without having to create a separate storage location (column) to store temporal information. Figure 17 shows an "LRS_Geometry" object representing the trajectory of a moving object. The point "(a,b,10)" represents an object at location "(a,b)" at time "10". The "Start" and "End" time of the trajectory can be extracted and indexed using functions provided by the LRS package. The spatial component of the trajectory is then indexed separately from the temporal using a standard 2-dimensional R-tree. Figure 18 visualizes the proposed LRS-based dual-index model. The key advantage of using LRS to index spatio-temporal data is that the spatio-temporal object can be stored as a single object in a table while still allowing for the flexibility of indexing the spatial and temporal domains using separate structures. In our experimental analysis, we will employ an indexing approach based on LRS as a means of comparison to our proposed method.

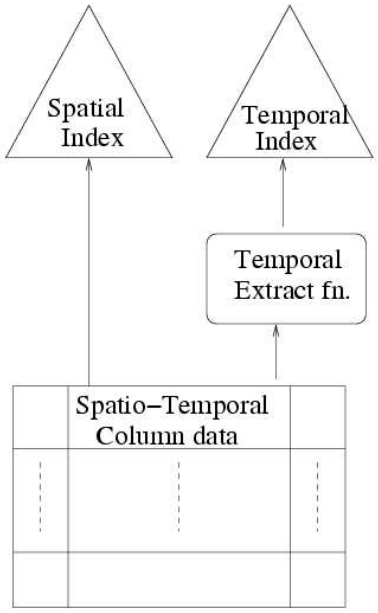


Figure 18: Indexing Spatio-Temporal Data using the LRS Approach (from [17])

3.1.2 Space-Filling Curve Approach

Another option (which we will also compare to our approach) by which to index spatio-temporal data using built-in RDBMS spatial data support is to employ space-filling curves. Given a data space that is partitioned into rectangular cells, a space-filling curve can be thought of as a thread that visits each cell in the grid once and only once, thus imposing a linear-order on the multi-dimensional space [24]. Because B+tree indexes are inherently linear (one-dimensional), a space-filling curve can be used to map multi-dimensional data into a 1-dimensional space that the B+tree can index. Objects can be approximated by an integer representing the cell number defined by the curve instead of using their exact spatial coordinates. Figure 19 illustrates various space-filling curves that map a two-dimensional space into a one-dimensional space, namely the Sweep, Z, and Hilbert curves. The mapping should be locality-preserving in the sense that points close together in two-dimensional space should still be close together in the one-dimensional space defined by the curve. Each type of curve defines a different linear order on the space with unique locality-preserving properties – a thorough analysis of the performance behavior of the various curves can be found in [24]. Ideally we want

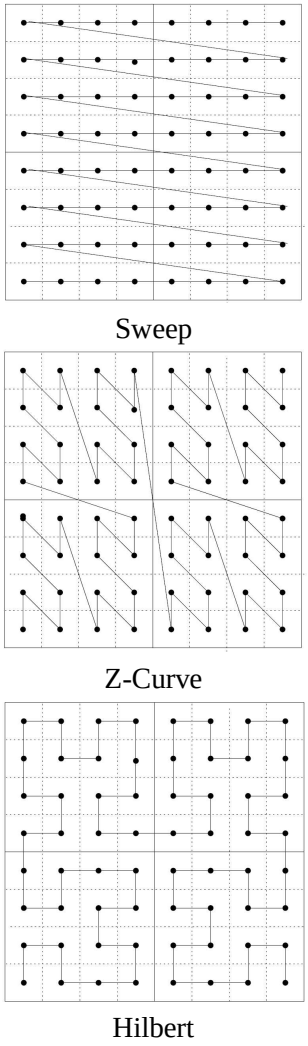


Figure 19: Space-Filling Curves (from [24])

two points that are close together in Euclidean space to be close in the linear order defined by a space-filling curve. Also, cell numbers should be easy to compute because data insertion, updates, and queries will require repeated cell number calculations.

Typically, there is a trade-off between the quality of the locality-preservation and the computational complexity of the curve. The sweep-curve, which we will use with SPIT, does not preserve spatial proximity especially well but can be

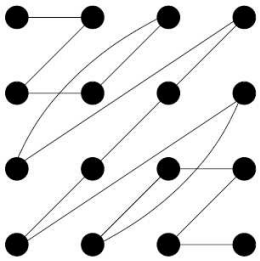


Figure 20: Spectral Curve (from [23])

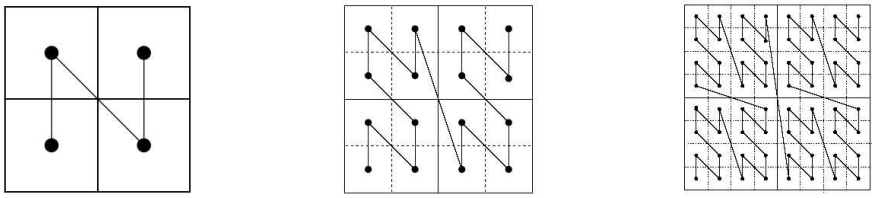
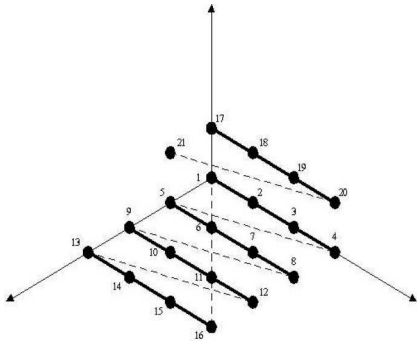


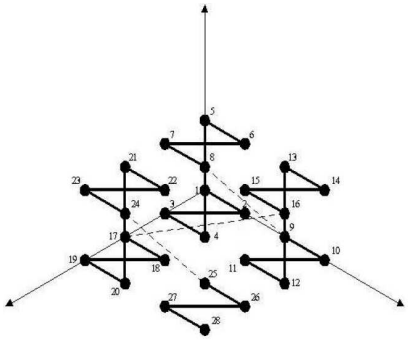
Figure 21: Recursive Construction of the Z-curve (from [24])

computed with ease. The Z-curve, labeled such due to its side-ways "Z" shape, can be calculated very efficiently using a bit interleaving process while providing reasonable locality-preservation. The Hilbert curve can be shown to provide a slightly better spatial clustering (locality-preservation) than the Z-curve [26] but is more expensive to compute. Figure 20 shows a unique type of space-filling curve named the spectral-curve. This curve can be shown to provide the optimal locality-preservation given a fixed set of multi-dimensional points [23]. Unfortunately, it is not practical for our purposes because it can only be computed over a static dataset, e.g., the curve would require a complete re-calculation whenever an update to the database occurred. Another important classification for space-filling curves is that of recursive versus non-recursive space-filling curves. As demonstrated in Figure 21, by starting with a basic shape, i.e., a side-ways 'Z" shape covering four cells, the Z-curve can be defined recursively within each sub-quadrant by further expanding the same "Z" shape. Likewise, the Hilbert curve can also be defined by following a recursive pattern. It should be noted that space-filling curves can be extended to cover a multi-dimensional space. Figure 22 illustrates the shape of the various space-filling curves discussed as they thread through a three-dimensional grid.

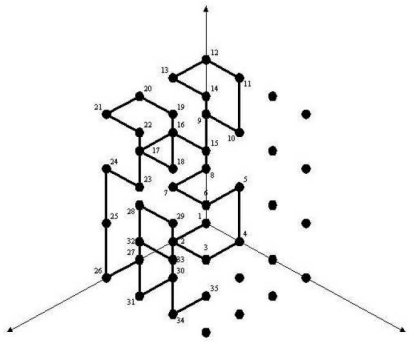
In [13] the authors show how to support geo-spatial operations in a RDBMS



Sweep



Z-Curve



Hilbert

Figure 22: Three-dimensional Space-Filling Curves (from [24])

using Z-curves and B-tree indexes. This approach can be readily extended into the spatio-temporal domain by approximating the spatial component of records using cell numbers (as defi ned by a space-fi lling curve) and creating a combined B-tree

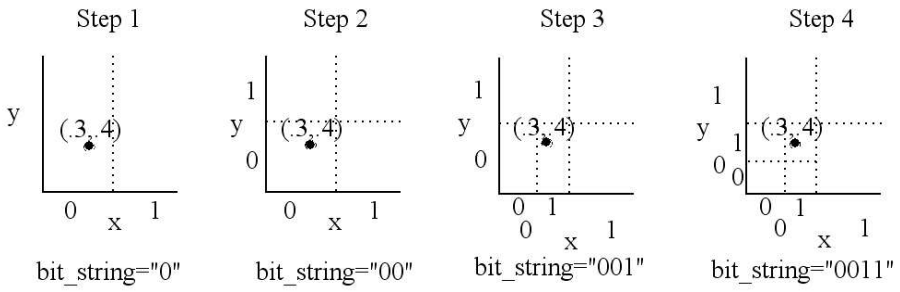


Figure 23: Bit Interleaving to Calculate Z-values

index over the cell numbers and temporal dimension of the records. A combined B-tree index is identical to a standard B-tree index, except that the indexed keys are the concatenation of the two or more values, i.e., cell number and temporal information. As in [13], we decided on the Z-curve for this approach because it provides a reasonable trade-off between computational overhead and locality-preservation. In order to compute the Z-value for point data, one uses a fixed resolution of the space in all dimensions, i.e., each cell has the came size. Each point is then approximated by one cell using a recursive bit interleaving process. The process works by recursively partitioning the data space into two halves and alternating between the x and y dimensions in order to append values to a bit string as follows: if the point lies in the left/bottom half of the partition append a "0" to the bit string, if a point lies in the right/top partition append a "1" to the bit string. The decimal value of the bit string is the Z-value of the point. For example, Figure 23 demonstrates computing the Z-value of the point (.3,.4) assuming a unit space and a "4×4" grid. In the first step, because $x = 0.3$ is to the left of 0.5 (half of the space), a "0" is appended to the bit string. In step 2, "0" is also appended for the y dimension because $y = 0.4$ is below 0.5. Step 3 divides the x dimension in half again, adding a 1 to the bit string because $x = 0.3$ is to the right of $x = 0.25$. Alternating to the y dimension (Step 4) adds another 1 to the bit string because $y = 0.4$ is above 0.25. The final bit string is "0011" which upon conversion to decimal gives a Z-value of 3. Note that this process can be readily extended to compute a three-dimensional Z-curve.

Using a Z-order with B-tree approach, the spatial component of records is approximated by cells defined according to the linear Z-order and a combined B-tree index over Z-values and time is created. At query time it is necessary to map the spatial component of a query to the one-dimensional space. As shown in Figure 24, a window query becomes a range query on the linear order, e.g., find all entries

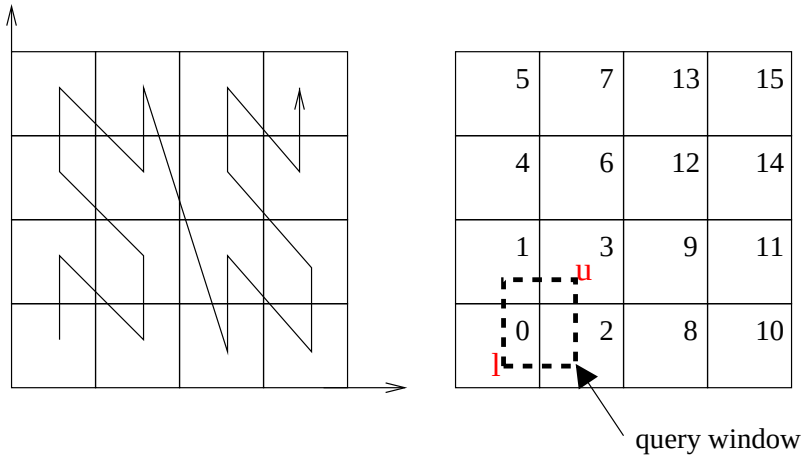


Figure 24: Z-Order Space-Filling Curve to Cell Number

(Z-values) in the range $[l, u]$ where l = smallest Z-value of the window (bottom left corner) and u = largest Z-value of the window (top right corner). The search over the combined index first narrows the search to those tuples within the range of Z-values $[l, u]$ and then to those tuples within the temporal range of the query window.

As mentioned, the Z-order with B-tree approach and the LRS-based R-tree strategy we described, are example techniques for indexing spatio-temporal inside the RDBMS by recasting the problem into a spatial data problem that can be integrated using existing RDBMS technology. Because spatio-temporal data has its own distinct properties, e.g., time is an unbounded dimension, approaches that recast the problem as a spatial indexing problem tend not to perform as well as specialized methods for spatio-temporal data. Our approach (SPIT) is to develop a true-spatio-temporal access method and provide a relational mapping thereof. We will use the Z-order with B-tree and LRS-based approaches for experimental comparison against our approach in order to understand the performance impact such design decisions have.

3.2 Loose Coupling

Many of the R-tree based structures previously discussed, e.g., [1, 32, 39, 47], could be used within a RDBMS using a loosely coupled approach, i.e., by taking an existing implementation of a STAM and wrapping it into an Application Programming Interface (API) that the RDBMS can access. Some RDBMSs support, for

example, Java classes that can be loaded directly into the database. A Java-based implementation of the STAM, which maintains its own set of data structures, could be loaded into the RDBMS and used to index spatio-temporal data. An example of such an approach in the spatial domain is [34] wherein a separate extra-database file structure is used to store spatial information and indexes.

We are not aware of any loosely coupled approaches proposed for the spatio-temporal domain, in part due to the serious drawbacks such a strategy entails. Because of the loose coupling, the STAM has limited ability to manage buffers, deal with disk page layout, or access internal database structures. Furthermore, the STAM designer is left with the daunting task of integrating standard RDBMS features, such as concurrency control and crash recovery mechanisms, into their access method. Our approach avoids these pitfalls because we map the problem directly onto relational tables and existing index structures, thus automatically inheriting the desirable properties that a RDBMS provides.

3.3 Tight Coupling

One of the key features of our proposal is that we employ a tight coupling between our STAM and the database using (only) native RDBMS functionality, i.e., relational tables and existing index support. Tightly coupling the STAM within the RDBMS involves providing a relational mapping of the STAM. Examples of such an approach in the temporal domain include the Relational Interval tree (RI-tree) [19] and the MAP21 technique [30]. As discussed, the key to both these approaches is the translation of a logical temporal index structure into an implementation relying on pre-existing RDBMS functionality. The RI-tree takes advantage of database extender technology while the MAP21 uses a data translation function to map the start and end points of temporal interval to 1-dimensional values that can indexed using a B+tree.

Such an approach is also used in the spatial domain to map the Quad-tree and R-tree into the RDBMS. For example, the R-tree in Oracle Spatial is a purely logical structure whose underlying implementation is based on relational tables and B+tree indexes [18]. Similarly, the mechanism underlying the Quad-tree relies on Z-order tessellation, relational tables, and B+tree indexes [18].

The main difficulty of integrating existing STAMs within a RDBMS is the enormous gap between conceptual model and relational mapping. In general, the mapping of a STAM inside the RDBMS is not obvious. Much of the importance of our work is that we, to the best of our knowledge, are the first to propose a STAM that can be tightly coupled inside a RDBMS using only native RDBMS functionality. As we will discuss, this makes it possible for our method to be smoothly integrated with any off-the-shelf RDBMS.

4 The Space-Partitioning with Indexes on Time Approach

In what follows we present our proposal for spatio-temporal data management integrated within a RDBMS – which we name the Space-Partitioning with Indexes on Time (SPIT) approach. 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 where the tuple is physically stored.

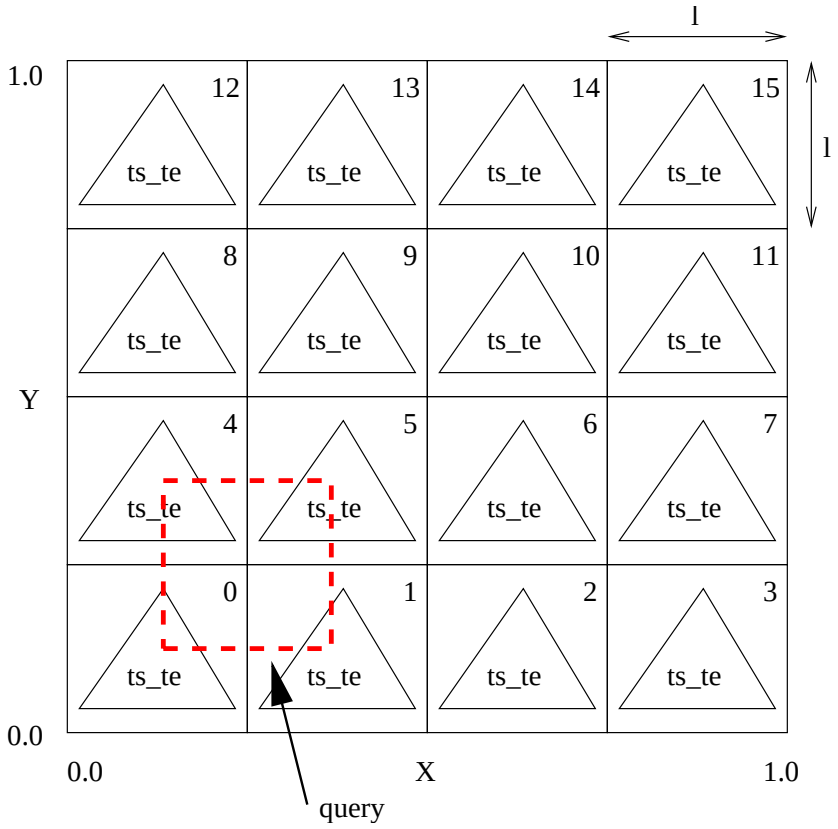


Figure 25: SPIT Approach for a 4 x 4 spatial grid

As shown in Figure 25, we use a static grid and number the cells using a horizontal sweep space-filling curve in order to give each cell a unique identifier pid. The pid of each cell is shown in the top right corner. The length l refers to the size of a grid cell in each spatial dimension. Recall that a spatio-temporal range query has a spatial component $\mathcal{R}$ and temporal interval $\mathcal{T}$ consisting of the start time t_s

and end time t_e of the query window. To efficiently support such queries, SPIT defines a local temporal index on $\langle t_s, t_e \rangle$ over the domain of tuples within each partition. The key advantage of spatial partitioning is that of partition elimination at query time. For example, given the query window shown in Figure 25, only cells $(0, 1, 4, 5)$ need be examined to answer the query window – cells that do not intersect the spatial component of the query window can be eliminated from consideration. For spatio-temporal data this works extremely well because we further apply a temporal filter within all intersecting cells using the temporal indexes on $\langle t_s, t_e \rangle$ inside each partition. 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 [8], 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 on the actual spatial location of tuples, and (4) duplicate elimination. Section 4.1 describes how to map SPIT's data and query processing model to a RDBMS. A critical performance factor in SPIT is choosing the number of grid cells to use, i.e., setting the length l, therefore Section 4.2 provides a disk access based cost model for choosing an optimal number of partitions to use.

4.1 Mapping SPIT into a RDBMS

Algorithm 1 $find\_pid()$ function

INPUT: $\langle x, y \rangle$
OUTPUT: pid

1: $x\_grid := \lfloor x \times N_p^* \rfloor$
2: $y\_grid := \lfloor y \times N_p^* \rfloor$
3: **if** $x = 1.0$ **then**
4: $\quad x\_grid := x\_grid - 1$
5: **end if**
6: **if** $y = 1.0$ **then**
7: $\quad y\_grid := y\_grid - 1$
8: **end if**
9: **return** $x\_grid + N_p^* \times y\_grid$

Mapping SPIT to a RDBMS requires defining and assigning partitions, creating partitioned temporal indexes, and providing a query mechanism. Algorithm 1 provides the pseudo-code of the $find\_pid()$ function which returns the pid of a record given its spatial coordinates $\langle x, y \rangle$. The algorithm assumes that each cell

is assigned its pid according to a horizontal sweep space-filling curve (c.f., Figure 25). The algorithm uses (as a constant) the number of partitions to use in one dimension, which we label N_p^*, and assumes a unit space. In Section 4.2, we provide a means to analytically determine N_p^*. Lines 1–2 of the $find\_pid()$ function calculate x_grid and y_grid, which correspond to the grid cell in the x and y dimensions where the $\langle x, y \rangle$ coordinate resides. Lines 3–8 deal with the special case of a x (y) value lying on the rightmost (uppermost) boundary of the grid. Line 9 returns the pid, which using a horizontal sweep curve consists of the grid number (x_grid) in the x dimension added to the number of cells in one dimension N_p^* multiplied by the grid number (y_grid) in the y dimension.

The pid attribute maps each tuple to a unique partition in the database where the tuple is physically stored. Conceptually, a RDBMS with partition support treats a partitioned table as a collection of separate tables that can be accessed as though they were a single table. Some RDBMSs provide built-in support for table partitioning. If this feature is unavailable, partitioned table support can be simulated by explicitly creating a set of tables (one for each partition) managed via a partition meta-data lookup table.

Next we create local indexes over the temporal domain of each partition. A combined B-tree index on $\langle t_s, t_e \rangle$ is used as the index of choice. Recall that a combined B-tree is simply an index over the concatenation of the two columns of data. Note that the options of creating a combined index on $\langle t_e, t_s \rangle$ or two separate B-trees on t_s and t_e also exist. Using two separate indexes, however, is unlikely to improve performance because for those tuples that satisfy t_e, the start time t_s must also be checked (and vice versa). There is also the option of creating all (or some) of the above indexes and allowing the RDBMS's query optimizer to choose which (if any) of the indexes to use. This may provide performance advantages in certain situations but has the drawback of extra index creation, maintenance and space overhead.

We considered the use of a 1-dimensional R-tree to index the temporal dimension. However, we abandoned the idea because in our data model, a large degree of overlap among the temporal intervals of objects occurs – querying almost any time interval will return nearly all oid's. The problem is that the selectivity in the temporal dimension is poor because most (if not all) objects exist somewhere in the space at all times. In such a situation the performance (and index creation times) of the 1-D R-tree approach is prohibitively expensive. 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. 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.

The order in which data is clustered on disk strongly impacts query perfor-

mance. Because the temporal index within each partition requires a local range scan of the data, query performance will be faster if the data on disk is already sorted according to time. Therefore, before inserting data it is beneficial (though not necessary) to ensure that all tuples are ordered by $\langle t_s, t_e \rangle$. This ensures that all those tuples satisfying the temporal component of a query will be located close together on disk.

4.1.1 Query Processing

Since a spatio-temporal query Q consists of a spatial range $\mathcal{R}$ and a temporal interval $\mathcal{T}$, query processing requires four steps:

1. spatial filtering,
2. temporal filtering,
3. spatial refinement, and
4. duplicate elimination.

Algorithm 2 provides the pseudo-code for the function $st\_query()$ which processes queries according to the four steps used by the SPIT model. Note that lines 4–9 assume the existence of a SQL interface in order to retrieve matching tuples from partitions in the RDBMS. A key advantage of SPIT is that filtering occurs in a pipelined fashion – at each step of query processing only those tuples satisfying the previous step are further examined. In what follows the details of the $st\_query()$ function are explained.

Spatial Filtering

Only those cells intersecting $\mathcal{R}$ need to be scanned to answer Q. Algorithm 3 provides the pseudo-code for the function $p\_intersect()$ which returns the list of pid's of (only) those cells which intersect $\mathcal{R}$ (assuming $\mathcal{R}$ is rectangular). For example, using the sample query window shown in Figure 25, the return value of $p\_intersect()$ would be $(0, 1, 4, 5)$. The $p\_intersect()$ function first (lines 1–2) sets the minimum and maximum pid of the bottom-left and top-right corners of the query rectangle by calling the function $find\_pid()$. Line 3 calculates the number of rows of the grid that the query window extends over. The loop from line 4–8 iterates over the columns that intersect the query window (line 4), and over the rows in each column (line 5), adding each intersecting cell to the pid_list (line 6). Because grid cells are mapped to partitions, the list allows the query algorithm to take advantage of partition elimination. Line 2 of Algorithm 2 uses the pid_list to

Algorithm 2 $st\_query()$ function

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

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

scan only those partitions corresponding to cells intersecting $\mathcal{R}$. No disk reads of other partitions occur.

Temporal Filtering

Within each partition that needs to be scanned, we can further improve performance by filtering out those tuples that do not intersect the temporal interval component of the query using the local temporal index (lines 6–7 of Algorithm 2).

When querying historical information aggregated over an extended period of time, it is often the case that the time between updates of an object's position do not exceed a certain maximum. In fact, one can always maintain, as meta-data, the length of the maximum stored temporal range. Query performance can be improved by assuming the largest temporal interval is known, as in [30], by which we can further restrict the temporal interval that needs to be scanned – we name this constant *MAX_TI*. For example, in fleet monitoring, it can be safe to assume that vehicles do not remain stationary for more than 2 or 3 days. Moreover, the value of *MAX_TI* will become relatively smaller as the database becomes "older". Objects within the query temporal interval $\mathcal{T}$ cannot have begun or ended their temporal before/after $T \pm$ *MAX_TI*. In our experimental section we show that the maximum interval assumption speeds up query performance. Note that 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. Using the maximum time interval assumption, lines 6–7 of Algorithm 2 are replaced with

Algorithm 3 $p\_intersect()$ function

INPUT: $\mathcal{R}$
OUTPUT: list of pid's

1: $pid_{min} := find\_pid(R.x_{min}, R.y_{min})$
2: $pid_{max} := find\_pid(R.x_{max}, R.y_{max})$
3: $num\_rows := \lfloor \frac{pid_{max} - pid_{min}}{N_p^*} \rfloor$
4: **for** $i := pid_{min}$ to $pid_{max} - num\_rows \times N_p^*$ **do**
5: **for** $j := 0$ to num_rows **do**
6: $pid\_list := pid\_list \cup (i + j \times N_p^*)$
7: **end for**
8: **end for**
9: **return** pid_list

the following.

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*

After the temporal filtering phase, only those tuples within each partition that intersect $\mathcal{T}$ are further scanned in the (next) spatial refinement step.

Spatial Refinement

The spatial coordinates of each tuple satisfying the above two checks are scanned by retrieving the tuple from the table partition where it is stored (lines 8–9 of Algorithm 2). If the $\langle x, y \rangle$ coordinates of tuple is inside $\mathcal{R}$ then it's oid is in the answer set. This check is necessary because a tuple may be inside of the partition but not inside of the query window. For example, using the query window from Figure 25, any tuples in grid cell 0,1,4 or 5 but not inside the query window would have to be removed from the answer set. Note that this could be improved by not scanning the disk when a partition is completely contained by the spatial component of the query $\mathcal{R}$, i.e., by performing an index-only scan. Such a strategy, however, would required a modified querying mechanism that could determine when complete containment occurs in order to take advantage of this special case. As well, because the actual answer set returns the oid of each object within the query window, the oid column would need to be included as an additional indexed column in the combined temporal index in order for the query mechanism to remain index-only. Otherwise, even though the query would not have to read from disk to perform the spatial refinement in this special containment case, it would still need to go to disk

to read the oid.

Duplicate Elimination

Duplicates answers can occur because within a partition there may be several tuples within the query answer set for the same object. Because our query should return only the oid's that satisfy Q, the final query phase is to eliminate any duplicate oid's in the answer set (line 11 of Algorithm 2). This can be easily accomplished by the use of a `unique` clause in the SQL query.

4.2 SPIT's Cost Model

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 accesses
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	size (fraction of the space) of query in one spatial dimension
q_t	size (fraction of the space) of the temporal aspect of the query
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: Symbols Used and their Meanings

We propose the following cost model to choose an optimal grid size for use with SPIT assuming a fixed regular grid. Table 1 lists the notation we will use.

Assuming a unit space $[0, 1]$ in each dimension then 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. This can be formalized as:

$$DA = GA \times (DA_g + IA_g) \quad (1)$$

In [44], the authors derive a formula for calculating the average number of boxes that intersect a known size query window, assuming a unit space. The average number of boxes that will be scanned is the total number of boxes multiplied by the average space the spatial component of a query covers extended by the average length of each box. The extended query covers the case of a query intersecting a box only partially. Based on [44], we formalize the average number of cells that will be scanned (GA) as the total number of cells multiplied by the average space the spatial component of a query covers extended by l:

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

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 dimension of our query q_t:

$$DA_g = \frac{N/N_g}{BS} \times q_t \tag{3}$$

Assuming 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 using: $IA_g = log_f N$. 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 [22].

Combining Equations 2 and 3, into Equation 1 yields:

$$DA \quad = (l+q)^2(\tfrac{N \times q_t}{BS} + \tfrac{3}{l^2}) \tag{4}$$

One immediate observation from Equation 4 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 effi ciently.

To fi nd the grid size l^* that will minimize disk accesses we take the fi rst derivative of (4) with respect to l and set it to 0, obtaining (after some algebraic manipulation):

$$l^* = \sqrt[3]{\frac{6q \times BS}{2N \times q_t}} \tag{5}$$

which can be shown to be a unique solution and can also be shown to be a minimum according to the second derivative test with respect to l^*. Finally, the optimal

number of grid cells (N_g^*) can be represented in terms of l^* using

$$N_g^* = \frac{1}{(l^*)^2} = (\frac{N \times q_t}{3q \times BS})^{2/3} \tag{6}$$

Note that in the special case where the average query size in the spatial and temporal dimensions is equal, i.e. $q = q_t$, an optimum number of grid cells regardless of query size can be determined. It is also interesting to note that N_g^* grows sublinearly with N. This suggests that SPIT can be sensitive to the database size; however, our experimental results will show that SPIT is fairly resilient to the growth of N.

Recall that Algorithms 1 and 3 relied on a constant N_p^* corresponding to the number of partitions in each dimension. Based on equation (6), N_p^* can be (near) optimally set as:

$$N_p^* = \lceil \sqrt{N_g^*} \rceil \tag{7}$$

In the next section we describe instantiating SPIT inside an Oracle database using the query processing and cost model just outlined.

5 RDBMS Support Implementation Within Oracle

Here we describe how to instantiate SPIT inside an Oracle database. We discuss the implementations of the R-tree and space-fi lling curve with B-tree approaches that we use as comparisons to SPIT. For completeness, we also describe other alternative methods that were considered during the course of this research and the reasons these methods were abandoned.

5.1 Space-Partitioning with Indexes on Time Approach

The SPIT grid is implemented using Oracle's built-in table partitioning support – a grid cell corresponds to a single Oracle table partition. The number of partitions (N_p^*) to use is calculated according to Equation 7. (Oracle allows a maximum of $2^{16} - 1 = 65535$ partitions per table.) The `ST_SPIT` table (whose DDL for an example 2×2 grid is provided in Table 2) stores records along with the additional pid attribute. Oracle range partitioning is used to automatically map the spatial grid to unique table partitions on disk. Conceptually, Oracle treats the `ST_SPIT` table as a collection of separate tables – one for each partition.

Within each partition, a local domain B-tree index is created over $\langle t_s, t_e \rangle$ using the index creation DDL provided in Table 2. In Oracle, the only difference between creating a local partitioned index versus a standard "global" index is the keyword `local` as part of the index creation DDL. Before inserting data into the `ST_SPIT` table we order all tuples by $\langle t_s, t_e \rangle$ through the use of a temporary table. For each tuple inserted into `ST_SPIT` the pid is calculated using a PL/SQL implementation of Algorithm 1.

We considered and rejected the use of techniques for RDBMS-support of temporal intervals to index the temporal dimension, e.g., the Relational Interval Tree (RI-tree) [19]. Unfortunately, the RI-tree implementation in Oracle does not currently support partitioned indexes and its data insertion times were prohibitively expensive for the datasets we used.

As mentioned earlier we do not consider the use of a 1-dimensional R-tree index over the temporal dimension. Experiments over a dataset of 6 million tuples showed that query performance for this approach was 5 times higher than a B-tree index on $\langle t_s, t_e \rangle$. Furthermore, it took nearly 16 times longer to create the 1-dimensional R-tree index as opposed to the B-tree index. The 1-dimensional R-tree is outperformed because of the high amount of overlap among the time intervals of the tuples. Also the R-tree's performance may suffer because Oracle Spatial does not provide native support for 1-dimensional objects – we were forced to "pad" the data with a second dimension in order to successfully build the 1-dimensional R-tree.

Table Creation DDL	`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 (MAXVALUE)`
	`)`

Index Creation DDL	`CREATE INDEX idx_st_spit_t` `ON ST_SPIT(t_s,t_e) LOCAL`
Sample SQL Query	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 0.6 and 0.6 + MAX_TI` 6: `AND  x  between 0.1 and 0.3` 7: `AND  y  between 0.2 and 0.4`

Table 2: SPIT DDL and SQL statements

Given the sample SQL query *"find the objects that were within the area 0.1–0.3 x and 0.2–0.4 y during during the interval 0.5 to 0.6"*, Table 2 provides the SQL query that would be issued against the `ST_SPIT` table. The query assumes a 4×4 grid by which to calculate pid's.

Line 3 of the sample query in Table 2 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 using a PL/SQL implementation of Algorithm 3. Only 4 out of 16 partitions need be scanned – a signifi cant reduction in I/O cost achieved with only a small computational overhead.

Lines 4–5 correspond to the *Temporal Filtering* stage of SPIT's query process-

ing. 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. Because the index on $\langle t_s, t_e \rangle$ provides pointers to the actual physical location on disk where the corresponding tuples are stored, the order in which data is clustered on disk will have an important impact on query performance. By ordering the data according to $\langle t_s, t_e \rangle$ at insertion time, query processing will be faster because those tuples with similar temporal intervals will be located close together on disk.

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. It is this step of query processing that performs disk access because the spatial coordinates are not represented in any index. By ensuring that the data is clustered according to $\langle t_s, t_e \rangle$, disk access will be minimized because the tuples in the query answer set will be located close together, i.e., within the same block on disk, thus reducing disk seek time.

Finally, line 1 corresponds to the *Duplicate Elimination* stage of SPIT's query processing. The unique clause on oid removes duplicates from the query result which may have occurred because several tuples for the same object were in the answer set.

We implemented the necessary SPIT functions (Algorithms 1, 2 and 3) using Oracle's built-in procedural language PL/SQL. The implementation is capable of generating SQL queries of the form provided in Table 2 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; however, any language capable of interacting with the RDBMS could be used.

5.2 The R-tree + Temporal B-tree Approach

As a method of comparison to SPIT, we adapt the LRS spatio-temporal indexing approach suggested by Oracle [17] 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. We do not utilize a Quad-tree based approach because Oracle reports that for most applications the R-tree is more effi cient [18] than the Quad-tree. The R-tree also has the advantage of requiring no parameterization except the choice of dimensionality – with the Quad-tree a tessellation level must be chosen. The "R-tree + Temporal B-tree" approach uses the `ST_RTREE` table whose creation DDL is provided in Table 3. The syntax for creating both the spatial index and temporal indexes is also provided in Table 3.

A sample query against the `ST_RTREE` table (using the same query range as with the `ST_SPIT` table) is provided in Table 3. Queries on this approach employ the Oracle Spatial [35] built-in spatial query predicate $sdo\_relate()$ which takes

Table Creation DDL

```
CREATE TABLE ST_RTREE (
       oid       INTEGER,
       position  MDSYS.SDO_GEOMETRY,
       t_s       NUMBER,
       t_e       NUMBER
)
```

Index Creation DDL

```
CREATE INDEX idx_st_rtree
ON ST_RTREE(position)
INDEX TYPE IS MDSYS.SPATIAL_INDEX

CREATE INDEX idx_st_rtree_t ON ST_RTREE (t_s,t_e)
```

Sample SQL Query

```
1: SELECT  unique oid
2: FROM    ST_RTREE
3: WHERE   sdo_relate(
4:           point,
5:           MDSYS.SDO_GEOMETRY(
               --spatial query window
               2003, --2-dimensional polygon
               NULL,NULL,
               MDSYS.SDO_ELEM_INFO_ARRAY(
               1,1003,3)
               MDSYS.SDO_ORDINATE_ARRAY(
               0.1,0.2,0.3,0.4)
             ),
6:           'mask=ANYINTERACT querytype=window'
           ) = 'TRUE'
7: AND     t_s between 0.5 - MAX_TI and 0.6
8: AND     t_e between 0.6 and 0.6 + MAX_TI
```

Table 3: R-tree + Temporal B-tree Approach DDL and SQL statements

an Oracle geometry column object (line 4), a query window (line 5), and a filtering predicate (line 6), and returns only those objects that intersect the query window. The B-tree index is used to further filter tuples based on the temporal aspect of the query (lines 7–8).

Table Creation DDL	`CREATE TABLE ST_ZORDER (`
	`  oid  INTEGER,`
	`  x  NUMBER,`
	`  y  NUMBER,`
	`  t_s  NUMBER,`
	`  t_e  NUMBER`
	`)`

Index Creation DDL	`CREATE INDEX idx_st_zorder`
	`ON ST_ZORDER(t_s,t_e,z_order(x,y))`
	
	`CREATE INDEX idx_st_zorder`
	`ON ST_ZORDER(z_order(x,y),t_s,t_e,)`

Sample SQL Query	1: `SELECT unique oid` 2: `FROM ST_ZORDER` 3: `WHERE z_order(x,y) between 0 and 3` 4: `AND t_s between 0.5 - MAX_TI and 0.6` 5: `AND t_e between 0.6 and 0.6 + MAX_TI` 6: `AND x between 0.1 and 0.3` 7: `AND y between 0.2 and 0.4`

Table 4: Z-value + B-tree Approach DDL and SQL statements

5.3 The Z-value + B-tree Approach

For the Z-value B-tree approach, the table `ST_ZORDER`, whose DDL is given in Table 4, is used to store records. Algorithm 4 provides the pseudo-code for the function z_order that calculates the Z-value of the cell where the spatial coordinates $\langle x, y\rangle$ of a tuple reside in. Lines 1–8 of the z_order algorithm are used to calculate the grid cell in the x and y dimensions where the point $\langle x, y\rangle$ resides, and are identical to lines 1–8 of Algorithm 1, the $find\_pid$ function. Line 9 of the z_order algorithm calculates the number of bits (num_bits) that will be used to code the Z-value (z_{value}). Recall that Z-values can be effi ciently calculated based on the bit representation of the cell number in each dimension of the point in the grid. The loop from lines 11–16 constructs the z_{value} by iteratively adding a bit from the bit-string representation of x_grid or y_grid. The $mask$ (line 12) is used to turn off all bits except for the bit from x_grid or y_grid we require to add to the

Algorithm 4 z_order function

INPUT: $\langle x, y \rangle$

OUTPUT: z_{value} of the cell containing $\langle x, y \rangle$

1: $x\_grid := \lfloor x \times N_p^* \rfloor$
2: $y\_grid := \lfloor y \times N_p^* \rfloor$
3: **if** $x = 1.0$ **then**
4: $\quad x\_grid := x\_grid - 1$
5: **end if**
6: **if** $y = 1.0$ **then**
7: $\quad y\_grid := y\_grid - 1$
8: **end if**
9: $num\_bits := \log_2 N_p^*$
10: $shift := num\_bits$
11: **for** $i := 1$ to num_bits **do**
12: $\quad mask := 2^{num\_bits - i}$
13: $\quad z_{value} := z_{value} + bitand(x\_grid, mask) * 2^{shift}$
14: $\quad shift := shift - 1$
15: $\quad z_{value} := z_{value} + bitand(y\_grid, mask) * 2^{shift}$
16: **end for**
17: **return** z_{value}

z_{value}. Line 13 adds to the z_{value} a bit from x_grid, and then line 14 decrements the $shift$ value which represents the bit position in the z_value string that we are currently interested in. Line 15 adds to the z_{value} a bit from the y_grid. The algorithm assumes the existence of a $bitand$ operator which performs the logical and operation between two bit strings. For each dataset, we calculate Z-values using the same number of cells in each dimension (N_p^*) that SPIT employs.

Because the Z-value of each tuple is only used for indexing purposes, it does not need to be stored as a separate column in the ST_ZORDER table. Instead, at index creation time, as shown in Table 4, a function-based index is created over the calculated Z-value of each tuple. Using the function-based index reduces table storage overhead which should help increase query performance because the number of tuples that can fit in one block on disk is correspondingly increased. A function-based index acts just as a regular column index, except the indexed value is the result of applying a function to each row in the table. The $z\_order()$ function is invoked at index creation time for every tuple in the database. The two Z-value B-tree approaches we consider for experimental comparison are called *t_z index* and *z_t index* which correspond to creating a combined B-tree index on $\langle t\_s, t\_e, z\_order(x, y) \rangle$ and $\langle z\_order(x, y), t\_s, t\_e \rangle$.

Query processing using the ST_ZORDER table proceeds by first computing the lower (l) and upper (u) Z-values of the spatial component of the query range. The sample query in Table 4 scans the z_value column over the range $\langle l, u \rangle$ (line 3), along with the temporal (line 4–5) and spatial (line 6–7) range components of the query. The *t_z index* and *z_t index* approaches use the same query; the difference between the approaches is whether the primary index filter is temporal or spatial. With both strategies, the actual $\langle x, y \rangle$ of records must be scanned after the index filtering stage to ensure that tuples actually intersect the spatial query range.

5.4 Other Approaches

For completeness, this section describes other approaches which we also explored during the course of this research. For each approach described below we discuss why the technique was infeasible or would be unlikely to provide efficient querying.

Naïve Indexing Approaches

Perhaps the most naïve spatio-temporal indexing approach would be the straightforward use of built-in B-tree indexes. By storing the data in a table with a similar schema as the ST_ZORDER table, various types of indexes on the spatial $\langle x, y \rangle$ and temporal $\langle t\_s, t\_e \rangle$ columns of the data can be created. For example, a separate index on each of the four columns could be used, or a set of combined indexes, perhaps on $\langle x, y, t\_s \rangle$ or $\langle t\_s, x, y \rangle$ could be created. Such approaches will suffer performance-wise because they do not take advantage of the semantics of the underlying spatio-temporal data model. The spatial region is a 2-dimensional space, therefore 1-dimensional index structures are characterized by poor locality-preservation and corresponding slow performance at query time. For example, given two separate 1-dimensional indexes on x and on y, an expensive intersection operation between the index pointers for those tuples within the x range and y range of the query will need to be performed. A 1-dimensional index on time, without any prior spatial discrimination, will also suffer from poor performance due to the high amount of overlap among the temporal intervals and the need to examine both the start and end time of temporal intervals.

3D Z-curve curve Approach

By treating the temporal dimension as a third spatial dimension, a 3-dimensional space-filling curve (for our purposes we considered the Z-curve) can be used for spatio-temporal indexing purposes. The code for calculating the 3-dimensional

z-curve is a modification of Algorithm 4 in order to take into account the extra dimension. This approach was rejected for several reasons. Treating time as a spatial dimension in order to create the 3-dimensional Z-curve ignores the unique properties of the temporal dimension, i.e., time is monotonically increasing. Furthermore, since time is unbounded and space-filling curves can only be computed within a fixed spatial area, creating the 3-dimensional curve necessitates the division of time into bounded intervals within which the Z-curve can be calculated. Another issue was the feasibility of mapping a 3-dimensional Z-curve into both the temporal interval and the spatial dimensions of the data – only the start time (t_s) of the data, and not the complete temporal interval could be indexed using this approach.

Quad-tree based Approaches

Oracle provides built-in support for Quad-tree spatial indexes. The creation of a Quad-tree in Oracle is very similar to the R-tree. By default, when the user creates a spatial index in Oracle the index type is the R-tree, by specifying the tessellation level (`SDO_LEVEL`) to use in the index creation statement a Quad-tree is created instead. We could easily substitute a Quad-tree for the R-tree used in the "R-tree + Temporal B-tree Approach" by changing the index creation DDL in Table 3 to:

```
create index idx_st_quadtree
on st_rtree(position)
indextype is mdsys.spatial_index
parameters('SDO_LEVEL=8')
```

We performed several initial experiments using such a Quad-tree index over the spatial component of the data and a temporal B-tree index. The performance in all cases was extremely poor. Indeed, as reported in [18], the R-tree in Oracle tends to outperform the Quad-tree index for the type of application domain we are interested in.

3-D R-tree

Similarly to the 3D Z-curve approach, we experimented with the use of a 3-dimensional R-tree to index the dataset by treating time as the third spatial dimension. Note that the Quad-tree in Oracle does not support 3-dimensional data. Unfortunately, such an approach is not compatible with the data model we use because the temporal dimension of the data is actually in the form of an interval.

Partitioned Spatial Indexes

Much like the partitioned local temporal indexes used with SPIT, Oracle allows the creation of partitioned spatial indexes. Given a partitioned table, the creation of a partitioned spatial index is identical to the regular syntax for spatial index creation except the keyword `local` is appended. We experimented with variations on this approach, such as creating separate R-tree or Quad-tree structures for each partition in SPIT, or for just the temporal interval in SPIT. We also experimented with a partitioned spatial index as a stand-alone indexing technique. Given the temporal interval component of the data model, however, this approach did not provide adequate query performance because only the spatial component of tuples could be indexed.

Z-curve Partitioning with Indexes on Time

We tested a variation of SPIT that uses a space-filling curve (again, the Z-curve) as the means to partition the space instead of using a sweep space-filling curve. Theoretically, this allows grid cells that are closer in space to be located closer on disk, however the problem with this approach is the overhead of calculating the intersecting cells at query time. With the current sweep-space filling curve approach, the intersecting cells can be computed directly; with a Z-curve approach however a range of cells is computed – which either requires a further refinement phase or the scanning of cells that are outside the spatial component of the query. In initial experiments, the extra overhead of such an approach could not compete with the performance of the sweep space-filling curve approach currently used by SPIT.

6 Experimental Results

In this section we experimentally compare SPIT with other approaches for spatio-temporal support in order to confirm the reliability of the SPIT cost model and to better understand the performance characteristics of the SPIT approach. Due to the need to generate data in a controlled manner for experimental purposes, we used the GSTD [46] to produce several datasets for testing purposes. Recall that GSTD allows for the generation of arbitrary sized datasets using prescribed statistical distributions describing the initial location of objects, the frequency of movement (snapshots) of objects through time, and the movement of objects through space. The datasets we experiment with consist of 1.5 million, 3 million, and 6 million tuples corresponding to 15000, 30000, and 60000 objects with 100 snapshots (sampled positions) each. We experiment with both a Gaussian and Skewed distribution of the data points, both distributions reflecting possible real-world application scenarios. (The specifics of which cannot be disclosed due to confidentiality reasons.) It is important to stress however, that the objects do not follow a simple uniform distribution in the data space. This is important because even though the cost model derived in Section 4.2 assumes a uniform distribution of the data points, our experiments will show that the optimum grid size suggested by the model is still nearly optimal for a non-uniform distribution.

For the Gaussian dataset, the initial location of objects is described by a Gaussian distribution centered in the middle of the space after which point objects movement through space is described by a uniform distribution. A snapshot of the Gaussian dataset at three snapshots is shown in Figure 26. With the Gaussian distribution, the density of points throughout the space is roughly equal after the initial snapshot. A typical real-world scenario would be to assume that certain areas of space have a higher density of objects, therefore we also created and experimented with a Skewed distribution dataset where the density of points through the space is non-uniform. The Skewed distribution was defined such that more data points occur in the bottom-left and top-right quadrants as compared to the bottom-right and top-left quadrants of the data space. A snapshot of the Skewed dataset is shown in Figure 27. The initial distribution is still Gaussian; however the skewness forces more points into the top-right and bottom-left quadrants. All experiments are carried out on both datasets.

We employ three sizes of spatio-temporal queries corresponding to 0.01%, 0.10% and 1.00% of the spatio-temporal space. Note that 1.00% of the spatio-temporal space corresponds to a selectivity of approximately 21.6% on each spatial and temporal dimension. To measure average time/query we issue 100 randomly generated queries and measure the total execution time using Oracle's built-in timing functionality (refer to Oracle's `timing` command for more details). Disk ac-

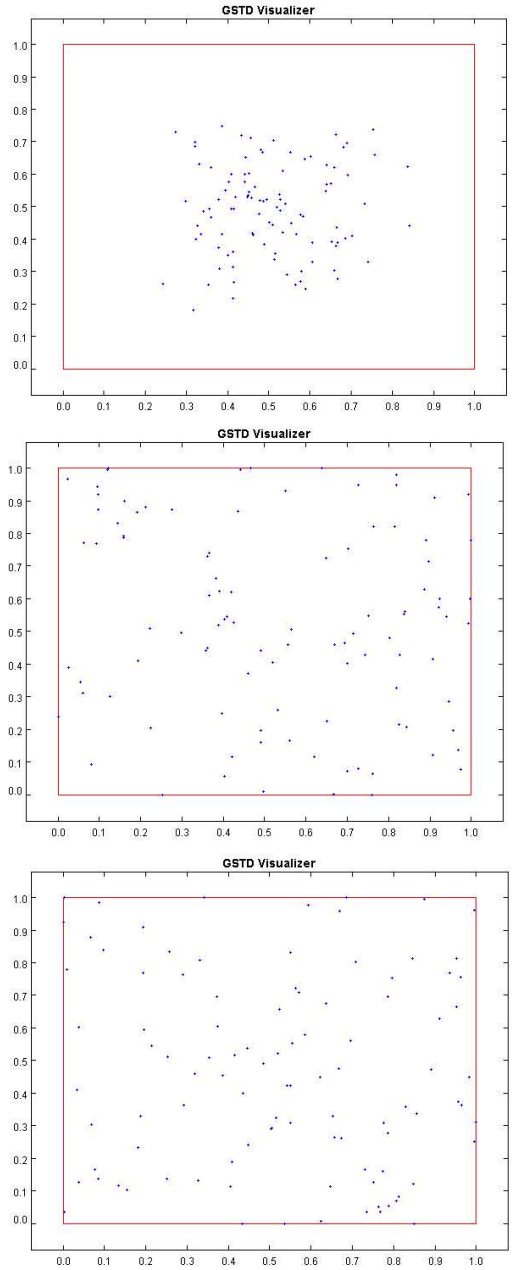


Figure 26: Snapshots of the Gaussian Dataset

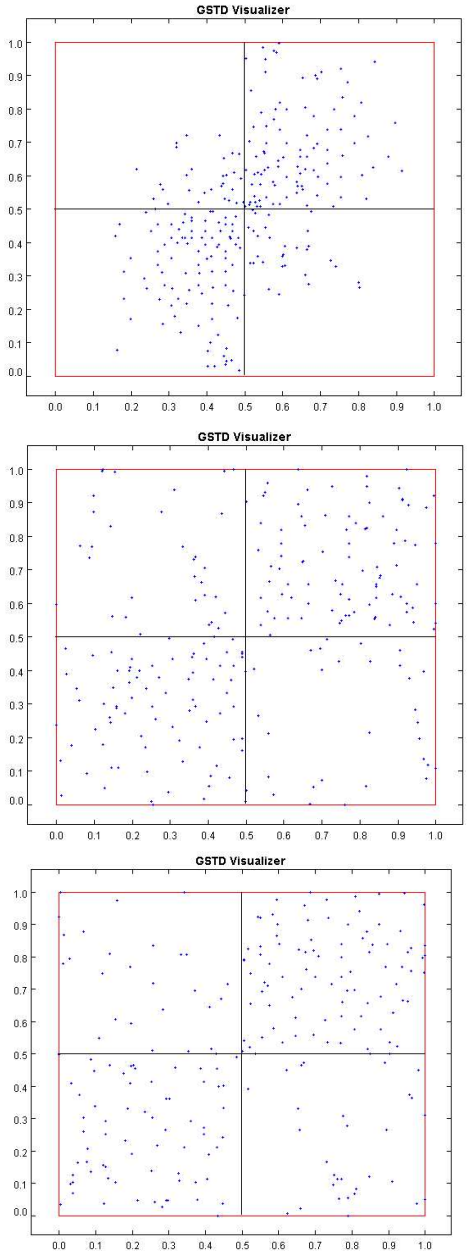


Figure 27: Snapshots of the Skewed Dataset

N	N_p^*	# of Grid Cells
1,000,000	10	100
3,000,000	16	256
6,000,000	20	400

Table 5: N_p^* and the corresponding Number of Grid Cells used in the `ST_SPIT` table for given dataset sizes.

cesses are measured according to the average number of "physical reads" reported by Oracle's query trace statistics report (refer to Oracle's `set autot` command for more details).

All experiments were carried out on a 4-processor IBM p690 system using Oracle 9.2i Enterprise Edition. (More details about the system used can be found on-line at *http://www.cs.ualberta.ca/hiso*.)

6.1 Cost Model Evaluation

We experimentally confirm the reliability of our cost model by reporting the performance of SPIT at grid sizes set below, at, and above the optimum grid size determined by the cost model. We hypothesize that if the performance of SPIT is best at the grid size analytically determined by the model to be optimum then the model is reliable.

For sake of clarity, we repeat Equation 6

$$N_g^* = (\frac{N \times q_t}{3q \times BS})^{2/3}$$

which calculates the optimal total number of cells in the grid given the size of the dataset (N), the query size in the temporal dimension (q_t), the query size in the spatial dimension (q) and the block size (BS) representing the number of tuples stored in a page on disk. The size of our datasets are known a priori. The query extents in the spatial and temporal dimensions are equal; therefore N_g^* can be calculated irregardless of query size. In order to estimate the block size we assume a page size on disk of 8192 bytes and a tuple size of 4 bytes per column. The `ST_SPIT` table consists of 6 columns, one for the object identifier, two for the spatial coordinates, two for the temporal interval, and one for the partition number. The block size can thus be calculated as:

$$BS = \frac{8192}{4 \times 6}\ tuples/block.$$

The SPIT algorithms require the number of partitions in one dimension (N_p^*) as a parameter. The value of N_p^* can be calculated in terms of N_g^* using Equation 7, which we repeat:

$$N_p^* = \lceil \sqrt{N_g^*} \rceil$$

The value of N_p^* determines the optimal number of grid cells (partitions) used when creating the `ST_SPIT` table for each dataset. These values are reported in Table 5. The methodology behind the following experiments is to vary the number of grid cells used by the `ST_SPIT` table and to measure average query performance over 100 randomly generated queries. If the best performance for each dataset occurs on the `ST_SPIT` table with the optimally determined number of grid cells then the model is reliable.

Figure 28 plots the number of grid cells used by the SPIT model against both the number of disk accesses and query time reported by Oracle for the three query sizes over the Gaussian Dataset. The shape of the curve reflects the trade-off between adding more partitions so as to benefit from partition elimination and the extra per partition cost of performing a local index range search of the data. The query performance times curves given in Figure 28 plot the time per query against the number of grid cells. As the plot shows, a strong correlation between time/query and disk accesses clearly exists. Although the model is only guaranteed to find the number of grid cells such that disk accesses are minimized, that point is also very likely to provide (near) optimum query performance in terms of time. The time/query results show that number of partitions can have a strong impact on real-time query performance.

For the 6 million tuple dataset, the empirical minimum number of disk accesses at all query sizes occurs at 400 cells, which is precisely the number of cells SPIT's cost model suggests we use. The results for the 1.5 million and 3 million tuple datasets are equally encouraging. For the 3 million tuple dataset the fastest query performance time occurs using 256 cells, which is the number of cells the model recommended we use. For the 1.5 million tuple dataset as well, the best query performance occurs at 100 grid cells – the grid size analytically determined by the model to minimize disk accesses.

The experiments confirm the model's ability to find the optimal number of grid cells irregardless of query size in the situation where the query extents in the temporal and spatial components are identical. The query size does not strongly impact the optimum performance point. For all datasets, as the number of partitions increases beyond the optimum, there is an increasing overhead due to the cost of accessing more partitions. This is even more pronounced for larger query sizes, which cover a larger number of partitions. When the number of partitions

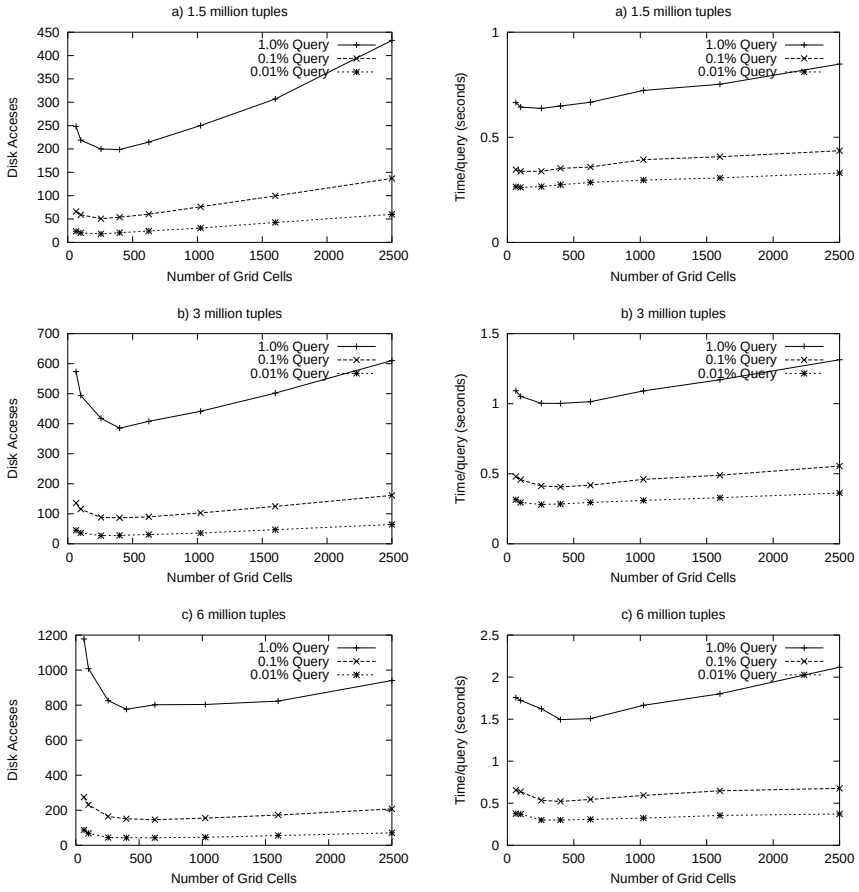


Figure 28: Disk Accesses and Query Processing Time – Gaussian Dataset

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 we hypothesized, the model is reliable because the analytical number of grid cells corresponds with the empirically reported best performance point.

The experimental results using the Skewed dataset are shown in Figure 29. The results confirm that the cost model is still reliable at determining an optimal number of grid cells to use given that the assumption of uniform data density does not hold. The ideal grid size for the Skewed data is slightly higher than what the model predicts because queries in areas of high data density benefit from the finer grid present when the number of grid cells is larger than the optimum. The cost model slightly underestimates the ideal grid size given the Skewed data distribution.

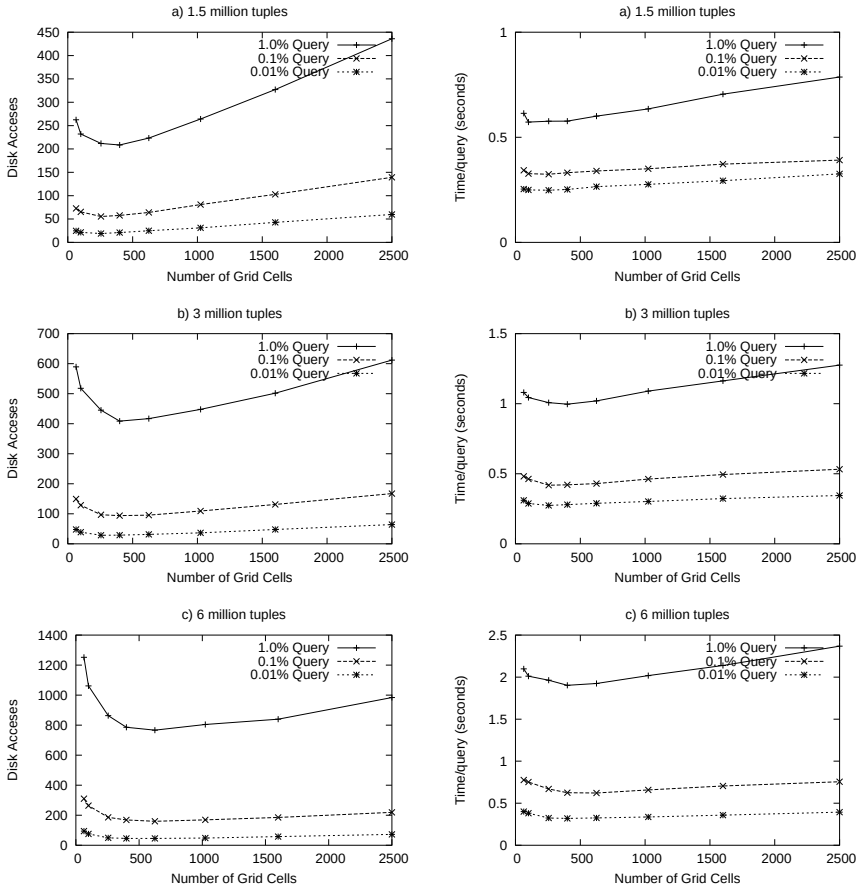


Figure 29: Disk Accesses and Query Processing Time – Skewed Dataset

Recall that N_p^* is determined for a given set of parameters, including N. As the lifespan of the database increases N is bound to increase, therefore the optimal value N_p^* should grow (sub-linearly with N) as well. Therefore a value of N_p^* determined for a given N is bound to be sub-optimal at some point in time. Fortunately, the experiments show that SPIT is fairly resilient to the growth of N especially for smaller query sizes. For instance, the index performance for a database of 6 million tuples using a value of N_g^* determined for 3 million tuples instead, is not too far from the optimal performance. Nevertheless, for a very large increase of N, performance can deteriorate, suggesting that N_g^* should be periodically recomputed and, if necessary, the index rebuilt.

Note that related work suggesting the use of a grid for spatio-temporal index-

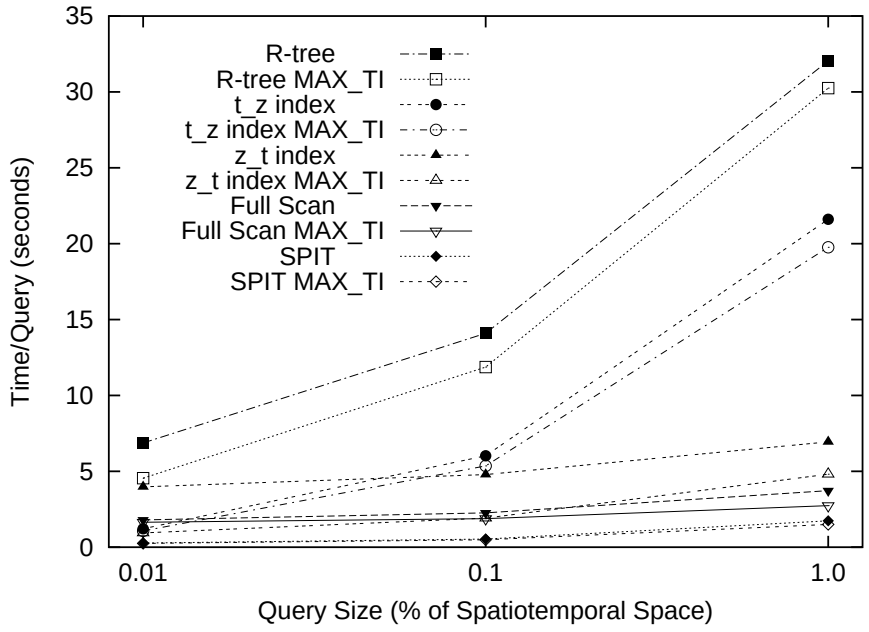


Figure 30: Query Performance on the 6 million tuple dataset with the maximum temporal interval (*MAX_TI*) assumption and without the assumption

ing, i.e., SETI [8], does not provide any means of tuning the number of grid cells beyond experimental trial and error. As our experiments have shown, the number of grid cells is a crucial parameter in determining the performance of a grid-based indexing approach. The ability to reliably determine the optimal number of partitions in an analytical manner is a major contribution of this work and is very important in making SPIT a practical solution for RDBMS spatio-temporal support.

6.2 Performance Evaluation

Next we compare the performance of SPIT against the other approaches for RDBMS-based spatio-temporal support described in Sections 5.2 and 5.3, denoted as the *R-tree*, *t_z index*, and *z_t index* approaches, respectively. The implementation of SPIT uses the optimal grid size set according to Table 5. In all experiments we use as a baseline a *Full Scan* of the data, i.e., no index support. Indeed, in many cases all approaches perform worse than a full scan of the data; SPIT, on the other hand, never did.

We first establish the benefit of using the maximum temporal interval assump-

Table	Index	Time	Gaussian Dataset			Skewed Dataset		
			1.5 M	3 M	6 M	1.5 M	3 M	6 M
ST_ZORDER	*t_z*	insert:	0:14	0:24	0:35	0:14	0:23	0:41
	index	index:	1:07	2:18	4:39	1:04	2:50	4:30
		total:	**1:13**	**2:42**	**5:14**	**1:18**	**3:13**	**5:11**
ST_ZORDER	*z_t*	insert:	0:14	0:24	0:35	0:14	0:23	0:41
	index	index:	1:07	2:16	4:37	1:07	3:02	4:30
		total:	**1:21**	**2:40**	**5:12**	**1:21**	**3:25**	**5:11**
ST_RTREE	*R-tree*	insert:	0:15	0:26	0:53	0:52	1:23	2:00
		index:	4:41	10:21	25:01	4:38	19:39	44:01
		total:	**4:56**	**10:47**	**25:54**	**5:30**	**22:02**	**46:01**
ST_SPIT	*B-tree*	insert:	1:05	2:24	4:04	1:08	2:23	5:18
		index:	0:17	0:29	1:32	0:22	0:46	1:40
		total:	**1:23**	**2:53**	**5:36**	**1:30**	**3:09**	**6:58**

Table 6: Data Insertion and Index Creation Time in min:sec

tion at query time. Figure 30 shows query performance of all approaches both with the maximum interval assumption and without the assumption for the 6 million tuple dataset. As expected, the assumption benefits all approaches because the temporal filter takes advantage of the tighter bound. Figure 30 establishes that the SPIT approach outperforms all other approaches both with and without the maximum interval assumption. For all further experiments, we report query performance using the maximum interval assumption and assume that the value of MAX_TI is known at query time.

Table 6 reports the time needed to insert and index the datasets for the four approaches used for comparison, assuming that the data is already sorted by $\langle t_s, t_e \rangle$. The insert time for the ST_SPIT table is higher than that for the ST_RTREE because of the overhead involved in calculating the pid of each tuple and for Oracle to lookup and write the tuple to the appropriate partition. However, the index creation time for the *R-tree* approach is prohibitively expensive. The indexing time for the ST_ZORDER is high because the Z-value calculation must be performed at index time. As the results show, creating the local partitioned indexes for SPIT does not incur a major overhead. In fact, SPIT's data insertion and index creation time is 5 times faster than the *R-tree* approach. Table 6 also provides the insertion and indexing times for the Skewed Dataset. Typically, insertion and index times tend to take longer for the Skewed dataset as compared to the Gaussian; however the relative order between the approaches is the same.

We now report the performance results of all approaches over the 100 ran-

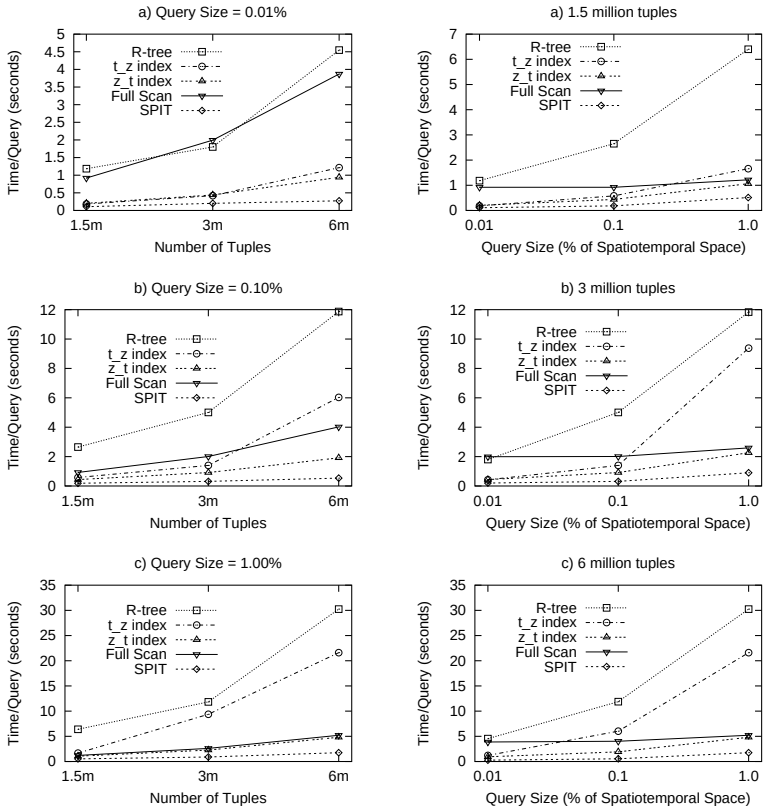


Figure 31: Query Performance for different database and query sizes – Gaussian Dataset

domly generated queries. The left-hand side of Figure 31 plots the performance of all approaches as the database scales between 1.5 million to 6 million tuples using a fixed query size. The right-hand side plots, on the other hand, show the results in terms of varying query size with a fixed database size. The *R-tree* approach has consistently poor performance and is outperformed by a sequential scan of the data in all cases. The *t_z index* approach also performs poorly, for nearly every test it too is outperformed by a sequential scan. The problem with a primary index on time is that there is an enormous amount of overlap among the temporal intervals of tuples because data about every moving object in the system is being continuously reported. As expected, the performance of the full scan of the data remains relatively constant throughout. Part of the reason that a full table scan can be difficult to outperform, especially as the query size increases, is that a full scan of the table

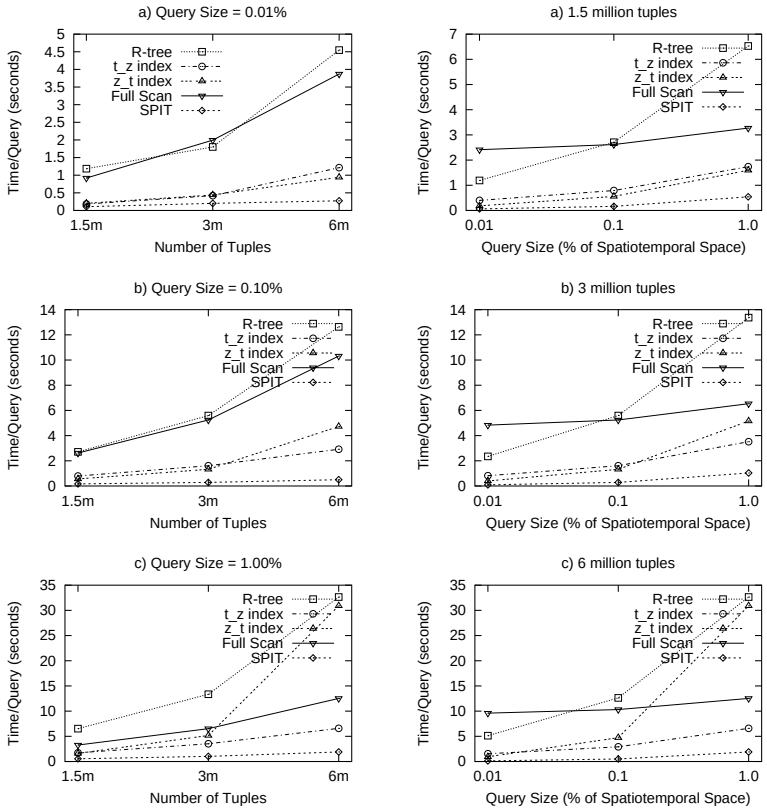


Figure 32: Query Performance for different database and query sizes – Skewed Dataset

requires very few seek operations on disk – data is scanned and filtered in continuous blocks. The *z_t index* approach, because of its efficient spatial discrimination, performs relatively well in all cases, although it is outperformed by the full table scan for the largest query size (1%). On average, SPIT outperforms the *z_t index* by a factor of three. SPIT is the only option to consistently outperform all other approaches (including a full table scan) in all tests.

The SPIT approach scales extremely well. In the worst case, i.e., the largest test query and database size, SPIT's query performance was still under 2 seconds. The *t_z index* approach, in the worst case, took 21.6 seconds to answer a query. For the *z_t index* approach worst case performance was 4.8 seconds. The *R-tree* approach, in the worst case, took 30.2 seconds to answer a query. The key advantage of SPIT lies in the performance advantage of partition elimination. Only those partitions

that intersect the spatial component of the query window are scanned. The local temporal index on each partition reduces the number of tuples read from disk. The ability to optimize the number of partitions further enhances performance. The R-tree, a general purpose spatial data structure, can not approach the performance advantage received from partition elimination. For the 3 million dataset and 0.1% query size, the *R-tree* approach takes 5 seconds whereas SPIT takes on average 0.3 seconds to answer a query – a speedup of 16 times.

Figure 32 show the performance of all approaches as database and query size varies on the Skewed dataset. On average, queries on the Skewed dataset take longer for all approaches as opposed to the Gaussian dataset. This can be partially attributed to the larger query answer set size for the Skewed dataset (queries in areas of high density return more tuples). For SPIT, query performance on the Skewed data degrades very little because queries in areas of low density return a small answer set extremely quickly, and queries that fall in areas of higher density are still serviced relatively quickly. Our experiments have shown that the SPIT approach offers the best overall query response time, scalability, and data loading performance on both the Gaussian and Skewed datasets. The experiments confirm that SPIT is an efficient approach for indexing spatio-temporal data inside the RDBMS.

7 Conclusions and Future Work

The need for built-in RDBMS support of complex data-types has long been acknowledged in the database community. Support for purely spatial or purely temporal data has been proposed, and a natural extension thereof is to develop a true Spatio-Temporal Access Method (STAM) with a relational mapping. The Space-Partitioning with Indexes on Time (SPIT) approach fills this crucial need by providing an efficient means to manage historical spatio-temporal data inside a RDBMS. SPIT uses a pipelined query mechanism wherein an initial coarse spatial partitioning based on a grid location is applied, followed by temporal filtering, spatial refinement and duplicate elimination steps. Excellent query performance is provided due to SPIT's ability to efficiently prune the search space so as to reduce the need to scan spatio-temporal records on disk. SPIT leverages existing RDBMS technology and has been shown to outperform other alternatives for spatio-temporal data management inside the RDBMS.

We have developed a disk access based cost model that can optimally choose the number of grid cells to use with SPIT. Empirical testing confirms that the model is reliable and that the SPIT approach offers excellent query performance as compared to other RDBMS-support alternatives. 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 tightly integrated RDBMS support has not been done before. Although a more traditional spatial index based approach, i.e., adapting an R-tree to the problem at hand, does not require a fixed grid-based partitioning, as we demonstrated experimentally our method dramatically outperforms the R-tree based approach. The coarse spatial partitioning combined with accurate temporal filter offer incomparable query performance to typical R-tree based approaches.

In [45], the authors identify three main requirements for indexing in spatio-temporal databases:

1. offer appropriate data types and query language support,
2. provide efficient indexing and retrieval methods, and
3. exploit cost models for query processing and optimization purposes.

Our method satisfies all three requirements. We provide a meaningful data type for tracking spatio-temporal objects as well as a mechanism for query support. Our indexing method, as shown experimentally, is very efficient largely due to the cost model we developed that optimizes the grid size used by SPIT.

7.1 Future Work

SPIT could be expanded in the future by adapting the model to take into account other types of spatio-temporal data models, i.e., the trajectory or parametric models. One possible approach that could be taken up would be to insert a data point into each cell where a trajectory (or the future parametric path) of an object follows. Such an approach would require a more complex insertion and query method, and would incur an extra storage expense. Given the already excellent performance of SPIT, however, we predict that such an approach would still outperform alternative methods.

Another opportunity for future work would be optimizing SPIT through the adaptation of a non-uniform spatial grid. Query performance could be further enhanced by making SPIT aware of areas of higher and lower density in the dataset. A denser data space may benefit from a finer grid, whereas in areas of low data density a coarse grid may be adequate. This would help make SPIT more scalable and adaptable to skewed data distributions.

As well, we are investigating manners in which SPIT's partitioning could be periodically re-adjusted as the database size increases. Even though the 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 useful to investigate whether a self-adaptation scheme, where the RDBMS would re-configure the partition 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.

References

[1] Mahdi Abdelguerfi , Julie Givaudan, Kevin Shaw, and Roy Ladner. The 2-3TR-tree, a Trajectory-Oriented Index Structure for Fully Evolving Valid-Time Spatio-Temporal Datasets. In *Proc. of ACM GIS*, pages 29–34, 2002.

[2] Tamas Abraham and John F. Roddick. Survey of Spatio-Temporal Databases. *Geoinformatica*, 3(1):61–99, 1999.

[3] David W. Adler. IBM DB2 Spatial Extender - Spatial Data within the RDBMS. In *Proc. of VLDB*, pages 687–690, 2001.

[4] Rudolf Bayer and Edward M. McCreight. Organization and Maintenance of Large Ordered Indices. *Acta Informatica*, 1:173–189, 1972.

[5] N. Beckman, H.P. Krigel, R. Schneider, and B. Seeger. The R*-tree: An Effi cient and Robust Access Method for Points and Rectangles. In *Proc. of ACM SIGMOD*, pages 332–331, 1990.

[6] R. Benetis, C. Jensen, G. Karciauskas, and S. Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *IDEAS*, pages 44–53, 2002.

[7] Thomas Brinkhoff. A Framework for Generating Network-Based Moving Objects. *Geoinformatica*, 6(2):153–180, 2002.

[8] V. Prasad Chakka, Adam C. Everspaugh, and Jignesh M. Patel. Indexing Large Trajectory Data Sets With SETI. In *Proc. of CIDR [Online: http://www-db.cs.wisc.edu/cidr/program/p15.pdf]*, 2003.

[9] Hae Don Chon, Divyakant Agrawal, and Amr El Abbadi. Range and kNN Query Processing for Moving Objects in Grid Model. *Mobile Networks and Applications*, 8(4):401–412, 2003.

[10] Federal Communications Commission. FCC: Enhanced 911. *[Online: http://www.fcc.gov/911/enhanced/]*, 2004.

[11] Martin Erwig, Ralf Hartmut Guting, Markus Schneider, and Michalis Vazirgiannis. Spatio-Temporal Data Types: An Approach to Modeling and Querying Moving Objects in Databases. *GeoInformatica*, 3(3):269–296, 1999.

[12] Michael Freeston. The BANG File: A New Kind of Grid File. In *Proc. of ACM SIGMOD*, pages 260–269, 1987.

[13] Johann Christoph Freytag, M. Flasza, and Michael Stillger. Implementing Geospatial Operations in an Object-Relational Database System. In *Proc. of SSDBM*, pages 209–219, 2000.

[14] Antonin Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. of ACM SIGMOD*, pages 47–57, 1984.

[15] Rittwik Jana, Theodore Johnson, S. Muthukrishnan, and Andrea Vitaletti. Location Based Services in a Wireless WAN Using Cellular Digital Packet Data (CDPD). In *Proc. of ACM MobiDE*, pages 74–80, 2001.

[16] Eija Kaasinen. User Needs for Location-Aware Mobile Services. *Personal Ubiquitous Computing*, 7(1):70–79, 2003.

[17] Ravi Kanth V Kothuri and Siva Ravada. Spatio-Temporal Indexing in Oracle: Issues and Challenges. *IEEE TCDE Bulletin*, 25(2):56–60, 2002.

[18] Ravi Kanth V Kothuri, Siva Ravada, and Daniel Abugov. Quadtree and R-tree Indexes in Oracle Spatial: A Comparison using GIS Data. In *Proc. of ACM SIGMOD*, pages 546–557, 2002.

[19] Hans-Peter Kriegel, Marco Pötke, and Thomas Seidl. Managing Intervals Effi ciently in Object-Relational Databases. In *Proc. of VLDB*, pages 407–418, 2000.

[20] David Lagesse. They know where you are. *U.S. News*, pages 23–36, 38, September 2003.

[21] Sitaram Lanka and Eric Mays. Fully Persistent B+-trees. In *Proc. of ACM SIGMOD*, pages 426–435, 1991.

[22] Philip M. Lewis, Arthur Berstein, and Michael Kifer. *Database and Transaction Processing*. Addison-Wesley, 2002.

[23] Mohamed F. Mokbel and Walid G. Aref. On Query Processing and Optimality Using Spectral Locality-Preserving Mappings. In *Proc. of SSTD*, pages 102–121, 2003.

[24] Mohamed F. Mokbel, Walid G. Aref, and Ibrahim Kamel. Performance of Multi-Dimensional Space-Filling Curves. In *Proc of ACM GIS*, pages 149–154, 2002.

[25] Mohamed F. Mokbel, Thanaa M. Ghanem, and Walid G. Aref. Spatio-Temporal Access Methods. *IEEE TCDE Bulletin*, 26(2):40–49, 2003.

[26] Bongki Moon, H. V. Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TKDE*, 13(1):124–141, 2001.

[27] MySQL AB. *MySQL Manual — Spatial Extentions in MySQL [Online: http://www.mysql.com/doc/en/Spatial_extensions_in_MySQL.html]*, 2002.

[28] Mario A. Nascimento nad Jefferson R. O. Silva and Y. Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *Proc. of STDBM*, pages 171–188, 1999.

[29] Mirco Nanni. Distances for Spatio-temporal clustering. In *Proc. of SEBD*, pages 135–142, 2002.

[30] Mario A. Nascimento and M. Dunham. Indexing Valid Time Databases via B+ -trees – the MAP21 Approach. *IEEE TKDE*, 11(6):1–19, 1999.

[31] Mario A. Nascimento, Dieter Profser, and Yannis Theodoridis. Synthetic and Real Spatiotemporal Datasets. *IEEE TCDE Bulletin*, 26(1):26–32, 2003.

[32] Mario A. Nascimento and Jefferson R. O. Silva. Towards Historical R-trees. In *Proc. ACM SAC*, pages 235–240, 1998.

[33] J. Nievergelt, H. Hinterberger, and K.C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM TODS*, 9(1):38–71, 1984.

[34] Beng Chin Ooi, Ron Sacks-Davis, and Ken J. McDonell. Extending a DBMS for Geographic Applications. In *Proc. of IEEE ICDE*, pages 590–597, 1989.

[35] Oracle Corporation. *Oracle Spatial User's Guide and Reference, Release 9.2*, 2002.

[36] Dieter Pfoser. Indexing the Trajectories of Moving Objects. *IEEE TCDE Bulletin*, 25(2):3–9, 2002.

[37] Dieter Pfoser and Christian S. Jensen. Querying the Trajectories of On-Line Mobile Objects. In *Proc. of ACM MobiDE*, pages 66–73, 2001.

[38] Dieter Pfoser and Christian S. Jensen. Indexing of Network Constrained Moving Objects. In *Proc. of ACM GIS*, pages 25–32, 2003.

[39] Dieter Pfoser, Christian S. Jensen, and Yannis Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. of VLDB*, pages 395–406, 2000.

[40] J. Roddick and B. Lees. *Geographic Data Mining and Knowledge Discovery*, chapter Paradigms for Spatial and Spatio-Temporal Data Mining, pages 33–50. Taylor and Francis, 2001.

[41] Simonas Saltenis, Christian S. Jensen, Scott T. Leutenegger, and Mario A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. of ACM SIGMOD*, pages 331–342, 2000.

[42] Hanan Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Comput. Surveys*, 16(2):187–260, 1984.

[43] Yufei Tao, Dimitris Papadias, and Jimeng Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proc. of VLDB*, pages 790–801, 2003.

[44] Yannis Theodoridis and Timos Sellis. A Model for the Prediction of R-tree Performance. In *Proc. of PODS*, pages 161–171, 1996.

[45] Yannis Theodoridis, Timos K. Sellis, Apostolos Papadopoulos, and Yannis Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In *Proc. of SSDBM*, pages 123–132, 1998.

[46] Yannis Theodoridis, Jefferson R. O. Silva, and Mario A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. of SSD*, pages 147–164, 1999.

[47] Yannis Theodoridis, Michalis Vazirgiannis, and Timos K. Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of IEEE ICMCS*, pages 441–448, 1996.

[48] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. On the Generation of Time-Evolving Regional Data. *Geoinformatica*, 6(3):207–231, 2002.

[49] Theodoros Tzouramanis, Michael Vassilakopoulos, and Yannis Manolopoulos. Overlapping Linear Quadtrees: a Spatio-Temporal Access Method. In *Proc. of ACM GIS*, pages 1–7, 1998.

[50] Ouri Wolfson, Bo Xu, and Sam Chamberlain. Location Prediction and Queries for Tracking Moving Objects. In *Proc. of ICDE*, pages 687–688, 2000.

[51] Yuni Xia and Sunil Prabhakar. Q+Rtree: Efficient Indexing for Moving Object Databases. In *Proc. of DASFAA*, pages 175–182, 2003.