University of Alberta

A Practical Indexing Technique for Spatio-Temporal Data

by

Viorica Botea ©

A thesis submitted to the Faculty of Graduate Studies and Research in partial fulfillment of the requirements for the degree of **Master of Science**.

Department of Computing Science

Edmonton, Alberta
Spring 2006

Your file Votre référence
ISBN: 0-494-13797-5
Our file Notre référence
ISBN: 0-494-13797-5

Abstract

Despite pressing need, current RDBMS support for spatio-temporal data is limited, and most existing spatio-temporal indexes cannot be readily integrated into existing RDBMSs. This thesis proposes SPIT$^+$, an indexing technique for historical spatio-temporal data, fully integrable in existing RDBMSs, and presents algorithms for processing typical spatio-temporal window and k-nearest neighbors queries. SPIT$^+$ separates the temporal and spatial components of data. A formal cost model and a partitioning strategy provide optimal space partitioning for uniformly distributed data and a heuristic partitioning leading to a very good query performance for arbitrary data distributions. The temporal layer's performance is improved if an optimal maximal temporal range is enforced, and a procedure to determine such an optimal value is presented. Extensive experiments show that SPIT$^+$ outperformes other RDBMS-based options by orders of magnitude, and is competitive to the MV3R-tree, with the unarguable advantage that it can be used on top of virtually any RDBMS.

Acknowledgements

I would like to express my sincere gratitude to my supervisor, Dr. Mario Nascimento, whose dedication in supporting me has well exceeded his professional duties. He agreed to work with me before I enrolled into this Master's program and has been a great help ever since. His assistance with program admission and subsequent research guidance have been invaluable.

I am also thankful to Dr. Jörg Sander, who played an important role in the development of this research by contributing with insightful comments and ideas. In addition, I would like to thank Dr. Arturo Sanchez-Azofeifa, the other committee member, for finding time to read this work and provide feedback.

My special thanks and appreciation go to my husband Adi, for being by my side during the entire time of this program. I wouldn't have been able to finish this thesis without his constant love and encouragement. Last by not least, I am very grateful to my parents and sister, for their unconditional and loving support. Thank you Flory, mom and dad.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Spatio-temporal data models the evolution in time of spatial objects, where a spatial object can be described as an entity that has associated a position in space. A spatio-temporal data object is thus characterized by a spatial attribute and a temporal attribute, describing spatial properties and the time period when the spatial properties were valid. Databases that store spatio-temporal objects are called *spatio-temporal databases.*

Consider the following scenario, where spatio-temporal data exist and can be used to answer practical questions. Assume that every cab in a city is capable of transmitting information to the headquarter every five minutes. The information transmitted contains the unique identifier of the cab, its current position represented by its spatial coordinates and the time the information is transmitted. Furthermore, assume that the information transmitted by all cabs is stored in a central database. As a first example of how this database can be useful, assume that a customer calls in and requests a cab at a given address. Instead of sending the customer request to every cab, the dispatcher can interrogate the database to identify the available cab that is the closest to that address. As a second example, assume that another customer forgot a piece of luggage in a cab that took him home two days ago. Rather than checking within all cabs in the city, the database could be used to identify only those cabs that were in the neighborhood of the customer house two days ago.

Spatio-temporal databases have received a great deal of research interest over the last years, mainly because the advances in mobile technology made possible the collection of large amounts of spatio-temporal data. Spatio-temporal data is generated in many real-life applications, the cab-tracking system mentioned above being just an example. Other applications include weather forecast prediction, traffic monitoring, spatio-temporal data mining, wildlife tracking, etc. For most of these applications, the data volume is usually huge, and data retrieval should work in real time. Standard models and techniques (e.g., indexing) used in "classical" databases are not efficient for spatio-temporal data. Hence developing models and indexing structures that work well for the latter type of data is very important. This is also the main focus of this thesis.

1.1 Problem Definition

As pointed out, many real-life applications, need to be modeled using spatio-temporal data. For some of these applications only the past locations of the moving objects are of interest, while for others it is more important to predict the objects future locations. This led to classifying the spatio-temporal data into historical and current/predictive [17]. This thesis addresses the issue of indexing and querying historical spatio-temporal data.

A typical way of representing historical spatio-temporal data is to model the objects' movement as discrete events in time. This kind of representation is also used here. Each mobile object has assigned an unique identifier and is associated with multiple database records. Each database record has the format $\langle oid, x, y, t_s, t_e \rangle$, where *oid* is the object's identifier, (x, y) represent the object's position along the spatial dimensions, and $[t_s, t_e)$ denotes the non-null time interval during which the object was located at position (x, y). The time intervals of all records having the same object identifier are disjoint and their union covers the whole temporal domain. For a given object o_k, two consecutive observations have the form $\langle o_k, x', y', t'_s, t'_e \rangle$, and $\langle o_k, x'', y'', t''_s, t''_e \rangle$, where $t''_s = t'_e$ and $(x', y') \neq (x'', y'')$. Between successive recorded positions, a stepwise interpolation is used; as long as the object's position is not updated in the database it is assumed to remain stationary at its last observed coordinates. This approach is preferred over the alternative of linear interpolating between two consecutive positions, as the later could result in erroneous assumptions. For instance, in the cabs example above, it would be wrong to assume that a cab followed a linear trajectory as obstacles/constraints (buildings, playgrounds, one-way streets, etc.) may be present.

There are two types of spatio-temporal queries that are considered in this thesis, namely *window* and k *nearest neighbors* queries. The following definitions are used for these types of queries:

- A spatio-temporal window query Q_W has the format $\langle \sigma, \tau \rangle$, where σ is a two-dimensional spatial region and τ is a time interval. The query answer consists of the set of unique *oid*'s of all records having (x, y) inside σ and $[t_s, t_e)$ overlapping with τ.

- A spatio-temporal kNN query Q_{NN} is defined as a tuple $\langle \rho, \tau, k \rangle$ with ρ being a static point in a two-dimensional space, τ specifying a time period and k being the number of requested neighbors. Q_{NN} returns the *oid*'s of k objects that were the closest to ρ during τ. The distance $dist(oid, \rho)$, between an object identified by *oid* and ρ, is defined as

$$dist(oid, \rho) = min_{(x,y) \in R_{oid,\tau}} \{d((x, y), \rho)\},$$

where $R_{oid,\tau} = \{(x, y) | \langle oid, x, y, t_s, t_e \rangle \in D \wedge [t_s, t_e) \cap \tau \neq \emptyset\}$, D is the set of all database records and $d((x, y), \rho)$ is the euclidian distance in the two-dimensional space.

1.2 Thesis Motivation and Contributions

Even though several spatio-temporal indexing structures have been developed, most of them suffer from the drawback that they cannot be easily integrated into existing RDBMSs. A spatio-temporal model is said to be fully integrable into an RDBMS if it can be implemented using only the standard functionality (e.g., data types, indexing structures) provided by that product. One important benefit of a fully integrable model is that it can take advantage of all features existing in an RDBMS, such as the capacity to manage large data volumes, concurrency control, query languages, and others. Furthermore, implementing a model on top of a standard product makes it accessible to more users and more application domains.

A first attempt to provide a spatio-temporal access method inside of an existing RDBMS has been made in [16], where the author proposed SPIT, a two-level indexing method fully integrated within the RDBMS via a relational mapping. It partitions the spatial dimension of the data space into a static grid and for each grid cell creates an index over the temporal dimension. A more detailed description of SPIT is provided in Section 2, where existing spatio-temporal indexing structures are reviewed.

The spatio-temporal indexing approach proposed in this thesis uses the same framework as SPIT does. However, it introduces several new and important features as detailed in the following:

- A new partitioning strategy is developed, that takes into account arbitrary data distribution. In contrast, SPIT was designed to work efficiently only for uniform data distributions.

- It introduces an algorithm for splitting the temporal ranges of data points with the goal of faster query response. An optimal decomposition, according with a new cost formula, is obtained with a histogram based method. No similar concern is present in SPIT.

- It shows how the presented indexing technique could be employed for solving new types of spatial-temporal queries such as kNN queries. Only spatio-temporal window queries were considered in SPIT.
- It provides a thorough experimental evaluation of the proposed method, for both real and synthetic datasets, by comparing it against other RDBMS-based alternatives for spatio-temporal data indexing, as well as a specialized spatio-temporal indexing structure.

1.3 Thesis Overview

The structure of the remaining chapters is the following: Chapter 2 reviews background concepts related to spatio-temporal databases domain and surveys spatio-temporal access methods present in the literature. An overview of several existing approaches focusing on answering spatio-temporal k nearest neighbors queries is also provided. Chapter 3 describes the proposed technique for indexing spatio-temporal data. A partitioning-based approach, the indexing method uses several decomposition algorithms designed to optimize the query processing cost. Besides in-depth description and discussion of query processing, this chapter includes the algorithms in pseudo-code and details about an actual implementation in an RDBMS system. An experimental evaluation of the proposed approach is presented in Chapter 4. Two major classes of experiments are performed: (1) model properties assessment and validation and (2) comparison against other spatio-temporal indexing approaches. The focus of the first class is to confirm the effectiveness of the decomposition strategies and to evaluate the model robustness with respect to its parameters.

The model efficiency is demonstrated with the second class of experiments, where it is shown that the proposed technique outperforms all compared approaches in answering both window and kNN spatio-temporal queries. Chapter 5 provides the conclusion of this thesis and offers directions for further research.

Chapter 2

Background and Literature Review

Representing spatio-temporal data inside a database can be done in several ways. A first choice is to consider the time as an additional dimension and to model the moving objects as points in a multidimensional space [1, 16, 19, 31]. In this way, the movement of an object in time could be represented by the object's spatial coordinates at each moment of time. If the object changes in shape and/or size over time and this change is relevant for the application's specifics, then the object is modeled as a multidimensional region and its spatial-extensions are stored in addition to its spatial coordinates. Even though object motion often is continuous in time, recording the position of an object for each time instance is impossible. Furthermore, for objects that do not move over a time period, this will result in storing redundant information. For this reason, only sampled positions, obtained by discretizing the whole time period, are stored. Figure 2.1 shows an example of time sampling for objects modeled as regions. For a query that falls between two neighboring sampled points, the location of the objects could be determined using, for instance, linear or step-wise interpolation.

A second possibility is to store the objects trajectories rather than their punctual locations. A trajectory is stored as a collection of linear segments,

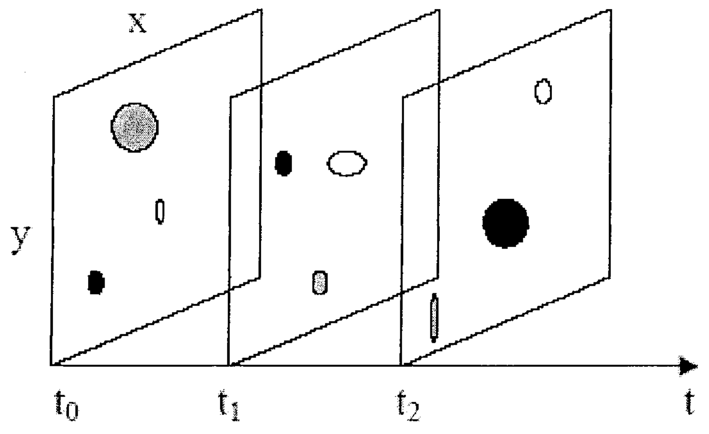


Figure 2.1: Spatio-Temporal Data Represented as Regions at Different Time Instances

where each line segment represents the motion of an object between two successive sampled positions. This type of data representation has been used in [5, 20, 22, 38]. While it offers a convenient way for retrieving all the objects satisfying a given spatial predicate at any moment of time without need for interpolation, a trajectory model usually requires more complex data and indexing structures than those employed by the previous model.

A third data model that can be used to represent spatio-temporal data is the parametric model [11, 26, 33]. Under this approach, an object's movement is represented as a function of time, and its motion (typically speed and direction) vector is stored together with its current spatial coordinates. By modeling the path followed by a moving object as a time variable information, a parametric model has the advantage that data updates are necessary only if the components of the motion vectors change. In addition, it allows to compute the objects future location, which makes it suitable for predictive spatio-temporal applications.

2.1 Spatio-Temporal Queries

The most common types of spatio-temporal queries are those inherited from the area of spatial databases. These include window queries, nearest neighbor, k-nearest neighbor [39], reverse nearest neighbor [2] and join queries. Spatio-temporal *window* queries retrieve all the objects that are contained in a specified query region during a given time interval or at a time moment. "*Find all the cars that were parked at the university between 5 p.m. and 6 p.m. two days ago*" could be an example of window query. Given the coordinates of a static or moving object (query point), a spatio-temporal *nearest neighbor* query retrieves the object that is the closest to the query point during a time interval, with the distance between two objects during a specified time interval being defined as the minimum of their distances at all time instances that belong to the specified time interval. An example of such a query would be "*which bus will be the closest to the bus station in the next five minutes*". A k-nearest neighbor query is an extension of a nearest neighbor query, where instead of returning only one object, the k closest objects are retrieved. In contrast, a *reverse nearest neighbor* query returns all the objects whose nearest neighbor is the query point. As opposed to the query types defined before, which are concerned with only one dataset, a spatio-temporal *join* query involves two sets of data and focuses on finding all the pairs of objects from the cartesian product of the datasets, that are located within a given distance from each other during a time interval.

Since the location (past, current or future) of the moving objects during the query time interval is the only information needed to compute a query answer set, all the query types defined above are often called *coordinate-based* [22] spatio-temporal queries. Recently, a new category a spatio-temporal queries, namely the *trajectory-based* queries [22], has emerged. Trajectory-based queries require information about the complete or partial trajectories of the moving objects and concentrate on answering questions regarding the topological and navigational aspects of objects' movements. Providing answers to questions such as whether a trajectory enters, leaves, stays within, crosses or bypasses a certain area during a given time interval is the focus of topological queries, while navigational queries involve derived information such as speed, heading, acceleration, traveled distance, etc. [22].

Using a complementary categorization, one can distinguish between historical and current/predictive spatio-temporal queries. Historical spatio-temporal queries are used in conjunction with historical databases and answer questions about the past movement of the objects. Current/predictive queries work with current/predictive spatio-temporal databases and are focused on returning information about the current state of the moving objects or about their projected movement. In order to answer questions about the objects location at a specified time moment in the future, information about their current velocities and positions is used. However, the answer to a predictive query is deemed accurate only at the time it is computed; it may be invalided with any update that happens between the current and the query times. To overcome this limitation, continuous spatial-temporal queries [28] have been introduced. Such queries involve constant monitoring of the database status and updating of the query answer with every occurring change.

2.2 Indexing Spatio-Temporal Data

Most of the indexing structures designed to handle spatio-temporal data are based on R-trees [7]. An R-tree is a balanced tree that indexes spatial objects based on their Minimum Bounding Rectangle (MBR). The leaf nodes contain the MBRs of all spatial objects and pointers to objects exact representations. At the directory level, each node consists of the smallest MBR that tightly encloses all the MBRs in the child nodes and of pointers to the child nodes. A sample two-dimensional R-tree is provided in Figure 2.2. The objects' MBRs are depicted by the rectangles R8–R19. Multiple MBRs are grouped together in the parent nodes R1–R7, in such a way that node's overlapping and the empty space introduced in the tree are minimized. When a window query has to be answered, the tree is traversed starting from the root to determine all the leaf entries whose MBRs overlap with the query window. For these entries, the objects exact representations are retrieved to test whether they satisfy the query. Since the MBRs of several entries of a node may overlap with the query, multiple paths from the root to leaves may be traversed.

Like spatio-temporal queries, spatio-temporal access methods could be classified into historical and current/predictive.

First, historical spatio-temporal indexing structures are reviewed. For this class of indexing structures, spatio-temporal data is usually represented using either the coordinate-based [1, 19, 31] or trajectory-based [5, 20, 22, 38] model.

RT-trees [40] are the first access structure intended to support the storage and retrieval of spatio-temporal data. An RT-tree uses a two dimensional R-tree to index data's spatial component and stores the temporal component as auxiliary information in the R-tree's nodes. While being able to answer spatial queries relatively well, an RT-tree provides no discrimination along the temporal dimension, which makes it extremely inefficient for queries based solely on a temporal predicate.

Designed to support both time-related and space-related queries at the same time, the 3D R-tree [38] considers the time as a third spatial dimension and indexes spatio-temporal data in a three-dimensional R-tree. The main problem with the 3D R-tree is the growth of the dead space introduced in the index, resulting in a larger degree of overlap among nodes, which in turn causes lower query performance. Research on addressing the issue of dead space in

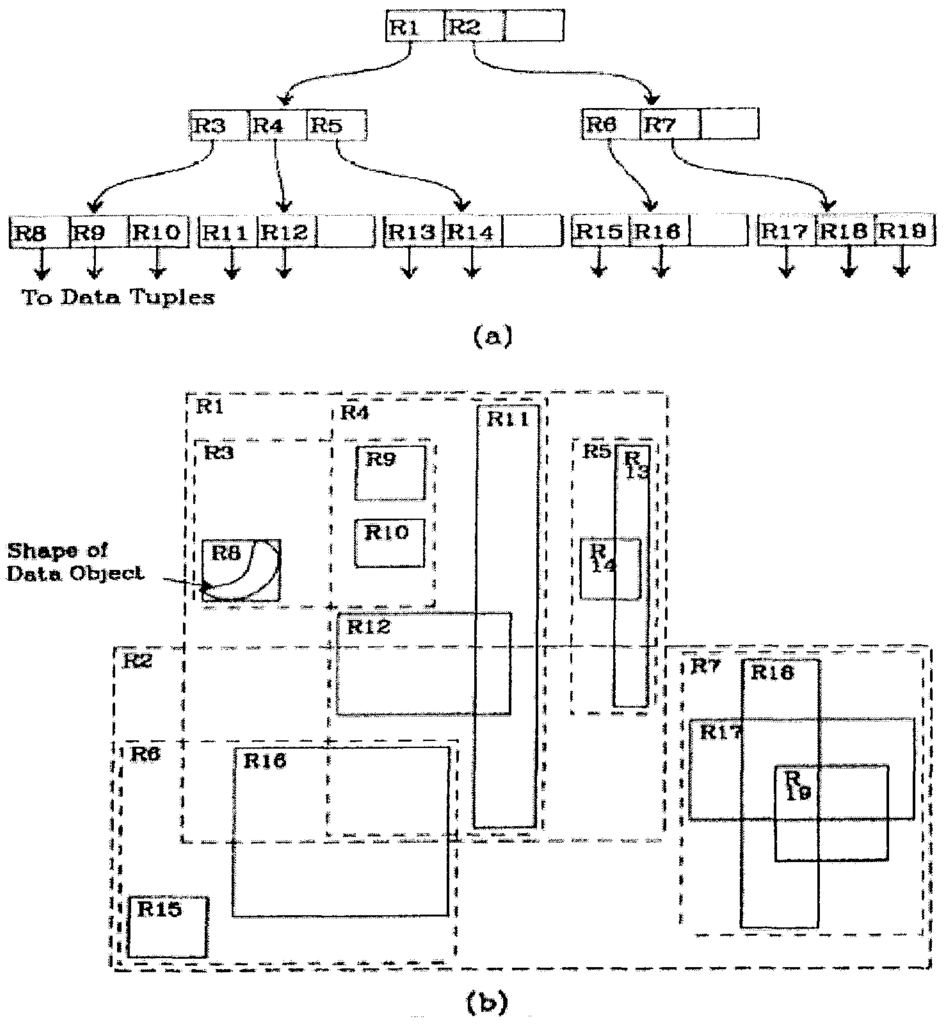


Figure 2.2: Example R-tree (from [7])

3D R-trees is reported in [8] and [24]. The common idea of these contributions is to split long trajectory segments, such that the total area of the indexed MBRs is reduced.

Historical R-trees [19] adapt the concept of overlapping B-trees [4] to spatio-temporal data. The main idea is to create a separate R-tree for each time instance in the history of the moving objects. To avoid duplication of nodes whose content doesn't change over consecutive timestamps and save disk space, common nodes are shared among multiple R-trees. If, instead, at least one node entry changes its location, then all the other entries have to be replicated. Very efficient for timestamp queries, a historical R-tree suffers from the drawback of having to search over multiple copies of the same objects in answering interval queries.

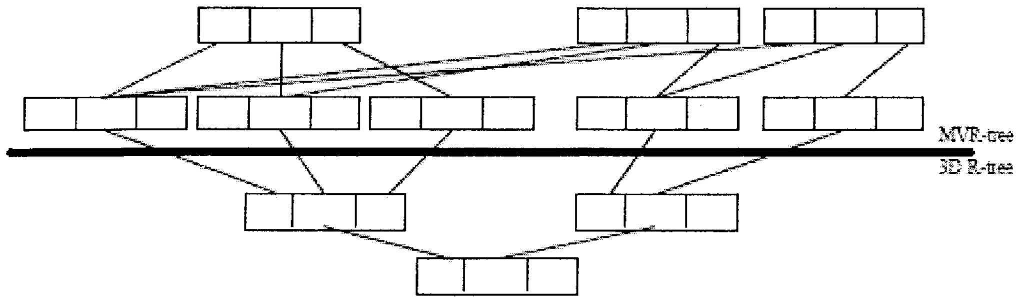


Figure 2.3: MV3R-tree (from [31])

The MV3R-tree [31] improves upon historical R-trees by providing more efficient support for interval queries. It consists of two correlated indexing structures: a *Multi-Version R-tree* (MVR-tree) and an auxiliary 3D R-tree built on MVR's leaf nodes, as shown in Figure 2.3. In answering a spatio-temporal query, either one or another structure is used: the MVR-tree offers good performance for time-slice queries, while the auxiliary 3D R-tree performs better for time-interval queries. As with any multi-version and overlapping structure, the replication of some information in an MVR-tree cannot be prevented. Object replication in an MVR-tree also induces extra space requirements for the 3D R-tree structure, making the entire structure less storage effective.

The Trajectory-bundle tree (TB-tree) [22], also a 3D R-tree data structure, aims at objects trajectory preservation, as explained below. It uses a modified R-tree insertion algorithm, such that each leaf node contains only line segments belonging to the same trajectory. To allow fast trajectory retrieval, leaf nodes containing information about the same object's trajectory are interconnected. The structure is very efficient for trajectory-based queries at the price of performance deterioration for coordinate-based queries.

SETI [5] offers an alternative solution to trajectory indexing by separating the temporal dimension from the spatial dimension. For the spatial dimension a static partitioning of the entire space is used. Within each partition cell, a sparse temporal R*-tree is created. Rather than indexing the time intervals of all segment trajectories that fall inside a cell, only one entry for each data page is maintained. When a trajectory segment intersects several partition cells, it is split into several segments, one for each cell. In order to facilitate updates, the last known locations of all objects are kept in an in-memory "front-line" structure. The performance of the SETI's indexing strategy depends to a great extent on the number of cells used to partition the space. The authors do not provide, however, any insights on how this parameter should be chosen.

Another method based on spatial decomposition is proposed by Song and Roussopoulos [30]. The space is partitioned into zones and each object's spatial location is represented only by the id of the zone that it belongs to, rather than the object's (x, y) coordinates. Inside each zone, a structure called a *SEB-tree* is used to index the objects, according to their start and end timestamps. The main drawback of this approach is that, when only part of a zone intersects with the query spatial range, the answer can be inaccurate, containing records from outside the query range.

Two related approaches capable of indexing both historical and current data are 2-3TR-trees [1] and 2+3TR-trees [20]. The common idea is to implement two R-trees: a two-dimensional tree for indexing current data and a three-dimensional one for historical data. Records whose time interval goes up to the current moment are stored in the two-dimensional tree. As soon as the time interval of such a record closes (i.e., the object changes its position), the record is transferred into the three-dimensional structure and a new record is inserted into the two-dimensional structure. The main difference between the two approaches lies in the nature of the data indexed on the three-dimensional

R-tree: multidimensional points in 2-3TR-tree and trajectories in 2+3TR-tree.

The previous indexing structures are suitable for retrieving information about the past or current location of moving objects. Next, structures capable of answering predictive spatio-temporal queries are reviewed.

A Time Parameterized R-tree (TPR-tree) [26] models future positions of moving objects as a linear function of time. It uses an R*-tree to store the current known location and velocity vector of objects. Each time when a change in an object motion pattern occurs, the record of the involved object is updated accordingly. The velocity vector of an internal node is determined by the velocity vectors of the enclosed objects such that the node's bounding rectangle will contain all this objects at any time in the future. Predictive queries are answered using the linear function and the currently known location of objects. TPR*-trees introduced in [33] are enhanced versions of TPR-trees. Operations such as insertion and deletion are performed differently than in standard R*-trees. This results in a different partitioning of the data inside the tree, with benefits reflected on the query response time.

The current state-of-the-art for predictive STAMs are the B^x-tree [11], which use a B^+-tree as their underlying structure. To use a B^+-tree efficiently, an ordering has to be defined for the involved data. In this method, the spatio-temporal data is ordered as follows. The spatial component is linearized with a space-filling curve, obtaining a spatial label for each object position. First, the temporal component is partitioned into several time intervals with unique ids, and each moving object is mapped to a time interval based on its update time. Next, the index value of an object is obtained by concatenating the id of its temporal interval and the object spatial label at t_{lab}, where t_{lab} is the end timestamp of the object's time interval. Since the B^x-tree indexes the position of objects only at certain moments of time, called *label timestamps*, queries having timestamps different from these label timestamps are handled by query-window enlargements.

Recently, the BB^x-index, an index structure based on B^x-trees, was proposed in [15]. By keeping not only one, but multiple trees with each tree having associated a lifespan, the BB^x-index is capable of managing past, present and future spatio-temporal queries at the same time.

Although most of the existing spatio-temporal indexing structures were designed to accommodate window queries, work that focuses on handling k nearest neighbor queries in the context of spatio-temporal data is also present in the literature. As mentioned before, a kNN spatio-temporal query retrieves the k objects from a dataset O that are the closest to a query object q, during a given interval of time. Depending on the dataset and the query object characteristics, three *scenarios* can be differentiated: 1) the query point is a moving object while the dataset is a collection of static objects, 2) the query point is a stationary object while the objects in the dataset are dynamic, and 3) both the query point and the dataset objects are moving objects.

One of the first works to consider answering kNN queries in the context of moving objects (query and data) in an one-dimensional space was reported by Kollios *et al.* [12]. Their algorithm is based on a dual transformation technique, which converts a line segment from the original space into a point in the transformed two-dimensional space. The method was extended to accommodate the case of objects moving in a two-dimensional space but whose movements are constrained by the existence of an underlying network.

Subsequent research on kNN queries mostly focused on predictive (either instantaneous or continuous, or both) queries. Using a TPR-tree as their supporting indexing structure, Benetis *et al.* [2] proposed both a reverse nearest neighbor algorithm and a nearest neighbor algorithm for moving queries, moving data case. They use a depth-first traversal of the tree, enhanced with metrics for pruning and directing the search. An algorithm for answering continuous queries is also presented in this work.

Tao et al. [34] also use a TPR-tree as their index of choice and provide a solution to kNN queries based on the concept of *time-parameterized queries*. Both the query point and the data points move in time (scenario 3), and a query is modeled as a line segment that shows the trajectory of the query point within a certain time interval. A query answer returns the closest neighbor of the query segment at any moment within the time interval. Initially, the closest neighbor c of the starting point s is retrieved, and the interval (s, p) where c remains the closest neighbor of the trajectory is computed (p is called a split point). Then, the procedure continues iteratively using p as the new start position.

The method proposed by Raptopoulou *et al.* in [23] addresses the same scenario and uses the same index structure to organize the moving objects as the work described above. The main improvement is a significant reduction of the number of queries issued for a time interval. In the previous work, the closest neighbor for each split point is retrieved with a separate query, whereas the new work performs only one query for the entire interval.

Li *et al.* [14] tackle the problem of continuous kNN queries in the context of scenario 3. A different approach is taken as compared to [23]. Rather than indexing the objects in their original space, a transformed time-distance space is used. In the transformed space, the movement of each object is represented as a curve in a plan where the x-axis is the time, and the y-axis is the euclidian distance from the object to the query point. Each such curve is divided into multiple segments, which are indexed in an R-tree. Their algorithm works as follows: first, the k objects that are the closest to the query point are retrieved by searching the whole dataset. As time passes, any changes in the query result set are detected by identifying all the curves that intersect with the curve of the k-th neighbor.

Other approaches [29, 32, 41] have studied the problem of continuous kNN queries over static data points (scenario number 1). In [29], the static objects are indexed in an R-tree. A sampling technique selects a set of points along the query object trajectory. For each split (sampled) point, the closest neighbor is retrieved using the R-tree index. For the rest of the points on the query segment, the result is obtained by applying linear splines to two consecutive sample points. The work of Tao *et al.* [32] differs in that the split points in a query trajectory are computed accurately. Moreover, a branch-and-bound strategy is used to traverse the R-tree that indexes the data. Zheng [41] addresses the same problem by using Voronoi diagrams. A key difference between these three contributions is that Zheng's method is hard to extend for more than 1 nearest neighbor.

All of the aforementioned research contributions offer solutions for answering kNN queries over either static or current/predictive databases. Frentzos *et al.* [6] were the first to study historical kNN queries over datasets of historical moving objects represented as trajectories (scenarios number 2 and 3). Several algorithms based on a branch-and-bound traversal of a TB-tree were presented, depending on the type of the query point (either moving or stationary) and the desired result of the query (continuous or not). A classical historical NN query retrieves the object(s) that came the closest to the query point during the entire time interval associated with the query. In contrast, the result of a continuous historical NN query consists of a list of objects $o_i, i = 1, n$, each of them associated with a time period t_i when the object o_i was the nearest neighbor of the query object.

The kNN contributions summarized before leave the problem formulation unchanged and develop algorithms specialized on kNN queries. Alternatively, it is possible to reformulate a kNN query as a window query in a spatio-temporal dataset. The basic idea is to determine a region that is assumed to contain all k nearest neighbors and to perform a window query for that region. A bigger query region increases the probability that all k neighbors are contained inside it but may imply a greater search effort (by accessing more data). A smaller region is processed faster, but the probability to contain all k neighbors decreases. Thus, an accurate region estimation is very important. For more details on research on this issue, see [3, 35].

With the notable exception of [11, 15], all methods presented before cannot be easily integrated into an RDBMS. The SPIT approach [16], on the other hand, can be fully integrated inside any RDBMS. Based on a two-layer approach, SPIT partitions the dataset according to the spatial location of the objects and creates temporal indexes over each spatial partition. The procedure is similar to the one described in the SETI approach [5] presented earlier: the space is divided into a fixed number of equal cells and a local index on the temporal dimension is created over the data within each cell. As opposed to SETI, SPIT uses B-trees for the temporal indexes, which allows it to be used within any existing RDBMS. In addition, the space partitioning is done according to a cost model aiming at minimizing query cost. However, the cost model assumes that data is uniformly distributed in space, which is often not true for real applications. The work of this thesis improves upon SPIT in several important aspects, as detailed in the introductory chapter, e.g., the use of a new partitioning algorithm, the splitting of time ranges and the development of nearest neighbors algorithm.

Chapter 3

SPIT$^+$

This chapter presents SPIT$^+$, a fully RDBMS integrable approach for historical spatio-temporal data storage and retrieval. SPIT$^+$ enhances the SPIT model introduced in [16] along the directions outlined in Section 1. Like SPIT, the proposed approach separates the spatial component from the temporal component of the data space and partitions the dataset into several disjunct subsets based on data's spatial coordinates. For each such subset a local index over the temporal dimension of data is created.

The dataset is partitioned using a grid that decomposes the spatial component of the data space into a number of cells. Each grid cell is assigned a unique identifier and, based on this identifier, is mapped to a different partition in the RDBMS. Within each grid cell, a B$^+$-tree is used to physically store the contained data. Figure 3.1 provides a conceptual representation of the SPIT$^+$ approach. Built on records time intervals $\langle t_s, t_e \rangle$, the B$^+$-tree organizes data based upon their temporal attribute. Besides the index keys, i.e., $\langle t_s, t_e \rangle$, the B$^+$-tree leafs also store both the identifiers and the spatial coordinates of all records.

Combined with the spatial decomposition, the separation of the spatial and temporal dimensions results in a very good search performance. When a spatio-temporal query is issued, only the subsets associated with the grid cells

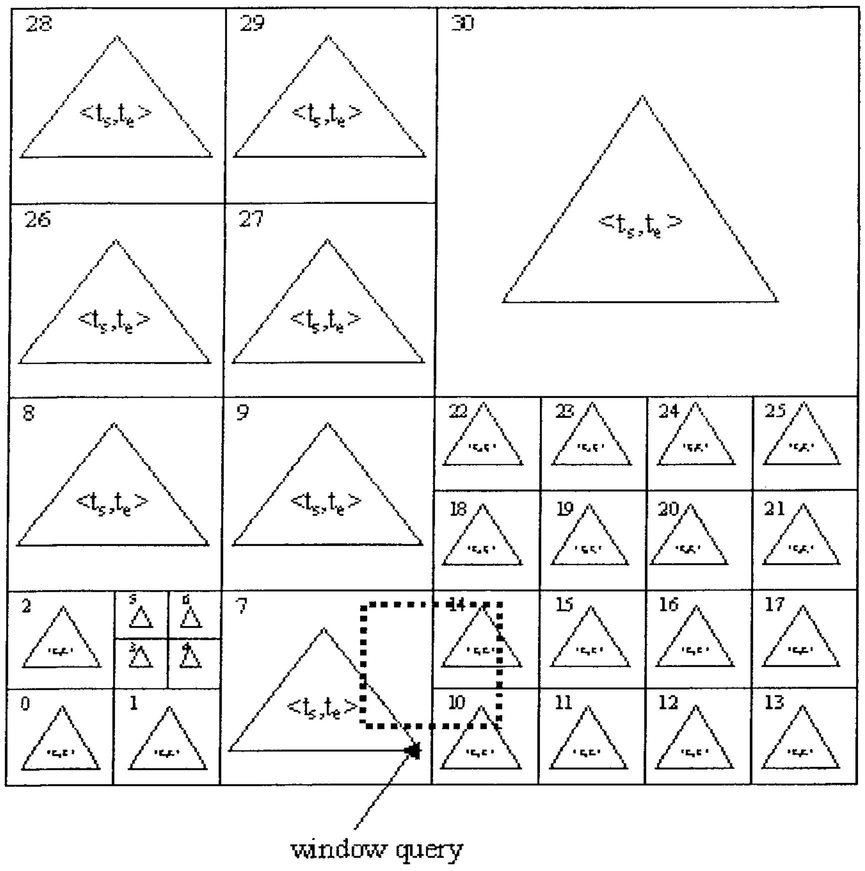


Figure 3.1: The SPIT$^+$ approach.

that intersect the query spatial component have to be searched. Typically, this is a small fraction of the entire dataset. The temporal index of each searched subset is then used to retrieve only the records that intersect the temporal interval of the query (query processing is presented in detail in the next section). Being a one-dimensional index, the temporal index does not exhibit the search performance degeneration which is characteristic to three-dimensional indexes (used when both the spatial and temporal dimensions are considered together).

Unlike SPIT, where all grid cells have the same size, SPIT$^+$ allows an irregular partitioning as the one shown in Figure 3.1. The irregular partitioning is beneficial for arbitrary (non-uniform) data distributions. It allows to use a coarser grid for areas with sparse data, and a finer grid for dense areas. How the spatial partitioning affects the query performance is discussed in Section 3.2.

Focusing on indexing and querying historical observations, which are completely available at index creation time, SPIT$^+$ constructs a static partitioning. A discussion on how the model can be extended to deal with new observations is provided later in this chapter.

3.1 Query Processing using SPIT$^+$

Like in [16], processing a spatio-temporal *window* query using SPIT$^+$ is a four-steps procedure. First, the grid cells that intersect the spatial component of the window query are identified. Second, the temporal index of each cell previously selected is used to retrieve only the records whose temporal intervals intersect the query temporal range. Next, the retrieved data is further filtered by removing the tuples whose spatial position falls outside the spatial range of the query. The final step is to eliminate any occurring duplicates (multiple instances of the same object).

Algorithm 1 *window_query*() function.

```
Input: $\langle \sigma, \tau \rangle$
Output: list of oid's
 1: pid_list := p_intersect($\sigma$)
 2: for all pid in pid_list do
 3:   oid_list := oid_list ∪
 4:       SELECT oid
 5:       FROM pid
 6:       WHERE $t_s$ BETWEEN $\tau.t_{min} - \mathcal{T}$ AND $\tau.t_{max}$
 7:       AND $t_e$ BETWEEN $\tau.t_{min}$ AND $\tau.t_{max} + \mathcal{T}$
 8:       AND $x$ between $\sigma.x_{min}$ and $\sigma.x_{max}$
 9:       AND $y$ between $\sigma.y_{min}$ and $\sigma.y_{max}$
10: end for
11: sort oid_list and remove duplicates
12: return oid_list
```

Algorithm 1 shows this procedure in pseudo-code. It assumes that the function *p_intersect()* already exists. Its task is to simply return the identifiers *pid* of the grid cells that intersect the query's spatial component. Only these cells have to be scanned (line 2). Lines 6–7 correspond to the second step of the query processing procedure. The tuples satisfying the temporal predicate of the query are retrieved by performing a range scan on the leaf nodes of each scanned partition B$^+$-tree. As done in [18], the algorithm uses the fact that the largest time interval, denoted as $\mathcal{T}$, is known. Knowledge of a dataset's $\mathcal{T}$ serves to further restrict the temporal range that needs to be inspected at query time, hence improving query processing time. In fact, given one value of $\mathcal{T}$ one can easily split all records whose temporal range length exceeds $\mathcal{T}$ into two or more records that adhere to the assumption. As can be easily seen in Algorithm 1, the smaller the value of $\mathcal{T}$, the smaller the range scan on the temporal index. There is, however, a trade-off in splitting the dataset's temporal ranges. It increases the number of indexed temporal ranges and hence the number of records in the database. A method for obtaining an optimal value of $\mathcal{T}$ for a given distribution of temporal ranges is presented in Section 3.3.

As some of the partitions determined at line 1 of the algorithm could intersect the query spatial range only partially, the actual $\langle x, y \rangle$ coordinates of the previously obtained records have to be checked against the query window (lines 8–9). Even though this test could be skipped for those partitions entirely contained in the query window, doing so would have no effect on the number of I/Os required to answer a query. When the B$^+$-tree is scanned for the necessary temporal filtering, all the leaf nodes containing records within the temporal range of the query have to be accessed. Since each object location is stored together with its index key in the B$^+$-tree leaf nodes, the above spatial test can be performed with no influence on the number of accessed leaf nodes, hence no additional I/Os.

3.2 Partitioning the Data Space

The spatial decomposition of the data space has a great impact on SPIT$^+$'s performance. The number of disk accesses required to answer a query depends both on the number of accessed data points and on the number of partitions containing these points. Consider the example in Figure 3.2, where two different decompositions are shown. The area that has to be searched in each case to answer the window query (represented as the dotted square in the figure) is shown in grey. Obviously, this area is larger when a coarser grid is used (case (a) in the figure). Assuming a uniform distribution of the data, a larger area will result in a bigger number of points that have to be accessed.

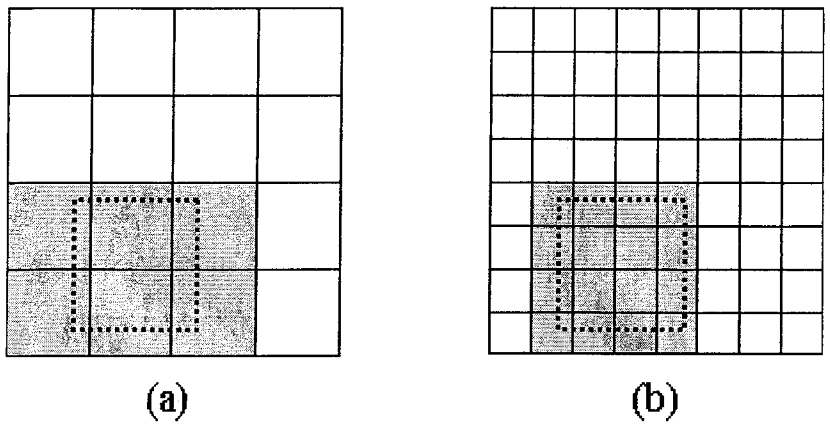


Figure 3.2: Different spatial decompositions.

In the second case, which uses a finer grid decomposition, the query intersects more partitions. Since the depth of a B$^+$ tree is logarithmic on the number of points, traversing several smaller trees (one for each intersecting partition) is more expensive than traversing one relatively larger tree that indexes the same information. Therefore, finding a good partitioning is very important for the model efficiency. A partitioning strategy aiming at minimizing the number of disk accesses is presented next.

A cost-based model, which refines the model introduced in [16], is used to compute the optimal number of cells in the grid. It assumes that the spatial domain is the unit square and that the temporal domain is formed by the (finite) set of recorded timestamps. Moreover, the average query sizes, on both the temporal and spatial dimensions, are supposed to be known – the robustness of SPIT$^+$ with respect to such an assumption is discussed in Section 4.1.3. All notations used for the cost model presentation are enumerated in Table 3.1.

Notation	Meaning
N	number of tuples, i.e., observations, in the dataset
DA	number of disk I/Os to answer a query
GA	average number of accessed grid cells
LA_c	number of leaf level I/Os per accessed grid cell
IA_c	number of directory level I/Os per accessed grid cell
BS	block size (number of tuples per data page)
q_s	average size (%) of the query in each spatial dimension wrt the modeled space
q_t	average size (%) of the temporal range of the query wrt the number of observed timestamps
l (l^*)	length (optimal length) of a grid cell per dimension
N_c (N_c^*)	total (optimal) number of cells in the grid
$\mathcal{T}$ ($\mathcal{T}^*$)	maximal (optimal maximal) length (%) of the indexed temporal ranges wrt the number of observed timestamps

Table 3.1: Notations used.

As in [16], the total number of disk accesses required to answer a query can be computed based on the average number of partitions (cells) that intersect the query and on the number of disk accesses performed inside each intersecting grid cell. Since a B$^+$-tree is used to store the data residing in each cell, the number of I/Os inside each partition can be determined as the number of accesses performed at the leaf-level of the tree combined with the number of accesses at the directory-level of the tree. As a result:

$$DA = GA \times (LA_c + IA_c) \tag{3.1}$$

The average number of cells intersected by a query can be estimated with the following formula [36]:

$$GA = N_c \times (l + q_s)^2 \tag{3.2}$$

When data is uniformly distributed, the average number of records inside each cell is N/N_c, whose storage requires $\frac{N/N_c}{BS}$ leaf pages. As the records are indexed on their temporal attribute, only the pages containing records that intersect the time interval of the query extended by $\mathcal{T}$ have to be accessed. Therefore, LA_c could be computed by the equation:

$$LA_c = \frac{N/N_c}{BS} \times (q_t + \mathcal{T}) \tag{3.3}$$

Finally, the cost of accessing the directory level of each cell is determined by the height of the B$^+$-tree, which can be calculated as $log_{b_f}(N/N_c)$, where b_f is the tree branching factor. For simplicity, IA_c's value is set to 3, a characteristic value for B-trees indexing millions of records and having the $b_f \approx 100$ [13], obtaining:

$$DA = (l + q_s)^2 (\frac{N \times (q_t + \mathcal{T})}{BS} + \frac{3}{l^2}) \tag{3.4}$$

The value of DA in the above equation is minimized for a cell size l^* given by

$$l^* = \sqrt[3]{\frac{3q_s \times BS}{N \times (q_t + \mathcal{T})}} \tag{3.5}$$

Consequently, the optimal number of grid cells N_c^* can be written as a function of l^* as:

$$N_c^* = \frac{1}{(l^*)^2} = (\frac{N \times (q_t + \mathcal{T})}{3q_s \times BS})^{2/3}. \tag{3.6}$$

Therefore, in order to minimize the number of disk accesses per query, given an average query size, the data space has to be decomposed using a *regular* grid that contains $\lceil\sqrt{N_c^*}\rceil$ equally sized cells in each dimension. In addition note that $\mathcal{T}$ in Eq. 3.6 is the only parameter one could fine-tune, the others are query or system dependent. In Section 3.3 a discussion is given on how this can be explored to further performance improvement.

3.2.1 Partitioning Data with Arbitrary Distributions

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

The idea is to recursively divide the space into four subspaces, as in a Quad-tree [27], until all obtained subspaces satisfy a uniform distribution criterion. The obtained subspaces are then partitioned using the cost model developed above. The uniformity of a data distribution can be tested using Pearson's Chi-Square test [25]. The test partitions the data into K equally sized cells (categories) and computes the sum (S^2) of squared differences between the actual number of objects inside each cell and the expected number of objects under the uniformity assumption. If the value of S^2 is smaller than $\chi^2_{K-1}(\alpha)$ (the $100(1-\alpha)$ percentile of a chi-square distribution with $K-1$ degrees of freedom) then the uniformity assumption is accepted, otherwise it is rejected. Algorithm 2 shows this procedure in pseudo-code.

Algorithm 2 *Partition()* recursive algorithm.

Input: An MBR containing data points

Output: A set of MBRs (each corresponding to a grid cell) and respective partitionings

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

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

It may appear at first that the partitioning strategy may result in many small cells with very few objects in each of them. This obviously would not be a good idea since there is an overhead cost to access a partition, and there is a point where accessing less data in more partitions is more expensive than accessing more data within less partitions. Fortunately, the heuristic above is able to identify such situation, stopping the partitioning accordingly. Recall that, during the partitioning, q_s^2 is the query size with respect to the current

MBR, and similarly N is the number of objects inside the current modeled space, i.e., the current MBR. Initially the current MBR is the whole unit square, but as the partitioning progresses, the MBRs are subdivided and the current MBRs become smaller. As an obvious consequence, q_s becomes larger with respect to the current MBR. On the other hand, the number N of objects per MBR becomes likely smaller as the MBRs are subdivided. Consider the case when the query size becomes equal to the current MBR, i.e., $q_s = 1$. It can be seen, from Eq. 3.6, that if $q_s = 1$ and BS, q_t and $\mathcal{T}$ are constants, then $N < \frac{3BS}{(q_t+\mathcal{T})}$ yields $N_c^* = 1$, i.e., no further partitioning is needed. This agrees with the intuition that as the partitioning progresses, there is a point where accessing less data in more partitions becomes more likely and more expensive than accessing more data within a single partition. At that point the partitioning process stops automatically.

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

3.3 Optimizing $\mathcal{T}$

As mentioned in Section 3.1, the size of the range scan on the B$^+$-tree that indexes the temporal intervals depends on the length of the largest indexed interval $\mathcal{T}$. There is nothing however, that prevents one to setting $\mathcal{T}$ "artificially" in order to optimize the index performance. As one decreases the value of $\mathcal{T}$ from its so-called intrinsic value, i.e., that inherent to the dataset, the number of temporal ranges that have to be split increases. As a result, the range scan on the index will become shorter, but the number of indexed objects will increase. The former consequence has potential positive impact on the performance of the temporal index while the latter has a negative effect. A method on how those two effects could be balanced in order to obtain an optimal value for $\mathcal{T}$, denoted as $\mathcal{T}^*$, is introduced next.

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

Let $l_k \in \{l_1, l_2, ...l_{M-1}\}$ be one given length that is going to be set as the maximal allowed length. (The case where $l_k = l_M$ induces no splits and therefore is not of interest.)

When splitting all ranges larger than l_k the current number of indexed ranges, originally N, will now become

$$N_k = N + \sum_{p=k+1}^{M} [C(l_p) \times (\lceil l_p/l_k \rceil - 1)] \tag{3.7}$$

Returning to Eq. 3.4 and substituting the expression of l^* (Eq. 3.5) for l, the first derivative of (3.4) with respect to $N \times (q_t + \mathcal{T})$ is positive when N, q_t and $\mathcal{T}$ are positive, which means that the number of disk accesses is monotonically increasing with $N \times (q_t + \mathcal{T})$. Consequently, the value $\mathcal{T}$ that minimizes $N \times (q_t + \mathcal{T})$ will also minimize the number of I/Os. Hence, the optimal value of the largest temporal interval is set to:

$$\mathcal{T}^* = l_k, \text{ where } k = \arg min_k\{N_k \times (q_t + l_k)\}.$$

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

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

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

3.4 Processing Spatio-Temporal kNN Queries

The previous sections of this chapter provided a partitioning algorithm designed to optimize the cost (in terms of the required number of I/Os) of a spatio-temporal window-query and showed how the proposed access structure can be used to process such a query. However, window queries are not the only type of spatio-temporal queries that SPIT$^+$ can handle. How SPIT$^+$ could be employed in solving spatio-temporal kNN queries is presented next.

First, a kNN search algorithm that assumes the data is already partitioned is introduced, and then a partitioning scheme that takes into account the query parameters (the value of k and the length of the time interval) is provided.

As already introduced, a spatio-temporal kNN query specifies a time interval and asks for the k objects that were the closest to the query point during that period. Provided that a list of grid cells containing all k neighbors could be determined, a kNN query can be processed similarly to a window query: only the partitions containing the closest neighbors are scanned and their temporal indexes are used to filter out those objects that do not satisfy the query temporal predicate. In addition, the list of retrieved objects has to be sorted according to the euclidian distance between the objects and the query point, and only the first k unique objects are returned as the query result set. However, as opposed to the case of window queries, here it is not possible to determine a priori the list of all partitions that need to be scanned, as they depend on the distance between the query object and its k-th neighbor, which is unknown.

Rather than trying to first determine all the partitions that need to be searched and then use them to compute the query answer, the k nearest neighbor algorithm proposed in this work computes the answer set in an incremental manner. The main idea employed by the proposed algorithm, which adapts the method proposed in [9] to a single-level data structure, is to keep an ordered list of all partitions and to scan one partition at a time, according with their order in the partitions list. When one partition is accessed, all its contained objects (points) having their temporal interval overlapping with the query temporal interval are retrieved and added to a list of potential k-nearest neighbors, based on their distance to the query point. Having found at least k objects, only those partitions lying within a distance smaller than $kDist$, the distance between the query point and its current k-th neighbor, from the query point need to be further accessed. Algorithm 3 states this procedure in pseudo-code.

Given a partition pid, the distance $dist(Q, pid)$ between the query point and pid is computed such that for each point P contained in pid, $dist(Q, pid) \leq dist(Q, P)$.

As mentioned above, once at least k points have been found (line 12), $dist(Q, P_k)$ is used to prune some of the partitions in $pList$. Since $\forall P \in pid$, $dist(Q, pid) \leq dist(Q, P)$, only those partitions having their distance to Q smaller than $dist(Q, P_k)$ can contain points that are closer to Q than the

Algorithm 3 *determine_kNeighbors()* function

```
Input: A query point $Q(x, y)$, a number $k$ of neighbors and a query time interval $q_t$
Output: The $k$ closest neighbors of $Q$
1: initialize oList to $\emptyset$
2: create pList, the list of all existing partitions
3: sort pList according with their distance to $Q$
4: initialize kDist with maxReal
5: pCurrent := the first partition in pList
6: remove pCurrent from pList
7: while dist(Q, pCurrent) <= kDist and pList != $\emptyset$ do
8:   for each point $P$ in pCurrent (and satisfying $q_t$) do
9:     if dist(q, P) <= kDist then
10:      add $P$ to oList
11:      sort oList according to the distance to $Q$
12:      if Card(oList) >= $k$ then
13:        $P_k$ := the k-th element in oList
14:        kDist := dist($Q, P_k$)
15:      end if
16:    end if
17:  end for
18:  pCurrent := the first partition in pList
19:  remove pCurrent from pList
20: end while
21: return the first $k$ objects in oList
```

current k-th neighbor. As long as *pList* contains partitions closer to Q than P_k, these partitions are scanned one at a time and both *oList* and $dist(Q, P_k)$ are updated accordingly. When the first partition having a distance greater than the current $dist(Q, P_k)$ is found, the algorithm stops and the first k points in *oList* are returned as the query answer set.

For a better understanding of the above algorithm, consider the example provided in Figure 3.3 and assume that the query asks for the 2 nearest neighbors of Q (the grey dot in the figure). Moreover, consider that the distances between Q and all partitions and objects are the ones provided in Table 3.2.

The steps executed by the algorithm are the following:

- *pList* = {P5, P6, P2, P3, P8, P9, P4, P1, P7}; *kDist* = maxReal;

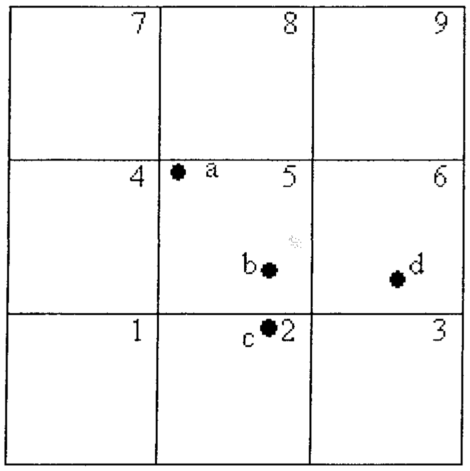


Figure 3.3: Example kNN query.

Partitions/ Objects	P1	P2	P3	P4	P5	P6	P7	P8	P9	a	b	c	d
Distances	8	3	4	7	0	1	9	4	5	7	2	4	6

Table 3.2: Distances of partitions and objects from the query point.

- *pCurrent* = P5; *pList* = {P6, P2, P3, P8, P9, P4, P1, P7}; *dist*(Q,pCurrent) $<$ *kDist* $\Rightarrow$ Scan P5; *oList* = {b, a}; *kDist* = dist(Q, a) = 7;
- *pCurrent* = P6; *pList* = {P2, P3, P8, P9, P4, P1, P7}; *dist*(Q, *pCurrent*) $<$ *kDist* $\Rightarrow$ Scan P6; *oList* = {b, d, a}; *kDist* = dist(Q, d) = 6;
- *pCurrent* = P2; *pList* = {P3, P8, P9, P4, P1, P7}; *dist*(Q, *pCurrent*) $<$ *kDist* $\Rightarrow$ Scan P2; *oList* = {b, c, d, a}; *kDist* = dist(Q, c) = 4;
- *pCurrent* = P3; *pList* = {P8, P9, P4, P1, P7}; *dist*(Q, *pCurrent*) $\leq$ *kDist* $\Rightarrow$ Scan P3; *oList* = {b, c, d, a}; *kDist* = dist(Q, c) = 4;
- *pCurrent* = P8; *pList* = {P9, P4, P1, P7}; *dist*(Q, *pCurrent*) $\leq$ *kDist* $\Rightarrow$ Scan P8; *oList* = {b, c, d, a}; *kDist* = dist(Q, c) = 4;

- *pCurrent* = P9; *pList* = {P4, P1, P7}; $dist(Q, pCurrent) > kDist \Rightarrow$ STOP; answerSet = {b, c}

The proposed k-nearest neighbor algorithm is optimal in the sense that the number of accessed partitions is exactly the same as if the distance to the k-th neighbor was known a-priori. This can easily be proven using the following remarks. Let P_k be the true k-th neighbor of the query point Q and let *kDist* be the distance between Q and P_k. To retrieve all the k-nearest neighbors, any optimal algorithm must scan all and only the partitions that intersect the circle centered at Q with radius *kDist*. These are the partitions *pid* having $dist(Q, pid) \leq kDist$ and are exactly the partitions that the algorithm accesses. Since *pList* is ordered based on the distance to Q, all the partitions accessed before finding P_k are closer to Q than the partition containing P_k and, consequently, their distance to Q is smaller than *kDist*. After finding P_k, the algorithm scans all partitions closer than P_k that were not scanned yet. No other partitions (i.e., with distance larger than *kDist*) are scanned by this algorithm (line 7 of Algorithm 3).

In order to answer kNN spatio-temporal queries using SPIT$^+$, a partitioning of the data space has to exist. Recall that the partitioning algorithm introduced in Section 3.2 requires the average query sizes in both spatial and temporal dimensions. While a kNN query provides a time interval whose length can be used for the average query size in the temporal dimension, an average size of the query in the spatial dimension has to be determined.

The first step in computing an average query size in the spatial dimension is to estimate E_{d_k}, the distance between the query point and its k-th nearest neighbor. According to [35], E_{d_k} can be estimated by the equation:

$$E_{d_k} = \frac{2}{\sqrt{\pi}} \left[1 - \sqrt{1 - \left(\frac{k}{N_U} \right)^{\frac{1}{2}}} \right] \qquad (3.8)$$

where N_U is total number of distinct objects in the dataset.

Once the expected distance E_{d_k} is determined, the query region in the spatial dimension could be approximated by the square that tightly encloses the circle centered at the query point and having radius the expected distance, $C(Q, E_{d_k})$, yielding $q_s = 2 \times E_{d_k}$.

3.5 SPIT$^+$'s Implementation

This section provides details on how the proposed indexing method is implemented inside an ORACLE database. Each record is represented as a tuple $\langle oid, x, y, t_s, t_e, pid\rangle$, where *oid* is the object identifier, $\langle x, y\rangle$ are the spatial coordinates, $\langle t_s, t_e\rangle$ indicates the time period during which the object was recorded at position $\langle x, y\rangle$, and *pid* is the identifier of the partition containing the object. All records are stored in an index-organized table called `SPIT_PLUS`. An index-organized table is a table that is embedded within an index, i.e., the index contains the tuples themselves in the leaf nodes of the index, e.g., a B$^+$-tree. Although index-organized tables are supported by most of the existing RDBMSs[1], their use is not a requirement for SPIT$^+$ to work. It could be implemented using a regular table and index, the only difference being some extra I/Os required for obtaining the actual tuples from a table after traversing the index. Built on the temporal attributes $\langle t_s, t_e\rangle$, the index-organized table is able to efficiently retrieve all tuples intersecting the query temporal interval by simply performing a sequential range scan of the index leaves. Moreover, the `SPIT_PLUS` table is range partitioned on *pid* to allow the mapping of SPIT$^+$'s grid to different table partitions. Each grid cell determined by the heuristic algorithm *Partition()* is associated with a single table partition. An unique partition id (*pid*) along with its MBR is stored in a table called

[1]A similar structure is called `clustered index` in Microsoft SQL Server, while the MySQL equivalent is an `InnoDB table`.

PARTITIONS. When an object is inserted into table SPIT_PLUS its coordinates are first checked against the PARTITIONS table to determine in which partition it should be inserted. Figure 3.4) shows the DDL of the SPIT_PLUS table for the example grid in Figure 3.1. In order to create an index-organized table in ORACLE, a primary key has to be defined and the clause organization index has to be included in the CREATE statement. The primary key specifies the attributes based on whose values the data is indexed. Besides the temporal attributes $\langle t_s, t_e \rangle$, it includes the partition identifier *pid* so that a local temporal index is created for each partition (as opposed to a global index for the entire table). For an RDBMS that does not provide partitioning functionality, the spatial grid and the local indexes could be implemented by simply creating a physical table and an index for each grid cell.

```
create table SPIT_PLUS (
   oid        integer,
   x          number,
   y          number,
   t_s        number,
   t_e        number,
   pid        integer,
   primary key(t_s, t_e, pid)
)
   organization index
   partition by range (pid) (
   partition p01 values less than (1),
   partition p02 values less than (2),
   ...
   partition p30 values less than (MAXVALUE)
)
```

Figure 3.4: A sample SPIT$^+$ table.

Two sample SQL queries issued against the SPIT_PLUS table defined above are shown in Figure 3.5. In particular, the SQL query in Figure 3.5 (a) corresponds to the spatio-temporal *window* query "find all the objects that were inside the area enclosed by the MBR determined by vertices (0.40,0.05) and

(0.55,0.20) during the time interval [0.6,0.8]", while the one in Figure 3.5 (b) was generated for the spatio-temporal *kNN* query "find the five closest objects to Q(0.5,0.1) during the time interval [0.6, 0.8]". In both examples T represents the optimal range length $\mathcal{T}^*$.

```
 1: SELECT DISTINCT oid
 2: FROM SPIT_PLUS
 3: WHERE pid IN (7,10,14)
 4: AND t_s BETWEEN (0.6 - T) AND 0.8
 5: AND t_e BETWEEN 0.6 AND (0.8 + T)
 6: AND x BETWEEN 0.40 AND 0.55
 7: AND y BETWEEN 0.05 AND 0.20;
```

(a) Sample window query.

```
 1: SELECT oid FROM (
 2:     SELECT oid, MIN(dist(oid, Q))
 3:     FROM SPIT_PLUS
 4:     WHERE pid IN (7,10,14)
 5:     AND t_s BETWEEN (0.6 - T) AND 0.8
 6:     AND t_e BETWEEN 0.6 AND (0.8 + T)
 7:     GROUP BY oid
 8:     ORDER BY MIN(dist(oid, Q))
 9: )
10: WHERE ROWNUM <= 5;
```

(b) Sample k-NN query.

Figure 3.5: Querying SPIT$^+$'s data.

In the case of the window query, first the list of grid cells intersecting the spatial component of the query is computed by performing a lookup in the PARTITIONS table. Only the table partitions associated with grid cells (7,10,14) are searched (line 3). For each scanned partition, the local B$^+$-tree index on $\langle t_s, t_e \rangle$ is used to perform a index range scan of the data in order to retrieve only those tuples intersecting the temporal interval of the query (lines 4–5). The tuples whose spatial location is outside the query's spatial range are filtered out on lines 6–7. Any duplicates occurring in the answer set are eliminated by including the DISTINCT clause in the line 1, which forces ORACLE to return only distinct values of *oid*.

For the kNN query, the inner SELECT retrieves the list of all unique objects residing inside the queried partitions (line 4) and satisfying the query temporal predicate (lines 5–6). The clause GROUP BY, used in conjunction with the MIN(*dist()*) function, groups the tuples into several categories based on their identifier and for each such category (unique *oid*) returns only the tuple whose distance to the query point has the smallest value. The list of unique objects is further ordered (line 8) according with the objects' distance to the query point. Finally, the outer SELECT returns only the first five objects (nearest neighbors) from the above list.

The SQL statements used to query the SPIT_PLUS table are dynamically generated using PL/SQL functions. The functions take as input the spatial query range (the query point and the number of nearest neighbors for the kNN queries) and the temporal query interval and produce SQL queries of the type shown in Figure 3.5. These functions, as well as all the other functions/algorithms required by SPIT$^+$, are implemented using the ORACLE JDeveloper environment. Supporting the development of both PL/SQL and Java procedures, JDeveloper offers the convenience of object-oriented and procedural languages while enabling the access of database objects.

3.6 Extending SPIT$^+$ to Handle New Observations

SPIT$^+$ was designed to handle historical spatio-temporal data, as such handling updates, i.e., new observations is not a concern. SPIT$^+$ is a feasible alternative for a scenario where data is collected to be queried at a later point in time. Given an indexed (historical) dataset, a new dataset can be merged with the current one using the existing partitioning or a new index could be built altogether for the newly combined database. The former may yield sub-optimal performance, depending on the size and spatial distribution of the new dataset. From this perspective the latter is a better option, and, as the empirical tests will show, index building times are quite reasonable for most practical purposes. Another possibility could be to create several indexes for different time periods, e.g., one per week. In this case queries would have to be re-written for handling the case where they span over several such indexes. Further exploration of these ideas is left as future work.

Chapter 4

Experimental Results

This section describes the empirical evaluation conducted to test the performance of SPIT$^+$. Two major classes of experiments have been performed. The experiments in the first class, described in Section 4.1, concentrate on evaluating the effectiveness of the decomposition strategies and the robustness of the model with respect to its parameters. The experiments on the second class, which are presented in Section 4.2, are used to show the performance of SPIT$^+$ in answering spatio-temporal queries, while also comparing it to other spatio-temporal indexing approaches.

For all experiments, both synthetic and real datasets have been used. One of the synthetic data sets, denoted as UNIFORM, has the objects uniformly distributed in the space and moving freely throughout the whole space. The second synthetic dataset was generated using the GSTD tool[1] [37] and shows a scenario where the objects have an initial gaussian distribution in the center of the data space and then migrate towards the north-east corner of the same. A sample instance of this dataset, denoted as GSTD, is illustrated in Figure 4.1(a), where all observed positions for a sample of 100 objects are shown. This dataset could depict a scenario where animals are migrating from one area to another in a park. It will serve to show how well the proposed partitioning scheme adapts for a truly non-uniform data distribution. The final

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

dataset, denoted as INFATI, contains real GPS positions of 20 cars roaming across the municipality of Aalborg, Denmark [10]. Each car's positions have been sampled every second, except when they were parked, for about 6 continuous weeks over a period of 3 months. The dataset contains approximately 1.9 million observations and is illustrated in Figure 4.1(b) where all observations are plotted.

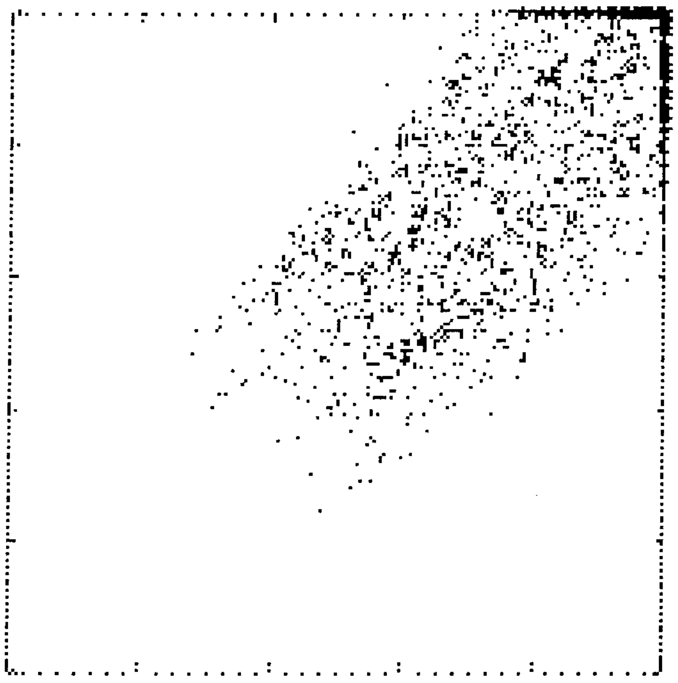
For each of the synthetic datasets, three different cardinalities have been tried, namely 1, 2.5 and 5 million observation data points, corresponding to 10, 25 and 50 thousand objects of interest with 100 sampled positions each.

Table 4.1 summarizes the parameters used for the experiments. Unless otherwise mentioned, whenever one parameter is being investigated, e.g., the robustness with respect to dataset size, all other parameters are kept constant at their default values. In all tests, the spatial domain of the search space was assumed to be the unit two-dimensional square, while the temporal domain is formed by the set of all recorded timestamps.

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

Table 4.1: Parameters and respective values investigated.

To investigate the average index access cost, 100 random queries following the same distribution as the dataset have been issued for each dataset. The average number of disk I/Os (physical reads) per query has been measured using the system's own internal tools and is used as performance indicator. All tests were carried out on a desktop using ORACLE 10g Enterprise for Windows Edition. The disk page size has been set to 8192 bytes. To avoid any influence on query performance, the DBMS's buffers were cleared before


(a) GSTD dataset

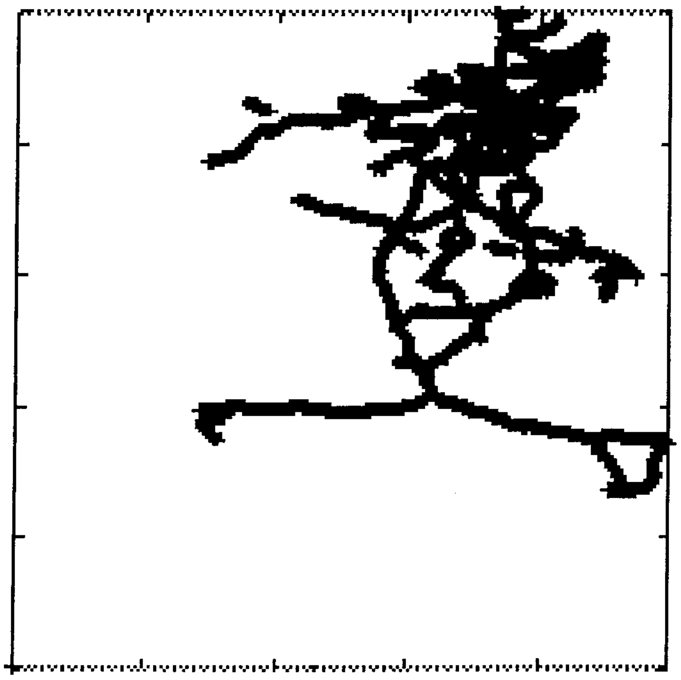

(b) INFATI dataset

Figure 4.1: Data distribution for the GSTD and INFATI datasets.

executing each query (using ORACLE's `alter system flush buffer_cache` command [21]). The task of query evaluation (execution plan) was left to the DBMS's query engine and the query optimizer mode was set to default, resulting in a cost-based mode as table statistics were collected before running the query workload. During the experiments, it has been observed that the execution plan created by the query engine when processing queries using the SPIT+ approach is exactly the expected one: a linear scan of the `PARTITIONS` table is used to determine the partitions that need to be accessed and for each such partition the local B^+-tree index is used to retrieve only the tuples whose temporal intervals overlap with the query temporal interval.

SPIT$^+$'s performance was compared against other two approaches that could also be implemented on top of ORACLE. The first approach is a simple *Linear Scan* which should provide the lower bound for expected performance. The second method uses an R-tree for the spatial component along with a B^+-tree for the temporal component. The R-tree is constructed over two-dimensional point objects consisting of the records spatial coordinates $\langle x, y\rangle$ and the B^+-tree is created on records temporal attributes (t_s, t_e). In what follows this scheme is referred as "R-tree+B-tree".

Just like within SPIT$^+$, in the R-tree+B-tree scheme, the temporal ranges are indexed using a B^+-tree, meaning that it can potentially benefit from the knowledge of $\mathcal{T}$ as well. However, finding an optimal value of $\mathcal{T}$ for this case is not trivial, as the increase in the number of total objects due to time intervals splitting will affect not only the B^+-tree but also the R-tree. If the number of split objects is not very large, e.g., as in the presence of relatively few long temporal ranges, the effects on the R-tree are, however, not very large. On the directory level a large number of splits would be needed to cause relevant changes in the structure. On the leaf level the number of I/Os will increase proportionally to the increase in the number of indexed objects, which is assumed to be not very high. In addition, this effect is mitigated by the efficiency of the underlying range scan of the B$^+$-tree. Considering all these factors, the value of $\mathcal{T}^*$ derived for SPIT$^+$ has also been used within the R-tree+B-tree. Nevertheless, as the following experimental results will show, the difference in performance between SPIT$^+$ and R-tree+B-tree is so large that finding the true optimal value for $\mathcal{T}$ for the latter would very unlikely improve its performance by a factor large enough to make it a competitive approach.

Finally, SPIT$^+$ is compared to the MV3R-tree [31] using the (unmodified) source code kindly made available by its authors[2]. Even though the MV3R-tree is not an index that can be easily mapped onto an RDBMS, and therefore lacks the practical aspect that SPIT$^+$ promotes, it is a well known index for historical spatio-temporal data and it has been shown to outperform a simple 3D R-tree, which would have been another competitor for SPIT$^+$.

4.1 Model Assessment and Validation

This section offers details on the experiments performed to investigate the effectiveness of the spatial partitioning and temporal intervals splitting strategies and presents the obtained results. The model robustness with respect to the query size is also examined here. As both the spatial partitioning and the temporal splitting used within SPIT$^+$ are designed to improve the index access cost in answering spatial-temporal window queries, the performance figures reported throughout this section refer to window queries performance.

4.1.1 Partitioning Effectiveness

In order to investigate the effectiveness of the space partitioning schemes described in Section 3.2 the number of disk accesses reported by ORACLE

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

when using SPIT$^+$'s partitioning algorithm (cost-based model for uniform distributed data) is compared against the number of disk accesses when an ad-hoc partitioning is used. The term "ad-hoc partitioning" refers to the case where the user chooses a grid size manually.

Figure 4.2(a) shows the performance of SPIT$^+$ and ad-hoc partitioning when the UNIFORM dataset is used. In this case, SPIT$^+$ uses the cost-based model to compute the optimal number of partitions and decomposes the space using a regular grid. When using all experimental default values, the SPIT$^+$ determined a 13×13 grid, which indeed is the best option when compared to several other choices for a regular partitioning of the data space as shown in the above mentioned figure.

As for the ad-hoc partitioning, when the number of partitions is smaller than the optimal number suggested by SPIT$^+$'s cost-based model, the number of disk accesses required to answer a query increases sharply with the decrease of the number of partitions. The reason is that, as the number of partitions is reduced, the area covered by each partition becomes larger, which, in turn, results in a larger amount of data that have to be accessed inside each partition. A search performance degeneration could also be observed when the number of partitions is greater than the optimum. In this case, the increased number of disk accesses is attributable to the cost of accessing more temporal indexes. For a number of partitions larger than the last value shown on the figure, it is expected that the number of disc accesses will grow at about the same rate as the total number of partitions.

For non-uniform distributions, SPIT$^+$ decomposes the space using a non-regular grid as detailed in Section 3.2.1. Again, its performance is compared to the ad-hoc alternative of having the user trying several different regular grids. As can be seen in Figures 4.2(b) and (c), for both non-uniform distributions the grid partitioning determined automatically by SPIT$^+$ provides performance at

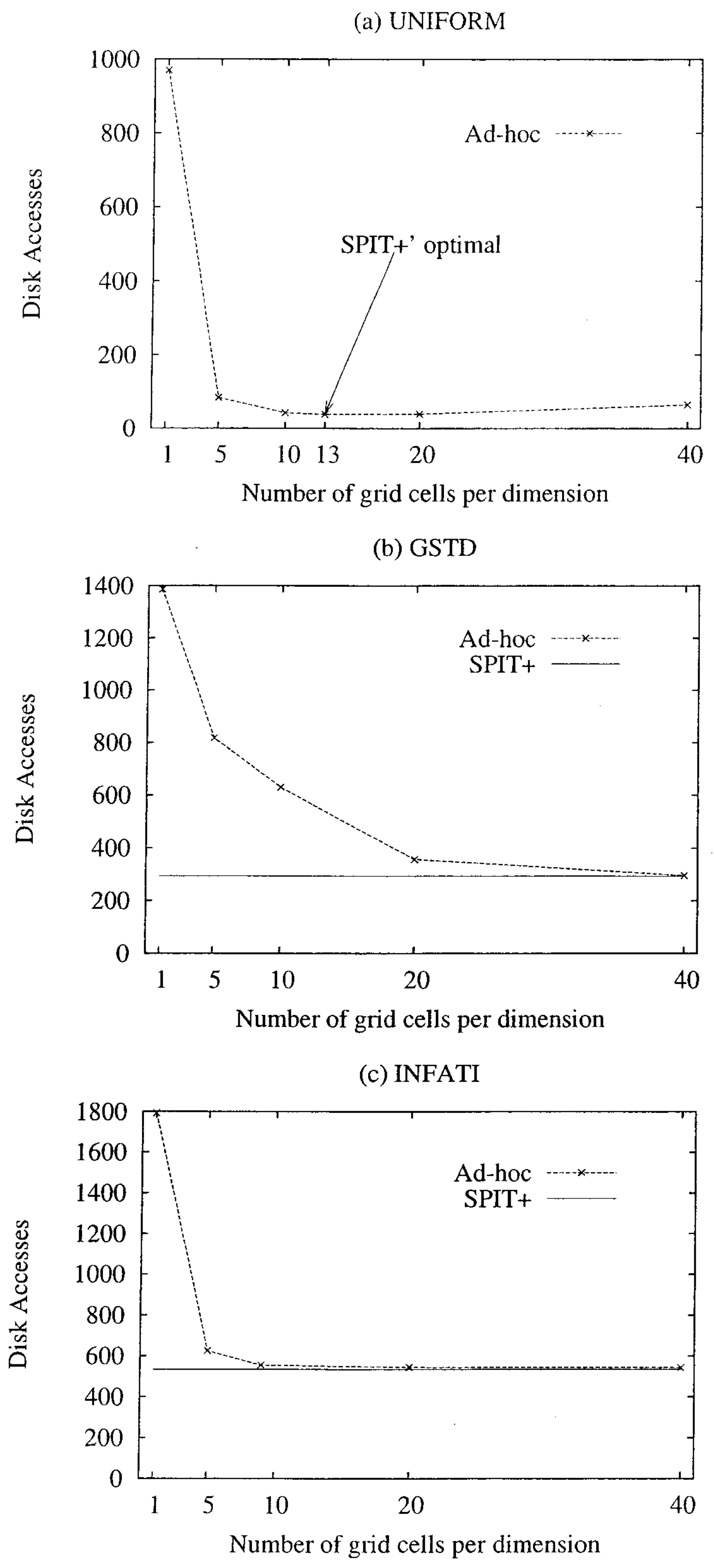


Figure 4.2: Comparing I/O performance yielded by SPIT^{+}'s partitioning against the use of ad-hoc regular grids.

least as good as the best ad-hoc partitioning. Since the resulting grid is non-uniform it does not make sense to plot performance as a function of the number of grid cells, hence the flat line for the SPIT$^+$ performance. Like in the case of the UNIFORM dataset, the additional cost of underpartitioning is very clear, but overpartitioning seems to be not as prejudicial.

4.1.2 Optimizing $\mathcal{T}$

As discussed in Section 3.3, adjusting the value of $\mathcal{T}$ yields a trade-off between improving query performance and enlarging the database. The argument used there was that the more skewed the distribution of temporal ranges towards shorter ranges, the more performance improvement can be achieved and the more worthwhile to enforce the optimal value $\mathcal{T}^*$ as the maximal length of the temporal ranges. In order to investigate this, four datasets have been used. Three datasets are synthetic datasets having a uniform spatial distribution (since the optimization is on the temporal level, the spatial distribution is irrelevant), with varying degrees of skewedness on the temporal ranges length. One is simply uniformly distributed and the other have the range lengths following an Exponential distribution $Exponential(\lambda)$, with rate parameters λ equal 0.5 and 4 (the larger λ the more skewed the distribution). Those are denoted by UNIF, UNIF+exp(0.5) and UNIF+exp(4) respectively. Finally, the INFATI dataset is used as a representative of a realistic distribution.

	I/Os w/ $\mathcal{T}$	I/Os w/ $\mathcal{T}^*$	Perf. Gain	Storage Overhead
UNIF	78.4	52.9	32%	80%
UNIF+exp(0.5)	73.5	39.5	46%	16%
UNIF+exp(4)	62.3	20.9	67%	4%
INFATI	2267.8	533.1	77%	0.5%

Table 4.2: Performance improvement and storage overhead due to $\mathcal{T}^*$.

Table 4.2 shows the obtained performance when using both the dataset's intrinsic $\mathcal{T}$ and the optimal $\mathcal{T}^*$ obtained as described in Section 3.3, as well as the yielded storage overhead. The dataset sizes as well as the query sizes used were the default values in Table 4.1.

Clearly, the more skewed the distribution of the lengths of the temporal ranges towards shorter ranges, the better the improvement in query processing time and the smaller the storage overhead. The skewedness of the temporal ranges is in fact a realist assumption, as evidenced by INFATI's distribution, which not coincidentally yielded the largest improvement with the smallest overhead. The gains are even larger when the temporal query range is smaller, as shown in Figure 4.3 where the performance gain when using the INFATI dataset is plotted for different lengths of the temporal range. This is due to the fact that the range scan on the B$^+$-tree leaves has a length of $q_t + \mathcal{T}$ (Eq. 3.3), the smaller q_t the more important $\mathcal{T}$ becomes, and thus the more important it is to optimize it accordingly. Hence, this optimization is used as an integral part of the SPIT$^+$ technique in the remainder of the experiments.

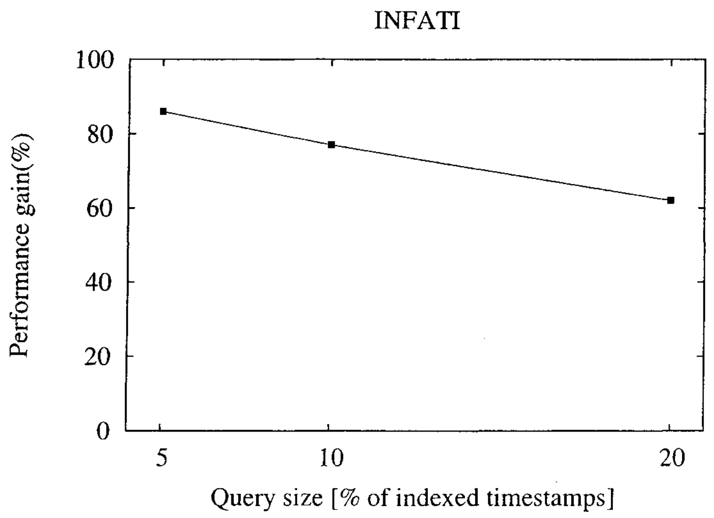


Figure 4.3: Performance gain as a function of the query temporal range length.

4.1.3 Robustness

As pointed out, SPIT$^+$'s spatial decomposition and, consequently, the query cost depend on an assumed query size, both for the temporal and the spatial component. In the next set of experiments it is shown how the performance is affected when the user estimates one query size but the actual posed queries have a different size. Ideally, one would want the performance to be robust, i.e., to not degrade much with reasonable variances between the assumed and actual query sizes. In all forthcoming tables the values in the first row represent the query sizes assumed at index construction time, with S being the size percentage-wise with respect to the spatial extent of the data space, and T being the size as a percent of the number of indexed timestamps. Following a similar notation, the values on the first column are the sizes of the issued queries. Hence, in the ideal case, the smallest values (shown in bold) should appear in the diagonal of the tables.

	$S = 0.25\%$ $T = 10\%$	$S = 1\%$ $T = 10\%$	$S = 4\%$ $T = 10\%$
$S = 0.25\%$, $T = 10\%$	**18.82**	21.22	25.58
$S = 1\%$, $T = 10\%$	39.51	**37.72**	42.23
$S = 4\%$, $T = 10\%$	95.25	**89.38**	91.03

(a) UNIFORM dataset

$S = 0.25\%$, $T = 10\%$	**141.85**	168.80	197.02
$S = 1\%$, $T = 10\%$	**278.39**	294.10	329.49
$S = 4\%$, $T = 10\%$	682.95	**670.52**	672.52

(b) GSTD dataset

$S = 0.25\%$, $T = 10\%$	390.53	**388.75**	399.94
$S = 1\%$, $T = 10\%$	545.22	**533.11**	552.37
$S = 4\%$, $T = 10\%$	704.64	**679.09**	682.77

(c) INFATI dataset

Table 4.3: I/O robustness of SPIT$^+$ for all three datasets with respect to spatial query size (temporal query size is fixed).

	$S = 1\%$ $T = 5\%$	$S = 1\%$ $T = 10\%$	$S = 1\%$ $T = 20\%$
$S = 1\%, T = 5\%$	**25.45**	27.63	27.35
$S = 1\%, T = 10\%$	43.73	**37.72**	37.93
$S = 1\%, T = 20\%$	69.01	61.82	**55.80**

(a) UNIFORM dataset

$S = 1\%, T = 5\%$	**204.82**	209.87	227.94
$S = 1\%, T = 10\%$	322.17	294.10	**281.11**
$S = 1\%, T = 20\%$	485.95	438.41	**416.83**

(b) GSTD dataset

$S = 1\%, T = 5\%$	298.91	**296.21**	312.28
$S = 1\%, T = 10\%$	553.78	**533.11**	544.01
$S = 1\%, T = 20\%$	1043.51	**991.90**	995.35

(c) INFATI dataset

Table 4.4: I/O robustness of SPIT$^+$ for all three datasets with respect to temporal query size (spatial query size is fixed).

Tables 4.3(a), (b) and (c) show the obtained performance when varying the size of the spatial component of the query, and fixing the temporal range length, for all three datasets. Tables 4.4(a), (b) and (c), on the other hand, show the performance when the temporal range varies and the spatial query remains fixed.

As can be observed, the smallest number does not always appear in the diagonal of the tables as in the ideal case. One reason for this is that the cost model often suggests a non-integer number of grid cells (Eq. 3.6), which is obviously not practical and has to be approximated to an integer. Another reason is the partitioning procedure is not guaranteed to deliver optimal results in the case of non-uniform spatial distributions, which is the case of the GSTD and INFATI datasets. Nevertheless, even in such cases, the difference between the value occurring on the diagonal and the smallest value in the corresponding row is very small.

Overall, the performance does not vary too much if one builds the dataset assuming a "wrong" (within reasonable limits) average query as can be seen throughout the tables. These results serve to show that SPIT$^+$ is indeed a robust approach with respect to the assumed query size. That is to say, even if the query size estimated at index building time is off by a factor of two or four in either the spatial or temporal dimension, SPIT$^+$ is still able to deliver good performance.

4.2 Query Performance

4.2.1 Window Queries

Next the performance of SPIT$^+$ in answering spatio-temporal window queries is compared against the R-tree+B-tree approach and a linear scan of the data. All approaches make use of the assumption that $\mathcal{T}^*$ is known at query time.

Figure 4.4 shows query performance as a function of the size of the spatial component of the window query, while Figure 4.5 shows the performance when varying the length of the temporal component. As expected, in both cases the performance of the linear scan is constant, as it depends only on the cardinality of the dataset. In all figures it is easy to see that the performance of the R-tree+B-tree approach degrades rather quickly, unlike for the other approaches. The case of the UNIFORM dataset is the only one where the R-tree+B-tree remains competitive with the linear scan for up to medium sized queries. For the GSTD and INFATI datasets the R-tree+B-tree is not competitive at all. This happens because for the GSTD dataset the density of the data in the occupied portion of the space is higher, causing the underlying R-tree to have more node overlaps and, consequently require more tree traversals. The case for the INFATI dataset is even more extreme, as even a simple linear scan performs relatively much better.

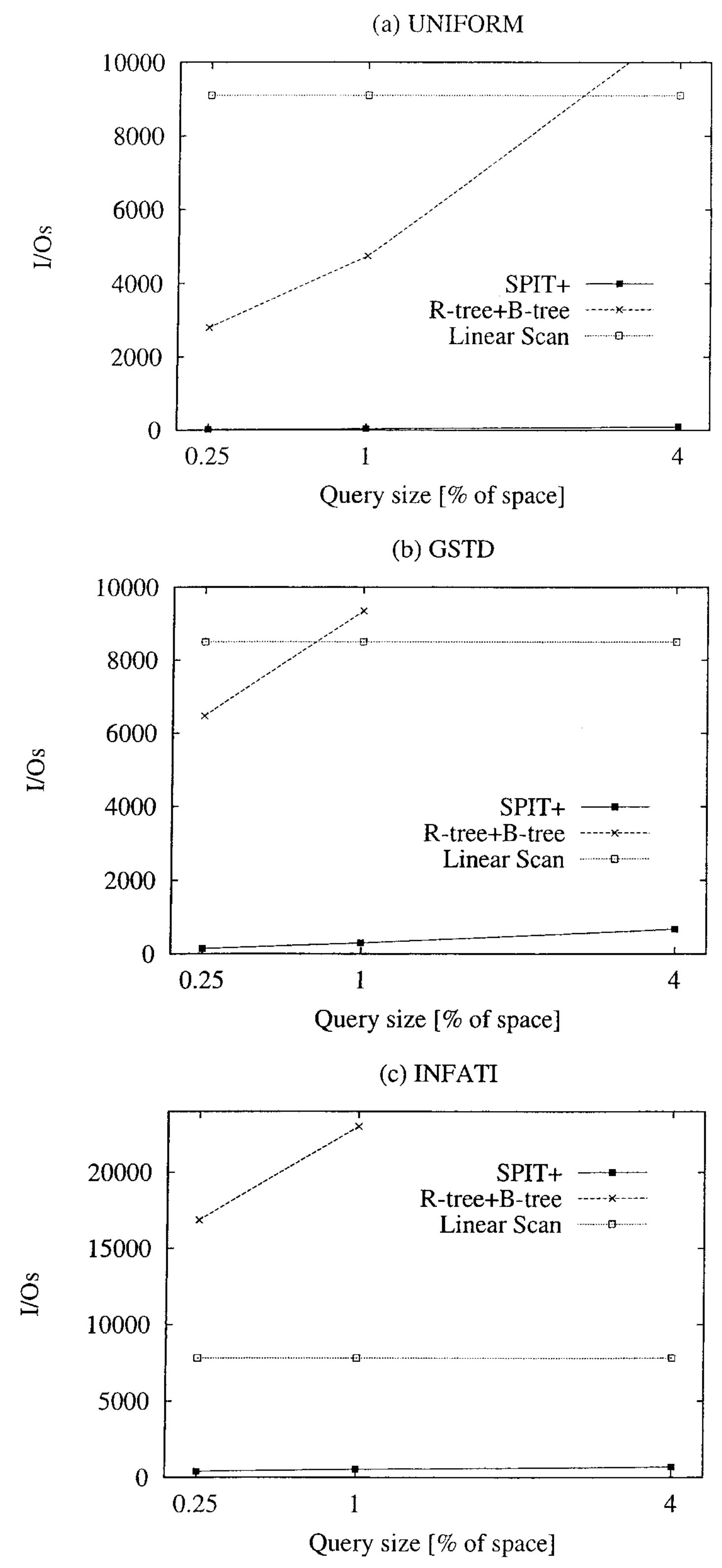


Figure 4.4: Comparing I/O performance as a function of the size of the spatial component of the query.

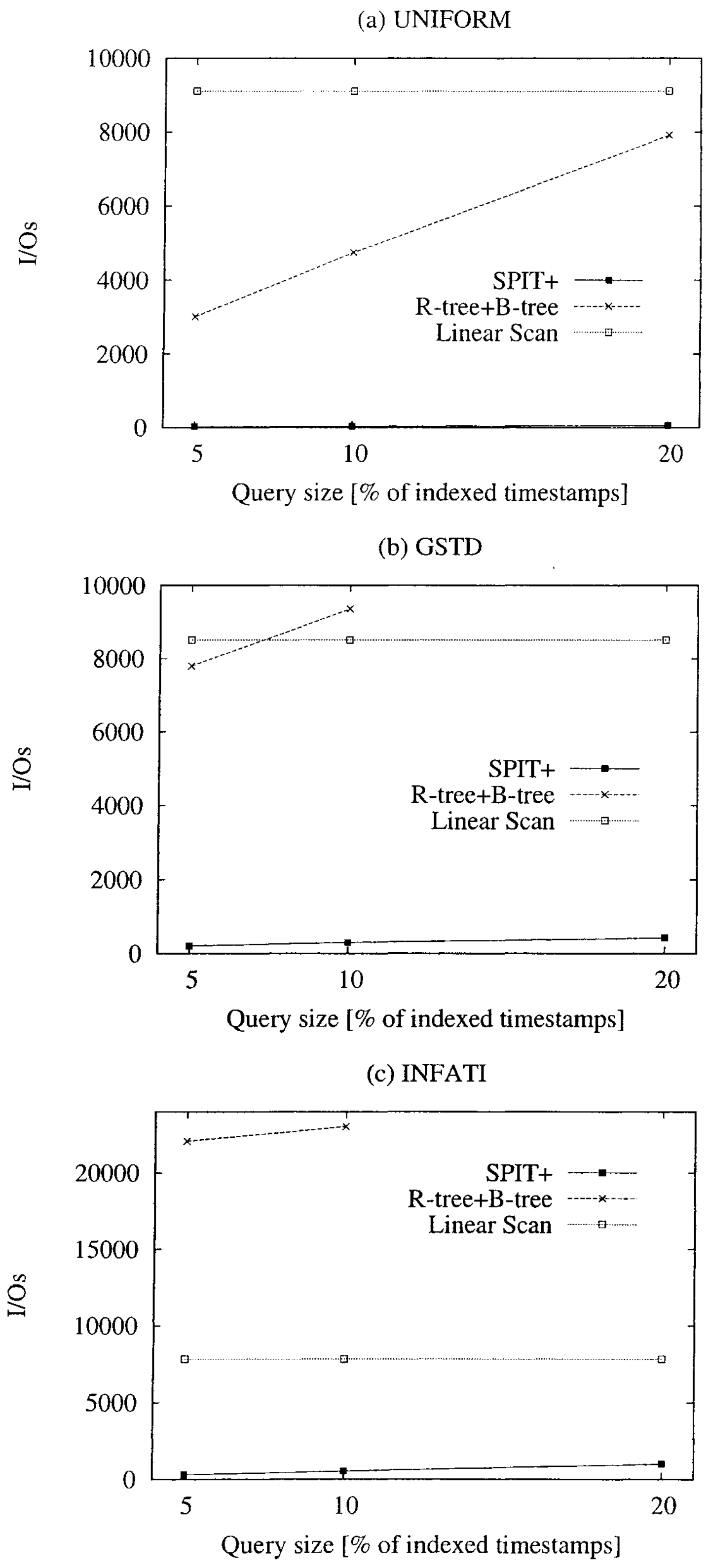
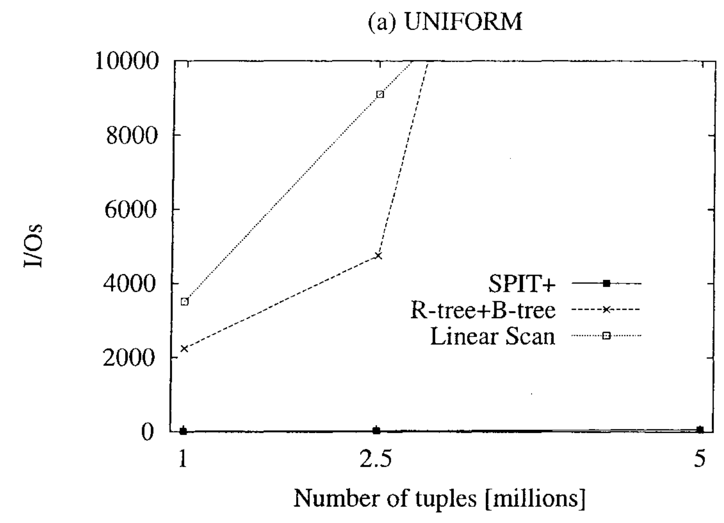


Figure 4.5: Comparing I/O performance as a function of the length of the temporal component of the query.

SPIT^+ consistently provides the best performance, being at least 10 times better than the other approaches. More importantly however, it is very robust with the increase of the query size for all distributions. This confirms that the proposed grid partitioning is able to cope well with variations in this parameter.



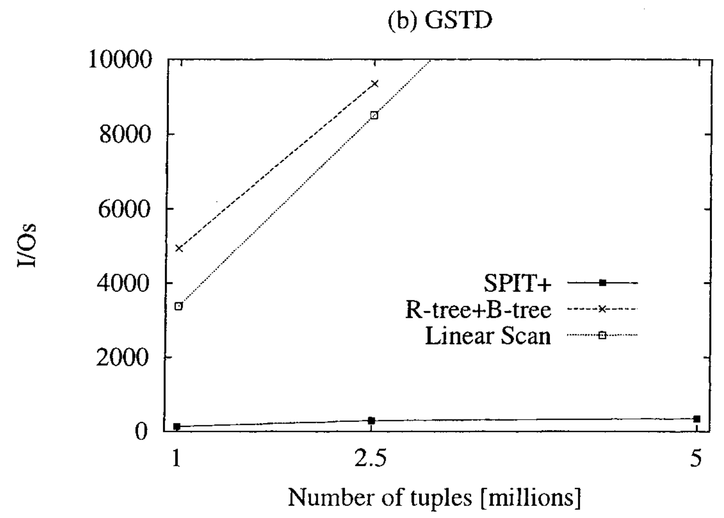


Figure 4.6: Comparing I/O performance as a function of the dataset cardinality.

SPIT^+ is also very robust with respect to the increase in the dataset size as can be seen in Figure 4.6. (Note that the INFATI data set was not used here as the dataset cardinality is fixed and an intrinsic part of the dataset features.)

The linear scan, as one would expect, does not scale well with the size of the dataset, and again the R-tree+B-tree approach is also a poor choice.

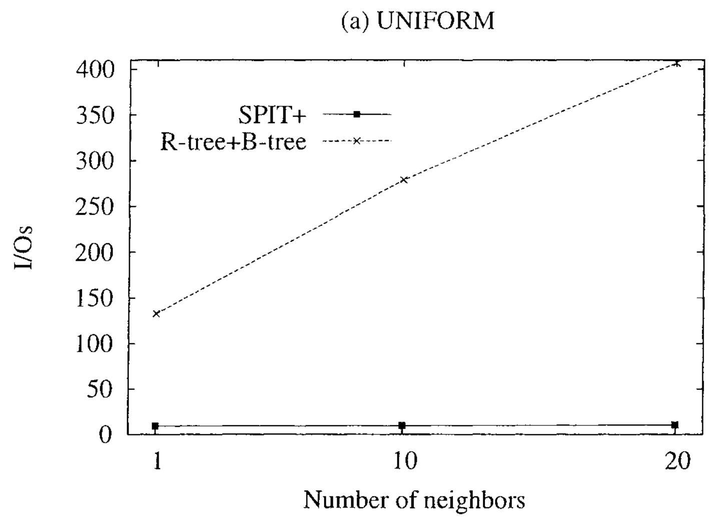
In summary, SPIT$^+$'s performance is often two or more orders of magnitude better than the other approaches, while being quite robust with respect to all parameters investigated. This is due to very effective filtering of heavily populated partitions that do not contribute to the query's answer, leading to highly efficient query processing.

4.2.2 kNN Queries

In this section, the efficiency of SPIT$^+$ in answering spatio-temporal kNN queries is evaluated by comparing it against the performance of the R-tree+B-tree approach. The *Linear Scan* method has much weaker performance (two orders of magnitude worse than SPIT$^+$'s performance) and has been omitted from the following plots.

Figure 4.7 shows the effect that the number k of required nearest neighbors has on the access cost of the index when the UNIFORM and GSTD datasets are used. The query size on the temporal dimension is set to 10% of the total number of recorded timestamps. The INFATI dataset has been left out of this type of experiments since it contains only 20 distinct objects and searching for 20 or even 10 nearest neighbors in a given time interval provides almost no discrimination on the spatial dimension. The performance of both SPIT$^+$ and R-tree+B-tree for one nearest neighbor search using the INFATI data is reported in the next section, when the effect of the temporal ranges length is studied.

Again, SPIT$^+$ delivers the best performance for both data distributions investigated and copes very well with the increase in the number of required neighbors. The difference between the number of I/Os required in SPIT$^+$ and in the R-tree+B-tree approach is, however, not as large as it was for window



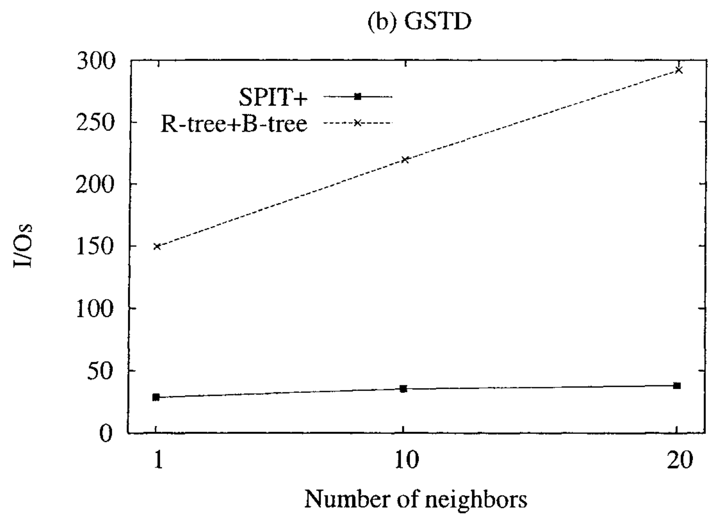


Figure 4.7: Comparing I/O performance as a function of the number of requested neighbors.

queries. To explain this observation, the way a kNN query is processed using an R-tree needs to be outlined first.

The kNN search algorithm used on the R-tree traverses the tree in a branch-and-bound manner. The algorithm keeps an ordered list of all the tree nodes that have to be visited and a list of the k closest neighbors found so far. The nodes list is ordered based on the euclidian distance, *minDist(q, MBR)*, between the query point q and the nodes MBRs. Initially, the list of nodes contain the root node and the list of closest neighbors is void. The tree is traversed by always visiting the first node in the nodes list. If the visited node is a directory node, all its children are added to the nodes list based on their *minDist(q, MBR)*; otherwise, the distance between the query point and all points contained in the leaf node and satisfying the query temporal predicate is computed and the k closest points are added to the list of k closest neighbors. Once the k-th point p from a leaf page is found, the distance *kNNDist(q,p)* between p and q is used to prune some of the nodes in the nodes list. Only those nodes having $minDist(q, MBR) \leq kNNDist(q,p)$ could contain points that are closer than the current k-th neighbor p. If such points are retrieved, the list of current k neighbors and the distance *kNNDist(q,p)* are updated accordingly. The algorithm stops when the nodes list is empty.

For the investigated query sets, the nearest distance *kNNDist(q,p)*, between the query point and its k-th closest neighbor, was usually much smaller than the sizes of the spatial component of the queries used in window queries evaluation, resulting in a smaller number of R-tree nodes that had to be accessed for kNN queries as compared with window queries. Moreover, as explained in Section 3.4, the size of the spatial component of the query used for data partitioning in SPIT$^+$ is based on some estimation, which may not be very accurate, especially for non-uniform data distributions. Hence, the smaller difference between the performances of SPIT$^+$ and R-tree+B-tree when processing kNN queries as compared with window queries.

The effect of the query temporal interval size on the number of I/Os required per query is shown in Figure 4.8. While for the UNIFORM and GSTD datasets the value used of k is set to the default value, for INFATI dataset k is set to 1 (recall that this dataset has only 20 distinct objects). As can be seen, the access cost for SPIT$^+$ increases slightly with the size of the temporal interval. In contrast, the number of accessed nodes for the R-tree+B-tree decreases as the length of the temporal ranges increases. The main reason is

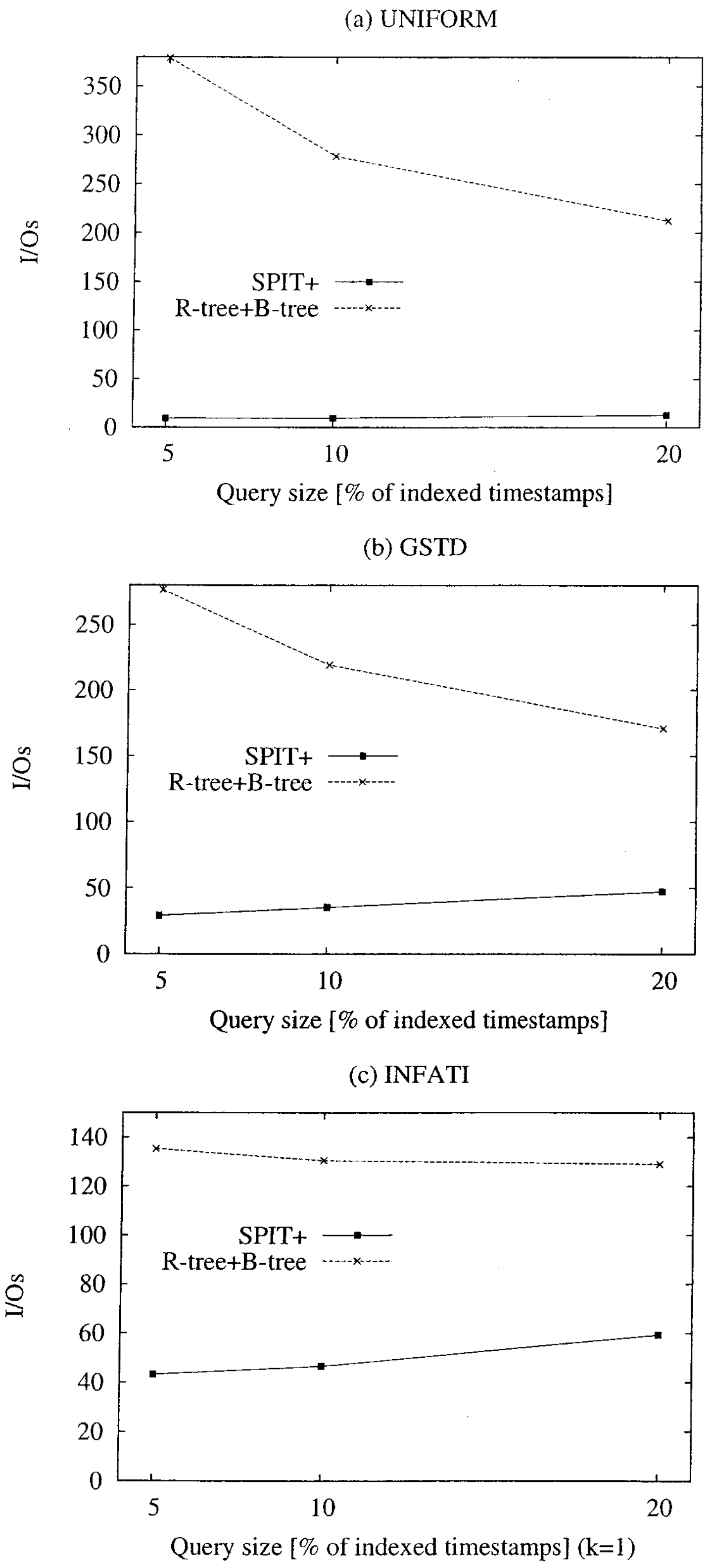


Figure 4.8: Comparing I/O performance as a function of the length of the temporal component of the query.

the following: the probability that the time intervals of data points found in the spatial neighborhood of the query point will intersect the query time range is greater when a larger query time range is considered. Hence, the nearest neighbor distance used by the R-tree search algorithm decreases with the increase of the query time range, resulting in accessing less nodes. However, the performance difference between the R-tree+B-tree approach and SPIT$^+$ is big enough to make SPIT$^+$ a very competitive approach, even for the largest considered query time interval, which is, indeed, a very large one.

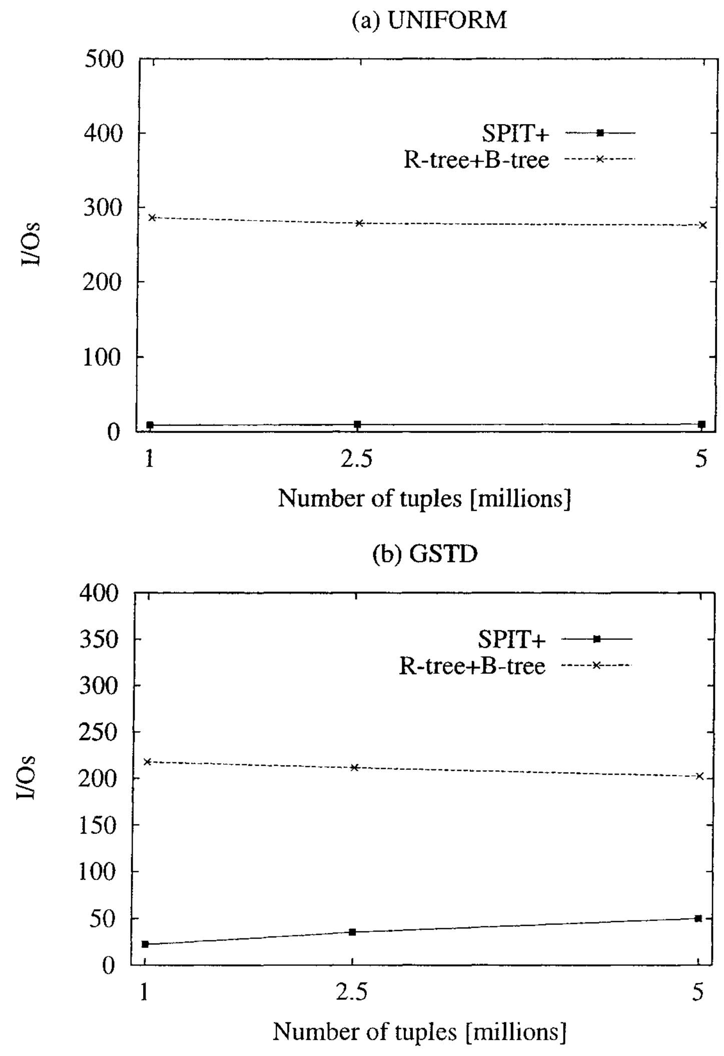


Figure 4.9: Comparing I/O performance as a function of the dataset cardinality.

Finally, SPIT$^+$'s performance in answering kNN queries is evaluated as a function of total number of objects in the dataset. As shown in Figure 4.9, the performance of both approaches scales very well with the size of the datasets. There is a very small improvement on the R-tree+B-tree performance, which is explained by the higher density of the data points in space when a larger dataset is used, resulting in smaller nearest neighbor distance. SPIT$^+$ outperforms the R-tree+B-tree, being better by at least a factor of four when the number of objects varies within the range 1M–5M.

4.2.3 Comparing with the MV3R-tree

Even though the main goal of this thesis is to provide a practical technique rather than a novel data structure for indexing spatio-temporal data, we also compare SPIT$^+$'s performance to the MV3R-tree [31]. Despite not being feasible to be implemented on top of existing RDBMSs, the MV3R-tree is arguably a good representative of special purposed indices for spatio-temporal data. The purpose of the experiments discussed next is to show that SPIT$^+$ can indeed offer performance at least comparable to a leading and specialized index structure. As the obtained MV3R-tree implementation does not support spatio-temporal kNN queries, only the performances in answering window queries have been compared.

Figures 4.10 and 4.11 shows the performance of SPIT$^+$ and the MV3R-tree when varying the size of the spatial and temporal component of the queries. While all other parameters remain at their default value as before, the GSTD dataset had to be downsized to 1 million observations as the obtained MV3R-tree source code was somehow unable to cope with larger datasets. The page size used to construct the MV3R-trees has been set to the same value as in SPIT$^+$; the other MV3R-tree parameters, like the strong/weak version overflow thresholds, have been kept to their default values.

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

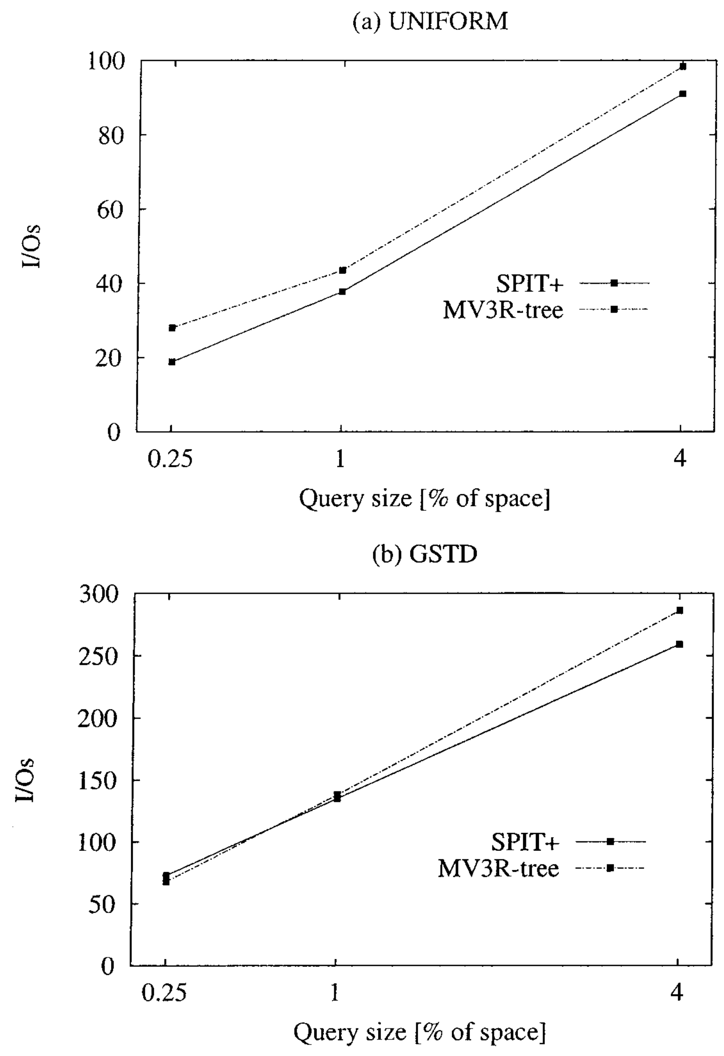
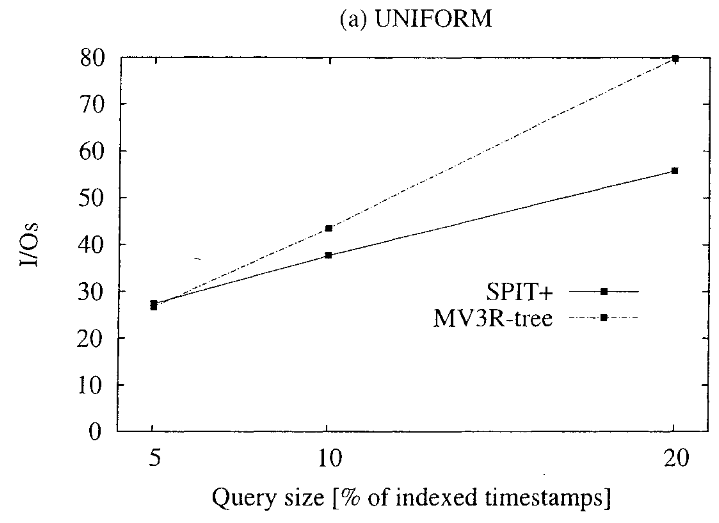


Figure 4.10: Performance of SPIT$^+$ vs. MV3R-tree when varying the size of the query's spatial component.

In terms of scalability, the inability of handling large datasets in the case of the MV3R-tree makes a fair comparison not possible. The reason being that for very small datasets, say in the order of up to a few hundred thousands of



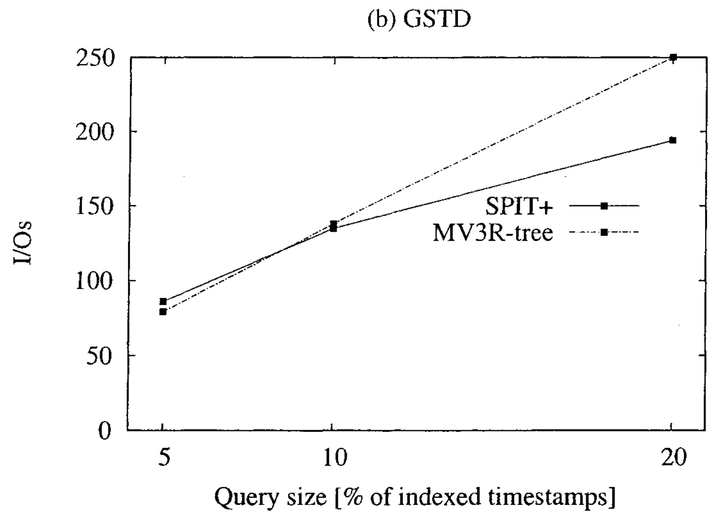


Figure 4.11: Performance of SPIT$^+$ vs. MV3R-tree when varying the length of the query's temporal component.

observations, there is an inherent overhead within SPIT$^+$ due to the underlying DBMS which is not present within the MV3R-tree. One should note though, that it is well known that for very small datasets a trivial linear scan is often the most efficient solution.

4.3 Index Creation

The final set of experiments deal with time required to create and index the database. In this regard, the Linear Scan approach is obviously the most efficient since there is no overhead associated to it. This comes at the expense of quite inefficient query performance as discussed above.

The times reported for index creation were obtained on a PC with an AMD Athlon XP 3200+ running at 2.19GHz and with 1.00GB of RAM, and using the GSTD dataset with 2.5 million objects. The partitioning was determined using the default query sizes on the spatial and temporal domains. When varying the size of the dataset, the time to build the index grows at a linear rate with the data volume. The results using the other data distributions follow the same trend.

There are two main tasks that need to be performed within SPIT$^+$. First, the partitioning must be obtained using the heuristic algorithm presented in Section 3.2.1. After that, the objects need to be inserted into the correct table partitions, i.e., the index-organized tables. The first parts took 76 sec. while the second required 843 sec. for a total of 919 sec. The R-tree+B-tree approach, on the other hand, needed only 200 sec. to insert the data on the (single) table but needed 784 sec. to build the associated indexes, for a total of 984 sec. It should be noted that both approaches made use of the SQL*Loader facility available in typical ORACLE installations.

Even though SPIT$^+$ is overall about 7% faster than the R-tree+B-tree approach, it was observed that in SPIT$^+$ the partitions lookup, i.e., finding in which partition an object should be inserted, caused most of the overhead at data insertion time. The idea of using an index, e.g., an R-tree, for the grid partitions themselves in order to speed up the partition lookup process, has also been considered. However, the number of partitions was fairly low (in the order of hundreds) for all experiments and it would most likely not benefit from an index, as compared to a simple linear scan of the partitions table.

Chapter 5

Conclusions and Future Work

This thesis addresses the issue of providing indexing support for historical spatio-temporal data, which is not only efficient from a query processing perspective but also practical, in particular that it can easily be implemented on top of any RDBMS using only standard facilities.

Based on a two-level structure that separates the temporal aspect from the spatial aspect of the data, the proposed indexing method, SPIT$^+$, enhances SPIT [16] with several optimization techniques to offer improved search performance. It uses a refined cost model that aims at optimizing the query cost in terms of number of disk accesses required to answer a query. For the case of a uniform data distribution the cost model provides an optimal partitioning of the dataset. For arbitrary data distributions, the cost model is used to provide a new heuristic partitioning which leads to very good query performance in practice. In addition, SPIT$^+$ offers the possibility of pre-processing the data, splitting the temporal ranges of some observations, in order to further improve performance.

An extensive empirical evaluation over both real and synthetic datasets have been performed, which demonstrates that SPIT$^+$ is both effective and efficient, outperforming other spatio-temporal data management alternatives for both window and kNN queries. It also shows that SPIT$^+$ is robust with respect to the query size assumed at index construction time.

5.1 Future Research

There are several directions that could be explored for future research. Some interesting ideas include:

- Expanding SPIT$^+$ to deal with new observations. As new data is appended to a previously indexed dataset, the index search performance will start to deteriorate especially if the data distribution changes considerably. As mentioned in Section 3.6, this problem could be solved by either re-constructing the whole index or using a series of archival indexes, each for a given time frame and fine tuned for the dataset distribution and size corresponding to that time frame. The latter approach has the advantage that only a small subset of the whole dataset has to be considered when each such index is created, and that previous indexes do not have to be rebuilt. A related idea would be developing a technique to automatically determine the point in time when the index performance degrades to a certain degree as the database is appended with new observation or major changes in the query workload occur.

- Extending the proposed SPIT$^+$ approach in order to handle trajectories. The research question to be investigated in this case is how to obtain a cost model to guide an optimal partitioning given a set of trajectory segments. Since a trajectory segment may cross several spatial partitions, some segments would have to be replicated and inserted into each intersected partition, increasing the total number of indexed segments. The cost model would have to be designed in such a way that this increase is taken into account.

- Optimizing SPIT$^+$ performance in answering kNN queries. Even though the experimental evaluation showed that SPIT$^+$ is very efficient, outperforming the R-tree based approach by factors of as much as 10, a better nearest neighbor distance estimation, which take into account the data distribution, would further improve query performance. Yet another nearest neighbor search related topic would be developing algorithms for processing kNN queries where the query object is also moving.

- Augmenting SPIT$^+$ so that not only historical but current spatio-temporal data could be efficiently indexed. When current observation are considered, the time-interval end value, t_e, of all database entries recording the current position of each mobile object is set to the so-called "now" value. When the current location of an object changes, the entry associated with its last recorded location has to be retrieved, this entry time interval has to be updated, and a new entry recording the new location has to be inserted into the database. The topic of having indexing structures, the B$^+$-tree included, able to sustain very high update ratios, e.g., several millions of updates per second, is still an open problem. Nevertheless, perhaps an approach similar to the one used within SETI, where a memory-resident, "front index" is used to alleviate the problem, could be adapted for use within SPIT$^+$.

Bibliography

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

[2] Rimantas Benetis, Christian S. Jensen, Gytis Karciauskas, and Simonas Saltenis. Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. In *In Proc. of the 2002 International Symposium on Database Engineering & Applications*, pages 44–53, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Stefan Berchtold, Christian Böhm, Daniel A. Keim, Florian Krebs, and Hans-Peter Kriegel. On Optimizing Nearest Neighbor Queries in High-Dimensional Data Spaces. In *ICDT*, pages 435–449, 2001.

[4] F. W. Burton, J. G. Kollias, D. G. Matsakis, and V. G. Kollias. Implementation of overlapping B-trees for time and space efficient representation of collections of similar files. *Comput. J.*, 33(3):279–280, 1990.

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

[6] Elias Frentzos, Kostas Gratsias, Nikos Pelekis, and Yannis Theodoridis. Nearest Neighbor Search on Moving Object Trajectories. In *SSTD*, pages 328–345, 2005.

[7] A. Guttman. R-trees: a dynamic index structure for spatial searching. In *Proc. of the ACM SIGMOD Conf.*, pages 47–57, 1984.

[8] Marios Hadjieleftheriou, George Kollios, Vassilis J. Tsotras, and Dimitrios Gunopulos. Efficient Indexing of Spatiotemporal Objects. In *In Proc. of the 8th International Conference on Extending Database Technology*, pages 251–268, London, UK, 2002. Springer-Verlag.

[9] Gísli R. Hjaltason and Hanan Samet. Distance Browsing in Spatial Databases. *ACM Trans. Database Syst.*, 24(2):265–318, 1999.

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

[11] C.S. Jensen, D. Lin, and B.-C. Ooi. Query and Update Efficient B^+-Tree Based Indexing of Moving Objects. In *Proc. of VLDB*, pages 768–779, 2004.

[12] George Kollios, Dimitrios Gunopulos, and Vassilis J. Tsotras. Nearest Neighbor Queries in a Mobile Environment. In *In Proc. of the International Workshop on Spatio-Temporal Database Management*, pages 119–134, London, UK, 1999. Springer-Verlag.

[13] P.M. Lewis, A.B., and M. Kifer. *Database and Transaction Processing.* Addison-Wesley, 2002.

[14] Li, Yang, and Han. Continuous k-Nearest Neighbor Search for Moving Objects. In *Proceedings of SSDBM04*, 2004.

[15] Dan Lin, Christian S. Jensen, Beng Chin Ooi, and Simonas Saltenis. Efficient indexing of the historical, present, and future positions of moving objects. In *In Proc. of the 6th international conference on Mobile data management*, pages 59–66, New York, NY, USA, 2005. ACM Press.

[16] D. Mallett. Relational Database Support for Spatio-Temporal Data. Technical Report TR04-21 (M.Sc. Thesis), Dept. of Computing Science, Univ. of Alberta, 2004. [http://www.cs.ualberta.ca/TechReports/2004/TR04-21/TR04-21.pdf].

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

[18] M.A. Nascimento and M. Dunham. Indexing valid time databases via B+ -trees – the MAP21 approach. *IEEE TKDE*, 11(6):1–19, 1999.

[19] M.A. Nascimento and J.R.O. Silva. Towards historical R-trees. In *Proc. ACM SAC*, pages 235–240, 1998.

[20] Mario A. Nascimento, Jefferson R. O. Silva, and Yannis Theodoridis. Evaluation of Access Structures for Discretely Moving Points. In *STDBM '99: Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 171–188, London, UK, 1999. Springer-Verlag.

[21] Oracle Corporation. *Oracle Database SQL Reference 10g Release 1 (10.1)*, December 2003.

[22] D. Pfoser, C.S. Jensen, and Y. Theodoridis. Novel Approaches in Query Processing for Moving Object Trajectories. In *Proc. of VLDB*, pages 395–406, 2000.

[23] Katerina Raptopoulou, Apostolos Papadopoulos, and Yannis Manolopoulos. Fast Nearest-Neighbor Query Processing in Moving-Object Databases. *GeoInformatica*, 7(2):113–137, 2003.

[24] Slobodan Rasetic, Jörg Sander, James Elding, and Mario A. Nascimento. A Trajectory Splitting Model for Efficient Spatio-Temporal Indexing. In *In Proc. of the 31st international conference on Very large data bases*, pages 934–945. VLDB Endowment, 2005.

[25] S.M. Ross. *Introductory Statistics.* McGraw-Hill, 1996.

[26] S. Saltenis et al. Indexing the Positions of Continuously Moving Objects. In *Proc. of the ACM SIGMOD Conf.*, pages 331–342, 2000.

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

[28] A. Prasad Sistla, Ouri Wolfson, Sam Chamberlain, and Son Dao. Modeling and Querying Moving Objects. In *ICDE '97: Proceedings of the Thirteenth International Conference on Data Engineering*, pages 422–432, Washington, DC, USA, 1997. IEEE Computer Society.

[29] Zhexuan Song and Nick Roussopoulos. K-Nearest Neighbor Search for Moving Query Point. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 79–96, London, UK, 2001. Springer-Verlag.

[30] Zhexuan Song and Nick Roussopoulos. SEB-tree: An Approach to Index Continuously Moving Objects. In *MDM '03: Proceedings of the 4th International Conference on Mobile Data Management*, pages 340–344, London, UK, 2003. Springer-Verlag.

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

[32] Y. Tao, D. Papadias, and Q. Shen. Continuous Nearest Neighbor Search. In *In Proc. of VLDB*, pages 287–298, 2002.

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

[34] Yufei Tao and Dimitris Papadias. Time-parameterized queries in spatio-temporal databases. In *In Proc. of the 2002 ACM SIGMOD international conference on Management of data*, pages 334–345, New York, NY, USA, 2002. ACM Press.

[35] Yufei Tao, Jun Zhang, Dimitris Papadias, and Nikos Mamoulis. An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces. *IEEE Transactions on Knowledge and Data Engineering*, 16(10):1169–1184, 2004.

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

[37] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the Generation of Spatiotemporal Datasets. In *Proc. of SSD*, pages 147–164, 1999.

[38] Y. Theodoridis, M. Vazirgiannis, and T.K. Sellis. Spatio-Temporal Indexing for Large Multimedia Applications. In *Proc. of IEEE ICMCS*, pages 441–448, 1996.

[39] Yannis Theodoridis, Timos K. Sellis, Apostolos Papadopoulos, and Yannis Manolopoulos. Specifications for Efficient Indexing in Spatiotemporal Databases. In Maurizio Rafanelli and Matthias Jarke, editors, *10th International Conference on Scientific and Statistical Database Management, Proceedings, Capri, Italy, July 1-3, 1998*, pages 123–132. IEEE Computer Society, 1998.

[40] X. Xu, J. Han, and W. Lu. RT-Tree: An Improved R-tree Index Structure for Spatiotemporal Databases. In *Proc. of the 4th Intl. Symposium on Spatial Data Handling*, pages 1040–1049, 1990.

[41] Baihua Zheng and Dik Lun Lee. Semantic Caching in Location-Dependent Query Processing. In *SSTD '01: Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*, pages 97–116, London, UK, 2001. Springer-Verlag.