Self-Tunable Spatio-Temporal B$^+$-tree Index for Moving Objects

Su Chen Beng Chin Ooi Kian-Lee Tan
School of Computing
National University of Singapore
Singapore
{chensu, ooibc, tankl}@comp.nus.edu.sg

Mario A. Nascimento
Department of Computing Science
University of Alberta
Edmonton, Alberta, Canada
mn@cs.ualberta.ca

November, 2007

Abstract

Database systems are increasingly being used to manage dynamically changing databases. One such application is a moving object database where the database and the workload change frequently: as the locations of moving objects change in space and time, the data distribution also changes; answers for the same query over the same region may vary widely in size over time. As a result, traditional static indexes cannot perform well, and it is critical to develop self-tuning indexes that can be reconfigured automatically based on the state of the system.

In this paper, we present the ST^{2}B-tree, a *S*elf-*T*unable *S*patio-*T*emporal *B*$^+$-Tree index to manage moving objects. The ST^{2}B-tree is amenable to tuning: frequent updates to the two subtrees allows the opportunity to rebuild the subtree with a different set of reference points and granularity of grid size easily without significant overhead. We propose an online tuning framework for the ST^{2}B-tree. The tuning procedure is conducted automatically online without intervention from database administrators (DBAs) or interfering with any regular functions of the DBMS. Our experimental results verify that the self-tuning process lessens the degradation in the effectiveness of the index caused by workload changes while incurring an extremely low overhead.

1 Introduction

Database tuning is crucial to the efficient operation of a database management system (DBMS). Almost every commercial DBMS, such as Microsoft SQL Server, IBM DB2 or Oracle, provides some tuning tools. The goal of tuning is to maintain the database always in a near "optimal" state. Variations in workload, including both queries and updates, can significantly impact the performance of the database. Usually, some components of the database, such as the indexes and the query optimizer, can be configured to adapt to workload changes.

Traditionally, the database administrator (DBA) was fully responsible for tuning the system to ensure optimal performance. However, it is impractical for the DBA to keep watch on the system all the time. To ease the burden on the DBAs, the only practical solution is to make a database self-tuning so that tuning proceeds automatically with minimal human intervention.

While some works have been done to develop self-tuning technologies in database systems, these works are largely restricted to traditional static databases. However, there are a number of emerging applications (e.g., Geographical Information System (GIS) and location aware applications such as traffic monitoring) that manage highly dynamic data. In particular, in *M*oving *O*bjects *D*atabases (MOD), a large number of objects are continuously moving, and their locations have to be frequently updated. Moreover, the distribution of moving objects varies over time and space. For example, in a traffic management system, objects are crowded in some places but less so in others. Moreover, the number of vehicles at certain locations is large in the day and relatively fewer at night. This means that the workload for the same query over the same region may be different at different time. Thus, there is a need to re-examine self-tuning methods for managing dynamic databases.

In this paper, we focused on designing self-tuning indexes to support MODs. Existing works on indexing MODs mostly focus either on designing indexing structures or developing efficient algorithms for various kinds of queries. Variability in data workload, i.e., cardinality and distribution of objects, has so far been overlooked in the design of moving objects indexes. On the other hand, while self-tuning indexes (e.g., [4]) have been designed to adapt to workload variations without the DBA's intervention, these are focused on relatively static databases. To the best of our knowledge, no previous work has investigated the problem of index tuning in the area of moving objects indexing.

Our contributions of this paper are:

- We examine three kinds of data diversities in moving objects databases and specify the impact on a moving object index based on space partitioning.

- We present a *S*elf-*T*unable *S*patio-*T*emporal index for moving objects, reusing the classical B^+-tree. The ST^{2}B-tree itself is amenable to tuning with respect to the way and granularity of space partitioning. No modification to the basic structure is required.
- We introduce an online tuning framework. In the framework, the ST^{2}B-tree tunes itself based on data variations. The tuning is performed online with low overhead. We also provide a guideline for determining parameters for the ST^{2}B-tree.
- We have conducted an exhaustive experimental study to evaluate the performance of the proposed self-tunable index. The results show that the self-tuning process lessens the degradation in the effectiveness of the index and hence improves the overall performance.

The remainder of the paper is organized as follows: Section 2 reviews some related works. Section 3 specifies the challenges that motivate the self-tuning of a moving object index. Section 4 presents the ST^{2}B-tree. Section 5 introduces the online tuning framework and how it works. Section 6 shows our experimental results. Finally, Section 7 concludes the paper.

2 Related Work

Existing moving object indexes can be classified into two categories. One is based on data partitioning, such as the TPR-tree [15] and its variant the TPR*-tree [17]. This type of indexes organizes objects in a R-tree-like structure. A *M*inimum *B*ounding *R*ectangle (MBR) is used to bound objects that are close to one another and that can be stored in a page. The MBRs are organized hierarchically in an R-tree-like structure. At the bottom level of the hierarchical structure, split occurs if too many objects are contained in a page. While two neighboring MBRs are merged into one if all objects can be stored in a single page. Data partitioning arises because each bottom level MBR contains no more than a given number of objects. Therefore, the regions in which objects are crowded always consist of many small regions bounded by leaf MBRs. Sparse regions are typically covered by few large MBRs.

The second category of indexes relies on space partitioning, such as the B$^+$-tree based indexes [8, 9, 20] and the grid based proposals [11, 13, 19]. These indexes partition space with a grid in advance. An object is indexed by the cell it belongs to. The indexing strategy completely ignores the feature (distribution) of objects.

Space partitioning based indexes surpass their counterparts that are based on data partitioning in two ways. First, both the B^+-tree and the grid index are well established indexing structures present in virtually every commercial DBMS. The index can be integrated into an existing DBMS

easily. No additional physical design is required to modify the underlying index structure, concurrency control and the query execution module of the DBMS. Second, in comparison with spatial indexes such as the R-tree, operations such as search, insertion and deletion on the B^+-tree and the grid index can be performed very efficiently.

The superiority is more remarkable especially in a concurrent environment. Because concurrency control in the R-tree-like indexes is more complex and time consuming, the R-tree based indexes are therefore not as scalable for real moving objects applications, where frequent updates and queries arrive simultaneously. Jensen, Lin and Ooi [8] have shown that the throughput of the TPR-tree [15] does not scale up in concurrent operations due to the fact that the R-tree based indexes hold the lock longer during updates. Guo et al. [7] have also shown that the pre-processing and tree optimization strategies employed in the TPR*-tree [17] results in extra delay in locking, and hence reduces the performance gain in query processing due to the preprocessing during insertions. They propose using the Buddy-tree [16] as the alternate structure to trade the data partitioning and the use of MBR in object bounding for a similar space partitioning R-tree like index.

However, despite the above deficiencies, R-tree based indexes such as the TPR-tree are less susceptible to data diversities and changes as a result of MBRs splitting and merging. In contrast, because existing space partitioning indexes, such as the B^x-tree [8], partition space using a single uniform grid, the workload across different parts of the index may not be balanced. Such imbalance does impact the performance of existing indexes based on space partitioning.

To address the imbalance of space partitioning, STRIPE [14] utilizes the conventional quad-tree as the underlying index. The quad-tree is a space partitioning index that is adaptive to data distribution. The space partitioning is guided by data distribution, i.e., the way of space partitioning is fixed but only dense regions are partitioned. However, different from the B$^+$-tree, the quad-tree is an unbalanced structure. The objects in the dense region are stored deeply in the tree. Updates and queries on these objects always incur higher overhead. Even using such distribution-adaptive indexes such as the quad-tree, the performance is still unsatisfactory when dealing with such data skew in moving objects databases. To avoid performance deterioration caused by imbalance and likely changes in workload, it is therefore crucial to tune the index explicitly.

2.1 Index Moving Objects Using the B$^+$-tree

The B^x-tree [8] is the first effort to adapt the B$^+$-tree to index moving objects. Each cell of the grid is assigned a unique id with the help of a space filling curve. Objects are indexed in a B$^+$-tree with the id of the cell it belongs to.

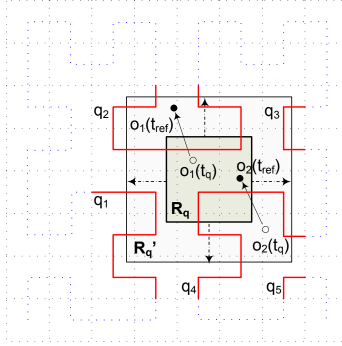


Figure 1: Query Processing in the B^x-tree

The time dimension is partitioned into intervals depending on the maximum time duration between two updates of an object. Each time interval has a certain reference time. A portion of the B$^+$-tree is reserved for each time interval. An object is indexed in the phase with its location at the reference time. An update deletes the old entry in the valid phase that covers the last updating time of the object and then inserts the new record into the newest phase. As time passes, the oldest phase expires and a new one is appended.

To process a range query, the B^x-tree searches each valid phase with an enlarged query region. The query region is expanded to the corresponding reference time according to the maximum velocity of objects. For example in Figure 1, given a range query $R_q = (\overrightarrow{x_1}, \overrightarrow{x_2})$ at t_q, the enlarged query R'_q at t_{ref} is:

$$R'_q = (\overrightarrow{x_1} - \overrightarrow{max_v} \cdot |t_{ref} - t_q|, \overrightarrow{x_2} + \overrightarrow{max_v} \cdot |t_{ref} - t_q|) \quad (1)$$

where $\overrightarrow{x_1}$ and $\overrightarrow{x_2}$ are the lower-left and the upper-right corner of the of R_q. $\overrightarrow{max_v}$ represents maximum velocity. Query enlargement is essential to avoid false negatives. Objects outside R'_q at t_{ref} cannot appear in R_q at t_q due to maximum velocity constraints. In Figure 1, at t_{ref}, o_1 is outside R_q and o_2 is inside. In contrast, o_1 is actually inside R_q while o_2 is outside R_q at t_q. However, since both o_1 and o_2 are inside the enlarged region R'_q at T_{ref}, there is no omission in the result.

The use of the space filling curve in deriving the $1d$ cell id destroys location proximity to some extent. A $2d$ range query is transformed into several $1d$ range queries. In Figure 1, the enlarged query region R'_q covers five $1d$ queries q_1 to q_5 which are shown as thick lines (note that q_5 is a single value query).

In [8], global maximum speed is used for query enlargement, and this could mean that oversized enlarged queries are used in regions with moving objects. In [9] Jensen, Tiesyte and Tradisauskas improve the B^x-tree query efficiency through more conscious query enlargements.

Recently, Yiu, Tao and Mamoulis present the B^{dual}-tree [20] which also uses a B$^+$-tree to index moving objects, like the B^x-tree. The B^{dual}-tree indexes objects in a dual space instead. A $2d$ moving object $o(x_1, x_2)$ is a $4d$ point $o(x_1, x_2, v_1, v_2)$ in dual space (x_i and v_i represent the coordinates and velocity in i-th dimension). The space partitioning is applied to the dual space, which means that both location and velocity are considered in deriving the $1d$ key. Unlike the B^x-tree, the B^{dual}-tree uses R-tree-like query algorithms as in the TPR-tree. Each internal entry in the B$^+$-tree maintains a set of *MO*ving *R*ectangles (MOR), indicating the spatial region and range of velocity covered by the subtree of the entry. A range query searches the subtree of an internal entry only if any of its MORs intersects with the query region. Likewise, a kNN query in the B^{dual}-tree is processed in a branch-and-bound manner just like that in the TPR-tree. By partitioning the velocity space, the B^{dual}-tree improves the query performance of the B^x-tree. However, maintaining the MORs introduces high computation workload, which slows down the fast update of the B$^+$-Tree. In a concurrent environment, the throughput is also lower, since during an update an internal node has to be locked for a longer period until the MORs are updated to ensure consistency between its entries and their MORs. In addition, unlike the B^x-tree, the B^{dual}-tree modifies the query algorithm of the B$^+$-tree, which can no longer be readily integrated into existing commercial DBMSs.

3 Challenges in MOD

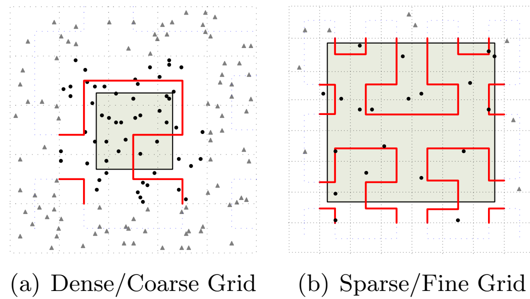
In this section, we first examine the impact of data density and granularity of space partition on index performance. Next, we present three kinds of data diversity in moving objects databases. Object diversities hinder an index from being "optimal". Index degradation caused by differences and changes of data engenders the demand for tuning the index online.

3.1 Impact of Data Density and Space Partition

To use the B$^+$-tree for indexing spatial data, the first step is to reduce dimensionality, i.e., mapping spatial data into $1d$ data. Typically, space is partitioned into small grid cells. Each cell is assigned a unique key for indexing, using a space filling curve for example. A $2d$ query is transformed into several $1d$ range queries that can be evaluated over the B$^+$-tree. The data density and granularity of space partition exert a joint effect on the index performance.

3.1.1 High Density vs. Coarse Grid

If the density of objects is high, or a coarse grid is used to partition space, each cell contains many objects and we have to check all objects in the cells



(a) Dense/Coarse Grid (b) Sparse/Fine Grid

Figure 2: The Co-relation of Data Density and Grid Granularity, Impact on Query Processing

that intersect with the query. Large number of false positives are incurred in the boundary cells. For example, in Figure 2(a), all objects (solid circles) in the 3×3 cells that intersect with the query (the dark square) must be examined. On the other hand, since objects in the same cell have the same indexing key, a cell with too many objects may incur overflow pages. This means that update cost will be higher as overflow pages have to be read and searched. The existence of too many overflow pages also compromises the balance property and bounded search cost of the B^+-tree structure.

3.1.2 Low Density vs. Fine Grid

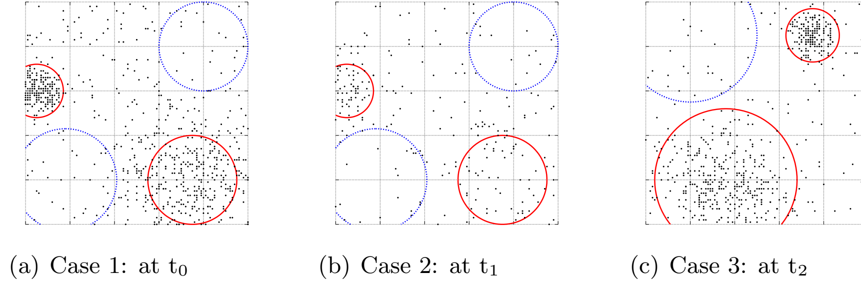
At the opposite end, if the density of objects is low, or a fine grid is used to partition space, few objects are contained in a cell. For example, we can partition the space in Figure 2(a) with a finer grid. Figure 2(b) zooms in on the query region in Figure 2(a). Now, most cells contain no more than 1 object. Obviously, no additional update overhead is incurred. The number of false positives also decreases in query processing because the boundary cells become smaller and contain fewer objects. However, the number of $1d$ range queries needed increases (from 2 to 9 in Figure 2, shown as red, thick lines). Although time is saved for pruning false positives, the increase in the number of $1d$ range queries deteriorates query performance.

3.2 Types of Data Diversity

Intrinsically, moving objects are spatial objects changing positions with time. The differences and changes of data fall into the following three categories as illustrated in Figure 3.

3.2.1 Diversity in Space

In general, the density of moving objects varies in different areas in space. Hotspots, such as commercial centers and major road junctions, always have



(a) Case 1: at t_0 (b) Case 2: at t_1 (c) Case 3: at t_2

Figure 3: Spatial and Temporal Data Diversity

higher object density than other places. Such a situation is portrayed in Figure 3(a). At time instance t_0, we have two hotspots enclosed in red, solid circles and the blue, dashed circles indicate regions with relatively lower object density.

3.2.2 Diversity with Time

In moving objects environment, the quantity of moving objects changes with time. For example, mass of vehicles travel in the day, causing heavy traffic load on the roads. In contrast, traffic is relative light at night. From time t_0 (Figure 3(a)) to t_1 (Figure 3(b)), hotspots are still hotspots; however, the density of all areas decreases significantly.

3.2.3 Diversity in Space with Time

As a combination of the above two kinds of diversities, both the density and distribution of moving objects change with time. From time t_1 (Figure 3(b)) to time t_2 (Figure 3(c)), besides an increase in the total number of objects, the hotspots move as well. The sparse areas may become dense while the dense areas may become sparse due to some external factors such as peak hours, road work and accidents. In a real scenario, from 8am to 9am, people drive from the residential suburbs to downtown. During office hours, most vehicles move in and around the downtown area. After 5pm, people start trickling home. The residential suburbs and downtown behave as hotspots alternately.

In summary, the object density and granularity of space partitioning greatly affect the efficiency of an index. Considering the three kinds of diversities in MODs, an index suffers from performance degradation if it partitions space evenly using a uniform grid consistently. As a result, a good moving object index must:

1. Discriminate between regions of different densities,

2. Adapt to density and distribution changes with time.

4 ST²B-tree: A Self-Tunable Moving Object Index

In this section, we first introduce the structure and basic query algorithm of the ST²B-tree. Then we explain why the ST²B-tree satisfies the two requirements of an adaptive index for moving objects outlined in earlier section.

4.1 ST²B-tree Structure

The ST²B-tree is built on the B$^+$-tree without any changes to the underlying B$^+$-tree structure and insertion/deletion algorithms. It indexes moving objects as $1d$ data points. A moving object is a spatio-temporal point in its natural space. The $1d$ key is composed of two components: KEY_{time} and KEY_{space}.

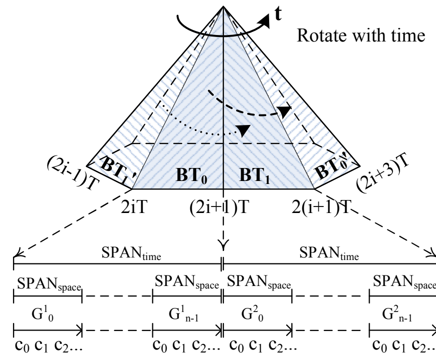


Figure 4: The Essence of the ST²B-tree

4.1.1 Index with Time

Assume that an object is updated at least once in time interval T. The ST²B-tree logically splits the B$^+$-tree into two subtrees, BT_0 and BT_1. Each subtree is assigned a range of T time. Specifically, the time ranges covering BT_0 and BT_1 are $[2iT, (2i{+}1)T)$ and $[(2i{+}1)T, (2i{+}2)T)$ respectively, where $i = 0, 1, 2\ldots$ As time elapses, the value of i increases and the time ranges of the two subtrees roll over alternately. The index therefore rolls over and self-adjusts with time. This behavior is illustrated in Figure 4. Suppose an object o issues an update $(\overrightarrow{x}, \overrightarrow{v})$ at t_{up}, where $\overrightarrow{x}$ and $\overrightarrow{v}$ represent the location and velocity of the object at t_{up}. The object will be indexed in the subtree whose time range covers t_{up}. For instance, updates issued in $[0, T)$ are indexed in the first subtree BT_0 while those in $[T, 2T)$ fall into the right subtree BT_1. Subsequent updates in $[2T, 3T)$ go back to BT_0, and so on and so forth.

Each subtree has a unique reference time T_{ref} and o is indexed with its location at T_{ref}, $\overrightarrow{x}' = \overrightarrow{x} + \overrightarrow{v} \cdot (T_{ref} - t_{up})$. T_{ref} is set to the upper bound of the time range, which is:

$$T_{ref} = \begin{cases} (2i+1)T, & if\ t_{up} \in [2iT, (2i+1)T) \\ (2i+2)T, & if\ t_{up} \in [(2i+1)T, (2i+2)T) \end{cases} \quad (2)$$

The temporal component KEY_{time}, which is used to identify the subtree that the object belongs to, is obtained as follows:

$$KEY_{time} = \begin{cases} 0, & if\ t_{up} \in [2iT, (2i+1)T) \\ 1, & if\ t_{up} \in [(2i+1)T, (2i+2)T) \end{cases}$$

4.1.2 Index in Space

Suppose we have a set of n reference points $\{RP_0, RP_1, \ldots, RP_{n-1}\}$, the data space is then partitioned into n disjoint regions $\{VC_0, VC_1, \ldots, VC_{n-1}\}$ in terms of the distance to the reference points. The partitioning forms a Voronoi Diagram of the n reference points as shown in Figure 5(a). Each reference point RP_i maintains a grid G_i, which centers at RP_i and covers its voronoi cell VC_i.

Given an object $o(\overrightarrow{x}, \overrightarrow{v})$ whose nearest reference point is RP_i. The spatial component KEY_{space} is:

$$KEY_{space}(o) = i \times SPAN_{space} + cid(\overrightarrow{x}', G_i)$$

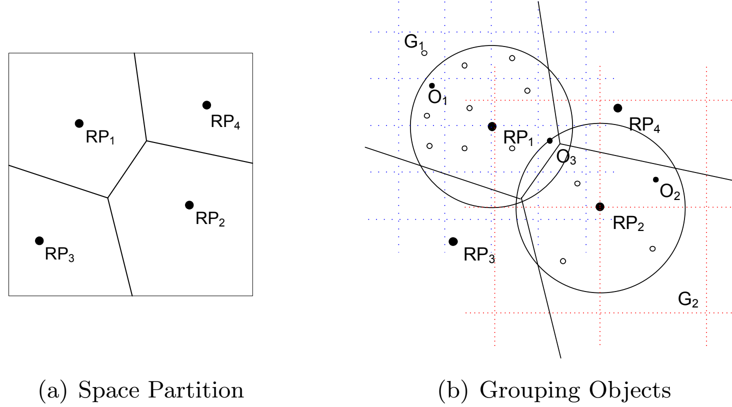
where i is the grid id, i.e., the id of the nearest reference point of o. A portion of $SPAN_{space}$ continuous keys in the B^+-tree are reserved for each grid. $i \times SPAN_{space}$ helps to locate the portion of key values reserved for grid G_i. $cid(\overrightarrow{x}', G_i)$ is the id of the cell in G_i that $\overrightarrow{x}'$ belongs to. The cell id is assigned with the help of a space filling curve. To preserve locality well, the ST^2B-tree utilizes the Hilbert curve as shown in Figure 2. In order to guarantee that the keys of adjacent grids do not intersect with each other, $SPAN_{space}$ must be an upper bound of cid in all grids.

Figure 5(b) illustrates how objects are indexed around two reference points RP_1 and RP_2. o_1, whose nearest reference point is RP_1, is indexed in G_1 and o_2 in G_2 likewise. Another object o_3, although covered by G_2 as well, is indexed in G_1 because o_3 is closer to RP_1 than RP_2 (i.e., in RP_1's voronoi cell VC_1). Although overlap may exist between adjacent grids, the voronoi cells of reference points are disjoint. Therefore, it is clear for an object which grid it belongs to.

In summary, in the ST^2B-tree, object o is indexed with KEY_{ST^2}:

$$KEY_{ST^2} = KEY_{time} \times SPAN_{time} + KEY_{space} \quad (3)$$

where $SPAN_{time}$, similar to $SPAN_{space}$, is the size of the key range reserved for each subtree. $SPAN_{time}$ must be an upper bound of KEY_{space} to avoid



(a) Space Partition (b) Grouping Objects

Figure 5: ST^{2}B-tree: Spatial Key Generation

overlap between keys in two subtrees. KEY_{time} and KEY_{space} are derived as described above.

Figure 4 shows the essence of the ST^{2}B-tree. The current time is in $[2iT, 2(i{+}1)T)$. The two logical subtrees are BT_0 and BT_1. At time $(2i{-}1)T$, the time range of BT_1' has changed to $[(2i{+}1)T, 2(i{+}1)T)$, shown as BT_1 in Figure 4. The next rotation will happen at $2(i+1)T$. BT_0 will be assigned with a new time range, shown as BT_0'. For key allocation, the key space is halved according to the update time. At the second level, each half is further partitioned for the n grids. Finally, at the bottom level, within the key space of each grid, objects are sorted in ascending order of the id of the cells they belong to.

4.2 Basic Query Algorithm

Algorithm 1 depicts the evaluation procedure of a simple range query in the ST^{2}B-tree. Both subtrees are searched. Since all objects in subtree BT_i are indexed with positions at time T_{refi}, the algorithm first enlarges the query region R_q from query time t_q to T_{refi} using the global maximum velocity max_v (line 3) (the same way as in the B^x-tree in Equation 1). Then, the grids of the reference points whose voronoi cell intersects with the enlarged query region needs to be further searched (lines 4-5). The cells that intersect with the enlarged query region are retrieved (line 7) (in ascending order of cell id assigned by using the space filling curve). Finally, an object is added to the result set if its position at time t_q is contained in the query region (lines 8-10). When the query q is a current query, $t_q = t_{now}$ (t_{now} is the current time when the query is issued). If q is a predictive query, $t_q = t_{now} + h$, where h denotes the prediction interval.

In the ST^{2}B-tree, a k Nearest Neighbor (kNN) query is conducted as incremental range queries until exact k nearest neighbors are found. It

Algorithm 1 Range Query

Input: Query region R_q, query time t_q
Output: All objects in R_q at t_q

```
1: result = ∅
2: for each subtree BT_i do
3:    R'_q = EnlargeRegion(R_q, max_v, t_q, T_refi);
4:    for each entry reference point RP_j do
5:       if R'_q intersects with VC_j then
6:          //VC_j stands for the voronoi cell of RP_j
7:          for each cell of G_j that intersects with R'_q do
8:             for each objects o in cell do
9:                p_{t_q} = Position(o, t_q, T_refi);
10:               if p_{t_q} ∈ R_q then
11:                  add o into result;
12: return result;
```

starts with an initial search radius r. If the kNNs are not found in the initial search region, it extend the search radius by *increment*. Both r and *increment* is set to D_k/k as in [8], where

$$D_k = \frac{2}{\pi}[1 - \sqrt{1 - \sqrt{\frac{k}{N}}}] \quad (4)$$

Here, D_k is the estimated distance to k'th nearest neighbor [18] and N is the number of objects in a unit space. We omit the detailed algorithm here and a similar procedure can be found in [8].

4.3 Why is the ST^{2}B-tree Tunable?

We now explain why the ST^{2}B-tree can be easily tuned to adapt to the three kinds of data diversities discussed in Section 3.

4.3.1 Diversity in Space

The ST^{2}B-tree partitions space using n reference points. Each reference point has its own grid and the cell sizes are not necessarily identical for all the grids. In fact, grid granularity can be determined by object density in the voronoi cell of the reference point. As shown in Figure 5(b), objects are relatively dense around RP_1. G_1 therefore is of finer granularity. For RP_2, objects are relatively sparse, so G_2 uses larger cells and partitions space at a coarser level. By using different grids in different areas (for different reference points), the ST^{2}B-tree can discriminate between regions of different densities.

4.3.2 Diversity with Time

At any point of time t, the older subtree (i.e., the one covering the earlier time range) in the ST^{2}B-tree is in a monotonic shrinking phase – only deletions affect that subtree. All insertions are conducted in the other subtree, whose time range covers the current time. At the next transition time iT, the older subtree becomes empty. All objects have been updated to the other subtree or dropped from the system.

If the database receives no update of an object in T time, we assume that the object leaves the system (e.g., parking) and it is deleted from the index. Alternatively, the object is migrated into the other part of the index with its position at new reference time. The new position is estimated using its last updated position and velocity. This is a design decision to be made by the system administrator. For the latter strategy, in order to guarantee the precision of the indexing, the system can pull updates from objects that are not updated in the last T time.

A new time range is assigned to the empty subtree, without interfering with any objects which are currently indexed in the other subtree. The older subtree is refreshed while the original younger subtree becomes the "older" one and enters the shrinking phase. At each transition time, one subtree in ST^{2}B-tree is empty. The ST^{2}B-tree can tune the granularity of grids for the empty subtree, according to the object densities investigated in the previous T time. The grids of the other non-empty subtree are kept unchanged. As a result, the two subtrees in the ST^{2}B-tree have their own set of grids. While searching/updating in a subtree, the corresponding set of grids are used. No collision happens.

4.3.3 Diversity in Space with Time

Since the two subtrees work independently without interference, the granularity of grids can be tuned to capture the change of object density with time. Likewise, the number and positions of reference points can also be tuned to capture the change of object distribution with time. For example, in Figure 3, at t_0 and t_1, the centers of the four circles are used as reference points. Later, at t_2, the centers of the three circles are used as reference points instead. Further, the two subtrees use their own set of reference points and corresponding grids.

In short, both reference points and grid granularity are tunable in the ST^{2}B-tree. The ST^{2}B-tree meets the two requirements we mentioned in Section 3. We discuss the guidelines of tuning these two parameters in the next section.

5 Self-Tuning of the ST^{2}B-tree

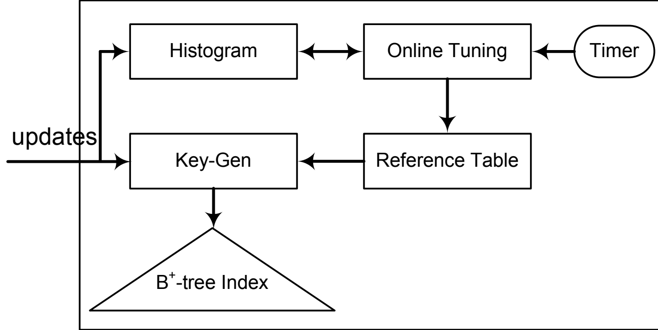


Figure 6: Online Tuning Framework

We now introduce how the self-tuning occurs on the ST^{2}B-tree. Figure 6 shows the self-tuning framework of the ST^{2}B-tree. The tuning framework adds four components on top of the underlying DBMS: the Reference Table for keeping information about the reference points, the Key-Gen for computing the index key, the Histogram for keeping statistics about the objects and the Online Tuning module for executing the tuning process.

The Reference Table maintains information about reference points, including their positions and voronoi cells. It consists of two parts, one for the reference points of the last T time (i.e., the older subtree) and one for the reference points of current time.

On receiving an object update, the Key-Gen module reads information about the reference points of the current time from the reference table and calculates the KEY_{ST^2} according to Equation 3. Then, the update is performed over the B$^+$-tree with KEY_{ST^2} and the new location and velocity of the object. Meanwhile, the statistics about the number of objects in the Histogram is updated accordingly.

At each transition time iT, the Timer triggers the Online Tuning module to start the tuning procedure. Based on the statistics in the Histogram, the Online Tuning module discovers new reference points and determines their grid granularity respectively. At the end of the tuning procedure, the reference table is updated accordingly.

Next, we present how the online tuning module discovers reference points and chooses the reasonable granularity for it.

5.1 Selection of Reference Points

The ST^{2}B-tree can dynamically adjust to different space partitioning. In a moving objects environment, both updates and queries arrive continuously. The tuning procedure must be done online without deferring any other op-

erations. Since the data is highly dynamic, it is difficult to find an optimal partitioning. Even if such an optimal partitioning exists, it is costly to discover it; moreover, its optimality is bound to be short-lived because of the dynamics of the system. Therefore, we aim to rapidly find a moderate set of reference points that roughly but effectively partitions the space based on density differences.

5.1.1 Histogram Maintenance

In the tuning framework, we have a $2d$ histogram, which consists of $n \times n$ square sized cells. The histogram of a cell c_{ij}, where i and j denote the row and column number of the cell, is a tuple $h_{ij} = (\overrightarrow{x_{ij}}, n_{ij})$, where n_{ij} is the estimated number of objects in that cell at $T_{ref} + T$ and $\overrightarrow{x_{ij}}$ is the centroid of objects in the cell.

Let the next transition time be T_{ref}. Then, we find the reference points for $[T_{ref}, T_{ref} + T]$. During that time, all updates will be indexed at a new reference time $T_{ref} + T$ (as explained in Section 4.3). This is why objects are estimated and counted at the time instance of $T_{ref} + T$.

Given an object $o(\overrightarrow{x}, \overrightarrow{v})$ that is updated at t_{up}, the histogram is updated as follows:

1. Estimate o's position at the time instance of $T_{ref} + T$:

$$\overrightarrow{x}'' = \overrightarrow{x} + \overrightarrow{v} \cdot (T_{ref} - t_{up} + T)$$

According to the definition of T_{ref} in Section 4.1.1,

$$\overrightarrow{x}'' = \overrightarrow{x} + \overrightarrow{v} \cdot [(\lfloor t_{up}/T \rfloor + 2)T - t_{up}]$$

2. H_{ij} of the cell that $\overrightarrow{x}''$ belongs to is updated

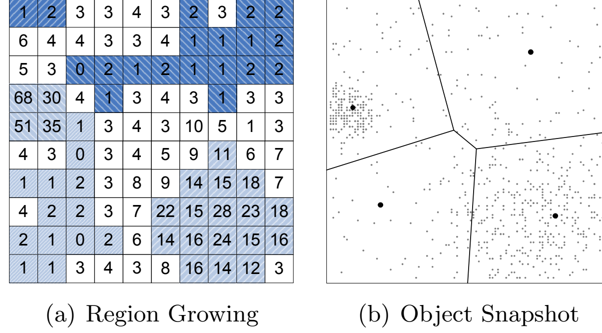
$$\overrightarrow{x_{ij}} = \frac{n_{ij} \cdot \overrightarrow{x_{ij}} + \overrightarrow{o.x}''}{n_{ij} + 1}$$
$$n_{ij} = n_{ij} + 1$$

Thus, $\overrightarrow{x_{ij}}$ is always the centroid of all objects estimated to be in the cell, as

$$\overrightarrow{x_{ij}} = \frac{\sum_{k=1}^{n_{ij}} \overrightarrow{x''_k}}{n_{ij}}$$

5.1.2 Finding Reference Points via Region Growing

The tuning procedure is triggered by the timer at each transition time iT. With the histogram, e.g., Figure 7(a), we identify dense and sparse regions by region growing.



(a) Region Growing (b) Object Snapshot

Figure 7: Finding Reference Points

Region growing is a technique widely used in image segmentation for finding adjacent similar pixels. In image processing, similarity of pixels is defined over color, brightness, etc. For us, each cell in the histogram acts as a pixel. Two cells are said to be similar if they have a similar number of objects. Algorithm 2 shows the procedure of region growing.

Algorithm 2 Region Growing

Output: a set of regions $\{R\}$

1: $RS = \emptyset$;
2: **for** each previous reference points RP_k **do**
3: $\quad c$ = the cell that contains RP_k
4: $\quad$ **if** c is unmarked **then**
5: $\quad\quad$ Add c to a new region R; mark c;
6: $\quad\quad$ Growing(c, R); Add R to RS;
7: **while** there is c that is unmarked **do**
8: $\quad$ Add c to a new region R; mark c;
9: $\quad$ Growing(c, R); Add R to RS
10: **return** RS**;**

Function Growing(c, R)

1: **for** each neighbor cell c' of c **do**
2: $\quad$ **if** c' is unmarked and $|c.n - R.max_n| \leqslant \epsilon$ and $|c.n - R.min_n| \leqslant \epsilon$ **then**
3: $\quad\quad$ Add c' to R; mark c;
4: $\quad\quad$ Growing(c', R);

First, we take the previous reference points as the seeds for growing. Since the distribution and densities of moving objects change gradually, the positions of reference points should move slightly. Starting from cell c, we examine its neighboring cells. If the neighboring cell c' does not belong to any existing region and $|c.n - R.max_n| \leqslant \epsilon$ and $|c.n - R.min_n| \leqslant \epsilon$, c' is added into the region R of c. $R.max_n$ and $R.min_n$ are the maximum and

minimum number of objects n of cells in R, and ϵ is a predefined threshold that defines similarity.

The growing procedure terminates when all the cells belong to some region. If too many regions are returned by the algorithm, the small regions can be dropped as noises and adjacent regions can be merged together based on a relaxed definition of similarity.

The output of the region growing algorithm is a set of regions that have similar densities as shown in Figure 7(a). The centers of the resultant regions are marked as the reference points. More specifically, a resultant region R consists of several adjacent cells. The center of R, i.e., the reference point RP, is calculated as:

$$RP.\overrightarrow{x} = \frac{\sum_{c_{ij} \in R} n_{ij} \cdot \overrightarrow{x_{ij}}}{\sum_{c_{ij} \in R} n_{ij}}$$

The object density for RP is:

$$RP.\rho = \frac{\sum_{c_{ij} \in R} n_{ij}}{|R|}$$

where $|R|$ is the number of cells in R.

As shown in Figure 7(b), the region growing method roughly identifies four reference points, which further partition the space into disjoint voronoi cells. We use the cells in R (with a similar number of objects) for estimating density for RP, ignoring the other cells which are noted as noises.

5.1.3 Alternative Methods

Intuitively, we can also apply density-based clustering methods to partition space. Examples of density based spatial clustering methods include DBSCAN [6] and OPTICS [3]. However, none of these existing methods accommodate the need of online tuning. First, they can only find dense areas; sparse regions may be completely disregarded. Second, density-based clustering methods are time-consuming. DBSCAN takes seconds to cluster a few thousand data points, even in the presence of a spatial index. While the tuning procedure is running, all updates have to be suspended. An update costs a few milliseconds over a B^+-tree on average, which means that thousands of updates may need to be postponed during the tuning procedure. This is not acceptable for online tuning of an index meant to support high update load.

Yet another practical approach to select reference points is to consider the characteristics of real world moving objects, e.g., city traffic. In reality, hotspots remain hotspots, no matter how many objects there are. Prominent landmarks, such as major road junctions and commercial centers, always attract more vehicles than the other places. These hotspots can be used as

reference points most of the time. On the other hand, we can also discover that real traffic often exhibits seasonal patterns, either daily, weekly or monthly. For example, many vehicles move towards the downtown area of a city between 8 to 9am and travel back to the residential suburbs at around 5 to 6pm every weekday. Based on the above observations, reference points can also be computed off-line based on historical data. We can compute and preserve the reference points for each time slice that the data shows similar patterns regularly. An online tuning module can then choose the set of preset reference points of the right slice of time as the tree rolls over with time.

5.2 Selection of Grid Granularity

As discussed in Section 3, data density and grid granularity are important factors that affect the performance of any index based on grid partitioning. Thus, grid granularity is a core parameter to be tuned in our tuning framework. To find the optimal granularity of space partition, we now analyze the effect of different grid granularity on the overall performance of an index.

For ease of analysis, we assume that objects are uniformly distributed in the entire space and the space is partitioned using a single grid. Without ambiguity, the result is directly applicable to each grid in the ST^2B-tree with local uniform assumption around each reference point.

The notations used are listed in Table 1. We start our analysis by giving a definition of the grid order λ.

Symbol	**Description**
N	the number of objects
λ	the resolution of space filling curve
IO_L	the number of leaf node I/O
N_L	the number of leaf nodes
n_o	average number of overflow nodes per leaf node
C_L	capacity of leaf nodes
C_O	capacity of overflow nodes
f	average fan-out of tree nodes
h	the height of the tree
V	the velocity used in query enlargement
L	the side length of square covered by a leaf node
L_q	the side length of a square-sized range query
t_q	the time of the query
T_{ref}	the reference time objects stored in the index
N_q	the number of $1d$ range queries
N_I	the number of internal node accesses

Table 1: Notations for Analyzing Grid Granularity

Definition 1 *The grid order* λ *is defined as the resolution of the space filling curve used for mapping grid cells into 1d values. A grid of order* λ *partitions data space into* $2^\lambda \times 2^\lambda$ *cells.*

Suppose that the entire space is a unit space which is partitioned by a grid of order λ. Then the side length of each cell is $2^{-\lambda}$. The following Lemma 1 estimates the number of leaf I/Os [20].

Lemma 1 *The number of leaf node accesses of a square-sized range query is estimated as:*

$$IO_L = N_L[L + V \cdot (t_q - T_{ref}) + L_q]^2 \tag{5}$$

Proof: N_L is the number of leaf nodes. Let $N_L = 2^{2i}$, where i is an integer no larger than λ. Then, each leaf node covers $2^{2(\lambda-i)}$ cells on average, which forms a square with side length $L = 2^{-i} = (1/N_L)^{1/2}$.

Let $L_Q = V \cdot (t_q - T_{ref}) + L_q$. L_Q is the side length of the enlarged query region. $(L + L_Q)^2$ is the probability that the enlarged query range intersects with the spatial region covered by a leaf node. The number of leaf nodes that to be accessed by a query is $N_L[L + L_Q]^2$.□

However, Equation 5 in Lemma 1 is valid only when there is no overflow pages in the tree, which means that there are very few objects having the same key. Considering the uniform distribution of objects, $\frac{N}{2^{2\lambda}} \leq 1$ ($\lambda \geq \frac{1}{2}\log_2 N$) is a necessary condition of Lemma 1.

Lemma 2 *If* $\lambda \geq \frac{1}{2}\log_2 N$*,* IO_L *does not change when* λ *increases.*

Proof: If $\lambda \geq \frac{1}{2}\log_2 N$, each cell contains 1 object at most and duplicate keys are rare. $N_L = \frac{N}{f}$, where $f = 69\% \cdot C_L$, where 69% is a typical fill factor of the B$^+$-tree.

$$\begin{aligned} IO_L &= N_L(L + L_Q)^2 = N_L\left((1/N_L)^{\frac{1}{2}} + L_Q\right)^2 \\ &= \frac{N}{f}\left((f/N)^{\frac{1}{2}} + L_Q\right)^2 \end{aligned}$$

Thus, IO_L is independent on λ.□

Lemma 3 *If* $\lambda \leq \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$*,* IO_L *increases when* λ *decreases.*

Proof: If $\lambda \leq \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$ ($\frac{N}{2^{2\lambda}} \geq C_L$), the number of objects contained in a cell is larger than the capacity of a leaf node. Each leaf node

has only one key; therefore $N_L = 2^{2i}, i = \lambda$. Suppose each leaf node has n_o overflow nodes, then:

$$IO_L = N_L(1 + n_o)\left((1/N_L)^{\frac{1}{2}} + L_Q\right)^2$$

Let $C_L = C_O$,

$$n_o = \frac{\frac{N}{2^{2\lambda}} - C_L}{C_O} = \frac{N}{2^{2\lambda}C_O} - 1$$

$$IO_L = \left(\frac{N}{C_O}\right)(2^{-\lambda} + L_Q)^2$$

Clearly, IO_L increases as λ decreases. □

Lemma 3 can be explained as follows. All objects contained in the boundary cells, which partially intersect with the query range, need to be checked. As λ decreases, the extent of a grid cell grows exponentially, bringing in more false positives. Access to these false positives incurs additional I/Os.

Corollary 1 *The grid order λ that minimizes IO_L is in range $[\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L, \frac{1}{2}\log_2 N]$.*

Proof: Easily deduced from Lemma 2-3, IO_L increases when λ is either larger than $\frac{1}{2}\log_2 N$ or smaller than $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$.□

According to Corollary 1, in order to minimize the number of leaf node accesses during a query, the space should be use the space filling curve with resolution $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \leq \lambda \leq \frac{1}{2}\log_2 N$.

While Corollary 1 focuses on the I/O overhead of the leaf nodes, we now consider the overhead of internal node accesses. The pitfall of dimensionality reduction is that a multidimensional range query is split into several $1d$ range queries. The number of $1d$ range queries has a significant impact on internal node accesses.

Lemma 4 *The number of 1d range queries N_q and internal node accesses N_I increases with λ.*

Proof: As proven in [12], the number of $1d$ range queries is about half the perimeter of the query range. Therefore, we have $N_q = 2 \cdot L_Q/2^{-\lambda}$. Suppose we do not modify the query algorithm of the B$^+$-tree. Each $1d$ range query starts from the root and searches for the lower boundary of the range. Then, the number of internal node accesses is:

$$N_I = N_q \cdot h = N_q \cdot \log_f N_L = 2^{\lambda+1} \cdot \log_f N_L \cdot L_Q$$

The height of the tree h is relatively stable when N_L varies. N_I is mainly determined by N_q. As λ increases, N_q decreases, and so does N_I. Therefore, a larger value of λ indicates a heavier overhead on internal nodes. □

To evaluate the query performance of a tree-index, the number of leaf I/O N_L is usually the main concern. However, as we shall see in Section 6.2, the number of I/O varies slightly with a wide range of grid order (in between $[\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L, \frac{1}{2}\log_2 N]$). Consequently, the number of accesses to internal nodes dominates query performance in terms of query time. In addition, the effect of internal node accesses becomes even more important in a concurrent environment. When queries and updates arrive simultaneously, each access to the internal node requires locking node and postponing concurrent updates accessing the same node. Therefore, according to Lemma 4 and Corollary 1, $\lceil \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \rceil$ is the best value for λ that minimizes the query costs.

We also need to consider the effect of grid granularity on update cost. An update consists of deleting the old record and inserting the new record. To find the old record, $1+\frac{1}{2}n_o$ nodes are searched on average, apart from the cost for node underflow. The old record is deleted and the node is written back. Despite the sporadic node overflow, inserting the new record incurs $(1+n_o)+1$ leaf and overflow node I/O. The insertion follows the overflow chain to obtain the last overflow nodes into which the new record is to be inserted. Writing back contributes another I/O. The update cost increases with the average number of overflow pages. The cell size increases with smaller λ and so is the number of overflow pages.

In summary, a smaller λ within the range indicated in Corollary 1 leads to better query performance; nevertheless it might incur higher update costs. As we shall see in Section 6.1, the query and the update cost achieves the best tradeoff when $\lambda = \lceil \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \rceil$.

5.3 Reference Time

We note that the reference time of a subtree also affects query performance. A query is enlarged into T_{ref} using maximum velocity. The maximum velocity and reference time both affect the query enlargement. Only the reference time is a variable parameter. Nevertheless, it does not need to be tuned with time. We now discuss how the reference time T_{ref} can be selected to minimize the query enlargement of the two subtrees

Objects update at least once during each time interval T and each subtree covers a time range of T time. Therefore, we investigate the average query enlargement E_{avg} in $[iT, iT+T)$. Supposing that queries arrive evenly in time,

$$E_{avg} = \frac{1}{T}\int_{iT}^{iT+T} E(t)dt, \quad i = 1, 2, ... \tag{6}$$

where $E(t)$ is the enlargement of a query at time t.

We consider the status that the ST^{2}B-tree has both subtrees BT_0 and BT_1. Without loss of generality, we assume that BT_0 is older than BT_1. The

time ranges of BT_0 and BT_1 are $[iT - T, iT)$ and $[iT, iT + T)$ respectively. Then, we have $T_{ref_1} = T_{ref_0} + T$.

The ST^{2}B-tree is designed for current and future queries of moving objects. H is a parameter constraining the maximum time that a query can predict into future. Assuming that the prediction time of a query is evenly distributed in $[0, H]$, the average enlargement of a query at time t is:

$$E(t) = \frac{1}{H} \int_0^H E(t, h) dh \tag{7}$$

where $E(t, h)$ is the enlargement of a query at time t with predict time h. Consider a square-sized range query with side length L enlarged with speed V along each side,

$$E(t, h) = \sum_{i=0}^{1} \left[(L + V|t + h - T_{ref_i}|)^2 - L^2 \right] \tag{8}$$

Combining Equations (6), (7) and (8),

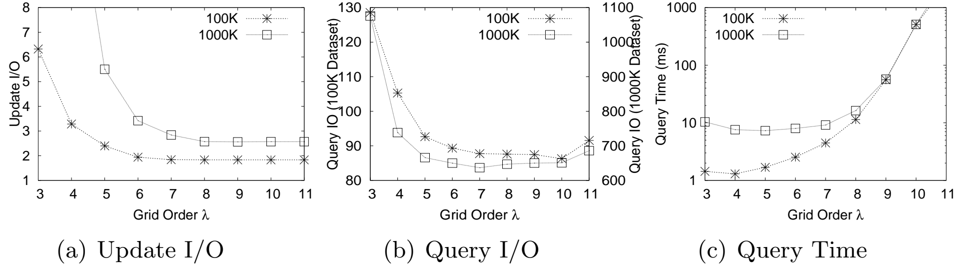
$$\begin{aligned} E_{avg} = \frac{1}{T} \int_{iT}^{iT+T} \frac{1}{H} \int_0^H & \Big[(L + V|t + h - T_{ref_0}|)^2 \\ & + (L + V|t + h - T_{ref_1}|)^2 - 2L^2 \Big] \, dh \, dt \end{aligned} \tag{9}$$

Lemma 5 *The average query enlargement E_{avg} is minimized when the reference time of the subtree with time range $[iT, iT + T)$ is in range $[iT + \sqrt{HT}, iT + \frac{1}{2}(H + T)]$.*

Proof: Let $\Delta t = t - T_{ref_1}$, then $\Delta t + T = t - T_{ref_0}$. Let $tr = T_{ref_1} - iT$, which represents the offset between the reference time to the lower boundary of BT_1's time range.

$$\begin{aligned} E_{avg} &= \frac{1}{T} \int_{-tr}^{T-tr} \frac{1}{H} \int_0^H \ (L + V|\Delta t + T + h|)^2 \\ &\quad + (L + V|\Delta t + h|)^2 - 2L^2 \ \ dh \, d\Delta t \\ &= E_1 + E_2 \\ E_1 &= \frac{1}{TH} \int_{-tr}^{T-tr} \int_0^H \ (2LVT + V^2T^2) + 2V^2|\Delta t + h|^2 \ \ dh \, d\Delta t \\ &= 2V^2 \ \ \frac{LT}{V} + T^2 \ \ + \frac{1}{3} \ \ T^2 + H^2 \ \ + \frac{1}{2}TH - (H + T)tr + tr^2 \\ E_2 &= \frac{1}{TH} \int_{-tr}^{T-tr} \int_0^H (4VL + V^2T)|\Delta t + h| dh \, d\Delta t \\ &= \frac{2V^2}{TH} \ \ \frac{1}{3}tr^3 - THtr + \frac{1}{2}TH^2 + \frac{1}{2}T^2H \end{aligned}$$

E_1 is minimized when $tr = (H + T)/2$ and E_2 is minimized when $tr = \sqrt{HT}$. Both E_1 and E_2 increase when tr is larger than $(H + T)/2$ or smaller than $\sqrt{HT}$. Therefore, E_{avg} is minimized when $tr \in [\sqrt{HT}, (H + T)/2]$.□



(a) Update I/O (b) Query I/O (c) Query Time

Figure 8: Grid Granularity

The exact tr that minimizes E_{avg} depends also on the value of V and L. Fortunately, we can obtain it in the special case that H is equal to T. In fact, it is reasonable to set H to T. An object issues at least one update during time interval T, which means the location and velocity of the object stored will expire after the same time interval. Therefore it makes sense to constrain a query to predict no further than T before the information stored becomes invalid.

Corollary 2 *When H is equal to T, the average query enlargement E_{avg} is minimized when the reference time of the subtree with time range $[iT, iT+T)$ is $iT + \frac{1}{2}(H+T)$, which is $iT + T$.*

Proof: When $H = T$, $\frac{1}{2}(H+T) = \sqrt{HT} = T$. Both E_1 and E_2 in Lemma 5 are minimized when $tr = T$. Therefore, E_{avg} is minimized when reference time $T_{ref} = iT + T$ for the time range $[iT, iT+T)$.□

6 Performance Evaluation

6.1 Experiment Setup

We implemented the ST^{2}B-tree with self-tuning function, and compared it with the B^x-tree [8], the B^{dual}-tree and the TPR*-tree [17], the most representative B$^+$-tree and R-tree based indexes for moving objects. The ST^{2}B-tree, the B^x-tree and the B^{dual}-tree are built on top of the same B$^+$-tree implementation for fairness, where the former two adopt the optimal query enlargement algorithm [9], which has been designed for the B^x-tree and is also applicable to the ST^{2}B-tree. We use the TPR*-tree implementation provided in [2]. The original version stores moving objects as rectangles, which reduces the capacity of the leaf nodes. We modified the code to store objects as points to enhance the performance.

The data space is $100 \times 100\text{km}^2$. We have five kinds of objects, with maximum speed of 30, 60, 90, 150 and 300km/h, simulating different real vehicles in different traffic conditions. Since we intend to investigate the

imbalance and changes in the workloads, we use nonuniform workloads in most experiments unless explicitly stated. We have randomly selected some points in the space as hotspots. Each hotspot uses a Gaussian distribution to generate objects around it. The queries are either uniform or non-uniform. The non-uniform queries follow the same distribution as the objects. The settings of workload used in the experiments are summarized in Table 2, where default values of variable parameters are shown in bold.

Parameter	**Setting**
Max Update Time (T)	**120** timestamps
Number of Hotspots	1, 2, ..., **10**
Number of objects(K)	100, 200, 300, ..., **1000**
Max speed of objects	30, 60, 90, 150 and 300km/h
Query Distribution	**uniform**, non-uniform
Number of queries	**200**
Range Query Size	100×100m^2, 200×200m^2, ..., **1×1km**^{2}
kNN k	10, 20,..., **100**
Query Predict Time (h)	**120** timestamps
Query-Update ratio	10:1, 1:1, 1:10, **1:100**, 1:1000

Table 2: Workload Settings

All the indexes are implemented in C++ and all of them use the same disk manager included in the TPR*-tree implementation [2]. The disk page size is 4KB. A LRU buffer of 50 pages is used. All experiments were conducted on a PC with Pentium IV 3.0GHz processor, 1.0GB memory and 80G SATA Disk, running Window XP.

6.2 Effect of Grid Granularity

We first study the effect of grid granularity empirically to verify the analysis in Section 5.2 and to determine the optimal grid order. We test on two workloads, including 100K and 1M moving objects. The positions of objects are randomly selected. Since the objects are uniformly distributed, the object density in the whole space is the same. The ST^{2}B-Tree will have only one reference point at the center of the space. The query workload consists of 200 1000×1000m^2 uniform range queries.

Figure 8 illustrates the overall performance with grid order λ varying from 3 to 11. We make the following observations:

1. The update I/O increases significantly when $\lambda \leq \frac{1}{2}(log_2 N - log_2 L)$ (about 7 for 1M objects) and hardly changes with finer partitioning.
2. When $\lambda \leq \frac{1}{2}(log_2 N - log_2 L)$, the query I/O increases with smaller value of λ. However, with larger λ, the number of average query I/O

hardly changes. Note that in Figure 8(b), the query I/O of the 1M dataset is labeled with the right y-axis with different values, because we do not intend to compare the I/O of 100K and 1M datasets but to show the trend of I/O changes with grid order λ.

3. The query processing time increases dramatically with a larger λ due to increasing number of key retrievals. Notice that the query processing time increases when λ becomes smaller. This can be explained by the fact that with excessively large grid cells, very few number of $1d$ searches are required. However, the I/O cost increases significantly, which contributes more to the processing time. In addition, it incurs more time to prune away a large number of false positives when the grid cells are too large.

Based on the observation in Figure 8 and the analysis in Section 5.2, we set the grid granularity λ to $\lceil \frac{1}{2}(log_2 N - log_2 L) \rceil$, which results in the best tradeoff between update and query performance. In the following experiments, this rule is applied to the selection of global space partitioning for the B^x-tree, while for the ST^2B-tree, it guides the selection of grid granularity of each reference point.

6.3 Spatial Diversity

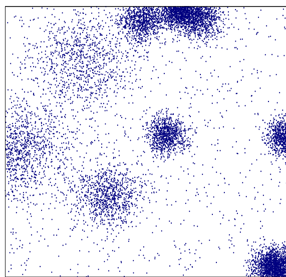
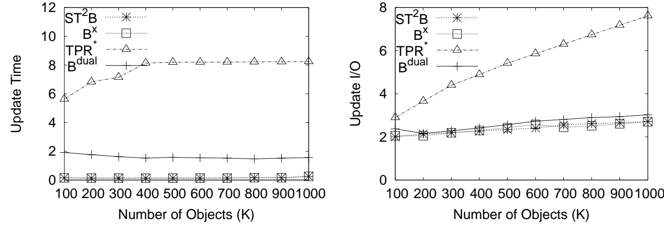
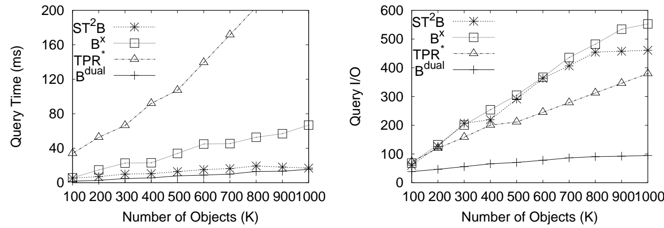


Figure 9: Hotspots in Spatial and Temporal Test

We now investigate the effectiveness of the ST^2B-tree with regard to the spatial diversity of moving objects. We use the default workloads that have been generated with 10 randomly selected hotspots as shown in Figure 9 (some hotspots are so close to others that they cannot be clearly seen). In this set of experiments, we keep the distribution and cardinality of workload unchanged with time and examine the effect of data skew only. The indexes run up to $1.5T$ (180s) and are evaluated on the average query performance for 200 queries.



(a) Update



(b) Range Query (Uniform)

Figure 10: Object Cardinality

Dataset Size	100K	200K	300K	400K	500K	1M
Time (ms)	953	6125	27282	32641	93188	306048

Table 3: Time for MOR Calculation

6.3.1 Scalability Test

First, Figure 10 shows the effect of the database size on query performance. The number of objects varies from 100K to 1M, with an increment of 100K. Because of the overlap between MBRs, the TPR*-tree has to search multiple paths in an update, resulting in a high update cost in both I/O and time Figure 10(a). The MBR adjustment during an update operation further degrades the update time. The B^x-tree and the ST^2B-tree both have fairly constant update time, both of which are less than 0.2ms. The update time of the TPR*-tree is about 40 times higher. The TPR*-tree update I/O grows with the number of objects. Because the TPR*-tree has a smaller internal node capacity, the height of the tree grows faster than the others and still due to the overlap between MBRs, more nodes have to be searched during an update.

The update time of the B^{dual}-tree is about 10 times higher than that of the ST^2-tree and the B^x-tree. In this implementation, the MORs of an internal entry in the B^{dual}-tree are updated online, along with the change (update) of the key range of the subtree of the entry. The MORs computation is time-consuming and deferrs the fast update of the B^+-tree. The MORs can also be computed at the query time so that the update time is as low as the other B^+-tree based indexes. Table 3 shows the time for com-

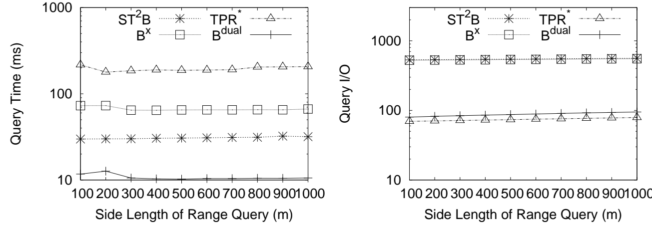


Figure 11: Range Query

puting the MORs for the whole tree, varying the number of objects. For the default 1M dataset, it needs more than 5 minutes to compute the MORs, which is unacceptable in an MOD.

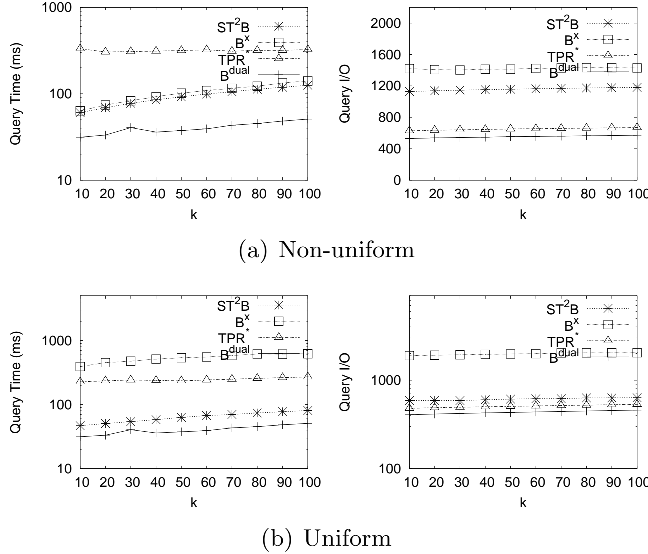
The query cost of all indexes increase linearly with the number of objects in Figure 10. For range queries (Figure 10(b)), the ST^{2}B-tree and the B^x-tree have similar in I/O overhead, which is mainly determined by the number of objects contained in the enlarged query region. The ST^{2}B-tree partitions low-density regions with coarser grid and thus reduces the processing time. The TPR*-tree has smaller I/Os because it does not need to extend the query range. However, the processing time is much higher because of the inefficient search operations of the R-tree (in comparing with the B$^+$-tree). As excepted, the B^{dual}-tree outperforms the others in both query time and I/O due to the partitioning in velocity dimensions.

6.3.2 Range Query

Figure 11 further investigates the performance of range queries. The size of the range queries varies from $100 \times 100m^2$ to $1 \times 1km^2$. The I/O costs of the B^x-tree and the ST^{2}B-tree are dominated by the query enlargement, yet less sensitive to the size of the query region. The TPR*-tree has lower I/O cost and much higher query processing time. As shown, the ST^{2}B-tree is about 2 times faster than the B^x-tree and 5 times than the TPR*-tree (The query is shown in log-scale in Figure 11).

6.4 kNN Query

Figure 12 examines the performance of kNN queries further, with k varying from 10 to 100. The I/O cost of the kNN queries, which depends on the number of objects in the expanded query region, is less sensitive to k for the B^x-tree and the ST^{2}B-tree. The ST^{2}B-tree incurs fewer I/Os than the B^x-tree. The ST^{2}B-tree surpasses the B^x-tree by a great margin in kNN queries. Both the time and I/O cost of the B^x-tree are quite high. The kNN queries in both B^x-tree and the ST^{2}B-tree are conducted as incremental range queries with the initial search region estimated from the objects density. The B^x-tree makes such estimation using global object density. As a result, in dense



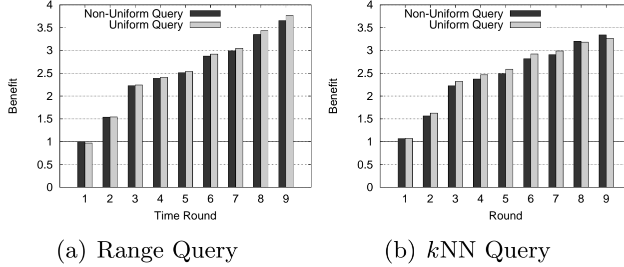
(a) Non-uniform

(b) Uniform

Figure 12: kNN Query

regions, the B^x-tree starts the kNN search with an oversized search region; in sparse regions, the B^x-tree starts with a small search region, but has to expand the region for many times to find the kNNs. Both affect the query processing time and I/Os. The ST^{2}B-tree, on the other hand, starts the search with a more accurate radius according to the object density around the reference points. Owing to more accurate search region, the performance of the ST^{2}B-tree on kNN queries is less affected by the data skew. The TPR*-tree incurs fewer I/Os and higher query processing time for the same reason as stated in the scalability test.

For the TPR*-tree and the B^{dual}-tree, the update time is high although they incurs fewer I/Os in comparison to the B^x-tree and the ST^2-tree. Since we are investigating an efficient index for moving objects, the updates arrives frequently. For the default 1M dataset, given the maximum update time $T = 120s$, there are at least $\frac{10}{12}K$ updates in one time unit. The B^x-tree and the ST^2-tree can complete this amount of updates in less than 2 seconds. However, as for the B^{dual}-tree, it spends about 20 seconds and as for the TPR*-tree, it needs minutes of time. Although the B^{dual}-tree and the TPR*-tree outperforms the others on query time, the udpate time of both are unacceptable in moving objects applications, where the number (and frequency) of location updates are much higher than that of queries. For clearer demonstration on the effect of self-tuning, we hence omit the results of the TPR*-tree and the B^{dual}-tree.



(a) Range Query (b) kNN Query

Figure 13: Increasing Data Cardinality

6.5 Temporal Diversity

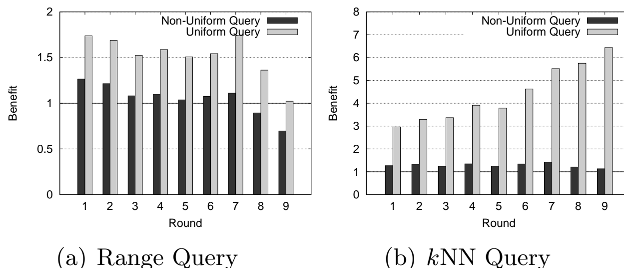
Next, we examine the effectiveness of the ST^{2}B-tree's self-tuning to adapt to the time-dependent changes in data cardinality. All the objects follow the same distribution with previous experiments (Figure 9). We build the indexes in the first round and run them for another 9 rounds of time. Each round is 120s. In each round, each object updates once and the whole index will be refreshed after the round. The number of queries is 1% of the number of updates each round. This is to simulate the real applications where many moving objects will keep on updating their positions, and the number of positional updates significantly outnumbers the number of queries. We study the total time of processing the updates and queries in each round as a measure of overall performance. Then we compute the benefit introduced by the self-tuning feature of the ST^{2}B-tree, which is defined as follows:

$$\text{Benefit} = \frac{\text{total processing time of the B}^x\text{-tree}}{\text{total processing time of the ST}^2\text{B-tree with self-tuning}}$$

The granularity of the B^x-tree is selected using the initial number of objects. When the data is uniformly distributed, the performance of the ST^{2}B-tree degrades to that of the B^x-tree with only one reference point at the center of the space. In other words, the B^x-tree is a static version of the ST^{2}B-tree which completely ignores the distribution and changes of objects. Hence we compare the ST^{2}B-tree with the static B^x-tree to show the effectiveness of the self-tuning features. The running time of the self-tuning process is included in the total processing time of the ST^{2}B-tree.

Firstly, we start with 100K objects and add another 100K objects in each round. Figure 13 shows the benefit of self-tuning in each round of time. The benefit of self-tuning grows with time, when the hotspots and data distribution change with time.

Initially, the B^x-tree selects the granularity of space partitioning with 100K objects. It then uses a grid with large cells (about $3000 \times 3000\text{m}^2$). With the increasing number of objects in the following rounds, the update performance degrades, because the increase in the number of overflow pages affects the balance of the underlying B$^+$-tree. On the other hand, since the



(a) Range Query (b) kNN Query

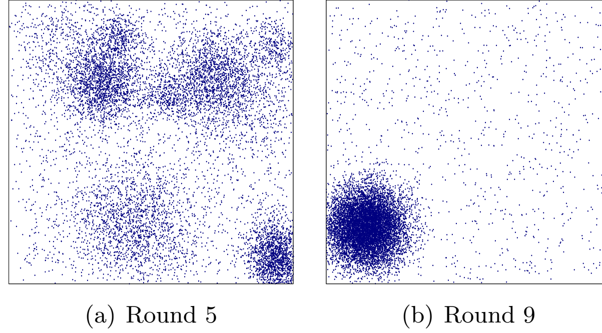
Figure 14: Decreasing Data Cardinality

ST^{2}B-tree partitions and indexes objects according to the distribution and density, the update cost remains at about 0.2ms all the time. Since the B^x-tree uses a large cell, it saves on query processing time according to our findings in Section 6.2. However, with carefully chosen granularity of space partition, the query processing time of the ST^{2}B-tree is higher than the B^x-tree only in the dense regions. In those sparse regions, the ST^{2}B-tree might use even larger grid cells, which would reduce the query processing time. Therefore, combining all these facts, the overall benefits of the self-tuning of the ST^{2}B-tree over static B^x-tree, especially when there are more updates than the queries, are obvious and significant.

Figure 14 shows the results of a reverse process. Starting with 1M objects, the number of objects being indexed decreases by 100K per round. The B^x-tree now uses a fine grid with smaller cells (about 200×200m^2) to partition the entire space. As we can see, the benefit of self-tuning is just a little higher with non-uniform queries. That is because the cost of the B^x-tree is also near optimal with such a fine grid. Non-uniform queries follow the same distribution as objects, and therefore the queries are concentrated at those dense regions. Now, in those dense regions, the ST^{2}B-tree also employs fine grid. Therefore, the benefit of tuning is less significant. However, for the uniform queries, the ST^{2}B-tree gains more by tuning with the data workload. The ST^{2}B-tree reduces the processing time of queries in the sparse regions by using larger grid cells. The overall performance gain is much more significant than for non-uniform queries.

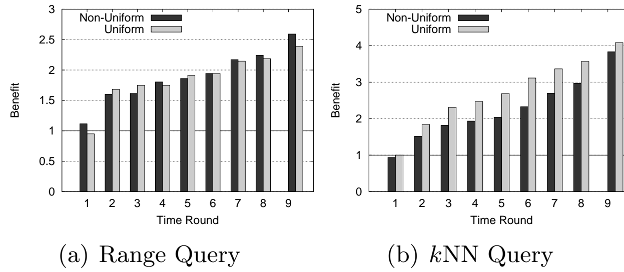
6.6 Spatio-Temporal Diversity

Now we further investigate the performance of the self-tuning phase of the ST^{2}B-tree with regard to the changes of objects distribution with time. We generate a set of workloads in which the skewness of objects increase with time. In round 0, we build the indexes with 1M uniformly distributed objects. Next, in round 1, the objects are generated with 10 hotspots. Subsequently, the number of hotspots is reduced by 1 each round. Finally, in round 9, there is only one hotspots. Figure 15 shows the snapshots of



(a) Round 5 (b) Round 9

Figure 15: Distribution of Spatio-Temporal Test



(a) Range Query (b) kNN Query

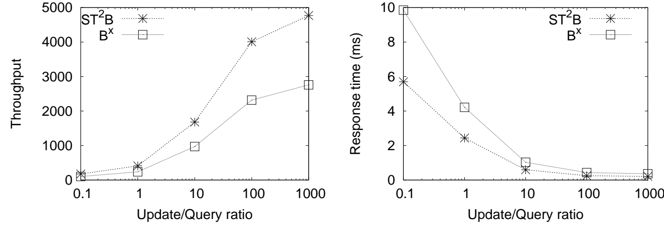
Figure 16: Change of Data Distribution with Time

objects at round 5 (moderately skewed) and round 9 (highly skewed). The query-update ratio is still 1:100.

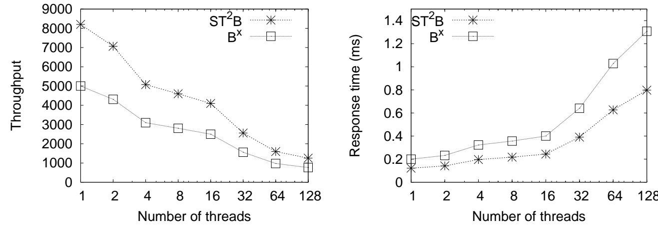
Figure 16 shows the changes of performance benefits with time. As expected, the gain of the self-tuning ST^{2}B-tree over the static B^x-tree increases when the data become even more skewed with time. During round 1, the ST^{2}B-tree is comparable to the B^x-tree. Since the objects are uniformly distributed, the region-growing algorithm will result in only one reference point and hence the ST^{2}B-tree degenerates to a B^x-tree with only one reference point. However, with skewed objects joining in the subsequent rounds, the ST^{2}B-tree gradually outperforms the B^x-tree owing to the self-tuning phases, which are equipped with adaptive space partitioning and granularity of indexing. For range queries (Figure 16(a)), the ST^{2}B-tree outperforms the B^x-tree by about 2 times in round 9 for both uniform and non-uniform query workloads. For kNN queries (Figure 16(b)), the performance gain is much higher, which is about 4 times.

6.7 Throughput Test

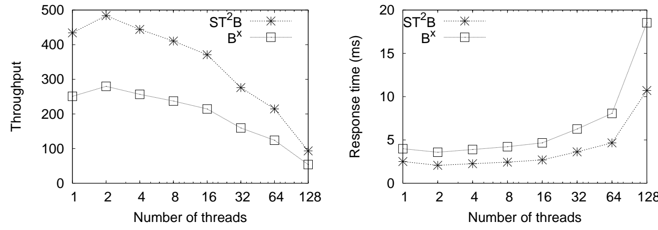
Finally, we evaluate the performance of the B^x-tree and the ST^{2}B-tree on concurrent operations. To highlight the difference between the two B$^+$-tree based indexes, we do not show the result for the TPR*-tree since it has



(a) Query-Update Ratio



(b) Number of Threads: Query-Update Ratio=1:100



(c) Number of Threads: Query-Update Ratio=1:1

Figure 17: Concurrent Operations

been shown to be inefficient in a concurrent environment in [16] and [8]. We implemented the B-link concurrency control mechanism [10]. A multi-thread program ran in a single PC to simulate a real multi-user environment. The default 1M dataset as shown in Figure 9 is used. The query workload of each thread is 200 non-uniform range queries with side length of 1000m.The results are the average of 10 runs of simulation.

Figure 17(a) shows the throughput and response time with the query-update ratio varying from 10:1 to 1:1000 using 8 working threads. In real moving object applications, the update load caused by the changes in object locations and moving speed is much higher than the query load, and the query-update ratio is to simulate such scenario. As expected, the throughput of the indexes increase significantly with more updates and the response time decreases. The queries, which hold shared lock on the node being accessed, do not prevent the other queries. However, although queries allow other read operations, they block the update operations, and by design of the experiment, the updates contribute more to the throughput. The updates access only a few nodes in the index and can finish very quickly. The (range)

queries, on the other hand, have to traverse multiple paths and read many leaf nodes (data nodes); hence they take longer than the updates. Since the throughput is defined as the number of operations completed by the indexes every second, the updates contribute more to it. Therefore, when the percentage of updates in the workload increases, the throughput increases and the response time decreases accordingly.

Figure 17(b) shows the effect of the number of threads under workload whose query-update ratio is 1:100. The number of threads varies from 1 to 128. In general, the throughput reduces with the number of threads for all indexes. Likewise, the response time increases with the number of threads being used. An update locks exclusively the node being accessed and all the concurrent requests for reading/writing the node are suspended. Considering the workload includes more updates than queries, the indexes are frequently being w-locked. As a result, the throughput decreases with more threads and each thread waits for a longer time for its turn to access the tree.

However, with more queries, the throughput first increases and then reduces with increasing number of threads. As shown in Figure 17(c), when the workload consists of 50% queries and 50% updates, the throughput reaches the peak with about 2 threads for both indexes. As more threads are introduced, they start to compete for resources and the throughput reduces as a result. Because the queries hold a shared lock on the node being accessed, it will not suspend the other query operations. Therefore, the degree of concurrency becomes higher with more queries. With more queries, the throughput reaches the peak with more threads. For example, when query-update ratio is 10:1, the peak of the throughput is 4 threads or so. However, in the MODs, there are typically more short updates than queries. Due to space constraint, we omit the results for such workload composition.

As can be observed from Figure 17(b) and 17(c), the indexes hit thrashing point after the number of threads increase to certain point and this is when the throughput starts to decrease after hitting the peak. We note that the throughput and the response time can be improved by implementing some admission controls to throttle the amount of work being performed concurrently. However, the admission control introduces another dimension of effect to the performance, we therefore ran our throughput tests without any admission control.

For different workload and number of threads, the ST^{2}B-tree obtains higher throughput and faster response time than the B^x-tree. The ST^{2}B-tree partitions sparse regions with larger cell, incurring much fewer key retrieves than the the B^x-tree. The internal nodes are locked less frequently than the B^x-tree. On the other hand, in dense regions, the ST^{2}B-tree uses finer grid than the B^x-tree. The B^x-tree has higher update I/O. This means that an update operation exclusively locks more nodes, preventing the other

updates or queries from accessing these nodes.

7 Conclusion

In this paper, we proposed the ST^{2}B-tree, a self-tunable B$^+$-tree index for moving objects databases. The ST^{2}B-tree partitions the data space using a set of reference points. Each reference point uses its own individual grid to partition its voronoi cell. The grid granularity is determined by the object density around a reference point. By monitoring the distribution and density of objects, the ST^{2}B-tree dynamically determines a different set of reference points, and adaptively adjusts the granularity of the space partitioning, The self-tuning procedure runs online, incurring almost no overhead. The experimental results confirm that the ST^{2}B-tree is efficient, robust and scalable with respect to data distribution, volume and concurrent operations. More importantly, equipped with the self-tuning capability, the ST^{2}B-tree is also adaptive to changes in workload with time.

8 Acknowledgments

This work was in part funded by ASTAR SERC, within the SpADE project under Grant 032-101-0026. M.A. Nascimento has been partially supported by NSERC Canada. Thanks to the anonymous referees for their insights and suggestions on improving the clarity of the paper.

9 Repeatability assessment result

Figure 8, Figures 10-13 and Figure 16 have been verified by the SIGMOD repeatability committee. Code and data used in the paper are available at http://www.sigmod.org/codearchive/sigmod2008/.

References

[1] SpADE: A SPatio-temporal Autonomic Database Engine for location-aware services. http://www.comp.nus.edu.sg/ spade/.

[2] TPR*-tree. http://www.rtreeportal.org/code.html.

[3] M. Ankerst, M. M. Breunig, H.-P. Kriegel, and J. Sander. OPTICS: Ordering Points To Identify the Clustering Structure. In *SIGMOD Conference*, pages 49–60, 1999.

[4] S. Chaudhuri and V. R. Narasayya. An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In *VLDB*, pages 146–155, 1997.

[5] S. Chen, B. C. Ooi, K. L. Tan, and M. A. Nascimento. Self-Tunable Spatio-Temporal B^+-tree Index for Moving Objects. *Technical Report*, 2007.

[6] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu. A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases with Noise. In *KDD*, pages 226–231, 1996.

[7] S. Guo, Z. Huang, H. V. Jagadish, B. C. Ooi, and Z. Zhang. Relaxed Space Bounding for Moving Objects: A Case for the Buddy Tree. *SIGMOD Record*, 35(4):24–29, 2006.

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

[9] C. S. Jensen, D. Tiesyte, and N. Tradisauskas. Robust B^+-Tree-Based Indexing of Moving Objects. In *MDM*, page 12, 2006.

[10] P. L. Lehman and S. B. Yao. Efficient Locking for Concurrent Operations on B-trees. *ACM Trans. Database Syst.*, 6(4):650–670, 1981.

[11] M. F. Mokbel, X. Xiong, and W. G. Aref. SINA: Scalable Incremental Processing of Continuous Queries in Spatio-temporal Databases. In *SIGMOD Conference*, pages 623–634, 2004.

[12] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *TKDE*, 13(1):124–141, 2001.

[13] K. Mouratidis, D. Papadias, and M. Hadjieleftheriou. Conceptual Partitioning: An Efficient Method for Continuous Nearest Neighbor Monitoring. In *SIGMOD Conference*, pages 634–645, 2005.

[14] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *SIGMOD Conference*, pages 637–646, 2004.

[15] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *SIGMOD Conference*, pages 331–342, 2000.

[16] B. Seeger and H.-P. Kriegel. The Buddy Tree: An Efficient and Robust Access Method for Spatial Database. In *VLDB*, pages 590–601, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

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

[18] Y. Tao, J. Zhang, D. Papadias, and N. Mamoulis. An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces. *TKDE*, 16(10):1169–1184, 2004.

[19] X. Xiong, M. F. Mokbel, and W. G. Aref. SEA-CNN: Scalable Processing of Continuous K-Nearest Neighbor Queries in Spatio-temporal Databases. In *ICDE*, pages 643–654, 2005.

[20] M. L. Yiu, Y. Tao, and N. Mamoulis. The B^{dual}-Tree: Indexing Moving Objects by Space Filling Curves in the Dual Space. *VLDB J.*, accepted for publication, 2008.