Continuous Online Index Tuning in Moving Object Databases

SU CHEN
National University of Singapore
MARIO A. NASCIMENTO
University of Alberta
and
BENG CHIN OOI and KIAN-LEE TAN
National University of Singapore

In a *moving object database* (MOD), the dataset, e.g., the location of objects and their distribution, and the workload change frequently. Traditional static indexes are not able to cope well with such changes, i.e., their effectiveness and efficiency are seriously affected. This calls for the development of novel indexes that can be reconfigured automatically based on the state of the system. In this paper, we design and present the ST^{2}B-tree, a *S*elf-*T*unable *S*patio-*T*emporal B^+-Tree index for MODs. In ST^{2}B-tree, the data space is partitioned into regions of different density with respect to a set of reference points. Based on the density, objects in a region are managed using a grid of appropriate granularity - intuitively, a dense region employs a grid with fine granularity, while a sparse region uses a grid with coarse granularity. In this way, the ST^{2}B-tree adapts itself to workload diversity in space. To enable online tuning, the ST^{2}B-tree employs a "multi-tree" indexing technique. The underlying B$^+$-tree is logically divided into two subtrees. Objects are dispatched to either subtree depending on their last update time. The two subtrees are rebuilt periodically and alternately. Whenever a subtree is rebuilt, it is tuned to optimize performance by picking an appropriate setting (e.g., the set of reference points and grid granularity) based on the most recent data and workload. To cut down the overhead of rebuilding, we propose an eager update technique to construct the subtree. Finally, we present a tuning framework for the ST^{2}B-tree, where the tuning is conducted online and automatically without human intervention, and without interfering with the regular functions of the MOD. We have implemented the tuning framework and the ST^{2}B-tree, and conducted extensive performance evaluations. The results show that the self-tuning mechanism minimizes the degradation of performance caused by workload changes without any noticeable overhead.

Categories and Subject Descriptors: H.2 [**Database Management**]: ; H.2.8 [**Database Applications**]: Spatial databases and GIS; H.3.1 [**Content Analysis and Indexing**]: Indexing methods

General Terms: Algorithms, Design, Experimentation, Performance

Additional Key Words and Phrases: Data distribution, Index tuning, Location-based services, Moving object indexing, Self-tuning

1. INTRODUCTION

Database tuning is crucial to the efficient operation of a database management system (DBMS). In fact, most commercial DBMS provides some tuning tools. The goal of tuning is to ensure the database system always operates 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 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 track of the system's performance all the time. The only practical solution is to make a database system self-tunable so that tuning proceeds automatically with minimal human intervention.

While some works have been done to develop self-tuning technologies in database systems, these are largely restricted to traditional static databases. A representative example of this line of work can be found in [Chaudhuri and Narasayya 1997]. However, there are a number of emerging applications (e.g., Geographical Information Systems and location aware applications such as traffic monitoring) that manage highly dynamic data. In particular, in a *M*oving *O*bjects *D*atabase (MOD), a large number of objects are moving continuously, and their locations have to be frequently updated. More importantly, the distribution of moving objects varies over time and space. For example, in a traffic management system, some places are likely to be more crowded and populated than others. The number of vehicles at certain locations may be larger during the day and relatively smaller at night. This means that the workload for the same query over the same region may be quite different at different times. For these dynamic databases, the workload changes much faster than in traditional databases, and hence they need to be tuned more frequently. Thus, there is a need to re-examine self-tuning methods for managing dynamic databases.

In this paper, we focus on designing self-tuning indexes for MODs. While some works, such as [Kalashnikov et al. 2004; Mokbel et al. 2004; Mouratidis et al. 2005; Xiong et al. 2005], studied in-memory indexing and query processing for moving objects, many others [Saltenis et al. 2000; Saltenis and Jensen 2002; Tao et al. 2003; Jensen et al. 2004; Patel et al. 2004; Yiu et al. 2008; Chen et al. 2008a] focused on disk-based indexes and query evaluation, considering the tremendous amount of ubiquitous moving objects. Our work belongs to the second category. However, existing works on disk-based moving object indexes mostly focused 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 object indexes.

In addition, for a tuning mechanism to be useful in a MOD, it must be automated, lightweight, and fully online. A MOD is operational at all times (24 by 7), making it impractical and uneconomical to require DBAs to tune the system manually. Since updates and queries arrive at the system continuously, we cannot afford to hold the system and postpone all regular operations (i.e., updates and queries) until the tuning procedure completes. The tuning should be performed online and completely in real time.

A preliminary report of this work appeared in [Chen et al. 2008], where we introduced the ST^{2}B-tree and an online tuning framework for it. The primary focus is on tuning the grid granularity and on determining an optimal space partitioning. In this extended paper, we addressed a key limitation of the “multi-tree” method - that of performance degradation caused by migrating objects from one subtree to another during rollover. A tunable eager update strategy is introduced to handle object migrations efficiently. In addition, we have a more rigorous analysis and comparative experimental study that examines the effects of various tuning knobs thoroughly. Specifically, the contributions of this paper are the following:

—We identify and examine three kinds of data diversities in MODs and specify their impact on a moving object index based on space partitioning.

—We present a *S*elf-*T*unable *S*patio-*T*emporal B^+-tree index (ST^{2}B-tree) for moving objects. The ST^{2}B-tree employs the "multi-tree" technique, which facilitates online tuning by maintaining two subtrees. The index dynamically adapts the granularity of space partitioning based on the object density within regions around a set of reference points.

—We discuss the potential performance problem of the "multi-tree" technique and propose an eager update approach to cut down the overhead of migrating objects in the ST^{2}B-tree between two subtrees before rebuilding the old subtree.

—We introduce an improved 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 propose methods to select the reference points and provide guidelines for determining the grid granularity and other tunable parameters.

—An extensive experimental study is conducted 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 with virtually no overhead.

The remainder of the article is organized as follows. In the next section, we review some related works. In Section 3, we first briefly introduce the B^x-tree, which is the base of the ST^{2}B-tree. Section 4 presents the ST^{2}B-tree, including the structure and basic query algorithms; we also provide insights on why the ST^{2}B-tree is amenable to tuning. In Section 5, we propose the eager update technique to manage object migrations during rollover from one subtree to another. Sections 6 and 7 analyze the effect of grid granularity and some temporal parameters. In Section 8, we present the online tuning framework and describe how it works. Section 9 reports the results of an extensive performance study. Finally, we conclude this paper in Section 10.

2. RELATED WORK

Moving object indexing is a well studied topic in database community. Specifically, a moving object is a multi-dimensional point in natural, whose coordinates keep changing with time. In most existing moving object indexes, objects are indexed as stationary points in the spatial space. Depending on the space where objects are indexed, Tao and Xiao [2008] classify moving object indexes into two categories, i.e., primal and dual indexes. Primal indexes [Saltenis et al. 2000; Tao et al. 2003; Saltenis and Jensen 2002; Procopiuc et al. 2002], index objects in their original spatial space. Dual indexes [Kollios et al. 1999; Jensen et al. 2004; Kollios et al. 2005; Yiu et al. 2008; Chen et al. 2008a], on the other hand, transform objects to some "dual" space first. Indexing and querying are performed in the "dual" space.

No matter in which category, an index aims to preserve moving objects' location proximity in the original space as much as possible. On the other hand, since objects are stored as stationary points in both categories, an index needs to adopt some policy to preserve the temporal information of each object, e.g., the updating time. Hence, in the remaining part of this section, we review existing moving object indexes from these two perspectives.

2.1 Space Partitioning vs. Data Partitioning Indexes

While objects are represented as stationary (multi-dimensional) points, one essential issue is how they are organized in the index. Considering this, existing moving object indexes can be classified into two major categories: data partitioning and space partitioning indexes.

In data partitioning indexes, objects are organized into dynamic partitions. Representative schemes in this category include the TPR-tree [Saltenis et al. 2000], the TPR*-tree [Tao et al. 2003], the R^{EXP}-tree [Saltenis and Jensen 2002] and the STAR-tree [Procopiuc et al. 2002]. All of these methods are based on traditional spatial indexes such as the R-tree [Guttman 1984] and R*-tree[Beckmann et al. 1990]. Specifically, at the bottom level, a leaf node accommodates up to a given number of objects that are close to one another. At higher levels of the hierarchical structure, each intermediate node contains up to a given number of entries, each of which contains a pointer to one of its children and the MBR (*M*inimum *B*ounding *R*ectangle) of the corresponding child. The MBR of a leaf is the smallest rectangle covering all the objects it contains. Similarly, the MBR of an intermediate node is the region which just bounds the MBRs of all its children. A node split occurs when the number of objects/entries to be stored in a node exceeds its capacity. On the other hand, two neighboring nodes are merged into one if objects/entries of both nodes can be accommodated in only one node. As a result of splitting and merging, objects/entires are clustered into groups based on their proximity. Therefore, the regions in which objects are crowded always consist of many small (leaf) MBRs. Sparse regions, on the contrary, are typically covered by a few large MBRs.

In the second category, space partitioning indexes adopt fixed space partitioning. These methods typically partition the data space in advance using a grid. An object is indexed by the cell it belongs to. Indexes in [Mokbel et al. 2004; Mouratidis et al. 2005; Xiong et al. 2005], utilize the grid index directly, while some other indexes such as those developed in [Jensen et al. 2004; Yiu et al. 2008] use a B$^+$-tree on top of the grid. Each grid cell is assigned a unique id, and objects are indexed by the B$^+$-tree with the id of the cell they belong to. As space partitioning indexes split the 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 the indexes, especially in the presence of skewed data. By comparison, data partitioning indexes are typically less susceptible to data diversities and changes as a result of MBR merging and splitting.

In [Chen et al. 2008b], the authors compare several state-of-art indexes experimentally and the result shows that space partitioning indexes outperform data partitioning indexes in most cases, especially on the update performance. In general, space partitioning based indexes surpass their counterparts that are based on data partitioning in two ways. First, both the grid index and the B$^+$-tree are well established indexing structures present in virtually every commercial DBMS. The index can be integrated into an existing DBMS easily. No fundamental (lower level) changes are required to the underlying index structure, concurrency control or 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 grid index and the B$^+$-tree can be performed very efficiently. To keep the objects well organized, updates in the R-tree are quite complex. Guo et al. [2006] have shown that the pre-processing and tree optimization strategies employed in the TPR*-tree [Tao et al. 2003] result in extra delay in locking, and hence reduce the performance gain in query processing due to the preprocessing during insertions. Other techniques, such as the bottom-up update [Kwon et al. 2002], lazy update [Lee et al. 2003] and update memo [Xiong and Aref 2006], have been developed in the literature to improve the update performance of the R-tree. However, the update cost, although decreases with varying degrees, is still higher than those space partitioning based indexes [Chen et al. 2008b].

A recent work, STRIPES [Patel et al. 2004], is a hybrid of both space partitioning and data partitioning indexes. It utilizes the quad-tree as the underlying index, so that the way of partitioning is fixed but guided by the data distribution. Since the quad-tree is an unbalanced structure, dense regions are partitioned into finer quads and stored deeply in the tree. Updates and queries on these objects always incur higher overhead. The ST^{2}B-tree presented in this paper is also a data supervised space partitioning index. The ST^{2}B-tree adopts the same technique used in [Yu et al. 2001; Jagadish et al. 2005]. The space is first partitioned into Voronoi cells of a set of reference points, and objects inside a Voronoi cell are clustered into a group, which occupies a contiguous segment of the entire key space. In [Yu et al. 2001; Jagadish et al. 2005], objects inside each Voronoi cell are sorted by their distances to the corresponding reference point. The ST^{2}B-tree, however, further partitions the space of each sub-space with a uniform grid. An object is indexed by the nubmer of the grid cell it belongs to.

2.2 Indexes with Single Reference Time vs. Multiple Reference Times

As existing moving object indexes store objects as stationary points, some mechanisms are required to get a synchronized view of object locations to enable spatio-temporal queries on the objects. There are in general two mechanisms.

First, in indexes in [Saltenis et al. 2000; Tao et al. 2003], objects are stored with their motion features (e.g., locations and velocities) at a specific reference time t_{ref}. Given an update at t_{up}, the location at t_{ref}, derived from the current motion features, is inserted into the index. An insertion considers the current locations of objects, which are estimated from the motion features stored in the index, and inserts the updating object into the most appropriate leaf node.

[Jensen et al. 2004] and [Chen et al. 2008b] both show that indexes which organize objects in single reference time such as the TPR-tree suffer from significant performance degradation with time. Consider the case where two objects, which are stored in the same leaf node, have not been updated for quite a long time. Then, the MBR of the leaf node could become quite large to bound both objects. With increasing large MBRs, the overlap between upper level MBRs increases and the index performance deteriorates. It is shown in [Tao and Xiao 2008] that such deterioration can be alleviated when the update frequency is high enough, i.e., 10% objects update in each timestamp. However, this frequency is too high that indexes such as TPR-tree cannot afford. In addition, since the performance deterioration is mainly caused by those objects with low update frequencies, even a few such objects will degrade the performance.

On the other hand, indexes such as those presented in [Jensen et al. 2004; Patel et al. 2004; Yiu et al. 2008; Chen et al. 2008a] have multiple (e.g., two) reference times. The time axis is partitioned into intervals. Each interval is assigned a certain reference time. Objects are distributed into different groups according to their last updating time. An update moves the object from its old group into the group whose time interval covers the current updating time. Indexes of this category also use a single tree structure, e.g., R-tree, B$^+$-tree and quad-tree, to manage all objects. The underlying tree is divided into several logical subtrees each of which manages objects of a group, i.e., whose updating time belongs to the same time interval. We call this method the “multi-tree” technique.

As a result of object updates, the “multi-tree” technique actually rebuilds the index periodically. All objects are required to update at least once within a given time interval T so that the subtree with the oldest time interval is empty, i.e., all objects have been moved

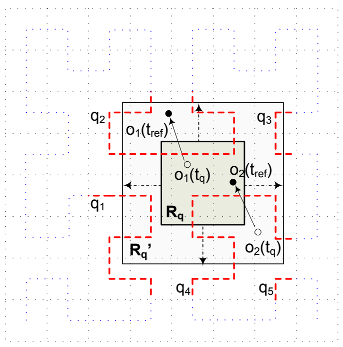


Fig. 3.1. Query Processing in the B^x-tree

to other subtrees. As a result, the oldest subtree can be reused for subsequent updates. A range query searches all subtrees with extended query regions. For each subtree, the query region is extended from the querying time to corresponding reference time using the maximum velocities of objects in the subtree.

Since T is the maximum update time interval of all objects, it could be quite large due to few infrequently updated objects. As a result, a query needs to search too many subtrees with large extended query regions. In [Chen et al. 2008a], the authors present a technique to improve the query performance at the expense of higher update cost, regarding the highly variable update frequencies. An object is inserted into multiple subtrees with different reference times (rather than the one covering the updating time only). The number of subtrees depends on the predicted time of the object's next update and a tunable parameter α. Instead of searching all subtrees, a query is performed on the α youngest subtrees only. α is used to balance the update and query performance. Query performance is improved as fewer subtrees are searched with smaller extended query regions.

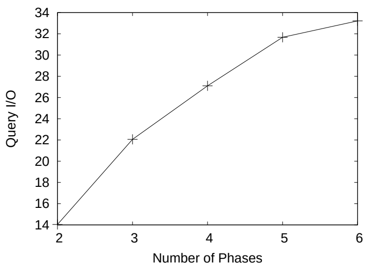
3. INDEX MOVING OBJECTS USING THE B$^+$-TREE

The ST^{2}B-tree shares the same basic design as the B^x-tree while bears the highly desired capability of self-tuning to avoid performance deterioration caused by imbalance and changes in workload. Before presenting the ST^{2}B-tree, we briefly introduce the B^x-tree first.

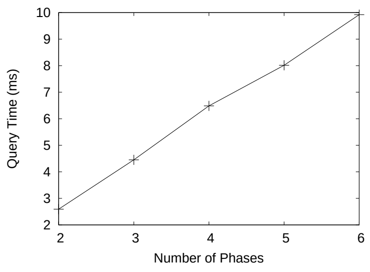
The B^x-tree [Jensen et al. 2004] is the first effort to adapt the B$^+$-tree to index moving objects. Grid cells are linearized with the help of a space filling curve. Objects are indexed in a B$^+$-tree with the id of the cell it belongs to. To process a range query, the B^x-tree searches each valid subtree 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 3.1, given a range query $R_q = (\overrightarrow{x_1}, \overrightarrow{x_2})$ at t_q, the enlarged query R'_q at t_{ref} is

$$R'_q = (\overrightarrow{x_1} - \overrightarrow{max_v} \cdot |t_{ref} - t_q|, \overrightarrow{x_2} + \overrightarrow{max_v} \cdot |t_{ref} - t_q|). \tag{3.1}$$

$\overrightarrow{x_1}$ and $\overrightarrow{x_2}$ are the lower-left and the upper-right corners of the of R_q. $\overrightarrow{max_v}$ represents the 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 3.1, at t_{ref}, o_1 is outside R_q and o_2 is inside. In contrast, o_1 is actually inside R_q



(a) Query I/O



(b) Query Time

Fig. 3.2. Effect of Number of Phases

while o_2 is outside R_q at t_q. 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 3.1, 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).

The basic query processing algorithm for the B^x-tree uses global maximum speed for query enlargement. This could result in oversized enlarged query, introducing a large number of false positives. Jensen et al. [2006] improve the B^x-tree query efficiency through more conscious query enlargements.

Recently, Yiu et al. [2008] present the B^{dual}-tree. The B^{dual}-tree is quite similar to the B^x-tree, except that space partitioning is applied to the dual space, i.e., both location and velocity are considered in deriving the $1d$ key. 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. With the MORs, the B^{dual}-tree uses R-tree-like query algorithms as in the TPR-tree. The B^{dual}-tree improves the query performance of the B^x-tree, since query enlargement is replaced by MORs enlargement. However, maintaining the MORs introduces high computation workload, which slows down the fast update of the B$^+$-Tree according to the findings in [Chen et al. 2008b]. 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.

As introduced in [Jensen et al. 2004], the B^x-tree can work with more than one subtree, each of which is called a "phase". Two phases are necessary to make the B^x-tree work properly, i.e., to enable the rollover between subtrees. In principle, the B^x-tree allows an arbitrary number of phases. With more phases, a query needs to search over more subtrees, but with smaller enlarged query regions. Figure 3.2 shows the effect of the number of phases on the index performance[1]. We find that the "multi-tree" strategy achieves the best query performance when the number of phases is minimized. Therefore, in the remaining part of the paper, we use two phases for all "multi-tree" indexes, including our ST^{2}B-tree. More phases are also applicable directly.

[1]These results are based on the settings discussed in Section 9.2

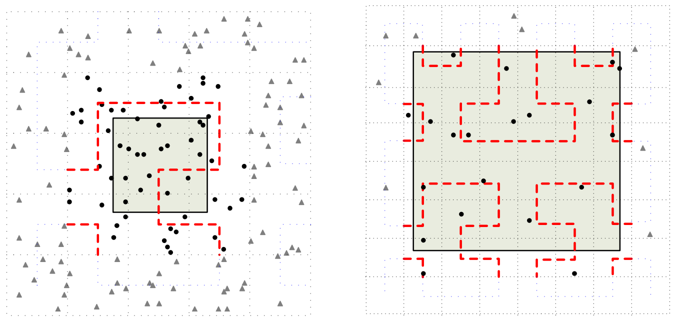


Fig. 4.1. The Co-relation of Data Density and Grid Granularity, Impact on Query Processing

4. ST^{2}B-TREE: A SELF-TUNABLE MOVING OBJECT INDEX

In this section, we first examine the challenges that motivate the self-tuning of a moving object index. Next, we introduce the structure and basic query algorithm of the ST^{2}B-tree. Then we explain why the ST^{2}B-tree is amendable to tune itself to fit the workload changes in MOD.

4.1 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 diversities 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.

4.1.1 *Impact of Data Density and Space Partition*

To use the B$^+$-tree for indexing spatial data, the first step is to reduce the dimensionality of objects, i.e., mapping spatial data into $1d$ data. Typically, the data 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 the partitions exert a joint effect on the index performance.

High Density or Coarse Grid: When the density of objects is high, or a coarse grid is used to partition the space, each cell will contain many objects. During query processing, we have to check all objects in the cells that intersect with the query. This may lead to a large number of false positives for those cells that are on the boundaries. For example, in Figure 4.1(a), all objects in the 3×3 cells (solid dots) that intersect with the query (the dark square) must be examined, and 8 of these are boundary cells where most of their points are not in the answers. 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. In most commercial databases such as IBM DB2, Oracle and MS SQL Server, records with duplicate keys are stored in overflow pages chained from the leaf nodes. This means that update cost will be higher as overflow pages have to be read and searched linearly. The existence of too many overflow pages compromises the balance property which bounds the search cost of

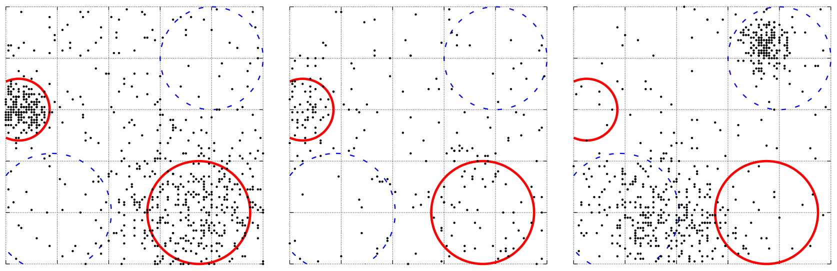

(a) Case 1: at t_0 (b) Case 2: at t_1 (c) Case 3: at t_2

Fig. 4.2. Spatial and Temporal Data Diversity

the B$^+$-tree structure.

Low Density or Fine Grid: At the opposite end, if the density of objects is low, or a fine grid is used to partition the space, few objects will be contained in a cell. For example, we can partition the space in Figure 4.1(a) with a finer grid. Figure 4.1(b) zooms in on the query region in Figure 4.1(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 smaller boundary cells contain fewer objects. However, the number of $1d$ range queries needed increases (from 2 to 9 in Figure 4.1, shown as dotted, thick lines). Although the overhead to prune false positives is reduced, the increase in the number of $1d$ range queries deteriorates query performance.

4.1.2 *Types of Data Diversity*

Intrinsically, moving objects are spatial objects whose positions change with time. The differences and changes of data fall into the following three categories as illustrated in Figure 4.2.

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 higher object densities than other places. Such a situation is portrayed in Figure 4.2(a). At time instance t_0, we have two hotspots (enclosed in solid circles), and two other regions with relatively lower object densities (enclosed in dashed circles).

Diversity with Time: In a moving object environment, the number of moving objects changes with time. For example, we expect many vehicles in the day, causing heavy traffic load on the roads. In contrast, traffic is relatively light at night (as fewer vehicles roam on the road). Referring to our running example, from time t_0 (Figure 4.2(a)) to t_1 (Figure 4.2(b)), hotspots are still hotspots; however, the densities of all areas decrease significantly.

Diversity in Space with Time: It is not uncommon to have a combination of the above types of diversities, i.e., both the density and distribution of moving objects change with time. In our example, from time t_1 (Figure 4.2(b)) to time t_2 (Figure 4.2(c)), besides an increase in the total number of objects, the hotspots move as well. Sparse areas may become dense while dense areas may become sparse due to some external factors such as peak hours, road work and accidents. As a real-life practical scenario, between 8am to 9am, people drive from the residential suburbs to downtown. During office hours, most

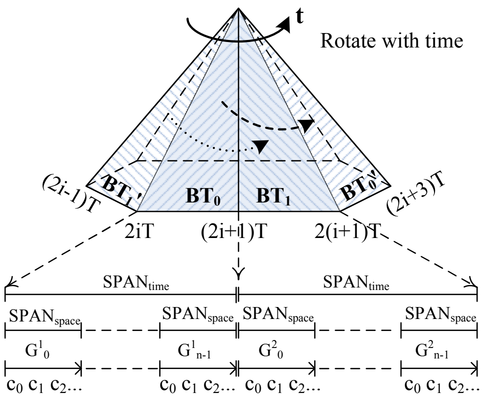


Fig. 4.3. The Essence of the ST^{2}B-tree

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, object density and the granularity of space partitioning greatly affect the efficiency of an index. Considering these three kinds of diversities in MODs, an index suffers from performance degradation when the space is evenly partitioned 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. Moreover, the index must be always online even in the midst of adaptation, so as not to disrupt or interfere with regular activities.

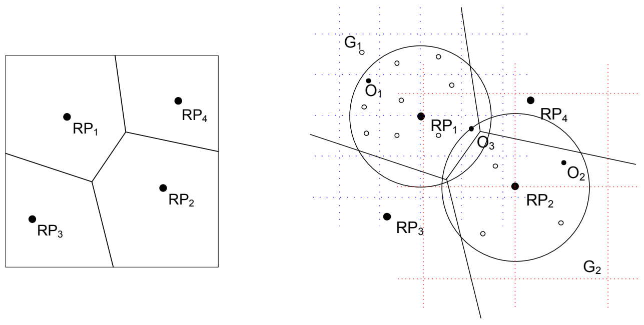
4.2 ST^{2}B-tree Structure

The ST^{2}B-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}.

4.2.1 *Index with Time*

In general, the ST^{2}B-tree adopts the "multi-tree" technique while dealing with the time dimension. Assume T_{up} is the maximal time interval between contiguous updates of an object, which means that an object is updated at least once in time interval T_{up}. The ST^{2}B-tree logically splits the B$^+$-tree into two subtrees, BT_0 and BT_1. Each subtree is assigned a range of T consecutive time points, where $T = T_{up}$. Specifically, the time ranges covering BT_0 and BT_1 are $[2iT, 2iT+T)$ and $[2iT+T, 2iT+2T)$ respectively, as time elapses, the value of i increases from zero and the time ranges of the two subtrees roll over alternately. The index therefore rolls over with time. This behavior is illustrated in Figure 4.3.

Without loss of generality, suppose that current time is $t \in [2iT, 2iT + T)$. As shown in Figure 4.3, BT'_1 is the older subtree, i.e., the one covering the earlier time interval $[2iT - T, 2iT)$, while BT_0 is the younger subtree, i.e., the one covering the current time interval $[2iT, 2iT+T)$. BT'_1 is now in a monotonic shrinking phase — only deletions affect BT'_1, and all insertions are conducted in the other subtree BT_0. Since $T = T_{up}$, all objects



(a) Space Partition (b) Grouping Objects

Fig. 4.4. ST^{2}B-tree: Spatial Key Generation

would be updated during $[2iT, 2iT + T)$. As a result, at the next transition time $2iT + T$, all objects are stored in the younger subtree BT_0, while the older subtree BT_1' becomes empty. Subsequently, a new time interval $[2iT + T, 2iT + 2T)$ is assigned to the empty subtree BT_1', which is now represented as BT in Figure 4.3. This change of time interval for the subtree BT_1 does not interfere with any objects which are currently indexed in BT_0. The older subtree BT_1 is refreshed while the original younger subtree BT_0 now becomes the “older” one and enters the shrinking phase. In the next transition time $2iT + 2T$, BT_0 will become empty and assigned with the newer time interval $[2iT + 2T, 2iT + 3T)$, and so on and so forth.

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}, $\overrightarrow{x}' = \overrightarrow{x} + \overrightarrow{v} \cdot (T_{ref} - t_{up})$. Herein, we set T_{ref} to the upper boundary of the time range, which is:

$$T_{ref} = (i + 1)T, \text{if } t_{up} \in [iT, iT + T). \tag{4.1}$$

We will discuss the selection of a proper reference time for a subtree later in Section 7.1.

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

$$KEY_{time} = \begin{cases} 0, & \text{if } t_{up} \in [2iT, 2iT + T), \\ 1, & \text{if } t_{up} \in [2iT + T, 2iT + 2T). \end{cases} \tag{4.2}$$

4.2.2 *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, that is, the partitioning forms a Voronoi Diagram of the n reference points as illustrated in Figure 4.4(a). Each reference point RP_i maintains a grid G_i, which is centered 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

Algorithm 4.1 computeKey($\overrightarrow{x}$)

```
1: $KEY_{time} = \lfloor t_c/T \rfloor \mod 2$;
2: $KEY_{space} = KEY_{space}(x)$;
3: $key = KEY_{time} * SPAN_{time} + KEY_{space}$;
```

Algorithm 4.2 Update($oid, \overrightarrow{x}, \overrightarrow{v}, \overrightarrow{x}_{pre}, \overrightarrow{v}_{pre}, t_{pre}$)

```
1: $Delete(\overrightarrow{x}_{pre}, \overrightarrow{v}_{pre}, t_{pre})$
2: $T_{ref} = \lceil t_c/T \rceil * T$;
3: $\overrightarrow{x}' = \overrightarrow{x} + (T_{ref} - t_c) * \overrightarrow{v}$;
4: $key = computeKey(x')$;
5: $Insert(key, (oid, \overrightarrow{x}', \overrightarrow{v}))$;
```

KEY_{space} is:

$$KEY_{space}(o) = i \times SPAN_{space} + cid(\overrightarrow{x}', G_i), \tag{4.3}$$

where i is the grid id, i.e., the id of the reference point closest to 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^{2}B-tree utilizes the Hilbert curve as shown in Figure 4.1. 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 4.4(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^{2}B-tree, object o is indexed with KEY_{ST^2}:

$$KEY_{ST^2} = KEY_{time} \times SPAN_{time} + KEY_{space}, \tag{4.4}$$

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

Algorithm 4.1 computes KEY_{ST^2} as in Equation 4.4 and Algorithm 4.2 shows the steps of an update, where t_c is the current time. KEY_{time}, T_{ref} and KEY_{space} are derived from the update time t_{up} based on Equations 4.2, 4.1 and 4.3 respectively. To process an update, the old entry of the object is first deleted from the index (line 1 in Algorithm 4.2). Together with the current location $\overrightarrow{x}$ and velocity $\overrightarrow{v}$, the object also sends the time, its location $\overrightarrow{x}_{pre}$ and velocity $\overrightarrow{v}_{pre}$ of the previous update. $Delete(\overrightarrow{x}_{pre}, \overrightarrow{v}_{pre}, t_{pre})$ first computes the indexing key of the old entry based on $\overrightarrow{x}_{pre}$, $\overrightarrow{v}_{pre}$ and t_{pre}. With the old indexing key, the old entry can be located and deleted from the B$^+$-tree. When the old record is deleted, Procedure $Insert(key, record)$ inserts the $record$ of the object into the B$^+$-tree using the new indexing key.

Figure 4.3 shows the essence of the ST^{2}B-tree. The current time is in $[2iT, 2iT + T)$. The two logical subtrees are BT_0 and BT_1. At time $2iT + T$, the time range of BT_1' has

Algorithm 4.3 Range Query

```
Input:  Query region $R_q$, query time $t_q$
Output: All objects in $R_q$ at $t_q$
 1: $result = \emptyset$
 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 $p_{t_q} \in R_q$ then
11:             add $o$ into result;
12: return $result$;
```

changed to $[2iT + T, 2iT + T)$, shown as BT_1. The next rotation will happen at $2iT + T$. BT_0 will be assigned a newer 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.

4.3 Snapshot Query Algorithms

Algorithm 4.3 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 3.1). Then, the grids of the reference points whose Voronoi cell intersects with the enlarged query region need 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 starts with an initial search radius r. If kNNs are not found in the initial search region, it extends the search radius by $increment$. Both r and $increment$ are set to D_k/k as in [Jensen et al. 2004], where

$$D_k = \frac{2}{\sqrt{\pi}}[1 - \sqrt{1 - \sqrt{\frac{k}{N}}}]. \tag{4.5}$$

D_k is the estimated distance to the k'th nearest neighbor [Tao et al. 2004] and N is the number of objects in a unit space. Due to space constraints we omit the detailed algorithm here and a similar procedure can be found in [Jensen et al. 2004].

4.4 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 4.1.

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 4.4(b), objects are relatively dense around RP_1, therefore G_1 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.

Diversity with Time: Based on the constraint that T is the maximum update time interval, at each transition time iT, the subtree in ST^{2}B-tree to be refreshed with a new time interval is always empty. At this point, the ST^{2}B-tree can use a different granularity of grids for this empty subtree, according to the object densities investigated in the previous T time. The grids of the other non-empty subtree are kept unchanged and all the objects currently in the index are unaffected. Therefore, the two subtrees in the ST^{2}B-tree can have their own set of grids. While searching/updating in a subtree, the corresponding set of grids is used. No collision happens. Once the granularity are determined, they cannot be changed during the next T time before the next transition time.

Diversity in Space with Time: In a similar way as the granularity of grids can be tuned to capture the change of object density with time, the number and positions of reference points can also be tuned to capture the change of object distribution with time. For example, in Figure 4.2, 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, thanks to the "multi-tree" structure of the ST^{2}B-tree, the two subtrees work independently without interference. Any settings about the indexing, including both reference points and grid granularity, are tunable in the ST^{2}B-tree, in addition, the ST^{2}B-tree meets the two requirements mentioned in Section 4.1. We shall discuss the self-tuning strategy of the ST^{2}B-tree in Section 8, which helps the ST^{2}B-tree perform better with time-dependent workload changes.

5. EAGER UPDATE: MANAGING OBJECT MIGRATION DURING ROLLOVER

As discussed in the previous section, the "multi-tree" indexing technique opens the door for tuning the index online. In addition, as indicated in Section 2.2, with the "multi-tree" technique, an object is indexed with a reference time t_{ref}, growing in step with the current time. The overall query performance of a "multi-tree" index will not deteriorate as time elapses. This is in contrast with the single-tree indexes such as the TPR-tree [Saltenis et al. 2000].

5.1 Effect of T: the length of the time interval covered by a subtree

The effectiveness and efficiency of the "multi-tree" structure depend largely on the value of T, i.e., the length of the time interval covered by a subtree.

Query Cost: The subtree rollover at the transition time iT is feasible only if no existing objects are affected, i.e. when the old subtree contains no objects. This condition is guaranteed by constraining the length of the time interval covered by a subtree T to be at least the maximum time interval between contiguous updates of an object T_{up}, as mentioned in Section 4.2.1. In practice, some objects update very infrequently, resulting in a large T_{up} and a large T as well. As aforementioned, in query processing, the ST^{2}B-tree avoids false negatives by query enlargement. The query performance largely depends on the query enlargement, while the query enlargement is proportional to the value of T. With a larger T, a query is enlarged into a larger size, which incurs higher query processing costs. In the extreme case when $T \rightarrow \infty$, the "multi-tree" index degrades to a single tree index. The query enlargement keeps increasing as time elapses and the query performance deteriorates significantly with a larger T.

Objects Migration: Since the length of T affects the query performance significantly, we consider setting T as small as possible. However, if T is set smaller than the maximum update interval T_{up}, at the point of transition, there may be some remaining objects in the older subtree. For the "multi-tree" technique to work correctly, we have to migrate all these un-updated objects to the younger subtree manually before assigning a new time interval to the subtree.[2] In particular, objects are inserted into the younger subtree with their locations at the corresponding reference time. This location is estimated from the location and velocity stored in the older subtree. The migration of un-updated objects incurs extra update workload on the index. Furthermore, since the migration happens at the transition time, it causes a burst of updates. All the upcoming regular updates are postponed until the migration finishes and the older subtree is empty. Although, a query can be processed as usual, the response time may be long due to lock contention caused by those updates. From this point of view, T should be as large as possible so as to minimize the migration cost. The smaller T is, the larger portion of objects need to be migrated, and the more frequent migration happens.

In brief, the value of T affects the query cost and the migration cost in opposite ways. We will discuss the effect of T on both the query and update theoretically in Section 7.2.

5.2 Eager Update

As discussed above, a larger T leads to a higher in query cost, while a smaller T may introduce extra migration cost. In order to avoid migration as much as possible while keeping T relatively small, we now introduce an eager update technique.

The principle of eager update is to update as many objects as possible without increasing the number of I/O accesses. Algorithm 5.1 shows the steps of an eager update. Consider an update in the form of $(oid, \overrightarrow{x}, \overrightarrow{v})$, where oid is the identity of the object, $\overrightarrow{x}$, $\overrightarrow{v}$ are the current location and velocity of the object.

The update procedure first finds the record of object oid. $oldkey$ is the current indexing key of oid and L is the leaf node that contains the record (lines 1-2). If the object is already indexed in the younger subtree (line 4), the update continues in a normal way as

[2] An alternative way of dealing with these un-updated objects is to discard them from the database, simply assuming that those objects have left the system (e.g., parking). No migration is required in this case. The older subtree is treated as empty. All the active objects are indexed by the other subtree and kept unaffected by the tuning. We do not consider this situation here.

Algorithm 5.1 EagerUpdate(oid, $\overrightarrow{x}$, $\overrightarrow{v}$)

```
1: $L = search(oid)$;
2: $oldkey$ = key of the existing record of $oid$ in $L$;
3: $KEY_{time} = \lfloor oldkey/SPAN_{time} \rfloor$;
4: if $KEY_{time} = t/T \mod 2$ then
5:    continue with Update($oid$, $\overrightarrow{x}$, $\overrightarrow{v}$);
6: else
7:    $T_{ref} = \lceil t/T \rceil * T$;
8:    $\overrightarrow{x}' = \overrightarrow{x} + \overrightarrow{v} * (T_{ref} - t)$;
9:    $key = computeKey(\overrightarrow{x}')$;
10:   $Q = \emptyset$
11:   for each record $r_i = (key_i, oid_i, \overrightarrow{x}_i, \overrightarrow{v}_i)$ in the leaf node $L$ do
12:      $\overrightarrow{x}' = \overrightarrow{x}_i + \overrightarrow{v}_i * T$;
13:      $key' = computeKey(\overrightarrow{x}')$;
14:      if $key == key'$ then
15:         Q = Q $\cup$ ($key, oid_i, \overrightarrow{x}', \overrightarrow{v}_i$);
16:         delete $r_i$ from L;
17:   $L' = search(key)$;
18:   for each $r_i \in Q$ do
19:      insert $r_i$ into L';
```

in Algorithm 4.2 (line 5). Otherwise, if the update needs to move the object from the older subtree to the younger one, the eager update checks all the other objects in the same leaf node L (line 11), and computes their estimated locations in the reference time of the younger subtree (line 12). If the object will be indexed by the same key as the object oid (line 14), this record is inserted into a queue Q (line 15) and then is deleted from the current leaf node L (line 16). Q maintains objects to be updated in an eager manner. Subsequently, the update procedure finds the leaf node L' in the younger subtree which object oid is to be inserted into (line 17). Finally, all objects in Q are inserted into L', since they have the same indexing key as object oid (lines 18-19).

Benefits of Eager Update:

As we can see, an eager update accesses the same tree nodes as a regular update. Therefore, additional I/Os are incurred only when the eager update causes node splitting or merging while the regular update does not. With eager update, a regular update moves more than one object to the younger tree. As a result, in T time (supposing T is smaller than the maximal update time interval), more objects are updated to the younger tree than usual. Fewer objects are left in the older subtree and the migration cost decreases. Another benefit of eager update is that it saves the cost of regular update as well. If an object has been updated into the younger subtree eagerly, the real update of the object affects the younger subtree only, i.e., both deletion and insertion happen in the younger subtree. Considering that objects move continuously in the space, it is likely that the deletion and insertion access the same leaf node. The insertion and deletion of an update work along the same path. The number of I/Os could decrease, with an LRU buffer, even if it is small in volume. On the contrary, if eager update is not supported, the deletion operation would delete an entry in the older subtree and the insertion operation would insert an entry in the younger subtree. In this case, even if the object does not move at all, different set of nodes will be visited during these two steps of the update. We shall see the effect of migrations and costs of different updates in Section 9.3.

Varying Degree of Eagerness:

In Algorithm 5.1, all objects, which have the same key as the core updating object in the younger subtree, are updated eagerly. Here, we introduce a tuning knob, denoted as $\mathcal{D}_e$, to more aggressively migrate objects from the older subtree to the younger subtree. Essentially, all objects whose indexing keys in the younger subtree differ no more than $\mathcal{D}_e$ are migrated from that of the core updating object. The intuition here is that objects with similar indexing key values are likely to be geographically close together. However, since objects move continuously in space, there is no guarantee on where those objects will be (in the younger subtree) after T time. As such, as $\mathcal{D}_e$ increases, the I/O overhead will also increase. Moreover, since objects with key values smaller than $\mathcal{D}_e$ may be found in other nodes, accessing these nodes will further increase the I/O cost. To handle the second problem, we restrict the migration to only those objects that are in the same leaf node as the core updating objects. In this way, $EagerUpdate$ corresponds to the case when $\mathcal{D}_e = 0$. On the other extreme, with $\mathcal{D}_e = \infty$, all objects within the leaf node will be migrated. A moderate value of $\mathcal{D}_e$ will result in migrating only subset of the objects within the node.

6. GRID GRANULARITY

As discussed in Section 4.1, 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 for the ST^{2}B-tree and any other space partitioning indexes. To find the optimal granularity, we now analyze the effects 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^{2}B-tree with local uniform assumption around each reference point.

The notations used are listed in Table 6.1. We start our analysis by giving a definition of the grid order λ.

DEFINITION 6.1. *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 the 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 6.2 estimates the number of leaf I/Os [Yiu et al. 2008].

LEMMA 6.2. *The number of leaf node accesses of a square-sized range query is:*

$$IO_L = N_L[L + L_q + V \cdot |t_q - T_{ref}|]^2. \tag{6.1}$$

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 $L_Q = L_q + V \cdot |t_q - T_{ref}|$. 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 to be accessed by a query is therefore $N_L[L + L_Q]^2$.

Table 6.1. Notations for Analyzing Grid Granularity

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

However, Equation 6.1 in Lemma 6.2 is valid only when there are 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 6.2.

LEMMA 6.3. *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 at most 1 object and duplicate keys are rare. $N_L = \frac{N}{f}$ and $f = 69\% \cdot C_L$, 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 of λ. □

LEMMA 6.4. *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 6.4 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 6.5. *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. This can be easily deduced from Lemma 6.3-6.4, IO_L keeps unchanged when λ is larger than $\frac{1}{2}\log_2 N$ (Lemma 6.3), and increases when λ is smaller than $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L$ (Lemma 6.4). □

According to Corollary 6.5, in order to minimize the number of leaf node accesses during a query, the space filling curve with resolution $\frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \leq \lambda \leq \frac{1}{2}\log_2 N$ should be used.

While Corollary 6.5 focuses on the I/O overhead of the leaf nodes, we now consider the overhead of internal node accesses. The problem with 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 6.6. *The number of $1d$ range queries N_q and internal node accesses N_I increases with λ.*

PROOF. As proven in [Moon et al. 2001], 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 increases, 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 9.2, 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 the node and postponing concurrent updates accessing the same node. Therefore, according to Lemma 6.6 and Corollary 6.5, $\lceil \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \rceil$ 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 node into which the new record is to be inserted. Writing it 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 does the number of overflow pages.

In summary, a smaller λ within the range indicated in Corollary 6.5 leads to better query performance; nevertheless it incurs higher update costs. As we shall see in Section 9.1, the

Table 7.1. Notations for Analyzing Time-related Parameters

Symbol	**Description**
N_i	number of objects in BT_i
T_{up}	maximum update time interval
T	length of the time interval covers a subtree
T_{refi}	reference time of subtree BT_i
T_{li}	lower boundary of BT_i's time range
T_{ui}	upper boundary of BT_i's time range
tr	offset between T_{refi} to T_{li} ($T_{refi} - T_{li}$)
CQ_{avg}	average cost of a query
CU_{avg}	average cost of an update
CU_1	average cost of an update involving only one subtree
CU_2	average cost of an update involving both subtrees
CU_m	average cost of migrating an object

costs of query and update achieve the best trade-off when

$$\lambda = \lceil \frac{1}{2}\log_2 N - \frac{1}{2}\log_2 C_L \rceil. \tag{6.2}$$

In the analysis in this section, we have an implicit uniform assumption on query distribution. The performance of an index is sensitive to query distributions as well. Here, we simplify our cost model to uniform queries for the ease of analysis. We will show the impact of query distribution experimentally in Section 9.

7. TIME RELATED PARAMETERS

In this section, we will discuss the selection of two parameters of the ST^{2}B-tree: the reference time T_{ref} and the length of the time interval T covered by a subtree in the ST^{2}B-tree currently. Both are critical parameters for the "multi-tree" structure, having significant impact on the overall index performance. Table 7.1 shows the notations we use in the following analysis in addition to the notations in Table 6.1.

7.1 Reference Time of a Subtree: T_{ref}

In Section 4, as shown in Equation 4.1, we simply set the reference time T_{ref} to be the upper boundary of the time interval of a subtree. In fact, the reference time of a subtree does affect query performance. Since a query is enlarged into T_{ref} using maximum object velocity max_v, T_{ref} and max_v have a joint effect on query enlargement. max_v is a data-related parameter, over which the system has no control, while T_{ref} is a system parameter. In the following, we try to find an "optimal" T_{ref} which minimizes the average query cost CQ_{avg}. Since T_{ref} only affects query performance of the indexand has no effect on the update performance, update cost is not our concern here.

Since each subtree covers a temporal range of length T, the query cost exhibits a periodic variation. Therefore, we investigate the average query cost CQ_{avg} in $[T_l, T_u)$, where T_l and T_u are the lower and upper boundary of the time interval covered by the younger subtree. Assuming that queries arrive evenly in time,

$$CQ_{avg} = \frac{1}{T_u - T_l}\int_{T_l}^{T_u} CQ(t)dt, \;\; i = 1, 2, ... \tag{7.1}$$

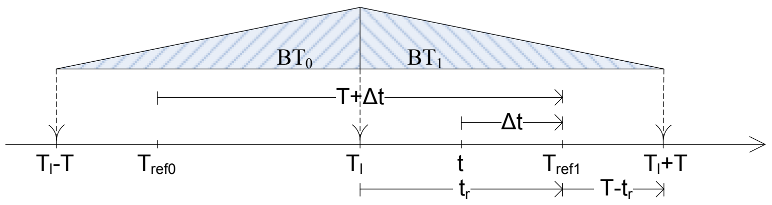


Fig. 7.1. Graphic Representations for Notations

where $CQ(t)$ is the cost of a query at time t. Given T as the length of the time interval,

$$CQ_{avg} = \frac{1}{T}\int_{T_l}^{T_l+T} CQ(t)dt,\;\; i = 1, 2, ...$$

In Section 6, the number of leaf I/O for a query is estimated as $IO_L = N_L[L + L_Q]^2 = \frac{N}{f}[(\frac{f}{N})^{\frac{1}{2}} + L_Q]^2 = 1 + 2(\frac{N}{f})^{\frac{1}{2}}L_Q + NL_Q^2$. The cost is dominated by NL_Q^2. To simplify the analysis, we now estimate the query cost using NL_Q^2, i.e., the number of objects contained in the enlarged query L_Q. The experimental results in Section 9 empirically confirm the fact that the query cost is proportional to the number of objects in the enlarged query. Consider a square-sized range query with side length L_q enlarged with speed V along each side,

$$CQ(t) = N_0(L_q + V|t - T_{ref_0}|)^2 + N_1(L_q + V|t - T_{ref_1}|)^2, \tag{7.2}$$

where N_i represents the number of objects in the corresponding subtree BT_i at time t. $(L_q + V|t - T_{ref_i}|)^2$ is the enlarged query range of subtree BT_i. The cost of a query consists of two parts, i.e., the cost of processing the enlarged query over each subtree.

Combining Equations 7.1 and 7.2,

$$CQ_{avg} = \frac{1}{T}\int_{T_l}^{T_l+T} \left[(L_q + V|t - T_{ref_0}|)^2 N_0 + (L_q + V|t - T_{ref_1}|)^2 N_1\right] dt.$$

Without loss of generality, we assume that BT_0 is older than BT_1 in the current ST^{2}B-tree. Figure 7.1 illustrates the meaning of the notations. For BT_1, the query region is enlarged by Δt, where Δt is the time difference between T_{ref1} to the query time t, i.e., $\Delta t = t - T_{ref_1}$. Since $T_{ref1} = T_{ref0} + T$, the query region is enlarged by $T + \Delta t$ for BT_0. , then $\Delta t + T = t - T_{ref_0}$. Let $tr = T_{ref_1} - T_l$, which is the offset between the reference time to the lower boundary of BT_1's time range. When the query time t varies from T_l to $T_l + T$, Δt varies from $-tr$ to $T - tr$. Then, CQ_{avg} can be alternatively represented by the following equation:

$$CQ_{avg} = \frac{1}{T}\int_{-tr}^{T-tr} \left[N_0(L_q + V|\Delta t + T|)^2 + N_1(L_q + V|\Delta t|)^2\right] d\Delta t.$$

(1) The case when $T \le T_{up}$:

Here, T_{up} is the maximum update time and N is the total number of objects. Suppose that object updates are uniformly distributed in T_{up} time. If $T \le T_{up}$, the numbers of objects in BT_0 and BT_1 are:

$$\begin{cases} N_0 = N - \frac{t-T_l}{T_{up}}N \\ N_1 = \frac{t-T_l}{T_{up}}N \end{cases} \qquad T \le T_{up}. \tag{7.3}$$

At the last transition time T_l, BT_1 is empty while BT_0 contains all N objects. Till timestamp t, $\frac{t-T_l}{T_{up}}N$ objects have been updated to BT_1 and $N - \frac{t-T_l}{T_{up}}N$ objects remain in the older subtree BT_0.

According to Equation 7.3, we have

$$CQ_{avg} = \frac{N}{TT_{up}} \int_{-tr}^{T-tr} \left[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) + (L_q + V|\Delta t|)^2 (\Delta t + tr) \right] d\Delta t.$$

LEMMA 7.1. *When $T \le T_{up}$, CQ_{avg} is minimized at*

$$tr = T + \alpha T_{up} - \sqrt{\alpha^2 T_{up}^2 + \alpha T^2 - \alpha T_{up} T} \in [T, 2T], \text{ where } \alpha = 1 + VT/2L_q.$$

PROOF.

$$\begin{aligned} CQ_{avg} &= \frac{N}{TT_{up}} \int_{-tr}^{T-tr} \left[(L_q + V|\Delta t + T|)^2 (T_{up} - \Delta t - tr) + (L_q + V|\Delta t|)^2 (\Delta t + tr) \right] d\Delta t \\ &= \frac{N}{TT_{up}} E_1 + 2L_q V \frac{N}{TT_{up}} E_2, \end{aligned}$$

where

$$\begin{aligned} E_1 &= \int_{-tr}^{T-tr} \left[L_q^2 T_{up} + V^2 [T_{up} \Delta t^2 + (T^2 + 2T\Delta t)(T_{up} - \Delta t - tr)] \right] d\Delta t, \\ E_2 &= \int_{-tr}^{T-tr} (\Delta t + tr)|\Delta t|) d\Delta t + \int_{-tr}^{T-tr} (T_{up} - \Delta t - tr)|\Delta t + T| d\Delta t, \end{aligned}$$

$$E_1 = L_q^2 T_{up} T + V^2 [T_{up} T tr^2 + T^3 tr - 3T_{up} T^2 tr + \frac{7}{3} T_{up} T^3 - \frac{7}{6} T^4].$$

E_1 is minimized at $tr = T + \frac{T_{up}-T}{2T_{up}}T$, which is between T and $2T$.

$$E_2 = \begin{cases} -\frac{1}{2}(T - T_{up})T^2 + T_{up}(T - tr)T + \frac{1}{3}tr^3 & \text{if } tr \le T, \\ \frac{1}{3}(T - tr)^3 + T_{up}(T - tr)^2 + T_{up}(T - tr)T + \frac{1}{2}(T_{up} - T + 2tr)T^2 - \frac{2}{3}T^3 & \text{if } T \le tr \le 2T, \\ \frac{1}{2}(T - T_{up})T^2 - T_{up}(T - tr)T & \text{if } tr \ge 2T. \end{cases}$$

When $tr \le T$, E_2 is minimized at $tr = T$, where

$$\min(E_2) = \frac{1}{2}(T_{up} - \frac{1}{3}T)T^2.$$

When $tr \ge 2T$, E_2 is minimized at $tr = 2T$, where

$$\min(E_2) = \frac{1}{2}(T_{up} + T)T^2.$$

Therefore, E_2 should be minimized when $tr \in [T, 2T]$.

Combining E_1 and E_2, if $T \le T_{up}$, CQ_{avg} should be minimized when $tr \in [T, 2T]$.

$$\begin{aligned} \min(CQ_{avg}) = & \frac{N}{TT_{up}} [L_q^2 T_{up} T + V^2 (T_{up} T tr^2 + T^3 tr - 3T_{up} T^2 tr + \frac{7}{3} T_{up} T^3 - \frac{7}{6} T^4)] \\ & + 2L_q V \frac{N}{TT_{up}} [\frac{1}{3}(T - tr)^3 + T_{up}(T - tr)^2 + T_{up}(T - tr)T + \frac{1}{2}(T_{up} - T + 2tr)T^2 - \frac{2}{3}T^3], \end{aligned}$$

which is minimized when $tr = T + \alpha T_{up} - \sqrt{\alpha^2 T_{up}^2 + \alpha T^2 - \alpha T_{up} T}$, where $\alpha =$ 1+$VT/2L_q$. □

From Lemma 7.1, we see that CQ_{avg} is minimized when the reference time T_{ref_1} is sometime between $T_l + T = T_u$ and $T_l + 2T = T_u + T$. The "optimal" reference time varies with different queries, i.e., different L_q and V. To simplify the problem, we consider the case that T is equal to T_{up}.

COROLLARY 7.2. *If $T = T_{up}$, all objects have been updated to the younger subtree in T time. CQ_{avg} is minimized when $tr = T$. T_{ref_1} is set to $T_l + T = T_u$, which is the upper boundary of its time range, the same as shown in Equation 4.1.*

PROOF. Derive from Lemma 7.1. □

Object migration can be reduced, or even avoided, by using the eager updates as introduced in Section 5. Since update cost is not our concern here, we can assume that all objects are eagerly updated to the younger subtree BT_1 within T time. As a result, although T is actually smaller than T_{up}, Corollary 7.2 also holds. The average query cost is minimized when T_{ref} is set to the upper boundary of the time range of a subtree, that is, when $tr = T$, we have:

$$CQ_{avg} = N\left[L_q^2 + \frac{V^2}{12}(4T^2 - 2T^2\frac{T}{T_{up}}) + \frac{L_qV}{6T}(6T^2 - 2T^2\frac{T}{T_{up}})\right] \tag{7.4}$$

(2) The case when $T > T_{up}$: If $T > T_{up}$, the numbers of objects in BT_0 and BT_1 are:

$$\begin{cases} N_0 = N - \frac{t-T_l}{T_{up}}N \text{ and } N_1 = \frac{t-T_l}{T_{up}}N & \text{if } T_{up} < t \leq T_l + T_{up}, \\ N_0 = 0 \text{ and } N_1 = N & \text{if } t \geq T_l + T_{up}. \end{cases} \tag{7.5}$$

Before $T_l + T_{up}$, i.e., $t \leq T_l + T_{up}$, N_0 and N_1 are the same with Equation 7.3. After $T_l + T_{up}$, i.e., $T_{up} < t \geq T_l + T_{up}$ all objects are moved to the younger subtree BT_1 and BT_0 is empty. According to Equation 7.5 , we have

$$\begin{aligned} CQ_{avg} = &\frac{N}{TT_{up}}\int_{-tr}^{T_{up}-tr}\left[(L_q + V|\Delta t + T|)^2(T_{up} - \Delta t - tr) + (L_q + V|\Delta t|)^2(\Delta t + tr)\right]d\Delta t \\ &+ \frac{N}{T}\int_{T_{up}-tr}^{T-tr}\left[(L_q + V|\Delta t|)^2\right]d\Delta t. \end{aligned}$$

LEMMA 7.3. *If $T \geq T_{up}$, CQ_{avg} is minimized when $tr = \frac{1}{2}(T + T_{up})$. $T_{ref_1} = T_l + \frac{1}{2}(T + T_{up})$ and*

$$CQ_{avg} = N\left[L_q^2 + \frac{V^2}{12}(T^2 + T_{up}^2) + \frac{L_qV}{6T}(3T^2 + T_{up}^2)\right]. \tag{7.6}$$

PROOF.

$$\begin{aligned} CQ_{avg} = &\frac{N}{TT_{up}}\int_{-tr}^{T_{up}-tr}\left[(L_q + V|\Delta t + T|)^2(T_{up} - \Delta t - tr) + (L_q + V|\Delta t|)^2(\Delta t + tr)\right]d\Delta t \\ &+ \frac{N}{T}\int_{T_{up}-tr}^{T-tr}\left[(L_q + V|\Delta t|)^2\right]d\Delta t \\ = &\frac{N}{TT_{up}}E_1 + 2L_qV\frac{N}{TT_{up}}E_2 + \frac{N}{T}E_3, \end{aligned}$$

where

$$\begin{aligned} E_1 &= \int_{-tr}^{T_{up}-tr}\left[L_q^2T_{up} + V^2[T_{up}\Delta t^2 + (T^2 + 2T\Delta t)(T_{up} - \Delta t - tr)]\right]d\Delta t, \\ E_2 &= \int_{-tr}^{T_{up}-tr}(\Delta t + tr)|\Delta t|)d\Delta t + \int_{-tr}^{T_{up}-tr}(T_{up} - \Delta t - tr)|\Delta t + T|d\Delta t, \\ E_3 &= \int_{T_{up}-tr}^{T-tr}\left[(L_q + V|\Delta t|)^2\right]d\Delta t. \end{aligned}$$

E_1 is minimized when $tr = \frac{1}{2}(T + T_{up})$ and,

$$\min(E_1) = L_q^2 T_{up}^2 + \frac{V^2}{12} T_{up}^2 [T_{up}^2 - 2TupT + 3T^2].$$

E_2 is minimized when tr is any time in $[T_{up} \leq T]$,

$$\min(E_2) = \frac{1}{6} T_{up}^2 (3T - T_{up}).$$

E_3 is minimized when $tr = \frac{1}{2}(T + T_{up})$

$$\min(E_3) = L_q^2 (T - T_{up}) + \frac{V^2}{12}(T - T_{up})^3 + 2L_q V \frac{1}{4}(T - T_{up})^2.$$

Therefore, CQ_{avg} is minimized, when $tr = \frac{1}{2}(T + T_{up})$ and

$$\min(CQ_{avg}) = N[L_q^2 + \frac{V^2}{12}(T^2 + T_{up}^2) + \frac{L_q V}{6T}(3T^2 + T_{up}^2)].$$

□

In summary, for the purpose of minimizing the average query cost, we set the reference time of a subtree as follows:

$$T_{ref} = \begin{cases} T_l + T & \text{if } T \leq T_{up}, \\ T_l + \frac{1}{2}(T + T_{up}) & \text{if } T > T_{up}. \end{cases} \tag{7.7}$$

Note that when $T \leq T_{up}$, Equation 7.7 is the same as Equation 4.1.

7.2 The Length of the Time Interval of a Subtree: T

While the reference time T_{ref} affects the query performance only, the length of the time interval T has an impact on both the queries and updates as discussed in Section 5.1.

Effect of T on Queries: Given that T_{ref} is determined according to Equation 7.7, the average query cost is as follows:

$$CQ_{avg} = \begin{cases} N[L_q^2 + \frac{V^2}{12}(4T^2 - 2T^2 \frac{T}{T_{up}}) + \frac{L_q V}{6T}(6T^2 - 2T^2 \frac{T}{T_{up}})] & \text{if } T \leq T_{up}, \\ N[L_q^2 + \frac{V^2}{12}(T^2 + T_{up}^2) + \frac{L_q V}{6T}(3T^2 + T_{up}^2)] & \text{if } T > T_{up}. \end{cases}$$

The average query cost CQ_{avg} always increases with larger T. In order to minimize the query cost, T should be as small as possible.

Effect of T on Updates: Now let us consider the update, concerning the average cost of each "real" update, i.e., update issued by the object actively.

LEMMA 7.4. *When $T \leq T_{up}$, the average cost of an update is:*

$$CU_{avg} = CU_2 + \frac{T_{up} - T}{T} CU_m.$$

PROOF. Suppose that objects update evenly in T_{up} time. $\frac{1}{T_{up}}N$ objects update at each timestamp. Just before the transition, a total number of $\frac{T}{T_{up}}N$ objects have been deleted from the older subtree and inserted into the younger subtree. The remaining N-$\frac{T}{T_{up}}N$ objects need to be migrated. The average cost of a "real" update is

$CU_{avg} = [CU_2 \frac{T}{T_{up}} N + \frac{T_{up} - T}{T_{up}} CU_m N] / \frac{T}{T_{up}} N$, which is $CU_{avg} = CU_2 + \frac{T_{up} - T}{T} CU_m$ □

LEMMA 7.5. *When $T > T_{up}$, the average cost of an update is:*

$$CU_{avg} = \frac{T_{up}}{T} CU_2 + \frac{T - T_{up}}{T} CU_1.$$

PROOF. Till $t = T_{up}$, all objects have been updated to the younger subtree actively and the total cost is $CU_2 N$. After T_{up} and before the transition, all subsequent updates operate on the younger subtree only. In the remaining $T - T_{up}$ time, there are a total number of $\frac{T-T_{up}}{T_{up}} N$ updates and the cost is $CU_1 \frac{T-T_{up}}{T_{up}} N$. As a result, the average update cost is $CU_{avg} = [CU_2 N + CU_1 \frac{T-T_{up}}{T_{up}} N] / \frac{T}{T_{up}} N$, which is $CU_{avg} = \frac{T_{up}}{T} CU_2 + \frac{T-T_{up}}{T} CU_1$ □

Combining Lemma 7.4 and 7.5,

$$CU_{avg} = \begin{cases} CU_2 + \frac{T_{up}-T}{T} CU_m & \text{if } T \leq T_{up}, \\ \frac{T_{up}}{T} CU_2 + \frac{T-T_{up}}{T} CU_1 & \text{if } T > T_{up}. \end{cases}$$

The three types of updates incur different costs, as we will see in Section 9.3. In essence, $CU_2 > CU_1 > CU_m$. If $T \leq T_{up}$, the average update cost is minimized when no migration happens at $T = T_{up}$. Otherwise, if $T > T_{up}$, it is better to maximize the percentage of single tree updates, hence T should be as large as possible.

The length of subtree time interval T has an opposite effect on query and update costs. In Section 9.5, we will investigate the effect of T with respect to various query/update ratios via empirical studies.

8. SELF-TUNING OF THE ST^{2}B-TREE

As discussed in Section 4, the multi-tree design makes the ST^{2}B-tree feasible for tuning. Sections 6 and 7 provide guidelines for choosing optimal values for the various parameters used in the ST^{2}B-tree. We now introduce how self-tuning is realized on the ST^{2}B-tree.

Figure 8.1 shows the tuning framework of the ST^{2}B-tree. The tuning framework adds four components on top of the underlying DBMS: the Index Profile, the Key-Gen, the Statistics and the Online Tuning.

8.1 Index Profile

Index Profile maintains the current settings of the ST^{2}B-tree. As shown in Figure 8.1, for each subtree BT_i, the global profile contains three parameters: the time interval of the subtree $[T_{l_i}, T_{u_i}]$ and the reference time T_{ref_i}. If we do not want to tune T—the length of the time interval covering a subtree—there is no need to maintain T_{l_i}, T_{u_i} and T_{ref_i}. When T is fixed, T_{l_i} and T_{u_i} can be derived from the current time, and T_{ref_i} is known according to Equation 4.1.

Besides global parameters, Index Profile keeps a reference table for the reference points in each subtree. Each reference point RP_j has an entry in the reference table, including its position, size of its grid G_j and granularity of its grid λ_j. If eager updates are allowed, Index Profile also keeps the degree of eagerness $\mathcal{D}_e$ currently used by the index.

8.2 Key-Gen

The Key-Gen module works as an interface between the ST^{2}B-tree and the underlying B$^+$-tree. On receiving an object update, it reads the index settings from the Index Profile that are necessary for computing KEY_{ST^2}, including the reference time and reference points of the younger subtree. It then calculates KEY_{ST^2} according to Equation 4.4. Finally, the

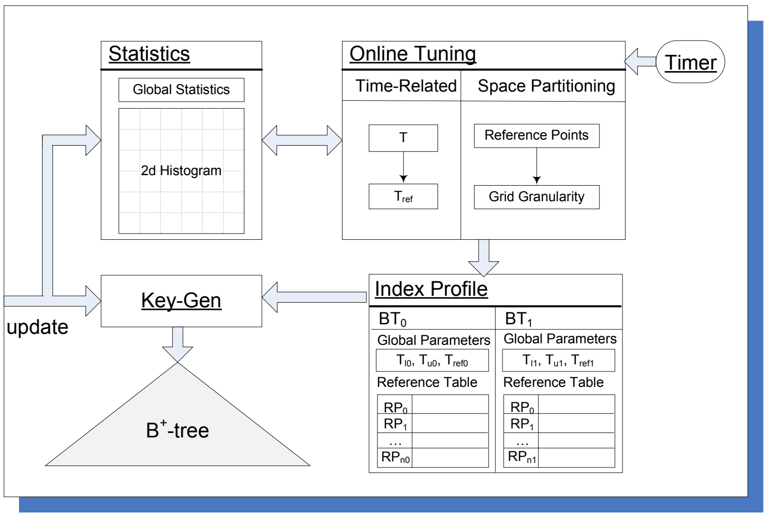


Fig. 8.1. Online Tuning Framework

update is performed over the B$^+$-tree with KEY_{ST^2} and new location and velocity of the object.

8.3 Statistics

The purpose of tuning is to make the index adaptive to the workload. The Statistics module maintains statistics about the workload in the current time interval, i.e., the time interval of the younger subtree. The statistics will be used to tune the index at the next transition time. While dealing with an object update, the statistics is updated accordingly. Right after the tuning process finishes, the statistics is cleared for the next time interval.

Generally, two kinds of statistics are maintained. The global statistics contains statistics about all objects and queries in the entire space. In order to tune the length of subtree time interval T as discussed in Section 7.2, the following statistics are required: 1) the total number of objects N, 2) maximum update time T_{up}, 3) three types of update cost CU_1, CU_2 and CU_m as defined in Table 7.1, 4) update/query ratio R, and 5) average query side length L and max_v for query enlarging. In addition, if the eager update technique is applied, the global statistics maintains one more field which is the number of objects N_m that are migrated at the last transition time.

Besides the global statistics, we use a $2d$ histogram to maintain regional statistics. The $2d$ histogram consists of $n \times n$ buckets, each of which maintains the statistics of a cell in the space. Specifically, the entire space is partitioned evenly into $n \times n$ square sized cells (different from the cells used for indexing in Section 4.2). In the histogram, the bucket of 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 and $\overrightarrow{x_{ij}}$ is the centroid of objects in the cell. The statistics maintained by the histogram summarizes the distribution in difference regions (cells) all over the space, which is necessary for the tuning purpose.

Histogram Maintenance:

Suppose the current time interval is $[T_l, T_u]$ and the reference time for the younger subtree is T_{ref}. The next transition time should be T_u, when the online tuning process starts. The tuning process aims to find the best space partitioning for the next time interval $[T_u, T_u + T]$. During that time, all objects will be indexed at a new reference time T'_{ref}. Therefore, objects are estimated and counted at the time instance of T'_{ref} in the histogram.

Specifically, given an update $o(\overrightarrow{x}, \overrightarrow{v})$ at t_{up}, the histogram is updated as follows:

(1) Estimate o's position at the time instance T'_{ref}:

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

Since the optimal length of the next time interval T is determined only when the tuning process completes, we simply assume that T will not change for the next time interval while maintaining the histogram. T'_{ref} can be determined accordingly, where $T'_{ref} = T_{ref} + T$. Then,

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

Note that if the update affects only the youngest subtree, the old record is deleted from the youngest subtree. The statistic should be updated in a reverse manner first, i.e., re-computing the centroid by subtracting old position of the object and decreasing the number of objects.

(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 cell c_{ij} at T'_{ref}.

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

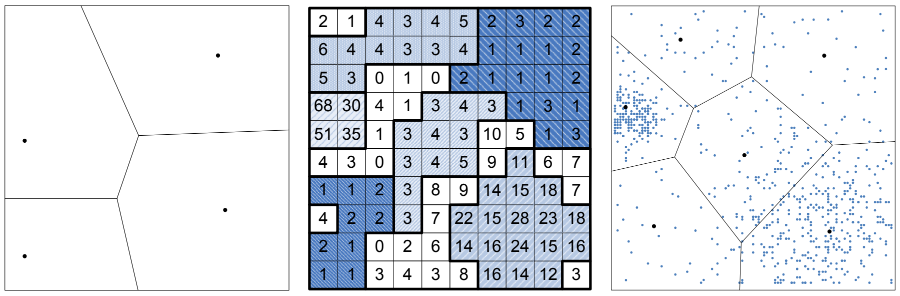
8.4 Online Tuning

The Online Tuning module is responsible for executing the tuning process. At each transition time, i.e., the upper boundary of current time interval, the Timer triggers the Online Tuning module to start the tuning procedure. Based on the statistics maintained, it determines new parameters for the ST^{2}B-tree. At the end of the tuning procedure, parameters in the Index Profile are updated accordingly.

As shown in Figure 8.1, the tuning can be applied in two aspects. On the one hand, we can tune the length of the subtree time interval and the reference time. On the other hand, we can improve the space partitioning by adjusting the set of reference points and their grid granularity.

8.4.1 *T and T_{ref}*

As shown in Section 7.2, the value of T affects update and query performance in opposite ways. In order to get the best tradeoff, we can also tune the value of T with respect to the latest query and update loads. With the global statistics maintained by the Statistics Module, the Online Tuning module can estimate the average query and update costs. In order to improve query performance, the system can tune down the value of T; in order to minimize the average update cost, the system can tune up the value of T. The final



(a) Old Reference Points (b) Region Growing (c) New Reference Points

Fig. 8.2. Finding Reference Points

decision is made depending on the main concern of the system, either query response time or supporting more updates/objects.

Given that T_{up} is maintained as a global statistics in the Statistics module, once T is determined, we can get the T_{ref} that minimizes average query cost according to Equation 7.7.

8.4.2 *Reference Points and Grid Granularity*

The ST^{2}B-tree can dynamically adjust to different space partitioning. However, 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, so that the tuning procedure can be done online without deferring any other operations.

Finding Reference Points via Region Growing:

The tuning procedure is triggered by the timer at each transition time. With the histogram, e.g., Figure 8.2(b), we identify dense and sparse regions by region growing. Recall that the Statistic module maintains a histogram with $n \times n$ buckets, each of which stores information about a cell in the space. Note here these cells of the histogram are detached from the cells which are used by the reference points for the purpose of indexing. The cells of the histogram are over all the space simply to have an idea of density distribution.

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 8.1 shows the procedure of region growing.

First, we take the previous reference points as the seeds for growing[3]. Since the distribution and density of moving objects change gradually, the positions of the reference points should move slightly. Starting from cell c, we examine its neighboring cells. If a neighboring cell c' does not belong to any existing region and $\frac{|c.n - R.max_n|}{R.avg_n} \leqslant \epsilon$ and $\frac{|c.n - R.min_n|}{R.avg_n} \leqslant \epsilon$, c' is added into the region R of c. $R.max_n$, $R.min_n$, $R.avg_n$ are the maximum, minimum and average number of objects of cells in R currently. ϵ is a prede-

[3]At the very beginning, reference points are obtained from historical data. In the absence of historical data, reference points are randomly picked during the initialization and will quickly fit the data after a few rounds of tuning.

Algorithm 8.1 Region Growing

```
Output: a set of regions $\{R\}$
 1: $RS = \emptyset$;
 2: for each previous reference points $RP_k$ do
 3:    $c$ = the cell that contains $RP_k$
 4:    if $c$ is unmarked then
 5:       Add $c$ to a new region $R$; mark $c$;
 6:       Growing($c$, $R$); Add $R$ to $RS$;
 7: while there is $c$ that is unmarked do
 8:    Add $c$ to a new region $R$; mark $c$;
 9:    Growing($c$, $R$); Add $R$ to $RS$
10: return $RS$;
Function Growing($c$, $R$)
 1: for each neighbor cell $c'$ of $c$ do
 2:    if $c'$ is unmarked and $\frac{|c.n-R.max_n|}{R.avg_n} \leqslant \epsilon$ and $\frac{|c.n-R.min_n|}{R.avg_n} \leqslant \epsilon$ then
 3:       Add $c'$ to $R$; mark $c$;
 4:       Growing($c'$, $R$);
```

fined threshold that defines similarity. The growing procedure terminates when all the cells belong to some region.

The output of the region growing algorithm is a set of regions that have similar object density. 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.

Figure 8.2 shows a running example of finding reference points by Algorithm 8.1. Region-growing starts with four previous reference points as shown in Figure 8.2(a). Figure 8.2(b) shows the resultant regions ($\epsilon = 1$) filled with different patterns and enclosed by thick lines. Then all regions that contain no more than 3 cells are pruned. Finally, we get 6 regions (shaded regions). As shown in Figure 8.2(c), the region growing method roughly identifies 6 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. When the reference points are determined, the grid granularity for each RP_i can be computed according to Equation 6.2.

Alternative Methods:

Intuitively, we can also apply density-based clustering methods to partition the space. Examples of density based spatial clustering methods include DBSCAN [Ester et al. 1996] and OPTICS [Ankerst et al. 1999]. However, none of these existing methods facilitates 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 toward 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. However, this method is only acceptable for a fairly stable environment. The preset settings may not be suitable once the environments changes for a non-trivial amount of time. For instance, road construction may last long enough to disrupt the system's responsiveness but not long enough to warrant changes to the indexes. It is a better choice if the tuning process can select the reference points and other parameters based on the latest workload at real-time and incurring imperceptible overhead.

8.4.3 *Degree of Eagerness* $\mathcal{D}_e$

Last but not least, if eager update is employed, we can also vary the degree of eagerness $\mathcal{D}_e$ according to the number of objects being migrated N_m last time. If $N_m \leq \xi$, we may increase $\mathcal{D}_e$ by one; otherwise decrease $\mathcal{D}_e$. Here, ξ is a user defined threshold, which depends on the latency that the system can afford to wait for object migration.

9. PERFORMANCE EVALUATION

9.1 Experiment Setup

In order to evaluate the ST^{2}B-tree and the self-tuning framework we used the most representative moving object indexes: the B^x-tree [Jensen et al. 2004], the B^{dual}-tree [Yiu et al. 2008], the TPR*-tree [Tao et al. 2003] and STRIPES [Patel et al. 2004], which are also based on different underlying structures. We use the implementations of the four indexes provided in [Chen et al. 2008b]. The ST^{2}B-tree is built on top of the same B$^+$-tree used by the B^x-tree and the B^{dual}-tree for fairness. The ST^{2}B-tree adopts the same optimal query enlargement algorithm [Jensen et al. 2006] as the B^x-tree.

[Chen et al. 2008b] provides a benchmark for evaluating and comparing the performance of moving objects. Our experiments follow the standard evaluation procedure introduced by that benchmark. Using the data generator included in the benchmark, we generate two kinds of workload, uniform and Gaussian datasets. Generally, the uniform datasets are used for evaluating the benefits of tuning time-related parameters, the eagerness of eager update strategy, etc. Since we intend to investigate the imbalance and changes in the workloads, we use Gaussian workloads to test the impact on the density-based space

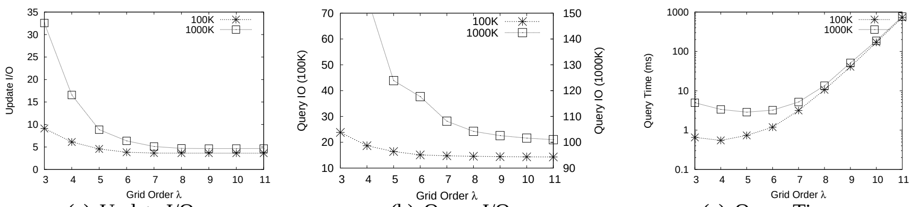
Table 9.1. Workload Settings

Parameter	**Setting**
Space domain	**100,000×100,000m^2**
Data size	**100K**, ..., 1M
Maximum object speed	10m/ts, ..., **100m/ts**
Maximum update interval T_{up}	**120ts**
Range query size	**1,000×1,000m^2**, ...,10,000×10,000m^2
Number of neighbors, k	**10**, ..., 100
Query Predictive Time	0ts, 10ts, ..., **60ts**, ..., 120ts
Time duration	**240ts**, 1200ts
Buffer size (number of pages)	**50**
Disk page size (KB)	**4**
Number of hotspots	**10**
Query/Update ratio	100:1, ..., **1:100**, ..., 1:10,000
Number of threads	1, 2, 4, ... , 128, **10**
Length of subtree time interval T	0.1T_{up}, 0.2T_{up}, ..., 0.9T_{up}, T_{up}
Offset of reference time T_{ref}-T_l	0, 0.2T, 0.4T, ..., T, 1.2T, ..., 2T
Degree of eagerness $\mathcal{D}_e$	**0**, 1, 2, 4, 8, 32, 64

partitioning scheme of the ST^{2}B-tree. Specifically, the data generator randomly selects some points in the space as hotspots. Each hotspot uses a Gaussian distribution to generate objects around it. The queries can be either uniformly distributed or not; in the case of non-uniform queries they follow the same distribution as the objects. The details about how the workloads are generated can be found in [Chen et al. 2008b].

Table 9.1 summarizes the settings of workload used in the experiments, where default values of variable parameters are shown in bold. ts is short for timestamp. Parameters are shown in groups based on their meanings. The experiments use the same range and default settings as the benchmark. In order to evaluate the tuning effect, we vary the tunable parameters such as T, T_{ref}, $\mathcal{D}_e$. Parameter ϵ for region growing is fixed at 1 in all the experiments. The execution time for region growing increases with the number of cells in the 2d-histogram. In our experiments, we use 100×100 cells in the 2d-histogram to keep the tuning time to be lower than 5ms, so that the system will not be stalled by the tuning process.

All the indexes are implemented in C++. All experiments are conducted on a IBM ThinkPad with Pentium M 1.86GHz processor, 1.0GB RAM and 60G SATA Disk, running Windows XP. All the results are the average for 10 runs. As the authors in [Jensen et al. 2004; Tao et al. 2003; Tao and Xiao 2008; Chen et al. 2008a] do in the experiments, we also use a block file with 4KB blocks to simulate a disk with 4KB pages. A LRU buffer with 50 pages is used. We use a tool, $Lavalys^{®}$ EVEREST, to test the random I/O speed on disk and memory respectively. The speeds of read and write of 4KB clusters on disk are ∼20MB/s and ∼12MB/s respectively. The speeds of memory reads and writes are ∼3356MB/s and ∼2770MB/s. The simulating experiments count the exact I/Os with our experimental settings. However, since we cannot prohibit the operating system from using the physical memory, the results actually shows the time for memory I/Os. Since all the indexes are implemented with the same disk manager, we did a fair lateral comparison between all indexes by simulation.



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

Fig. 9.1. Grid Granularity

9.2 Tunable Parameters

Before comparing with other moving object indexes, we first investigate the effect of the tunable parameters discussed in Sections 6 and 7. In this set of experiments, we use uniform datasets to get a clearer understanding of the effect of these parameters.

Effect of Grid Granularity:

We first study the effect of grid granularity empirically to verify the analysis in Section 6 and to determine the optimal grid order. We test on two uniform workloads, including 100K and 1M moving objects. 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 100 uniform range queries with default settings.

Figure 9.1 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 $\lambda \leq \frac{1}{2}(log_2 N - log_2 C_L)$ (about 7 for 1M objects) and hardly changes with finer partitioning.

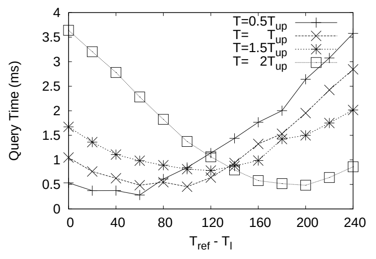
(2) When $\lambda \leq \frac{1}{2}(log_2 N - log_2 C_L)$, the query I/O increases with smaller value of λ. However, with larger λ, the number of average query I/O hardly changes. In Figure 9.1(b), the query I/O of the 1M dataset is labeled with the right y-axis with different values, because we do not 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.

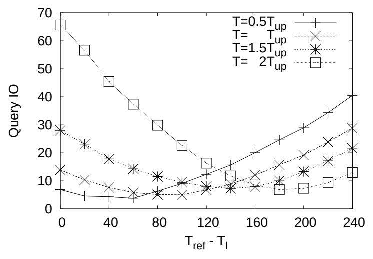
Based on the observation in Figure 9.1 and the analysis in Section 6, we set the grid granularity λ to $\lceil \frac{1}{2}(log_2 N - log_2 C_L) \rceil$ as in Equation 6.2, which results in the best tradeoff between update and query performance. In the following experiments, this rule is applied to the selection of global space partitioning for the B^x-tree, while for the ST^{2}B-tree, it guides the selection of grid granularity of each reference point.

Effect of Reference Time:

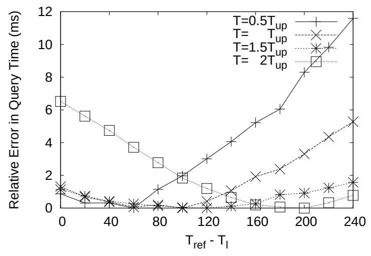
We now see the effect of the reference time. As discussed in Section 7.1, the reference time only affects the query performance. Therefore, we investigate the query time and I/O with respect to different choices of reference time. We vary the value of $T_{ref} - T_l$ from 0



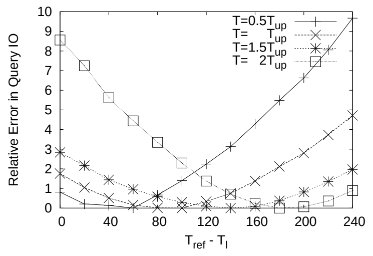
(a) Query Time (ms)



(b) Query I/O

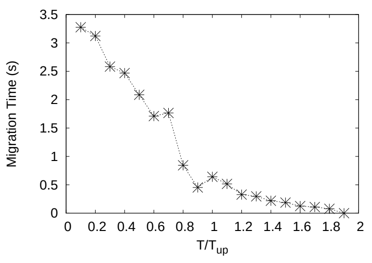


(c) Relative Error in Query Time (ms)

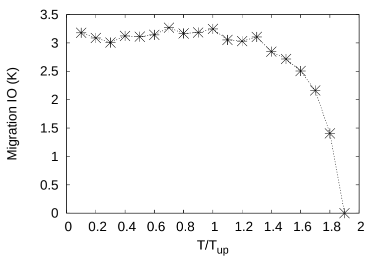


(d) Relative Error in Query I/O

Fig. 9.2. Reference Time-relative



(a) Migration Time



(b) Migration I/O

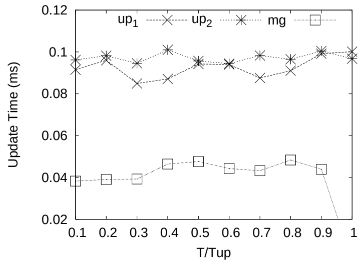
Fig. 9.3. T vs. Migration Cost

to $240ts$ ($2T_{up}$), i.e., the offset from the reference time to the lower boundary of the time interval of the subtree. The maximum update time T_{up} is fixed at the default value of 120ts. Figure 9.2 shows the average cost when the length of the subtree time interval T is set to $0.5T_{up}$, T_{up}, $1.5T_{up}$ and $2T_{up}$.

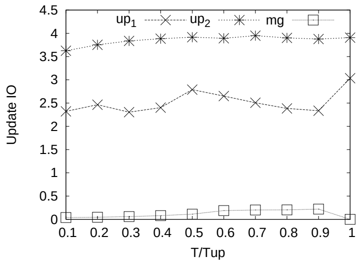
As shown, when $T = 0.5T_{up}$, the query cost is minimized when $T_{ref} - T_l = 60$, which be equal to T. When $T \geq T_{up}$, i.e., $T = T_{up}, 1.5T_{up}$ and $2T_{up}$, the query cost is minimized when $T_{ref} - T_l = 120ts, 140ts, 200ts$ respectively. The results in Figure 9.2 verify the analysis in Section 7.1. The optimal reference time is selected as Equation 7.7. While T_{ref} deviates from the optimal value, the query cost increases monotonically.

Effect of the Length of Subtree Time Interval T:

In this experiment, we investigate the problem caused by object migration during rollover. The length of the subtree time interval T varies from $0.1T_{up}$ to T_{up} (if T is larger than T_{up}, no migration occurs). In order to see the pure effect of object migration, eager update is disabled.

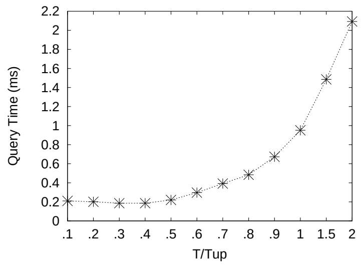


(a) Update Time (ms)

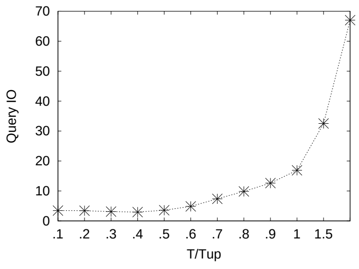


(b) Update I/O

Fig. 9.4. T vs. Update Cost



(a) Query Time (ms)



(b) Query I/O

Fig. 9.5. T vs. Update Cost

Figure 9.3 shows the total running time and number of I/Os of one migration. As expected, the total migration time is proportional to the number of un-updated objects left in the old subtree, which decreases with larger T. When T is only 10% of the maximum update time T_{up}, there is a delay of more than 3s to finish the migration. If T increases to about $0.9T_{up}$, the migration time is reduced to around 0.5s. Note that the migration I/Os is less affected when T is smaller than $0.9T_{up}$. This is because migration is done by deleting objects from the older subtree and inserting them into the younger subtree in a batch mode, which saves a number of I/Os, especially with the help of the LRU buffer.

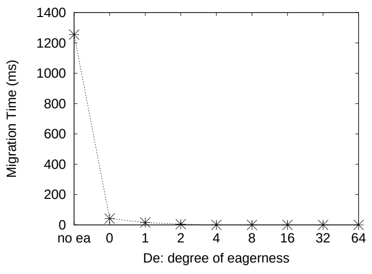
Figure 9.4 illustrates the corresponding costs for different types of updates, where "up$_1$" represents updates that involve only one subtree, "up$_2$" represents updates that affect both subtrees and "mg" denotes the amortized cost of migrating one object. The amortized migration cost is the minimum, i.e., around 0.04ms and no more than 0.1 I/O access. A single-subtree update (up$_1$) incurs about 2/3 I/Os of an update involving both subtrees.

Although the total migration time seems to be not very long (only a few seconds), it is still not acceptable in a continuous running MOD. As shown in Figure 9.4, the average update time is only about 0.1 millisecond. During the migration time, the MOD is capable of handling tens of thousands of updates.

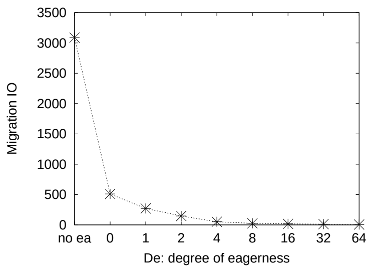
Figure 9.5 shows the effect of T on query performance. As expected, the query cost, including the CPU time and the number of I/O accesses increase with T.

9.3 Effect of Eager Updates

Now we proceed to investigate the effect of eager updates. T is set to be half of the maximum update time, i.e., $0.5T_{up}$. In this set of experiments, we vary the degrees of eagerness $\mathcal{D}_e$ from 0 to 64. The corresponding benefits on migration cost and expense

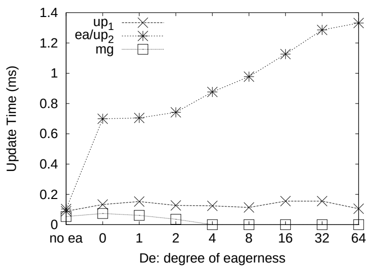


(a) Migration Time (ms)

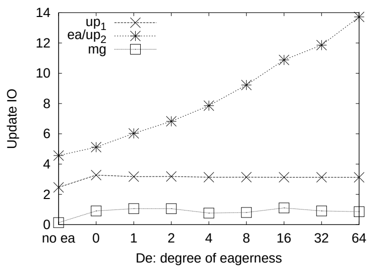


(b) Migration I/O

Fig. 9.6. Degree of Eagerness vs. Migration Cost



(a) Update Time (ms)



(b) Update I/O

Fig. 9.7. Degree of Eagerness vs. Update Cost

on update cost are shown in Figure 9.6 and Figure 9.7 respectively. The leftmost point (marked as "no ea") in both Figures represents the corresponding result of the case where eager update is disabled.

First Figure 9.6 shows the benefits of eager updates in terms of total time and I/O cost of an migration. If there are no eager updates, it requires about 1.3 seconds for migrating objects during rollover. However, when eager update is enabled, the migration cost decreases dramatically. Specifically, when $\mathcal{D}_e = 0$, which means the smallest degree of eagerness, the migration time is reduced to about 50ms. When $\mathcal{D}_e = 4$, the migration time is around 5ms only, which is much more acceptable considering the continuity of a MOD.

Eager update technique undoubtedly relaxes the constraints on T by reducing the migration cost, but at the expense of an increase in update cost. Figure 9.7 shows the average update cost. As shown, the migrating cost ("mg") is still the smallest. The cost of single tree update ("up_1") is more or less the same as that in Figure 9.4, since eager update is only applicable to updates involving two subtrees as described in Algorithm 5.1. There is a significant increase on the average cost of updates ("up_2") when eager update is applied. When $\mathcal{D}_e = 0$, the update is about 7 times slower than normal update (the one marked with "no mig"), while the increase in the number of I/Os is no more than 1. Intuitively, when $\mathcal{D}_e = 0$, the update cost should not increase since the deletion and insertion access the same tree nodes as a normal update. In practice, by deleting and inserting more records from a leaf may cause leaf merge or split, which incurs additional I/Os. With larger $\mathcal{D}_e$, the I/O cost increases more significantly, since the insertions may happen in leaf nodes other than the leaf node where the core object, i.e., the object causes the eager update, is inserted.

Figure 9.8 shows the number of updates of different types. With higher degree of eagerness, i.e., larger $\mathcal{D}_e$, not only the number of objects to be migrated decreases, but also the

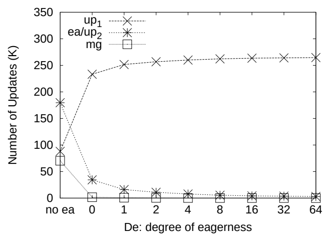


Fig. 9.8. Number of Updates

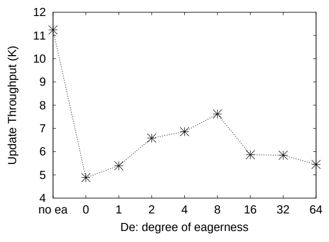


Fig. 9.9. (Active)Update Throughput

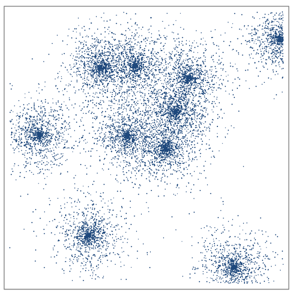


Fig. 9.10. Distribution of Default Gaussian Workload (10 hotspots)

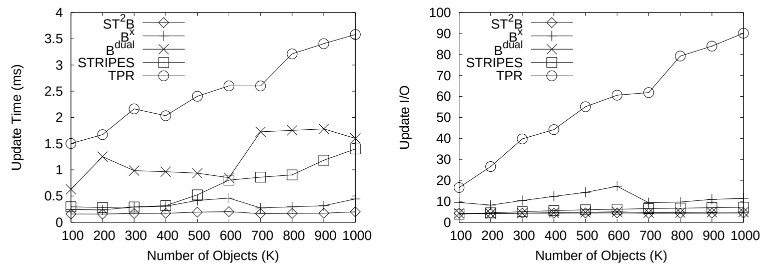
number of updates involving both subtrees. Through eager updating, a number of updates are avoided and reduced into single subtree updates. Therefore, as we can see, the number of single tree updates (“up_1”) increases with $\mathcal{D}_e$.

Finally, Figure 9.9 shows the effect of $\mathcal{D}_e$ on the total update throughput, i.e., the total number of active updates processed in unit time. Updates caused by migration are not counted here since they are additional workload introduced by the design constraints of the system. When there is no eager update, the throughput is quite high since the update cost is the lowest as shown in Figure 9.7. With eager updates, although the throughput becomes much smaller, the system will not be paused for object migration. From Figure 9.8, we can observe that the percentage of single tree updates increases with larger $\mathcal{D}_e$. Since single tree updates are less costly, the total update throughput increases when $\mathcal{D}_e$ increases. When $\mathcal{D}_e$ is larger than 8, the throughput starts to drop. This is because when $\mathcal{D}_e$ is large enough, the percentage of single tree updates does not increase substantially, while the cost of eager updates does increase.

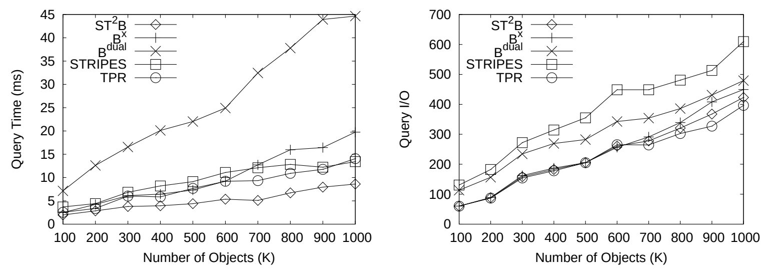
In summary, regarding the migration cost, the benefit of eager updating is already significant when $\mathcal{D}_e$ is as small as 0 or 1. With larger $\mathcal{D}_e$, the further improvement is minor. Considering the migration time and the update cost, we believe that setting $\mathcal{D}_e$ to 0 or 1 is sufficient enough for the tuning purpose.

9.4 Spatial Diversity

We now investigate the effectiveness of the ST^{2}B-tree with regard to the spatial diversity of moving objects. We use a Gaussian workload generated with 10 randomly selected hotspots as default. Figure 9.10 shows a sample of the workload used (some hotspots are close to others and cannot be clearly seen). In order to examine the effect of data skew only, we keep the distribution and cardinality of workload unchanged with time in this set of experiments. For the same purpose, eager updating is disabled and T is fixed to T_{up} so



(a) Update Cost



(b) Query Cost

Fig. 9.11. Object Cardinality

as to guarantee that there is no object migration. The indexes run up to $2T_{up}$ (240ts). The query workload consists of 100 queries. The queries are evaluated every 12 timestamps and the average costs are record.

Scalability Test:

First, Figure 9.11 shows the effect of the data size on query performance. The number of objects varies from 100K to 1M, with an increment of 100K. The results of all the indexes (except the ST^{2}B-tree) conform to the findings in [Chen et al. 2008b]. Specifically, as shown in Figure 9.11(a), the TPR*-tree incurs a high update cost in both I/O and time. This is because of the overlap between MBRs that results in the TPR*-tree having to search multiple paths in an update. The MBR adjustment during an update operation further degrades the update time. The update time of the TPR*-tree is about 7 and 14 times higher than that of the ST^{2}B-tree with 100K and 1M objects respectively. The B^x-tree has fast update, but the number of update I/Os is higher than the others except the TPR*-tree. The number of update I/Os of the B^x-tree is affected by the granularity of space partitioning with regard to the data distribution. Since the workload is skewed, a uniform grid leads to some densely populated cells which break the balance of the B$^+$-tree by introducing many overflow pages. Consequently, the update I/O cost increases. Owing to the data-adaptive space partitioning, the ST^{2}B-tree has both fairly constant update time, i.e., about 0.2ms, and number of I/Os, which is around 4. STRIPES and the B^{dual}-tree, although having smaller number of update I/Os than the B^x-tree, take longer time to process a query.

Figure 9.11(b) shows the average query cost with respect to the number of objects. As expected, for all indexes, the query cost increases linearly with the data size. This is because more objects need to be retrieved in a given query region for a larger dataset. The ST^{2}B-tree, the B^x-tree and the TPR*-tree incur similar number of I/Os. The cost of the ST^{2}B-tree and the B^x-tree are mainly determined by the number of objects contained in

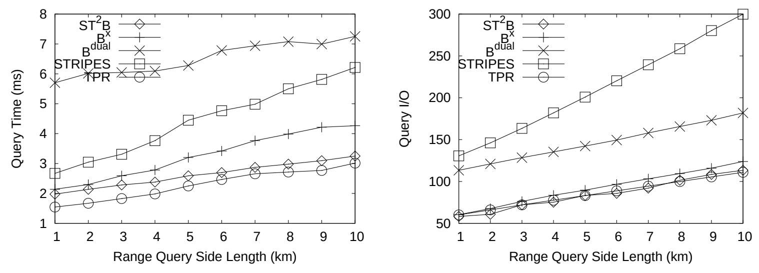


Fig. 9.12. Range Query Side Length

the enlarged query region, while the TPR*-tree has been specifically designed to reduce its I/O cost over the original TPR-tree. The B^{dual}-tree incurs a larger number of I/Os. This is partially because the partitioning in the velocity dimensions makes some nearby objects of different velocities distributed into different leaf nodes, while in the ST^{2}B-tree and B^x-tree, nearby objects are clustered together. Among all the indexes, STRIPES is the most expensive regarding the I/O cost due to the unbalancing structure and low utilization space of the quad-tree.

Regarding the query processing time, the B^x-tree is not able to handle hotspots well due to its use of one single grid granularity, and causes many false positives to be retrieved and examined, and therefore incurs higher I/Os. The ST^{2}B-tree is most efficient in terms of query time due to its adaptive use of appropriate grid granularity when it rolls forward with time.

In Figure 9.11, the B^x-tree consistently outperforms the B^{dual}-tree. This is somewhat surprising, since it does not conform to the findings in [Yiu et al. 2008], where the B^{dual}-tree is expected to beat the B$^+$-tree in terms of the number of I/Os. We find the following reasons for this inconsistency. In [Yiu et al. 2008], the disk page is only 1K bytes and no buffer is used. Query processing of the B^x-tree incurs larger number of I/Os, since it has to revisit upper-level tree nodes several times. However, in our experiments, a small LRU buffer is used and intermediate nodes are kept in memory most of the time. The buffering effect brings a considerable decrease on the number of querying processing I/Os. As for the query processing time, [Yiu et al. 2008] does not provide any result on the query processing time of the B^{dual}-tree. In our experiments, we find that the decomposition and overlapping testing of the MORs in the B^{dual}-tree are both time-consuming tasks. Therefore, the query processing time of the B^{dual}-tree is consistently higher than all the other indexes.

Size of Query:

Figure 9.12 shows the average cost of processing range queries. All the indexes are tested with square-sized range queries with side length varying from 1km to 10km. As expected, the query cost of each index increases with an increasing query window size. Larger windows contain more objects and therefore lead to more node accesses. In general, Figure 9.12 shows the same relative order between the curves as in Figure 9.11(b). The TPR*-tree, the B^x-tree and the ST^{2}B-tree have the similar number of I/Os. With the default 100K Gaussian workload, the B^x-tree spends twice the query processing time of the TPR*-tree. As discussed in Section 2, the TPR*-tree is a data partitioning index, which is less affected by the data distribution comparing with the B^x-tree. The ST^{2}B-tree shortens the processing time of the B^x-tree by about 30% for queries with 10km side length. The ST^{2}B-tree improves the B^x-tree with more adaptive space partitioning. As a result, the effect of

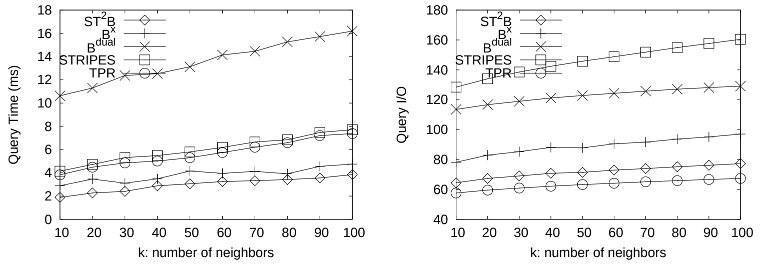


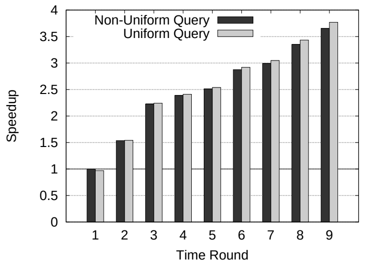
Fig. 9.13. kNN Query k

data skew is mitigated. The B^{dual}-tree takes the longest processing time and incurs twice number of I/Os than the TPR*-tree, the B^x-tree and the ST^{2}B-tree. As for STRIPES, the processing time increases the fastest among all indexes because of the significant increase in the number of I/Os.

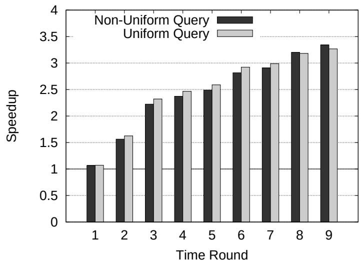
Figure 9.13 examines the performance of kNN queries further, with the number of neighbors k varying from 10 to 100. As for kNN queries, all indexes have a slight increase in I/O cost and query processing time. The I/O cost of the kNN queries, which depends on the number of objects in the expanded query region, is less sensitive to k for the B^x-tree and the ST^{2}B-tree. In terms of the number of I/Os, the ST^{2}B-tree surpasses the B^x-tree by a greater margin for kNN queries than for range queries. 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 cost depends largely on the accuracy of the estimated search radius. The B^x-tree makes such estimation using global object density. As a result, in dense 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 because of its branch-and-bound kNN search algorithm. The kNN query performance of STRIPES and the B^{dual}-tree are similar to their performance on range queries, contributing the highest number of I/Os and the highest processing time respectively.

9.5 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 used in the previous experiments (Figure 9.10). 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 the overall performance. Then we



(a) Range Query



(b) kNN Query

Fig. 9.14. Increasing Data Cardinality

compute the speedup introduced by the self-tuning feature of the ST^{2}B-tree, defined as:

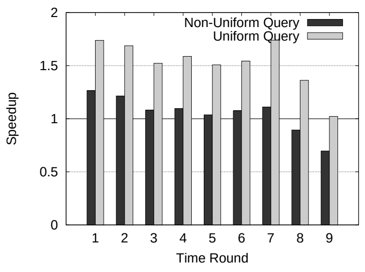
$$\text{Speedup} = \frac{\text{total processing time of the B}^x\text{-tree}}{\text{total processing time of the ST}^2\text{B-tree}}.$$

The granularity of the B^x-tree is selected using the initial number of objects, while the space partitioning of the ST^{2}B-tree is dynamically tuned in accordance to the workload. 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.

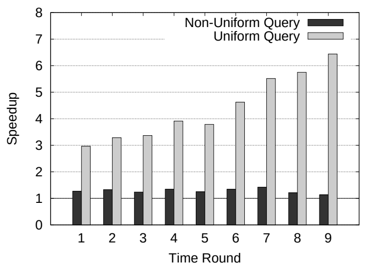
First, we start with 100K objects and add another 100K each round. Figure 9.14 shows the speedup brought by self-tuning in each round of time. The speedup introduced by self-tuning grows with time, when the hotspots and data distribution change with time.

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$3000m^2). With the increasing number of objects in the following rounds, the update performance degrades, because the increase in the number of overflow pages affects the balance of the underlying B$^+$-tree. On the other hand, since the 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 9.2. 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 speedup introduced by the self-tuning of the ST^{2}B-tree over static B^x-tree, especially when there are more updates than queries, are obvious and significant.

Figure 9.15 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$\times$200m^2) to partition the entire space. As we can see, the speedup introduced by the self-tuning to the system 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 queries are concentrated at those dense regions. Now, in those dense regions, the ST^{2}B-tree also employs fine grid. Therefore, the system speedup introduced by tuning is less significant. However,

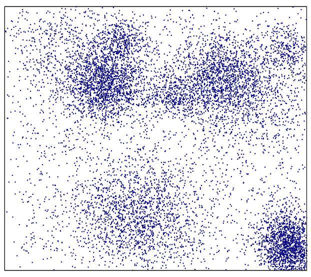


(a) Range Query



(b) kNN Query

Fig. 9.15. Decreasing Data Cardinality


(a) Round 5

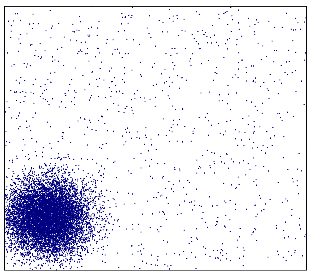
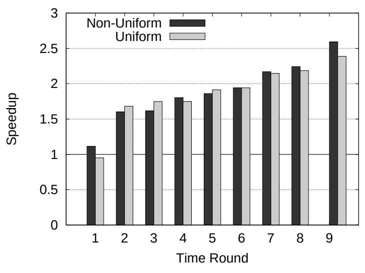
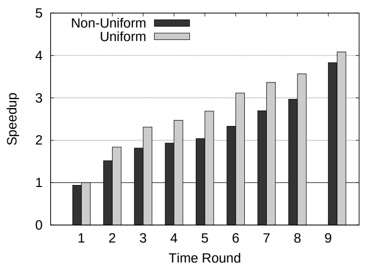

(b) Round 9

Fig. 9.16. Distribution of Workload in Spatio-Temporal Test



(a) Range Query



(b) kNN Query

Fig. 9.17. Change of Data Distribution with Time

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.

9.6 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 increases 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 hotspot. Figure 9.16 shows the snapshots of objects at round 5 (moderately skewed) and round 9 (highly skewed). The query-update ratio is still 1:100.

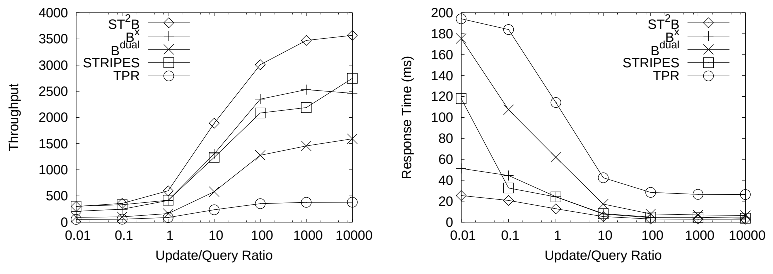
Figure 9.17 shows the changes of system speedup with time. As expected, the gain of the self-tuning ST^{2}B-tree over the static B^x-tree increases when the data become even more skewed with time. During round 1, the ST^{2}B-tree is comparable to the B^x-tree. Since the objects are uniformly distributed, 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 9.17(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 9.17(b)), the performance gain is much higher, which is about 4 times.

9.7 Throughput Test

Finally, we evaluate all indexes in a multiple-user environment. We use a multi-thread program to simulate the real multiple-user environment. The B$^+$-tree adopts the B-link concurrency control mechanism as presented in [Lehman and Yao 1981]. The R-tree (for TPR*-tree) employs the R-link technique [Ng and Kameda 1994]. We implement a native concurrency control of the quad-tree for STRIPES. Specifically, a search operation holds a read lock on each node on its current searching path while an insertion/deletion holds a write lock on the current node. The write lock is released when locks on its children are granted and split/merge will not happen to the current node after the insertion/deletion. The default 1M dataset as shown in Figure 9.10 is used. The query workload consists of range queries with default settings, following the same distribution of the data objects. Updates and queries arrive evenly in time and are loaded into a task pool and then randomly distributed to each free working thread. The performance is measured by two metrics: throughput and response time. The throughput is defined as the average number of tasks finished in a unit time. The results are the average of 10 runs of simulation.

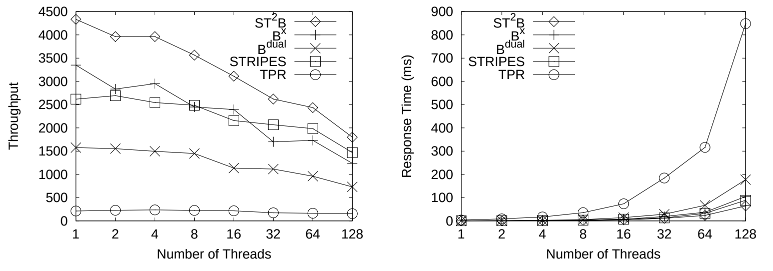
Figure 9.18 shows the throughput and response time with the query-update ratio varying from 100:1 to 1:1000 using 10 working threads. In real moving object applications, the update load caused by the changes in object locations and moving speed is much higher than the query load, and the query-update ratio is to simulate such scenario. As expected, the throughput of the indexes increases 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 9.19 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. The throughput reduces with increasing number of threads for all indexes, and 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. As the workload includes more updates than queries, the indexes are frequently being write-locked. The throughput decreases with more threads and each thread waits for a longer time for its turn



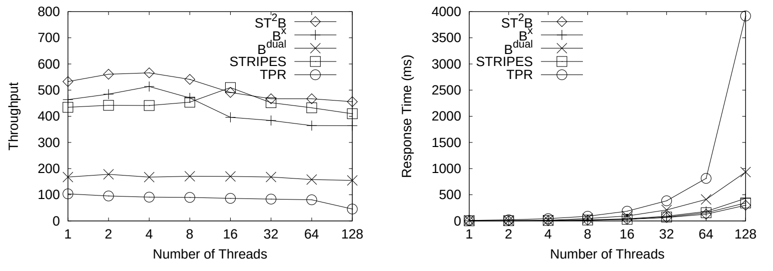
(a) Throughput (b) Response Time

Fig. 9.18. Query-Update Ratio



(a) Throughput (b) Response Time

Fig. 9.19. Number of Threads: Query-Update Ratio=1:100



(a) Throughput (b) Response Time

Fig. 9.20. Number of Threads: Query-Update Ratio=1:1

to access the tree.

However, with more queries, the throughput first increases and then reduces with increasing number of threads. As shown in Figure 9.20, when the workload consists of 50% queries and 50% updates, the throughput reaches the peak with about 2 threads or 4 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, so we omit the results for such workload composition.

As can be observed from Figures 9.19 and 9.20, the indexes hit thrashing point after the number of threads increases to a 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, which has not been taken into account here.

10. CONCLUSION

In this paper, we have re-examined the problem of indexing moving object databases (MODs). We identified several forms of data diversity (namely, space, time, and spatial-temporal diversities) in MODs that existing static indexes are unable to handle effectively. We then proposed the ST^{2}B-tree, an online self-tunable B$^+$-tree index that continuously adapt to the changes in object locations and distributions. To adapt to space diversity, 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 object density around a reference point. By monitoring the distribution and density of objects continuously, the ST^{2}B-tree dynamically determines a different set of reference points, and adaptively adjusts the granularity of space partitioning. We also proposed methods to choose the reference points, and provide a guideline on the optimal choice of granularity. To deal with the time diversity, the ST^{2}B-Tree employs a "multi-tree" approach, where two subtrees are used to index objects regarding their last update time. The basic idea is to rebuild the subtrees periodically and alternately, during which the newly identified reference points and granularity are used. We have also proposed a novel eager update mechanism to facilitate object migration from one subtree to another. We have conducted an extensive performance evaluation of the ST^{2}B-Tree against several state-of-the-art techniques. The experimental results showed the superiority of the ST^{2}B-Tree over these methods, confirming 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.

ACKNOWLEDGMENTS

Su Chen and Beng Chin Ooi were supported by Singapore NRF Grant R-252-000-376-279. Mario A. Nascimento was supported by a NSERC Canada Discovery Grant.

REFERENCES

ANKERST, M., BREUNIG, M. M., KRIEGEL, H.-P., AND SANDER, J. 1999. Optics: Ordering points to identify the clustering structure. In *Proc. ACM SIGMOD*. 49–60.

BECKMANN, N., KRIEGEL, H. P., R.SCHNEIDER, AND B.SEEGER. 1990. The R*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD*. 322–331.

CHAUDHURI, S. AND NARASAYYA, V. R. 1997. An efficient cost-driven index selection tool for Microsoft SQL server. In *Proc. VLDB*. 146–155.

CHEN, N., SHOU, L.-D., CHEN, G., AND DONG, J.-X. 2008a. Adaptive indexing of moving objects with highly variable update frequencies. *JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 23,* 6 (Nov.), 998–1014.

CHEN, S., JENSEN, C. S., AND LIN, D. 2008b. A benchmark for evaluating moving objects indexes. In *Proc. VLDB*. 1574–1585.

CHEN, S., OOI, B. C., TAN, K.-L., AND NASCIMENTO, M. A. 2008. The ST^{2}B-tree: A self-tunable spatio-temporal B$^+$-tree index for moving objects. In *Proc. ACM SIGMOD*. 29–42.

ESTER, M., KRIEGEL, H.-P., SANDER, J., AND XU, X. 1996. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Proc. KDD*. 226–231.

GUO, S., HUANG, Z., JAGADISH, H. V., OOI, B. C., AND ZHANG, Z. 2006. Relaxed space bounding for moving objects: A case for the buddy tree. *SIGMOD Record 35,* 4, 24–29.

GUTTMAN, A. 1984. R-trees: A dynamic index structure for spatial searching. In *Proc. ACM SIGMOD*. 47–57.

JAGADISH, H. V., OOI, B. C., TAN, K.-L., YU, C., AND ZHANG, R. 2005. iDistance: An adaptive B^+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst 30,* 2, 364–397.

JENSEN, C. S., LIN, D., AND OOI, B. C. 2004. Query and update efficient B^+-tree based indexing of moving objects. In *Proc. VLDB*. 768–779.

JENSEN, C. S., TIESYTE, D., AND TRADISAUSKAS, N. 2006. Robust B^+-tree-based indexing of moving objects. In *Proc. MDM*. 12.

KALASHNIKOV, D. V., PRABHAKAR, S., AND HAMBRUSCH, S. E. 2004. Main memory evaluation of monitoring queries over moving objects. *Distrib. Parallel Databases 15,* 2, 117–135.

KOLLIOS, G., GUNOPULOS, D., AND TSOTRAS, V. 1999. On indexing mobile objects. In *Proc. ACM PODS*. 261–272.

KOLLIOS, G., PAPADOPOULOS, D., AND GUNOPULOS, V. 2005. Indexing mobile objects using dual transformations. *The VLDB Journal 14,* 2, 238–256.

KWON, D., LEE, S., AND LEE, S. 2002. Indexing the current positions of moving objects using the lazy update. In *Proc. MDM*. 113–120.

LEE, M. L., HSU, W., JENSEN, C. S., CUI, B., AND TEO, K. L. 2003. Supporting frequent updates in R-trees: A bottom-up approach. In *Proc. VLDB*. 608–619.

LEHMAN, P. L. AND YAO, S. B. 1981. Efficient locking for concurrent operations on B-trees. *ACM Trans. Database Syst. 6,* 4, 650–670.

MOKBEL, M. F., XIONG, X., AND AREF, W. G. 2004. Sina: Scalable incremental processing of continuous queries in spatio-temporal databases. In *Proc. ACM SIGMOD*. 623–634.

MOON, B., JAGADISH, H. V., FALOUTSOS, C., AND SALTZ, J. H. 2001. Analysis of the clustering properties of the hilbert space-filling curve. *TKDE 13,* 1, 124–141.

MOURATIDIS, K., PAPADIAS, D., AND HADJIELEFTHERIOU, M. 2005. Conceptual partitioning: An efficient method for continuous nearest neighbor monitoring. In *Proc. ACM SIGMOD*. 634–645.

NG, V. AND KAMEDA, T. 1994. The R-link tree: A recoverable index structure for spatial data. In *Proc. DEXA*. 163–172.

PATEL, J. M., CHEN, Y., AND CHAKKA, V. P. 2004. Stripes: An efficient index for predicted trajectories. In *Proc. ACM SIGMOD*. 637–646.

PROCOPIUC, C. M., AGARWAL, P. K., AND HAR-PELED, S. 2002. Star-tree: An efficient self-adjusting index for moving objects. In *ALENEX*. 178–193.

SALTENIS, S. AND JENSEN, C. S. 2002. Indexing of moving objects for location-based services. In *Proc. ICDE*. 463–472.

SALTENIS, S., S.JENSEN, C., LEUTENEGGER, S. T., AND LOPEZ, M. A. 2000. Indexing the positions of continuously moving objects. In *Proc. ACM SIGMOD*. 331–342.

TAO, Y., PAPADIAS, D., AND SUN, J. 2003. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *Proc. VLDB*. 790–801.

TAO, Y. AND XIAO, X. 2008. Primal or dual: which promises faster spatiotemporal search? *The VLDB Journal 17,* 5, 1253–1270.

TAO, Y., ZHANG, J., PAPADIAS, D., AND MAMOULIS, N. 2004. An efficient cost model for optimization of nearest neighbor search in low and medium dimensional spaces. *TKDE 16,* 10, 1169–1184.

XIONG, X. AND AREF, W. G. 2006. R-trees with update memos. In *Proc. ICDE*. 22.

XIONG, X., MOKBEL, M. F., AND AREF, W. G. 2005. SEA-CNN: Scalable processing of continuous k-nearest neighbor queries in spatio-temporal databases. In *Proc. ICDE*. 643–654.

YIU, M. L., TAO, Y., AND MAMOULIS, N. 2008. The B^{dual}-tree: indexing moving objects by space filling curves in the dual space. *The VLDB Journal 17,* 3, 379–400.

YU, C., OOI, B. C., TAN, K.-L., AND JAGADISH, H. V. 2001. Indexing the distance: an efficient method to knn processing. In *Proc. VLDB*. 421–430.

Received Month Year; revised Month Year; accepted Month Year