B^s-tree: A Self-tuning Index of Moving Objects

Nan Chen, Lidan Shou, Gang Chen, Ke Chen, and Yunjun Gao

College of Computer Science, Zhejiang University, China
{cnasd715,should,cg,chenk,gaoyj}@cs.zju.edu.cn

Abstract. Self-tuning database is a general paradigm for the future development of database systems. However, in moving object database, a vibrant and dynamic research area of the database community, the need for self-tuning has so far been overlooked. None of the existing spatio-temporal indexes can maintain high performance if the proportion of query and update operations varies significantly in the applications. We study the self-tuning indexing techniques which balance the query and update performances for optimal overall performance in moving object databases. In this paper, we propose a self-tuning framework which relies on a novel moving object index named B^s-tree. This framework is able to optimize its own overall performance by adapting to the workload online without interrupting the indexing service. We present various algorithms for the B^s-tree and the tuning techniques. Our extensive experiments show that the framework is effective, and the B^s-tree outperforms the existing indexes under different circumstances.

Keywords: spatio-temporal database, moving object, index, self-tuning.

1 Introduction

With the rapid progress in hardware and software technologies, the database systems and applications are becoming increasingly more complex and dynamic, however at lower costs than ever. As a result, the traditional solution of employing database administrators to maintain and fine-tune the DBMS for higher performance has appeared to be inefficient and uneconomical. The self-tuning functionality of databases is expected to be the substitute for DBAs. Intuitively, a self-tuning database provides practical solutions to adjust itself automatically for optimal performance with minimal human intervention. In the past years, a lot works (e.g., [3]) have been done to extend the self-tuning capabilities of database systems. However, the existing works so far are mainly restricted to traditional static databases.

1.1 Motivation

The recent emergence of numerous moving object database applications, such as traffic control, meteorology monitoring, mobile location computing and so on, has called on for self-tuning techniques in the moving object databases as well. In the numerous techniques proposed for managing moving object databases, index design appears to be the spotlight. Therefore, we shall look at the self-tuning techniques for moving object indexes. The problem that we consider for tuning is the update and query performance.

H. Kitagawa et al. (Eds.): DASFAA 2010, Part II, LNCS 5982, pp. 1–16, 2010.
© Springer-Verlag Berlin Heidelberg 2010

We observe that the update and query performances of moving object indexes display interesting patterns if we classify the existing indexes into two major categories regarding the data representations that they use [5]. The first class includes R/R*-tree-based indexes, such as the TPR-tree [9] and the TPR*-tree [11]. This class is characterized by good query performance but expensive update costs. The second class includes techniques that employ space partitioning and data/query transformations to index object positions, such as the B^x-tree [8]. As it is based on the B+-tree, the B^x-tree has good update performance. However, it does not achieve satisfactory query efficiency, as shown in [13]. In addition, the second class has better concurrency performance than the first, because, as shown in [8,6], the concurrency control in a R/R*-tree-based index is more complex and time-consuming than that in a B+-tree-based index.

The above observation leads to the implication that the current moving-object indexing techniques encounter difficulty when handling variable needs in the proportion of query and update operations. Given some examples of these applications, an aviation monitoring system may need more query operations, and oppositely make fewer updates to the database. While an animal position tracking system may require far more updates than queries. In addition, there are also applications which require variable ratio of updates and queries at different time. For example, in a traffic monitoring system of a city, the proportions of updates and queries may vary widely by time. It may receive increased updates during the rush hour, and process more queries at the report time.

To solve the problem with these dynamic applications, we need a self-tunable index structure which is able to strike a balance between the performance of queries and updates, thereby achieving good overall performance for different proportion of updates and queries. In addition, the self-tuning of the index should not interrupt the index service. Based on the discussion in the above, none of the existing moving object indexes can satisfy such requirements readily. They suffer from either poor query performance or large update costs, and does not have the ability of self-tuning. In fact, the performance of queries and that of updates usually affect each other, and are often anti-correlated in most of the existing indexes. Therefore, they are not appropriate for these environments. We believe this problem has so far been overlooked.

1.2 Overview of the Proposed Techniques

In this paper, we propose an online self-tuning framework for indexing moving-objects based on a novel data structure called B^s-tree. The B^s-tree provides the functionality of self-tuning, and can adaptively handle both queries and updates efficiently. Our framework uses an *update parameter* α and a *query parameter* β to tune the performance of the B^s-tree. Figure 1 shows the components of the proposed self-tuning framework. When an update arrives, the Key Generator Module (KGM) computes the index keys according to the update parameter α of the index, inserts them into the B^s-tree and reports the update cost to the Online Tuning Module (OTM). On the other hand, when a query arrives, the Query Executor Module (QEM) processes the query according to the query parameter β of the index and reports the query cost to OTM. OTM monitors the performance and controls the tuning.

The proposed B^s-tree has many desirable features. It can provide self-tuning for optimal overall performance without interrupting the indexing service. When the query

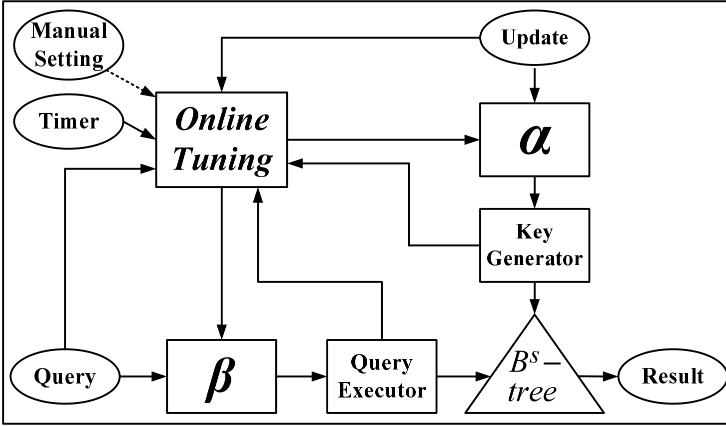

Fig. 1. Online Self-tuning Framework

cost dominates the overall performance, the B^s-tree achieves considerably higher performance of queries, at the expense of slightly more update costs. In contrast, if the update cost dominates the overall performance, the B^s-tree will sacrifice some query performance to obtain lower update costs. Our extensive experiments show that the B^s-tree outperforms the existing indexes in different environments. Another advantage of the B^s-tree is that it is based on the classical B+-tree. Therefore it can be easily implemented in the existing DBMSs and is expected to have good concurrency performance. Table 1 summarizes the properties of the B^s-tree compared to the previous indexes.

Table 1. Comparison of predictive indexes of moving objects.

	Query Cost	Update Cost	DBMS integration	Self-Tuning	Concurrency
B^x	HIGH	LOW	EASY	NO	HIGH
TPR*	LOW	HIGH	HARD	NO	LOW
B^s	LOW	LOW	EASY	YES	HIGH

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the structure of the B^s-tree, and presents the associated query and update operations. Section 4 analyzes the I/O cost of the B^s-tree, and introduces how the online self-tuning framework works. Section 5 presents the results of the experiments. Section 6 conludes the paper and discusses possible future work.

2 Related Work

Lots of index structures have been proposed to index the current and predicted future positions of moving objects. According to [5], these indexes can be classified into two major categories depending on the trees they are based on. The first category is the R/R*-tree-based indexes which base on the R-tree [7] and the R*-tree [1], such as the Time-Parameterized R-tree (TPR-tree) [9] and its variant, the TPR*-tree [11]. A node in the TPR-tree is represented as a MOving Rectangle (MOR), including a Minimum

Bounding Rectangle (MBR) and a velocity Bounding Rectangle (VBR). The extent of a MBR grows according to its VBR and it never shrinks, although it is generally larger than strictly needed. This guarantees that every MBR always bounds the objects that belong to it at all times considered. The update algorithm of the TPR-tree is exactly the same as those of the R*-tree, by simply replacing the four penalty metrics of the R*-tree with their integral counterparts. Based on the same structure, the TPR*-tree provides a new set of insertion and deletion algorithms aiming at minimizing a certain cost function. In the TPR-tree and the TPR*-tree, predictive query for a future time t is processed by visiting MBRs expanded to t. The TPR-tree and the TPR*-tree are deployed to solve a large number of spatio-temporal query problems (e.g., [2] for KNN queries). However, they have a major weakness: the update costs are expensive.

The second category of indexes relies on space partitioning and data/query transformations, such as the B+-tree based indexes [8] [5] [13]. The B^x-tree [8] is the first effort to adapt the B+-tree to index moving objects. It partitions the time axis into equal intervals, called phases, and partitions the data space into uniform cells. An index partition of the B+-tree is reserved for each phase. When an insertion operation of a moving object arrives, the B^x-tree computes the position of this object at the end timestamp of the next phase, and transforms this position to a value of the space-filling curve, such as the Hilbert-curve and Z-curve. This value is then indexed in the corresponding partition of the B+-tree. To process a range query, the B^x-tree checks each existing partition for qualifying objects. Specifically, in one partition, the B^x-tree enlarges the query window to the end timestamp of the corresponding phase. It first enlarges the query window according to the minimum and maximum velocities of all moving objects, and then makes the enlargement exacter using the space velocity histogram which records the minimum and maximum velocities of the space cells. Then the B^x-tree scans all multiple ranges of index keys which fall within the enlarged query window, and checks the corresponding moving objects to finds the ones satisfying the query. Since each partition of the B+-tree has to be checked, the query processing of the B^x-tree is not efficient, as shown by both the experiment results of [13] and those of ours.

The B^{dual}-tree in [13] also uses a B+-tree to index moving objects. However, it indexes objects in a dual space instead, considering both location and velocity information to generate index keys. The B^{dual}-tree maintains a set of MORs for each internal entry, and uses R-tree-like query algorithms as in the TPR/TPR*-tree. A range query searches the subtree of an internal entry only if any of its MORs intersects with the query region. By partitioning the velocity space, the B^{dual}-tree improves the query performance of the B^x-tree. However, maintaining the MORs introduces high computation workload, which slows down the fast update and high concurrency of the B+-Tree. It has worse update performance than the B^x-tree, and worse query performance than the TPR*-tree. In addition, by modifying the update and query algorithms of the B+-tree, the B^{dual}-tree can no longer be readily integrated into existing DBMSs.

Besides the query and update performance, there are some other issues with the moving object indexes. [4] proposes a series of B+-tree-based indexes structures to handle the problem of *doubtful positive*. When the update durations of moving objects are highly variable, the result of a predictive query may contain frequently updated objects whose states would be updated much earlier than the future query timestamp. These result

objects are apparently doubtful and are called doubtful positives. The work in [4] introduces the concept of *prediction life period* for every moving object to handle predictive queries with no doubtful positive. In addition, different indexes in [4] have different balance of update performance and query performance. However, these indexes have no ability of self-tuning. The DBA has to decide to choose which index to satisfy the application. And, when the requirement of the application changes, the DBA has to stop the index service, delete the current index, choose and rebuild another index. The ST^2B-tree [5] is a self-tuning index of moving objects, based on the B+-tree. However, the problem it addresses is the varying distribution of moving objects over time and space, while our work focuses on the change of the proportion of query and update operations.

3 The B^s-tree

3.1 Index Structure

The B^s-tree is built on the B+-tree without any change to the underlying B+-tree structure and insertion/deletion algorithms, thus is readily implementable in existing DBMS. In order to support B-link concurrency control, each internal node contains a pointer to its right sibling. A moving object is transformed to one or more 1D keys and indexed by the B^s-tree. We now introduce how the transformation works.

As [4], in order to handle the problem of doubtful positive, we also define a *prediction life period*, t_{per} for each moving object, as the interval from its last update timestamp to the next update timestamp. It is the future within which the current information of this moving object is predicted to be right, and it can be updated. Similar to all the above existing works of moving object indexes, moving objects in the B^s-tree are modeled as linear functions of time. Therefore, the position of a moving object at time t is given by $O = (\boldsymbol{x}, \boldsymbol{v}, t_{ref}, t_{per}) = \boldsymbol{x}(t) = (x_1(t), x_2(t), ..., x_d(t))$, where $t_{ref} < t < t_{ref} + t_{per}$. Here, $\boldsymbol{x}$ and $\boldsymbol{v}$ is the location and velocity of the moving object at the reference time t_{ref}. Thus, $\boldsymbol{x}(t) = \boldsymbol{x} + \boldsymbol{v} * (t - t_{ref})$. For each query, the *query interval* indicates how far the query "looks" into the future. If the query interval of a query exceeds t_{per} of a moving object, this object is considered as a doubtful positive and will not appear in the result.

We partition the time axis into intervals of duration T_m, and then sub-partition each T_m into n equal-length sub-intervals of duration, each at length of T_{sam}, called *phases*. We label each phase with a timestamp t_{lab} which is the end time point of this phase. Unlike the B^x-tree which samples a moving object exact once, the B^s-tree samples a moving object one or more times. We introduce an *update parameter* α for the B^s-tree, where $1 \leq \alpha \leq n + 1$. When an insertion of a moving object arrives, the B^s-tree computes the positions at t_{lab} of every α phases located between $t_{lab} = [t_{ref}]_l$ and $t_{lab} = [t_{ref} + t_{per}]_l$, where $[t]_l$ means the nearest future t_{lab} of t. For each sampling, the B^s-tree maps the position to a signal-dimensional value, called KEY_{sam}, using the Hilbert curve. This value is concatenated with the phase number of its t_{lab}, called KEY_{lab}, to generate the index key values B_{value}, which is then inserted to a B+-tree. The relationship and computation methods are as follow:

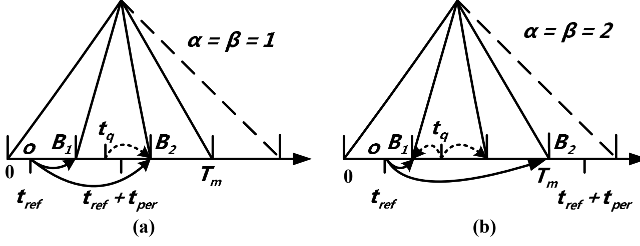

Fig. 2. An example of Insertion and Query in the B^s-tree

$$B_{value} = KEY_{lab} \oplus KEY_{sam},$$
$$KEY_{lab} = (t_{lab}/(T_m/n)) \bmod (n+1),$$
$$KEY_{sam} = Curve(\boldsymbol{x}(t_{lab})).$$

In figure 2, the solid lines with arrowheads show an insertion case of the B^s-tree when $\alpha = 1$ and $\alpha = 2$. We can see how a moving object is sampled, according to its prediction life period t_{per} and the update parameter α. The B^s-tree contains data belonging to $n + 1$ phases, and thus it includes $n + 1$ sub-trees, or partitions. As time passes, repeatedly the oldest partition expires, and a new one is appended. Therefore, the index rolls over. An update of a moving object deletes its old sampled values and inserts new ones. Except maintaining the space velocity histogram H_s as the B^x-tree, the B^s-tree also maintain a partition velocity histogram H_p, which stores the maximum and minimum velocities of the moving objects that inserted into each partition.

3.2 Query Algorithms

A range query $Q_r(R, t_q)$ retrieves all moving objects which intersect the query region R at the query time t_q, which does not precede the current time. We introduce a *query*

Algorithm Range Query $Q_r(R, t_q)$
Input: R is the query window and t_q is the query time.
Output: All objects in R at t_q within their $t_{per}s$
1. $t_{lab} \leftarrow [t_q]_l$
2. **For** $k \leftarrow 1$ to β
3. $R' = ExpandWindow(R, t_q, t_{lab}, H_p, H_s)$
4. **For** each key range in R'
5. **For** each moving object in the key range
6. Compute its position at t_q
7. **If** it is in R at t_q within its t_{per}
8. Add it to the result set
9. Find the next key range
10. **If** not found
11. Break
12. $t_{lab} \leftarrow [t_q]_l - T_{sam}/n$
13. Return result set

Fig. 3. Range Query Algorithm

parameter β for the B^s-tree. Unlike the B^x-tree which expands query windows to each partition, the B^s-tree only expands query windows β times to the phase which t_q belongs to and the $\beta - 1$ ones before. In figure 2, the dashed lines with arrowheads show the phases which a query window is expanded to when $\beta = 1$ and $\beta = 2$. We will discuss the relationship between α and β in section 4.2.

Figure 3 shows the range query algorithm of the B^s-tree. The algorithm searches the partition which the query time belongs to and the $\beta - 1$ ones before (Line 2). For each partition, it expands the query window from the query time t_q to the end timestamp t_{lab} of the corresponding phase (Line 3). Here, H_p indicts the partition velocity histogram and H_s indicts the space velocity histogram. The enlargement first expand the window according to the maximum and minimum velocities of the corresponding phases in H_p to get a preliminary window R_{pre}, and then finds the minimum and maximum velocities in the cells that R_{pre} intersects, using H_s, to get the exacter expanded window R'. After the enlargement, the algorithm can get several *key ranges* in R', which are one-dimensional intervals of index values falling within the expanded query window R'. For each key range, a range query of the B+-tree is executed (Line 4). For each object found in the key range, the algorithm checks whether it is in R at t_q within its t_{per}. If so, the object satisfies the query and is added to the result set (Line 5-8).

A KNN query $Q_{KNN}(p, t_q, k)$ retrieves k moving objects for which no other moving objects are nearer to o at t_q. As in the B^x-tree, a KNN query of the B^s-tree is handled as incremental range queries $Q_r(R, t_q)$ until exact k nearest neighbors are found. The initial search range is a region centered at p_t with extension $r = D_k/k$, where p_t is the position of p at t_q. If the KNNs are not found in this region, it extends the search radius by r as in [12]. D_k is estimated by the equation as in [8] [5]:

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

Note that, during the incremental range queries, the query window of a range query will contain the former one. Therefore, for the KNN queries of the B^s-tree, we record the expanded window of a range query for each partition. When processing the next range query, the area covered by the former expanded window does not need to be checked.

4 Self-tuning of the B^s-tree

In this section, we first analyze the update and query I/O cost of the B^s-tree. And then basing on the analysis, we introduce the online self-tuning framework of the B^s-tree.

4.1 Update and Query Performance Analysis

First, let us focus on the update performance of the B^s-tree. For a B+-based-tree, the height of the tree is usually significantly important for the update performance. Assuming that the B^s-tree maintains N values, has the fan-out of F, and its nodes are filled with at least $F/2$ values. So the height is $H = \log_{F/2} N + 1$ at the most. Since sampling a moving object for one or more times, the B^s-tree may ask for a bigger N than the B^x-tree. However, with a big F which is usually in the hundreds, such a bigger N will

seldom affect H. As most indexes of moving objects, update of a moving object in the B^s-tree is a deletion operation followed by an insertion operarion. Therefore, the total update I/O cost of the B^s-tree is the sum of the deletion and insertion cost:

$$IO_{upd} = IO_{del} + IO_{ins} = \left(\left\lceil \frac{\left[t_{ref} + t_{per}\right]_l - \left[t_{ref}\right]_l}{T_{sam} * \alpha} \right\rceil + \left\lceil \frac{\left[t'_{ref} + t'_{per}\right]_l - \left[t'_{ref}\right]_l}{T_{sam} * \alpha'} \right\rceil \right) * H.$$

Here, $\lceil a \rceil$ means the smallest integer which is no smaller than a. Note that, for a moving object, among different updates, the prediction life period t_{per} and the update parameter α can change. We can see that the I/O cost of an update is only several times the tree height. Since H and T_{sam} are fixed, and t_{ref} and t_{per} are depended on the moving object itself, the update parameter α is the most important issue of the update cost. The update performance increases with α. We can tunes the update performance by tuning α.

Second, let us turn to the query performance of the B^s-tree. The B^s-tree handles range queries using the method of query window enlargement. It expands the query window for β times and traverses β partitions of the tree. Therefore, the total query I/O cost is the sum of the cost of the β partition traverses:

$$IO_{upd} = \sum_{i=1}^{\beta} \left(\sum_{j=1}^{m} \left(H - 1 + R_j \right) \right).$$

Here, for each partition traverse, m is the number of key ranges falling in the expanded window. For each key range, there is a unique search followed by a range search in the B+-tree. Specially, the I/O cost of a unique search is $H - 1$ internal nodes of the index, while the I/O cost of the range search is R_j leaf nodes of the index. Figure 4 gives an range query example of the figure 2 (b) where $\alpha = \beta = 2$. O is the position of the moving object o at the reference time. B_1 is its first key value sampled in the B^s-tree, while B_2 is not visible in figure 4. The shadowed rectangle is the original query window. It is expanded twice, one to the phase t_q belongs to and one for the phase before. In the first enlargement, there are two key ranges without containing any sampling value of o. While, in the second enlargement, after two unique-range scans of the B+-tree, it finds B_1. There are totally two enlargement and four unique-range scans for this range query.

The total cost of a range query is decided by the number of partition traverses: β. While, for each partition traverse, the cost is decided by the number and length of the

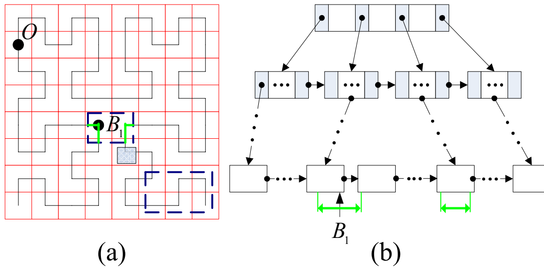

Fig. 4. An example of Query Analysis

key ranges. The larger the query window is expanded, the more and longer the key ranges it contains. The enlargement velocities and the enlargement time interval are the two issues of the query window enlargement. For the first issue, except the space velocity histogram used by the B^x-tree, the B^s-tree also uses the partition velocity histogram to make the enlargement velocities exacter. As for the second issue, the *enlargement time interval* set S_{eti} of the β enlargement is estimated as follow:

$$S_{eti} = \{\left|t_q - [t_q]_l\right|, T_{sam} - \left|t_q - [t_q]_l\right|, ..., (\beta - 1) * T_{sam} - \left|t_q - [t_q]_l\right|\}.$$

Here, $\left|t_q - [t_q]_l\right|$ is the enlargement time interval of the phase which t_q belongs to, while the $k * T_{sam} - \left|t_q - [t_q]_l\right|$ is the enlargement time interval of the kth phase of the $\beta - 1$ phases before. Therefore, the biggest enlargement time interval is $MAX(\left|t_q - [t_q]_l\right|, (\beta - 1) * T_{sam} - \left|t_q - [t_q]_l\right|)$. The enlargement time interval increases with β. The bigger β is, the larger a query window is expanded, and the higher the partition traverse cost is. In summary, we can see that the query parameter β is the most important issue of the range query costs. It not only decides the times of query window enlargement, but also effects how large a query window is expanded to.

As for KNN queries of the B^s-tree, since a KNN query is handled as incremental range queries, the I/O cost can be approximately estimated as the I/O cost of the last range query. However, it will be a little bigger. This is because, although the range queries in a KNN query of the B^s-tree do not check the same key values repeatedly, a continuous key range within the expanded window of the last range query may be divided by different range queries. Some additional unique scans are asked for, and some leaf nodes shared by different range queries are accessed several times. However, for the cost of KNN queries, the query parameter β is still the most important issue. To sum up, we can tunes the query performance of the B^s-tree by tuning β.

In addition, from the above analysis, we can also observe that since $T_{sam} = T_m/n$, n effects both update performance and query performance. A larger n results in more partitions and may require more update costs in the B^s-tree. While a smaller n results in larger query window enlargement, which increases the query cost. In this paper we choose $n = 3$, as the previous experiments of B+-based-tree of moving objects in [4] show that it is an appropriate balance value.

4.2 Self-tuning Framework

From the above analysis, we find that the update and query performance of the B^s-tree can be tuned by the update parameter α and the query parameter β. However, α and β can not be changed arbitrarily. We should guarantee the correctness of the queries.

Theorem 1. *As long as β is no less than all α of all moving objects, the query correctness of the B^s-tree can be guaranteed. That is, given a query, the B^s-tree can find all moving objects which satisfy the query in their t_{per}s.*

Proof: For a moving object $o(t_{ref}, t_{per}, \alpha_o)$, where α_o is its current update parameter, it will be sampled at following time:

$$\{[t_{ref}]_l, [t_{ref}]_l + \alpha_o * T_{sam}, [t_{ref}]_l + \alpha_o * 2 * T_{sam}, ..., [t_{ref}]_l + \alpha_o * m * T_{sam}\}.$$

Here, $[t_{ref}]_l + \alpha_o * m * T_{sam}$ is the biggest t_{lab} which is not bigger than $t_{ref} + t_{per}$. Given a future query $Q(t_q)$ where $t_{ref} \le [t_q]_l \le t_{ref} + t_{per}$, assuming that o satisfies Q at t_q and the query parameter $\beta < \alpha_o$, if $[t_q]_l = [t_{ref}]_l + (\alpha_o * i + j) * T_{sam}$ where $i < m$ and $\beta \le j < \alpha_o$, the B^s-tree will not search any partition which o is sampled to, therefore will not find o and the result is not correct. In reverse, if β is no less than all α of all moving objects, the B^s-tree will not omit any partition which any result object is sampled to. As long as the partition is traversed, the method of query window enlargement guarantees that any moving object which satisfies the query in this partition will be found. Proved. ■

Note that, if α of the B^s-tree changes, the moving objects which are already in the index have the old α, while a new insertion or update is according to the new α. Therefore, in the index, the moving objects may have different α. According to Theorem 1, β of the B^s-tree should be equal to the biggest α of all moving objects to guarantee the query correctness. Bigger β is meaningless, as it increases query costs. If all moving objects in the B^s-tree have the same α, we call the B^s-tree is in a *steady state*, and $\beta = \alpha$. Also note that, there is some restriction for the longest update interval of a moving object. In the B^s-tree, the time interval T_m should be larger than most $t_{per}s$ of all moving objects in the system. In addition, similarly to [8] [5], for those rare moving objects who are not updated in the last T_m, they are "flushed" to new partitions. The new positions are estimated using their last updated positions and velocities. Their new $t_{per}s$ are the rest $t_{per}s$. However, their α could be changed. Thus, the longest update or flush interval of a moving object is restricted to T_m.

Now we introduce how the online self-tuning framework works. In the framework, the cost we focus on is the I/O cost, since it is usually more important than the CPU cost. Recall that, as shown in figure 1, OTM is the key module which monitors the performance of the B^s-tree and controls the tuning. OTM works in a "self-learning and self-tuning" way. And it has three tasks. First, it maintains a histogram $H_{rec} = \{m, n, C_{upd}, C_{que}\}$ which includes the number and average cost of updates and queries during the recent period of time T_{rec}, represented as m, n, C_{upd}, C_{que}. T_{rec} is the "self-learning" time window. It should be a multiple of T_{sam}, $T_{rec} = l * T_{sam}$. Specially, in order to get H_{rec}, OTM maintains the number and total cost of updates and queries of the l most recent T_{sam}. Thus, H_{rec} can be easily carried out at the end timestamp of each T_{sam}. T_{rec} can be set according to different applications, and it can be reset at any time without breaking the index service. In this paper, we set $T_{rec} = 2 * T_{sam} = 2 * n * T_m$.

Second, OTM also maintains a set S_{base} which includes the average cost of updates and queries for each steady state of the B^s-tree, ranging $\alpha = \beta$ from 1 to $n + 1$.

$$S_{base} = \{C^1_{upd}, C^1_{que}, C^2_{upd}, C^2_{que}, ..., C^{n+1}_{upd}, C^{n+1}_{que}\}$$

For initialization, we give predictive values for the elements in S_{base}. Then, when in a steady state, the corresponding elements of the current $\alpha = \beta$ are updated by C_{upd}, C_{que} in H_{rec} at the end timestamp of each T_{sam}. The initialized predictive values in S_{base} can be given by experiments. We give the guide line of setting the initialized S_{base} in the experiment department.

Third, OTM adjusts the B^s-tree in a self-tuning way. As shown in figure 1, there is a Timer trigger for OTM. At the end timestamp of each T_{sam}, OTM not only updates H_{rec} and S_{base}, but also starts a *checking procedure*. It decides weather to tune the update

parameter α and query parameter β of the index. The overall performance is the average performance of each operation. It is estimated by the following equation:

$$C_{all} = \frac{m * C^{\alpha}_{upd} + n * C^{\beta}_{que}}{m + n}.$$

Here, m is the number of updates and n is the number of queries. OTM uses the current m and n in H_{rec}, and different C^{α}_{upd} and C^{β}_{que} in S_{base} for each $\alpha = \beta$ to compute C_{all}. Then, it chooses $\alpha = \beta$ which gets the lowest C_{all}. In order to avoiding thrash, if this C_{all} is 15% lower than the C_{all} with the current $\alpha = \beta$, OTM will decide to tune α and β, and enter a *tuning procedure*. Note that, the checking procedure is efficient, since there are only a little computation and comparison during the checking procedure.

For the tuning procedure, the current update parameter and query parameter are represented as α_{cur} and β_{cur}, while the target update parameter and query parameter which OTM decides to tune to is represented as α_{tar} and β_{tar}. In addition, α_{tun} and β_{tun} are the update parameter and query parameter during the tuning procedure. Since the current and target states both are steady states, we have $\alpha_{cur} = \beta_{cur}$ and $\alpha_{tar} = \beta_{tar}$. If $\alpha_{tar} < \alpha_{cur}$, at the beginning of the tuning procedure, OTM sets $\alpha_{tun} = \alpha_{tar}$ and $\beta_{tun} = \alpha_{cur}$. During the tuning procedure, the B^s-tree will include different moving objects with α_{cur} or α_{tar}. Thus, β_{tun} has to be equal to the biggest update parameter of all available partitions, α_{cur}, to guarantee the query correctness. Note that, a moving object will be updated or flushed at least once during the last T_m. Therefore, after time period of T_m, the update parameter of all moving object will be the new one α_{tar}, the B^s-tree will reach a steady state, and OTM tunes the query parameter to $\beta_{tar} = \alpha_{tar}$. While, on the other hand, if $\alpha_{tar} > \alpha_{cur}$, at the beginning of the tuning procedure, OTM sets $\alpha_{tun} = \beta_{tun} = \alpha_{tar}$. After time period of T_m, the B^s-tree reaches a steady state, with no need to reset the query parameter. The length of the tuning procedure is T_m. During the tuning procedure, OTM does not update H_{rec} and S_{base}, and does not enter the checking procedure. When the tuning procedure finishes, OTM resets H_{rec} to empty and continues its work. Note that, the index service is not broken.

Although OTM has the ability of self-tuning, it allows for manual configuration. This enhances the flexibility and usability of the B^s-tree. The administrator can set $\alpha_{tar} = \beta_{tar}$ at any time. OTM will then enter the tuning procedure at the end timestamp of the current T_{sam}. On the other hand, the administrator can also "freeze" the B^s-tree to avoid too frequent tuning. As a result, OTM will not enter the checking and tuning procedures. In addition, the initial value of $\alpha = \beta$ when creating the B^s-tree, is also customizable and should be set according to specific applications.

5 Experiments

In this section, we experimentally compare the B^s-tree with the B^x-tree [8] and the TPR*-tree [11], which are the most representative B+-tree and R-tree based indexes for moving objects. All experiments are implemented in the C++ language, and conducted on a 2.6GHz Pentium 4 Personal Computer with 1GB memory, running Windows XP Professional. The page size and index node size are both set to 4 KB.

We use a data generator similar to the one used by the B^x-tree [8]. The objects move in the space domain of $1000 * 1000$. The initial object positions are generated randomly, so are the moving directions. The moving speed in each dimension is selected randomly from -3 to 3. The update frequencies (indicated by t_{per}) of the moving objects are variable among various moving objects. 35% of the moving objects have $t_{per}s$ ranging from 0 to $1/3T_m$, while another 35% are in range $(1/3T_m, 2/3T_m]$. And the remaining 30% moving objects are in $(2/3T_m, T_m]$. As the experiments in [8], T_m is set to 120 time units. For the B^x-tree and the B^s-tree, we choose $n = 3$ and use the Hilbert-curve as the space-filling curve. For the TPR*-tree, the horizon H (how far the queries can "see" in the future) is set to 60 time units. For each dataset, we execute same 100 predictive queries and evaluate their average cost. The query time q_t ranges from 1 to H. For range queries, the side length of query windows is chosen from 10 to 50.

5.1 Basic Performance

In this set of experiments, we study the basic query and update performance of the B^s-tree, the B^x-tree and the TPR*-tree. Figure 5 $(a)-(d)$ show the average number of I/Os and CPU time for each range query and update at various dataset sizes, ranging from $100K$ to $500K$. For a more clear view, we represent the query and update performance of the B^s-tree for different steady states. B^s-tree(j) indicts the B^s-tree in the steady state with $\alpha = \beta = j$, from 1 to $n+1 = 4$. As expected, when $\alpha = \beta = 4$, the query and update performance of the B^s-tree is very similar with that of the B^x-tree. This is because both the B^s-tree(4) and the B^x-tree sample once for an update and expand query windows $n + 1 = 4$ times for a range query. Therefore, in these figures, we omit the query and update performance result of the B^s-tree(4), using that of the B^x-tree instead. From these experiments, we can observe that the B^x-tree has better update performance than the TPR*-tree, while the TPR*-tree has better query performance than the B^x-tree. The B^s-tree has the ability to tune the query and update performance by different α and β. It

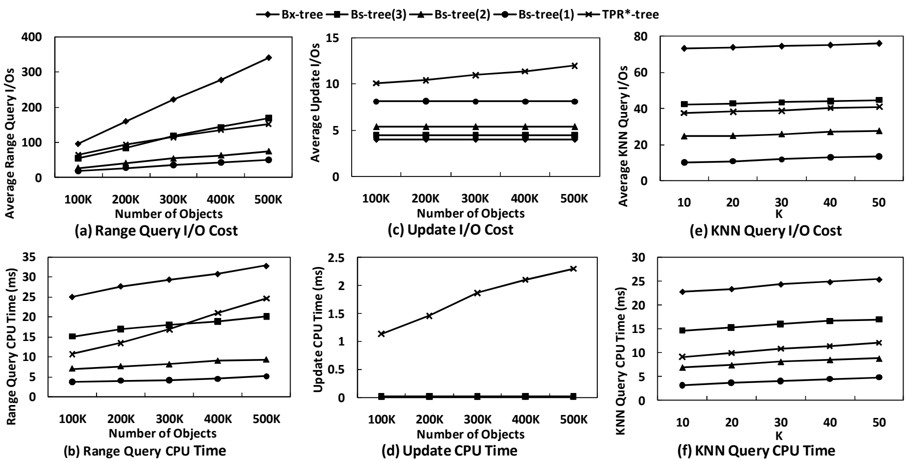

Fig. 5. Basic Query and Update Performance

can have almost the same query and update performance as the B^x-tree when update cost is the major part of the overall cost, while in other cases, it can pay slightly more update cost to achieve considerably higher performance of queries. Therefore, it "dominates" the B^x-tree. In addition, when $\alpha = \beta = 2$ or $\alpha = \beta = 3$, the B^s-tree has both better query performance and update performance than the TPR*-tree. It "dominates" the TPR*-tree. We can also see that the cost of queries increases with the date cardinality for all the three kinds of indexes. However, for the B^x-tree and the B^s-tree, the update cost is not apparently affected by the dataset size. In addition, for all the three kinds of indexes, the update cost is much lower than the query cost. In the rest experiments, the default dataset size is $100K$. Figure 5 (e) and (f) shows the average number of I/Os and CPU time per KNN query while varying K from 5 to 50. Observe that the effect of k is not very significant. The relationship of the performance of the three indexes is similar with that of the range query performance.

5.2 Overall Performance

In this experiment, we study the overall performance of the three trees by combining both update and query operations in the same workload. For the query operations, the range queries and KNN queries are mixed with the proportion of 5 : 1, considering that the range queries are more common. We vary the ratio of the number of queries over the number of updates from 1 : 1000 to 100 : 1. Figure 6 shows the average overall I/Os and CPU time for each operation. For the B^s-tree, S_{base} of OTM is initialized using the results in subsection 5.1. The overall performance of the B^s-tree with different proportion of updates and queries was recorded when the B^s-tree reached the steady states. From this experiment, we can observe that the B^s-tree adjust its performance in a self-tuning way. When the total update cost is much higher than the total query cost, it automatically pays some more query cost to achieve lower update cost. While, oppositely, when the total query cost is much higher than the total update cost, it automatically pays some more update cost to achieve better query performance. In this way, the B^s-tree keeps good and smooth overall performance despite the change of the workload. It can be seen that in almost all cases, the B^s-tree outperforms the TPR*-tree and the B^x-tree in overall performance.

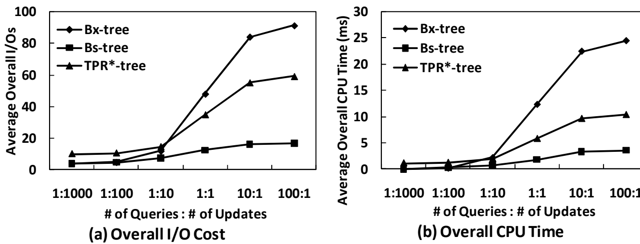

Fig. 6. Overall Performance

5.3 Self-tuning Performance

The above experiments show that the B^s-tree has good performance when it reaches the steady states. However, if the performance degrades markedly when the B^s-tree is in the

tuning procedures, it is still not appropriate for real use. In this experiment, we study the self-tuning performance of the B^s-tree. We vary the proportion of the number of queries over the number of updates from 1 : 100 to 100 : 1, and then return to 1 : 100. Correspondingly, $\alpha = \beta$ of the B^s-tree tunes from initial value 4 to 1, and then return to 4. Note that as shown in section 5.2, since the query cost is much higher than the update cost, the overall cost will increase with the ratio of queries, though it is much slighter for the B^s-tree than for the TPR*-tree and the B^x-tree. Figure 7 shows the average overall I/Os and CPU time for each operation. Here T indicts the performance when the B^s-tree is in the tuning procedures, while S indicts the performance when the B^s-tree reaches the steady states. We can see that the self-tuning cost of the B^s-tree is slight. During the tuning procedures, the B^s-tree pays only a little more cost than that of reaching the steady state later. And note that the length of a tuning procedure is only T_m. In addition, we can see that the extra tuning cost of the B^s-tree from a higher $\alpha = \beta$ to a lower $\alpha = \beta$ is slighter than that from a lower $\alpha = \beta$ to a higher $\alpha = \beta$, since the methods of tuning procedure are different. In conclusion, The B^s-tree can provide non-break index service with good and smooth performance.

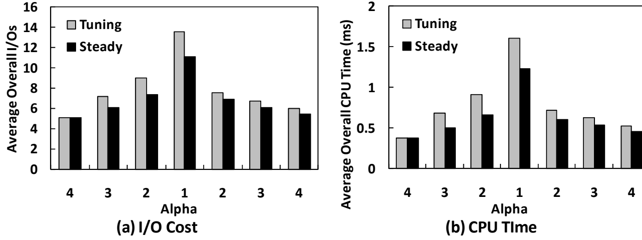

Fig. 7. Tuning Performance

5.4 Concurrency Performance

Finally, we study the concurrency performance, using a multi-thread program to simulate multi-user environments. To highlight the difference between the two B+-tree based indexes, we do not show the result for the TPR*-tree since it has been shown to be inefficient in a concurrent environment in [8] and [6]. The B-link technique [10] is used as the concurrency control technique for the B^x-tree and the B^s-tree. We use a workload varying the proportion of the number of query operations over the number of update operations from 1 : 100 to 100 : 1, and then return to 1 : 100, with initial $\alpha = \beta = 4$. Figure 8 shows the average throughput and response time of the whole workload, while varying the number of threads from 1 to 6. Throughput is the rate at which operations could be served by the system and response time is the time interval between issuing an operation and getting the response from the system when the task is successfully completed. As expected, the B^s-tree outperforms the B^x-tree, basing the same B+-tree structure and the same concurrency control technique. This is because when the update operations dominate the overall performance, the B^s-tree with $\alpha = \beta = 4$ has almost the same update and query performance. While, in other cases, the B^s-tree with smaller $\alpha = \beta$ pay slight more update cost to achieve significantly higher query performance, which results much better overall performance than the B^x-tree.

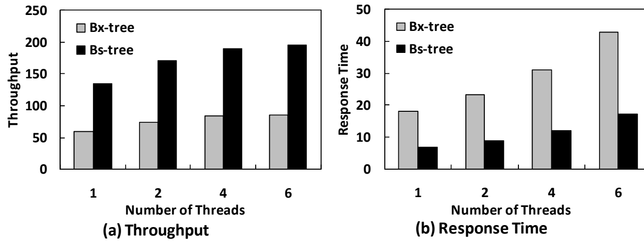

Fig. 8. Concurrent Performance

6 Conclusion

With the development of moving object databases, requirements to handle the applications, where the ratio of update and query operations varies widely with time, are becoming essential. In this paper, we propose a moving object index structure, namely B^s-tree. The B^s-tree has the ability of adapting its update and query performance to meet different requirements. We implement various algorithms for the B^s-tree. In addition, we present an online self-tuning framework which provides self-tuning for optimal overall performance without interrupting the indexing service. Our experiment studies show that, the B^s-tree achieves good and smooth overall performance with efficient self-tuning procedures. Therefore, the proposed B^s-tree is efficient and suitable for dynamic applications in which the proportion of query and update operations varies significantly by time. For future work, we would study other self-tuning techniques for moving object databases.

Acknowledgement

This work was supported in part by the National Science Foundation of China (NSFC Grant No. 60803003, 60970124) and by Chang-Jiang Scholars and Innovative Research Grant (IRT0652) at Zhejiang University.

References

1. Beckmann, N., Kriegel, H.-P., Schneider, R., Seeger, B.: The R*-Tree: An Efficient and Robust Access Method for Points and Rectangles. In: SIGMOD Conference, Atlantic City, NJ, May 1990, pp. 322–331 (1990)
2. Benetis, R., Jensen, C.S., Karciauskas, G., Saltenis, S.: Nearest Neighbor and Reverse Nearest Neighbor Queries for Moving Objects. VLDB J. 15(3), 229–249 (2006)
3. Chaudhuri, S., Narasayya, V.R.: An Efficient Cost-Driven Index Selection Tool for Microsoft SQL Server. In: VLDB, Athens, Greece, August 1997, pp. 146–155 (1997)
4. Chen, N., Shou, L.-D., Chen, G., Dong, J.-X.: Adaptive Indexing of Moving Objects with Highly Variable Update Frequencies. Journal of Computer Science and Technology (JCST) 23(6), 998–1014 (2008)
5. Chen, S., Ooi, B.C., Tan, K.-L., Nascimento, M.A.: S2TB-Tree: A Self-Tunable Spatio-Temporal B+-Tree Index for Moving Objects. In: SIGMOD Conference, Vancouver, BC, Canada, June 2008, pp. 29–42 (2008)

6. Guo, S., Huang, Z., Jagadish, H.V., Ooi, B.C., Zhang, Z.: Relaxed Space Bounding for Moving Objects: A Case for the Buddy Tree. SIGMOD Record (SIGMOD) 35(4), 24–29 (2006)
7. Guttman, A.: R-Trees: A Dynamic Index Structure for Spatial Searching. In: SIGMOD Conference, Boston, Massachusetts, June 1984, pp. 47–57 (1984)
8. Jensen, C.S., Lin, D., Ooi, B.C.: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. In: VLDB, Toronto, Ontario, Canada, August 2004, pp. 768–779 (2004)
9. Saltenis, S., Jensen, C.S., Leutenegger, S.T., Lopez, M.A.: Indexing the Positions of Continuously Moving Objects. In: SIGMOD Conference, Dallas, Texas, USA, May 2000, pp. 331–342 (2000)
10. Srinivasan, V., Michael, Carey, J.: Performance of B-Tree Concurrency Algorithmss. In: SIGMOD Conference, Denver, Colorado, May 1991, pp. 416–425 (1991)
11. Tao, Y., Papadias, D., Sun, J.: The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In: VLDB, Berlin, Germany, September 2003, pp. 790–801 (2003)
12. Tao, Y., Zhang, J., Papadias, D., Mamoulis, N.: An Efficient Cost Model for Optimization of Nearest Neighbor Search in Low and Medium Dimensional Spaces. IEEE Trans. Knowl. Data Eng. (TKDE) 16(10), 1169–1184 (2004)
13. Yiu, M.L., Tao, Y., Mamoulis, N.: The Bdual-Tree: indexing moving objects by space filling curves in the dual space. VLDB J. (VLDB) 17(3), 379–400 (2008)