

Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects

Dan Lin¹

Christian S. Jensen²

Beng Chin Ooi¹

Simonas Šaltenis²

¹School of Computing
National University of Singapore, Singapore
{lindan, ooibc}@comp.nus.edu.sg

²Department of Computer Science
Aalborg University, Denmark
{csj, simas}@cs.aau.dk

ABSTRACT

Although significant effort has been put into the development of efficient spatio-temporal indexing techniques for moving objects, little attention has been given to the development of techniques that efficiently support queries about the past, present, and future positions of objects. The provisioning of such techniques is challenging, both because of the nature of the data, which reflects continuous movement, and because of the types of queries to be supported. This paper proposes the BB^x -index structure, which indexes the positions of moving objects, given as linear functions of time, at any time. The index stores linearized moving-object locations in a forest of B^+ -trees. The index supports queries that select objects based on temporal and spatial constraints, such as queries that retrieve all objects whose positions fall within a spatial range during a set of time intervals. Empirical experiments are reported that offer insight into the query and update performance of the proposed technique.

Categories and Subject Descriptors

H.2.2 [Database Management]: Physical Design—*access methods*; D.3.2 [Information Storage and Retrieval]: Information Storage—*file organization*

General Terms

Algorithms, Performance

Keywords

Mobile objects, indexing, B-tree

1. INTRODUCTION

We are approaching a situation where it is practical for individuals to be online always and everywhere. This situation is in large measure brought about by advances in consumer electronics and mobile communications. Further, relatively accurate positioning of mobile objects is becoming practical. An infrastructure for location-enabled services [15] is thus emerging.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

MDM 2005 05 Ayia Napa Cyprus

Copyright 2005 ACM 1-59593-041-8/05/05 ...\$5.00.

Motivated in part by such advances, new spatio-temporal indexing techniques are being developed. Some techniques index the current and (anticipated) future positions of spatial objects [13, 17], while others aim to index the past, historical, positions of spatial objects [8, 11, 16].

Techniques of the latter type support a variety of queries. Timeslice queries retrieve all objects that meet specified spatial constraints at a single point in time, while time range queries retrieve all objects that satisfy the spatial constraints at some time point during a specified time interval. The possibly most basic spatial constraint states that an object must belong to a rectangular region. Indexes that support these basic types of queries are often also useful for the processing of more complex queries.

However, we are not aware of any single index in the literature that is capable of indexing the historical, present, and future positions of moving objects. The closest technique, proposed recently by Sun et al. [14], is a summary structure that supports approximate aggregate queries.

Motivated by the absence of indexing support for the historical, present, and future positions of moving objects, we propose the Broad B^x indexing technique (BB^x -index), which is designed with storage and query efficiency in mind. More specifically, the BB^x -index structure is based on the B^x -tree [4], which applies a novel linearization technique to timestamped locations—so that the resulting values preserve spatial proximity and are also time-wise partitioned—and indexes the linearized positions using a B^+ -tree.

The BB^x -index inherits the ability to index present and future positions from the B^x -tree, and it extends this ability with support for also past positions. Three factors contribute to its storage efficiency. First, object positions are represented as linear functions (as is done by, e.g., the TPR-tree family). In comparison to constant functions, this representation reduces the number of updates and thus data volume to one third for data deriving from cars [3]. Second, an object is usually indexed only once in the BB^x -index. Third, being a B^+ -tree based index, the index does not rely on bounding rectangles, rendering the internal directory relatively compact. The uses of linear functions and the B^+ -tree contribute to obtaining good update performance.

Having a single index for historical, present, and future positions not only results in indexing that is both space and query efficient; it also makes it possible to support some complex queries neatly and efficiently, with no need for complicated optimization strategies. For example, we demonstrate that it is possible to support a new time- and location-constrained set query that retrieves all objects whose positions fall within a spatial range during a set of time intervals.

We conducted the the performance study, and our experimental

results demonstrate that the BB^x -index significantly outperforms the latest existing method MV3R-tree [16] with respect to storage and historical query efficiency, and performs similar to the B^x -tree with respect to the predictive query efficiency.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the structure of the proposed BB^x -index. Section 4 presents the associated query and update operations. Section 5 covers comprehensive performance experiments. Finally, Section 6 concludes.

2. RELATED WORK

As pointed out, indexing techniques for moving objects generally come in two variants: techniques for indexing the present and future positions of objects and techniques for indexing the historical positions of objects. Several recent surveys of moving-object indexing techniques exist that focus on different aspects [1, 6, 9].

Representatives of the first variant of indices include the TPR-tree (Time-Parameterized R-tree) family of indexes [12, 13, 17]. Next, Patel et al. [10] have recently proposed a practical indexing technique, termed STRIPES, that supports efficient updates and queries at the expense of higher space requirements. STRIPES uses dual transformation [5], where the linear movement of a point object moving in d -dimensional space is represented as a point in a $2d$ -dimensional space. Another recent index is the B^x -tree [4], which uses the B^+ -tree to manage moving objects efficiently; we cover the B^x -tree in more detail in the next section.

We proceed to cover the indexing of historical positions in more detail, since our proposed indexing technique aims to solve problems that are prevalent in existing index structures.

One of the earliest works is the Historical R-tree (HR-tree) [8], which logically constructs a “new” R-tree each time an update occurs. HR-trees are efficient for timeslice queries, as the search degenerates to a static query, for which R-trees are efficient. Their disadvantage is extensive duplication of objects, which leads to a very high space consumption. Consequently, the performance of the HR-tree on interval queries is very poor.

The historical movements of objects can be represented by trajectories composed of sequences of connected, spatio-temporal line segments. For example, an object moving in two-dimensional (x, y) space can be represented by such a trajectory in three-dimensional (x, y, t) space. Such trajectories can be indexed by an R-tree because they, as well as their individual, constituent line segments, can be enclosed by bounding boxes. Based on this idea, Pfoser et al. [11] propose the Spatio-Temporal R-tree (STR-tree) and the Trajectory-Bundle tree (TB-tree). The former organizes line segments not only according to spatial proximity, but attempts also to group the segments according to their trajectory membership. The TB-tree aims only for trajectory preservation and basically ignores spatial proximity. These indices experience degrading performance as the length of the history increases.

Next, Tao and Papadias [16] propose the Multi-Version 3D R-tree (MV3R-tree), which consists of a Multi-Version R-tree (MVR-tree) and an auxiliary 3D R-tree. MVR-trees utilize the concepts of Multi-Version B-trees [2], which have good performance for timeslice queries. The auxiliary 3D R-tree compensates for the inefficiency of interval queries in MVR-trees. However, in order to offer good timeslice performance, the MV3R-tree uses so-called version splits that duplicate data during update operations, and the auxiliary structure requires extra storage space.

A recent proposal by Sun et al. [14] supports queries about the past, present, and future. However, on approximate aggregate query results can be computed. In applications where accurate results are needed, other proposals are needed.

Despite the existence of several indexing techniques for historical positions and for present and future positions, no single moving-object index for the past, present, and future positions of moving objects has yet been reported in the literature that achieves the goals of storage efficiency and query and update efficiency.

3. BB^x -INDEX

As the BB^x -index borrows from the B^x -tree, we first introduce the B^x -tree and then present the structure of the BB^x -index. Section 4 covers the associated algorithms.

3.1 The B^x -Tree

The B^x -tree [4] is a B^+ -tree index structure that indexes the current and future positions of moving objects. To be able to use the B^+ -tree, which is capable only of indexing data belonging to a totally ordered linear domain, moving-object positions are linearized.

The movement of an object is given by a linear function of time and a time when the function is valid. For movement in two dimensional space, such a movement O can be described by a two dimensional velocity vector \vec{v} and a two dimensional point \vec{x} , in addition to the time value t_u , resulting in a point in a higher dimensional space. These points are mapped to a one-dimensional space by means of a space filling curve. The specifics are given shortly.

A space-filling curve is a function that enumerates every point in a discrete, multi-dimensional space. In our context, a space-filling curve is generally considered good if it preserves proximity, meaning that points close in multidimensional space tend to be close in the one-dimensional space obtained by the curve. The Peano curve and the Hilbert curve are prime examples of good space-filling curves [7]. The B^x -tree may use any space-filling curve x (indicated by the superscript in its name), and we choose the Hilbert curve which has better performance than other curves as reported in [4].

The motivation for representing the position of an object as a linear function of time is fourfold. First, it is possible to estimate such linear functions in prominent applications, e.g., applications that perform frequent sampling and applications that use GPS for positioning. Second, linear functions have quite compact descriptions, using only two more parameters than do the conventional, constant functions for two-dimensional movement. More complex functions are harder or impossible to obtain, and they are less compact. Third, linear functions offer better predictions of future positions than do constant functions. Fourth, the use of linear functions reduces the volume of updates. Intuitively, an update occurs when the position predicted by an existing function is deemed inaccurate. Studies of GPS data from cars indicate that for a range of accuracies, the use of linear functions reduces the number of updates to one third of the updates needed when using constant functions [3].

The B^x -tree effectively “partitions” the positions it indexes, placing positions in so-called phases based on their update time. Specifically, the time axis is first partitioned into intervals of duration Δt_{mu} (the anticipated maximum duration in-between two updates of any object location), and each such interval is further partitioned into n equal-length sub-intervals, which are the phases. A value of $n = 2$ has been shown to yield an index with good query and storage efficiency [4]. Figure 1 shows a B^x -tree with $n = 2$.

For a moving object, the key value indexed by the B^x -tree is the concatenation of a phase number and the result of applying the underlying space-filling curve to the position of the object as of the end time of the phase, termed the label timestamp of the phase. The phase used when computing the key value is the one that follows the phase that intersects with the insertion time of the object. Using

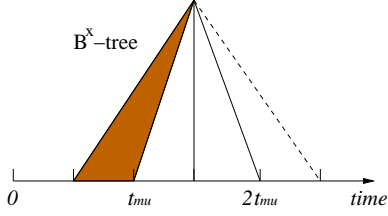


Figure 1: The B^x -Tree

the notation introduced above, the mapping function is:

$$B^x \text{ value}(O, t_u) = [\text{phase}]_2 \oplus [x\text{-rep}]_2 \quad (1)$$

Here, $[x]_2$ denotes the binary representation of x , and \oplus denotes concatenation. The two components of the function are defined as follows:

$$\begin{aligned} \text{phase} &= (t_{lab}/(\Delta t_{mu}/n) - 1) \bmod (n + 1) \\ x\text{-rep} &= x\text{-value}(\vec{x} + \vec{v} \cdot (t_{lab} - t_u)) \end{aligned}$$

Here, $t_{lab} = \lceil t_u + \Delta t_{mu}/n \rceil_l$, where operation $\lceil x \rceil_l$ returns the nearest future label timestamp of x . Next, $x\text{-value}$ is obtained from the space-filling curve, \vec{x} , \vec{v} are the position and the velocity of the given object, and t_u is the time when the object issues an update.

To handle queries occurring at timestamps other than the label timestamps, the B^x -tree enlarge the query window to enclose objects that are not in the query window at the label timestamps, but may possibly be in the query window at the query timestamp. Because the B^x -tree stores an object's location as of some time after its update time, the enlargement involves two cases: a location must either be brought back to an earlier time or forward to a later time. Consider the example in Figure 2, where t_{ref} denotes the

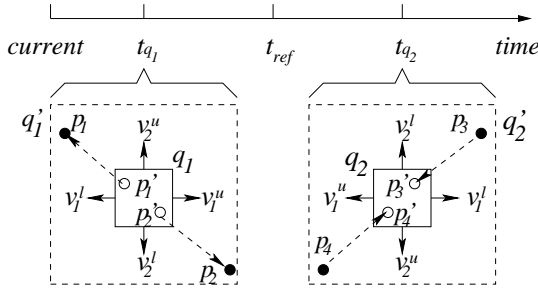


Figure 2: Query Window Enlargement

label timestamp that four moving objects p_1 , p_2 , p_3 and p_4 are indexed in the tree, and where queries q_1 and q_2 (solid rectangle) have time parameters t_{q_1} and t_{q_2} respectively. The figure shows the locations of four moving objects at the query time as circles, from which we can see that these four objects are the answers of the two queries. In order to obtain these answers, the query windows are enlarged (dashed rectangles) using the expansion speed. The details of computation of the expansion speed can be found in [4].

3.2 BB^x -Index Structure

The BB^x -index consists of nodes that in turn consist of entries, each of which is of the form $\langle x\text{-rep}, t_{start}, t_{end}, pointer \rangle$. For leaf nodes, $pointer$ points to the objects with the corresponding $x\text{-rep}$, where $x\text{-rep}$ is obtained from the space-filling curve; t_{start} denotes the time when the object was inserted into the database (corresponding to the t_u in the description of the B^x -tree), and t_{end} denotes the

time that the position was deleted, updated, or migrated (migration refers to the update of a position done by the system). For non-leaf nodes, $pointer$ points to a (child) node at the next level of the index: t_{start} and t_{end} are the minimum and maximum t_{start} and t_{end} values of all the entries in the child node, respectively. In addition, each node contains a pointer to its right sibling to facilitate query processing.

Unlike the B^x -tree, the BB^x -index is a forest of trees, with each tree having an associated timestamp signature t_{sg} and a lifespan (see Figure 3). The *timestamp signature* parallels the value t_{lab} from the B^x -tree and is obtained by partitioning the time axis in the same way as for the B^x -tree. The lifespan of each tree corresponds to the minimum and maximum lifespans of objects indexed in the tree. The roots of the trees are stored in an array, and they can be accessed efficiently according to their lifespan. This array is relatively small and can usually be stored in main memory.

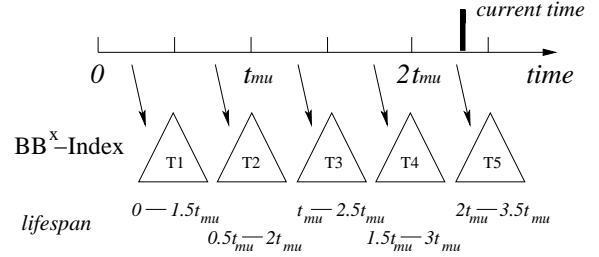


Figure 3: The BB^x -Index

Objects inserted during the same phase will be stored in the tree with the t_{sg} that is equal to the end timestamp of that phase. In particular, an update with timestamp t_{start} is assigned a timestamp signature $t_{sg} = \lceil t_{start} \rceil_l$, where $\lceil x \rceil_l$ returns the smallest timestamp signature that does not precede x .

The position of an object is represented by a single-dimensional value $x\text{-rep}$ obtained from a space-filling curve. In order to retain the proximity-preserving property of the space-filling curve, we index objects within a time interval by their positions as of the time given by the timestamp signature of this interval. Hence, we need to determine an object's position at the timestamp signature according to its moving function.

An object's linear movement $O = (\vec{x}, \vec{v})$ is given by a position and a velocity at the time of update, t_{start} . The transformation from the current position to the position \vec{x}_{ind} that will be indexed is: $\vec{x}_{ind} = \vec{x} + \vec{v} \cdot (t_{sg} - t_{start}) = \vec{x} + \vec{v} \cdot (\lceil t_{start} \rceil_l - t_{start})$. We thus place the position $x\text{-rep}$ computed by applying the space-filling curve to \vec{x}_{ind} in the tree with timestamp signature t_{sg} .

Note that we do not concatenate the timestamp signature and $x\text{-rep}$ as in the B^x -tree. There are two reasons for this. First, our index aims to handle moving objects from the past to the future. Thus, the index must contend with timestamps that keep growing in value. Inclusion of such values in the key would pose an efficiency problem since we must then allocate substantial space for the key in order to cater to its growth. In contrast, the B^x -tree only indexes current positions of moving objects and hence is able to fix the length of the key value (by using the modulo function). Second, without considering the timestamp, we obtain a shorter key and a simpler mapping function. Imagining that the index runs for one year, the accumulated timestamp value ($\approx 2^{24}$ minutes) would require a long key value representation, which will significantly reduce the node capacity and fanout, which increases index size and decreases query performance.

Let us illustrate the BB^x -index with an example. Figure 3 shows

a BB^x -index with $n = 2$. Objects inserted between timestamps 0 and $0.5t_{mu}$ are stored in tree T_1 with their positions as of time $0.5t_{mu}$; those inserted between timestamp $0.5t_{mu}$ and t_{mu} are stored in tree T_2 with their positions as of time t_{mu} ; and so on. Each tree has a maximum lifespan: T_1 's lifespan is from 0 to $1.5t_{mu}$ because objects are inserted starting at timestamp 0 and because those inserted at timestamp $0.5t_{mu}$ may be alive throughout the maximum update interval t_{mu} , which is thus until $1.5t_{mu}$; the same applies to the other trees.

4. QUERY PROCESSING

In the following, we first unify historical and predictive queries, then consider the range query and the set query. Finally, updates are considered.

4.1 Unifying Historical and Predictive Queries

To understand how the BB^x -index is able to support historical and predictive queries, consider Figure 4.

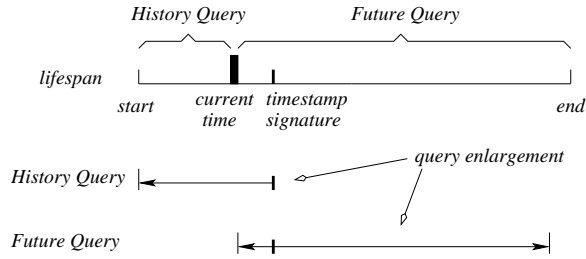


Figure 4: Historical Versus Predictive Query

Figure 4 shows the possible queries that happen in the latest tree in a BB^x -index: (i) historical queries refer to times between the start time of the lifespan and the current time; (ii) predictive queries refer to times between the current time and the maximum update interval from the current time.

Recall that object positions are indexed as of times equal to timestamp signatures. Thus, in order to answer queries as of other times than the timestamp signatures, the BB^x -index enlarges query windows to capture those objects that are outside the query window at the time specified by the query. Figure 4 illustrates the maximum query window enlargements for historical and predictive queries. The enlargements are similar for both kinds of queries. Consequently, the algorithms we propose in the following sections can answer queries on the historical as well as the future positions of moving objects.

4.2 Range Query

An interval range query retrieves all objects whose position falls within the rectangle $q = ([qx_1^l, qx_1^u], [qx_2^l, qx_2^u])$ at some time during a time interval $[t_l, t_u]$ (“l” denotes the lower bound, “u” denotes the upper bound).

Queries are handled separately for each tree whose lifespan intersects with the query interval. Figure 5 illustrates how three types of intersection may be discerned: (i) the query interval ends before the timestamp signature of the tree; (ii) the query interval intersects the timestamp signature; and (iii) the query interval starts after the timestamp signature.

Like the B^x -tree, the BB^x -index uses query window enlargement to counter the data transformation used. Different types of intersections are handled by different strategies for query window enlargement. For the first case, we use the maximum of the start time of the query interval and the start time of the lifespan to perform

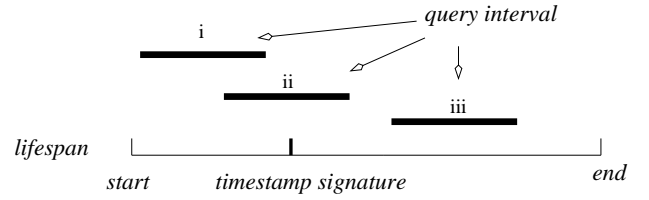


Figure 5: Lifespan Intersected with Query Interval

a backward enlargement. Similarly, for the third case, we use the minimum of the end time of the query interval and the end time of the lifespan to perform a forward enlargement. In the second case, the query interval is partitioned into two parts by the timestamp signature. Both forward and backward enlargements of the query window are performed. Then we compare the sizes of the enlarged query windows and keep the larger one.

Having obtained an enlarged query window, we travel the index. In each tree of the index, a range query in the native, two-dimensional space becomes a set of range queries in the transformed, one-dimensional space (due to the use of a space-filling curve).

The pseudo code of the algorithm is shown in Figure 6. In line 7, function `LeftmostIntersection` calculates the leftmost interval of intersection between the sequence of intervals q'' and the interval that extends from low . Then the leaf node containing the start point of this interval is located, and all entries in this leaf node that are after the start point are checked. If the last entry is smaller than the end point, we continue traversing right siblings by the right link (lines 10–12). Otherwise, we compute the next leftmost intersection (line 7) and then continue searching. This process repeats until we can no longer find any intersections.

Algorithm `Range_query(q, [t_l, t_u])`

Input: q is the query range, and $[t_l, t_u]$ is the query time interval

1. **for** each tree in the BB^x -index **do**
2. **if** the tree's lifespan intersects with $[t_l, t_u]$ **then**
3. enlarge q to q'
4. compute q'' , the space filling curve representation of q'
5. $low \leftarrow 0$
6. **repeat**
7. $[start, end] \leftarrow \text{LeftmostIntersection}(q'', [low, \infty))$
8. locate leaf node containing $start$
9. store objects with $x_{rep} \geq start$ from this node in L
10. **while** $entry.x_{rep} \leq end$, for the node's last $entry$
11. access right sibling node
12. store all objects from this node in L
13. $low \leftarrow entry.x_{rep} + 1$
14. **until** $\text{LeftmostIntersection}(q'', [low, \infty)) = \emptyset$
15. **for** each object in L **do**
16. **if** the object's position during $[t_l, t_u]$ is inside q **then**
17. add the object to the `result_set`
18. **return** `result_set`

Figure 6: Range Query Algorithm

We note that this algorithm, in contrast to that of the B^x -tree, is able to skip processing for some of the intervals in q'' . In particular, if an interval is contained in a leaf node that has already been accessed, the interval is skipped. Moreover, it is reasonable to assume that the query interval will not exceed the maximum update interval. Since there is only one copy of each object within the maximum update interval, the BB^x -index avoids redundant accesses to the same objects when compared to the MV3R-tree.

4.3 Time and Location Constrained Set Query

Queries on historical data may relate to temporal patterns. This is exemplified by the following query: “Which taxis were in the vicinity of a given hotel between times 18.00 and 18.30 on a weekday during a certain past month?” In this section, we define a new type of query, the *time and location constrained set query*, that supports this type of querying. The query retrieves the objects whose position falls within the rectangular range $q = ([qx_1^l, qx_1^u], [qx_2^l, qx_2^u])$ at some time during a set of time intervals $(t_{int1}, t_{int2}, \dots, t_{intk}) = ([t_1^l, t_1^u], [t_2^l, t_2^u], \dots, [t_k^l, t_k^u])$.

A naive method (NM) to process this query is to treat it as a series of independent interval range queries and then combine the results obtained from these. For instance, in Figure 7, query q is comprised of three sub-queries q_1 , q_2 , and q_3 . NM handles each single sub-query as an independent range query. For q_1 , NM queries tree T_1 ; for q_2 , trees T_1 and T_2 are queried since both of their lifespans intersect with the query time interval; for q_3 , trees T_1 , T_2 , and T_3 are queried. As a result, tree T_1 has been searched three times to answer set query q . As we shall see next, such repetitive search can be avoided.

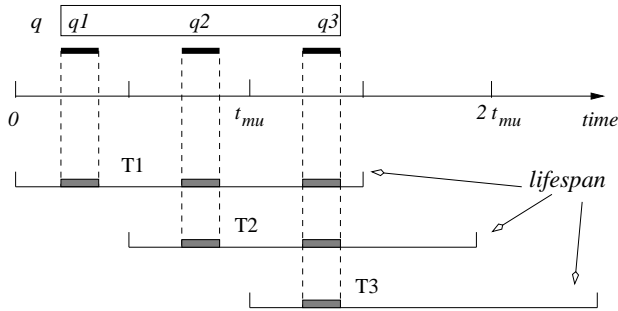


Figure 7: Example Time and Location Constrained Set Query

Figure 8 shows more details about the search in tree T_1 . Assuming that each query window enlargement has similar expansion speed, we obtain three ranges r_1 , r_2 , and r_3 for the sub-queries

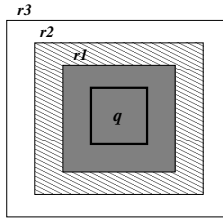


Figure 8: Query Enlargements in Tree T_1

q_1 , q_2 , and q_3 , respectively, according to their query time interval. Then, for sub-query q_1 , we need to search the range r_1 , and for sub-query q_2 , we need to search the range r_2 . We can see that r_2 covers r_1 , which means the objects inside r_1 are retrieved twice here due to the overlap of ranges. Similarly, to answer the sub-query q_3 , we need to search the range r_3 , where r_3 covers r_2 and objects inside r_2 will also be retrieved twice. Thus, region r_1 has been searched three times and r_2 has been searched twice.

Based on this observation, we propose a more efficient method for computing the set query, termed the grouping method (GM). The main idea is to group sub-queries for each tree and handle sub-queries in the same group simultaneously, thus avoiding repetitive traversals of trees.

Given the example set query q from Figure 7, GM first considers trees T_1 , T_2 , and T_3 because their lifespans intersect with time interval $[t_1^l, t_3^u]$, where t_1^l and t_3^u are the earliest and latest query times of the set queries, respectively. In each tree, GM identifies all sub-queries whose query intervals intersect with the lifespan of current tree and then computes the intersections of the time intervals. Using the resulting time intervals, GM enlarges the query window and obtains several candidate query ranges. For example, the query time intervals of three sub-queries all intersect with the lifespan of T_1 , and hence GM will compute the enlargements of these three sub-queries and obtain ranges r_1 , r_2 and r_3 . Then, GM will only search using the largest query range (in the example, r_3), and further distinguishes results for each sub-query. The pseudo code for GM is shown in Figure 9.

Algorithm Set_query($q, (t_{int1}, t_{int2}, \dots, t_{intk})$)

Input: q are the query range, and $(t_{int1}, t_{int2}, \dots, t_{intk})$ is the query time intervals

1. **for** each tree in the BB^x -index **do**
2. **if** the tree's lifespan intersects with $[t_1^l, t_k^u]$ **then**
3. **for** $i \leftarrow 1$ to $i \leq k$ **do**
4. **if** the tree's lifespan intersects with t_{inti}
5. enlarge q to q'
6. $maxq'$ stores the largest q'
7. $maxq'' \leftarrow$ space filling curve representation of $maxq'$
8. $low \leftarrow 0$
9. **repeat**
10. $[start, end] \leftarrow$ LeftmostIntersection($maxq''$, $[low, \infty)$)
11. locate leaf node containing $start$
12. store objects with $x_{rep} \geq start$ from this node in L
13. **while** $entry.x_{rep} \leq end$, for the node's last entry
14. access right sibling node
15. store all objects from this node in L
16. $low \leftarrow entry.x_{rep} + 1$
17. **until** LeftmostIntersection($maxq''$, $[low, \infty)$) = \emptyset
18. **for** each object in L **do**
19. **if** object's position during $(t_{int1}, t_{int2}, \dots, t_{intk})$ is inside q **then**
20. add the object to the result_set
21. **return** result_set

Figure 9: Group Method (GM)

4.4 Insertion, Deletion, and Migration

Insertion into the BB^x -index is similar to insertion into the B^x -tree [4]. To delete an object, we first find the tree where this object is stored. Rather than physically removing the object, we modify the end time of its lifespan, t_{end} , to be the current time.

Objects in an old tree that have not been updated within the maximum update interval should be migrated, by means of a deletion and an insertion, to the current tree. This system-controlled migration is needed to maintain the index structure. Although migration introduces additional updates, the amortized cost is small in practice (as shown in the Section 5.5) and is controllable (by increasing the maximum update interval).

The algorithm outline is shown in Figure 10.

5. PERFORMANCE STUDIES

In this section, we present the results of experimental performance studies conducted on the BB^x -index. Following a description of the experimental setting, the section considers in turn storage requirements, range queries, time- and location-constrained set

Algorithm Update(P_o, P_n)

Input: P_o and P_n are old and new objects respectively

1. $t_{index} \leftarrow$ the time P_o is indexed in the tree
2. find tree T_x whose lifespans contain t_{index}
3. $pos_{index} \leftarrow$ the position of P_o at t_{index}
4. $key_o \leftarrow$ the x -value of the pos_{index}
5. locate P_o in T_x according to key_o
6. modify the end time of P_o 's lifespan to current time
7. $t'_{index} \leftarrow$ the time P_n will be indexed
8. $pos'_{index} \leftarrow$ the position of P_n at t'_{index}
9. $key_n \leftarrow$ the x -value of the pos'_{index}
10. insert P_n into the latest tree according to key_n

Figure 10: Update Algorithm

queries, and updates. Comparisons with the B^x -tree and the MV3R-tree are included where appropriate.

5.1 Experimental Settings

In the experiments, we compare the historical and predictive query performance of the BB^x -index to the MV3R-tree and the B^x -tree, respectively. Query and update costs are measured in terms of node accesses. A page size of 1024 bytes is used, which results in fanouts of 36 in both the MV3R-tree and the BB^x -index, and 50 in the B^x -tree. The MV3R-tree source code [16] that we use in the experiments uses this page size; to be fair, we want to preserve whatever optimization may have been done to that code. Our other parameters are also similar to those used for the MV3R-tree [16]. See Table 1, where values in bold denote the default values used.

Parameter	Setting
Page size	1K
Agility	1, ..., 3% , ..., 25%
Spatial extent of queries	2%
Time interval length of queries	1.38% , 3.75%, 7.5%
Number of intervals in a query	2,4,6,8, 10
Query frequency	0.5, 1, 2 , 4, 8
Number of queries	100
Dataset size	5K, 50K

Table 1: Parameters and Their Settings

For all experiments, both the initial positions and the movements of the objects are uniform. At each timestamp, the percentage of objects that will update their movement function is roughly the same and corresponds to the agility of the dataset, i.e., a dataset has *agility* p , if on average $p\%$ of the objects change their movement function at each timestamp [16].

Two parameters are used when generating interval range queries for the experiments, namely the spatial extent, expressed as a fraction of the spatial universe in Table 1, and the temporal length of the queries, expressed as a fraction of the recorded history [16]. Queries are generated so that all timestamps are queried with the same probability.

The set query has two more parameters: the number of time intervals and the query frequency. Query frequency f_q indicates how many intervals occur in one maximum update interval and is represented as the division of the maximum update interval t_{mu} by the average length between each interval start time. Given a query q and its time intervals $([t_1^l, t_1^u], [t_2^l, t_2^u], \dots, [t_k^l, t_k^u])$, the query frequency is given as $f_q = (k - 1) \cdot t_{mu} / \sum_{i=1}^{k-1} (t_{i+1}^l - t_i^l)$.

5.2 Storage Requirements

As the space efficiency is important when indexing historical data, we compare the sizes of the MV3R-tree and the BB^x -index. We created datasets with cardinality 5K and agilities in the range [1, 25] (same as in reference [16]). Figure 11 shows the size of each structure after 100 timestamps as a function of the agility. We observe that the BB^x -index has the smaller size. This is because one

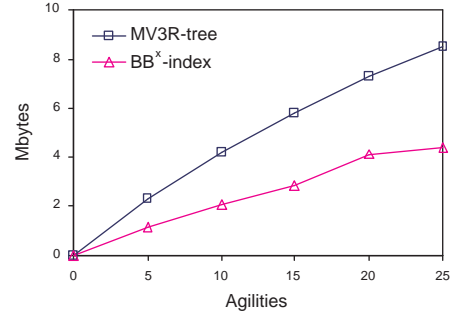


Figure 11: Storage Requirements

object may be duplicated several times in the MV3R-tree due to the version splits. In the BB^x -index, each object has only one copy, except those (few) objects that are not updated within the average update interval and need to be migrated. While it is possible to use a variety of settings for the parameters that offer some control over the replication done by the MV3R-tree, we note that we have used the settings in the MV3R-tree that were used in reference [16].

5.3 Range Queries

We proceed to consider the historical and predictive query performance of the BB^x -index, the MV3R-tree, and the B^x -tree, respectively. We evaluate range query performance over datasets of 50K objects that evolve for 200 timestamps. We use a workload with 3% agility and 2% query extents.

5.3.1 Historical Queries

The historical queries have 3.75% query length and are distributed uniformly over the past 200 timestamps. As shown in Figure 12, the BB^x -index achieves significantly better performance than the MV3R-tree. This is due partly to the fact the BB^x -index is smaller in size and simpler in structure.

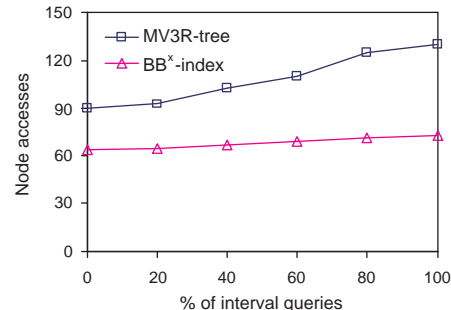


Figure 12: Historical Range Query Performance

Next, we study the behavior of the MV3R-tree and the BB^x -index as time passes. We executed the same query workloads (with 2% extents and 7.5% query length) after the dataset (3% agility) evolves for 200 and 500 timestamps. The results are shown in Figure 13.

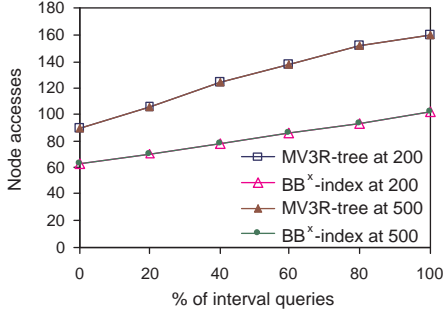


Figure 13: Deterioration of MV3R-Trees and BB^x -Indexes as Time Progresses

Note that the lines of the same index are overlapping. The results show that both the MV3R-tree and the BB^x -index are nearly unaffected by the passing of time. By comparing the results with Figure 12, we note that both are, however, affected by the query length. In fact, when the query length is large, the effect of ratio of interval queries on the query performance is more profound.

5.3.2 Predictive Queries

In this experiment, we evaluate the performance of the BB^x -index for predictive queries. From this experiment and onwards, we can no longer use the MV3R-tree as the benchmark index since it does not support predictive queries. Instead, we compare the performance of the BB^x -index against the more specialized (and smaller) B^x -tree, which is now applicable.

In the experiment, the maximum update interval t_{mu} is about 34 timestamps, as a result of a 3% agility. The predictive query time is randomly selected from $[now, now + t_{mu}]$. Figure 14 shows the predictive range query performance of the B^x -tree and the BB^x -index after every 50 timestamps. Observe that the two indices have

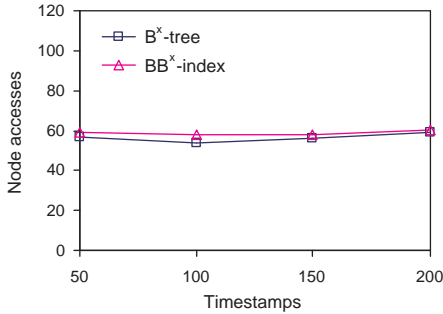


Figure 14: Predictive Range Query Performance

comparable performance as time passes, though the query cost of the BB^x -index is slightly larger than that of the B^x -tree. This is because the BB^x -index needs to store more information than the B^x -tree, which results in a smaller fanout. Moreover, the difference between the mapping functions makes the average length of query window enlargement for the predictive query in the BB^x -index larger than that of the B^x -tree, because the BB^x -index also has to guarantee efficient historical query performance.

5.4 Time- and Location-Constrained Set Query

In this set of experiments, we study the properties of the set query by testing various parameters including the number of time intervals, time interval length, and query frequency. The BB^x -index is

created with 50K objects and run for 500 timestamps before performing 100 set queries (number of intervals is 10, interval length is 1.375% and f_q is 2 by default). We compare the efficiency of the algorithms NM and GM.

5.4.1 Effect of Number of Time Intervals

In this experiment, we measure the effect of the number of time intervals by varying the number from 1 to 10. Figure 15 shows the set query performance of two algorithms. Note that when there is only one interval, algorithms NM and GM are the same. When the

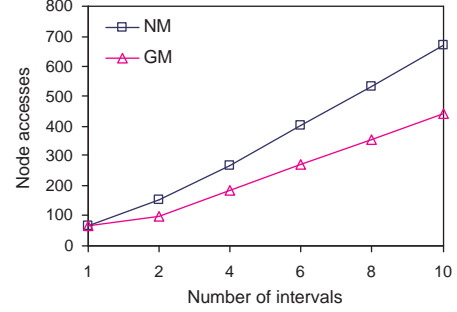


Figure 15: Effect of Number of Intervals

number of intervals increases, GM performs much better than NM. The query cost of NM increases linearly with the number of time intervals since it is essentially a sum of a series of range queries. In contrast GM can save the cost when several time intervals fall into the same lifespan of one tree of the BB^x -index.

5.4.2 Effect of Time Interval Length

Next, we evaluate the effect of the time interval length. Figure 16 illustrates the query performance of both algorithms. It is no sur-

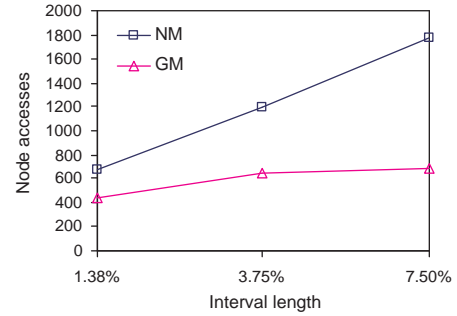


Figure 16: Effect of Interval Length

prise that NM deteriorates very fast with an increase of the interval length. This is due to the accumulated effect of range queries with long time intervals. However, the query cost of GM first increases a little and then stays almost constant. This is because the performance of the GM is only affected by the sub-queries with the longest time interval inside each tree. When the time interval becomes longer, the query window enlargement may reach the maximum lifespan of the tree, and thus the performance will no longer degenerate.

5.4.3 Effect of Query Frequency

We now study the effect of query frequency. Figure 17 shows the query performance with the query frequency f_q ranging from 0.5 to 8. It is interesting to observe that NM and GM achieve the same

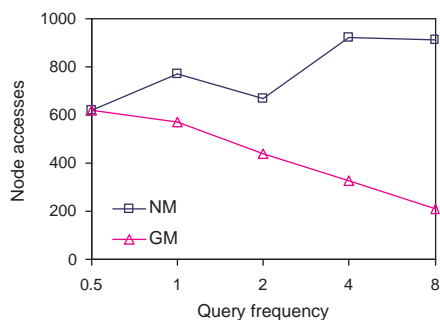


Figure 17: Effect of Query Frequency

performance when f_q is equal to 0.5 (i.e., each sub-query happens after $2t_{mu}$). At this time, no sub-queries fall into the same tree of the BB^x -index, and hence GM also needs to handle all the sub-queries separately. However, when the query frequency increases, GM starts to show its efficiency. The higher the frequency, the better GM performs since more sub-queries can be grouped and done together in one tree traversal.

5.5 Update Cost

In order to investigate the performance degradation across time, we measure the update cost (amortized over insertion and deletion) of the BB^x -index after every 20 timestamps over a 50K dataset. As shown in Figure 18, the BB^x -index retains almost constant performance and is not affected by time. This is because given the key value, each deletion or insertion only needs to traverse one path from the top to the bottom of the tree.

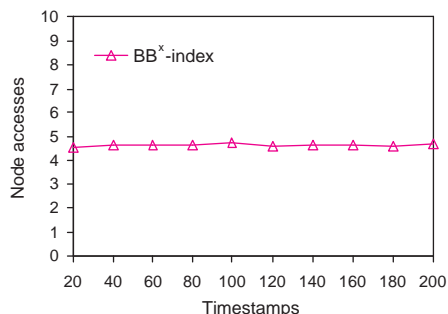


Figure 18: Update Performance

6. SUMMARY AND FUTURE RESEARCH

This paper presents a new indexing technique, the BB^x -index, which can answer queries about the past, the present, and the future. The BB^x -index is based on the concepts underlying the B^x -tree. For the indexing of historical information, it avoids duplicating objects and thus achieves significant space saving and efficient query processing. For predictive queries, we improve the algorithms of the B^x -tree. Moreover, we define and offer a technique for processing a new type of query, a spatio-temporal set query. Extensive performance studies were conducted that indicate that the BB^x -index outperforms the existing state-of-the-art method, the MV3R-tree, with respect of historical queries, and performs similarly to the B^x -tree with respect to predictive queries.

Several promising directions for future work exist. One is to improve current algorithms after considering extreme cases. For

example, one such case occurs when half objects are updated frequently while half are not, which results in relatively many forced updates. Another direction is to experiment with the proposed structure in new applications. One desirable application would be to predict possible traffic jams and to guide drivers based on historical and current information.

Acknowledgments

This work was supported in part by grant 216 from the Danish National Center for IT Research. In addition to his primary affiliation, the second author is an adjunct professor at Agder University College, Norway.

7. REFERENCES

- [1] P. K. Agarwal and C. M. Procopiuc. Advances in Indexing for Mobile Objects. *IEEE Data Eng. Bull.*, 25(2): 25–34, 2002.
- [2] B. Becker, S. Gschwind, T. Ohler, B. Seeger, and P. Widmayer. An Asymptotically Optimal Multiversion B-Tree. *VLDB Journal* 5(4): 264–275, 1996.
- [3] A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. In *Proc. MobiQuitous*, pp. 164–173, 2004.
- [4] C. S. Jensen, D. Lin, and B. C. Ooi. Query and Update Efficient B+-Tree Based Indexing of Moving Objects. *Proc. VLDB*, pp. 768–779, 2004.
- [5] G. Kollios, D. Gunopulos, V. J. Tsotras. On Indexing Mobile Objects. In *Proc. PODS*, pp. 261–272, 1999.
- [6] M. F. Mokbel, T. M. Ghanem, and W. G. Aref. Spatio-Temporal Access Methods. *IEEE Data Eng. Bull.*, 26(2): 40–49, 2003.
- [7] B. Moon, H. V. Jagadish, C. Faloutsos, and J. H. Saltz. Analysis of the Clustering Properties of the Hilbert Space-Filling Curve. *IEEE TKDE*, 13(1): 124–141, 2001.
- [8] M. A. Nascimento and J. R. O. Silva. Towards Historical R-trees. In *Proc. ACM Symposium on Applied Computing*, pp. 235–240, 1998.
- [9] B. C. Ooi, K. L. Tan, and C. Yu. Fast Update and Efficient Retrieval: an Oxymoron on Moving Object Indexes. In *Proc. of Int. Web GIS Workshop, Keynote*, 2002.
- [10] J. M. Patel, Y. Chen, and V. P. Chakka. STRIPES: An Efficient Index for Predicted Trajectories. In *Proc. ACM SIGMOD*, pp. 637–646, 2004.
- [11] D. Pfoser, C. S. Jensen and Y. Theodoridis. Novel Approaches in Query Processing for Moving Objects. In *Proc. VLDB*, pp. 395–406, 2000.
- [12] S. Šaltenis and C. S. Jensen. Indexing of Moving Objects for Location-Based Services. In *Proc. ICDE*, pp. 463–472, 2002.
- [13] S. Šaltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the Positions of Continuously Moving Objects. In *Proc. ACM SIGMOD*, pp. 331–342, 2000.
- [14] J. Sun, D. Papadias, Y. Tao, and B. Liu. Querying about the Past, the Present, and the Future in Spatio-Temporal Databases. In *Proc. ICDE*, pp. 202–213, 2004.
- [15] J. Schiller and A. Voisard, editors. *Location-Based Services*. Morgan Kaufmann Publishers, 2004.
- [16] Y. Tao and D. Papadias. MV3R-Tree: A Spatio-Temporal Access Method for Timestamp and Interval Queries. In *Proc. VLDB*, pp. 431–440, 2001.
- [17] Y. Tao, D. Papadias, and J. Sun. The TPR*-Tree: An Optimized Spatio-Temporal Access Method for Predictive Queries. In *Proc. VLDB*, pp. 790–801, 2003.