

Algorithms for Database Systems Seminar, ETH Zurich  
Spring semester 2010

Report on

# **Query and Update Efficient B<sup>+</sup>-Tree Based Indexing of Moving Objects**

Corina Bassi  
ETH Zurich  
[cbassi@student.ethz.ch](mailto:cbassi@student.ethz.ch)

## **Abstract**

This report presents a way of indexing moving objects with a B<sup>+</sup>-tree based data structure. This concept was first described 2004 by Jensen, Lin and Ooi in the paper „Query and Update Efficient B<sup>+</sup>-Tree Based Indexing of Moving Objects“ [1]. Apart from the structure of the index this report also describes how to execute range and *k*NN queries and how to perform updates on the index.

## 1. Introduction

Moving objects can be many different things. For example users of wireless devices, cars or any other moving vehicle. Building an index for such data poses problems, which traditional indexes for spatial data can't handle. Traditional indexes have good query performance. However, they are not efficient for updates which is crucial for an index for moving objects, as they change their position frequently.

The proposal of using a  $B^+$ -tree offers several advantages. The  $B^+$ -tree is very efficient for insertion and retrieval. This is mainly due to the high fanout, which means that several objects are stored in the same node in the tree. In a  $B^+$ -tree objects are only stored in leaf nodes. The internal nodes are directory nodes. An important concept in the  $B^+$ -tree are the right sibling pointers. They are stored in every node and point to the right neighbor on the same level in the tree. They allow a fast in-order traversal of the stored objects.

As the  $B^+$ -tree is a one-dimensional index, one has to find a way to represent moving objects linearly. This linear representation should preserve proximity. Meaning that objects which are close in reality should be close in the linear representation. For the linearization a space-filling curve like the Peano curve or the Hilbert curve can be used [1]. They both preserve proximity.

The paper [1] presented in this report proposes an index called the  $B^x$ -tree, where the  $x$  denotes the space dimension for the index. This index is based on the  $B^+$ -tree and profits by all the advantages like the good performance for insertion and retrieval. However, the  $B^x$ -tree is able to index multidimensional data, using a space-filling curve for the linearization of the data.

This report is organized as follows. The next section describes the Time-Parametrized R-tree (TPR-tree) [2] which was state-of-the-art before the proposal of the  $B^x$ -tree. Section 3 describes the structure of the index as well as queries and updates on it. Section 4 looks at the performance of the  $B^x$ -tree and compares it to the TPR-tree and finally section 5 draws conclusions and looks at some follow-ups on this topic.

## 2. Related Work

The goals of the TPR-tree [2] and the  $B^x$ -tree [1] are the same: Index moving objects to allow queries on the current and near-future. However they choose a different approach. While the  $B^x$ -tree is based on the  $B^+$ -tree, the TPR-tree is based on the R-tree and therefore indexes multidimensional data without the linearization which is necessary for the  $B^x$ -tree.

A detailed description of the R-tree [3] and the TPR-tree [2] can be found in the referenced papers. In this report only those features of these trees are described, which are essential to see the differences in structure and performance between the TPR-tree and the  $B^x$ -tree.

The core concept of the R-tree are minimum bounding rectangles (MBRs). They allow the tree to store multidimensional data. MBRs group objects that belong together and enclose them with the smallest possible rectangle. For retrieval of the objects one traverses the resulting tree. For a large number of objects MBRs are overlapping and multiple paths have to be traversed for retrieval of objects. This leads to bad performance results. Figure 1 shows a simple example of an R-tree. The tree on the right hand side is the R-tree that results from the MBRs on the left hand side.

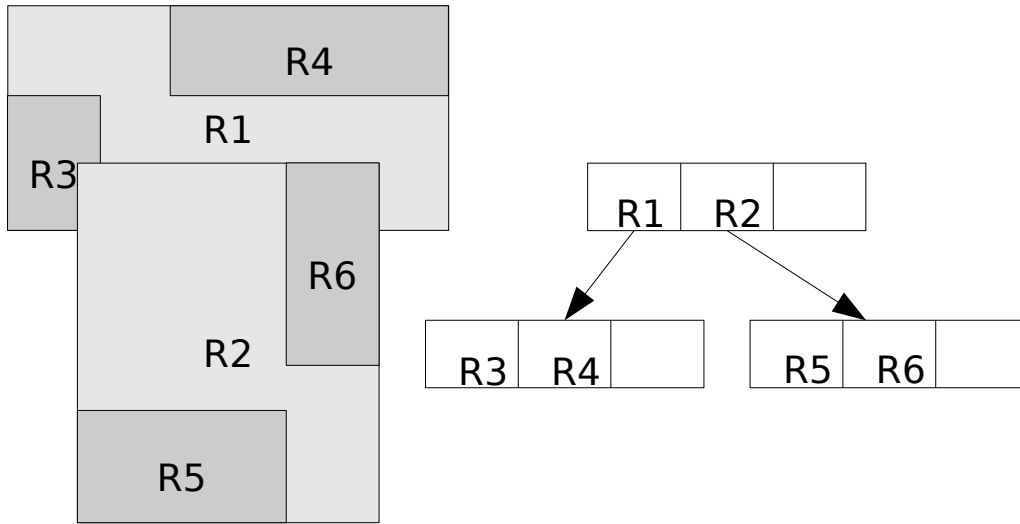


Figure 1: Simple R-tree.

In the TPR-tree not only the moving objects but also the MBRs are represented as linear functions of time. The lower bounds of the MBRs move with the maximum downward speed of all enclosed objects and their upper bounds do likewise with the maximum upward speed. Therefore, one can be sure that objects can't move out of their bounding rectangles at any time. One MBR always contains the same objects. However the MBRs are constantly growing. Whenever they are accessed they have to be tightened to the actually necessary size, but they still tend to overlap more and more after some time.

### 3. Structure and Algorithms

#### 3.1 Structure

The B<sup>x</sup>-tree [1] is an index for moving objects which allows queries on the current and near-future. For efficient querying the index is built taking not only the location, but also the time information into account. The B<sup>x</sup>-tree always consists of a number of subtrees. Every subtree contains objects which were last updated in a certain time interval. Within the subtree the objects are stored according to their current location. This structure ensures the index is as close to the real situation as possible at any point in time.

##### 3.1.1 Object Representation

Objects represented in the B<sup>x</sup>-tree are linear functions of time. They are represented by the following parameters (for a two-dimensional space):

$$\mathbf{O} = ((x, y), (v_x, v_y), t)$$

- $x/y$ : coordinates describing the location of the object at the current point in time
- $v_x/v_y$ : velocities in the  $x$  and  $y$  direction (valid for the current point in time)
- $t$ : current point in time (time of the update)

Whenever the velocity vector of an object changes, it has to be updated.

##### 3.1.2 Time Information

As mentioned above the data in the B<sup>x</sup>-tree is partitioned into phases according to the update times of the objects. In figure 2 one can see an example for a B<sup>x</sup>-tree. The triangles like the brown colored one denote the partitions of the B<sup>x</sup>-tree. There is always a fixed number of partitions with the same size as can be seen on the time axis in figure 2. In [1] the authors find the number of partitions as follows:

- Define  $\Delta t_{mu}$  as the maximum interval between two updates of an object.
- Split the interval  $\Delta t_{mu}$  into  $n$  phases. The authors of [1] claim  $n = 2$  to be the best choice.

The authors of [1] do not elaborate on this any further.

Every object is part of one partition according to its update time. Upon insertion of a new or updated object into the index one has to find the partition this object belongs to. In figure 2 this is described by the arrows. Objects with an update time 0 go into partition 0, objects with an update time between 0 and  $(1/2) \cdot \Delta t_{mu}$  go into partition 1 and so on.

To compare the positions of the objects of one phase with each other, one has to compare them at the same point in time. Therefore, every partition has a label time which is called  $t_{lab}$ . The objects of one phase are all mapped to this label time. They are stored with their location at the label time.

To calculate the partition number for an object that one wants to insert one can use the following formulas:

- $t_{lab} = (t_u + \Delta t_{mu} / n)$  where  $t_u$  is the last update time of the object
  - The result is rounded to next future label timestamp.
  - The rounded result tells to which partition the object belongs.

- partition number =  $(t_{lab} / (\Delta t_{mu} / n) - 1) \bmod (n + 1)$ 
  - The modulo operation is done because there is a fixed number of partitions and only so many numbers are used instead of counting the absolute number of partitions.
  - The -1 is because the partition numbering starts from 0.

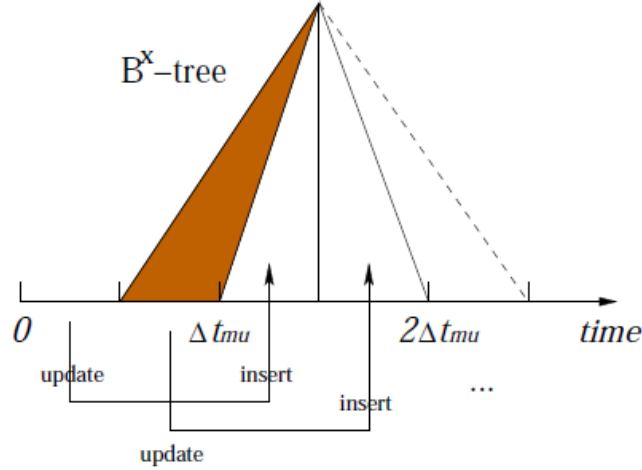


Figure 2: B<sup>x</sup>-tree. Wikipedia.

### 3.1.3 Location Information

To store the spatial properties of an object a linearization of the data is needed. This can be done using the Peano-curve (Figure 3) or the Hilbert-curve (Figure 4), which are space-filling curves that preserve proximity. They visit every point in a multidimensional space. Points that are close in space are still close on the resulting curve.

Starting at the lower left corner in figure 3 one can follow the curve to give an increasing number to every cell on the grid. Every object receives the number of the cell in which it is. For figure 4 the procedure is the same.

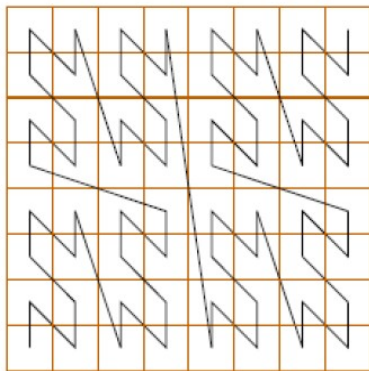


Figure 3: Peano-curve. [1].

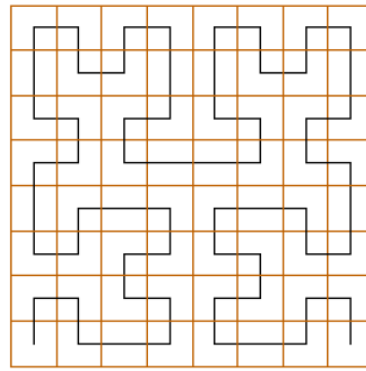


Figure 4: Hilbert-curve. [1].

These curves are applied to every partition of the B<sup>x</sup>-tree separately, because the partitions represent objects at different points in time.

### 3.1.4 Index Key

The index key that is inserted into the B<sup>x</sup>-tree is a concatenation of time information and location information.

- Time information: Binary representation of the partition number
- Location information: Binary representation of the value on the space-filling curve.

### 3.1.5 Example

The following is a small example for the computation of the index key for a given object. This could be a new one or one where the velocity vector has changed.

The initial setup is the following:

- $\Delta t_{mu} = 60$
- $n = 2$

This means that every partition contains objects of a time frame of 30 time units.

The example will look at two different objects:

- Object 1:  $O_1 = ((5, 6), (0, -0.1), 0)$
- Object 2:  $O_2 = ((3, 2), (0.1, -0.025), 20)$

The first step is the calculation of the partition number, where the objects belong:

- $O_1$ 
  - $t_{lab}^1 = (0 + 60 / 2) = 30$
  - $partition\ number^1 = (30 / (60 / 2) - 1) \bmod (2 + 1) = 0$
- $O_2$ 
  - $t_{lab}^2 = (20 + 60 / 2) = 60$  (50 rounded to the next label timestamp)
  - $partition\ number^2 = (60 / (60 / 2) - 1) \bmod (2 + 1) = 1$

In a second step one has to calculate the location of the object at the label time of the corresponding partition:

- $O_1$ : location at time  $t_{lab}^1 = (5, 3)$
- $O_2$ : location at time  $t_{lab}^2 = (7, 1)$

Next the value on the space-filling curve is calculated for the new locations. In this example the Hilbert-curve is used:

- $O_1$ : hilbert-value = 52
- $O_2$ : hilbert-value = 62

Finally the index key is a concatenation (+) of the binary representation of the partition number and the binary representation of the hilbert-value:

- $O_1$ : index key = 00 + 0110100
- $O_2$ : index key = 01 + 0111110

### 3.2 Insertion, Deletion, Updates

Insertion and deletion in the  $B^x$ -tree are almost the same as in the  $B^+$ -tree. For insertion one has to compute the index key before inserting it into the index. For deletion an additional data structure is needed. This data structure stores the index keys for every objects latest insertion. When deleting an object one can look up the key in this data structure and then delete it from the index like in the  $B^+$ -tree.

However, updating is a bit different in the  $B^x$ -tree. Objects are only updated when the parameters of their moving function changes. Updates are clustered over some time and then executed.

The time interval for updates must be chosen carefully. Setting it to the longest update interval occurring among the objects might be a bad choice if all the others have considerably shorter update intervals. Therefore, it is set to the longest interval among a certain percentage of the objects. If objects aren't updated in an interval they are transferred to a new partition with their positions recalculated for the new timestamp.

### 3.3 Queries

#### 3.3.1 Range Query

A range query returns all objects which are within a rectangle with some specific properties at some specific point in time. The  $B^x$ -tree allows range queries for the current and near-future.

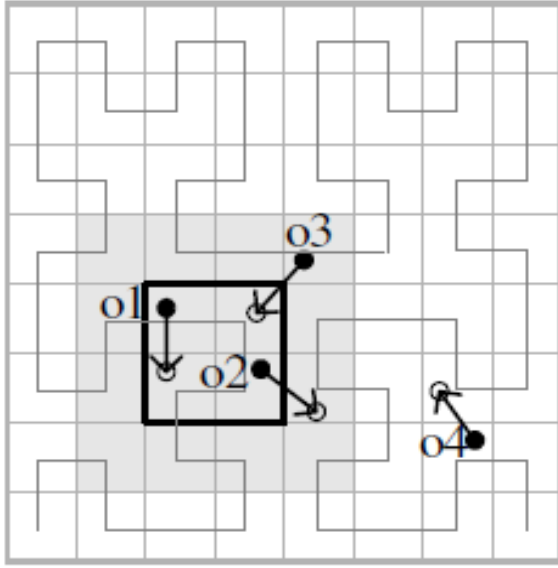
As objects are stored at the label time of their partitions, objects which are not in range at the label time when their position is stored could still move into the range by the time the query is executed. A new concept has to be introduced to account for this problem. The solution is query-window enlargement as shown in figure 5. The enlarged query-window must enclose all objects which fulfill one of the following properties:

- The objects speed would allow the object to move into range by the time of the execution of the query.
- The objects speed would have allowed the object to have moved out of the range in between query time and the label time when the object is stored in the index.

The easiest way to make this true is to move the bounds of the query region outwards with the maximum possible speed among all objects. One has to be aware that this results in many false positives, meaning objects which are in the enlarged query-window but are not in the actual result set of the query. Especially for dense data this approach is not ideal. An example for this is shown in figure 5. The correct result of the query should be objects o1 and o3. However, o3 is not in the original query region and therefore the query-window enlargement as described above is needed.

To find all the queried objects in the  $B^x$ -tree one has to traverse all the valid partitions at the time of the query. In each partition it might be necessary to execute several range queries. This is because of the linearization with space-filling curves. The relevant parts of the real space might be split up in several intervals in the one-dimensional space. This can also be seen in figure 5 in the shaded region. For efficiency reasons only the start and end points of these intervals are stored for traversal.

After retrieving all the possible objects a check is needed to filter out the ones that really are in the query window. For example one can look at o2 in figure 5. This object is in the enlarged query-window, but it is not in the query region at query time.



- Rectangle: query region
- Shaded: enlarged query-window
- Objects: o1, o2, o3, o4
- Filled circles: objects at reference time of phase
- Empty circles: objects at query time
- Arrows: velocity vectors

Figure 5: Query-window enlargement. [4]

### 3.3.2 $k$ NN Query

A  $k$ NN Query ( $k$  Nearest Neighbors Query) returns for a given object  $q$  at a given location  $k$  objects. For all these  $k$  objects it is true that there doesn't exist an object which is closer to  $q$  and not in the result set.

The  $k$ NN Query is basically a set of range queries as described in the last section. The range is enlarged as long as there aren't  $k$  objects in the result set.  $q$  is always at the center of the rectangle that represents the queried range. For the initial range an estimation is used. This estimation takes the total number of objects and the number  $k$  into account. If at least  $k$  objects are in the result set the nearest  $k$  objects are returned. Otherwise the range must be adjusted. All sides of the rectangle are moved away from the center. As a distance to extend with every step, the one that is calculated for the initial estimation can be used.

For every iteration of the query the objects that have not been part of the previous result set, but are now within the range are retrieved.

### 3.3.3 Continuous Query

Continuous queries have similar implementations as the queries explained in the last two sections. However, they are not executed only once. Their result sets are continuously maintained over time. Such queries run in a time frame from time of issue until deactivation. The query is recomputed at given time intervals and maintained correct for updates in between.

Maintaining a correct result set during updates is easy for a range query. An inserted object can either be in the range or outside. A deleted object is simply removed from the result set if it was there.

For a continuous  $k$ NN Query the situation is somewhat different. During insertion the new object is compared to the  $k$  objects in the result set. The result set is adapted if necessary. However, deletion is a bit different in case the deleted object was in the result set. The query has to be recomputed.

## 4. Performance Results

This section discusses the performance of two example operations on the  $B^x$ -tree. The first example is a range query which is a representative example for queries, as it is part of every query this report described. The second example is updates. They consist of deletion and insertion of an object, and are therefore a representative example for modifications.

### 4.1.1 Range Query

There are mainly two factors which are critical for the performance of a range query on the  $B^x$ -tree. The first one is the number of sections on the space-filling curve which have to be taken into account for the query. As mentioned in chapter 3.3.1 one has to execute a range query for every section on the space-filling curve which is in the enlarged query window. Switching from one section to the next when retrieving the objects in the result set includes backtracking in the  $B^x$ -tree and jumping (following the right-sibling pointers to the next section).

The second factor is the number of objects retrieved. The best case would be that only the objects that really belong to the result set are retrieved. However with the concept of query-window enlargement a number of false positives is in the retrieved set of objects. This can slow the query down.

In comparison with the TPR-tree the  $B^x$ -tree performs quite well. In [1] the authors did many performance tests comparing these two trees. One example is the variation of dataset sizes. For larger datasets the performance of TPR-tree degrades rapidly. Increasing the number of objects leads to higher probability of overlapping MBRs, for a query multiple paths have to be traversed. The  $B^x$ -tree is not really affected by large datasets. There the slightly worse performance is only due to larger result sets.

### 4.1.2 Updates

In this case an update corresponds to deletion of the object followed by an insertion of the object with new parameters. As described in chapter 3.2 there is an additional datastructure to store the index key of every objects latest insertion. This makes it possible to look up the key for deletion there and then delete it from the  $B^x$ -tree with the same performance as in the  $B^+$ -tree. Insertion is also done as efficiently as in the  $B^+$ -tree after calculating the index key.

When compared to the TPR-tree the  $B^x$ -tree shows the same advantages for updates as for range queries. As soon as the dataset size increases the TPR-tree slows down because of the overlapping MBRs and the need for multiple traversals of paths.

## 5. Conclusion

The  $B^x$ -tree as described in the previous chapters offers a way around the performance problems with updates that many other index structures have when it comes to indexing moving objects. It offers efficient operations for insertion and retrieval of objects as it is based on the  $B^+$ -tree. The  $B^x$ -tree is therefore a one-dimensional datastructure. However, with some adaption it is able to store multidimensional data.

At least for range queries and  $k$ NN queries concerning the current or near-future positions of the objects, the  $B^x$ -tree achieves good performance results and outperforms the TPR-tree.

There is another advantage to the  $B^x$ -tree. As it is based on the  $B^+$ -tree without changing the basic operations it can be built on top of existing database systems very easily and very cost effectively.

### 5.1.1 Outlook

The  $B^x$ -tree has some issues remaining. For example the parameters like  $\Delta t_{mu}$  or the initial range of the  $k$ NN query are only estimates which can in some setups be bad choices.

Two other problems are solved by later extensions to the  $B^x$ -tree:

- The size of the cells in the grid does not adapt to dense or sparse data.
- Query-window is mostly larger than needed.

The second problem with the size of the query-window is solved by the  $B_r^x$ -tree [4]. There only the relevant cells of the grid are added to the enlarged query-window.

The first problem which applies to special data distribution is solved by the  $ST^2B$ -tree [5]. It is based on the  $B^+$ -tree as well but it adapts very good to dense or sparse data as it chooses the size of the grid cells dynamically. For further explanation see the paper referenced in [5].

## References

- [1] C. S. Jensen, D. Lin and B. C. Ooi. Query and Update Efficient B<sup>+</sup>-tree Based Indexing of Moving Objects. In *VLDB*, pages 768-779, 2004.
- [2] S. Saltenis, 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.
- [3] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Searching. In *Proc. ACM SIGMOD*, pp. 47-57, 1984.
- [4] Jensen, C.S., D. Tiesyte, N. Tradisaukas. Robust B<sup>+</sup>-Tree-Based Indexing of Moving Objects. In *Proceedings of Seventh International Conference on Mobile Data Management*. 2006.
- [5] S. Chen, B. C. Ooi, K. Tan and M. A. Nascimento. ST<sup>2</sup>B-tree: A Self-Tunable Spatio-Temporal B<sup>+</sup>-tree Index for Moving Objects. *SIGMOD'08*, June 9–12, 2008, Vancouver, BC, Canada.