INDEXING FOR MOVING OBJECTS

Guo Shuqiao

Bachelor of Science
Fudan University, China

A THESIS SUBMITTED
FOR THE DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE

2005

Acknowledgement

I would like to take this opportunity to express my gratitude to all those who gave me the possibility to complete this thesis. First of all, I am so much grateful to my supervisors Prof. Ooi Beng Chin and Dr. Huang Zhiyong, for their guidance, encouragement and constant support. Their advice, insights and comments have helped me tremendously in all the time of research for and writing of this thesis in NUS. I would also like to thank Prof. Jagadish for his valuable suggestions and help during the research, and to thank Dr. Chan Chee Yong for his guidance and kindness as my mentor during my first semester in NUS. I sincerely wish to thank NUS and SoC for providing scholarship and facilities for my study.

Also, my acknowledgements go out to Lin Dan, Cui Bin, Dai Bingtian, Pavan Kumar B Sathyanarayan, Yao Zhen, Cao Xia, Song Yaxiao, Li Shuaicheng, Xiang Shili, Chen Chao, and all my colleagues in Database Group for their willing to help in my research. They have given me quite a lot happy hours. It is my pleasure to get to know all of them and working together with them. Special thanks go to Ni Yuan, Liu Chengliang, Huang Yicheng and Yu Jie for their great help in various ways. Their support and friendship make my life more enjoyable.

Foremost, I would like to express my deep appreciation to my family, especially my beloved parents. They always share my good and bad experiences, my gains and pains, my happiness and sadness. Their support, understanding, patience and love accompany me and encourage me whenever and wherever.

CONTENTS

LIST OF FIGURES

Summary

Rapid advancements in positioning systems such as GPS technology and wireless communications enable accurate tracking of continuously moving objects. This development poses new challenges to database technology since maintaining up-to-date information regarding the location of moving objects incurs an enormous amount of updates. Furthermore, some applications require high degree of concurrent operations, which introduces more difficulties for indexing technology. In this thesis, we shall examine a simple yet efficient technique in moving objects indexing.

Most of existing techniques for indexing moving objects depend on the use of a minimum bounding rectangle (MBR) in a multi-dimensional index structure such as the R-tree. The association of moving speeds with its MBR often causes large overlaps among MBRs. This problem becomes more severe as the number of concurrent operations increases due to lock contention. Thus, it cannot handle heavy update load and high degree concurrent update efficiently. We observe that due to the movement of objects and the need to support fast and frequent concurrent operations, MBR is a stumbling block to performance. To address the problem, we believe that indexes based on hash functions are good alternatives, since they are able to provide quickly update and do not suffer from the overlapping problem. However, region based retrieval must be supported. Consequently, we propose a "new", simple structure based on the Buddy-tree, named Buddy*-tree. The Buddy*-tree is a hierarchical structure without the notion of tight bounding spaces. In the proposed structure, a moving object is stored as a snapshot, which is composed of its position and velocity at a certain timestamp. The status of an indexed object is not changed unless there is an update for it. Instead of capturing speed in an MBR, we enlarge the query rectangle to handle future queries. To support concurrent operations efficiently we employ sibling pointers like the B-link-tree and R-link-tree in the Buddy*-tree. An extensive experimental study was conducted and the results show that our proposed structure outperforms existing structures such as the TPR*-tree and B^x-tree by a wide margin. To this end, we believe that our contributions have successfully addressed some of the issues of moving objects indexing techniques.

CHAPTER 1

Introduction

Database management system (DBMS) has become a standard tool to assist in maintaining and utilizing large collection of data. To facilitate efficient access to the data records, index structures are used. An *index* is a data structure that organizes data records on disk to optimize certain kinds of retrieval operations [45]. To index single-dimensional data, hash functions (e.g. [29] and [19]) and the B^+*-tree* [16] are widely recognised as the most efficient indexes.

During the last decade, *spatial databases* have become increasingly important in many application areas such as multimedia, medical imaging, CAD, geography, or molecular biology. Spatial databases contain multi-dimensional data or high-dimensional data which require much more sophisticate access methods. To support efficient retrieval in such databases, many indexes have been proposed ([20] and [8]).

With rapid advancements in positioning systems (e.g. GPS technology), sensing technologies, and wireless communications in recent years, *spatio-temporal databases* that manage large volumes of dynamic objects have attracted the attention of researchers.

In order to track accurately the movement of thousands of mobile objects in such databases, to develop techniques of efficient storage and retrieval of moving objects is an urgent need. In addition, some applications such as traffic control system and wireless communication also require the support for high concurrent operations. These requirements have posed new challenges to database technology. Indeed, this topic has received significant interest in recent years.

1.1 Motivation

Mobile objects move in (typically two or three-dimensional) space. As such, traditional index techniques for multi-dimensional data are a natural foundation upon which to devise an index for moving objects. Indeed, most index structures for moving objects are developed by making suitable modifications to appropriate multi-dimensional index structures.

A standard technique for indexing objects with spatial extent is to create a *minimum bounding rectangle* (MBR) around the object, and then to index the MBR rather than the object itself. Since most index structures cannot deal with the complexity of object shape, the MBR provides a simple, indexable representation at the cost of some (hopefully, not too many) false positives. Many multi-dimensional index structures, including in particular the *R-tree* [22] and its derivatives (e.g. [53] and [2]), follow such an approach.

Moving objects, even if they are modeled as points, are in different locations in space at different times. In an index valid over some period of time, if we wish to make sure to locate a moving object, we can do so by means of a bounding rectangle around the location of the object within this period of time. To handle the mobility of objects, most spatio-temporal indexes also have explicit notions of object velocity, and make linear, or more sophisticated, extrapolations on object position as a function of time. But an MBR is still required to make sure that a search query does not suffer a false dismissal. Among such techniques, the *TPR-tree* [49] is one of the most popular indexes. The TPR-tree (the Time Parameterized R-tree), an R-tree based structure, adopts the idea from [54] to model positions of the moving objects as functions of time with the velocities as parameters. While the use of linear rather than constant functions may reduce the need for updates by a factor of three [15], and provides query support for current and future queries, performance remains a problem. Various strategies have since been proposed to improve the performance of the TPR-tree such as [59].

Individual updates on the R-tree based structures, such as the TPR-tree, tend to be costly due to modification of MBRs and long duration splitting process of nodes. Frequent tree ascents caused by node splitting and propagation of MBR updates lead to costly lock conflicts. The concurrency control algorithms of the R-trees, such as the *R-link-tree* [32], are not able to adequately handle a high degree of concurrent accesses that involve updates. This causes us to question about the need of MBR in a highly mobile database, where moving objects change positions frequently. That is, can we do without the bounding rectangles?

Another problem of the TPR-tree is the use of enlarged MBRs by taking speed and the last update time into consideration during query processing. The enlarged MBRs can cause severe overlap between them – the degree of which is much more severe than the MBR overlapping problem in the R-tree. The problem lies in the fact that the information about velocity is embedded in the MBRs. Instead of embedding the velocity information with the MBR, can we capture it into the query?

In this thesis, we attempt to address these difficulties by redefining the problem of indexing mobile objects.

1.2 Objectives and Contributions

Our idea is that, instead of embedding the velocity information within the index, we attempt to capture it in the query. Now, instead of point objects ballooning into large MBRs, we will have point queries being turned into rectangular range queries. On the surface, this appears to make no difference in terms of performance – so one wonders why bother to make this equivalence transformation?

It turns out that the benefit we get is that we can now build much simpler indexes – we only need to consider static objects rather than mobile objects. Simpler multi-dimensional structures are essential to support high update loads. In particular, we propose a simple indexing structure based on the *Buddy-tree* [52] – the Buddy*-tree. The bounding rectangles in the internal nodes are not minimum, and are based on the pre-partitioned cells. They are different sizes, and the union of the lower level bounding spaces spans the bounding space of the parent.

To allow concurrent modifications, we adapt the concurrency control mechanism of the R-link-tree. Since the Buddy*-tree is a space partitioning-based method, it does not suffer from the high-update cost of the R-tree, and due to the decoupling of velocity information from bounding rectangles, it does not suffer from the overlap problem of the TPR-tree.

Our work makes the following contributions:

1. The proposed structure does not suffer from the MBR overlap problem and hence is able to support more efficient update and range queries for moving object;

2. Node entries only contain space information, and are relatively small, permitting a larger fanout and requiring less storage space than competing techniques. This also leads to better performance.

3. An extremely aggressive lock release policy can be applied to obtain high con-

currency, through the use of a secondary right link traversal process. Since high update rates are common for mobile objects, this high concurrency renders the Buddy*-tree even more attractive.

The contribution is not so much on the design of a new structure, but insights on simple and yet elegant solutions in solving the difficult problem of moving object indexing, which has received a great amount of attention lately.

The rest of this thesis will give a detailed description of the above contributions. Experimental studies were conducted, and the results show that the Buddy*-tree is much more efficient than the *TPR*-tree* [59], an improved variant of the TPR-tree, and the B$^+$-tree based *B^x-tree* [26].

1.3 Layout

The thesis is organized as follows.

- Chapter 2 surveys previous index techniques for single-demensional and multi-dimensional objects and moving objects, as well as techniques for concurrency control for index trees.
- Chapter 3 describes the structure and concurrency control of the Buddy*-tree.
- Chapter 4 introduces the operations and algorithms of the Buddy*-tree.
- Chapter 5 describes a careful experimental evaluation.
- We conclude our work in Chapter 6 with some final thoughts and a summary of our contributions. We also discuss some limitations and provide directions for future work.

CHAPTER 2

Preliminaries

In this chapter, we review some existing structures that are relevant to our work, and existing index structure concurrency control mechanisms that our concurrency control is based upon.

Since mobile objects move in (typically two or three-dimensional) space, traditional index techniques are a natural foundation upon which to devise an index for moving objects. Indeed, most index structures for moving objects have been developed by making suitable modifications to appropriate single-dimensional and multi-dimensional index structures. Therefore, in this chapter, we review some traditional indexing techniques first.

2.1 Single-dimensional Indexing Techniques

In this section, we introduce some popular indexes for single-dimensional data.

2.1.1 The B$^+$-tree

For disk-based databases, I/O accesses dominate the overall operational cost, hence, the main design goal for index structures is to reduce data page accesses. The widely used B$^+$-tree [16], a variant of the *B-tree* [1], requires as many node accesses as the number of levels to retrieve a data item. The B$^+$-tree (as shown in Figure 2.1) is a multi-way balanced and dynamic index tree in which the internal nodes direct the search and the leaf nodes contain the data entries. To facilitate range search efficiently, the leaf nodes are organized into a doubly linked list. The B$^+$-tree as a whole is dynamic and adaptive to data volume. It is robust and efficient.

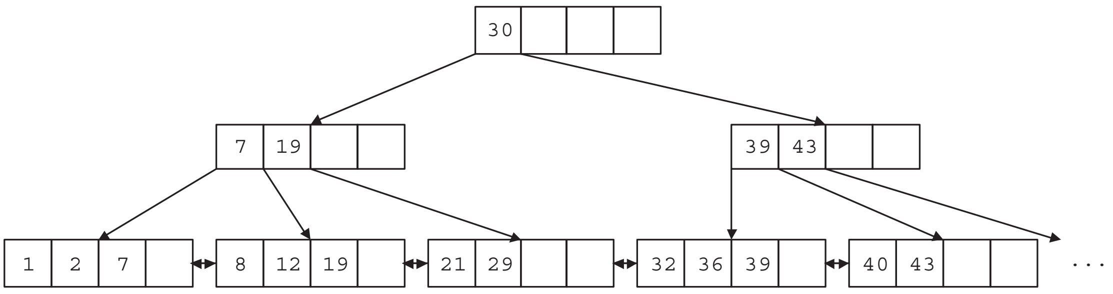


Figure 2.1: An Example of B$^+$-Tree

2.1.2 Hash Structures

The basic idea of hash-based indexing techniques is to use a hash function, which maps values in a search field into a range of bucket numbers. Random accesses on the hash structure are fast. However, the hash structure cannot support range searches. Further, skew distributions may cause collisions and cause the performance to degrade.

The *Extendible Hashing* [19], a dynamic hashing method, employs a directory to support dynamic growth and shrinkage of data volume and handle data skewness more effectively (see Figure 2.2). When overflow occurs, instead of chaining the overflow page or rehashes, it splits the bucket into two and double the directory to hold the new

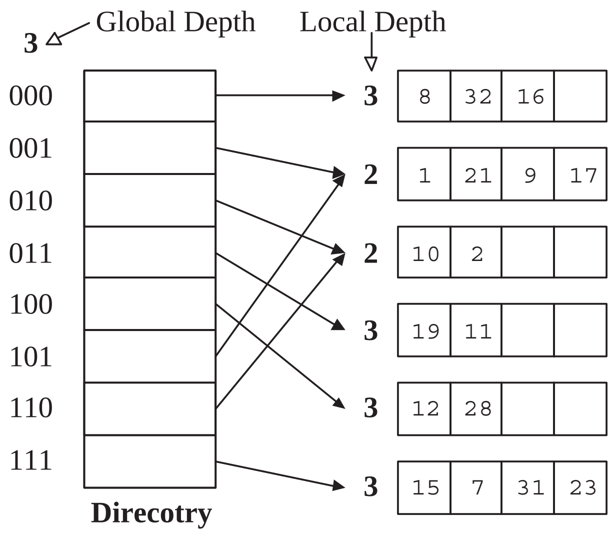


Figure 2.2: An Example of Extendible Hashing

bucket. Since the growth of the directory is always in power of two, it can be very large if the hash function is not sufficiently random. Fortunately, the directory size is not very large in terms of storage requirement.

The *Linear Hashing* [36] is another dynamic hashing technique, an alternative to Extendible Hashing (see Figure 2.3). It handles the problem of long overflow chains without directory. The dynamic hash table grows one slot at a time as it splits the nodes in predefined linear order. Since the buckets can be ordered sequentially, allowing the bucket address to be calculated from a base address, no directory is required. Overflow chain is allowed in Linear Hashing, thus, if the data distribution is very skewed, overflow chains could cause its performance to be worse than that of Extendible Hashing.

2.2 Multi-dimensional Index Techniques

Many multi-dimensional indexes have been proposed to support applications in spatial and scientific databases. In this section, we provide review on general multi-dimensional

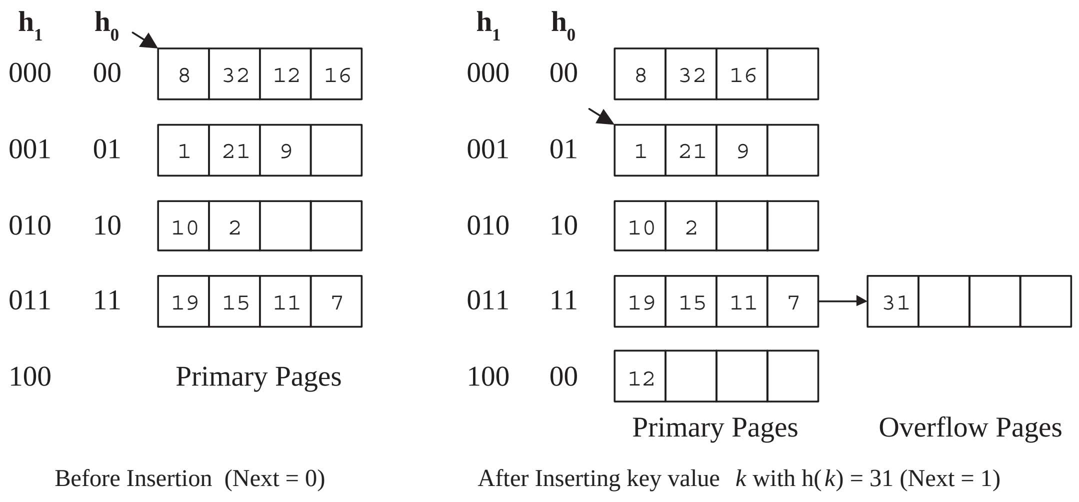


Figure 2.3: An Example of Linear Hashing

indexing.

Existing multi-dimensional index techniques can be traditionally classified into *Space Partitioning-Based* and *Data Partitioning-Based* index structure.

A Space Partitioning(SP)-Based approach recursively partitions a data space into disjoint subspaces. The subspaces (often referred to as regions, buckets) are accessed by means of a hierarchical structure (search tree) or some d-dimensional hash functions. Popular SP index structures include the *k-d-B-tree* [46], the *Grid File* [41], the R^+*-tree* [53], the *LSD-tree* [23], the *hB-tree* [38], the Buddy-tree [52], the *VAM k-d-tree*[56], the *VAMsplit R-tree* [62]), the *VP-tree* [11], the *MVP-tree* [9], etc.

A Data Partitioning(DP)-Based approach partitions the data into subpartitions based on proximity such that each subpartition can fit into a page. The hierarchical index is constructed based on space bounding, where the parent data space bounds the subspaces. As such, it is also known as *bounding region* (BR) approach. In such indexes, BRs may or may not overlap. In the case where BRs do not overlap, spatial objects have to clipped and stored in multiple leaf nodes. The R-tree [22] is one of the earliest Data Partitioning-Based indexes which all the other DP approaches are derived from. The shape of the bounding region can be rectangle (also referred as bounding box) (the R-tree, the *R*-tree* [2], the *TV-tree* [35], the *X-tree* [7]) or sphere (the *SS-tree* [63], the *SS*$^+$*-tree* [33]) and both of the two shapes (the *SR-tree* [28]).

Alternatively, we can classify the multi-dimensional index techniques into *Feature-Based* and *Metric-Based* techniques.

The feature based techniques split the space or partition the data based on the feature values along each independent dimension. The distance function used to compute the distance among the objects or between the objects and the query points is transparent to feature based techniques. In the SP-based index structures, feature based approaches include the k-d-B-tree, the R$^+$-tree, the LSD-tree, the hB-tree, the Buddy-tree, the VAM k-d-tree, the VAMsplit R-tree. In the DP-based index structures, feature based approaches include the R-tree, the R*-tree, the TV-tree, the X-tree.

The metric based techniques split the space or partition the data based on the distances from database objects to one or more suitably chosen pivot points. This technique is sensitive to the distance function. Popular distance based structures include the SS-tree, the VP-tree, the MVP-tree and the *M-tree* [14].

Hybrid approaches have also been proposed to combine the advantages of different techniques and improve the performance (the *Pyramid-tree* [6], the *Hybrid-tree* [10], the *IQ-tree* [5]).

Here we introduce and briefly discuss most popular index structures.

2.2.1 The Grid File

The Grid File is a multi-dimensional index structure based on extendible hashing. It employs a directory and a grid-like partition of the space. In each dimension, the Grid File uses $(d-1)$-dimensional hyperplanes parallel to the axis to divide the whole space into subspaces, called grid cells. The mapping from grid cells to data buckets is n-to-

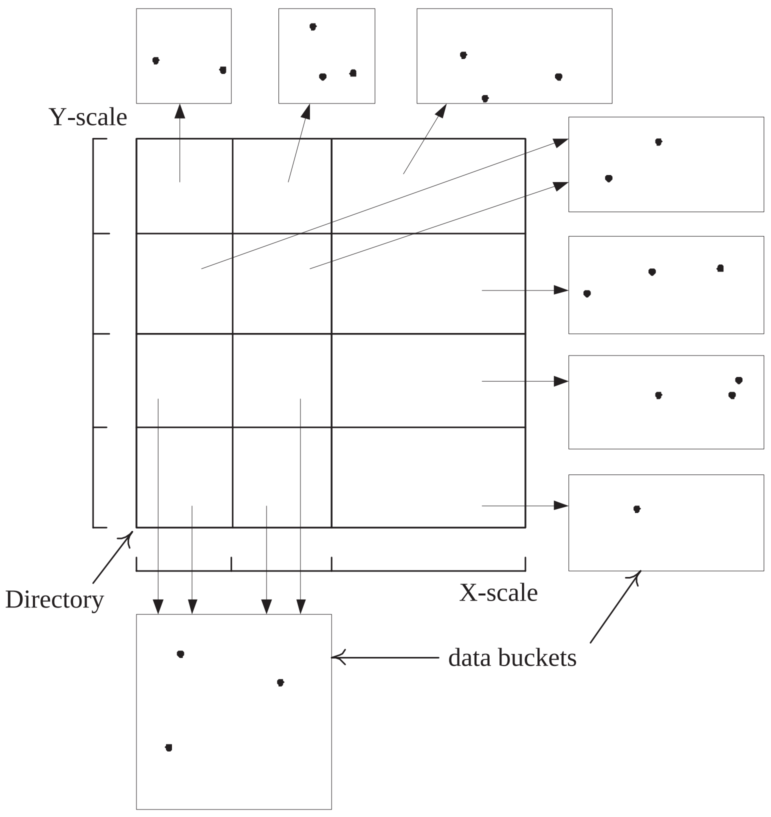


Figure 2.4: An Example of Grid File

1, that is to say, each grid cell is associated to only one data bucket, but one bucket may contain the regions of several adjacent buddy grid cells (see Figure 2.4). The bucket management system uses the data structure of d 1-dimensional arrays called linear scales to describe the partition in each dimension. Another structure is a d-dimensional array called directory. Each element in the directory is an entry to the corresponding data bucket. It is used to maintain the dynamic mapping between grid cells and data buckets. Linear scales are usually kept in the main memory, while the directory is kept on the disk due to its size.

The Grid File guarantees that a single match query can be answered with two disk accesses: one read on the directory to get the bucket pointer and the other read on the data bucket. For a range query, all grid cells which intersect the query region and their corresponding data buckets are inspected.

When a data bucket is overflowing and only one grid cell is associated to the bucket, a split of the grid cell occurs. Both grid cell and data bucket are split, and linear scales and directory are updated. If the Grid File maintains an equal-distant interval between each partitioning hyperplane in every dimension, there is no requirement to maintain linear scales. A simple hash function is used instead. In such case, a split of a grid cell is also a split of scale in this dimension, which will cause the directory to double in size.

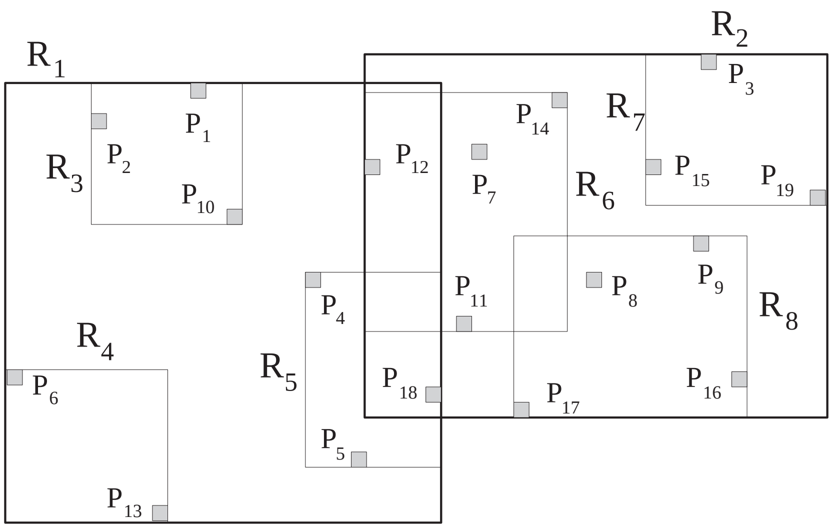
To reduce the split of directory and increase the space utilization some variances of Grid File (e.g. the *Two-Level Grid File* [24], the *Multilevel Grid File* [61] and the *Twin Grid File* [25]) have been proposed.

2.2.2 The R-Tree

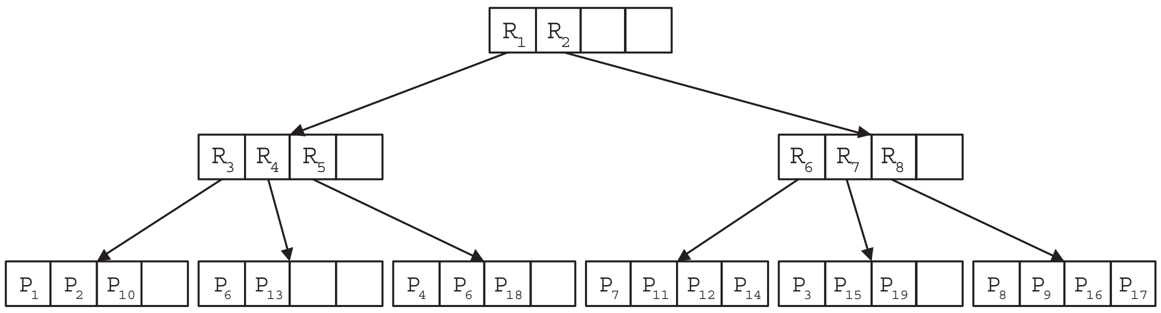
The R-Tree The R-tree is a multi-dimensional generalization of the B$^+$-tree, a dynamic, multi-way and balanced tree. As shown in Figure 2.5, in an R-tree leaf node, an entry consists of the pointer to the object and a d-dimensional bounding rectangle covering its data object. An entry in a non-leaf node contains a pointer to its child, a lower level node, and a bounding rectangle which covers all the rectangles in the child node. All the bounding rectangles are tight, so call *MBRs*, short for *minimal bounding rectangles*. The union of the MBRs on the same level may not be the whole space. Furthermore, there might be overlaps among the MBRs.

To do a range search, which is to retrieve all the objects that intersect a given query window, the algorithm descends the tree starting from root and recursively traverses down the subtree whose MBR intersects the query window. When a leaf node is reached, all the objects inside are examined and qualified ones for the query window are returned.

To insert an object, such a recursive process starting from the root is done until reaching a leaf node: choose a subtree whose MBR needs least enlargement to enclose the new



(a) A planar representation



(b) The R-tree

Figure 2.5: An Example of R-Tree

object. The new object then is added into the leaf node and the MBRs along the search path must be adjusted for the new object. If the node overflows, a split occurs.

The R*-Tree The R*-tree is a variant of the R-tree. The objective of the R*-tree is to reduce the area, margin and overlap of the directory rectangle. New insertion, split algorithms and forced reinsertion strategy are introduced. Contrary to the R-tree where only area is considered, overlap, margin and area are considered in the insertion algorithm of the R*-tree. The R*-tree outperforms the R-tree particularly if the data is non-uniformly distributed.

Other variants of the R-tree are proposed to overcome the problem of the overlapping covering rectangles of the internal nodes of the R-tree, including the R^+-tree, the Buddy-tree and the X-tree. The R^+-tree and the Buddy-tree avoid overlapping by employing SP method, and the objective of the X-tree is to reduce overlap for increasing dimensionality.

The Buddy-Tree The Buddy-tree is a dynamic hashing scheme with a tree-structured directory. It inherits the idea of MBR from the R-tree, however, it behaves as a SP-based structure. A Buddy-tree is constructed by cutting the space recursively into two subspaces of equal size with hyperplanes perpendicular to the axis of each dimension. The subspaces are recursively partitioned until the points inside one subspace fit within a single page on disk. Besides a space partition, each internal node in the Buddy-tree corresponds to an MBR, which is a minimal rectangle that covers all the points accessible by this node. Figure 2.6 gives an example of a 3-level Buddy-tree, where the space partitions are showed by plain rectangles and the MBRs by shadowed rectangles. As in all tree-based structures, the leaves point to the records of points on disk.

To insert a new point, the MBRs along the path from root to the target leaf node must be adjusted to guarantee that the new point is under cover. If a node is full, the space partition is halved and the MBRs are calculated for the two new partitions.

Since the Buddy-tree does not allow overlap among the space partition, the MBRs on the same tree level are mutually disjoint. Therefore, although the idea of MBRs is similar to R-tree, the Buddy-tree guarantees single-path search for insertions, deletions and exact match queries, contrary to the multi-path searching behavior in the R-tree. And compared to the k-d-B-tree, the Buddy-tree offers better performance for range query due to that the MBRs help to filtrate unqualified nodes. Additionally, the performance of the Buddy-tree is almost independent of the sequence of insertions, which is an essential drawback of previous tree-structures (such as the k-d-B-tree or the hB-tree).

One problem of the Buddy-tree is the relatively low fanout, since it maintains both

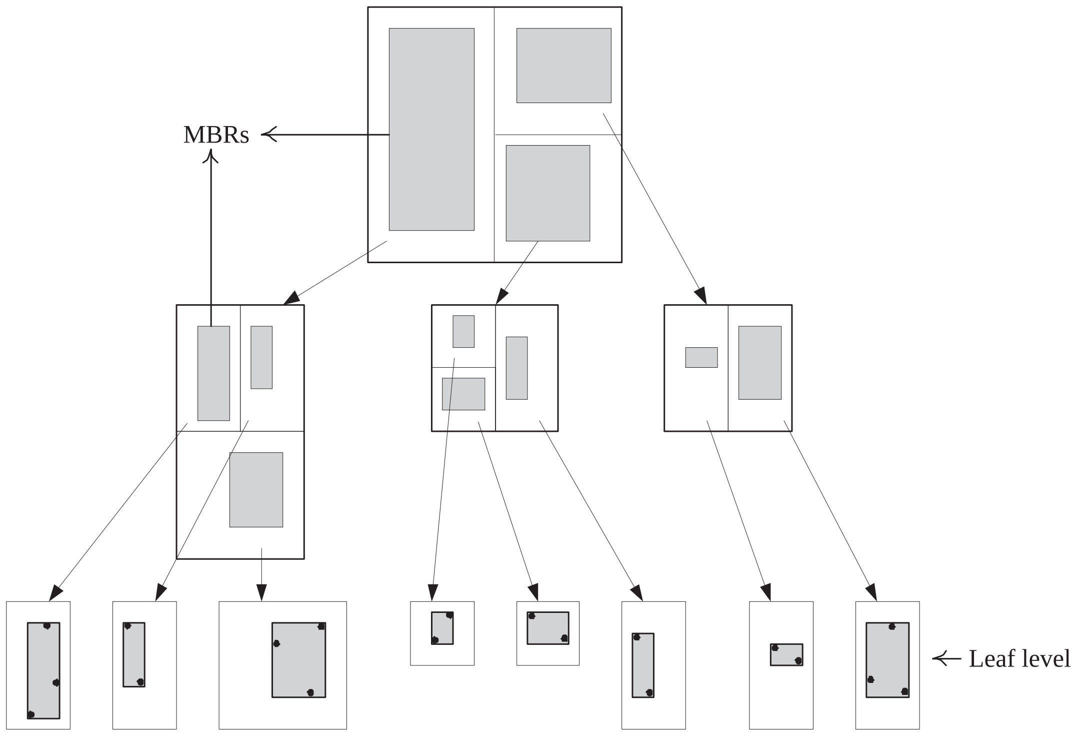


Figure 2.6: An Example of a 3-level Buddy-Tree

space partition and MBR in each entry. To solve this problem, a representation of the rectangles which is similar to that of the so-called hash-trees ([43], [44]) was suggested. That is, to employ two hash values (lower left and upper right corners), instead of two d-dimensional points, to represent a rectangle. Another disadvantage of the Buddy-tree is that although it does not suffer from the problem of forced splits, skewed data possibly introduces empty or nearly empty regions as well, since a subspace is always split at the median position.

The X-Tree The X-tree (eXtended node tree) is designed to solve the problem of high overlap and poor performance of R*-tree in high-dimensional databases by using larger fanout. The notion of supernode with variable size is introduced to keep the directory as flat as possible. Furthermore, the main objective of the insertion and split algorithm is to avoid those splits that would result in high overlap. The two concepts, supernode and overlap-free split, improve the performance of point query in the X-tree.

2.2.3 Use of Bounding Spheres

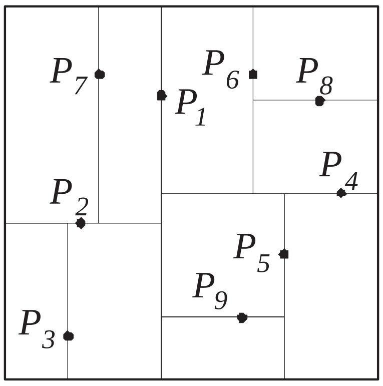
The SS-Tree The SS-tree is a distance-based variant of the R-tree. It uses d-dimensional spheres as BRs instead of bounding rectangles. In insertion algorithm, the choice of subtree is dependant on the distance between the new entry and the centroid of the node. The structure of the SS-tree enhance the performance of nearest neighbor queries, since on average the minimum distance of a query point from a bounding sphere is lower than that from a bounding rectangle. Furthermore, since the SS-tree stores only the centroid and radius for each entry in the node instead of the bounding rectangle, it only requires nearly half storage compared to the R^*-tree. Hence, it increases the fanout and reduces the height of the tree. The SS^+-tree is a variant of SS-tree, which uses k-means clustering algorithm as the split heuristic. An approximately smallest enclosing sphere is employed in the tree and it is a tighter bounding sphere than that of the SS-tree.

The SR-Tree The performance of bounding rectangles and bounding spheres are compared and analyzed in [28]. The conclusion is (1) Bounding rectangles divide points into smaller volume regions. However they tend to have longer diameters than bounding spheres, especially in high-dimensional space. Since the lengths of region diameters have more effects on the performance of nearest neighbor queries, SS-trees, which use bounding spheres for the region shape, outperforms the R^*-trees; (2) Bounding spheres divide points into short-diameter regions. However they tend to have larger volumes than bounding rectangles. Since large volumes tend to cause more overlap, bounding rectangles are advantageous in terms of volume. The SR-tree (sphere/rectangle-tree) [28] combines bounding spheres with bounding rectangles, as the properties are complementary to each other. The characteristic of SR-tree is that it partitions points into regions with small volumes (rectangles) and short diameters (spheres). Compared to the SS-tree, the SR-tree's smaller regions reduce overlap. Compared to the R*-tree, its shorter diameters enhance the performance of nearest neighbor queries. However, the SR-tree suffers from the fanout problem. Since it stores more information than the SS-tree and R*-tree do, the reduction of fanout may require more nodes to be read during query processing.

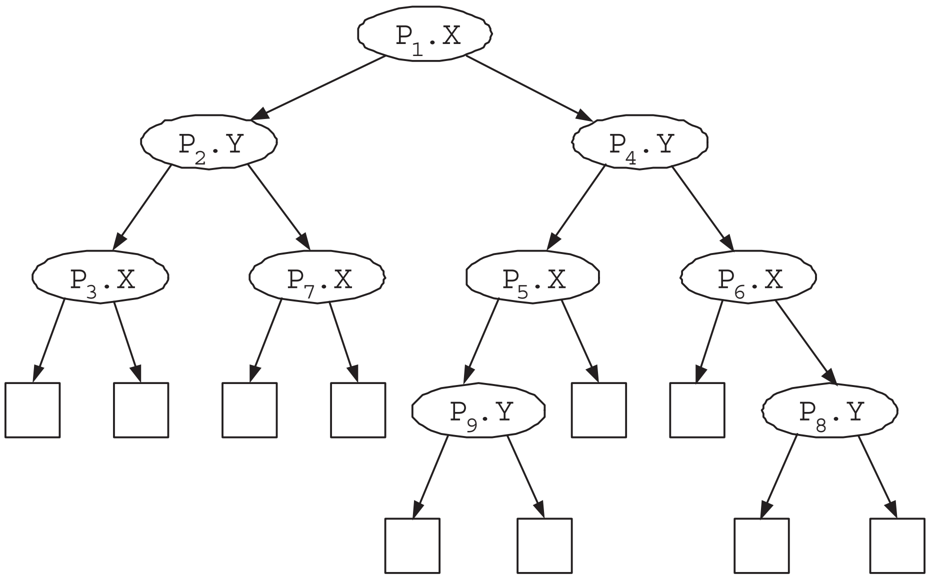
2.2.4 The k-d-Tree

The k-d-Tree The *k-d-tree* (k-dimensional tree) [3, 4], a main memory index structure, is a binary tree designed to index multi-dimensional data points. Most of SP-based hierarchical structures are derived from the k-d-tree. The k-d-tree is constructed by recursively partitioning point sets using hyperplanes that are perpendicular to one of the coordinate system axes. An internal node in the tree stores a data point and the dimension the data value is used to partition the data space. The child nodes, which contain the left and right (or up and down) subspaces of their parent respectively, are again partitioned using planes through a different dimension. An example of the k-d-tree is shown in Figure 2.7.

The k-d-B-Tree The k-d-B-tree is one of the earliest disk-based multi-dimensional index structures. It combines the properties of the adaptive k-d-tree and the B-tree, which we have introduced in the last section. Like a B-tree, the k-d-B-tree is a disk based and height-balanced tree. The structure is constructed by dividing the search space into subregions, which are represented by a k-d-tree (see Figure 2.8). B-tree like pages management is employed in the k-d-B-tree. If a node (a disk page) overflows, the tree chooses one dimension to split. In other words, a $(d-1)$-dimension hyperplane is chosen to split the space into two nonoverlapping subregions. It is noticeable that the subregions on the same tree level are mutually disjoint. The disjointness of the subspaces is also the



(a) A planar representation



(b) The k-d-tree

Figure 2.7: An Example of k-d-Tree

distinctive characteristics of all the SP-based approach. A split of the internal node may also affect the regions in the subtree, which have to be split by this hyperplane as well. Since such forced splits of the k-d-B-tree may cause empty or nearly empty nodes, it is not possible to have a lower bound on the occupancy node to guarantee the storage utilization. Furthermore, the high cost in cascading splitting is another problem, causing the tree to be sparse.

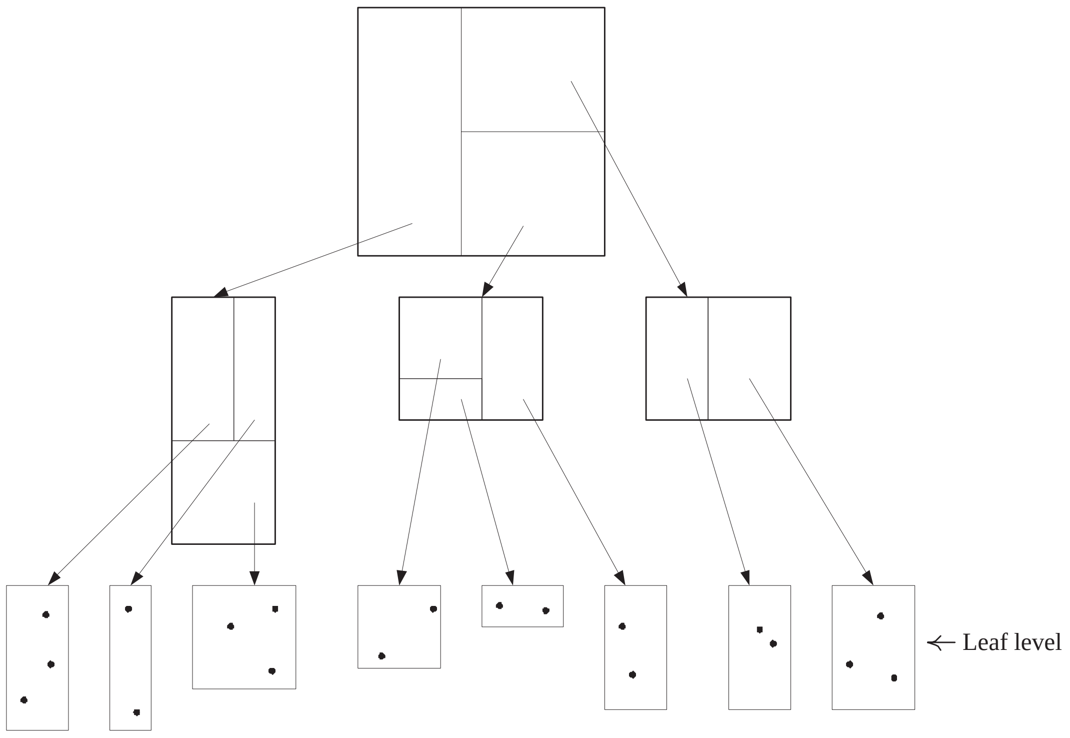


Figure 2.8: An Example of a 3-level k-d-B-Tree

The VAM k-d-Tree and VAMsplit R-tree The VAM k-d-tree (Variance, Approximately Median k-d-tree) is a refinement of the adaptive k-d-tree. It chooses the dimension with the largest variance to split instead of choosing the dimension with the greatest spread. The split position is approximately the median. The VAMsplit R-tree is derived from such an optimized k-d-tree. Since the VAMsplit R-tree provides more information such as upper and lower bounds on each dimension (characteristic as a R-tree) than the VAM k-d-tree, it reduces the I/O cost in searching.

2.2.5 Indexes for High-dimensional Databases

In the last subsection, we reviewed index techniques for multi-dimensional databases. These indexes have been designed primarily for low-dimensional databases, and hence most of them suffer from the '*dimensionality curse*'. In this subsection, we shall briefly review some existing works that have been designed or extended specifically for high-dimensional databases.

The TV-Tree The TV-tree (Telescopic-Vector tree), an R*-tree based index, is one of the first index structures for high-dimensional databases. The main idea is to reduce dimensionality based on important attributes. That is, the TV-tree telescopes active dimensions by activating a variable (typically small) number of dimensions for indexing. Since more entries can be stored in a node, the TV-tree reduces the effect of the 'dimensionality curse'.

The MVP-Tree The MVP-tree (Multi-Vantage Point-tree) is a distance-based indexing for high-dimensional space. It is an extension of the VP-tree, which partitions a data set according to the distance between the data and the reference (vantage) point, and uses median value of such distances as a separator to choose appropriate path for insertion. The MVP-tree extends the idea by introducing multiple vantage points. Another improvement is that the distances between parent nodes and child nodes are pre-computed in order to reduce the number of distance computations at query time.

The M-Tree In the M-tree the objects are indexed in metric space and the data structure is parametric on the distance function. The design of the M-tree is based on the principles of both metric tree and spatial access methods, which leads to the optimization of reducing both I/O cost (by using the R-tree like structure) and the number of distance computations (by exploiting the triangle inequality). The distance-based characteristic makes the approach appropriate for similarity range and nearest neighbor queries.

The Hybrid-Tree The Hybrid-tree is a feature based index. It mixes ideas from both DP-based and SP-based structures. Similar to the SP-based approaches, the Hybrid-tree always splits a node using a single dimension and stores the partition information inside the index nodes as the k-d-trees. Compared to the pure SP-based, the Hybrid tree keeps two split positions and the indexed subspaces need not be mutually disjoint. The tree operations (search, insertion and deletion) are performed like a DP-based index by treating the subspaces as BRs in a DP-based data structure.

The VA-File The *VA-File* (Vector Approximation File) [60] employs the compressing technique in indexing for high-dimension database. It is simple and yet efficient. The VA-File divides the data space into 2^b rectangular cells where b is a user specified number of bits. A unique bit-string of length b is allocated for each cell. And data points (vectors) that fall into a cell are approximated by the corresponding bit-string. Similarity queries are performed by scanning the VA-File, which keeps the array of compact bit-strings, to find the potential candidates (filtering step), and then accesses the vectors for further checking. In a very high-dimensional situation, the VA-File outperforms most tree structures since most hierarchical indexes suffer from the dimensionality curse and their performance deteriorate rapidly when the number of dimensions gets higher.

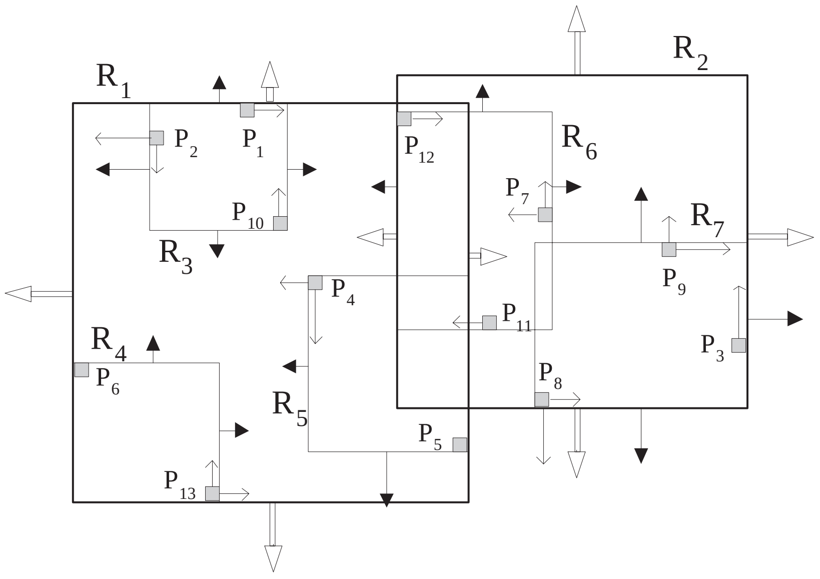
The A-Tree The *A-tree* [48] combines positive aspects of the VA-File and SR-tree by applying both partitioning and approximation techniques. The basic idea of the A-tree is to store a compressed representation of bounding boxes of child nodes in the inner nodes by using *virtual bounding rectangles* (VBRs) which contain and approximate BRs or data objects by quantization. Since VBRs can be represented rather compactly, the fanout of the tree is bigger and consequently the tree is able to achieve better performance than the VA-File and SR-tree (as shown in [48]). However, the effect is similar to that of the X-tree, and is only effective up to certain number of dimension. Further, this is good only for databases that are fairly static, since insertion and deletion may cause bounding regions to change and affect the relative addressing.

2.3 Index and Query of Moving Objects

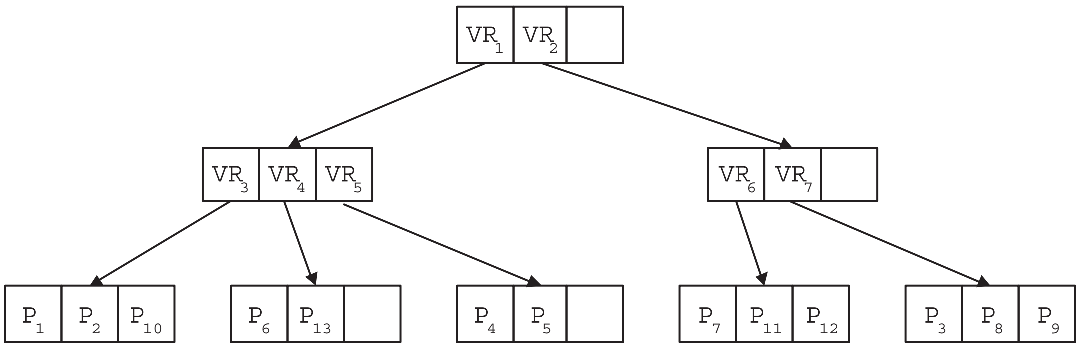
There is a long stream of research on the management and indexing of spatial and temporal data, which eventually led to the study of spatio-temporal data management. Since the traditional index techniques for multi-dimensional data such as the R-tree and its descendants cannot support heavy update efficiently and do not support queries on the future state of moving objects, several efficient spatio-temporal presentation and access methods [31, 57, 42] as well as approaches of querying for moving objects [30, 13] were proposed. All these approaches are based on the static index techniques we have discussed in the last two sections. In this section, we introduce several popular access methods and index structures for mobile objects.

MOST *MOST* [54] is one of the earliest spatio-temporal data models. It proposes to address the problem of representing moving objects in database systems by representing the position of moving objects as a function of time and the motion vector as an attribute. By treating time as one dimension, moving objects in d-dimension space can be indexed in $(d+1)$-dimension. Hence, near future state of an object can be queried. However this work did not propose any detailed access or processing method.

The TPR-Tree The TPR-tree (the Time Parameterized R-tree) [50] is an R-tree based index that has been designed to handle objects and predictive queries. The underlying idea of the TPR-tree is conceptually similar to MOST. Velocity vectors of objects or MBRs as well as the dynamic MBRs at current time are stored in the tree with the time as one attribute, as shown in Figure 2.9. At a non-leaf node, the velocity vector of the MBR is determined as the maximum value of velocities in each direction in the subtree and such velocity vector is called a *velocity bounding rectangle* (VBR). The VBR often causes the associated MBR to change its position; the different edge velocities will even



(a) A planar representation



(b) The TPR-tree (VR denotes the VBR and MBR at time t; P consists of position and velocity vector)

Figure 2.9: An Example of TPR-Tree

cause an object or an MBR to grow with time.

The query behavior of the TPR-tree is similar to that of the R-tree. To handle the near future query with query time t_q, when an MBR with time attribute t is examined for the query window, it is enlarged based on the VBR and the time distance between t and t_q. The algorithms of insertion and deletion for the TPR-tree are based on those of the R*-tree. The method of maintaining dynamic MBRs in the TPR-tree grantees that the MBRs always enclose the underlying objects or MBRs with time. However the dynamic MBRs are not necessarily tight. When an object is inserted or removed, the MBR of its parent node is tightened. But the other nodes that are not affected by the insertion or deletion are not adjusted.

The TPR-tree provides efficient support for querying of the current and future position of moving objects. However, it inherits the property of multi-path traversal of the R-tree, and the different edge velocities cause an object or an MBR to grow, resulting in more severe overlap, thus, degrades the performance.

[58] proposes a general framework for Time-Parameterized queries in spatio-temporal database based on the TPR-tree. The concept of "influence time" T_{INF} is introduced to compute the expiry time of the current result. By treating T_{INF} as the distance metric, some types of TP query (e.g. window query) can be reduced to nearest neighbor query, for which *branch-and-bound algorithm* [47] is employed.

The TPR*-Tree A performance study of the TPR-tree in [59] shows that the TPR-tree is far from being optimal by the means of the average number of node accesses for queries. Subsequently, the TPR*-tree was proposed to improve the TPR-tree by employing a new set of insertion and deletion algorithms.

In the insertion algorithm of the TPR*-tree, a QP (priority queue) is maintained to record the candidates paths which have been inspected. By visiting the descendant nodes, the TPR*-tree extends the paths in QP until that a global optimal solution is chosen, while the TPR-tree only chooses a local optimal path. In the node splitting algorithm, a set of worst objects whose removal benefits the parent node the most are removed and reinserted into the tree. These strategies improve the performance of the TPR-tree, however, additional I/O operations are incurred during updates, and since the core features of the TPR-tree, such as coupling of VBR to the MBR, remain. The query performance is achieved at the expense of costlier updates, which require the lock to be held for a longer period in concurrent operations, hence lock contention is expected to be more severe.

The B^x-Tree The B^x-tree [26] is a B$^+$-tree structure that makes use of transformation for indexing moving objects in a single-dimensional space. The main idea are linearization of the locations and vectors of moving objects using space-filling curve and indexing of transformed data points in a single B$^+$-tree. In the B^x-tree, the objects are partitioned based on time, but indexed in the same space. Insertions and deletions are straightforward and are similar to those of the B$^+$-tree. However, the index rolls on time based on the update interval to keep the index size stable. Range queries and predictive queries involve multiple traversal due to the partitioning on time. The B^x-tree is shown to be very efficient for range and kNN queries as it does not have the problem of enlarging MBRs over time. Further, it does not have the time consuming splitting problem. The concurrency control based on the *B-link-tree* [34] is adopted in the B^x-tree. However, unlike R-tree based indexes, the B^x-tree is not scalable in terms of dimensionality.

Other Structures Indexes based on hashing have been proposed to handle moving objects (e.g. [55] and [12]). In [55], the data space is partitioned into a set of small cells (subspaces). A moving object is stored in a corresponding cell based on its latest position. However, no detailed information such as exact position and velocity is stored. The database is updated only if an object moves to a new cell and asks for an update. To find the right cell for a certain object, a set of Location Pre-processing parts (LPs) is used. LPs work based on hashing functions, from which the cell that contains the target object can be found and accessed from the index. (In [55], the indexing method employed is Quad-tree Hashing. The space is organized as a quad-tree [51], in which each leaf node contains the objects inside the associated cell at current time. A node fits to a data page and splits if overflowing.) One challenge of this approach is that the LPs have to know the current structure of the index, which is dynamic. Another limitation is that the index only provides approximate locations for the indexed objects, hence it is not suitable for the applications that require exact locations or velocities of objects.

Some other novel indexes for moving objects have been proposed. However most methods are only suitable in particular environment. For example, Kalashnikov et.al. [27] proposed a new idea of indexing the continuous queries instead of indexing the moving objects to efficiently answer continuous queries based on the assumption that the queries are more stable compared to moving objects. The authors claimed that the query index may use any spatial index structure (e.g. the R-tree). However, this approach is specifically designed for continuous queries and is not suitable for other application.

Hybrid structures have also been proposed. For instance, in [17], hashing on the grid cells is used to manage hot moving objects in memory, while the TPR-tree is used to manage cold moving objects on disk, as a way to provide efficient support for frequent updates.

2.4 Concurrency in the B-Tree and R-Tree

In order to provide correct result for concurrent operations, earlier works on concurrency of the B^+-tree employ top-down lock-coupling. Lock-coupling implies that during descending the tree, the lock on the parent node can only be released after the lock on the child node is granted. Obviously, the update operations can be blocked by coupled read locks during tree ascent. Furthermore, if an update operation backing up the tree also employs lock-coupling, dead lock occurs.

The B-link-tree [34] was subsequently proposed to solve the problem. The structure of the B^+-tree is slightly modified to offer no block search for multiple searches and updates. In a B-link-tree, every node keeps a right link pointing to the right sibling node in the same level. On each level all the nodes buildup a right link chain and the nodes are ordered by their keys. In the modified structure, when a search process without lock-coupling goes down in the tree, it will not miss any splits, since it will aware of a split by comparing the keys and hereby visits the new split node along the right link chain before the new node is installed into the tree.

The R-link-tree [32] employs the similar modification for the R-tree. The main difference between the R-tree and the B-tree is that keys in the R-tree do not keep the order. Therefore, a structural addition LSN (logical sequence number) is introduced. A unique LSN within the tree is assigned to each node and an expected LSN is kept in each entry of the internal nodes. If a node is split, the new split out node is inserted into the right link chain and it holds the old node's LSN. The original node is assigned a new LSN which is higher than the old one. Before the new node installed, the expect LSN in the corresponding entry of the parent node is not updated. The split of a node can be detected by comparing the expect LSN taken from the entry in the parent node with the actual LSN in this node. If the latter is higher than the former, there is an uninstalled split. Travel along the right link chain, therefore, is necessary. The traversal is terminated if meeting a node with an LSN equal to the expect LSN. Another difference is that if the bounding rectangle in the leaf node is changed, we must propagate the change to its ancestor nodes. This process employs down-top lock-coupling.

The locking strategies of the B-link-tree and R-link-tree are deadlock-free since there's always only one lock in the B-link-tree, and the R-link-tree only employs lock-coupling in the down-top process.

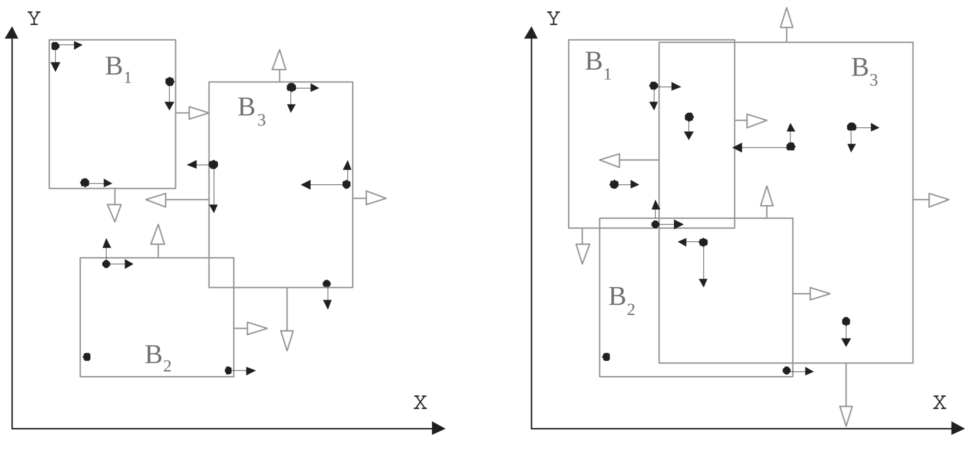
CHAPTER 3

The Buddy*-Tree

3.1 Motivation

A popular approach indexing spacial objects is to employ MBRs (such as the R-tree and its variants). In order to adapt such indexes for moving objects, VBRs (velocity bounding boxes) are stored with MBRs (such as the TPR-tree and TPR*-tree). One shortcoming of MBR-based index is that the overlaps among the nodes in the same level leads to possible multi-path search to retrieve an object. In the TPR-tree, due to the existence of VBRs, the MBRs keep enlarging as time progresses, and the overlapping problem becomes more and more severe. The TPR*-tree made some changes to alleviate the problem but, as an MBR/VBR-based index, it still suffers from the MBR overlapping problem.

Consider the example shown in Figure 3.1. This is a typical representation of moving objects using an MBR. The arrows denote the velocity of each object, broken up into components along the axes to obtain what are called velocity bounding rectangles (VBRs). The length of an arrow denotes the absolute value of velocity in the direction. Note that velocities are associated not just with the data objects, but also with the MBRs. MBR velocities are independently assigned to each boundary of the MBR, and is the maximum of the velocities in that direction in any of its included objects.



(a) MBRs (b) MBRs after one time unit

Figure 3.1: MBRs vs Speed

Suppose that all the MBRs and VBRs are tightened and each object is inserted into an optimized node (following the algorithm of the TPR*-tree), as in Figure 3.1 (a). One time unit later, the MBRs have expanded as shown in Figure 3.1 (b). At this time, the MBRs overlap each other, and do not tightly bound their constituent points any more. This problem becomes even more severe as time progresses since the overlapped area among MBRs B_1, B_2 and B_3 becomes increasingly larger.

Figure 3.2 shows the overlap ratios (the sum of area of all the MBRs / the area of union of all MBRs) at leaf level in a TPR*-tree with time elapsed. In this experiment, we use a uniform data set with 500K moving objects spreading in a 1000×1000 space, and the speed of objects are randomly chosen in range 0 to 3. There are no update operations in the experiment period. The overlap ratio increases quickly as time passes. In fact, we can make the following observation:

Let $x_i^l(0), x_i^u(0)$ be the lower bound and upper bound of some MBR respectively on dimension i at time 0, and $\vec{\boldsymbol{u}}_i^l, \vec{\boldsymbol{u}}_i^u$ be the minimum and maximum velocity of it on dimension i . After t time units, the volume of this MBR is $V = \prod_{i=1}^{d}(x_i^u(t) - x_i^l(t))$. Since $x_i^l(t) = x_i^l(0) + \vec{\boldsymbol{u}}_i^l \cdot t$ and $x_i^u(t) = x_i^u(0) + \vec{\boldsymbol{u}}_i^u \cdot t$, the volume of MBR can be

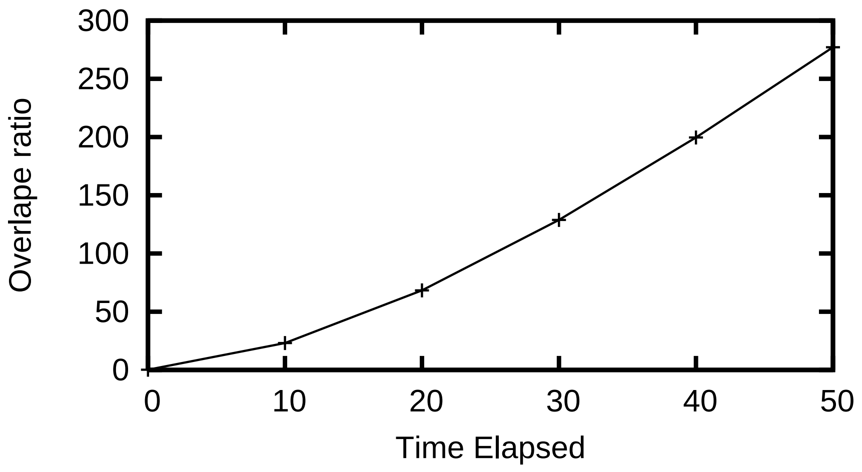


Figure 3.2: Overlap vs Time for Leaf Level MBRs

rewritten as $V = \prod_{i=1}^{d} [(x_i^u(0) - x_i^l(0)) + (\vec{\boldsymbol{u}}_i^u - \vec{\boldsymbol{u}}_i^l) \cdot t]$. Therefore,

$$\frac{\partial V}{\partial t} = \sum_{i=1}^{d} \{ (\vec{\boldsymbol{u}}_i^u - \vec{\boldsymbol{u}}_i^l) \cdot \prod_{i'=1, i' \neq i}^{d} [(x_i'^u - x_i'^l) + (\vec{\boldsymbol{u}}_i'^u - \vec{\boldsymbol{u}}_i'^l) \cdot t] \}$$

That is, $\frac{\partial V}{\partial t}$ is $O(t^{d-1})$.

The probability of any MBR being accessed by a random point search query, assuming uniform distributions, is proportional to the volume of the MBR. Therefore the expected number of MBRs accessed at any level of the index tree is proportional to the sum of their volumes. This leads to the following Lemma:

Lemma 1. *The rate of increase of the expected number of MBRs to be accessed at some level l is $O(t^{d-1})$, where t is the elapsed time and d is the dimensionality.*

As for concurrent operation, another disadvantage of MBRs for indexing moving objects is that an insertion in a leaf node even without split may involve several internal nodes, since a backing up process for modifying the MBRs or VBRs of it's ancestor nodes is necessary. In concurrent operations, locks on internal nodes affect the throughput a lot. Since update operations are quite frequent in moving objects database, the backing up process seriously reduces the performance.

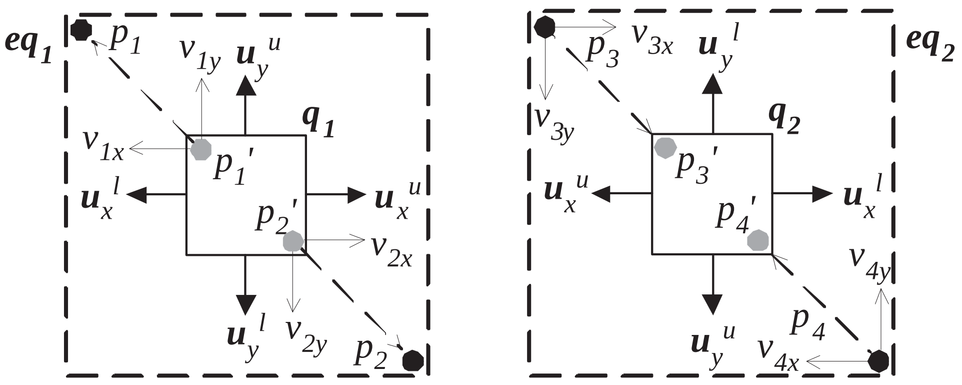
To overcome the challenges described above, we propose a complementary technique described in Section 3.2 for indexing mobile objects. We concretize these ideas into an index structure we call the Buddy*-tree, a variant of the well-known Buddy-tree, in Section 3.3. Issues of concurrency are important for good performance in an update-intensive environment, such as one would expect with moving objects. These issues are studied in the last part of this chapter.

3.2 Using Velocity for Query Expansion

Our central idea is that movement of objects can be handled by expanding queries rather than actually perturbing objects in the index. To know how much to expand a query by, we need to know what the velocities of the objects are, so these must also be stored. But all of this information can be stored as a static snapshot, taken at some time t_{ref}. We store, in the index, the velocities and positions of all objects at this reference time.

As in so many other moving object index structures, we use linear interpolation to estimate object position at times other than t_{ref}. The position of an object at time t can be calculated by the function $\boldsymbol{x}(t) = \boldsymbol{x}(t_{ref}) + \vec{\boldsymbol{v}} \times (t - t_{ref})$.

Since we index the objects at a reference time which is some time after current time, the enlargement of query window involves two cases: (1) if query time t_q is before t_{ref}, the location must be brought back to an earlier time (as shown in Figure 3.3 (a)); (2) otherwise, the location must be forwarded to a later time (as shown in Figure 3.3 (b)). Based on this, we can suitably enlarge a query as follows: Suppose the query is q with query window [qx_i^l, qx_i^u] ($i = 0, 1, ...d - 1$,where d is dimension of the space), and the query time is t_q, the enlarged query window [eqx_i^l, eqx_i^u] is obtained as ([26]):



(a) $t_q < t_{ref}$ (b) $t_q \geq t_{ref}$

Figure 3.3: Two Cases of Query Window Enlargement (p_i is the indexed location at t_{ref} of objects and p_i' is the actual location at t_q)

$$eqx_i^l = \begin{cases} qx_i^l + \vec{\boldsymbol{u}}_i^l \cdot (t_{ref} - t_q) & if t_q < t_{ref} \\ qx_i^l + (-\vec{\boldsymbol{u}}_i^u) \cdot (t_q - t_{ref}) & otherwise \end{cases}$$

$$eqx_i^u = \begin{cases} qx_i^u + \vec{\boldsymbol{u}}_i^u \cdot (t_{ref} - t_q) & if t_q < t_{ref} \\ qx_i^u + (-\vec{\boldsymbol{u}}_i^l) \cdot (t_q - t_{ref}) & otherwise \end{cases}$$

where $\boldsymbol{u}_i^l$ and $\boldsymbol{u}_i^u$ are the minimum and maximum velocities respectively of objects inside the query window in dimension i. Note that we would ideally have liked to enlarge the query by precisely the velocities of the objects included in the query. But this raises a chicken and egg problem, since the whole purpose is to determine which objects are in the query. We get around this problem by separately keeping track of the minimum and maximum velocities in each region.

Theorem 1. *Enlargement of query window provides the correct answer.*

Proof. Suppose q is an original query with query time t_q and query window R = [qx_i^l,

qx_i^u] ($i = 0, 1, ...d-1$), let S_q be the set of points returned by the original query. We also denote S_{ref} as the set of points returned by query q' with query time t_{ref} and enlarged query window R' calculated by the above formula. We wish to show that $S_q = S_{ref}$.

For any point $\boldsymbol{x} \in S_q$, let $\boldsymbol{x}(t_q)$ and $\boldsymbol{x}(t_{ref})$ be its positions at time t_q and t_{ref} respectively, we show $\boldsymbol{x}(t_{ref})$ is returned by our algorithm.

First, we suppose that $t_{ref} > t_q$. $\boldsymbol{x}(t_q)$ and $\boldsymbol{x}(t_{ref})$ are related by $\boldsymbol{x}(t_{ref}) = \boldsymbol{x}(t_q) + \vec{\boldsymbol{v}} \cdot \Delta t$, where $\Delta t = t_{ref} - t_q > 0$. If we can prove $\boldsymbol{x}(t_{ref}) \in R'$, then enlarged query q' will return $\boldsymbol{x}$.

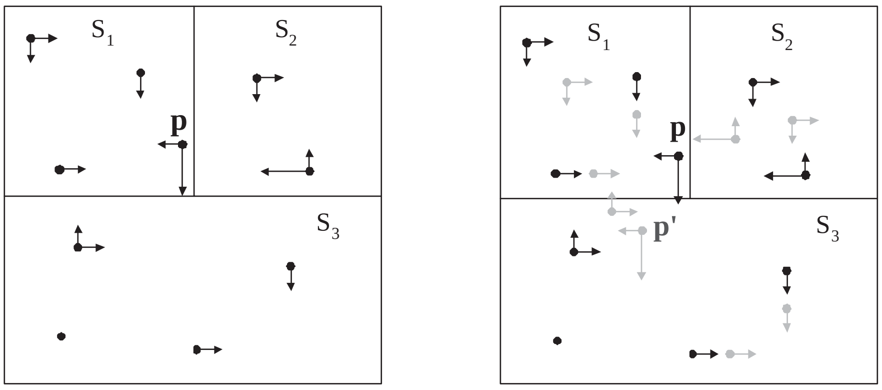
We use the same representation as the above formula for R, R' and the minimum and maximum velocities of objects inside R. Notice that $\boldsymbol{x} \in S_q$ and $\vec{\boldsymbol{v}}$ is velocity of $\boldsymbol{x}$, thus $\vec{\boldsymbol{u}}_i^l \leq \vec{\boldsymbol{v}}_i \leq \vec{\boldsymbol{u}}_i^u$ and $qx_i^l \leq \boldsymbol{x}(t_q)_i \leq qx_i^u$. $\Delta t > 0$ ($i = 0, 1, ...d-1$). Therefore, $qx_i^l + \vec{\boldsymbol{u}}_i^l \cdot \Delta t \leq \boldsymbol{x}(t_q)_i + \vec{\boldsymbol{v}}_i \cdot \Delta t \leq qx_i^u + \vec{\boldsymbol{u}}_i^u \cdot \Delta t$, i.e. $eqx_i^l \leq \boldsymbol{x}(t_{ref})_i \leq eqx_i^u$, hence we have $\boldsymbol{x}(t_{ref}) \in R'$. We can prove that $\boldsymbol{x}(t_{ref}) \in R'$ when $t_{ref} \leq t_q$ similarly. Hence, $S_q \subseteq S_{ref}$.

For any point $\boldsymbol{x} \in S_{ref}$ we have $\boldsymbol{x} \in S_q$, since every candidate point will be examined and unqualified ones will be removed. That is $S_{ref} \subseteq S_q$

Therefore, $S_q = S_{ref}$, i.e. enlargement of query window provides the correct answer.

□

Our overall index structure is thus to create a number of snapshot indexes, each at a selected reference time point. Queries with respect to times that lie between these reference points are resolved by using the closest time reference and extrapolating linearly using the formulae above. This is illustrated in Figure 3.4. Figure 3.4 (a) shows the snapshots of the objects at reference time t_{ref} and also illustrates the contents of the three subspaces in the index. Figure 3.4 (b) shows the status of objects and index one time unit after t_{ref}, where the dark points denote the snapshots and the light points denote the real locations of objects at this time. Consider the object p in Figure 3.4 (a), which is indexed



(a) Snapshots at time t_{ref} (b) Snapshots at time $t_{ref} + 1$

Figure 3.4: Indexing Moving Objects with Snapshots

in the node with space S_1 based on its location at time t_{ref}. Although after one time unit, p's actual position is inside space S_3 (p' in Figure 3.4 (b)), it is still stored in S_1, since we are only interested in the positions of objects at time t_{ref}.

With the passage of time, we use different reference time points to index objects, which are called *timestamps*, denoted as t_{l_1}, t_{l_2}, t_{l_3},$\ldots$ We let the timestamps always be the medial time in a whole update interval. If the maximal update interval of the index is T_{ui}, and index is constructed at time 0, then, the first t_l is $T_{ui}/2$, second is $T_{ui} \times 3/2$ and so on. Given an object whose updated time is t_u, we will index it at the $(\lfloor t_u/T_{ui} \rfloor + 1)$th timestamp. For example, if $T_{ui} = 120$, $t_u = 130$, since $\lfloor t_u/T_{ui} \rfloor + 1 = 2$, we will index this object using the second timestamp, that is $t_l = T_{ui} \times 3/2$ = 180. After determining the indexed timestamp, we can calculate the object's position at t_l according to its position and velocity at t_u. The position at t_l and velocity compose the snapshot, which is used to insert the object into the index. After every T_{ui} time, the old t_l expires and all new incoming objects are inserted using the new t_l. In some operations (such as deletion), we might have the old status of an object and want to find it in the index. For example, given an object p, whose updated time is $t_u(p)$ and the location and velocity at time $t_u(p)$ are also available, to retrieve it from the index, we first use the above function to calculate the timestamp t_l for $t_u(p)$, followed by computing the snapshot that we have used to index p, and finally, search the index for such snapshot. The search algorithm is to be introduced in Chapter 4.

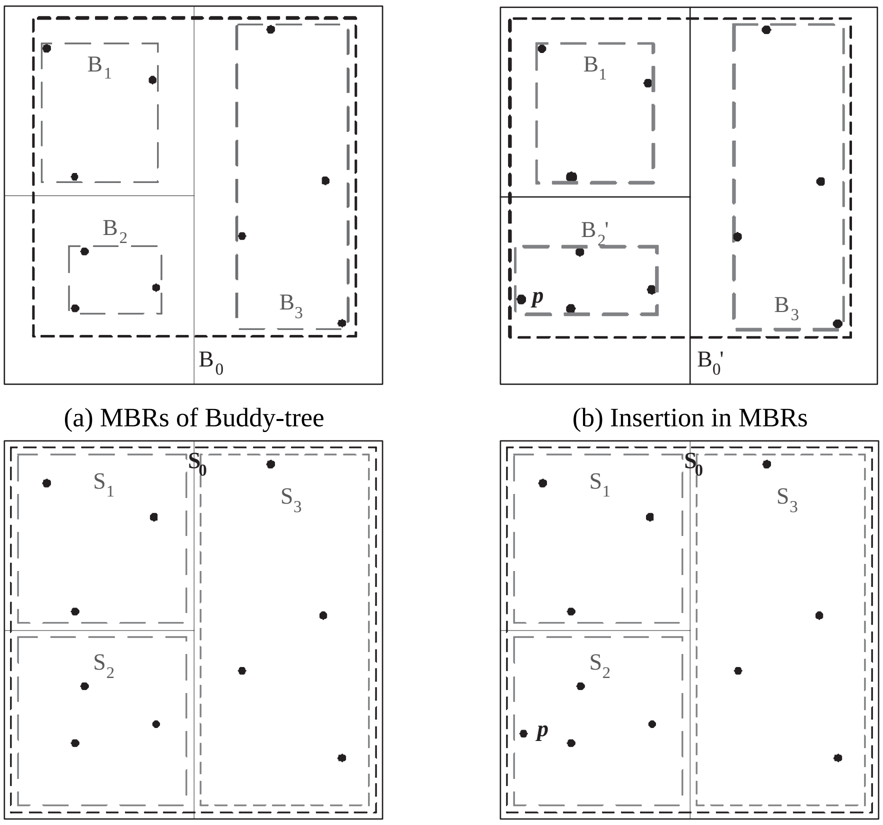
3.3 Structure of Buddy*-Tree

Given that we have a set of static points to index in each snapshot, and given the importance of fast update, we choose the Buddy-tree [52] as the basic structure of our proposed index. The index tree is constructed by cutting the space recursively into two subspaces of equal size with hyperplanes perpendicular to the axis of each dimension. Each subspace is recursively partitioned until the points in the subspace fit within a single page on disk.

We make several alterations to this basic Buddy-tree structure to suit our needs. We call the new index structure a Buddy*-tree. A traditional Buddy-tree creates tight bounding rectangles around the data points in each node, as shown in Figure 3.5 (a). Although the MBRs help in the efficiency in query operation, one disadvantage is that insertion and deletion of an object (no node splitting or merging occurs) probably changes the MBR of the located leaf node. Furthermore, if the MBR of a child node is changed, the parent node should be visited to adjust its MBR. Consider such an example in Figure 3.5 (b). After object p is inserted into the leaf node with MBR B_2, B_2 is enlarged to B_2' and, as a result, the MBR of the parent node B_0 is enlarged to B_0' as well. It is same to delete operations. Therefore, backing up the tree is a potential part of an insert or delete process. It is notable that a backing up process costs a lot in high degree concurrency, since visiting and locking an internal node is very likely to block other threads' operation and, hence, reduces the degree of concurrency. Since such tight MBRs are costly to update, we choose instead to use loose bounding. Specifically, we partition space, and use the entire space partition as the bounding rectangle for indexing purposes (see Figure 3.5 (c)), thus completely avoiding the need for bounding rectangle update (as illustrated in Figure 3.5 (d), an insertion of object does not make any change to the bounding spaces in the tree), at the cost of having some bounding rectangles be unnecessarily large (and hence require needless access at search time). We call this a *Loose Bounding Space* (LBS) associated with the index tree node. Although the MBR outperforms the LBS in query operation, to achieve efficient update and high degree concurrency LBS is a better choice. Furthermore, since the LBS is same as the space partition, for the Buddy*-tree there is no need to maintain additional information in the node entries. Therefore, we can gain a higher fanout which benefits the performance by reducing node accesses. Additionally, another reason to support this choice is for concurrency control purpose, which is discussed later.

To know how much to expand a query rectangle by, we need knowledge of the minimum and maximum velocities in each node. A naive method is using the global maximum speed to enlarge the query window. However this method might introduce unnecessary node access. We improve it by maintaining a list of local maximum velocities for all the index nodes. This information is computed for each node and then stored in the main memory at our first visit of that node. In the search process for a range query, when we visit a node, and need to determine which children of this node to visit, we thus have available to us not just the bounding rectangles for each child, but also the extremal velocities of objects in it.

To support high degree concurrent operations on Buddy*-tree, we absorb the idea of right links among each level from B-link-tree [34] and R-link-tree [32]. Thus, at any given level all nodes are chained into a singly-linked list. The Buddy*-tree, like the R-tree, is a multi-dimensional index structure, and hence does not have a natural ordering of keys at each level available in the case of a B-tree. To solve this problem,



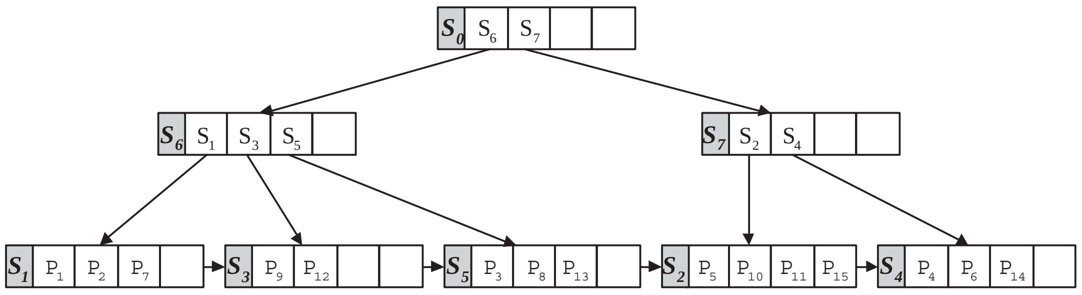
(a) MBRs of Buddy-tree (b) Insertion in MBRs

(c) LBSs of Buddy*-tree (d) Insertion in LBSs

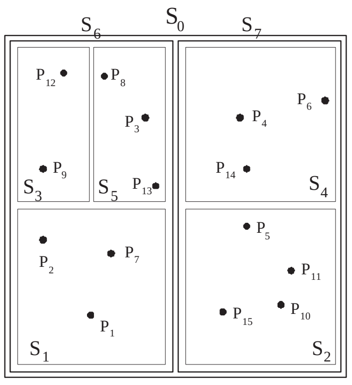
Figure 3.5: The difference of bounding methods between Buddy-Tree and Buddy*-Tree

[32] assigned an additional parameter LSN as the timestamp to each node and recorded the expected LSNs of the child nodes in each entry. The LSN is used to detect the split and determine where to stop when moving right along the right link chain. However, this structural addition is not required in the Buddy*-tree since we are guaranteed not to have overlaps between nodes. Instead, any lexicographic ordering of keys, constructed by following the path from root to leaf, will suffice. We can then rely upon these sideways links to delay the upward propagation of node splits, thereby allowing update operations to give up locks on ancestor nodes quickly rather than having to retain them against the possibility of a node split.

Figure 3.6 (a) shows an example of the Buddy*-tree in 2-dimensional space, with the



(a) The Buddy*-tree

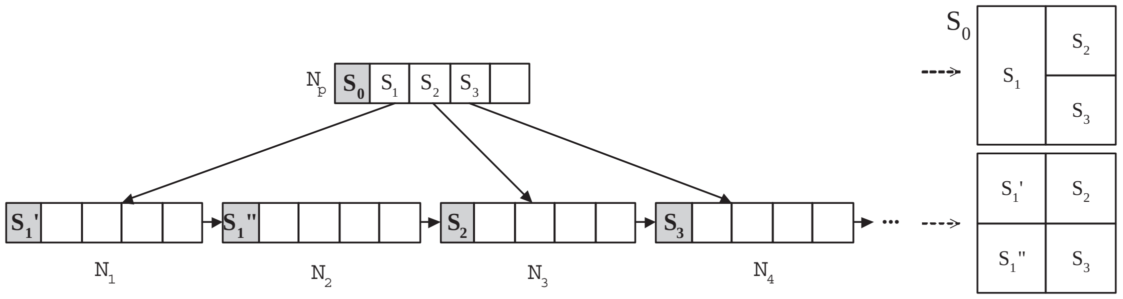


(b) A planar representation

Figure 3.6: An Example of the Structure of Buddy*-Tree

corresponding data space illustrated in Figure 3.6 (b). The first capital letter in a node denotes the LBS of it, followed by the entries with key LBS (expected LBS for the child node) or points.

Figure 3.7 shows an example in a Buddy*-tree fragment of how to detect an uninstalled split. Consider the second entry in the parent node N_p in figure 3.7(a). It points to node N_3, where the key LBS in the entry (S_2) is same as the actual LBS found in N_3. This is the normal case. However, there can be another case due to delayed propagation of node splits. Consider the first entry with key LBS S_1, which points to node N_1 with LBS S_1', where S_1' is a subspace of S_1. N_1 has a right link to node N_2, which also has LBS that is a subspace of S_1. Continuing farther along right links, N_3 is the first node that does not overlap S_1. This stops the right link traversal. All the nodes encountered in the right traversal, up to and excluding the last node, are covered by a single entry in the



(a) A Buddy*-tree fragment (b) Space partition

Figure 3.7: An Example of Uninstalled Split in Buddy*-Tree

parent, and must all be explored during a search, until such time as the split is installed at the parent node, with separate entries for each child.

A Buddy-tree may not be balanced due to the property that each directory node in it contains at least two entries. However, the Buddy*-tree is height balanced since it omits the more-than-one-entry property. Due to the frequent movement of objects strict adherence to certain "fill-factor" is not only quite redundant, it is not cost effective and it will unnecessarily slow down the concurrent operations. From the observation we made at the experimental study, leaf nodes of the Buddy*-tree are about 63%-76% filled, while internal nodes are 35%-67% filled. The occupancy rate is comparable to multi-dimensional indexes such as the R-tree.

3.4 Locking Protocols

The top nodes in an index structure can become hot-spots for concurrency as each of multiple processes need to access these en route to various leaf nodes and data. Locking protocols for tree indexes have been studied extensively. We use the following variation of a tree protocol for concurrency:

1. **Top-down**: In traversal of the tree top-down, only one lock is required at a time, that is, we release the lock of parent before we lock the child. For example, in

Figure 3.8 we lock node N_P and obtain pointer to N_1. Then we release lock of N_P and request lock for N_1. This is more aggressive than typical tree locking, which will continue to hold the lock on N_P until it receives lock on N_1. The worrisome case due to such an early lock release on our part is that a user gets the pointer to node N_1 from its parent N_P but N_1 is subsequently updated (and split) by another inserter before the user gets access to it. In this case, the user will subsequently detect the update by comparing the LBS in the entry of parent level with LBS in N_1. For instance, if a split has occurred, it will be found as discussed above in the example shown in Figure 3.7.

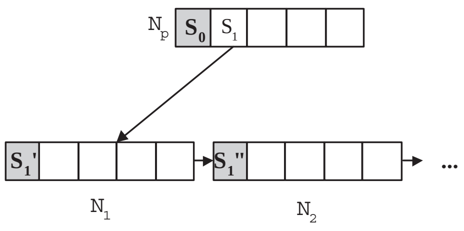


Figure 3.8: An Example of Lock Protocol

2. **Left-to-right**: In traversal of the right link chain, only one lock is required at a time. The example in Figure 3.8 is that we get the right link from the left node N_1 and release the lock before we apply lock for the right node N_2. The reason for correctness is similar to case (1). Right link chain can help to solve the problem if any other thread overtakes us and splits the node before we reach it. Entries in the original node can only move right, hence, when we keep looking right it is impossible for us to miss the object or the entry even if we hold only one lock at any time.

3. **Bottom-up**: To install changes (node splits), we have to move back up the tree. In this backing up process, we employ lock-coupling. For example, in Figure 3.8 if we have updated node N_1 and should back up the tree from it, we must hold the

write-lock of child node N_1 until we obtain write-lock of the parent N_P. Lock-coupling avoids the situation that another inserter causing a split overtakes us and installs the split before us, and finally we install our changes without being aware of the other inserter.

Due to the use of right links, we have in effect rendered node splitting (and merging) atomic at each level. In conjunction with the locking described above, it is easy to see that the tree will appear consistent at all times to any user. Furthermore, the locking protocols are deadlock-free. This is because only one lock is held at a time except during back-up. But during back-up, the process only moves upwards: after locking a node it never seeks to lock a node below it. Due to this ordering, we are guaranteed to be deadlock-free.

3.5 Consistency and Recovery

The highest degree of transactional isolation is defined as Degree 3 consistency [1]or repeatable read (*RR*) isolation [2][21]. This is a common requirement for concurrent access in database systems. A simple solution would be to lock all involved leaf nodes, that is to lock the leaf nodes in which several entries are returned by the search for the duration of the entire transaction. This is not sufficient due to the phantom insertions [18].

The phantom problem can occur with a tree locking protocol on an index, including the specific protocol we described above. To solve the phantom problem, the B-link-tree employs key-range locking (key-value locking) [39] and R-link-tree uses a simplified form of predicate locks [18]. Notice that the Buddy*-tree is a space-partition based index, there is no gap among the subspaces covered by all leaf nodes, that is, the whole

[1]Transaction T is degree 3 consistency if (1) T does not overwrite dirty data of other transactions; (2)T does not commit any writes before EOT (end of the transaction); (3) T does not read dirty data of other transactions; (4) Other transactions do not dirty any data read by T before T completes.

[2]Repeatable read implies that if a search operation is run twice within the same transaction it must return the exact same result (even if that result set is empty).

space is covered by leaf nodes. Therefore, we are able to employ a simpler solution: we simply retain locks on leaf nodes until the end of transaction, thereby ensuring that no one can modify these nodes. Since we use LBS that completely cover pace partitions, this is sufficient to guarantee repeatable reads.

Consider the example using MBRs in Figure 3.9. At the beginning, the query range R intersects leaf MBR B_1 and B_3 (Figure 3.9 (a)). Later, the insertion of object p enlarges B_2 (Figure 3.9 (b)) and causes it to intersect R so that a repeat search of R will find p in the result. Avoiding this problem is not easy, and requires the use of expensive predicate locks in general. In contrast, the situation with the Buddy*-tree is shown in Figure 3.10. There is no gap among the subspaces covered by leaf nodes and if we lock the leaf nodes corresponding to S_1, S_2 and S_3 the whole query range is accordingly "locked", and no inserter can make any update inside R. Thus repeatable reads are guaranteed.

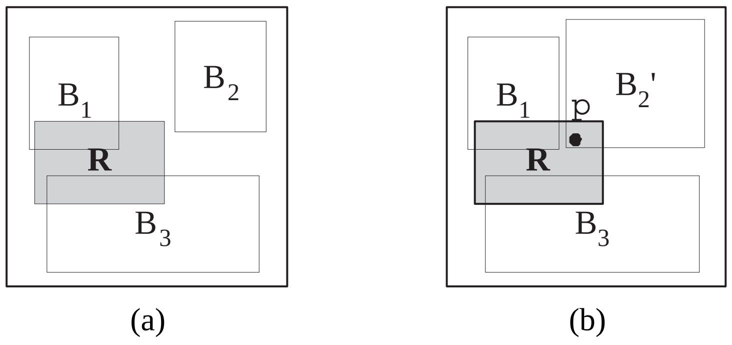


Figure 3.9: An Example of Phantom in R-Link-Tree

We take in the recovery method of R-link-tree based on the idea from [40] and [37]. The brief idea is to divide an update operation into contents-changing and structure-modifying part and employ a logical undo and redo. For the content-change, which involves the update on a leaf, write-ahead-logging (WAL) is used for recovery purpose. As for structure-modification, which may be a node split or update of any internal nodes, it does not have to be locked until the initiating transaction commits and can be visible immediately, i.e. if an atomic action such as split is committed, it will not roll back even

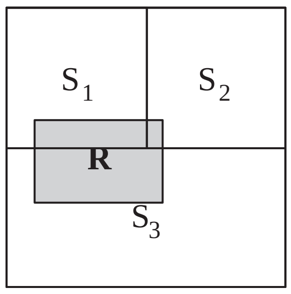


Figure 3.10: An Example of RR in Buddy*-Tree

if the initiating transaction fails.

CHAPTER 4

Buddy*-Tree Operations

In this chapter, we will describe the individual operations on the Buddy*-tree.

4.1 Querying

Buddy*-tree enlarges query range instead of enlarging MBRs. The formulae to calculate the enlarged window were presented in Section 3.2. These formulae require knowledge of minimum and maximum object velocities. Rather than computing these globally, we do this on a per node basis, suitably enlarging the query window when determining whether there is the possibility of overlap with a node.

Pseudocode for the range query algorithm is shown in Algorithm 1 and 2. The procedure *Range_Search()* is recursively called to examine a candidate node beginning from the root. During the visit of each node, if this is an interior node, its child nodes are added into the $tobeVisited$ list, when they are identified as having possible overlaps with the

Algorithm 1 Range_Query($Root$, r)

```
/* Input: $Root$ is the root node. $r$ is the query including query window, the predictive time ($t_{pre}$) and the query time ($t_q$)*/
1: $l$ := the indexed space
2: Range_Search($r$, $Root$, $l$)
3: r_unlock(all the locked leaf node)
```

suitably expanded query region. And then, *Range_Search()* is called for all the nodes in the list. If this is a leaf node, candidate matching data points are returned. Next, a right link is followed and additional linked nodes are visited as needed.

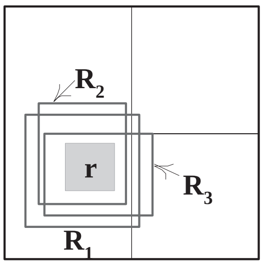


Figure 4.1: An Example of Range Query

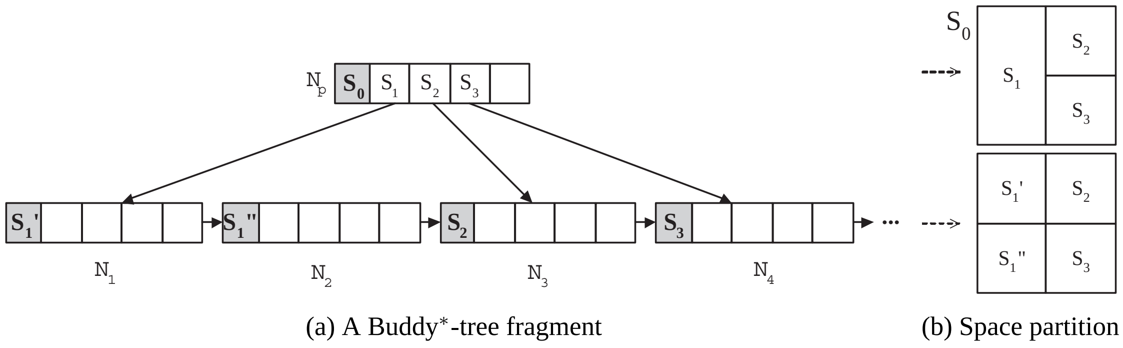


Figure 4.2: An Example of Uninstalled Split in Buddy*-Tree

Figure 4.1 gives a range query example. Suppose part of the index is shown as in Figure 4.2 and the query range is r, illustrated in Figure 4.1. First, the function *Range_Search* is called for root node N_p and its LBS S_0. LBSs of the three entries S_1, S_2 and S_3 are compared with the query range R_1, R_2, R_3 respectively, suitably expanded based on the time stamps and the recorded velocity extreme in these entries. The first and third entries are qualified and the child node pointers inside are added into $tobeVisited$ list with their LBS S_1 and S_3. We call procedure *Range_Search* for N_1 and N_4 in turn.

As a leaf node, N_1 is checked and qualified object entries are returned to the user. The current LBS of N_1 is S_1', which is not equal to S_1, which means that node N_1 has

Algorithm 2 Range_Search(r, N, l)

/* Input: r is the query window. N and l are the pointer of the node to be examined and its LBS obtained from its parent node, respectively*/

```
1:  r_lock($N$)
2:  if $N$ is marked by this thread then
3:     UnMark($N$)
4:  end if
5:  for each entry $e$ in $N$ that $e.LBS$ overlaps $R$, obtained by enlarging $r$ according to
    the time difference and extremal velocities in $e.node$ do
6:     if $N$ is not a leaf node then
7:        Add($tobeVisited$, ($e.node$, $e.LBS$))
8:        Mark($e.node$)
9:     else
10:       output qualified points in $e$
11:    end if
12: end for
13: if $N$ is not leaf then
14:    r_unlock($N$)
15: end if
16: while $tobeVisited$ is not empty do
17:    ($N'$, $l'$) := Get($tobeVisit$) // get next child $N'$ which is to be visited, $l'$ is the LBS
       added to the list together with $N'$
18:    Range_Search($r, N', l'$)
19: end while
20: if $N.LBS$ is not equal to $l$ /* there exists a split of $N$ hasn't been installed */ then
21:    traverse the right link chain starting at $N$ to first node whose LBS is not contained
       in $l$
22:    for each node $M$ along the chain except the last one do
23:       r_lock($M$)
24:       $l'$ := $M.LBS$
25:       r_unlock($M$)
26:       Range_Search($r$, $M$, $l$)
27:    end for
28: end if
```

split. So its right link is followed and N_2 is reached. LBS S_1'' of N_2 is contained in S_1, hence, it is tested by the query window for qualified object entries as if it were part of original node N_1. Continuing along right links, N_3 is next reached and the fact that its LBS S_2 is not part of S_1 completes the right traversal.

Consider such a case in the above example–after N_P is visited, node N_1 and N_4 are added into $tobeVisited$ list and if before we visit N_4, other users delete some points which causes N_3 merge with N_4, then when it is our turn to visit N_4, it has already been deleted. To avoid such problem we mark a node if it is added into the $tobeVisited$ list in query process (line 8 in Algorithm 2) and remove our mark for this node when we obtain the lock for it (line 3). To delete a node, the marks for it must be check (see Section 4.3).

4.2 Insertion

The insert process of the Buddy*-tree consists of three main steps as outlined in Algorithm 3. First, the snapshot of the object must be computed, i.e. the location at t_l, according to its location and velocity at the update time t_u and it is put in an entry (line 1). Second, we must locate the leaf node N that the object (snapshot) should be inserted into, following a procedure similar to that for search described above. The path from root to the leaf N is noted (line 2). Third, the actual insertion is performed (line 4) after locking N, using the algorithm *Insert_Entry()*.

The function of *Insert_Entry(s, N)* (outlined in Algorithm 4) is to insert the entry s into the node N. A possible situation is that after we locate N and before we lock

Algorithm 3 Insert(p,t_u)

```
/* Input: $p$ is the point to insert and $t_u$ is the operation time */
1: $s$ := the entry containing $p$'s snapshot according to $t_u$ and $t_l$
2: $N$ := Locate($root$, $s$, $path$)
3: w_lock($N$)
4: Insert_Entry($s$, $N$)
```

it, another inserter overtakes us and splits N. To ensure correctness in this case, before putting s in N, we must check whether it is the right node to insert, i.e. whether N's LBS covers s's (line 1). If not so, which means that N is split, we move right, hence *Move_Right(s, N)* is called (line 2). The function of *Move_Right(s, N)* (outlined in Algorithm 5) is to find a node in the right link chain beginning from N, which is suitable for s. Then, s is simply put in the new N that is returned by *Move_Right()* if there is any empty entry (line 4-7). Otherwise, a split is caused by the overflow (line 8-27).

Algorithm 4 Insert_Entry(s, N)

/* Input: s is the entry containing a point or a branch to install into node N. Node N as input is write-locked and it is unlocked after the procedure, */

```
1:  if $N.LBS$ doesn't cover $s.LBS$ /* $N$ has been split */ then
2:      $N$ := Move_Right($N$, $s.LBS$)
3:  end if
4:  if find an empty entry $e$ in $N$ then
5:      put $s$ in $e$
6:      $N.num++$
7:      w_unlock($N$)
8:  else
9:      Node_Intlz($newN$)
10:     Split_Node($N$, $newN$)
11:     if $N$ is not the root then
12:         $P$ := $N$'s parent node /* get information from the memoried path */
13:         w_lock($P$)
14:         if $P.LBS$ doesn't cover $N.LBS$ /* $P$ has been split */ then
15:             $P$ := Move_Right($P$, $N$.LBS)
16:         end if
17:         w_unlock($N$)
18:         update LBS in the corresponding entry in $P$ for $N$
19:         $s'$ := the entry containing $newN$
20:         Insert_Entry($s'$, $P$)
21:         w_lock($N$)
22:         Insert_Entry($s$, $N$)
23:     else
24:         Node_Intlz($P$)
25:         insert $N$ and $newN$ in $P$ and make it new root for the index tree
26:         Insert_Entry($s$, $N$)
27:     end if
28: end if
```

Algorithm 5 Move_Right(N, s)

/* Input: N is the beginning node in the right travel and s is the entry with the wanted point or LBS. The procedure will release lock for original N and write-lock the new N*/

```
1: while $N.LBS$ does not cover $s.LBS$ do
2:     $tempN$ := $N$'s right link
3:     w_unlock($N$)
4:     $N$ := $tempN$
5:     w_lock($N$)
6: end while
```

Algorithm 6 Split_Node($N, newN$)

/* Input: N is the node to split and $newN$ is a new empty node for split purpose. The write-lock for N is already hold and will kept in the entire procedure*/

```
1: $splitdim$ := Choose_Spl_Dim($N$)
2: split the LBS and repartition pointers between $N$ and $newN$ according to $splitdim$
3: insert $newN$ into the right link chain next to $N$
```

The LBS S of an overflowed node N is split into two equal sized parts S_1 and S_2, where S_1 occupies the original node N and S_2 is assigned as LBS to a new node $newN$. The split dimension is chosen by turn in rotation, and the split position is the median. The new node $newN$ is firstly inserted into the right link chain of N. The above process is done in function *Split_Node()* (Algorithm 6). At this stage, the LBS in the corresponding entry of N's parent node is not updated, but any concurrent accesses can reach $newN$ through the right link chain from N. After that, we install the split into the parent if N is not the root (line 11-22). The parent node P of N is accessed (line 12), using the root-to-leaf path remembered from line 2 of Algorithm 3. Since P is possibly split by some other users after our last access, we should check P's LBS with that of N and $newN$ (line 14). If N's LBS is not a subspace of P's, we can conclude that P has been split. The right link chain beginning with P is searched to reach the real parent node for N and $newN$ (line 15). N's LBS is updated in P (line 18) and the entry with new node $newN$ is installed (line 19-20). This new entry installation could cause node P to split, and so on recursively until a node with empty entry is reached or the root is split (line

23). In the latter case a new root is created (line 24-25).

4.3 Deletion

Deletion is similar to insertion; first locate the key value at leaf level and then delete it. It is possible that this causes the leaf node to have very few (non-empty) entries left. Such underflow is handled in the Buddy*-tree by merging it with its buddy, where the buddy has few enough entries itself.

Node splits and merges require “backing up” the tree at a potential loss of concurrency. For the more common case of node splits, we have described in detail how to use right-links to manage node splits with minimal impact on concurrency. For the less common case of node merges, the same ideas apply, in reverse order. That is to say, when a node merger is to be undertaken of two buddy nodes, first fix the entry at the parent to point only to the “left” buddy. At this stage, the “right” buddy is no longer linked from the parent, but is only accessible by right linking from its buddy. Then actually perform the node merge into the left buddy and eliminate the right buddy. The algorithm details are along the lines described for insertion above (shown in Algorithm 7, 8, 9 and 10).

Note that, for typical scenarios with moving objects, updates are much more frequent than pure insertions or deletions. While performing a node merge is possible algorithmically, from an engineering perspective we are frequently better off leaving alone the underflowed node since it is quite likely to fill up again after a while.

Algorithm 7 Delete(p,t_u)

/* Input: p is the point to delete and t_u is the last update time of p */

```
1: $s$ := the entry containing $p$'s snapshot according to $t_u$ and $t_l$
2: $N$ := Locate($root$, $s$, $path$)
3: w_lock($N$)
4: Del_Entry($s$, $N$)
```

Algorithm 8 Del_Entry(s, N)

/* Input: s is the entry containing a point or a branch to delete from node N. Node N as input is write-locked and it is unlocked after the procedure. */

```
1: if $N.LBS$ doesn't cover $s.LBS$ /* $N$ has been split */ then
2:    $N$ := Move_Right($N$, $s.LBS$)
3: end if
4: if find the entry $s$ in $N$ then
5:    delete $s$ from $N$
6:    $N.num--$
7: end if
8: if $N.num < MinFill$ and $N$'s right neighbor $\neq NULL$ then
9:    $M$ := $N$'s right neighbor
10:   w_lock($M$)
11:   if IsBuddy($N$, $M$) and $N.num + M.num \leq Fanout$ /* merge $N$ and $M$ */ then
12:      $P$ := $N$'s parent node
13:      w_lock($P$)
14:      if $P.LBS$ doesn't cover $N.LBS$ /* $P$ has been split */ then
15:         $P$ := Move_Right($P$, $N$.LBS)
16:      end if
17:      $e_1.LBS := e_1.LBS + e_2.LBS$ //$e_1$ and $e_2$ are entries in $P$ for child nodes $N$
         and $M$ respectively
18:      Del_Entry($e_2$, $P$)
19:      Merge_Node($N$, $M$)
20:      w_unlock($P$)
21:      Del_Node($M$)
22:   else
23:      w_unlock($M$)
24:   end if
25: end if
26: if $N$ is the root and $N.num = 1$ then
27:    make $N$'s child the root
28:    Del_Node($N$)
29: else
30:    w_unlock($N$)
31: end if
```

Algorithm 9 Merge_Node(N, M)

/* Input: M is the node to be merged into N. The write-locks for N and M are already hold and will kept in the entire procedure, i.e., they are still locked after the procedure*/

```
1: update $N.LBS$ by merging the LBSs of $N$ and $M$
2: copy all the no-empty entries from $M$ to $N$
3: copy the right link of $M$ to $N$
```

Algorithm 10 Del_Node(N)

/* Input: N is the node to be deleted*/

```
1: if $N$ is marked by any thread //$N$ in a certain $tobeVisited$ list then
2:     put $N$ into $tobeDel$ list //the nodes in $tobeDel$ list will be deleted periodically
3:     w_unlock($N$)
4: else
5:     delete this node
6: end if
```

CHAPTER 5

Experimental Evaluation

In this chapter, we perform experimental study to evaluate the performance of the Buddy*-tree and present the results.

We implemented the Buddy*-tree, and compared its performance to that of the TPR*-tree and B^x-tree. All of these structures were implemented in C. All experiments were conducted on a single CPU 3G PentiumIV Personal Computer with 1 G bytes of memory.

We ran two sets of experiments, one with a single thread of activity, and another with multiple concurrent threads. In both sets of experiments we use synthetic uniform datasets. The position of each object in the data set is chosen randomly in a 1000×1000 space. Each object moves in a randomly chosen direction with a randomly chosen speed ranging from 0 to 3. We constructed the index at time 0. For the test on the effect of data distributions, we use the network dataset [49]. The parameters used in the experiments are summarized in Table 5.1, and the default values are highlighted in bold.

5.1 Storage Requirement

Storage requirement is an important issue for database index. Firstly, a comparatively smaller size index can be whole cached in the main memory to improve performance. Secondly, a smaller size index means less data pages (nodes), such that, it reduces the I/O

Parameter	Setting
Page size	4K
Max update interval	60,**120**,180,240
Max predictive interval	120
Query window size	**10**,20,...,100
Number of queries	200
Dataset size	100K,...,**500K**,...1M
Number of threads	2,4,8,...,**64**,128,256
Number of operations per thread	200
Data distributions	**uniform**, network

Table 5.1: Parameters and Settings

cost due to an operation such as query or update may visit fewer nodes. The fanout of an internal index node plays a most important role in the storage requirement. Obviously, if the size of the information maintained for a child node is smaller, there's more child nodes can be kept in one internal index nodes.

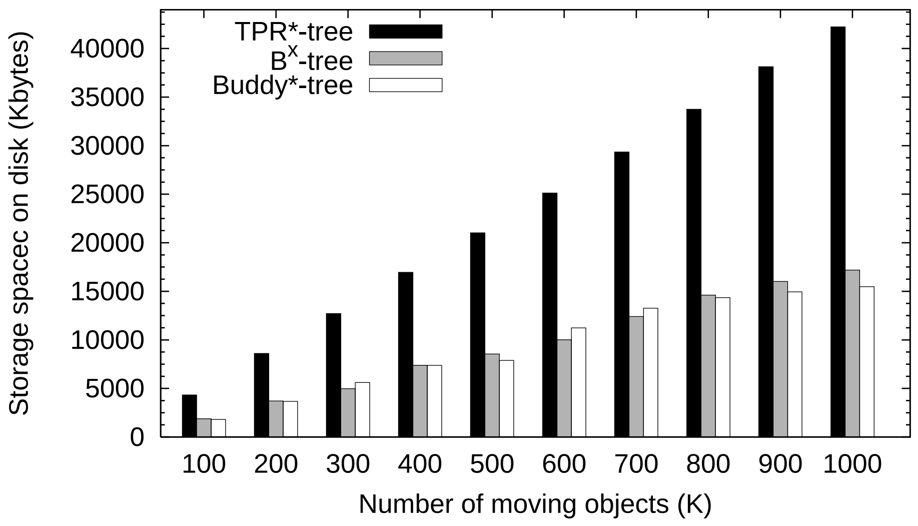


Figure 5.1: Storage Requirement

In a Buddy*-tree internal entry, the space partition is kept for the child node (at least 8 Bytes for 2-dimensional space). We use the global speed for enlarging the query space, and hence speed local to each LBS is not stored. We observe in the experiments that local speeds only improve the performance slightly since most maximum local speeds are close to the global maximum speeds. As for B^x-tree, each entry contains a 64bit key (8 Bytes). However, a TPR*-tree internal node stores MBRs and VBRs for each child entry (24 Bytes for 2-dimensional space). The storage requirement of the indexes is shown in Figure 5.1. As anticipated, TPR*-tree requires more than twice storage space of the others, which are comparable.

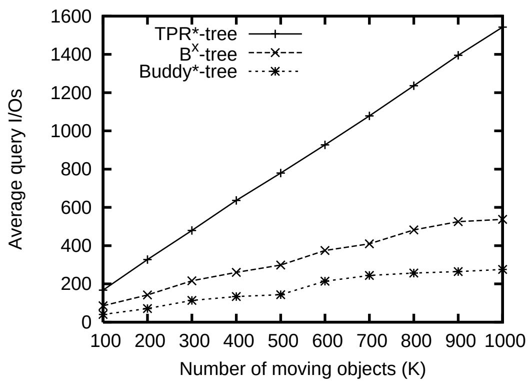
5.2 Single Thread Experiments

In this part there is only one thread in the experiments. We study the performance of the Buddy*-tree by comparing I/O cost and CPU cost to TPR*-tree and B^x-tree.

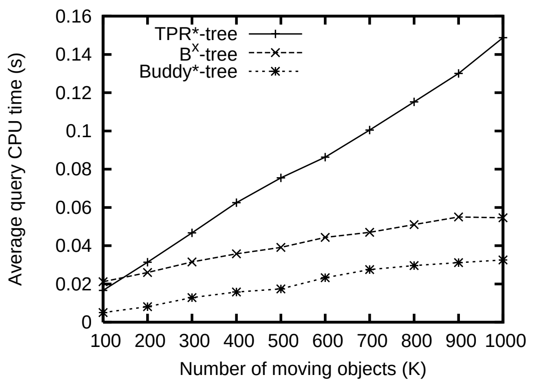
5.2.1 Effect of Dataset Size

First, we study the range query performance with different sizes of dataset by comparing the costs when the number of moving objects in the dataset varies from 100K to 1M. 200 window queries with size 10 are issued after the index running for an entire maximum update interval of 120 time units. The predictive intervals of the queries are randomly chosen in the range from 0 to 120. Figure 5.2 shows the average cost of I/O operation and CPU time per query for the three inspected indexes.

As expected, the results show that the window query costs of all the indexes increase with the number of objects. However, The increasing speed of the TPR*-tree is much higher than that of the others. When there are 1M objects in the dataset, the cost of the TPR*-tree is nearly 3 times over that of the B^x-tree and more than 5 times over that of the Buddy*-tree. The explanation of this result is as follows. The Buddy*-tree is a space partition based index whose range query cost increases mainly due to the number of objects inside the range. However, with increasing number of data, the TPR*-tree suffers



(a) I/O cost



(b) CPU time

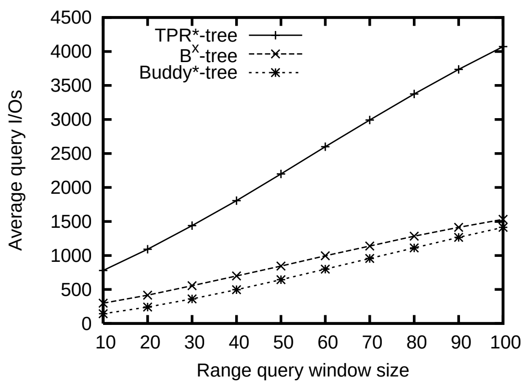
Figure 5.2: Effect of Dataset Size on Range Query Performance

seriously from the overlap among MBRs. Furthermore, the small fanout also limits the performance of the TPR*-tree. The B^x-tree employs a space filling curve to map objects in 2-dimensional space to single dimension space. The curve is cut into a set of intervals by a query window. To search for all the intervals we must “jump” among the subtrees, and during such jumps several internal nodes are likely to be visited more than once. This behavior introduces a few more I/O operations.

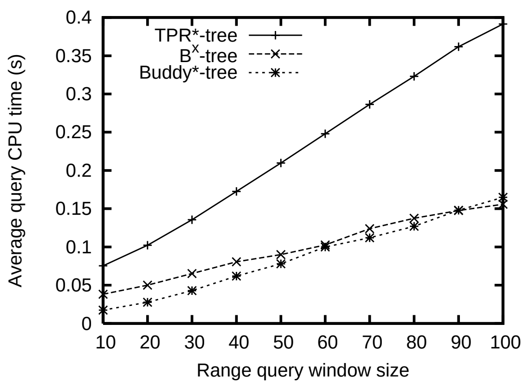
5.2.2 Effect of Query Size

We next investigate the performance of the indexes with respect to query size.

In the experiments we vary the query window size from 10 to 100 on a dataset of size 500K. The same 200 queries with predictive interval randomly chosen from 0 to 120 are issued in the three indexes after they run for 120 time units. As shown in Figure 5.3, query costs of all the indexes increase with the query window size. This behavior is straightforward, since a larger window covers more objects and accordingly, more index nodes will be accessed and examined. However, the TPR*-tree degenerates considerably over the other indexes. This is attributed to the overlap problem of TPR*-tree. Since a larger query window contains more overlaps of the MBRs, and hence, more accesses of nodes result. The B^x-tree costs a little more than the Buddy*-tree does. As can be observed, with query window increasing, the gap between B^x-tree and Buddy*-tree declines. With query window size 10, the Buddy*-tree reduces the query cost by about 50% compared to that of the B^x-tree and this performance gap is only about 10% when the window size increases to 100. This is because that in the query process on B^x-tree, a larger query window tends to get a smaller set of longer intervals of the space curves rather than a larger set of shorter intervals in a smaller window. This reduces the high cost of "jumps" in a way.



(a) I/O cost



(b) CPU time

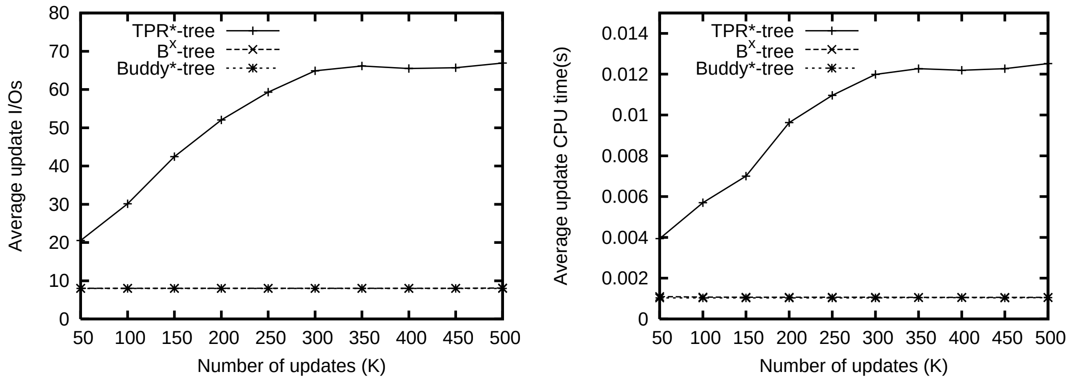
Figure 5.3: Effect of Query Window Sizes on Range Query Performance

5.2.3 Effect of Updates

In this subsection, we compare the average update cost of Buddy*-tree against the TPR*-tree and the B^x-tree.

First, to study the update costs of the indexes evolving with the passage of time, we compute the average update cost of the three indexes after every 50K updates in a 500K dataset. Note that each update involves an insertion and a deletion and leave the size of the tree unchanged. Figure 5.4 summarizes the results, showing that TPR*-tree degrades considerably faster than the B^x-tree and Buddy*-tree, which are comparable.

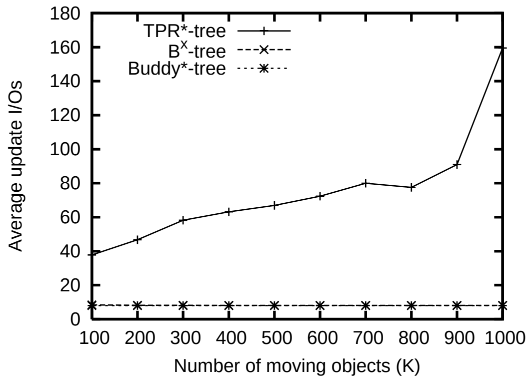
The reason is that each deletion entails a search to retrieve the object to remove and since the behavior of multiple path travel in the TPR*-tree, the cost of search inevitably increases with time due to the continuous enlargements of the MBRs which are not updated as time passes. Fortunately, the update cost approaches a saturation point after some time as the enlargement of MBRs grows at a much slower pace due to the overall coverage. In contrast, we observe that the average update costs of B^x-tree and Buddy*-tree are not very sensitive with respect to elapse time. This is because that in the B^x-tree and Buddy*-tree, an insertion or deletion only travel down one path by comparing the key (a value in B^x-tree and a rectangle or a point in Buddy*-tree). No matter how large the dataset is, only the nodes along the path from root to the leaf node that contains the desired object are accessed. Thus, the number of I/Os only depends on the height of the tree and does not change much over time. In fact, they are almost time independent.



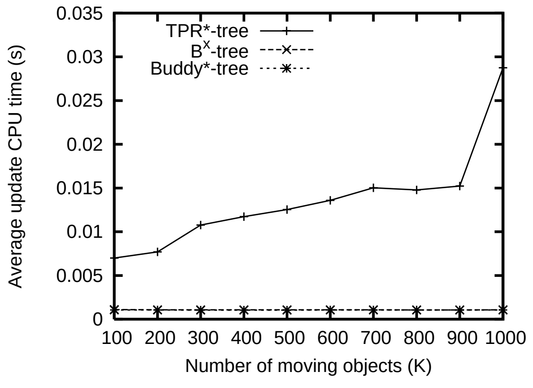
(a) I/O cost (b) CPU time

Figure 5.4: Effect of Time Elapsed on Update Cost

We next compare the update performance of indexes with respect to the size of dataset. In this experiment, we vary the number of objects in the dataset from 100K to 1M, and investigate the average update costs after the indexes running for a maximum update interval of 120 time units. Figure 5.5 shows the update cost as a function of the number of moving objects.



(a) I/O cost



(b) CPU time

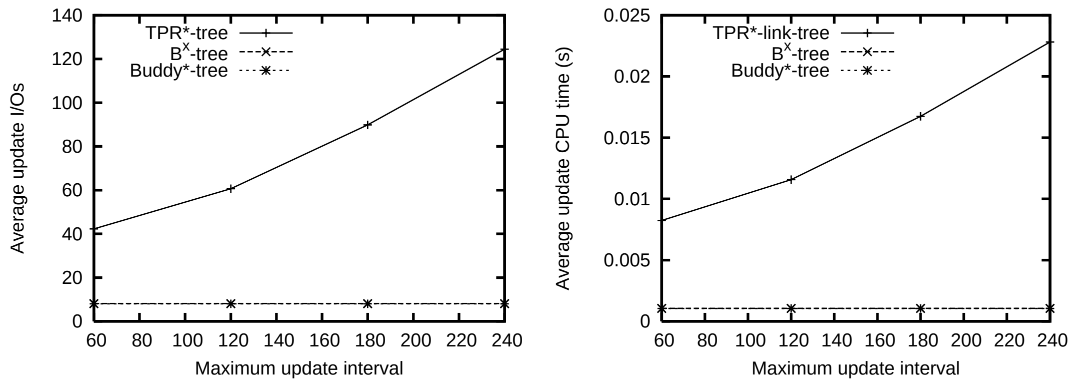
Figure 5.5: Effect of Dataset Size on Update Cost

As shown in the figure, an update in the B^x-tree or Buddy*-tree only incurs several I/Os and the cost is not affected a lot by the size of dataset. The reason is that the retrieval of an object in B^x-tree or Buddy*-tree is single-path and, thus, the number of I/Os only depends on the height of the tree. Only when the increasing data size causes the tree to grow, an update in the two indexes will incur about 2 more I/Os. However, in this experiment, when the range of dataset size is varied from 100K to 1M, both the B^x-tree and the Buddy*-tree remain 3-level tall, hence, no change of the update costs is observed.

We observe that the performance of the TPR*-tree degrades with the increasing size of the dataset. The explanation is that in the TPR*-tree, traversing multi-path is inevitable due to the overlaps among MBRs. The increase in data size causes the increase in density of objects, resulting in more overlap and higher update cost. The performance curve of TPR*-tree can be observed at the point of ‘900K’ due to the growth of the tree from 3 levels to 4 levels. The TPR*-tree grows faster than the others due to the smaller fanout of it.

5.2.4 Effect of Update Interval Length

In this experiment, we study the effect of maximum update interval length on the update performance of indexes. Figure 5.6 shows the average update costs after the indexes run for one maximum update interval, varying from 60 to 240. Observe that the B^x-tree and Buddy*-tree are not affected by the length of maximum update interval, whereas the TPR*-tree degrades fairly quickly. As we have discussed above, this is because the number of I/Os of these two indexes only depends on the height of the tree and does not change over time. As the update interval increases, MBRs in TPR*-tree keep enlarging; overlaps among them become more severe. Therefore the update cost increases significantly.



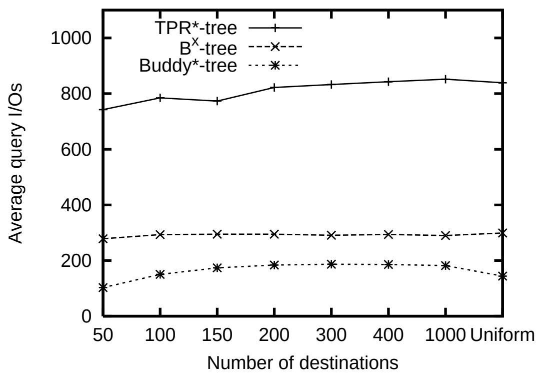
(a) I/O cost (b) CPU time

Figure 5.6: Effect of Maximum Update Interval

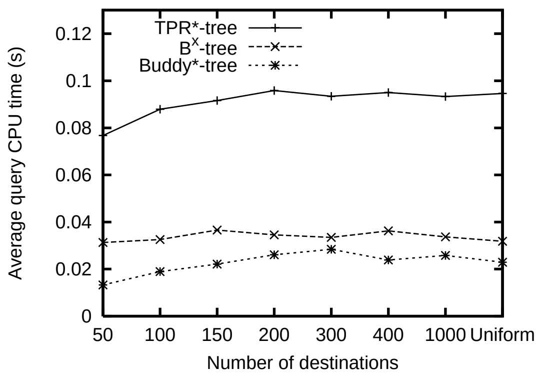
5.2.5 Effect of Data Distribution

This experiment uses the network dataset to study the effect of data distribution on the indexes. The dataset is generated by an existing data generator, where objects move in a road network of two-way routes that connect a given number of uniformly distributed destinations [49]. The dataset contains 500K objects, that are placed at random positions on routes and are assigned at random to one of three groups of objects with maximum speeds of 0.75, 1.5, and 3. Objects accelerate as they leave a destination, and they decelerate as they approach a destination. Whenever an object reaches its destination, a new destination is assigned to it at random.

Figure 5.7 summarizes the average range query costs of the three indexes when the number of destinations in the simulated network of routes is varied. Decreasing the number of destinations adds skew to the distribution of the object positions and their velocity vectors. Thus, uniform data is an extreme case. As shown, increased skew leads to a decrease in the range query cost in the TPR*-tree. This is expected because when there are more objects with similar velocities, they are easier to be bounded into rectangles that have small velocity extents and also are not too big. The results are consistent with the performance of the TPR-tree reported in [49]. As expected, the performance of the B^x-tree is not affected by the data skew because objects are stored using space-filling curves and hence, the density has less of an effect on the index. Observe that the range query cost of the Buddy*-tree firstly increases with the number of destinations and after the point that the number of destination is 300, the cost descends. A main reason for this interesting behavior is that the data distribution affects two factors, which make contrary effects on the performance of the Buddy*-tree – (i) with the increasing skewness of the dataset, the objects inside the same data page tend to possess more similar velocities, which leads to less enlargements of the query windows during the range queries and thus, less cost of queries; (ii)since the split algorithm of the Buddy*-tree is not adaptive for data distribution, skewed dataset may introduce empty or nearly empty data nodes for the index, resulting in poor disk utilization and hence cause the decline in query performance. As shown, when the dataset is very skewed (the number of destinations is less than 300), the cost decreases with the skewness of dataset because factor (i) dominates the performance. After the turning point of 300, factor (ii) affect the performance a little more, therefore the cost decreases with number of destinations.



(a) I/O cost



(b) CPU time

Figure 5.7: Effect of Data Distribution on Range Query Performance

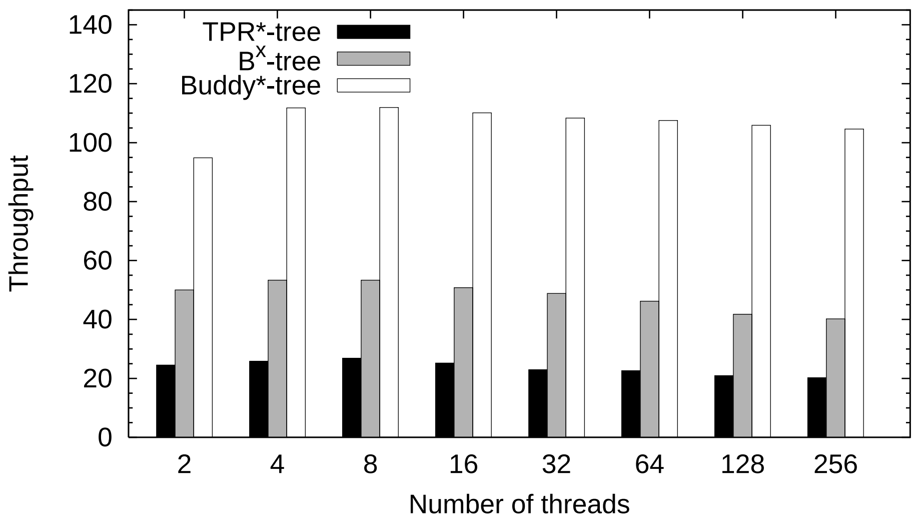
5.3 Multiple Thread Experiments

In this part we compare the performance of indexes on concurrent operations. We used multi-thread programs on the PC to simulate multi-user environment. We implemented B-link for B^x-tree. We note that the TPR*-tree employs different update algorithms from the TPR-tree (e.g. remove and reinsert a set of entries in split algorithm), it cannot grantee RR (repeatable read) even if we implement R-link for it. Designing a new and efficient concurrency control mechanism for the TPR*-tree is possible, but not straightforward. For baseline comparison purposes, we simply locked the whole TPR*-tree for concurrency control. For illustration purposes, we also implemented the R-link structure for the TPR-tree, and show its performance in Section 5.3.2.

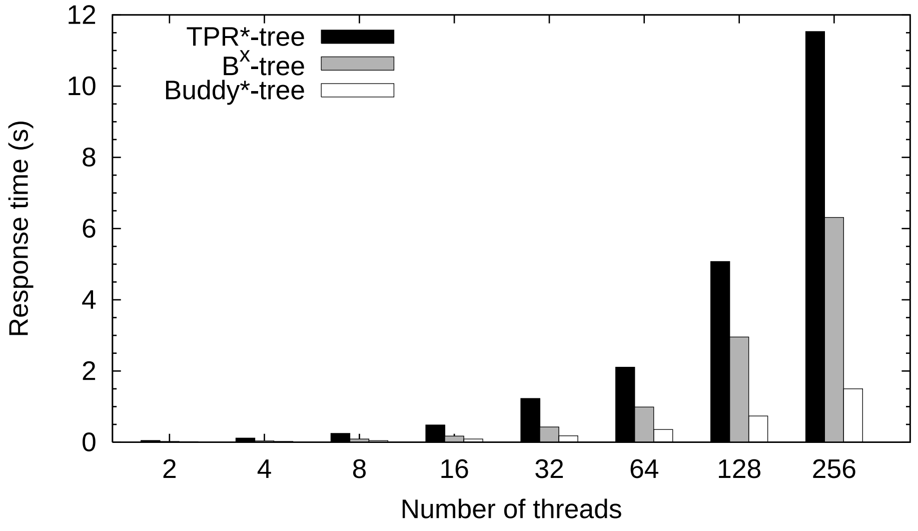
In the following experiments, each thread issues 200 operations, and the default workload of each thread contains the same number of queries and updates.

5.3.1 Effect of Number of Threads

First, we investigate the effect of the number of threads. Figure 5.8 shows the throughput and response time for the indexes by varying the thread number from 2 to 256 (each thread issues 100 queries and 100 updates).



(a) Throughput



(b) Response time

Figure 5.8: Effect of Threads on Concurrent Operations

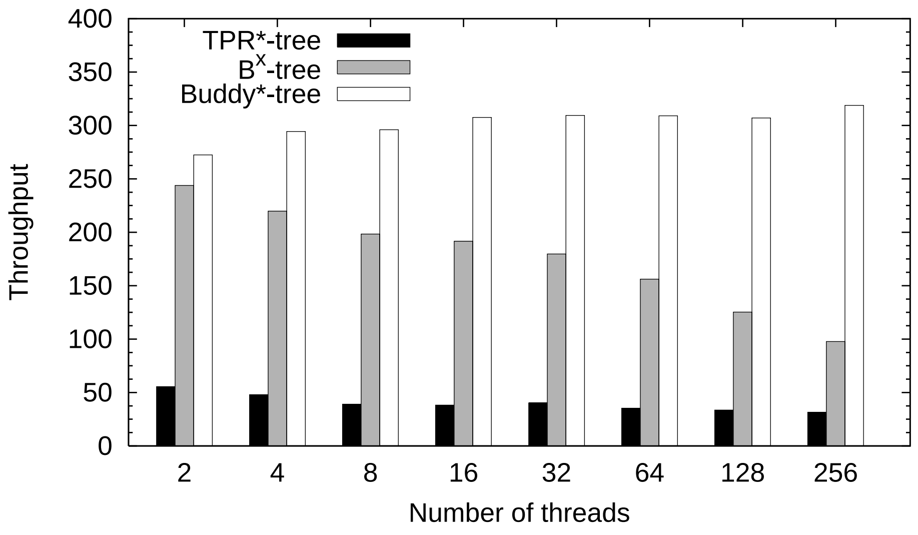
All the indexes reach the highest throughput at around 8 threads and thereafter show deteriorating performance as the number of threads is increased. Measuring the decline as we go from 8 to 256 threads, we find this decrease to be only 6.5% for Buddy*-tree, but 24.5% and 24.6% for B^x-tree and TPR*-tree respectively. Since the Buddy*-tree has been designed for high concurrency, its superior performance with multiple threads validates our design. The decline in performance of the TPR*-tree is also to be expected. The surprise is the decline in performance of the B^x-tree in spite of the use of B-link chain for high concurrency. The main reason for this is that a lot of "jumps" in the B^x-tree for range query increase the number of accesses of and locks on internal nodes, which reduces the degree of concurrency.

Note that we measured multi-thread operation on a single-CPU PC, on which different threads could not be really run at the same time. We believe that in a real concurrent environment both B^x-tree and Buddy*-tree will provide even better performance, compared to the TPR*-tree.

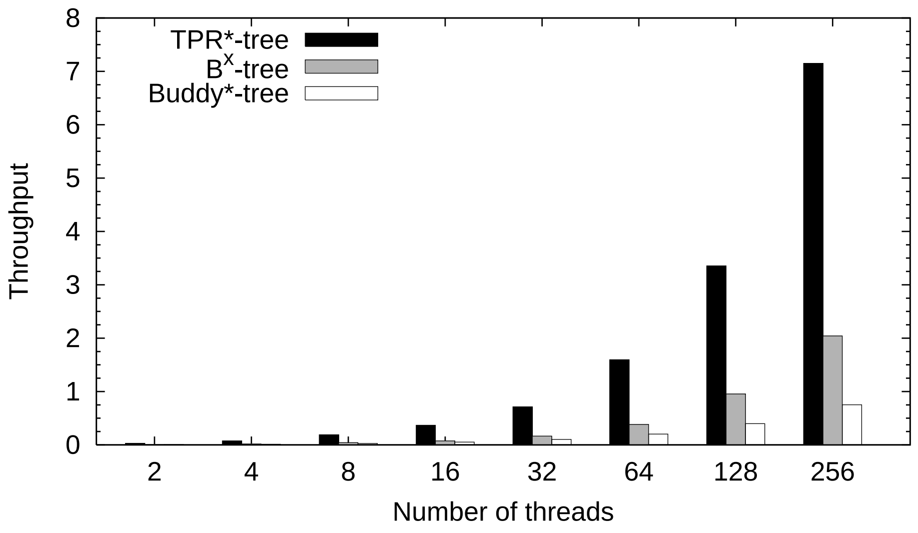
The results for response time show the impact of concurrency even more starkly. The response time of TPR*-tree with 256 threads is about 240 times of that with 2 threads. This ratio approximately 200 and 100 for B^x-tree and Buddy*-tree respectively.

To study the update performance with respect to the number of threads, in this experiment, we vary the thread number from 2 to 256, where each thread is assigned a workload of 200 updates.

The throughput and response time for the indexes are summarized in Figure 5.9. The throughput of the TPR*-tree and the B^x-tree descend with the thread number increasing. As shown in Figure 5.10, with the increase of threads number, average update I/Os of all the indexes are not effected much. Therefore, we can conclude that the degenerations of the TPR*-tree and B^x-tree are not caused by I/O operations, but caused by lock contention. As before, the degradation of the TPR*-tree is to be expected. However, the performance of the B^x-tree is much below expectation. The main reason is that, there're



(a) Throughput



(b) Response time

Figure 5.9: Effect of Threads on Concurrent Updates

three subtrees for different timestamps in the B^x-tree and the insertions at a same time only involve one of the subtrees, that is, all the insertions “crowd” in 1/3 part of the index tree. The crowd becomes more severe with threads number increasing, resulting in more access conflicts and hence lower degree of concurrency. In contrast, the performance of the Buddy*-tree is almost not affected by the thread number due to the right link structure, and quicker release of locks due to simpler split and no bounding box update.

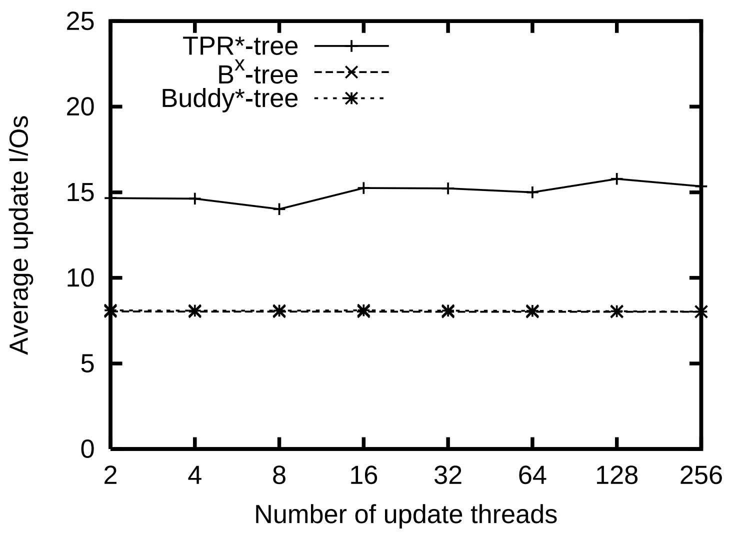
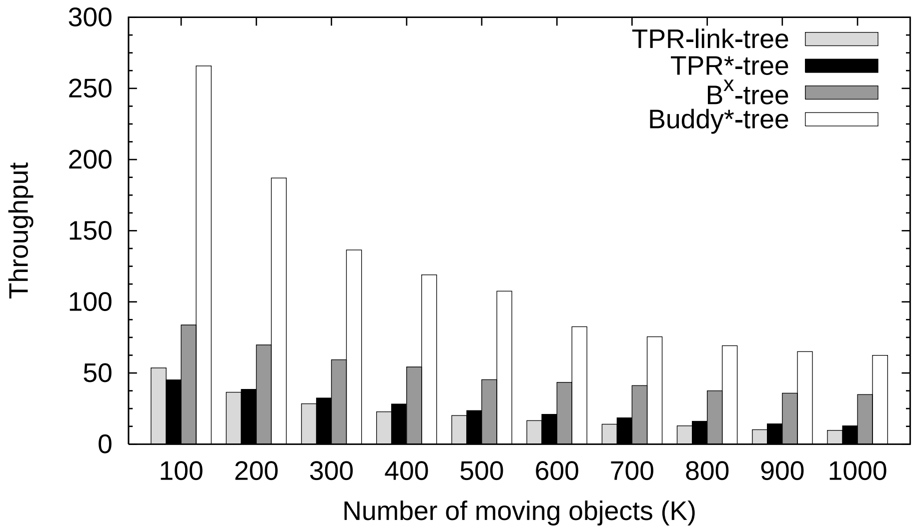


Figure 5.10: Effect of Threads on Update I/O Cost

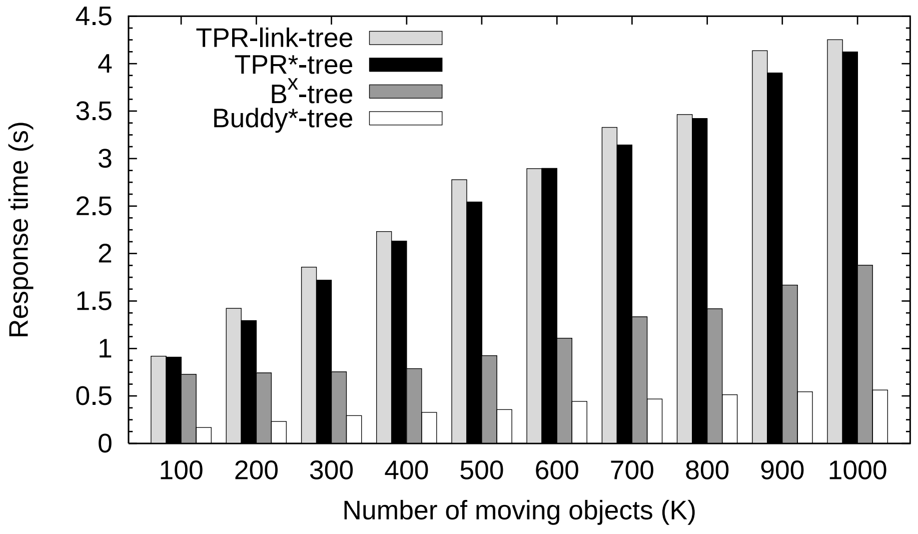
Note that, I/O cost of TPR*-tree in Figure 5.10 is not consistent with that in Figure 5.4. This is because that all the updates in Figure 5.10 issue at a same time, and the MBRs are not enlarged by time. While, updates in Figure 5.4 spread randomly in a period of 120 time units and the MBRs keep enlarging due to time elapsing and hence I/O cost increases.

5.3.2 Effect of Dataset Size

We next investigate the performance of the indexes with different numbers of moving objects on concurrent operations. In these experiments, the throughput and response time are compared after running 64 threads on the dataset whose size varies from 100K to 1M.

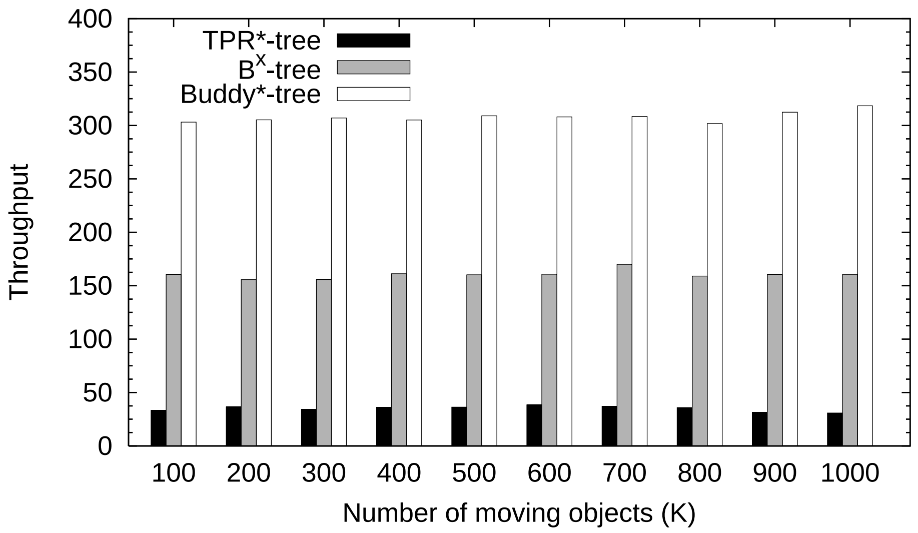


(a) Throughput

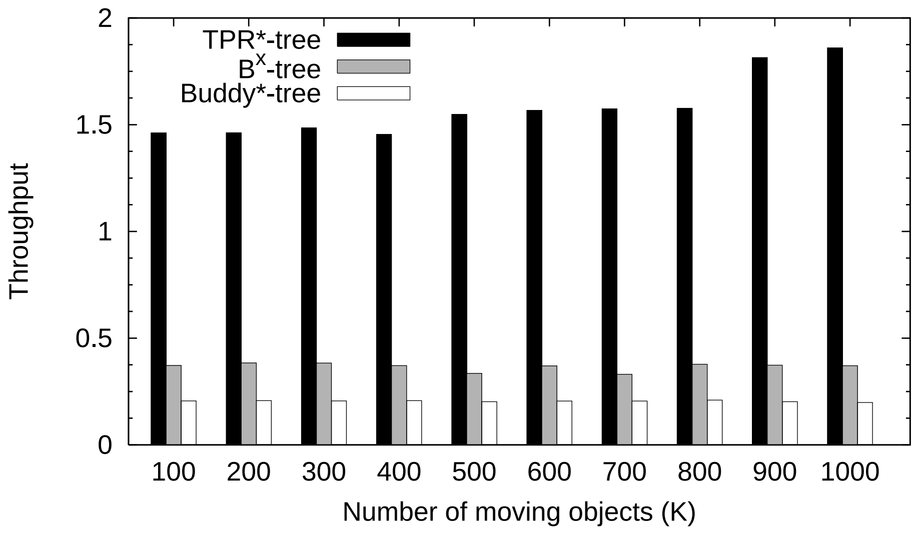


(b) Response time

Figure 5.11: Effect of Data Size on Concurrent Operations



(a) Throughput



(b) Response time

Figure 5.12: Effect of Data Size on Concurrent Updates

As shown in Figure 5.11, the performance of all indexes reduces with the increasing number of moving objects. This is straightforward, since the larger the dataset is, the more nodes an index contains and the more I/O operations a query or update brings. However the Buddy*-tree outperforms the other indexes for both throughput and response time. As before, TPR*-tree is the worst, because that small fanout and overlap cause poor range query performance. However, its performance is comparable to the TPR-tree whose more efficient R-link based concurrent accesses are being compromised by poorer query efficiency.

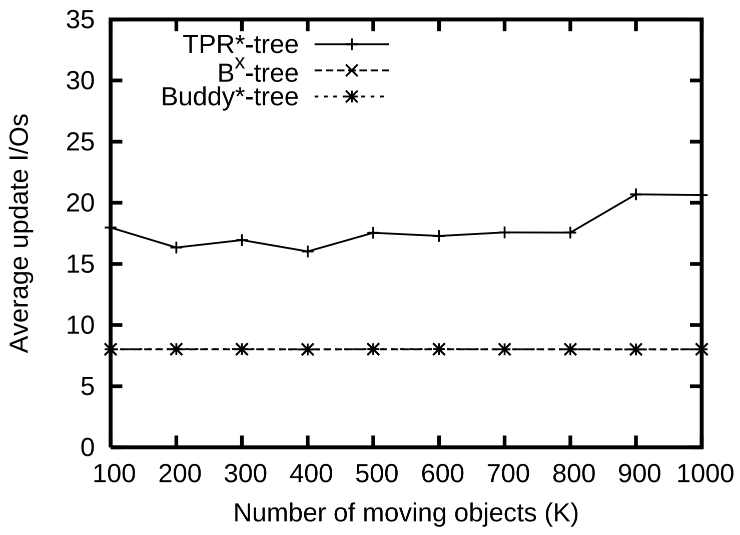


Figure 5.13: Effect of Data Size on Update I/O Cost

Also, we conduct experiments to study the effect of data size on concurrent updates. The setting of this experiment is same with that of the last one, except that the workloads only contain updates. From Figure 5.13, we can see that the average update I/O costs of three indexes do not affected obviously by the dataset size. TPR*-tree costs about twice of the others do. However, compared to the performance showed in Figure 5.5, the cost of TPR*-tree in this experiment is much lower and does not degrade much with data size increasing. As before, this is due to the different distribution of updates with respect to time. The B^x-tree and the Buddy*-tree achieve almost the same I/O performance. Whereas, the results summarized in Figure 5.12 shows that Buddy*-tree outperforms B^x-tree in the throughput and response time. This attributes to that the right links enable the Buddy*-tree to handle concurrent updates efficiently, however, the B^x-tree suffers from the “crowd” problem. The concurrent performance of TPR*-tree is still the worst. This is without doubt, since the highest I/O cost and worst concurrency control of it.

The performance study in this chapter shows that the Buddy*-tree outperforms the TPR*-tree and the B^x-tree in both single-thread experiments and experiments with concurrent operations. Especially for high degree concurrent operations, B^x-tree outperforms the other two indexes by a wide margin.

CHAPTER 6

Conclusion

In this thesis, we investigated the problem of indexing for moving objects. We presented a thoroughly review of traditional index structures and existing indexing techniques for mobile objects. In order to support frequent updates and concurrent operations, we proposed a space partitioned based index structure Buddy*-tree, a generalization of Buddy-tree, for indexing mobile objects.

The central idea is to use an adaptive query expansion technique to allow for object motion, while indexing only static snapshots. Therefore, we only need to consider static objects rather than mobile objects. So we can choose a multidimensional structure with good update properties. Buddy-tree, as a well performed SP based index, thus, is used as the basic for our new index. We create a Buddy*-tree based on a standard Buddy-tree with two key differences. First, the LBSs (loose bounding spaces) are employed instead of the MBRs (minimum bounding rectangles). This strategy benefits both the update process and concurrent operations, at the cost of having some bounding rectangles be unnecessarily large. However, the cost is worthy, which is verified by the experiment results in Chapter 5. Secondly, since high update rates are common for mobile objects, a right link structure is additionally used to permit high concurrency. We make use of the properties of Buddy*-tree to realize concurrent control without any other structure additional (such as LSN in R-lint-tree). An object is partitioned to an index node based on its location at a certain reference time, and is stored as a snapshot consisting of velocity vector and location with the timestamp. To handle future range query, we employ query window enlargement instead of MBRs enlargement in TPR-tree. We also proved that these two methods of enlargement create the same query results.

The main advantages of our proposed index structure are as follows:

1) As a space partitioned based structure without MBRs, it does not suffer from the overlap problem and hence is able to support more efficient update and range queries for moving objects;

2) According to the presentation method of Buddy*-tree, node entries only contain space information for the subtree or objects, and are relatively small, permitting a larger fanout and requiring less storage space than competing techniques such as the TPR*-tree. This also leads to better performance.

3) An extremely aggressive lock release policy can be applied to obtain high concurrency, through the use of a secondary right link traversal process. Since high update rates are common for mobile objects, this high concurrency renders the Buddy*-tree even more attractive.

These advantages are verified by the experiments. The performance study shows that the Buddy*-tree outperforms the TPR*-tree and the B^x-tree. This is even more strongly the case when multiple concurrent operations are applied.

The main limitation of Buddy*-tree, which is inherited from Buddy-tree, is the storage utilization for skewed data. Since we employ almost the same split algorithm with that of Buddy-tree – when a node overflows, the corresponding space is always split into two subspaces of equal size. Skewed data, thus, will introduce empty or nearly empty nodes. Therefore, in the future work, we will try to study the split strategy (such as combining with the split idea of VAM k-d-tree) to improve storage utilization for skewed data without much effect on query performance.

BIBLIOGRAPHY

[1] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

[2] N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R*-Tree: an efficient and robust access method for points and rectangles. In *the Proceedings of ACM SIGMOD international Conference on Management of Data*, pages 322–331, 1990.

[3] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, pages 509–517, 1975.

[4] J. L. Bentley and J. H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[5] S. Berchtold, C. Bohm, H. V. Jagadish, H.-P. Kriegel, and J. Sander. Independent Quantization: An index compression technique for high-dimensional data spaces. In *the Proceedings of the International Conference on Data Engineering*, pages 577–588, 2000.

[6] S. Berchtold, C. Bohm, and H.-P. Kriegel. The pyramid-technique: Towards breaking the curse of dimensionality. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 142–153, 1998.

[7] S. Berchtold, D. A. Keim, and H.-P. Kriegel. The X-Tree: An index structure for high-dimensional data. In *the Proceedings of International Conference on Very Large Databases*, pages 28–39, 1996.

[8] C. Bohm, S. Berchtold, and D. Keim". Searching in high-dimensional spaces: index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 33(3):322–373, 2001.

[9] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 357–368, 1997.

[10] K. Chakrabarti and S. Mehrotra. The Hybrid Tree: An index structure for high dimensional feature spaces. In *the Proceedings of the International Conference on Data Engineering*, pages 440–447, 1999.

[11] T. Chiueh. Content-Based Image Indexing. In *the Proceedings of the 20th International Conference of Very Large Data Bases*, pages 582–593, 1994.

[12] H. D. Chon, D. Agrawal, and A. E. Abbadi. Using space-time grid for efficient management of moving objects. In *the Proceedings of the 2nd ACM international workshop on Data engineering for wireless and mobile access (MobiDE)*, pages 59–65, 2001.

[13] H. D. Chon, D. Agrawal, and A. E. Abbadi. Range and kNN query processing for moving objects in grid model. *Mobile Networks and Applications(MONET)*, 8(3):401–412, 2003.

[14] P. Ciaccia, M. Patella, and P. Zezula. M-Tree: An efficient access method for similarity search in metric spaces. In *the Proceedings of International Conference of Very Large Data Bases*, pages 426–435, 1997.

[15] A. Civilis, C. S. Jensen, J. Nenortaite, and S.Pakalnis. Efficient tracking of moving objects with precision guarantees. DB *Technical Report TR-5*, 2004.

[16] D. Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[17] B. Cui, D. Lin, and K. L. Tan. Towards optimal utilization of main memory for moving object indexing. In *the Proceedings of the 10th International Conference on Database Systems for Advanced Applications*, pages 600–611, 2005.

[18] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger. The notions of consistency and predicate locks in a database system. *Communications of the ACM*, 19(11):624–633, 1976.

[19] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendible hashing - a fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

[20] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2), 1998.

[21] J. Gray. Notes on database operating systems. *Operating Systems - An Advanced Course, number 66 in Lecture Notes in Computer Science*, pages 393–481, Springer-Verlag, 1978.

[22] A. Guttman. R-Trees: A dynamic index structure for spatial searching. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 47–57, 1984.

[23] A. Henrich, H.-W. Six, and P. Widmayer. The LSD-Tree: spatial access to multidimensional point and non-point objects. In *the Proceedings of International Conference of Very Large Data Bases*, pages 45–53, 1989.

[24] K. Hinrichs. Implementation of the Grid File: design concepts and experiences. *BIT*, 25(4):569–592, 1985.

[25] A. Hutflesz, H.-W. Six, and P. Widmayer. The Twin Grid File: space otimizing access schemes. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 183–190, 1988.

[26] C. S. Jensen, D. Lin, and B. C. Ooi. Query and update efficient B+-Tree based indexing of moving objects. In *the Proceedings of International Conference on Very Large Data Bases*, pages 768–779, 2004.

[27] D. V. Kalashnikov, S. Prabhakar, S. Hambrusch, and W. Aref. Efficient evaluation of continuous range queries on moving objects. In *the Proceedings of the 13th International Conference on Database and Expert Systems Applications*, pages 731–740, 2002.

[28] N. Katayama and S. Satoh. The SR-Tree: An index structure for high-dimensional nearest neighbor queries. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 369–380, 1997.

[29] D. E. Knuth. *The Art of Computer Programming*. Addison-Wesley, 1973.

[30] G. Kollios, D. Gunopulos, and V. J. Tsotras. Nearest neighbor queries in a mobile environment. In *the Proceedings of the International Workshop on Spatio-Temporal Database Management*, pages 119–134, 1999.

[31] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. pages 261–272, 1999.

[32] M. Kornacker and D. Banks. High-concurrency locking in R-Trees. In *the Proceedings of the 21st International Conference on Very Large Data Bases,* pages 134–145, 1995.

[33] R. Kurniawati, J. S. Jin, and J. A. Shepherd. The SS$^+$-Tree: An improved index structure for similarity searches in a high-dimensional feature space. In *the Proceedings of Storage and Retrieval for Image and Video Databases,* pages 516–523, 1997.

[34] P. Lehman and S. Yao. Efficient locking for concurrent operations on b-trees. *ACM-TODS,* 6(4), 1981.

[35] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-Tree: An index structure for high-dimensional data. *The International Journal on Very Large Data Bases*, 3(4):517–542, 1994.

[36] W. Litwin. Linear hashing: A new tool for file and table addressing. In *the Proceedings of the 6th International Conference on Very Large Data Bases,* pages 212–223, 1980.

[37] D. Lomet and B. Salzberg. Access method concurrency with recovery. In *the Proceedings of the ACM SIGMOD International Conference on the Management of Data,* pages 351–360, 1992.

[38] D. B. Lomet and B. Salzberg. The hB-tree: a multi-attribute indexing method with good guaranteed performance. *ACM Transactions on Database Systems*, 15(4):625–658, 1990.

[39] C. Mohan. ARIES/KVL: A key-value locking method for concurrency control of multiaction transactions operating on B-Tree indexes. In *the Proceedings of the 16th International Conference on Very Large Data Bases*, pages 392–405, 1990.

[40] C. Mohan and F. Levine. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. In *the Proceedings of the ACM SIGMOD International Conference on the Management of Data*, pages 371–380, 1992.

[41] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, 1984.

[42] B. C. Ooi, K. L. Tan, and C. Yu. Frequent update and efficient retrieval: an oxymoron on moving object indexes? In *the Proceedings of International Web GIS Workshop*.

[43] E. J. Otoo. Balanced multidimensional extendible hash tree. In *the Proceedings of the 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 100–113. ACM, 1986.

[44] A. M. Ouksel. The interpolation-based grid file. In *the Proceedings of the 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 20–27. ACM, 1985.

[45] R. Ramakrishnan and J. Gehrke. Database management system (third edition).

[46] J. T. Robinson. The k-d-B-tree: a search structure for large multidimensional dynamic indexes. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 10–18, 1981.

[47] N. Roussopoulos, S. Kelley, and F. Vincent. Nearest neighbor queries. In *the Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 71–79, 1995.

[48] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The A-Tree: An index structure for high-dimensional spaces using relative approximation. In *the Proceedings of International Conference of Very Large Data Bases*, pages 516–526, 2000.

[49] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *the Proceedings of ACM SIGMOD International Conference on Mangement of Data*, pages 331–342, 2000.

[50] S. Saltenis, C. S. Jensen, S. T. Leutenegger, and M. A. Lopez. Indexing the positions of continuously moving objects. In *the Proceedings of ACM SIGMOD International Conference on Mangement of Data*, pages 331–342, 2000.

[51] H. Samet. The Quadtree and related hierarchical data structures. *ACM Computing Surveys*, pages 187–260, 1984.

[52] B. Seeger and H. P. Krieger. The buddy-tree: An efficient and robust access method for spatial data base systems. In *the Proceedings of International Conference on Very Large Data Bases*, pages 590–601, 1990.

[53] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R$^+$-Tree: a dynamic index for multi-dimensional objects. In *the Proceedings of the 13th International Conference of Very Large Data Bases*, pages 507–518, 1987.

[54] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *the Proceedings of the International Conference on Data Engineering*, pages 422–432, 1997.

[55] Z. Song and N. Roussopoulos. Hashing moving objects. In *the Proceedings of the 2nd International Conference on Mobile Data Management*, pages 161–172, 2001.

[56] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(4):579–589, 1991.

[57] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. *The International Journal on Very Large Data Bases*, pages 431–440, 2001.

[58] Y. Tao and D. Papadias. Time-parameterized queries in spatio-temporal databases. In *the Proceedings of the ACM SIGMOD international conference on Management of data*, pages 334–345, 2002.

[59] Y. Tao, D. Papadias, and J. Sun. The TPR*-tree: An optimized spatio-temporal access method for predictive queries. In *the Proceedings of International Conference of Very Large Data Bases*, pages 790–801, 2003.

[60] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *the Proceedings of International Conference of Very Large Data Bases*, pages 426–435, 1998.

[61] K.-Y. Whang and R. Krishnamurthy. The multilevel grid file - a dynamic hierarchical multidimensional file structure. In *the Proceedings of the 2nd International Symposium on Database Systems for Advanced Applications*, pages 449–459, 1991.

[62] D. A. White and R. Jain. Similarity indexing: Algorithms and performance. In *the Proceedings of Storage and Retrieval for Image and Video Databases*, pages 62–75, 1996.

[63] D. A. White and R. Jain. Similarity indexing with the SS-Tree. In *the Proceedings of the International Conference on Data Engineering*, pages 516–523, 1996.