Indexing for Efficient Main Memory Processing

Cui Bin

NATIONAL UNIVERSITY OF SINGAPORE

2003

Indexing for Efficient Main Memory Processing

Cui Bin

Bachelor of Engineering
Xi'an Jiaotong University, China

A THESIS SUBMITTED
FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
SCHOOL OF COMPUTING
NATIONAL UNIVERSITY OF SINGAPORE
2003

Acknowledgement

Although only one name appears on the cover, this thesis would not exist without support of various people who accompanied me during the last four years. I take this opportunity to express my thanks to all of them.

At the outset, I would like to express my appreciation to Prof. Ooi Beng Chin for his guidance, encouragement, and friendship through all my years in National University of Singapore. As my supervisor, he has constantly forced me to remain focusing on achieving my goal. His observations and comments helped me to establish the overall direction of the research and to move forward with investigation in depth. I have leaned a great deal from him about how to do and present research. Without his help, this thesis would never have been come into being.

I sincerely wish to thank Prof. Tan Kian Lee, whose valuable suggestions and comments concerning my research have not only contributed significantly to enrichment of thesis, but also shaped my research capabilities to a considerable extent.

Special thanks go to Li Hanyu, Ng Weesiong, Shen Hengtao, Wang Hongyu, Wang Wenqiang and all other colleagues in Database Group for their friendship and willing to help in various ways. Working together with them has been an

enlightening experience and a great pleasure. Further, I would like to thank the University for providing me with a scholarship for my doctoral study.

Finally, I would like to thank my beloved parents. They have always supported me in all my decisions.

CONTENTS

LIST OF FIGURES

Summary

Database management systems have become a standard tool for manipulating large volumes of data on secondary storage. To enable fast access to stored data according to content, organizational methods or structures known as indexes are used. For the past several decades, most database research has focused on large databases that do not fit into main memory. This focus is increasingly being challenged as RAM becomes cheaper and larger. With increasingly larger main memory sizes, it is now possible to store an entire database into main memory. To access data quickly in such an environment, a database requires novel memory-based structures that optimize CPU cycles and memory space. In this thesis, we shall examine some advanced techniques in main memory indexing.

To efficiently support queries of single-dimensional data in the memory environment, we optimize the BD-tree for main memory data processing. The BD-tree is essentially a B^+-tree where the leaf nodes are large-sized partitions, and each partition is organized by hash tables. To compare the BD-tree against the B^+-tree and CSB^+-tree, we present various cost models using these indexes to process exact match queries. The cost models include the L2 cache and translation lookahead

buffer (TLB) miss model, and the execution time model. We implement these structures and conduct experimental studies on them. Our analytical and experimental results show that a well-tuned BD-tree is superior in most cases.

In main memory systems, the L2 cache typically employs cache line sizes of 32-128 bytes. However, these values are relatively small compared to high-dimensional data with 32 dimensions or more. Consequently, the existing techniques (meant for use on low-dimensional data) that minimize cache misses are no longer effective. We present a novel index structure, called Δ-tree, to speed up the processing of high-dimensional queries in the main memory environment. The Δ-tree is a multi-level structure where each level represents data spaces of different dimensionalities: the number of dimensions increases towards the leaf level which contains data in all their dimensions. The remaining dimensions are obtained using *Principal Component Analysis*, which has the desirable property that the first few dimensions capture most of the information in the dataset. Each level of the tree serves to prune search space more efficiently as the reduced dimensions can better exploit the small cache line size. Moreover, distance computation on lower dimensionality is less expensive. We also propose an extension, called the Δ^+-tree, that globally clusters data space and then further partitions clusters into small regions to reduce search space. The Δ^+-tree can significantly reduce search space, and can hence lead to lower computational cost and fewer cache misses. We conduct extensive experiments to evaluate the proposed structures against existing techniques on different kinds of datasets. The experimental results show that the Δ^+-tree outperforms other indexes by a wide margin.

To facilitate the concurrent operation of multi-dimensional indexes, (e.g., the R-tree) in the main memory environment, we present a novel main memory concurrency control algorithm, called OPUS (One-Pass UpdateS). In most cases, OPUS

traverses each R-tree node *once* during an update operation. Insertions are performed top-down using *preparatory operations*, based on early node-splitting and MBR modification. By *preparing* insert operations, the proposed OPUS requires the R-tree to be traversed down once for an insertion, even if there are MBR modifications. Deletions are bottom-up, through the support of an auxiliary hashing structure on object identifiers. Modifications of spatial content are done by combining a bottom-up deletion and a localized insertion. OPUS offers several advantages towards achieving high throughput. First, the One-Pass traversal mechanism reduces lock conflict, cache misses and computational cost. Second, the localized modification constrains the affected area of update and reduces interference to other concurrent operations, thereby improving throughput. Third, for delete and update operations, the secondary hash table can eliminate the tree traversal cost involved in locating an object, and hence reduces workload on the tree and lock conflicts. We implement OPUS and run extensive sets of experiments to evaluate its performance in the main memory environment. Our results show that our proposed algorithm provides more efficient support for concurrent updates on R-trees.

We believe that our contributions have successfully addressed some of the issues of main memory indexing techniques, including the proposal of two index structures [31, 32] and a concurrency control algorithm [33]. We conduct extensive performance studies, and the results show that the proposed methods are flexible, efficient and easy to implement. Through ongoing research work, we will seek to improve the overall performance of database systems.

CHAPTER 1

Introduction

Database management systems (DBMS) have become a standard tool for manipulating large volumes of data on secondary storage. To enable fast data access according to content, organizational methods or structures known as indexes are used. Traditionally, data in a DBMS is stored on storage devices such as disks and tapes. As random access memory (RAM) gets cheaper, it becomes increasingly possible to keep the whole database memory resident, i.e., a main memory DBMS (MMDBMS) emerges as an economically viable alternative to a disk-resident DBMS (DRDBMS). Although MMDBMS can show orders-of-magnitude higher performance than DRDBMS, such a significant performance gain does not come automatically by just loading the database onto memory, but requires main memory specific optimization techniques. In the absence of expensive disk I/O cost, the main factors of memory indexes are cache behavior and computational cost. The new requirements have invalidated the design principles of indexes which have been optimized to reduce disk I/O. This calls for the design of new indexes to facilitate

memory processing. The T-tree [63] is a binary tree index proposed especially for memory processing. More recently, the sharp drop in memory prices has re-ignited the interest in MMDBMS. In the aspect of indexes, a few indexes have been proposed, such as the CSB$^+$-tree [77] and the CR-tree [55]. However, there remain some open issues to be solved. In this thesis, we shall revisit the problem of main memory indexing techniques, including single- and high-dimensional indexing and main memory concurrency control mechanisms on indexes.

1.1 The Concept of Indexing

A database index is meant to improve the efficiency of data lookup at rows of a table by a key access retrieval method. Typically, an index consists of a sequence of index entries that are stored on disk. One *index entry* for each row exists in the index, and the index is responsive to future row updates. This means that if a row is to be accessed by the value in a column, and the index provides this access, then if the row is updated in this column value, the index will change to reflect it. Index entries look like rows of a table with two columns: the *index key*, consisting of the concatenation of values from certain column values in this row (often just one column), and a *row pointer* to the disk position of the row from which this specific entry was constructed. Data entries can be actual data records or $< search\text{-}key, rid >$ pairs. Index entries are usually in sorted order by index key (although hashed access is also possible), and are then used by the system to reduce the volume of data that must be fetched and examined in response to a query. Standard lookup through an index locates one or more index entries with a given key value or range of key values, and follows entry pointers to the associated rows. In practice, large database files must be indexed to meet performance requirements.

If the structure of the information to be searched is simple, such as in one-dimensional numerical attributes or character strings, the problem may be considered simple. Database management systems provide index structures for the management of such data [30] which are well-understood and widely applied. Other kinds of applications, such as multimedia, medical imaging, and molecular biology, employ the so-called feature transformation which transforms important features or properties of data objects into high-dimensional points. Searching objects based on these features is thus a search of points in feature space. To support efficient retrieval in high-dimensional databases, many indexes have been investigated [19].

1.2 Motivation

Most research that has been carried out so far is based on conventional disk-resident DBMS. The fact that a database index normally resides on disk has far-reaching ramifications, because access to disk is extremely slow compared with memory access. This fact has an important effect on database index structures, such as B-trees. The most important aim in the design of a database index is to minimize the number of disk accesses needed to read desired data. It has been noted [15] that indexes are the primary and most direct means of reducing redundant disk I/O in DRDBMS.

Disk access speed, an extremely important aspect of DRDBMS performance, has not kept pace with the enormous improvements in CPU execution speed. To improve the efficiency of DRDBMS operations, buffer managers are used to bring data from disk to memory as needed. If the memory buffer of a DRDBMS is large enough, copies of the data will reside in memory at all times. Although such a system will perform well, it is not taking full advantage of memory. First, the

index structure is designed for disk access (e.g., B-trees), even though the data are in memory, and hence performance is not optimized. Second, applications may have to access data through a buffer manager, as if the data were on disk. For example, every time an application seeks to access a given record, its disk address have to be computed, and then the buffer manager will be invoked to check if the corresponding block is in memory. Once the block is found, the record will be copied into an application buffer, where it is actually examined. Clearly, if the record is always in memory, it is more efficient to refer to it directly at its memory address.

Therefore, main memory processing is not as simple as increasing the buffer pool size. It also differs in design from disk-based systems. To fully utilize memory, some have considered using memory as a primary device for data access, i.e., as in main memory DBMS (MMDBMS). In MMDBMS, data reside permanently in main memory, although there may be a backup copy on disk. This differs from DRDBMS, where disk data may be buffered into memory for access, but do not reside permanently in memory. Because data can be accessed directly from memory, MMDBMS can provide much better response times and transaction throughputs, as compared to DRDBMS. This is especially important for real-time applications where transactions have to be completed by specific deadlines.

We can easily identify two major trends on hardware development over the last two decades. First, CPU speed keeps on following Moore's law, i.e., doubling about every 18 months. Second, main memory keeps up with CPU development in size but not speed. Hence, the performance gap between CPU speed and memory latency is expected to widen even more in the near future. To bridge the gap, small but fast cache memories have been introduced between CPU and main memory. Nowadays, cache memories are often organized in two or three cascading levels, with

their size and latency growing with the distance from the CPU. Caches can reduce memory latency, only if requested data are found in one of the cache levels. This depends mainly on the application's memory access pattern. Hence, it becomes the responsibility of software developers to design and implement algorithms that make optimal use of the cache/memory architecture.

To efficiently support index processing in the main memory environment, an important issue which needs to be explored is cache behavior. With the ever-widening gap between CPU and memory access cost, the assumption that memory references have uniform cost is no longer valid. Therefore, improving cache behavior, i.e., minimizing L2 cache misses, becomes an imperative task in improving main memory database performance. With the emergence of new applications, such as multimedia and scientific databases, we have to manipulate high-dimensional data to facilitate fast query processing in main memory, which incurs expensive computational cost. Distance computation cost incurred in similarity comparisons contributes significantly to the overall cost of multi-dimensional or high-dimensional index structures. Therefore, the primary goals of a main memory oriented index are to exploit the L2 cache efficiently and minimize computational cost to improve overall performance. Research into main memory database systems has attracted great interest of the research community [70, 17, 76, 77, 64, 55, 26, 22], and remains to be so due to the continuous trends in hardware advancement and the demand for MMDBMS to support new applications such as Location Based Services and Online Query Answering.

1.3 Objectives and Contributions

MMDBMS needs to provide index structures for efficient management of different data. Some structures of data information are simple, such as single-dimensional numerical attributes or character strings. However, an increasing number of applications has recently emerged processing complicated application-specific high-dimensional data. In this thesis, we present our solutions to address the issues of indexing in a main memory environment. Two novel indexing methods are proposed to speed up query processing on data with different dimensionalities. Concurrency control is crucial for running real-world main memory database applications, and we propose a main memory concurrency control algorithm to support concurrent operations involving index updates. Extensive experimental studies were conducted to demonstrate the superiority of the proposed methods.

1.3.1 Exploitation of Bounded Disorder for Memory B^+-tree Indexing

We first revisit the problem of single-dimensional indexing in the main memory environment. To improve search performance, an index structure must facilitate the effective use of the L2 cache and CPU. The CSB^+-tree [77] is a promising structure that is efficient for range queries. However, it does not perform well for exact match queries (when compared to hash-based schemes). This is because the tree tends to be tall as the fanout is limited by cache line size and therefore small. In contrast, the hash-based technique can support very efficient exact match queries. This prompts us to design a structure that can combine the benefits of hashing and tree indexing. To facilitate efficient query processing on single-dimensional data, we optimize the Bounded Disorder (BD) method [67, 68] for main memory processing.

The BD-tree is essentially a B^+-tree where the leaf nodes are large-sized partitions, and each partition is organized by a hash table.

To optimize the BD-tree for memory processing, we shall apply the hash function with low computation cost to reduce the CPU overhead at leaf level. Since the use of the tree structure essentially acts as a dynamic range partitioning mechanism, the average performance should be accurately predicted under the uniformity assumption. Moreover, since data may be skewed or the hash functions may not distribute the data uniformly within each partition, we allow each bucket to have at most k chains (for some integer $k >= 1$). Since the fanout of the main memory tree structure is typically small compared to a disk-based index, we can reduce the height of the BD-tree significantly by setting a relatively large partition size. Within the partition, we set the bucket size to correspond to the cache line size. For exact match queries in the BD-tree, we only need one cache line access for each chain bucket and at most k buckets to be retrieved at leaf level. For range queries, because leaf partitions are ordered, we merely scan the partitions that overlap the desired query ranges. Additionally, we store the chained buckets contiguous with the primary buckets of the partition, and hence reduce the TLB misses of scanning for efficient range query processing.

To demonstrate the performance of the BD-tree, we shall present analytical models to evaluate the cost of exact match queries. We model the cost of a exact match query as a function of three variables: the number of L2 cache misses, TLB misses and instructions. The cost analysis consists of a cache and TLB misses model and an execution time model. The results of our experimental analysis confirm the theoretical results and show the efficiency of BD-tree.

1.3.2 Exploitation of Dimensionality Reduction for Memory High-dimensional Indexing

Many emerging database applications such as image, time series and scientific databases manipulate high-dimensional data. In these applications, the most frequently used and yet expensive operation is finding objects in a high-dimensional database that are similar to a given query object, e.g., nearest neighbor search, range search.

Query processing on high-dimensional data is computationally expensive, involving large amounts of distance calculation. Additionally, the size of a high-dimensional point can be much larger than a typical L2 cache line size. Therefore, an efficient main memory index should minimize distance computation and exploit the L2 cache effectively. We propose a novel multi-tier index structure, called the Δ-tree, that can facilitate efficient similarity search in the main memory environment. Each tier in the Δ-tree represents data space as clusters in different number of dimensions, and tiers closer to the root partition data space using fewer number of dimensions. The numbers of tiers and dimensions are obtained using the Principal Component Analysis technique. By reducing the number of dimensions at internal nodes, we can better utilize the L2 cache and reduce distance computation. We present the insertion, deletion and different kinds of queries for the Δ-tree. An extension of the Δ-tree, called the Δ^+-tree, is also proposed to further reduce search space. The Δ^+-tree method globally clusters data space and partitions the clusters into small regions before building the tree. The experimental results demonstrate the overall effectiveness of our proposed techniques.

An indirect contribution of this work is the comparative study of the various indexing structures. To our knowledge, this is the first comprehensive performance study that involves as many high-dimensional structures. The results offer insight

into the strengths and limitations of these schemes which should help researchers and practitioners pick an appropriate scheme to adopt.

1.3.3 Exploitation of One-Pass Traversal for Memory Concurrency Control

Traditional concurrency control algorithms on the R-tree are disk-based, and cannot adequately handle a high degree of concurrent accesses that involve updates. Unlike B^+-tree, there are more lock conflicts in the R-tree due to frequent tree ascents, e.g., an ascent is needed for node split or MBR modification propagation.

In this thesis, we present a novel main memory concurrency control algorithm for the R-tree, called OPUS (One-Pass UpdateS), that facilitates high throughput in the midst of frequent updates. In most cases, OPUS traverses each R-tree node *once* during an update operation. Insertions are performed top-down using *preparatory operations*, based on early node-splitting and MBR modification. By *preparing* insert operations, the proposed OPUS requires the R-tree to be traversed down once for an insertion, even if there are MBR modifications. Only when there is a split, do we need to effect the split to the parent node. Deletions are bottom-up through the support of an auxiliary structure on object identifiers. We use a hash table to locate the objects to be deleted. The bottom-up traversal is necessary to tighten the MBR of ancestor nodes. Modifications of spatial content are done by combining a bottom-up deletion and a localized insertion. While the deletion may be performed up the tree to tighten the MBR, the insertion may be performed midway up the tree to locate the bounding node. OPUS has some advantages over the existing concurrency control algorithms. First, the One-Pass traversal mechanism not only reduces lock conflict, but also computational cost and memory access (cache misses). Second, since the modification is localized, the affected area of

update is constrained locally as much as possible and causes less interference to other concurrent operations, thereby improving throughput. Third, for delete and update operations, the secondary hash table can significantly reduce workload on the tree and cut down lock conflicts.

OPUS is presented in the context of the R-tree. However, it is also applicable to other multi-dimensional structures with minor variation. We have implemented OPUS and run extensive sets of experiments to evaluate its performance in the main memory environment. The results show that OPUS outperforms the existing concurrency control algorithms.

1.4 Thesis Synopsis

The thesis is organized as follows:

- Chapter 2 presents a general introduction to the problem of main memory indexing techniques and related work.
- In Chapter 3, we study single-dimensional index processing in the main memory environment. We optimize the BD-tree for memory processing, and present a cost analysis and an experimental evaluation of the BD-tree against the B^+-tree and CSB^+-tree.
- In Chapter 4, we propose a novel multi-tier index structure, called the Δ-tree, for high-dimensional indexing. An extension of the Δ-tree, called the Δ^+-tree, is also proposed to further improve the efficiency of search performance. Extensive experiments were conducted to evaluate the performance of proposed methods with other existing high-dimensional index structures.
- Chapter 5 provides a description of our novel main memory concurrency con-

trol algorithm, called OPUS. OPUS has been specially designed to facilitate high throughput in the midst of frequent updates. The performance study of OPUS is also presented.

- We conclude our work in Chapter 6 with a summary of our contributions. We also discuss some limitations in this research and suggest directions for future work.

CHAPTER 2

Preliminaries

Database management systems have become a standard tool for manipulating large volumes of data. To enable fast data access, index structures are used popularly. Advances in hardware-related technologies promise to incur the demanding for the advanced index technologies. In this chapter, we shall discuss background information pertaining to the topic in this thesis, e.g. memory hierarchical structure, single/high-dimensional index structures and concurrency control algorithms.

2.1 Hardware Architecture

Custom computer systems have experienced tremendous improvements in the past two decades. However, this growth has not been equally distributed over all aspects of system performance and capacity. The CPU speeds have been increasing at a much faster rate than memory speed, and the gap between CPU and memory speed is expected to widen in the future. This affects all computer systems, making it

increasingly difficult to achieve high processor efficiency. The way to reduce the memory latency for applications has been to incorporate caches in the memory subsystem. Caches are high-speed buffers that are used to hold items in current use. They appear in CPUs as buffers for main memory, in the same way as main memory for disk.

Since the data values must be loaded into CPU before computations may take place, the topic of data motion becomes important. As shown in Figure 2.1, data actually is present at different levels of memory locations during execution, and efficient movement between these levels, known as the memory hierarchy, becomes a driving force in performance programming.

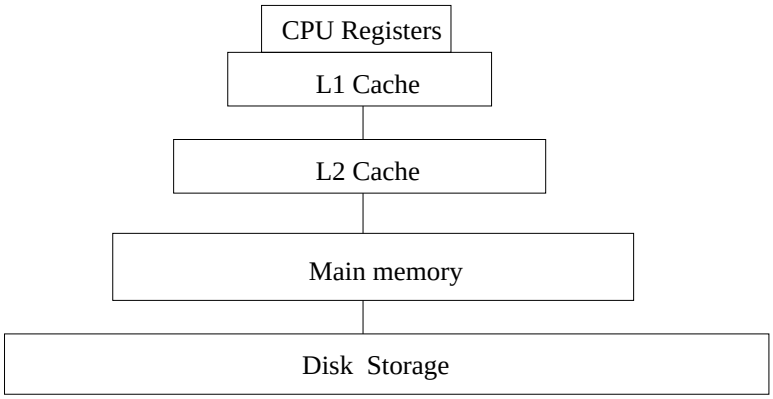


Figure 2.1: The structure of memory hierarchy

Register

Registers represent the hierarchy level closest to the CPU. Registers are usually incorporated as part of the chip architecture; consequently, the time needed to access data from a register is the shortest. On the other hand, because the registers are incorporated into the chip, the "real estate" needed for registers is limited, thereby limiting the number of registers available.

In today's common machines, a data item may be retrieved from a register into

the CPU during a single cycle. Because the access time is the fastest, fetching data from a register will not cause a delay in the instruction pipeline (unless the value in the register is one that is still being computed). The fastest computer applications are written so that needed data reside in registers as often as possible. There are not too many computers that have more than 64 general-purpose registers.

Cache

A cache is a small buffer of relatively fast memory that is used to hold data that has recently been read from memory. The cache sits below the register in the hierarchy, but above the main memory. Caches are of limited size, but can hold more data than registers. Data values that have been loaded into registers will most often be in blocks that have already been loaded into the cache, and blocks that have been loaded into cache reflect blocks of data sitting in the main memory.

The access time for data from the cache is very fast when compared to the main memory. A cache is designed to be a low-latency, high-bandwidth module. Most systems are organized with more than one level of caches. For example, there may be small caches (used for instructions, data, or both) called level-1 caches (L1 cache) located on the microprocessor chip, and a much larger off-chip cache called a level 2 (L2) cache. Typically, data residing in the L1 cache may be accessed in several clock cycles, while a L2 miss incurs more than 100 clock cycles when data is required to be fetched from main memory. Therefore, effective utilization of L2 caches becomes an important optimization of system performance.

Main Memory

The main memory level is the next level below the cache. Memory serves both as a repository for all the data and code in a program and as an interface for I/O

for the applications. Memory has a higher latency than the cache, and a lower bandwidth. Data are stored and referenced in a chip in blocks that are organized as a 2-dimensional array of rows and columns. The time to access memory is broken up into two parts: the access time, which is the time it takes to access a memory reference; and the cycle time, which is the time required between requests to memory.

Memory can be constructed from different kinds of memory chips: the dynamic random access memory chip (DRAM) or the static RAM (SRAM). The *dynamic* part of a DRAM chip indicates the characteristic of the chip that each row in the memory must be accessed within a short time span or else the data are lost. This forces the DRAM chip to write back data after it is read. Thus the cycle time is increased, because the chips may be busy refreshing the data. This is opposed to the *static* RAM, which, through the use of more hardware, does not need to be refreshed. The additional hardware allows for a faster overall memory reference time, because there is a reduction in the cycle time. However, the cost of SRAM chips is significantly higher than that of DRAM chips. Also, SRAM chips hold much less data than DRAM chips. DRAM chips are the most popular for main memory modules. SRAM chips are used in specialized hardware needing fast access memory and are the primary kind of memory chip used for caches.

Translation Lookahead Buffer

The translation lookahead buffer (TLB) is an essential part of the modern processor. Poor memory reference locality causes TLB thrashing, which increases the cost of reading data from the memory. When the processor issues a read or write instruction from memory, it must first translate the logical memory address seen by the user program to a physical memory address understood by the memory

hardware. In order to reduce the cost of this translation, processors contain a fast associative memory, the TLB, of recently used address translations. If the address being referred to cannot be mapped by the TLB, a TLB miss has occurred and the mapping is resolved by referring to the data structures of the operating system. Currently the cost of handling a TLB miss in modern machines can be around 100 cycles. TLBs usually contain from 64 up to hundreds of TLB entries, each covering the mapping of one virtual memory page, typically 4 or 8KB. The more pages an application uses (which is also dependent on the often configurable size of the memory pages), the higher the probability of TLB misses. Depending on the implementation and hardware architecture, TLB misses can be more costly even than a main-memory access. Moreover, as address translation often requires accessing some memory structure, this can in turn trigger additional cache misses.

2.2 Traditional Indexing Techniques

2.2.1 Single-dimensional Index Structures

Order preserving indexes are commonly used to index single-dimensional data, most of which are tree structures, e.g. the B-tree. These structures can support both exact match query and range query efficiently. Except tree structures that preserve some natural ordering in the data, there are other methods that randomize the data, such as hashing method. However, although random accessing of the hashing methods is fast, sequential accessing is slow. There are also indexes using a combination of hashing and tree indexing. In this section, we introduce some well-known indexes.

Array

Array is the simplest index structure. It uses minimal space, providing that the size is known in advance or that growth is not a problem (e.g. that one can somehow use the underlying mapping hardware of virtual memory to make the array grow gracefully). The biggest drawback of the array is that data movement is O(N) for each update, so it appears impractical for anything but a read-only environment.

The B-tree

The B-tree [6] is a multi-way balanced tree and is well suited for disk-based database systems since it requires few node accesses to retrieve a value. For disk-based databases, I/O access dominates the overall operation cost. Most of the database systems use the B^+-tree [30](see Figure 2.2), a variant of the B-tree, which keeps all the data in the leaves of the tree, while internal nodes are used only to direct the search to the target leaf nodes. Thus, it is not space efficient compared with B-tree. However, the B^+-tree has a larger internal node fanout than the B-tree for the same node size, furthermore the insertion and deletion are more efficient.

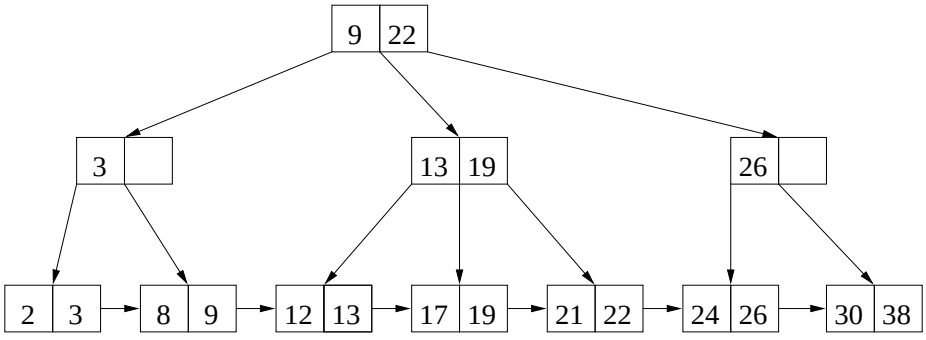


Figure 2.2: The structure of B-tree

Hashing Structures

There are some hashing methods that randomize the data, such as Chained Bucket Hashing [56], Extendible Hashing [36], Linear Hashing [66], etc. A version of hashing method classifies keys using a suffix of the binary representation of the keys. Random accessing is fast but sequential accessing is not. By using a prefix instead, sequential accessing is possible but not efficient.

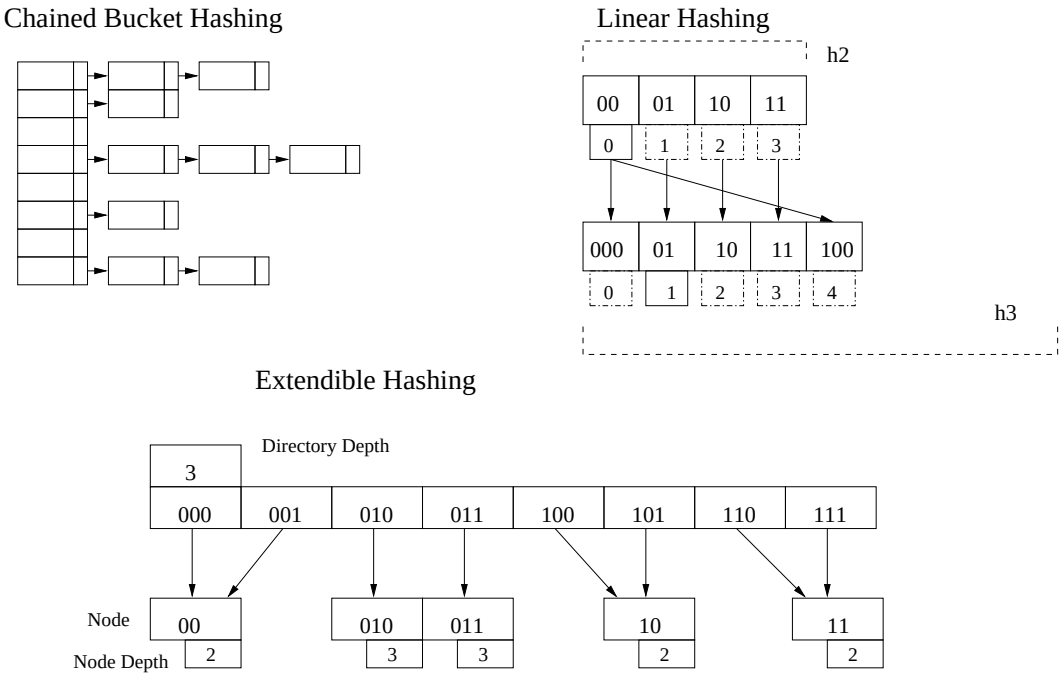


Figure 2.3: Hashing-based indexes

Chained Bucket Hashing [56] is a static structure used both in main memory or on disk (see Figure 2.3). It is very fast because it is a static structure - it never has to reorganize its data. But, this advantage is also its disadvantage; because it is static, it may have very poor behavior in a dynamic environment because the size of the hash table must be known or guessed before the table is filled. If the estimated size is too small, then performance can be poor, and if the estimated size is too large, then much space is wasted. At best, there is some space wasted, since each data item has a pointer associated with it.

Extendible Hashing [36] employs a dynamic hash table that grows with the data, so the table size does not need to be known in advance (see Figure 2.3). A hash node contains several items and splits into two nodes when an overflow occurs. The directory grows in powers of two, doubling whenever a node overflows and has reached the maximum depth for a particular directory size. One problem with Extendible Hashing is that any node can cause the directory to split, so the directory can grow to be very large if the hash function is not sufficiently random.

Linear Hashing [66] also uses a dynamic hash table, but it is quite different from Extendible Hashing (see Figure 2.3). A linear hash table grows linearly as it splits nodes in predefined linear order - as opposed to Extendible Hashing, which splits the nodes that overflow. The decision to split a node and extend the directory in a controlled fashion can be based on criteria other than overflowing nodes, providing several advantages over the uncontrolled splitting of Extendible Hashing. At first, the buckets can be ordered sequentially, allowing the bucket address to be calculated from a base address - no directory is needed. Secondly, the event that triggers a node split can be storage utilization, keeping the storage cost constant for a given number of elements.

The above hashing methods are dynamic and provide access to the primary page of any bucket in exactly one disk access. But they are not indexed-sequential data structure (ISDS) because sequential accessing can not be performed quickly: the address of the successor of R, a record in the file, is not related to the address of R. The hashing methods can be made order-preserving in a trivial way, also called range partition hashing, e.g. by using the hash function h(R)=R/S where R is an integer and S is a fixed partition factor. Both random and sequential accessing will then be possible, but the performance is significantly affected by the data distribution.

Bounded Disorder Structure

In [67], the authors proposed a key associative access method, called the bounded disorder (BD) method, which uses a combination of hashing and tree indexing. The idea explored in that paper is to increase the ratio of file size to index size by using hashing as a way to improve the distribution of keys in a multi-bucket node and to provide a fast way of accessing data. Hence, a BD file index can be sufficiently small so that it can fit entirely in main memory. Random access can then be accomplished in close to one disk access to read the appropriate pages of the data node, while the B-tree may require two disk accesses per query. The data nodes are the leaf nodes of an index tree. Hence they are in key sequence and key sequential access can also be supported.

2.2.2 High-dimensional Index Structures

In this section, we will give an introduction about the basics of query processing in multi/high-dimensional data spaces. We start with some definitions which formalize the problem description. Then, we will give an overview of well-known index structures for multi-dimensional query processing.

First of all, we will introduce some notions and formalize the problem description. We define the notion of the high-dimensional database, distance metrics and variant query types. Any combination between distance metrics and query types can be deployed depending on the different applications.

We assume that data objects are feature-transformed into points of a vector space with a fixed, finite dimension D. Therefore, a database is a set of points in a D-dimensional data space. The data space is a subset of R^D. Typically the data space is restricted to the unit hypercube $[0..1]^D$ for analysis. In the database, insertions of new points and deletions of points are possible and can be

handled efficiently. The number of point objects currently stored in our database is abbreviated as n. In some applications, objects cannot be mapped into feature vectors, however, there exists some notion of similarity between objects that can be expressed as a metric distance between objects. Thus, the objects can also be embedded in a metric space, and these object distances can directly be used for query evaluation. Our research work on multi-dimensional database is restricted to vector spaces with finite number of dimensions.

All neighborhood queries are based on the notion of the distance between two points P and Q in the data space. Depending on the application to be supported, several metrics to define the distances are applied as follows.

- *Manhattan distance* : The distance between two points measured along axes at right angles. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $|x_1 - x_2| + |y_1 - y_2|$. This is easily generalized to higher dimensions. Manhattan distance is often used in integrated circuits where wires only run parallel to the X or Y axis.

- *Euclidean distance* : The straight line distance between two points. In a plane with p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $\sqrt{((x_1 - x_2)^2 + (y_1 - y_2)^2)}$. In D dimensions, the Euclidean distance between two points p and q is $\sqrt{\sum_{i=1}^{D}(p_i - q_i)^2}$ where p_i (or q_i) is the coordinate of p (or q) in dimension i.

- L_m *distance* : The generalized distance between two points. In a plane with point p_1 at (x_1, y_1) and p_2 at (x_2, y_2), it is $(|x_1 - x_2|^m + |y_1 - y_2|^m)^{\frac{1}{m}}$. This is easily generalized to higher dimensions. Euclidean distance is L_2 distance. Rectilinear or Manhattan distance is L_1 distance. L_∞ distance is $\max(|x_1 - x_2|, |y_1 - y_2|)$.

Among them, the Euclidean metric L2 is the most common metric. Queries

using the L2 metric are hyper-sphere shaped. Queries using the maximum metric or the Manhattan metric are hypercubes and rhomboids, respectively.

Typically high-dimensional indexes have been designed to support different types of queries, such as point query, range query, K-nearest neighbor query, etc. Now we briefly explain these types of queries and assume that database is a set of points in a d-dimensional data space.

- *Point query*: "find all objects whose attributes are identical to the query point" and can be formally expressed with respect to the database DB as follows:

  $PQ(DB, Q) = \{P \in DB | P = Q\}$

- *Range query*: "find all objects in the database which are within a given distance ϵ from a given object". Similarity range queries can be formally expressed as follows:

  $RQ(DB, Q, \epsilon) = \{P \in DB | Dist(P, Q) < \epsilon\}$

  where $Dist()$ is the similarity measure applied. We note that $Dist()$ is highly application-dependent, and may have different metrics. A window query specifies a rectangular region in data space from which all points in the database are selected. Simply, we can regard the window query as a range query around the center point of the window using a weighted maximum metric.

- *K-nearest neighbor* (KNN) *query*: "find the k-most similar objects in the database which are closest in distance to a given object". KNN queries can be formally expressed as follows:

  $KNNQ(DB, Q, k) = \{P_1 ... P_k \in DB | \forall P' \in DB \setminus \{P_1 ... P_k\} \wedge Dist(P', Q) > Dist(P_i, Q)\}$

In high-dimensional databases, due to the low contrast in distance, we may have more than k objects with similar distance to the query object Q. In such a case, the problem of ties is resolved by randomly picking enough objects with similar distance to make up k answers or allowing more than k answers.

There are other variations of queries, e.g., the reverse nearest neighbor queries (RNN), ranking queries. A reverse nearest neighbor query finds all the points in a given database whose nearest neighbor is a given query point [57]. RNN queries arise because certain applications require information that is most interesting or relevant to them. RNN queries can be extended to RKNN queries. RNN queries generally require different data structure support and query processing strategies. In a ranking query, the user specifies neither a range in the data space nor a result set size. Even though, the ranking query is more related to nearest neighbor queries than to range queries, because the first answer of a ranking query is always the nearest neighbor. The user has then the possibility to ask for further answers. Upon this request, the second nearest neighbor is reported, then the third and so on. The user decides after examining an answer if he needs further answers or not. Ranking queries can be especially useful in the filter step of a multi-step query processing environment. Here, the refinement step usually takes the decision whether the filter step has to produce further answers or not. We shall not deal with these two queries in this thesis and a comprehensive survey on high-dimensional indexes and query processing strategies, with more vigorous definition of terms, can be found in [16].

Many multi-dimensional indexes have been proposed to support applications in spatial and scientific databases. Multi-dimensional access methods can be classified into two broad categories based on the data types that they have been designed to support: Point Access Methods (PAMs) and Spatial Access Methods (SAMs).

While PAMs have been designed to support range queries on multi-dimensional points, SAMs have been designed to support multi-dimensional objects with spatial extents. However, all SAMs can function as PAMs, and since high-dimensional data are points, we shall not differentiate SAMs from PAMs. On the other hand, the multi-dimensional indexes can be further classified into two classes: hierarchical, data organizing structures such as R-trees and space organizing structures such as Multi-dimensional Hashing and grid-files. We will concentrate here on the first class, the data organizing structures, since hashing-based methods do not play an important role in high-dimensional indexing. To our best knowledge, there exists no serious approach to solve the high-dimensional indexing problem with a space organizing structure.

Many hierarchical multi-dimensional indexes are based on the principle of hierarchical clustering of the data space. Structurally, they are similar to the B^+-tree [6, 7, 30]: The data vectors are stored in leaf nodes such that spatially adjacent vectors are likely to reside in the same node. The leaf nodes are organized in a hierarchically structured directory. Each directory (internal) node points to a set of subtrees. Usually, the structure of the information stored in leaf nodes is different from the structure of the internal nodes. The root node serves as an entry point for query and update processing. The index structures are height-balanced. The wide acceptance of the tree-like structure is due to its elegant height-balanced characteristic, where the I/O cost is somewhat bounded by the height of the tree. Further, each node is in the unit of a physical page, thereby making it ideal for disk I/O. Thus, it has become an underlying structure for many multi-dimensional indexes.

Here we will introduce and briefly discuss some most popular index structures for multi/high-dimensional data spaces, such as the R-tree [44], the X-tree [14],

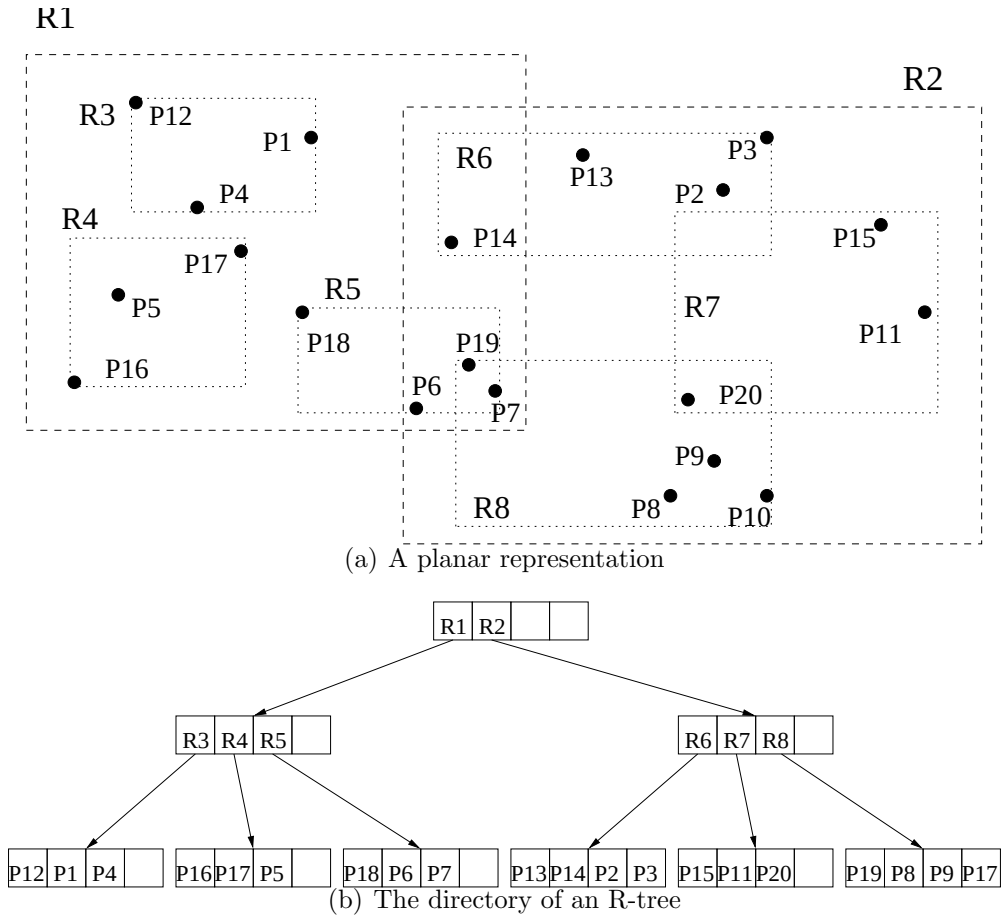
the TV-tree [65], the M-tree [29], idistance [90], VA-files [88]. There are many other multi-dimensional index structures which are excluded from our discussion, e.g. the kd-tree [9], the k-d-B-tree [79], the hBtree [69], the Hilbert-R-tree [52], the SS-tree [89], the SR-tree [54], the VAMSplit R-tree [48], the LSD^h-tree [46], the A-tree [82], the Pyramid tree [10], the iminmax [74], the IQ-tree [12], the space filling curve [80], the Gray Codes [37] and the grid-files [73, 60, 39], etc.

The R-tree

The R-tree [44] is a multi-dimensional generalization of the B^+-tree, that preserves height-balance. Like the B^+-tree, node splitting and merging are required for inserting and deleting objects. The R-tree has received a great deal of attention due to its well-defined structure and the fact that it is one of the earliest proposed tree structures for indexing spatial objects. Figure 2.4 illustrates the structure of an R-tree. An entry in the R-tree leaf node consists of an object-identifier of the data object and a D-dimensional bounding box which bounds its data objects. In a non-leaf node, an entry contains a child-pointer pointing to a lower level node in the R-tree and a bounding box covering all the boxes in the lower nodes in the subtree.

In order to locate all the objects which intersect a given query box, the search algorithm descends the tree starting from the root. The algorithm recursively traverses down the subtrees of those bounding boxes that intersect the query box. When a leaf node is reached, bounding boxes are tested against the query box and their objects are fetched for testing whether they intersect the query box.

To insert an object, the tree is traversed and all the boxes in the current non-leaf node are examined. The constraint of least coverage is employed to insert an object: the box that needs least enlargement to enclose the new object is selected.



(a) A planar representation

(b) The directory of an R-tree

Figure 2.4: The structure of R-tree

The nodes in the subtree indexed by the selected entry are examined recursively. Once a leaf node is reached, a straightforward insertion is made if the leaf node is not full. However, the leaf node needs splitting if it is full before the insertion is made. For each node that is traversed, the covering box in the corresponding parent node is readjusted to bound tightly the entries in the node. For a newly split node, an entry with a covering box is inserted in the parent node if there is sufficient space in the parent node. Otherwise, the parent node will be split and the process may propagate to the root.

To remove an object, the tree is traversed to locate the leaf containing the object. The deletion of an object may cause the leaf node to underflow and may

also cause further deletion of nodes in the upper levels. Deletion may change the bounding box of entries in the ancestor nodes. Hence adjustment of these entries is required.

For searching, the decision to visit a subtree depends on whether the covering box intersects the query box. It is quite common for several covering boxes in an internal node to intersect the query box, resulting in the traversal of several subtrees. Therefore, the minimization of overlaps of the covering boxes as well as the coverage of these boxes is of primary importance in constructing the R-tree. The overlap between bounding boxes becomes more severe as the dimensionality increases, and precisely it is due to this weakness that the R-tree does not perform well for high-dimensional databases.

Minimization of both coverage and overlap is crucial to the performance of the R-tree. However, it may not be possible to minimize both at the same time [15]. A balancing act must be performed such that the near optimal of both these minimizations can produce the best result. The R*-tree [8] introduces an additional optimization objective concerning the margin of the covering boxes, e.g. squarish covering boxes. Since clustering objects with little variance in the lengths of the edges tends to reduce the area of the cluster's covering box, the criterion, which ensures quadratic covering boxes, is used in the insertion and splitting algorithms. Intuitively, squarish covering box is compact and hence facilitates packing and reduces overlap.

Dynamic hierarchical spatial indexes are sensitive to the order of the insertion of data. A tree may behave differently for the same data set but with a different sequence of insertions. Data objects inserted previously may result in a bad split in the R-tree after some insertions. Hence it may be worthwhile to do some local reorganization, which is however expensive. The R*-tree deletion algorithm

provides reorganization of the tree to some extent, by forcing the entries in underflowed nodes to be inserted from the root. Performance studies have shown that the deletion and reinsertion can improve the R-tree performance significantly. The reinsertion increases the storage utilization; but this can be expensive when the tree is large.

Other variants of R-trees have been proposed, including the R^+-tree [85] and the buddy-tree [84]. The R^+-tree and the buddy-tree were proposed to overcome the problem of the overlapping covering boxes of internal nodes of the R-tree for low-dimensional databases. It should be noted that the R-tree and its variants can be used for high-dimensional indexing and similarity search. The R-tree, due to its balanced structure, has made it one of the most popular structures for extension for supporting other form of retrievals, including distance based high-dimensional search.

The X-tree

The R-trees have primarily been designed for the management of spatially extended 2-dimensional objects, but also have been used for high-dimensional data. Empirical studies [14, 89], however, showed a deteriorated performance of the R-trees for high-dimensional data. The major problem of R-tree-based index structures in high-dimensional data spaces is the overlap. In contrast to low-dimensional spaces, there exists only few degrees of freedom for splits in the directory. In fact, in most situations there exists only a single *good* split axis. An index structure that does not use this split axis will produce highly overlapping MBRs in the directory and thus show a deteriorated performance in high-dimensional spaces. Unfortunately, this specific split axis might lead to unbalanced partitions. In this cases, a split should be avoided in order to avoid underfilled nodes.

The X-tree [14] is an extension of the R-tree which is directly designed for the management of high-dimensional objects and based on the analysis of problems arising in high-dimensional data spaces. In the general split operation of R-trees, the index starts with a single data page covering almost the whole data space and inserts data items. If the page overflows, the index splits the page into two new pages. Most of the time, some MBRs of the partitions will span the whole data space. Obviously, this leads to a high overlap with the other partitions, and hence reduces the performance of the tree. In this case the X-tree does not split and creates an enlarged directory node instead - a supernode. This guarantees an overlap free directory. The higher the dimensionality, the more supernodes will be created and the larger the supernodes become. The split algorithm works as follows: in case of a data page split, the X-tree uses the R*-tree split algorithm or any other topological split algorithm. In case of directory nodes, the X-tree first tries to split the node using a topological split algorithm. If this split would lead to highly overlapping MBRs, the X-tree applies the overlap-free split algorithm, e.g. creating a supernode.

The X-tree shows a high performance gain compared to the R-trees for all query types in medium-dimensional spaces. For low-dimensional data, the X-Tree shows a behavior almost identical to the R-trees. For high-dimensional data, the X-tree also has to visit such a large number of nodes, but a linear scan in supernode is much less expensive.

The TV-tree

The TV-tree [65] is designed especially for real data that are subject to the Principal Component Analysis (PCA) [50], a mapping which preserves distances and eliminates linear correlations. Such data yield a high variance and therefore, a good

selectivity in the first few dimensions while the last few dimensions are of minor importance for query processing. Indexes storing PCA-transformed data tend to have the following properties:

- The last few attributes are never used for cutting branches in query processing. Therefore, it is not useful to split the data space in the corresponding dimensions.
- Branching according to the first few attributes should be performed as early as possible, i.e. in the topmost levels of the index. Then, the extension of the regions of lower levels (especially of data pages) is often zero in these dimensions.

Regions of the TV-tree are described by so-called Telescope Vectors (TV), i.e. vectors which may be dynamically shortened. A region has inactive dimensions and active dimensions. The inactive dimensions form the greatest common prefix of the vectors stored in the subtree. Therefore, the extension of the region is zero in these dimensions. In the α active dimensions, the region has the form of an L_p-sphere where p may be 1, 2 or ∞. The region has an infinite extension in the remaining dimensions which are supposed either to be active in the lower levels of the index or to be of minor importance for query processing.

The region description comprises floating point values for the coordinates of the center point in the active dimensions and one float value for the radius. The coordinates of the inactive dimensions are stored in higher levels of the index (exactly in the level where a dimension turns from active into inactive). To achieve a uniform capacity of directory nodes, the number of active dimensions is constant in all pages. The concept of telescope vectors increases the capacity of the directory pages. It was experimentally determined that a low number of active dimensions

($\alpha = 2$) yields the best search performance.

The insert-algorithm of the TV-tree chooses the branch to insert a point according to some criteria, such as *minimum increase of the number of overlapping regions*, *minimum increase of the radius*. To cope with page overflows, the authors proposed to perform a re-insert operation, like in the R*-tree. The split algorithm determines the two seed-points (seed-regions in case of a directory page) which have the least common prefix or the maximum distance. The objects are then inserted into one of the new subtrees using the above criteria for the subtree choice.

The TV-tree was proposed to avoid the dimensionality curse problem. It uses a variable number of dimensions for indexing, adapting to the number of objects to be indexed, and to the current level of the tree. Since the TV-tree indexes the active dimensions and the number of active dimensions is usually small, the method saves space and leads to a larger fan-out. As a result, the tree is more compact and shallower and performs better than the R^*-tree. Even though the TV-tree's internal nodes have fewer dimensions, the algorithm is the same as the R-tree based algorithm. Furthermore, the number of active dimensions of the TV-tree can be large, which means that it still suffers from the dimensional scalability problem.

The M-tree

In [29], the authors proposed the height-balanced M-tree to organize and search large datasets from a generic metric space, where object proximity is defined by a distance function, satisfying the positivity, symmetry, and triangle inequality postulates. The M-tree is a balanced tree, able to deal with dynamic data files, and as such it does not require periodical reorganizations. The M-tree partitions objects on the basis of their relative distances, as measured by a specific distance function,

and stores these objects into fixed-size nodes, which correspond to constrained regions of the metric space.

In an M-tree, leaf nodes store all indexed objects, whereas internal nodes store the routing objects. For each routing object O_r, there is an associated pointer, denoted ptr(T(O_r)), that references the root of a sub-tree, T(O_r), called the covering tree of O_r. All objects in T(O_r) are within the distance r(O_r) from O_r, r(O_r) > 0, which is called the covering radius of O_r. Finally, a routing object O_r is associated with a distance to P(O_r), its parent object, that is the routing object which references the node where the O_r entry is stored. An entry for a database object O_j in a leaf node is quite similar to that of a routing object, but no covering radius is needed, and the pointer field stores the actual object identifier, which is used to provide access to the whole object possibly resident on a separate data file.

Since the design of M-tree is inspired by both principles of metric trees and database access methods, performance optimization concerns both CPU (distance computations) and I/O costs. Experimental results show that the M-tree yields good scalability with the size of data sets from generic metric spaces.

The Slim-tree

In [51], the authors proposed a new dynamic tree for organizing metric datasets, named Slim-tree. The Slim-tree uses the triangle inequality to prune distance calculations needed to answer similarity queries over objects in metric spaces, which is similar to the M-tree. The proposed insertion algorithm uses new policies to select the nodes where incoming objects are stored. When a node overflows, the Slim-tree uses a Minimal Spanning Tree to help with the split. The new insertion algorithm leads to a tree with high storage utilization and improved query performance. The Slim-tree tackles the overlap problem between nodes and proposes

"fat-factor" method to minimize it. The authors also presented a new visualization tool for interactive data mining and for the visual tracking of the behavior of a tree under updates. Although the Slim-tree performs similarly for distance calculations compared with the M-tree, it reduces the number of disk accesses.

The Pyramid-tree

The basic idea of the Pyramid Technique [11] is to transform the D-dimensional data points into 1-dimensional values and then store and access the values using a conventional index such as the B^+-tree. It splits the data space into 2D pyramids, which share the center point of the data space as their top and have (D-1)-dimensional surface of the data space as their base. The location of a point within its pyramid is indicated by a single value, which is the distance from the point to the center point according to dominant dimension. To perform a range query, the pyramids that intersect the search region are first obtained, and for each pyramid, a subquery range is worked out. Each subquery is then used to search the B^+-tree. For each range query, 2D subqueries may be required, one against each pyramid.

The Omni-technique

The Omni-technique [38] was proposed to be used with existing index structures and reduce the distance computational cost. The Omni-technique chooses a number of objects from the dataset as "foci", from which distance calculations to answer queries can be pruned. The foci can be used as global reference points to any object in the database, and improve any underlying index structure. The authors also proposed a method to define and select an adequate number of objects to be used as Omni-foci, with the best tradeoff between increasing memory requirements and

decreasing distance computations. By applying the Omni-concept with sequential scan and R-tree, they can achieve up to ten times fewer calculations, disk accesses and overall time.

The VA-file

In [88], the authors described a simple vector approximation scheme, call VA-file. The VA-File divides the data space into 2^b rectangular cells where b denotes a user specified number of bits. For each dimension i, a small number of bits (b_i) is assigned (b_i is typically between 4 and 6), and 2^{b_i} slices along the dimension i are determined in such a way that all slices are equally full. These slices are kept constant while inserting, deleting and updating data points. Let b be the sum of all b_i. Then, the data space is divided into 2^b hyper-rectangular cells, each of which can be represented by a unique bit-string of length b. Each data point is approximated by the bit-string of the cell into which it falls. Depending upon the accuracy of the data points and the number of bits chosen, the approximation file is 4 to 8 times smaller than the vector file. The VA-File overcomes the difficulties of high dimensionality, and has been shown to perform well for disk-based systems as it reduces the number of random I/Os.

Nearest neighbor queries are performed by scanning the entire approximation file, and by excluding the vast majority of vectors from the search (filtering step) based only on these approximations. When searching for the nearest neighbor, the entire approximation file is scanned and upper and lower bounds on the distance to the query can easily be determined based on the rectangular cell represented by the approximation. Analogously, we can define a filtering step when the k nearest neighbor must be retrieved. A critical factor of the search performance is the selectivity of this filtering step since the remaining data objects are accessed in

the vector file and random I/O operations occur. After the filtering step, a small set of candidates remain. These candidates are then visited in increasing order of their lower bound on the distance to the query point Q, and the accurate distance to Q is determined. However, not all candidates must be accessed. Rather, if a lower bound is encountered that exceeds the (k-th) nearest distance seen so far, the VA-file method stops. The VA-file has been shown to perform well for disk-based systems as it reduces the number of random I/Os. However, it incurs higher computational cost making it less attractive for main memory databases: besides computing the actual distances of candidate points, it has to decode the bit-string and compute all the lower and some upper bounds on the distance to the query point.

The iDistance

In [90], the authors presented an efficient method for KNN search in a high-dimensional space, called iDistance. iDistance partitions the data and selects a reference point for each partition. The data points in each cluster are transformed into a single dimensional space based on their similarity with respect to a reference point. It then index the distance of each data point to the reference point of its partition. Since this distance is a simple scalar, with a small mapping effort to keep partitions distinct, it is possible to used a standard B^+-tree structure to index the data and KNN search be performed using one-dimensional range search. In the iDistance, the authors employ two data structures:

- A B^+-tree is use to index the transformed points to facilitate speedy retrieval. We use the B^+-tree since it is available in all commercial DBMS. In the B^+-tree, leaf nodes are linked to both the left and right siblings. This is to facilitate searching the neighboring nodes when the search region is gradually

enlarged.

- An array is also required to store the reference points and their respective nearest and farthest radii that define the data space.

The choice of partition and reference point provides the iDistance technique with degrees of freedom that most other techniques do not have. The iDistance technique can permit the immediate generation of a few results while additional results are searched for. In other words, it is able to support online query answering: an important facility for interactive querying and data analysis. The weakness of iDistance is that the transformation to 1-D space may introduce too much information loss, and hence reduce the prune efficiency.

2.2.3 Concurrency Control Algorithms

Multiprogramming or concurrent processing of transactions in computer systems is required to take advantage of multiple processors and CPU-I/O overlap to attain a higher transaction throughput. However, interactions among the database transactions can cause the database state to become inconsistent, even when the transactions individually preserve correctness of the state, and there is no system failure. In this case, the requirement for concurrency control arose to ensure correctness when a shared database is updated by multiple transactions concurrently.

Concurrency control is the activity of preventing harmful interference between concurrent operations. Concurrency search index structures, especially concurrent B^+-trees, are most often used in database systems. For example, applications such as airlines, telecommunications, banks and real-time databases require *thousands* of transactions per second; each transaction usually consists of several data record accesses, and most of them through index structure. If each transaction requires

0.1 second, the multiprocessing level will be more than 100. At such high multiprocessing levels, restrictive serialization techniques result in a serialization bottleneck [49]. In answer to this challenge, researchers have proposed a variety of concurrency control algorithms for different indexes [49, 62, 25, 58]. We can distinguish these algorithms into two basic classes, i.e. lock-coupling and link-type technique.

Concurrency Control on B$^+$-tree

Lock-coupling is a basic technique for the concurrency control on B$^+$-trees: a lock on an internal node n is never released until a lock on a child of n is obtained. Search operations use read-lock, while the insert and delete operations use write-lock. The write-lock conflicts with any other locks. The main issue regarding the performance of this kind of algorithms is what kind of lock is obtained and how long it is hold. The naive lock-coupling algorithm involves locking index nodes heavily, and can be improved by the optimistic descent algorithm [7], because indexes rarely restructure for single update operation. One variant of lock coupling is the optimistic descent algorithm. Upon insertion, the algorithm first traverses to a leaf with the optimistic assumption that the target leaf is a safe node. If the assumption is not true, it tries again without the optimistic assumption.

The radical linking approach, B-link tree was originally introduced in [62]. Instead of avoiding possible conflicts by lock coupling, the tree structure is modified so that the search has the opportunity to compensate for a missed split. The crucial addition is the right link - a pointer to the right sibling node on the same level. When a node is split, the new right sibling that is created is inserted into the right link chain directly to the old node. The effect is that all nodes at the same level are chained together through right links. Searching in a B-link tree can therefore be performed without lock coupling. In a descent, the lock on the parent

node is released before the lock on the child node is granted. However, the node may be split before the lock is achieved. Consequently, the right sibling node has to be searched based on the fact that the highest key on the node is less than the search key. When a leaf has to be split due to an insertion, it avoids lock coupling its parent [81]. As soon as the page has been split and by inserting the new right sibling into the right link chain, acquiring the lock on the parent and possibly moving right if the concurrent split has occurred before the split of the leaf node, the insertion process can drop the lock on the leaf that was overflowing and then acquire a lock on the parent, possibly moving right to compensate for concurrent splits and splitting up the tree recursively. This linking strategy offers very high concurrency because search and insert processes only need to lock one node at a time.

Concurrency Control on Multi-dimensional Indexes

Non-standard database applications such as robotics, computer vision, CAD/CAM, and geographic data processing are becoming increasingly important, and geometric data play a crucial role in many of these areas. Over the last two decades, researchers have proposed many spatial indexes to facilitate query operations on large geometric databases [8, 19, 44]. To support concurrent operations on multi-dimensional indexes, some concurrency control algorithms [25, 53, 58, 59, 72, 86] have been proposed. Similar to those of the B^+-trees, they can be categorized into lock coupling and linking algorithms. The techniques of lock coupling were used to design concurrency control algorithms without changing the original R-tree structure [25, 72], and a stack is used for the operations in the R-tree. The search algorithm starts at the root and descends toward the desired leaf nodes. All the nodes meeting the search predicate need to be pushed onto a stack with a shared

lock, and then popped off for examination. The nodes can be unlocked only after checking. Thus all the nodes in the stack are locked and no update operations can modify them until they are unlocked. The search algorithm ought to hold multiple locks simultaneously. On the other hand, the update algorithm also needs to hold multiple locks during the tree traversal, node splitting and MBR modification, which cause significant deterioration in the throughput of the concurrency control algorithm. The search and update operations need to hold multiple locks simultaneously, which cause significant deterioration in the throughput of the concurrency control algorithm.

To solve the lock-coupling problem, linking algorithms for multi-dimensional indexes were proposed in [58, 59] to lock one node during searching and employ lock coupling only while splitting the node or propagating the MBR modification. The main obstacle to the use of the linking mechanism is the lack of linear ordering among the MBRs in the multi-dimensional indexes. To overcome this problem, the R-Link tree [58] was proposed to provide high concurrency on the R-tree through the right link approach. It assigns logical sequence numbers (LSNs) to each node and each entry in the internal nodes. The LSNs are similar to time stamps in that they monotonically increase over time, but are not synchronous with any real-time clock. The nodes, the entries and the search and insert algorithms are designed so that these LSNs can be used to make correct decisions about how to move through the tree. An R-Link tree is basically the standard R-tree with two major differences. First, all of the nodes on any given level are chained together in a singly-linked list via right links. Second, the main structural addition is an LSN in each node and entry. These LSNs provide us with a mechanism for determining when a given node is obsolete. Each entry in a node consists of an MBR, a pointer to the child node and the child node's LSN. If a node has to be split, the new right

sibling is assigned the old node's LSN and the old node receives a new LSN. A process traversing the tree can detect the split even if it has not been installed in the parent by comparing the expected LSN, as taken from the entry in the parent node, with the actual LSN of the child node.

In the above algorithm, each entry of internal nodes maintains the LSNs of child nodes and the fan-out is reduced as a result. Consequently, an extension for concurrency control was proposed to deal with the extra information problem, called Concurrent GiST (CGiST) [59]. CGiST extends every node with only one node sequence number (NSN) and a right link, and uses NSN to detect splits. The NSN is taken from a tree-global, monotonically increasing counter variable. During a node split, this counter changes and a new value is assigned to the original node; the new sibling node receives the original node's prior NSN and right link. In general, a traversing operation can now detect a split by memorizing the global counter value when reading the parent entry and comparing it with the NSN of the current node. If the latter is higher, the node must have been split and the operation follows right links until it sees a node with an equal or smaller NSN. When a node is split, its parent node must first be locked. The node is then split, the NSN is set and the tree-global counter is incremented. So multiple locks must be held for split operations, which may delay the concurrent operations. The process, however, improves on the R-link tree design by eliminating the space overhead in node entries. In contrast, for CGiST to work properly, it must lock the parent before the node split, which reduces the efficiency of concurrency control. In Figure 2.5, we show a simple example where a search operation cannot yield the correct results if a concurrent insertion does not lock the parent before split. For example, suppose we have two operations on the CGiST simultaneously, a search operation P_1 and an insert operation P_2. P_2 splits Node C to C' and C", and

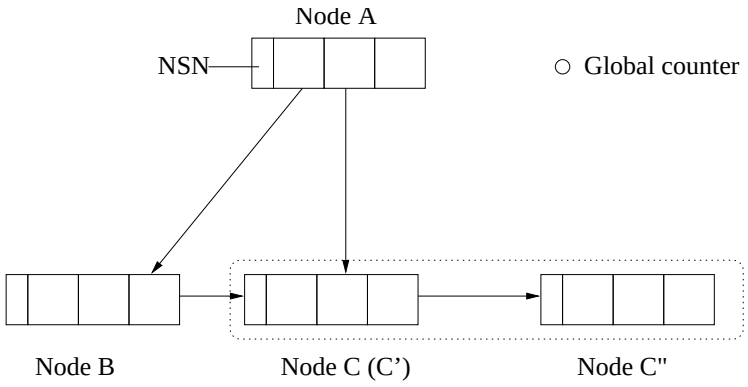


Figure 2.5: Example of an illegal operation

increases the NSN of Node C'. But before P_2 updates the content of Node A, P_1 accesses Node A, reads the current global counter, say n, then pushes n and the address of C into the stack. When the addresses of C and n come up, p_1 accesses the node C'. Unfortunately, p_1 will miss the Node C", because n is not less than the NSN in node C', which means p_1 needs to access the right sibling through the right link.

So far, we can see that the linking techniques also needs to request multiple locks exclusively for split and MBR change. Some mechanisms were proposed to improve the concurrency based on them. In [53], the authors proposed a new linking scheme, which employed a new approach for MBR modification, called top-down index region modification (TDIM). This scheme performs MBR modification from top down by operating on at most one node along the insertion path for most insertions. TDIM combines MBR modification with tree traversal and avoids locking of nodes from multiple levels of the tree at the same time. Also the MBR modification is done in a piecemeal fashion without excluding query access, queries are not blocked except during node split. Additionally, they proposed a split algorithm, named copy based concurrent update (CCU). The basic idea of CCU is to split the node in a local copy. Queries are free to access the original node while the split is processing.

The content of the original node is changed after the split in local copy completes. Thus queries only block during the copy back. The main disadvantage of TDIM is that it does not supports the delete operation. To locate the delete object, we need to traverse multiple path to get the object and we do not sure which node we access is the ancestor of target object. Additionally, even we know the ancestor, we still can not decide whether we can shrink the certain MBR. For CCU, extra spaces are needed for split and each split incurs garbage node, which increases the complexity of the algorithms.

In [86], the authors proposed a concurrency control method to minimize the query delay. To avoid the query delay by MBR updates, they introduced partial lock coupling (PLC) technique. The PLC technique increases concurrency by using lock coupling only in case of MBR shrinking operations that are less frequent than MBR expansion operation. To reduce the query delay by split operation, they optimize exclusive latching time on a split node. The weakness of PLC is that the x-lock is hold during the propagation, and the algorithm did not provide phantom protection. Additionally, this algorithm is based on CGiST, and it must lock the parent of split node before the split; hence multiple locks need to be hold.

Concurrent access to data through multi-dimensional indexes introduces the problem of protecting query range from phantom update. The CGiST method [59] uses a modified predicate locking mechanism to provide phantom protection over Generalized Search Trees. In [23], the dynamic granular locking (DGL) approach was proposed to phantom protection in R-trees. DGL method dynamically partitions the embedded space into lockable granules that adapts to the distribution of objects. They define the lowest level BRs of the R-tree as the lockable granules. Since the R-tree partitions may not cover the entire embedded space, they present an additional structure that partitions the non-covered space into a set of granules

referred to as external granules. Following the principles of granular locking, each operation requests locks on enough granules to guarantee that any two conflicting operations request conflicting locks on at least one granule in common. They also proposed two locking strategies, the *cover-for-insert and overlap-for-search* policy and the *overlap-for-insert and "cover-for-search.* The DGL approach addressed phantom protection problem in multi-dimensional access methods and granular locks can be implemented more efficiently compared to predicate locks, but DGL may offer lower concurrency because of its complexity. Additionally, DGL must integrate with other methods to form the complete concurrency control algorithm for multi-dimensional access.

2.3 Main Memory Index Techniques

For the past several decades, most of the database researches have focused on large databases that do not fit into the main memory. This assumption is increasingly being challenged as RAM gets cheaper and larger. It is now not uncommon to find affordable computer systems with main memory in the order of magnitude of gigabytes, thus it is possible for us to keep the whole database in the main memory.

Index structures designed for main memory are different from those designed for disk-based systems. Recent advances in hardware technology have reduced the access times of both memory and disk tremendously. However, the number of disk I/O remains an important parameter to optimize the performance of index structures in disk-resident DBMS. On the contrary, a memory-oriented index structure is contained in main memory, hence there are no disk accesses to minimize, while memory access and CPU cost become dominant. Thus the primary goals of a main memory index are to reduce the cache and TLB misses and overall computation

cost.

With the ever-widening gap between CPU and memory speed, fast cache memory was introduced to fill the gap. Take the SUN Fire 4800 for example, a L1 cache hit incurs about 2 processor clocks, while a L2 miss incurs more than 170 processor clocks when data is required to be fetched from slower but larger capacity RAM. Therefore, effective utilization of L2 caches becomes an important optimization issue for main memory index processing. Indeed, optimizing the utilization of L2 cache is somewhat similar to optimizing the utilization of RAM to the disk-resident DBMS. Clearly, disk-based indexing structures are applicable in main memory context too, though one can expect their performance to be less satisfactory since they are designed to minimize I/Os. There are some recent works that propose cache conscious index structures [26, 18, 76, 77, 55], which are designed to optimize the memory cache utilization. Indexes that have designed to exploit smaller but much faster cache memory are said to be cache conscious. Like the disk-based indexes which page the structure into 4K or 8K blocks due to the way data blocks are read from disk, cache conscious indexes page the structure based on cache lines.

The AVL-tree

The AVL-tree [3] was designed as an internal memory data structure (see Figure 2.6). It is a binary tree, which is very fast since binary search is intrinsic to the tree structure (no arithmetic calculations are needed). Updates always affect a leaf node and may result in an unbalanced tree, so the tree is kept balanced by rotation operations. The AVL-tree has two major disadvantage: it is poor in storage utilization, each tree node only keeps one record, but there are two pointers and a balance factor for every record; because each node only has one key, the AVL-tree is usually higher than other trees, thus the cache and TLB misses are more when

we go through the AVL-tree.

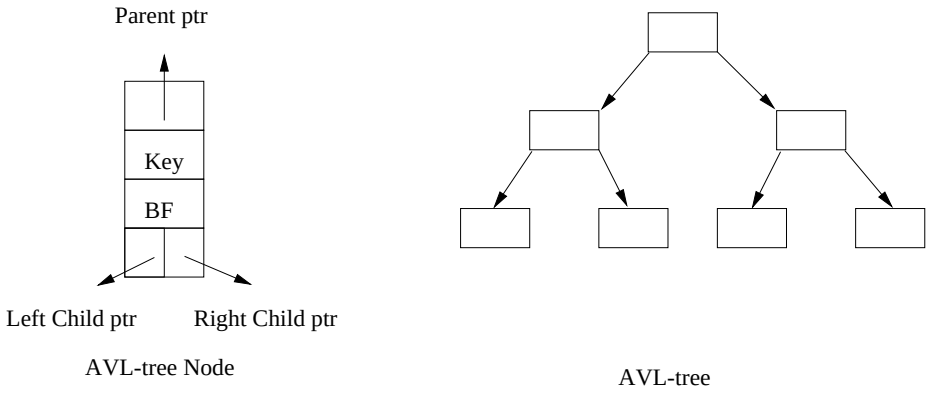


Figure 2.6: The structure of AVL-tree

The T-tree

The T-tree [63], which evolved from the AVL-tree and B-tree, is a balanced binary tree designed as an index structure especially for main memory systems. Its main feature is that it allows each node to contain more than one element (see Figure 2.7). In consequence, the T-tree inherits the good update and storage characteristics of the B-tree, and at the same time, it also retains the intrinsic binary search nature of the AVL-tree. Data movement, which is required for insertion and deletion, is usually needed within a single node. Rebalancing operation is similar to that of the AVL-tree, but the frequency of rebalancing operation is much lower than in an AVL-tree because of the intra-node data movement.

Figure 2.7 shows a T-tree node which consists of a balance factor (BF), a number of keys, one parent pointer and two child pointers. The balance factor represents the difference between the right sub-tree height and the left sub-tree height. A minimum count and a maximum count are associated with a T-tree. Internal nodes keep their occupancy, i.e. the number of keys in one node, in this range. During the insert operation, the corresponding key will be inserted into the node

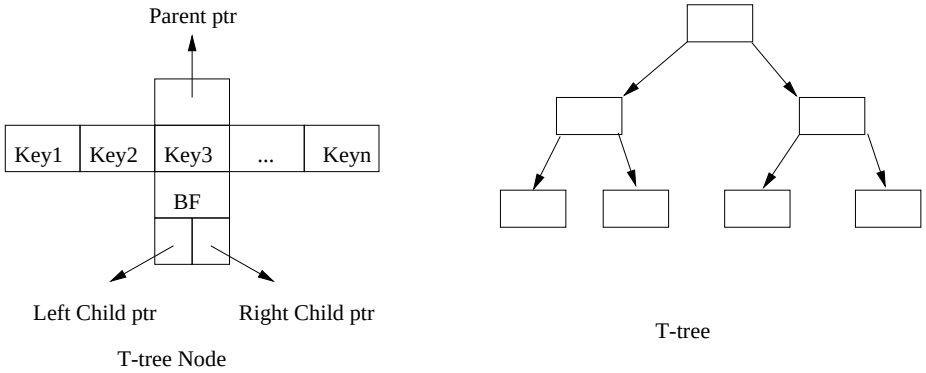


Figure 2.7: The structure of T-tree

that bounds the key value of the data entry. When the node to be inserted is full, it may cause inserting a new node to the tree. Inserting a new node may not only require redistributing the keys, but also rebalancing the tree. Similarly, deletion may cause a node to become underflowed or empty, deleting a node may also cause the tree to be rotated.

Although the T-tree performs better than the B^+-tree when it was proposed, it performs much worse than B^+-tree in main memory now, because CPU speed has improved by two orders of magnitude relative to memory latency during the past twenty years and the cache misses become more expensive [76].

The CSB$^+$-tree

In [76], the authors proposed a new index structure called *Cache-Sensitive Search Trees* (CSS-tree) that has even better cache behavior than a B^+-Tree. The CSS-trees augment binary search by storing a directory structure on top of the sorted list of elements. The CSS-trees avoid storing child pointers explicitly by embedding the directory structure in an array sequentially, and thus have a better utilization of each cache line. Although this approach improves the searching speed, it also makes incremental updates difficult since the relative positions between nodes are

important. As a result, we have to batch updates and rebuild the CSS-tree.

To overcome the rebuilding problem, the authors introduced a new index structure called the *Cache-Sensitive B+-Tree* (CSB^+-Tree) that retains the good cache behavior of the CSS-trees while at the same time being able to support incremental updates [77]. A CSB^+-Tree has a structure similar to a B^+-Tree, as shown in Figure 2.8. Instead of storing all the child pointers explicitly, a CSB^+-tree puts all the child nodes for a given node contiguously in an array and stores only the pointer to the first child node. Other child nodes can be found by adding an offset to the first-child pointer. This approach allows good utilization of a cache line. Additionally, the CSB^+-trees can support incremental updates in a way similar to the B^+-trees. The experimental results demonstrate that the CSB^+-trees dominate B^+-trees in terms of search in memory.

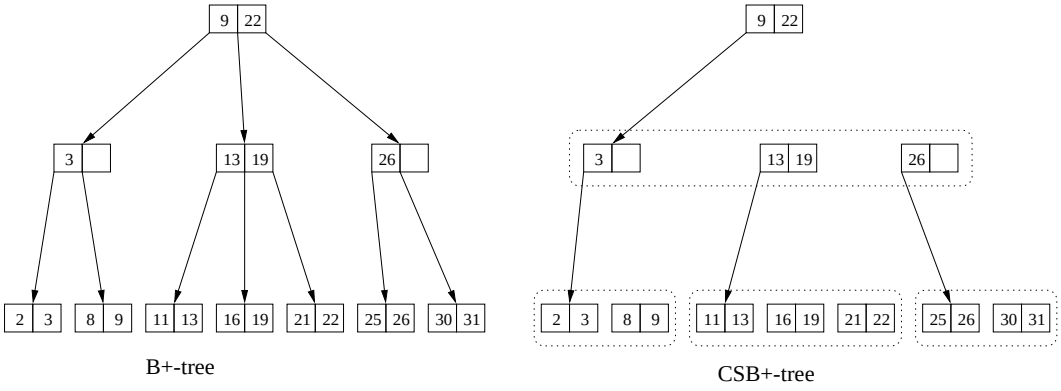


Figure 2.8: The comparison between B^+-tree and CSB^+-tree

The CSB^+-trees need to maintain the property that sibling nodes are contiguous, even in the face of updates. A set of sibling nodes is called a node group. There are several ways to keep node groups contiguous, all of which involve some amount of copying of nodes when there is a split. The authors presented several variations on the CSB^+-tree idea that differ in how they achieve the contiguity property. The simplest approach is to deallocate a node group and allocate a new larger node

group on a split. *Segmented* CSB$^+$-trees reduce the update cost by copying just segments of node groups. *Full* CSB$^+$-trees pre-allocate extra space within node groups, allowing easier memory management and cheaper copying operations.

The CR-tree

In [55], the authors proposed a cache-conscious version of the R-tree called the CR-tree. To pack more entries in a node, the CR-tree compresses MBR keys. It first represents the coordinates of an MBR key relatively to the lower left corner of its parent MBR to eliminate the leading 0's from the relative coordinate. Then, it quantizes the relative coordinates to further cut off the less significant trailing bits. These relative coordinates have fewer bit, and the resultant MBR is called QRMBR (quantized relative representation of MBR). Consequently, the CR-tree becomes significantly wider and smaller than the ordinary R-tree. The experimental and analytical studies on the CR-tree showed that it performs faster than the R-tree. The algorithms of of CR-tree are shown as follow:

- The search algorithm is similar to the one used in other R-tree variants. The only difference is that the CR-tree compares a query rectangle with QRMBRs. Instead of recovering MBRs from QRMBRs, the CR-tree transforms the query rectangle into the corresponding QRMBR using the MBR of each node as the reference MBR. Then, it compares two QRMBRs to determine whether they overlap.

- To insert a new object, the CR-tree traverses down from the root choosing the child node that needs the least enlargement to enclose the object MBR. When visiting an internal node to choose one of its children, the object MBR is first transformed into the QRMBR using the reference MBR. Then, the enlargement is calculated between a pair of QRMBRs. When a leaf node

is reached, the node MBR is first adjusted such that it encloses the object MBR. Then, an index entry for the object is created in the node. If the node MBR has been adjusted, the QRMBRs in the node are recalculated because their reference MBR has been changed. If the node overflows, it is split and the split propagates up the tree.

Concurrency Control of Main Memory Indexes

Main memory CC scheme is crucial for running real-world main memory database applications involving index updates and taking advantage of the multiprocessor systems for scaling up the performance of such applications.

The physical versioning was proposed for the CC of T-tree [78]. Its key idea is to create a new version of the node for the updater so that the index readers do not interfere with the updaters. The major advantage of this scheme is the lock-free traversal of indexes. As a result, a high degree of concurrency comparable to that of no concurrency control can be achieved for read transactions. Incorporation of the updated version into the index involves obtaining locks either at the tree-level or both on the tree and on the node to update. The major problem with this scheme is the high cost of creating versions. The index performance degrades sharply with the increasing update ratio, and the scalability of update performance is also poor.

In [22], the authors present an optimistic, latch-free index traversal (OLFIT) concurrency control scheme for B^+-tree variants. For each node, this scheme maintains a lock, a version number, and a link to the next neighbor node at the same level. The next node link is borrowed from the Blink-Tree to facilitate the split handling. The index traversal involves consistent node reads starting from the root. Here, consistency means that no update occurs between the start and the end of a node read. To ensure the consistency of node reads without locking, every node

update first obtains the node lock, updates the node content, increments the version number, and releases the lock. The node read begins with reading the version number into a register and ends with verifying if the node lock is free and the current version number is equal to the register-stored one. If these two conditions are true, the read is consistent. Otherwise, the node read is retried until the conditions become true.

CHAPTER 3

Exploitation of Bounded Disorder for Memory B$^+$-tree Indexing

3.1 Introduction

The impact of advances in hardware and main memory capacity is beginning to be felt by the software community. Not only is it possible to store the entirety of a database in memory now, many algorithms that were designed in the past need to be reexamined. For example, the memory-based index structure T-tree [63] is no longer attractive because of its low fanout, poor cache behavior and excessive utilization of pointers. In recent years, many new schemes that are cache-aware have been designed [40, 77, 18, 26, 27]. In this chapter, we reexamine the single-dimensional indexing issue in main memory databases.

To improve performance, an index structure must facilitate effective use of the L2 caches and CPU. One promising structure is the CSB$^+$-tree [77] that reduces the cache misses by storing all the child nodes of any given node contiguously, and keeping only the address of the first child in each node, thereby increasing the utilization of the cache line. However, while the CSB$^+$-tree is efficient for range

queries, it does not perform well for exact match queries (compared to hash-based schemes). This is because the height of the tree could be tall due to the small fanout limited by cache line size. This prompted us to search for a structure that can combine the benefits of hashing and tree structures. Here we present our solution to facilitate efficient query processing in main memory environment. We optimized the Bounded Disorder (BD) method [67] for memory processing. The BD-tree is essentially a B^+-tree where the leaf nodes are large-sized partitions, and each partition is organized by hash tables. Records within a leaf node are not ordered.

Figure 3.1 shows the structure of the BD-tree, which comprises two tiers. In the first tier, a hierarchical multi-way tree structure (i.e. B^+-tree) *range partitions* the data. The second tier begins at the leaf level of the tree, which organizes the data in each leaf node using a hashing method.

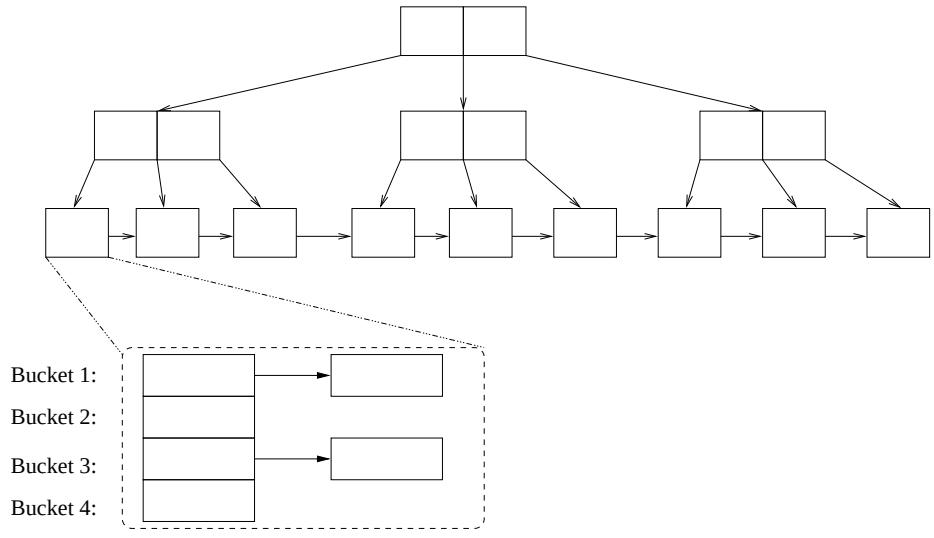


Figure 3.1: The BD-tree

The BD-tree is efficient for memory query processing. First, the leaf node size of a BD-tree is much larger than that of a B^+-tree. This means fewer internal nodes are needed, i.e. a shorter tree. Therefore fewer cache and TLB misses incur

during the tree traversal. Moreover, since leaf nodes are organized as hash buckets, only the necessary buckets need to be searched. This clearly benefits exact match queries. Second, the use of a tree structure essentially acts as a dynamic range partitioning mechanism, allowing each leaf node to have a different range size. As such, only the necessary leaf partitions need to be examined for range queries.

To optimize the BD-tree for main memory processing, we shall apply a hash function with low computation cost to reduce the CPU overhead in leaf, because we need to calculate the hash value of the key for search. Since the use of tree structure essentially acts as a dynamic range partitioning mechanism, the average performance should be accurately predicted under the uniformity assumption. Additionally, memory hashing technique [5] can be applied on BD-tree to further improve its performance. Moreover, since data may be duplicated or the hash functions may not distribute the data uniformly within each leaf node, we allow each bucket to have k chains (for some integer $k >= 1$). In other words, a leaf node will be split only when a new record is to be inserted into a bucket whose chained buckets are overflowed. Since the fanout of a main memory tree structure is typically small compared to a disk-based index, we can reduce the height of the BD-tree significantly by setting relatively large leaf partition sizes. Within the leaf node, we set the bucket size to correspond to the cache line size. Note that while we can keep the leaf node of a B^+-tree/CSB^+-tree larger than its internal nodes, a larger leaf node size does not result in better search performance, because a large node size also means more cache misses when the leaf node is accessed. In this case, even if we use binary search, some non-contiguous segments of the leaf need to be loaded into the cache. For exact match queries in the BD-tree, we only need one cache miss to access each chain bucket and at most k buckets to be retrieved in leaf level. For range queries, because the leaf partitions are ordered, we just scan

the leaves that overlap the desired query ranges. To speed up the range query, we store the chained buckets contiguous with the primary buckets of the leaf node, and hence reduce the TLB misses of scanning.

There are exceptions where the duplicated keys overflow the chained buckets and no splitting will help. We store the number of unique keys in each bucket in the leaf node, which is determined by k. For example, if each chain has 8 keys, this maximum value can be $8k$. Thus, a leaf node will be split when a new record is to be inserted into a bucket whose number of unique keys are more than $8k$. Because of the existence of the duplicated keys, we allow more than k chains in this case.

3.2 The Operations on BD-tree

Since the BD-tree is a hybrid structure between the B^+-tree and the hash-based method, the operations on BD-tree is also an integration of their operations.

3.2.1 Exact Match and Range Query Algorithms

Searching in the BD-tree is quite straightforward. The operation to find a target leaf is the same as that of finding a leaf node in B^+-tree. After finding the target leaf, we use hash-based method to get the exact key in the node; while we use binary search in searching a B^+-tree node. The search algorithm for exact match queries, outlined in Figure 3.12, works as follows:

1. The search starts from the root of the tree.

2. Within an internal node, we use binary search to determine the child node to be searched further. The sub-tree pointed by the associated child pointer between two keys, whose data space bounds the search key, is traversed to search. This operation is recursively conducted until we find the target leaf.

3. Within the leaf node, we calculate the hash value of the search key first and access the proper bucket. We scan the keys in all the chained buckets; all matching answers are returned; otherwise (i.e. no matching answer is found after all chained buckets have been searched), the search fails.

Algorithm Search(R, k)
Input: search key k, node of BD-tree R (root initially)
Output: offset of key

```
1.  while (R != LEAF)
2.        i=search_node(R, k);
3.        R=R→children[i];
4.  h=hash(k);
5.  search_bucket(h,k);
6.  return(offset);
end Search.
```

Figure 3.2: Exact match search in BD-tree.

A range query is an extension of an exact match query. Like the B^+-tree, for a range query defined by two search keys, we use the first search key of the query range to search the index as an exact match query. Once the leaf node containing the search key is obtained, we scan all the records in the leaf node (since keys are randomly distributed across buckets within the leaf partition), sort the records and return answers in the query range. If the maximum value in the leaf is smaller than the end point of the query range, we follow the next pointer of the leaf node to retrieve the next leaf. The search process continues until all leaves that intersect the query range have been searched.

3.2.2 Insert Algorithm

To insert a new key, we first locate the target leaf node, then calculate the hash value of the insert key and insert the key into the proper bucket. If the number

of unique keys is beyond the constraint of buckets, we split the leaf and insert the new leaf node into the tree. The insertion algorithm shown in Figure 3.3 works as follows:

1. Search for the bounding leaf of the insert key. If the search exhausts the tree and no bounding leaf is found, the last accessed leaf node on the search path is considered to be the "bounding leaf".

2. Calculate the hash value of the key, check the uniqueness of the key by locate the position of key. If the number of unique keys is less than the bound, insert the key in the proper bucket, then the insert operation succeeds and stops.

3. If the number of unique keys is beyond the bound, we need to split the leaf. We first sort the new key and keys of the leaf node, and use the middle key of the sorted array to split the leaf. We create a new leaf node, and move the keys larger than the middle keys to the newly created leaf. Records are rehashed into buckets within the leaves.

4. If the parent node is not full, insert the pointer to the newly created leaf into the node; else split the parent node and adjust the tree. This step may cascade up the tree.

3.2.3 Delete Algorithm

The delete algorithm is similar to the insert algorithm: access the leaf partition that bounds the key to be deleted, delete the key and adjust the tree. However, after deletion, merging of leaves and movement of data may be involved to ensure good space utilization of the index. In practice, people implement the deletion by

Algorithm Insert(R, k)
Input: insert record k, root node of BD-tree R

```
1.  If (R==NULL)
2.      create_node(R)
3.      create_leaf(leaf)
3.      insert_key(leaf, k)
4.  else
5.      leaf=access(R, k)
6.      h=hash(k);
7.      check_duplicate(k);
8.      if (number of unique keys is less than bound)
9.          insert_key(leaf, k)
10.     else
11.         mid=split_leaf(leaf, k)
12.         insert_key(mid, leaf→parent)
```

insert_key(key, node)

```
1.  if (node is not full)
2.      add_key(key, node)
3.  else
4.      while (node→parent is full)
5.          mid_key=split_node(node, key)
6.          node = node→parent
7.          insert_key(mid_key, node)
8.      mid_key=split_node(node, mid_key)
9.      if(node→parent != NULL)
10.         add_key(mid_key, node→parent)
11.     else
12.         new_node=create_node()
13.         add_key(mid_key, new_node)
end Insert.
```

Figure 3.3: Insert in BD-tree.

simply removing the key and record without tree adjustment. The detail of the algorithm is outlined in Figure 3.4.

1. Search for the leaf that bounds the value to be deleted. If it is found, calculate the hash value and search for the delete value in the proper bucket. If the value does not exist in the bucket, stop.

2. Delete the key, if the delete does not cause the leaf to underflow, then stop; else check whether we need to coalesce the leaf with its neighbors. If the node is underflowed, which means the number of keys in the leaf is below a certain threshold value; we must either redistribute keys from an adjacent sibling or merge the partition with a sibling to maintain the minimum occupancy. The last operation may cascade up the tree. Note that care has to be taken as two partitions may not be able to be merged as merging them may result in a long chain of buckets.

3.3 Cost Analysis

To have an insight into the performance of the B^+-tree, CSB^+-tree and BD-tree, we present analytical models to evaluate the cost of an exact match query. We model the cost of exact match query as a function of three variables: number of L2 cache misses, TLB misses and instructions. There are other factors that affect performance, such as branch mispredictions, instruction cache misses and L1 data cache misses. However these factors do not play a significant role in determining the overall cost [4]. The cost analysis consists of a cache and TLB misses model and an execution time model. The representative parameters used for the analysis are shown in Table 3.1. Some of the values were produced by measuring the performance of indexes in the experiment.

```
Algorithm Delete(R, k)
Input: delete key k, root of BD-tree R.

1.  leaf=access(R, key)
2.  if ( leaf == NULL)
3.      return FALSE
4.  else
5.      delete( leaf, k)
6.      if (leaf is not underflow)
7.          return TRUE
8.      else
9.          if (coalesce(leaf, leaf.next)==TRUE)
10.             delete_key(key, leaf→parent))

delete_key(key, node)
1.  remove_key(key, node)
2.  if (node is underflow)
3.      if (merge(node, node.sibling)==TRUE)
4.          node = node→parent
5.          if (node ! = NULL)
6.              delete_key(key, node)
7.      else
8.          borrow_key(node.sibling);
end Delete.
```

Figure 3.4: Delete in BD-tree

The number of chained buckets is a parameter of the BD-tree. The larger bucket number k may reduce the tree height, but increase the cost in the leaf level, both cache misses and computational cost. We can easily use the formula to estimate the cost for variant values of k. Since we focus on the comparison with other structures, we assume that the buckets in the leaf nodes of the BD-tree have no overflow chains to clarity the analysis.

Variable	Value	Description
c	10M	the cardinality of the dataset
s	4 bytes	the size of a key
p	4 bytes	the size of pointer
r	8 bytes	the size of a $< key, RID >$ pair
l	64 bytes	the size of a cache line (UltraSPARC processor)
n	32-256 bytes	the size of a node
b	64 bytes	the size of a hash bucket
u	70%	the average utilization of a node/bucket
d	64	the directory size of leaf partition (the BD-tree)
m_c	150 cycles	the cost of an L2 cache miss
m_t	100 cycles	the cost of a TLB miss
m	12 bytes	the size of metadata in the node of tree
I_b	40	number of instructions to evaluate a key and calculate next evaluation position in binary search

Table 3.1: Parameters for cost analysis

3.3.1 Cache and TLB Miss Model

The L2 cache misses and TLB misses play an important role in the tree operations. For each node access, there may be several cache misses and one TLB miss for various node sizes (< virtual memory page size). The node size and tree height are two main parameters in this cost model. A larger node size implies a shorter tree, and vice versa.

To determine the height of the tree, we first present the formula for fanout, f.

$$f = \begin{cases} \lfloor u * \frac{n-m}{s} + 1 \rfloor & : \quad CSB^+\text{-}tree \\ \lfloor u * \frac{n-m}{s+p} + 1 \rfloor & : \quad B^+\text{-}tree/BD\text{-}tree \end{cases}$$

Also with the given dataset, the number of leaf nodes, L is :

$$L = \begin{cases} \lceil \frac{c}{u * \frac{n-m}{r}} \rceil & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \lceil \frac{c}{u * d * \frac{b}{r}} \rceil & : \quad BD\text{-}tree \end{cases}$$

Therefore, the height of the tree, h, is given as follows:

$$h = \lceil \log_f L + 1 \rceil.$$

For an exact match query, we must traverse the tree from the root to the leaf to get the answer. At each level, a child node is determined by a binary search in the parent node, so the number of L2 cache misses of a node is $\lceil (\log_2 (u * n_c) + 0.5) \rceil$, where n_c is the number of cache lines spanned by one node ($n_c = \lceil \frac{n}{l} \rceil$). Because we always need to access the first cache block of the node, we add 0.5 cache miss for each node access.

On the first query of the index, the number of cache misses is the cache lines accessed. However, there is a higher probability of finding some nodes in the cache for subsequent queries, e.g. the nodes in the upper levels of the tree are accessed frequently and may always reside in the cache. Re-accessing of these nodes will result in fewer cache misses. To model cache misses more efficiently, we use the Cardenas's formula [21]:

$$X(n, q) = N * (1 - (1 - 1/N)^q),$$

where X is the number of unique nodes accessed, N is the number of nodes available and q is the number of queries. Using this formula, the average number of accessed nodes for a single query, i.e. $\frac{X(N,q)}{q}$, reduces when we increase the number of queries. We combine the Cardenas's formula with the tree structure to determine the number of unique nodes that are accessed when we traverse the tree. The number of cache misses in the leaf node of BD-tree is 1 as we assume no chained buckets in the model. Thus we can model the number of cache misses per query as:

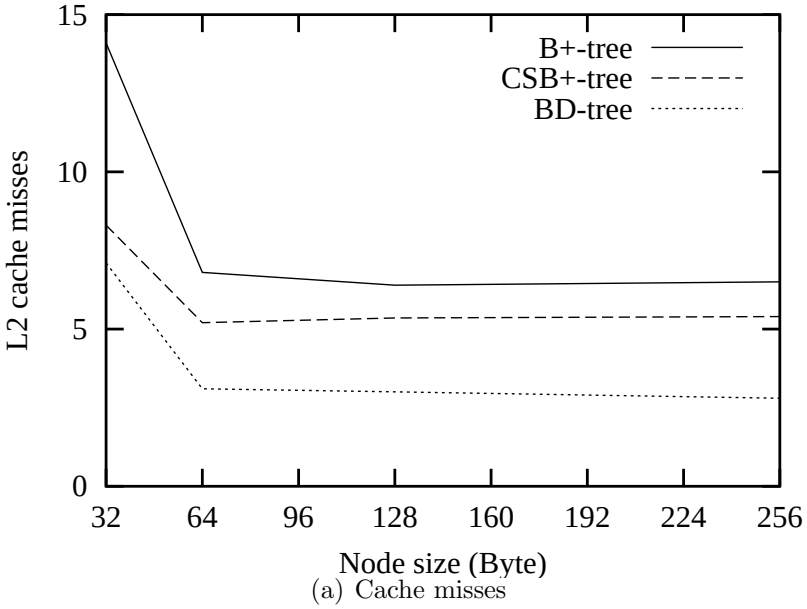
$$C_m = \begin{cases} \frac{\sum_{i=1}^{h} X(n(i),q) * \lceil (\log_2 (u * n_c) + 0.5) \rceil}{q} & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \frac{\sum_{i=1}^{h-1} X(n(i),q) * \lceil (\log_2 (u * n_c) + 0.5) \rceil + X(n(h),q)}{q} & : \quad BD\text{-}tree \end{cases}$$

where $n(i)$ is the number of nodes at level i in the tree ($n(i) = f^{i-1}$). The model gives the compulsory cache misses, but not the capacity cache misses [45]. Because the cache size is finite, eventually some highly accessed cache lines can be replaced after running many queries. However, with large cache size, these effects are not significant for an approximation.

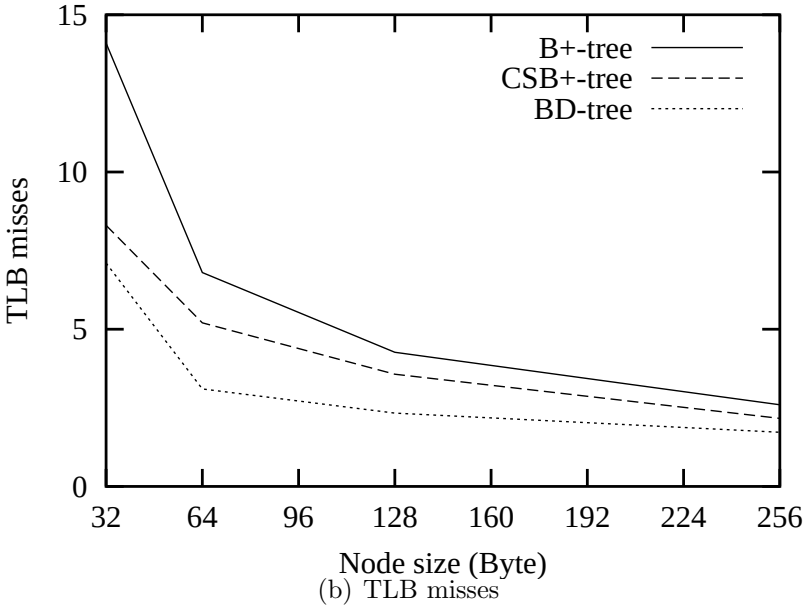
For an individual query, the number of TLB misses can be equal to the height of the tree, while the TLB misses may be satisfied from the L2 cache hits and top levels of the tree may stay cached for subsequent queries. Therefore, the number of the TLB misses, TLB_m can be much smaller than the tree height and also smaller than C_m because of larger TLB page size. TLB_m is represented by the following equations:

$$TLB_m = \begin{cases} \frac{C_m}{\log_2(u*n_c)+0.5} & : \quad B^+\text{-}tree/CSB^+\text{-}tree \\ \frac{C_m-1}{\log_2(u*n_c)+0.5}+1 & : \quad BD\text{-}tree \end{cases}$$

We show the average number of TLB and cache misses for 1000 exact match queries using the above models in Figure 3.5. The B$^+$-tree performs worst among these trees. Compared with the CSB$^+$-tree, the B$^+$-tree has a smaller branching factor than the CSB$^+$-tree since it needs to store more child pointers. Hence, the B$^+$-tree is taller than the CSB$^+$-tree, and incurs more TLB and cache misses. Not surprisingly, the BD-tree is best because its leaf node (partition) has more keys, thus the BD-tree is shorter. When we increase the node size, both the number of cache and TLB misses reduce. The number of cache misses is much smaller than the number of cache lines accessed, because some of the frequently accessed nodes can reside in the cache. The number of TLB misses is fewer than that of cache misses when the node size is larger than 64 bytes, because one node access only incurs one TLB miss but several cache misses in this case. For the CSB$^+$-tree,



(a) Cache misses



(b) TLB misses

Figure 3.5: The number of cache and TLB misses for a single query

when the node size is 64 bytes (cache line size), the number of cache misses is the smallest. For the B^+-tree and BD-tree, the optimal node size is 128 bytes. In all cases, the cache misses do not increase much for larger node sizes. Because the tree is shorter, more proportional accessed nodes can be resident in the cache, although more cache lines need to be accessed. The number of cache misses is a compromise of these factors.

3.3.2 Execution Time Model

The time to execute a single query (T_Q) includes computation time (T_C), L2 cache misses time (T_{CM}) and TLB misses time (T_{TLBM}). Thus, we can estimate the execution time T_Q as:

$$T_Q = T_C + T_{CM} + T_{TLBM}.$$

The above equation ignores the effects of out-of-order execution which can hide a portion of cache miss latency by continuing to execute the instructions out-of-order on the processor [4]. It is easy for us to get T_{CM} and T_{TLBM} because we know the number of TLB and cache misses from the TLB and cache miss model. We obtain the cost of each TLB and cache miss using the Calibrator Tool [2], e.g. a L2 cache miss latency m_c is 150 cycles and a TLB miss latency m_t is 100 cycles for SUNFire 4800.

The computation cost is the time taken to execute all the instructions for a query. The number of instructions executed at each level is mainly the binary search inside the node, so we can give the equation to calculate the number of total

instructions I_Q as follows:

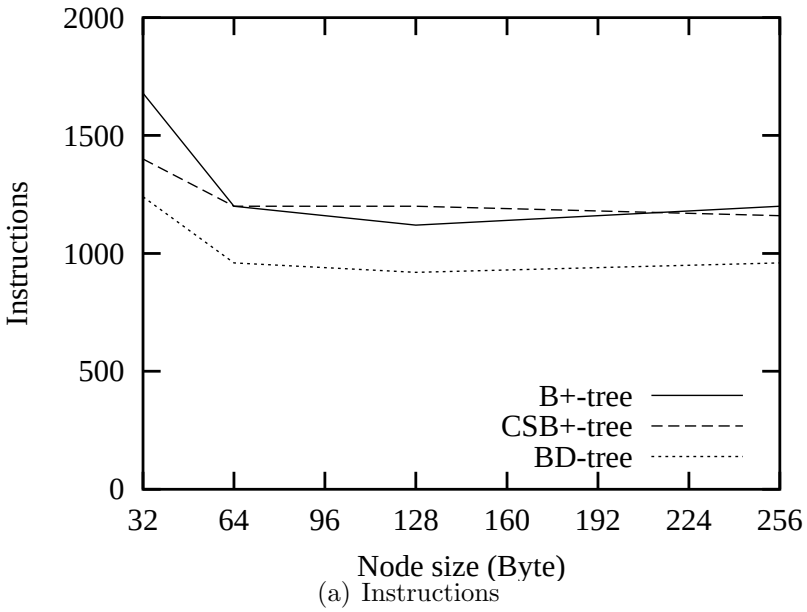
$$I_Q = \begin{cases} I_b * ((h-1) * \lceil \log_2 (u * \frac{n-m}{s+p} + 1) \rceil + \lceil \log_2 (u * \frac{n-m}{r} + 1) \rceil) & : \quad B^+\text{-}tree \\ I_b * ((h-1) * \lceil \log_2 (u * \frac{n-m}{s} + 1) \rceil + \lceil \log_2 (u * \frac{n-m}{r} + 1) \rceil) & : \quad CSB^+\text{-}tree \\ I_b * ((h-1) * \lceil \log_2 (u * \frac{n-m}{s} + 1) \rceil + \lceil \log_2 (u * \frac{b}{r} + 1) \rceil) & : \quad BD\text{-}tree \end{cases}$$

where I_b is the number of instructions required to evaluate a key and select the next position in a binary search. The Perfmon tool [34] can be used to count the number of instructions/cycles of an evaluation. Hence, we can get the number of instructions for an exact match query using the above equation. The CPU cycles per instruction (CPI) used in the model is equal to 2. The number of instructions and total CPU cycles for a query are shown in Figure 3.6.

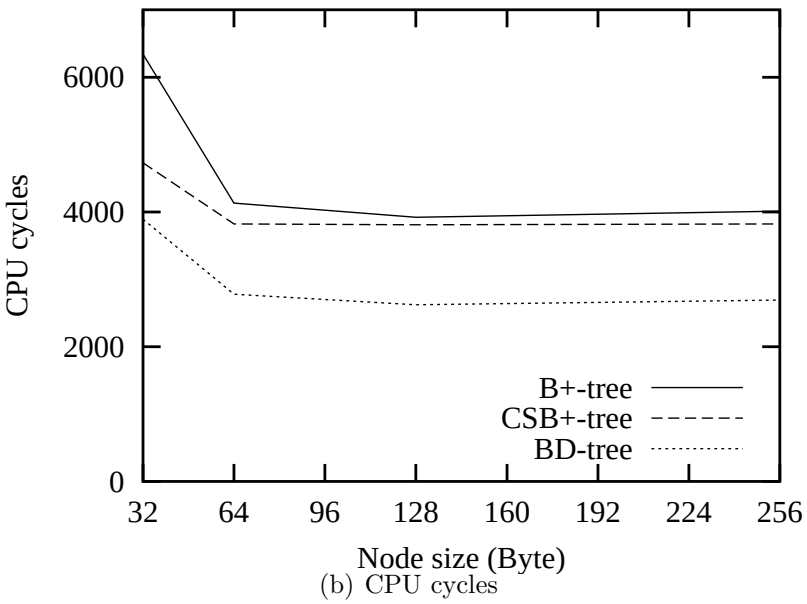
Figure 3.6 (a) shows the number of instructions executed versus node size per query. The number of instructions is quite stable when the node size is larger than 64 bytes, because we execute binary search in each node and the overall computation cost is similar. More instructions are incurred when the node size is equal to 32 bytes, because the binary search is not efficient when the node size is too small. The total execution time is the combination of TLB and cache misses and CPU cost. When the node size is equal to the cache line size, the CSB$^+$-tree performs best, while 128 bytes is an optimal node size for B$^+$-tree and BD-tree. Clearly, the BD-tree performs best among these trees, followed by the CSB$^+$-tree and B$^+$-tree.

3.4 Performance Study of BD-tree

In this section, we present an experimental evaluation of the optimized BD-trees. We compare the BD-tree against the B$^+$-tree and CSB$^+$-tree. We run our exper-



(a) Instructions



(b) CPU cycles

Figure 3.6: The number of instructions and cycles for a single query

iments on SUN Fire 4800 machine, which has 750MHz CPU, 16 GB RAM and $< 8M, 64B, 1 >$ L2 cache ($<$ *cache size*, *cache line size*, *associativity* $>$). We implemented these index structures in C. The machine is running SunOS 5.8 and the compiler is GNU's gcc 2.95.2. The dataset we used consists of 10 million integers. The data in the structures consists of $< key, pointer >$ pairs, both 4 bytes long, and the keys are uniformly distributed random data. We also conducted sensitivity analysis on the techniques by varying distributions of datasets (e.g. Zipf distributed in terms of key duplication [91]), cardinality of datasets, and so on. For the efficient processing in the leaf node, we allow at most $2 * m$ unique keys in one bucket chain, where m is number of keys in a bucket. All the buckets in a leaf node are allocated in the contiguous memory space to facilitate the range query.

3.4.1 Tuning the BD-tree

The BD-tree has two extra parameters for the hash structure in the leaf level, i.e. the number of buckets in a leaf and the bucket size. Our preliminary study to tune these parameters shows that the exact match query performance of BD-tree improves with a large number of buckets. On the other hand, the large leaf size may reduce the performance of range queries with small query range. In the first experiment, we tune these two parameters for the optimal performance. The internal node size is set to 128 bytes, which is the optimal value according to the cost analysis. The Figure 3.7 shows the exact match query performance of BD-tree for different bucket sizes and directory sizes.

We found that the performance of tree is best when the leaf has buckets of size 64 bytes, i.e. which is equal to the cache line size. Also it shows that the performance is better when we increase the directory size. Because we use hash table in the leaf level, the cost in leaf level is almost constant and the larger directory size means

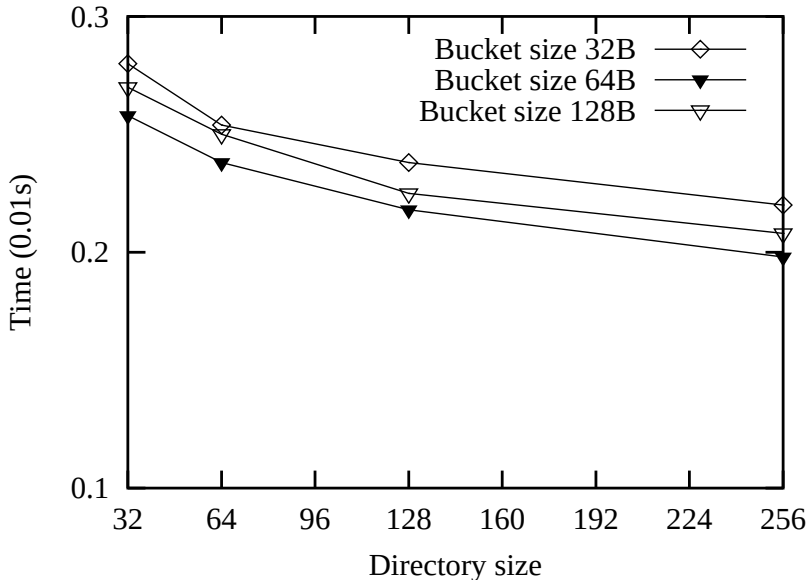


Figure 3.7: Performance of exact match query

the shorter tree. We use 64 bytes as a default bucket size and test the range query performance of BD-tree, the results are shown in Figure 3.8.

We can see that the larger directory size shows poorer results when the range selectivity is small, e.g. 0.01% selectivity. Because larger directory size means larger node size, and the range query of BD-tree needs to access the whole leaf, the disorder within the leaf incurs some additional cost. From the above two experiment, we found that there is no optimal parameter values for both exact match query and range query. For simplicity of presentation, we use the leaf with 64 buckets of size 64 bytes each as the default values in our study here, where both queries yield near optimal performance.

3.4.2 Performance of Exact Match Query

In this experiment, we study the effect of node size on the performance of exact match query of the various schemes by varying the node size from 32 bytes to 256 bytes. This also serves to tune the various schemes for optimal settings of

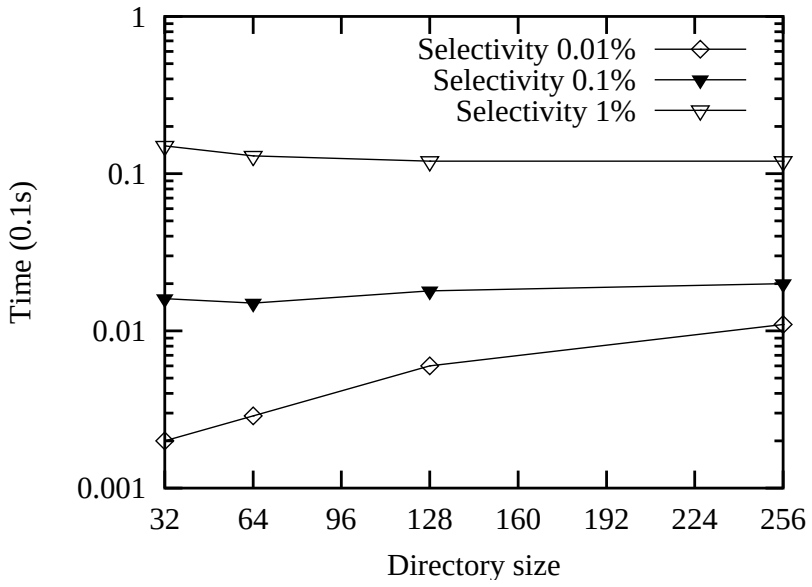
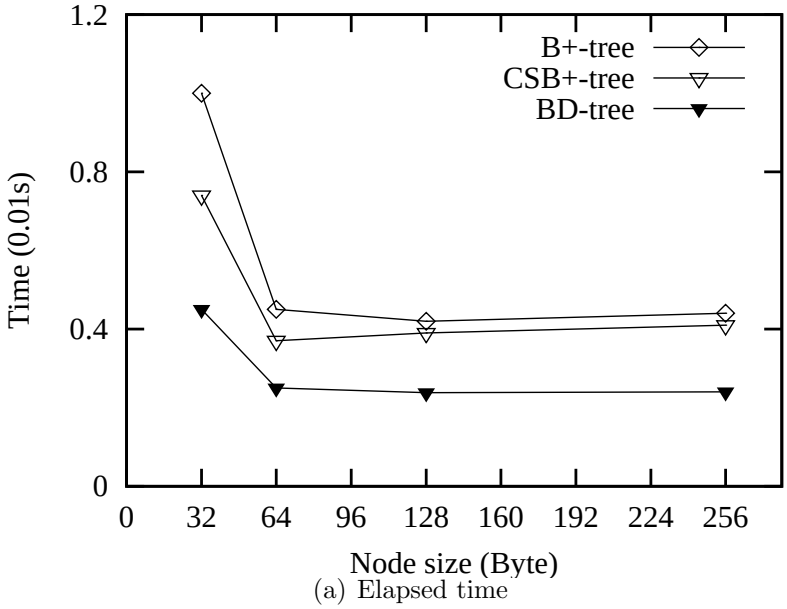


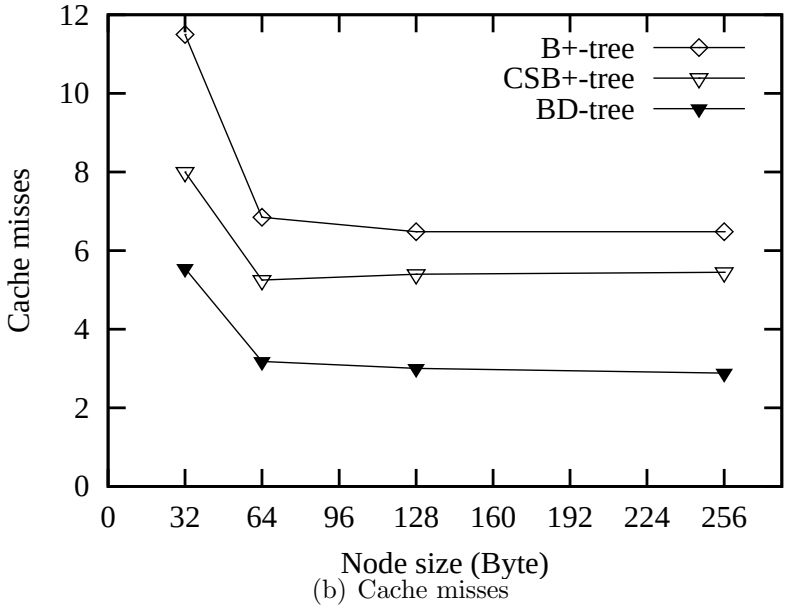
Figure 3.8: Performance of range query

node sizes. We conducted 1000 operations of exact match queries, and recorded their performance. The results are shown in Figure 3.9 and it is clear that the performance of the three indexes is consistent with the analytical results.

Figure 3.9 (a) shows the elapsed time of exact match query. For all tree-based schemes, when the node size is too small (e.g, 32 bytes), the fanout becomes so small that it results in a very tall tree. This leads to more TLB and cache misses as the tree is traversed. When the node size increases, the number of TLB and cache misses decreases. For the B^+-tree, its optimal performance for exact match queries occurs when the node size is equal to 128 bytes. However, the performance starts to degenerate as the node size reaches beyond this point. Our result is consistent with the findings in [76]. For the CSB^+-tree, we observe that it is optimal when the node size is 64 bytes which is the cache line size of L2. The reason for the poor performance for small node size is the same as that for the B^+-tree. For the BD-tree, we fixed the leaf size to contain 64 buckets and the bucket size is equal to cache line size, 64 bytes. From the results, we find that exact match query



(a) Elapsed time



(b) Cache misses

Figure 3.9: Effect of node size

performance is almost optimal when the node size is equal to 128 bytes which is the same as the B^+-tree.

Comparing these various B^+-tree based structures, we see that the CSB^+-tree is better than the B^+-tree in all cases, but the difference is not significant when the node size is larger than 128 bytes. This result is consistent with what was observed in [76] and in the analytical section. The BD-tree performs best and is about 40% better because of the shorter tree traversal and the efficient bucket access in the leaf level.

In Figure 3.9 (b), we show the numbers of average cache misses of exact match query, which were obtained by Perfmon [34]. A typical replacement strategy for cache misses is the LRU (*least recently used*) algorithm. Because we did not flush the L2 cache after each query and the L2 cache can be fairly large on modern machines, running many queries on the index will eventually result in fewer cache misses per query, e.g. some highly accessed cache lines can always reside in the L2 cache. Note that there is no index node residing in the cache before the operations. So smaller index size can benefit more from this mechanism as the upper levels of the tree can always reside in the cache.

Based on the results of the previous experiment, we pick the node size for the B^+-tree to be 128 bytes, the node size for the CSB^+-tree to be 64 bytes, and the bucket size for both Range Hashing to be 64 bytes. For the BD-tree, the internal node has size 128 bytes, and each leaf node has 64 buckets, and each bucket has 64 bytes. These will be used as the default settings for this and all subsequent experiments.

Now we evaluate the performance of each scheme on exact match queries with optimal node sizes. To simulate this, we first built a tree with 10,000,000 keys, and then conduct up to 1M operations of exact match queries, and record their

performance. Figure 3.10 shows the results.

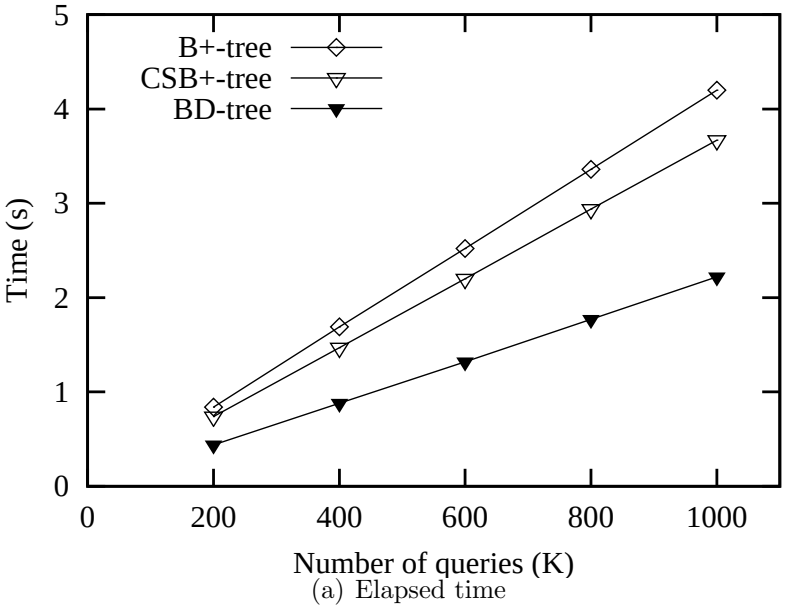
For exact match query, as expected, the BD-tree provides the best performance; while the B^+-tree and CSB^+-tree is about 40% slower. The reasons are obvious. The BD-tree performs well for two main reasons. First, the search within each leaf is very fast: only need to access one cache line size bucket (at most two) and perform a search within the bucket. Second, the tree is shorter, thus it's faster to access the target leaf, because it incurs less cache misses. The CSB^+-tree performs better than the B^+-tree for exact match query (about 15% faster) because it better utilizes each cache line and has fewer cache misses.

We note that as the number of searches increases, the gaps between the various schemes widen, showing that the BD-tree makes better use of the L2 caches compared to the CSB^+-tree and B^+-tree. We also note that the gap between the CSB^+-tree and B^+-tree is less than what is reported in [77]. The reason is that the optimal node sizes for two trees are different, 64 bytes for the CSB^+-tree and 128 bytes for the B^+-tree. In [77], the authors set both the node size to 64 bytes; in that case, the gap is much larger as the B^+-tree is not optimally tuned. In our experiment, when both node sizes are 64 bytes, the B^+-tree is about 25% slower than the CSB^+-tree.

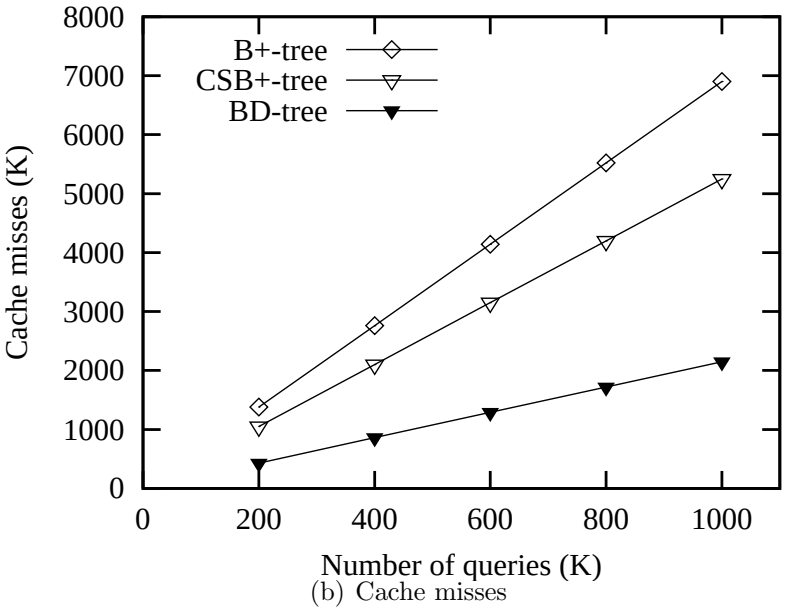
3.4.3 Performance of Range Query

In this experiment, we study the various schemes' performance using range queries. First we study the effect of node size on the performance by varying the node size. For range queries, we have a mix of four queries with different selectivity (0.01%, 0.1%, 1%, 10% of total number of records). The results are shown in Figure 3.11.

As expected, we note that a large node size benefits all the schemes. In our experiments, our range queries retrieve a large number of data, so the cost to



(a) Elapsed time



(b) Cache misses

Figure 3.10: More on search performance

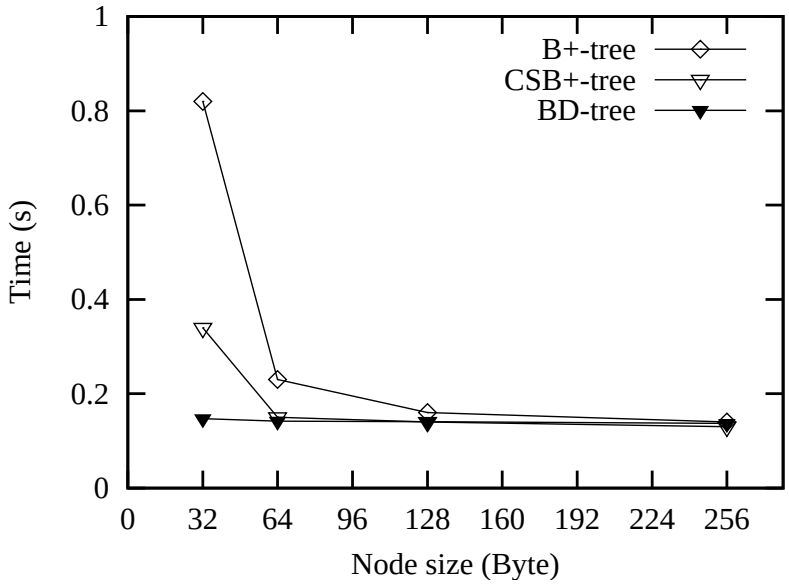
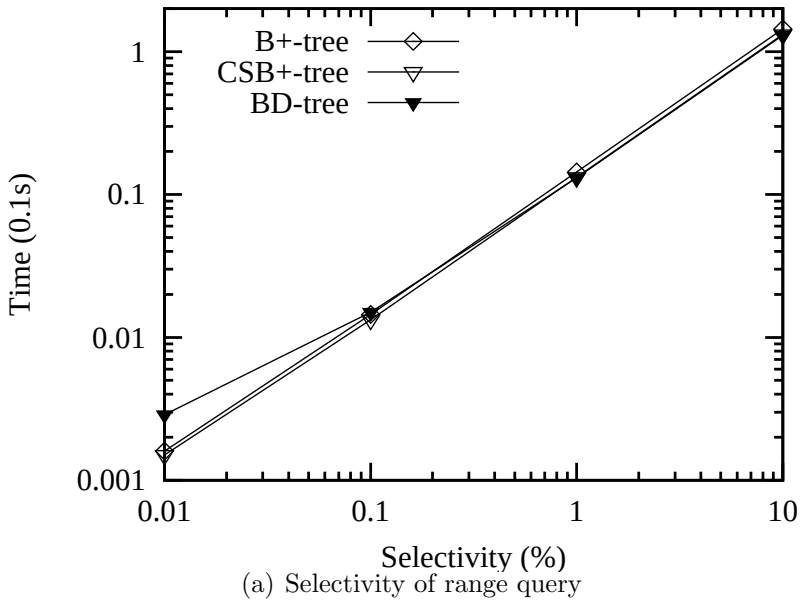


Figure 3.11: Effect of node size for range query

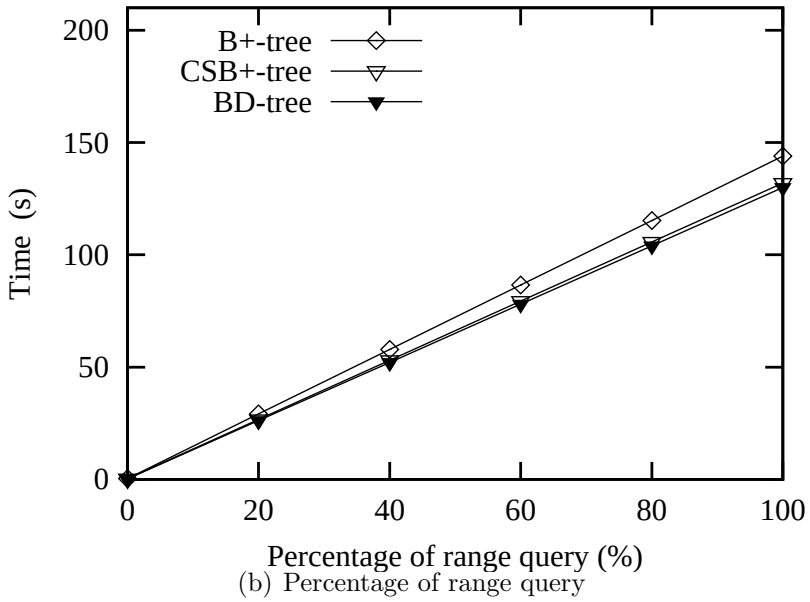
traverse the tree is negligible compared to that of leaf nodes to be accessed. Larger node size causes fewer cache and TLB misses.

The CSB$^+$-tree is better than the B$^+$-tree in all cases, because in our implementation of CSB$^+$-tree, leaf nodes with the same parent node are stored sequentially in memory. Thus, fewer TLB misses are incurred when we scan the leaf nodes. For the BD-tree, each leaf node (partition) has much more keys than the CSB$^+$-tree and the scan of leaf nodes causes fewer TLB misses. This results in the BD-tree providing good range query performance. This is expected since the BD-tree is able to "balance" the distribution of key values across uneven ranges.

We also conducted two additional experiments to evaluate the performance of the various schemes on range queries. Before the experiment we built the trees with 10,000,000 keys. In this experiment, we study the various schemes' performance on range queries. We vary the selectivities of the queries ranges from 0.01% to 10% of the total number of keys. The result is shown in Figure 3.12 (a). The cost of range queries falls into two parts, tree traversal to locate the leaf node containing



(a) Selectivity of range query



(b) Percentage of range query

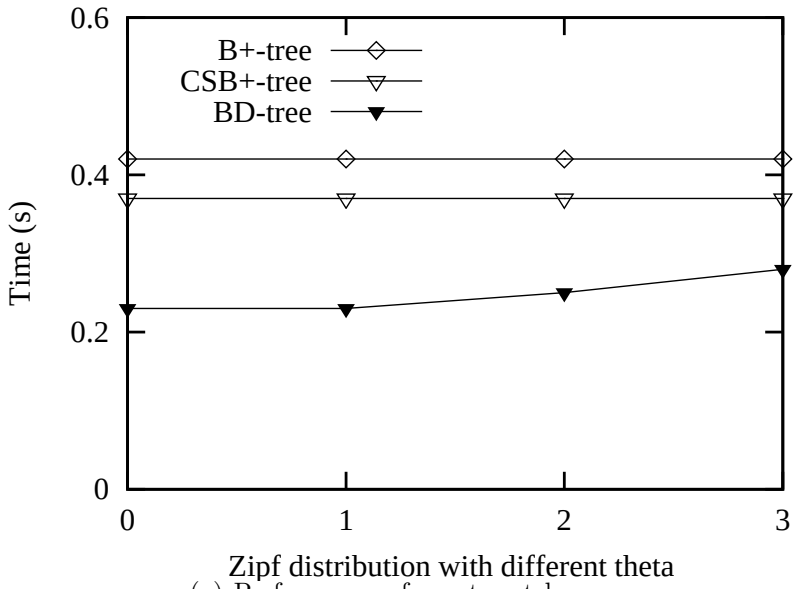
Figure 3.12: More on search performance

the first record and scanning all leaf nodes that contain the answers. The CSB$^+$-tree is always better than the B$^+$-tree, because the CSB$^+$-tree can locate the leaf faster and fewer TLB misses are incurred for scan, as the leaf nodes with the same parent node are stored sequentially in memory. Although the BD-tree can locate the leaf node much faster, the disorder within the leaf incurs some additional cost. Therefore, when the range selectivity is small, e.g. 0.01% selectivity, the BD-tree shows poorer results because it needs to access the whole leaf. On the other hand, the BD-tree performs as well as the CSB$^+$-tree and better than the B$^+$-tree when the selectivity is larger than 0.1% in our experiment. Each leaf node of the BD-tree has much more keys than the other two trees, and hence the scan of leaf nodes causes fewest TLB misses.

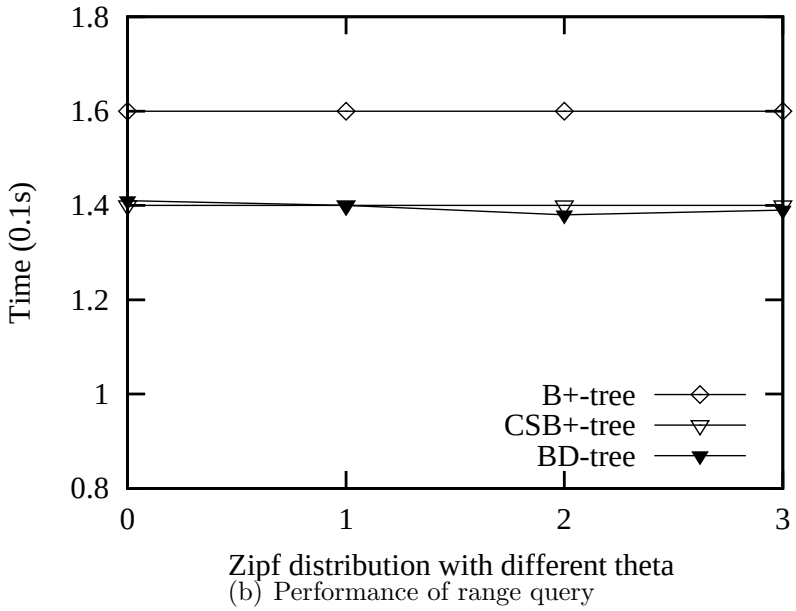
In the second experiment, we evaluated a mix of exact match and range queries over 100K operations. Figure 3.12(b) shows the results (total processing time) as the percentage of range queries increases from 0% (i.e. 100% exact match queries) to 100% (i.e. 0% exact match queries). The relative performance of the various scheme is similar to that of the previous experiment (see Figure 3.12(b)) where the B$^+$-tree is the most inefficient, and both the CSB$^+$-tree and BD-tree are equally good.

3.4.4 Effect of Duplication

In this experiment, we study the effect of key duplication. This is realized by changing the θ value of the Zipf distribution. We vary θ from 0 (unique keys) to 3 (highly duplicated), the maximum duplication is 10. Figure 3.13 shows the result of this experiment. For range query, we have a mix of four queries with different selectivity as in previous experiment. For exact match query, the B$^+$-tree and CSB$^+$-tree yield similar performance for different key distributions. While the



(a) Performance of exact match query



(b) Performance of range query

Figure 3.13: Effect of θ

BD-tree performs worse when the keys are highly distributed, as the chain can be long if there are more duplicates. However, it is still at least 20% better in all the cases. For the range query, we can see that the BD-tree can benefit more from the data duplication, because all the chained buckets are stored consecutively in the memory, and the long chain means better utilization of space, and hence incurs fewer cache misses and TLB misses.

3.4.5 Performance of Insertion

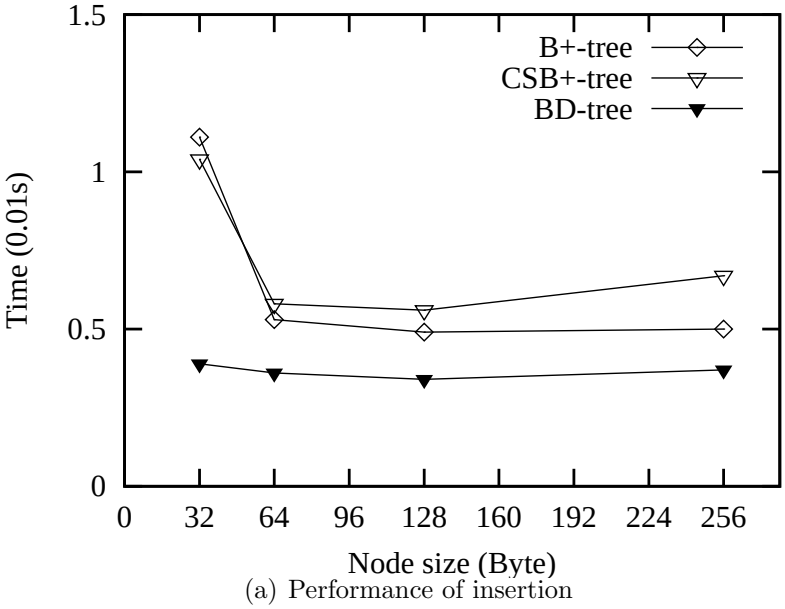
We also study the insertion performance of the various schemes. Figure 3.14(a) shows the result of 1000 insertion for various node sizes; Figure 3.14(b) shows the result as the number of keys inserted increases from 200K keys to 1000K keys.

In most of the cases, we observe similar performance: the BD-tree is the most efficient; the CSB$^+$-tree is the worst. The B$^+$-tree is slightly better than the CSB$^+$-tree.

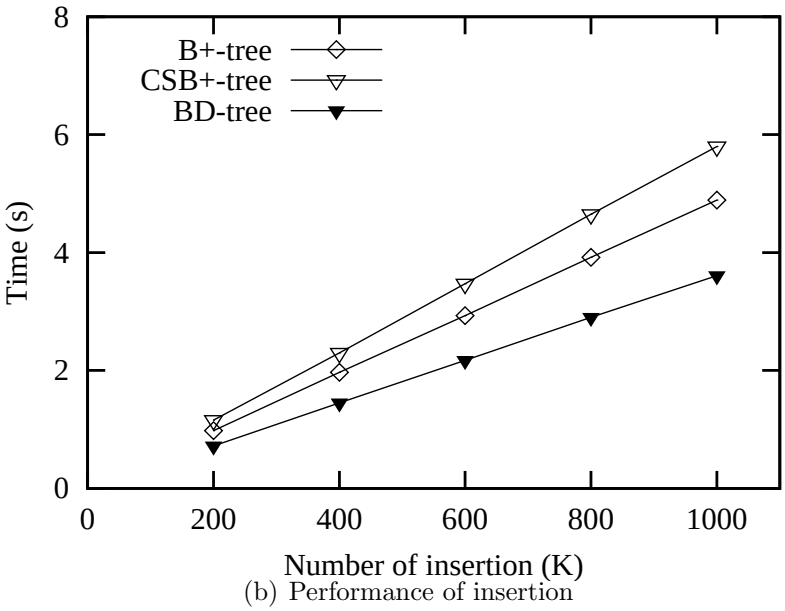
The CSB$^+$-tree is worse than the B$^+$-tree for the following reasons. The insertion cost has three parts: search cost, sort cost and split cost. The split cost of the CSB$^+$-tree includes copying a complete node group, whereas a node split of the B$^+$-tree is creating a new node. The BD-tree performs better than the B$^+$-tree because the leaf nodes of the BD-tree have much more keys than those of the CSB$^+$-tree and B$^+$-tree, thus the tree is shorter; at the same time inserting a key within a leaf partition is also very efficient as we only need to calculate the hash value and insert the key into the target bucket.

3.4.6 Storage Efficiency

We also looked at the storage efficiency of the various schemes. We evaluated the various schemes on datasets with sizes from 2 million to 10 million records. The



(a) Performance of insertion



(b) Performance of insertion

Figure 3.14: Comparison on insertion performance

result is shown in Figure 3.15.

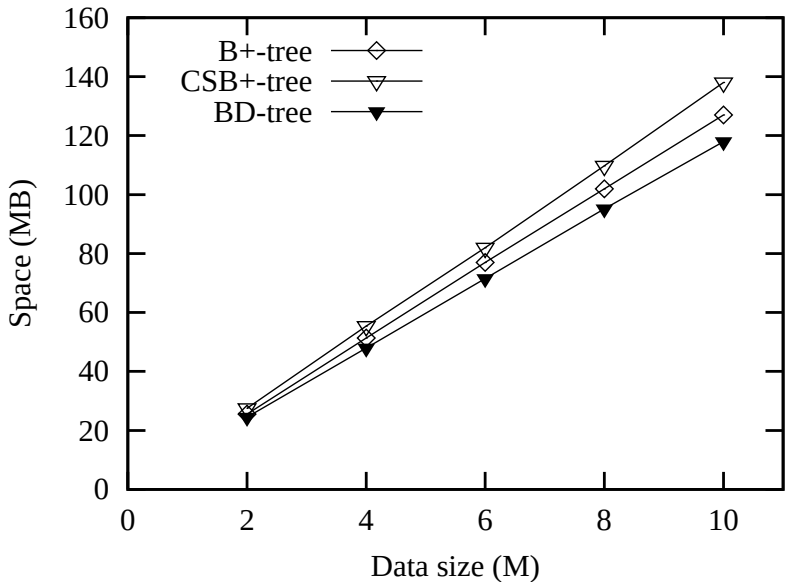


Figure 3.15: Space cost for different index methods

The space requirement of the CSB$^+$-tree is a little more than the B$^+$-tree, because although the internal node space of the CSB$^+$-tree is less, the utilization of space in nodes is about 70% for them and some space of leaf nodes is occupied by pointers and the counters on the number of keys. This overhead is relatively large as the node size is small, note that the node sizes of CSB$^+$-tree and B$^+$-tree are 64B and 128B respectively. In case that the node sizes are same, the CSB$^+$-tree is more efficient.

It turns out that the BD-tree incurs the least storage space. While this may appear surprising, our investigation finds that this is so because each leaf can hold a lot more keys. Recall that each bucket is allowed to be chained with extra buckets. In our case, we find that some of the buckets are chained whose first bucket is full, therefore the overall space utilization of buckets can be improved. Furthermore, because each leaf holds many more keys, the number of internal nodes are much fewer.

3.4.7 Performance of Join

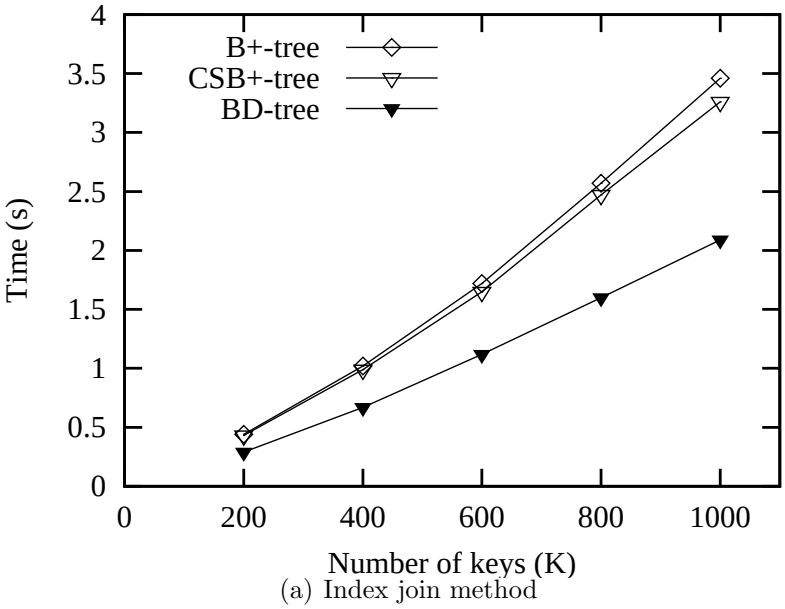
In this experiment, we investigate how the three *tree* index schemes will perform for join operations. We use binary relations of 8-bytes-wide tuples [key, pointer]. We assume that the two join relations are of the same size. Both the source relations have the same set of tuples. However, one of the relations, say R, is assumed to have no index, while the other, say S, is indexed by one of the methods. In addition, we randomized the tuples of R so that records are not ordered in the same manner as S[1]. We implement two join methods. The first is an indexed nested loops join that operates as follows: for each tuple of R, traverse the index of S to find matching tuples. The second is a sort-merge join algorithm. We sort the keys of R first, and a merge join can be used on the sorted R relation and the index of S.

We vary the relation cardinalities from 200K to 1M. The relations are joined on keys (i.e. no duplicates) so that the join relation has the same size as the source relations. Moreover, the join result contains the [key, pointer, pointer] combinations of matching tuples. The results of the experiments are shown in Figure 3.16.

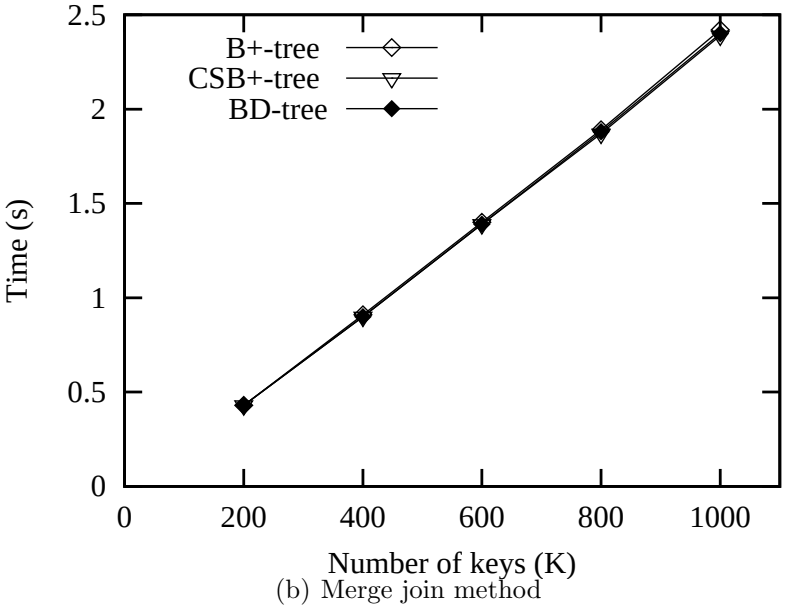
From Figure 3.16(a), we note that BD-tree is superior for index join. This is not surprising as the main cost of the operation is the search cost to find a matching S tuple for every R tuple, and searching in the BD-tree is much faster than both the B^+-tree and CSB^+-tree. The index join cost of BD-tree is about 40% faster than the other two methods.

Figure 3.16(b) shows the results for the merge join cost of the various schemes. As shown, all the three schemes perform equally well. Upon investigation, we find that the sort cost dominates the total join processing cost. For merge join, we only need to scan the leaf nodes. Because the keys in the leaf nodes of BD-tree are not

[1]Another way of looking at this is a key-foreign key join: there exists a secondary index on R but the secondary key does not match the primary key of S; so, the leaf nodes of R must be scanned.



(a) Index join method



(b) Merge join method

Figure 3.16: The comparison of join

sorted, the match cost in leaf nodes is more expensive; however, the TLB misses of the BD-tree are fewer. It turns out that the overall performances of all structures are almost the same.

Comparing the two figures (Figure 3.16(a) and (b)), we make an interesting observation. For the B^+-tree and CSB^+-tree, the merge join cost is less than the index nested loops join. On the contrary, for the BD-tree, the index nested loops join algorithm is the preferred method!

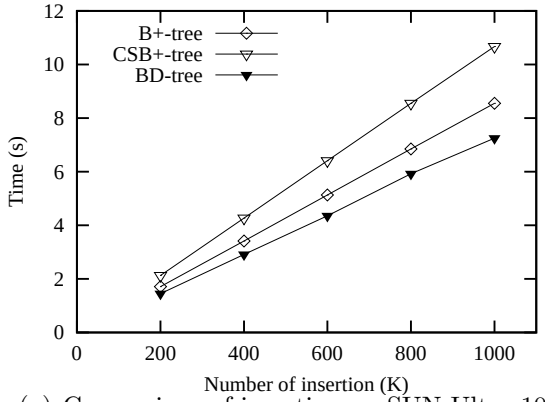
3.4.8 Performance on Different Architectures

In this experiment, we study the performance of the three tree index schemes on other three different machines with different system capacity, e.g. SUN Ultra 10, Sun E450 and an Intel PC. Table 3.2 shows a set of specific parameters that describes the respective hardware configuration and characteristics.

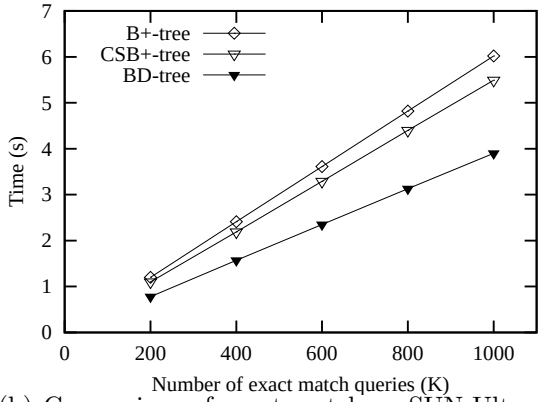
Description	SUN Ultra 10	SUN E450	Intel PC
OS	SunOS 5.7	SunOS 5.7	Windows XP
CPU Speed (MHZ)	UltraSPARC-II 333	UltraSPARC-II 480	P4 1.6G
Main-memory Size	256 MB	4 GB	1 G
L2 cache Size	2 MB	4 MB	256 KB
L2 cache line size	64 B	64 B	64 B
L2 miss latency	122 ns = 41 cy	195 ns = 94 cy	105 ns = 168 cy
TLB entries	64	64	64
Page size	8KB	8KB	4KB
TLB miss latency	158 ns = 53 cy	106 ns = 51 cy	80 ns = 120 cy

Table 3.2: Hardware configuration and characteristics

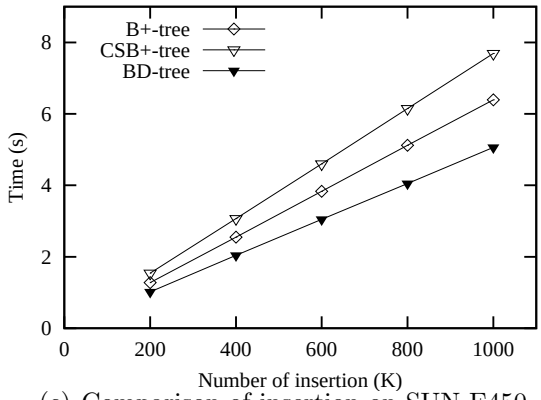
The dataset we used in this experiment consists of 10 million integers; and then we conduct 200K to 1M exact match query and insertion operations on these three machines and record the time. The results are shown in Figure 3.17. From the figure, we can see that the performances on all the machines exhibits similar pattern; the BD-tree performs best for exact match query and insertion on all the



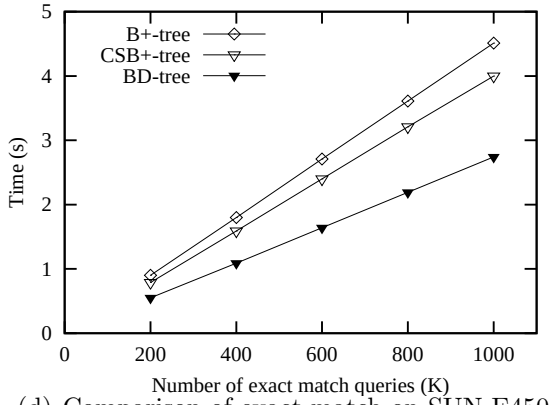
(a) Comparison of insertion on SUN Ultra 10



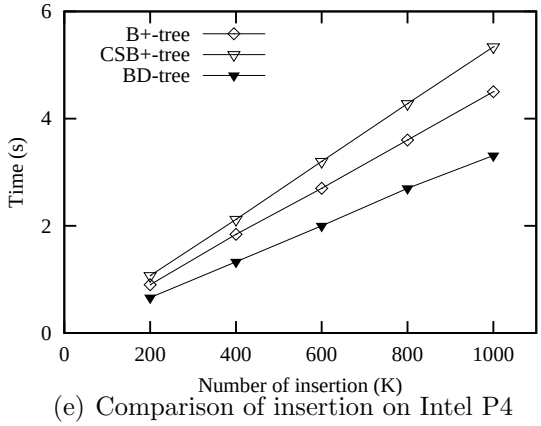
(b) Comparison of exact match on SUN Ultra 10



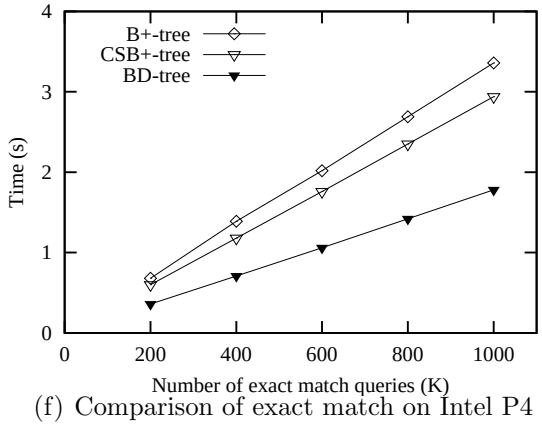
(c) Comparison of insertion on SUN E450



(d) Comparison of exact match on SUN E450



(e) Comparison of insertion on Intel P4



(f) Comparison of exact match on Intel P4

Figure 3.17: Performance on different architectures

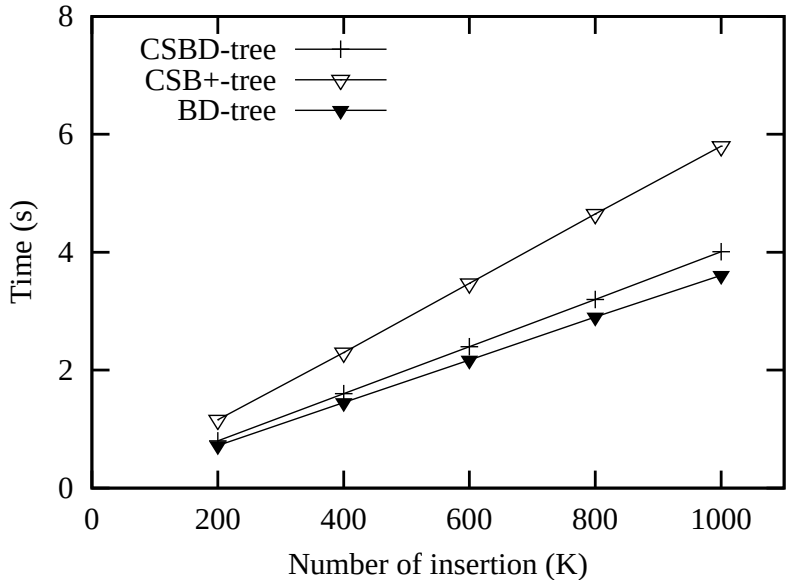


Figure 3.18: Insertion on CSBD-tree

machines. The gap between the BD-tree and the other indexes is largest on the Intel P4, which has the largest CPU speed. The performance gain that can be achieved by the BD-tree is likely to be more due to a shorter tree height and better utilization of cache memory.

3.4.9 Performance of the CSBD-tree

As mentioned earlier, the first tier of the BD-tree can be applied to any hierarchical indexing structure. In this section, we extended the idea to CSB$^+$-tree, i.e. we built a CSBD-tree which essentially replaces the B$^+$-tree component of BD-tree by the CSB$^+$-tree. In this experiment, we compare the performance of the BD-tree, CSB$^+$-tree and CSBD-tree. Figure 3.18, 3.19 and 3.20 show the results.

As shown in Figure 3.18, the insertion cost of the CSB$^+$-tree has been greatly reduced when hashing is introduced to form the CSBD-tree, although it remains costlier than that of the BD-tree due to the CSB$^+$-tree structure. For search performance, we observe that the CSBD-tree provides the best exact match query

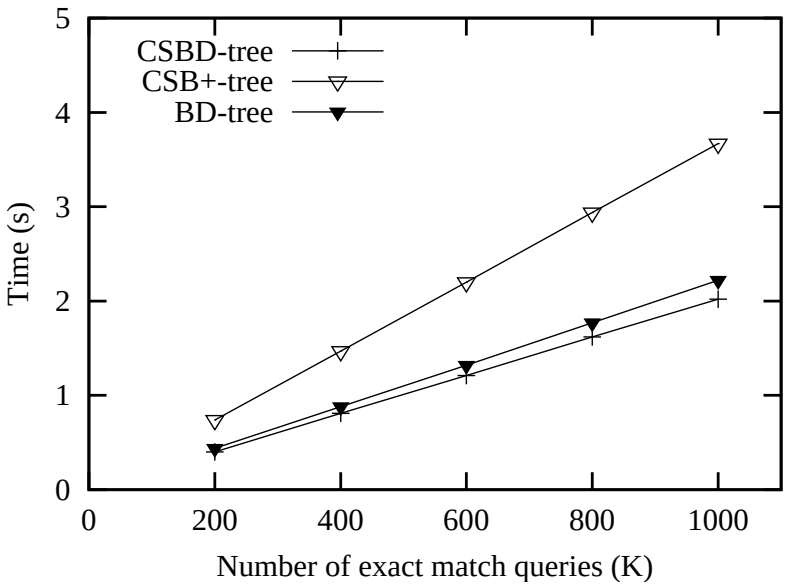


Figure 3.19: Exact match query on CSBD-tree

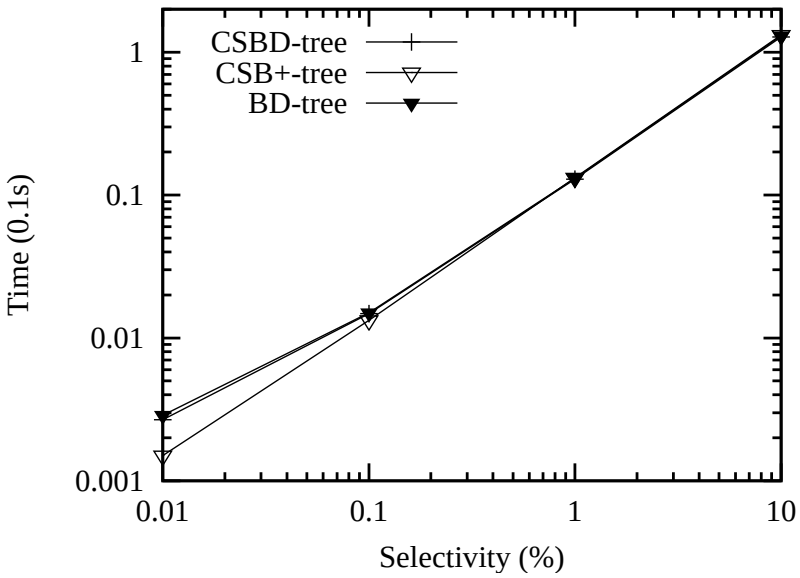


Figure 3.20: Range query on CSBD-tree

performance (Figures 3.19). This is expected as CSBD-tree has the advantages of BD-tree now: fewer internal nodes and hence shorter tree structure, and hash-based method to facilitate fast lookup at the leaf. The range query performance is better than that of BD-tree, shown in Figure 3.20, as the sibling leaf nodes of the CSBD-tree are saved locally, making the scanning of leaves more efficient.

3.5 Summary

In this chapter, we have revisited the problem of accessing single-dimensional data in main memory databases. We optimized the BD-tree to facilitate fast search in main memory environment. The BD-tree taps on the strengths of a hierarchical tree structure to evenly distribute and range partition the keys (facilitating efficient range queries processing) and the efficiency of hash-based methods for processing exact match queries. We studied the BD-tree against the B^+-tree and CSB^+-tree analytically and empirically. Our results showed that the BD-tree is a promising index structure for memory-based processing.

CHAPTER 4

Exploitation of Dimensionality Reduction for Memory High-dimensional Indexing

4.1 Introduction

With an increasing number of new database applications such as multimedia content-based retrieval, time series and scientific databases, the design of efficient indexing and query processing techniques over high-dimensional datasets becomes an important research area. These applications employ the so called feature transformation which transforms important features or properties of data objects into high-dimensional points, i.e. each feature vector consists of D values, which correspond to coordinates in a D-dimensional space. Searching for objects based on these features is thus a search of points in this feature space. In these applications, one of the most frequently used and yet expensive operations is to find objects in the high-dimensional database that are similar to a given query object, e.g. nearest neighbor search and similarity range query.

There is a long stream of research on processing high-dimensional data, and many index structures have been proposed [19, 8, 20, 28, 29, 65, 88, 90, 41, 13,

14, 82]. However, these index structures have largely been studied in the context of disk-based systems where it is assumed that the databases are too large to fit into the main memory. This assumption is increasingly being challenged as RAM gets cheaper and larger. The existing index structures are not appropriate for main memory indexes. For example, the number of active dimensions of the TV-tree can be large, which means that it still suffers from the dimensional scalability problem; the $full$ dimensionality of M-tree introduces high cache misses and distance computation; although the VA-file can reduce the cache misses, it incurs more computational cost.

As random access memory gets cheaper, it becomes increasingly affordable to build computers with large main memories, but main memory data processing is not as simple as increasing the memory buffer size. This has prompted renewed interest in research in main memory databases [18, 22, 26, 55, 77]. In main memory systems, minimizing L2 cache misses and computation cost has been an active area of research. Several main memory indexing schemes have been designed to be *cache conscious* [22, 55, 77, 32], i.e., these schemes page the structure based on cache blocks whose sizes are usually 32-128 bytes. However, these schemes are targeted at single or low dimensional data (that fits in a cache line), and cannot be effectively deployed for high-dimensional data. First, for high-dimensional data, the query processing is computationally expensive, which involves large amounts of distance calculation [20]. Second, the size of a high-dimensional point (e.g., 256 bytes for 64 dimensions) can be much larger than a typical L2 cache line size. Actually [55] shows that even for 2-dimensional data, the optimal node size can be up to 512 bytes, and increases as the dimensionality increases. Therefore an efficient main memory index should minimize the distance computation to improve the performance, and also exploit the L2 cache effectively as well.

To deal with the high-dimensional data in main memory environment, we propose a novel multi-tier index structure, called Δ-tree[1], that can facilitate efficient KNN search in main memory environment. Each tier in the Δ-tree represents the data space as clusters in different number of dimensions and tiers closer to the root partition the data space using fewer number of dimensions. The numbers of tiers and dimensions are obtained using the Principal Component Analysis (PCA) technique [50]. After PCA transformation, the first few dimensions of the new data space generally capture most of the information, and in particular two points that are distance d_i apart in i dimensions have the property that $d_i \leq d_j$ if $i \leq j$. More importantly, by reducing the number of dimensions in internal nodes, we can better utilize the L2 cache. We present the algorithms of insertion, deletion and different kinds of queries for the Δ-tree. An extension of the Δ-tree, called Δ^+-tree, is also proposed to further reduce the search space. The Δ^+-tree globally clusters the data space and then partitions clusters into small regions before building the tree. We compare the proposed schemes against other known schemes including the M-tree [29], TV-tree [65], iDistance [90], VA-file [88], CR-tree [55], Slim-tree [51], Omni-technique [38], Pyramid-tree [11] and Sequential Scan. Our experimental study shows that the Δ^+-tree is superior to the other indexes.

Not that, the BD-tree that we introduced in previous chapter is not suitable for high-dimensional databases. Although some space filling curves are mappings from a D-dimensional space into a single dimensional space, most distance information is lost when the dimensionality is high. And hence, the queries can not be conducted efficiently.

[1] Δ reflects the structure of the tree where nodes closer to the root index keys with lower dimensions, while those towards the leaf index keys with higher dimensions.

4.2 Principal Component Analysis

The Principal Component Analysis (PCA) [50] is a widely used method for transforming points in the original (high-dimensional) space into another (usually lower dimensional) space [24, 47]. It examines the variance structure in the dataset and determines the directions along which the data exhibits high variance. The first principal component (or dimension) accounts for as much of the variability in the data as possible, and each succeeding component accounts for as much of the remaining variability as possible. Using PCA, most of the information in the original space is condensed into a few dimensions along which the variances in the data distribution are the largest.

We shall briefly review how the principal components are computed. Let the dataset contains N D-dimensional points. Let A be the $N \times D$ data matrix where each row corresponds to a point in the dataset. We first compute the mean and covariance matrix of the dataset to get the *eigenmatrix*, V, which is a $D \times D$ matrix. The first principal component is the eigenvector corresponding to the largest eigenvalue of the variance-covariance matrix of A, the second component corresponds to the eigenmatrix with the second largest eigenvalue and so on.

The second step is to transform the data points into the new space. This is achieved by multiplying the vectors of each data point with the *eigenmatrix*. More formally, a point P(x_1, x_2, ..., x_D) is transformed into $V \times P = (y_1, y_2, \ldots, y_D)$. To reduce the dimensionality of a dataset to k, $0 < k < D$, we only need to project out the first k dimensions of the transformed points. The mapping (to reduced dimensionality) corresponds to the well known Singular Value Decomposition (SVD) of data matrix A and can be done in $O(N \cdot D^2)$ time [42].

Suppose we have two points, P and Q, in the dataset in the original D-dimensional space. Let P_{k1} and P_{k2} denote the transformed points of P projected

on $k1$ and $k2$ dimensions respectively (after applying PCA), $0 < k1 < k2 \leq D$. Q_{k1} and Q_{k2} are similarly defined. The PCA method has several nice properties:

1. $dist(P_{k1}, Q_{k1}) \leq dist(P_{k2}, Q_{k2}) \quad 0 < k1 < k2 \leq D$, where $dist(p, q)$ denotes the distance between two points p and q (See [24] for a proof).

2. Because the first few dimensions of the projection are the most important, $dist(P_k, Q_k)$ can be very near to the actual distance between P and Q for $k \ll D$ [24].

3. The above properties also hold for new points that are added into the dataset (despite the fact that they do not contribute to the derivation of the *eigenmatrix*) [24]. Thus, when a new point is added to the dataset, we can simply apply the *eigenmatrix* and map the new data from the original space into the new PCA space.

In [24], the data are organized into clusters, and PCA is employed to find the optimal number of dimensions for each cluster. Our work applies PCA differently. We use it to facilitate the design of an index structure that allows pruning at different levels with different number of dimensions. This can reduce the computational overhead and L2 cache misses.

4.3 The $\triangle$-tree

Handling high-dimensional data has always been a challenge to the database research community because of the dimensionality curse. In main memory databases, the curse has taken a new twist: a high-dimensional point may not fit into the L2 cache line, whose size is typically 32-128 bytes. Additionally, the distance computation of high-dimensional query occupies a large portion of the overall cost in

the absence of disk I/O. As such, existing indexing schemes are not adequate in handling high-dimensional data. In this section, we present a new index structure, called Δ-tree, to facilitate fast query processing in main memory databases. For the rest of this chapter, we assume that the dataset consists of D-dimensional points and use the Euclidean distance as the metric distance function.

4.3.1 The Index Structure of Δ-tree

The proposed structure is based on three key observations. First, dimensionality reduction is an important technique to deal with the dimensionality curse. In particular, by reducing the dimensionality of a high-dimensional point, it is possible to "squeeze" it into the cache line. Second, ascertaining the number of dimensions to reduce to is a non-trivial task. In addition, even if we can decide on the number of dimensions, it is almost impossible to identify the dimensions to be retained for optimal performance. Third, PCA offers a very good solution: the first component captures the most dominant information of points, the second the next most dominant, and so on. Moreover, as discussed in Section 4.2, it has several very nice properties.

Consider a dataset of D-dimensional points. Suppose we apply PCA on the dataset to transform the points into a new space that is also D-dimensional. We refer to the transformed space as PCA-Space. Consider a data point P in the PCA-Space, say $(x_1, \ldots, x_D)$. We define $\prod(P, m)$ to be an operator that *projects* point P on its first m dimensions ($2 \leq m \leq D$):

$$\prod((x_1, \ldots, x_D), m) = (x_1, \ldots, x_m).$$

Figure 4.1 shows a Δ-tree, which is essentially a multi-tier tree. The data space is

split into clusters and the tree directs the search to the relevant clusters. However, the dimensionality of indexing keys at each level of the tree is different — nodes closer to the root have keys with fewer dimensions, and the keys at the leaves are in the full dimensions of the data. We shall discuss how the number of levels of the tree and the number of dimensions to be used at each level can be determined shortly. For the moment, we assume that the tree has L levels and the number of dimensions at level k is m_k, $1 \leq k \leq L$, $m_i < m_j$ for $i < j$. Moreover, we note that the m_i dimensions selected for level i are given by $\prod((x_1, \ldots, x_D), m_i)$.

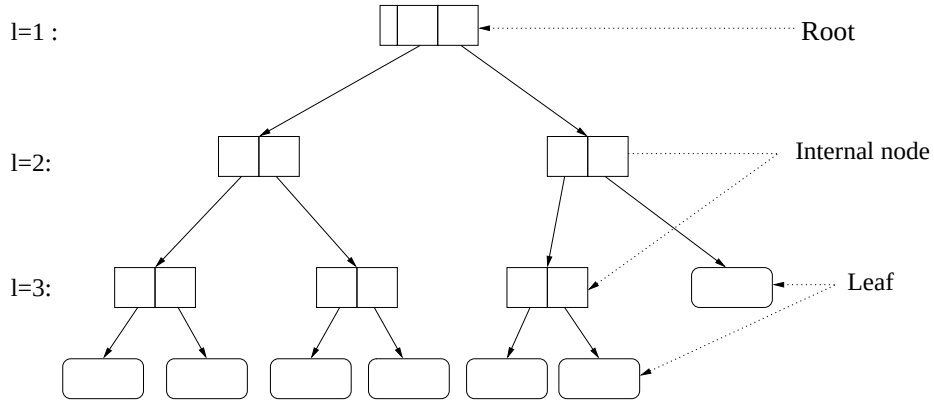


Figure 4.1: The Δ-tree

In the Δ-tree, the data is recursively split into smaller clusters at each level. This is done as follows. At level 1 (root), the data is partitioned into n clusters $C_1, C_2, \ldots C_n$. We employ a clustering algorithm for this purpose, and in our implementation we use the K-means scheme. The clustering is, however, performed using the m_1 dimensions in the PCA-Space of the transformed data. In other words, C_1 contains points that are clustered together in m_1 dimensions in the PCA-Space. At level 2, C_i is partitioned into $C_{i1}, C_{i2}, \ldots, C_{in}$ sub-clusters using the m_2 dimensions of the points in C_i in the PCA-Space. This process is repeated for each sub-cluster at each subsequent level l where each subsequent clustering process operates on the m_l dimensions of the points in the sub-cluster in the PCA-Space. At the leaf level

(level L), the full dimensions in the original space are used as the indexing key, i.e., the leaf nodes correspond to clusters of the actual data points.

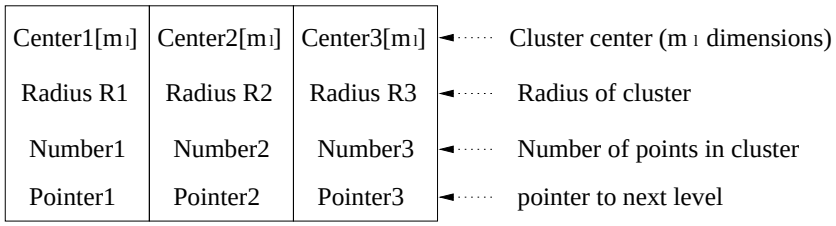


Figure 4.2: The structure of internal node

Figure 5.2 shows an example of an internal node of Δ-tree with three sub-clusters. An internal node at level l contains information of the cluster it covers at m_l dimensions, and each entry corresponds to information of a sub-cluster. Each entry is a 4-tuple (c_l, r, num, ptr), where c_l is the center of the sub-cluster obtained at level l, r is the radius of the sub-cluster, num is the number of points in the sub-cluster, and ptr is a pointer to the next level node. The root node has the same structure as an internal node except that it has to maintain additional information on the dataset. This is captured as a triple (L, m, $eigenmatrix$) header, where L represents the number of projection levels, $m = (m_1, m_2, \ldots, m_{L-1})$ is a vector of size $L-1$ representing the number of dimensions in each projection level (excluding the last level which stores the full dimensions of points), and $eigenmatrix$ is the eigenmatrix of the dataset after PCA processing.

We note that the Δ-tree can be used to prune the search space effectively. Recall (in property 1) that the distance between two points in a low dimensionality in the PCA-Space is always smaller than the distance between the two points in a higher dimensionality. Thus, we can use the distance at low dimensionality to prune away points that are far away (i.e., if the distance between a database point and the query point at low dimensionality is larger than the real distance of the current K-th NN, then it can be pruned away). More importantly, the lower dimensionality at

upper levels of the tree decreases the distance computational cost, and also allows us to exploit the L2 cache more effectively to minimize cache misses.

For the Δ-tree to be effective, we need to be able to determine the optimal number of levels and the number of dimensions to be used at each level. There are some methods to decide these values, such as fixed fan-out and fixed node size. However the last method incurs high implementation cost: the fan-out is related to the reduced dimensionality if the node size is fixed. For the simplicity of implementation, we fix the fan-out of the tree. For the number of levels, we adopt a simple strategy: we estimate the number of levels based on the fan-out of a node, e.g., given a set of N points, and a fan-out of f, the number of levels is $L = \lceil log_f N \rceil$.

To determine the number of dimensions m_l to be used at level l, our criterion is to select a cumulative percentage of the total variation that these dimensions should contribute [50]. Let the variance of the j-th dimension be v_j. Then, the percentage of variation accounted for by the first k dimensions is given by

$$V_k = \frac{\sum_{j=1}^{k} v_j}{\sum_{j=1}^{D} v_j}$$

With this definition, we can choose a cut-off V_l^* for each level. Suppose there are L projection levels, we have

$$V_l^* = \frac{l}{L}, 1 \leq l \leq L$$

Hence we can retain m_l dimensions in level l, where m_l is the smallest k, for which $V_k \geq V_l^*$. In practice, we always retain the first m_l dimensions which preserve as much information as possible.

Figure 4.3 shows the effect of cumulative variation of a dataset after apply-

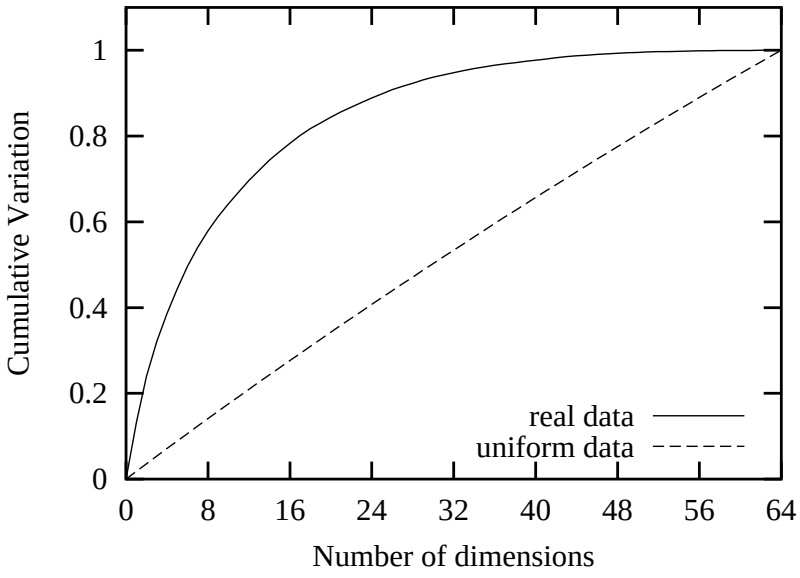


Figure 4.3: The proportion of cumulative variation

ing PCA. Two 64-dimensional datasets are used: a real dataset containing color histograms extracted from the Corel Database [1] and a synthetic dataset that is uniformly distributed. It is clear that for the real dataset (where the data is skewed), the first few dimensions in the PCA-Space is sufficient to capture the variation , e.g., the first 8 dimensions already capture the 60% of variation in the data. On the other hand, for uniformly distributed data, all the dimensions have similar variance. Suppose we have 5 projection levels, the resultant m_l for each level is shown in Table 4.1.

Dataset	$l = 1$	$l = 2$	$l = 3$	$l = 4$	$l = 5$
Real data	2	4	8	17	64
Uniform data	12	24	37	50	64

Table 4.1: m_l for different level

We note that for efficiency reason, we do not require the Δ-tree to be height-balanced. Since the data may be skewed, it is possible that some clusters may be large, while others contain fewer points. If the points in a sub-cluster at a level l

($< L$) fit into a leaf node, we will not partition it further. In this case, the height of this branch may be shorter than L. On the other hand, for a large cluster, if the number of points at level L is too large to fit into a leaf node, we further split it into sub-clusters using the full dimensions of the data. We have $m_l = D$ for $l > L$. However, in practice, we find that the difference in height between different subtrees is very small. Moreover, if we should bound the size of a cluster, we can control the height differences.

4.3.2 The Operations on Δ-tree

After the presentation of Δ-tree structure, we now introduce algorithms for various operations in this section.

The Δ-tree construction

Figure 4.4 shows the algorithm for constructing a Δ-tree for a given dataset. We have adopted a top down approach. At first, routine PCA() transforms the dataset into the PCA-Space (line 1). We treat the whole dataset as a cluster and refer to these new points as pC. In line 2, the function Init() initiates parameters of root node according to the information of PCA, such as the $eigenmatrix$, the value of L and the vector m. The default value of $m_l = D$ for $l > L$, and we do not save this value in the root node explicitly. In line 3, we call the recursive routine R_insert(node, pC, lev) that essentially determines the content of the entries of the node at level lev - one entry per subcluster. Note that we are dealing with points in the transformed space (i.e., pC), and that lev determines the number of dimensions that this node is handling.

In line 1 of R_insert(node, pC, lev), we partition the data of the cluster pC into K sub-clusters (by K-means). However, this partitioning is performed only

Algorithm Buildtree(dataset, tree)
Input: the high-dimensional dataset
Output: Δ-tree

1. $pC = PCA(dataset)$;
2. Init(root);
3. R_insert(root, pC, 1);

R_insert(node, pC, lev)
1. pClusters = LevCluster(pC, K, m_{lev});
2. for each $pC_j \in pClusters$
3. node.center[j] = $pCenter_j$;
4. node.radius[j] = $pRadius_j$;
5. if ($sizeof(pC_j) <$ leafsize)
6. New (leaf);
7. Insert_leaf(pC_j);
8. node.children[j]=leaf;
9. else
10. New (inter_node);
11. node.children[j]=inter_node;
12. R_insert(inter_node, pC_j, lev+1);

Figure 4.4: The algorithm of building Δ-tree

on the m_{lev} dimensions of the cluster. For each sub-cluster, in lines 3-4, we fill the information on the center and radius into the corresponding entry in *node*. If the number of points in a sub-cluster fit into the leaf node (lines 5-8), we insert the points into the leaf node directly. Otherwise (lines 9-12), we recursively invoke routine **R_insert()** to build the next level of the tree.

KNN Search of the Δ-tree

To facilitate KNN search, we employ two separate data structures. The first is a priority queue that maintains entries in non-descending order of distance. Each item in the queue is an internal node of the Δ-tree. The second is the list of KNN candidates. The distance between the K-th NN and the query point is used to

prune away points that are further away.

Algorithm KNNSearch(QueryPoint, tree, K)
Input: Δ-tree, query point Q, K
Output: K nearest neighbors

```
1. Queue = NewPriorityQueue();
2. KNN = NewKNN();
3. prune_dist = ∞;
4. Q' = GetPCA(eigenmatrix, Q);
5. Enqueue(Queue, root);
6. while (Queue is not empty)
7.     node = RemoveFirst(Queue);
8.     for( each child of node)
9.         if ( P_dist(child, Q', m_child.lev)<prune_dist )
10.            if ( child is an internal node )
11.                Enqueue(Queue, child);
12.            else
13.                for ( each point in leaf )
14.                    if(dist(point, Q)<prune_dist)
15.                        Insert (point, KNN);
16.                        Adjust(prune_dist);
```

Figure 4.5: KNN search algorithm for Δ-tree

We summarize the algorithm in Figure 4.5. In the first stage, we initialize the priority queue, KNN list and the pruning distance (lines 1-3). After that we transform the query point from the original space to the PCA-Space using the *eigenmatrix* in the root (line 4). In line 5, we insert the root node into the priority queue as a start. After that, we repeat the operations in lines 7-16 until the queue is empty. We get the first item of the queue which must be an internal node (line 7). For each child of the node, we calculate the distance from Q' to the sub-cluster in PCA-Space (distance is computed with $m_{child.lev}$ dimensions using P_dist()). If the distance is shorter than the pruning distance, we do as follows. If the child node is an internal node, it means that there is a further partitioning of the space into sub-clusters, and we insert the node into the queue(lines 10-11). Otherwise, the

child must be a leaf node, we access the real data points in the node and compute their distances to the query point; points that are nearer to the query point are then used to update the current KNN list (lines 12-15). The function Adjust() in line 16 updates the value of pruning distance when necessary, which is always equal to the distance between the query point and the K-th nearest neighbor candidate.

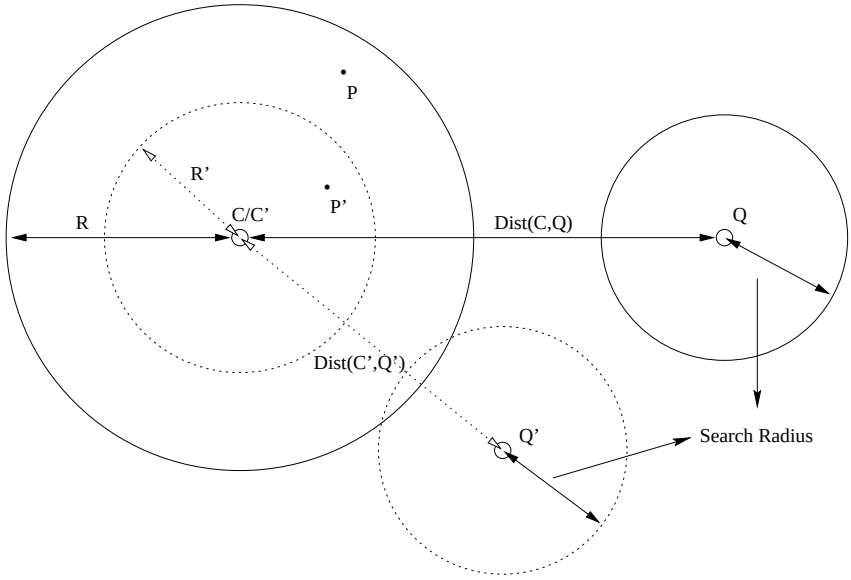


Figure 4.6: Prune with projection distance

We illustrate the effect of pruning a cluster in Figure 4.6. In the figure, Q represents the query point, and the solid circle represents the search space bounded by its distance to the current K-th NN. Q' represents the transformed point in a lower dimensional space. Note that the search radius remains the same, and the search region is denoted by the dotted circle. Consider a cluster centered on C (region bounded by solid circle). Since we only consider distance here, suppose the cluster on the transformed low-dimensional space is also centered at C (denoted C') but the region is smaller (bounded by dotted line). Suppose a point P in the

former is transformed to a point P' in the latter. We have the following equation:

$$dist(P, Q) \geq dist(P', Q') \geq dist(C', Q') - R'.$$

If $dist(C', Q') - R'$ is larger than the search radius, then all the points in this cluster cannot be nearer than the current K-th NN (since $dist(P, Q)$ must be larger than search radius also). Thus, we can prune away the cluster. As the values of C' and R' are already maintained in the Δ-tree, the computational cost is low, making the proposed scheme efficient.

Range Query of the Δ-tree

As we mentioned previously, a *window/range query* can be treated as a *similarity range query* around the center point of the given query range. Therefore we only present the algorithm which supports *similarity range query*, and call it *range query* shortly. Like KNN search algorithm, we employ two separate data structures. The first is a priority queue that maintains entries within the search distance. Each item is an internal node of the Δ-tree. The second is the list of current query results. The search distance, which defines the query range, is used to prune away points directly.

We summarize the range query algorithm in Figure 4.7. The algorithm is very straightforward. At first, we initialize the priority queue. After that we transform the query point from the original space to the PCA-Space using the *eigenmatrix* in the root (line 2). In line 3, we insert the root node into the priority queue as a start. After that, we repeat the operations in lines 5-13 until the queue is empty. We get the first item of the queue which must be an internal node (line 5). For each child of the node, we calculate the distance from Q' to the sub-cluster in PCA-Space (distance is computed with $m_{child.lev}$ dimensions using P_dist()). If the

Algorithm RangeQuery(QueryPoint, tree, dis)
Input: Δ-tree, query point Q, dis
Output: points within dis to Q

```
1. Queue = NewPriorityQueue();
2. Q' = GetPCA(eigenmatrix, Q);
3. Enqueue(Queue, root);
4. while (Queue is not empty)
5.     node = RemoveFirst(Queue);
6.     for( each child of node)
7.         if ( P_dist(child, Q', m_child.lev)< dis )
8.             if ( child is an internal node )
9.                 Enqueue(Queue, child);
10.            else
11.                for ( each point in leaf )
12.                    if(dist(point, Q)<prune_dist)
13.                        Insert point into result set;
```

Figure 4.7: Range query algorithm for Δ-tree

distance is shorter than the search distance dis, we do as follows. If the child node is an internal node, it means that there is a further partitioning of the space into sub-clusters, and we insert the node into the queue(lines 8-9). Otherwise, the child must be a leaf node, and we access the real data points in the node and compute their distances to the query point; if the distance between point and the query point is less than the search distance, the point is inserted into the result set.

Insertion of the Δ-tree

So far, we have seen the Δ-tree as a structure for static databases. However, the Δ-tree can also be used for dynamic databases. This is based on the properties of PCA. When a new point is inserted, we simply apply *eigenmatrix* on the new point to transform it into PCA-Space and insert it into the appropriate sub-cluster. To reduce the complexity of the algorithm, we only update the radius and keep the

original center of the affected cluster. This may result in a larger cluster space and degrade the precision of *eigenmatrix* gradually. Note that, the K-means clustering method can not guarantee no overlap between clusters, and more overlaps incurred by new updates may degenerate the query performance, but as our study shows, the Δ-tree is still effective after a great amount of updates.

Algorithm Insert(newpoint, tree)
Input: the Δ-tree, newpoint P

```
1. P' = GetPCA(eigenmatrix, P);
2. node = root;
3. while ( node is not leaf )
4.     NearSubCluster = FindBestCluster(node, P);
5.     Adjust( node_radius[NearSubCluster]);
6.     if (node_child[NearSubCluster] is internal node)
7.         node = node_child[NearSubCluster];
8.     else
9.         leaf = node_child[NearSubCluster];
10. if (leaf is not full)
11.    Insert(leaf, P);
12. else
13.    Split(leaf, newleaf)
14.    if (leaf.parent is not full )
15.        InsertLeaf(leaf.parent, leaf, newleaf)
16.    else
17.        New(inter_node)
18.        InsertNode(leaf.parent, inter_node)
19.        InsertLeaf(inter_node, leaf, newleaf)
```

Figure 4.8: Insert algorithm for Δ-tree

The algorithmic description of the insert operation is shown in Figure 4.8. We first transform the newly inserted point into the PCA-Space using the *eigenmatrix* in the root node (line 1). We then traverse down the tree (beginning from the root node (line 2)) for the leaf node to insert the point by always selecting the nearest sub-cluster along the path (lines 3-10). If the leaf node has free space, we insert the new point into the leaf (line 12). Otherwise, we must split the leaf node before

insertion. To split the leaf, we generate two clusters. If the parent node is not full, the routine InsertLeaf( ) (line 15) inserts two new clusters into the parent. Otherwise, we generate a new internal node and insert the leaf nodes (lines 17-19). In this case, a new internal node level is introduced. Another way to deal with the splitting is to propagate the split up to the root node. However since all the internal nodes are full when we build up the tree, the propagation will affect the space utilization of internal nodes and arise a new problem that how to control the reduced dimensionality of new root. We do not adapt this method in the implementation.

We note that our algorithm does not change the cluster *eigenmatrix* as new points are added. As such, it may not reflect the real feature of a cluster as more points are added, since the optimal *eigenmatrix* is derived from all known points. On the other hand, once we change the *eigenmatrix*, we have to update the keys of the original data points as well. As a result, insertion may make it necessary to rebuild the tree. Two mechanisms to decide when to rebuild the tree are as follows:

1. *Insertion threshold*: In this naive method, once the newly inserted points exceed a pre-determined threshold, say 50% of the original data size, we rebuild the tree regardless of the precision of original *eigenmatrix*.

2. *Distance variance threshold*: Given a set of N points, we form a cluster with center c. After PCA transformation, we have another center c' in the projected lower dimensional space. Originally we have the average distance variance V_a:

$$V_a = \frac{\sum_{i=1}^{N} |dist(P_i, C) - dist(P_i', c')|}{N}.$$

Once we insert a point P, we get a new average distance variance, called V_a':

$$V_a' = \frac{V_a * N + |dist(P, C) - dist(P', c')|}{N + 1}.$$

V_a' may change after every insertion, where the V_a' for N+1 points will be used as V_a when the V_a' for N+2 points is to be computed. Once $\frac{V_a' - V}{V}$ is larger than a pre-defined threshold, V is the original V_a. we rebuild the tree. This method is expected to be more efficient because the decision factor is related to the distribution of newly inserted points.

Deletion of the Δ-tree

In the above section, we have discussed the algorithm to deal with insertion of the Δ-tree. Although deletion is different from the insertion, we can apply the similar scheme on the delete operation. The details are shown as follows:

1. We first transform the point into the PCA-Space using the *eigenmatrix*. We then traverse down the tree for the leaf node holding the point. Either the KNN search or range query algorithm can be used to locate the point.

2. We delete the point, and check whether merging with a sibling node is possible in case of underflow.

3. Note that we do not change the cluster *eigenmatrix* in the first two steps. However, we record the effect of deletion on *eigenmatrix*. Like insert operations, large volumes of deletion may trigger the rebuilding of tree.

4.3.3 The Δ^+-tree: A Partition-based Enhancement of the Δ-tree

While the proposed Δ-tree can efficiently prune the search space, it has several limitations. First, its effectiveness depends on how well a dataset is globally correlated, i.e., most of the variations can be captured by a few principle components in the transformed space. For real datasets that are typically not globally correlated, more clusters may have to be searched. Second, the complete dataspace has to be examined for each query. Third, there is a need to periodically rebuild the whole tree for optimal performance.

In this section, we propose an extension called Δ^+-tree that addresses the above three limitations. To deal with the first limitation, we globally partition the dataspace into multiple clusters, and manage the points in each cluster with a Δ-tree. For simplicity, we also employ the K-means clustering scheme to generate the global clusters and apply PCA for clusters individually. We use a directory to save the information of global clusters. Each entry of the directory represents a global cluster, and has its own *eigenmatrix*, L, m, cluster center, radius and a pointer to the corresponding Δ-tree that manages its points.

Even with the proposed enhancement, the second limitation remains: each global cluster space has to be examined completely. Our solution is to partition the cluster into smaller regions so that only certain regions need to be examined. We made the following observations of a cluster:

1. Points close to each other have similar distance to a given reference point. The distance value is single dimensional and it can be easily divided into different intervals.

2. A cluster can be split into regions ("concentric circle") as follows. First, each

point is mapped into a single-dimensional space based on the distance to the cluster center. Second, the cluster is partitioned. Let $Dist_{min}$ and $Dist_{max}$ be the minimum and maximum distance of points within the cluster to the center. Let there be k regions (k is a predetermined parameter). The points in region i must satisfy the following equations:

$$\begin{cases} Dist_{min} + i * f \leq Dist_i \leq Dist_{min} + (i+1) * f & i = 0 \\ Dist_{min} + i * f < Dist_i \leq Dist_{min} + (i+1) * f & 1 \leq i < k \end{cases}$$

where $f = (Dist_{max} - Dist_{min})/k$. Figure 4.9 shows an example of a cluster with six regions.

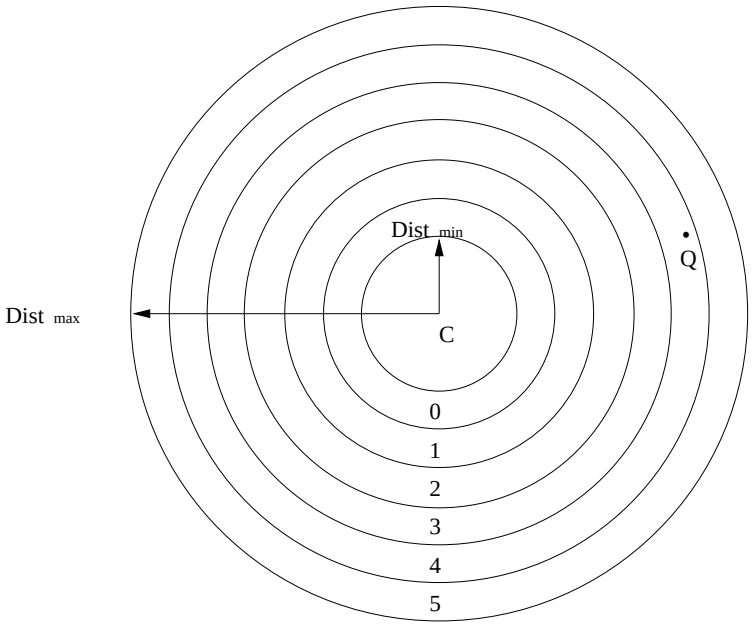


Figure 4.9: Cluster partitioning and searching

3. Given a query point, we can order the regions in non-descending order of their minimum distance to the query point. The regions are then searched in this order. This step can be efficiently performed by checking against the partitioning vectors (i.e., $Dist_{min}$, $Dist_{min} + f$, ..., $Dist_{min} + (k-1)f$) of

the region. For example, consider the query point Q in Figure 4.9. Q falls in region 4. As such, region 4 will be examined first, followed by regions 5, 3, 2, 1 and 0.

4. We note that this partitioning scheme can potentially minimize the search space by pruning away some regions. Using the same example as before, if the current KNN points after searching say region 5 are already nearer than the minimum distance between the query point and region 3, then regions 3, 2, 1 and 0 need not be examined. As shown in Figure 4.10, we only need to search the subtrees of certain partitions.

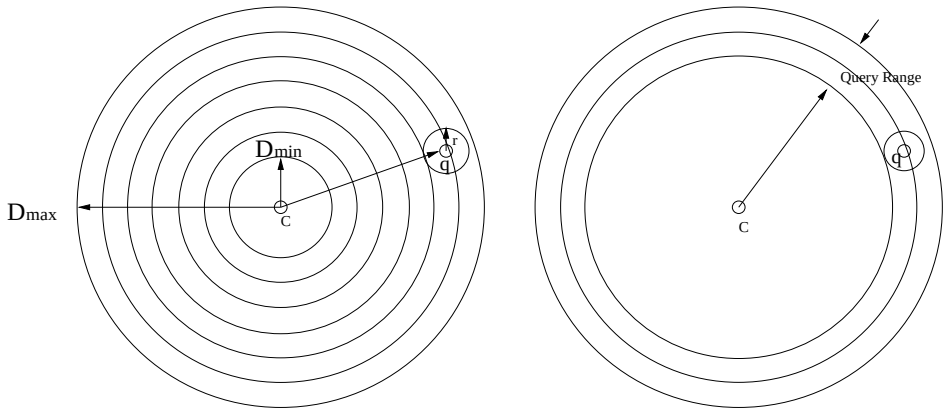


Figure 4.10: An example of pruning searching space

Based on these observations, we can introduce a new level immediately after the directory. In other words, instead of building a Δ-tree for each global cluster, we partition it as described above. For each region, we build a Δ-tree. We shall refer to this new structure as the Δ^+-tree. Figure 4.11(b) shows a Δ^+-tree structure. The whole dataset has two global clusters and we partition the cluster into 3 regions. For comparison purpose, we also show the Δ-tree (Figure 4.11(a)).

As described above, the operations on the Δ^+-tree are quite similar to those on the Δ-tree. For all the operations, we start from the nearest global cluster

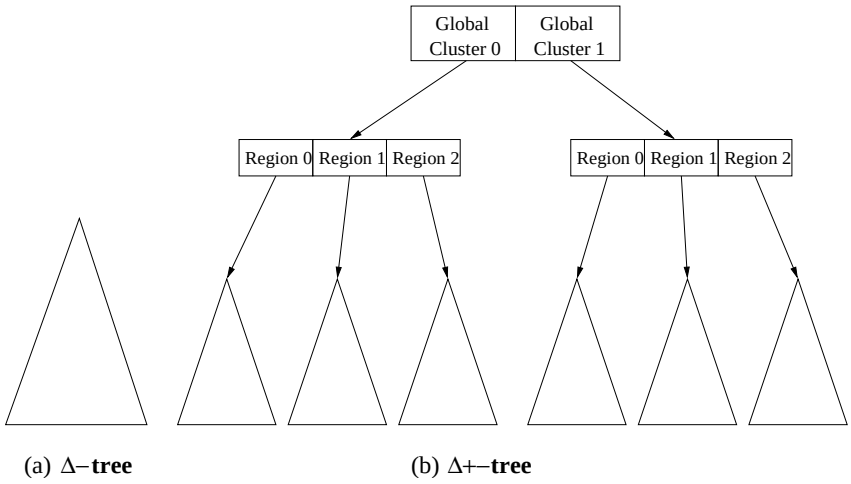


Figure 4.11: The structure of Δ-tree variants

and region, and then traverse the subtree. This process continues with other clusters, while cluster or regions containing points further than the K-th NN are not traversed.

The proposed Δ^+-tree is also more update efficient - while it cannot avoid a complete rebuild, it can defer a complete rebuild to a longer period (compared to Δ-tree). Recall that the structure partitions the data space into global clusters before PCA transformation. As such, it localizes the rebuilding to only clusters whose *eigenmatrix* is no longer optimal as a result of insertions, while other clusters are not affected at all. A complete rebuild would eventually be needed if the global clusters are no longer optimal. As we shall see in our experimental study, the index remains effective for a high percentage of newly added points.

4.4 Performance Study of Δ-trees

In this section, we present an experimental study to evaluate the Δ-tree and Δ^+-tree. The performance is measured by the average execution time, cache misses and distance computation for KNN search over 100 different queries. We use the

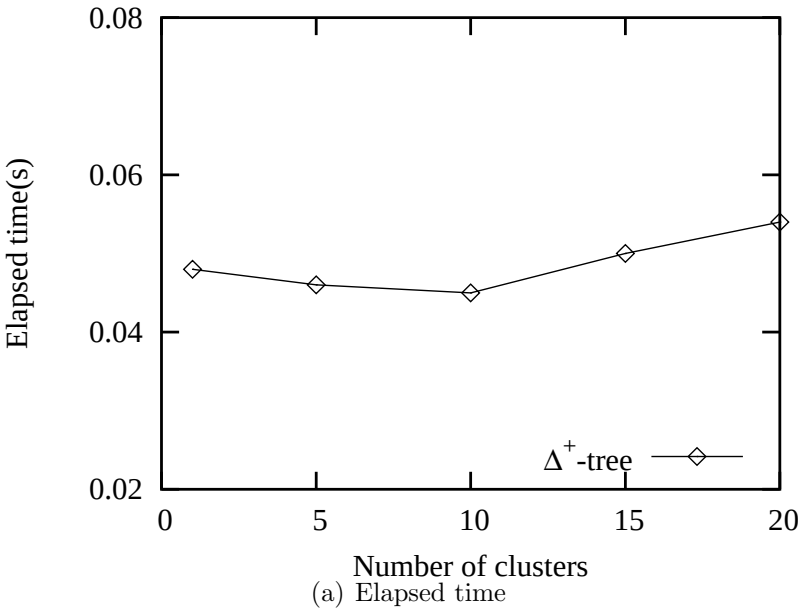
Perfmon tool [34] to count L2 cache misses. All the experiments are conducted on SUN Fire 4800 machine with 750MHz CPU, 16 GB RAM and 8 MB L2 cache. All the data and index structures are loaded into the main memory before each experiment begins. We demonstrate the results on the random dataset, synthetical clustered dataset and real-life datasets.

4.4.1 Tuning the Δ^+-tree

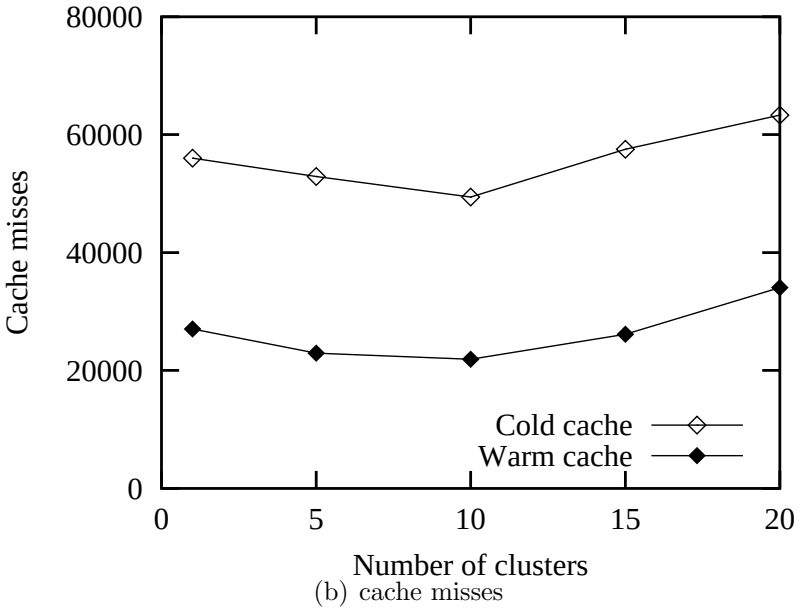
For the Δ^+-tree, it has two extra parameters: the number of global clusters and regions. When both parameters are set to 1, the Δ^+-tree becomes the Δ-tree. In the first experiment, we tune these two parameters of the Δ^+-tree. We use a real-life dataset consisting of 64 dimensional color histograms extracted from 70,000 color images obtained from the Corel Database [1].

First, we set the number of region to 1 and vary the number of clusters. The default fanout of the subtree we used is 10, which is near optimal as shown in the following section 4.4.2. Figure 4.12 shows the NN search performance of our new structures for different cluster numbers.

We observe that the performance of Δ^+-tree improves with relatively large number of clusters. As shown in Figure 4.12 (a), when the number of clusters is fewer than 10, the larger the cluster number the better the performance, although the performance is almost stable when the number is larger than 5. Because the image dataset is typically skewed, and hence it is possible for queries to be constrained within a certain cluster. In this case, the search space can be reduced efficiently compared with the Δ-tree, as we only need to search a subtree. However, the performance starts to degenerate when the number exceeds 10. If we partition the dataset into too many clusters, the query may incurs multiple subtree traversal, which introduce more cost overhead. The optimal number of cluster is a compro-



(a) Elapsed time



(b) cache misses

Figure 4.12: NN search for different cluster number

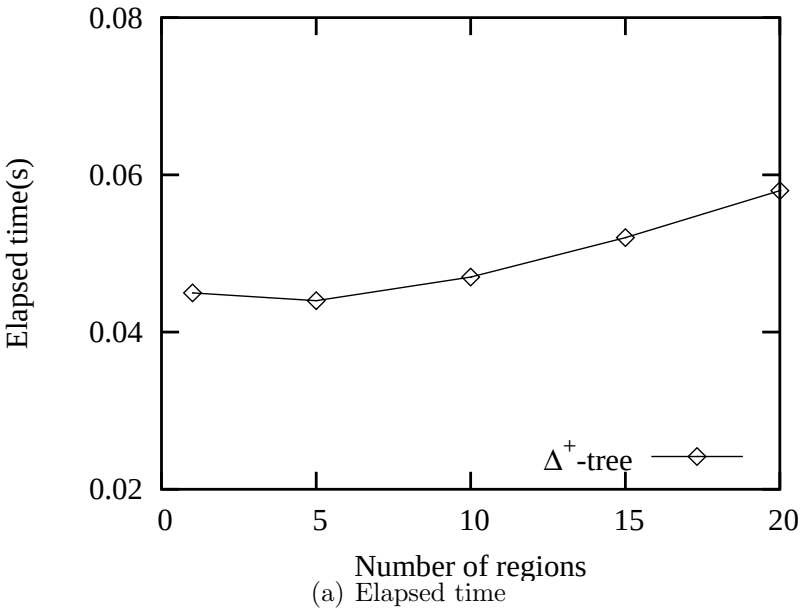
mise of the above factors, and may vary for different datasets. If we know the data distribution well, the tree can yield optimal performance when the number exactly matches the apriori number of clusters.

We conduct two sets of experiments to test the number of cache misses:

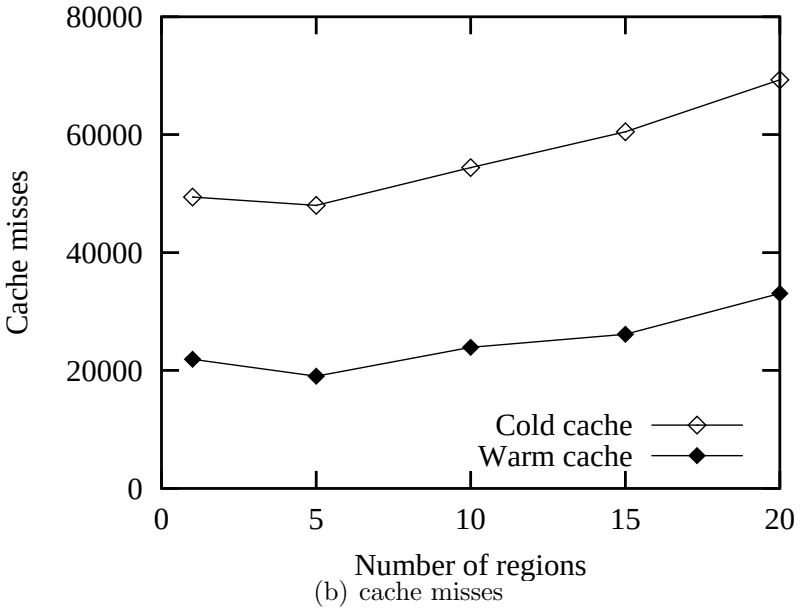
- *cold cache* : cache is flushed after every query; that is, the cache does not contain any index nodes or pages,
- *warm cache* : the queries run consecutively without cache flushing.

In both cases, we record the average number of cache misses for 100 queries. We found that the actual cache miss cost is only around 10% of the overall time cost if we run the queries consecutively, i.e. *warm cache*. This is the case in our studies as we do not perform any cache flushing between queries. Since we have run many queries on a single index structure, the cache hits are high, because some highly accessed cache lines can always reside in the L2 cache. However, the number of cache misses can be much larger in the case of *cold cache*. Our investigation shows that the number of cache misses for an independent query can be twice as much.

Another parameter of the Δ^+-tree is the number of regions. We set the cluster number to 10, and show the experiment result for varying number of regions in Figure 4.13. We found that the performance is near optimal when the number of regions is between 3 and 5 for this dataset. The query can be limited to a region, and hence the search space is reduced. Partitioning too many regions also introduces more subtree traversal which degrades the performance. The dataset affects the selection of region. When the KNN distance is small, we can partition the cluster into more regions to reduce the search space. On the other hand, when the KNN distance is large, the search space will overlap with some regions if we partition many regions, thus it incurs more tree operations and computations.



(a) Elapsed time



(b) cache misses

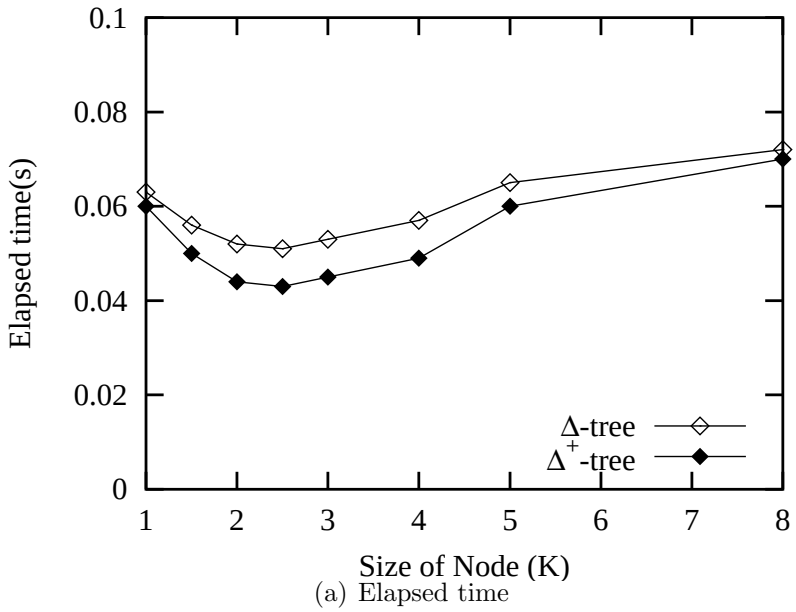
Figure 4.13: NN search for different region number

In this experiment, we test the performance of two extra parameters of the Δ^+-tree. However, the optimal values of the parameters may vary with different datasets. If we know the distribution of the dataset, we can apply the optimal value to the index structure. For simplicity, we set the default number of clusters and regions to be 10 and 5 respectively throughout the performance study. Additionally, we only show the average number of cache misses for 100 consecutive queries without cache flushing, i.e. *warm cache*.

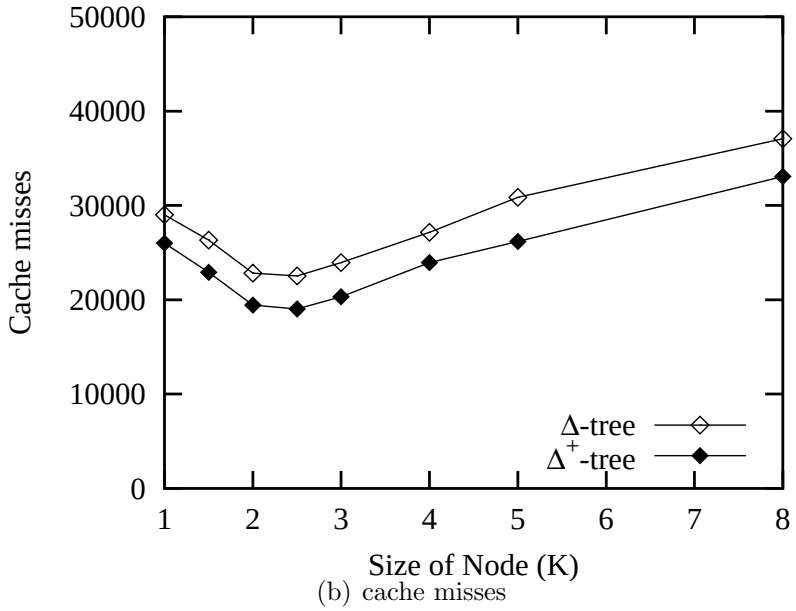
4.4.2 Comparing Δ-tree and Δ^+-tree

We conduct an extensive performance study to tune the two proposed schemes for optimality. Here we present one representative set that studies the effect of node size, i.e. we vary the node size from 1 KB to 8 KB. Although the optimal node size for single-dimensional datasets has been shown to be the cache line size [77], this choice of node size is not optimal in high-dimensional cases – the L2 cache line size on a typical modern machine is usually 64 bytes, which is not sufficient to store a high-dimensional data point (256 bytes for 64 dimensions) in a single cache block. [55] shows that even for 2-dimensional data, the optimal node size can be up to 256-512 bytes, and increases as the dimensionality increases. The minimum cache misses is a compromise of node size and tree height. High-dimensional indexes require more space per entry, and therefore the optimal node size is larger than the cache line size.

Figure 4.14 shows the NN search performance of our new structures for different node sizes. The node size here represents the size of leaf node. Since we fix the fan-out of tree in our implementation, i.e. all the levels have same fan-out, we have varied internal node size depending on the remaining dimensions in each level. As shown in the figure, there is an optimal node size that should be used. When the



(a) Elapsed time



(b) cache misses

Figure 4.14: NN search for different node size

node size is small ($< 2K$), the fan-out of the tree is also small. In multi-dimensional indexes, more than one node of the same level need to be accessed and the small fan-out introduce high overlap between nodes. As a result, more nodes will have to be accessed. We observe that as the node size increases, the number of accessed nodes decreases. The performance is optimal when the node size is around 2-3K. However, as the node size reaches beyond a certain point ($> 3K$), the performance starts to degenerate again. This is because too large a node size results in more cache misses per node. Therefore, the total cache misses increase. The optimal node size (2-3K) is a compromise of these factors.

The results, shown in Figure 4.14, clearly demonstrate the superiority of the Δ^+-tree over the Δ-tree: it is about 15% better than Δ-tree. This is because the Δ^+-tree searches a smaller data space compared to the Δ-tree. First, the Δ^+-tree globally partitions the dataset into clusters before PCA transformation, thus the *eigenmatrix* is more efficient than that of the Δ-tree. Second, the Δ^+-tree may only need to search a few clusters and exclude the other clusters that are far to the query point. Third, partitioning the cluster into regions can further reduce the search space compared to the Δ-tree that examines the whole data space. Hence, the total cost of cache misses and computation is reduced by the Δ^+-tree.

Since Δ^+-tree performs better than Δ-tree, in the following experiments, we shall restrict our discussion to the Δ^+-tree, and use the optimal node size determined above.

4.4.3 Comparison with other structures

In this section, we compare the Δ^+-tree with some existing methods on different datasets, e.g. the TV-tree, Slim-tree, M-tree, Pyramid-tree, CR-tree, VA-file, iDistance, Omni-sequential and Sequential Scan. To ensure a fair comparison, we

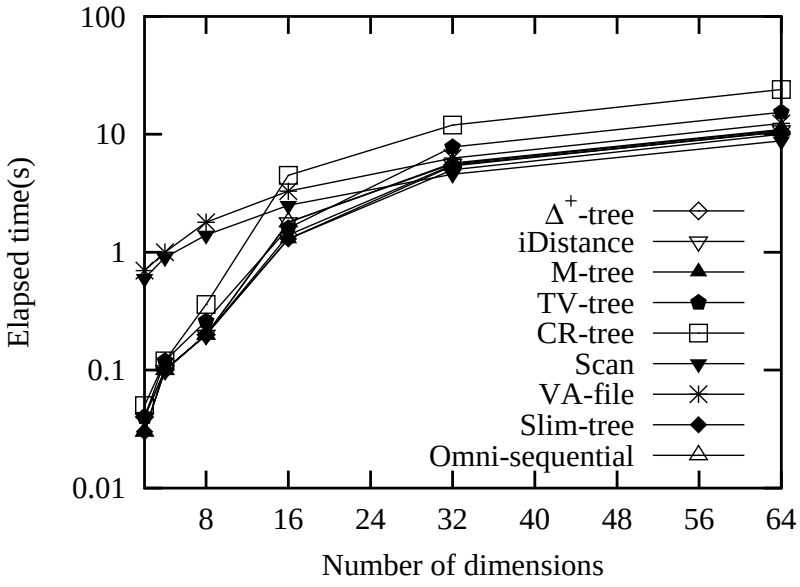


Figure 4.15: The comparisons of NN search time

optimize these methods for main memory indexing purposes such as tuning the node size[2]. We only present the optimal result of each structure.

On uniformly distributed dataset

In this experiment, we first generate a uniformly distributed dataset with up to 64 dimensions. The data size is 1,000,000 points. We present the results for NN queries only, and the results are summarized in Figure 4.15.

For uniformly distributed random data, all the index structures yield similar performance when the dimensionality is beyond 30. Because of large NN distance, all the methods have to access most of the data for a single query when dimensionality is high. As shown in [13], we can determine the expected distance of the query point to the nearest neighbor in the database. Assuming uniformly distributed dataset in a normalized data space $[0,1]^d$ with N points, the nearest neighbor distance can be approximated by the volume of the sphere which on the average

[2]One parameter of TV-tree is the number of active dimensions α, e.g. optimal α is around 20 for real dataset.

contains one data point. Since the data space with radii r can be calculated by

$$sp^d(r) = \frac{\sqrt{\pi^d}}{\Gamma(d/2+1)} \cdot r^d,$$

where $\Gamma(n)$ is the gamma function ($\Gamma(x+1) = x \cdot \Gamma(x), \Gamma(1) = 1$ and $\Gamma(1/2) = \sqrt{\pi}$). Since $sp^d(dist^{nn}) = \frac{1}{N}$ we can get the the expected nearest neighbor distance

$$dist^{nn}(N,d) = \frac{1}{\sqrt{\pi}} \cdot \sqrt[d]{\frac{\Gamma(d/2+1)}{N}}.$$

Based on this formula, the $dist^{nn}$ can become larger than the length of the data space, i.e. 1, when the dimensionality is higher than 30. Because of large NN distance, we can see that it is almost impossible to partition the data space well, thus a tree index cannot be efficient for uniformly distributed data.

From the figure, we can see that Sequential Scan is the best scheme for high-dimensional data space ($D > 30$). This is expected for uniformly distributed dataset as the distance to the NN is too large, and we must scan all the data even if we have an index. Since we need to access all the data when the number of dimensions is high, the tree structures cause more TLB misses and cache misses – accessing the internal nodes is the overhead. Not surprisingly, the VA-file is worse than Sequential Scan. As mentioned, the VA-file needs three cost overhead: decoding cost, computational cost and actual data access. In disk-based environments, disk I/O cost is dominant, so the cost overhead does not affect the performance of the VA-file much. But, in main memory systems, the search cost is bounded by CPU cost. Although the VA-file can reduce the cache misses compared to Sequential Scan, the cost overhead overwhelms the gain. The performances of the Δ^+-tree, Slim-tree, Omni-sequential, M-tree and iDistance are quite similar, because they are all distance-based schemes. When the data are uniformly distributed, PCA is

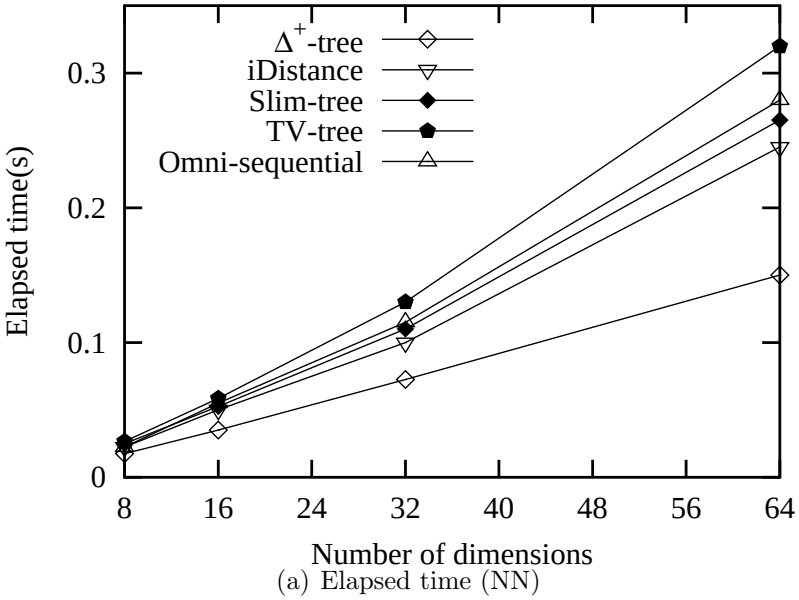
useless as all the dimensions have the same weight. When the dimension is low (2 or 4), the CR-tree is efficient and performs as well as the Δ^+-tree. However, since the R-tree is not scalable to high-dimensionality, the CR-tree's performance starts to degrade as the number of dimensions increases; additionally it incurs higher computational cost (to uncompress the MBRs). The performance of the TV-tree is only better than the CR-tree when the dimensionality is high, because although the TV-tree uses reduced dimensions, its structure is similar to the R-tree and dimensionality reduction is not efficient for uniformly distributed dataset.

For the rest of this chapter, we only focus on the TV-tree, Slim-tree, iDistance, Omni-sequential, Sequential Scan and the Δ^+-tree. The VA-file is worse than Sequential Scan, and is expected to perform worse when the dataset is skewed because of bit conflict. We exclude the CR-tree due to its poor performance for high-dimensional datasets. Because the Slim-tree is optimized from the M-tree and performs better, we also omit the M-tree in the comparison for the clarity of figures.

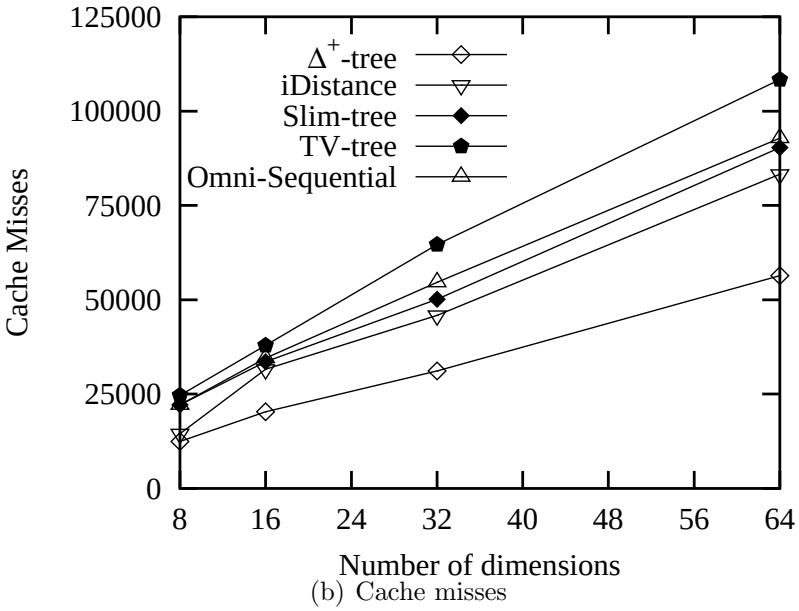
On clustered dataset

In many applications, data points are often correlated in some ways. In this set of experiments, we evaluate the Slim-tree, Omni-sequential, TV-tree, iDistance, Sequential Scan and Δ^+-tree on clustered datasets. We generate the data for different dimensional spaces ranging from 8 to 64 dimensions, each having 10 clusters. We use a method similar to that of [24] to generate the clusters in subspaces of different orientations and dimensionalities. All datasets have 1,000,000 points. We vary the cluster number of the Δ^+-tree, as we expected, the tree yields optimal performance when the number of clusters exactly matches the apriori number of clusters used in the Δ^+-tree (which is 10).

Figure 4.16 shows the time and cache misses of NN search as we vary the



(a) Elapsed time (NN)



(b) Cache misses

Figure 4.16: NN search for clustered datasets

number of dimensions from 8 to 64. Since the cost of memory access is mainly determined by cache access, and the node sizes varies for different indexes, we omit the details of node accesses. Because Sequential Scan performs poorly and all these tree structures achieve a speedup by a factor of around 15 over the Sequential Scan, we omit Sequential Scan from the figures to clearly show the differences between the other schemes. The reason is clear: for Sequential Scan, we must scan the whole dataset to get the nearest neighbors, and the cost is proportional to the size and dimension of the dataset. When the dataset is clustered, the nearest neighbors are (almost) always located in the same cluster with the query point. Thus, the tree structures can prune most of the data when traversing the trees.

Our Δ^+-tree can be 60% faster than the other three methods, especially when the dimensionality is high. The Δ^+-tree has three advantages over the Slim-tree. First, the Δ^+-tree reduces the cache misses compared to the Slim-tree. Because we index different levels of projections of the dataset, the number of projected dimensions in the upper levels is much smaller than that of real data. When the original space has 64 dimensions, the dimensions in the first three upper levels are fewer than 15. The node size of the Δ^+-tree in the upper levels can be much smaller than that of the Slim-tree; consequently the index size of the Δ^+-tree is also smaller. In the tree operations, upper levels of the tree will probably remain in the L2 cache as they are accessed with high frequency, so the Δ^+-tree can benefit more from this property. Furthermore, the internal nodes of the Δ^+-tree are full and fewer node accesses are needed because of hierarchical clustering. The effect is that the Δ^+-tree can reduce more cache misses compared to the Slim-tree. Second, the computational cost of Δ^+-tree is also smaller than the Slim-tree. The reason is in the projection level the distance computation is much faster because of reduced dimensionality. For example, when the dimension of the projection is 8, it already

captures more than 60% of the information of the point. This means we can prune the data efficiently in this projection level as the computational cost is only 12.5% of the actual data distance computation. Third, we build the tree from top down exploiting the global clustering compared with local partition of Slim-tree, so the benefit of the Δ^+-tree is also from the improved data clustering which reduce the search space, and hence the fewer operations for a query.

Comparing with the iDistance, although the B^+-tree structure of iDistance is more cache conscious, the iDistance incurs more distance computation and cache misses than the Δ^+-tree, because the search space of iDistance is larger. Omni-sequential also transforms the high-dimensional data into single dimensional space based on a set of global foci, and use pre-computed distance to prune the distance calculations. Although these two methods exploit different pruning mechanisms, mapping high-dimensional data to 1-dimensional space is less efficient in filtering data compared to the Δ^+-tree because of the heavy information loss.

Not surprisingly, the TV-tree performs worst among these index methods, especially when dimensionality is high. There are several reasons for this behavior. First, the TV-tree is essentially similar to the R-tree and its effectiveness depends on α, the number of active dimensions. It turns out that for the datasets used, the data are not globally correlated. As a result, the optimal α value for the TV-tree remains relatively large. For example, for 64 dimensions, we found that $\alpha \approx 20$. The reduced number of dimensions is still too large for an R-tree-based scheme (like the TV-tree) to perform well. Moreover, searching the reduced dimensions leads to false admissions which result in more nodes being accessed.

Next, we test the scalability of NN performance with respect to the data size. We fix the number of dimensions at 64, and vary the dataset size from 1,000,000 to 5,000,000 points. The result of NN search is shown in Figure 4.17. Because the cost

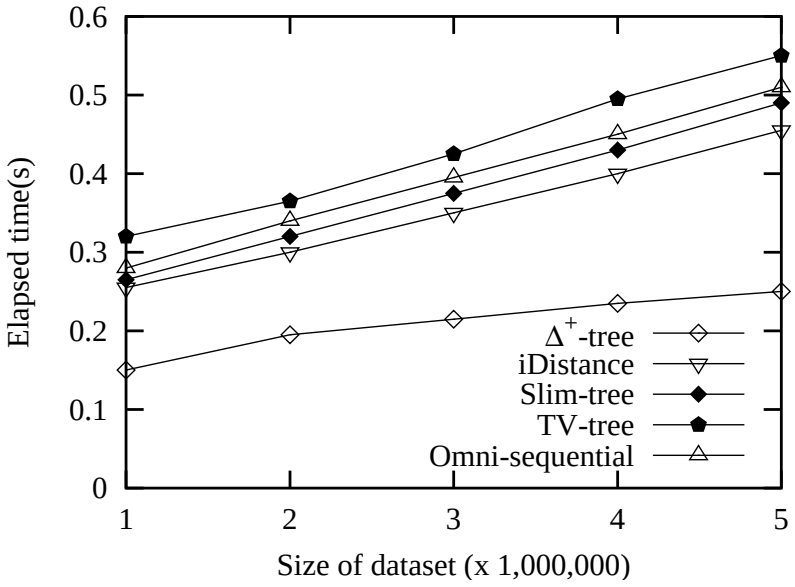
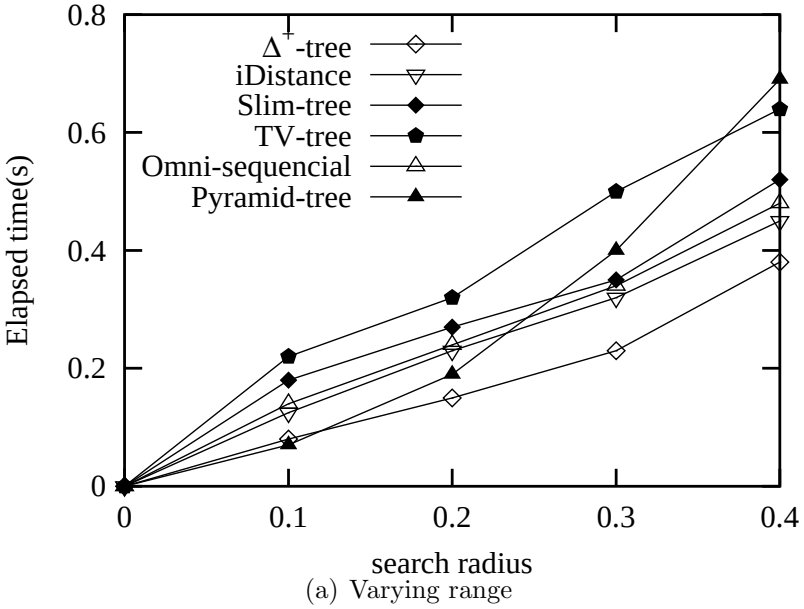


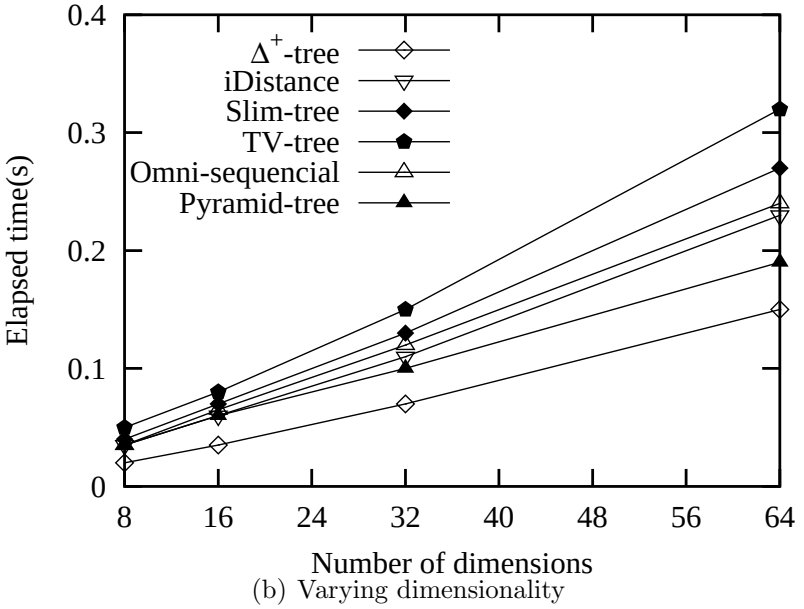
Figure 4.17: Scalability of index structures

of Sequential Scan increases almost linearly and is more expensive than other tree structures, we exclude it from the figure. The Slim-tree, TV-tree, Omni-sequential, iDistance and Δ^+-tree remain very effective for large datasets, demonstrating their scalability. In fact, the gap between these schemes and Sequential Scan widens as the dataset size increases. The relative performance between the five schemes remain largely the same as earlier experiments: the Δ^+-tree is the best scheme, followed by the iDistance, Slim-tree, Omni-sequential and finally TV-tree.

In the next experiment, we compare the range query performance for these indexes. Since the Pyramid-tree was specifically proposed for range search in high-dimensional space, we also include it in the comparison. Figure 4.18 shows the results of range search for 64-d clustered dataset with varying search radius from 0 to 0.4, and 0.2 search range with varying dimensionality. It is clear that all the indexes perform worse as we increase the search radius or the dimensionality, because the query have to search more space, which means more cache misses and distance computation. Although the Pyramid-tree performs well when the



(a) Varying range



(b) Varying dimensionality

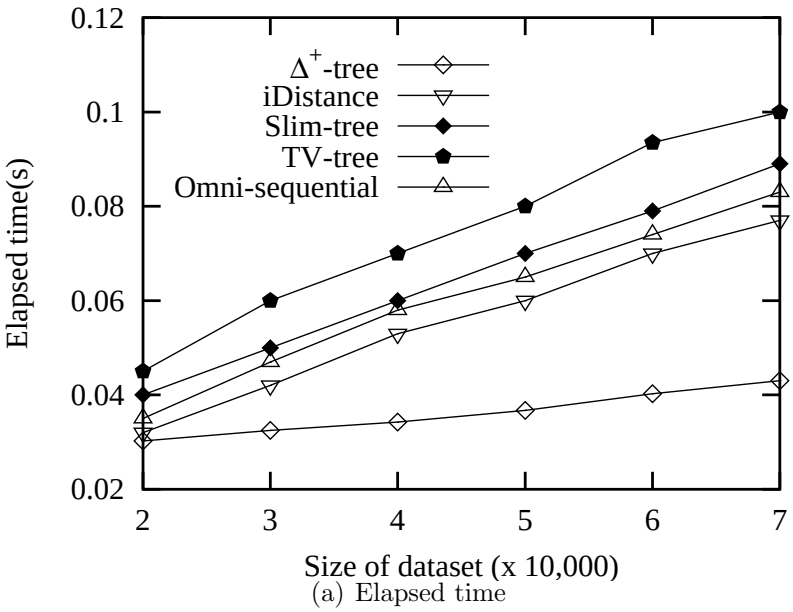
Figure 4.18: Range query for clustered dataset

search radius is small, it degrades faster than other methods. The Pyramid-tree is primarily designed and optimized for queries of small side length on uniform data, but it does not work well with big sized queries. The Δ^+-tree is the best among these structures for most of the case, however the gap between other indexes is narrowed. The reason is that a large search radius reduces the efficiency of the index technology.

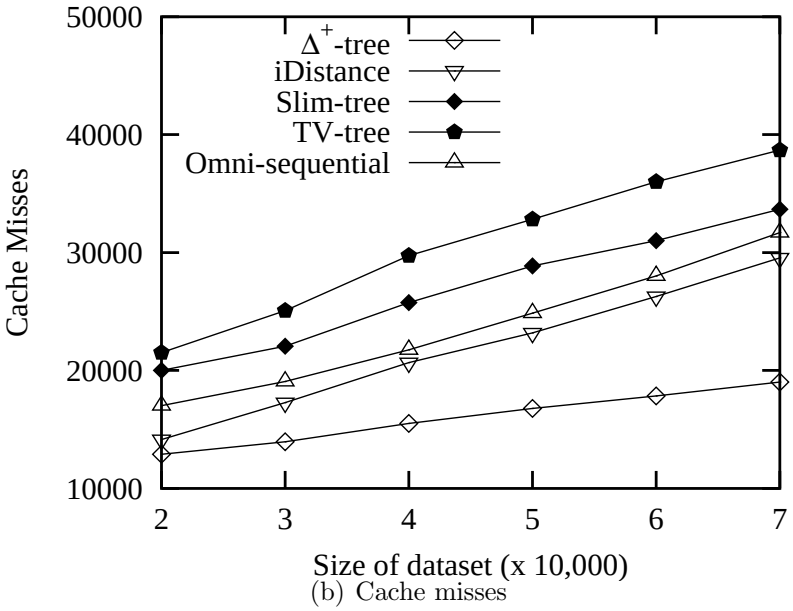
On real dataset

In this experiment, we evaluate the various schemes on the different real-life datasets, 70K 64 dimensional color histograms from the Corel Database [1] and 11K 79 dimensional motion capture dataset [**?**]. First, we test the performance of KNN search. The performance is quite similar to the clustered datasets. The tree-based methods are at least 10 times faster than Sequential Scan because the real dataset is generally skewed. As such, we will not present the results for Sequential Scan.

The NN performance comparisons for 64D dataset among the Δ^+-tree, Slim-tree, TV-tree, Omni-sequential and iDistance are shown in Figure 4.19. The Δ^+-tree is about 50% faster than other methods for NN search in terms of cache misses and time cost. In Figure 4.20, the comparisons of KNN performance for different K also show that the Δ^+-tree performs best. These results clearly show the effectiveness of the Δ^+-tree even if the number of clusters employed may not match that of the dataset. The Slim-tree is poor because it incurs more computation and it uses all the dimensions in the internal nodes resulting in more cache misses. Although it adjusts the nodes to reduce the overlap, the partition is not as good as that of the Δ^+-tree, because the Δ^+-tree clusters the dataset from top-down by capturing the overall data distribution and makes optimal partition. The iDistance and Omni-sequential are worse than the Δ^+-tree, because the distance information



(a) Elapsed time



(b) Cache misses

Figure 4.19: NN search for 64D real dataset

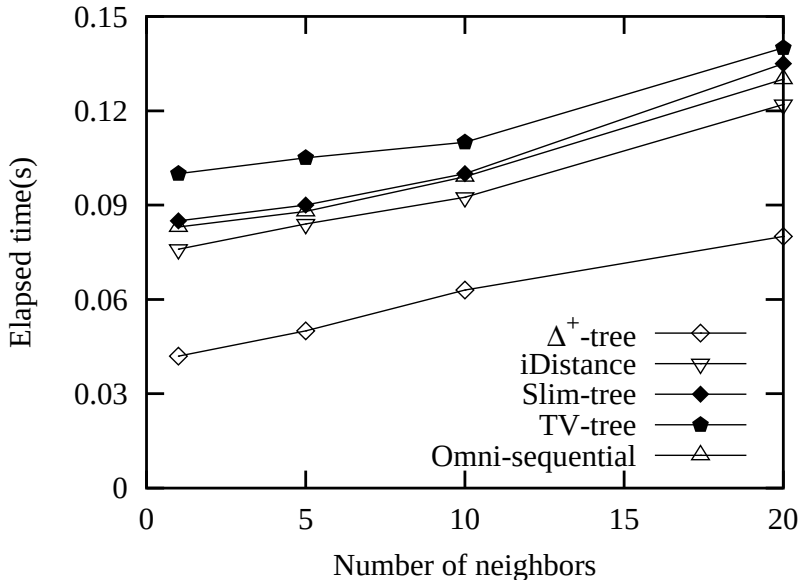
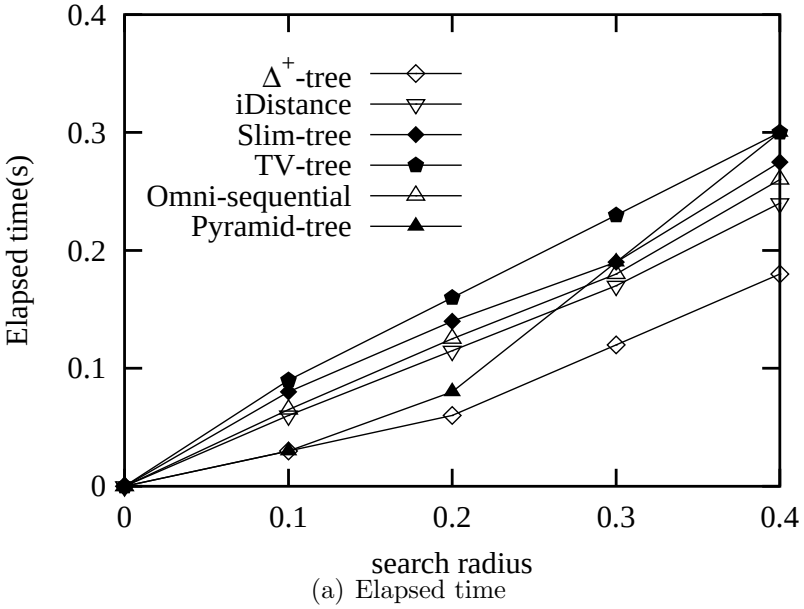


Figure 4.20: KNN search for 64D real dataset

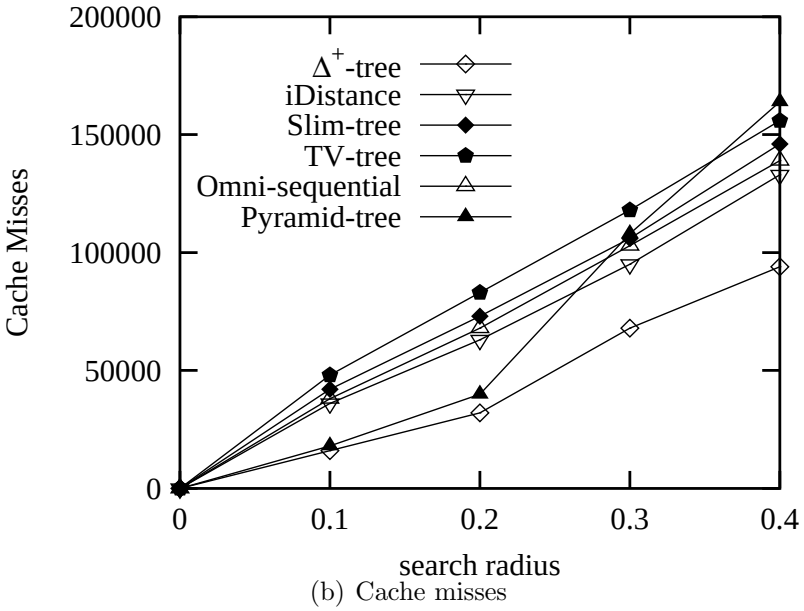
loss incurs larger search space and hence more computation and cache misses. The number of active dimensions for the TV-tree remains large, so its performance is affected by the scalability problem of the R-tree. In the context of main memory indexing, this translates into higher computational cost and number of cache misses.

As in section 4.4.3, we also conduct the range queries for these indexes. Figure 4.21 shows the range query results with varying the search radius from 0 to 0.4. All the indexes degrades as we increase search radius, because a large search range incurs more cache misses and distance computation. However, our method is still 30% better for relatively large search radius. We can expect that the performance of all the index structure will degrade to that of the Sequential Scan for very large search radius where all the data needs to be accessed.

The results for KNN and range queries in the motion capture dataset are presented in Figure 4.22. The performance differences are quite similar to that of color histogram dataset. Moveover, the gap between the Δ^+-tree and other indexes is widened compared with Figure 4.20. The reason is that the motion dataset is more

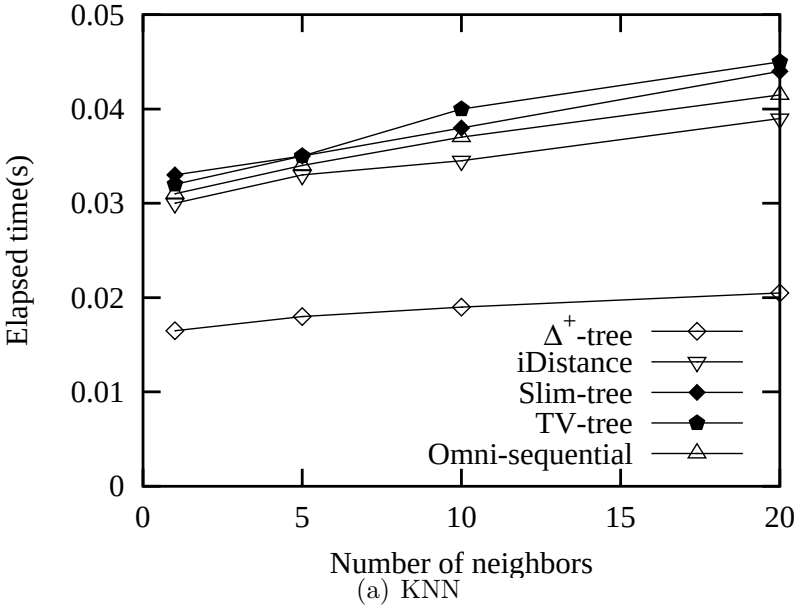


(a) Elapsed time

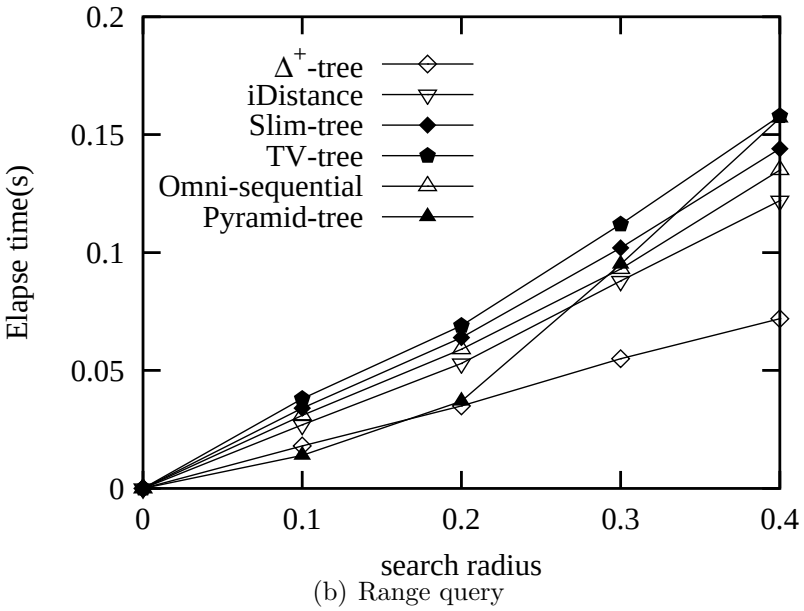


(b) Cache misses

Figure 4.21: Range query for 64D real dataset



(a) KNN



(b) Range query

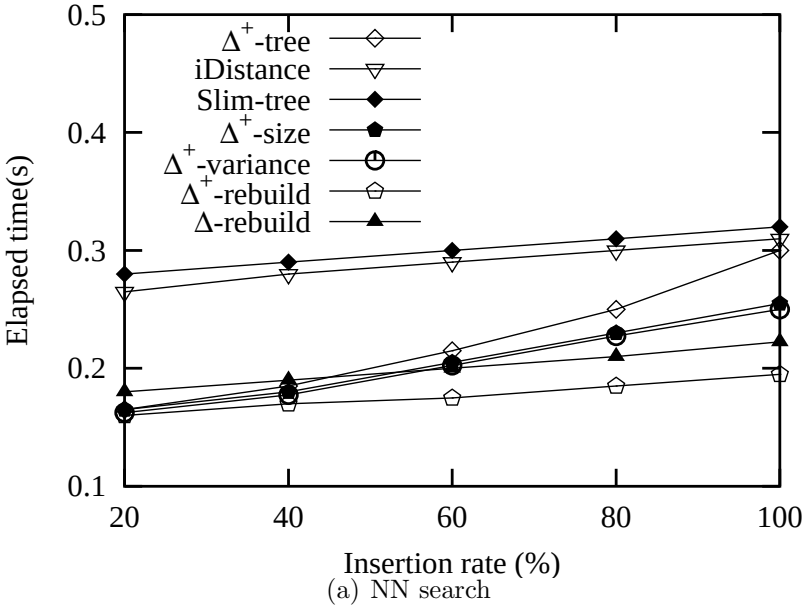
Figure 4.22: Performance for 79D real dataset

skewed, and hence the Δ^+-tree can benefit more from the PCA transformation. Checking the effect of cumulative variation of the dataset after applying PCA, we found the first 6 dimensions in the PCA space can capture 80% of the data information, furthermore the first 17 dimensions in the PCA space capture 95% of the information. Therefore, the Δ^+-tree can apply the reduced dimensionality to decrease the distance computational cost and cache misses without much information loss. We also note that the TV-tree performs better as we can use fewer active dimensions for this dataset.

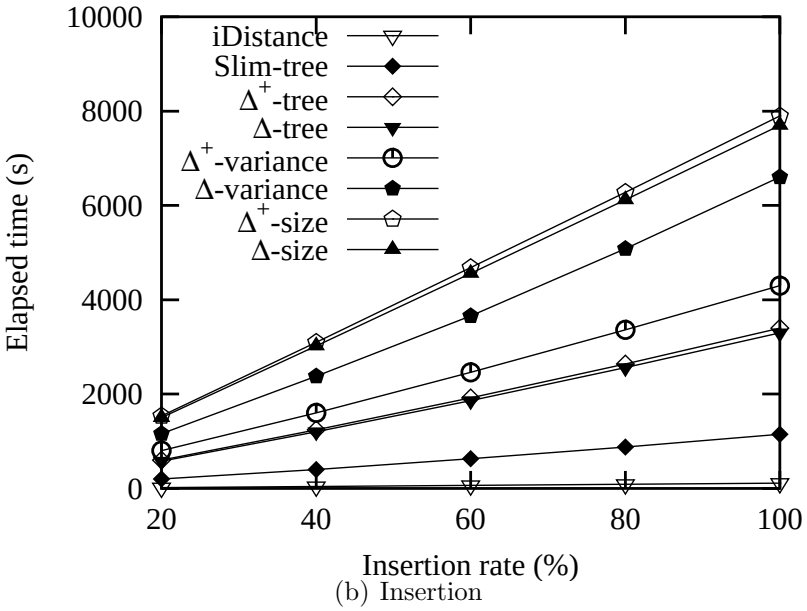
On effect of insertion

In the last experiment, we study the effect of insertion on the Δ^+-tree and Δ-tree. The delete operations yield similar performance and we omit it. For the Δ^+ tree and Δ-tree, we evaluated four versions, e.g. Δ^+-tree represents the version that is based on the proposed insert algorithm (without rebuilding); Δ^+-rebuild represents the version that always rebuilds the tree upon insertion, i.e. this is ideal and represents the optimal Δ^+-tree; Δ^+-size represents the version that rebuilds the tree after a certain size threshold is reached; and Δ^+-variance represents the version that rebuilds the tree after a certain variance threshold is reached. In our experiment, we set the threshold as 10% for the latter two schemes. We used the 64D clustered synthetic dataset for this experiment. We first build the tree structure with 1,000,000 data. Subsequently, we insert up to 100% more new data points. The results of insertion and NN search are shown in Figure 4.23. The performances of Omni-sequential and TV-tree are similar to iDistance and Slim-tree respectively, and we omit them for the clarity of figure.

Figure 4.23 (a) shows the average execution time of NN search on the new datasets after 20% of newly inserted data. First, we observe that the Δ^+-trees and



(a) NN search



(b) Insertion

Figure 4.23: Performance for the effect of insertion

Δ-trees outperform the other tree structures. We only show the performance of the Δ-rebuild for comparison, because the Δ-trees show the similar performance degeneration as the Δ^+-trees. This clearly demonstrate the effectiveness of the proposed schemes. Second, as expected, as more points are inserted, the performance of Δ^+-tree degenerates as the newly inserted data affect the precision of cluster *eigenmatrix*. However, the degradation of performance is marginal. More importantly, the accuracy is not affected - the Δ^+-tree may examine more nodes because of the relatively larger radius. Third, it is clear that the rebuilding algorithms can reduce the performance degradation. For Δ^+-size and Δ^+-variance, the degradation is around 30% even for 100% new insertions. So the Δ^+-tree remains very effective after new data points are inserted. We also observe that Δ^+-size is slightly better than Δ^+-variance. This is because it incurs more rebuilding. Once the new points of a global cluster is more than the threshold, it rebuilds the subtree regardless of the data distribution. The results show Δ^+-tree's robustness with respect to updates in the sense that it can take sufficiently large number of updates before we need to rebuild the tree structure.

Figure 4.23 (b) shows the execution time for insertions. The Δ^+-tree and Δ-tree incur more expensive insertion cost, because K-means clustering and PCA transformation are necessary for the reorganization of the tree. We did not show the rebuild version as the rebuild cost for each insertion is extremely high. However, our two proposed update mechanisms can significantly reduce the insertion cost, especially the *Distance variance threshold* method. The Δ^+-variance yields better performance than Δ-variance as the points are inserted into respective global cluster and the efficiency of *eigenmatrix* degenerates less than that of Δ-tree, hence reduce the rebuild frequency. The iDistance is the most efficient in terms of insertion, as we only need to insert a new record into the B^+-tree. However, the insertion cost

is small compared with NN query cost, because the query in high-dimensional space typically needs to access a great amount of nodes. Additionally, we can do the rebuilding offline while not interfering with the other queries. This makes the Δ^+-tree a promising candidate even for dynamic datasets.

4.5 Summary

In this chapter, we have addressed the problem of accessing high-dimensional data in main memory databases. We presented an efficient novel index method, called Δ-tree, for KNN search. The Δ-tree employs hierarchical clustering and multiple level of projections of points to allow nodes to better fit into the L2 cache. Thus, the search process can be accelerated by reducing computational cost and cache misses. We also proposed an extension, called Δ^+-tree, that further partitions a cluster into regions. We conducted extensive experiments to evaluate the Δ-tree and Δ^+-tree against several known techniques, and the results showed that our technique is superior in most cases.

CHAPTER 5

Exploitation of One-Pass Traversal for Memory Concurrency Control

5.1 Introduction

As random access memory gets cheaper and larger, it becomes increasingly possible to keep the whole database memory resident. To efficiently process the query-intensive transactions, recent research addressed the problem of optimizing L2 cache utilization and minimizing computation cost in the design of main memory index structures. While these memory indexes effectively improve the search performance, they were studied without much consideration of concurrency control (CC) which is essential for real-world main memory database applications. Recently some main memory CC algorithms have been proposed for single-dimensional indexes, such as physical versioning [78] and optimistic latch-free index traversal [22]. However these CC algorithms cannot be applied to multi-dimensional indexes, e.g. the

R-tree. Compared with the B^+-tree, the R-tree introduces more lock conflicts due to frequent tree ascents and multiple sub-tree traversal. Furthermore, we have seen an increasing demand for dynamic spatial indexes for supporting location-based services lately. In such applications, the spatial index is frequently updated as moving objects' locations change [61, 75, 83]. Hence, an efficient main memory CC algorithm for R-trees not only needs to provide high throughput of the concurrent query operations, but also must handle the updates effectively.

Traditional concurrency control algorithms on the R-tree are disk-based and cannot adequately handle a high degree of concurrent accesses that involve updates. *Lock coupling* that performs well in single dimensional indexes (e.g., B^+-tree) is not effective for spatial indexes like R-tree [25, 72]. This is because in the R-tree, there are more lock conflicts due to frequent tree ascents: an ascent is needed for a node split; and it is also needed to propagate an MBR modification (which is much more frequent than node split in multi-dimensional indexing). In addition, the search operations in R-tree, such as range query and KNN search, usually require traversal of multiple paths, and hence multiple locks need to be held simultaneously. The R-link tree [58], an R-tree version of the B-link tree [62], offers a high degree of concurrency when the query load is high and the demand for update is light. As we shall see in our experimental study, these schemes are not optimized for high update load.

In this chapter, we present a novel main memory concurrency control algorithm, called OPUS (One-Pass UpdateS), that enables the R-trees to provide efficient concurrency control in the midst of frequent updates. Note that, OPUS can employ different R-tree variants as underlying tree structures, such as the CR-tree [55] introduced in chapter 2 which is a cache-conscious version of the R-tree for main memory access. Since these R-tree variants employ similar algorithms, e.g. query

and insertion, we use the R-tree for the clarity of presentation of OPUS. The previous work shows MBR modification propagation and node splitting - both arising from insertion - to be the main causes for blocking other concurrent operations and decreasing the concurrency of the index. OPUS alleviates these two problems by using preparatory splitting and MBR modification, and hence reduces the query blocking overhead. OPUS traverses each R-tree node *once* during an update operation in the most of cases. Insertions are performed top-down using *preparatory operations* [43], based on early node splitting and MBR modification. By *preparing* insert operations, the proposed OPUS requires the R-tree to be traversed down once for an insertion, even if there are MBR modifications. Only when there is a split, we need to effect the split to the parent node. Deletions are conducted bottom-up through the support of an auxiliary structure on the object identifiers and backward pointers. We use a hash table to locate the leaf node that bounds the object to be deleted. The bottom-up traversal is necessary to tighten the MBRs of ancestor nodes. Location modifications of spatial objects are done by combining a bottom-up deletion and a localized insertion. OPUS has some advantages to achieve high throughput. First, the One-Pass traversal mechanism can not only reduce lock conflict, but also computational cost and memory access (cache misses). Second, since the modification is localized, the affected area of update is constrained locally as much as possible and incur less interference to other concurrent operations, thereby improving the throughput. Third, for delete and location modification operations, the secondary hash table can significantly reduce the work load on the tree and also reduce the lock conflicts.

We implement OPUS and run extensive sets of experiments to evaluate its performance in main memory environment. We compare our method with other schemes and the results show that OPUS outperforms the existing well-known

concurrency control algorithms [58, 59].

5.2 Basic Index Structure

The OPUS algorithm employs the R-link tree as the base structure, i.e. the right link structure. To facilitate the efficient OPUS algorithm, two new structures are introduced to speed up the update process, which is shown in Figure 5.1.

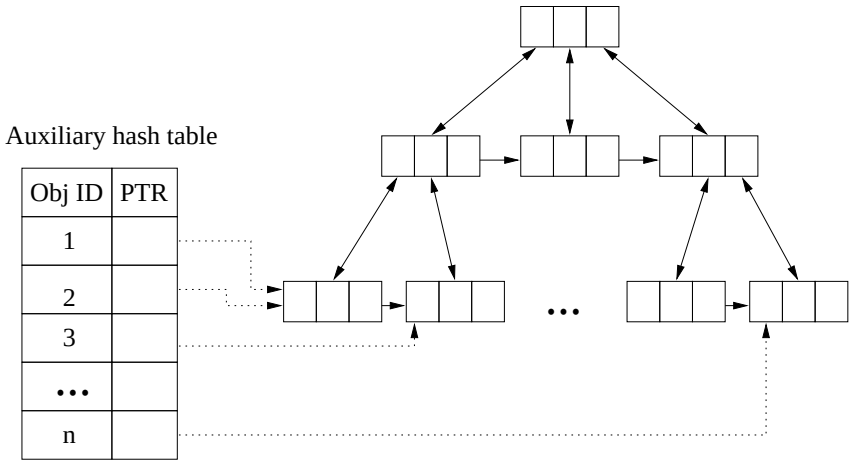


Figure 5.1: Optimized OPUS for update

First, we need to locate the object to be deleted or modified. To reduce the cost of tree traversal, we introduce a secondary hash table to locate the objects in the leaf nodes. In the hash table, we keep the information of the $objectID$, and a pointer to the leaf node that holds the object. The pointer can be the memory address of the leaf node. Because the $objectID$s are typically consecutive integers, we just adapt the Chained Bucket Hashing method to index the $objectID$. The size of hash table is $8 * N$ bytes, where N is the number of objects. Since OPUS accesses the leaf using the hash table, we need not access and lock the internal nodes in the tree; we thus avoid any conflict with other concurrent update operations. Additionally, the hash table is much faster than tree structures in terms of exact point search. When

an object is inserted, deleted or modified, the leaf node containing the object may be changed. Hence, the hash table must be updated to maintain the valid address of leaf node. The update cost of the hash table is not expensive, and it doesn't incur heavy concurrency overhead.

Second, since we propagate up the tree for object modification or MBR shrinkage, we assign a parent pointer to each node to identify the bounding ancestor. In the traditional R-link tree, deletion and modification must start from the root to identify the path to the target leaf node that holds the object.

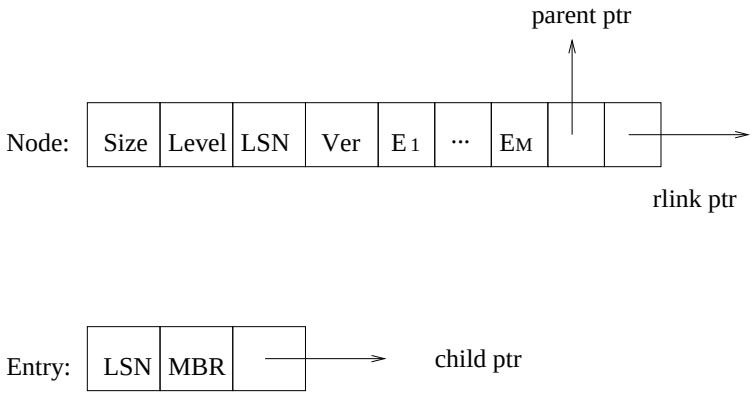


Figure 5.2: Node structure

An example of a node structure for OPUS is shown in Figure 5.2. *Size* is the number of entries in the node and *Level* represents the level of the node in the tree. Each node is assigned a Logical Sequence Number (LSN) and a right link pointer to a sibling node. The $LSNs$ monotonically increase over time and provide us with a mechanism for determining whether an accessed node is obsolete. If the node has to be split, the new right sibling is assigned the old node's LSN and the old node receives a new LSN. The concurrent operations can detect the split by comparing the expected LSN (taken from the entry in the parent node) with the actual LSN. If the latter is higher, it is proof that there has been a split and the operations then traverse right. Each operation needs to acquire a suitable lock before it can read

or update the content of the node. As shown in Figure 5.2, a node can have up to M entries and each entry is composed of an MBR, a child pointer and an LSN which records the LSN of its child node.

To implement the OPUS algorithm efficiently and correctly, we associate each node with an additional variable, which is a version of node Ver. Ver is monotonically and incrementally changed with each update operation that changes the content of the node, and its value provides us with a means for determining whether the node has been updated. When an update operation accesses the node with shared lock, it records the Ver of the node. If the operation needs to change the node content, it gets the exclusive lock and compares the new Ver with the old value to verify the consistency of the node content. If Ver is not changed, the operation can update MBR in the identified entry directly; otherwise, it needs to re-examine the node.

5.3 The OPUS Algorithm

With the presentation of index structure, we now introduce OPUS algorithms for various concurrent operations in this section.

5.3.1 Search Algorithm

The traditional concurrency control algorithms, such as lock-coupling, are too pessimistic as they lock more nodes when they traverse the trees. In the R-Link tree, the search operation only needs to lock the node before it examines the content of the node, and releases the lock after that. OPUS employs the similar search algorithm as that of the R-link tree, which is more efficient and locks at most one node at a time. Using the linking technique, we can guarantee the correctness of

the algorithm using fewer locks.

Note that in OLFIT [22], the main memory CC algorithm for B^+-tree, the authors proposed to read the node without lock. To ensure the consistency of node reads, the node read begins with reading the version number and ends with verifying if the node latch is free and the current version number is equal to the previous one. If these two conditions are true, the read is consistent. Otherwise, the node read is retried until the conditions become true. However, this mechanism is not suitable for multi-dimensional index structures because of high retry cost overhead. The search operation on a B^+-tree node is very simple, typically a binary search. A typical search operation in R-trees, either range query or KNN search, generally has to descend multiple paths within the tree, since the MBR keys can overlap. The underlying data structure to support this is a stack, which is used to remember which nodes still have to be visited. Therefore, the operation cost on a R-tree node is much more expensive. First, the R-tree node operation incurs more computational cost, e.g. coordinate comparison for range query and distance computation for KNN query. Second, we may push some node addresses into the stack after checking the node. If we retry the node read, we have to pop the obsolete addresses from the stack. Furthermore, since we want the OPUS to support the concurrency control in the midst of frequent updates, the possibility of node re-read is high. Therefore, we still use the shared lock for query operations in OPUS. The details of the search algorithm are shown in Figure 5.3

The process starts by initially pushing the root and the corresponding LSN into the stack. A node p that has not yet been examined is popped off the stack. If the LSN is higher than the one on the stack, we know that this node has been split in the meantime. Therefore, we traverse the rightlink chain until the node whose LSN is equal to the LSN popped from the stack, and push these nodes

Search(r, root)

```
1. push(stack, root, root.LSN);
2. While not empty(stack)
3.     pop(stack, p, lsn);
4.     r_lock(p);
5.     if ( lsn < p.LSN )
6.         traverse the rightlink chain until the node with LSN=lsn;
7.         for each node p' along the chain
8.             r_lock(p');
9.             push(stack, p', p'.LSN);
10.            unlock(p');
11.    if(p is leaf)
12.        for all entries e of p intersecting r
13.            insert e into result set;
14.    else
15.        for all entries e of p intersecting r
16.            push(stack, e, e.LSN);
17.    unlock(p);
```

Figure 5.3: The search algorithm of OPUS

into the stack. If p is a leaf node, we can add the qualifying objects into the result list. Otherwise, all entries in the node that qualify for the search predicate are in turn pushed in the stack. This operation is repeated until the stack is empty. This mechanism can make sure the split node will never be misses without any lock-coupling for search operations. We can know whether the node has been split by the comparison of he the LSN values.

5.3.2 Insert Algorithm

In this section, we present how OPUS handles insertions. Traditionally, an insert operation is carried out in four phases:

1. The insert operation traverses the tree from root to leaf, following the path with the lowest insert penalty.

2. If the leaf overflows after insertion, it has to be split, and that may cause recursive splitting of ancestor nodes.

3. If the insertion changes the leaf's MBR, the MBR modification needs to be propagated to the ancestor nodes recursively, until a node whose MBR bounds the new object is encountered.

4. The new object is inserted into the leaf.

In the B^+-tree, we only need to propagate the node split to the upper levels. In contrast, an insertion in the R-tree may propagate to the ancestor node for two reasons: node split and MBR modification. The second operation, MBR modification, is much more frequent than node split [86]. As with the algorithm proposed in [58, 59], we must employ lock coupling during the tree propagation to ensure correctness of tree structure. Because we need to hold the exclusive lock during

the propagation, both of these requirements become the main block factors for the concurrent operations in R-tree.

To deal with frequent node splitting and MBR modifications, OPUS employs preparatory node split and MBR modification. It tries to modify the structure when traversing down the tree, reducing the lock blocking overhead, and hence improves the throughput of concurrency control algorithm. We discuss OPUS in more detail as follows.

Node Split:

OPUS performs preparatory split on the node which is full along the traversal path down the tree. When a full node is encountered, OPUS will split the node and install the new entry to the parent node. To avoid overtaking of conflicting updates, OPUS employs lock coupling and holds an exclusive lock of the child until it obtains an exclusive lock on the parent. While it is possible for us to preparatorily modify the MBR for insert operations, we still need to propagate the MBR shrinkage bottom up due to delete operations. If we do not use lock coupling, the tree structure would be incorrect. As an example, consider the following special case. When a node split causes an update on the entries in the parent node, another delete operation causing MBR shrinkage can overtake the operation and modify the entries in the parent node. The update in the parent node due to the node split is performed regardless of the previous shrinkage of the MBR. Also, the MBR in the node would not reflect the bounding rectangle of the child node.

In the R-link tree, the split may propagate from the leaf level to the root. In OPUS, by exploiting preparatory MBR modifications, most of the insertions require only top-down traversal, because MBT modification propagations are more frequent. Split installation to the parent only propagates one level most of the

time. Only in an extreme case, as shown in Figure 5.4, would it be necessary to propagate to upper ancestor nodes.

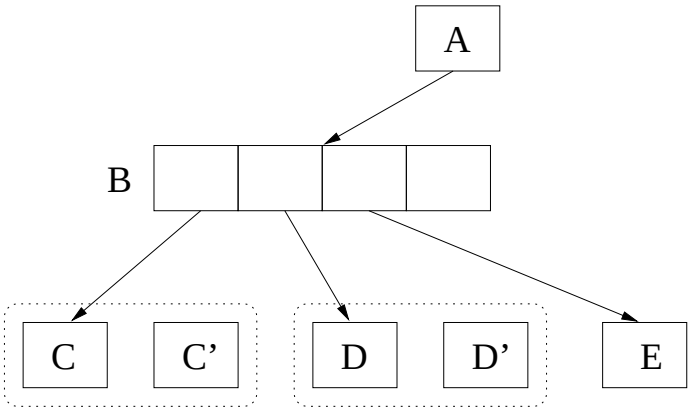


Figure 5.4: The extreme case of further propagation

For simplicity, a three level branch of R-tree is illustrated for our analysis: node A is the parent of node B and node B is the parent of nodes C, D and E. Assume that each node can have up to 4 entries. There are two concurrent processes traversing simultaneously down the tree from node B to C and D respectively. Both C and D need to be split. So we must install the new entries of C' and D' to the node B. Thus the node B is overflowed and need to be split and a new entry need to be inserted to node A. We want to estimate the possibility of such a case. Assume M is the maximum number of entries in a node, and m, which typically equals $\lceil M/2 \rceil$, is the minimum number of the entries in a node. For a split process to propagate to node A, it must satisfy the conditions: (1) the number of entries in node B is $M-1$, (2) the number of entries in C is M, and (3) the number of entries in D is M. Each of these three conditions is independent and has the probability $\frac{1}{M-m+1}$, hence the possibility of all these three conditions being satisfied is $(\frac{1}{M-m+1})^3$. As shown in [55], the optimal node size for the R-tree for memory processing is around 512 bytes. For a dataset with 2-dimensional rectangles, M would be 24 and m would be equal to 12. This will result in a possibility equal to

$4.55 * 10^{-4}$. More generally, considering the lower level nodes with the same parent node, probability of both processes accessing the same parent is $\frac{1}{m}^d$, which leads to much lower probability of the case; where d is the depth of parent node in the tree, and the depth of root node is considered as zero. Even if there are more than two concurrent processes, the possibility of propagating to more than one level is still low.

On the other hand, the installation to the parent does not interfere with the further downward tree traversal. The two operations can run in parallel, and this can speed up the overall insert operation and hence reduce the response time.

MBR Modification (for Insertion):

In OPUS, MBR modification is performed while traversing down the tree. For nodes where the MBR contains the point to be inserted, nothing needs to be done; otherwise, if there is any enlargement needed, MBR is enlarged before the child node is traversed. In the original CGiST or R-link tree, they employ lock coupling bottom up to avoid overtaking conflicting update. The OPUS algorithm starts with the examination at the root of the tree, and identifies the entry e which bounds the insert object. It then updates the corresponding MBR of e and continues with the insertion of the object in the subtree rooted at e. Since OPUS does not employ lock coupling for top-down traversal, the particular child node may be split before the insertion reaches it. In this case, the insertion must resume at the parent node. The possibility of resumption is very small, because two insertions must access the same child node and the node must be full. The possibility of accessing the same node is at most $\frac{1}{2}$, i.e. the minimal fanout of the root is 2, and the possibility that a node is full is $\frac{1}{M-m+1}$. Therefore, even for two simultaneous insertions, the resumption possibility is $\frac{1}{M-m+1} * \frac{1}{2}$, i.e. 3.8%. Typically the MBR in the root

node is large since it has to cover all the existing objects. Most of the time, the root MBR can bound the new points, and hence the possibility is much smaller in reality. Also considering the whole tree, the resumption probability in the lower levels is reduced to $\frac{1}{m}^d$, and the overall probability is much less.

In the extreme case, in which the parent node is also split, the resumption needs to recursively propagate up to the ancestors. At least three insertions come in together, which may cause the split of the parent. We again refer to Figure 5.4 for illustration. For example, we have three simultaneous insertions $P1$, $P2$ and $P3$. After $P1$ preparatorily enlarges one entry in node B, it tries to access node C. At that time B is not full; otherwise, $P1$ needs to split B. When $P1$ reaches C, it finds that C has been split after it changes the MBR in B by comparing the value of LSN. So then it needs to resume the MBR modification in node B. If B needs to be split, at least two of its children will be split. Suppose $P2$ causes the split of C, and $P3$ causes the split of D. Now let us see the possibility of this case. In this situation, the number of entries in B is $M-1$ and the numbers in C and D are both equal to M. Each has the possibility $\frac{1}{M-m+1}$ as we showed previously. The possibility that $P1$ and $P2$ access the node C is $\frac{1}{M-1}$. The possibility that $P3$ accesses D is $\frac{M-2}{M-1}$, noting here C and D are different nodes. Thus the total possibility is $\frac{1}{M-1} * (\frac{1}{M-m+1})^3 * \frac{M-2}{M-1}$, which is only $1.9 * 10^{-5}$. Although the resumption propagation incurs some cost overhead, its practical value is almost nothing.

In contrast to the algorithms of [58, 59], OPUS combines MBR modification with tree traversal and avoids locking multiple levels simultaneously. Another side benefit is the lower computation cost and fewer cache misses. In traditional algorithms, when we traverse down the tree, we need to examine the whole node to identify the bounding rectangle. And if we need to propagate the MBR modification to the ancestor, we also need to access the parent node again to locate the

corresponding entry. The OPUS algorithm, however, needs to examine the node only once to make the modification.

The Insert Algorithm

Here we give the details of the insert operation. We employ the preparatory operation in our OPUS algorithm to improve the concurrency. The details of the insert algorithm are shown in Figures 5.5-5.7.

Typical concurrency control algorithms use two types of locks: the shared lock and the exclusive lock, and only two shared locks are compatible. Query operations use the shared lock and insert operations use the exclusive lock. However, insertions do use the shared lock during tree traversal. As we employ preparatory operations for insertions, we need to change the content of the node, which is different from the insert algorithm of the R-link tree. A naive way is to acquire an exclusive lock on nodes of the traversing path. This approach is too pessimistic because the insertion may not change the content of the node while the exclusive lock always blocks concurrent operations. Our optimization provides for the insertion to acquire a shared lock first, and then converts it to an exclusive lock in the event of an MBR modification or a node split. But this approach has a weakness when used on the R-link tree structure directly. For example, two insertions, A and B, access the same node p when they traverse down the tree. They examine node p, identify the suitable entry and try to acquire a lock on p in exclusive mode. Suppose A gets the exclusive lock first and updates the key MBR in p (A may split node p or change an MBR). When it is finally B's turn to change the entry in p, the previous identification would be invalid because A has changed the content of node p. But B does not know whether the content has been changed or not. To guarantee the correctness of the algorithm, B has to re-examine the node p holding

Insert(r, root)

```
1. p = root;
2. old_lsn = root.LSN;
3. repeat
4.     r_lock(p);
5.     if( old_lsn < p.LSN )
6.         unlock(p);
7.         pop(stack, p, old_lsn);
8.     else if ( p is leaf )
9.         unlock(p) and x_lock(p);
10.        if (p is safe)
11.            insert_leaf(r, p);
12.            unlock(p);
12.        else
13.            Split(p, p', r, p.LSN);
14.            Install_parent(p, p');
15.        InsertLeaf(leaf.parent, leaf, newleaf)
16.    else /* p is an internal node */
17.        ver = p.Ver;
18.        branch = pick_branch(p, r);
19.        if( branch.mbr needs to be extended )
20.            unlock(p) and x_lock(p);
21.            if ( ver != p.Ver )
22.                branch = pick_branch(p, r);
23.            extend branch.MBR;
24.            p.Ver++;
25.        if ( p is full )
26.            Split(p, p', NULL, p.LSN);
27.            Install_parent(p, p');
28.        push(stack, p, old_lsn);
29.        if( p is not split )
30.            unlock(p)
31.        p = branch.child
32.        old_lsn = branch.LSN
33. until(the new object is inserted to the leaf)
```

Figure 5.5: The insert algorithm of OPUS

the exclusive lock, and this increases the block time and cost overhead. The Ver which we introduced in Section 5.2 addresses this problem as it verifies whether the node is in a consistent state. When an operation A accesses the node with shared lock, A records the Ver of the node. If A needs to change the node content, it gets the exclusive lock and compares the new Ver with the old value. If Ver is not changed, A can update MBR in the identified entry directly; otherwise, A needs to re-scan the node to get the suitable entry.

As mentioned earlier, we combine preparatory operations, including MBR modification and node split, with tree traversal. In Insert(), we start from the root, and repeat the loop until we insert the new object r into the leaf. Inside the loop, we first check whether node p is split after we examine the parent, i.e. $old\_lsn < LSN(p)$. If so, the split nodes will overtake the MBR modification in the parent and the tree becomes incorrect. In this case, we must resume the insertion at its parent. If p is an internal node, we record the node version, ver, and examine the node to pick the most geometrically optimal entry e for r. If the MBR key in e needs to be modified, we release the read lock and acquire an exclusive lock on p. If the node content has been changed, i.e. ver is different, we need to examine the node again and may pick a different entry. After that, we can change the MBR. Then if the node p is full, we conduct the preparatory split and install the split nodes p and p' to the parent. And then we push the node into the stack, and release the lock on p. Note if p has been split, we need not unlock(p), because we would have called unlock() in Install_parent(). Once we reach the leaf level, we acquire an exclusive lock and insert the object r into the leaf. If the leaf node is full, we split the leaf and install the new nodes to the parent.

The Split() function is very straightforward. If the lock is not in exclusive mode, we release the r-lock and acquire an x-lock on the node, compute the split

Split(p, p', r, lsn)

```
1. if ( lock type of p is read-lock )
2.     unlock(p) and x_lock(p);
3.     if ( lsn < p.LSN )
4.         p = suitable node for r in rightlink from p;
5.         change lock to new p
6.         if( p is safe )
7.             insert r into p and return;
8. compute the split information;
9. pick_split(p, p', r);
10. p'.LSN = p.LSN;
11. p.LSN++
12. p.Ver++
13. create a new entry e' for p'
```

Figure 5.6: The split algorithm

information, split into two nodes and assign the new LSNs to the nodes. If the node is split when we change the lock mode of the node, we need to select an optimal node in the rightlink and insert the new entry.

In Install_parent(), we first check whether the stack is empty. If it is empty, we create a new root, and install the two entries for p and p' into the root. Otherwise, we pop off a node from the stack which is the parent of the node when descending. We may need to move right if the parent node has been split. In this case, LSN is not necessary to recognize the correct parent, because an entry in a node can be uniquely identified by the child pointer and the address of the child node is unchanged during the operation. After we get the lock of parent, we release the lock on node p. We employ lock coupling during installation to the parent. Then we update the entry of parent node for p, and insert the new entry e'(for p'). As we have mentioned earlier, the parent node needs to be split only in an extreme case. Therefore, we only need to propagate the split to ancestor by one level typically.

Install_parent(p, p')

```
1. if ( stack is empty )
2.     create a new root node;
3.     unlock(p);
4.     install the entry for p and p' into root;
5. else
6.     pop(stack, parent, lsn);
7.     x_lock(parent);
8.     if( p is not the child of parent )
9.         parent = node holding p in rightlink from p;
10.        change lock to new parent;
11.    unlock(p)
12.    update the entry in parent for p
13.    if ( parent is safe )
14.        insert newentry e' into parent
15.    else
16.        Split(parent, parent', e', parent.LSN);
17.        Install_parent(parent, parent')
```

Figure 5.7: The install_parent algorithm

5.3.3 Delete Algorithm

Typically a delete operation requires the object ID of the object, as well as the location. The preparatory operation is not suitable for delete operations in the R-tree due to the fact that the MBRs cannot be strictly ordered. To locate the delete object, we need to traverse multiple paths to get the object using search algorithm. We do not know which node we have accessed is the ancestor of the object before reaching the target leaf node. Additionally, even if we know the ancestor, we still cannot decide whether we can shrink the MBR of any particular entry unless we know the bounding rectangle of the particular child node.

As shown in the R-link tree, we have to traverse the tree to find the leaf holding the object we want to delete. This procedure is the same as an exact point match query. When we traverse the tree downward, we get the path from the root to

the leaf. After the object has been removed from the leaf, we propagate the MBR shrinkage up the tree. The hash table cannot be employed on the R-link tree directly. This is because even if it can access the leaf node via the hash table, it cannot propagate the MBR shrinkage up without knowing the path to the root. The locking behavior is the same as for the search and insert algorithms and therefore key deletion is also deadlock-free [58].

The deletion of OPUS is quite similar to that of the R-link tree except of two differences, i.e. the hash table and the backward pointers are used to support efficient delete operations. First, via the hash table, OPUS can access the leaf very quickly and does not introduce much concurrency control overhead. In hashing structure, we only need to lock one bucket to remove the offset of leaf and all the buckets scatter about the table, therefore it introduces much less interference between each other. Note that, the upper level nodes of tree are typically the bottleneck of concurrency operations. Second, once OPUS deletes the target object, it needs to propagate the MBR modification up. The propagation is in the same manner as that of the R-link tree. The distinction is that the R-link tree gets the path from root to leaf when it traverses down the tree, while the OPUS propagates the modification up using the backward pointer. Maintaining these backward links from child to parent is necessary to make sure the correctness of the algorithm. These pointers are essential for the delete scheme as only with them is it possible to avoid a downward tree traversal. To update the backward pointer, the split may involve the updates of half child nodes. Although the cost overhead is high, this phenomenon occurs very rarely. Because the leaf nodes do not have children, updates of backward pointers only occur in case of internal node split, i.e. both the certain internal node and one of its child node must be full. To estimate the possibility of such a case, we assume $M = 24$ and $m = 12$ as in Section 5.3.2.

For the first level internal node from leaf level, the possibility of split is $(\frac{1}{M-m+1})^2$, which is only $0.6 * 10^{-2}$. Furthermore, the possibility in the upper levels decreases exponentially. Hence it is worthwhile to apply backward pointer mechanism for deletion.

5.3.4 Modification Algorithm

With rapid advances in wireless communications and positioning techniques, some applications need to track the positions of moving objects. In [61], the authors show that the indexing technique based on the R-tree can also perform well on the indexing of moving objects.

In the traditional databases, the positions of objects do not change with time, and the typical operations are query, insertion and deletion. To index moving objects using the R-tree, location modification is an essential operation. The objects in the R-tree will change positions within a given period. To update a moving object, traditional algorithms will do as follows: locate the object, delete the object and insert the object with a new position. Only if the new position is bounded by the leaf MBR, the update operation will be required to change the position. This method is very expensive, because the reinsertion always starts from the root of the tree, entails many locks on the downward path to the leaf, and hence reduces the degree of concurrency. The traditional R-link tree and CGiST can be employed straightforwardly to support the concurrency control of location modification of objects.

It is common that the new position of a moving object is near the old one, and in such case we should try to modify the position of the object bottom up. The OPUS algorithm works as follows: locate the leaf containing the target object via the hash table, delete the object, find the lowest level ancestor node which

bounds the new position, and reinsert the new object into the subtree rooted in that ancestor (using the preparatory operations). Figure 5.8 shows the detail of a modification operation.

Modify(object, new_object)

```
1. locate(hashtable, leaf, object);
2. x_lock(leaf);
3. mbr = MBRArea(leaf);
4. if ( new_object is bounded by mbr )
5.      change the position of the object;
6.      unlock(leaf);
7. else
8.      remove(object, leaf);
9.      propagate the MBR shrinkage if any;
10.     while ( new_object is not bounded by mbr )
11.          parent=p→parent;
12.          r_lock(parent) and unlock(p);
13.          p = parent;
14.          mbr = MBRArea(p);
15.     Insert(new_object, p);
16.     adjust hash table
```

Figure 5.8: The update algorithm of OPUS

In Modify(), we first locate the leaf node of the object using the hash table. If the new position of the object is still bounded by the leaf MBR, we just update the position. Otherwise, we delete the old object and propagate the MBR shrinkage [1]. Then we traverse the tree from the bottom up until we reach a node p whose node MBR bounds the new position. Finally, we insert the new object into the subtree rooted in p. Because the moving objects typically move locally, the bottom up mechanism constrains the affected area locally and does not interfere much with other concurrent operations. Additionally, the bottom up algorithm reduces the

[1]The MBR shrinkage propagation is same as what we do in the delete operation. Additionally, the propagation can combine with the bottom up traversal for bounding ancestor. For simplicity, we separate two operations in Figure 5.8.

number of locks, because the path is shorter typically. Hence OPUS can improve the throughput of location modification operations significantly.

5.3.5 Phantom Protection and Recovery

Two common requirements for concurrent access in multi-dimensional database systems are phantom protection and recovery. In R-trees, concurrent accesses to data introduces the problem of protecting ranges specified in the retrieval from phantom insertion and deletion. For example, an insertion into the key range of a current search can succeed even though the search locked all leaves it accessed, and the key will be visible to a re-search. We proposed an enhanced method to handle phantom.

A naive way to avoid the phantom problem is for scans to keep every node they traversed locked until the end of the transaction, including internal nodes. This method has two disadvantages. First, setting locks on internal nodes reduces concurrency significantly, because it will block all the concurrent insert operations. Second, it may introduce deadlocks. A preceded insertion cannot propagate the MBR modification up the tree. [58, 35] proposed a simplified form of predicate locks to avoid the phantom problem. A search checks the still-active insertions and suspend itself if its query rectangle collides with any of the uncommitted new keys. An update operation would in turn check the active searches and also suspend itself on a collision with a query rectangle. Only one operation commits, the conflicted operation can continue. This method is more efficient compared the former naive method and no deadlock will be incurred. we simply check a key value against a query rectangle and a lock request is only rejected if there is a guaranteed collision with another active lock. The major disadvantage of this method is that the throughput will be reduced because of block of conflicted operations. We

propose an enhanced method to avoid phantom. The solution assigns a timestamp to each active operation, and records it. Once the search operation commits, it checks whether there is any collisions with other concurrent update operations in the meantime. If there is any phantom, we just adjust the answer set accordingly. While for insert operation, we do not need to check the collision with concurrent searches, because all the affected search operations will check the phantom. Key values, query rectangles and timestamps are organized into an separate data structure to speed up checking for lock conflicts.

To ensure an index functions correctly, the tree should reflect updates of committed transactions and none of those of uncommitted transactions. Moreover, the tree structure must be brought back into a consistent state after a system crash or failure of update. Since the concurrency control mechanism of OPUS is based on the R-link tree structure, the main idea of the recovery method is drawn from [58, 71]. We also split an update operation into its contents-changing and its structure-modifying part. A content change is an update of a key on a leaf; a structure modification can be a node split or deletion, an update of an index entry on an internal node. By separating these two parts, the semantic effects of an index operation are attributed to the initiating transaction whereas the structure modification is handled independently of any transaction. Additionally, write-ahead logging (WAL) is used for recovery purposes. When changing the contents of a leaf, we log this operation within the context of the executing transaction, obeying the WAL protocol. This ensures that these leaf changes can always be undone if the transaction aborts and redone if it commits. However, structural modifications are treated as separate recoverable units *atomic actions*, which are logged and recovered separately from the surrounding transaction. The obvious advantage is that the effects of structure modifications can be made visible to other transactions

although the initiating transaction has not committed, without creating an abort dependency between the transactions. Interested reader can refer to [71] for more details.

5.4 Performance Study of OPUS

In this section, we present an experimental study to evaluate the performance of OPUS in main memory environment. We implemented the OPUS, and compared it with CGiST and the R-link tree. All the indexes are loaded into memory before the concurrent operations are conducted. The CGiST[2] evolved from the R-link tree, so the concurrency control of these two mechanisms are quite similar. The main difference is that the CGiST must lock the parent node before it splits the node. Although CGiST reduces the capacity of tree node, the performance is not improved as shown in our experimental study.

All the experiments were conducted on SUN Fire 4800 machine with 750MHz CPU and 16 GB RAM. Concurrent operations on the index structures are simulated using multiple threads with different concurrency levels, which control the number of threads that can run simultaneously. For each experiment, we present the result with different workloads and variant threads. The proportion of search and update operations was used to generate different degrees of node contention. The main metrics used in the simulation for evaluating concurrency control system performance are throughput and response time. Throughput is the rate at which operations can be serviced by a CC system. It can be computed by dividing the number of completed operations by total simulation time. Response time is the interval between issuing an operation an getting the system response. It can

[2]We employ the concurrency control algorithm of CGiST over the R-tree. For the clarity of presentation, we simply call it CGiST.

be computed by dividing the summation of each operation's response time by the number of completed operations. The larger the throughput, the better the performance; in contrast, the shorter the response time, the better the performance. To understand the effectiveness of the various techniques of OPUS, we study them in isolation.

5.4.1 Performance on the CR-tree

The CR-tree [55] is a cache-conscious version of the R-tree for main memory access. It compresses MBR keys to pack more entries in a node, and hence reduces the cache misses of the tree operation. The algorithms of CR-tree are similar to the ones used in other R-tree variants. When we consider the concurrency control of multi-dimensional index trees in main memory, people may think that the CR-tree is the best candidate as the underlying tree structure for OPUS, since it is proposed for memory processing. However, although the CR-tree performs search faster than the ordinary R-tree, it yields worse update performance. Therefore, we test the performance of OPUS on different underlying tree structures in the first experiment with mixed search and insert workloads.

The underlying trees, i.e. CR-tree and R-tree which are represented by OPUS-CR and OPUS-R respectively, are initialized with 1,000,000 2-dimensional points that are generated randomly. The coordinates of points are selected from [0,1]. There are different kinds of search operations, such as point query, range query, nearest neighbor search, etc. We focus on the performance on range query and the default query window size is 0.001% of the space. As shown in [55], the optimal node size for the R-tree and CR-tree is around 512 bytes to optimize the cache behavior in main memory environment, and hence the node size used in the experiment is also set as 512 bytes. Figure 5.9 shows the throughput and response time of a

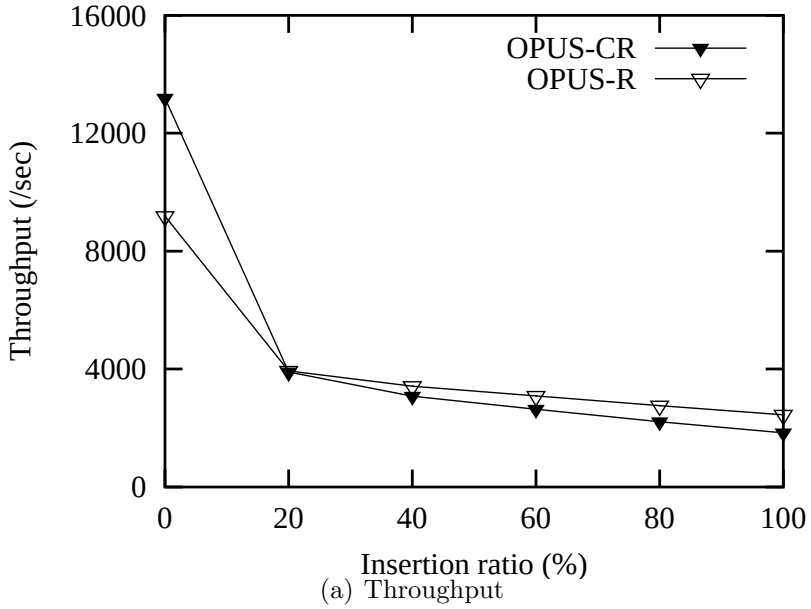
mixed workload with varying ratios of insertion and query. The workload consists of a sequence of 10,000 operations, each of which is randomly determined to be an insert or query operation based on the given ratio. 32 threads run concurrently for these operations.

When the insertion ratio is equal to 0, we have a read-only scenario with 32 threads running concurrently and the CR-tree performs better than the R-tree. Because the search operations do not block other concurrent query, thus the throughout of concurrent operations depends on the efficiency of the query. The CR-tree outperforms the R-tree as it optimizes the cache behavior of R-tree for use in the main memory environment by MBR compression.

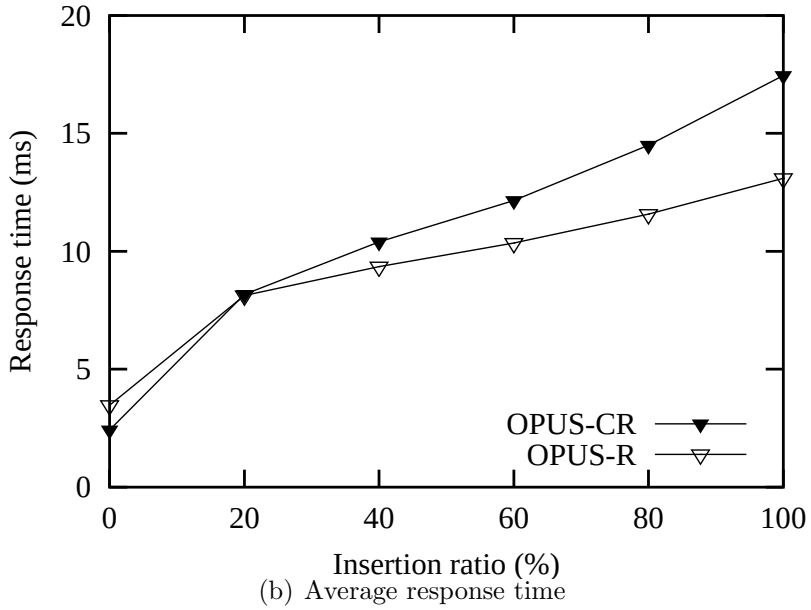
However, we observe that the throughput decreases when the ratio of insertion increases for all the methods as more blocks result from more insertions. The R-tree provides the better performance when the insertion ratio exceeds 20%. Although the CR-tree reduces cache misses for queries, it introduces cost overhead for insertions. For the mixed workload of insertion and query, the cost of insertion becomes dominant when the insertion ratio increases. Not only the cost of insertion is more expensive than query, but also it evokes high lock overhead and may block other concurrent operations. Since we presents efficient concurrency control in the midst of frequent updates, we employ the R-tree as the underlying structure for OPUS in the following experiments because it performs better for concurrent updates.

5.4.2 Effect of Insertion

In this set of experiments, we study the various concurrency control algorithms to see how they perform on insertions. The underlying R-tree is initialized with 1,000,000 2-dimensional points that are generated randomly. We use uniform data but different query (insertion) distribution, and a mixture of operations to evaluate



(a) Throughput



(b) Average response time

Figure 5.9: Performance with different underlying structures

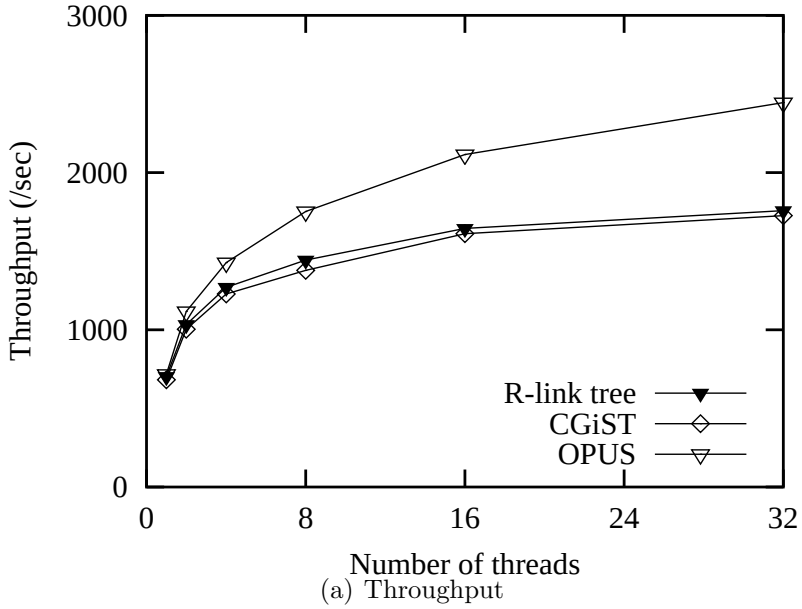
the efficiency of the structure. The skewed operation distributions are used to evaluate the schemes under different contention levels. We also investigate the throughput and the response time of mixed search and insert operations on the different indexes. The OPUS uses more space compared other two schemes, as it uses a secondary hash table to store the object ID. The space overhead is around 8M bytes in this experiment.

We ran each concurrency control algorithm with the following workloads: 100% insertion and a mixture of insertion and query with varying ratio. Each workload had 10,000 operations to access the indexes. We did not present the result for the pure search operation here, because all three concurrency control mechanisms yield similar performance. The reason is obvious - regardless of whether a lock is acquired, the search operation of all three methods does not block other concurrent operations due to lock compatibility. The difference in the tree fanout does not affect the performance much for our dataset.

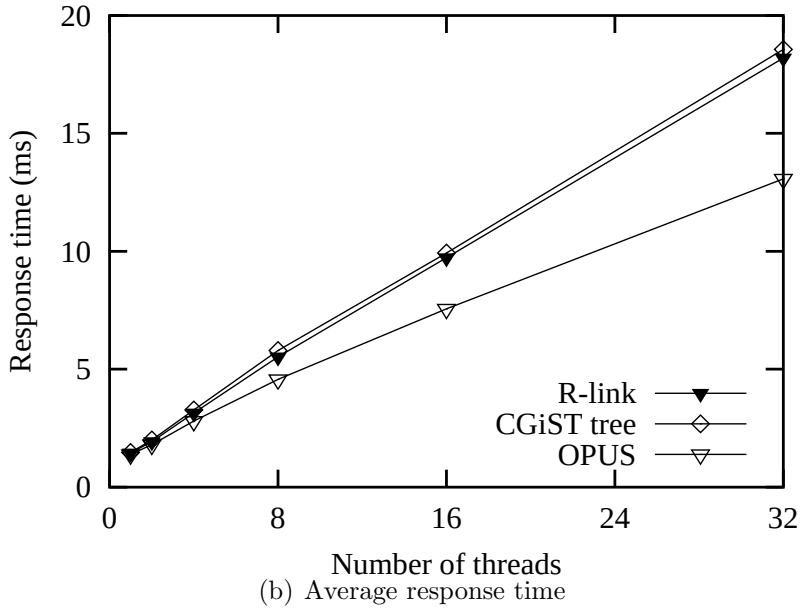
Performance of Insertion-only Workload

In this experiment, we evaluate the performance of the concurrent workload with pure insertions. We measure the throughput and response time as we vary the number of threads that perform insertions. The results are shown in Figure 5.10. The graphs show respectively the aggregated throughput (in operations per second) and the average response time for each operation, against the number of threads performing insertions.

When there is one thread, the three algorithms have similar performance, although OPUS is slightly better than the other two methods. This is because OPUS modifies the MBR as it traverses down the tree, thereby reducing the cost to propagate the MBR modification in the traditional way. The tree propagation not only



(a) Throughput



(b) Average response time

Figure 5.10: Performance of insertion

introduce extra memory access, i.e. more cache misses, but also more computational cost, because we have to reexamine the MBRs in ancestor nodes. Although OPUS needs to update the hash table, the cost overhead is not expensive, we only need to update one hash bucket most of time. We note that when the number of threads increases, all three methods' throughput also increases. Not surprisingly, the R-link tree is better than CGiST although the concurrency control algorithm of CGiST has evolved from the R-link tree. CGiST was proposed for reducing the additional information, i.e. LSN in the entry of R-link tree. However, it reduces the efficiency of the concurrency to make the algorithm work properly. The NSN of CGiST is taken from a tree-global and monotonically increasing counter variable. For node splitting, we must acquire the exclusive lock on its parent node first, split the node, assign NSN and finally increase the global counter to make the algorithm work properly. As a result, one must keep multiple locks on two levels for quite a long time, and this may block other concurrent operations. Meanwhile, the R-link tree only requests the lock in the parent node after the split of the child node.

The OPUS scheme scales better than the other two mechanisms in terms of number of threads. First, this is due to lower locking overheads and fewer blocking effects. OPUS is able to reduce the two blocking effects caused by MBR modification propagation and node splitting on two counts. OPUS traverses down the tree with preparatory operations, either MBR modification or node splitting. The OPUS avoids multiple locks on an insertion path in case of MBR modification. In certain situations, OPUS can release the lock on the node before acquiring the lock on the child node, which means that OPUS abstains from lock coupling in the case of MBR modification. Note that in the R-link tree or CGiST, the insertions must employ lock coupling to propagate the MBR modifications. Second, the One-Pass traversal reduces the cache misses and computational cost for the tree

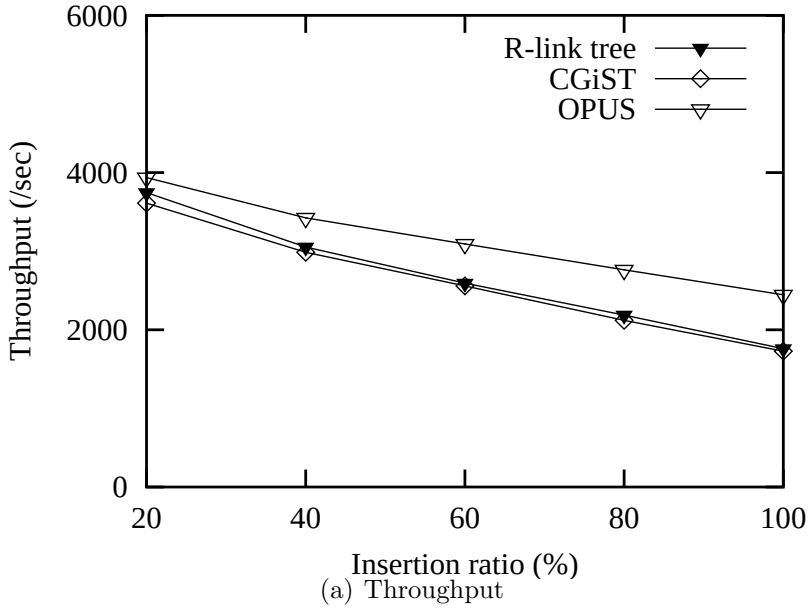
ascents. Note, once the operation accesses a node, it has to request a lock, either shared lock or exclusive lock, i.e. a memory write occurs. The OPUS algorithm can significantly reduce this kind of operation. In addition, the split operations of OPUS typically propagate to an ancestor by only one level, and the parent node still resides in cache in most cases. When the number of threads is small (< 8), all three methods yield good scalability. After that point, the rate at which the throughput increases for the R-link and CGiST become slower due to the lock conflict which add to the wait time of the all blocked operations, e.g. the throughputs do not change much after 16. As the number of threads increases, the gap between the OPUS and the other two methods increases, due to better scalability achieved by OPUS. The OPUS algorithm is around 30% better than the R-link tree and CGiST when the number of threads is large.

Performance of Insertion-Query Workload

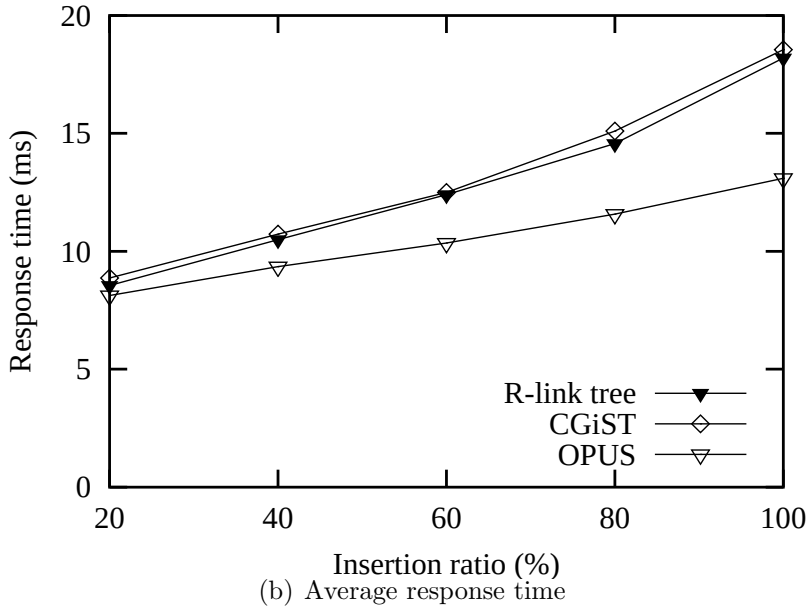
Figure 5.11 shows the throughput and response time of a mixed workload with varying ratios of insertion and query. The workload consists of a sequence of 10,000 operations, each of which is randomly determined to be an insert or query operation based on the given ratio. 32 threads run concurrently for these operations.

When the insertion ratio is equal to 0, we have a read-only scenario with 32 threads running concurrently, and all the three schemes perform equally well. The reason is obvious - regardless of whether a lock is acquired, the search operation of all three methods does not block other concurrent operations due to lock compatibility. When the insertion ratio is minor, the performances of these three methods do not differ much. To demonstrate the difference well, we plot the results where the insertion ratio is larger than 20%.

We observe that the throughput decreases when the ratio of insertion increases



(a) Throughput



(b) Average response time

Figure 5.11: Performance with varying insertion ratio

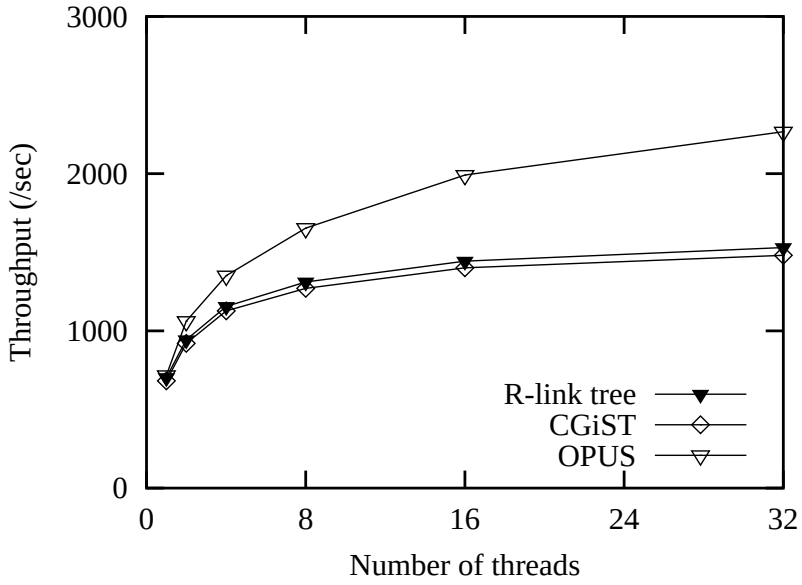


Figure 5.12: Effect of skew insertions

for all the methods as more blocks result from more insertions. OPUS provides the best performance when the insertion ratio is high, and outperforms the R-link tree and CGiST by 30%. The gain is due to the removal of the lock coupling incurred by MBR modification propagations. The lock coupling is the most expensive operation in the context of the R-tree's concurrency control, which entails the holding of multiple locks on nodes for multiple levels. The multiple locks undermine the performance of concurrent mixed operations for CGiST and R-link tree significantly.

On Skew Insertion and Query Distributions

In the above experiments, the data points are randomly distributed and queries are uniformly distributed, and so are the operations on the index. To evaluate the performance of the algorithms on skew insertion/query distribution, we use *Zipf* distribution to generate skewed workloads toward certain spatial region in order to create hot spots on some branches. The throughputs for insertion-only and insertion-query workloads are shown in Figures 5.12 and 5.13.

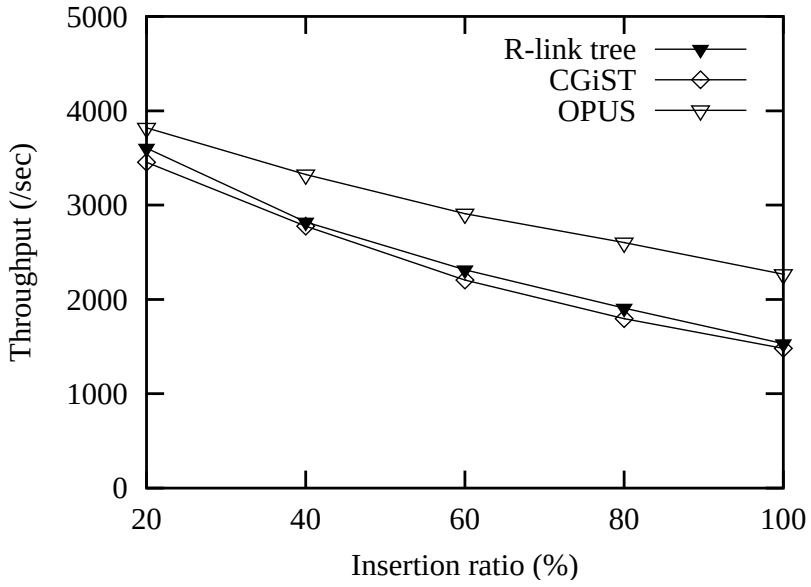


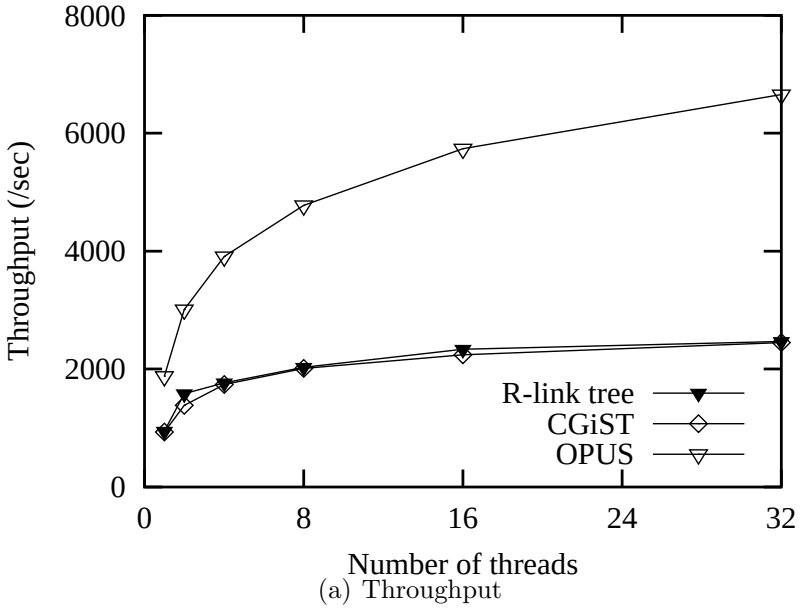
Figure 5.13: Effect of skew insertion/query

Compared with Figures 5.10 (a) and 5.11 (a), we found that all the methods deteriorate in performance. This is expected as a skew distribution led to a high proportion of operations accessing the same regions, resulting in more lock conflicts between the concurrent insertion/query operations, and more blocking time. Among the three methods, OPUS still yields the best throughput.

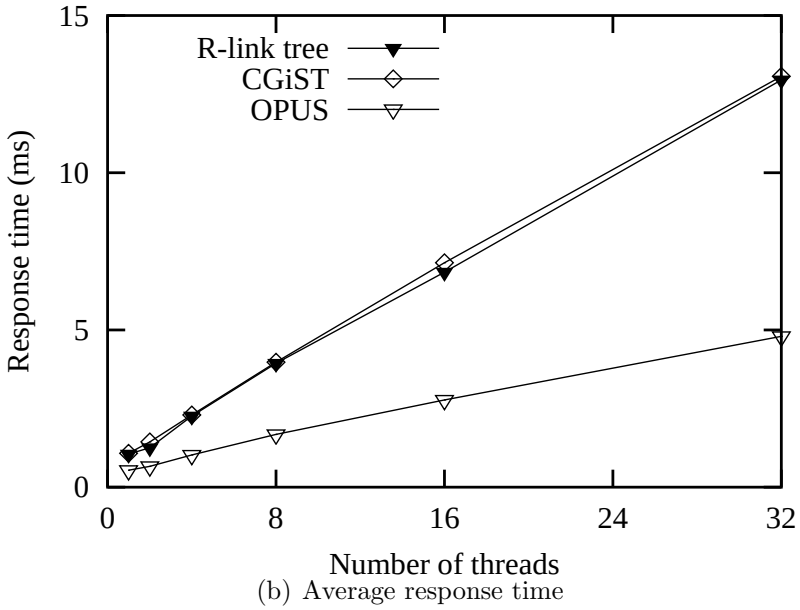
5.4.3 Effect of Deletions

In this experiment, we study the performance on deletions. The underlying R-tree is initialized with 1,000,000 2-dimensional points that are generated randomly. We then ran each concurrency control algorithm with 10,000 deletions. As in general, we apply lazy deletion, i.e. do not merge nodes when they are underflow. Figure 5.14 show the throughput and average response time against the the number of threads performing deletions.

When there is one thread, R-link tree and CGiST have similar performance, while the OPUS is much faster. OPUS can access the objects via the hash table,



(a) Throughput



(b) Average response time

Figure 5.14: Performance of deletions

thus the tree traversal time to locate the object is eliminated. Furthermore, the OPUS scheme scales better than the other two mechanisms in terms of number of threads. Using the hash table to locate the leaf node, OPUS can reduce the lock conflicts with other concurrent operations in the tree. As in the hash table, all the buckets scatter about the table, thus it incurs less interference between concurrent operations. For delete operation, the OPUS is about 100% better in most cases.

5.4.4 Effect of Frequent Modifications

In this set of experiments, we will evaluate the performance of the three concurrency methods under the setting where updates are frequent. An application that requires fast and frequent updates is the moving object database, and we shall use such a database for our testing purposes. In 2-dimensional moving object databases, moving objects disclose their spatial location frequently and each disclosure causes a modification on the index where a data point is moved from one region to another. To simulate such volatile situations, we employ the GSTD [87] algorithm to generate 1,000,000 moving objects in a 2-dimensional space according to different distributions, different movement natures (random, directed) and different speeds (distance moved between updates, from 0.001 to 0.01).

Effect of Speed

First, we investigate the ability of OPUS to handle moving objects by varying the speed at which objects move, from 0.001 to 0.01. We built a tree with 1,000,000 data points, then perform 10,000 location modifications on the index with single thread. Figure 5.15 shows the total execution time for different index structures. The random movement and directed movement yield similar performances in our experiment, and therefore we only show the results for the random movement.

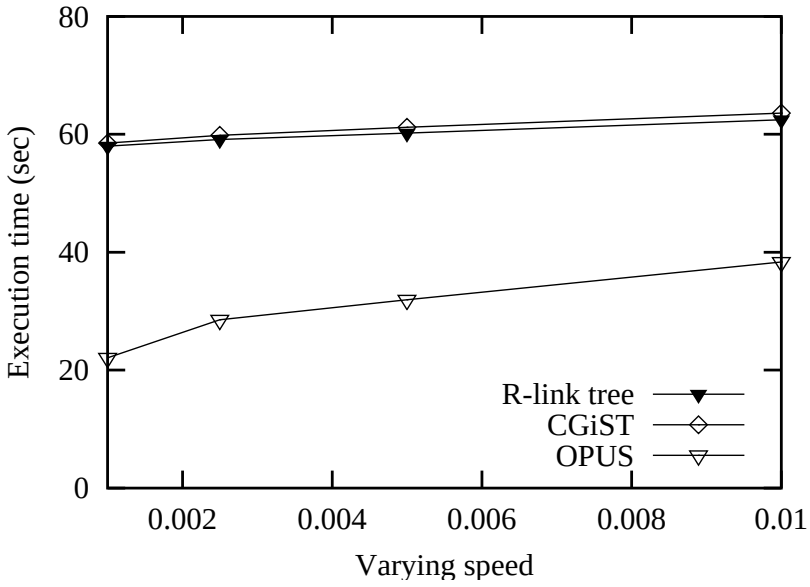


Figure 5.15: Effect of varying speed

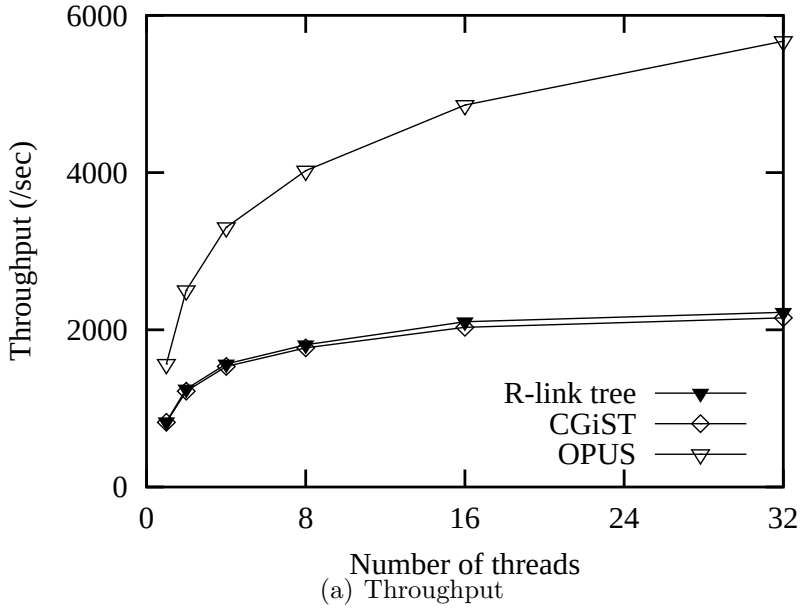
As illustrated in Figure 5.15, an increase in speed inevitably results in an increase in the update cost for all three index structures. The effect of speed on the R-link tree and CGiST is not significant, because once the points moves out of the leaf MBR, they always reinsert the points from the root. However, OPUS consistently outperforms the R-link tree and CGiST even when the speed is as high as 0.01. The performance of OPUS improves significantly, and is at least 60% faster for varying speeds. OPUS updates the objects bottom up and most of the update is localized in the lowest levels, therefore fewer nodes (memory) need to be accessed, i.e. less computational cost and fewer cache misses. Additionally, we use a secondary hash table to locate the objects in the leaf nodes, thus eliminate the tree traversal time to search for the object, which is one of the main costs in the location modification operation. But when the speed is high, say 0.01 in the experiment, the gap between OPUS and other methods becomes smaller. When the speed is high, the possibility is higher for OPUS to traverse up to the upper levels in order to locate the bounding node for some objects.

Concurrent Modifications

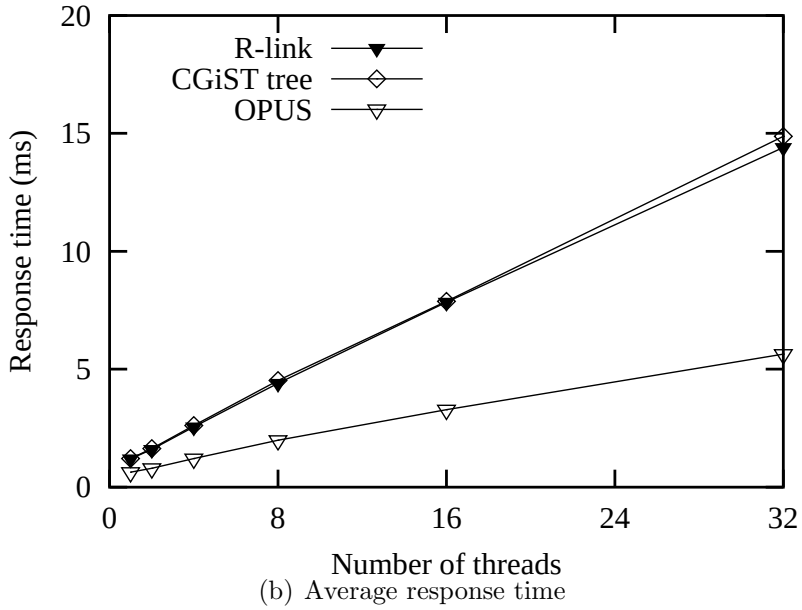
In this experiment, we shall evaluate the concurrent modification algorithms on data points due to object movement. The speed is set at 0.005 as a default value. Figures 5.16 shows the performance of pure modifications with multiple threads.

We can observe that OPUS is around 80% better than the R-link tree and CGiST when there is one thread. First, OPUS accesses the objects via the hash table, so it eliminates the tree traversal time to locate the object. Second, OPUS attempts to update the objects in the lowest levels first to avoid the expensive re-insertion, which can significantly reduce the cache misses and computational cost. In the experiment, it is noted that most of the modifications can be limited to the lowest two levels of the tree. Even if the objects move out of the leaf MBR, the objects are still in the bounding rectangle of the parent node in most of the cases. In the worst case, OPUS may however trace back to the root. Even so, the insertion of OPUS with preparatory operation provides higher throughput. In contrast, for the R-link tree and CGiST, the modification must re-insert the object unless it moves inside the leaf node.

When the number of threads increases, all the methods increase in their throughput. In particular, OPUS yields better scalability, and the improvement is more than 150% over the other two schemes for 32 threads. The gain is due to the fact that most of the modifications occur in the lower levels and since the target objects are randomly selected, they do not interfere with other concurrent operations. With the R-link tree and CGiST, the process must locate the leaf node whose MBR bounds the object via tree traversal and it may re-insert the object into the tree. Both operations need to start from the root of the tree. In this case, the re-insertion may block the other operations, and hence reduce the throughput of modifications. On the contrary, OPUS can locate the leaf node very efficiently



(a) Throughput



(b) Average response time

Figure 5.16: Performance of modification

with very low concurrency overhead by using the hash table, thus reduces the lock conflicts with other concurrent updates.

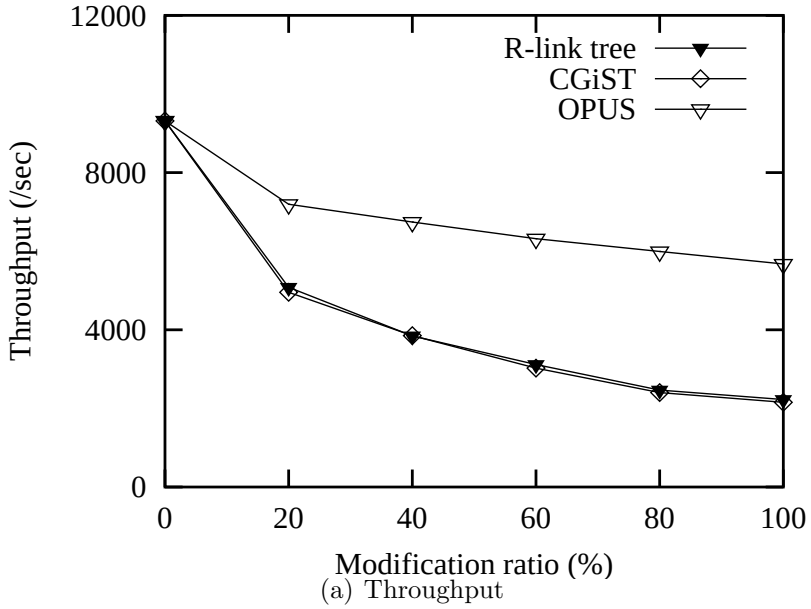
Mixed Concurrent Modifications and Queries

In this experiment, we investigate how OPUS performs when we mix concurrent modifications and queries with varying ratio. 32 threads run concurrently in the experiment. Figure 5.17 shows the results.

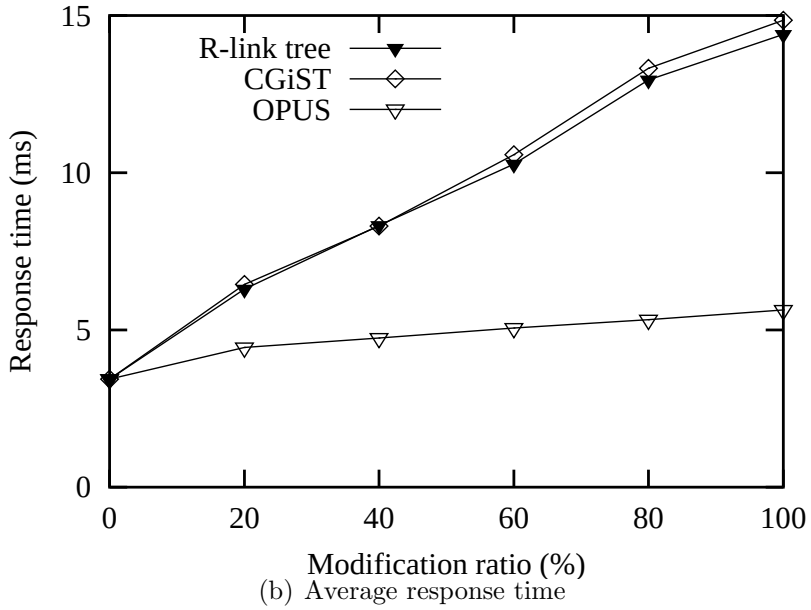
For all read-only queries, all the algorithms yield similar performance for the same reasons as explained previously. As the ratio of modifications increases, OPUS yields better scalability compared to the other two structures, and is more than 60% better when the modification ratio is above 40%. Because the queries and modifications are generated randomly, the operations scatter about the index for all indexes. The modification of OPUS is performed bottom up, hence the modification will not block other concurrent operations. OPUS can access the leaf node directly using the hash table, and the update is localized. Therefore, the least blocking is induced by OPUS. In contrast, the modifications of the R-link tree and CGiST may block the concurrent operation because they traverse and re-insert the new object from the root node of the indexes, thereby reducing the overall throughput.

5.5 Summary

In this chapter, we have addressed the problem of main memory concurrency control for R-tree. We proposed an efficient CC algorithm, called OPUS, that facilitates high throughput in the midst of frequent updates. The main blocking factors for insertions in the R-tree is the node split and MBR modification. The OPUS algorithm enables the R-tree to perform insertions in a top-down fashion using *preparatory operations*, based on early splitting and MBR modification. To deal



(a) Throughput



(b) Average response time

Figure 5.17: Performance with varying modification ratio

with the deletion and modification operation, a secondary hash table and backward pointer are introduced to speed up the processing. OPUS algorithms has some advantages over existing CC algorithms. First, the One-Pass traversal mechanism can not only reduce lock conflict, but also the cache misses and computational cost. Second, the localized modification constrains the affected area of update and incur less interference to other concurrent operations, thereby improving the throughput. Third, for delete and location modification operations, the secondary hash table can eliminate the tree traversal cost to locate the object, and hence reduce the work load on the tree and lock conflicts. In the experimental comparisons with the R-link and CGiST-based schemes, we have shown that our proposed algorithm outperforms other techniques in main memory environment.

CHAPTER 6

Conclusion

Database systems are used to manipulate large volumes of data on secondary storage, and pervade more and more areas of research, business and daily life. With increasingly larger main memory sizes, it is now possible to store whole databases into main memory. Hence indexing memory-resident data becomes an emerging domain of research.

6.1 Thesis Contributions

In this thesis, we have examined the problem of main memory index techniques. A summary of our main contributions is as follows:

1. We have revisited the single-dimensional indexing in the main memory environment. To improve performance, an index structure must facilitate effective use of L2 caches and the CPU. We have optimized the Bounded Disorder (BD) method for main memory processing. To demonstrate the performance of the B^+-tree, CSB^+-tree and BD-tree, we have presented analytical models to evaluate query cost. Query cost has been modelled as a function of three variables: the number of L2 cache misses, TLB misses and instructions. We

have also conducted an experimental evaluation of the BD-tree and compared it against the B^+-tree and the CSB^+-tree.

2. To deal with high-dimensional data, we have proposed a novel multi-tier index structure, called the Δ-tree, that can facilitate efficient search in the main memory environment. Each tier in the Δ-tree represents data space as clusters in different number of dimensions, and tiers closer to the root partition data space using fewer number of dimensions. The numbers of tiers and dimensions are obtained using the PCA technique. By reducing the number of dimensions in internal nodes, we can better utilize the L2 cache and reduce distance computation. We have demonstrated insertion, deletion and different kinds of queries on the Δ-tree. An extension of the Δ-tree, called the Δ^+-tree, has also been proposed to further reduce search space. The Δ^+-tree method globally clusters data space and then partitions clusters into small regions before building the tree. We have implemented the proposed index structure and conducted extensive experiments for performance evaluation on different kinds of datasets. The experimental results show that the Δ^+-tree outperforms other indexes by a wide margin.

3. Traditional concurrency control algorithms on the R-tree are disk-based, and cannot adequately handle a high degree of concurrent accesses involving updates. We have presented a novel main memory concurrency control algorithm for the R-tree, called OPUS (One-Pass UpdateS), that facilitates high throughput in the midst of frequent updates. In most cases, OPUS traverses each R-tree node *once* during an update operation. Insertions are performed top-down using *preparatory operations*, based on early node-splitting and MBR modification. Deletions are bottom-up through the support of an auxiliary structure on object identifiers. A hash table is used to locate the ob-

jects to be deleted. The bottom-up traversal is necessary to tighten the MBR of ancestor nodes. Modifications of spatial content are done by combining a bottom-up deletion and a localized insertion. OPUS offers several advantages towards achieving high throughput. First, the One-Pass traversal mechanism reduces lock conflict, cache misses and computational cost. Second, the localized modification constrains the affected area of update and causes less interference to other concurrent operations, thereby improving throughput. Third, for delete and update operations, the secondary hash table can eliminate the tree traversal cost of locating objects, and hence reduce workload on the tree and lock conflicts. We have implemented OPUS and studied its performance against other well-known concurrency control algorithms. Our results show that OPUS is superior and yields high throughput.

6.2 Future Work

Like most other research, the work presented here leaves some questions unanswered and even uncovers new problems. Some of these open issues on main memory index techniques should be mentioned here.

First, further experimental study against other methods can be conducted. In [26], the prefetching technique is proposed to accelerate query performance using the B^+-tree. Prefetching can effectively overlap multiple cache misses when accessing a tree node, and hence reduce cache stalls in tree operations. Since the prefetching scheme is orthogonal to the index structures, i.e., it can be applied to any indexes, it is interesting to see how performance may improve if we combine the prefetching technique with our index structures.

Second, note that we use K-means to cluster data in the Δ-tree, and the num-

bers of reduced dimensions are identical at the same levels of each sub-tree. This mechanism does not yield optimal performance as clustering cannot reflect the overall information of the dataset if the clusters are skewed. Therefore, we would like to apply other clustering algorithms that can cluster datasets more intelligently according to data distribution, and decide the level of the sub-tree and the numbers of reduced dimensions in each level respectively.

Third, rapid advancements in positioning systems such as GPS technology have made it feasible to track and record the changing positions of continuously moving objects. In order to keep track and manage the large number of moving objects, the locations of moving objects are maintained in databases; the efficient processing of queries on databases of moving objects has become an important problem. However, most current moving object indexing techniques are disk-based. Clearly, more efficient processing of updates on moving object indexes may be obtained through the aggressive use of all the available main memory. Simply using buffering does not imply that all the available main memory is being utilized in any optimal fashion. One may instead expect the index to reside in main memory, in a form that is optimized for the particular main memory and processor environment. In this respect, the area of main memory database management – where main memory is aggressively exploited – may have ideas to offer. For example, more elaborate index structures can be employed to optimize CPU performance, or hash structures can be used to eliminate the expensive tree operation.

Fourth, recently a new class of data-intensive applications has become widely recognized: applications in which the data is modelled as transient *data streams*, e.g. financial applications and sensor networks. In the data stream model, individual data items arrive continuously in multiple, rapid, time-varying, unpredictable and unbounded streams. It is not feasible to load the data stream into a tradi-

tional DBMS and operate on it. The queries on data stream are typically time sensitive, main memory techniques on data streams must be developed to provide fast response. However, since data streams are potentially unbounded in size, the amount of memory required to compute an exact answer to a data stream query may also grow without bound. Given a bounded amount of memory, it is not always possible to produce exact answers for data stream queries; however, high-quality approximate answers are often acceptable in lieu of exact answers. Thus, there exist challenges in designing algorithms that can give approximate answers using the available limited memory, e.g. how to produce approximate answers under bounded memory? how to optimally consider a set of queries, and minimize overall approximation with the best memory allocation?

BIBLIOGRAPHY

[1] *Corel Image Features*. available from http://kdd.ics.uci.edu.

[2] The calibrator tool. http://www.cwi.nl/ manegold/Calibrator/, 1999.

[3] G. Adel'son-Vel'skii and E. Landis. An algorithm for the organization of information. *Soviet Mathematics*, 3:1259–1263, 1962.

[4] A. Ailamaki, D. J. DeWitt, M. D. Hill, and D. A. Wood. Dbms on a modern processor: Where does time go. In *Proc. 25th International Conference on Very Large Data Bases*, pages 266–277, 1999.

[5] A. Analyti and S. Pramanik. Fast search in main memory databases. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 215–224, 1992.

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

[7] R. Bayer and M. Schkolnick. Concurrency of operations on B-trees. *Acta Informatica*, 9:1–21, 1977.

[8] N. Beckmann, H. P. Kriegel, R.Schneider, and B.Seeger. The r*-tree: An efficient and robust access method for points and rectangles. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 322–331, 1990.

[9] J. L. Bentley. Multidimensional binary search in database applications. In *IEEE Transactions on Software Engineering. 4(5)*, pages 397–409. 1979.

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

[11] S. Berchtold, C. Bőhm, and H. P. Kriegel. The Pyramid-tree: Breaking the curse of dimensionality. In *Proc. of the ACM SIGMOD Conference*, pages 142–153, 1998.

[12] 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 *Proc. 16th International Conference on Data Engineering*, pages 577–588, 2000.

[13] S. Berchtold, C. Bohm, D. Keim, F. Krebs, and H. P. Kriegel. On optimizing nearest neighbor queries in high-dimensional data spaces. In *Proc. 8th International Conference on Database Theory*, pages 435–449, 2001.

[14] S. Berchtold, D. A. Keim, and H. P. Kriegel. The x-tree: An index structure for high-dimensional data. In *Proc. 22th International Conference on Very Large Data Bases*, pages 28–39, 1996.

[15] E. Bertino, B. C. Ooi, R. Sacks-Davis, K.L. Tan, J. Zobel, B. Shidlovsky, and B. Cantania. *Indexing Techniques for Advanced Database Systems.* Kluwer Academic, 1997.

[16] C. Bhm, S. Berchtold, and D. Keim. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. *ACM Computing Surveys*, 2001.

[17] P. Bohannon, D. Lieuwen, R. Rastogi, A. Silberschatz, S. Seshadri, and S. Sudarshan. The architecture of the dali main memory storage manager. *Multimedia Tools and Applications*, 4(2):115–151, 1997.

[18] P. Bohannon, P. Mcllroy, and R. Rastogi. Main-memory index structures with fixed-size partial keys. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 163–174, 2001.

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

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

[21] A. Cardenas. Analysis and performance of inverted database structures. In *Communication of ACM*, volume 18, pages 253–264, May 1975.

[22] S. K. Cha, S. Y. Hwang, K. Kim, and K. Kwon. Cache-conscious concurrency of main-memory indexes on shared-memory multiprocessor systems. In *Proc. 27th International Conference on Very Large Data Bases*, pages 181–190, 2001.

[23] K. Chakrabarti and S. Mehrotra. Dynamic granular locking approach to phantom protection in r-trees. In *Proc. 14th International Conference on Data Engineering*, pages 446–454, 1998.

[24] K. Chakrabarti and S. Mehrotra. Local dimensionality reduction: A new approach to indexing high dimensional spaces. In *Proc. 26th International Conference on Very Large Data Bases*, pages 89–100, 2000.

[25] J. K. Chen and Y. F. Huang. A study of concurrent operations on r-trees. In *Information Science*, pages 263–300, 1997.

[26] S. Chen, P. B. Gibbons, and T. C. Mowry. Improving index performance through prefetching. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 139–150, 2001.

[27] S. Chen, P. B. Gibbons, T. C. Mowry, and G. Valentin. Fractal prefeching b+-tree: optimizing both cache and disk performance. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 157–168, 2002.

[28] Y. S. Chen, Y. P. Hung, and C. S. Fuh. Fast algorithm for nearest neighbor search based on a lower bound tree. In *Proc. 8th International Conference on Computer Vision*, pages 446–453, 2001.

[29] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proc. 24th International Conference on Very Large Data Bases*, pages 194–205, 1997.

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

[31] B. Cui, B. C. Ooi, J. W. Su, and K. L. Tan. Contorting high dimensional data for efficient main memory processing. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 479–490, 2003.

[32] B. Cui, B. C. Ooi, J. W. Su, and K. L. Tan. Main memory indexing: The case for bd-tree. *IEEE Transactions on Knowledge and Data Engineering*, 2003.

[33] B. Cui, B. C. Ooi, K. L. Tan, and Z. Y. Huang. *OPUS: The Tune of Fast Concurrent Updates on R-trees*. Technical Report, School of Computing, National University of Singapore, 2003.

[34] R. Enbody. *Perfmon: Performance Monitoring Tool*. available from http://www.cps.msu.edu/ enbody/perfmon.html, 1999.

[35] K. Eswaren, J. Cray, R. Lorie, and I. Traiger. On the notions of consistency and predicate locks in a database system. In *Communication of ACM*, volume 19, pages 624–633, Nov 1976.

[36] R. Fagin, J. Nievergelt, N. Pippenger, and H. R. Strong. Extendibale hashing: A fast access method for dynamic files. *ACM Transactions on Database Systems (TODS)*, 4(3):315–344, September 1979.

[37] C. Faloutsos. Gray codes for partial match and range queries. In *IEEE Transactions on Software Engineering 14*, pages 1381–1393. 1988.

[38] R. F. S. Filho, A. Traina, C. Traina Jr., and C. Faloutsos. Similarity search without tears: the omni-family of all-purpose access methods. In *Proc. 17th ICDE Conference*, 2001.

[39] M. Freeston. A general solution of the n-dimensional B-tree problem. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 80–91. 1995.

[40] H. Garcia-Molina and K. Salem. Main memory database systems: an overview. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):509–516, 1992.

[41] J. Goldstein and R. Ramakrishnan. Contrast plots and p-sphere tree: Space vs. time in nearest neighbor searches. In *Proc. 26th International Conference on Very Large Data Bases*, pages 429–440, 2000.

[42] G.H. Golub and C.F. Van Loan. *Matrix Computations*. The Johns Hopkins University Press, 1989.

[43] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. of the 19th IEEE Symposium on Foundations of Computer Science*, pages 8–21, 1978.

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

[45] J. Hennessy and D. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan kauffman, 1998.

[46] A. Henrich. The lsd^h-tree: An access structure for feature vectors. In *Proc. 14th International Conference on Data Engineering*, pages 577–588. 1998.

[47] J. Hui, B. C. Ooi, H. Shen, C. Yu, and A. Zhou. An adaptive and efficient dimensionality reduction algorithm for high-dimensional indexing. In *Proc. 19th International Conference on Data Engineering*, 2003.

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

[49] T. Johnson and D. Shasha. The performance of current b-tree algorithms. In *ACM Transactions on Database Systems (TODS)*, 1993.

[50] I. T. Jolliffe. *Principal Component Analysis.* Springer-Verlag, 1986.

[51] C. Traina Jr., A. Traina, C. Faloutsos, and B. Seeger. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, 2002.

[52] I. Kamel and C. Faloutsos. Hilbert r-tree: An improved r-tree using fractals. In *Proc. 20th International Conference on Very Large Data Bases*, pages 500–509. 1994.

[53] K. V. Ravi Kanth, F. D. Serena, and A. K. Singh. Improved concurrency control techniques for multi-dimensional index structures. In *12th International Parallel Processing Symposium*, pages 580–586, 1998.

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

[55] K. Kim, S. K. Cha, and K. Kwon. Optimizing multidimensional index trees for main memory access. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 139–150, 2001.

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

[57] F. Korn and S. Muthukrishnan. Influence sets based on reverse nearest neighbo queries. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 210–212, 2000.

[58] M. Kornacker and D. Banks. High-concurrency locking in r-trees. In *Proc. 21th International Conference on Very Large Data Bases*, pages 134–145, 1995.

[59] M. Kornacker, C. Mohan, and J. M. Hellerstein. Concurrency and recovery in generalized search trees. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 62–72, 1997.

[60] H. Kriegel and B. Seeger. PLOP-Hashing: A grid file without directory. In *Proc. 4th International Conference on Data Engineering*, pages 369–376. 1988.

[61] D. Kwon, S. J. Lee, and S. H. Lee. Index the current positions of moving objects using the lazy update r-tree. In *3rd International Conference on Mobile Data Management*, 2002.

[62] P. Lehman and S. Yao. Efficient locking for concurrent operations on b-trees. In *ACM Transactions on Database Systems (TODS)*, volume 6, pages 650–670, 1981.

[63] T. Lehman and M. Carey. A study of index structures for main memory database management systems. In *Proc. 12th International Conference on Very Large Data Bases*, pages 294–303, 1986.

[64] T. Lehman, E.J. Shekita, and L. Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, 4(6):555–566, 1992.

[65] K. Lin, H. V. Jagadish, and C. Faloutsos. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994.

[66] W. Litwin. Linear hashing: A new tool for file and table addressing. In *Proc. 6th International Conference on Very Large Data Bases*, 1980.

[67] W. Litwin and D. Lomet. The bounded disorder access method. In *Proc. 17th International Conference on Data Engineering*, pages 38–48, 1986.

[68] D. Lomet. A simple bounded disorder file organization with good performance. In *ACM Transactions on Database Systems (TODS)*, volume 13, pages 525–551, October 1988.

[69] D. Lomet and B. Salzberg. The hb-tree: A multiattribute indexing method with good guaranteed performance. In *ACM Transactions on Database Systems (TODS)*, volume 15, pages 625–658, 1990.

[70] H. Lu, Y.Y. Ng, and Z. Tian. T-tree or b-tree: Main memory database index structure revisited. In *Proc. Australasian Database Conference*, pages 65–73, 2000.

[71] C. Mohan and F. Levine. Aries/im: An efficient and high concurrency index management method using write-ahead logging. In *Proc. of the ACM SIGMOD Conference*, pages 371–380, 1992.

[72] V. Ng and T. Kameda. Concurrent accesses to r-trees. In *Proc. Symposium on large spatial databases*, pages 142–161, 1993.

[73] J. Nievergelt, H. hinterberger, and K. C. Sevcik. The grid file: an adaptable, symmetric multikey file structure. In *ACM Transactions on Database Systems (TODS)*, 1984.

[74] B. C. Ooi, K. L. Tan, C. Yu, and S. Bressan. Indexing the edge: a simple and yet efficient approach to high-dimensional indexing. In *Proc. 18th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 166–174. 2000.

[75] D. pfoser, C. S. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. 26th International Conference on Very Large Data Bases*, pages 395–406, 2000.

[76] J. Rao and K. Ross. Cache conscious indexing for decision-support in main memory. In *Proc. 25th International Conference on Very Large Data Bases*, pages 78–89, 1999.

[77] J. Rao and K. Ross. Making b+-trees cache conscious in main memory. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 475–486, 2000.

[78] R. Rastogi, S. Seshadri, P. Bohannon, D. Leinbaugh, A. Silberschatz, and S. Sudarshan. Logical and physical versioning in main memory databses. In *Proc. 23th International Conference on Very Large Data Bases*, pages 86–95, 1997.

[79] J. T. Robinson. The k-d-b-tree: A search structure for large multidimensional dynamic indexes. In *Proc. ACM SIGMOD International Conference on Management of Data*, pages 10–18. 1981.

[80] H. Sagan. *Space Filling Curves*. Springer-Verlag, 1994.

[81] Y. Sagiv. Concurrent operations on b*-trees with overtaking. In *Journal of computer and system sciences*, volume 33, pages 275–296, 1986.

[82] Y. Sakurai, M. Yoshikawa, S. Uemura, and H. Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. In *Proc. 26th International Conference on Very Large Data Bases*, pages 516–526, 2000.

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

[84] B. Seeger and H. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. In *Proc. 16th International Conference on Very Large Data Bases*, pages 590–601. 1990.

[85] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multi–dimensional objects. In *Proc. 13th International Conference on Very Large Data Bases*, pages 507–518. 1987.

[86] S. Song, Y. Kim, and J. Yoo. An enhanced concurrency control scheme for multi-dimensional index structures. In *Proc. 7th International Conference on Database Systems for Advanced Applications*, pages 190–199, 2001.

[87] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. In *Proc. 6th International Symposium on Large Spatial Databases*, 1999.

[88] R. Weber, H. J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proc. 24th International Conference on Very Large Data Bases*, pages 194–205, 1998.

[89] D. A. White and R. Jain. Similarity indexing with the ss-tree. In *Proc. 12th International Conference on Data Engineering*, 1996.

[90] C. Yu, B. C. Ooi, K. L. Tan, and H. V. Jagadish. Indexing the distance: An efficient method to knn processing. In *Proc. 27th International Conference on Very Large Data Bases*, pages 421–430, 2001.

[91] G.K. Zipf. *Human Behavior and the Principle of Least Effort.* Addison Wesley, 1949.