O_2-Tree: A Shared Memory Resident Index in Multicore Architectures

Daniel Ohene-Kwofie

Faculty Of Science

University Of The Witwatersrand

A Dissertation submitted to the Faculty Of Science in fulfilment of the requirements for the degree of

Master Of Science by Research only

signed on October 2012 in Johannesburg

Declaration

I declare that this Dissertation is my own, unaided work under the supervision of Prof. Ekow Otoo. It is being submitted for the Degree of Master of Science at the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination at any other University.

____

____________________day of ________________20 ________________in ________________

Abstract

Shared memory multicore computer architectures are now commonplace in computing. These can be found in modern desktops and workstation computers and also in High Performance Computing (HPC) systems. Recent advances in memory architecture and in 64-bit addressing, allow such systems to have memory sizes of the order of hundreds of gigabytes and beyond. This now allows for realistic development of main memory resident database systems. This still requires the use of a memory resident index such as T-Tree, and the B$^+$-Tree for fast access to the data items.

This thesis proposes a new indexing structure, called the O_2-Tree, which is essentially an augmented Red-Black Tree in which the leaf nodes are index data blocks that store multiple pairs of key and value referred to as "key-value" pairs. The value is either the entire record associated with the key or a pointer to the location of the record. The internal nodes contain copies of the keys that split blocks of the leaf nodes in a manner similar to the B$^+$-Tree. O_2-Tree structure has the advantage that: it can be easily reconstructed by reading only the lowest value of the key of each leaf node page. The size is sufficiently small and thus can be dumped and restored much faster.

Analysis and comparative experimental study show that the performance of the O_2-Tree is superior to other tree-based index structures with respect to various query operations for large datasets. We also present results which indicate that the O_2-Tree outperforms popular key-value stores such as BerkelyDB and TreeDB of Kyoto Cabinet for various workloads. The thesis addresses various concurrent access techniques for the O_2-Tree for shared memory multicore architecture and gives analysis of the O_2-Tree with respect to query operations, storage utilization, failover and recovery.

Acknowledgements

I would like to acknowledge the Center For High Performance Computing (CHPC) South Africa for their financial support for this work. I will also like to thank my advisor, Prof. Ekow Otoo, for the the untold number of hours spent reviewing my work, and also for the many ideas that he suggested. Special thanks to Prof. Michael Sears for his immense help and support. The people with whom I've shared an office (G. Nimako, O. Damilola, Lani) have been a great group. I'd like to thank them for their friendship, moral support, and willingness to answer questions.

Contents

List of Figures

Chapter 1

Introduction

Indexing is the process of associating a key with the location of a corresponding data record in a Database management system (DBMS). Tree structured indices, in particular, are critical to database processing systems since they allow for both random and range queries. Today's data processing tasks in scientific data mining, financial analysis, network monitoring, etc., handle large volumes of data which require fast access with high throughput. Fast indices are thus implemented to facilitate fast query processing in such DBMS. A more common architecture is to maintain an in-memory index to data records that are chunked or grouped into pages. The data items are then accessed through a large in-memory cache pool. In the past, however, it was common for database systems to store index schemes for the database on disk since it was expensive to have the entire index structure permanently in memory. The required index blocks are then transferred to main memory on demand for manipulation. B^{+}-Tree is a widely used index for disk-based database management systems since it guarantees few disk accesses to read and write disk blocks during query processing and supports both random and range queries efficiently.

Recent advances in memory architectures with 64-bit addressing, allow for memory sizes of the order of hundreds of gigabytes and beyond at a reasonable cost. It has, therefore, become feasible to have sufficiently large shared memory such that the entire index of either, a memory resident or disk-resident database, can be maintained in main memory. For instance, the latest Oracle Exadata X2-8 system ships with over 2TB of main memory (Oracle 2012*b*). This has, therefore, motivated much research to exploit memory as well as the the many-cores available on such architectures to provide fast application processing for main-memory databases.

In-memory database systems provide extremely fast response time and very high throughput. There are real-time access applications where high performance is required and as such the entire database needs to be loaded in a large main memory store (Kong-Rim & Kyung-Chang 1996). Such memory-resident databases have been receiving the attention of database researchers for quite some time now (DeWitt et al. 1984, Ammann et al. 1985, Lehman & Carey 1986, Bitton et al. 1987, Lehman et al. 1992, Garcia-Molina & Salem 1992, Lu et al. 2000, Oracle 2009). Examples of such in-memory databases are Oracle's TimesTen, IBM's SolidDB, the SAP HANA database and Datablitz. These are desirable for mission critical, embedded, or real-time systems. Examples of these include routing or real-time billing in Telecommunications, financial services applications, e-commerce systems, etc.

However they require mechanism for making both the data records and memory resident index persistent and fault-tolerant. The Oracle TimesTen in-memory database, for instance, delivers real-time performance by managing the entire data in memory, using variants of the T-Tree, without any disk-accesses (Oracle 2006). Such in-memory databases require the optimization and design of appropriate efficient data structures for the data manipulation.

Algorithms designed for main memory database systems (MMDBS), unlike disk based algorithms, focus on efficient use of CPU cycles and memory (for high throughput and low latency) rather than minimizing disk accesses and disk utilization (Lehman & Carey 1986). Such database systems still require the use of a memory resident index for fast access to the data items and in general guarantee very high processing (insertion, deletion, exact-match queries, range queries, and largest/smallest key values searches).

Index structures proposed in the literature for MMDBS includes tree structures as well as hashing. Though hashing provides fast query updates and lookups, it may not be space-efficient and also its does not support range traversals. Therefore, for the purposes of supporting all forms of query operations including range queries, minimum and maximum key lookups efficiently, tree-based indices such as the T-Tree and the B$^+$-Tree, are the preferred structures for indexing. Lehman & Carey (1986) proposed the T-Tree which is well known as a preferred index structure in MMDBS. Several variants have been proposed in the literature to optimise and make the T-Tree efficient index scheme for main memory databases. The T-Tree, however, has the drawback of having a complex balancing algorithm particularly when deletions occur and, also both the T-Tree and B$^+$-Tree require scanning the entire database to rebuild the index. The B$^+$-Tree, which was designed mainly for disk-based index, has been characterized as not suitable for in-memory index structure due to its high space requirement. However, recent research has proven that, they can actually be used for in-memory index as well due to the current trend of memory technology (Rao & Ross 1999).

There has also been a recent flood of main memory index schemes characterised as *NoSQL* databases also referred to as *key/value* pair index structures (Marcus 2012). Notably in this pack are index schemes such as BerkeleyDB (Oracle.com 2011), LevelDB (Google.com 2011) and Kyoto cabinet (FAL Labs 2012). Such in-memory index optimized for in-memory databases and running on multi-core processors can support very high query processing rates. Another challenge with such in-memory indices, however, is how to efficiently ensure that the concurrently executing processes are isolated from each other in such concurrent environment.

1.1 Problem Motivation

As memory sizes increase with the advances in technology, main memory database systems have become increasing popular. The T-Tree, which is the most widely used in-memory index structure, has several drawbacks. The T-Tree, even though proposed over a decade ago, is still relevant in modern database systems and research. High performance in-memory databases such as the Oracle Timesten (Oracle 2006), DataBlitz (Bell Labs 1996), FastDB (FastDB 2012), and MonetDB (MonetDB 2012) use the T-Tree as index schemes. The MySQL Cluster (Oracle 2012*a*) also uses the T-Tree to provide high performance on large databases.

The T-Tree index requires that the entire database be traversed to rebuild the index after failure. The structure also has a high system overhead cost in query operations due to the multiple data comparison at each node. For example, searching for a key which is *l-levels* down from the root will involve *2l* (minimum and maximum key) comparisons at each node before the bounding node is located for further search. Further, the T-Tree requires several up and down traversals to restore the tree's invariant especially during insertion/deletion of keys. Several variants (Kong-Rim & Kyung-Chang 1996, Lu et al. 2000, Lindström 2007) have been proposed to limit the number of up and down traversals. The issues of index recovery (rebuilding after failure), and fault-tolerance still involve high system overhead. The strict adherence to the AVL-Tree balance condition of the T-Tree affects the performance significantly. For instance, the removal or insertion of a node may require that the height or balance factor of each node along the search path be updated. The B$^+$-Tree, which is also a widely used index structure also requires scanning the entire records in the database to rebuild the index.

To resolve some of these issues, this thesis proposes the O_2-Tree as an alternative to the T-Tree and similar related index structures. The O_2-Tree can easily be constructed with minimal traversals of the data pages or data chunks in a main memory database. The O_2-Tree structure provides better performance in terms of query operations for very-large datasets. It also provides an efficient way to ensure that the in-memory data is persistent if desired. It is fault-tolerant since the entire index structure can be easily reconstructed without traversing every record in the database. A concurrent access protocol for the O_2-Tree is presented which inherently scales well in highly concurrent environment.

1.2 Contributions

The major contribution in this thesis is the development of a main memory index structure for database systems. More specifically, the results being reported include:

- Development and implementation of the O_2-Tree as an in-memory index data structure. Other existing structures such as the T-Tree, B^+-Tree, Red-Black Tree, and AVL-Trees are also implemented for comparative studies.
- A proof of the fundamental theorem upon which the development of the O_2-Tree is based. Namely that null leaf pointers in a standard Red-Black Tree that are colored black, never become internal nodes during rotation operations.
- Comparative studies of the T-Tree, O_2-Tree, B^+-Tree, AVL-Tree and the Top-down Red-Black Tree with varying dataset to simulate real world datasets.
- Implementation of a persistent store with the O_2-Tree using *BerkeleyDB DBMpoolFile* in-memory cache pool.
- Comparative study and analysis of the O_2-Tree through in-memory cache, with NoSQL key-value stores such as the TreeDB of Kyoto Cabinet, BerkeleyDB and Google's LevelDB library.
- Development and implementation of a concurrent access protocol for the O_2-Tree for shared memory multicore machines.

1.3 Organisation Of the Thesis

The remainder of this thesis is organised as follows. Chapter 2 gives a brief survey of different data structures that can be used for main memory index. Examples include the T-Tree, AVL-Tree, the B^+-Tree and their variants. In Chapter 3, we give the details of the O_2-Tree in-memory index structure. The structure is described with discussions of the various query operations supported by the O_2-Tree. Chapter 4 describes the analytical model for the tree index and a generalisation of the storage utilisation of the in-memory index. We present the various algorithms for concurrent access of the O_2-Tree in shared memory in Chapter 5. We examine the strict balance algorithms as well as relaxed balanced algorithms for concurrency control. In Chapter 6, we present the mechanisms employed to implement index persistence and fault recovery using existing in-memory cache libraries. Chapter 7 discusses the various experimental performance evaluations conducted. The experimental environment and setup are discussed and the various simulated and real-life workloads used are also discussed. We finally summarize the work of this research in Chapter 8, and give some directions for future work.

Chapter 2

Related Work

Indexing remains important in database systems. Though sequential data access in main memory is cheaper than in disk-based systems, the different classes of queries do not always submit to sequential processing. Random and range queries require efficient random search to the first record followed by sequential scan for the required range. Many index structures exist for main memory indexes. A few from the literature relevant to the thesis are discussed.

2.1 Main Memory Index Structures

2.1.1 The T-Tree

Lehman and Carey (1986) proposed the T-Tree as a new data structure for main memory database management. The T-Tree was developed by combining features of the AVL-Tree and the B-Tree. A T-Tree structure consists of T-nodes. In the T-Tree, each T-node stores more than one pair of a “key, value” pairs. The T-Tree maintains the intrinsic binary search nature of an AVL-Tree but has better storage utilization since each node contains several data items in sorted order. Each T-node has only two children pointers to lower level nodes just as in AVL-Trees. Therefore, query operations on a T-Tree are similar to that of an AVL-Tree, but the T-Tree has a better performance due to the following reasons:

1. Each T-Node contains more than one data item and hence the height of the tree is much smaller than the AVL-Tree
2. Data movement is required for updates (insertion, deletion) but it’s usually within a single node and thus it is much faster.
3. By grouping several data items in a single node, the number of rotations to balance the tree is significantly reduced. Rotations are only performed when a node overflows due to insertion or underflows due to deletion.

A T-Tree consists of three different types of nodes; an internal node, a leaf node and a half-leaf node. An internal node is a T-node which has a left and right child. A leaf node is a T-node with NIL child pointers (no children), and a T-node with only one child is a half-leaf node.

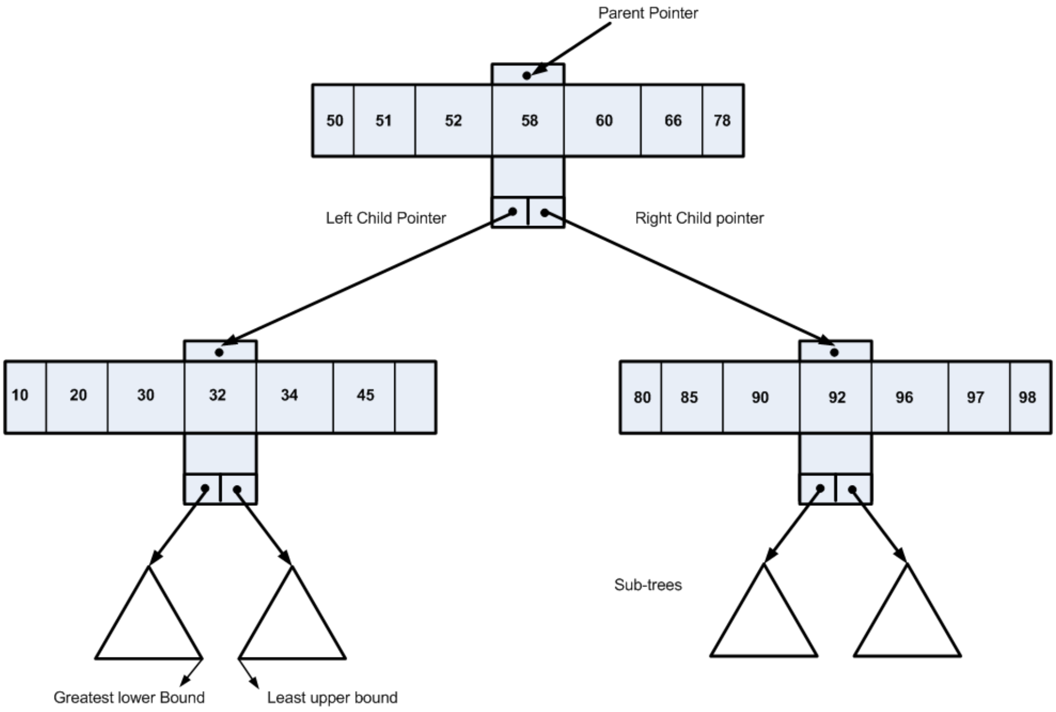


Figure 2.1: Structure of the T-Tree indicating the pointers in each T-Node

Let the records being organised consist of pairs of keys and values of the form $\langle key, value\rangle$. The value is either an entire record that includes the key or a pointer to the location where the record is stored. Only the keys are shown in the illustrations below. To search for an item with key, k, the T-Tree is traversed starting from the root of the tree and continues to the children just as in the AVL-Tree. Let x denote a node visited during the traversal from the root node and let $x.k_{min}$ and $x.k_{max}$ denote the minimum and maximum keys respectively of the node x. If the search key k is less than the minimum k in the node x $(k < x.k_{min})$, the search continues along the left child. However, if $(k > x.k_{max})$, the search continues along the right child. Otherwise the current node will be searched either using a binary search or a sequential scan for the item with key k.

To insert a new data item q_{new}, the bounding T-node is identified using the search algorithm. If a node is found and there is room for another entry, the item is inserted; otherwise the data item q_{min}, with the minimum key in the node is removed while q_{new} is inserted. The item q_{min} then becomes the new item to be inserted. The search continues directly to the node holding the greatest lower bound of q_{min}. The item value is inserted here as the new item whose key is the greatest lower bound if there is room. Otherwise, a new leaf is created and the tree is checked for balancing.

The deletion algorithm is similar to the insertion operation. The data item to be deleted is first located by searching, and the operation performed as necessary. If the deletion does not cause an underflow (i.e. node has more than the minimum allowable number of data items), the item is deleted; else if it's an internal node then the item key is replaced with the one whose key is the greatest lower bound from a leaf or half-leaf to keep its occupancy. However, if the node is a leaf, then the item is simply removed. Leaves are permitted to underflow in a T-Tree. On the other hand, if the node is a half-leaf node, and can be merged with a leaf node, the two nodes are merged into one node and the other discarded. The tree is then checked for balance as in the AVL-Tree. The T-Tree supports other query operations such exact-match searches and range searches. The T-Tree has a significant low depth which guarantees fast queries as well as better storage utilization as compared to the AVL-Tree. A brief discussion of the AVL-Tree is provided below.

2.1.2 The AVL-Tree

The AVL-Tree is one of the earliest balanced binary tree, first discovered by Adelson-Velskii and Landis(1962). It is a height balanced tree such that for each node x, the height of the left and right subtrees differ by 1. Suppose $height(x)$ returns the height of a node x, and let $x.left$ and $x.right$ denote the left and right subtrees of node x respectively. Then, for an AVL-Tree we have that:

$$|x.right - x.left| < 1$$

An extra height attribute or balance factor is assigned to each node in an AVL-Tree to help maintain balance (Cormen et al. 2009). This imposes a strict balance on the tree's depth and therefore ensures a worst-case height of $\mathrm{O}(\log_2 N)$, where N is the number of keys in the tree. Exact-match search and update operations start from the root and traverse the tree down to the target node. Insertions and deletions operations on the AVL-Tree, as in any binary tree, may alter the height of the tree, which can result in an unbalanced tree. Single or double rotations along the search path are used to restore the balance of the tree. Query operations on an AVL-Tree take $\mathrm{O}(\log_2 N)$ and are therefore useful candidates for main memory database management systems. The major drawback with AVL-Tree is its height and poor storage utilisation. Each node stores only one data item, as well as two pointers and additional control information to maintain balance. In an attempt to reduce the height of the tree, the T-Tree and it variants were proposed.

2.1.3 Variants of the T-Tree

The T-Tree has major drawbacks with traversal during range query processing as well as data movement during underflows and overflows. This is due to the fact that, for range queries, the retrieval algorithm must traverse up and down the tree to get the range of values. A similar problem exists with handling overflow and underflow of internal T-nodes. The in-order predecessor or successor, which is a leaf or half-leaf, must be located to fix the underflow/overflow of internal nodes as discussed in Section 2.1.1 above. Also, the change in height of a sub-tree requires that the tree be traversed along the search path to update the height accordingly.

Kong-Rim and Kyung-Chang (1996) proposed the T*-Tree to improve the T-Tree for range query searches. T*-Tree is basically a T-Tree with additional pointer from each node to its successor node. This pointer enables a node to directly jump to its successor without moving along the search path from node to node. Lindstrom (2007) proposed the T^{LINK}-Tree which is similar to the T*-Tree. All records in a T^{LINK}-Tree are kept in the leaf nodes and therefore the internal nodes serve as routers to the leaves. T^{LINK}-Tree also adds two additional pointers to the nodes, successor and predecessor pointers. Furthermore, a node contains a low key value of records pointed by a key pointer. Thus, every key value represents a range of records between that key value and next key value. The last key value of the node represents the high-value of the left most record page (Lindström 2007). The T^{LINK}-Tree does not only provide a further reduction in the height of the T-Tree but also minimizes tree traversals especially during range queries and updates.

These and many other variants have been developed to enhance the structure and improve the performance of the T-Tree especially for range queries. However, the issue of having to balance the tree (when T-nodes are inserted or deleted) is still of a major concern. This is due to the strict balance requirement of the AVL-Tree balance invariant of the tree. Lu et al. (2000), proposed the T-Tail Tree which seeks to delay the balance in such cases. However, eventually, the tree must be balanced all the way up the root and the height of each node along the update path is updated accordingly.

2.1.4 The B$^+$-Tree

B$^+$-Tree is a multi-way search tree in which each node holds more than one data item. B$^+$-Tree is the most common dynamic index structure in database systems, especially for disk-based Database Management Systems(DBMS). In such DBMS, the number of disk access per search is proportional to the height of the tree. B$^+$-Tree therefore has a significantly low height for high *fanout* referred to as the tree order. B$^+$-Tree is an improvement of the B-Tree introduced by Bayer & McCreight (1972). A B$^+$-Tree requires that all keys reside in the leaves. Formally, a B$^+$-Tree of order m has the following properties.

1. The keys are stored at the leaves and all leaves are at the same level.
2. Internal nodes store as many as $m - 1$ keys to guide the search path.
3. Every internal node (except the root) has at least $\lceil m/2 \rceil$ children.
4. The root is either a leaf or has between 2 and m children.
5. Leaf nodes have between $\lceil m/2 \rceil$ and m data items.

Comer (1979) surveyed and described several variants of a B-Tree, besides the B$^+$-Tree. B*-Tree requires that every node be at least two-thirds full i.e., $(2m - 1)/3$. This ensures efficient use of space and also enhances search time. However, update operations (insert and delete) gets slower since nodes require extra attention as they fill up. We focus on the B$^+$-Tree for this research since it provides better performance for the update operations.

The order of a B$^+$-Tree determines its branching factor or fanout. A large branching factor significantly reduces the height of the tree and the traversal depth to find a key. For instance a B$^+$-Tree of order m and N keys will have a height of $O(\log_{\lceil m/2 \rceil} N)$. A large m implies a greater logarithmic base and hence a lower tree depth which guarantees reasonably faster query operations.

In a B$^+$-Tree, when an overflow occurs, the affected node is split into two and a copy of the median key is promoted to the upper level internal node whiles retaining the actual key as the minimum in the right leaf key in the leaf node. The leaves are also linked together in a symmetric order to form a sequence set. This makes it possible to traverse the tree efficiently for either random key searches or for sequential access in retrieving range of keys.

Recent research (Lee et al. 2007, Rao & Ross 1999, Rao & Ross 2000), focuses on improving performance of these index structures by making use of modern architectural design of CPUs and the cache-line behaviour in particular. Rao et. al proposed the CSS+ Tree and showed that B$^+$-Tree has better performance on modern architecture. They also optimised it to support fast updates in CSB+ (Rao & Ross 2000). It was observed that having node sizes of a B$^+$-Tree larger than the cache line reduces the translation lookaside buffer (TLB) misses and thus effectively improves the performance of the tree index (Hankins & Patel 2003).

A much recent research also proposed fast architecture sensitive tree search on modern CPUs and GPUs. *FAST* (Kim et al. 2010), is an architecture sensitive layout of a binary search tree, which is logically organised to optimise for architecture features like page size, cache line size, and Single instruction, multiple data (SIMD) width of the underlying hardware. *FAST* is designed to eliminate the impact of memory latency, and exploit thread-level as well as data-level parallelism on both CPU and GPUs (Kim et al. 2010). Our objective in this research however, is to optimise and improve the performance of in-memory index database schemes to support fast recovery and persistent storage while minimizing the number of rebalancing procedures required by an ordinary binary tree index.

2.2 NoSQL key-value Databases

NoSQL is a broad class of database management systems, different from the traditional relational database model, that have been gaining grounds in recent times. These data stores usually have no concept of tables, relations, joins and they do not use structured query language (SQL). They consist basically of a $\langle key - value \rangle$ pair and therefore such databases are able to scale easily with huge datasets. The structure of the O_2-Tree is suitable for implementing a NoSQL data store. It is therefore, necessary to compare the performance of the O_2-Tree with other structures that have been used in implementing NoSQL databases. The few NoSQL key-value databases used for experimentation and performance comparisons are discussed below.

BerkeleyDB (Oracle 2011), is one of the most popular key-value data stores. It is a library suite that provides the data management features, such as high throughput, low-latency reads, non-blocking writes, high concurrency, data scalability, in-memory caching, ACID transactions, automatic and catastrophic recovery when the application fails, and other features. The BerkeleyDB provides a number of access methods; the B-Tree, Hash,Queue and Recno. The B-Tree access method stores key-value tuples in an ordered manner which support ordered traversals and range searches. BerkeleyDB is included in the benchmarks because of its performance and wide usage for in-memory database applications.

Kyoto Cabinet (FAL Labs 2012) is another library of routines written in the C++ for managing NoSQL database. It also provides APIs for C++, C, Java, Python, Ruby, Perl, and Lua. The data in Kyoto Cabinet is organised as simple data file containing records of key-value pairs. Every key and value, is serial bytes with variable length. Both binary data and character string can be used as a key and a value. Each key must be unique within a database of the Kyoto Cabinet. The records are organized either as hash table or B^+-Tree. Kyoto Cabinet supports various methods for managing data. The data is either completely managed in memory or on disk through an in-memory cache using either a B$^+$-Tree or hash table.

The other key-value store considered in the benchmarking experiment is the LevelDB by Google. LevelDB is a fast key-value storage library which provides an ordered mapping from keys to values. It actually uses Sorted String Table (SSTable), which is a simple abstraction to efficiently store large numbers of key-value pairs while optimizing for high throughput and sequential read/write workloads. It thus, supports forward and backward iteration over the stored data. LevelDB writes are asynchronous by default (i.e. a separate thread writes the data to the persistent storage asynchronously). It therefore, returns after pushing the write process into the operating system memory.

2.3 Concurrent Access of Tree Structured Indexes

In a multi-user multiprocessor environment, the database system must permit several user requests to be processed simultaneously and correctly. It is possible in such situations for one process to be changing data while another is reading that same data. In order to ensure consistency and correctness of the whole system, there is the need to implement some constraints on the data to prevent the processes from interfering with one another. In DBMS, the term *ACID* – (*atomicity*, *consistency*, *isolation*, *durability*) is a desirable characteristics in processing transactions or concurrent queries. A system may not necessarily guarantee all five properties. As shared memory multiprocessors are widespread, it has become increasingly important to examine and implement efficient algorithms for concurrent manipulations in database indexes. There are several proposed protocols for implementing concurrency control in database indexes (Bronson et al. 2010, Srinivasan & Carey 1993, Nurmi & Soisalon-Soininen 1996, Nurmi et al. 1987). Since databases are for multi-user purposes, a non-concurrent index will surely become a major bottleneck in any database management system. Some techniques in the literature relevant to our work are discussed below.

2.3.1 Concurrent Control Techniques

Several techniques, strategies and frameworks have been proposed for concurrency control in tree-based indexes (Agrawal et al. 1987, Srinivasan & Carey 1993, Kung & Robinson 1981). Concurrency control mechanisms are categorised mainly as pessimistic, optimistic or semi-optimistic.

The pessimistic (blocking) concurrency control mechanism involves the use of locks (latches). The lock mechanism limits other users' access to the same record when one user has a lock on the record. The lock prevents (i.e., blocks) other users from changing (and in some cases reading, based on the lock compatibility modes), that record. Lock modes used include; S (traditional shared mode); X (exclusive lock mode); and I (intent - used to establish a lock hierarchy, e.g. SIX- shared with intent exclusive; IS - intent shared; IX - intent exclusive).

Optimistic concurrency control mechanism, also referred to as *commit-time validation or certification*, however, prevents the overwriting of data by using timestamps. In this case, unlike the pessimistic strategies, transactions are validated only after they have reached their commit points. When a transaction reaches its commit point and if it finds that any object that it read has been written by another transaction that committed during its lifetime, it is restarted. Kung & Robinson (1981) concurrency strategy is based on this technique. Another technique described in Agrawal et al. (1987) is the immediate-restart. This is similar to the locking technique. Transactions initially read-lock the objects they read and upgrade to write-locks latter before a write operation. However, if a request for a lock is denied, the transaction restarts immediately after some delay referred to as restart-delay (Agrawal et al. 1987). Since the locking and immediate-restart algorithms are based on dynamic locking, conflicts are detected as they occur, unlike the optimistic algorithms which only detect conflicts at transaction-commit time.

The semi-optimistic on the other hand, combines features of both the optimistic and pessimistic approaches. Some operations are blocked if they cause violations of some rules in some situations. They do not block in other situations while delaying rules checking (if needed) until the transaction end, as is the case with the optimistic approach.

The different concurrency control categories and methods provide different performance. Factors such as the average completion rates (throughput), transaction types mix, level of parallelism, and other factors inform the choice of concurrency control approaches adopted. A knowledge about trade-offs of the various approaches, is therefore a key to the selection of the method that will provide the highest performance. Agrawal et al. (1987) concluded from their study that, a blocking concurrency control algorithm is a better choice than a restart-oriented algorithm in an environment where physical resources are limited. However, in a highly resourced environment with low resource utilization where a greater degree of concurrency is expected, the restart-oriented algorithms provide superior performance.

2.3.2 Relaxed Balancing

In balanced tree-based indexes, the major challenge to concurrency control algorithms is in the process of restoring the tree's invariant (i.e., the balance condition) during or after an update. The balance operations ensure that the tree does not degenerate into linear list and also, search and update operations are performed in logarithmic time based on the number of keys stored in the tree. In a concurrent environment, however, uncoupling the rebalance operations from the update operations may increase the possible amount of concurrency as well as the effective performance of the tree-index (Hanke et al. 1997). This led to the idea of relax balanced trees, in which the balance condition is violated by update operations and then eventually restored. The relaxed balance techniques, effectively uncouple the mutating operations from the restructuring operations by allowing the invariants to be violated but restored by separate rebalancing operations (Nurmi et al. 1987, Nurmi & Soisalon-Soininen 1991, Boyar et al. 1995, Hanke 1998, Larsen 1998).

The rebalance of the tree is delayed in this case. Thus, speeding up and improving the performance of the tree since the rebalancing operation is decoupled from the update operation. The deletion of an internal node, especially in a concurrent tree, is even much more complicated compared to insertion. Deletion of an internal node in a binary tree requires that the successor of that node be located to replace the deleted node. This may involve several other nodes along the search path/nodes along which rebalancing the tree will encompass (Bronson et al. 2010). The idea of relax balance introduced by (Guibas & Sedgewick 1978), was actually first implemented by Kessels (1983) in AVL-Trees. The strategy was also applied to B-Trees in (Nurmi et al. 1987). Nurmi & Soisalon-Soininen applied the idea of relaxed balancing to their version of Red-Black Tree which they called Chromatic Binary search trees.

These separate rebalancing operations for relax balance trees, involve only local changes. Larsen (Larsen 1998) showed that for a relaxed Red-Black Tree the number of restructuring changes after update is bounded by $O(1)$ and the number of color changes by $O(\log\ N)$, where N is the size of the tree. The process of restoring the invariants in relaxed Red-Black Tree also has an amortized constant of $O(1)$ (Larsen 1998).

2.4 Performance Measures

A naive concurrency control algorithm in which the entire tree is locked down from the root whenever an operation is being carried out in the tree, will obviously provide the worst concurrency performance since all operations will be completely serialised. An important conclusion from the relaxed balance literature indicates that, it provides better level of concurrency since the updates are decoupled from the tree restructuring process (Hanke 1999). Such decoupling allows readers to proceed concurrently with updaters without any blocking. However, markings are used to keep track of violation of the invariant of the tree which is eventually settled to keep the tree from degenerating into a list.

The type of locks used in a concurrent algorithm also affect the performance. A protocol which uses exclusive locks for traversals provides a poor performance especially when there are several searches. Though the updates will be serialised to ensure correctness, *readers* will also be blocked. This causes the performance of the algorithm to be slower and also result in a low degree of concurrency. A much efficient approach is the use of read-write locks for traversals. *Readers* and *Updaters* acquire read-locks for traversals. Updaters, however, lock exclusively all nodes which are involved in the actual update once they have been identified.

In a system where there is a high degree of contention, using a *fine grain granularity* locking scheme, in which fewer nodes are locked, provides better performance. In this scheme, only nodes which are involved in the updates are locked, and immediately released. The blocking time is thus greatly minimized. The lock coupling technique with relaxed balance algorithms is adopted in this thesis to implement the concurrent access protocol (thread-safe version) for the O_2-Tree index structure. In this scheme, the tree is allowed to grow and any in-balance is immediately restored in a top-down fashion. Top-down restructuring in a concurrent environment is preferred since it guarantees minimal or no interferences between transactions. Top-down restructuring also ensures that deadlocks are eliminated.

Chapter 3

The O_2-Tree Main memory Index

3.1 Overview and Features

The O_2-Tree is a Red-Black Tree in which the leaf nodes are index blocks, pages or chunks that store the records of "key-value" pairs. The internal nodes contain copies of the keys of the middle "key-value" pairs that split blocks of the leaf nodes when they become full. These internal nodes are formed into a simple binary search tree that is balanced using the Red-Black Tree algorithms. The Red-Black Tree balancing algorithm is less complex than that of the AVL-Tree which has a more strict balance condition and hence, the choice of the Red-Black Tree balance algorithm for the O_2-Tree. The new structure has the advantage that it can be easily reconstructed by reading only the lowest key of each leaf-node and as such explicitly making the index tree persistent is therefore unnecessary. However, the index tree can be made explicitly persistent at the end of each session if desired. The height of the tree is also significantly reduced compared to that of an equivalent Red-Black Tree with same number of keys.

The name "O_2-Tree" is motivated by the fact that the forward and backward leaf-link pointers as well as the general structure of the tree from the root to the leaves indicate an O-like structure.

3.2 Definition

The O_2-Tree is a self-regulating binary tree. Associated with the O_2-Tree is the order of the tree. The order is the maximum number of "key-value" pairs a leaf node can hold. The value associated with the key-value pair corresponds to the data. This could also be a pointer to the location where the record is stored. The key-value is stored in the leaf nodes; whiles the internal nodes are only keys

and are simply binary placeholders or routers to facilitate and guide the tree traversal.
An O_2-Tree of order m ($m \geq 2$, *the minimum degree of the tree*) satisfies the following properties:

1. Every node is either Red or Black. The root is always Black.
2. Every leaf node is colored Black and consists of a block or page that holds key value pairs.
3. If a node is Red, then both its children are Black.
4. For each internal node, all simple paths from the node to descendant leaf-nodes contain the same number of black nodes. Each internal node holds a single value.
5. Leaf-nodes are blocks that have between $\lceil m/2 \rceil$ and m "key-value" pairs.
6. If a tree has a single node, then it must be a leaf which is the root of the tree, and it can have between 1 to m "key-value" records.
7. Leaf nodes are linked together to form a doubly linked list to facilitate range searches.

The advantages of the O_2-Tree are summarized as follows:

1. A considerable low height which provides faster tree traversal.
2. Tree rotations for balancing are minimized and therefore guarantee fast query processing. The greater the order of the tree, the larger the leaf-nodes can grow and thus guarantees minimized splitting which could consequently lead to rotations.
3. The doubly-linked list leaf-nodes provide an easy mechanism to traverse the tree in sorted order for key range searches.
4. Leaf nodes keep their occupancy and thus provide locality of reference for keys.
5. The O_2-Tree also has a superior performance compared to the T-tree since the O_2-Tree makes only one key comparison internally (per node) in order to make a traversal.
6. The tree can easily be rebuilt from the data pages since the internal nodes are simply copies of the minimum key values of leaf nodes. These are equivalent to the middle keys after a split occurs.

3.3 Basic Properties & Features

The internal structure of the O_2-Tree is essentially a Red-Black Tree with one major difference. In the standard Red-Black Tree, when a search key compares equal to the key stored at a node, the search terminates since the key would have been found. In the O_2-Tree, however, the search continues to the right child. Searches in the O_2-Tree always terminate at a leaf node. Since the O_2-Tree is based on the features of the standard Red-Black Tree, we explain the well studied standard Red-Black Tree next for completeness.

3.3.1 Structure of the Red-Black Tree

The Red-Black Tree is a self-regulating balanced binary tree. It uses node color as well as tree rotations to keep the tree balanced. A Red-Black Tree is a binary tree that satisfies the following Red-Black properties:

1. Every node is either Red or Black.
2. The root is Black.
3. Every leaf node (NIL) is Black.
4. If a node is Red, then both its children are Black.
5. For each node, all simple paths from the node to descendant leaf nodes contain the same number of black nodes (Cormen et al. 2009). This is referred to as the Black-Condition.

The Red-condition of a Red-Black Tree states that if a node is Red, then its parent must exist and must be Black. This implies that, no more than half the nodes on such a path can be red, whereas the black condition implies that, there are same number of black nodes (termed the *black-height*) on each such path. It therefore follows from these conditions that, no path from the root of a sub tree to a leaf can be more than twice as long as the other. This height restriction of the tree makes it highly effective for query operations. The height of a Red-Black Tree of N nodes is no more than $2\log_2\ N$ and thus, guarantees worst-case query (insertion, deletion, and searching) times of $O(\log_2\ N)$.

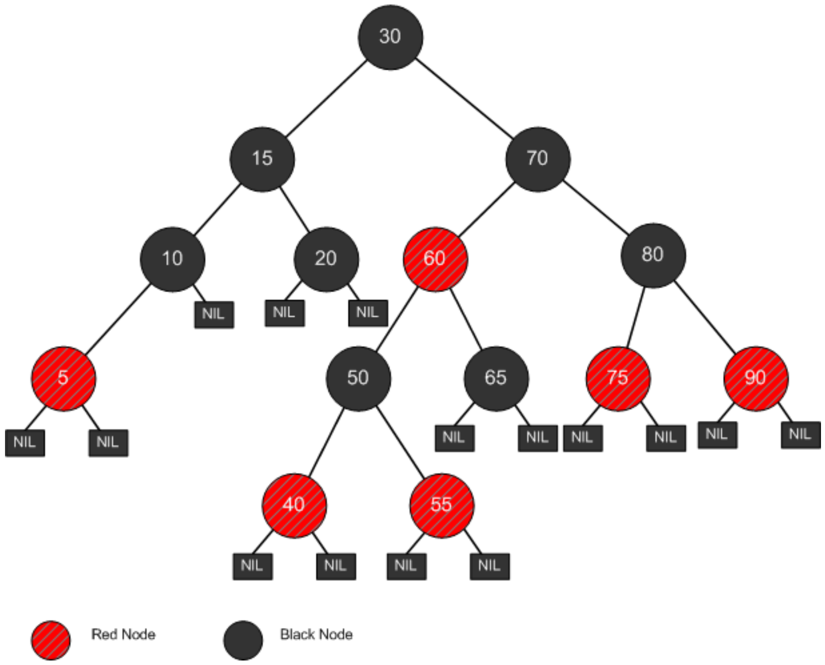


Figure 3.1: Structure of the Red-Black Tree

There are two basic approaches in balancing Red-Black Trees after updates (insert or delete); Top-Down and Bottom-Up approaches. The Top-Down approach involves a series of rotations and color flips as the tree is traversed from the root to the insert/delete node. This is done to ensure that, when the insert/delete operation is done, there will be no need to balance the tree again. Thus, in a single pass, the tree is both balanced and the update operation done. The Bottom-Up approach, however, involves two passes. The first pass involves traversing the tree from the root to the insert/delete location. Once the update operation is done, the tree is then checked for balance to ensure the Red-Black conditions are not violated. If the tree is not balanced, a series of balancing steps are applied from the insert/delete point up to the root tree along the update traversal path. The Top-Down Red-Black Tree is considered in this research since it has a faster performance. It is also a suitable option for concurrency control since the likelihood of deadlocks with up-ward traversals is eliminated. The major drawback of the Red-Black Tree is the height of the tree. Since each node stores a single key, the depth of the tree becomes high and therefore affects query operations. Figure 3.1 shows the structure of a Red-Black Tree after inserting keys $\{5, 10, 30, 70, 15, 20, 60, 50, 40, 65, 55, 75, 80, 90\}$.

The standard Red-Black Tree is utilized to implement the O_2-Tree structure. The fundamental proposition with which the O_2-Tree is implemented is the following:

Proposition 1. *In a Red-Black binary search Tree all black leaf-nodes (NIL) are guaranteed to remain as leaf nodes under single and double rotations.*

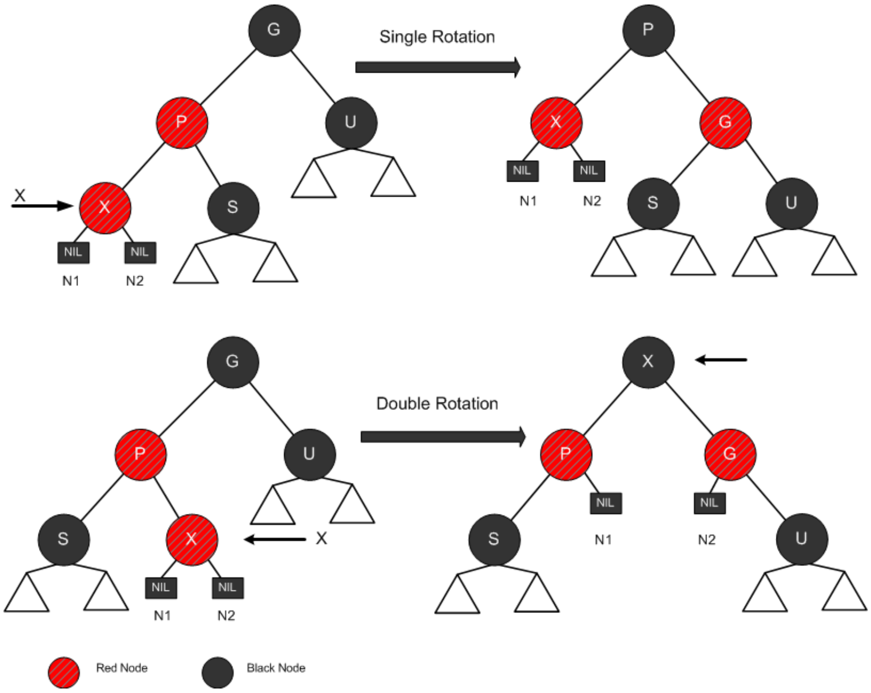


Figure 3.2: Single and double rotation in a Red-Black Tree

In a Red-Black Tree, the two rotations used to restore the tree's invariant after update operations are the single and double rotations. Rotations basically swap (using pointer manipulations) the roles of the parent and the child while maintaining the search order of the binary tree. Single and double rotations are illustrated in Figure 3.2. Only leaf-nodes affected by the rotation are indicated. A single rotation between P and G restores the tree's balance after insertion of X caused a violation. It is evident from the illustration that, all leaf-nodes (NIL) remain leaves even after the single rotation. Similarly, leaf-nodes in a double rotation still remain leaf-nodes. Though node X, has become the new parent of the sub-tree, its leaves *N1* and *N2* still remain leaves but with different parents and the binary search order is still maintained. This leads to a second but important proposition.

Proposition 2: *A Red-Black binary tree with N leaf-nodes will still have N leaf-nodes after either single or double rotations.*

It follows from the above that leaf nodes, formed into data pages, will always remain at the leaf nodes irrespective of the rotations performed on the tree. This fact is also true for other binary search trees.

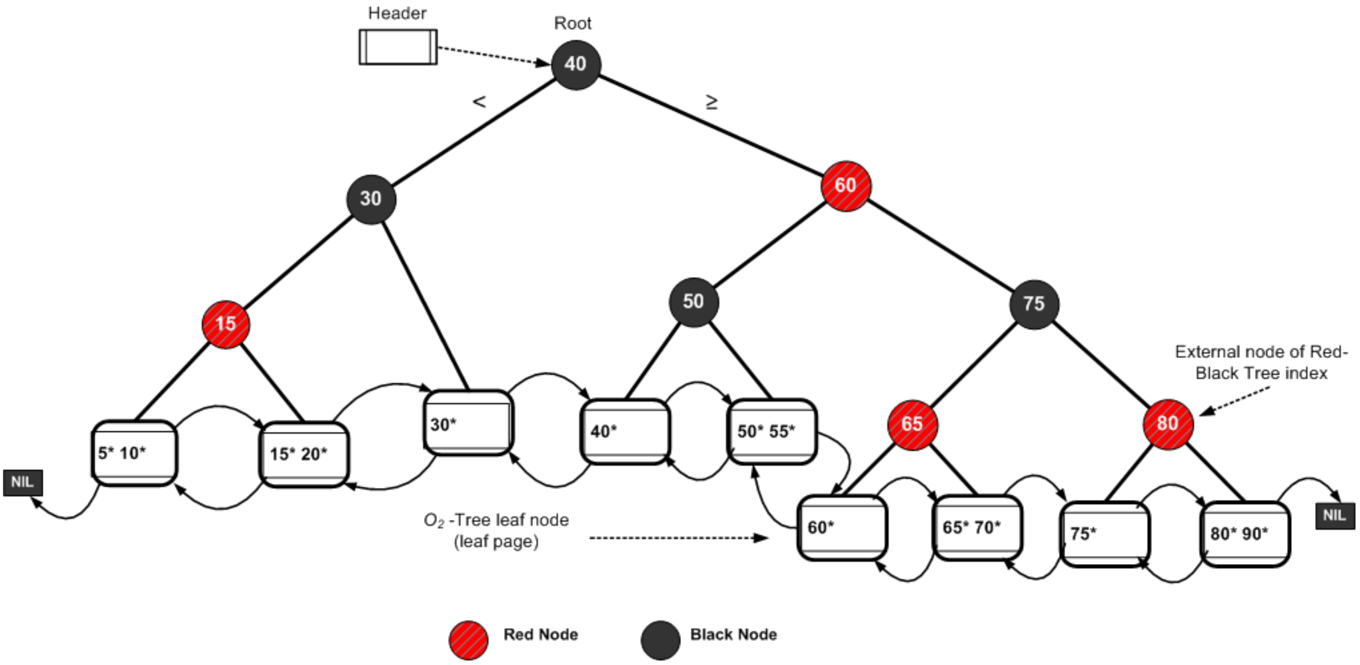


Figure 3.3: General Structure of O_2-Tree

3.3.2 O_2-Tree Structure Implementation & Notations Used

For the purposes of the algorithms presented in Sections 3.4 and beyond, we present the class definitions for the various nodes of the O_2-Tree. Some notations used are also discussed.
A node is specified using a C++ abstract class interface definition as:

```
class O2node
{
    O2node *left
    O2node *right
    int color;
    virtual int nodeType = 0 /* 0=internalNode, 1=leaf */
}
```

Both internal nodes and leaf nodes inherits from the *O2node* abstract class defined above. The class definition for an internal node is illustrated below:

```
class O2InternalNode : public O2node
{
      int key
      int nodeType = 0
}
```

The structure of a leaf node is a page or block and is of the form illustrated by the class definition below. The *record* datatype is a *struct* which defines the “key-value” pairs.

```
class O2leaf : public O2node
{
      Record record[m]  /* m is the order of the tree */
      int nodeType = 1
}
```

Let T denote an O_2-Tree. The root node will be designated as *Root*(T). T is implemented with a dummy *header* $Header(T)$, such that all keys in the tree are greater than the key of the *header*. It also serves as the parent to the root of T. Supposing a type declaration for the *key, value and color* where *color* is enumeration type defined as *enum color* {*Red, Black*}. Let z denote a *node* in T. Then $z.left$ and $z.right$ refer to the left and right child respectively of z. Let *z.parent* denote the parent of z and *z.sibling* refer to the sibling of z such that z and *z.sibling* have the same parent (i.e if z is a left child of its parents then *z.sibling* will be the right child of the parent and vice versa). Also *z.key* is the value of the key in z, if z is an internal node (i.e. $nodeType \neq leaf$). Additionally, $z.key[i]$ and $z.value[i]$ refer to the key and value respectively in the *ith* position of z given that z is a leaf node (i.e., $nodeType = leaf$).

3.4 O_2-Tree Search Algorithms

3.4.1 Exact-Match Search Algorithm

In an exact-match search, a key x is given and the key-value pair $\langle x, val_x \rangle$ is returned if x exist, otherwise a null value is returned. Searching the O_2-Tree is similar to searching in a Red-Black Tree. However, the internal Red-Black Tree structure serves as placeholders to locate the actual key in the leaf node. A significant difference between the O_2-Tree traversals and that of the Red-Black Tree is the fact that all traversals end at the leaf node even when the key is found in an internal node. Having located the leaf node z in which the search key x resides, the binary search function "*binarySearch*(x, z)" is used to locate the "key-value" pair $\langle x, val_x \rangle$ within the node z. The search algorithm is illustrated in Algorithm 3.1.

The process cost of exact-match search is $O(\log_2 N)$ where N is the number of "key-value" pairs stored. This is formally shown in Chapter 4 where we discuss the analytical properties.

Algorithm 3.1: Get(key x, $Header\ T$)

```
Data: $key\ x$
Result: corresponding "$\langle x, val_x \rangle$" pair if $found$, otherwise $null$
begin
    $**\ node\ denotes\ the\ pointer\ to\ the\ current\ node\ during\ traversal\ **$
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    return $binarySearch(x, node)$
    $done$
end
```

3.4.2 Range Search Algorithm (Get Next, Get Previous)

The range search begins from the root just as in the exact-match search query. The search begins by locating the minimum key, x_{min} in a given range e.g., $x_{min} \leq x \leq x_{max}$, where x_{max} is the maximum key in the range. The search returns the range of key-value pairs $\langle x, val_x \rangle$ within the specified range. Once the leaf with key x_{min} is located, the algorithm proceeds with a sequential scan of the leaf node until the last key in the leaf node is reached. If the maximum key x_{max} is still not located, the scan continues to the next leaf following the *forward-link* pointer between the leaf nodes. This continues until the last key x_{max} in the range is found. The *rangeScan* function accomplishes this purpose. The algorithm is illustrated in Algorithm 3.2.

Algorithm 3.2: Range Scan(*key* x_{min}, *key* x_{max}, *Header* T)

Data: *key* x_{min}, *key* x_{max}

Result: corresponding key-value for each existing key in the range

```
begin
    $x \longleftarrow x_{min}$
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    return $rangeScan(x_{min}, x_{max}, node)$
    $*$Range search starting from the min key in the range from the current leaf
    $*$Continue scan in the next leaf if the the maximum key in the range is not encountered
end
```

3.5 Insertion Algorithms

There are two basic approaches to insertion in the O_2-Tree just as is the case with the Red-Black Tree; the *Top-Down* as well as the *Bottom-Up* approaches. The difference in these algorithms lies in the manner in which restructuring of the tree during updates are handled. In the Top-Down approach, rotations and color changes are applied as the tree is been traversed. This is done to ensure that when a new internal node is inserted, the tree will be balanced without any upward traversal. The Bottom-Up approach, however, involves traversal of the tree without any restructuring operations. When a new internal node is inserted which results in the subsequent violation of the invariants, then either double or single rotation as well as well as color changes are applied to restore the invariants. The standard Red-Black Tree restructuring operations are used in both cases to restore the invariants.

To insert the key-value pair $\langle x, val_x \rangle$, the bounding leaf node of x is located using the search algorithm. When the leaf is located and there is room, the new key-value pair $\langle x, val_x \rangle$ is inserted in order by the function $insertInOrder(x, val_x, node)$ into $node$ (the bounding leaf node), based on x. If the leaf is already full, then a split is performed using the function $splitInsert(x, val_x, node)$, where $node$ is the leaf node to be split. After the split, a new internal node is inserted which becomes the parent of the two new leaf nodes. The tree grows in height only when a split of a full leaf node occurs. In the Top-

Down approach the function $preBalance(x, node)$, which apply standard Red-Black Tree Top-Down balance operations is called during the traversal. The Bottom-Up approach, on the other hand does not perform any restructuring during traversal. However, it calls the function $insertFixUp(node)$, to restore the invariants if only a split and subsequent insertion of a new internal node causes a violation. The Bottom-Up and Top-Down approaches are illustrated in Algorithms 3.3 and 3.4 respectively.

3.5.1 Bottom-Up Algorithm

As discussed above, the Bottom-Up algorithm adopts an optimistic approach. In that, it assumes that the insert operation, which we refer to as *Put()*, will not result in any invariant violation. However, if this assumption fails, the *insertFixUp()* (Cormen et al. 2009), method is called to restore the invariants. Algorithms 3.3 shows the Bottom-Up approach as discussed.

3.5.2 Top-Down Algorithm

The objective of the Top-Down algorithm is to ensure that when a new internal node is inserted, there will be no need to move up the tree again. Unlike the Bottom-Up approach, the Top-down algorithm adopts a pessimistic approach, and applies the restructuring method *preBalance()*, to restore the invariant along the search path to the insert location. The *preBalance()* method, applies standard Red-Black Tree top-down rebalancing steps in literature (Weiss 1993), from the root to the bounding leaf node. It thus, guarantees that when a new internal node is inserted its parents *color* will not be Red. The algorithm is illustrated in below in Algorithm 3.4.

The *splitInsert* function is responsible for the split of full leaf nodes. The function basically allocates a new leaf node and redistribute the “key-value” pairs amongst the new leaf and the full leaf node. For instance, the insertion of the “key-value” pair $\langle x, val_x \rangle$ into the full leaf node denoted as $leaf$, will involve the allocation of a new leaf node denoted by $newLeaf$ to the right of $leaf$. Half of the “key-value” pairs in $leaf$ are distributed to $newLeaf$. A new internal node $newNode$ is then created with the lowest key of $newLeaf$ and its left and right pointers set to $leaf$ and $newLeaf$ respectively. The “key-value” pair $\langle x, val_x \rangle$ is then inserted into the appropriate leaf node (either $leaf$ or $newLeaf$). The function returns $true$ if the entire operation succeeds, otherwise it returns $false$. The algorithm is shown in Algorithm 3.5.

Algorithm 3.3: Put($Key\ x$, $value\ val_x$, $Header\ T$)

```
Data: $key\ x,\ value\ val_x$
Result: $true$ for success $false$ otherwise
begin
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    if $!leaf.isfull()$ then
        return $insertInOrder(x, val_x, node)$
    else
        $splitInsert(x, val_x, node)$ /* Algorithm 3.5 */
        /*
        * If parent node is Red
        * Use standard Red-Black Tree bottom-up algorithms
        * */
        if $node.parent.color = Red$ then
            $insertFixUp(node)$
    return $done$
end
```

3.6 Deletion Algorithms

The deletion algorithm which is denoted as $Delete(x)$ removes the key x and its associated value val_x from the index structure. The delete algorithm begins with a traversal from the root to locate the leaf node which contains the key x to be deleted. During the traversal, if key x is found in an internal node, a pointer to the node is stored in the variable denoted as $found$ while the traversal proceed to the leaf node. When x is located in the leaf node, the key-value pair $\langle x, val_x \rangle$ is removed from the leaf node using the $removeKey()$. If the leaf node underflows (i.e results in fewer keys than $\lceil m/2 \rceil$), then "key-value" are *"borrowed"* or appended from the sibling of the leaf to restore the occupancy. This is handled by the function $appendKeyFrom()$. However, if the sibling is exactly

Algorithm 3.4: Put(*Key x*, *value val*$_x$, *Header T*) : Top-Down

```
Data: key x, value val_x
Result: true for success false otherwise
begin
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    if $node.left.color = Red$ and $node.right.color = Red$ then
        /* use standard Red-Black Tree top down algorithms */
        $preBalance(x, node)$
    if $!leaf.isfull()$ then
        return $insertInOrder(x, val_x, node)$
    else
        return $splitInsert(x, val_x, node)$ /* Algorithm 3.5 */
    return done
end
```

half-full (i.e., has exactly $\lceil m/2 \rceil$ keys) then, the leaf and its sibling are merged into the left leaf by the *mergeLeaf()* method. The *mergeLeaf()* function always merges the leaf and its sibling into which ever is the *parent.left* (either leaf or sibling) and deletes the *parent.right*. When leaf nodes are merged, their parent node is also removed from the tree. This may result in the reduction of the height of the tree. If the key x was located in an internal node, the least key in the restored leaf node replaces the original key in the node pointed to by $found$.

There are two approaches for the *Delete()* algorithm just as the *Put()*; the Top-Down and the Bottom-Up approaches. These approaches basically differ only in the way violation of invariants in the tree structure are handled. The Bottom-Up and Top-Down approaches are discussed in the subsequent sections below.

Algorithm 3.5: splitInsert($Key\ x$, $value\ val_x$, $O2node$ node)

Data: $key\ x,\ value\ val_x, O2node\ node$

Result: $true$ for success $false$ otherwise

```
begin
    $newLeaf \longleftarrow new\, leaf()$
    $newNode \longleftarrow new\, internalNode()$
    $midpoint \longleftarrow \frac{m}{2}$
    $where\ m$ is the order of the tree
    $j \longleftarrow 0$
    for $i \leftarrow midpoint$ to $m - 1$ do
        $newLeaf[j] \longleftarrow leaf.remove(i)$
        $j++$
    insert "key, value" into the appropriate leaf
    $newNode.key \longleftarrow newLeaf.key[0]$
    $newLeaf.parent \longleftarrow newNode$
    $leaf.parent \longleftarrow newNode$
    reset forward and backward links of leaf nodes
end
```

3.6.1 Bottom-Up Algorithm

In the Bottom-Up approach, the *Delete*() algorithm proceed as discussed in Section 3.6. However, when a merger of leaf nodes occur, which results in the subsequent removal of the parent node, the invariants of the index structure is violated if the removed node is a *Black* node. The black-height of the Red-Black Tree internal structure is reduced by one for that particular subtree. The *deleteFixUp*() function is thus called to restore the invariants from the leaf node up the tree by applying standard Red-Black Tree rebalance rotations and color changes in the literature (Cormen et al. 2009). Algorithm 3.6 shows the Bottom-Up algorithm.

3.6.2 Top-Down Algorithm

The Top-Down algorithm is similar to the Bottom-Up algorithm except that the algorithm ensures that, the node to be deleted is *Red* by pessimistically 'pushing' Red internal nodes down the tree along the search path. This is accomplished by the function *deleteBalance*() using standard Red-Black Tree top-down rotations and color changes in the literature (Weiss 1993). Leaf underflows are

Algorithm 3.6: Delete(*Key x*, *Header T*) : Bottom-Up

```
Data: key $x$
Result: $true$ for success $false$ otherwise
begin
    /* minKeys ensures that node is at least half full */
    $minKeys \longleftarrow \frac{m}{2}$
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
        if $x = node.key$ then
            /* If key is found in an internal node as well, store
            pointer to that node and continue */
            $found \leftarrow node$
    $done \leftarrow removeKey(x, node)$
    if $done$ and $node.underflow()$ then
        if $node.sibling.keys > minKeys$ then
            /* Borrow from sibling to keep occupancy */
            $done \leftarrow node.appendKeyFrom(sibling)$
        else
            /* Merge leaf and sibling into the left node; delete right node and parent */
            $done \leftarrow mergeLeaf(leaf, sibling)$
            /* Set Parent for merged leaf to the grand */
            if $deletedParent.color = Black$ then
                $node \leftarrow grand$
                /* Apply standard Red-Black Tree bottom-up delete algorithm */
                $deleteFixUp(node)$
    if $found \neq null$ then
        $found.key \leftarrow node.key[0]$
    return $done$
end
```

handled in the same way as the Bottom-Up algorithm which involves borrowing or merging of leaf nodes. The algorithm is presented in Algorithm 3.7.

The actual implementation of the O_2-Tree requires some special cases in the balancing of the tree to ensure that leaf-nodes keep their occupancy. If a leaf-node underflows and its sibling is an internal node, it cannot borrow from this sibling. In this case, we perform an in-order traversal of the sibling to the in-order predecessor leaf-node. Thus, if the leaf-node which caused the underflow is a left child, we borrow from the leftmost leaf node (child) of the sibling and if it is a right child, we borrow from the rightmost leaf node (child) of the sibling. The internal nodes are adjusted appropriately and the current pointer is set to the sibling's child. The child is checked for balance to ensure it keeps its occupancy. Since each leaf node has direct links to sibling leaf nodes as well as other leaf nodes to its left and right, this is easily achieved by referencing either the left or right pointer of the leaf.

Algorithm 3.7: Delete($Key\ x$, $Header\ T$) : Top down

Data: $key\ x$

Result: $true$ for success $false$ otherwise

```
begin
    /* minKeys ensures that node is at least half full */
    $minKeys \longleftarrow \frac{m}{2}$
    $node \longleftarrow root(T)$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node \leftarrow node.left$
        else
            $node \leftarrow node.right$
    if $x = node.key$ then
        $found \longleftarrow node$
    if $node.color = Black$ then
        /* Apply standard Red-Black Tree top-down delete algorithm */
        $deleteBalance(node)$
    $done \longleftarrow removeKey(x, node)$
    if $done$ and $node.underflow()$ then
        if $node.sibling.keys > minKeys$ then
            /* Borrow from sibling to keep occupancy */
            $node.appendKeyFrom(sibling)$
        else
            /* Merge leaf and sibling into the left node; delete right node and parent */
            $mergeLeaf(leaf, sibling)$
            /* Set Parent for merged leaf to the grand */
    if $found \neq null$ then
        $found.key = node.key[0]$
    return $done$
end
```

Chapter 4

Analytical Properties of the O_2-Tree

4.1 Introduction

In this chapter, analysis of the O_2-Tree, with respect to the height, the rate of splits and restructuring operations for that matter, as well as the generalised storage cost model for bucket based tree structures is presented. The height of the tree affects the performance of the index structure for all query operations. The model for the worst-case height as a function of the number of keys within the index structure is thus presented.

The rate at which a split of a leaf node will occur is also considered, given a tree structure with a particular leaf size and keys. Generally, larger bucket sizes result in fewer splits and therefore fewer restructuring operations during updates. However, the intra-node search times increases as there are more keys and comparisons to make per node. But this overhead cost is minimal as compared to the restructuring operation.

4.2 Worst-Case Height Analysis

The worst case height (h) of a Red-Black Tree (Cormen et al. 2009) is given by :

$$h = 2\log_2(N+1) \; where \; N \; is \; the \; number \; of \; keys$$

The time complexity for algorithms of Red-Black Tree operations is thus O($\log_2 N$). This makes the tree very efficient for query operations. The O_2-Tree, thus builds on this efficiency to provide faster query operations. The analysis is provided below:

Let N =*key-value* pairs stored, then the maximum number of leaf nodes or blocks in an O_2-Tree of order m is given by

$$M = \frac{N}{\left\lceil \frac{m}{2} \right\rceil}$$

The number of external nodes of the Red-Black Tree internal index structure is $M/2$.
Since every external node of the Red-Black Tree has 2 block pointers, the total number of nodes in the Red-Black Tree index is given by:

$$\begin{aligned} &= \frac{M}{2} + \frac{M}{2} - 1 \\ &= M - 1 \end{aligned}$$

Since, the worst case height of a Red-Black Tree is $2\log_2(N+1)$, then the worst case height, h, of the internal Red-Black Tree index structure of the O_2-Tree is given by

$$\begin{aligned} h &= 2\log_2(M - 1 + 1) \\ &= 2\log_2(M) \\ &= 2\log_2\left(\frac{N}{\left\lceil \frac{m}{2} \right\rceil}\right) \\ &= 2(\log_2 N - \log_2 \left\lceil \frac{m}{2} \right\rceil) \\ h &\approx 2(\log_2 N - \log_2\ m) \qquad (4.1) \end{aligned}$$

It follows from equation (4.1) that the worst-case cost of query operations(*Get(), Put(), Delete()*) on the O_2-Tree is bounded by O($\log_2\ N$).

4.3 Number of Nodes

Given an O_2-Tree of N keys, the estimated maximum number of leaf pages or blocks is $N/\lceil m/2 \rceil$ where m is the order of the tree. This is the case if each leaf page is only half full. Also, if each page is completely full, then the number of leaf pages is given by N/m. The respective total number of internal nodes (making up the Red-Black Tree index) in the O_2-Tree, given by n in both cases is $2N/m - 1$ and $N/m - 1$. Therefore, the total number of internal nodes in an O_2-Tree is no more than $2N/m$ and no less than N/m i.e ($N/m \leq n \leq 2N/m$). A larger bucket size m will therefore result in fewer internal nodes and may subsequently result in faster traversal.

4.4 Data Comparison

The number of data comparison during traversal is proportional to the number of internal nodes visited plus the number of comparisons within a leaf page. Since the worst case height (h) of the O_2-Tree is given by $2\log_2 N/\lceil m/2\rceil$ from equation (4.1), and also, since the internal traversal requires only one data comparison, it therefore implies that the number of data comparison during the internal Red-Black Tree traversal is equivalent to the height h of the tree. Also, at the leaf page, the worst case data comparison to locate a key using binary search is given by $\log_2 m$. This is the case when the leaf page is completely full and the search key is the largest in the page. Consequently, the total number of data comparison to locate a key is :

$$\begin{aligned}
&= h + \log_2 m \\
&= 2\log_2 \frac{N}{\left\lceil \frac{m}{2} \right\rceil} + \log_2 m \\
&\approx 2(\log_2 N - \log_2 m) + \log_2 m \\
&\approx 2\log_2 N + \log_2 m \qquad (4.2)
\end{aligned}$$

4.5 Average Number And Probability Of Splits

How often do node splits occur in an O_2-Tree? In an O_2-Tree, new internal nodes are only created when there is a split of a full leaf node (leaf overflow). A similar analysis to that in (Drozdek 2001) for B^+-Trees is discussed below.

It follows that during the construction of an O_2-Tree of height h and n number of internal nodes, there are n splits. More so, an O_2-Tree of n internal nodes will have at least:

$$(n+1)\left\lceil \frac{m}{2} \right\rceil \textit{ keys.}$$

The rate of splits in the tree with respect to the number of keys is given by the ratio of the number of internal nodes (resulting from splits) to the total number of keys. Therefore, the rate of splits

$$\begin{aligned}
&= \frac{n}{(n+1)\left\lceil \frac{m}{2} \right\rceil} \qquad (4.3)\\
&= \frac{1}{\frac{(n+1)}{n}\left\lceil \frac{m}{2} \right\rceil}\\
&= \frac{1}{(1+\frac{1}{n})\left\lceil \frac{m}{2} \right\rceil}
\end{aligned}$$

Now we observe that

$$\lim_{n\to\infty}(1+\frac{1}{n}) = 1$$

Therefore, as n increases, the average probability of split is given *by* :

$$= \frac{1}{\left\lceil \frac{m}{2} \right\rceil} \qquad (4.4)$$

Therefore the larger the value of m, the order of the tree, the lower the probability and the less frequent splits will occur. For instance, given a capacity of $m = 512$ keys the probability of splits will be 0.0039, while a capacity of $m = 1000$ will result in 0.002 probability.

4.6 Storage Utilization

The expected storage utilization, from the fact that it grows and shrinks from block splitting and merging respectively, is $O(\ln 2)$. It can easily be shown using a similar approach as in the approximate storage utilisation of B-Trees (Leung 1984). Let N be the total number of keys in the tree and let n denote the number of index blocks at the leaves of the tree. Let m be the order of the tree. Each leaf block has at least $\lceil m/2 \rceil$ and m keys. The storage utilization denoted by μ is the total number of keys stored divided by the total storage capacity of all the nodes.

$$\mu = \frac{N}{m \times n}$$

The expected storage utilisation is

$$E(\mu) = \frac{N}{m}E\left(\frac{1}{n}\right)$$

To evaluate $E(1/n)$ we note that n lies in the interval $[N/m, 2N/m]$. By approximating the distribution as a continuous random rectangular distribution over the interval we have

$$E(\mu) \approx \frac{N}{m} \int_{N/m}^{2N/m} \frac{dn}{n} = \ln 2$$

Chapter 5

The O_2-Tree Concurrency Control

5.1 Overview

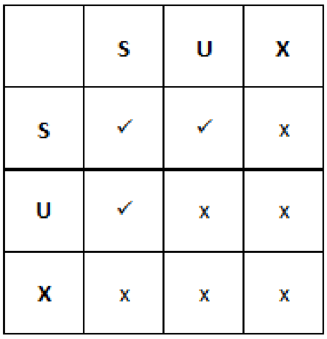
Concurrency control mechanism is required to enable the index structures scale in multi-user shared memory environment. A non-concurrent index structure will surely be a bottleneck in any database transaction processing. Locks and latches are a common means of implementing concurrency in index structures. Concurrent control algorithm for relaxed balance tree implementations based on fine-grained read-write locks provide good scalability for tree-indexes.

Optimistic concurrency control (OCC) schemes using version numbers are attractive for concurrency control especially for in-memory index. They naturally allow *readers* (i.e., thread or user process requiring read access to the index structure) to proceed without locks, and thus avoid the coherence contention inherent in read-write locks. The readers simple read version numbers updated by *writers/updaters* (i.e., thread or user process requiring write access - *Put() / Delete()*) to detect concurrent mutation. Since, readers assume that no mutation will occur during a critical region, they retry if that assumption fails i.e if a mutation occurs. This could however, lead to spurious retries and wasted work.

Software transactional memory (STM) provides a generic implementation of optimistic concurrency control. STM groups shared-memory operations into transactions that appear to succeed or fail atomically. The aim of STM is to deliver a simple parallel programming at an acceptable performance. However, performance gains and scalability are amongst the most important goals of a data structure library, and not just simplicity (Bronson et al. 2010). More so, its been observed that, in practice, STM systems also suffer a performance hit relative to fine-grained lock-based systems on small numbers of processors (1 to 4 depending on the application) (Bronson et al. 2010).

A concurrent access control mechanism, which is based on the lock coupling technique (also known as *hand-over-hand locking*), for the O_2-Tree index is presented. This technique is similar to the Bayer–Schkolnick (Bayer & Schkolnick 1988) concurrency algorithm. In this technique, a *reader* or *updater* gets a lock on a node while requesting for a lock on the next node along the search path. It only releases the lock on the current node if the request is granted for the other node. The scheme also adopts the relaxed balance Red-Black Tree algorithm by Larsen (1998). However, the index structure is managed such that the number of restructuring steps after mutation operations are further reduced. To achieve maximum concurrency, the O_2-Tree thread-safe algorithms are implemented with *page-level* or *node-level* locking. In this case, each node can be locked and unlocked. This simple fine-grained lock-coupling technique ensures that multiple threads can proceed concurrently as long as they don't interfere with each other at the same node.

Locks are used to prevent concurrent interleaved operations from interfering with one another. A tree with fewer locks provides greater concurrency and scalability in high workloads. The thread-safe algorithm presented, is implemented with three types of locks; the shared (S-lock), exclusive (X-lock) and Upgrade (U-lock) locks. The S-locks are compatible with themselves as well as the U-locks but not with X-locks. The U-locks are intermediate read-write locks between the S-locks and the X-locks. They allow for the atomic upgrade of S-locks into X-locks by an updater. X-locks are incompatible with themselves and all other locks. Several user processes can *S-lock* a node at the same time, whereas, only one process can *U-lock* a node at a time but can coexist with other processes with *S-lock* on the same node. The *X-lock* on the other hand ensures exclusive access to a node and cannot coexist with any other process locks. The lock compatibility matrix is as shown below.



	S	U	X
S	✓	✓	x
U	✓	x	x
X	x	x	x

Figure 5.1: Lock Compatibility Matrix

The concurrency control mechanism using fine-grain locks which is adopted allows multiple *readers* to proceed simultaneously without blocking other node traversals except for leaf nodes where an *updater* needs to hold a lock. This further reduces the lock overhead which might have resulted due to blocking of concurrent interleaved query operations. The scheme ensures further performance gains by using the following mechanisms;

- search operations are interleaved using the hand-over-hand locking technique;
- mutations perform rebalancing separately which encompasses smaller fixed sized atomic regions.

5.2 Concurrent Index Restructuring

The major challenge with the concurrent access of the index, especially with updates, is how to restore the balance of the tree after an update. The standard strict top-down algorithm limits the amount of concurrency of the index and does not scale well, since every update will proceed with several top-down balancing steps before exiting. The process of restoring the tree invariant therefore becomes a bottleneck for such concurrent tree implementations. The mutating operations must acquire not only locks to guarantee the atomicity of their operations, but also locks to guarantee that no other mutation affects the balance condition of any nodes or the sub-tree that will be involved in the restoration process. To overcome this challenge, the idea of relax balance trees is adopted. The relax balance tree algorithm, proposed by Larsen (1998), is implemented for the concurrent protocol. Structural changes after an update in this relaxed balanced condition is bounded by $O(1)$ and the number of color changes by $O(\log\ N)$ (Larsen 1998). This algorithm is preferred over that of the Chromatic tree (Nurmi & Soisalon-Soininen 1996) since it has a reduced number of cases for handling structural conflicts.

The relief algorithm ensures that *updaters* perform no re-structuring activities. In this case, only a few number of nodes are locked at a time. Node colors are also characterised by weights. A Red node has a weight of 0, a Black node has a weight of 1 and an over-weight node has a weight greater than 1. When an update is made, the updater only adjust the weights of the affected nodes appropriately.

The entire process of handling conflicts in the tree is handled by a rebalancing process which is denoted as *rebalancer*(). The *rebalancer*() may be run in the background especially during off-peak times when there is little activity on the index. The *rebalancer*() locates nodes in the tree with *overweight* ($weight > 1$) as well as *red-red*(i.e., consecutive red nodes) conflicts and resolves them appropriately. Several techniques have been proposed in the literature with regards to the way in which the rebalancing process is invoked. One suggestion is to traverse the data structure randomly in search of rebalancing request (Nurmi & Soisalon-Soininen 1991). This strategy does not scale well since the *rebalancer*() will be involved in spurious traversals. Another proposal is to use a *problem queue* (Boyar et al. 1995, Boyar & Larsen 1994). The problem queue approach is adopted in this implementation since a random traversal by the *rebalancer*() may result in several interference with other query processes and subsequent degradation in the performance of the index. In this approach, when an imbalance situation is created in the tree, a pointer to the parent of the node involved is placed in the *problem queue*. The *rebalancer*() continuously reads the queue and purposefully proceeds to the exact location to fix the imbalance. The tree is balanced if the *problem queue* is empty.

Updaters append requests to the tail of the *problem queue* while, the *rebalancer*() pop these request from the head of the queue. To prevent interference and guarantee consistency between *updaters* and the *rebalancer*() the problem queue is implemented to allow for maximum concurrency with locks similar to that in (Hanke 1999). This allows for simultaneous *push()* and *pop*() operations such that neither the *rebalancer()* nor user *updater* processes are blocked. Whiles *updater* appends requests to the *tail* of the *problem queue*, the *rebalancer*() pops these request from the *head* of the queue. This prevents interference and guarantees consistency between *updaters* and the *rebalancer*() process. The relaxed balancing algorithm is illustrated in Appendix A.

Before presenting the algorithm for the concurrent operations, the same notations discussed in Section 3.3.2 of Chapter 3 are adopted. Additionally, let the three locks used be denoted as *rlock*(), *wlock*(), *and xlock*(), similar to those used by (Nurmi & Soisalon-Soininen 1996). Several user processes can hold *rlock*() on a node at the same time, whereas, only one process can hold a *wlock*() on a node at a time but can coexist with other processes that hold *rlock*() on the same node. *xlock*() held on a node on the other hand ensures exclusive access to that node and cannot coexist with any other process. Request to hold an *rlock*, *wlock*, *and xlock* are handled by the respective functions *rlock*(), *wlock*(), *and xlock*(). To unlock a node that a process holds any of *rlock*, *wlock*, *or xlock*, the *unlock()* function is invoked.

5.3 Thread-Safe Search Algorithm

The Search function denoted as *Get*(*key x*) returns the exact-match key-value pair $\langle x, val_x \rangle$ associated with the key x from the O_2-Tree index T, if x exist. Otherwise a null value is returned. The search traverses nodes from the root by using lock-coupling with *rlocks* until the the leaf page with the given x is found. This prevents mutation operations from changing data whiles the current thread is reading that same data (thus preventing data races). Once the bounding leaf-page z for the search key x is located, the binary search function $binarySearch(x, z)$ is used to locate the "key-value" pair $\langle x, val_x\rangle$ from z. The thread-safe search algorithm is given in Algorithm 5.1.

The algorithm for the range search is the same as the above except that the reader makes request to hold an *rlock()* on adjacent leaf nodes that need to be scanned for keys within the required range $(\langle x_{min} \leq x \leq x_{max}\rangle)$. It follows a similar approach as discussed in Section 3.4.2 of Chapter 3.

Algorithm 5.1: Thread-Safe $Get(key\ x, Header\ T)$

```
Data: $key\ x$
Result: corresponding $\langle x, val_x\rangle$ pair if $found$, otherwise $null$
begin
    ** node is the current node pointer for traversal **;
    $node \longleftarrow root(T)$
    $node.rlock()$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node.left.rlock()$
            $node.unlock()$
            $node \leftarrow node.left$
        else
            $node.right.rlock()$
            $node.unlock()$
            $node \leftarrow node.right$
    $done \leftarrow binarySearch(x, node)$
    $node.unlock()$
    return $done$
end
```

5.4 Thread-Safe Insert and Update Algorithm

The $Put()$ operation proceeds with a traversal similar to that of the $Get()$. However, a much more elegant approach is to use a *wlock*, which allows several *rlocks* by other threads on the resource but not another *wlock* or *xlock*. This allows for interleaved $Get()$ operations to overtake *writer* operations if necessary and not be blocked. To insert the key-value pair $\langle x, val_x\rangle$, the leaf page (denote this as *node*) in which the key-value pair belongs is first located. When the page is located, it is locked exclusively and if there is room, the new key-value pair $\langle x, val_x\rangle$ is inserted in order by the function *insertInOrder*$(x, val_x, node)$ into the page based on the value of the key x. If the page is already full, then a split is performed using the function *splitInsert*$(x, val_x, node)$, where *node* is the leaf node to be split. A split basically allocates a new page in the in-memory cache pool and assigns half of the key-value pairs $\langle x, val_x \rangle$ from the overflow page to the new page. The *previous* and *next* page pointers are updated appropriately. After the split, a new internal node is inserted which becomes the parent of the two page blocks. The tree may grow in height only when a page (leaf-node) overflows. The thread-safe algorithm is presented below as Algorithm 5.2.

5.5 Thread-Safe Delete Algorithm

The delete algorithm follows a similar pattern as the insert algorithm. However, the delete may result in page underflow. In this case, either key-value pairs $\langle x, val_x \rangle$ are borrowed from adjacent pages (*previous or next pages*) using the *appendKeyFrom()* function or pages are merged (using the *mergeLeaf()* function) with the leaf-node that underflowed and the other page is deallocated or released into the cache pool(memory pool). The *sibling* from which a key is borrowed to keep the occupancy of the underflowed node or page is the one which has the same parent as the underflowed node. However, if the *sibling* node is not a leaf, then the $\langle x, val_x \rangle$ pair is borrowed from the previous or next leaf node using the *backward-link or forward-link* respectively. If the underflowed leaf-page is a left child of its parent, the *forward-link* is used. Otherwise, the *backward-link* is used. A merger of pages also results in the subsequent removal of the parent node. If this results in the violation of the invariant condition, the grandparent of the new parent node is pushed to the problem queue. The thread-safe delete algorithm is given in Algorithm 5.3.

5.6 Correctness

5.6.1 Deadlock Freedom

The concurrent protocol presented guarantees deadlock free access by all user processes or threads. This ensures correctness of all transactions. The algorithm does define lock order for traversals such that all request are made in the same top-down approach. For instance, a thread that holds no locks may request a lock on any node, and a thread that has already acquired one or more locks may only request a lock on one of the children of the node most recently locked. Each critical region in a mutation operation preserves the binary search tree property.

Let p and n denote nodes in the critical section of a mutation operation, where p is the parent of n. Consider a rotation which changes *n.left* or *n.right* to point to p. This will require locks on both p, n, and the old parent (denoted by p_1) of p, if it exists. It therefore implies that, it is not possible for two threads T_1 and T_2 to hold locks on nodes p_1 and n, respectively, and for T_1 to observe that p is a child of p_1 while T_2 observes that p is a child of n. Despite concurrent changes to the tree structure, this protocol is therefore deadlock free , due to the defined lock order which is enforced in the index traversal. Locks are acquired only on children of the previously locked node and each child traversal is protected by a lock held by that same thread. Also there is no deadlock cycle loop where thread T_1 waits on a lock by T_2 whiles T_2 waits on a lock by T_1. Because no such loop exists in the tree structure, and all parent-child relationships are protected by the required locks to make them consistent, there is no such deadlock cycle in the concurrent protocol.

5.6.2 Linearisability

In order for the algorithms to behave as expected in a concurrent environment, they require that their implementations be linearisable, meaning that they appear to occur atomically at some point between when they are invoked and when they return (Herlihy & Wing 1990). This implies that operations for a particular key produce results consistent with sequential operations on the tree-index structure. Atomicity and ordering is trivially provided between *Put()* and *Delete()* operations by the *wlock* hand-over-hand tree traversal. This ensures that no two of such operations overtake or interfere with each other. It is not possible for two threads , T_1 and T_2 to lock the same node resource simultaneously. This ensures that the updates are serialised. Since a mutation operation only changes child and parent links after acquiring all of the required locks in the critical region, atomicity of the transaction is guaranteed. Eventually, the overall *correctness* of the concurrent algorithms on the tree structure is ensured.

Algorithm 5.2: Thead-Safe Put(*Key x*, *value* val_x, *Header T*)

```
Data: key x, value $val_x$
Result: true for success false otherwise
begin
    ** node is the current node pointer for traversal **
    $parent \longleftarrow header$
    $node \longleftarrow root(T)$
    $parent.wlock()$
    $node.wlock()$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node.left.wlock()$
            $parent.unlock()$
            $parent \leftarrow node$
            $node \leftarrow node.left$
        else
            $node.right.wlock()$
            $parent.unlock()$
            $parent \leftarrow node$
            $node \leftarrow node.right$
    /* Upgrade both node and parent locks to xlock */
    $parent.xlock()$
    $node.xlock()$
    if $!leaf.isfull()$ then
        $done \leftarrow insertInOrder(x, val_x, node)$
    else
        $done \leftarrow splitInsert(x, val_x, node)$
        Update problem queue if invariant is violated
    $parent.unlock()$
    $node.unlock()$
    return done
end
```

Algorithm 5.3: Thread-Safe Delete($Key\ x, Header\ T$)

```
Data: key x
Result: true for success false otherwise
begin
    /* minKeys ensures that node is at least half full */
    $minKeys \longleftarrow \frac{m}{2}$
    $parent \longleftarrow header$
    $node \longleftarrow root(T)$
    $parent.wlock()$
    $node.wlock()$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node.left.wlock()$
            $parent.unlock()$
            $parent \leftarrow node$
            $node \leftarrow node.left$
        else
            $node.right.wlock()$
            $parent.unlock()$
            $parent \leftarrow node$
            $node \leftarrow node.right$
    /* Upgrade both node and parent locks to xlock */
    $parent.xlock()$
    $node.xlock()$
    $done \leftarrow removeKey(x, node)$
    if $done$ and $node.underflow()$ then
        $sibling.xlock()$
        if $node.sibling.keys > minKeys$ then
            /* Borrow from sibling to keep occupancy */
            $done \leftarrow node.appendKeyFrom(sibling)$
        else
            /* Merge leaf and sibling into the left node;
            release page block and delete parent node */ ;
            $done \leftarrow mergeLeaf(leaf, sibling)$
        $sibling.unlock()$
    $parent.unlock()$
    $node.unlock()$
    return $done$
end
```

Chapter 6

Persistence, Fault Tolerance And Recovery

6.1 Overview

A major concern with main-memory databases and and their memory resident indexes is the guarantee that the database will be persistent, recoverable and fault-tolerant. Since main memory is volatile, it is essential that one adopts recovery techniques for the entire database as well as the index, such that the mechanism to restore the database to a consistent and operational state is not expensive, time consuming and that it constitutes the least bottleneck in the overall performance of the database. These recovery mechanism are essential to ensure that the database and its associated index can be quickly repaired and restored into a *usable state* from which normal processing can resume. The faster the index can be restored or recovered, the less impact it will have on the performance of the entire database recovery process. Generally, transactional logging, checkpointing and reloading techniques are employed.

6.1.1 Transactional Logging

Logging maintains a log of transactions that occur during normal execution. Log information may be recorded at two different levels of abstraction in main memory databases (MMDB), either physical logging and logical logging. In physical logging, the state of the modified database and the physical information (such as page id and offset) is recorded for each record update, whereas logical logging records the *state transition* of the database and the logical location affected by the update. Though logical logging results in fewer logs, measures must be taken to ensure that each committed transaction is executed exactly once which adds complexity to the database both during normal processing and during recovery (Gupta & Chen 1999).

The write-head logging (WAL) is the most popular and widely used type of logging in database management systems (DBMS). In this protocol, the log data is written to non-volatile memory before the update (transaction) can be made to the database. The protocol provides atomicity and durability (two of the ACID - atomicity, consistency, isolation, durability, properties), in a DBMS and thus ensures that the effect of uncommitted transactions can be rolled back. The logs, therefore, ensure that all committed transactions are stable (transaction durability). When there is failure, the logs are 'replayed' to bring the database to the state of the last logs.

6.1.2 Checkpointing

Checkpointing takes a snapshot of the database periodically and copies it onto persistent storage for backup purposes. It reduces the amount of work that must be done during crash recovery. Since, checkpointing takes snapshot of the database during normal processing, the algorithms must be designed such that there is minimal interference with normal transaction processing. MMDB checkpointing approaches are categorized into :

- *non-fuzzy checkpoint*,
- *fuzzy checkpoint*, and
- *log-driven checkpoint*

The non-fuzzy checkpointing approach ensures consistency of the database snapshot by obtaining a lock on the data items before taking the snapshot. This consistency, can either be transaction or action oriented. Action consistency is better and results in less interference than transaction consistency (Gupta & Chen 1999). The fuzzy checkpointing, on the other hand, does not secure any locks before taking a snapshot. The approach, however, writes a record to the log file after a checkpoint to synchronize with normal transactions. More so, the database is generally dumped in segments. In the log-driven checkpointing, a new dump is generated from previous dumps rather than dumping from the main-memory database. This is achieved by applying logs to the previous dumps. This technique is less efficient than flushing the dirty pages directly to persistent storage.

6.1.3 Reloading

Reloading is one of the most important parts of the recovery process. This process must ensure that the database is in a consistent state after a system crash. After a system failure, the persistent copy of the database is reloaded into main memory. The MMDB is then restored to a consistent state by applying information in the undo and redo logs to the reloaded last checkpointed copy of the database in-memory. Reloading can either be done in a simple manner in which case, normal operation of the database is halted until the entire database is reloaded in memory to resume normal processing. In concurrent reloading approach however, the reload activities and transaction processing are performed in parallel. More specifically, the database is reloaded on demand on a per-partition basis. This strategy, therefore eliminates the need to wait for all the database to be loaded in memory.

6.2 Persistence And Fault Tolerance

The O_2-Tree index structure is implemented as a persistent key-value store by reading and writing the leaf-nodes using an in-memory cache pool in which the leaf nodes of blocks of key-values pairs are managed by the *BerkeleyDB Mpoolfile* (**BDB** for short) subsystem only. The internal nodes provide simply binary place holders for fast tree index traversal. New internal nodes are only added when leaf-nodes split as a result of overflows. The index tree may grow in height after a split of a leaf-block. The reverse occurs when there is an underflow resulting in the merging of leaf-nodes and the subsequent removal of the parent of the nodes that are merged.

The BDB Memory Pool subsystem is a general-purpose shared memory buffer pool which can be used by applications requiring page-oriented, shared and cached file access. The BDB memory pool is a memory cache shared among any number of threads of control (Oracle 2011). The O_2-Tree in-memory key-value store ensures persistence by organising the leaf-pages through the in-memory cache pool.

The *DB_MPOOLFILE* class provides the APIs for reading and writing pages to the in-memory buffer cache. The $DB\_MPOOLFILE->get()$ fetches a page from the cache, whereas $DB\_MPOOLFILE->put()$ returns that page back to the cache pool. The pages within the cache are replaced by *LRU* (least-recently-used) policy, with each new page replacing the page that has been unused the longest. When the specified cache size is full, the LRU scheme ensures that dirty pages are written to the underlying backing file before being discarded from the pool. In addition, $DB\_MPOOLFILE->sync()$ method is used to flush all dirty pages within the pool to the backing file. The API inherently provides a transaction subsystem which allows a group of database changes to be treated atomically so that either all of the changes are done, or none of the changes are done at all to ensure data consistency. The transaction model which is inherent in the memory pool subsystem is supported by a write-ahead logging to keep track of all committed transactions and to ensure data durability

(Oracle 2011). During normal processing, a separate thread periodically flushes dirty pages to the persistent store asynchronously to ensure durability of transactions.

Besides storing the leaf-pages by a background process, such that the entire Red-Black Tree structure can be rebuilt from the minimum key values of each leaf-page, the internal-nodes of the O_2-Tree that form the Red-Black Tree can be occasionally dumped onto disk during checkpoint or after each session of usage. Just before a session starts and as part of the initialisation phase, the Red-Black Tree can be restored from persistent store.

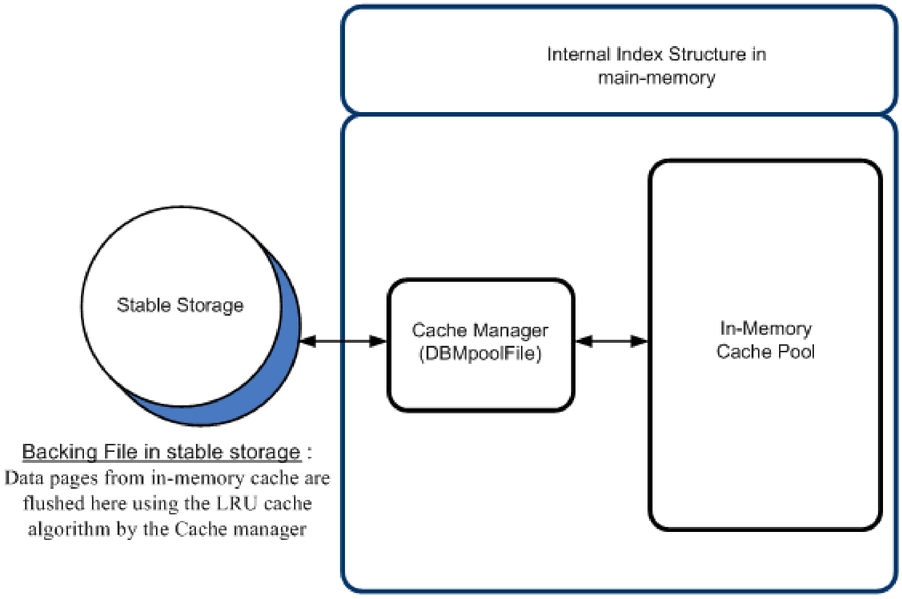

Internal Index Structure in main-memory
Stable Storage
Cache Manager (DBMpoolFile)
In-Memory Cache Pool
Backing File in stable storage :
Data pages from in-memory cache are flushed here using the LRU cache algorithm by the Cache manager

Figure 6.1: Schematic for O_2-Tree Data Persistence Through In-Memory Cache Design

6.3 Recovery

Since main memory is volatile, it is essential to adopt recovery techniques for the entire database as well as the index, such that restoring will not be a bottleneck on the overall performance of the database. These recovery mechanisms are essential to ensure that the database can be repaired and restored into a *usable state* from which normal processing can resume should there be any errors or faults. Since disk (persistent storage) reads are expensive, reducing the disk overhead during recovery from persistent dumps is very crucial in designing the recovery techniques for in-memory databases.

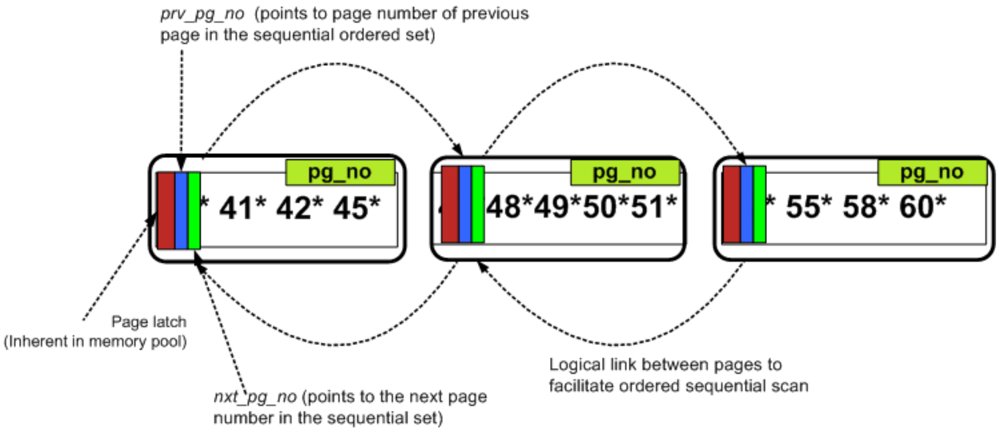

prv_pg_no (points to page number of previous page in the sequential ordered set)
pg_no
* 41* 42* 45*
pg_no
48*49*50*51*
pg_no
* 55* 58* 60*
Page latch (Inherent in memory pool)
nxt_pg_no (points to the next page number in the sequential set)
Logical link between pages to facilitate ordered sequential scan

Figure 6.2: Schematic of O_2-Tree key-value Store Page Structure

6.3.1 Crash Recovery

A crash recovery is performed when the volatile in-memory index is lost due to system failure. This involves rebuilding the index from the database persistent dumps. The O_2-Tree persistent store provides an efficient and simply approach for index recovery. Rebuilding the index structure of the O_2-Tree from persistent store, unlike B^+-Tree, T-Tree structures, requires reading only the first key values in each of the leaf-page. This eliminates the performance bottleneck of traversing the entire "key-value" pairs of data in the leaf-pages and therefore ensures a quick means of recovery should there be a failure. It also reduces the disk reads (Input/Output I/O). In systems where the index data is too large to fit into available memory, pages could be paged-in and paged-out of the in-memory cache using a cache replacement policy such as the least recently-used protocol. The key advantage of the proposed O_2-Tree with regards to recovery is the fact that the tree index can easily and quickly be reconstructed from reading just the first key-value pair of each page from the backing store. In this way, the entire index can be built at a very fast rate and a such the database recovery time is greatly reduced.

6.3.2 Normal Recovery

When a usage session ends normally or during periodic checkpoints, the entire internal index structure could be dumped to a persistent store. This technique further minimizes the impact of reconstructing the entire index from the leaf pages.

To save the internal index structure at the end of a usage session or during a checkpoint, the structure is dumped to a persistent storage in a *depth-first pre-order traversal array sequence*. The use of the

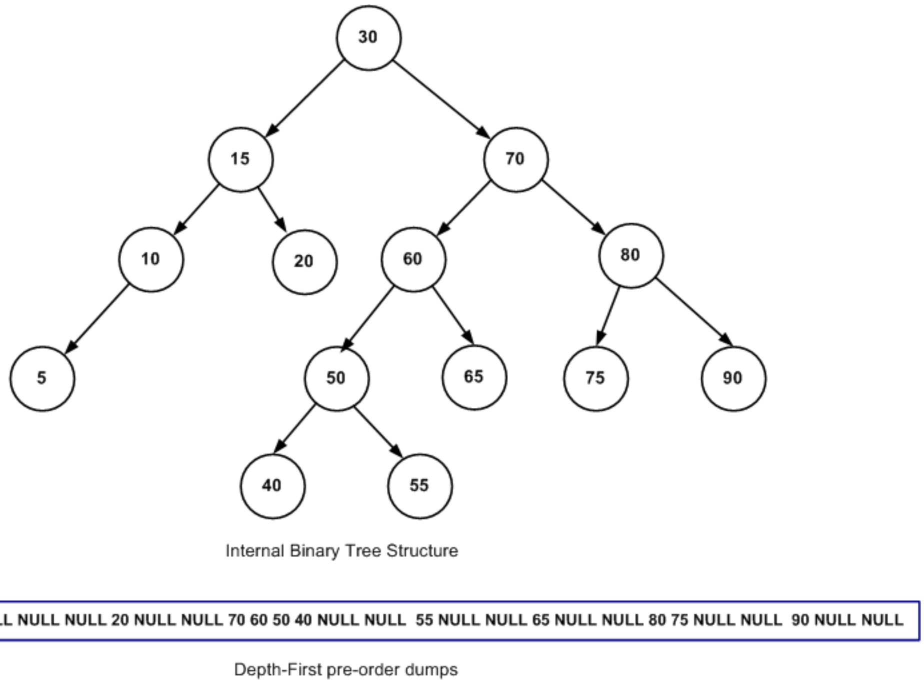

30
15
70
10
20
60
80
5
50
65
75
90
40
55
Internal Binary Tree Structure
30 15 10 5 NULL NULL NULL 20 NULL NULL 70 60 50 40 NULL NULL 55 NULL NULL 65 NULL NULL 80 75 NULL NULL 90 NULL NULL
Depth-First pre-order dumps

Figure 6.3: Depth-first pre-order traversal showing persistent dump sequence

depth-first traversal precludes the need for a stack for the dumping process. In the depth-first pre-order traversal approach, the root of the subtree is visited, followed by the left subtree and then the right subtree. With this approach, the index can be dumped iteratively from the root to the external nodes which point directly to the leaf pages. Figure 6.3 illustrates the depth-first pre-order traversal strategy and shows the associated dump for this technique. A $NULL$ value is used to indicate the end of a particular subtree. Another approach will be to use a *nodeType* data field in each node to indicate if it is an external node or an internal node. In which case external nodes indicate the end of that particular subtree.

The internal index structure could therefore be reconstructed following the same iterative depth-first pre-order traversal using the keys in the persistent array store. Hence, the index is loaded at a fast rate and thereby improving the overall performance of the MMDB reload and recovery.

Chapter 7

Experimental Performance Evaluations

A number of experiments and comparative studies were conducted on the index structures discussed in Chapter 2 and the O_2-Tree. Different groups of experiments, each consisting of a series of other tests were conducted to ascertain the performance of the structures discussed. Some experiments were also conducted with the persistent O_2-Tree with in-memory cache, against common NoSQL (key-value) stores such as the BerkeleyDB, TreeDB of Kyoto Cabinet and LevelDB. These experiments were conducted primarily to compare the performance of O_2-Tree with these key-value stores using various access methods.

7.1 Experimental Environment

The O_2-Tree index was implemented in C++ and the program was compiled and built using the GNU g++ compiler on a 64-bit Intel i7 multi-core (with hyper threading enabled) processor machine running the Ubuntu 11.10 Operating System (OS). Another 64-bit Intel Xeon E5630 CPU machine having 72GB of RAM and running the Scientific Linux release 5.4 OS was used for the experimental evaluations of the O_2-Tree index as a persistent key-value data store in concurrent environment. Additionally, some of the basic existing index structures for main memory index were implemented, while others were modified from the available source codes. Each index structure considered, holds pointers to records (i.e., the memory addresses).

The concurrency control algorithms were implemented with the Boost Thread Library, (a portable C++ multi-threading library), version 1.48.0. The Boost Thread library was adopted for the implementation since it provides simple and extensive interfaces and functions for thread and lock management. Particularly, it provides the intermediary read-write lock called the *upgrade-lock* which is an extension to the multiple-reader / single-writer model concept: a single thread may have upgradable ownership at the same time as others have shared ownership (Williams 2008). These upgrade locks could be converted to exclusive locks atomically once all shared locks have been released. However, a second design using traditional primitive POSIX Threads (Pthreads) was implemented due to the observed overhead in using the Boost Thread Library. The Pthread library was used for the thread-safe implementation of the persistent O_2-Tree key-value data store.

The Netbeans Integrated Development Environment (IDE) equipped with the GNU Project Debugger (gdb) as well as the Sun Studio were used to facilitate the code implementation, debugging and profiling of these structures. Totalview was used to facilitate debugging of the multi-threaded code. TotalView is a GUI-based source code defect analysis tool that provides control over processes and thread execution. It also provides visibility into program state and variables. TotalView allows the debugging of one or many processes and/or threads with complete control over the program execution, and debugging operations(e.g., stepping through code etc.). Additionally, Valgrind's Tool Suite consisting of tools such as Helgrind, DRD and memcheck were used for profiling and debugging.

7.2 Data Sources

The performance of each index structure was evaluated experimentally in a simulated database environment. The keys and pointers in each experiment were set to 32 bits. All keys were randomly generated in advance within the range of 1 to $30 Million$. The same 4-bytes data randomly generated were used for the value in each $\langle key-value \rangle$ pair. The average time was reported for each experiment.

Data sets generated from the Transaction Processing Council (TPC-H) orders table (TPC-H Benchmark: 2011) were used for benchmarking the persistent O_2-Tree key-value data store as well.

7.3 Performance Evaluation And Discussion

7.3.1 Index Structures with No Concurrency Control

The first group of experiments were conducted to measure the performance of the index structures in various simulated database environment with no concurrency. The first set of experiments within this group involved a series of insertions. This was setup to measure how fast each data structure could construct the tree index given a random set of keys. The other query operations were also subjected to a similar model. The simulation was done such that each memory index used the same set of data in the same random order. In the second set of performance evaluations, each data structure was subjected to a mix of insertions, deletions and searches with different percentages (update ratios) of each operation. The total time to perform the entire mix of operations was then reported. A set of experiments were also carried out to evaluate the effect of the node size for the leaf-based structures (O_2-Tree, B$^+$-Tree, and the T-Tree).

7.3.1.1 Building the Index

Figure 7.1 shows the performance of the five data structures for a simple build of the index. Each experiment started with an empty tree and performed up to $30Million$ unique $\langle key - value\rangle$ pair insertions. The *order* of the T-Tree, B$^+$-Tree, and the O_2-Tree used was set at $m = 256$. The plot shows the times for building the respective data structures.

As can be observed from the graphs, AVL-Tree performed worst among the index structures considered while the O_2-Tree performed best. The Top-Down Red-Black Tree also performed better than the AVL-Tree. AVL's strict balancing requirement accounts for its worst performance. The O_2-Tree on the other hand, required comparatively fewer splits and rotations which accounts for its superior performance. The B$^+$-Tree performed better than the T-Tree due to its significant low depth and less complexity in restoring the tree's invariants. The O_2-Tree, however, outperformed the B$^+$-Tree due to the fact that the B$^+$-Tree makes multi-way decision during its traversal down the tree whiles the O_2-Tree makes single data comparison to determine the search path during traversal. Further, splitting and redistribution of keys in the nodes of a B$^+$-Tree may propagate all the way to the root of the tree. However, only color changes, which are less expensive, may propagate up to the root of an O_2-Tree. Figure 7.2 indicates the same experiment with sequential key insertions. Generally, the results were similar, as in the random key insertions but much lower average times for all the structures were observed.

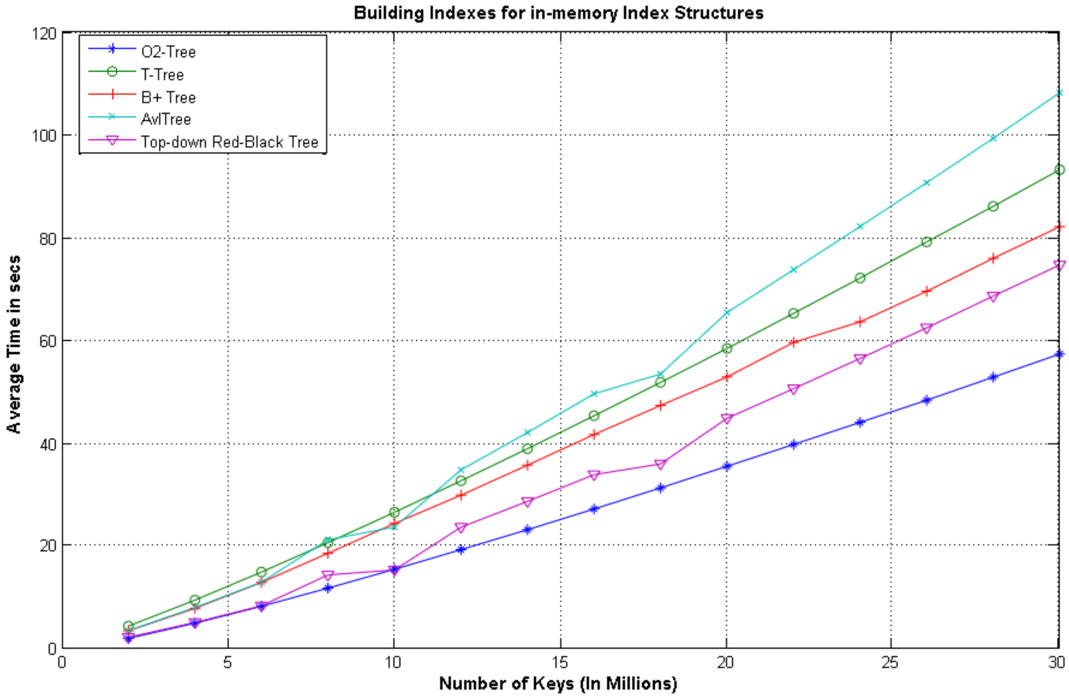


Figure 7.1: Building the Index Using Randomly Generated Keys

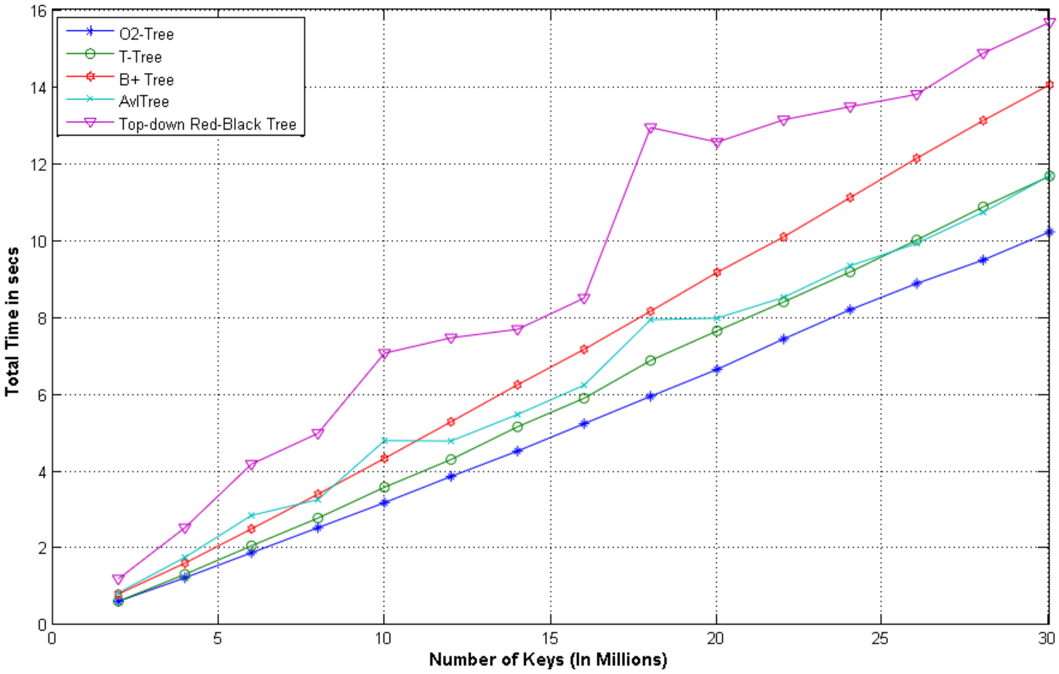


Figure 7.2: Building the Index with Sequential key Insertions Using Randomly Generated Keys

7.3.1.2 Exact-Match Query Searches

Figure 7.3 shows the performance of each structure with only successful exact match search queries. This evaluated the performance of each index in an environment where only search queries are run. The B$^+$-Tree performed better than the O_2-Tree and the T-Tree. The B$^+$-Tree and the T-Tree do have significantly low depth than that of the O_2-Tree due to the multiple element per node. Therefore, the search queries proceeded faster. Also, the B$^+$-Tree has a much lower depth than the T-Tree and hence its superior performance.

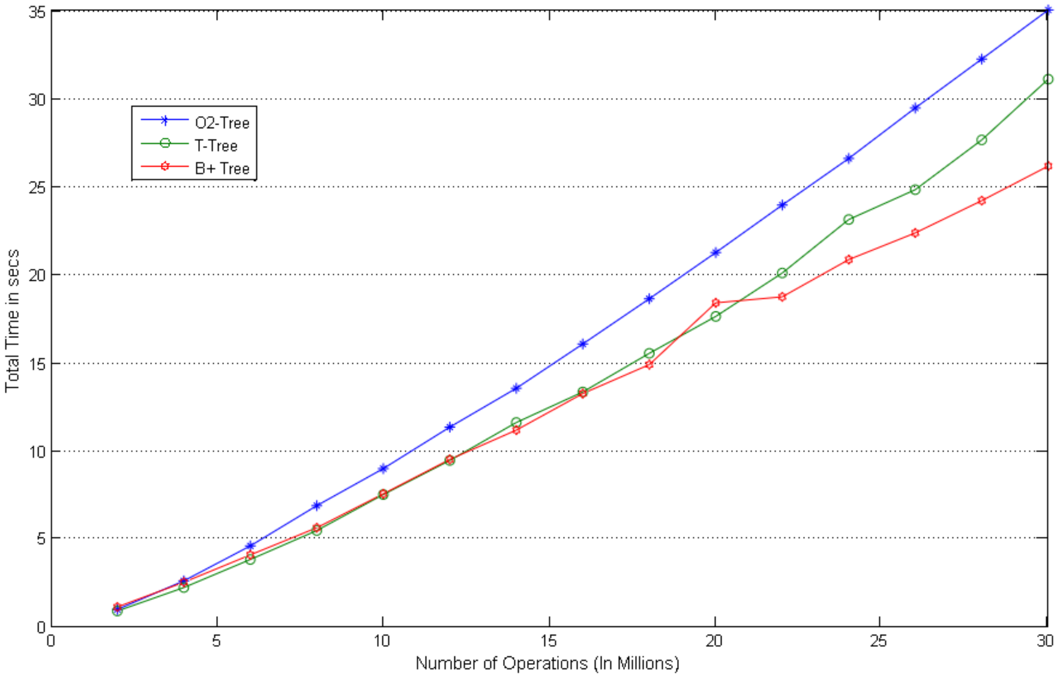


Figure 7.3: Workload of Successful Exact-Match Search Queries Using Randomly Generated Dataset

7.3.1.3 Query Mix with Different Workloads Using Randomly Generated Dataset

Figure 7.4 shows a similar set of experiments, with the same data structures except that now, the times were recorded for a mix of updates and searches. In the workload driven construction of the indexes, 50% of the keys generated were used for searches while 50% of the keys generated were for new insertions. The O_2-Tree outperformed all the other index structures tested in the experiment. Figure 7.5 and Figure 7.6 show similar results with different workload ratios in which 60%, 75% of the total operations are searches, 30%, 20% inserts and 10%, 5% deletes respectively.

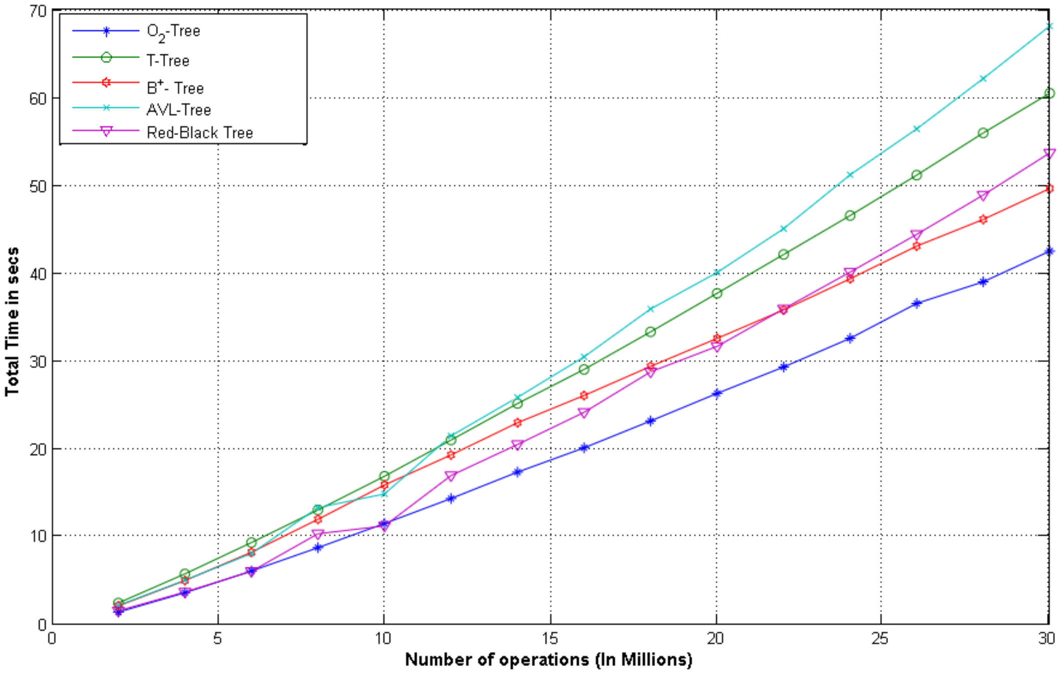


Figure 7.4: Workload of a Mix of 50% Exact-Match Searches and 50% Inserts

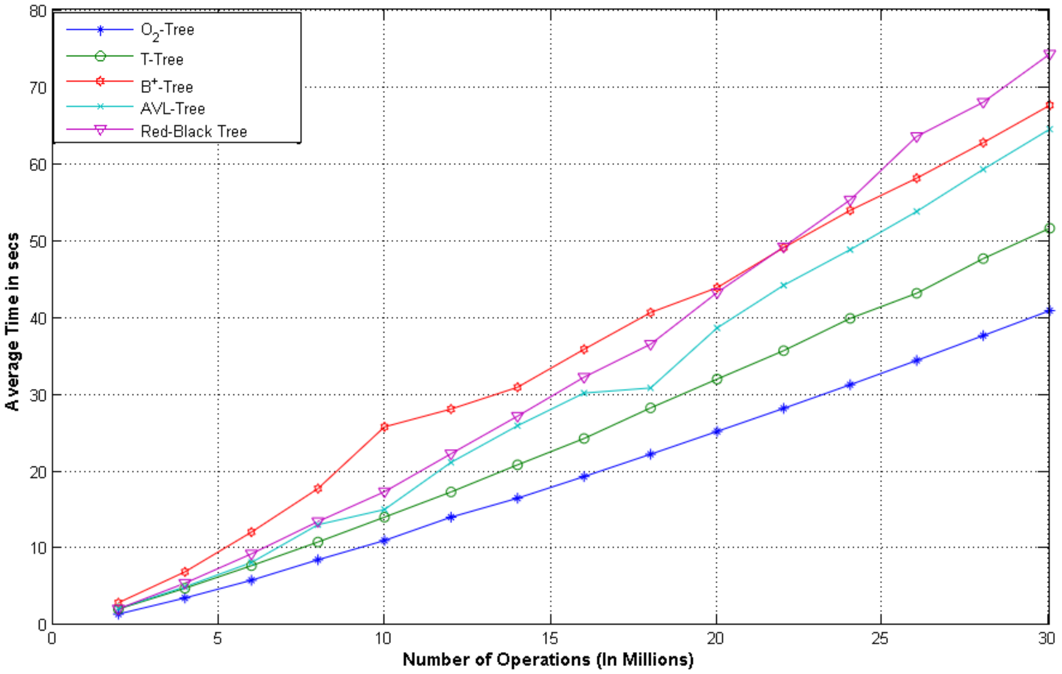


Figure 7.5: Workload of a Mix of 60% Exact-Match Searches, 30% Inserts, 10% Deletes

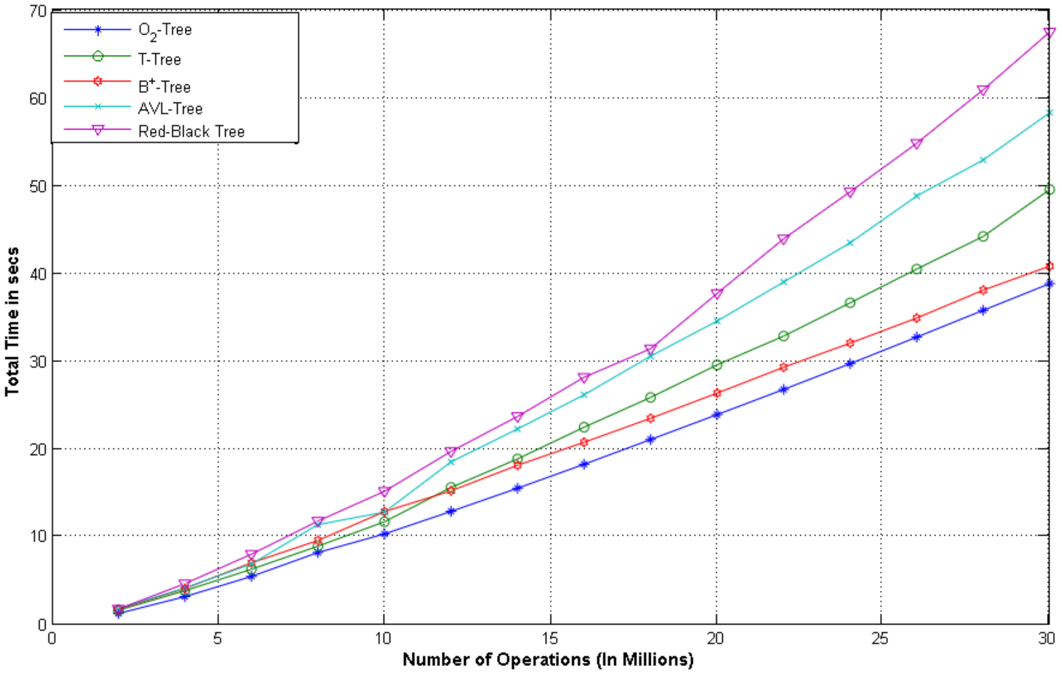


Figure 7.6: Workload of a Mix of 75% Searches, 20% Inserts and 5% Deletes

Figure 7.7 shows the general performance of each index structure under a workload of $10 Million$ (M) interleaved query operations and update ratios ranging from 0% to 100% update ratios (0% update ratio implied only searches while a 100% update indicated only updates). For each update ratio, 30% were delete operations while 70% were insertions. Generally, it was observed that as the update ratio increased, the average times increased as well. This was because updates may involve mutation operations and index restructuring. The O_2-Tree outperformed all the structures considered due to same reasons discussed in the previous Sections.

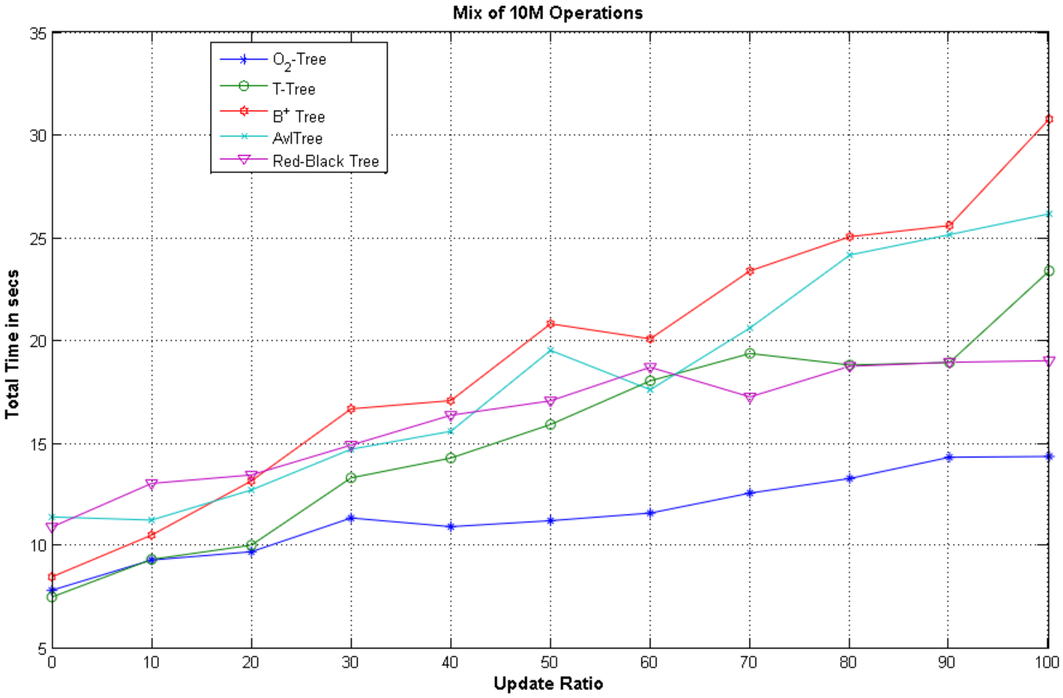


Figure 7.7: Performance of Index Structures With Varying Update Ratios

7.3.1.4 Effect of Node Size

Comparative experimental study were conducted on the effect of the leaf bucket size of the O_2-Tree, T-Tree and the B$^+$-Tree index structures. This was done to determine the optimum bucket size for best results for each leaf-based index structure. The results as indicated in Figure 7.8 shows a general drop in the performance as the node size increased. As the node size increased, the height of each tree generally reduced, it also resulted in fewer splits and therefore fewer internal nodes. However, the number of data comparisons per leaf-node increased. More so, larger bucket sizes resulted in a lot more data movements and rearrangement during the insertion of a new $\langle key - value \rangle$ pair and therefore accounted for the general drop in performance.

The O_2-Tree generally outperformed all the indices considered. More so, it was observed that, as the bucket size increases, the O_2-Tree relative performance increased considerably with respect to the other indices. This is because, the *internal nodes* of the O_2-Tree are still traversed with single data comparison unlike the T-Tree and the B$^+$-Tree. However, with small bucket sizes of 32 and 64, the B$^+$-Tree, performed better than the O_2-Tree and the T-Tree. This is due to the fact that the small bucket sizes result in more splits and restructuring operations for the O_2-Tree and the T-Tree indices.

Since the O_2-Tree and the T-Tree have more complex and *expensive* restructuring algorithms, they generally performed worse with small bucket sizes.

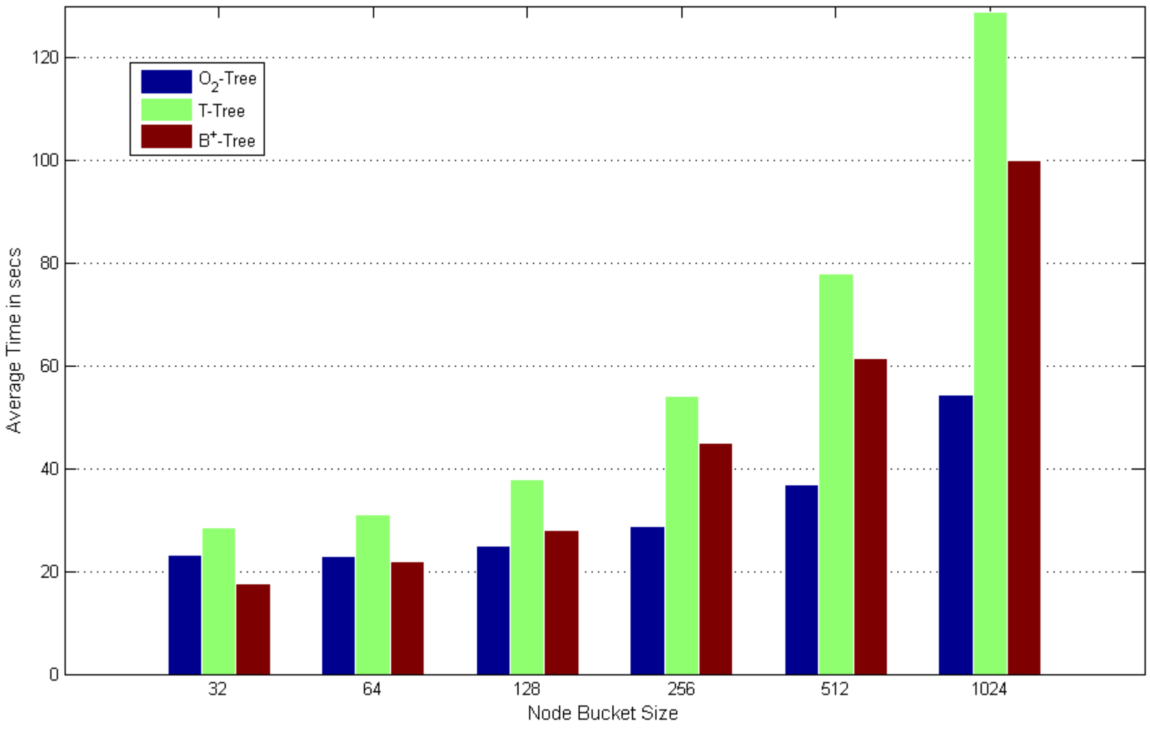


Figure 7.8: Effect of Node Size on the In-Memory Indices Using Randomly Generated Dataset

7.3.2 O_2-Tree with In-Memory Cache

The second group of experiments were conducted to compare the persistent implementation of the O_2-Tree index structure through an in-memory cache with NoSQL key-value databases such BerkeleyDB, Kyoto Cabinet as well as LevelDB, as discussed in Section 2.2. In this experiment, operations were performed in the in-memory cache and pages periodically flushed to disk based on the *Least Recently Used* (LRU) cache algorithm. The experiments were setup such that each key-value data store had enough cache to keep the data being processed. The data was flushed to disk at the end of each experiment. The total time to complete the entire operation was then reported. This was repeated for varying data sizes and database workloads similar to the first group of experiments. The cache size and the page sizes were also tuned to determine their effect on the performance of each key-value store and for best performance.

7.3.2.1 Persistent Storage with In-Memory Cache

The experiments illustrate the performance of the in-memory persistent O_2-Tree with the NoSQL key-value data stores discussed. Each key-value data store was tuned with a page size of *4k (4096 bytes)* as well as a $2GB$ cache size. The various optimization mechanisms for the NoSQL databases as indicated and recommended from their respective documentations were also applied for best results. Compression was disabled for LevelDB and the Kyoto Cabinet TreeDB to achieve better performance rather than storage optimisation.

The results indicated that the persistent O_2-Tree performed comparably to LevelDB and Kyoto Cabinet HashDB (using *4GB* map, *5 Million* Buckets). The LevelDB was ran in asynchronous mode in which a separate thread flushed the cache contents concurrently to disk. Comparing the HashDB and the multi-threaded LevelDB with the persistent O_2-Tree was more of an "apple-orange" comparison biased against the O_2-Tree persistent store since it does not asynchronously flush cache contents to disk. However, the objective was to evaluate how the tree-based persistent O_2-Tree index compared with these popular industry-standard NoSQL key-value stores.

The persistent O_2-Tree, however, performed over $5X$ faster than the Kyoto Cabinet TreeDB (using the B$^+$-Tree access method) and more than $2X$ as fast as BerkeleyDB using the *Btree* access method. The Kyoto Cabinet HashDB, performed excellently well when tuned with a *4GB* map and $5Million$ buckets. The Kyoto Cabinet HashDB, however, does not support ordered traversals as is the case with hash data structures. The results are as shown in Figure 7.9. It was also observed that, the persistent O_2-Tree actually performed comparable and even better than the LevelDB for keys sizes up to $20Million$.

Figure 7.10 shows the performance of the key-value data stores using TPC-H generated dataset up to $20Million$. The persistent O_2-Tree (referred to as O_2-Tree KV) outperformed all other key-value data stores. The difference between this result and that of Figure 7.9 may be due to the differences in the datasets used for both experiments. The dataset used in Figure 7.9 was uniformly generated while the TPC-H data was not uniformly generated and very representative of real life data.

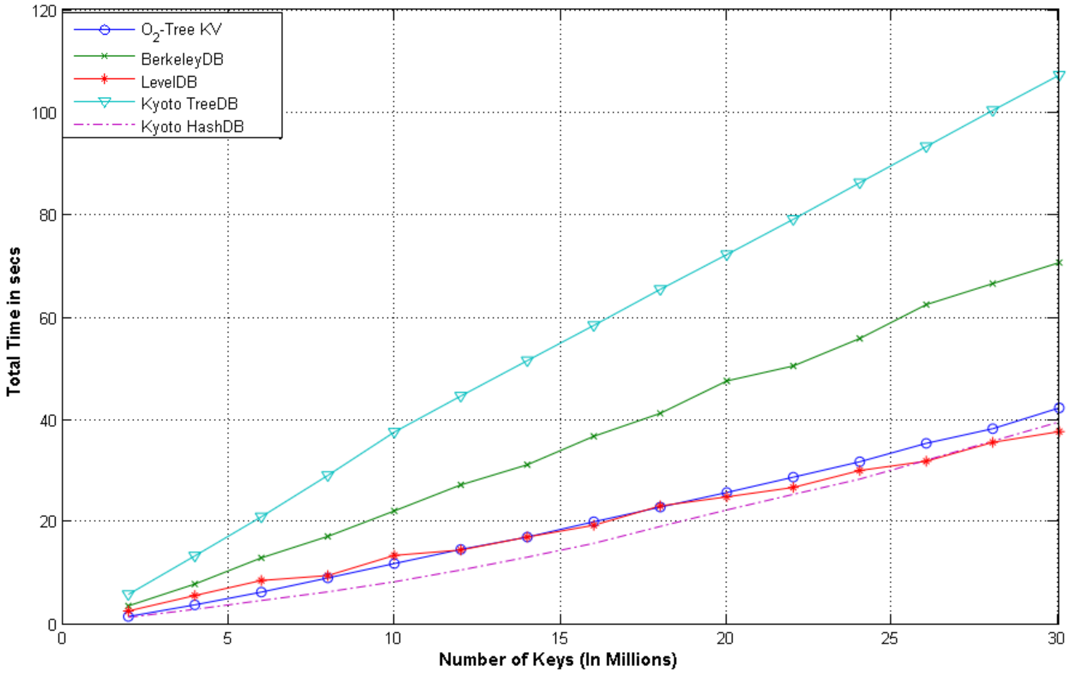


Figure 7.9: Persistent Storage through In-Memory Cache Using Randomly Generated Dataset

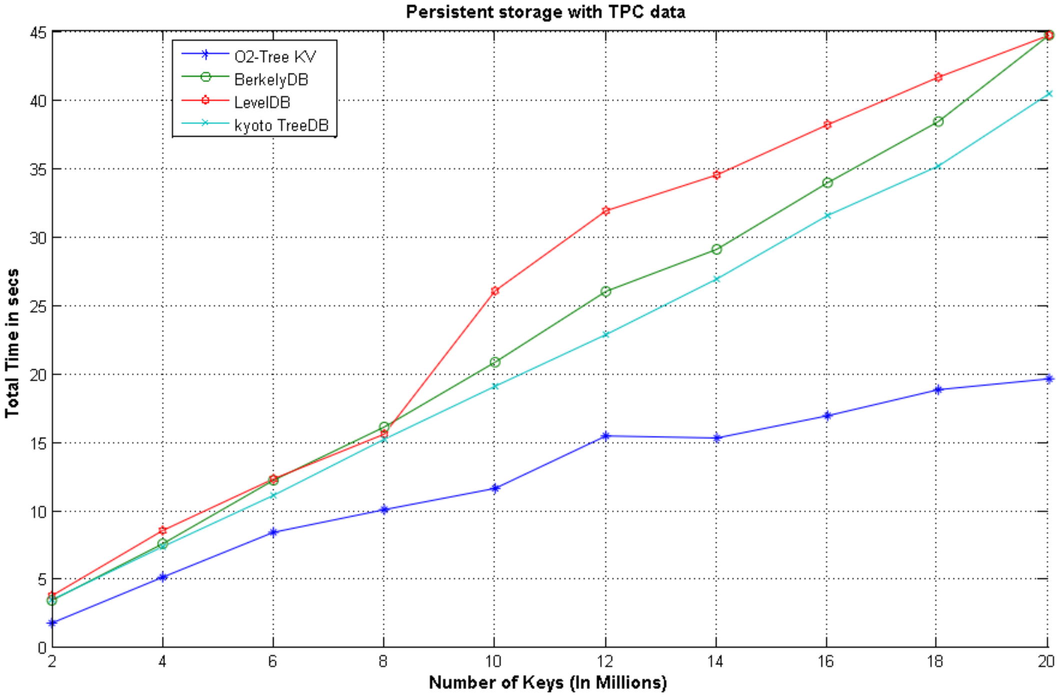


Figure 7.10: Persistent Storage through In-Memory Cache Using TPC-H Dataset

7.3.2.2 Effect of Different Workloads

The second set of experiments with the NoSQL databases illustrated in Figure 7.11 and Figure 7.12, shows the performance of each key-value data store under different workloads; 50% inserts and 50% searches, as well as a workload of 75% searches, 20% inserts and 5% deletes. Again, the persistent O_2-Tree showed a performance comparable to the LevelDB and the Kyoto Cabinet HashDB. It did however, outperformed the BerkeleyDB and the Kyoto Cabinet TreeDB by several orders of magnitude. The HashDB outperformed all the structures due to the fact that it does not arrange keys in an ordered manner and therefore does not support ordered traversals or range scans.

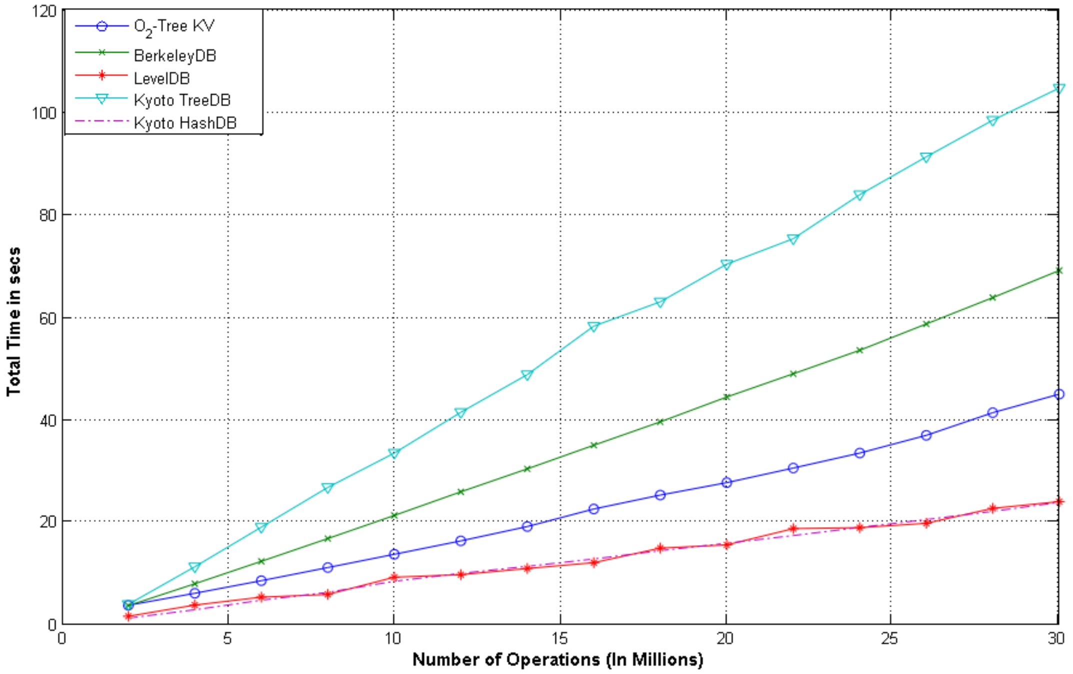


Figure 7.11: Workload of a Mix of 50% Exact-Match Searches and 50% Inserts Using Randomly Generated Dataset

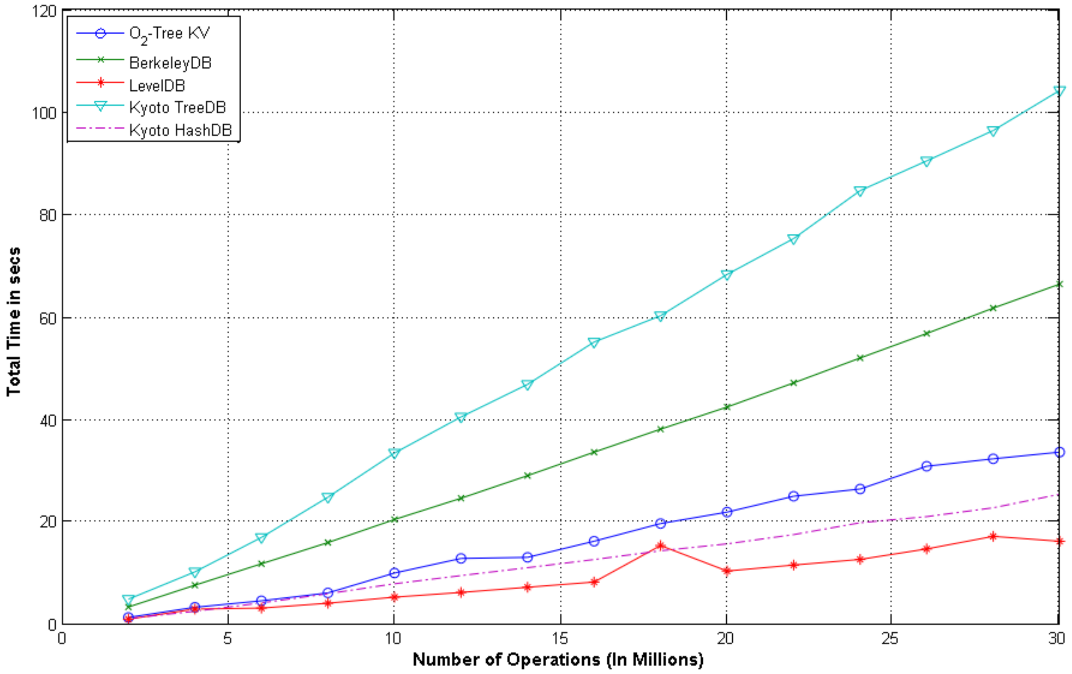


Figure 7.12: Workload of a Mix of 75% Searches, 20% Inserts and 5% Deletes Using Randomly Generated Dataset

Figure 7.13 shows the results of a similar experiment but with TPC-H generated data for the O_2-Tree KV, BerkeleyDB, Kyoto Cabinet TreeDB (referred to as KyotoDB) and LevelDB. Generally, as the update ratio increased, the performance of each data store dropped due to mutations and restructuring operations as discussed above. The O_2-Tree KV, however, outperformed the other key-value data stores considered in this thesis.

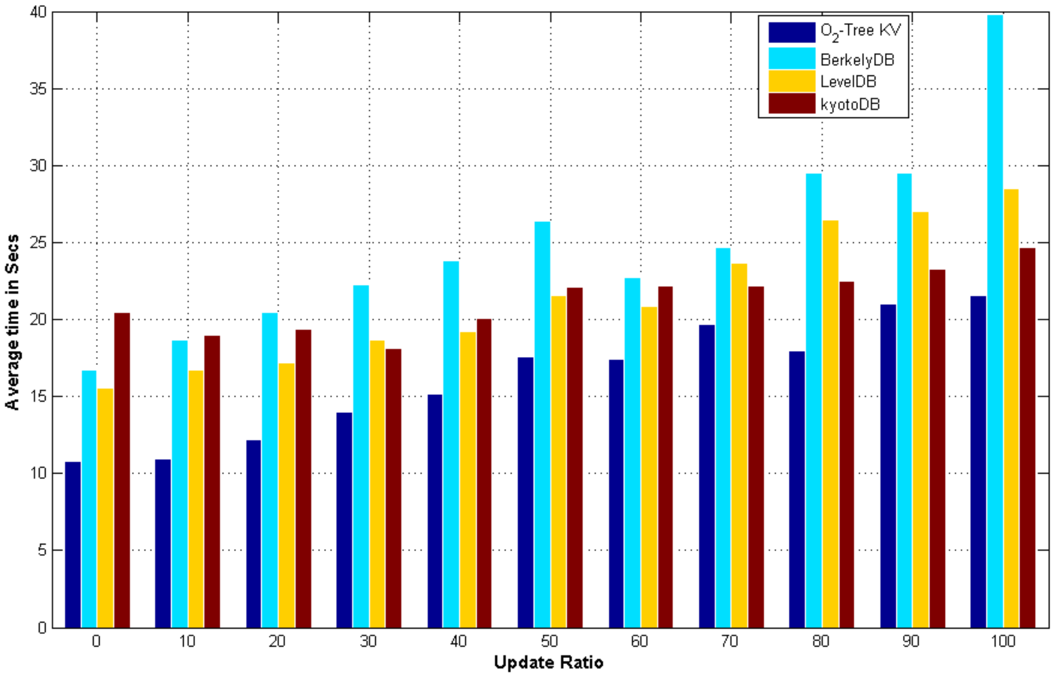


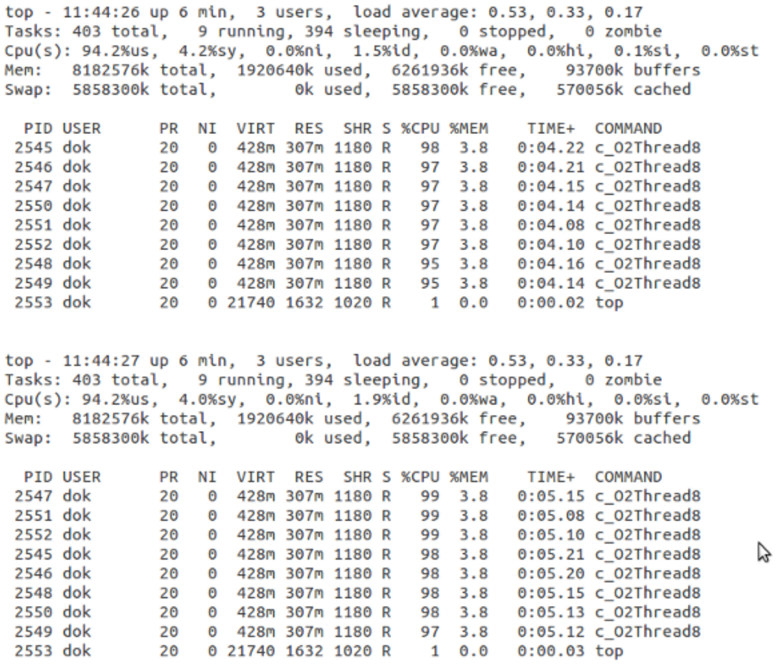
Figure 7.13: Query Mix Operations of key-value Data Stores with Varying Update Ratios Using TPC-H Dataset

7.3.3 O_2-Tree with Concurrency Control

In this group of experiments, the performance of the thread-safe version of the O_2-Tree compared with the T-Tree were evaluated. T-Tree was chosen for this benchmark due to the fact that it is the index structure which was specifically designed for in-memory Databases. More so, recent high speed in-memory databases such as the Oracle Timesten, DataBlitz, SAP HANA etc., do use variants of the T-Tree for indexing.

Both structures were implemented with their respective relaxed-balance restructuring algorithm. Different sets of experiments were then conducted to determine the effect of concurrent access on these structures in-memory. *Two* to *eight* threads from the Intel i7 CPU with hyper-threading enabled was used in the test environment. The unix *time* or *top* commands were used to capture the average time of each experiment for different access patterns. The unix *top* command was run in batch mode to capture the time of each thread. For instance, the command *"top -b -H -i -d 1 > outputfile.txt"*, captures the *top* output every *second* for each thread and redirects the output to the file "$outputfile.txt$". The output captured for each thread includes, the RAM usage, percentage of

CPU usage as well as the CPU time, etc. The average CPU time for each run was then reported for the experiment. (i.e., the average CPU time = sum of CPU time per thread / total number of threads). Figure 7.14 shows a sample output from the *top* command as discussed.



```
top - 11:44:26 up 6 min,  3 users,  load average: 0.53, 0.33, 0.17
Tasks: 403 total,   9 running, 394 sleeping,   0 stopped,   0 zombie
Cpu(s): 94.2%us,  4.2%sy,  0.0%ni,  1.5%id,  0.0%wa,  0.0%hi,  0.1%si,  0.0%st
Mem:   8182576k total,  1920640k used,  6261936k free,    93700k buffers
Swap:  5858300k total,        0k used,  5858300k free,   570056k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2545 dok       20   0  428m 307m 1180 R   98  3.8   0:04.22 c_O2Thread8
 2546 dok       20   0  428m 307m 1180 R   97  3.8   0:04.21 c_O2Thread8
 2547 dok       20   0  428m 307m 1180 R   97  3.8   0:04.15 c_O2Thread8
 2550 dok       20   0  428m 307m 1180 R   97  3.8   0:04.14 c_O2Thread8
 2551 dok       20   0  428m 307m 1180 R   97  3.8   0:04.08 c_O2Thread8
 2552 dok       20   0  428m 307m 1180 R   97  3.8   0:04.10 c_O2Thread8
 2548 dok       20   0  428m 307m 1180 R   95  3.8   0:04.16 c_O2Thread8
 2549 dok       20   0  428m 307m 1180 R   95  3.8   0:04.14 c_O2Thread8
 2553 dok       20   0 21740 1632 1020 R    1  0.0   0:00.02 top

top - 11:44:27 up 6 min,  3 users,  load average: 0.53, 0.33, 0.17
Tasks: 403 total,   9 running, 394 sleeping,   0 stopped,   0 zombie
Cpu(s): 94.2%us,  4.0%sy,  0.0%ni,  1.9%id,  0.0%wa,  0.0%hi,  0.0%si,  0.0%st
Mem:   8182576k total,  1920640k used,  6261936k free,    93700k buffers
Swap:  5858300k total,        0k used,  5858300k free,   570056k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2547 dok       20   0  428m 307m 1180 R   99  3.8   0:05.15 c_O2Thread8
 2551 dok       20   0  428m 307m 1180 R   99  3.8   0:05.08 c_O2Thread8
 2552 dok       20   0  428m 307m 1180 R   99  3.8   0:05.10 c_O2Thread8
 2545 dok       20   0  428m 307m 1180 R   98  3.8   0:05.21 c_O2Thread8
 2546 dok       20   0  428m 307m 1180 R   98  3.8   0:05.20 c_O2Thread8
 2548 dok       20   0  428m 307m 1180 R   98  3.8   0:05.15 c_O2Thread8
 2550 dok       20   0  428m 307m 1180 R   98  3.8   0:05.13 c_O2Thread8
 2549 dok       20   0  428m 307m 1180 R   97  3.8   0:05.12 c_O2Thread8
 2553 dok       20   0 21740 1632 1020 R    1  0.0   0:00.03 top
```

Figure 7.14: Sample Output of the *top* command

7.3.3.1 Concurrent Search Operations

The effect of simultaneous search operations on the thread-safe index structures was evaluated in this experiment. *Readers* proceeded with the traditional shared locks. The various worker/user threads, therefore proceeded with no blocking. In this experiment, threads share the workload equally. The dataset used consisted of $10Million$ unique 4-bytes keys and 4-bytes values randomly generated from the set of $30Million$ unique keys.

As indicated in Figure 7.15, as the number of threads increased, each index performed relatively better. A significant improvement was observed from the case where a single thread was used to the case where 2 worker threads were used. The performance almost doubled in this case. Also, beyond the 2-threads case, as the number of threads increased, there were general performance gains but not as significant as the first case described above. The T-Tree generally did perform better than the O_2-Tree especially when fewer threads were used. The relatively low depth of the T-Tree compared to the O_2-Tree accounts for this better performance. However, as the number of threads increased, the O_2-Tree's performance improved much more than that of the T-Tree and as such the performance difference reduced significantly. The O_2-Tree actually outperformed the T-Tree when 8 threads were used, though not very significant.

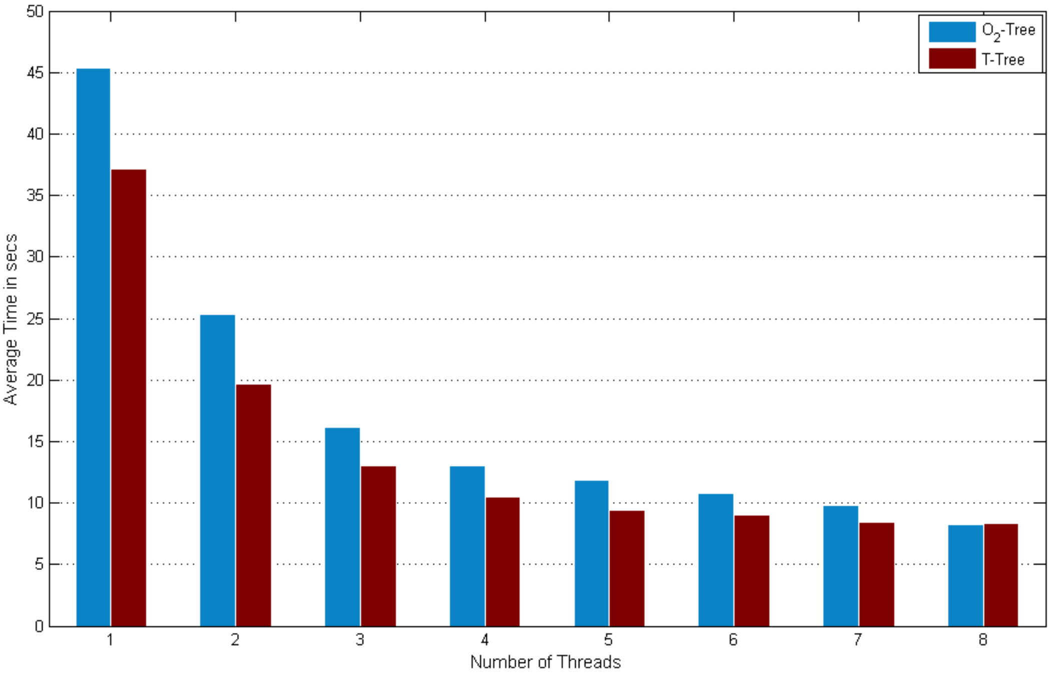


Figure 7.15: Performance of Concurrent Search Queries Using Randomly Generated Dataset

7.3.3.2 Building the In-Memory Tree Index

Updates in the thread-safe implementation, unlike the search queries, involve several restructuring operations to keep the tree balanced. To ensure correctness, therefore, updaters block other threads and proceed with *upgrade-locks* as discussed in Chapter 5. In these experiments each tree was initialised with 512 $\langle key-value\rangle$ pairs. Each thread was assigned equal number of random $\langle key-value\rangle$ pairs to insert in the tree concurrently. A single *rebalancer()* thread was also initialised to restore the invariants of the index structures.

Generally, as the number of threads increased, the performance of the index structures increased as well. However, there was a drop in performance when using 5 threads onwards. Beyond 5 threads, the performance remained approximately the same. As the number of threads increased, the number of contention and blocking increased as well, this affected the overall performance and hence the reason for the almost constant time when 6 or more threads were used. It was also observed that the

O_2-Tree outperformed the T-Tree in all cases. This is due to the fact that the O_2-Tree does single data comparison per node, while the T-Tree does multiple comparison per node and also restoring the invariant a T-Tree involves several data movements. The T-Tree insert operation requires blocking several nodes especially when a bounded internal node is full. When a bounded internal node is full, the minimum key is removed to make room for the new key to be inserted in that node. The in-order predecessor is then located to insert the removed key. These operations and data movements involved exclusive locks and therefore affected the performance. However, data movements with exclusive locks only happen at the leaf-nodes of the O_2-Tree. Figure 7.16 illustrates the performance of building the index with $10 Million$ unique 8-byte key-value pairs when using different number of threads.

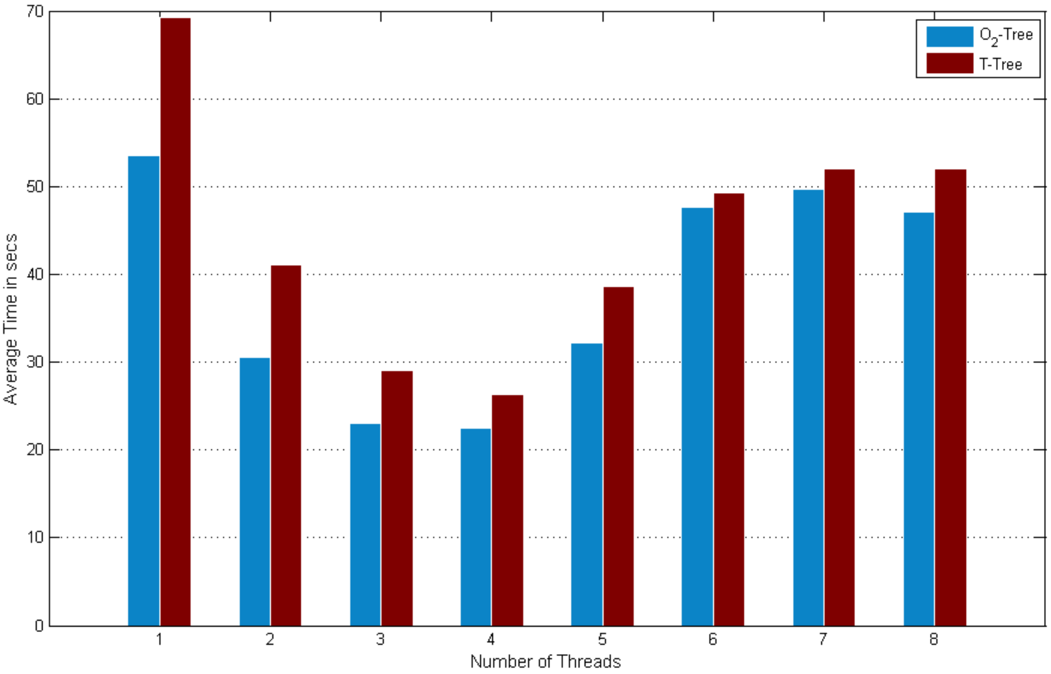


Figure 7.16: Performance of Multiple Threads when Building the Tree Index Using Randomly Generated Dataset

7.3.3.3 Concurrent Deletes from the In-Memory Tree Index

These experiments were conducted to measure the performance of each index structure when multiple delete threads were concurrently spawned. The restructuring of each index was left to the *rebalancer()* process to handle. Figure 7.17 illustrates the performance of multiple delete threads on the indices. A general trend similar to that of the concurrent insertions was observed. The performance improved from a single thread to the case when 4 threads were used, after which the performance dropped. However, there was no significance change in performance from the 6 worker threads onwards. The O_2-Tree, did outperform the T-Tree in all cases. The same reasons discussed in the previous section accounted for this performance as well.

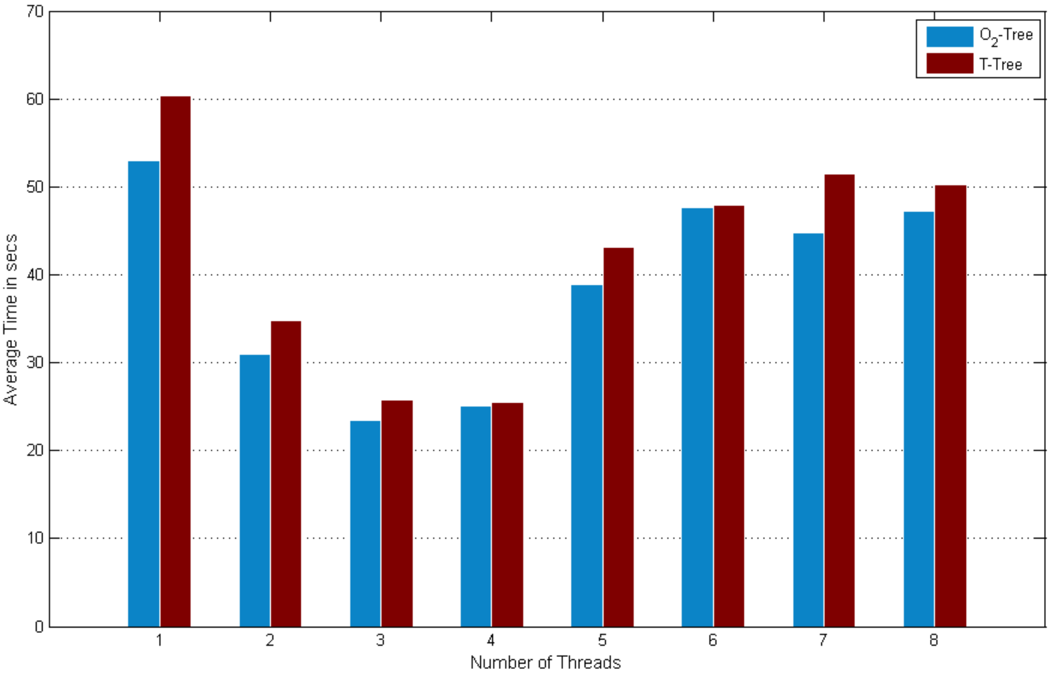


Figure 7.17: Effect of Multiple Delete Threads on the Tree Index Using Randomly Generated Dataset

7.3.3.4 Concurrent Access with Varying Workloads

The series of experiments in this section involved varying workloads with different number of concurrent operations. This was done to evaluate the effect on the performance of the data structures in a simulated environment where different users perform different mix of query operations. Query operations were interleaved and the ratio of updates to searches was varied from 0 to 100%, where a 0% update ratio indicated that only search operations were carried out, and a 100% update ratio indicated that only query update operations were carried out. For each update, the probability that it was an insertion was 0.6 and the probability that it was a deletion was 0.4. Five (5) Million query operations with the respective update ratios were performed using 8 threads. Each thread had equal share of data with which to perform the query operations.

Figure 7.18 illustrates the plots. Generally, as the update ratio increased, the average time increased as well. Since update queries resulted in restructuring operations to restore the tree's invariants, the more the updates, the more the restructuring operations and hence the greater the average time. Further, updaters block each other during traversal and lock exclusively nodes which they update. This affected the average time to perform the query operations. The best performance was observed when the update ratio was 0% (100% searches) whiles the 100% update ratio (100% updates) resulted in the worst performance. The O_2-Tree performed better than the T-Tree. The difference was, however, significant when the update ratios were low. Even though the relaxed algorithm for the T-Tree was much simpler with few conditions, the multi-way decision and multiple data comparison per node, and the complexity of restructuring during updates accounted for its worse performance.

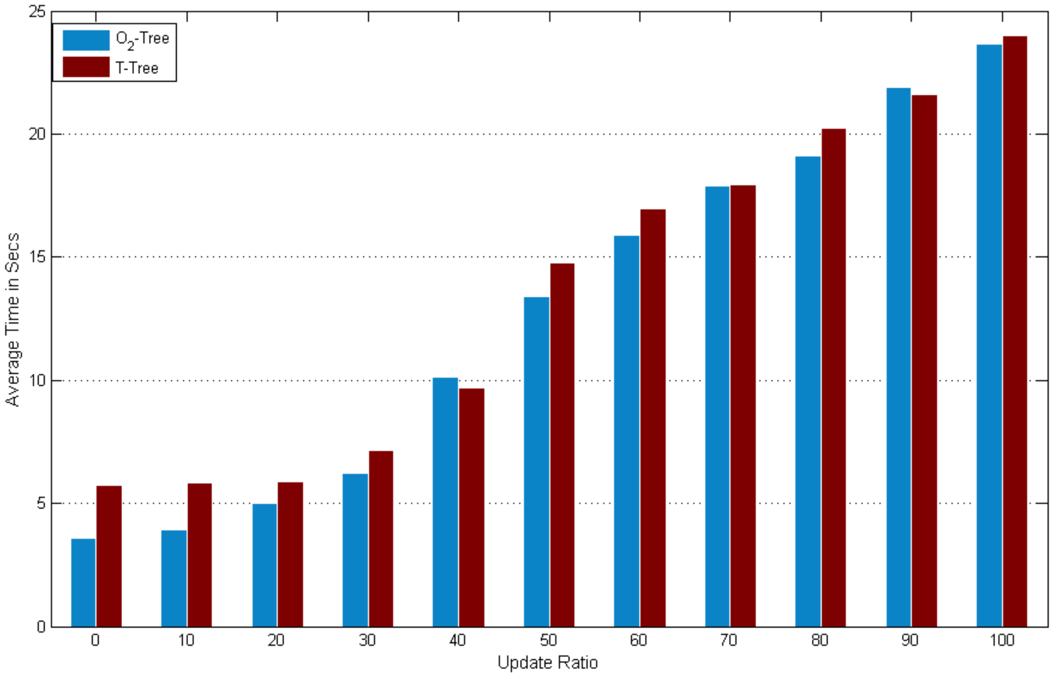


Figure 7.18: Concurrent Interleaved Operations with Varying Workloads Using Randomly Generated Dataset

Figure 7.19 shows similar results but with respect to the number of operations per second (OPS). This indicated how many query operations were performed by the eight (8) worker threads given the same update ratios as discussed above. It was observe that for an update ratio of 0%, the O_2-Tree performed over 1.3 million operations per second (OPS) while the T-Tree resulted in approximately eight-hundred and seventy one thousand (871000) query OPS. However, for higher update ratios, the O_2-Tree performed better but not as significant as in the cases with lower update ratios. For instance, for an update ratio of 100% the O_2-Tree resulted in approximately 212000 OPS and the

T-Tree 209000 OPS.

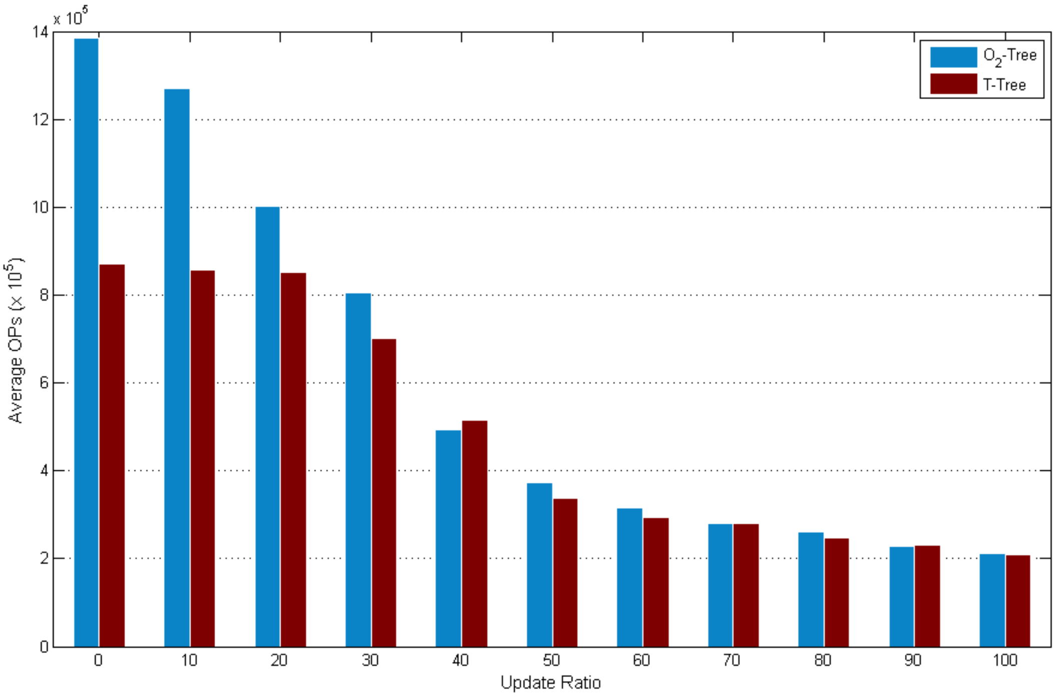


Figure 7.19: Operational Throughput on Varying Workloads Using Randomly Generated Dataset

7.3.3.5 CPU Utilisation

The CPU utilization of each thread during the program execution with varying update ratios was captured for a workload of $5 Million$ query operations. This was done to determine the efficiency of the algorithm in utilising the CPU cores. The average utilization for each run of the experiments was reported. Algorithms which blocked other threads for long periods will obviously result in poor efficiency in terms of CPU utilization. It was observed that, with 0% update ratio (i.e 100% search queries), the average utilization was over 95% (i.e., the highest), whereas an update ratio of 100% resulted in the lowest average utilization (≈ 46%). The reason being that, with 100% search queries, all threads proceeded at full capacity with no blocking. For an update ratio of 100%, however, threads blocked each other to ensure consistency and thereby serializing operations which traversed the same search path. Generally, the more updates there were, the more operations were blocked and the less the CPU utilization. Lower percentages of updates, therefore, resulted in higher utilization of the CPU cores.

For lower update ratios, the T-Tree had a relatively higher utilisation than the O_2-Tree. However, the O_2-Tree indicated a relatively higher CPU utilisation for higher update ratios ($\geq$ 60%). The graph shown in Figure 7.20 illustrates the CPU Utilization as discussed.

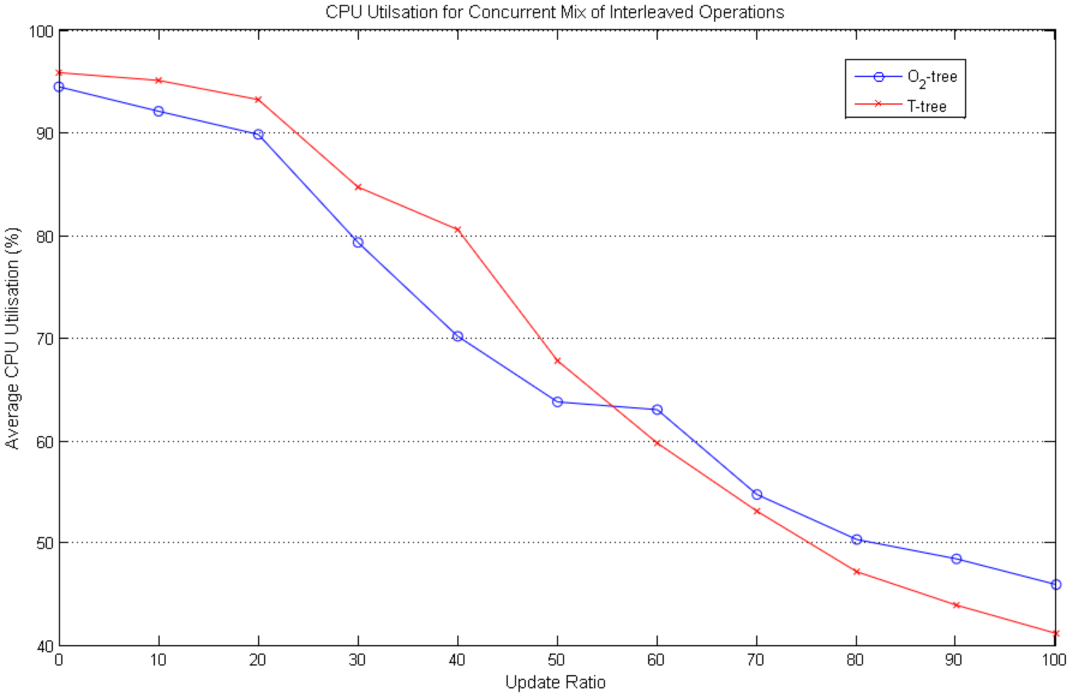


Figure 7.20: CPU Utilization at Various Workloads

7.3.4 Persistent O_2-Tree Data Store with Concurrency Control

The experiments in this section were conducted on the 64-bit Intel Xeon E5630 CPU machine having 72GB of RAM and running the Scientific Linux release 5.4 OS. The performance of the O_2-Tree index KV , BerkeleyDB, KyotoDB, as well as LevelDB were evaluated under various workloads in a multi-threaded concurrent environment. Each key-value store was tuned with a page size of 4k (4096 Bytes) as well as a 2GB in-memory cache. The experiment were conducted with threads varied from 2 to 16.

7.3.4.1 Scalability In Multi-threaded Insertions

The first experiment in this section evaluated how each key-value data store scaled with increased contention and concurrency. $20 Million$ concurrent insertions were performed with uniformly generated random dataset. These operations were performed in the in-memory cache and pages were periodically flushed to disk based on the LRU cache policy.

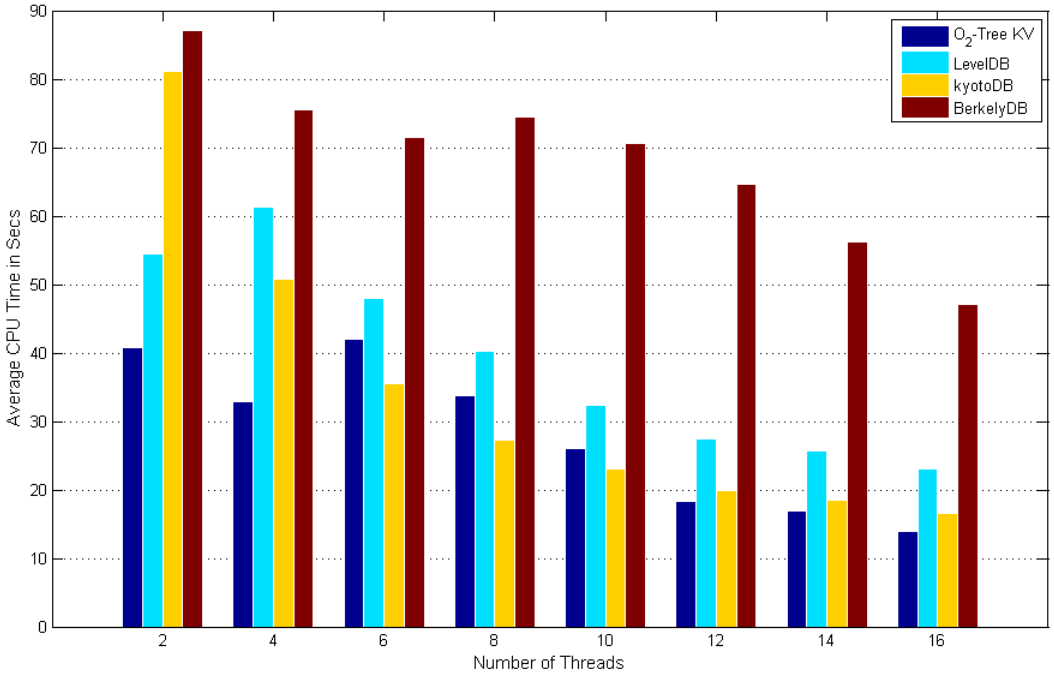


Figure 7.21: Scalability of key-value Data Stores Under High Workloads Using Randomly Generated Dataset

A general performance gain was observed as the number of threads and contention levels were increased from 2 to 16. However, the O_2-Tree KV performed better than the other key-value data stores and showed a good level of scalability due to its simplistic approach to persistent data storage and index mechanism. The O_2-Tree, performed over 2.5X faster than the KyotoDB and BerkeleyDB (both using the *Btree* access method). The results are shown in Figure 7.21.

7.3.4.2 Throughput for Query Mix Operations

The second set of experiments with the NoSQL databases were conducted to evaluate the throughput of each key-value data store under different workloads. For each update workload, 30% were delete operations while 70% were insert operations. All query operations were interleaved such that a user thread performed either an update or a search operation at a time. All operations were performed with 16 user threads. Figure 7.22 shows the results from the experiment.

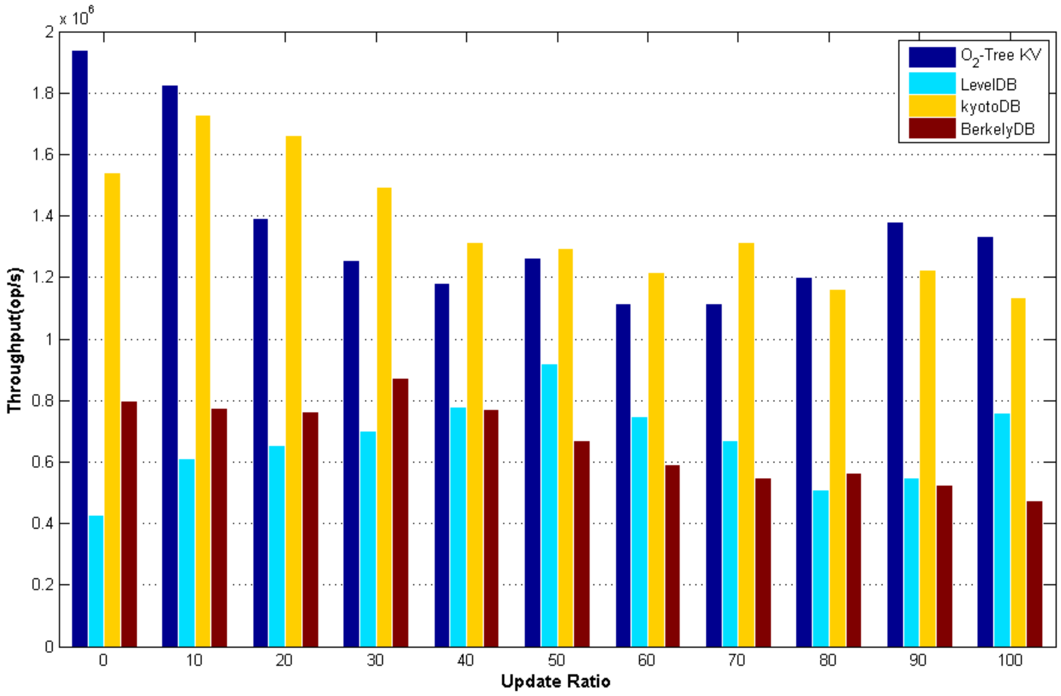


Figure 7.22: Throughput Comparison for Different Mix of Workloads using Randomly Generated Dataset

It was observed that, the throughput decreased as the update ratio increased. This was due to the fact that, updates required restructuring which affected the overall performance. More updates resulted in several blocking by threads holding exclusive locks to a data item or node. The O_2-Tree KV recorded the highest throughput of about $1.9M$ operations per second (op/s) which later dropped to 1.3Mop/s at 100% updates. A similar trend was observed for all key-value data stores considered.

Chapter 8

Conclusion

8.1 Summary

We have studied and discussed some of the existing index structures for main memory resident databases. We have also discussed some of the favourable and unfavourable properties of these earlier index structures. We have proposed the O_2-Tree, a new index scheme for in-memory database systems which is based on the Red-Black Tree. The index supports exact-match as well as range search queries. We have also shown that the O_2-Tree guarantees a logarithmic query processing as in other balanced binary trees. However, the new structure has a relatively low depth as compared to the Red-Black-Tree, and as such guarantees a faster processing time than other well known existing indices such as AVL-Tree, T-Tree as well as the B$^+$-Tree. We have also shown experimentally the superior performance of the O_2-Tree. The height of the O_2-Tree is in general less than that of an equivalent Red-Back Tree with the same number of keys but comparatively higher than an equivalent T-Tree or B$^+$-Tree. The difference in height is a function of the *order* of the tree. Even though the O_2-Tree has a greater height than that of the B$^+$-Tree and the T-Tree, the O_2-Tree has a lower branching factor (2 per node) and therefore does fewer data comparisons per node than a B$^+$-Tree and the T-Tree of the same order. Our experimental results showed that the O_2-Tree is much more superior to earlier main memory index structures, especially when the datasets are large and also when insertions occur both in random as well as sequential order of keys. We observed that in a workload consisting of only search queries, the B$^+$-Tree was a better and much faster index structure. However, in real-time database environments, where there are frequent insertions and possibly deletions, and rebuilding the index after failure, the O_2-Tree provides a relatively better performance. It becomes, therefore the index structure of choice for such systems. The B$^+$-Tree on the other hand provides a better performance in systems where searches are predominant.

Regarding index persistence and fault recovery, we observed that the O_2-Tree could easily and quickly be rebuilt from persistent dumps. Unlike, earlier index structures, only the leaf nodes in an O_2-Tree index needs to be dumped to persistent store. From these persistent stores, only the first key-value pairs have to be read to rebuild the entire index. The O_2-Tree is thus inherently fault tolerant. The other structures such as the T-Tree and the B^+-Tree, however, require that the entire data pages of the database be traversed in order to reconstruct the index. Additionally, we have shown that the persistent implementation of the O_2-Tree outperformed popular NoSQL key-value data stores such as the BerkeleyDB with the B-tree access method as well as the Kyoto Cabinet TreeDB by several orders of magnitude. Our persistent implementation of the O_2-Tree performed comparably to the LevelDB persistent key-value data store in asynchronous mode even though the O_2-Tree was disadvantaged in this particular experiment.

We have also presented the performance of the concurrent tree indices in shared memory multi-core architectures, thus exploiting the modern trends in multi-core processor designs. We have experimentally studied the performance of the thread-safe algorithms of the T-Tree and the O_2-Tree using their respective relaxed balance algorithms. We observed a general improvement in the performance of the indices as more worker threads share the workload, however as the number of threads increased beyond four(4), there was a drop in performance due to locking overhead and thread contentions. Regarding search queries, which proceed without blocking there was a significant improvement in performance as the number of threads increased. We conclude from the experimental results that there is an optimum level of concurrency beyond which there is no significant improvement in performance. Performance could actually drop, especially for updates due to locking overhead, beyond this optimal point. In our case, such optimal number of worker threads for updates was four(4) for our testing platform. This could be due to the fact that we had four(4) actual cores, although hyper-threaded to eight(8). The number of lock contentions increase with increasing number of threads which affect the overall performance of the concurrent update algorithms. However, we anticipate a much greater optimal point for platforms with much more cores. We also, anticipate much better performance gains with lock-free concurrent models such as Software Transactional Memory (STM).

8.2 Future work

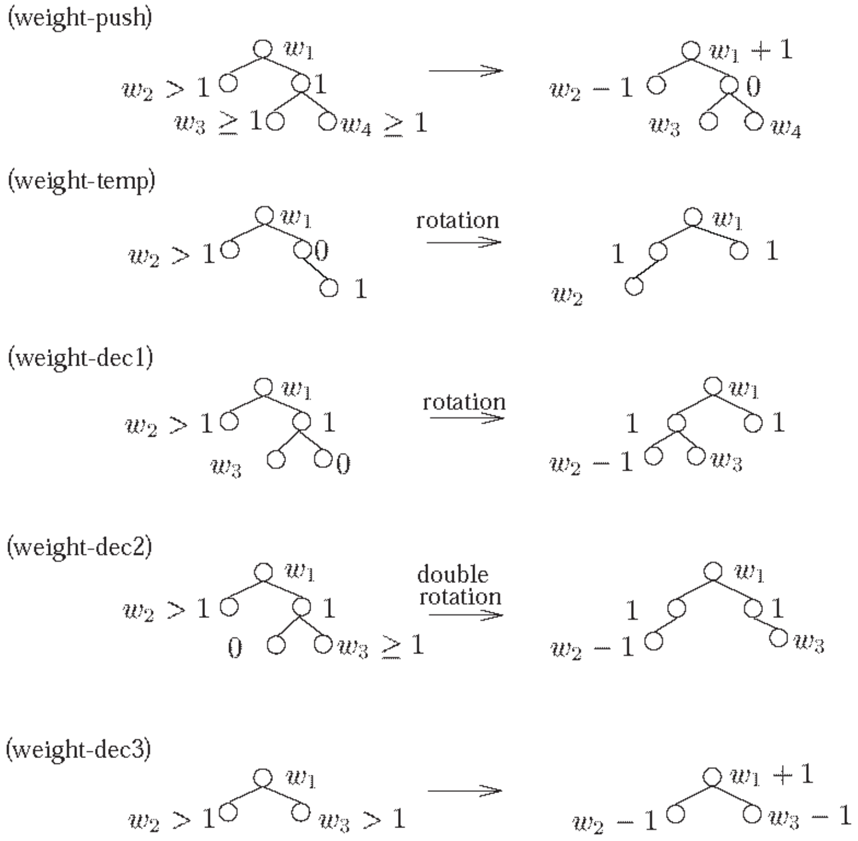
The future direction will involve studies of the O_2-Tree in a distributed programming environment such as the UPC to determine how index structures perform in such systems. Owing to the speed gap between cache access and main-memory access, there have been several research activities to make tree indices cache conscious. A study of cache conscious implementation of the tree index structure should exploit architectural improvements in modern CPU and GPU designs to reduce the impact of memory latency during traversals and thus improve performance of these tree index structures. Lock-free concurrency control algorithms such as STM will also be exploited to reduce the lock overhead, and thus increase the performance gains of the concurrency protocol for the O_2-Tree index structure.

Appendix A

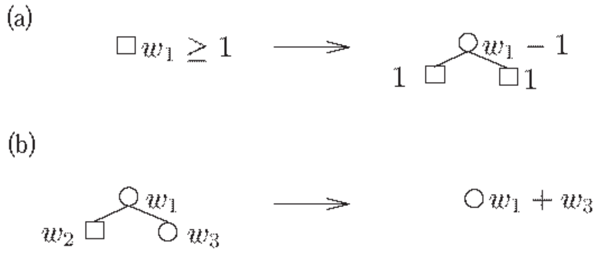
Relaxed Balance Algorithm

The relaxed Balance technique for Red-Black tree as proposed by Larsen (1998) is indicated below. The squares are used to denote leaf-nodes while circles denote general nodes(leaf or internal). The weight label 0 is for Red nodes,and 1 is for Black nodes. Overweighted nodes are denoted by > 1. We present the left cases. The right cases are however symmetric.

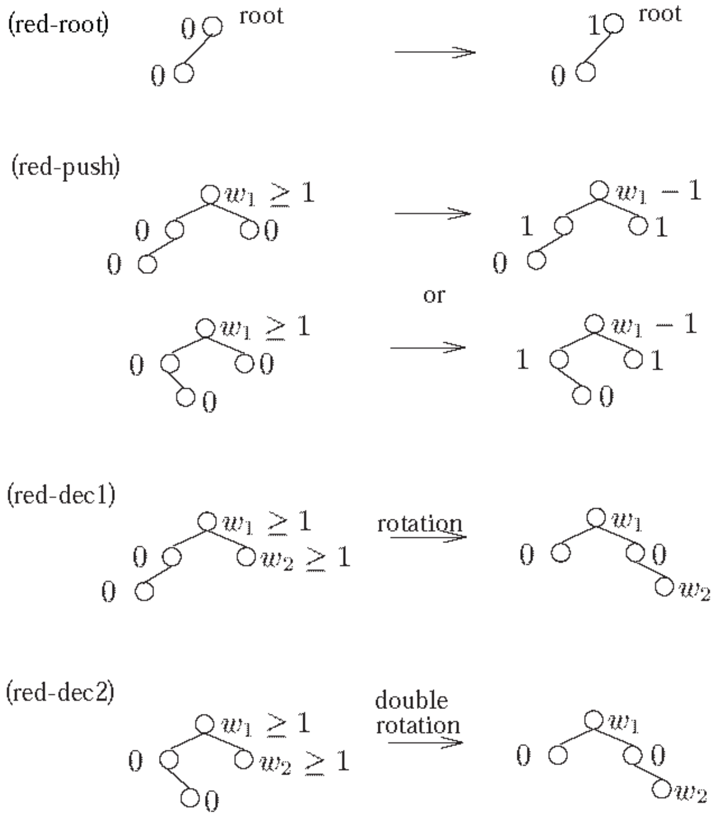
Handling a overweight conflict:



(a) Insertion and (b) Deletion of an item in a chromatic tree:



Handling a red-red conflict:



References

10gen, Inc (2011), 'MongoDB: High-Performance, Open Source NoSQL Database', `http://www.mongodb.org//`.

Adelson-Velskii, G. & Landis, E. M. (1962), 'An Algorithm for the Organization of Information', *Doklady Akademii Nauk USSR* **16**, 263–266.

Agrawal, R., Carey, M. J. & Livny, M. (1987), 'Concurrency Control Performance Modeling: Alternatives and Implications', *ACM Transactions on Database Systems* **12**, 609–654.

Ammann, A. C., Hanrahan, M. & Krishnamurthy, R. (1985), Design of a Memory Resident DBMS, *in* 'Proceedings of IEEE COMPCON Conference', Los Alamitos, CA, pp. 54–58.

Bayer, R. & McCreight, E. M. (1972), 'Organization and Maintenance of Large Ordered Indices', *Acta Informatica* **1**, 173–189.

Bayer, R. & Schkolnick, M. (1988), Readings in database systems, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, chapter Concurrency of Operations on B-Trees, pp. 129–139.

Bell Labs (1996), 'The Dali Main-Memory Storage Manager', `http://www.bell-labs.com/project/dali/`.

Bitton, D., Hanrahan, M. & Turbyfill, C. (1987), Performance of Complex Queries in Main Memory Database Systems, *in* 'Proceedings of the Third International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 72–81.

Bohannon, P., Lieuwen, D., Rastogi, R., Silberschatz, A., Seshadri, S. & Sudarshan, S. (1997), 'The Architecture of the *Dali* Main-Memory Storage Manager', *Multimedia Tools Appl.* **4**, 115–151.

Boyar, J., Fagerberg, R. & Larsen, K. S. (1995), Amortization Results for Chromatic Search Trees, with an Application to Priority Queues, *in* 'Proceedings of the 4th International Workshop on Algorithms and Data Structures', WADS '95, Springer-Verlag, London, UK, pp. 270–281.

Boyar, J. & Larsen, K. S. (1994), 'Efficient Rebalancing of Chromatic Search Trees', *Journal of Computer and System Sciences* **49**(3), 667–682.

Bronson, N. G., Casper, J., Chafi, H. & Olukotun, K. (2010), A Practical Concurrent Binary Search Tree, *in* 'Proceedings of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming', PPoPP '10, ACM, New York, NY, USA, pp. 257–268.

Chilimbi, T. M., Hill, M. D. & Larus, J. R. (2000), 'Making Pointer-Based Data Structures Cache Conscious', *Computer* **33**(12), 67–74.

Comer, D. (1979), 'Ubiquitous B-Tree', *ACM Computing Survey* **11**, 121–137.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009), *Introduction to Algorithms, Third Edition*, The MIT Press.

DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R. & Wood, D. A. (1984), Implementation Techniques for Main Memory Database Systems, *in* 'Proceedings of the 1984 ACM SIGMOD international conference on Management of data', SIGMOD '84, ACM, New York, NY, USA, pp. 1–8.

Drozdek, A. (2001), *Data Structures and Algorithms in C++* , second edn, Brook/Cole ,Thomson Press, Brook/Cole,USA.

Ellis, C. S. (1980), 'Concurrent Search and Insertion in AVL-Trees', *IEEE Transactions on Computers* **29**, 811–817.

FAL Labs (2012), 'Fundamental Specifications of Kyoto Cabinet', `http://fallabs.com/kyotocabinet/spex.html`.

FastDB (2012), 'Main Memory Relational Database Management System', `http://www.garret.ru/fastdb.html`.

Fraser, K. (2003), Practical Lock Freedom, PhD thesis, Cambridge University Computer Laboratory, Cambridge CB3 0FD, UK.

Garcia-Molina, H. & Salem, K. (1992), 'Main Memory Database Systems: An Overview', *IEEE Trans. on Knowl. and Data Engineering* **4**, 509–516.

Google.com (2011), 'LevelDB: A Fast key-value Storage Library written at Google', `http://code.google.com/p/leveldb/`.

Graefe, G. & Larson, P. (2001), B-Tree Indexes and CPU Caches, *in* 'Proceedings of the 17th International Conference on Data Engineering', IEEE Computer Society, Washington, DC, USA, pp. 349–358.

Gray, J. (1978), Notes on Data Base Operating Systems, *in* 'Operating Systems, An Advanced Course', Springer-Verlag, London, UK, pp. 393–481.

Guibas, L. J. & Sedgewick, R. (1978), A Dichromatic Framework for Balanced Trees, *in* 'Proceedings of the 19th Annual Symposium on Foundations of Computer Science', IEEE Computer Society, Washington, DC, USA, pp. 8–21.

Gupta, A. & Chen, H.-Y. (1999), Recovery System for Main-Memory Databases, Technical report, Madison – 53706, WI.

Gupta, G. & Srinivasan, B. (1986), 'Approximate Storage Utilization of B-Trees', *Information Processing Letters* **22**(5), 243 – 246.

Hanke, S. (1998), The Performance of Concurrent Red-Black Tree Algorithms, Technical report.

Hanke, S. (1999), The Performance of Concurrent Red-Black Tree Algorithms, *in* 'Proceedings of the 3rd International Workshop on Algorithm Engineering', WAE '99, Springer-Verlag, London, UK, pp. 286–300.

Hanke, S., Ottmann, T. & Soisalon-Soininen, E. (1997), Relaxed Balanced Red-Black Trees, *in* 'Proceedings of the Third Italian Conference on Algorithms and Complexity', CIAC '97, Springer-Verlag, London, UK, pp. 193–204.

Hankins, R. A. & Patel, J. M. (2003), 'Effect of node size on the Performance of Cache-conscious B$^+$-Trees', *SIGMETRICS Perform. Eval. Rev.* **31**(1), 283–294.

Herlihy, M. P. & Wing, J. M. (1990), 'Linearizability: A Correctness Condition for Concurrent objects', *ACM Transactions on Programming Languages and Systems* **12**(3), 463–492.

Kessels, J. L. W. (1983), 'On-the-fly Optimization of Data Structures', *Communications of the ACM* **26**, 895–901.

Kim, C., Chhugani, J., Satish, N., Sedlar, E., Nguyen, A. D., Kaldewey, T., Lee, V. W., Brandt, S. A. & Dubey, P. (2010), FAST: Fast Architecture Sensitive Tree Search on modern CPUs and GPUs, *in* 'Proceedings of the 2010 international conference on Management of data', SIGMOD '10, ACM, New York, NY, USA, pp. 339–350.

Kong-Rim, C. & Kyung-Chang, K. (1996), T*-Tree: A Main Memory Database Index Structure for Real Time Applications, *in* 'Proceedings of IEEE Real-Time Computing Systems and Applications', South Korea, pp. 81 – 84.

Kung, H. T. & Robinson, J. T. (1981), 'On Optimistic Methods for Concurrency Control', *ACM Transactions Database Systems* **6**, 213–226.

Larsen, K. S. (1998), 'Amortized Constant Relaxed Rebalancing Using Standard Rotations', *Acta Informatica* **35**, 35–10.

Larson, P., Blanas, S., Diaconu, C., Freedman, C., Patel, J. M. & Zwilling, M. (2011), 'High-Performance Concurrency Control Mechanisms for Main-Memory Databases', *Proceedings of VLDB Endowment* **5**(4), 298–309.

Lee, I., goo Lee, S. & Shim, J. (2011), 'Making T-Trees Cache Conscious on Commodity Microprocessors', *J. Inf. Sci. Eng.* **27**(1), 143–161.

Lee, I., Shim, J., Lee, S.-g. & Chun, J. (2007), CST-Trees: Cache Sensitive T-Trees, *in* 'Proceedings of the 12th international conference on Database systems for advanced applications', DASFAA'07, Springer-Verlag, Berlin, Heidelberg, pp. 398–409.

Lee, S.-S. & Yoon, Y.-I. (1997), An Index Recovery Method for Real-Time DBMS in Client-server Architecture, *in* 'Proceedings of the 4th International Workshop on Real-Time Computing Systems and Applications', RTCSA '97, IEEE Computer Society, Washington, DC, USA, pp. 110–.

Lehman, T. J. & Carey, M. J. (1986), A Study of Index Structures for Main Memory Database Management Systems, *in* 'Proceedings of the 12th International Conference on Very Large Data Bases', VLDB '86, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 294–303.

Lehman, T. J., Shekita, E. J. & Cabrera, L. F. (1992), 'An Evaluation of Starburst's Memory Resident Storage Component', *IEEE Trans. on Knowl. and Data Eng.* **4**, 555–566.

Leung, C. H. (1984), 'Approximate Storage Utilisation of B-Trees: A simple Derivation and Generalisations', *Information Processing Letters* **19**(4), 199–201.

Lindström, J. (2007), T^{link}-Tree: Main Memory Index Structure with Concurrency Control and Recovery, *in* 'Proceedings of the third conference on IASTED International Conference: Advances in Computer Science and Technology', ACST'07, ACTA Press, Anaheim, CA, USA, pp. 533–538.

Lu, H., Ng, Y. & Tian, Z. (2000), T-Tree or B-Tree: Main Memory Database Index Structure Revisited, *in* 'Proceedings of the Australasian Database Conference', ADC '00, IEEE Computer Society, Washington, DC, USA, pp. 65– 73.

Manegold, S., Boncz, P. A. & Kersten, M. L. (2000), 'Optimizing Database Architecture for the new Bottleneck: Memory Access', *The VLDB Journal* **9**(3), 231–246.

Manegold, S., Boncz, P. & Kersten, M. L. (2002), Generic Database Cost models for Hierarchical Memory Systems, *in* 'Proceedings of the 28th international conference on Very Large Data Bases', VLDB '02, VLDB Endowment, pp. 191–202.

Marcus, A. (2012), 'The Architecture of Open Source Applications: Chapter 13. The NoSQL Ecosystem', `http://www.aosabook.org/en/nosql.html`.

Mellor-Crummey, J. M. & Scott, M. L. (1991), 'Scalable Reader-writer Synchronization for Shared-memory Multiprocessors', *ACM SIGPLAN Notices* **26**(7), 106–113.

MonetDB (2012), 'Architecture Overview', `http://www.monetdb.org/Documentation/Manuals/MonetDB/Architecture`.

Nurmi, O. & Soisalon-Soininen, E. (1991), Uncoupling Updating and Rebalancing in Chromatic Binary Search trees, *in* 'Proceedings of the tenth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems', PODS '91, ACM, New York, NY, USA, pp. 192–198.

Nurmi, O. & Soisalon-Soininen, E. (1996), 'Chromatic Binary Search trees: A Structure for Concurrent Rebalancing', *Acta Informatica* **33**, 547–557. 10.1007/BF03036462.

Nurmi, O., Soisalon-Soininen, E. & Wood, D. (1987), Concurrency Control in Database Structures with Relaxed Balance, *in* 'Proceedings of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems', PODS '87, ACM, New York, NY, USA, pp. 170–176.

Oracle (2006), 'Oracle TimesTen In-Memory Database Architectural Overview', `http://www.oracle.com/us/products/database/timesten/overview/index.html`.

Oracle (2009), Extreme Performance Using Oracle TimesTen In-Memory Database , *in* 'An Oracle White Paper', Oracle Corporation, pp. 3– 25.

Oracle (2011), *Programmer's Reference Guide*, Oracle BerkeleyDB, Oracle Corporation,USA.

Oracle (2012*a*), 'MySQL Cluster CGE', `http://mysql.com/products/cluster/`.

Oracle (2012*b*), 'Oracle Exadata Database Machine X2-8 DataSheet', `http://www.oracle.com/us/products/database/exadata/database-machine-x2-8/overview/index.html`.

Oracle.com (2011), 'Oracle BerkeleyDB 11g', `http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html`.

Qiaoyu, L., Jianwei, L. & Yubin, X. (2010), Performance Analysis of Data Organization of the Real-Time Memory Database Based on Red-Black Tree, *in* 'Proceedings of the 2010 International Conference on Computing, Control and Industrial Engineering - Volume 01', CCIE '10, IEEE Computer Society, Washington, DC, USA, pp. 428–430.

Rao, J. & Ross, K. A. (1999), Cache Conscious Indexing for Decision-Support in Main Memory, *in* 'Proceedings of the 25th International Conference on Very Large Data Bases', VLDB '99, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 78–89.

Rao, J. & Ross, K. A. (2000), Making B^+-Trees Cache Conscious in Main Memory, *in* 'Proceedings of the 2000 ACM SIGMOD international conference on Management of data', SIGMOD '00, ACM, New York, NY, USA, pp. 475–486.

Saikkonen, R. & Soisalon-Soininen, E. (2008), Cache-sensitive Memory Layout for Binary Trees, *in* G. Ausiello, J. Karhumäki, G. Mauri & L. Ong, eds, 'Fifth Ifip International Conference On Theoretical Computer Science – Tcs 2008', Vol. 273 of *IFIP International Federation for Information Processing*, Springer Boston, pp. 241–255.

Sedgewick, R. (2008), 'Left Leaning Red-Black Tree ', http://www.cs.princeton.edu/~rs/talks/LLRB/.

Seltzer, M. & Bostic, K. (2012), 'The Architecture of Open Source Applications: Chapter 4. BerkeleyDB', http://www.aosabook.org/en/bdb.htm.

Sponsored by VMWARE (2011), 'Redis: An Open Source, Advanced key-value Store', http://redis.io/.

Srinivasan, V. & Carey, M. J. (1993), 'Performance of B$^+$-Tree Concurrency Control Algorithms', *The VLDB Journal* **2**, 361–406.

T., R. & Runger, G. (2010), *Parallel Programming For Multicore and Cluster Systems*, first edn, Springer, Springer-Verlag Berlin Heidelberg.

TPC-H Benchmark: (2011), 'Transaction Processing Council', http://www.tpc.org/tpch/.

Weiss, M. A. (1993), *Data Structures and Problem Solving Using C++*, second edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Weiss, M. A. (1998), *Data Structures and Algorithm Analysis in C++*, second edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Williams, A. (2008), 'Boost Documentation', http://www.boost.org/doc/libs/1_48_0/doc/html/thread.html.