EAI Endorsed Transactions
on Scalable Information Systems
Research Article ICST.ORG

Concurrent Operations of O_2-Tree on Shared Memory Multicore Architectures

Daniel Ohene-Kwofie[1,†], E. J. Otoo[1,*], Gideon Nimako[2]

[1]School Of Electrical and Information Engineering, University Of the Witwatersrand, South Africa
[2]School of Public Health, University Of the Witwatersrand, South Africa

Abstract

Modern computer architectures provide high performance computing capability by having multiple CPU cores. Such systems are also typically associated with very large main-memory capacities, thereby allowing them to be used for fast processing of in-memory database applications. However, most of the concurrency control mechanism associated with the index structures of these memory resident databases do not scale well, under high transaction rates. This paper presents the O2-Tree, a fast main memory resident index, which is also highly scalable and tolerant of high transaction rates in a concurrent environment using the relaxed balancing tree algorithm. The O2-Tree is a modified Red-Black tree in which the leaf nodes are formed into blocks that hold key-value pairs, while each internal node stores a single key that results from splitting leaf nodes. Multi-threaded concurrent manipulation of the O2-Tree outperforms popular NoSQL based key-value stores considered in this paper.

Keywords: Pessimistic Concurrency, Indexing, In-Memory Databases,Performance, Algorithms

Received on 14 September 2013, accepted on 16 February 2014, published on 27 May 2014

doi: 10.4108/ sis.1.3.e 6

1. Introduction

Indexes in database managements system (DBMS) facilitate fast query processing. Tree structured indexes in particular, are critical to database processing systems since they allow for both random and range query processing. Today's data processing tasks in transaction processing, scientific data management, financial analysis, network monitoring, data analytics, etc., handle large volumes of data which require fast accesses and very high throughput.

Recent advances in memory architectures, with 64-bit addressing, now allow for memory sizes of the order of hundreds of gigabytes and beyond at a reasonable cost. It is, therefore, 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 X3-8 system ships with 4TB of main memory [25]. This has, therefore, motivated much research to exploit memory as well as the many-cores available on such architectures to provide fast application processing for main-memory databases.

Recently, there has been a flood of developments and implementations of in-memory data stores with associated index schemes. These are characterised in general as *NoSQL* databases. They are also referred to as *key-value* pair index structures [21]. Notably in this pack are index schemes such as BerkeleyDB [26], LevelDB [12], Kyoto Cabinet [10], RedisIO [30] and MongodB [1]. Such in-memory indexes, optimized for in-memory databases and running on multi-core processors, can support very high query processing rates. The challenge with such systems is how to efficiently ensure that the concurrently executing processes are isolated from each other in such an

*Corresponding author. Email: papaotu@gmail.com
†Corresponding author. Email: danielkwofie@gmail.com

environment. Current DBMS typically rely on locking but in a traditional implementation with a separate lock manager, the lock manager becomes a bottleneck and results in much overhead cost, especially at high transaction rates [17].

In this paper we present an in-memory index structure, referred to as O_2-Tree with emphasis on its implementation in a shared memory multi-core architecture. Such achitectures have a common shared memory that can be accessed by multiple programs running concurrently on several cores, as a result of the multiprocessing design capability, on the same node. We address primarily its concurrency control and fault recovery mechanism. The O_2-Tree is essentially a Red-Black Binary Search Tree in which the leaf nodes are data blocks that store multiple records of "key-value" pairs. The internal nodes contain copies of the keys that result from splitting the blocks of the leaf nodes in a manner similar to the B^+-Tree. However, the O_2-Tree is structurally different from the minimal order of a B^+-Tree also called the 2−3-Tree. A 2−3-Tree is a tree in which each internal or non-leaf node has either 2 or 3 children, all leaf nodes are at the same level, and every node may contain 1 or 2 keys. On the other hand, all leaf nodes of the O_2-Tree may not necessary be at the same level but the leaves contain $m \geqslant 2$ keys. One could question whether the AA-Tree [2] could not be used in place of the Red-Black-Tree (or RB-Tree) that the O_2-Tree uses for the internal nodes. The answer is yes and this leads to a generalization of the O_2-Tree where the internal nodes could be organized as an AVL-Tree [**?**], a 2−3-Tree [**?**], an AA-Tree [**?**] or a SkipList [**?**].

In the O_2-Tree, internal nodes are simply binary placeholders or routers to facilitate and guide the tree traversal. The tree index is fault tolerant in the sense that it is easily reconstructed by reading only the lowest key values of each leaf node that is always made persistent. It is inherently persistent and scales well in highly concurrent environment.

We use a pessimistic concurrency control, but allow multiple *readers* to proceed without blocking internal nodes except for leaf nodes where an updater needs to hold a lock. This allow us to reduce the lock overhead due to blocking of concurrent interleaved query operations. We achieve further performance gains by using the following mechanisms; search operations are interleaved using the hand-over-hand (also referred to as lock-coupling) locking technique; and mutations perform rebalancing separately which encompasses smaller fixed sized atomic regions.

We use the relaxed balance algorithm for Red-Black Tree presented by Hanke et al.[13], to maintain the invariants of the O_2-Tree. We have explored and evaluated the O_2-Tree, and done extensive experimental evaluations and comparisons with some of the well known key-value storage schemes, in multi-core environment under high contentions and index workloads. The experiments confirmed that the concurrent O_2-Tree has a superior performance compared to popular NoSQL key-value stores (*Tuple Store category*) which are often used as in-memory database indexes. These include the BerkeleyDB key-value store (BerkeleyDB), the TreeDB of Kyoto Cabinet and Google's LevelDB.

The major contribution being reported in this paper is the development, implementation and comparative experimental tests of the O_2-Tree main memory index structure. This is usable as a *NoSQL* key-value store for database systems that require a high performance concurrent access in shared memory multi-core architectures. We present results which show that the O_2-Tree in-memory index has high scalability in highly shared concurrent environment, and performs comparatively better than most popular *NoSQL* key-value storage schemes.

The remainder of this paper is organised as follows. Section 2 presents the background of our study. In Section 3, we describe the O_2-Tree in-memory index and present our basic algorithms for concurrency control. A mechanism for persistent storage and recovery is presented in Section 4. In Section 5, we describe our experimental setup and report the performance results of the experimental comparative study of the O_2-Tree with representative NoSQL key-value stores. We conclude in Section 6 and give some directions for future work.

2. Related Work

Tree structured index operations are fundamental in database management systems (DBMS). These provide for fast transaction processing in the DBMS. They allow for both efficient random as well as sequential processing of keys and are therefore widely used in DBMS. Recent advances in main memory technology and the availability of configured systems with memory sizes of the order of hundreds of gigabytes and tens of terabytes have motivated several research in developing main memory index schemes [4, 15, 18, 20]. The usage is such that the index of a main memory resident database or a disk-resident database is kept entirely in memory for high transaction throughput. Some of the widely used tree-based index structures include the B^+-Tree, and the T-Tree. However, recently a number of such index-driven databases have emerged under the banner of *NoSQL* databases. *NoSQL* stores consist basically of a key-value pair and and as such these databases are able to scale easily.

The B^+-Tree [3, 8] is one of the well studied and well understood index structure for database systems. It is generally characterised as a multi-way search tree of order m in which each node holds at least $\lceil m/2 \rceil - 1$ and at most $m - 1$ data item. B^+-Tree was specially designed to speed-up index searches on disk-based DBMS. In such DBMS the number of disk accesses to retrieve a record, is proportional to the height h of the tree, where $h \leq \log_{\lceil m/2 \rceil} N$ for a tree of order m or *fanout* of m. B^+-Tree therefore has a significantly low height for a high *fanout*.

An alternative to the B^+-Tree, designed specifically for main-memory indexing, is the T-Tree [18]. It was proposed as the preferred index structure for main-memory databases. Though the T-Trees has less storage overhead than the B^+-Tree, research in [27, 28] has shown that the B^+-Tree is able to efficiently utilise the cache line in modern processors to provide a better performance. Another index structure which has been widely studied is the Red-Black binary Tree (or RB-Tree) [9]. It is noteworthy that in the use of an RB-Tree as main-memory index, each internal node stores a key-value pair while external nodes are represented as NULL values. The RB-Tree provides an efficient scheme for main memory indexing. However, the performance deteriorates as the datasets become very large. This is due to the fact that, the height of the tree increases greatly and hence traversals and restructuring after updates become expensive especially in concurrent environment with high contention. Further, the CPU cache-line is poorly utilised since each node, either internal or leaf, is visited once for a single key-value access.

Restructuring of the RB-Tree after insertions and deletions can be done during the *top-down* traversal before the operation or *bottom-up* after the operation. One would expect that the concurrency control in RB-Tree would be efficiently implemented with *top-down* insertions and deletions algorithms. Unfortunately standard top-down restructuring algorithm, does not scale well with the RB-Tree and other index structures in general. The process of restoring the tree's invariant becomes a bottleneck for 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. The strict standard top-down algorithm limits the amount of concurrency of the index since every update will proceed with several top-down balancing steps before exiting. This difficulty led to the idea of relaxed balance trees [13, 16, 22].

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 [5, 6, 13, 14, 16, 22, 23]. These separate rebalancing operations involve only local changes. Larsen [16], showed that for a relaxed RB-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 RB-Tree has an amortized constant of $O(1)$ [16].

Concurrent control algorithm for relaxed balance tree implementations based on fine-grain read-write locks provide good scalability for tree-indexes. Optimistic concurrency control (OCC) schemes using version numbers are also attractive for concurrency control especially for in-memory index. They naturally allow readers to proceed without locks, and thus avoid the coherence contention inherent in read-write locks. The readers simple read version numbers updated by writers to detect concurrent mutations since readers assume that no mutation will occur during access to 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 goal of a data structure library, and not just simplicity [7]. In practice STM systems also suffer a performance hit relative to fine-grain lock-based systems on small numbers of processors (1 to 4 depending on the application) [7].

In this paper, we present the concurrent operations of the O_2-Tree memory resident index structure that can be used also as a persistent key-value store. It utilizes an in-memory cache for the leaf nodes and a fine-grain relaxed balance concurrent algorithm in a manner similar to the approach in [16]. This effectively allows for greater degree of concurrency in the O_2-Tree. We discuss this in detail in Section 3. The distinctive differences in the T-Tree, B$^+$-Tree, RB-Tree and the O_2-Tree are clearly illustrated in Figure 1. The approach we advocate here where the RB-Tree is used as the memory resident index can easily be generalised to replace the internal RB-Tree with any of the following: a 2-3-Tree, an AA-Tree and a SkipList.

3. The O_2-Tree In-memory Index

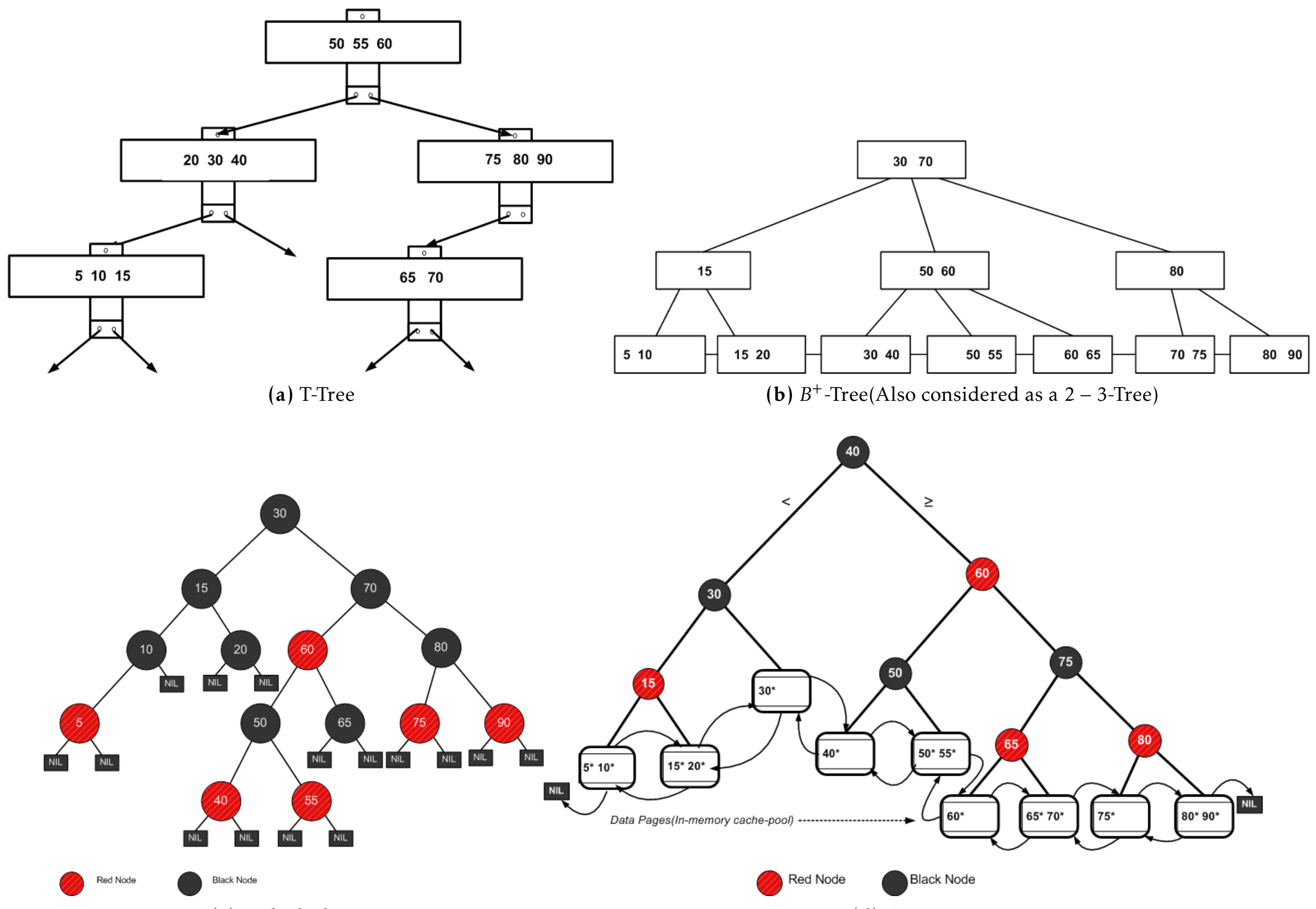
3.1. Structure of the O_2-Tree

The O_2-Tree is basically a binary search tree, managed as a Red-Black Binary-Search Tree, whose leaf nodes are organised as index blocks, data pages, or chunks that store records of "key-value" pairs of the form ⟨key, value⟩. The "value" may also represent a pointer to the location where the record is held in memory. In which case we could also denote it as "⟨key, recptr⟩", where "recptr" denotes the record pointer.

The internal nodes contain copies of only the keys of the middle "key-value" pairs that split the leaf nodes when they become full. These internal nodes are formed into a simple binary search tree that is balanced using the RB-Tree rotation algorithms. Let K_s be the search key and let K_p be key stored at a node p. During a traversal from the root node to a leaf node, a left branch of the node p is followed if $K_s < K_p$ and the right branch is followed if $K_s \geq K_p$. The process continues until a leaf node is reached.

We adopt the RB-Tree balancing algorithm for the O_2-Tree since it is less complex than that of the AVL-Tree which has a more strict balancing condition. The RB-Tree has been widely studied and known for its excellent performance. The O_2-Tree structure, however has a number of advantages over existing indices such as the T-Tree and some of the recent NoSQL key-value stores. The O_2-Tree can easily be reconstructed by reading only the lowest "keys" of each of the leaf nodes. Further, by maintaining only the leaf nodes persistent, the index tree is inherently persistent. The height of the internal RB-Tree is also significantly reduced compared to the situation where each node stores a single "key-value" pair and the entire tree is maintained as a simple RB-Tree. By grouping multiple 'key-value" pairs in the leaf nodes, we optimise the tree so that it also exhibit much better cache sensitivity especially during operations of the leaf nodes. The leaf nodes are therefore able to utilise the cache-line architectural features of the machine, and as such reduce the number of cache misses which would have otherwise resulted from making single node comparison of "key-value" pair. We also achieve significant performance gains by doing single data comparison internally per node during traversal, unlike other structures such as the B$^+$-Tree and the T-Tree that require at most m comparisons.

The *order* of the tree, denoted by m, is the maximum number of "key-value" pairs a leaf node can hold. Data is stored in the leaf nodes; whiles the internal nodes



(**a**) T-Tree (**b**) B^+-Tree(Also considered as a $2-3$-Tree)

(**c**) Red-Black-Tree (**d**) O_2-Tree

Figure 1. Diagram of the various tree structures

are simply binary place holders that facilitate or guide the tree traversal to reach a leaf node. All successful or unsuccessful searches always terminate at a leaf node. This is reminiscent of the search process in a B^+-Tree except that now internal nodes hold only single key values as opposed to m key values. Figure 1d illustrates the schematic layout of the O_2-Tree of order $m = 3$. We show only the keys in the leaf nodes. The corresponding equivalent Red-Black-Tree is shown side-by-side in Figure 1. Detailed explanation of the RB-Tree can be found in [9].

The properties of the O_2-Tree index include all of the RB-Tree [9] properties, plus the following:

1. Each internal node holds a single key value which is a copy of the minimum key value at the leaf node. These keys are equivalent to the middle keys after a leaf node splits.
2. Leaf-nodes are blocks that have between $\lceil m/2 \rceil$ and m "key-value" pairs.
3. 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 values.
4. Leaf nodes are doubly-linked in forward and backward directions. These links provide easy mechanism to traverse the tree in sorted order for key range searches.

We implemented the O_2-Tree index structure 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* subsystem. The BerkeleyDB Mpool subsystem is a general-purpose shared memory buffer pool which can be used for page-oriented, shared and cached file access. The BerkeleyDB Mpool library implementation uses the same on-disk format as its in-memory format as well. This provides a simple mechanism to flush cached pages since a page can be flushed from the cache without format conversion [21]. The internal nodes of the O_2-Tree 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. It should be noted here that this does not constitute an implicit use of BerkeleyDB Tree search. Only the cache functionality of the *Mpool* subsystem is used.

3.2. Some Analytical Results

We state some analytical properties of the O_2-Tree without formal proofs.

Proposition 3.1. In the O_2-Tree, the black leaf-nodes of blocks of "key-value" pairs remain as leaf nodes under all rotations of the internal nodes which are structured as a Red-Black tree.

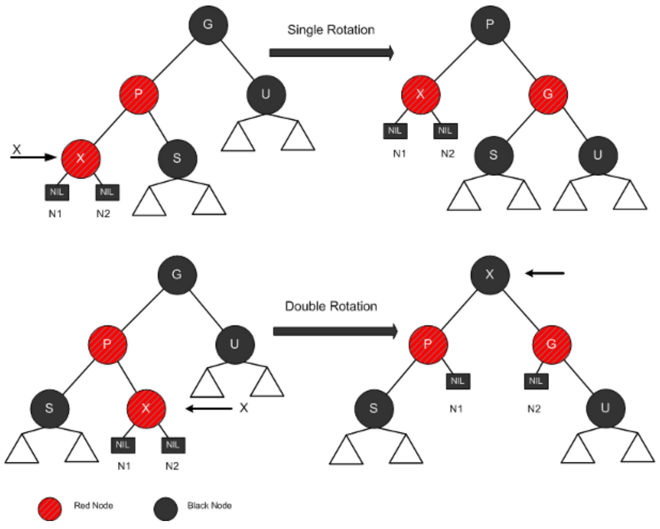


Figure 2. Single and double rotations in a Red-Black Tree

Proof. 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 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. □

Proposition 3.2. An O_2-Tree with n black leaf-nodes will still maintain its n black leaf-nodes after single or double rotations.

Proposition 3.3. The O_2-Tree, supports the query operations of *Put()*, *Delete()*, and *Get()* in time $O(\log_2 N/\lceil m/2\rceil)$, where N is the number of "key-value" pairs in the structure.

Proof. This follows from the fact that the number of leaf-node blocks is at most $n_b = N/\lceil m/2\rceil$. The number of nodes of supporting internal RB-Tree is $n_b - 1$. The height h of the internal RB-Tree is given by $h \leq \log_2 n_b$ [9]. This implies that a search (given by *Get()*), and insertion (given by *Put()*) and a deletion (given by *Delete()*) is each computed in time $O(log_2 N/\lceil m/2\rceil)$. □

Proposition 3.4. Assuming the response set of key-value pairs retrieved in a range search is s, such a range search can be carried out in an O_2-Tree of order m and N key-value pairs in time $O(log_2 N/\lceil m/2\rceil + s)$.

Proof. Given a lower bound k_l and an upper bound k_u values of keys, a range search retrieves the set of key-value pairs whose keys lie in the interval $[k_l, k_u]$. Using the key k_l the search for the leaf node bucket B_l that should contain the key k_l is first retrieved. This takes time $O(\log_2 N/lceil m/2\rceil)$. Once the bucket B_l is retrieved, the forward pointer from this bucket, and all subsequent buckets, can followed to retrieve all the key-value pairs whose their keys are less than k_u. The process stops when the maximum key value k_u is retrieved. The sequential scan performed, retrieves only leaf-buckets that contain s key-value pairs satisfying the range search. The total time to conduct the range search is therefore $O((\log_2 N/lceil m/2\rceil) + s)$. □

3.3. Concurrency Control in the O_2-Tree

We present our concurrent control scheme based on the relaxed balance RB-Tree algorithm by Larsen [16], but we manage our index structure such that the number of restructuring steps after mutation operations is further reduced. To achieve maximum concurrency, we implement the thread-safe algorithm with *page-level* or *node-level* locking. In this case, each node can be locked and unlocked. This simple fine-grain lock-coupling technique ensures that multiple threads can proceed concurrently as long as they don't interfere with each other at the same node. We use three locks as in [23, 24] which we denote as *rlock, wlock, and xlock*. Several user processes can *rlock* a node at the same time, whereas, only one process can *wlock* a node at a time but can coexist with other processes with *rlock* on the same node. *xlock* on the other hand ensures exclusive access to a node and cannot coexist with any other process.

The entire process of handling contentions in the tree is also handled by a rebalancing process which we denote as the *rebalancer*() process and runs in the background. The *rebalancer*() process locates nodes in the tree with conflicts and resolves them appropriately. We adopt the *problem queue* approach to manage contentions instead of random traversal by the *rebalancer*() which could result in several interferences with other query processes and causes degradation in the performance of the index. Let a user operation intending to insert/delete a "key-value" pair be denoted as an *updater()* process. In the problem queue approach, when a lock conflict 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. We implemented a concurrent problem queue to allow for simultaneous *push()* and *pop*() operations such that neither the *rebalancer()* nor the *updater()* processes are blocked while traversing and manipulating the O_2-Tree. An *updater()* appends requests to the *tail* of the *problem queue*, while the *rebalancer*() pops these request from the *head* of the queue. This prevents interference as much as possible and guarantees consistency between *updaters*() and the *rebalancer*() processes. The problem queue is temporarily locked and released both by an *updater()* and a *rebalancer()* during times that they access the problem queue only.

Before presenting the algorithm for the concurrent operations, we first define the following notations. Let T denote an O_2-Tree. The root node will be designated as $Root(T)$ whose parent is the *header* of the index. If z denotes an $Internalnode$ 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 let $z.sibling$ 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., $nodeType \neq leaf$). In addition, $z.key[i]$ and $z.value[i]$ refers to the key and value respectively in the ith position of z given that z is a page block (i.e., $nodeType = leaf$).

Exact-Match Search Algorithm: $Get(x, T)$. The $Get(key\ x)$ function returns the exact-match key-value pair $\langle x, val_x \rangle$ associated with the key x from the data store T, if x exist, otherwise a null value is returned. The search traverses nodes from the root by lock-coupling with $rlocks$ until the the leaf page with the given x is found. Once the leaf-page z, in which the search key x resides, is located, we utilise a binary search function $binarySearch(x, z)$ to locate the "key-value" pair $\langle x, val_x \rangle$ from z. Unlike the T-Tree and the B^+-Tree, the search proceeds with only one key comparison in the internal node. The T-Tree and the B^+-Tree do on the average $m/2$ comparisons before continuing with the search for a given key. The thread-safe search algorithm for the O_2-Tree is given in Algorithm 1.

Range Search Algorithm ($Get\ Next, Get\ Previous$). The range search traverses nodes from the root by lock-coupling with $rlocks$ just as in the exact-match search query. The search begins by locating the minimum key, x_{min} in the 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

Algorithm 1: $Get(key\ x)$

```
Data: $key\ x,\ T$
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 $done$
```

Algorithm 2: Range Scan$(key\ x_{min},\ key\ x_{max})$

```
Data: $key\ x_{min},\ key\ x_{max}$
Result: corresponding key-value for each existing
        key in the range
begin
    $x \longleftarrow x_{min}$
    $node \longleftarrow root(T)$
    $node.rlock()$
    while $node.nodeType \neq leaf$ do
        if $x < node.key$ then
            $node.left.rlock()$
            $node.unlock()$
            $node \leftarrow node.left$
        else
            $node.right.rlock()$
            $node.unlock()$
            $node \leftarrow node.right$
    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 maximum
    key in the range is not encountered
```

range. Once the leaf page with key x_{min} is located, the algorithm proceeds with a sequential scan of the leaf node until the last key in that node is reached. If the maximum key x_{max} is still not located, the scan continues with an *rlocks* 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 algorithm is illustrated in Algorithm 2.

Insert and Update Algorithm: $Put(x, val_x, T)$. 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 *rlock* by other threads on the resource but not another *wlock* or *xlock*. This allows for interleaved $Get()$ operations to overtake $updater()$ operations if necessary and not be blocked. To insert the key-value pair $\langle x, val \rangle$, the leaf page (denoted as *node*) in which the key-value pair belongs is first located. Once the page is located, it is locked exclusively with an *xlock* 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 $splitInsert(x, val_x, node)$ (see Algorithm 4), 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 pair $\langle x, val_x \rangle$ 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 new internal node is coloured *Red*. The tree may grow in height only when a page (leaf-node) overflows. If the operation results in the violation of the invariant condition, the parent of the new parent node is pushed to the problem queue. The thread-safe *Put* algorithm is presented in Algorithm 3.

Delete Algorithm: $Delete(x, T)$. The delete algorithm follows a similar pattern as the insert algorithm. However, the delete may result in page underflow. In

Algorithm 3: Put($key\ x, value\ val_x,\ T$)

```
Data: $key\ x,\ value\ val_x T$
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 $!(node.isfull)$ 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 $done$
```

this case, either key-value pairs $\langle x, val_x \rangle$ are borrowed from adjacent pages (*previous or next pages*) or pages are merged with the leaf-node that underflowed and the other page is deallocated or released into the cache pool. A merger of pages also results in the 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 as given in Algorithm 5.

Algorithm 4: splitInsert($Key\ x,\ value\ val_x,\ O2node$ node)

```
Data: $key\ x,\ value\ val_x, O2node\ node$
Result: $true$ for success $false$ otherwise
begin
    $newPage \longleftarrow new\ leafPage()$
    $newNode \longleftarrow new\ internalNode()$
    $midpoint \longleftarrow \frac{m}{2}$
     where $m$ is the order of the tree */
    $j \longleftarrow 0$
    for $i \leftarrow midpoint$ to $m-1$ do
        $newPage[j++] \longleftarrow node.remove(i)$
    /* insert "key, value" into the appropriate leaf page */
    $newNode.key \longleftarrow newPage.key[0]$
    $newPage.parent \longleftarrow newNode$
    $node.parent \longleftarrow newNode$
     /* reset forward and backward links of leaf nodes */
```

3.4. Correctness

The concurrent protocol presented guarantees linearisability as well as deadlock freedom. 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. This ensures freedom from deadlock. For instance, a request by one thread for a lock on a child node can only be granted after a lock request on the parent node has been granted. Each critical region preserves the binary search tree property. The lock ordering ensures that there is no deadlock cycle loop where a thread, T_1 waits on a lock by another thread, T_2 whiles T_2 waits on a lock held by T_1. Since no such loop exists in the tree structure, and all parent-child relationships are protected by the required locks to make them consistent, the concurrent protocol algorithm is deadlock free.

In order for the algorithms to behave as expected in a concurrent environment, they require that their implementations be linearisable. 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

Algorithm 5: Delete($key\ x,\ T$)

```
Data: $key\ x\ T$
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 */
            $done \leftarrow mergeLeaf(node, node.sibling)$
        $sibling.unlock()$
    $parent.unlock()$
    $node.unlock()$
    return $done$
```

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. More over, each critical region during a mutation operation, only changes child and parent links after acquiring all of the required locks, hence guaranteeing the atomicity of the transaction.

3.5. Storage Utilisation

The expected storage utilisation the O_2-Tree, 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 [19]. 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 utilisation 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

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

4. Persistence and Recovery

A major concern with main-memory databases and and their memory resident indexes is the guarantee of the database persistence, recovery and fault-tolerance. 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 and time consuming. An expensive and time consuming recovery index technique will obviously become a bottleneck in the overall performance of the database. Fast recovery mechanisms 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, check-pointing and reloading techniques are employed. Logging maintains a log of transactions that occur during normal execution, whereas check-pointing takes a snapshot of the database periodically and copies it onto persistent storage for backup purposes. After a system failure, the persistent copy of the database is reloaded into main memory. The indexes are rebuilt and the database is then restored to a consistent state by applying information in the undo and redo logs to the reloaded copy.

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. The O_2-Tree in-memory key-value store ensures persistence by accessing the leaf-pages through the in-memory cache pool. A separate thread periodically flushes dirty pages to the persistent store asynchronously.

The O_2-Tree persistent store provides an efficient and simply approach for index recovery. The reason being that rebuilding the index structure of the O_2-Tree from persistent store, unlike the $B^+ - Tree$ and the 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. In systems where the index data is too large to fit into available memory, pages are paged-in and paged-out of the in-memory cache using a cache replacement policy such as the least recently-used protocol. In addition, bulk-loading the index from the persistent pages provides a much faster approach to restoration as the amount of restructuring is minimal.

Besides storing the leaf-pages by a background process, such that the entire RB-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 RB-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 RB-Tree can be restored from the persistent store.

5. Performance Evaluation

We evaluated the performance of the O_2-Tree index as a key-value persistent store, on the Intel Xeon E5630 CPU machine. We enabled hyper-threading for all performance evaluations. We conducted all the implementations and code compilation with the GNU GCC/G++ compiler on a 64-bit machine having a *72GB* of RAM and running the Scientific Linux release 5.4 Operating system. We generated 32-bit uniform distributed keys with which we formed key-value pairs where the values were also uniform random generated values. We also performed some experiments with live data read from the flight statistics datasets [11] as well as the records of the *Order* table generated from the TPC-H dbgen data generator [31].

5.1. Sequential Evaluations

For completeness, we present the comparative results of the performance of the O_2-Tree with the basic index structures such as the B^+-Tree, T-Tree, AVL-Tree, and the Top-Down Red-Black-Tree. Figure 3 shows the performance of the five data structures for a simple build of the index. We performed up to 50M unique key insertions. The *order* of the T-Tree, B^+-Tree, and the O_2-Tree used was $m = 512$. The graphs show the times for building the respective data structures in memory. As can be observed from the graphs, T-Tree performed worst among the index structures considered while the O_2-Tree had the best performance. 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 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 from the simple fact that the B^+-Tree makes multi-way-decision during its traversal down the tree while the O_2-Tree makes single data comparison to determine the search path during traversal. Splitting and redistribution of keys in the nodes of a B^+-Tree may continue all the way to the root of the tree. In the O_2-Tree only colour changes may propagate to the top. These operations are less expensive compared to the splitting and redistribution in B^+-Trees. The poor performance of the T-Tree and the B^+-Tree, compared to the O_2-Tree, is due to the fact that several data comparisons are required to locate a bounding node for child node to be visited during traversals.

Figure 4 illustrates the performance evaluation of these structures when subjected to interleaved mix of insertions, deletions and searches with different percentage of each operation for a total of 50 million (50M) query operations with a single thread. Figure 5 shows the operational throughput (*query operations per second*) for varying workloads with 50% updates. The query mix operations involved generating either an update (insertion or deletion) and conducting a lookup with varying probabilities. We refer to the probability of generating an update multiplied by 100 as the update ratio. Thus, a 0% update ratio indicates only data look-ups whiles a 100% update ratio indicates only update operations. Each update ratio consist of 30% deletions and 70% insertions. The preliminary results from the graphs indicates that the O_2-Tree clearly outperforms all the basic structures considered.

We however observed that for lower update ratios such as 0% and 10% , the T-Tree and the B^+-Tree provided better and comparable performance to the O_2-Tree. Figures 5 and 6 show the operational throughput (*query operations per second*) for varying workloads with 50% and 100% updates respectively. We observed a general decline in throughput as the workload and update ratios increase. However, the O_2-Tree index provided the best operational throughput in all cases.

We also present the results with the single threaded persistent implementation of the O_2-Tree index structure, where the key values and their associated data are kept persistent through an in-memory cache, with some NoSQL(key-value) data stores such as the BerkeleyDB [29], the Kyoto Cabinet TreeDB [10] and LevelDB [12]. These experiments were conducted primarily to compare the O_2-Tree index structure to other popular NoSQL(key-value) data stores that use tree structured access methods and have been reported in the literature as being extremely fast. The data operations are performed with the leaf nodes read and written through the memory resident cache. The leaf nodes are then written to disk using the Least Recently Used (LRU) cache replacement algorithm. At the end of a session, the cache is flushed so that all memory resident leaf-nodes are written to disk and the overall time to complete the operation is reported. We repeated this for varying data sizes.

We refer to the persistent O_2-Tree implementation as O_2-Tree-KV. This scheme was compared with other popular NoSQL key-value stores. Each key-value store was initialised with a page size of 4K, as well as a 2.5GB in-memory cache size. We applied the tuning mechanisms to the NoSQL databases as indicated and recommended in their respective documentations. Furthermore, we did not enable compression. The TPC-H generated dataset from the *Order* table was used for all experiments involving the persistent key-value stores.

Our results indicate that the O_2-Tree, using BerkeleyDB Mpool subsystem as a cache, outperformed the BerkeleyDB and the Kyoto Cabinet when using the TreeDB, by several orders of magnitude. However, O_2-Tree-KV performance is very comparable to that of the LevelDB. We actually did run the LevelDB in asynchronous mode in which a separate thread concurrently flushed the cache contents to disk. Even though in comparing the LevelDB and the BerkeleyDB with with O_2-Tree, the O_2-Tree had the disadvantage that it does not have asynchronous back-end persistent store. Our objective however, was to evaluate how the O_2-Tree-KV, in writing to disk through an in-memory cache, compared with these popular industry-standard NoSQL key-value stores without multi-threading.

The O_2-Tree-KV with a cache support however, performed over $5X$ faster than the Kyoto Cabinet when using the TreeDB access method of the Kyoto Cabinet and about $1.5X$ as fast as BerkeleyDB using the *B-Tree* access method. The results are as shown in Figure 7. The second set of experiment with the NoSQL databases, illustrated in Figure 8, show the performance of each key-value store under different mix of queries where the update ratio was varied from 0% to 100%. Again,

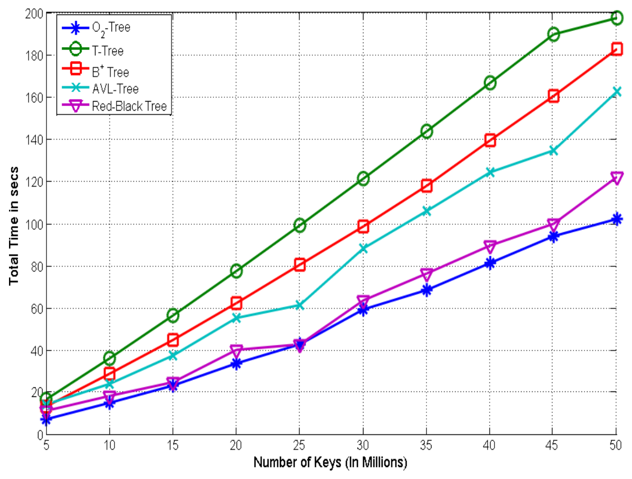


Figure 3. Index build with randomly generated keys

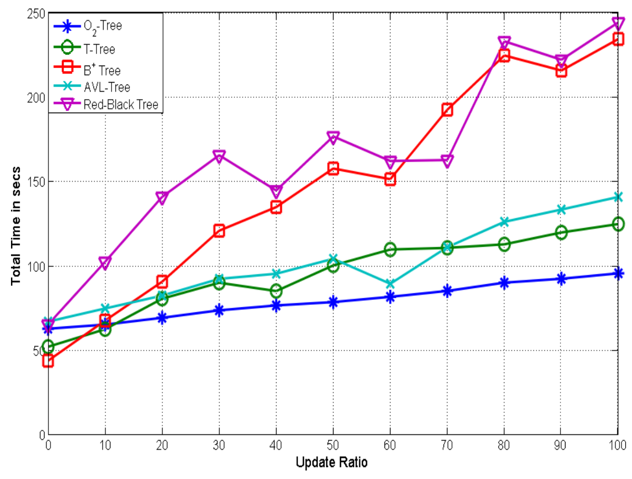


Figure 4. Mixed Operations of Searches, Inserts and Deletes

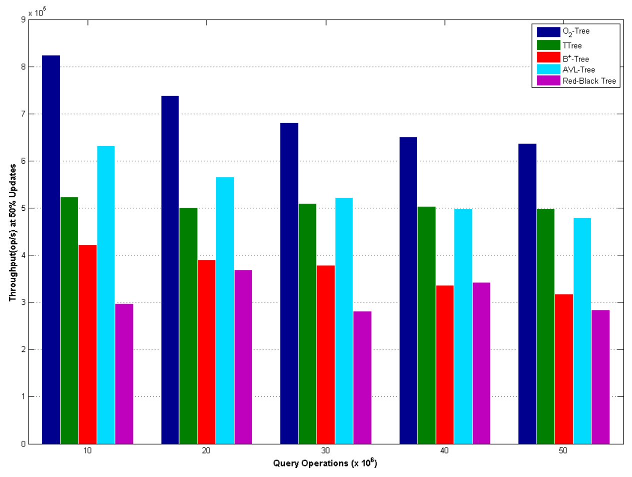


Figure 5. Operational Throughput with 50% Updates for Basic Indexes using TPC Dataset

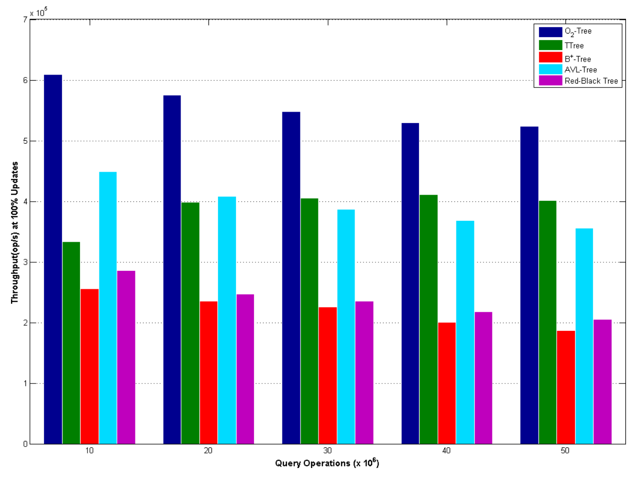


Figure 6. Operational Throughput with 100% Updates for Basic Indexes using TPC Dataset

the O_2-Tree, when reading and writing through a cache shows performance characteristics that are superior to the NoSQL databases considered. It outperformed the LevelDB and the Kyoto Cabinet that used the TreeDB access method, by several orders of magnitude.

Figure 9 and Figure 10 show the corresponding operational throughput results for mixed query operations on the NoSQL data stores with varying data sizes from 10M to 50M operations for 100% and 50% updates respectively. The O_2-Tree-KV demonstrated a superior operational throughput at high update ratio and large dataset. The efficient index mechanism of the O_2-Tree-KV accounts for its superior performance.

5.2. Multi-threaded Evaluations

We evaluated the average time for a multi-threaded insertion of "key-value" pairs of generated data into each of the following storage schemes: the O_2-Tree persistent store, which we refer to as O_2-Tree-KV, the BerkeleyDB and Kyoto-Cabinet TreeDB using the $B-Tree$ access method as well as the LevelDB NoSQL key-value store. These experiments were conducted primarily to compare the performance of O_2-Tree with these key-value stores where the data blocks are written and read through an in-memory cache to a disk file using multiple threads. We evaluated the average time it takes to perform 20 million (20M) insertions of "key-value" pairs concurrently with the number of threads

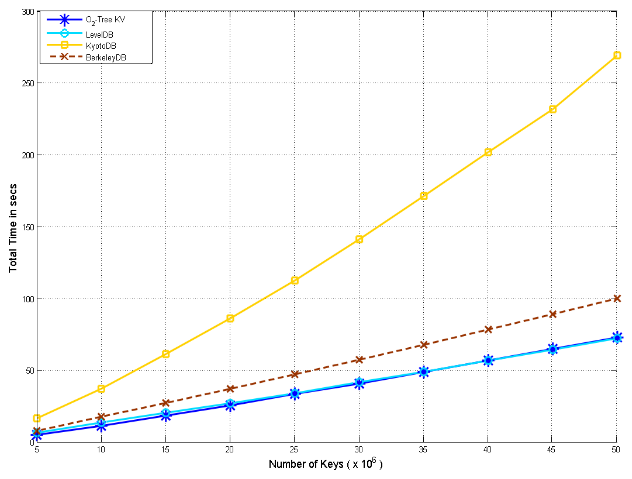


Figure 7. Persistent store: Insertions With In-Memory Cache using TPC dataset

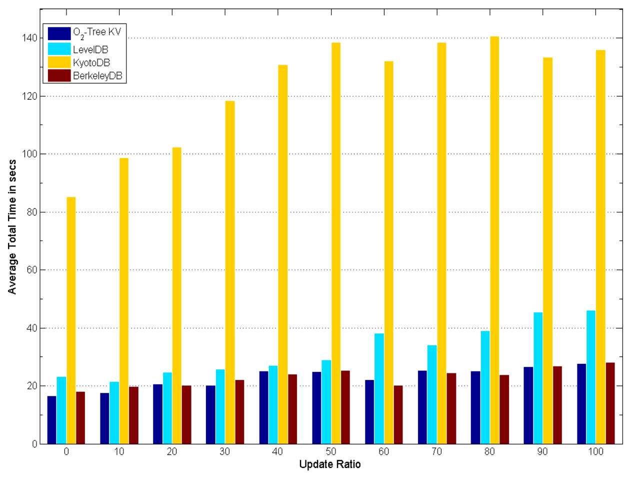


Figure 8. Persistent store: With In-Memory Cache and 50M Mixed Queries

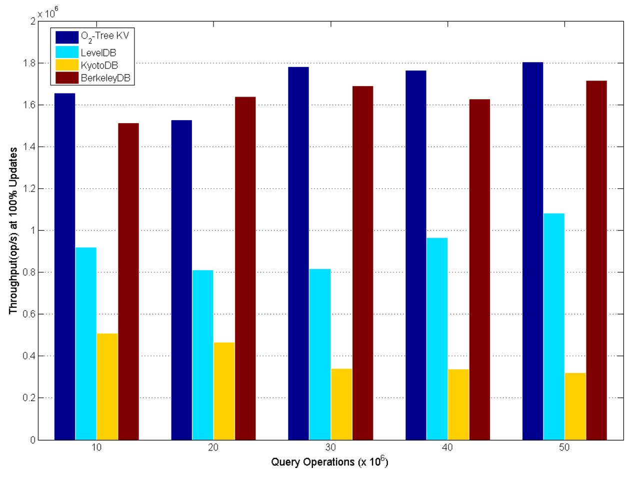


Figure 9. Operational Throughput for Persistent Storage with 100% Updates

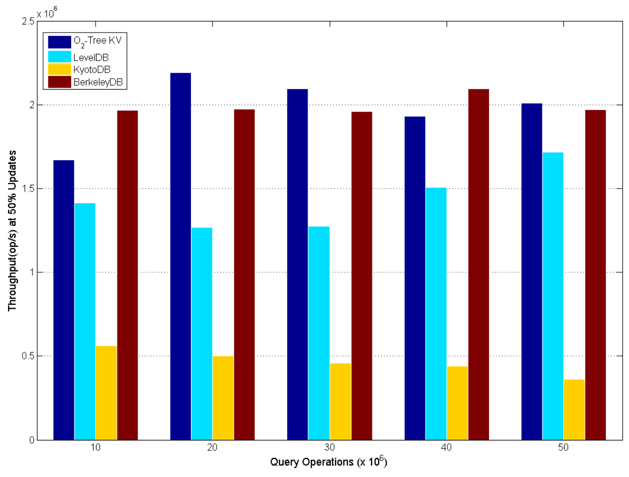


Figure 10. Operational Throughput for Persistent Storage with 50% Updates

varying from 2 to 16. The page size as well as the in-memory cache size for each key-value store was set to 4k and 2.5GB respectively for all experiments in this group. We ensured that the operations were performed with the index tree in memory while the data pages were maintained in the in-memory cache pool. The data pages were read and written to disk according to the *Least Recently Used* (LRU) cache replacement policy of the *Mpool*. The results are shown in Figure 11. The general observation was that the average time to perform insertions decreased with increasing number of threads. This was due to the fact that the degree of access contentions increased as the number of threads increased. Threads must block to ensure correctness of query operations. Therefore, more threads result in lock contention as several threads are blocked. This eventually affects the overall performance of the structures. However, the O_2-Tree-KV performed better than the other "key-value" storage schemes discussed in the paper. The O_2-Tree-KV employs a simple index mechanism which accounts for its better performance. The O_2-Tree-KV, performed about $2 \sim 3X$ faster than the KyotoDB and BerkeleyDB both of which use the B-Tree access method. The results are shown in Figure 11.

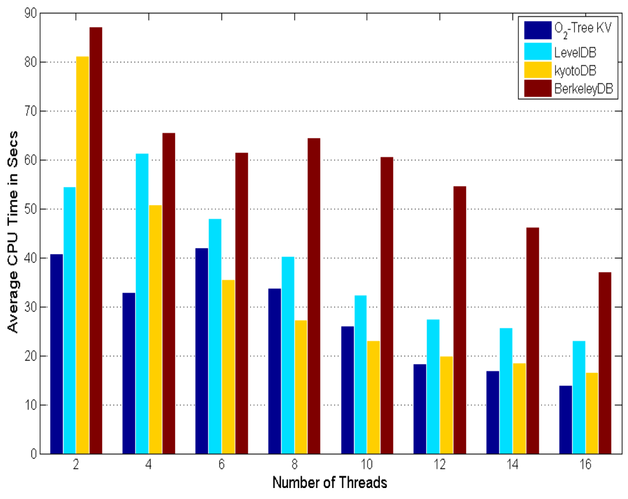


Figure 11. Index Construction with Varying Number of Threads

Figure 12 shows the operational throughputs of each of the key-value stores under different workloads. Each workload consisted of a mix of look-ups, insertions and deletions referred to as update ratio from the previous discussion. For each update ratio, we interleaved all operations such that a thread performed either an update or a lookup. All operations were performed by a maximum 16 threads we had on the machine. We observed a general decrease in throughput as the update ratio increased. This was due to the fact that, updates require restructuring of the index which affects the overall performance. The O_2-Tree-KV did record the highest throughput which was about *1.9M* operations per second (op/s). This rate later dropped to *1.3M op/s* at 100% updates. A similar trend was observed for all the other key-value stores considered.

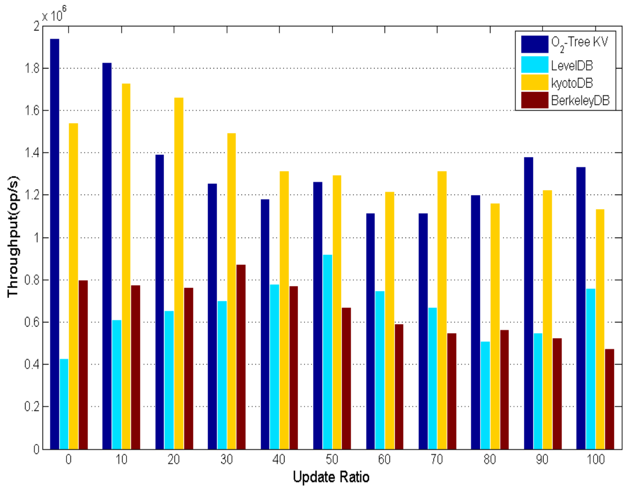


Figure 12. Operational Throughput for Different Mix of Workloads

We also compared the average time to conduct a search or lookup for all key-value stores. One objective of *NoSQL* key-value store is to provide effective lookup without the bottlenecks of traditional Relational database systems (RDBMS). We conducted the experiments with 20M $32-bit$ keys. We gradually increased the number of threads to ascertain the effect of shared memory multi-threaded concurrent access of these different data storage systems. The results show that, as the number of threads increased, the lookups proceeded faster since there was relatively little work per thread. During lookups, threads do not block and thus, can proceed immediately with expected linearisable results. Though the O_2-Tree-KV outperformed all the key-value stores considered, it rather exhibited a poor performance gain as the number of worker threads increased. This could be due to the cache coherence problem associated with single node traversals. We anticipate a much better performance with a lock-free protocol such as Software transactional memory *STM*.

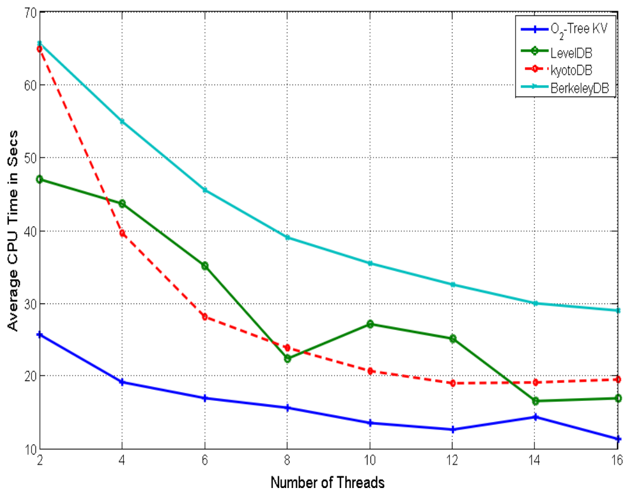


Figure 13. Concurrent Look-ups for 20M "key-value" using Varying Workloads

Additionally, we performed multi-threaded scalability evaluations on the O_2-Tree-KV as well as the BerkeleyDB, Kyoto-Cabinet TreeDB and the LevelDB NoSQL key-value stores. We adopted the *strong scalability* test approach in which we doubled the dataset as well as the number of threads for each run of the experiment. The dataset was varied from 5M with 2 threads and doubled for each run to 40M with 16 threads for the last run.

The first set of scalability test shown in Figure 14 illustrated the results with only insertions (*Puts*). Figure 15 however, indicates similar experiment but this time for a mix of query operations in which 50% were look-ups and 50% updates (of which 70% were insertions and 30% deletions). We observed a comparable and even better performance for the O_2-Tree-KV which exhibited a high level of scalability. Generally, a gradual increase in CPU times for all the key-value stores considered was observed as the number of threads and datasets were doubled.

We also evaluated the total size of the *problem queue* which is used by the relax balance algorithm of the O_2-Tree. We varied the data size as well as the number of threads in each run of the experiment. We observed that the total problem queue size was a function of the size of the dataset used to build the index. Large datasets resulted in larger problem queue size. Figure 16 shows the graph for the total problem queue sizes for the tree index. We observe a rapid increase in the problem queue size given a small increase in the dataset. For instance, the queue increases rapidly from about $6X10^4$ when the dataset is $3M$ to an average of about $12X10^4$ (double the previous size) when the dataset is increased by $1M$.

However, a series of experiments conducted indicated that the average problem queue size was comparatively small at any instance using a single *rebalancer* thread. Since, the *rebalancer* thread does not traverse the index from the root but goes directly to the offending node, it is able to process problem queue items faster than the update threads. This accounts for the minimal average problem queue size observed in the experiment. Further, as the number of the *rebalancer* threads increases, the problem queue size reduces significantly as more threads are able to concurrently process the queue with minimal interference to ensure that the tree is balanced.

Finally, we evaluated the performance of each key-value store using real life flight statistics data [11] that consisted of 32bit keys and their corresponding data values. The physical size of the file was about 600MB. We loaded 10M keys and their corresponding values into each key-value store using varying concurrent threads up to to 16 threads. The operational throughput

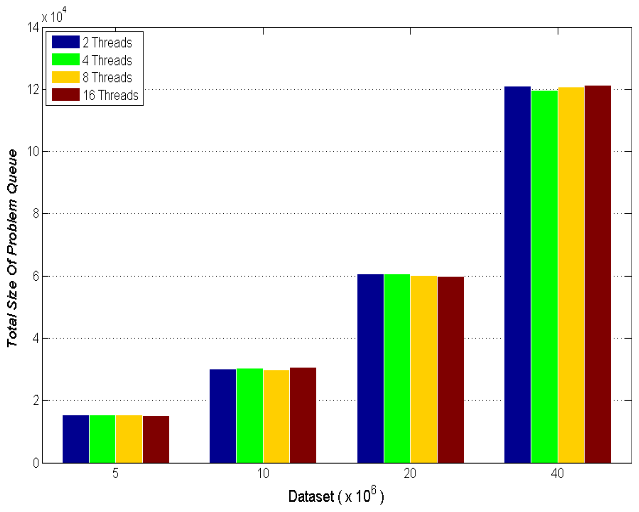


Figure 16. Total *Problem Queue* size for Varying Data sizes and Threads

to load the data from the persistent dump was then reported. The primary objective of this experiment was to measure the performance with real life data besides the synthetic data used in the previous experiments. We observed a comparable performance between all the key-value stores considered. The O_2-Tree-KV exhibited a much better throughput even though the others were comparable. Figure 17 illustrates the results.

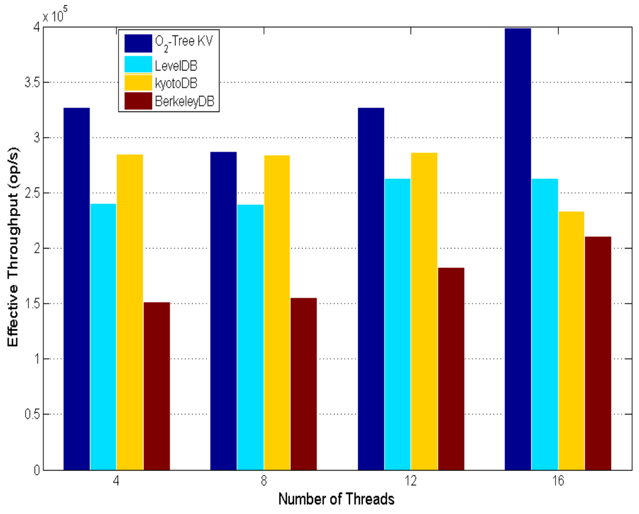


Figure 17. Concurrent Loading of Real-life Persistent data

6. Conclusion and Future Work

In this paper we have presented the O_2-Tree as an in-memory resident index for a persistence key-value store. It delivers high performance and exhibits good

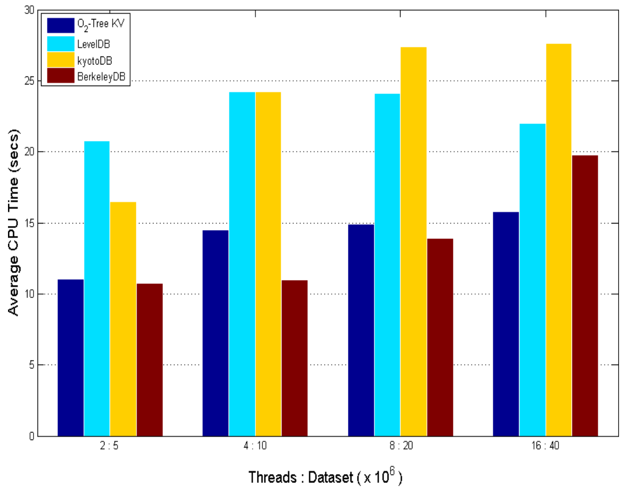


Figure 14. Scalability Test with 100% Insertions

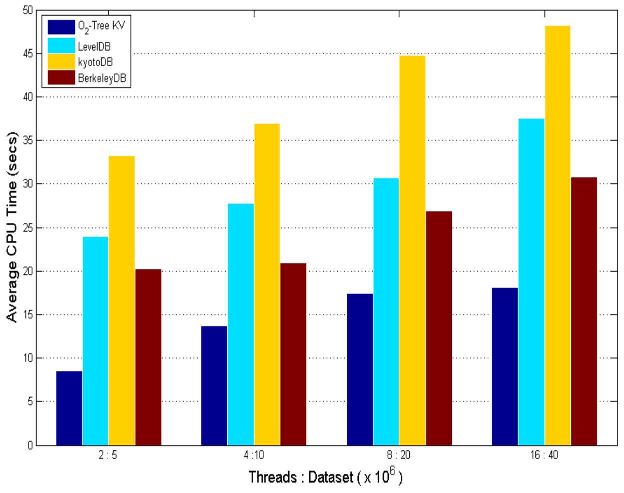


Figure 15. Scalability Test with 50% Update Ratio

scalability while being tolerant of contention. We have also presented a concurrent access protocol based on the relax balance tree technique which allows the scheme to attain high performance as well.

We compared our index persistent O_2-Tree implemented through an in-memory cache against popular high performance and widely used *NoSQL* key-value stores such as the BerkeleyDB, Google's LevelDB and Kyoto-Cabinet TreeDB. Our experiments show that O_2-Tree key-value store outperforms both BerkeleyDB, and Kyoto-Cabinet TreeDB by $2-3X$. It also performs comparatively well against Google's LevelDB for many access patterns. More importantly, the experimental results show that O_2-Tree index structure exhibit a good scalability and tolerates contention. It also exhibit superior performance especially during high updates. It's therefore the index structure of choice in applications with frequent updates such as Online Transaction Processing (OTP).

Future work anticipated involves using optimising techniques to make the structure much more cache aware using blocking techniques to improve CPU cache usage as well as bulk loading techniques for greater throughput. We are also exploring the use of GPU traversals for O_2-Tree for even higher throughput. Other future work include replacing the RB-Tree internal structure of the O_2-Tree with other tree structures such as the *AA-Tree*, 2−3-*Tree* and the *SkipList*, and compare their performance measures.

6.1. Acknowledgements

This work was supported with funding from the Centre For High Performance Computing (CHPC), and the Department of Science and Technology (DST), South Africa.

References

[1] 10gen, Inc (2011) Mongodb: High-performance, open source NoSQL database., http://www.mongodb.org//.

[2] Andersson, A. (1993) Balanced search trees made simple. In *In Proc. 3rd Workshop on Algorithms and Data Structures* (Springer): 60–71.

[3] Bayer, R. and McCreight, E.M. (1972) Organization and Maintenance of Large Ordered Indices. *Acta Inf.* **1**: 173–189.

[4] Bohannon, P. et al. (1997) The Architecture of the DALA Main-Memory Storage Manager . *Multimedia Tools Appl.* **4**: 115–151.

[5] Boyar, J., Fagerberg, R. and 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 (London, UK, UK: Springer-Verlag): 270–281.

[6] Boyar, J. and Larsen, K.S. (1993) Efficient rebalancing of chromatic search trees. *Journal of Computer and System Sciences* **49**: 667–682.

[7] Bronson, N.G., Casper, J., Chafi, H. and Olukotun, K. (2010) A Practical Concurrent Binary Search

tree. In *Proc. of the 15th ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPoPP '10 (New York, NY, USA: ACM): 257–268. doi:http://doi.acm.org/10.1145/1693453.1693488.

[8] Comer, D. (1979) Ubiquitous B-Tree. *ACM Comput. Surv.* **11**: 121–137. doi:http://doi.acm.org/10.1145/356770.356776.

[9] Cormen, T., Leiserson, C., Rivest, L. and Stein, C. (2009) *Introduction to Algorithm*, **1** (Cambridge, Massachusetts London, England: MIT Press), 3rd ed.

[10] FAL Labs (2011) Kyoto cabinet: a straightforward implementation of dbm, http://fallabs.com/kyotocabinet/.

[11] FlightStats: (2005) FlightStats Database, http://dl.flightstats.us/, http://dl.flightstats.us/.

[12] Google.com (2011) Leveldb: A fast key-value storage library written at google, http://code.google.com/p/leveldb/.

[13] Hanke, S., Ottmann, T. and Soisalon-Soininen, E. (1997) Relaxed balanced red-black trees. In *Proc. of the Third Italian Conference on Algorithms and Complexity*, CIAC '97 (London, UK: Springer-Verlag): 193–204.

[14] Hanke, S. (1998) *The Performance of Concurrent Red-Black Tree Algorithms*. Tech. rep.

[15] Kong-Rim, C. and Kyung-Chang, K. (1996) T *-Tree: A Main Memory Database Index Structure for Real Time Applications. In *Proc. IEEE Real-Time Computing Systems and Applications* (South Korea): 81 – 84.

[16] Larsen, K.S. (1998) Amortized constant relaxed rebalancing using standard rotations. *Acta Informatica* **35**: 35–10.

[17] Larson, P.A., Blanas, S., Diaconu, C., Freedman, C., Patel, J.M. and Zwilling, M. (2011) High-performance concurrency control mechanisms for main-memory databases. *Proc. VLDB Endow.* **5**(4): 298–309.

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

[19] Leung, C.H. (1984) Approximate storage utilisation of B-trees: a simple derivation and generalisations. *Inf. Process. Lett.* **19**: 199–201.

[20] Lu, H., Ng, Y.Y. and Tian, Z. (2000) T-Tree or B-Tree: main memory database index structure revisited. In *Proc. of the Australasian Database Conference*, ADC 2000 (Washington, DC, USA: IEEE Computer Society): 65– 73.

[21] Marcus, A. (2012) The architecture of open source applications: Chapter 13. the NoSQL ecosystem, http://www.aosabook.org/en/NoSQL.html.

[22] Nurmi, O., Soisalon-Soininen, E. and Wood, D. (1987) Concurrency control in database structures with relaxed balance. In *Proc. of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, PODS '87 (New York, NY, USA: ACM): 170–176. doi:http://doi.acm.org/10.1145/28659.28677.

[23] Nurmi, O. and 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 (New York, NY, USA: ACM): 192–198.

[24] Nurmi, O. and Soisalon-Soininen, E. (1996) Chromatic binary search trees: A structure for concurrent rebalancing. *Acta Informatica* **33**: 547–557. 10.1007/BF03036462.

[25] Oracle (2013) Oracle exadata database machine x3-8 datasheet, http://www.oracle.com/us/products/database/exadata/database-machine-x3-8/overview/index.html.

[26] Oracle.com (2011) Oracle berkeleydb 11g, http://www.oracle.com/technetwork/products/berkeleydb/overview/index-085366.html.

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

[28] Rao, J. and Ross, K.A. (2000) Making B^+-trees cache conscious in main memory. In *Proc. of the 2000 ACM SIGMOD international conference on Management of data*, SIGMOD '00 (New York, NY, USA: ACM): 475–486. doi:http://doi.acm.org/10.1145/342009.335449.

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

[30] Sponsored by VMWARE (2011) Redis: An open source, advanced key-value store., http://redis.io/.

[31] TPC-H (2001) Transaction Processing Council, http://www.tpc.org/tpch/.