Concurrent Operations of O2-Tree on Shared Memory Multicore Architectures

Daniel Ohene-Kwofie[1] **E. J. Otoo**[1*] **Gideon Nimako**[1+]

[1] School of Computer Science
The University of the Witwatersrand,
South Africa, 2050
Email: {danielkwofie, *papaotu,+gideonnimako}@gmail.com

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, of the order of tens to hundreds of gigabytes, thereby allowing such architectures to be used for fast processing of in-memory databases 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 due to the overhead incurred. This paper presents the O_2-Tree, a fast main memory resident index, which is highly scalable and tolerant of high transaction rates in a concurrent environment using the relaxed balancing tree algorithm. The O_2-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 only a single key that results from splitting leaf nodes in a manner reminiscent of the B^+-Tree. The scheme adopts well to implementing key-value store as in the NoSQL database. Multithreaded concurrent manipulation of the O_2-Tree, in shared memory outperforms popular NoSQL based key-value stores considered in this paper. An added feature of the scheme is its resiliency to system failure since the memory resident index can be restored from the minimum keys in each of the leaf nodes.

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

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 access with 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 X2-8 system ships with 2TB of main memory (Oracle 2012). 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 (Marcus 2012). Notably in this pack are index schemes such as BerkeleyDB (Oracle.com 2011), LevelDB (Google.com 2011), Kyoto Cabinet (FAL Labs 2011), RedisIO (Sponsored by VMWARE 2011) and MongodB (10gen 2011). Such in-memory indexes, optimised 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 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 (Larson et al. 2011).

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. 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. The 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. It is inherently persistent and scales well in highly concurrent environment.

Another alternate approach to fault tolerance is to store the memory resident internal nodes of the O_2-Tree after a session and reload it before a session. A post-order binary tree traversal of the internal nodes can easily dump the nodes and a similar traversal allows for reconstruction of the internal nodes of the tree. During a session, the internal binary tree structure can be occasionally dumped by check-point and supported with logs of operations of insertions and deletions between check-points. The most current dump is pointed to by a header value.

The mechanism of locating the last usable dump is very much like *shadow-paging*. To recover, the most recent check-point dump is loaded and the update logs, since the last check-point dump, is used to restore the internal nodes of the tree into a consistent state.

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 allows 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 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.(Hanke et al. 1997) to maintain the invariants of the O_2-Tree. We have explored and evaluated the O_2-Tree, and done extensive experimental evaluations, in comparison 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. We compare with these specific key-value NoSQL data stores since they can be used in non client-server mode and they allow for their use in both efficient random and range searches. For example the hash-based Kyoto Cabinet has superior performance to the O_2-Tree but performs poorly compared to the O_2-Tree for range searches.

The major contribution being reported in this paper is the development, implementation and comparative experimental tests of a the O_2-Tree main memory index structure 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 the 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 (Lehman & Carey 1986, Kong-Rim & Kyung-Chang 1996, Bohannon, P. et al. 1997, Lu et al. 2000). 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 (Bayer & McCreight 1972, Comer 1979), 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 $\lceil m/2 \rceil$ and at most m 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 (Lehman & Carey 1986). 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 (Rao & Ross 1999, 2000) 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) (Cormen et al. 2009). 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. This fact is exploited to build the O_2-Tree. The otherwise NULL values are used form the leaf-nodes of groups of records of "key-value" pairs.

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. Additionally, the CPU cache-line is poorly utilised since each node including the leaf-nodes are 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 standard strict 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 (Nurmi et al. 1987, Hanke et al. 1997, Larsen 1998).

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 & Larsen 1993, Boyar et al. 1995, Hanke et al. 1997, Hanke 1998, Larsen 1998). These separate rebalancing operations involve only local changes. Larsen (Larsen 1998) 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 $O(1)$ (Larsen 1998).

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 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 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) (Lourenço et al. 2009), 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). 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).

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, as provided by the BerkeleyDB Mpool subsytem, for the leaf nodes and a fine-grained relaxed balance concurrent algorithm in a manner similar to the approach in (Larsen 1998). 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.

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 into index blocks, data pages, or chunks that store the 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 this 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 the bounding 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. 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.

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 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 1c. Detailed explanation of the RB-Tree can be found in (Cormen et al. 2009).

The properties of the O_2-Tree index include all of the RB-Tree (Cormen et al. 2009) properties, plus the following:

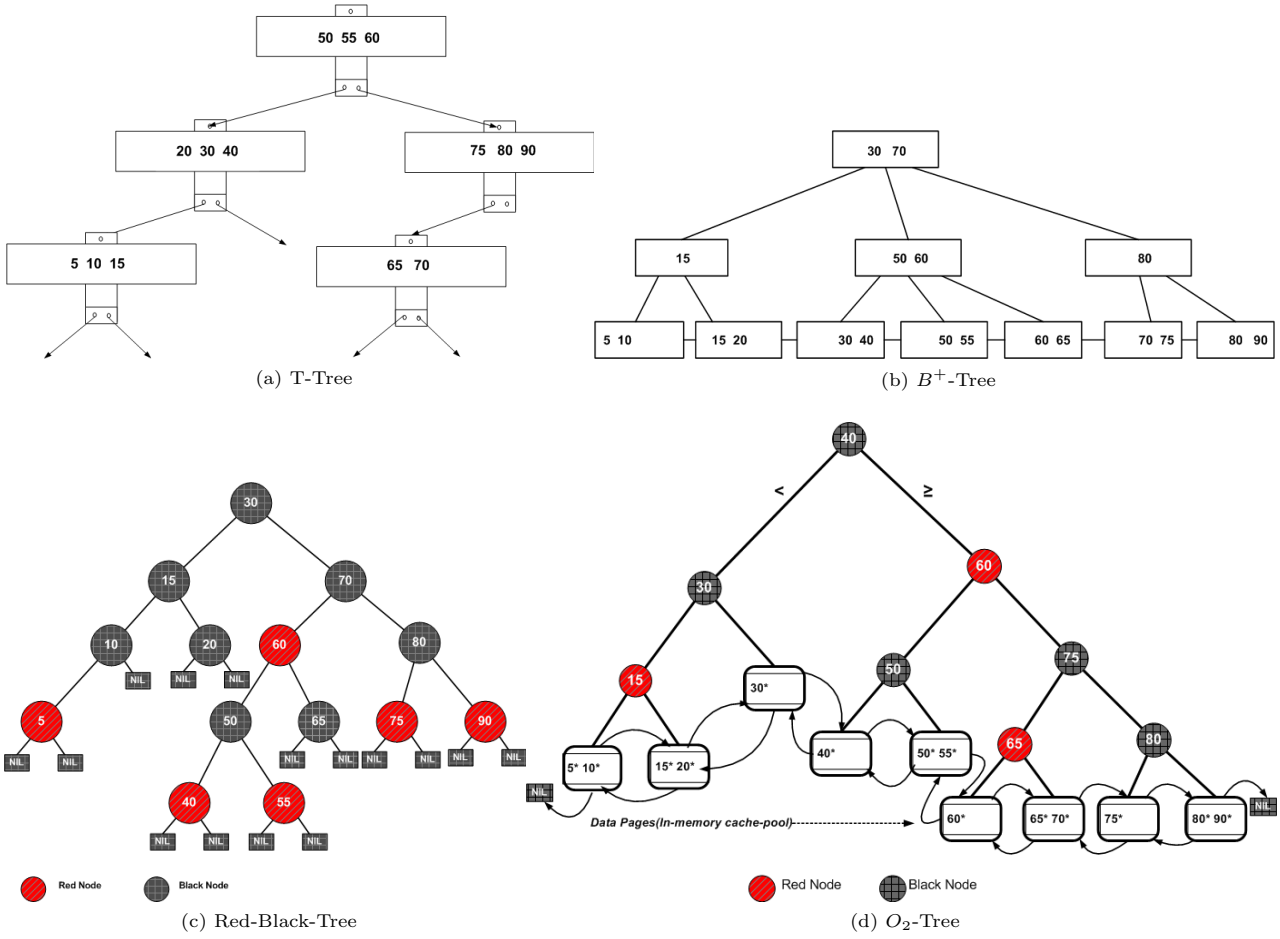
i) Each internal node holds a single key value which is a copy of the minimum key value at the leaf node. These keys are equivalent to the middle keys after a leaf node splits.

ii) Leaf-nodes are blocks that have between $\lceil m/2 \rceil$ and m "key-value" pairs.

iii) 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.

iv) 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 where the



(a) T-Tree (b) B^+-Tree

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

Figure 1: Diagram of the various tree structures

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 (Marcus 2012). It should be noted that the BerkeleyDB access method is never used in our structure. Only the functions of the MpoolFile subsystem are called.

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.

3.2 Some Analytical Results

We state some analytical properties of the O_2-Tree without formal proofs. Our focus is on the concurrent operations.

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.*

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.*

Sketch of 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 $h \leq \log_2 n_b$. This implies that a search (given by *Get()*), an insertion (given by *Put()*) and a deletion (given by *Delete()*) is each computed in time $O(log_2 N/\lceil m/2 \rceil)$. □

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 $O(log_2 N/\lceil m/2 \rceil + s)$.

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 (Larsen 1998), 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-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. We use three locks as in (Nurmi & Soisalon-Soininen 1991, 1996) 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 cause 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 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.

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 *Internal 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 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., $nodeType \neq leaf$). 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., $nodeType = leaf$).

3.3.1 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 leaf page with the given x is found. Once the leaf-page z, in which the search key x must resides, is located, we utilise a binary search function $binarySearch(x, z)$ to locate the "key-value" pair $\langle x, val_x \rangle$ in z. The thread-safe search algorithm is given in Algorithm 1.

Algorithm 1: $Get(key\ x)$

```
Data: key x, T
Result: corresponding ⟨x, val_x⟩ pair if found,
        otherwise null
1  begin
2      ** node is the current node pointer for
       traversal**
3      node ⟵ root(T)
4      node.rlock()
5      while node.nodeType ≠ leaf do
6          if x < node.key then
7              node.left.rlock()
8              node.unlock()
9              node ← node.left
10         else
11             node.right.rlock()
12             node.unlock()
13             node ← node.right
14     done ← binarySearch(x, node)
15     node.unlock()
16     return done
```

3.3.2 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* of other threads to traverse the tree but not with 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 $\langle x, val_x \rangle$, the leaf page (denoted as *node*) in which the key-value pair belongs is first located. When the page is located, it is locked exclusively (*xlock*) and if there is room, the new key-value pair $\langle x, val_x \rangle$ is inserted in order based on the value of the key x into the page, by the function $insertInOrder(x, val_x, node)$. 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 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 tree may grow in height only when a page (leaf-node) overflows. The thread-safe *Put* algorithm is presented in Algorithm 2.

3.3.3 Delete Algorithm: $Delete(x, T)$

The delete algorithm follows a similar pattern as the insert algorithm. However, the delete may result in a page underflow. In 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 in 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 given in Algorithm 3.

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

Algorithm 2: Put($key\ x, value\ val_x,\ T$)

```
Data: key x, value val_x, T
Result: true for success false otherwise
1  begin
2      **node is the current node pointer for traversal**
3      parent ⟵ header
4      node ⟵ root(T)
5      parent.wlock()
6      node.wlock()
7      while node.nodeType ≠ leaf do
8          if x < node.key then
9              node.left.wlock()
10             parent.unlock()
11             parent ← node
12             node ← node.left
13         else
14             node.right.wlock()
15             parent.unlock()
16             parent ← node
17             node ← node.right
18     /* Upgrade both node and parent locks to xlock */
19     parent.xlock()
20     node.xlock()
21     if !(leaf.isfull) then
22         done ← insertInOrder(x, val_x, node)
23     else
24         done ← splitInsert(x, val_x, node)
25         Update problem queue if invariant is violated
26     parent.unlock()
27     node.unlock()
28     return done
```

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 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.

Algorithm 3: Delete($key\ x,\ T$)

```
Data: key x, T
Result: true for success false otherwise
1  begin
2      /* minKeys ensures that node is at least half full */
3      minKeys ⟵ m/2
4      parent ⟵ header
5      node ⟵ root(T)
6      parent.wlock()
7      node.wlock()
8      while node.nodeType ≠ leaf do
9          if x < node.key then
10             node.left.wlock()
11             parent.unlock()
12             parent ← node
13             node ← node.left
14         else
15             node.right.wlock()
16             parent.unlock()
17             parent ← node
18             node ← node.right
19     /* Upgrade both node and parent locks to xlock */
20     parent.xlock()
21     node.xlock()
22     done ← removeKey(x, node)
23     if done and node.underflow() then
24         sibling.xlock()
25         if node.sibling.keys > minKeys then
26             /* Borrow from sibling to keep occupancy */
27             done ← node.appendKeyFrom(sibling)
28         else
29             /* Merge leaf and sibling into the left node; release page block and delete parent node */
30             done ← mergeLeaf(leftNode, rightNode)
31         sibling.unlock()
32     parent.unlock()
33     node.unlock()
34     return done
```

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.

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 organising 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 B^+, 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. 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 (FlightStats: 2005) as well as the records of the *Order* table generated from the TPC-H dbgen data generator (TPC-H 2001).

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, O_2-Tree, AVL-Tree, and the Top-Down Red-Black-Tree. Figure 2 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 operation with a single thread. Figure 3 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 indicate that the O_2-Tree clearly outperforms all the basic structures considered.

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 using TreeDB, the B-Tree access method, and the LevelDB, a 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. We evaluated the average time it takes to perform 20 million (20M) concurrent insertions of "key-value" pairs with the number of threads varied 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. We ensured that the operations were performed with the index tree in memory while the leaf-nodes were accessed via the in-memory cache pool. The data pages were periodically flushed to disk based on the *Least Recently Used* (LRU) cache replacement algorithm. The results are shown in Figure 4. The general observation was that the average time to perform insertions decreased with increasing number of threads. This was due to the fact that even though the degree of access contentions appeared to increase, each thread did less amount of work and consequently encountered less blocking and continued to carry out the correct query operations. Therefore, more threads result in overall better 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 4.

Figure 5 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 the 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.

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

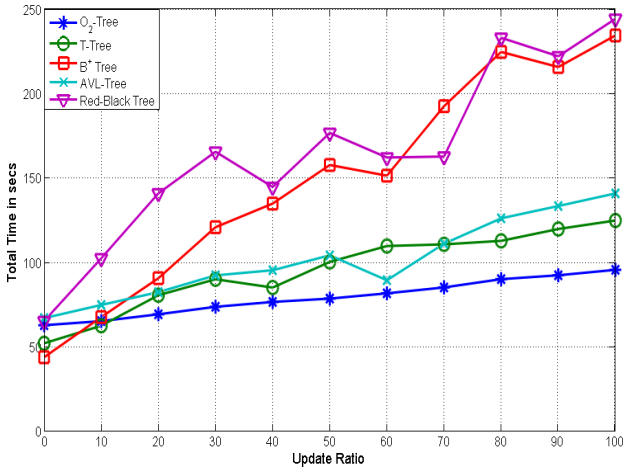


Figure 2: Mixed Operations of Searches, Inserts and Deletes for Basic Indexes using TPC Dataset

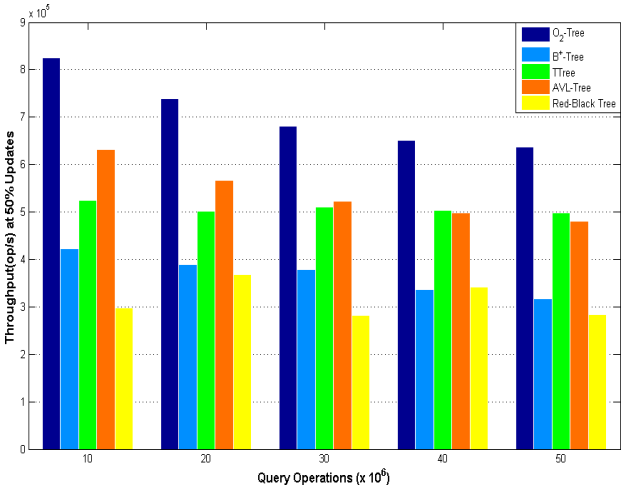


Figure 3: Operational Throughput with 50% Updates for Basic Indexes using TPC Dataset

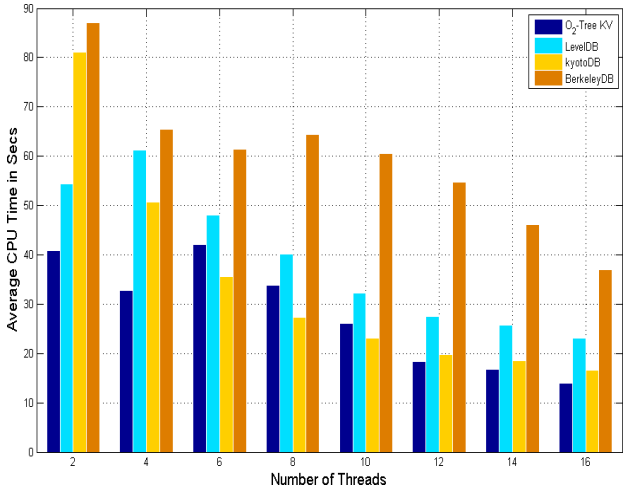


Figure 4: Index Construction with Varying Number of Threads

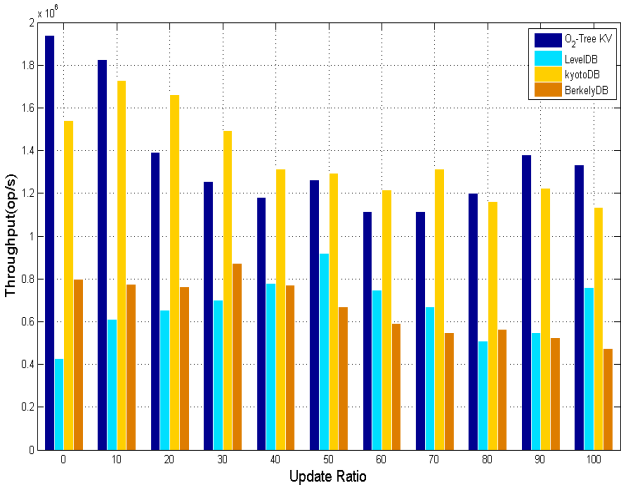


Figure 5: Operational Throughput for Different Mix of Workloads

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 *STM*.

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 tests, shown in Figure 7, illustrated the results with only insertions (*Puts*). Figure 8 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

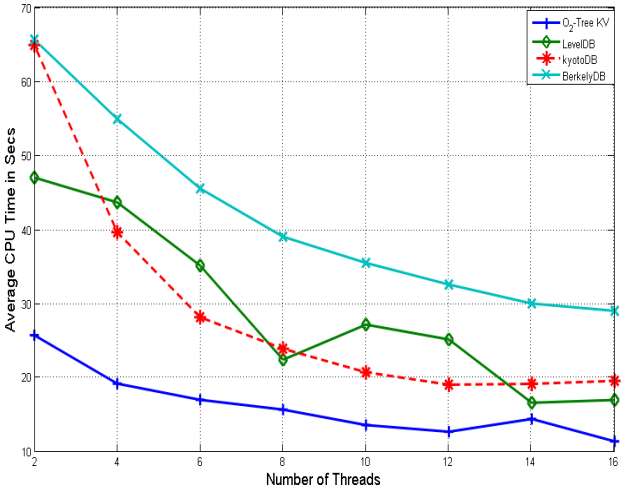


Figure 6: Concurrent Look-ups for 20M "key-value" using Varying Workloads

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

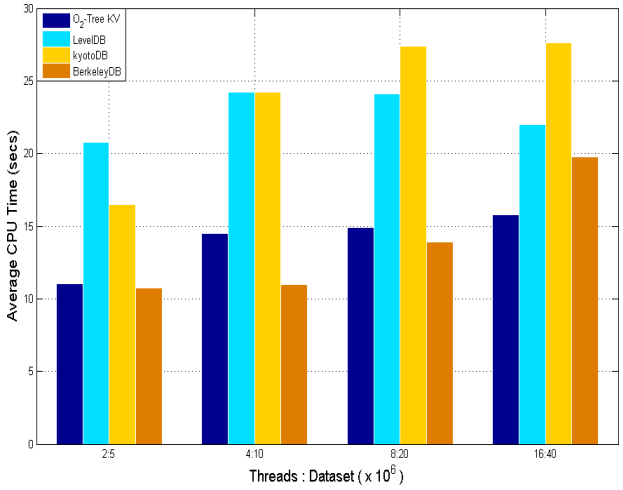


Figure 7: Scalability Test with 100% Insertions

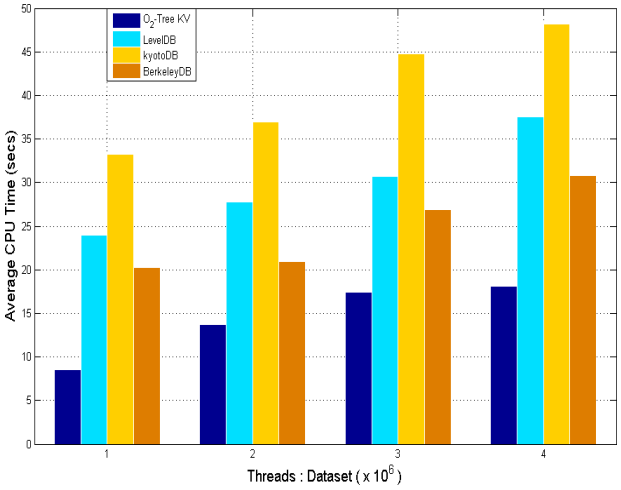


Figure 8: Scalability Test with 50% Update Ratio

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 9 shows the graph for the total problem queue sizes for the tree index. However, a series of experiments conducted indicated that the average problem queue size was about *8* 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.

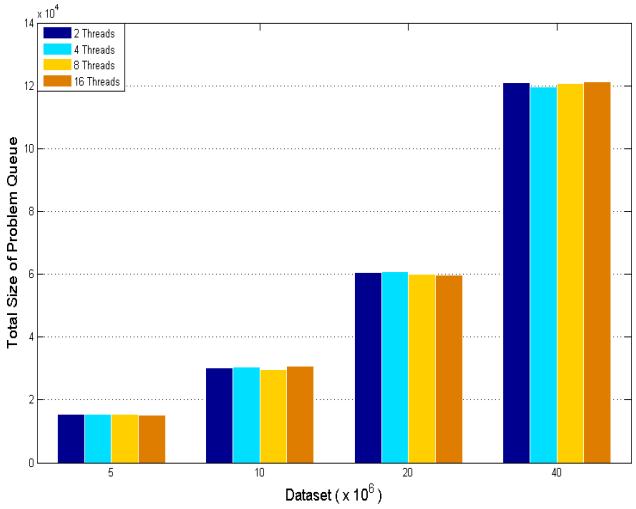


Figure 9: Total *Problem Queue* size for Varying Data sizes and Threads

Finally, we evaluated the performance of each key-value store using real life flight statistics data (FlightStats: 2005) 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 16 threads. The operational throughput 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 10 illustrates the results.

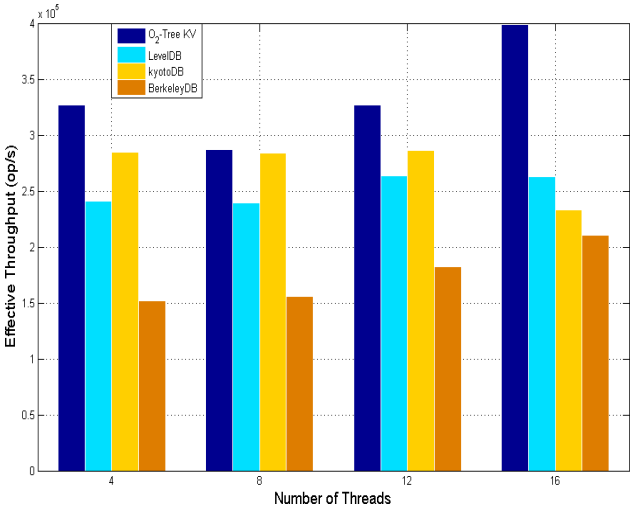


Figure 10: Concurrent Loading Of Real-life Persistent data

6 Summary 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 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 using TreeDB. Our experiments show that O_2-Tree key-value store outperforms both BerkeleyDB, and Kyoto-Cabinet TreeDB by some order of magnitude. 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. Future work anticipated involve 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 the O_2-Tree with GPU for even higher throughput.

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.

Bibliography

10gen, I. (2011), 'Mongodb: High-performance, open source nosql database', `http://www.mongodb.org//`.

Bayer, R. & McCreight, E. M. (1972), 'Organization and maintenance of large ordered indices', *Acta Inf.* **1**, 173–189.

Bohannon, P. et al. (1997), 'The architecture of the dala 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, UK, pp. 270–281.

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

Bronson, N. G., Casper, J., Chafi, H. & 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, ACM, New York, NY, USA, pp. 257–268.

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

Cormen, T., Leiserson, C., Rivest, L. & Stein, C. (2009), *Introduction to Algorithm*, Vol. 1, 3rd edn, MIT Press, Cambridge, Massachusetts London, England.

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

FlightStats: (2005), 'FlightStats Database, http://dl.flightstats.us/', `http://dl.flightstats.us/`.

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

Hanke, S. (1998), The performance of concurrent red-black tree algorithms, Technical report.

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

Kong-Rim, C. & 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, pp. 81 – 84.

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

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

Lehman, T. J. & 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, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp. 294–303.

Lourenço, J., Dias, R. & Luís, J. (2009), Understanding the behavior of transactional memory applications, *in* 'PADTAD '09: Proc. 2009 ACM workshop on Parallel and distributed systems: testing and debugging'.

Lu, H., Ng, Y. Y. & Tian, Z. (2000), T-Tree or B-Tree: main memory database index structure revisited, *in* 'Proc. of the Australasian Database Conference', ADC '00, IEEE Computer Society, Washington, DC, USA, pp. 65– 73.

Marcus, A. (2012), 'The architecture of open source applications: Chapter 13. the nosql ecosystem', `http://www.aosabook.org/en/nosql.html`.

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* 'Proc. of the sixth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems', PODS '87, ACM, New York, NY, USA, pp. 170–176.

Oracle (2012), '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-085366.html`.

Rao, J. & 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, 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* 'Proc. of the 2000 ACM SIGMOD international conference on Management of data', SIGMOD '00, ACM, New York, NY, USA, pp. 475–486.

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

TPC-H (2001), 'Transaction Processing Council', `http://www.tpc.org/tpch/`.