2012 International Conference on Information and Knowledge Management (ICIKM 2012)
IPCSIT vol.45 (2012) © (2012) IACSIT Press, Singapore

O_2-Tree: A Fast Memory Resident Index for In-Memory Databases

DanielOhene-Kwofie, E. J.Otoo[+] and Gideon Nimako

School of Computer Science,
The University of the Witwatersrand,
Johannesburg, South Africa

Abstract. Modern computer architecture with 64-bit addressing has now become common place. The consequence is that moderately large databases can be maintained, during a usage session, as main memory resident database systems. Such in-memory resident databases still require the use of a memory resident index for fast access to the data items. The T-Tree and its variants were developed as the appropriate in-memory index structure but recently a number of such index-driven databases have emerged under the banner of *NoSQL* databases.We propose the *O_2*-Tree as an alternative to the T-Tree that is usable as an index for in-memory databasethat out-performs a number of *NoSQL* databases. The *O_2*-Tree is essentially a Red-Black Binary-searchtree in which the leaf nodes are index blocks that store multiple records of key-value pairs. The internal nodes contain copies of the keys that split blocks of the leaf nodes in a manner similarto the B^+-Tree. Thetree index can be easily reconstructed by reading only the lowest key-value of each leaf node block.

Keywords: Index, In-Memory Database, NoSQL, Memory Resident Index, Key-value store.

1. Introduction

Indexing is the process of associating a key with the location of a corresponding data record in a Database management system (DBMS). Indexes are implemented to facilitate fast query processing in DBMS. In the past, it was common for database systems to store index schemes for the database on disk since it was expensive to have the entire index structure permanently in memory. The required index blocks are then transferred to main memory on demand for manipulation. B^+-tree is a widely used index for disk-based database management systems since it guarantees few disk accesses to read and write disk blocks during query processing. Recent advances in memory architecture and 64-bit addressing allow for memory sizes of the order of hundreds of gigabytes and beyond at a reasonable cost. It has, therefore, become feasible to have sufficiently large shared memory such that the entire index of either, a memory resident or disk-resident database, can be maintained in main memory.

In-memory database systems provide extremely fast response times and very high throughput. The Oracle TimesTen in-memory database, for instance, delivers real-time performance by managing the entire data in memory without any disk-accesses. Such database systems still require the use of a memory resident index for fast access to the data items and in general guarantee very high processing (insertion, deletion, and search operations for exact-match queries, range queries, and largest/smallest key values searches). Lehman and Carey (Lehman and Carey, 1986) proposed the T-Tree which is well known index structure for Main Memory Database Systems (MMDBS).

1.1. Problem Motivation

[+] Corresponding author Tel.: +27736446201;
E-mail address:papaotu@gmail.com

As memory sizes increase with the advances in technology, main memory database systems have become increasing popular. There has been a recent flood of main memory index schemes characterised as *NoSQL* databases also referred to as *key/value* pair index structures (Marcus, 2012). Notably in this pack are index schemes such as BerkeleyDB (Oracle.com), LevelDB (Google.com, 2011) and Kyoto cabinet (FAL Labs, 2011).

The T-Tree, which is an earlier designed and still used in-memory index structure, has several drawbacks. For example the tree index requires that the entire database be traversed to rebuild the index after failure. The T-Tree also has a high system overhead cost in query operations due to the multiple data comparison at each node. For example, searching for a key which is *l-levels* down the root will involve *2l* (minimum and maximum key) comparisons at each node before the bounding node is located for further search. Further, the T-Tree requires several up and down traversals to restore the tree's invariance especially during insertion/deletion of keys. Several variants (Kong-Rim and Kyung-Chang, 1996; Lu et al., 2000; Lindstrom, 2007) have been proposed to limit the number of up and down traversals. Even though the paper on a T-Tree appeared in 1986, as recent as in 2007 (Lindstrom, 2007), research on the T-Tree was still being explored for its feasibility in embedded systems.

To resolve some of the drawbacks in the use of the T-Tree, we propose the O_2-Tree as an alternative to the T-Tree for main-memory resident index structure. It can also be perceived as another solution to the *NoSQL* evolution. The O_2-Tree can easily be constructed with minimal traversals of the data pages or data chunks in main memory database systems. Our solution is a variant of the existing Red-Black-Tree. The O_2-Tree structure provides better performance in terms of query operations for very-large datasets. It also provides an efficient way to ensure that the in-memory data is persistent if desired. It is fault-tolerant since the entire index structure can be easily reconstructed without traversing every record in the database. We also note that although some recent results in *NoSQL* and memory resident indexes indicate extremely fast inserts and searches operations, the constraints on the query classes that can be supported under such fast processing rates are not reported. For example some index schemes have high processing rates under random insertions and look-ups;but they might perform poorly under range searches. We focus primarily on structures that have good performance for both random and range searches.

1.2. Contributions

The major contributions being reported in this paper is the development of a main memory index structure for database systems. The results being reported include:

- Development and implementation of the O_2-Tree as an in-memory index data structure.
- A comparative performance studies by experimentally derived results of the O_2-Tree with other existing structures such as the T-Tree, B^+-Tree, Red-Black Binary-Search tree (or Red-Black-Tree for short), and AVL-Trees.
- Performance studies of the persistent version of the O_2-Tree that uses an in-memory write-through cache to store values in a file compared with *NoSQL* data stores such as LevelDB (Google.com, 2011), BerkeleyDB (Seltzer and Bostic, 2012) and Kyoto Cabinet (FAL Labs, 2011).

1.3. Organization Of the Paper

The remainder of this paper is organised as follows. Section 2 gives a brief survey of different data structures relevant to the implementation of O_2-Tree, which can be directly compared for main memory index. Examples include the T-Tree, AVL-Tree, the B^+-Tree and their variants. In Section 3, we give some details of the O_2-Tree index structure. Section 4 gives an overview of our experimental setup. We give the results of some preliminary experiments conducted with random data datasets. We summarise the work of this paper in Section 6, and give some directions for future work.

2. Background and Related Work

Many index structures exist for main memory databases. A few fromthe literature relevant to our work are discussed.

2.1. The T-Tree

Lehman and Carey (1986) proposed the T-Tree as a new data structure for main memory database management. The T-Tree was developed by combining features of the AVL-Tree and the B-Tree. In the T-Tree, each node stores more than one pair of "key-value, record-pointer" records. The T-Tree maintains the intrinsic binary search tree property of an AVL-tree but has better storage utilisation since each node contains several data items in sorted order. A T-Tree consists of three different types of nodes; an internal node is a T-node which has a left and right child, a leaf node is a T-node with NIL child pointers (no children). A T-node with only one child is called a half-leaf node. Each T-node has only two children pointers to lower level nodes just as in the AVL-trees. Query operations on a T-tree are similar to that of an AVL-tree, except that the T-tree has been shown to have better performance. The general structure of the T-tree is illustrated in Figure 1. Variants of the T-Tree include T^*-Tree (Kong-Rim and Kyung-Chang, 1996) and the T^{LINK}-Tree (Lindstrom, 2007).

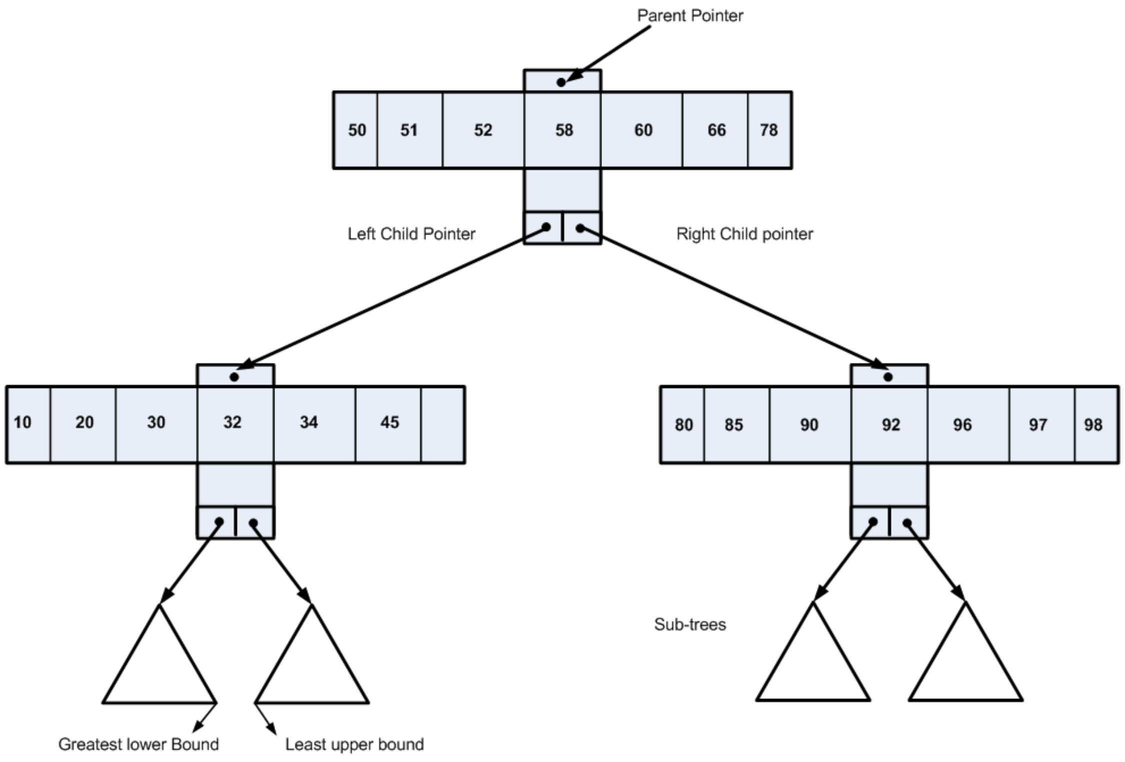


Fig. 1: Structure of the T-Tree indicating the pointers in each T-Node

2.2. The B^+-Tree

A B^+-Tree is one of the much discussed and well understood index structure for database systems. It is a multiway search tree in which each node holds more than one data item. B^+-Tree is one of the most common dynamic index structures in database systems, especially for disk-based DBMS. In such DBMSs the number of disk accesses, to retrieve a record, is proportional to the height of the tree. B^+-Tree therefore has a significantly low height for high fanout known as the tree order. B^+-Tree is an improvement of the B-Tree introduced in (Bayer and McCreight, 1972). A B^+-Tree requires that all keys reside in the leaves. A survey of variants of the B-Tree is given in (Comer, 1979).

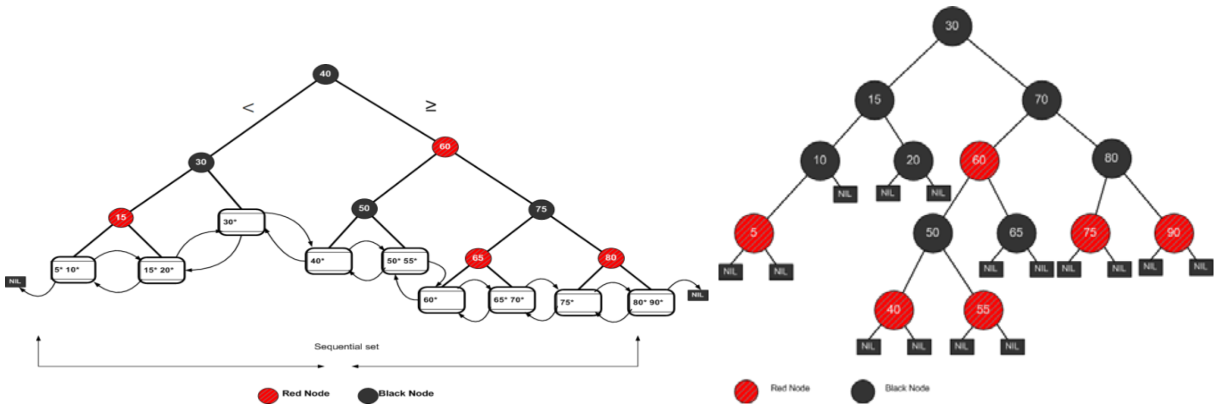
While the T-Tree derives its basic concepts from the AVL-Tree, the O_2-Tree that we describe briefly in the next section, due to limitation in the number of pages allowed, is derived by combining the concepts of the Red-Black Tree and the B^+-Tree.

3. The O_2-Tree

3.1. Overview And Features

The O_2-Tree is basically a Red-Black Binary-Search tree, in which the leaf nodes are formed into index blocks, or chunks that store the records of "key-value, record-pointer" pairs. Let this be denoted as "[keyval, recptr]". The internal nodes contain copies of only the key-values of the middle "key-value, record-pointer" pairs that split blocks of the leaf nodes when they become full. These internal nodes are formed into a simple binary search tree that is balanced using the Red-Black-Tree rotation algorithms. Let K_s be the search key-value 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 Red-Black Binary-Tree balancing algorithm is less complex than that of the AVL-Tree which has a more strict balancing condition. Hence, the choice of the Red-Black-Tree balancing algorithm for the O_2-Tree. The new structure has a number of advantages over the T-Tree and some of the recent *NoSQL* key-value stores. First the O_2-Tree can easily be reconstructed by reading only the lowest "key-value" of each the leaf pages. By maintaining only the leaf blocks persistent, the index tree is inherently persistent. The height of the internal Red-Black-Tree is also significantly reduced compared to the situation where each node stores a single "key-value, record-pointer" pair and the entire tree maintained as a simple Red-Black-Tree.

Associated with the O_2-Tree is the order of the tree denoted by *m*. The order is the maximum number of "key-value, record-pointer" 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 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 *2a* illustrates the schematic layout of the O_2-Tree of order $m = 3$ showing only the key-values in the leaf nodes. The corresponding equivalent Red-Black-Tree is shown side-by-side in Figure *2b*. Detailed explanation of the Red-Black-Tree can be found in (Cormen et al., 2009).



(a) Structure of O2-Tree

(b) A Regular Red-Black-Tree

Fig. 2: Contrast between the O_2-Tree and the Red-Black-Tree

3.2. Definition

An O_2-Tree of order *m* $(m > 2)$, the minimum degree of the tree, satisfies the following properties:

1. Every node is either red or black. The root is black.
2. Every leaf node is coloured black and consists of a block or page that holds "key-value, record-pointer" pairs.
3. If a node is red, then both its children are black.
4. For each internal node, all simple paths from the node to descendant leaf-nodes contain the same number of black nodes. Each internal node holds a single key value.
5. Leaf-nodes are blocks that have between *m/2* and *m* "key-value, record-pointer" pairs.
6. If a tree has a single node, then it must be a leaf which is the root of the tree, and it can have between *1* to *m* key data items.
7. Leaf nodes are double linked in forward and backward directions.

The properties of the O_2-Tree are summarised as follows:

- It has a relatively low height which provides for faster tree traversal.
- Tree rotations for balancing are minimised and therefore guarantee fast query processing. The greater the order of the tree, the larger the room in the leaf-nodes. This is able to contain more "key-value, record-pointer" pairs thereby minimising splitting which could consequently lead to rotations.
- The doubly-linked leaf-nodes provide an easy mechanism to traverse the tree in sorted order for key range searches.
- The O_2-tree also has a superior performance compared to the T-tree since the O_2-tree makes only one key comparison per node internally in order to determine which branch to follow.
- The tree can easily be rebuilt from the leaf nodes since the internal nodes are simply copies of the minimum key values at the leaf nodes. These are equivalent to the middle keys after a split occurs.

Two fundamental properties of the Red-Black-Tree allow for the construction of the O_2-tree. We state them below without formal proof due to the limitation on the number of pages.

Proposition 3.1. *In a Red-Black-Tree all black leaf-nodes (that are NIL) are guaranteed to remain as leafnodes under all rotations.*

Proposition 3.2.*A Red-Black-Tree with N leaf-nodes will still maintain its N leaf-nodes after single ordouble rotations.*

It follows from the above that data pages stored in the leaf nodes will always remain at the leaf nodes irrespective of the rotations performed on the tree. As a result, the O_2-Tree simply utilise the otherwise NIL pointers of the terminal nodes of the Red-Black-Tree to retain the leaf-blocks.

The O_2-Tree, supports the query operations of inserts, deletes, and searches in time $O(log_2 N/\lceil m/2\rceil)$, where N is the number of "key-value, record-pointer" pairs in the structure. This follows from the fact that the number of index blocks of the leaf nodes is at most$n_b = N/\lceil m/2\rceil$, and the number of nodes of supporting internal Red-Black-Tree is $n_b - 1$. The height $h \leq \log_2 n_b$which leads to the conclusion that the search, inserts and deletion is $O(log_2 N/\lceil m/2\rceil)$.New internal nodes are only added when leaf-nodes split as a result of overflows in the leaf-blocks. The tree may grow in height after a split of a leaf-block. The reverse occurs when there is an underflow resulting in the merging of leaf-nodes and the subsequent removal of the parent of the nodes that are merged. The tree operations are outlined and discussed below.

3.3. Searching In An O_2-Tree

Searching the O_2-Tree is similar to searching in a Red-Black binary Tree. However, the internal Red-Black tree structure serves as place holders to locate the actual key in the leaf-node. The search algorithm is given Algorithm 1. 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,perform on the average,*m/2* comparisons before continuing with the search. We give the algorithm for the top-down insertion of the O_2-Tree in the subsequent subsection, but leave out details of the top-down deletion.

3.4. Top-down Insertion Algorithms

The objective of the Top-down algorithm is to ensure that when a new internal node is inserted, there will be no need to move up the tree again. It thus, guarantees that when a new internal node is inserted, its parent will not be Red. The algorithm is given in Algorithm 2. The supporting splitinsert() and the prebalancing algorithms are stated as Algorithm 3 and Algorithm 4 respectively. For the purposes of the algorithm, we assume that a node, *x*of an O_2-Tree has a key denoted as *x.key.* The left and right child pointers of *x,*are denoted by *x.left* and *x.right* respectively. The parent node is also denoted by *x.parent*. We also assume that some functions invoked in the algorithms perform actions implied by their names.

Algorithm 1 The Search Algorithm

```
Search(key $x$ ,Node $node$)
$node \leftarrow root$
while $node \neq leaf$ do
  if $x < node.key$ then
    $node \leftarrow node.left$
  else
    $node \leftarrow node.right$
  end if
end while
Do binary or sequential search
for key in the current $node$
return $binarySearch(x, node)$
$done$
```

Algorithm 2 Key-value Insertion

```
$insert(key\,x, Value\,y, Node\,node)$
$node \leftarrow root$
while $node \neq leaf$ do
  if $x < node.key$ then
    $node \leftarrow node.left$
  else
    $node \leftarrow node.right$
  end if
  if $node.left.color == RED$ and $node.right.color == RED$ then
    $preBalance(x, node)$
  end if
end while
if $!full(leaf)$ then
  $insertInOrder(x, y, node)$
else
  $splitInsert(x, y, node)$
end if
$done$
```

Algorithm 3 The SplitInsert Algorithm

```
$splitInsert(Key\,x, Data\,y, Node\,leaf)$
$newLeaf \leftarrow new\,leaf()$
$newNode \leftarrow new\,internalNode()$
$midpoint \leftarrow m/2$
$where\ m$ is the order of the tree
$j \leftarrow 0$
for $i = midpoint$ to $m - 1$ do
  $newLeaf[j] \leftarrow leaf.remove[i]$
  $j{+}{+}$
end for
insert "$key, value$" into the appropriate leaf
$newNode.key \leftarrow newLeaf[0].key$
$newLeaf.parent \leftarrow newNode$
$leaf.parent \leftarrow newNode$
Set forward and backward links of $leaf\ nodes$
$done$
```

Algorithm 4 The PreBalancing Algorithm

```
$preBalance(Key\,x, node\,N)$
$N.color \leftarrow RED$
$N.left.color \leftarrow BLACK$
$N.right.color \leftarrow BLACK$
$P \leftarrow N.parent$
$G \leftarrow N.parent.parent$
if $P.color == RED$ then
  $G.color \leftarrow RED$
  if $N\,and\,P\,are\,same\,side\,children$ then
    $singleRotate(P)\ around\,G$
  else
    $doubleRotate(N)\ around\,G$
  end if
  $N.color \leftarrow BLACK$
end if
$done$
```

3.5. Storage Utilisation

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

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

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 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. Experimental Environment

We performed experiments to compare the Top-down Red-Black-Tree, AVL-Tree, B^+-Tree, the T-Tree and our proposed O_2-Tree as simple memory resident indexes. We then performed a second set of experiment with the persistent O_2-Tree compared with some *NoSQL* (key-value) stores such as the BerkeleyDB (Seltzer and Bostic, 2012), Kyoto Cabinet (FAL Labs, 2011) and LevelDB (Google.com, 2011) where the data blocks are written and read through an in-memory cache to a disk file. These experiments were conducted primarily to compare the performance of O_2-Tree with these key-value stores using various access methods.

Our implementation of the O_2-Tree is in C++ and the program was compiled and built using the GNU g++ compiler on a 64-bit Intel i7 multi-core processor machine running the Ubuntu 11.10 operating system. We also took the liberty to implement some of the basic existing index structures for main memory index and also modified some source codes for our testing platform. Each index holds pointers to records (i.e. the memory address). The Netbeans Integrated Development Environment (IDE) equipped with the GNU Project Debugger (gdb) as well as the Sun Studio were used in developing, implementing, debugging and profiling of these index structures. Our test data, the data values, that also serve as the keys, are generated as a set of 4-byte random integers. Results of the performance tests are described in the next sections.

5. Performance Evaluation and Results

5.1. Performance Features Evaluated

The performance of each index structure is evaluated experimentally in a simulated database environment. We set the keys and pointers in each experiment to 4 bytes. All keys are randomly generated in advance within the range of 1 to 30 million. We repeat each experiment 3 times and the average time is reported.

The first set of experiment conducted involved a series of insertions. We set this to measure how fast each data structure can construct the tree index given a random set of keys. The other query operations were also subjected to the 3 different sets of similarly generated random data. The experiments were done such that each index structure obtains the same set of data in the same random order. In the second set of performance evaluations, each data structure was subjected to a mix of insertions, deletions and searches with different percentage of each operation. The total time to perform the entire mix of operations is then reported.

We also compared the implementations of the index structure where the key values and their associated data are kept persistent through an in-memory cache. This is to compare our index structure to other popular *NoSQL (key-value)* databases that have been reported in the literature as being extremely fast. These include BerkeleyDB, Kyoto Cabinet as well as LevelDB. In this setup, we implemented a persistent index structure through an in-memory cache. Operations are performed in the cache and pages are periodically flushed to disk based on the Least Recently Used (LRU) cache algorithm. We setup the experiment such that each key-value store has enough cache to keep the data being processed. The data is flushed to disk and the overall time to complete the operation is reported. We repeat this for varying data sizes and database workloads. The O_2-Tree used the BerkeleyDB Mpool caching subsystem.

5.2. Discussion of Results

Figure 3 shows the performance of the five data structures for a simple build of the index. We perform30 Million unique key insertions. The order of the T-Tree, B^+-Tree, and the O_2-Tree used was $m = 256$.The plot shows the times for building the respective data structures.

As can be observed from the graphs, AVL-Tree performed worst among the index structures consideredwhile the O_2-Tree performed best. The Top-down Red-Black Tree also performed better than the AVL-Tree.AVL's strict balancing requirement accounts for its worst performance. The O_2-Tree on the other hand,required fewer splits and rotations which accounts for its superior performance. The B^+-Tree performedbetter than the T-Tree due to its significant low depth and less complexity in restoring the tree's invariants.The O_2-Tree, however, outperformed the B^+-Tree due to the fact that the B^+-Tree makes multiway-decisionduring its traversal down the tree whiles the O_2-Tree makes single data comparison to determine the searchpath during traversal. Also, splitting and redistribution of keys in the nodes of a B^+-tree may propagate allthe way to the root of the tree. However, only colour changes, which are less expensive, may propagate upthe O_2-Tree.

Figure 4 shows a similar set of experiments, with the same data structures except that now, the timesare recorded for a mix of insertions and searches. In the workload driven construction of the indexes, 50% ofthe keys generated were used for searches while 50% of the keys generated were for new insertions. Again,the O_2-Tree outperformed all the other index structures tested in the experiments. Figure 5 shows similarresults withdifferent workload ratio in which 75% of the total operations are searches, 20% inserts and 5%deletes.

The second group of experiments illustrates the performance of the in-memory O_2-Tree that writes into a file for persistence using the BerkeleyDB Mpool subsystem as a cache. This scheme is compared with otherpopular *NoSQL key-value* stores. Each key-value store was tuned with a page size of *4k* (4096 bytes) as wellas a *2GB* cache size. We applied tuning mechanisms to the NoSQL databases as indicated and recommendedfrom their respective documentations. We also did not enable compression.

Our results indicated that the O_2-Tree, using BerkeleyDB memory pool, performed comparatively to theLevelDB and Kyoto Cabinet HashDB (using 4GB map, 5 million Buckets). We actually run the LevelDB inasynchronous mode in which a separate thread concurrently flushed the cache contents to disk. Even thoughin comparing the LevelDB and the BerkeleyDB with O_2-Tree, the O_2-Tree has the disadvantage that itdoes not have asynchronous backend persistent store, our objective is to evaluate how the O_2-Tree writing todisk through an in-memory cache, compares with these popular industry-standard *NoSQL* key-value stores.

The O_2-Tree with a cache support however, performed over *5X* faster than the Kyoto Cabinet TreeDBand more than *2X* as fast as BerkeleyDB using the B-Tree access method. The Kyoto Cabinet HashDBhowever performed excellently when tuned with a 4GB map and 5 million buckets. The results are as shownin Figure 6. We also observed that, the O_2-Tree actually performed comparatively and even better thanLevelDB for keys sizes up 20 million.

The second set of experiment with the NoSQL databases illustrated in Figure 7 and Figure 8, showsthe performance of each key-value store under different workloads; 50% inserts and 50% searches, as wellas a workload of 75% searches, 20% inserts and 5% deletes. Again, the O_2-Tree when reading and writingthrough a cache exhibits performance characteristics that are comparable to the LevelDB and the KyotoCabinet HashDB. It did however; outperform the BerkeleyDB and the Kyoto Cabinet TreeDB by severalorders of magnitude.

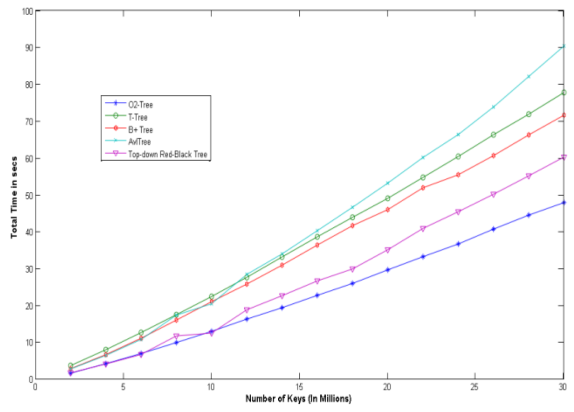


Fig. 3: Index build with randomly generated keys

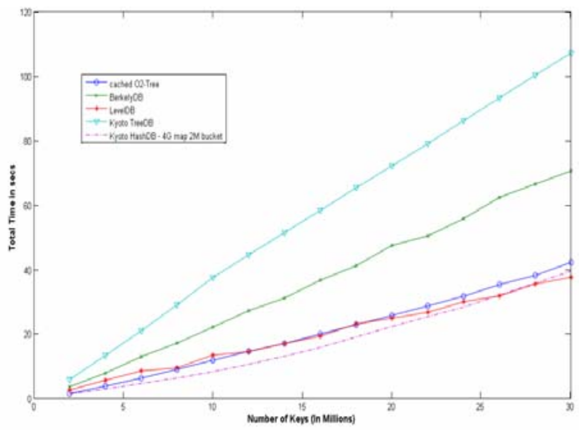


Fig. 6: Persistent store: With in-memory cache insert

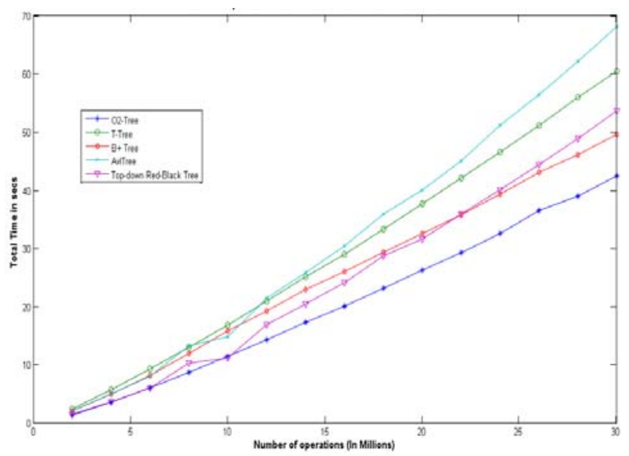


Fig. 4: Mix of 50:50 searches & inserts

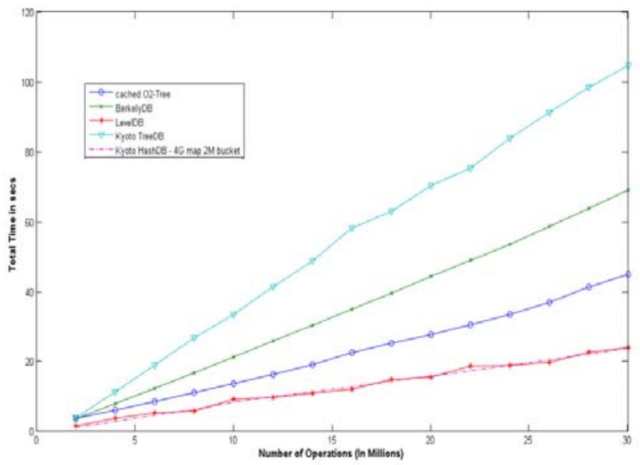


Fig. 7: Persistent Store: Mix of 50:50 searches & inserts

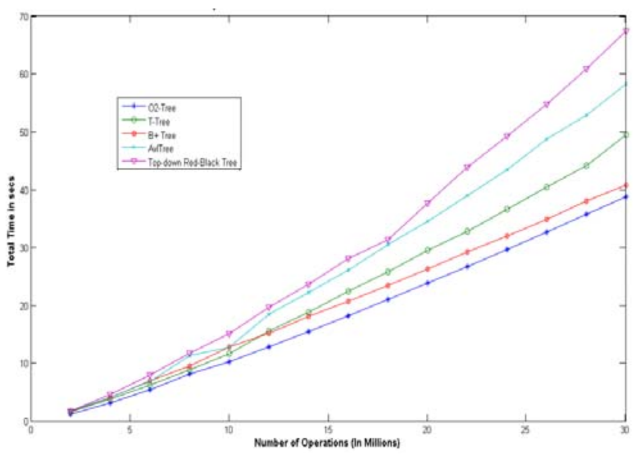


Fig. 5: Mix of 75:20:5 searches, inserts & deletes

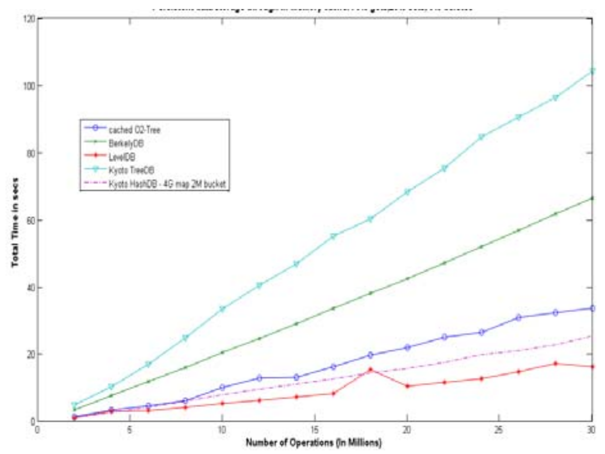


Fig. 8: Persistent Store: Mix of 75:20:5 searches,inserts & deletes

6. Summary and Future Work

In this paper, we have discussed some of the existing index structures for main memory resident databases. We have discussed some of the favourable and unfavourable properties of these earlier index structures. The main theme of this paper is the development and implementation of the O_2-Tree as a new database index structure that serves as a better alternative to the T-Tree and possible some new NoSQL key-value stores. The O_2-Tree guarantees a logarithmic query processing as in other balanced binary trees. However, the new structure has a low depth which guarantees a faster processing time than the existing indexes such as the B^+-Tree, AVL-Tree, T-Tree and the Red-Black-Tree. Our preliminary results indicate that, the height of the O_2-Tree is in general less than that of an equivalent Red-Back Tree with the same number of keys depending on the block-size of leaf nodes. In particular, even though the O_2-Tree has a greater height than that of the B^+-Tree and the T-Tree, the O_2-Tree has a lower branching factor (2 per node) and therefore does fewer comparisons per node than a B^+-Tree and the T-Tree of the same order. This is due to the fact that a B^+-Tree as well as the T-Tree make multi-way decision per node. Our experimental results show that the O_2-Tree is much more superior to earlier main memory index structures, especially when the datasets are large.

Our future work is to implement concurrent protocols on the use of the structure in a shared memory multicore architecture. Further studies are anticipated in a real multiuser environment with concurrent accesses of thread-safe variants of some of the index structures used in our studies. This will enable us conduct comparative performance tests of some of these index structures, with the O_2-Tree, in a shared memory environment. The concurrent protocols anticipated on the O_2-Tree will explore both pessimistic, as well as optimistic concurrency control protocols with relaxed balance of the index structures.

7. Acknowledgements

This work was supported with funding from the Centre For High Performance Computing, CHPC, South Africa.

8. References

[1] R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indices.*Acta Informatica*, 1: 173-189, 1972.

[2] D. Comer. Ubiquitous B-Tree. *ACM Computing Surveys*, June 1979, pp. 121-137.

[3] T. Cormen, C. Leiserson, L. Rivest, and C. Stein. *Introduction to Algorithm*, volume 1. MIT Press, Cambridge, Massachusetts, London, England, 3rd edition, July 2009.

[4] FAL Labs. Kyoto cabinet: *A Straight forward Implementation of DBM*. http://fallabs.com/kyotocabinet/, 2011.

[5] Google.com. *LevelDB: A fast key-value storage library* written at Google. http://code.google.com/p/leveldb/, 2011.

[6] C. Kong-Rim and K. Kyung-Chang. T *-Tree: A main memory database index structure for real time applications. *In Proc.IEEE Real-Time Computing Systems and Applications*, pp. 81 -84, South Korea, Nov. 1996.

[7] T. J. Lehman and M. J. Carey. A study of index structures for main memory database management systems. *In Proc. Ofthe 12th International Conference on Very Large Data Bases*, VLDB '86, pp. 294-303, San Francisco, CA, USA, 1986.

[8] C. H. Leung. Approximate storage Utilization of B-Trees: a simple derivation and generalizations. *Information Processing Letters*, 19:199-201,Nov. 1984.

[9] J. Lindstrom. T^{link}-tree: main memory index structure with concurrency control and recovery. *In Proc. of the third conferenceon IASTED International Conference: Advances in Computer Science and Technology*, ACST'07, pp. 533-538, Anaheim, CA, USA, 2007.

[10] H. Lu, Y. Y. Ng, and Z. Tian. T-Tree or B-Tree: Main memory database index structure revisited. *In Proc. of the AustralasianDatabase Conference*, ADC '00, pp. 65-73, Washington, DC, USA, 2000. IEEE Computer Society.

[11] A. Marcus. The architecture of open source applications: Chapter 13. *The NoSQL ecosystem*. http://www.aosabook.org/en/nosql.html, 2012.

[12] Oracle.com. *Oracle BerkeleyDB 11g*. http://www.oracle.com/technetwork/products/berkeleydb/overview/index.html.

[13] M. Seltzer and K. Bostic. *The architecture of Open source Applications*: Chapter 4. BerkeleyDB. http://www.aosabook.org/en/bdb.htm, 2012.