Fast Indexting: Techniques to Optimize Database Management Systems

Anna Kovalevskaya
University of Toronto
Toronto, Canada

Sam Shareski
University of Toronto
Toronto, Canada

Alexandru Urda
University of Toronto
Toronto, Canada

***Abstract*—The spread use of databases brought up a variety of data structures and methods to support the growing demand. It makes sense to have different databases to support different needs, which is why data management systems have been optimized with respect to their most used operations. Some systems became best for specific operations such as read, write, update and so on. For example, Cache Sensitive B+ (CSB+) Trees aimed to provide better index performance for searches for in memory databases while maintaining good performance for insert and deletion which B+ Trees already provide. Log-Structured-Merge (LSM) trees were inspired in 1990s to optimize write by writing to disk in batches of records. This feature of LSM resulted in slower read, no-in place update and challenges for index maintenance. To improve deferred operations such as read, delete or some queries, we propose bLSM tree. To improve index maintenance we propose diff-index, different index schemas (*sync-full*, *sync-insert*, *async-simple* and *async-session*) [2] based on a stronger need for consistency or latency. Fractal Trees is an optimization of B-trees. We then compare performance of various database structures [3]. The discussion is closed with respect to new hardware and new data structures that efficiently take advantage of advances in technology.**

Keywords—Database, optimization, Cache, Log-Structured-Merge (LSM) tree, diff-index, Fractal Tree, Bw-tree.

I. INTRODUCTION

A. Cache Sensitive B+ (CSB+) Tree

CSB+ trees are a modification of B+ trees designed in mind for the better cache utilization. This is especially beneficial for in-memory databases when main memory accesses are the largest performance bottleneck. CSB+ tree use partial pointer elimination to increase the space efficiency of the tree nodes while maintaining good performance for incremental updates.

B. Log-Structured-Merge (LSM) Tree

NoSQL databases have become more popular in the recent years for its advantages over relational database management systems. Examples are BigTable [4], Dynamo [5] and many more. The LSM-tree minimizes I/O costs by moving batches of data from main memory to disk. By storing and moving data in contiguous chunks, the algorithm greatly reduces disk arm movement [6] as well. To optimize write, LSM does not support in-place updates. Instead, the algorithm adds a new record with an updated timestamp. LSM is great for heavy write operations but not recommended for heavy reading.

C. Optimization using bLSM

Deferred read, deletes, updates and range queries in LSM trees motivated new features in LSM to improve overall performance. The solutions proposed are extensions to the existing merge and read algorithms. In particular, the bLSM tree achieves a near-optimal read and scan performance by efficiently scheduling merges, taking advantage of threading, terminating read as early as possible and using Bloom filters.

D. Diff-Index

Contrary to bLSM, diff-index offers a different approach to improve LSM read and index maintenance. In particular, Diff-Index proposes different schemes aimed at different performance goals. According to the CAP theorem, any network and replicated data store can enjoy two of the three properties: consistency, availability and network partition-tolerance [2]. Therefore it makes sense to provide different schemes for different data stores depending on the need for consistency and latency.

E. Fractal Tree

Fractal Trees are an alternative to LSM trees. FT is a simple extension of B tree where each internal node of the tree contains a buffer. The buffers are filled from root to lower nodes. Data is moved when a buffer is full.

F. B Tree, Fractal Tree and LSM Cost Comparison

This section is a brief summery of time and space costs between B Tree, Fractal Tree and LSM.

G. Future Discussion

With the emergence of new hardware, it is important to look at most recent structures. In particular, we view a variation of a B-tree that optimizes multi-core systems and flash storage.

II. CACHE SENSITIVE B+ (CSB+) TREE

A. Motivation

Memory costs are continually decreasing which makes in-memory databases a viable option for many applications. With the entire database in memory the cost of disk I/O no longer has to be considered. While main memory access is orders of magnitude faster than disk latency, it still has a considerable cost compared to CPU speed, around 100 cycles of latency compared to accessing the L1 cache [1]. The task of reducing the amount of memory accesses, or in other words, increasing

the amount of cache hits, cannot accomplished in a direct manner since the cache is not controlled by the application layer. Instead, a more subtle approach is required to design a data structure conscious of the cache.

B. B+ Trees in the Cache

B+ trees can cooperate well with the cache as long as we tune the structure with the cache in mind. To have good cache performance, we need to set the nodes of the tree to be the same size as the cache line (block size). This will be between 32 and 128 bytes. If the nodes are the same size as the cache line then there will be no wasted space and the cache will be able to hold more of the tree structure.

But how much information about the B+ tree can we actually store in a node? Let's assume our node size is 32 bytes, and the keys and pointers are 4 bytes each. We also need to store the number of keys used for another 4 bytes. If we use first 4 bytes to store the number of keys, we can store 3 keys for another 12 bytes which leaves the remaining 16 bytes for the 4 pointers we need to store. This means that half of the space used for the node is consumed by pointers. To improve cache utilization we need to have more efficient use of space in our node structure.

C. Pointer Elimination

To be more efficient with the cache line we need to have some strategy to remove pointers from the data structure since we cannot remove the keys. There have been many different structures proposed to address this issue (Some examples are CSS-Trees, PLI-Trees and V-Trees) but they all have the same basic idea of removing pointers from the data structure.

It's possible to not have any pointers in our data structure but this applies certain restrictions. With no pointers, we have to know the structure of the entire tree implicitly so that we can use arithmetic to identify child nodes [1]. If this is accomplished, then we can greatly improve search performance in an in-memory database since we can store many more keys per cache line. However, this approach has drawbacks. Eliminating pointers means we have to pre-allocate space for the entire tree and follow very strict allocation policies. This makes modification to the tree very difficult and very expensive. Either restrictions must be applied to the types of updates that can be made or updates need to be done in batches. This is acceptable for certain applications but is not a great general purpose solution.

D. CSB+ Tree with One Child Pointer

The goal for the CSB+ tree is to provide good cache performance while retaining the good all around performance of a B+ tree [1]. This is achieved by using *partial* pointer elimination [1]. In a CSB+ Tree inner node, we store the following: a single pointer to a child node *group*, the number a keys, and a list of the keys themselves. A node group contains nodes who all share the same parent node. The node group is stored contiguously in memory so when given a pointer to the first node in the node group, we can use arithmetic to access any node in the group.

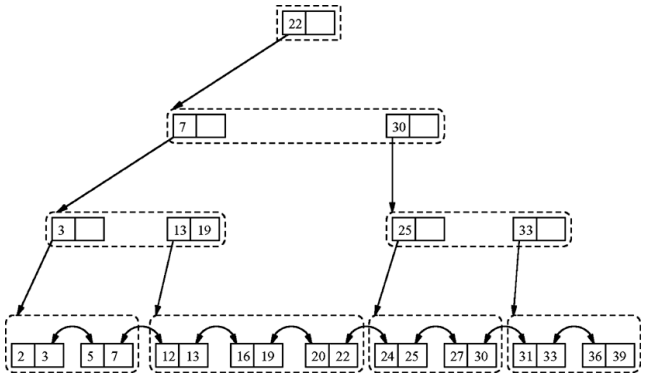

Fig. 1. Schematic Picture of CSB+ cache

To optimize cache utilization we apply the same strategy we used in B+ trees. We set the size of the node to be the size of the cache line. Using the same size cache line as the previous example, 32 bytes, we now need 4 bytes for the one pointer, 4 bytes for the number of keys, and can use the remaining 24 bytes to store a total of 6 keys. This is double the number of keys we were able to store in a regular B+ tree node of 32 bytes.

E. Operations on a CSB+ Tree

The advantage of using a CSB+ Tree as opposed to another cache optimized data structure is that we can apply the efficient algorithms of a B+ tree for searching, inserting and deleting with only slight modifications which we will describe.

Search. Searching in a CSB+ tree is nearly identical to searching in a B+ tree. The only difference is how we locate the child node. When we find the correct key which is the *k*th key in the node, we can locate the child node using a simple offset: child_node = first_child_pointer + *k* * OFFSET.

Insert. As with a B+ tree inserting is tricky when inserting a key causes a node to split. We have two cases, splitting a leaf node and splitting a parent node. The main difference in a CSB+ tree is instead of splitting a node, we split the entire node group.

In the first case, suppose we have to split a leaf node in a node group *g* which is pointed to a parent node *p* which has room for another key. In this case we allocate a new node group *g'* and copy the nodes for *g* into *g'* with a new node added. We then update the pointer in *p* to point to *g'*.

In the case where the parent node *p* does not have room then we still create a new node group *g'* and split the keys evenly between *g* and *g'*. The keys in parent node *p* needs to be redistributed between *p* and *p'* which will point to *g* and *g'* respectively. The node group containing *p* will also need to be copied and its parent node updated as it parent node was in the first case.

Delete. Deletions are handled similar to insertion but it's often simpler to avoid merging and do a lazy delete since deletions are typically much less common than insertions [1].

In all operations CSB+ trees offer similar algorithms to a B+ tree the difference being the amount of information that needs to be copied is higher for a CSB+ tree.

F. Variations and Performance

Depending on what the requirements are for the application we can use two variations of a CSB+ tree depending on whether we want to prioritize space efficiency or speed. If space is a concern we can use a Segmented CSB+ tree which simply divides the node groups into small segments. However, for every division we have add more pointers to the nodes. When space is not a concern, we can use a Full CSB+ Tree. In this alternative, we allocate space for a full node group whenever we allocate a group. This causes splits to a node group to be much less frequent.

Method	Branching Factor	Total Key Comparisons	Cache Misses	Extra Comparisons per Node
Full CSS-Trees	$m+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m+1)}$	0
Level CSS-Trees	m	$\log_2 n$	$\frac{\log_2 n}{\log_2 m}$	0
B^+-Trees	$\frac{m}{2}$	$\log_2 n$	$\frac{\log_2 n}{\log_2 m-1}$	0
CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	0
CSB^+-Trees (t segments)	$m-2t+1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-2t+1)}$	$\log_2 t$
Full CSB^+-Trees	$m-1$	$\log_2 n$	$\frac{\log_2 n}{\log_2 (m-1)}$	0

Fig. 2. Table of Comparison of Different Methods

In this table we can look at the theoretical performance of different variations of CSB+ trees and other options. The parameters are *m* which is the number of slots per node and *n* is the number of leaf nodes in the tree. Another benefit in addition to better cache usage is that the branching factor for a CSB+ tree is double the branching factor of a B+ tree.

G. Conclusion

The CSB+ tree is a clever example of a data structure which can take advantage of a specific hardware configuration. It's also interesting because it builds upon a very common and mature data structure which allows to take advantage of improvements for a regular B+ tree. While the CSB+ tree is a good all around performer, it does not necessarily provide the best performance for any particular operation. If we have a very specialized use case it may be better to consider an even more specialized data structure.

III. LOG-STRUCTURED-MERGE (LSM) TREE

A. Motivation

Log-Structured-Merge (LSM) tree was inspired in the 90s to support heavy write operations. The structure was indented for history and logging purposes but it also spread to industries such as banking where making a transaction is much more likely than reading one.

B. Components of LSM

The LSM structure consists of k components. The first component resides in main memory and the rest reside on disk.

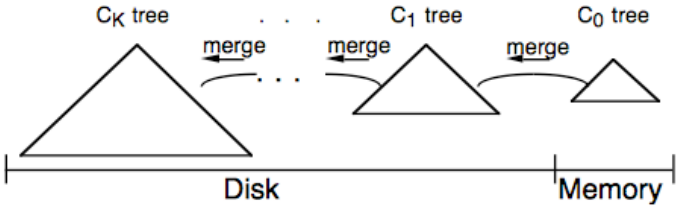

Fig. 3. Schematic Picture of LSM Components

When a new row is inserted, it is put into the first component in memory. Because it resides in memory, this component's depth does not have to be minimized. It can be kept with any structure such as an AVL tree [6]. When any component reaches a pre-set threshold, it is merged into the next component.

C. The Merging Algorithm

The algorithm begins by copying a multi-page block containing leaf nodes of the larger component into a buffer. It then picks up a disk page size leaf node from the buffer and merges with entries taken from [6]. This decreases the size of the smaller component and adds a new merged leaf node in the larger component.

IV. OPTIMIZATION USING BLSM

A. Motivation

Although LSM was intended for heavy write loads, it is important to look at how it could be improved for other operations as well. bLSM is an extension of LSM that addresses read amplification and write pauses.

B. Reducing Read Amplification

Bloom filters is a probabilistic technique to test if an element is in the set. It might return a false positive but it will never return a false negative. To use Bloom filters, allocating 10 bits per item leads to 1% false positive rate and is a reasonable trade-off in practice [7]. Bloom filters reduce element lookup amplification from
N to 1+N/100 [7]. On top of Bloom filters, it is possible to terminate read sooner by reading from smallest component to largest. This is because newest entries are always in the smaller components.

C. Dealing with Write Pauses

Write pauses happen when the merge algorithm locks its components. A few workarounds have been proposed. For example, it is possible to make a new temporary component while the real one is being merged and then merge it later on but this majorly slows scan [6]. Another approach is to merge at the application level when demand for write is low but this requires specialized labour at the application level. Smaller partitions are the last approach but they eventually lead to the same problem of write pauses.

D. Merge Scheduling

Because leaving tree components to accumulate sacrifices scan performance, merging is important. bLSM trees use a merge scheduler to minimize write pauses. In particular, a gear scheduler ensures that all merges finish at the same time.

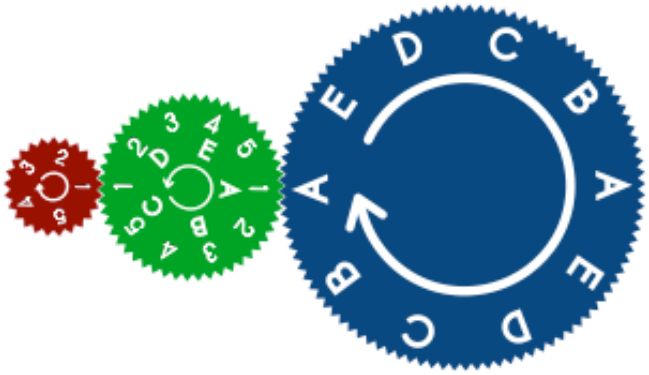

Fig. 4. Schematic Picture of Gear Scheduler

It works by assigning each process two indicators: $in_{progress}$ *and* $out_{progress}$ and only merging every component at the same time. In figure 3, the numbers represent the $in_{progress}$ of C_0 and $out_{process}$ of C_1 and letters represent $in_{progress}$ of C_1 and $out_{progress}$ of C_1 [7]. This way all trees fill up at the same time as more space becomes available downstream [7]. In particular:

$$in_{progress} = \text{Bytes read by merge}_i / (|C_{i-1}| + |C_i|)$$

$$out_{progress} = (\text{floor}(|C_i|/|RAM|_i) + \text{inprogress})/\text{ceil}(R)$$

It is important to note that $in_{progress}$ only grows by some constant factor and always ranges from 0 to 1. $out_{progress}$ also ranges from 0 and 1 and is set to 1 immediately before a new merge is triggered [7]. It is possible for one tree to shrink due to a delete. In this case, downstream merges are pauses until this tree fills up again.

V. DIFF-INDEX (DIFFERENTIATED INDEX)

A. Motivation

Diff-Index was inspired to maintain index based on a need for latency or consistency. For example, consider a bank and Youtube both using LSM. When a transaction is made in a bank, it is important for everybody to see the transaction at the same time. Consider a client that takes out all money from their debit bank account twice in a row. If the database is not updated right away, such a transaction is possible twice. Then it would be possible for a client to go into a negative debit balance – typically not a desired operation in banks. On the contrary, consider a user that makes a comment on Facebook. A user posting a comment on a video does not notice if there is a delay between the moment "Post" is clicked and the time that other users around the world see the comment. This inconsistency of data might be acceptable in order to improve latency.Diff-Index provides schemes depending on most prioritized functions of the database.

B. Diff-Index Schemas

More precisely, index schemas were motivated by the variety of data store goals - consistency, availability and partition-tolerance [7].

Fig. 5. Schematic Picture of Data Store Goals

Consider a scenario where User A makes a change to the database and User B tries to access that data. Potential concerns are:

- Will User A be able to see their changes right away?
- Will User B be able to see User A changes right away?
- Will User A have to wait to for their change to process and, if so, how long?

The four index update schemes are as follows that address these questions are as follows:

1) *sync-full*: to complete all index update tasks synchronously;
2) *sync-insert*: to insert new index synchronously but lazily-repair old index entries;
3) *async-simple*: to asynchronously execute index updates and guarantee eventual execution;
4) *async-session*: to achieve "read-your-write" semantics on top of *async-simple*.

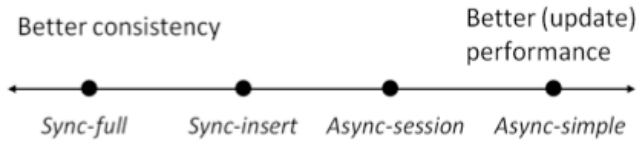

Fig. 6. Schematic Picture of Diff-Index Functionality

The following abbreviations are used throughout the section: $<k, v, t_s>$ is a tuple where k is the key, v is the value and t_s is the timestamp.

C. Scheme Sync-Full Implementation

Algorithm 1 Index update in sync-full

When a put $\langle k, v_{new}, t_{new}\rangle$ to base table is observed, do **SU1** to **SU4**:
SU1. Put data into base table: $P_B(k, v_{new}, t_{new})$;
SU2. Put index into index table: $P_I(v_{new} \oplus k, null, t_{new})$ or $P_I(v_{new} \oplus k, t_{new})$ to omit the null value;
SU3. Read the value of key k before time t_{new}: $v_{old} \leftarrow R_B(k, t_{new} - \delta)$;
SU4. Delete old index from t_{new} from index table: $D_I(v_{old} \oplus k, t_{new} - \delta)$.

Fig. 7. Algorithm for Scheme Sync-Full

Essentially, *sync-full* completely updates the entire database with respect to this insert. Consistency is guaranteed because every insert is fully processed all at once. Therefore every user will see the change at the same time.

D. Scheme Sync-Insert Implementation

Algorithm 2 Index read in sync-insert

Input: Index value v_{index}.
Output: List of rowkeys in base table K.
$K \leftarrow \emptyset$
SR1: Read index table and get a list of rowkeys with timestamps: $List(\langle k, ts\rangle) \leftarrow R_I(v_{index})$
SR2:
for all $\langle k, ts\rangle$ in $List(\langle k, ts\rangle)$ **do**
  read base table and get newest value of k with timestamp: $\langle v_{base}, tb\rangle \leftarrow R_B(k)$
  if $v_{index} == v_{base}$ **then**
    $\langle v_{index} \oplus k\rangle$ is an up-to-date index entry and add k to K
  else
    $\langle v_{index} \oplus k\rangle$ is stale and delete $\langle v_{index} \oplus k, ts\rangle$ from index table: $D_I(v_{index} \oplus k, ts)$
  end if
end for
return K

Fig. 8. Algorithm for Scheme Sync-Insert

The immediate procedure for sync-insert is to insert the new record and invalidate timestamps of all older indexes. The indexes are removed later on. This procedure is faster to insert but it slows down queries, as they have to check timestamps of every index entry. Consistency is still maintained but looking up the update might take longer as invalidated indexes are still being processed.

E. Scheme Async-Simple Implementation

Algorithm 3 Index update in async-simple

When a put $\langle k, v_{new}, t_{new}\rangle$ to base table is observed, do **AU1** to **AU2**:
AU1. Do base put $P_B(k, v_{new}, t_{new})$; If an index is defined, add the put to AUQ;
AU2. Acknowledge put SUCCESS to the client.

Algorithm 4 Background index processing in async-simple

BA1. If AUQ is not empty, dequeue a put $\langle k, v_{new}, t_{new}\rangle$ from it;
BA2. Read from base table the value for k at $t_{new} - \delta$: $v_{old} \leftarrow R_B(k, t_{new} - \delta)$;
BA3. Delete old index from t_{new}: $D_I(v_{old} \oplus k, t_{new} - \delta)$;
BA4. Insert new index $P_I(v_{new} \oplus k, t_{new})$.

Fig. 9. Algorithm for Scheme Async-Simple

The immediate procedure for async-simple is to put the request for update on a queue. The queue then processed one request at a time and eventually updates the database for every user. It is often problematic that even the client can not see their changes. Consistency is sacrificed because the update is not done right away, other users will see the older index even though it has been changed.

F. Scheme Async-Session Implementation

Async-session is the extension of async-simple that allows for the client to see their changes right away even though the index is not fully updated on the system. This does not improve consistency but it could prevent the client to think their transaction did not go through the database at all.

```
session s = get_session()
put(s, table, key, colname, colvalue)
getFromIndex(s, table, colname, colvalue)
end_session(s)
```

Fig. 8. Algorithm to Add Sessions to Async-Simple

VI. FRACTAL TREE

A. Motivation

Fractal Tree (FT) indexing was motivated to be a simple modification of the B Tree.

B. Implementation

Fractal Tree (FT) is an extension of B trees. In FT, every node contains a buffer where data is initially put. When a data is filled up, data is moved down one level. It is important to note that data is not moved in continuous blocks but rather one insert at a time [3].

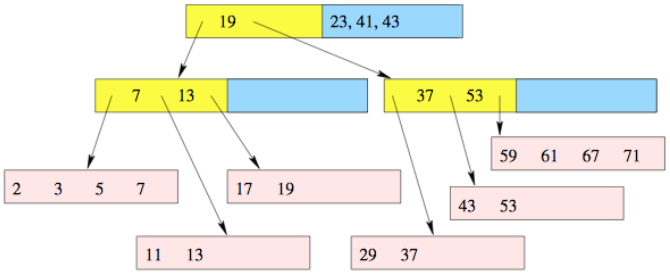

Fig. 10. Schematic Picture of Fractal Tree

C. FT Write Amplification

In the worst case, each object moves down the tree, once per level. Let B be the block size, N the size of data and k be the number of children. Then the total worst case cost is O(k $log_k N/B$) [3]. Read cost is O($log_k N/B$).

VII. BTREE, FRACTAL TREE AND LSM COST COMPARISION

A. A Light Comparison

Data Structure	Write Amp (worst case)	Read Amp (range) (cold cache)	Read Amp (range) (warm)	Space Amp
B Tree	$O(B)$	$O(\log_B N/B)$	1	1.33
FT index	$O(k\log_k N/B)$	$O(\log_k N/B)$	1	negligible
LSM leveled	$O(k\log_k N/B)$	$O((\log^2 N/B)/\log k)$	3	2 (file-per-run) 1.1 (many files)
LSM size-tiered	$O(\log_k N/B)$	$O(k(\log^2 N/B)/\log k)$	13	3

Fig. 11. Table Compares B, FT and Two LSM variations

Databases are often a trade off between space and time. While B Tree has best space amplification, it is the slowest of the three. Different LSM implementations provide us with a choice between more better write amplification or read amplification. Nevertheless, both LSM versions are faster on read and write than B trees and FT trees. This is a general pattern where efficiency is distributed between time and space constraints.

VIII. FUTURE DISCUSSION

A. An Overview of New Platforms and Hardware

Although many database structures have been around for decades, it is important to analyze their performance in today's technology. In particular, right now we have a high peak performance multi-core world [8]. While cores can help optimize time performance, the improvements in storage devices are a major breakthrough for database indexing as well. Flash storage in particular offers higher I/O ops per second at lower cost [8].

B. Bw-tree Architecture

Bw-tree is highly depended on these new technologies as it utilizes cores and new storage techniques.

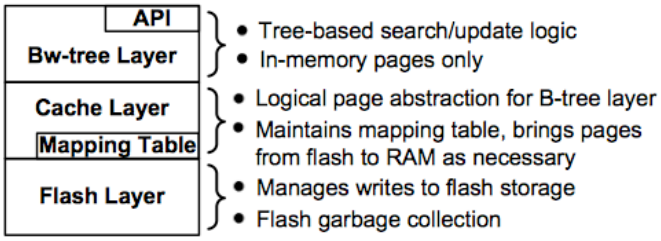

Fig. 12. Schematic Picture of Bw-tree Architecture

Although we don't give details on Bw-tree implementation in this paper, it's performance is impressive in comparison to B Trees in BerkeleyBD database.

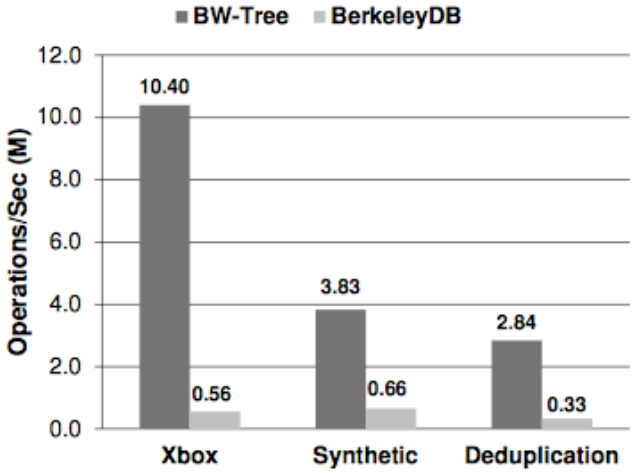

Fig. 13. Bw-tree Compared To BerkeleyDB

[1] J. Rao, K.A. Ross, "Making B+ trees cache conscious in main memory," SIGMOD '00 Proceedings of the 2000 ACM SIGMOD international conference on Management of data, pp. 475-486, *ACM*, 2000.

[2] W. Tan, S. Tata, Y. Tang., L.Fong, "Differentiated Index in Distributed Log-Structured Data Stores," *IBM,* 2013.

[3] B. Kuszmaul, "A Comparison of Fractal Trees to Log-Structured-Merge (LSM) trees," *Harvard*, April 2014.

[4] F. Chang, J. Dean, S. Ghemawat, W. Hsieh,D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber. Bigtable: A distributed storage system for structured data. ACM Transactions on Computer Systems (TOCS), 26(2):4, 2008.

[5] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels. Dynamo: Amazon's highly available key-value store. In SOSP, volume 41, pp. 205–220. ACM, 2007.

[6] P. O'Neil, D. Gawlick, E. Cheng, E. O'Neil, "The Log-Structured Merge-Tree (LSM)," *Acta Informatica,* vol. 33, pp. 351-385, 1996.

[7] R. Sears, R. Ramakrishnan, "bLSM: A General Purpose Log Structured Merge Tree," *Harvard*, pp. 217-228, 2012.

[8] J.J. Levandoski, D.B. Lomet, S.Sengupta, "The Bw-Tree: A B-tree New Harware platforms, " 2013 IEEE 29th International Conference on Data Engineering (ICDE), 2013.