CS5234 Project Report: Write Optimized Data Structures

Luo Wen Han
l.wenhan@u.nus.edu

Kennard Ng
e0036319@u.nus.edu

Abstract

The advent of deep learning has accelerated the rate at which enterprises learn from data. For these algorithms to work, large amounts of data are needed. This presents a challenge in today's context where the amount of data is overwhelming for trivial database systems. In this regard, we study write-optimized data structures used by modern database infrastructures to understand their theoretical and empirical performance and provide insight on improving data storage capabilities. Additionally, we show that improvements introduced in one data structure is also transferable to other data structures. Finally, we also give our recommendation on the more effective write-optimized data structures for modern database systems.

1. Introduction

With the rise of Industry 4.0, data has become an important resource for businesses and institutions alike. With more streams of data being readily available such as data from social media platforms and advertisements, the challenge in today's information is centered around storing and processing the overwhelming volume of data efficiently and effectively.

To manage the problem of data storage, most corporations use third-party database infrastructures that are built upon write-optimized data structures such as Log-Structured Merge Trees (LSMs) and Fractal tree index. LSMs are widely used in the storage layers of NoSQL systems such as BigTable [12], Dynamo [13], HBase [2], LevelDB [3], RocksDB [4], and AsterixDB [1] while MySQL databases such as TokuDB [5] use Fractal tree index as the underlying storage data structure. TokuDB which uses Fractal tree index has its underlying implementation originally built using Cache Oblivious Lookahead Arrays [7] (COLA) but has recent switched to using an extension of the B^ϵ tree [10].

Given that database infrastructures use different write-optimized data structures, an interesting research direction would go towards comparing the performance between these data structures and offering recommendations for effective data management. Additionally, another interesting study would be to compare these data structures theoretically and making empirical observations as to whether these theoretical guarantees hold in practice. Finally, we would want to observe if the improvements made in a single write-optimized data structure *e.g.* LSM, could be shared with another write-optimized data structure *e.g.* COLA for improved performance.

Hence, in this work, we investigate and compare the theoretical performance of three write-optimized data structures that are/were used in large-scale databases: namely the B^ϵ-tree, the log-structured merge tree (LSM), and the cache oblivious lookahead array (COLA). We also compare their empirical performance of these data structures on different insert/query patterns. In these experiments, we show that the best data structure depends largely on the system workloads where different data structures perform well across different workloads. Additionally, we improve the search query performance of LSM and COLA by introducing Bloom Filters to reduce the query IO cost of queries in COLA, and make empirical comparisons with their vanilla data structures. In doing so, we show that improvements in LSMs can also be transferred over to other data structures such as COLA. Our improvements have also shown to scale better than prior data structures as the size of data increases. Finally, we make our code publicly available at `https://github.com/Kennard123661/cs5234-project`.

2. Background Information

In this section, we describe the external memory model used for theoretical analysis of write-optimized data structures, the overarching theme for write-optimized data structures and analyze bloom filters, which are commonly used to improve the query performance of LSMs.

2.1. Memory Model

In modern machines, data storage is complicated where there is a hierarchy of different storage mediums with each medium having different storage capacities and access times. Memory at higher levels *i.e.* caches are often located near the processor where they can be accessed quickly whereas memory at lower levels *i.e.* external disks are located further away from processor and have significant IO access times. Since memory at higher levels need to have fast access times, they are often made up of expensive hardware such as RAM and have smaller storage capacities whereas memory at lower levels do not need to have faster access times and are made of less expensive materials, leading to larger storage capacities.

Given this hierarchy of faster but smaller storage mediums at higher levels and slower but larger storage mediums, memory accesses begin with higher levels, where lower levels are accessed if the required data is not found at higher levels. Accessing lower levels often happens when higher levels do not have sufficient capacity to contain the data to be loaded due to their limited capacity. In such scenarios, the expensive IO cost of accessing the disk memory located at lower levels becomes a bottleneck for algorithms where access times are approximately $\sim$1000x longer for memory at lower levels than memory at higher levels.

From earlier discussions on the memory structures of modern machines, it is evident that modern machines have complicated memory models that are difficult to analyze, given multiple levels of memory with different properties. In contrast to this, theoretical discussions of write-optimized data structures often use the simpler external memory model proposed by Aggarwal and Viter [6] for theoretical analysis of these algorithms. In this model,

- There are only two levels of data storage: main memory and disk.
- IO communication is done in blocks of B where B units of data is written/read between memories.
- The main memory has a limited size of M.
- The disk has a much larger capacity than main memory but has a higher IO cost, where a cost of 1 is incurred for each disk access to load a block of size B.

The external memory model also has a similar behavior to modern machine memory structure where, when an item location is accessed, we first look for the item in main memory. If the item is found in the main memory, we incur no cost. However, if the item is not found and has to be loaded from disk, the block containing the item's location and copy the block onto disk, which incurs a cost of 1.

Finally, given the limited capacity of the main memory, a block is evicted from memory to load another block from disk when the main memory is full. The block to be evicted depends on the cache management policy of the main memory *e.g.* Least Recently Used (LRU) where blocks that haven not been accessed recently are evicted first. Following recent literature, we also use the external memory model in our theoretical discussions.

2.2. Write-Optimized Data Structures

Write-Optimized data structure such as LSMs, COLA [7] and the B^{ϵ}-tree [10] improve upon the read and write speeds of database infrastructure. This is done by reducing the cost incurred by insert *i.e.* write and query *i.e.* read operations, under the external memory model. These two operations can be described to be the following:

- `insert <key,value>`: inserts `<key,value>` pair into the database
- `query <key>`: searches for `<key>` in the database and returns its corresponding `value`, other return `Invalid`.

In this work, we describe a `<key,value>` pair as an item and we assume `value` to be any arbitrary data type. For ease of analysis, we will only be handling the insertion of `keys` and assume that the corresponding `key` to `value` mapping in handle within a black-box. We represent each `key` as an integer in our work.

2.3. Bloom Filters

Bloom filters [9] are data structures used to determine set membership where the data structure returns false if the item definitely does not belong in the set and returns true if the item is possibly in the set *i.e.* bloom filters can return false positives but not false negatives. Bloom filters are often implemented as a vector of n bits with K hash functions. Given their small memory foot, bloom filters can be held in memory and they are often used in LSMs to reduce the cost of false queries *i.e.* queries where the item does not exist in the database.

To reduce the cost of false queries, bloom filters are first queried to check for set membership within each component before the more expensive query operation is performed. If the bloom filter returns False, the item definitely does not exist in the component and the expensive query operation does not happen. In this section, we will describe the insertion of items into the bloom filter, set membership queries within bloom filters, and analysis insert and query performance within bloom filters.

Insert Operation. To add an item x to a bloom filter, the item is first hashed through multiple hash functions $\{f_k\}_{1:K}$ to get the digests $\{f_k(x)\}_{1:K} \subseteq \{0, \cdots, n-1\}$. The bits of the vector indexed by the digests $f_k(x)$ will be set to 1 *e.g.* Figure 1.

Query Operation. To query the existence of an item within a bloom filter, the item is hashed through the hash functions to get digests $f_k(x) \in \{0, \ldots, n-1\}$. Then the bits indexed by the digests in the n-bit vector are extracted.

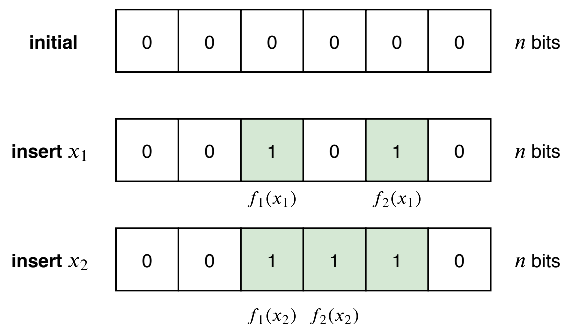

Figure 1. Initially, all bits in the vector are set to 0. When x_1 is added, it is first hashed by the hash functions $f_1(\cdot)$ and $f_2(\cdot)$ where $K = 2$. Then bits indexed by $f_1(x_1)$ and $f_2(x_2)$ are set to 1. The same procedure happens for x_2.

If all extracted bits are equal to 1, then the bloom filter returns True. Otherwise, it returns False.

Intuitively, if an item x exists within the set, the bits indexed by its digest will be set to 1 during the insert operation such that query operations returns True. On the other hand, if the item does not exist, bloom filters may return True during query operations when inserting other items $x' \neq x$ set the bits indexed by $\{f_k(x)\}_{1:K}$ to 1. An example is given in Figure 1, where x_2 and x_1 has overlapping digests such that inserting x_1 or x_2 sets the third bit to 1. We will analyze the false positive rate in section 2.3.1.

2.3.1 Analysis

In this section, we will discuss some claims on the performance of bloom filters, and discuss the optimal parameters that we use for bloom filters. We assume that the bloom filters have K hash functions, n bits and at most m items are added to the bloom filter.

Claim: The running time of insert/query operations in bloom filters is $O(K)$ where K is the number of hashes. **Proof:** During insertions, we hash an item K times for each of the hash functions and set/check the corresponding bit in the n-vector, where we check at K bit indexes.

Claim: The space complexity of bloom filters is $O(n)$ bits where n is the length of the bloom filter vector. **Proof:** n bits are needed to store the n-vector of the bloom filter. Further, we will discuss optimal choices of n and K, assuming a known m.

Claim: Bloom filters have false positive error rate of $p = \left(1 - e^{-Km/n}\right)^K$. **Proof**:

1. Given that an arbitrary hash function maps an integer x to $\{0, \dots, n-1\}$ with uniform probability, the probability that an arbitrary bit is not set by a hash function k is $(1 - 1/n)$.
2. Given K independent hash functions, the probability that a bit is not set when an item is inserted is $(1 - 1/n)^K$.
3. If m items are inserted, the probability that an arbitrary bit is not set is $(1 - 1/n)^{Km}$. Hence, the probability that the bit is set is:

$$\Pr[\text{bit set}] = 1 - \left(1 - \frac{1}{n}\right)^{Km}. \tag{1}$$

.

4. For a false positive to occur, all bits indexed by the digest of false positive x has to be 1. This probability is:

$$\Pr[\text{False Positive}] = \left[1 - \left(1 - \frac{1}{n}\right)^{Km}\right]^K. \tag{2}$$

5. Using the Taylor expansion, $e^x \approx 1 + x + \dots$, we have $1 - 1/n \approx e^{-1/n}$, where

$$\begin{aligned} \left[1 - \left(1 - \frac{1}{n}\right)^{Km}\right]^K &\approx \left[1 - \left(e^{-1/n}\right)^{Km}\right]^K \\ &= \left[1 - e^{-Km/n}\right]^K. \end{aligned} \tag{3}$$

Claim: Given a fixed m and n, the optimal number of hashes $K^* = n \ln 2/m$. **Proof** (from [11]):

1. We let the false positive rate p be:

$$\begin{aligned} p &= \left[1 - e^{-Km/n}\right]^K \\ &= \exp\left(K \ln\left(1 - e^{-Km/n}\right)\right) \end{aligned} \tag{4}$$

2. Given that e^x increases monotonically with x, minimizing $g = K \ln\left(1 - e^{-Km/n}\right)$ also minimizes p. Hence, taking the partial derivative of g w.r.t. K:

$$\begin{aligned} \frac{\partial g}{\partial K} &= \ln\left(1 - e^{-Km/n}\right) + K \cdot \frac{\frac{m}{n} e^{\frac{-Km}{n}}}{1 - e^{\frac{-Km}{n}}} \\ &= \ln\left(1 - e^{-Km/n}\right) + \frac{Km}{n} \frac{e^{\frac{-Km}{n}}}{1 - e^{\frac{-Km}{n}}} \end{aligned} \tag{5}$$

3. Setting $K = n \ln 2/m$, we see that:

$$\begin{aligned} \frac{\partial g}{\partial K} &= \ln\left(1 - e^{-ln2}\right) + \ln 2 \frac{e^{-\ln 2}}{1 - e^{-\ln 2}} \\ &= \ln\left(1 - \frac{1}{2}\right) + \ln 2 \frac{\frac{1}{2}}{1 - \frac{1}{2}} \\ &= -\ln 2 + \ln 2 = 0. \end{aligned} \tag{6}$$

This implies that p is minimum when $K = n \ln 2/m$.

Claim: Using $K^* = n \ln 2/m$, the optimal number of bits $n^* = -m \ln p/(\ln 2)^2$. **Proof**:

1. Taking the the natural logarithm on both sides of Equation 4:
$$\ln p = K \ln \left(1 - e^{-Km/n}\right). \tag{7}$$
2. Using $K = n \ln 2/m$, we have:
$$\begin{aligned} \ln p &= K \ln \left(1 - e^{-Km/n}\right) \\ &= \frac{n \ln 2}{m} \ln \left(1 - e^{-\ln 2}\right) \\ &= \frac{n \ln 2}{m} \ln \left(\frac{1}{2}\right) \\ &= -\frac{n(\ln 2)^2}{m} \end{aligned} \tag{8}$$
3. Then, the optimal number of bits is:
$$n = -\frac{m \ln p}{(\ln 2)^2} \tag{9}$$

3. Write-Optimized Data Structures

In this section, we will introduce the three write-optimized data structures that we will be studying: the B^ϵ tree, Log-Structured Merge Trees (LSMs) and Cache Oblivious Look-ahead Arrays (COLA). We assume there are N items in the data structure, a cache *i.e.* main memory size of M and block size of B.

3.1. B^ϵ-Tree

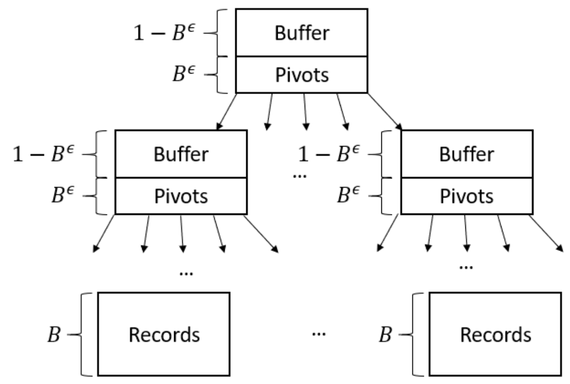

Figure 2. B^ϵ-tree

The B^ϵ-Tree is an augmentation of the classic B-Tree with buffers at each node to increase the write performance while maintaining the same query performance. All nodes of a B^ϵ-Tree are size B: branch and root nodes have B^ϵ space allocated to pivots and keys and $B - B^\epsilon$ space allocated to the buffer while leaf nodes have the whole B space used to store items.

$\epsilon \in [0, 1]$ is a hyperparameter in the B^ϵ tree where a higher value of ϵ increases the degree each non-leaf node and decrease the height of the tree, improving query performance. Conversely, a lower value of ϵ increases the size of the buffer, which batches more inserts for flushing and thus improving write performance, at the cost of lower query performance.

Insert Operations. Items are always first inserted into the buffer of the root node. If the buffer of the root node is full after an insertion, a flush operation is performed where items belonging to the child with the most pending items are removed and inserted into the buffer of the child.

The tree then recursively checks for fullness and flushes when necessary the child nodes, until the items are collected in the leaf nodes. When leaf nodes are full, the $B^epsilon$-tree mirrors the B-tree and splits the leaf node, and proceeds to climb up the tree and recursively splitting any branch nodes with full pivots. In the case of splitting the branch nodes the buffer is distributed to the split nodes using the median of the keys.

Query Operations. Query operations are the same as that of a B-tree, except that at every node the tree additionally checks for the existence of the queried item in the buffer.

3.1.1 Analysis

Claim: A B^ϵ-tree with N items has a height of $O(log_{B^\epsilon} N)$. **Proof**: The maximum number of children per node is $O(B^\epsilon)$, i.e. the size of the pivots in the node. Thus the maximum height of the tree is $O(log_{B^\epsilon} N)$.

Claim: Insert operations into the B^ϵ-tree has an amortized cost of $O(\log_{B^\epsilon} N/B^{1-\epsilon})$ block transfers. **Proof**:

1. When the buffer of a branch node is full (i.e. the buffer contains $B - B^\epsilon$ items), the full node and the child node with the most pending number of items must be retrieved from disk, incurring a cost of $O(1)$.
2. Since a flush can remove at least $(B - B^\epsilon)/B^\epsilon = O(1/B^{1-\epsilon})$ items from the buffer, the number of inserts between flush is at least $O(1/B^{1-\epsilon})$.
3. Flushes can only happen from the root of the tree down to the leaves, therefore the maximum number of flush for an insert is the height of the tree: $O(\log_{B^\epsilon} N)$.
4. Therefore, the insert cost of an item is $O(\log_{B^\epsilon} N/B^{1-\epsilon})$ amortized block transfers.

Claim: Queries operations into the B^ϵ-tree has a cost of $O(\log_{B^\epsilon} N)$ block transfers. **Proof**: The maximum height of the tree is $O(\log_{B^\epsilon} N)$, which is the max length the tree has to traverse for a query.

Corollary: At $\epsilon = 0.5$, insert costs $O\left(\log_B N/\sqrt{B}\right)$ amortized disk reads, an improvement over classic B-Trees by a factor of $\sqrt{B}$. Query costs $O\left(\log_B N\right)$, the same as classic B-Trees.

3.2. Log Structure Merge Tree

In the original paper on log-structure merge-tree (LSM-Tree) by O'Neil, Patrick, et al [14], the data structure is described as a ensemble of 2 or more separate "components" or tree structures of increasing size. Each component is optimized for their underlying hardware, e.g. a component in the RAM can be implemented as a skip list while a component in the disk can be implemented as a B-Tree.

LevelDB [3] is a popular implementation of the LSM-Tree. It uses a multi-component structure where the first component resides in memory while the remaining components are stored in the disk. We assume LevelDB's LSM implementation for our analysis of the LSM-Tree.

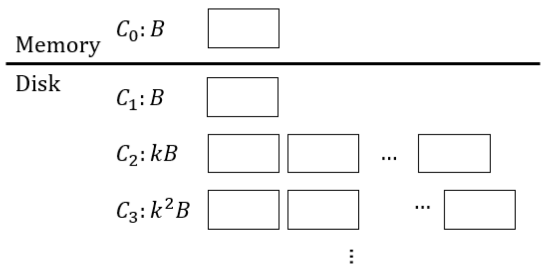

Figure 3. Basic LevelDB Structure

Structure. LevelDB's LSM-Tree consists for multiple components: C_0 of size B located in memory and $C_1 \cdots C_i$ located on disk with increasing size of a factor of k. At every level, records are guaranteed to be in sorted order. In practice LevelDB implements $k = 10$.

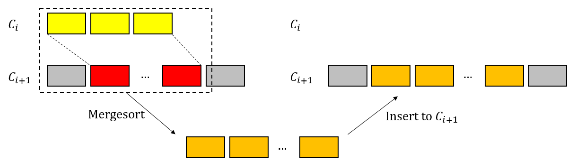

Figure 4. LSM Compaction

Insert Operations. When a record is inserted into the LSM-tree, the record is inserted to the only block in C_0; as such no IO cost is incurred. When the block in C_0 is full, the block is made immutable and dumped into the next level, C_1, which resides on disk. If at any point a level in the disk is full, compaction occurs where the full level is merged into the next level. The compaction is optimized for time and space by only retrieving overlapping blocks in the next level and performing a mergesort with the blocks of the full level, after which the sorted blocks are inserted into the next level (see Figure 4).

Query Operations. For queries, each component C_0 to C_L where L is the total number of levels is searched in sequential order. In order to search a component, a simple binary search can be performed since each component is sorted.

3.2.1 Analysis

Claim: A LSM-tree with N items contains $O(\log_k(N/B))$ components. **Proof**: The number of items in $C_0 = C_1 = B$. For every subsequent level, $C_i = C_{i-1} \cdot k$ for some growth factor k. Then, $\sum_{i=0}^{L} C_i = B + B + kB + k^2B + \cdots + k^{L-1}B = O(k^L B)$. Therefore, L = $O(\log_k(N/B))$.

Claim: An item is written to disk at most $O(k \log_k(N/B))$ times. **Proof**:

1. An item is inserted into a level once.
2. When merging with a previous level, The same item is merged back into the same level at most k times.
3. Therefore, an item is written to disk at most $O(k \log_k(N/B))$ times.

Claim: Insert operations into the LSM-tree has an amortized cost of $O(\log_k(N/B)/B)$ block transfers. **Proof**:

1. When a level containing X items C_i is full, the cost to merge C_i to C_{i+1} is $O(kX/B)$ (Note that the number of items in $C_{i+1} = kX$).
2. Hence, the cost per item to merge C_i to C_{i+1} is $O(1/B)$.
3. Each item is moved at most $O(\log_k(N/B))$ times.
4. Therefore, the insert cost of an item is $O(\log_k(N/B)/B)$ amortized block transfers.

Claim: Query operations of a LSM-tree has a cost of $O((\log^2 N/B)/\log k)$ block transfers. **Proof:**

1. The number of items per level in terms of N is $C_{L-n} = O(N/k^n)$, where L is the highest level. The highest level C_L contains $O(n)$ items.
2. Then, the number of blocks per level is $C_{L-n} = O(N/k^n B)$.
3. The number of disk reads need to perform binary search on a level is $C_{L-n} = O(\log N/k^n B)$.

4. In the worst case, binary search must be performed at every level. Therefore the cost of a query is

$$\sum_{i=0}^{\log_k (N/B)} C_{\log_k (N/B)-i} = O((\log^2 N/B)/\log k)$$

.

3.3. Log Structure Merge Tree with Bloom Filters

As we can see from the analysis of the LSM-tree, the LSM-tree performance on queries are a downgrade compared to classic B-trees. One of the ways to improve query performance is to implement memory-resident bloom filters (see 2.3.1) at every component of the LSM-tree. When querying for an item, the bloom filters are used to determine which level the item may be in, allowing the LSM-tree to only perform binary search on levels where the items could possibly be in. We shall show that implementing bloom filters improves the query performance of the LSM-tree to be comparable to classic B-trees.

3.3.1 Analysis

Claim: Query operations of a LSM-Tree with bloom filters at every level has a cost of $O(\log(N/B))$ block transfers. **Proof:**

1. We assume that the bloom filters have a low false positive rate.
2. In the worst case, the bloom filters return that the item exist at the highest level.
3. Binary search on the highest level incurs a cost of $O((\log(N/B))$ disk reads.
4. Therefore the cost of a query is $O(\log(N/B))$.

3.4. Basic Cache-Oblivious Look-ahead Array

A cache-oblivious look-ahead array (COLA) stores items in a contiguous array array **A**. **A** is divided into smaller sorted sub-arrays $\mathcal{A}_i$ that have size 2^i, and sub-array $\mathcal{A}_i$ contains items only if the i^{th} least significant bit of N is 1. For example, if there are $N = 4$ items in COLA, which corresponds to 100_2 in binary, only sub-array $\mathcal{A}_2$ will contain items while other sub-arrays will be empty. The basic structure of COLA can be visualized in Figure 5.

Insert Operations. When item x is inserted in COLA, it is first inserted into the first sub-array $\mathcal{A}_0$. If $\mathcal{A}_0$ is empty, the item can be inserted in $\mathcal{A}_0$. Otherwise, the item is merged with $\mathcal{A}_0$ in sorted order to produce a merge array $\mathcal{A}'_0$, which contains the items in $\mathcal{A}_0$ and the item x in sorted order. Notice that the size of this merge array is 2 given that the size of $\mathcal{A}_0$ is one and x is only one item.

Once the merge array at level 0 *i.e.* $\mathcal{A}'_0$ is formed, it is inserted into sub-array $\mathcal{A}_1$ which has size 2. If $\mathcal{A}_1$ contains

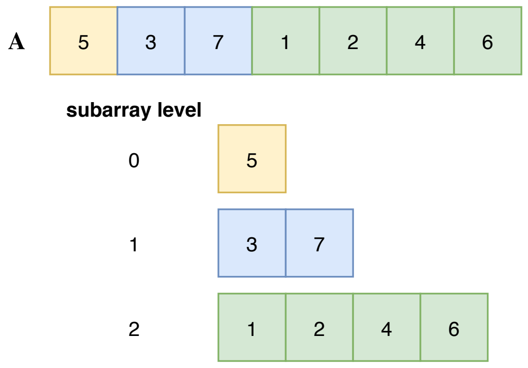

Figure 5. Basic COLA structure. Notice that items are in each sub-array $\{\mathcal{A}_i\}_{0:2}$ are sorted within the sub-array but not across the entire array **A**.

items. The same procedure occurs where the sorted merge array $\mathcal{A}'_1$ is formed and is inserted into $\mathcal{A}_2$. This process occurs recursively down the sub-array level until the merge array can be complete inserted at the level. Notice that at each level, the merge array at $\mathcal{A}'_i$ contains 2^{i+1} items and the sub-array at the next level has size 2^{i+1}. An example of this insertion process is illustrated in Figure 6

Query Operations. During query operations, we search down the levels of sub-arrays starting from sub-array $\mathcal{A}_0, \mathcal{A}_1, \ldots$. Given that each sub-array $\mathcal{A}$ is sorted, binary search can be performed at within each sub-array to search for any arbitrary item x.

3.4.1 Analysis

Claim: To store N items, $\lceil \log_2(N+1) \rceil$ levels of sub-arrays are needed. **Proof**: Each sub-array $\mathcal{A}_i$ stores 2^i items and $\sum_{i=0}^{L-1} |\mathcal{A}_i| = 2^0 + 2^1 + \ldots + 2^{L-1} = 2^L - 1$. Hence, $L = \log_2(N+1)$.

Claim: The lowest sub-array *i.e.* $\mathcal{A}_{\log_2(N+1)-1}$ of basic COLA has size $\lceil (N+1)/2 \rceil$. **Proof:** From the size of the last sub-array:

$$\begin{aligned} 2^{\log_2(N+1)-1} &= 2^{\log_2((N+1)/2)} \\ &= \frac{N+1}{2} \end{aligned} \tag{10}$$

Claim: Insert operations in basic COLA has an amortized cost of $O(\log N/B)$ block transfers. **Proof**:

1. The number of block transfers to merge an array of length k is $O(k/B)$ at an arbitrary level.
2. Hence, the cost to merge at each item in the array is $O(1/B)$.
3. In the worst case, an item experiences $O(\log N)$ merges.

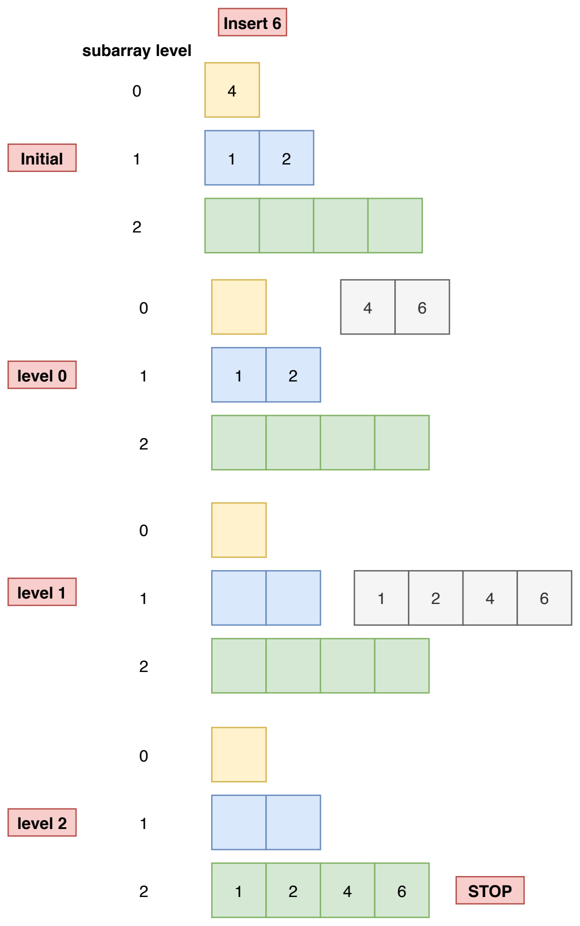

Figure 6. COLA insert example. In this example, we insert 6 into COLA. Given that $\mathcal{A}_0$, we merge the items to get $\mathcal{A}'_0 = [4, 6]$ Since $\mathcal{A}_1$ is full as well, we merge again to get $\mathcal{A}'_1 = [1, 2, 4, 6]$. Finally, since $\mathcal{A}_2$ has enough space, we can insert $\mathcal{A}'_1$ into $\mathcal{A}_2$ without merging any further, and the insert operation can be terminated.

4. As such, each item incurs an amortized cost of $O(\log N/B)$ block transfers.

Claim: Query operations in basic COLA has a cost of $O(\log N \cdot \log(N/B))$ block transfers. **Proof:**

1. From our previous claim, there are $O(\log N)$ levels in COLA.
2. At each level, binary search is performed, the lowest level contains $O(N)$ items, where binary search takes $O(\log N)$ access $O(\log N)$ locations. Given that data is loaded in blocks of B, we load $O(\log(N/B)$ blocks. Hence, binary search has $O(\log(N/B))$ block transfers.
3. Hence, the total number of block transfers for $O(\log N)$ levels is $O(\log N \cdot \log(N/B))$ block transfers.

Claim: The best case for true queries *i.e.* query for items that exist in COLA, incurs no IO cost. **Proof**: If the item is found in the first sub-array A_0 which has size 1, the query trivially returns True. Given that the block size $B \geq 1$, the first array can be fit into memory without incurring IO cost.

Claim: The best case bound for false queries *i.e.* is $\Omega(\log N \cdot \log(N/B))$. **Proof:** Query operations in COLA returns false if the item is not found after searching $O(\log N)$ levels, which incur a cost of $\Omega(\log N \cdot \log(N/B))$ block transfers. Hence, the best case bound for false queries in COLA is $\Omega(\log N \cdot \log(N/B))$. From our previous claim, it can be seen that the best case cost of block transfers for true and false queries in COLA is skewed. To reduce the best case cost for false queries, we can use bloom filters.

Claim: COLA with bloom filters can reduce the best case bound for false query operations to incur no IO cost. **Proof:** Given that bloom filters are space efficient and leave minimal memory footprint, they can be stored in main memory, incurring no IO cost. Prior to querying items in COLA, the bloom filter can be queried before starting query operations in COLA. This incurs no IO cost in the best case when the false query item detected to not exist in COLA by the bloom filter before querying COLA.

Claim: COLA with bloom filters implemented at every sub-array level can reduce the cost of query operations to $O(\log N)$. **Proof**:

1. From previous proofs, there are $O(\log N)$ sub-array levels in COLA.
2. With Bloom filters implemented at every level, the bloom filter can be checked before performing binary search at every level. For sub-arrays that do not have the item, the bloom filter check prevents binary search from happening. Given that the bloom filter is memory efficient, it incurs no IO cost.
3. If the bloom filter allows the binary search, it is probable that the array contains the item. In this case, it is the only array where the item is found and incurs cost $O(\log(N/B))$ when performing binary search in this array.
4. Hence, the total cost of query is $O(\log N + \log(N/B)) = O(\log N)$ block transfers.

3.5. COLA with Fractional Cascading

From the previous section on basic COLA, we see that the cost of naive query operations in basic COLA is $O(\log^2 N)$. Bender et. al. [7] improve the cost of search queries in COLA by introducing fractional cascading. In fractional cascading, every 8th item in sub-array $\mathcal{A}_{i+1}$ has real look-ahead pointers in sub-array $\mathcal{A}_i$. These real look-ahead pointers:

1. have a pointer to the location of the original item in A_{i+1}.
2. contain a duplicate of the item that they copied in A_{i+1}.

Additionally, every 4th item at each sub-array A_i is reserved to contain virtual look-ahead pointers that point to their nearest real look-ahead pointers to the left and right in A_i, allowing for fast localization of real look-ahead pointers. An illustration of real and virtual look-ahead pointers can be seen in Figure 7.

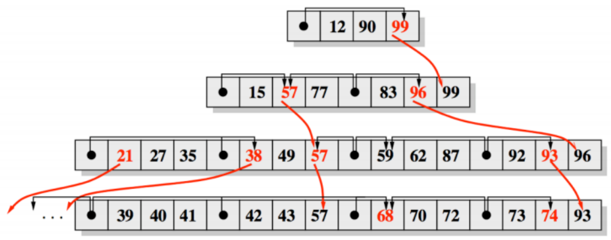

Figure 7. **Fractional Cascading Example**. Real look-ahead pointers (in red) have pointers to their originals in the next sub-array. On the other hand, virtual look-ahead pointers (in black dots) point to their nearest real look-ahead pointers to their left and right.

Insert Operation. Insertions for COLA with fractional cascading is performed similarly to insertions for basic COLA, where items are merged if the sub-array is full and items are insert down into the next sub-array. However, during merges, real look-ahead pointers are not included in the merge array and are removed.

Additionally, once the original merge operation from basic COLA has completed, real look-ahead pointers and virtual look-ahead pointers are added to sub-arrays above the final sub-array where the items are inserted. Every 8th item is duplicated with a real look-ahead pointer in the sub-array on the previous level. This process happens recursively through COLA *i.e*. real look-ahead pointers can also be created to point real look-ahead pointers *e.g*. Figure 7, until no more real look-ahead pointers can be created.

Query Operation. Queries for COLA are sped up using fractional cascading where the real look-ahead pointers can reduce the search space at the next level. During binary search at sub-array A_i where the item is not found, the lower and upper bound of the search is returned by the binary search algorithm. Then, using the virtual pointers, the nearest look-ahead pointers encasing the lower and upper bound can be obtained. Within the next sub-array, only the region enclosed by the real look-ahead pointers need to search rather than the entire sub-array.

An example can be in given Figure 7 where we search for key 92. At the first sub-array, the binary search algorithm returns the lower bound 90 and upper bound 92. The nearest look-ahead pointers are then found to be 99 only. Then, the search range in the 2nd sub-array is between the start of the array and the location of 99 pointed to in the 1st sub-array. The search is performed on the reduced search space in the second sub-array and terminates at [83,96]. The two nearest look-ahead pointers are then 57 and 96. The nearest look-ahead pointers are then used to reduce the search space in the 3rd sub-array where we only search within [57,96] and use binary search to finally find the key 92.

3.5.1 Analysis

Claim : Search queries in COLA with fractional cascading takes $O(\log N)$ search cost. **Proof:**

1. From our earlier proofs, there are $O(\log N)$ sub-array levels that need to be searched
2. At each sub-array with look-ahead pointers, the search space can be reduced to the constant of 8 given that each sub-array has every 8th item duplicated in its previous sub-array as a real look-ahead pointer. This implies that the distance between two look-ahead pointer is a constant factor of 8. As such, the cost of binary search at each sub-array level is $O(1)$.
3. Given $\log N$ levels, the total cost of query operations is $O(\log N)$ for COLA with fractional cascading.

Claim: Insert queries in COLA with fractional cascading has an amortized cost of $O\left(\log N/B\right)$ block transfers. **Proof**:

1. From our previous, we see that there is an amortized cost of $O\left(\log(N/B)\right)$ for basic COLA. We use this cost for the downward insertion and merge operation.
2. Then, we now consider the case of inserting real pointers in the reverse order *e.g.* $\mathcal{A}_i, \mathcal{A}_{i-1}, \ldots$. Given that every 8^{th} item is inserted, there are $O(N/8 + N/64) = O(N)$ real-lookahead pointers inserted, where $O(N/B)$ block transfers are needed.
3. If we amortize this among the items merged, we see that the cost is $O(1/B)$. Hence, the total amortized cost is $O\left(\log(N/B) + 1/B\right) = O\left(\log(N/B)\right)$

3.6. Theoretical Comparisons Across Data Structures

From our claims in previous sections, we compare the theoretical bounds for each of the write-optimized data structures. We added B-trees as a baseline comparison with other data structures. We present our theoretical comparisons in Table 1.

Data Structure	**Insert**	**Query**
B Tree	$O\left(\log(N/B)\right)$	$O\left(\log(N/B)\right)$
B^ϵ Tree ($\epsilon = 0.5$)	$O\left(\log(N/B)/\sqrt{B}\right)$	$O\left(\log(N/B)\right)$
LSM	$O\left(\log_k(N/B)/B\right)$	$O\left((\log^2 N/B)/\log k\right)$
LSM+BF	$O\left(\log_k(N/B)/B\right)$	$O\left(\log(N/B)\right)$
COLA	$O\left(\log(N/B)\right)$	$O\left(\log N \cdot \log(N/B)\right)$
COLA+BF	$O\left(\log(N/B)\right)$	$O\left(\log N\right)$
COLA+FC	$O\left(\log(N/B)\right)$	$O\left(\log N\right)$

Table 1. Comparison of Theoretical Bounds Across Write-Optimized Data Structures.

3.7. Implementation Details

For all data structures, we ensure they support the $insert(i)$ and $query(i)$ operations for any 32-bit integer i. In addition to the write-optimized data structures discussed earlier, we added B-tree as a baseline data structure.

B and B^ϵ-Tree. The B and B^ϵ-Tree implementation uses a file backed LRU cache to access its nodes. The cache have a set memory size that can be set during experimentation. Similar to the cache model, nodes are retrieved from disk if not already loaded into the cache, writes to the cache does not incur a IO cost to the disk, and when the cache is full the least-recently used node is dumped back onto disk.

The B tree implementation is implemented as a $B+$ tree, where nodes only contain keys and leaf nodes contain all of the items. The B tree implementation differs from the B^ϵ-Tree implementation in that only the leaf nodes are disk-backed, the root and branch nodes of the B tree remains permanently in memory.

Rather than having a set ϵ, we instead set B^ϵ (i.e. the branching factor) as other real-world implementations of the B^ϵ-Tree also sets B^ϵ ranging from 4 to 16. [8]. This also makes it easier to make a fair comparison to the LSM-Tree and COLA by setting a similar growth factor in those data structures.

Log-Structured Merge Tree. For our implementation of the LSM-tree, all components are implemented with a sorted list. When the memory-resident component is full, it is dumped into disk into a folder denoting the level of the component. When the number of files in the folder (blocks in the component) exceeds the limit, overlapping blocks from the next level are retrieved and a merge-sort is performed. The sorted blocks are then dumped in the next folder. Each level has a metadata object stored as a file which contains the name of the blocks and the starting key of each block in sorted order.

The implementation of the metadata object may have caused an unintended optimization of the LSM-tree that deviates from the original. Instead of needing to perform a binary search on a level incurring $O(\log k^i)$ disk reads for the ith level, we can simply read the metadata, perform a binary search on the starting index of the blocks to determine which block the item is likely in, then read the block to check if the item is in the level. This reduces the cost of a query to $O(\log_k(N/B))$, i.e. the total number of levels in the tree.

The LSM-Tree has also been implemented with memory-resident bloom filters for each disk-resident component, which can be turned off with a flag.

Cache Oblivious Lookahead Array. For our COLA implementations, we use an arbitrary growth factor g that can be set during experiments. Our previous examples in our analysis use $g = 2$. To simulate our disk, we read/write to a HDF5 storage file given that our data is stored contiguously in an array. This allows us to stream the data dynamically and extract our locations of the disk through indexing.

Furthermore, to reduce the amount of disk accesses, we implement a counter at every sub-array level that counts the number of items in the sub-array level. In doing so, COLA does not need to write to disk at every merge to set arrays that were flushed during merges to null values, and can instead set the counter to 0.

In our COLA + Fraction Cascading (FC) implementation, we represent each sub-array as two arrays, following a similar implementation used in experiments of [7]. In our implementation, one array represents the visible items in the array, which includes the duplicates from real look-ahead pointers. The second array is a pointer array where array items that are real look-ahead pointers have pointers to their original copies in the next sub-array while other array items contain virtual pointers that point to the nearest look-ahead pointer to its left. Additionally, we have a third bit array for each sub-array to indicate which array items are real look-ahead pointers. For our COLA + Bloom Filter (BF) implementation, we add a global bloom filter that registers membership for all items in COLA and a bloom filter at each sub-array level to register membership for items at each sub-array level.

4. Experiments

In this section, we discuss the experiments we perform to compare the empirical performance of each data structure. In our experiments, we are interested in comparing the performance of the different data structures across different types of data. We split this into two categories: the first category compares across different natures of data *e.g.* sorted, random, and the second category of data relates to the distribution of operations where we vary the proportion of query and insert operations. We present the former experiments in Section 4.1 and the latter experiments in Section 4.2. Finally, we compare the scalability of the write-optimized data structures across different number of input operations.

To ensure a fair comparison, we set the branch factor of the B Tree and the B^ϵ Tree to 8 and the growth factor of LSM Tree and COLA to 16. We also set the same block size and memory limit for all data structures.

4.1. Non-interspersed inserts and queries

In this experiment, we generate three different data sets where inserts operations are performed prior to queries. The sequential data set inserts and queries integers from 0 to N in ascending order. The reversed data set inserts and queries integers from N to 0 in descending order. The random data set inserts and queries integers from 0 to N in a random order. In all data sets queries are guaranteed to return true. These data sets represent real-world usage of logging massive amount of data, then inspecting them afterwards. We set the value of $N = 10^6$. We present the average number of operations per second for across the three data sets for each write-optimized data structure in Table 2.

Discussions. Across all three data sets, we observe that LSM Tree has the best performance for inserts. In fact LSM Tree outperforms the other non-LSM data structures by almost a factor of 10 on random inserts. This is likely due to compact operations being extremely efficient compared to flush operation in the B^ϵ Tree or the merge operation in the COLA. The bloom filter versions of the LSM Tree and COLA performed worse than their original counterparts, which can be attributed to the performance loss from rebuilding the bloom filters.

In terms of query performance, B-Tree performed the best when the data arrives in some sorted order *i.e.* forward/reverse order, followed by B^ϵ Tree by a huge margin over other data structures. This difference can be attributed to implementation differences between B-Tree based data structures and other data structures; B and B^ϵ Trees both uses a LRU cache to cache to leaf nodes. Thus for a sequential range of B queries, either forward or reversed, B and B^ϵ only needs to read the leaf node from disk once and perform the rest of the queries from the cache. Random queries will likely result in reading random leaf nodes and thus evicting others from the limited cache, resulting in a poorer query performance for B and B^ϵ Trees in random queries, which is proven by the query performance of both being the worst among all of the other data structures for random queries. We can also see that implementing bloom filters does increase the query performance of the data structures.

On the random data set, we see that COLA+BF performs the best for random queries where each sub-array level is queried in the Bloom filter before performing a binary search at each sub-array level. Given that binary search at each level cost $O(\log N/B)$ as shown earlier and the bloom filter incurs no IO cost, we only incur a single binary search cost for the sub-array that the item is located in where our theoretical search cost is $O(\log(N/B) + \log N) = O(\log N)$

Finally, we observe that fractional COLA does not perform well relative to other write-optimized data structures. We attribute this to the possibility that having additional pointers significantly results in more blocks being transferred, and also to the possibility that our implementation may not faithfully replicate the implementation of the authors of COLA [7].

For further experiments, we use data sets with random distributions. We believe that this suits the theme of real-world data where the data has no particular ordering. For meaningful and practical experiments, we use the random data distribution for subsequent experiments.

4.2. Interspersed inserts and queries

Another interesting experiment we perform is to compare the performance of the write-optimized data structures with different distributions of query/insert operations. Hence, we compare the time taken to perform interspersed inserts and queries of differing ratios. With a given N and insert ratio $0 < R < 1$, RN random integers from range $[0, N)$ are inserted and $(1 - R)N$ random integers from range $[0, N)$ are queried. In this data set there would false queries where the item queried does not exist in the data structure yet. This data set represents a real-world usage where inserts and queries are happening concurrently (e.g. a web server). We set the value of $N = 10^6$ and compare $R = 0.2, 0.4, 0.6, 0.8$.

Additionally, we omit the performance of Fractional COLA in our results in Figure 8 given that its poor query performance in Table 2 result in significantly longer time to completion when compared to other write-optimized data structures.

Discussions. Consistent with Table 2, we see that B Tree performs better with a smaller proportion of insert operations where the data structure suffers badly when the num-

	Sequential		Reversed		Random	
Data Structure	Inserts	Queries	Inserts	Queries	Inserts	Queries
B Tree	7523	**122951**	7631	**133202**	473	385
B^ϵ Tree	14959	40269	14906	40496	2917	566
LSM Tree	**250000**	4543	**250000**	3074	**31250**	789
LSM + BF Tree	71429	5138	71429	4590	19467	1356
COLA	3573	1018	3912	1079	3628	1037
BF-COLA	2641	1655	2727	1597	2623	**4142**
Fractional COLA	1284	502	1920	528	1278	4

Table 2. **Mean operations per second**

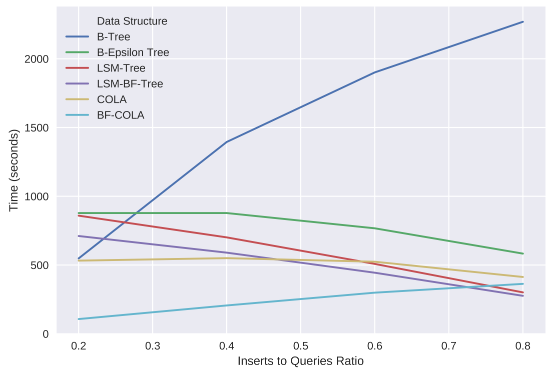

Figure 8. **Plot of time to complete against inserts to queries ratio**

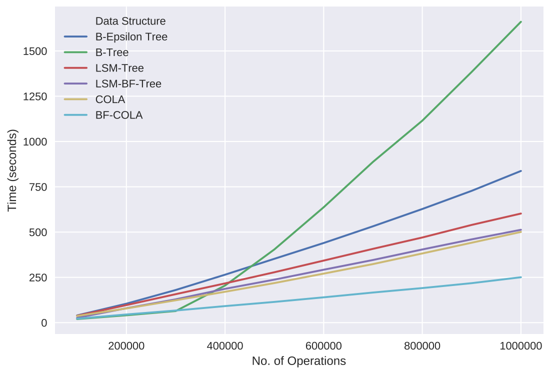

Figure 9. **Plot of time to complete against number of operations**

ber of inserts is increased. On the other hand, other write-optimized data structures have better performance as the proportion of inserts increase. We see that the performance COLA+BF and COLA begin to converge but do not intersect even when the insert ratio is 0.8 where we can observe that the overhead from the BF do not add significant overhead during inserts when increase in the insert ratio marginally increases the time required for COLA+BF to reach completion.

Additionally, we see that our results are consistent in that LSM performs the best when the proportion of insert operations increases but COLA+BF performs the best when the proportion of query operations dominate given that there are more false queries.

4.3. Varying input size with interspersed inserts and queries

In this experiment, we want to analyze empirically the growth rate of the time required to complete a series of interspersed inserts and queries based on the operations. We set the $R = 0.5$ (i.e. equal number of inserts and queries) and varied N from 10^5 to 10^6 in increments of 10^5.

Discussions. We can see from Figure 9 that the time taken for write-optimized data structure to complete the operations seem to increase linearly with the number of operations. For the B-Tree, the time taken increases linearly slowly at first then slowed down. The point it slows down is likely when the internal cache of the B-Tree have been depleted. The rest of the results are consistent with our previous experiments: for write-optimized data structures COLA+BF seem to perform the best while B-Epsilon Tree performed the worst almost all write-optimized data structures throughout all of the input size.

Additionally, we see that write-optimized data structures are more scalable than their baseline B-tree counterpart which experiences an exponential increase in time taken for completion compared to write-optimized data structures, which further motivates the use of write-optimized data structures for database operations.

5. Conclusion

In conclusion, we show that while write-optimized data structures may appear to have the same theoretical bounds, there are subtleties in their performance that allows some data structures to empirically outperform others with similar theoretical performance. Additionally, we see that write-optimized data structures scale better with increasing number of data compared to non-write optimized data structures such as the B-tree which motivates their use for large-scale data management that is prevalent in today's digital age.

Furthermore, we see that no write-optimized data structure surpasses the performance of all other data structures across insert and query operations, and across all data distributions. This hints that data administrators should choose data structures that fit their desired workloads. Despite this, we see that any choice of modern database infrastructure today surpasses a trivial database system that implements B-tree for large scale data sets. Hence, our recommendation would be to continue using modern write-optimized data structures for data management, but choose write-optimized data structures based on the workload requirements.

Finally, we show that improvements in LSM can also be transferred over to COLA where we observe significant improvements in the performance of COLA through the use of bloom filters. This motivates further research into transferring improvements across adata structures, and developing a universal write-optimized data structure that generalize well over different conditions.

References

[1] Asterixdb. https://asterixdb.apache.org/. Accessed: 2019-11-12. 1

[2] Hbase. https://hbase.apache.org/. Accessed: 2019-11-12. 1

[3] Leveldb. https://github.com/google/leveldb. Accessed: 2019-11-12. 1, 5

[4] Rocksdb. https://rocksdb.org/. Accessed: 2019-11-12. 1

[5] Tokudb. https://www.percona.com/software/mysql-database/percona-tokudb. Accessed: 2019-11-12. 1

[6] A. Aggarwal and S. Vitter, Jeffrey. The input/output complexity of sorting and related problems. *Commun. ACM*, 31(9):1116–1127, Sept. 1988. 2

[7] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J. Nelson. Cache-oblivious streaming b-trees. In *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, pages 81–92, New York, NY, USA, 2007. ACM. 1, 2, 7, 9, 10

[8] M. A. Bender, M. Farach-Colton, W. Jannen, R. Johnson, B. C. Kuszmaul, D. E. Porter, J. Yuan, and Y. Zhan. And introduction to be-trees and write-optimization. *login; magazine*, 40(5), 2015. 9

[9] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970. 2

[10] G. S. Brodal and R. Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics. 1, 2

[11] A. Broder and M. Mitzenmacher. Network applications of bloom filters: A survey. *Internet Mathematics*, 1(4):485–509, 2004. 3

[12] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Trans. Comput. Syst.*, 26(2):4:1–4:26, June 2008. 1

[13] 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. *SIGOPS Oper. Syst. Rev.*, 41(6):205–220, Oct. 2007. 1

[14] P. O'Neil, E. Cheng, D. Gawlick, and E. O'Neil. The log-structured merge-tree (lsm-tree). *Acta Informatica*, 33(4):351–385, 1996. 5