**Undergraduate Research & Development Project**

# I/O Optimized Index Structures

*by*

J. Guna Prasaad
110050082

*under the guidance of*
Prof. S. Sudarshan

Department of Computer Science & Engineering
Indian Institute of Technology Bombay
Mumbai, India

# Contents

# Chapter 1

# Introduction

With the advent of Big Data, high performance in a database management system has become more essential than ever. This need takes on various forms in various kinds of applications. Most of the traditional database systems were designed to be used either for transactional or analytical workloads. Transactional sysems are generally deployed in interactive, user facing portions of the applications. Such systems generally optimize for random reads and worst case write latencies. However, the analytical systems are designed for higher write throughput and sequential reads over latency or read access.

In recent times, there is an increasing demand for low latency processing of both reads and writes. Unlike traditional write heavy workloads, social media and mobile devices generate data at unprecedented rates and demand that updates be synchronously exposed. The emphasis of low latency workloads are shifting from reads to writes and is expected to reach a break even where both these become equally significant [1]. Moreover, the need for performant index probes in write optimized systems is increasing.

Google's BigTable, Yahoo!'s PNUTS are some of the widely-used services to support such low latency demands over large data. These systems use log structured indexes to provide superior write throughputs. LSM Trees despite being very effective at write throughput, could result in a notoriously poor read performance [1]. In this project, we look at alternatives for index structures and compare them in order to improve the read performance without hampering the write throughput.

## 1.1 Organization of the report

The report is organised as follows: In the next chapter, we discuss the performance of B$^+$-tree as an index structure. In chapters 3 and 4, we explain and analyse log structured merge trees and buffer trees respectively. In chapter 5, we briefly explain some other index structures and comment on their suitability to the problem. Chapter 6 provides an overview of our implementation of buffer trees that integrates with `HBase`, an open source implementation of Google's BigTable. Chapter 7 discusses the future experiments and improvements in the implementation of buffer trees and its integration with `HBase`.

# Chapter 2

# B$^+$-Trees

B$^+$-tree (or any of its variants such as B-tree) is the most widely used index structure due to its capability to maintain its efficiency despite random insertion or deletion of data [2]. It takes the form of a balanced tree in which every non-leaf node has between $\lceil n/2 \rceil$ and $n$ children. The performance characteristics of B$^+$-trees are quite familiar in the literature and hence we briefly discuss them here.

## 2.1 Performance

The cost of insertion and deletion operation on the B$^+$-tree is proportional to the height of the tree. In the worst case, this is $log_{\lceil n/2 \rceil}(N)$, where $n$ is the fanout and $N$ is the number of records. Given that these are worst case bounds and the memory sizes of today are of several gigabytes, these costs would be much lower because the non-leaf nodes can be expected to be in the database buffers in most cases. Typically, only one or two I/O operations are performed for a lookup or update. However, this will not be applicable in a setting that is as large as what we are interested in.

**Effective Depth** In case of large indexes, we are interested in the effective number of I/Os, taking into account the advantage of a page buffered system. [3] defines *effective depth* ($d_e$) of a B$^+$-tree to be the average number of pages not found in buffer during a random key value search down the directory level of the tree. To perform an update to a B$^+$-tree, we perform a key-value search to a leaf level page with $d_e$ I/Os, update it and write out the leaf page using 1 I/O. The effective depth of B$^+$-trees for different sizes of data at the leaf level is given in figure 2.1.

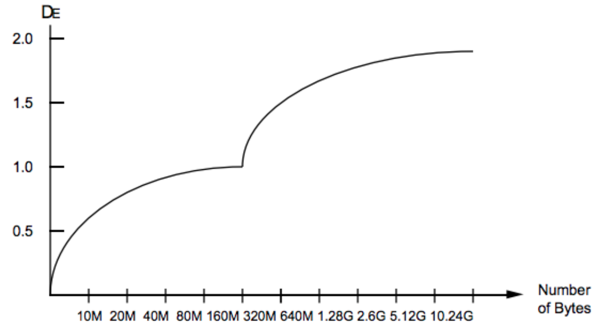

Figure 2.1: Effective depth of a B$^+$-tree vs the number of bytes at the leaf level [3]

# Chapter 3

# LSM Trees

## 3.1 Introduction

A *log structured merge tree* is essentially an extension of the idea of B$^+$-trees designed carefully in order to exploit the advantage of *batching*. An LSM tree [4] consists of multiple trees with a smaller component that resides in memory and many larger components that reside on the disk as shown in figure 3.1. It uses an algorithm that effectively defers and batches index changes, migrating the changes out to disk on a particularly efficient way similar to merge sort.

**Advantage of batching** The idea of batching stems from the fact that bulk loading of sorted records in a B$^+$-tree is faster than inserting one by one. This is because we can avoid the index probes to find the leaf page every time we want to insert a new record. This advantage occurs even in case of a succession of inserts of ever increasing key values, an *insert-on-the-right* situation. The algorithm described below will explain how LSM trees use this in an efficient manner to provide higher write throughput.

## 3.2 Algorithm

Whenever an insertion occurs, the index entry for the new record is inserted into the memory resident $C_0$ tree, after which it will migrate to the disk resident $C_1$ tree along with other index entries to take advantage of the batching. There is a certain amount of delay before the index entries get to the disk, which implies a need for recovery mechanisms to recover entries that don't get out to disk prior to crash. The operation of inserting an entry into the $C_0$ tree has no I/O cost

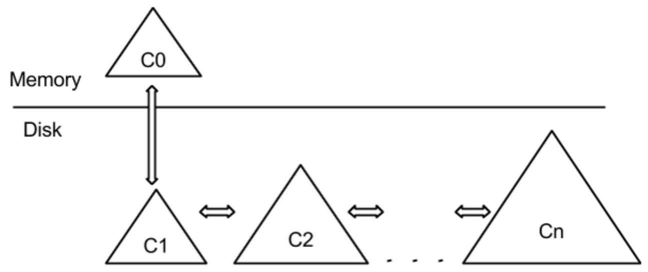

Figure 3.1: Structure of a multi component LSM Tree

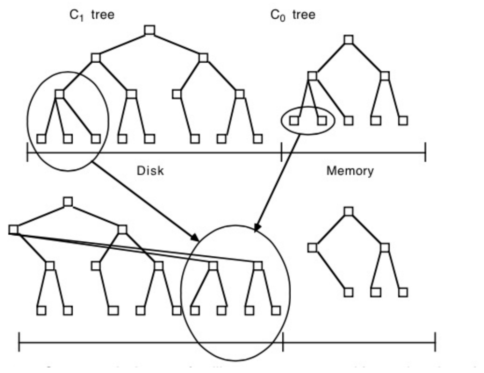

Figure 3.2: Conceptual picture of rolling merge, with results written to disk [4]

but the cost of memory capacity to house the $C_0$ tree imposes a limitation on its size.

**Rolling merge** The transfer of entries from $C_0$ tree to $C_1$ tree happens in a series of merge steps. A read of the multi-page block of the $C_1$ tree makes a range of entries of the leaf of $C_1$ tree buffer resident. Then in each merge step, we read a disk page sized leaf of the $C_1$ tree buffered in this block, merge these entries with entries from the leaf of the $C_0$ tree creating a new leaf of the $C_1$ tree. This step is shown in figure 3.2. This is called as *rolling merge* [4]. The newly written blocks are written to new disk positions, so that the old blocks will not be overwritten and will be available for recovery in case of a crash. These rolling merges happen between the on disk components too but always between $C_{n-1}$ and $C_n$ components.

**Why multiple disk components?** The advantage of the batch merge occurs when we merge the entries of $C_0$ and $C_1$ leaves. Every step leads to some I/Os and in each step, the advantage is proportional to the number of entries of $C_0$ we have transferred to $C_1$. This can be quantified by the following equation, where $M$ is the batch merge parameter [4] and $S_0$ and $S_1$ are sizes of the leaves of $C_0$ and $C_1$ respectively.

$$M \propto \frac{S_0}{S_0 + S_1}$$

We would like to keep this parameter $M$ as high as possible. But as the entries of $C_0$ are transferred to $C_1$, the value of $S_1$ needs to grow. To keep this greater than a bound, we can shift entries of $C_1$ tree to another tree on the disk. Extending this intuition, we can maintain this parameter by having multiple trees on disk with data migrating among these trees as and when they reach a certain size. This prevents the degradation in write performance due to large data. However, this situation presents a trade-off between read and write performances as explained in section 3.3

## 3.3 Performance

All of the I/O costs in an LSM tree is only due to movements during the rolling merge steps. Consider the notations mentioned in the table 3.1. In steady state, the rate at which entries move from component $C_{i-1}$ to $C_i$ in the rolling merge is the same as $R$. Then the merge from $C_{i-1}$ to $C_i$ entails a multi-page block read from $C_{i-1}$ at a rate of $R/S_p$ pages per second and multi-page reads at the rate of $r_i * R/S_p$ from $C_i$ component. Finally merge step writes out entries at the rate of $(r_i + 1) * R/S_p$ per second. Summing over all disk resident components the number of I/Os per

| | |
|---|---|
| $R$ | rate of insertion of data in bytes per second |
| $S_p$ | number of bytes per page |
| $S_i$ | size of leaf of $C_i$ tree in bytes |
| $r_i$ | $S_i/S_{i-1}$ |

Table 3.1: Notations

second, denoted by $H$ is given by

$$H = \frac{R}{S_p} * ((2r_1 + 2) + (2r_2 + 2) + ... + (2r_k + 2)$$

in a $k+1$-component LSM tree. We wish to minimize $H$ under the condition that the size of indexed data is constant, which is roughly the proportional to the size of the largest component.

$$\Pi_{i=1}^{k} r_i = \frac{S_k}{S_0} = N$$

This minimization problem is solved when each of $r_i$ is same and is equal to $C^{\frac{1}{k-1}}$. It immediately follows that the amortized cost of insertion is $\mathcal{O}(N^{\frac{1}{k-1}})$ [4]. Further more, in case of reads the LSM tree performs $k-1$ times as many seeks performed in a simple B$^+$-tree, as in the worst case it has to look up all the $k-1$ components on the disk. This value can be unacceptably high in some cases.

# Chapter 4

# Buffer Trees

## 4.1 Introduction

Like LSM Trees, buffer trees [5] also are an extension of the $B^+$-trees in combination with a buffering technique. The key idea in a buffer tree is to perform all the operations in a *lazy* manner using main memory sized buffers associated with internal nodes of the tree. Each internal node of the buffer tree is associated with a memory buffer of the size of the main memory as shown in the fighure 4.1.

## 4.2 Algorithm

Whenever an insertion happens, it is inserted into the buffer of the root node. The buffer tree waits until it collects enough entries in its buffer for a block and then these entries are pushed one level down to buffers on the next level. We call this a buffer emptying process (figure 4.2) When the buffer of the leaf node is emptied, the entries are inserted into the respective leaves using a merge step similar to external merge.

When the leaf overflows, a leaf split occurs similar to that of a $B^+$-tree. These leaf splits are then propagated upwards resulting in node splits and tree rebalancing operations. The deletes are also handled in a lazy manner. The delete entries are not deleted until they reach the leaves. In case of leaf underflow, leaf coalescing happens similar to $B^+$-tree.

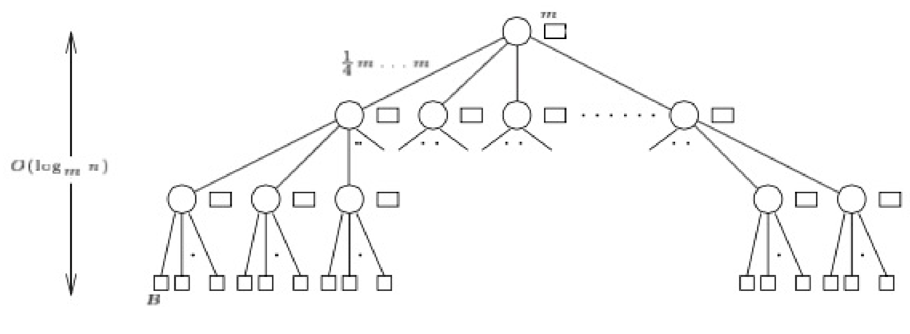

Figure 4.1: Structure of a buffer tree [5]

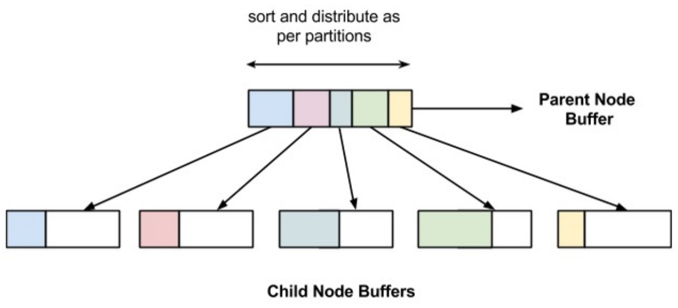

Figure 4.2: Buffer emptying process

| | |
|---|---|
| $N$ | total number of elements |
| $M$ | number of elements that can fit into main memory |
| $B$ | number of elements per block |

Table 4.1: Notations

## 4.3 Performance

Consider the notations in table 4.1. We assume the fanout ($m$) is proportional to size of memory buffers in terms of the number of elements. We use shorthand $n$ and $m$ for $N/B$ and $M/B$ respectively.

**Write performance** I/O costs for write result mainly from two steps : buffer emptying process and tree rebalancing operations to maintain the height. Each element is shifted from a parent node buffer to a child node buffer only once. But along with itself many other elements are shifted and they share this I/O cost. In total $\mathcal{O}(n * log_m(n))$ I/Os are shared by $N$ elements. The total number of rebalancing operations in a sequence of $N$ updates is bounded by $\mathcal{O}(n/m)$ and each rebalance operation takes $\mathcal{O}(m)$ I/Os and thus the total cost is bound by $\mathcal{O}(n)$ I/Os. So, the amortized cost of insertion is bound by $\mathcal{O}(log_m(n)/B)$ I/Os [5].

**Read Performance** In case of read performance, the I/O cost reduces to traversing the tree collecting all the parent buffer elements and the leaf elements. This is bound by the height of the tree and hence the worst case cost of reads is $\mathcal{O}(log_m(n))$ I/Os [5]. This cost is reduced by buffering of higher level internal nodes and their buffers in the cache. The read performance of buffer tree is better than the LSM trees without compromising on the write performance.

**Why is buffer tree better than LSM tree?** Intuitively, both LSM trees and buffer trees are based on the same idea that batching of operations results in sharing of costs. But the essential difference is in how these batched elements are managed. In case of LSM trees they are managed in multiple trees with overlapping ranges. Buffer trees on the other hand, use the structural organisation provided by the B$^+$-tree itself to organize this batched data. This results in better performance both in case of writes as well as reads.

# Chapter 5

# Other Index Structures

## 5.1 bLSM Trees

A bLSM (pronounced *blossom*) tree is an LSM tree with appropriate tuning and merge scheduling to supplant the poor performance of LSM trees in reads. In bLSM tree, the number of tree components are limited to three, one in memory and two on-disk components as shown in figure 5.1. Both these on-disk components are covered by bloom filters so as to prevent unnecessary lookups in these trees [1]. Following the analysis for write performance in section 3.3, the write performance is expected to be $\mathcal{O}(\sqrt{N/M})$ [1], where $N$ is the total size of the data and $M$ is the size of main memory. But, bLSM implements other optimization in the merge step that reduces the write amplification as explained below. The performance characteristics of bLSM trees are provided in table 5.1

**Spring and Gear Scheduler** We ideally would need to synchronize merge completions with the processes that fill each tree component in order to provide write availability [1]. The gear scheduler tracks the progress of the merges and ensures they complete at the same time, preventing them from stalling the application. To ensure this synchronisation, downstream merges are paused until the tree reaches a certain size. This clock-like synchronization is shown in figure 5.2(a). A spring and gear scheduler attempts to keep the size of $C_0$ between a low and high water mark. It pauses downstream merges if $C_0$ begins to empty and applies backpressure to the application as $C_0$ fills as shown in figure 5.2(b). The downstream merge processes behave as they did in the gear scheduler.

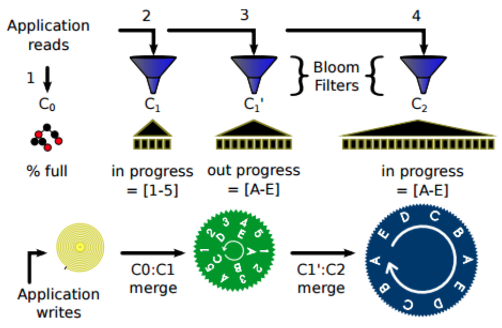

Figure 5.1: bLSM-Tree Architecture

| **Operation** | **#seeks** |
|---|---|
| Point lookup | 1 |
| Read modify write | 1 |
| Insert of overwrite | 0 |
| Short page scans ($\leq$ 1 page) | 2 |
| Long page scans (N pages) | 2 |

Table 5.1: Performance characteristics of a bLSM tree [1]

Figure 5.2: (a) Gear Scheduler (b) Spring and Gear Scheduler [1]

**Snow shoveling** The bLSM tree implements snow shoveling [1] in merging the trees and hence increases the effective size of the RAM. In a normal merge sort, we divide the data into memory sized runs, sort and merge them. We would ideally like to have smaller number of runs to ease the merge step. This can be done by exploiting the implicit order of values that occur in the input. Fill the memory with values and keep outputting the lowest value that comes after the previous value printed. Each of these would be a run and it is likely that the number of runs obtained thus is lesser than the in the previous case. This increases the effective RAM and hence reduces the write amplification.

## 5.2 Fractional Cascading

Fractional cascading [6] is a technique where leaves of a smaller tree contains pointers to leaves of a larger tree thereby avoiding the logarithmic search down the path of the larger tree. This is depicted in the figure 5.3. The problem with this scheme is that the cascade steps of the search examine pages that likely reside on disk. In effect, it eliminates a logarithmic in-memory overhead by increasing read amplification by a logarithmic factor [1].

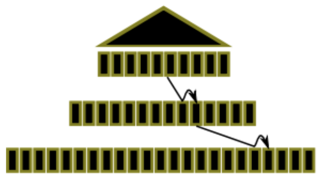

Figure 5.3: Fractional Cascading : leaves of the smaller tree has pointers to the leaves of the larger tree so that we can avoid traversing the larger tree [1]

# Chapter 6

# Implementation Details

## 6.1 Introduction

In this section, we explain the implementation of the buffer tree that is integrated with `HBase`, an open source implementation of Google BigTable by Apache Software Foundation. In section 6.2 we provide a brief overview of Google BigTable. This overview concentrates mainly on the components of BigTable that are of importance to understand the indexing stucture deployed in it. Contents in section 6.2 are produced direcly from [7].

## 6.2 Google BigTable

Google BigTable [7] is a sparse, distributed, persistent multidimensional sorted map. The indexing is done by a *row key, column key and a timestamp* and each value in the map is an uninterpreted array of bytes

$$(\text{row:string, column:string, time:int64}) \rightarrow \text{string}$$

### 6.2.1 Data Model

Each *row key* in a table is arbitrary string (of max 64KB). Every read or write of data under a single row key is atomic (regardless of the number of columns it has). Rows are lexicographically ordered. The row range for a table is dynamically partitioned and each row range is called a *tablet* (relates to distribution and load balancing). Column keys are grouped into sets called *column families* which form the basic unit of access control. A column key is specified as `family:qualifier`. Each cell in a bigtable can contain multiple versions of the same data; these version indexed by *timestamp*.

### 6.2.2 Storage Structures

BigTable uses Google File System (GFS) to store logs and data files - plays the typical role of a disk in traditional database systems. A Google SSTable is used to store the actual data as key-value pairs as a persistent, immutable ordered map (key and value are both arbitrary strings). An SSTable provides facilities to look up value associated with a specified key or to iterate over keys within a specified key range. An SSTable contains a sequence of blocks and a block index. Either the block index or the complete sequence might be loaded onto main memory for processing.

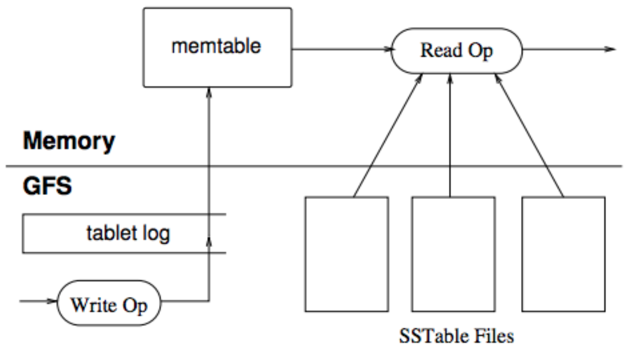

Figure 6.1: Tablet Representation [7]

### 6.2.3 Overall Architecture

There are three major components to a BigTable system: the client servers, master server and tablet servers: tablet servers are dynamically loaded. Master is responsible for allocation of tablets to tablet servers, detecting the addition and expiration of tablet servers, load balancing and garbage collection in GFS. Each tablet server manages a set of tablets (10-10000 tablets per server): handles read and write requests to the tablets that it has loaded and also splits tablets that have grown large. Client servers are allowed direct access to tablet servers and the system is out of typical disadvantages of a centralized architecture.

### 6.2.4 Tablet Serving: Instance of LSM Trees

The persistent state of a tablet is stored in the GFS. Updates are committed to a commit log that stores redo records. Of these updates recently committed ones are stored in memory in a sorted buffer called as *memtables*. Older updates are stored in a sequence of SSTables. When a write operation hits, it is first entered into the commit log. After the write has been performed its contents are inserted into the memtable. When a read operation hits, it is executed on a merged view of the sequence of SSTables and memtable (both are lexicographically sorted data structures and hence merged view is easy to form). This is depicted in the figure 6.1.

### 6.2.5 Compactions

As incoming write operations get executed the size of memtable increases and when it crosses a threshold, the memtable is frozen, a new memtable is created and frozen memtable is converted into a SSTable and written to GFS. This is called a **minor compaction**.

Every minor compaction leads to a new SSTable and if this increase is unchecked, read operations might have to merge an arbitrary number of such files periodically. So, periodically a **merging compaction** is performed, which reads a few SSTables, the memtable and creates a new SSTable. The input SSTables and memtable can be discarded. This is called a **major compaction**. This also discards deleted data and hence by periodic major compactions the system getz back unused resources.

### 6.2.6 `HBase` and Google BigTable

For the purposes of this project we use Apache `HBase` [8], which is an open source implementation of Google BigTable. One of the few main differences between BigTable and `HBase` is the nomenclature,

| `HBase` | **BigTable** |
|---|---|
| Region | Tablet |
| RegionServer | Tablet server |
| Flush | Minor compaction |
| Minor compaction | Merging compaction |
| Major compaction | Major compaction |
| Write-ahead log | Commit log |
| HDFS | GFS |
| Hadoop MapReduce | MapReduce |
| MemStore | memtable |
| HFile | SSTable |
| ZooKeeper | Chubby |

Table 6.1: Nomeclature in `HBase` and Google BigTable [8]

which is presented in the table 6.1.

## 6.3 BFile Design

The main component in the buffer tree is the file that stores the data, we call it `BFile`. This file contains the complete buffer tree. The blocks of the `BFile` are of three types: `BMeta`, `BNode`, `BLeaf` blocks. There is a single `BMeta` block at the start of the file, which contains meta data of the buffer tree such as number of nodes, block offset of root node, etc. The `BNode` block, as shown in figure 6.2, consists of meta information, node information and buffer data. Node information consists of block offsets of child nodes or leaves, and separators key values. The buffer data is an append-only unsorted array of key value pairs. `BLeaf` block, similar to the `BNode` block, consists of meta information and sorted array of key value pairs. It also contains a index on this array as the size of each key value is variable and such an index will allow faster search in the leaf. This is depicted in figure 6.3.

## 6.4 Code Design

In this section, we will explain the building blocks of the buffer tree implementation.

### 6.4.1 Frames

The buffer tree nodes are represented as blocks in the `BFile` and each type of the block has an associated class that will be used to read and write onto the file. We preferred to retain the data as an array of bytes, instead of converting them into member fields of the class, in order to minimize the memory usage. In essence, the `Frame` objects provide functions to access the data stored in the form of array of bytes. There are three kinds of frames : `BMetaFrame`, `BNodeFrame` and `BLeafFrame`. Frames may be opened in two modes: *read* and *write*. Frames are opened in read modes generally during scans and in write mode during buffer emptying and relocation operations.

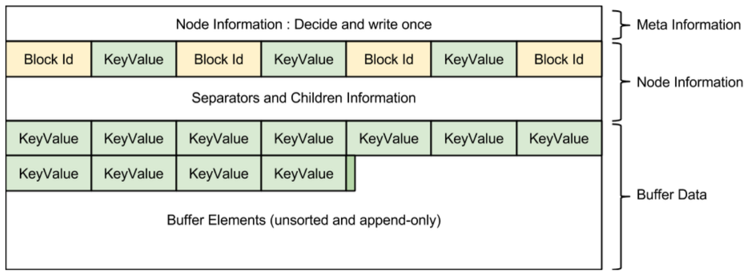
Node Information : Decide and write once
Meta Information
Block Id
KeyValue
Block Id
KeyValue
Block Id
KeyValue
Block Id
Node Information
Separators and Children Information
KeyValue
Buffer Data
Buffer Elements (unsorted and append-only)

Figure 6.2: Structure of `BNode` block

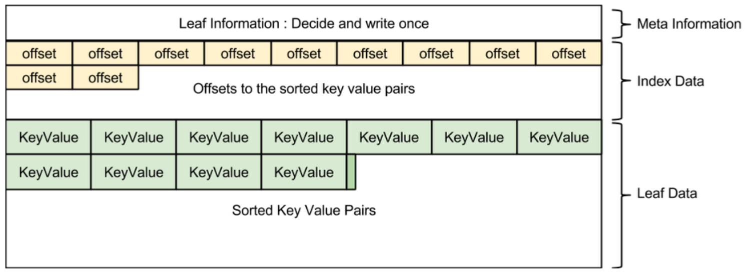
Leaf Information : Decide and write once
Meta Information
offset
Index Data
Offsets to the sorted key value pairs
KeyValue
Leaf Data
Sorted Key Value Pairs

Figure 6.3: Structure of `BLeaf` block

### 6.4.2 Scanners

In `HBase`, every read operation is modelled as a scan. So, we built a structure of scanners around the buffer tree to directly integrate with `HBase`. We designed three scanners : `LeafScanner`, `NodeBufferScanner` and `NodeScanner`.

Every `LeafScanner` is linked to a `BLeafFrame` object and acts as a scanner for the sorted array of key value pairs in the leaf block. Seeking to a particular key value in the leaf can be done faster by using the offset index stored in the leaf block. Every `NodeBufferScanner` is linked to a `BNodeFrame`. The scanner creates a copy of the key values in the node buffer, sorts them and acts as a scanner over this sorted list.

`NodeScanner` is an encapsulating class which provides a sorted view of the complete sub-tree rooted at the node including the node buffers of all its children. It maintains a pointer to one of its child nodes and keeps advancing towards higher key value ranged children. This scanner probes the node's own `NodeBufferScanner` and the `NodeScanner` or `LeafScanner` of its current child and chooses the appropriate key value.

### 6.4.3 Putting it all together

In `HBase`, every `HStore` corresponds to a buffer tree. Currently, every `HStore` is linked to a `Memstore` and a number of `StoreFile` [8]. `Memstore` is the memory resident component. We replaced this with `BufferTree` object, which consists of the root node's frame and is also linked to the `BFile` on the disk so that it can access other nodes and leaves. All inserts or updates are added into the buffer of the root node. These key values get written onto the other node buffers in the file before they eventually enter a leaf block. These transfers are actuated by emptying the root node buffer everytime it gets full.

An observation about adding elements into a buffer, buffer emptying and rebalancing operations, simplified the protocol of communication between the internal nodes and their children. If the buffer of a node at depth $h$ is emptied, then it can be triggered only when all its ancestors at heights $< h$ are emptying their respective buffers. Similarly, if there is a rebalancing operation that occurs at a node of height $h$, then it can happen only when all its children and grand children in that particular traversal, undergo rebalancing. So, we can simplify the protocol of communication between parent and child by awaiting a signal to rebalance from each of its child whenever its buffer is being emptied.

### 6.4.4 Read and Write Paths

Read path of a query along the buffer tree is very similar to that of the B$^+$-tree, in addition to which we need to read the node buffers too. Since, in our `BFile` design we store buffers in the same block as the node itself, there is no need for a separate seek. We obtain scanners for the particular instance using the root node, which will inturn read the node and leaf frames along the traversal path. In the worst case the number of seeks would be equal to the height of the tree. However, we expect many of the internal nodes at higher levels to be in the system cache and hence avoid some disk seeks.

Write path of a insert or update operation starts at the root node buffer. They are buffered in the root node and only when it is full, is it emptied onto its children node buffers. This emptying process will result in number of seeks equal to the number of its children. These emptying results in subsequent emptying of children nodes and eventually the inserted key value gets into a leaf block.

## 6.5 Conclusion

Currently, the status of the project is still *work in progress* and we are yet to evaluate the performance of our implementation of buffer trees using experiments, which would be our immediate agenda. In the next chapter, we present some experiments and improvements that are yet to completed.

# Chapter 7

# Future Work

## 7.1 Experiments

Intuitively buffer trees are expected to perform better than LSM trees based on the theoretical analysis presented in the previos sections. However, we need to establish this through experiments. We chose `HBase` as the testing platform to test this hypothesis and as a first step, we have completed the implementation of buffer tree and integrated it with `HBase`. Following are some of the scenarios in which we would like to evaluate the performance of buffer trees typically in:

- insert-heavy scenarios
- random lookup performance
- varying read:write workloads

There are a number of parameters that must be tuned to achieve the best performance of buffer trees such as fanout, size of buffers, etc. Experiments to fine tune such parameters are some other essentials. A higher fanout size leads to more I/O and less batch advantage during buffer emptying. But, a higher fanout reduces the height of the buffer tree and hence reduce the number of I/Os during reads. This presents a clear trade-off situation. Does varying the size of buffers according to height of the node have an impact on the performance? Currently the tree allots the earliest non-allocated block for a new leaf or node. Is there a better heuristic that can be used to take advantage of multipage blocks similar to that in SB-trees?

## 7.2 Improvements

**Region Splits** Region splits in `HBase` happen when the size of the component HStores exceed a certain limit. The region split starts by linking parts of the same store file to different stores and eventually they get transferred to a different `StoreFile` in the new `Store` [8]. Partitioning the elements in the `BFile` is clearly very easy but the subsequent migration of data to a new buffer tree is very expensive as it involves reading from a lot of random blocks. We need to devise a scheme for block allocation wisely.

**Concurrency** Concurrency between buffer emptying and reads is the most critical among others. We need to work out mechanisms so that these do not block each other without compromising on the write throughput and read performance.

**Recovery issues** In the current `HBase` architecture, the `WAL` (Write Ahead Log) acts as the backbone of recovery for data that is in the `Memstore`, which has not migrated to the disk [8]. In case of buffer trees a similar mechanism has to be worked out for the root node buffer elements. To recover from crashes during buffer emptying processes, we can retain the elements in the parent node's buffer until we check if the transfer is successful. We can also implement error detection schemes for every element to detect corruption of data, which is highly likely in a system that performs in-place updates such as ours.

**Write-once read-many architectures** Distributed file systems such as GFS, HDFS are generally designed as write-once read-many architectures [8]. This imposes a great constraint on the implementation of buffer trees as the operations in buffer trees have update-in-place requirements. Modern alternatives such as copy-on-write methods could be considered but these are simply underperforming substitutes that remove the performance advantages of the buffer trees.

# Bibliography

[1] Russell Sears and Raghu Ramakrishnan. blsm: A general purpose log structured merge tree. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD '12, pages 217–228, New York, NY, USA, 2012. ACM.

[2] Abraham Silberschatz, Henry Korth, and S. Sudarshan. *Database Systems Concepts*. McGraw-Hill, Inc., New York, NY, USA, 5 edition, 2006.

[3] Patrick E. O'Neil. The sb-tree: An index-sequential structure for high-performance sequential access. *Acta Inf.*, 29(3):241–265, June 1992.

[4] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. The log-structured merge-tree (lsm-tree). *Acta Inf.*, 33(4):351–385, June 1996.

[5] Lars Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[6] Bernard Chazelle and Leonidas J. Guibas. Fractional cascading: I. a data structuring technique. *Algorithmica*, 1:133–162, 1986.

[7] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E. Gruber. Bigtable: A distributed storage system for structured data. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation - Volume 7*, OSDI '06, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[8] Lars George. *HBase: The Definitive Guide*. O'Reilly Media, 1 edition, 2011.