CuttleTree: Adaptive Tuning for Optimized Log-Structured Merge Trees

Nicholas Joseph Ruta, Jr.

A Thesis in the Field of Software Engineering

for the Degree of Master of Liberal Arts in Extension Studies

Harvard University

March 2018

Abstract

The standard implementation of a Log-Structured Merge-tree (LSM-tree) (O'Neil, 1996) is described as a disk-based data structure designed to provide low-cost indexing for a file experiencing a high rate of record inserts and deletes over an extended period. The standard LSM-tree design provides better I/O access patterns for write intensive workloads but the tradeoff is a decrease in performance of read queries. The typical implementation of the LSM-tree is initialized with its attributes set and it remains constant throughout the lifetime of the program. These attributes include the amount of data contained within each portion of the LSM-tree, the way in which data is moved from one part of the data structure to another and the frequency of data transmission.

The purpose of this thesis is to design an adaptive LSM-tree that captures workload patterns by collecting statistics during its runtime and uses different versions of tunable parameters in order to optimize the performance. These tunable parameters adjust the behavior of the LSM-tree and allow it to store and read data using different techniques. This way, instead of having a single and fixed design as in current state-of-the-art implementations, our new adaptive LSM-tree can transition between alternative designs and accommodate varying workloads. LSM-trees are prevalent in many modern systems and so this work finds applications in numerous systems categories from classic database systems to key-value stores.

The project was implemented in C++ using object oriented principles in order to create a modular design that makes testing and extending more productive and efficient. This includes an implementation of a B-tree as each level's data structure. Results from extensive testing are provided to show an increase in performance for workloads that include a change from read to write-heavy, write to read-heavy, or when a combination of several of these changes occur during runtime.

Acknowledgments

I would like to express my special appreciation and thanks to my thesis director Professor Dr. Stratos Idreos for his encouragement, enthusiasm, immense knowledge and guidance while I created this thesis project. The decision to work on this project was based on the experience that I had in Dr. Idreos's class *Big Data Systems CS 265* during the Spring 2016 term. The class project was to implement an LSM-tree data structure. Professor Idreos had us read and discuss many research papers on the subject including material on adaptive approaches that are a focal point of the research conducted in the Harvard DAS lab. It is through this learning process that I was exposed to (and quite frankly, became passionate about) the application of new and creative processes to optimize the performance of data systems. Dr. Idreos is an inspirational role model as a scholar and educator and I am truly grateful to have been his student.

I want to thank my research advisor, Dr. Jeff Parker, for helping me to refine my ideas during the early stages of development and providing valuable resources to help me bring this thesis project to fruition. I would also like to thank Dr. Niv Dayan for kindly making time for me and acting as a valuable mentor. He provided support, feedback and helped me determine the best direction for my research.

Table of Contents

Table of Contents vi

List of Figures ix

Chapter 1 Introduction 1

1.1 The Log-structured Merge-tree 1

1.2 The Problem: Inadequate Adaptive Behavior During Runtime 3

1.3 The Solution: CuttleTree Statistics Collection & Adaptive Tuning 4

Contributions 4

Chapter 2 Background 6

2.1 Rolling Merge Strategies of Log-structured Merge-trees 6

2.2 Related Work 7

Chapter 3 The CuttleTree Log-structured Merge-tree 10

3.1 Adaptive Nature of CuttleTree 11

Tunable Parameters as Design Knobs 11

Implementation 15

Bloom Filter 17

Rolling Merge Algorithm 19

C_0 Memory Level Initial Fill-Up 19

CuttleTree Benchmark System 20

3.2 CuttleTree Statistics Collection Feature 21
Implementation 22
3.3 Cost & Complexity Analysis 23
Rolling Merge Cost 23
Cost to Merge Levels 24
Block Size Latency Cost 24
Chapter 4 Experimental Analysis 26
4.1 Hardware, Dataset and Workload 26
4.2 CuttleTree Static Experiments 27
B-tree vs std::vector 27
Finding the Optimal B-tree Node Size 27
Read Optimization Using Range Detection & Tombstone Delete 29
Parallelization 32
HDD vs SSD 34
Inserts Throughput as Levels Increase 39
Read to Write Ratio 40
4.3 CuttleTree Adaptive Experiments 41
Merging Levels to Reduce Read Amplification 42
Decreased Rolling Merge Times 43
Adaptive Behavior & The Multi-Phase Workload 44
4.4 CuttleTree Statistics Collection & LevelDB 45
Using Block Statistics to Improve Random Reads Performance 45
4.5 Strengths and Weaknesses 46

Chapter 5 Summary and Conclusions 48

Future Work 48

References 50

Appendix 1 CuttleTree Sample Output 51

CuttleTree Output 51

CuttleTree Statistics & LevelDB Output 55

List of Figures

Figure 1 LSM-tree design (O'Neil, 1996) showing a two-level implementation. 2

Figure 2 LSM-tree (O'Neil, 1996) rolling merge process transferring data from memory to disk. 3

Figure 3 The cuttlefish using its "adaptive camouflage" optimization. 10

Figure 4 Table showing the time complexities for inserts and reads of an unsorted array and B-tree. 13

Figure 5 Excerpt from the documentation of CuttleTree's LSM-tree class showing the constructor's tunable parameters. 15

Figure 6 Example of possible parameter settings for read and write optimized workloads. 17

Figure 7 An example of a Bloom filter showing the set {x,y,z}. The element w is not in the set {x,y,z} because it hashes to one bit-array position containing a 0. 19

Figure 8 Using statistics up until the first rolling merge to set the disk level configuration. 20

Figure 9 Showing the initialization parameters for the CuttleTree benchmark system. 21

Figure 10 An example Tree-like data structure showing nodes of various sizes. The 2-4-6 node on the left shows a size of 3. 28

Figure 11 Displaying 20 as the optimal node size. 28

Figure 12 The throughput for 20k reads using read optimization and not using it. 30

Figure 13 Displays TOMBSTONE vs BLIND deletes and shows TOMBSTONE outperforms when reads are frequently repeated. 32
Figure 14 Showing an increase in throughput as additional cores are added to the multi-thread rolling merge. 33
Figure 15 Showing an increase in throughput as additional cores are added to worker queue for reads until 5 threads. 34
Figure 16 Comparing HDD vs SSD for writes. 35
Figure 17 Comparing HDD vs SSD for reads. 36
Figure 18 Showing higher number of inserts for SSD drive. 37
Figure 19 Showing higher number of reads for SSD drive. 38
Figure 20 Showing updates for the SSD drive. 39
Figure 21 Showing throughput with total levels increase. 40
Figure 22 Showing Read vs Write ratio. 41
Figure 23 Showing Throughput for a read-heavy workload as a result of merging levels. 42
Figure 24 Showing adaptive behavior for decreased rolling merge times. 43
Figure 25 Showing adaptive behavior for a dynamic workload. 45
Figure 26 Showing throughput using variable LevelDB block sizes. 46

Chapter 1 Introduction

"…relational databases are the foundation of western civilization…If the database can be made to run faster on a small system, then you don't have to buy as much hardware to get the job done. In fact, performance is a critical measure of what you're doing. It's like house cleaning: you just have to do it, because if you don't, you can make silly things happen." – Dr. Bruce Lindsay, IBM

1.1 The Log-structured Merge-tree

The general concept of the traditional design of a Log-Structured Merge-tree is to provide a data structure that can do writes in batches in order to achieve a speed-up due to the cost of expensive I/O transactions. This design trade-off can create a system that minimizes the cost associated with storing data on disk by using a standard B-tree implementation. This is because a B-tree requires two I/O transactions per random key blind-write (a write that does not involve a read before the data is written). This is due to the fact that a B-tree is first loaded from disk, modified, and then finally written back to disk. In order to reduce this behavior to minimize disk activity, an LSM-tree writes periodically in batches in a method referred to as a rolling merge. This rolling merge process occurs over two or more levels designed to contain the data in its entirety. The trade-off with this design decision is that reads will have to consult multiple levels in order to attempt to find data values. While the inherent behavior of the LSM-tree is set as

a write efficient data structure, certain design decisions can be modified in order to reduce the impact this design has on read queries.

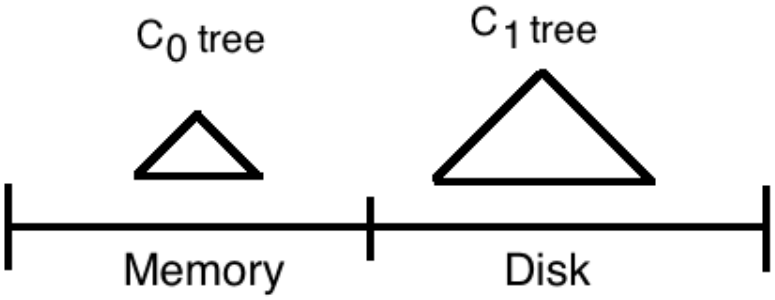

Figure 1 LSM-tree design (O'Neil, 1996) showing a two-level implementation.

An early paper on the LSM-tree design (O'Neil, 1996) describes the basic structure as being composed of two levels. The first level, called C_0, resides completely in memory while the second level, C_1, is on disk. The LSM-tree design provided by this project uses tunable parameters to experiment with the number of disk levels, size of each disk level, size ratio between levels, size of the C_0 memory level and the percentage of C_0 that is to be copied to the disk levels when a rolling merge occurs. The disk levels contain data that resides in B-tree implementations. In order to optimize the binary file creation and random access patterns associated with the B-tree, the C++ programming language was used and technical references were consulted throughout the design and implementation phase. With the initial implementation complete, a parallelized version was created. The rolling merge process was implemented in a separate background thread and multiple threads were tested to improve read performance. The parallelized version was used to create an adaptive data structure that relies on occasional system testing in order to analyze current workload conditions and optimize performance.

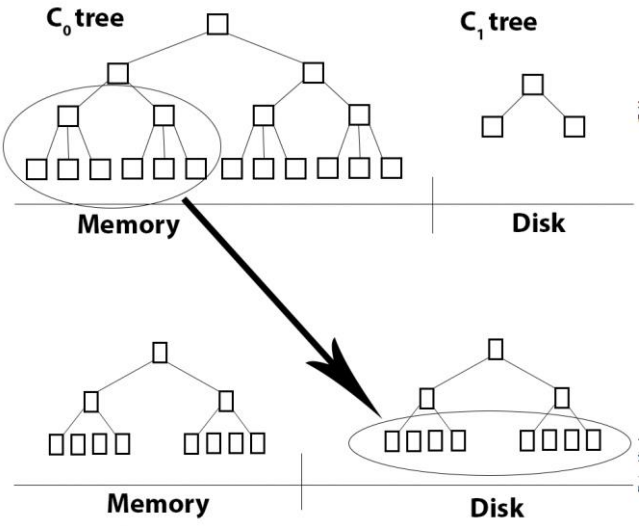

Figure 2 LSM-tree (O'Neil, 1996) rolling merge process transferring data from memory to disk.

1.2 The Problem: Inadequate Adaptive Behavior During Runtime

Modern LSM-trees come with a large array of tunable parameters that are typically configured during the setup stage. These include, but are not limited to, the amount of data that can be stored in a memory-resident buffer before a compaction is executed, the maximum number of concurrent compactions and the size ratio between the levels of the LSM-tree. This requires that we know what to expect during the lifetime of an application's execution at the beginning. It also means that the system may not optimally handle dynamic environments, even if a significant amount of time is spent monitoring and continuously keeping it tuned. This is because optimal configuration can change during different phases of the workload and at an instance. Improvements made on these structures could have a significant impact on the performance of database systems that are used by the vast majority of technology companies today.

1.3 The Solution: CuttleTree Statistics Collection & Adaptive Tuning

We introduce CuttleTree, an LSM-tree based key-value store that uses an array of tunable parameters and statistics collection to allow for optimized decision making at runtime. CuttleTree shows the potential that a self-designing system can have towards automatically adapting to its environment in order to handle the ever-changing data-driven world of today.

Contributions

In summary, the contributions provided by this project are as follows:

- In Chapter 3, we introduce CuttleTree, an LSM-tree based key-value store that enables adaptive responses to dynamic workload patterns by utilizing twelve tunable parameters. It includes a statistics collection mechanism that allows for decision making during runtime by detecting changes such as read-heavy and insert-heavy workload patterns.
- In Chapter 3, we will describe the CuttleTree benchmark system's architecture. This detailed implementation was used to model various workloads and test CuttleTree's ability to adapt. It allowed for more fine-grained control over the conditions of our experiments than any state-of- the-art data system, such as LevelDB, that we observed.
- In Chapter 3, we introduce the CuttleTree statistics collection platform. It is a key facet of the project which allows for dynamic decision making to occur. It maintains more statistical information about the runtime of the data system than any state-of-the-art project we have observed. We describe how it was implemented on top of

LevelDB to reveal more detailed statistics than it otherwise would provide. In Chapter 4, we demonstrate the usefulness of these statistics and compare the results of using them to make dynamic decisions with the results of a static version of the system while keeping in mind the cost associated with maintaining these statistics. These dynamic decisions are based on a collection of "tuning knobs" that include (1) setting an upper bound constraint on the number of levels, (2) adjusting the size ratio between levels, (3) enabling bloom filters, fence pointers, and tombstone deletes for read optimization, (4) altering the size of memory buffers and the percentage of a buffer to be merged to a disk-resident level during a compaction, and (5) determining the frequency for the usage of the statistics collection mechanism.

Chapter 2 Background

2.1 Rolling Merge Strategies of Log-structured Merge-trees

Most LSM-tree designs belong to one of two categories based on their choice of rolling merge algorithm. An early paper on the LSM-tree design (O'Neil, 1996) describes the process that occurs when data is moved from memory to disk in order to address the size constraints of a memory resident data structure as it increases its total data allocation. When the memory resident portion of the LSM-tree reaches a certain defined capacity, an ongoing rolling merge process serves to delete some contiguous segment of entries from the memory resident portion and copy them to disk. This is referred to as a "leveling" merge policy. The second type of merge policy is called "tiered." This policy makes updates less costly at the expense of read performance. It accomplishes this by splitting levels into multiple "runs", and therefore avoids having to merge levels each time the C_0 memory level is at capacity. Since levels are made up of segments, they are no longer sorted. This results in more expensive read operations.

Currently, there are several popular implementations that use different rolling merge strategies. The LevelDB/RocksDB system (Menon et al., 2014) relies on a B-tree based intermediate layer called the file-system. It divides layers up into non-overlapping segments of around 2mb in size. Anytime a segment-set is modified, the entire file-system file must be rewritten. It performs merges using one of these segments and an adjacent segment. Since the data does not necessarily reside in contiguous locations on

disk, writing the data is not always efficient. Shuttle-Trees and Fractal-Trees (Kuszmaul, 2014) take a different approach. They add write-deferral to their B-tree implementations. They accomplish this by using smaller block sizes than LevelDB's segment sizes and writing portions of the b- tree as "shuttle buffers" as opposed to making an immediate write each time an update occurs.

Our implementation will use the "leveling" rolling merge policy. In Chapter 4, we include several experiments that demonstrate the nature of our rolling merge process. We'll take into consideration the number of CPU cores available and the hardware profile of the system in addition to the total time needed to complete a rolling merge.

2.2 Related Work

WiredTiger is a storage engine that was acquired by MongoDB. It is a high performance, scalable, production quality platform for data management. MongoDB bought the technology in order to address its need to do a better job on high write-volume workloads. It uses LSM-trees for sustained throughput under random insert workloads. The only configurable parameter of WiredTiger's LSM-tree implementation is the size of the in- memory level. This thesis project will provide multiple parameters related to the number of levels and their sizes in order to improve the average read time.

Read amplification is the amount of work done per logical read operation. The work done can include key comparisons or the cost of decompressing data read from storage. The bLSM general purpose LSM-tree (Sears et al., 2012) design proposes a technique to reduce read amplification. It uses Bloom filters to improve index performance. The paper describes their choice to scan the LSM-tree levels in order until a

matching record is found. They terminate the search early, if a record is found, instead of continuing to search levels for duplicates. They are able to do this since their focus is on the most recent version of the element. While this does address the need for improved read amplification, the design does not incorporate tunable parameters in order to further adjust the performance. This thesis project experiments with adjusting the number of levels and level sizes in addition to the use of a Bloom filter to improve read amplification.

The Apache product HBase is an open-source, distributed storage manager well suited for real-time read/write access. It can provide an improvement on the MongoDB and bLSM designs in terms of read amplification by offering additional tunable parameters including the size of the C_0 memory level and the number of disk files at each level. HBase works well with random read and write access patterns, especially for those organizations already heavily invested in the Apache Hadoop HDFS file system for storing large volumes of data. This thesis project will demonstrate a more fine-grained approach by including numerous tunable parameters and allowing the data structure to adjust the settings during runtime.

LevelDB is a key-value storage library written at Google that provides an ordered mapping from string keys to string values. It incorporates some adaptability during runtime. The testing shared by Google suggests that read performance was not a priority. This is made apparent by the benchmarks provided for read performance being "quite small." It was intended to characterize the performance of LevelDB when the working set fits in memory. The maximum number of levels are not configurable. The Google team states that write-heavy workloads should have more levels and read-heavy ranges should

have fewer levels. They address this by automatically triggering rolling merges for ranges that have seen a lot of lookups recently. They do this in order to avoid having fewer and larger data compactions that are typically far more stressful on the system. This thesis project experiments with a variable total number of levels in order to address the optimization needed for both read and write amplification.

It will be demonstrated that the ability to change parameter settings during the lifetime of the application will have a positive effect on the performance. For example, increasing the size of the memory resident portion of the data structure, decreasing the number of levels and increasing the size of each level will decrease the average read time. This is because more data residing in memory and fewer levels to contain the data will result in fewer disk reads. Also, as the amount of levels increases, the throughput of inserts per second will decrease, since multiple levels involve multiple smaller data structures that need to be maintained. This will cause a more frequent transmission of data from one level to another.

Chapter 3 The CuttleTree Log-structured Merge-tree

The name CuttleTree was inspired by one of the best adaptive behavior specialists in nature: the cuttlefish. In order to achieve the most effective results, the cuttlefish automatically adapts to its environment, blending in with its surroundings in order to go unnoticed and survive. It accomplishes this through the usage of "tunable parameters" such as the level of pigmentation associated with each of its tiny organs called chromatophores. Similar to the cuttlefish, our Log-structured Merge-tree provides a unique ability to adapt to the situation at hand. Our LSM-tree based key-value store uses an array of tunable parameters and statistics collection to allow for optimized decision making at runtime. Going forward, we will refer to our design simply as CuttleTree.

Figure 3 The cuttlefish using its "adaptive camouflage" optimization.

3.1 Adaptive Nature of CuttleTree

Tunable Parameters as Design Knobs

The foundation of CuttleTree is a standard and well-documented LSM-tree (O'Neil, 1996). What makes CuttleTree unique, and grants it the ability to adapt during runtime, is its tunable parameters. CuttleTree has 12 tunable parameters that collectively establish 8 design knobs. The details of each design knob are as follows:

1. The first tunable parameter is called **is*_read_Optimized***. It is a *Boolean* that determines if the implemented version of a bloom filter, described later in this chapter, and a tombstone delete procedure, detailed in Chapter 4, should be used for read queries in order to increase performance. This allows performance testing to be conducted at different points in the runtime of the application in order to determine if these read optimization features create efficiency. If a large amount of read queries are being executed, they can be enabled by setting the *Boolean* value to *true*. This will decrease the amount of I/O interactions but it will require more memory to maintain the bloom filter and tombstone log file.
2. The second tunable parameter is ***m_node_size.*** It sets the number of values present in each node of the B-tree associated with each level of the CuttleTree design. It is an *integer* that represents the number of values to be stored in each node. The general theory is that node size should be as small as possible, not to bring too much data from disk. In Chapter 4, we will demonstrate a method to determine the optimal node size and set the ***m_node_size*** parameter based on the hardware that CuttleTree is running on. The experiment will solidify our belief that optimal node size is governed primarily by access latency, transfer bandwidth and the record size.

3. The data structure that will be used for the memory-resident C_0 level is determined by the third parameter, ***c0DataStructure***, which has an option for a B-tree and another for an array set as an *integer* value of 1 or 2 respectively. The need for decision-making related to the memory-resident data structure stems from the inherent trade-off between the speed of reads and inserts of these two data structures versus the workload and amount of memory allocated to the data structure. The standard LSM-tree is implemented with the C_0 memory-resident level as an unsorted array to allow for inserts to be appended at the end with one simple operation. When C_0 becomes full, it is first sorted and then written to the disk-resident levels. An array is a good option when C_0's main purpose is random access or iteration of the data. The B-tree, on the other hand, keeps keys in sorted order for sequential traversing. The B-tree uses a recursive algorithm that involves searching the tree to find the point, referred to as a leaf node, where the new element should be added. Since the B-tree keeps the index balanced, it must determine if there is enough space in that leaf node. This results in either no further action or the leaf node must be split into two new nodes and potentially merged with other nodes to maintain its balanced nature. This procedure leads to its faster search time complexity but a slower insert time complexity relative to an unsorted array. In terms of big O notation, both have a space complexity of O(n). The worst-case time complexity for a search is O(n) for an unsorted array and O(log n) for the B-tree. As described previously, the unsorted array appends new data to the end in O(1) time complexity while the B-tree takes longer for sufficiently large enough data at O(log n).

Time Complexity of Inserts & Reads in Big O Notation		
	Inserts	Reads
Unsorted Array	O(1)	O(n)
B-tree	O(log n)	O(log n)

Figure 4 Table showing the time complexities for inserts and reads of an unsorted array and B-tree.

4. The fourth parameter, ***num_levels***, is an integer value that sets the upper bound on the total number of levels for the CuttleTree. Subsequent parameters will be described to determine the conditions required to elicit the creation of a new level but this design knob enabled us to have a stopping point. This allowed us to debug during the development phase by creating a forced stopping point to level creation in order to assess the correctness of our data compactions. It also helped complete planned experiments. For example, what performance implications could we address by observing an LSM-tree that has a minimal size difference between levels but is limited to only 5 levels. In this situation, our implementation would have a 5^{th} level that would continuously expand as new inserts were processed.
5. The CuttleTree is a "leveling" LSM-tree design that is a made up of a series of levels each with its own determined maximum size. The file size for the first disk-resident level, C_1, is set using a *long* value fifth parameter named ***firstLevelFileSize***. The ***sizeBetweenLevels*** (also known as level fan-out) sixth parameter is a *float* value that determines the capacities of adjacent levels by using the product of itself and ***firstLevelFileSize*** to set the file sizes of each subsequent level after C_1. Tuning this relationship between levels allows us to manage the balance between read and write performance of the design. For example, lookup cost depends on the number of levels. We reduce the number of levels by increasing the size ratio between levels. We could achieve this outcome in a series of experiments with CuttleTree by setting

the ***firstLevelFileSize*** to 200,000 bytes and reducing the ***sizeBetweenLevels*** from 4 to 2.

6. LSM-trees manage the compaction of their data using the rolling merge process described in Chapter 2. CuttleTree provides a design knob to tune the amount of data that is transferred between the memory and disk levels. This design knob was used during experiments to validate the "Cost to Merge Levels" portion of the section "Cost & Complexity Analysis" found later in this chapter. It involved increasing or reducing the time required to execute a rolling merge. First, we have to define the size of CuttleTree's C_0 memory-resident level. We do this as a percentage of the ***firstLevelFileSize*** described in the previous design knob's section with the seventh tunable *float* parameter: ***c0_percentage_of_c1***. When C_0 is smaller, less data is moved during a rolling merge. When the eighth tunable *Boolean* parameter, ***copyallFromC0***, is set to *false*, the *float* type ninth parameter, ***c0percentage_to_copy***, is used to determine the percentage of the total memory-resident values that should be moved to the first disk level during a rolling merge. This allows us to throttle the amount data transferred per rolling merge in order to reduce the total time of its execution.
7. The single threaded CuttleTree can be made to a concurrent program when the tenth *Boolean* parameter, ***threadedRollingMerge***, is set to *true*. This makes rolling merge processes execute in a separate thread allowing more inserts to occur simultaneously. It also enables a worker queue and thread pool implementation that allow for concurrent reads and deletes to enable the program to scale as the number of threads/cores grows. This parameter was useful during experimentation to understand

the impact of parallelization on the system during workloads that involved concurrent reads, writes and rolling merges.

8. Later in this chapter, we will describe the statistics collection techniques that give CuttleTree its unique ability to adapt to various workloads for optimal performance. In order to measure CuttleTree's performance, we had to create two parameters that disabled aspects of its adaptability in order to establish a baseline. The eleventh *Boolean*, ***use_initial_c0_stats_tuning***, allowed us to disable adaptive behavior that occurred during the initial phase of runtime by setting its value to *false*. The twelfth *Boolean*, ***use_print_current_stats_and_adapt***, determined whether CuttleTree would be an adaptive design by capturing the workload it experiences and reacting by tuning the design knobs outlined in this section. When set to true, CuttleTree collects statistics related to each operation and incrementally assesses the workload and current settings of the tunable parameters to decide if changes should be made.

```
Constructor to initialize CuttleTree using tunable parameters.

   @param is_read_optimized boolean value to determine if read optimization is on or off
   @param m_node_size represents the node size. There are M fields in each node of each B-tree
   @param c0_data_structure what data structure should be used at c0 (1 for BTree or 2 for std::Vector)
   @param num_levels the max. number of levels for the LSM-tree
   @param first_level_file_size what is the maximum size of the c1 level?
   @param size_between_levels what is the size increase for each additional level?
   @param copy_all_from_c0 should we copy all of c0 to c1 on a rolling merge?
   @param c0_percentage_to_copy what percentage of c0 do you copy to c1 on a rolling merge?
   @param c0_percentage_of_c1 what percentage of c1 can c0 be before a rolling merge occurs?
   @param threaded_rolling_merge should the threaded rolling merge occur in its own thread?
   @param use_initial_c0_stats_tuning set the disk levels after adaptive tuning of first fill-up of c0?
   @param use_print_current_stats_and_adapt make adaptive tuning decisions based on current statistics?
   @param cuttle_tree_statistics_ptr pointer for CuttleTreeStatistics class used to collect statistics
```

Figure 5 Excerpt from the documentation of CuttleTree's LSM-tree class showing the constructor's tunable parameters.

Implementation

CuttleTree provides insert, read, delete, and update functions. It was written in C++ and uses the object-oriented programming (OOP) paradigm to include three core

classes. The *LSMTree* class is the primary class that uses data structures that are both in memory, found in the *LsmLevelMemory* class, and on disk, found in the *LsmLevelDisk* class. The *LSMTree* class is where the tunable parameters, described in the previous section, were created.

Once the implementation was complete, attention was focused on the adaptive nature of the application. Testing was conducted to determine several versions of the application with different settings for the tunable parameters. These versions were applied to different workload profiles in order to optimize the performance of the data structure. The system was modified to automatically adapt to different workloads as the program is running based on occasional tests performed by the system on recent activity. This was accomplished by collecting statistical information about recent data workload and running tests to report the current performance of the system. This statistics collection mechanism is described later in this chapter under "CuttleTree Statistics Collection Feature." Testing the current performance of the system includes operations such as speed tests for a set amount of insert or read queries and finding the optimal node size of each level's B-tree to support top performance. For example, when there are more reads occurring, the system can adapt by changing the tunable parameters in a way that optimizes the performance. One possible adjustment could be to make the LSM-tree contain fewer levels in order to reduce I/O operations when many reads are attempted in a short period of time.

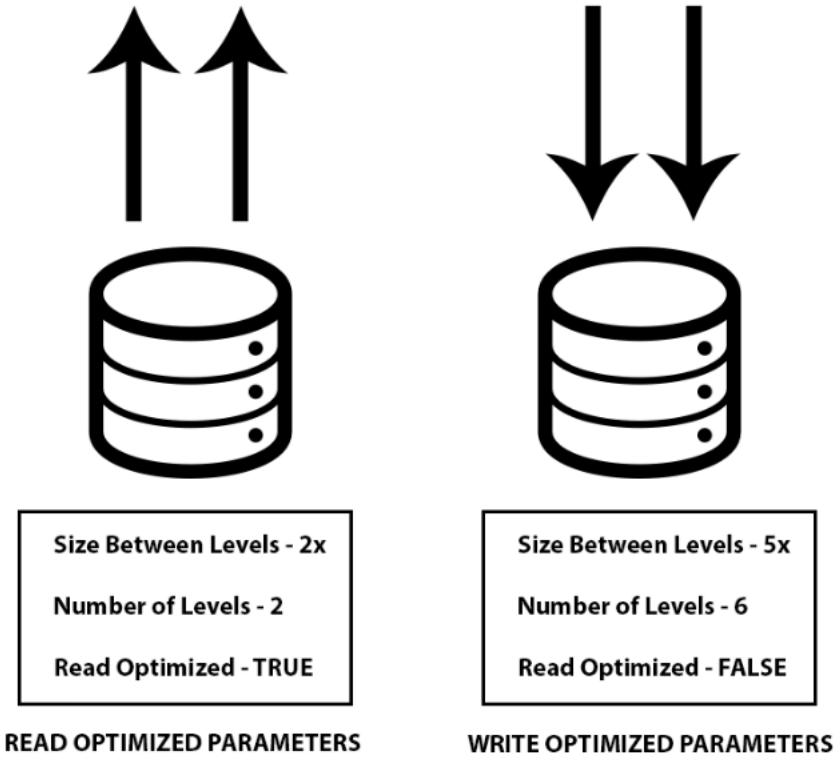

Figure 6 Example of possible parameter settings for read and write optimized workloads.

Bloom Filter

The standard implementation of an LSM-tree found in use today is inherently designed to handle high write throughput. This is the result of using data structures, such as a B-tree, to contain the disk resident data and writing to these structures in a deferred batch process as opposed to taking immediate action. The multiple level design, with each having at least one data structure to contain a portion of the total data, of an LSM-tree decreases the efficiency of read queries. If a read is requested, the LSM-tree will need to probe multiple levels and one read will be required per level. This was described in a research paper assessing the efficiency of LSM-trees (Kuszmaul, 2014). The researchers addressed the issue by using a Bloom filter.

The Bloom filter is a probabilistic data structure that is designed to rapidly determine whether an element is present in a data set. The term probabilistic means that the Bloom filter cannot tell us if the element is definitely in the database, but if the item is not in the database, the Bloom filter might be able to tell us that it is missing. An empty bloom filter is a bit array with all bits initially set to 0. Hash functions are used to map each element of the set to one of the array positions. To add an element to the Bloom filter array, feed it to each of the hash functions to get an assigned array position. The bits at the assigned array positions provided by each hash function are then set to 1. To query for a particular element, feed it to the hash functions to retrieve array positions. If any of the bits at these positions are set to 0, the element is definitely not in the set. If it were, the bits would have been set to 1 at all positions. If all values are set to 1, then either the element is in the set or the bits have by chance been set to 1 during the insertion of another element. This is why the Bloom filter is not able to say with certainty that the item is present in the set. It will be used in an attempt to provide a significant reduction in the time required to perform a read query as described by Adam Kirsch and Michael Mitzenmacher (Adam et al., 2007). Using the Bloom filter for read queries performed on the LSM-tree allows the application to avoid the need to do a disk I/O for levels that do not actually contain the element of interest. Research conducted (Kuszmaul, 2014) showed that in most cases, this means that a point query requires only one disk I/O. It is important to note that Bloom filters have limited capacity and depend on their configured size. Once all of the bits are set, the probability of a false-positive is 1.

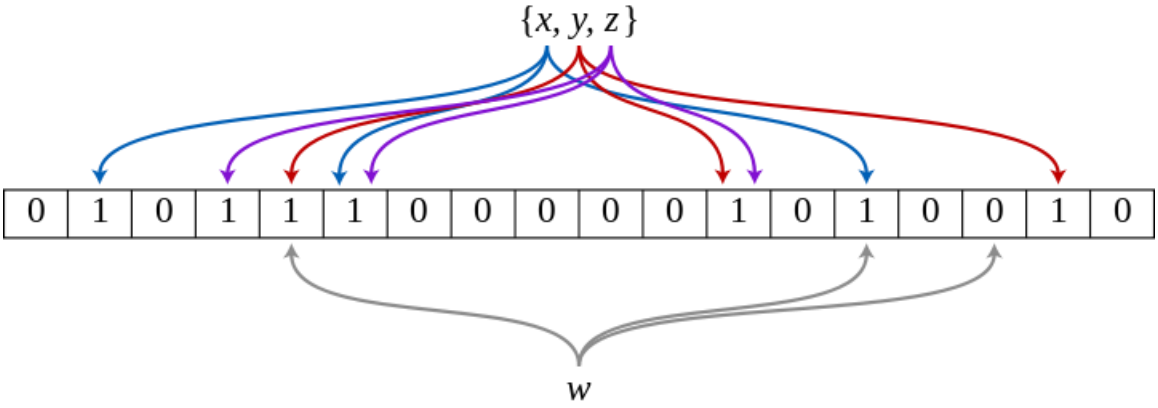

Figure 7 An example of a Bloom filter showing the set {x,y,z}. The element w is not in the set {x,y,z} because it hashes to one bit-array position containing a 0.

Rolling Merge Algorithm

CuttleTree uses a "leveling" LSM-tree design that consists of one B-tree data structure at each level. Here, we describe the algorithm that CuttleTree uses to make a complete compaction of the levels when a rolling merge is triggered.

Rolling Merge Algorithm

Here is a list of terms used for this algorithm:

P = CuttleTree tunable parameter to control the percentage copied to the next level
L = the total number of levels in the CuttleTree
C_n = the nth level
F = tunable parameter for the current level's max file size

1. **Move** P from C_0 to C_1
2. **For** C_1 to C_L
3. **If** F exceeded **then** move P to $C_{current} + 1$
4. **empty** $C_{current}$

C_0 Memory Level Initial Fill-Up

Most state-of-the-art LSM-tree based data systems in use today allow for configuring settings related to the level-based architecture. Although the primary way that CuttleTree adapts is though ongoing statistical collection, we can learn something

from the first fill-up of the C_0 memory level. An additional feature was added to CuttleTree in order to determine the optimal number of disk levels dynamically as opposed to setting them at initialization. It accomplishes this by deciding if, up until the first rolling merge is triggered, there have been more reads than writes. This works since a fill-up of the memory level triggers a rolling merge. If we haven't had a fill-up yet, we don't have to know the number of levels required. It has been noted, by interviewing engineers in industry, that such an automated adaptation can save time and effort when implementing data infrastructures with many individual data systems. We utilize this concept for an adaptive experiment in Chapter 4.

```
-----------------------------
INITIAL C0 FILL-UP STATISTICS
-----------------------------
4416 TOTAL OPERATIONS.

4001 INSERTS ~ 90.6024% OF TOTAL OPERATIONS.
0 READS ~ 9.39764% OF TOTAL OPERATIONS.
0 UPDATES ~ 0% OF TOTAL OPERATIONS.
0 DELETES ~ 0% OF TOTAL OPERATIONS.

SETTING LSM DISK LEVEL CONFIGURATION...
      ~~SETTING 5 DISK LEVELS TO HANDLE INSERT HEAVY WORKFLOW.~~
```

Figure 8 Using statistics up until the first rolling merge to set the disk level configuration.

CuttleTree Benchmark System

The CuttleTree benchmark implementation created allowed us to model various workloads and test CuttleTree's ability to adapt. It provided fine-grained control over the conditions of our experiments in order to isolate patterns where CuttleTree's adaptive behavior could be highlighted. To initialize the benchmark, the user provides CuttleTree with the total number of operations requested, what percentage of the workload should each of the four types of operations (read, insert, update and delete) be and the

benchmark type. The benchmark has three options, the first being to run the operations in sequential order. If 100 total operations are requested, with 50% inserts and 50% reads, CuttleTree will execute 50 inserts followed by 50 point queries. The second option creates a random execution of the total operations based on the percentages provided. The third option is a fully customizable user-created sequence. For example, a custom benchmark consisting of a sequential insert phase followed by a random insert and write phase and ending with a ready-heavy phase was used to mimic a real workload described to us by a Facebook engineer. A company like Facebook experiences a workload like this during a typical day and their engineers need to prepare for it. A database could be set up in the morning by Facebook engineers, followed by peak user activity during the day with people adding posts and clicking various links, and ending with a lighter period of user searches at night.

```
static void runBenchmark(int N, int benchmarkType, float insertPercent,
                         float readPercent, float updatePercent, float deletePercent);
```

Figure 9 Showing the initialization parameters for the CuttleTree benchmark system.

3.2 CuttleTree Statistics Collection Feature

Statistics collection allows for the dynamic decision making of CuttleTree to occur. Collecting detailed statistical information about the runtime of CuttleTree allowed us to answer questions that led determining the correct adaptive behavior for a given workload. Do we accurately capture the workload? Is the workload read or write-heavy? How long does a rolling merge take to complete? What is the cost of making the suggested adaptation and is it worth it? Our statistics collection implementation allowed us to answer these questions.

Implementation

We knew that we planned to use the CuttleTree statistics collection logic with other data systems, such as LevelDB, in addition to CuttleTree. For that reason, our C++ implementation was created in a separate class: *CuttleTreeStatistics*. An object of type *CuttleTreeStatistics* is created and passed to the data system at its initialization. Functions related to operations (read, write, update and delete) and modified to include appropriate calls to inform *CuttleTreeStatistics* of their actions. With each occurrence of these operations recorded, *CuttleTreeStatistics* is able to do the heavy-lifting. Its abilities include:

- Recording details about each level of the LSM-tree. For each level:
  - For each type of operation, record how many occurred.
  - Each level consists of one or more files which consists of one or more blocks. The number of these disk-resident blocks and how many of each type of operation occurred was recorded.
  - The minimum, maximum, mean and median number of times per level/block that each operation occurred.
- Counting how many rolling merges occurred and the average time to complete one.
- Recording the runtime duration and total number of operations, along with the throughput in operations per second.

Some of these statistics are collected in widely-used data systems such as LevelDB. With *CuttleTreeStatistics*, we are able to see more detailed information like activity at the block level. Since this *CuttleTreeStatistics* class was created as a separate module, we were able to implement it on top of LevelDB with relative ease. In Chapter 4, we describe experiments that utilized these statistics. The full output of these statistics, for CuttleTree and LevelDB, can be found in Appendix 1. In Appendix 1, we

demonstrate two LevelDB benchmarks that show accurate collection of block-level activity given differing read query patterns.

3.3 Cost & Complexity Analysis

Here is a list of terms used throughout this section:

N = Total number of entries
D_i = Total number of entries at the i level of the LSM-tree
L = Number of levels
E = Average size of data entries
B = Block size
B_{min} = The minimum block size, based on the filesystem
T = Size ratio between levels
T_{lim} = Size ratio value at which point L converges to 1
$IOPS$ = I/O operations per second

Rolling Merge Cost

As we have previously described, data are stored into multiple levels in an LSM-tree. New records are inserted to the C_0 memory level. When this memory buffer is full, its content is written to the C_1 disk-resident level. This potentially triggers one or more rolling merges, based on a threshold for file sizes, and involves subsequent levels to participate. We can optimize for read or write-heavy workloads based on this concept by adjusting the size ratio between levels. CuttleTree provides tunable parameters, *firstLevelFileSize* and *sizeBetweenLevels*, in order to manage a balance that is present in all LSM-trees between read and write performance. We can reduce the cost associated with the total number of rolling merges executed by using a larger value for *T*. We use

this cost analysis in Chapter 4 with the experiment entitled “Decreased Rolling Merge Times.”

Cost to Merge Levels

We know that when the memory-resident C_0 level reaches its threshold size, its data is synced to disk using a rolling merge. This data is merged with other disk-resident data in order to minimize the cost of read queries. If the number of levels is very large, it hurts read performance by increasing read amplification. If $L > 1$ at *CuttleTree's* initialization, and we detect a change to a read-heavy workload, we could merge the data in order to have $L = 1$ and see an improvement in read throughput. We learned from Monkey (Dayan et al., 2017) that as T approaches T_{lim}, the number of levels L approaches 1. We can make this happen by first adjusting *CuttleTree's* tunable parameter: *firstLevelFileSize* for C_1 to equal T_{lim}. Next, we merge subsequent levels $C_{2\text{-}L}$ to C_1. This involves copying all values at each subsequent level to C_1 so that the number of values in C_1 = N. The total number of operations (number of entries not in C_1) needed to accomplish this is:

$$N\text{-}D_1 = \sum_{i=2}^{L} D_i$$

We will utilize this levels merge strategy in Chapter 4 under the experiment entitled “Merging Levels to Reduce Read Amplification.”

Block Size Latency Cost

We know that CuttleTree statistics collection added to LevelDB exposes details at the block level such as how many blocks were created and how many times they were

read from. This LevelDB block has an adjustable size that does not necessarily have to coincide with the filesystem's notion of a block size. It is the size of the request that LevelDB uses to perform an I/O operation from disk. This size can have an impact on performance, as we will see in Chapter 4 with the experiment entitled "Block Statistics to Improve Random Reads Performance."

We can think of the cost to retrieve an entry from an LSM-tree as it relates to latency and throughput. Throughput, for our purpose, is impacted by the block size and I/O operations per second *IOPS* such that: $Throughput = IOPS \times B$

As B increases, so does the throughput. The amount of throughput required will determine the latency experienced by the LSM-tree per entry request. Requests made to the storage device will be based on B_{min}. If $B > B_{min}$ then multiple sectors of disk will be read from. This can be an important consideration, especially for range queries. For exact searches, low latency can be obtained by using Bloom Filters to identify the blocks that need to be read off storage. However, for range queries, bloom filters cannot be used to provide low latency reads in the same way. As B increases in size, the latency associated with each range query should tend to increase as well.

Chapter 4 Experimental Analysis

The main purpose of the experimental phase of this project was to test the system in order to learn about its characteristics. The CuttleTree implementation was tested using various hardware and all of the twelve tunable parameters. This allowed for the performance of different configurations of CuttleTree to be analyzed in order to make decisions based on design trade-offs inherent in its execution.

4.1 Hardware, Dataset and Workload

The primary computer used for testing was a 2015 Apple Macbook Pro Retina with 2.6GHZ Intel i7 quad core processor, 16gb RAM and a 1TB Apple SSD drive. An additional Apple Macbook Pro 2.6GHZ Intel i7 quad core processor with 16gbRAM and a 500GB HDD drive was used in order to test performance on various disks. The datasets used varied in size between 20k random numbers to 256 million random numbers generated using C++ and a seeded implementation of the *rand()* function in addition to a series of files that were loaded at application startup by CuttleTree's benchmark system. The workload varied from write-heavy and read-heavy to a mixture of the two at different ratios. In order to test different scenarios, workloads such as repetitive value reads (reading for the same value many times) and updates followed by deletes and reads were added.

4.2 CuttleTree Static Experiments

B-tree vs std::vector

The CuttleTree implementation provided a tunable parameter to change the C_0 memory resident level's data structure from a B-tree to a *C++ standard-library* vector. Various workloads were tested to see if there was a benefit to be had from either data structure. Reads and writes were tested, in sizes of 50k to 500k values, and no notable difference is throughput was determined. The design trade-off was made to choose a *C++ std::vector* for the final testing. This meant that inserts could be done in a simpler way, using the *push_back* method, as opposed to the B-tree implementation which had to readjust the shape of the tree after each insert. Various tests also showed that for smaller sets of data flow, a sorted vector can perform better than a B-tree. (Kuszmaul, 2014)

Finding the Optimal B-tree Node Size

Each level of the LSM-tree contains one or more tree-like data structure. These tree-like data structures contain many nodes. Each node contains a size that determines how many values may reside inside of it. How big should each node of the tree be? The well-established theory (see, for example, the Disk-Access Model (DAM) of (Aggarwal et al., 1988)) usually assumes that the underlying disk drives employ a fixed-sized block, and sets the block size of the tree to match that device block size. The theory also assumes that there is no difference between a small or large block size, leaving it to just an assessment of the number of blocks fetched from and stored to disk. The informative paper titled "Modern B-tree Techniques" (Graefe, 2011) offers the theory that optimal

node size is governed primarily by access latency and transfer bandwidth as well as the record size.

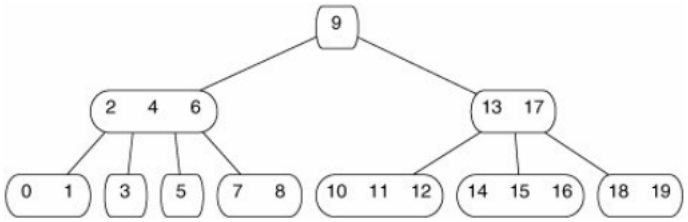

Figure 10 An example Tree-like data structure showing nodes of various sizes. The 2-4-6 node on the left shows a size of 3.

The CuttleTree implementation provided by this thesis project contains a tunable parameter to set the node size for both memory and disk resident data structures. It was determined, after extensive testing, that the optimal node size for the test machine used (a MacBook pro i7 quad core with 16gb of ram and a modern apple SSD drive) was 20. See Figure 2.0 below.

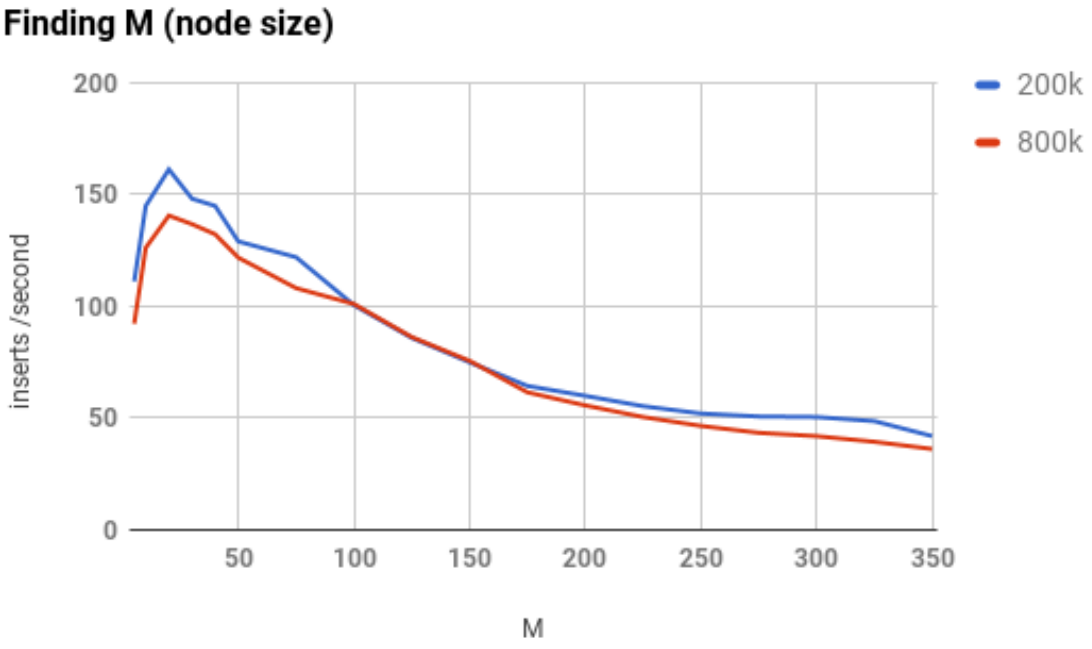

Figure 11 Displaying 20 as the optimal node size.

The results show an optimal value for the node size, M, to be 20. The above figure clearly displays that the best inserts per second throughput of both 200,000 and 800,000 values. The test inserts were done using random numbers and the data finally resided 50% in memory and 50% on disk. Modern B-tree Techniques (Aggarwal et al., 1988) declared that for a disk with .1ms access latency and 100 MB/s transfer bandwidth, a 10KB node size would be optimal for a modern flash device. Since the paper was several years old, it is possible that the newer SSD drive used for the testing reached higher levels of performance. Based on a 50byte allocation per value in CuttleTree, the test resulted in the size of 20 for nodes of the disk resident B-trees found in the $C_{1\text{-}n}$ levels.

Read Optimization Using Range Detection & Tombstone Delete

Two techniques were used in order to attempt to create a "read optimized" LSM-tree implementation. First, when an insert or read occurs, the data is examined by memory-resident checks in order to determine if disk access should occur. The CuttleTree contains two variables to contain the min and max values of the dataset. This creates a range with which future read requests can use in order to determine if a new value to be read could possibly reside in the CuttleTree. The second is the usage of tombstone deletes as opposed to traditional deletes or blind deletes. A traditional delete will probe the entire database to find if and where the key-value pair is located while a blind delete will skip the probing step and just execute the delete. CuttleTree can skip the on-disk read portion or execution of delete commands entirely by first checking if a value has already been placed in a memory-resident data structure that manages values "to be deleted." This is known as a "tombstone" value. In read-heavy flow patterns, especially

in which frequent read misses occur on the same data, this behavior can reduce the amount of disk I/O activity and speed up read performance.

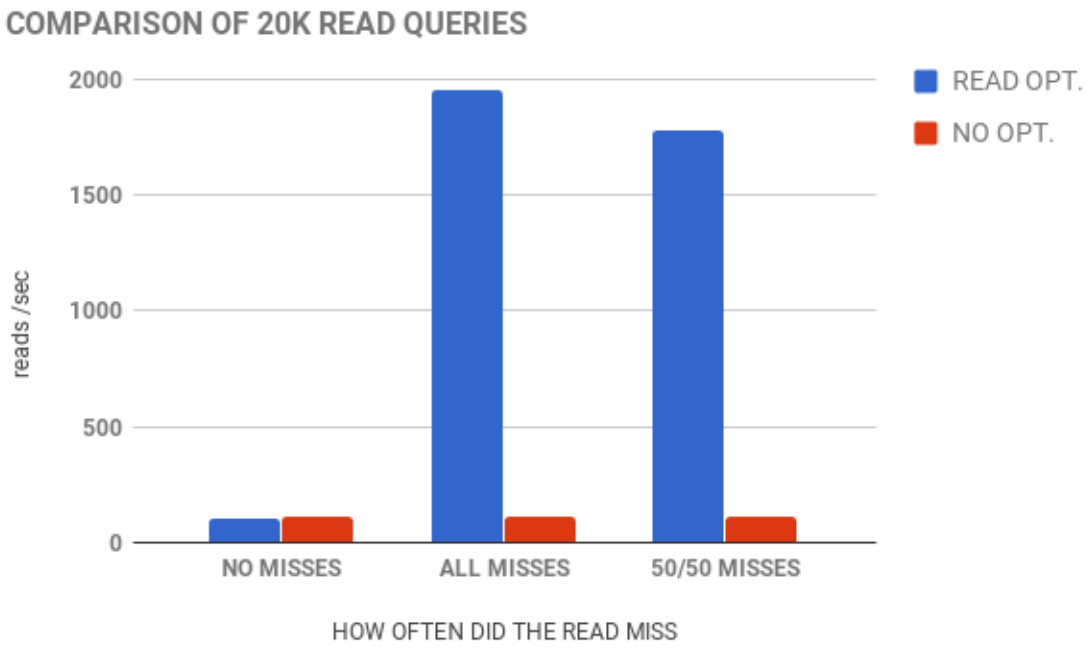

Figure 12 The throughput for 20k reads using read optimization and not using it.

The figure above shows that when 20k read attempts resulted in no misses, the throughput for the reads was very low using both read optimizations and not using them. This makes sense since we always have to access disk for successful reads from the disk-resident levels regardless of whether we make a read optimization or not. When the reads almost entirely resulted in all misses (the value did not exist in the database), the read optimized technique resulted in much better performance. This is also true, to less of a degree, when the read attempts resulted in about 50/50 success rate of finding a value. This improvement in performance can be attributed to the I/O cost savings realized by not making unneeded disk reads.

The tombstone delete technique was then isolated and compared to a blind delete form of deleting values from the database. This leads to a key design tradeoff found in this system. If the blind deletes are done, CuttleTree sends a request to each level to delete the value requested for removal. This results in potentially many disk operations since the value could be present on the last disk of CuttleTree. If the tombstone method is applied, data could be inconsistent in the B-trees found on disk, causing different results when inserting, reading, updating and deleting. Since the focus of this experiment was to create a read optimized CuttleTree, the tombstone delete was preferred. The tombstone method was found to outperform the blind delete method when there was a flow of many reads, especially when lots of reads are repeating the search for a particular value. The figure below shows the results of inserting 500k values following by deleting 50k values and then searching 500k values. In this instance, it does not matter if the 50k values are marked as tombstone or blindly deleted, the results are still very similar. In contrast on the right, when there are 500k inserts following by 50k deletes and 100k reads, of which 20% are repeated 3 times, the tombstone method is able to outperform blind delete. This is a situation in which the CuttleTree is able to quickly read from memory the tombstone vector which declares values that have been marked for deletion.

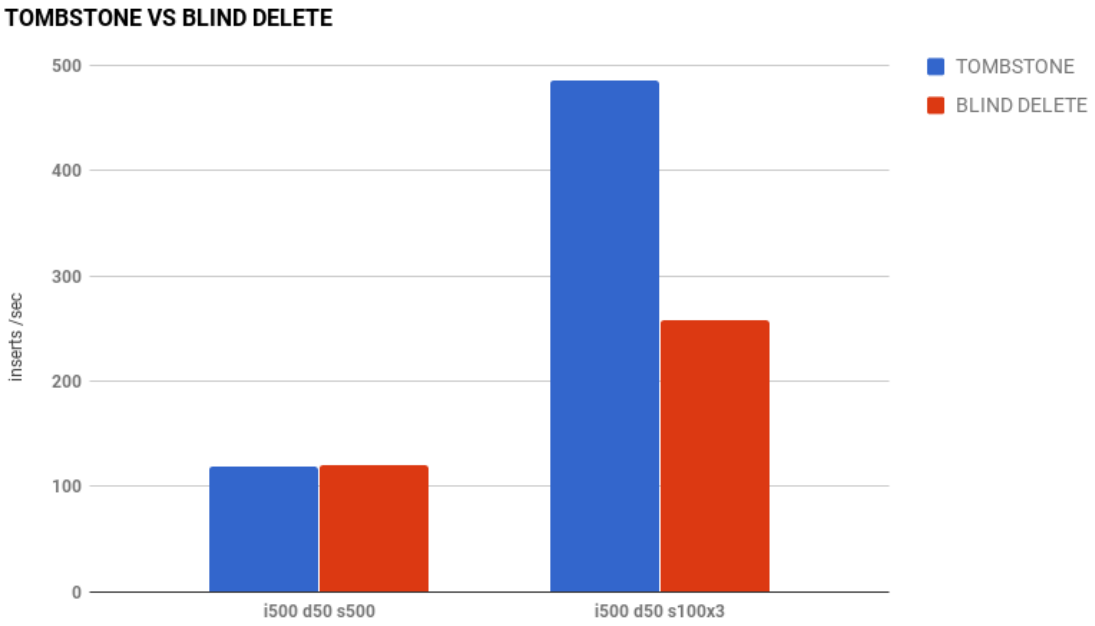

Figure 13 Displays TOMBSTONE vs BLIND deletes and shows TOMBSTONE outperforms when reads are frequently repeated.

Parallelization

In order to increase the performance of the overall CuttleTree design on a multicore system, two parallelization techniques were implemented. Since the memory and disk levels have certain functionality that coexists, the rolling merge calls were able to be moved to a separate thread. The current implementation has one thread execute all inserts to memory until a rolling merge is requested (since C_0 has reached its defined limit). The rolling merge is then executed in a separate thread and is then joined concurrently by a second thread which will fill C_0 back up until another rolling merge is requested. This process allows for C_0 to quickly absorb more inserts without having to wait for the rolling merge to compete as it would in a single threaded program. The figure below shows the results from such a dual threaded insert rolling merge technique.

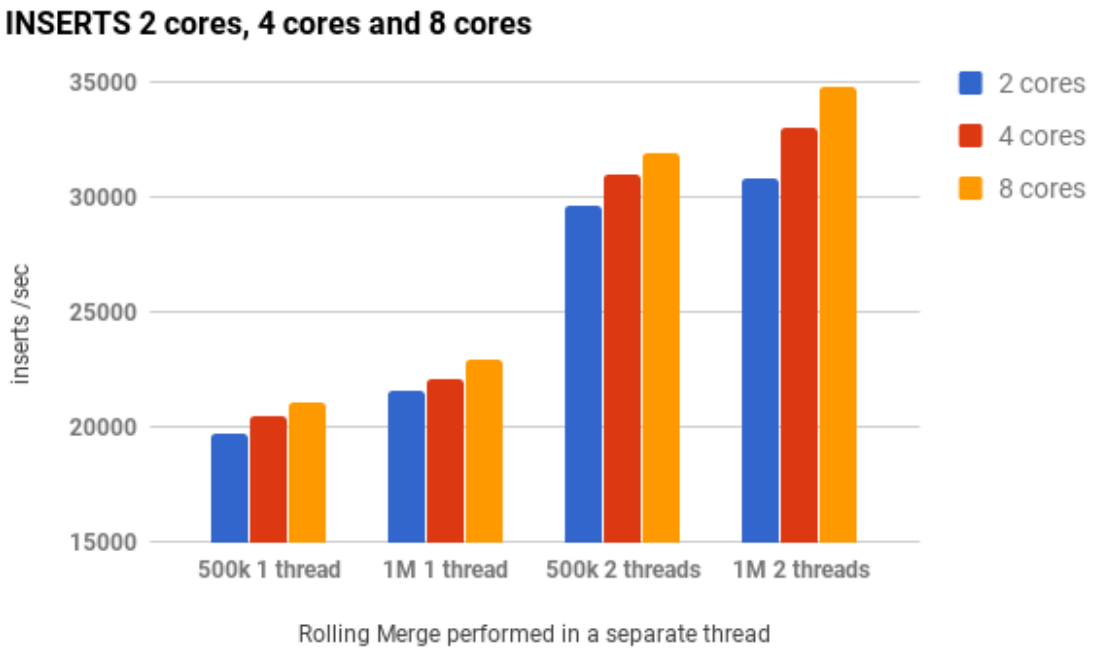

Figure 14 Showing an increase in throughput as additional cores are added to the multi-thread rolling merge.

It is apparent from the testing that when 2, 4, and 8 cores were tested using the multi-threaded process to run rolling merge, the throughput of inserts increased steadily.

In order to attempt to perform further read optimizations for CuttleTree, a worker queue was implemented. The worker queue is a thread safe object that maintains a list of "work" to be done by a pool of threads. Inserts were followed by large reads and the results were tested for a steady increase in threaded performance. The results showed that as an additional thread was spawned and able to take work from the queue, the overall throughput increased. This was true until the point when the additional threads spawned did not have the ability to further deplete the worker queue fast enough. It appears that for the system used for this experiment, worker threads were waiting for work starting at 5 threads.

The figure below shows an increased performance for both 500k and 1 million reads up until 5 threads are accessing the worker queue.

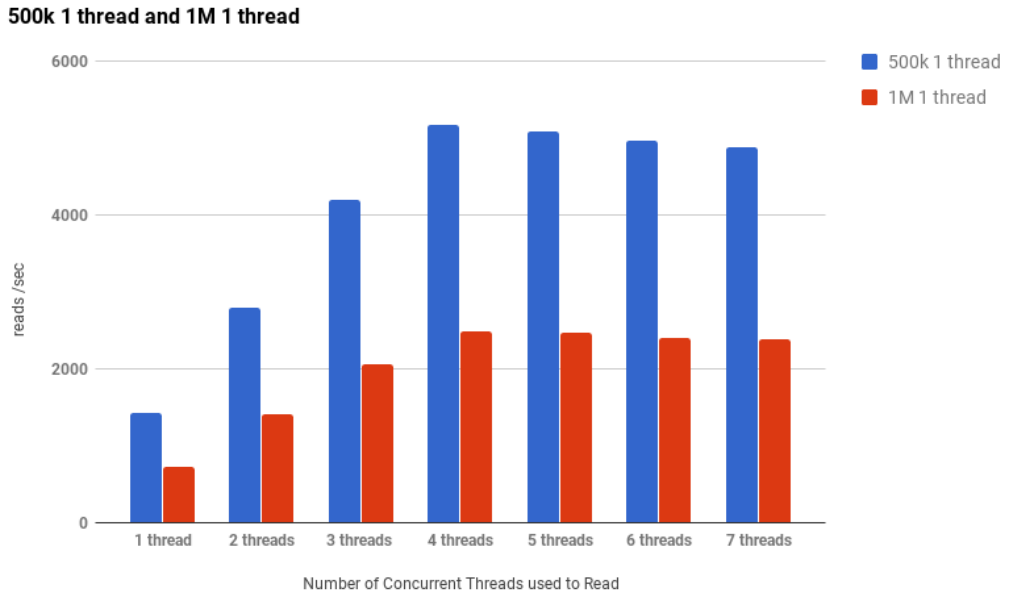

Figure 15 Showing an increase in throughput as additional cores are added to worker queue for reads until 5 threads.

HDD vs SSD

The major principal of the LSM-Tree is that it is used to create a better usage of disk I/O computation. Since the disk levels are stored as B-trees, it made sense to test the performance of these major industry-standard forms of hard drive storage. Tests were run to compare the HDD vs SSD in an Apple Macbook Pro. The first test to realize results from was starting with an empty CuttleTree and inserting incrementally from 10k to 80k inserts. The below figure shows the time that elapsed from start to finish for HDD and SSD drives. (time elapsed was chosen to highlight the near linear decrease in throughput for HDD)

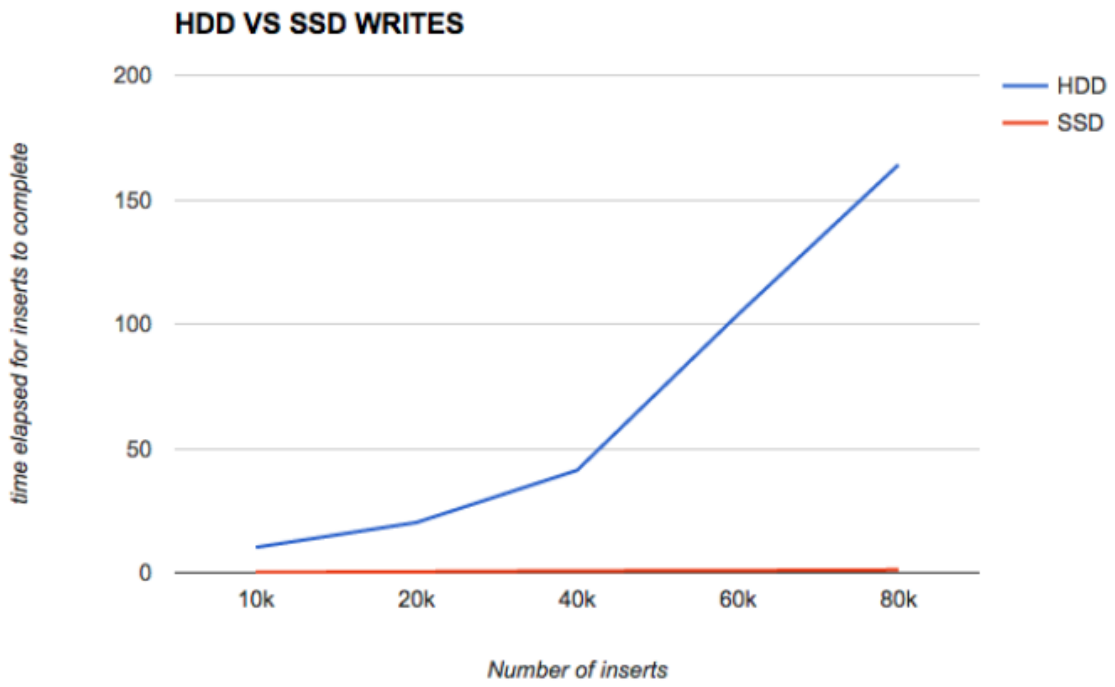

Figure 16 Comparing HDD vs SSD for writes.

The next test run was to run a workload of sequential reads from 10k to 80k using the same SSD and HDD equipped computers. The results, shown below, were that the HDD took more time to complete each incremental increase in reads while the SSD remained steady. It was noted that reads tend to perform slower than writes and that the SSD drive did start to show a decrease in throughput at approximately 70k inserts.

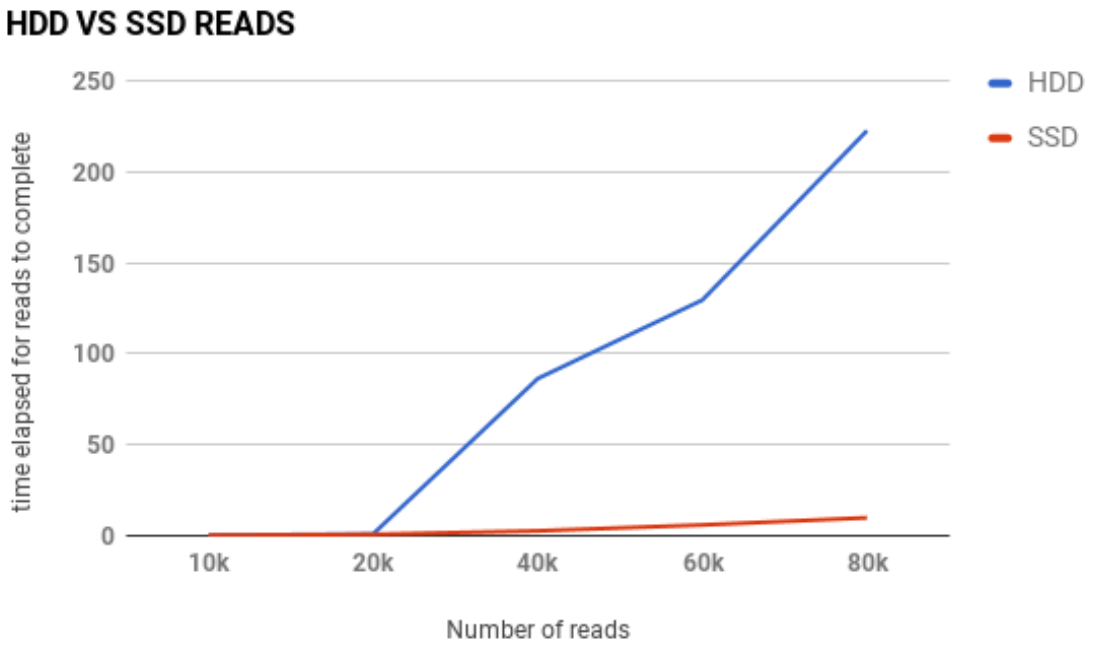

Figure 17 Comparing HDD vs SSD for reads.

The results produced by these tests were not surprising. They did act as confirmation that the CuttleTree implementation was steady enough to produce reliable results. Testing the HDD at higher levels of transactions would take a lot more time to run and didn't seem to yield any interesting further results. It was interesting to put the SSD to a more stressful test to see the throughput behavior. The figure below shows the results of SSD sequential writes from 100k to 3.2 million total.

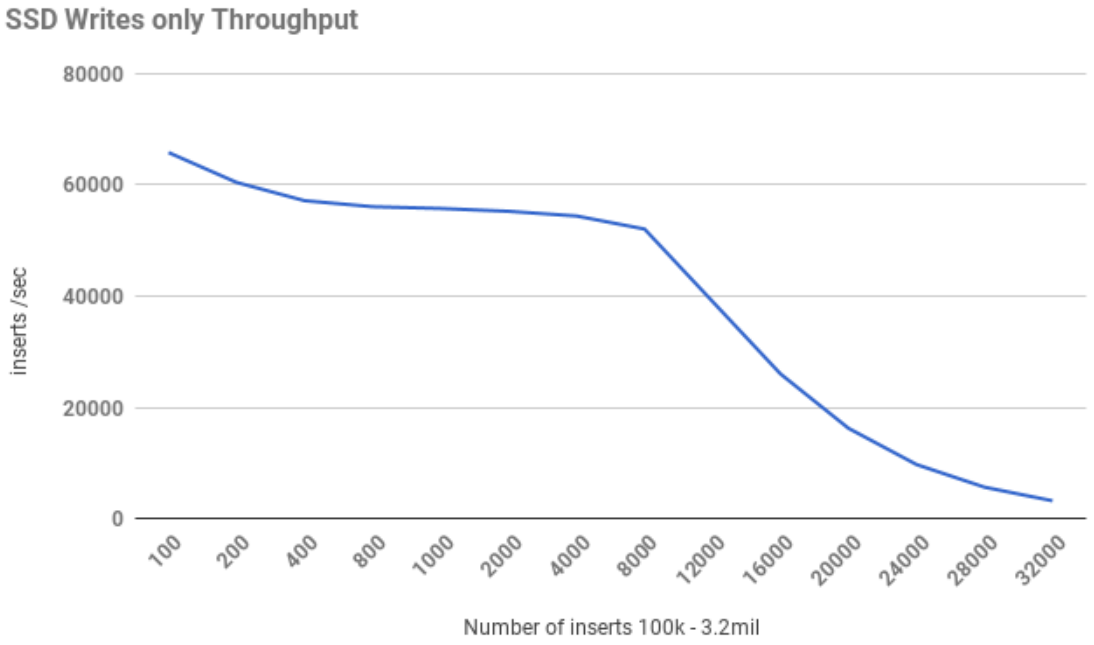

Figure 18 Showing higher number of inserts for SSD drive.

The results confirm that the throughput decreases in a near linear manner as the number of inserts increases.

The figure below demonstrates that even with attempts made to increase read performance, the CuttleTree still has faster write patterns than read. It shows a much more severe decrease in throughput and becomes very slow at the 8 million reads point.

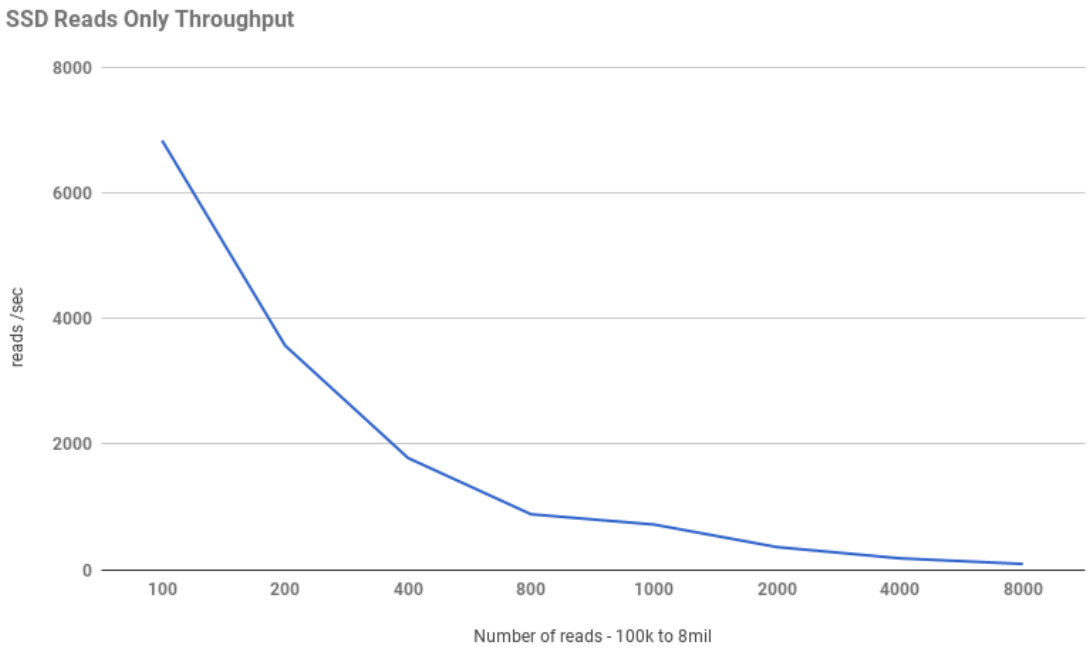

Figure 19 Showing higher number of reads for SSD drive.

In order to confirm consistent behavior for the design, updates were executed on the SSD drive as well. The range of 100k to 1 million updates were tested and a linear decrease in throughput was determined. The figure below shows these findings which confirmed consistent and expected results.

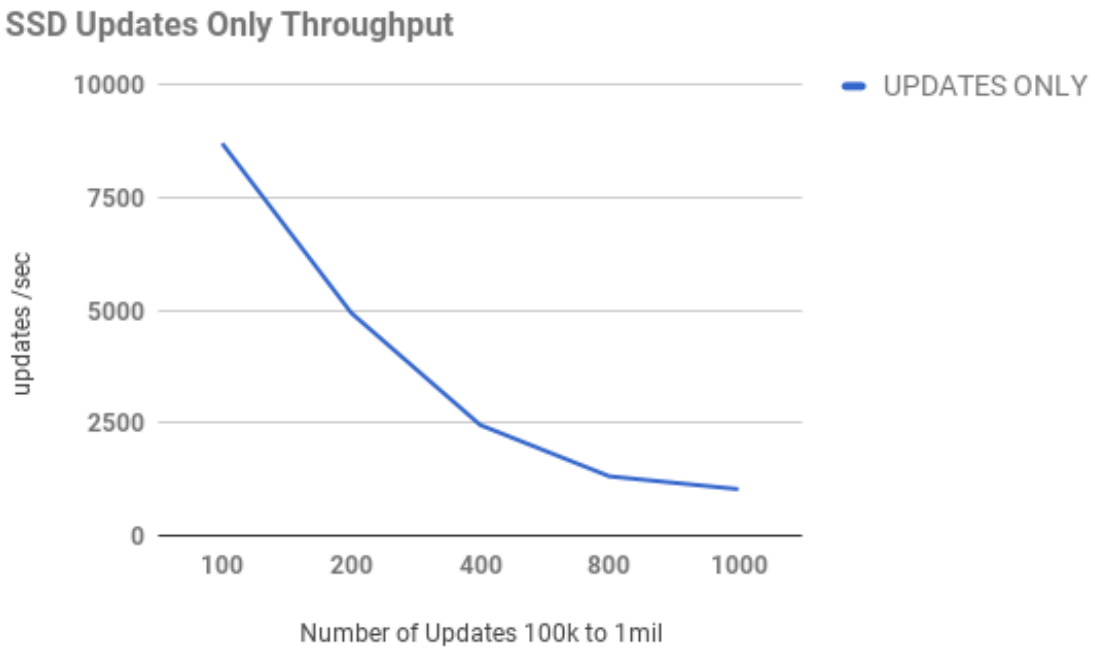

Figure 20 Showing updates for the SSD drive.

Inserts Throughput as Levels Increase

The first test was designed to measure the throughput as the amount of levels used by CuttleTree increased. This was easy to perform based on the tunable parameters present in the system for setting the amount of levels of CuttleTree, the size of each level, the size ratio between levels, the size of the memory level at C_0, and the amount of data that is to be copied from level to level. The experiment was run with 200k and 400k inserts and a maximum of 5 disk resident levels in addition to the C_0 memory level. The results in the figure below show that as the system has to insert to deeper levels, the throughput decreases. There is particular point noted at levels after C_2 that suggests that it may be optimal to have no more than two disk resident levels in the CuttleTree.

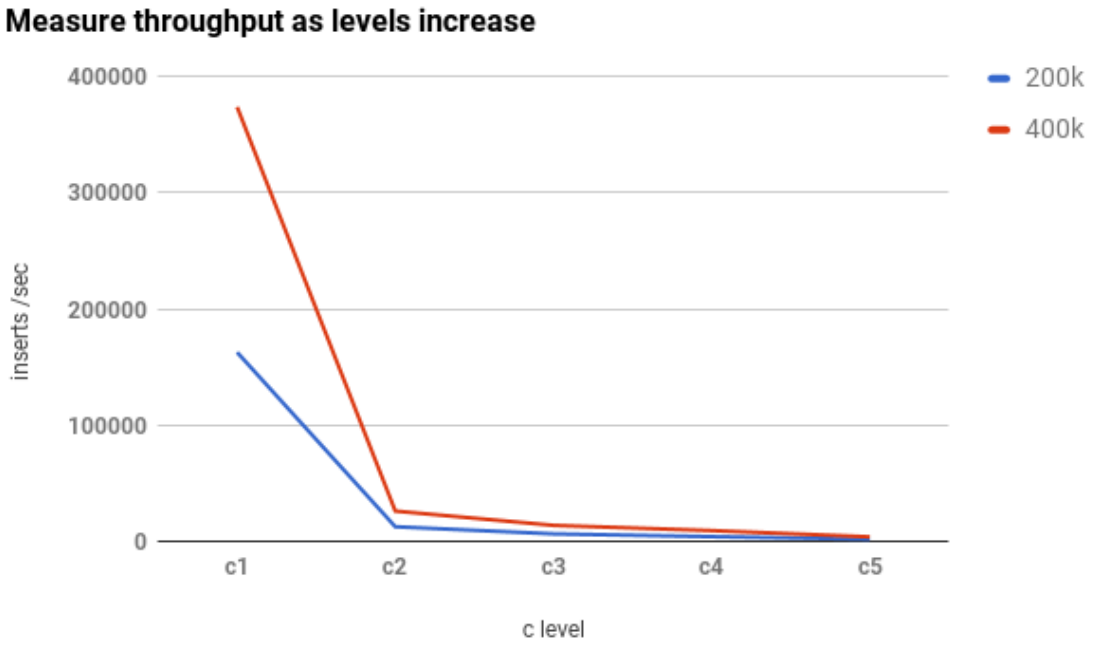

Figure 21 Showing throughput with total levels increase.

Read to Write Ratio

This test was to alter the read vs write ratio of the benchmark. Starting with 100% writes and moving to 90% writes and 10% reads all the way to 100% reads measured for transactions per second yielded the following results demonstrated in the figure below.

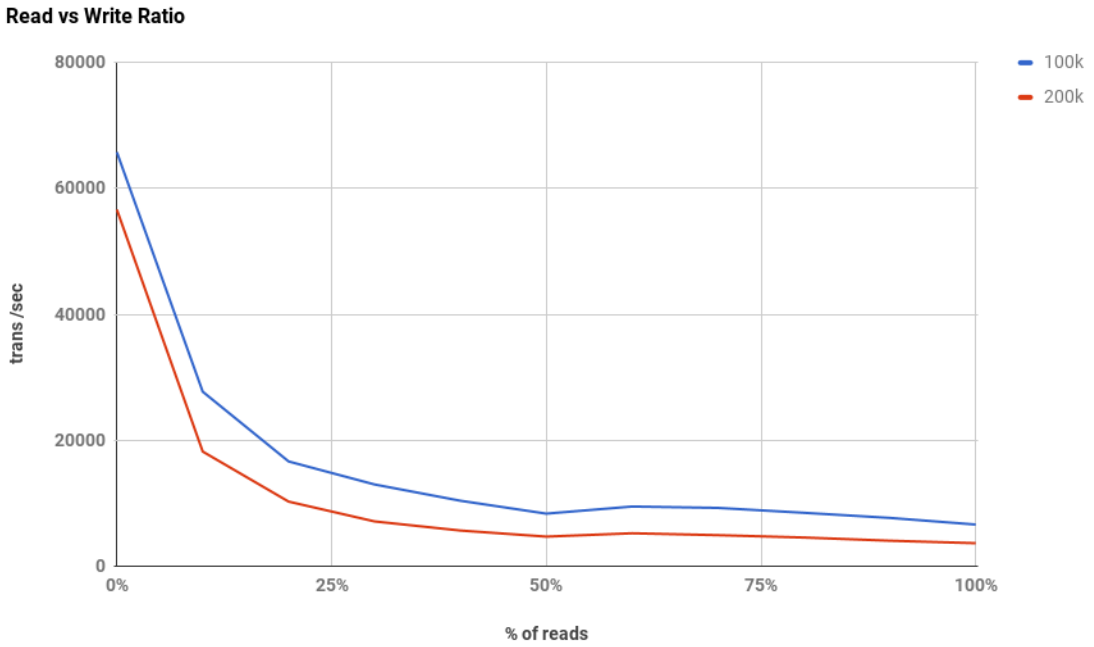

Figure 22 Showing Read vs Write ratio.

The results verify that as the read-heavy workload increased, the over transaction per second decreased. This test offered verification that the CuttleTree design was behaving in a reliable and predictable way that gave some credibility to the slight increase in performance witnessed by the read optimizations offered by this project.

4.3 CuttleTree Adaptive Experiments

The implementation of CuttleTree has twelve tunable parameters. In order to fully understand the system, and now that the consistency of the system has been confirmed by the previous tests, it was time to create experiments that could lead to future research in adaptive and tunable performance parameters for an active LSM-tree.

Merging Levels to Reduce Read Amplification

In the case when CuttleTree detects a change from a write-heavy to read-heavy workload, an adaptive decision can be made to improve overall throughput. After CuttleTree's first several statistics collection actions, it detected a write-heavy workload and set the maximum number of levels to 5. Many more insert requests followed but at some point, CuttleTree detected a change to a read-heavy phase. We know that read amplification can be reduced by decreasing the total number of levels needed to probe. We changed the number of levels to one based on the cost described in Chapter 3 under "Cost to Merge Levels." Having one level reduced the number of disk seeks per read. The figure below demonstrates that from a range of 100k to 1 million operations, this adaptive behavior improves CuttleTree's read throughput.

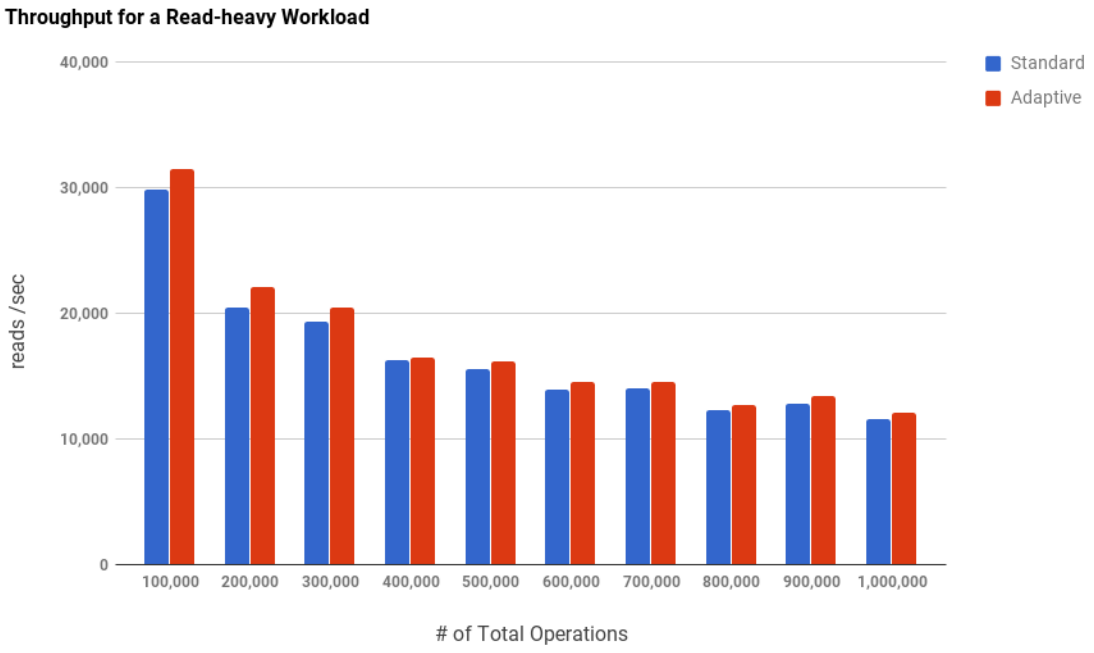

Figure 23 Showing Throughput for a read-heavy workload as a result of merging levels.

Decreased Rolling Merge Times

CuttleTree's statistics collection allowed us to analyze the average time needed to complete a rolling merge. We can isolate this statistic and give CuttleTree an adaptive behavior based on the size ratio between levels for a merge policy improvement. This adaptation is based on the cost analysis provided in Chapter 3 under "Rolling Merge Cost." We ran between 100k and 1 million total operations of which 90% were inserts and 10% were random reads. The benchmark was set to randomize the order of these operations. Since there were 90% inserts, CuttleTree detected an insert-heavy workload and adjusted by increasing the size ratio between levels *T*. This adaptive behavior, demonstrated in the figure below, was compared to a standard execution that did not modify *T* during runtime.

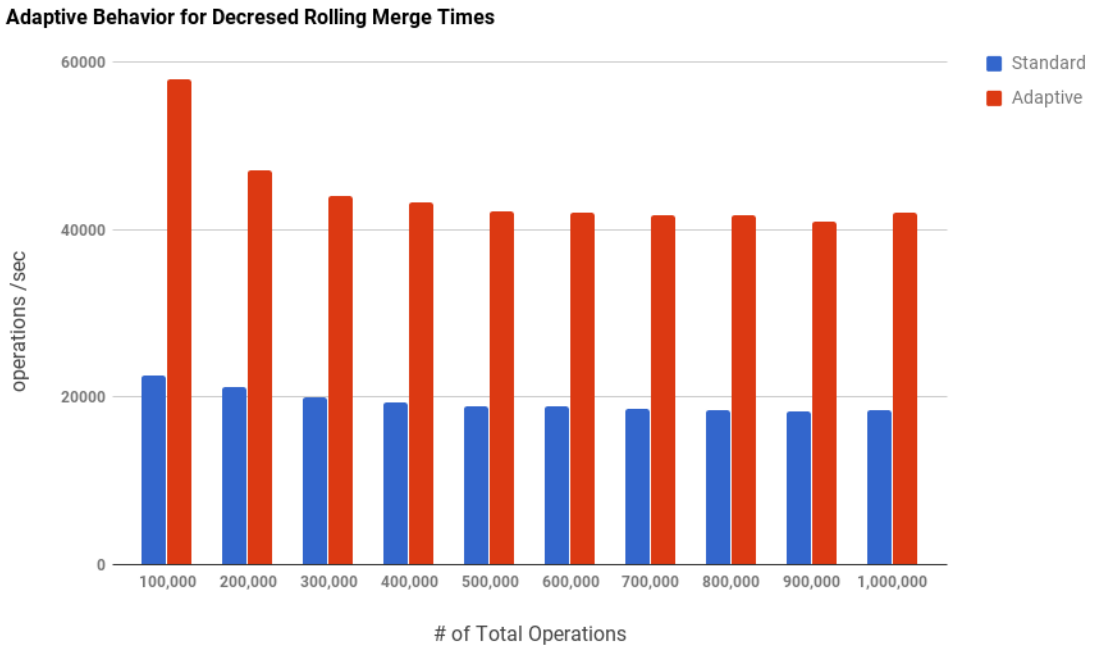

Figure 24 Showing adaptive behavior for decreased rolling merge times.

Adaptive Behavior & The Multi-Phase Workload

From interviews conducted with engineers in industry, we heard comments such as "I want the LSM-tree to adapt at runtime. For example, you might start with a write-heavy workload while loading, then do read-write during the day, then do read-mostly overnight, repeat." We decided to take the techniques described thus far and adapt to this multi-phase workload. We used CuttleTree's benchmark system to create a workload that starts as write-heavy, changes to a mixed read and write phase, and ends as read-heavy. We did this for 250k to 2.5 million total operations.

From the previous experiment entitled "Inserts Throughput as Levels Increase", we learned that CuttleTree achieves the best insert-per-second throughput with two disk-resident levels. Therefore, CuttleTree initially detects an insert-heavy phase and creates an upper bound constraint of two disk levels as an optimization. As more reads come in, CuttleTree statistics eventually detects that the workload has changed from a mixed insert/read phase to a read-heavy phase. The levels are combined to make one disk resident level and C_0 is flushed to disk so that there were fewer places to probe.

On the other hand, a non-adaptive version of CuttleTree is set for the standard 5 level configuration and does not adapt during runtime. It incurs higher cost penalties, as described in Chapter 3 under "Cost & Complexity Analysis" and the figure below shows its inferior performance when compared to its adaptive counterpart. The figure below suggests that the cost to compact the data to one level is higher as N increases (total throughput decreases ~1.5 million total operations) but still yields improved overall throughput when compared to the standard non-adaptive version of CuttleTree.

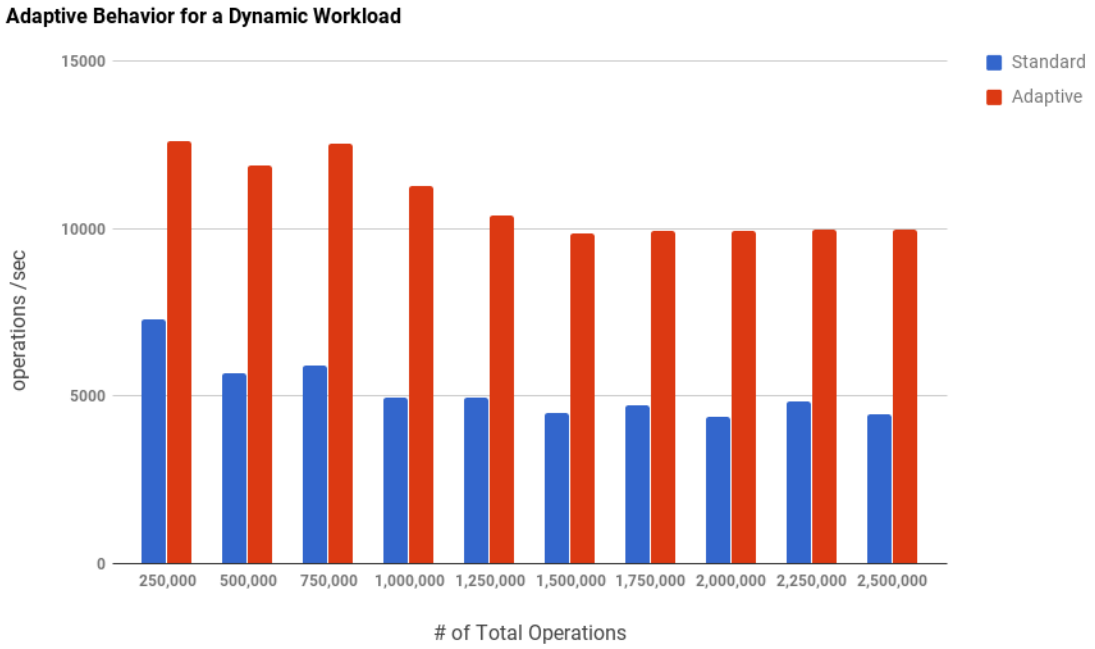

Figure 25 Showing adaptive behavior for a dynamic workload.

4.4 CuttleTree Statistics Collection & LevelDB

Using Block Statistics to Improve Random Reads Performance

Our implementation of CuttleTree statistics on top of LevelDB allowed us to reveal more detailed statistics than it otherwise would provide. In this experiment, we demonstrate the usefulness of these statistics in an attempt to find the optimal block size for LevelDB. We know that CuttleTree statistics added to LevelDB exposes details at the block level such as how many blocks were created and how many times they were read from. Based on the "Block Size Latency Cost" analysis in Chapter 3, we can use the fact that latency associated with LevelDB's block size impacts throughput of random reads, especially for range queries since they don't benefit from bloom filters like point queries can. For this experiment, we set LevelDB's block size incrementally higher from

100bytes to 2mb and used a standard LevelDB benchmark (db_bench). We started with a fill sequence of 10 million entries and then did read queries randomly for 10 million times. The figure below shows the total throughput. From these results, we can determine that a LevelDB block size of ~100k provides the optimal balance when considering the trade-off between minimizing the cost associated with the latency to retrieve data from disk and the insert-friendly nature of maintaining larger block sizes.

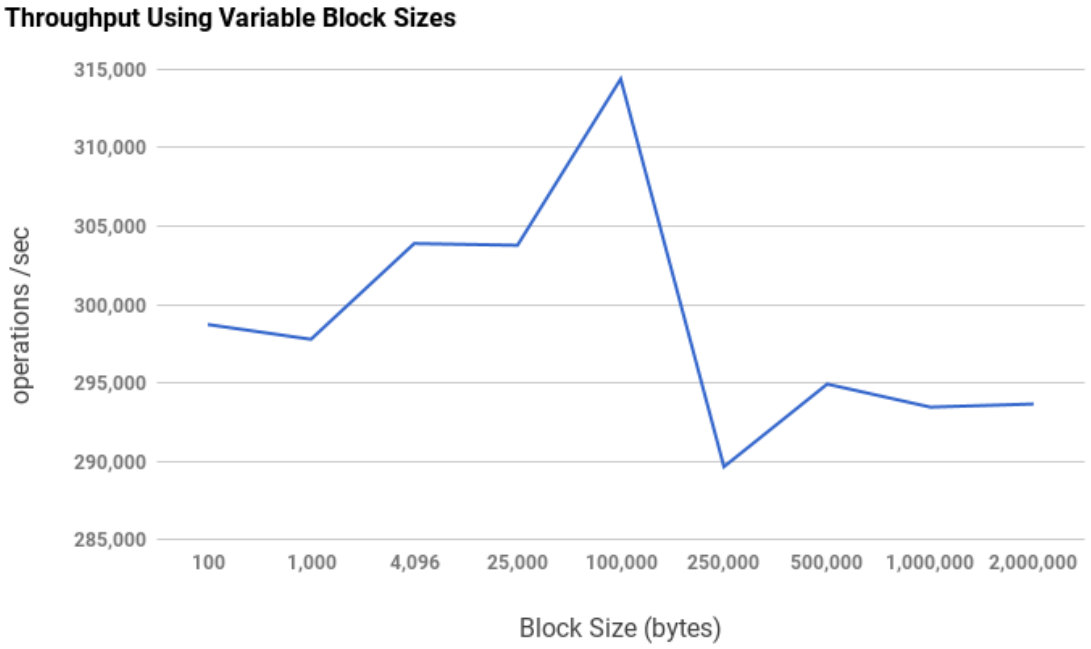

Figure 26 Showing throughput using variable LevelDB block sizes.

4.5 Strengths and Weaknesses

The main strength of CuttleTree is its ability to adapt to certain workload-related changes during runtime. It uses its tunable parameters and statistics collection to facilitate dynamic decision making. While it does allow for improved performance with occasional workload changes, like one shift from insert to read-heavy, it fails to calculate and weigh

the cost to make some of its changes on the fly. For example, in this chapter's experiment entitled "Adaptive Behavior & The Multi-Phase Workload", the cost to compact all the entries *N* is not weighed against the gain in performance at the time of decision making. For example, if the read queries up to that point resulted mostly in bloom filter intervention that prevented disk I/O activity, then CuttleTree should probably not pay the cost to decrease read amplification theoretically.

Chapter 5 Summary and Conclusions

While modern LSM-trees come with a large array of tunable parameters, configuring them typically requires that we know what to expect during the lifetime of an application's execution at the beginning. Improvements made on these structures could have a significant impact on the performance of database systems that are used by the vast majority of technology companies today. We introduce CuttleTree, an LSM-tree based key-value store that uses an array of tunable parameters and statistics collection to allow for optimized decision making at runtime. CuttleTree's benchmark provides more fine-grained control over the conditions of our experiments than any state-of-the-art data system we reviewed. The CuttleTree statistics platform facilitated dynamic decision making by encouraging adaptive behavior based on the fast detection of workload changes.

Future Work

CuttleTree makes decisions during runtime in order to optimize performance. Future work could incorporate real-world constraints, such as limited memory or a company's choice to limit budget towards additional disk space, when executing decision making. Further, we could incorporate the cost-benefit analysis of rolling merges related to "leveling" vs "tiered" LSM-tree implementations to determine which type is better for

the workload at hand and determine the cost to shift the data from one type to the other dynamically.

We designed CuttleTree to detect the workload phases as the program runs. We began to create a hardware diagnostic sequence that would allow for more adaptive setup. In Appendix 1, we demonstrate this "Initial Adaptive Tuning" phase. First, it attempts to understand the health of the hardware by measuring the change in CPU temperature before and after running a stress test. It then collects disk related information such as block size and how much physical memory is available. Future work could use this information to determine the appropriate size for the C_0 memory level and, based on the anticipated size of each entry, we could use the results from our "Finding M (node size)" experiment found in Chapter 4 to determine the optimal node size for disk-resident B-trees. The trade-off between the cost of memory and disk capacity could be weighed to alter the demands placed on the hardware by CuttleTree.

References

Kirsch, Adam, Mitzenmacher, Michael (2007), “Less Hashing, Same Performance: Building a better Bloom Filter”, Random Structures & Algorithms, 2008, Vol.33(2), pp.187-218 [Peer Reviewed Journal]

A. Aggarwal and J. S. Vitter. The input/output complexity of sorting and related problems. Commun. ACM, 31(9):1116– 1127, 1988.

Ammeraal, L. (1996). Algorithms and Data Structures in C++, 1st ed. John Wiley & Sons Ltd.

N. Dayan, M. Athanassoulis, and S. Idreos. (2017). Monkey: Optimal navigable key-value store.

Goetz Graefe (2011), "Modern B-Tree Techniques", Foundations and Trends® in Databases: Vol. 3: No. 4, pp 203-402. http://dx.doi.org/10.1561/1900000028

Bradley C. Kuszmaul. (2014). A Comparison of Fractal Trees to Log-Structured Merge (LSM) Trees. Retrieved June 06, 2016, from http://insideanalysis.com/wp-content/uploads/2014/08/Tokutek_lsm-vs-fractal.pdf

Prashanth Menon, Tilmann Rabl, Mohammad Sadoghi, Hans- Arno Jacobsen. (2014). Optimizing Key-Value Stores for Hybrid Storage Architectures. Retrieved June 01, 2016, from http://www.msrg.org/publications/pdf_files/2014/CASCON1 5-Menon-Optimizing_Key-V alue_Stores_fo.pdf

Patrick O’Neil, Edward Cheng, Diter Gawlick and Elizabeth O’Neil. (1996). The Log-Structured Merge-Tree (LSM-tree). Retrieved June 11, 2016, from http://paperhub.s3.amazonaws.com/18e91eb4db2114a06ea61 4f0384f2784.pdf

Russell Sears, Raghu Ramakrishnan. (2012). bLSM: A General Purpose Log Structured Merge Tree. Retrieved June 20, 2016, from http://www.eecs.harvard.edu/~margo/cs165/papers/gp- lsm.pdf

Appendix 1 CuttleTree Sample Output

CuttleTree Output

The CuttleTree output starts with an “Initial Adaptive Tuning” phase which collects hardware-related statistics to be used for initial configuration. It uses the entry size and disk block size to calculate the optimal node size for the disk-resident B-trees. It displays the total amount of physical memory so that a C_0 size can be selected.

```
----------------------------------
STARTING INITIAL ADAPTIVE TUNING...
----------------------------------
The CPU Temperature is 68.0°C

Running the stress test...
Last Number in the Fib. sequence - 514229
Stress test complete!

The CPU Temperature is 68.2°C

The disk block size is 4096 bytes.
The size of one key value pair is: 16
The calculated value for the optimal node size is: 256

Total Physical Memory  =   16777216 k
Total Physical Memory  =      16384 MB
Total Physical Memory  =         16 GB

The optimal size for c0 =    8388608 k

Processor Information:
Intel(R) Core(TM) i7-4960HQ CPU @ 2.60GHz
--------------------------------
INITIAL ADAPTIVE TUNING COMPLETE!
--------------------------------


--------------------------------
RUNNING THE LSM TREE BENCHMARK
--------------------------------
1100000 TOTAL OPERATIONS
```

```
100000   INSERTS.
1000000  READS.
0        UPDATES.
0        DELETES.


-----------------------------
INITIAL C0 FILL-UP STATISTICS
-----------------------------
6001 TOTAL OPERATIONS.

6001 INSERTS ~ 100% OF TOTAL OPERATIONS.
0 READS ~ 0% OF TOTAL OPERATIONS.
0 UPDATES ~ 0% OF TOTAL OPERATIONS.
0 DELETES ~ 0% OF TOTAL OPERATIONS.


SETTING LSM DISK LEVEL CONFIGURATION...

      ~~SETTING 5 DISK LEVELS TO HANDLE INSERT HEAVY WORKFLOW.~~
```

Next, during runtime, it optionally outputs information related to each increment of operations that CuttleTree statistics uses to determine workload conditions. It displays actions related to all adaptive decision making.

```
15000 OPERATIONS HAVE OCCURRED. CALIBRATING CUTTLETREE using ADAPTIVE
TUNING...
INSERTS/READS -> 15001/0
INSERT PERCENTAGE IS 100
INSERT HEAVY WORKFLOW DETECTED, ADAPTIVE TUNING COMPLETE!
…
…
…
15000 OPERATIONS HAVE OCCURRED. CALIBRATING CUTTLETREE using ADAPTIVE
TUNING...
INSERTS/READS -> 765051/0
INSERT PERCENTAGE IS 100
~~~ADAPTIVE TUNING COMPLETE!~~~
…
…
…
15000 OPERATIONS HAVE OCCURRED. CALIBRATING CUTTLETREE using ADAPTIVE
TUNING...
INSERTS/READS -> 1008244/416851
INSERT PERCENTAGE IS 70.7492
~~~ADAPTIVE TUNING COMPLETE!~~~
…
…
…
```

```
15000 OPERATIONS HAVE OCCURRED. CALIBRATING CUTTLETREE using ADAPTIVE
TUNING...
INSERTS/READS -> 1199999/1200161
INSERT PERCENTAGE IS 49.9966
READ HEAVY WORKFLOW DETECTED, ADAPTIVE TUNING COMPLETE!

~~~ADAPTIVE TUNING COMPLETE!~~~
…
…
…
15000 OPERATIONS HAVE OCCURRED. CALIBRATING CUTTLETREE using ADAPTIVE
TUNING...
INSERTS/READS -> 1199999/1995214
INSERT PERCENTAGE IS 37.5562
~~~ADAPTIVE TUNING COMPLETE!~~~
…
…
…
~~~IT TOOK 81.7556 SECONDS TO PROCESS 1,100,000 OPERATIONS~~~
```

After runtime completes, CuttleTree outputs basic statistics about each level including the final values count and total size of the files.

```
C0 VALUES COUNT: 3976
--------------------------------------------------------------------------
--------
C1 file size is - 968808
LEVEL NUMBER - 1
maxFileSize - 200000
lsmLevelDisk - 0x7fff5fbe5050
fileName - c1.bin
the values count in this level is 23017

--------------------------------------------------------------------------
--------
C2 file size is - 872888
LEVEL NUMBER - 2
maxFileSize - 400000
lsmLevelDisk - 0x7fff5fbe7820
fileName - c2.bin
the values count in this level is 19698

--------------------------------------------------------------------------
--------
C3 file size is - 1035952
LEVEL NUMBER - 3
maxFileSize - 800000
lsmLevelDisk - 0x7fff5fbe9ff0
fileName - c3.bin
the values count in this level is 22520

--------------------------------------------------------------------------
--------
```

```
C4 file size is - 1505960
LEVEL NUMBER - 4
maxFileSize - 1600000
lsmLevelDisk - 0x7fff5fbec7c0
fileName - c4.bin
the values count in this level is 31682

------------------------------------------------------------------------
--------
C5 file size is - 297368
LEVEL NUMBER - 5
maxFileSize - 3200000
lsmLevelDisk - 0x7fff5fbeef90
fileName - c5.bin
the values count in this level is 4404
```

Finally, CuttleTree statistics displays summary statistics. It shows the number of operations that occurred at each level (and each block for LevelDB). It displays total operations taking internal operations, like those incurred during a rolling merge or reads at multiple levels, into consideration, and displays the number and average running time for the rolling merges. Finally, the elapsed time and throughput is displayed.

```
--------------------------
~~CUTTLE TREE STATISTICS~~
--------------------------

THERE ARE 6 LEVELS (including C0 memory level)

Level 0
Number of Reads = 999987
Number of Writes = 99999
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Level 1
Number of Reads = 960725
Number of Writes = 96023
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Level 2
Number of Reads = 756375
Number of Writes = 128383
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Level 3
Number of Reads = 555035
Number of Writes = 279781
```

```
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Level 4
Number of Reads = 338711
Number of Writes = 118341
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
Level 5
Number of Reads = 39206
Number of Writes = 15266
Number of Deletes  = 0
Number of Updates = 0
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~
TOTAL READS = 3650039
TOTAL WRITES = 737793
TOTAL DELETES = 0
TOTAL UPDATES = 0
TOTAL OPERATIONS = 4387832

NUM. ROLLING MERGES = 249
AVERAGE TIME FOR A ROLLING MERGE = 0.194368 SECONDS

~~~IT TOOK 81.7556 SECONDS TO PROCESS 4387832 INTERNAL OPERATIONS~~~


THE THROUGHPUT WAS 53670.1 operations per second
--------------------------------
CUTTLE TREE STATISTICS COMPLETE!
--------------------------------
```

CuttleTree Statistics & LevelDB Output

Here, we show the CuttleTree statistics class built on top of LevelDB to demonstrate block-level details. For this output, we display the LevelDB benchmark system running a fill sequence followed by a read-heavy phase that isolates ~1% of the database. We can see that the total blocks read from is only 5 of the 525 created.

```
LevelDB:    version 1.20
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compression)
Entries:    1000000
RawSize:    110.6 MB (estimated)
FileSize:   62.9 MB (estimated)
------------------------------------------------
fillseq      :       2.833 micros/op;   39.1 MB/s
readhot      :       2.793 micros/op;
```

```
-------------------------
~~CUTTLE TREE STATISTICS~~
-------------------------

TOTAL NUMBER OF BLOCKS -> 525
TOTAL BLOCKS READ FROM -> 5
TOTAL BLOCK READS -> 779256
MIN NUMBER OF BLOCK READS -> 117022
MAX NUMBER OF BLOCK READS -> 221122

THE AVERAGE (mean) # of BLOCK READS -> 155851
THE AVERAGE (median) # of BLOCK READS -> 220152

TOTAL MEMORY LEVEL (c0) READS -> 0
TOTAL DISK LEVELS (c1-n) READS -> 1000000
TOTAL WRITES FOR ALL LEVELS -> 1000000


~~~IT TOOK 5.6434 SECONDS TO PROCESS 2000000 OPERATIONS~~~


THE THROUGHPUT WAS 354396 operations per second
--------------------------------
CUTTLE TREE STATISTICS COMPLETE!
--------------------------------
```

For this output, we display the LevelDB benchmark system running a fill sequence followed by a read-heavy phase that is random across the entire database. As opposed to the previously displayed output, we can see that the total blocks read from is now 455 of the 525 created.

```
LevelDB:    version 1.20
Keys:       16 bytes each
Values:     100 bytes each (50 bytes after compression)
Entries:    1000000
RawSize:    110.6 MB (estimated)
FileSize:   62.9 MB (estimated)
------------------------------------------------
fillseq      :       2.839 micros/op;   39.0 MB/s
readrandom   :       3.585 micros/op; (1000000 of 1000000 found)

-------------------------
~~CUTTLE TREE STATISTICS~~
-------------------------

TOTAL NUMBER OF BLOCKS -> 525
TOTAL BLOCKS READ FROM -> 455
TOTAL BLOCK READS -> 985162
MIN NUMBER OF BLOCK READS -> 1644
```

```
MAX NUMBER OF BLOCK READS -> 2350

THE AVERAGE (mean) # of BLOCK READS -> 2165
THE AVERAGE (median) # of BLOCK READS -> 2202

TOTAL MEMORY LEVEL (c0) READS -> 12164
TOTAL DISK LEVELS (c1-n) READS -> 987836
TOTAL WRITES FOR ALL LEVELS -> 1000000


~~~IT TOOK 6.44264 SECONDS TO PROCESS 2000000 OPERATIONS~~~


THE THROUGHPUT WAS 310432 operations per second
--------------------------------
CUTTLE TREE STATISTICS COMPLETE!
--------------------------------
```