STORAGE AND AGGREGATION FOR FAST ANALYTICS SYSTEMS

A Thesis
Presented to
The Academic Faculty

by

Hrishikesh Amur

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
College of Computing

Georgia Institute of Technology
December 2013

STORAGE AND AGGREGATION FOR FAST ANALYTICS SYSTEMS

To my parents, who continue to show me the way.

ACKNOWLEDGEMENTS

I would like to express my deep gratitude to my advisor, Karsten Schwan. Karsten is encouraging and patient, and always gave me the freedom to pursue my interests. During our meetings, he brought perspective in terms of the utility of an idea, when I would be excitedly lost in its intricacies. He completely eliminated the need to worry about lack of equipment or sources of funding, allowing me to fully concentrate on my work. I am also grateful to Karsten for all the opportunities for research collaborations and internships, that he facilitated through his leadership at CERCS and personally.

Dave Andersen and Michael Kaminsky were my mentors during an internship at Intel Labs, and unofficial advisors for this dissertation. An important thing that I learned working with Dave is that interesting research problems only become visible when systems are pushed to their limits. Dave's questioning during our meetings always pointed towards verification of this in my experiments. Michael was extremely helpful by often making me re-examine assumptions or shaky reasoning, and gave me valuable advice. I am also grateful to them for the time they spent in improving my technical writing, with detailed feedback on research papers drafts.

Greg Ganger was my mentor for an internship at CMU and was instrumental in piquing my interest in distributed storage, and putting me on track for this dissertation. Among other things, Greg taught me how to deliver a talk by critiquing multiple iterations of a talk until I got it right.

Ada Gavrilovska and Matt Wolf provided me with invaluable feedback about my dissertation and during other presentations and informal discussions throughout my time as a graduate student. My early collaborations with Ada helped shape my work, and I enjoyed my free-wheeling conversations on graphs with Matt.

Rich Vuduc provided crucial feedback when there were multiple potential directions for my work, and helped guide me to a fruitful one.

I would also like to express my gratitude to Karthick Rajamani and Freeman Rawson, my mentors during my internship at IBM Research in Austin. As an inexperienced, first-year PhD student, the experience reminded me why I liked working on systems.

I am sure that I would not have chosen to pursue my PhD. had it not been for the excellent experiences that I had, before coming to Georgia Tech., working with Dr. Sridhar Varadarjan, and my first exposure to systems work with Dipankar Sarma and Srivatsa Vaddagiri at IBM's Linux Technology Center.

A large part of the credit for this work goes to the students that I have collaborated with. Ripal Nathuji mentored me during my first project. Jim Cipar answered many questions when I was starting out with working on distributed storage. Vijay Vasudevan critiqued multiple versions of my first conference talk, and also gave me helpful advice on delivering a talk. Thanks to Wolf Richter for discussions on Minni. I also enjoyed working with Bhavani Krishnan and Bharath Ravi. Thanks to Vishal and Shruthi for their prompt and useful feedback on paper drafts.

Chad Hunnycutt made sure that I always had access to the right infrastructure to do my work, and throughout my time at Georgia Tech., Susie McClain ensured that administrative tasks caused minimal distraction.

The many students at CERCS and outside made life as a graduate student bearable and fun. Vishal was my office-mate for the last two years, and was usually my first critic for research ideas. Partha and Mahendra were companions for conversations on topics academic and otherwise. Thanks to Sudarsun for always being ready for another cup of coffee. I tremendously enjoyed our twenty-minute discussions on all the kernel-hacking problems he was working on. Thanks to Alex for his infectious enthusiasm for systems work, and for GPU-related discussions. Thanks to Mukil, Priyanka, Adit, Min, and Chengwei for all the discussions during our reading groups and meetings.

Karthik Raveendran was my house-mate for almost the entire duration of my PhD; I tremendously enjoyed our discussions on books, movies, football and research. Satya, Varun, Deepak, Utsav and Aranya made my two summers in Pittsburgh enjoyable.

My family has been a constant source of support. When I felt that I could not meet a deadline, Sphurti's constant encouragement kept me going, and her understanding and humor made setbacks and rejections easier to handle. My aunt, Shashi, has been like a third parent here in the US. I would also like to thank my grandparents, who remain a huge inspiration to me. Thanks to Boos for being the ultimate stress-buster, and for his reminders to keep life simple. I am incredibly lucky to have my parents for their life-long support and guidance. I think it will suffice to say that if I can follow their path through life, I will be content.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Computing in the last decade has been characterized by the rise of data-intensive scalable computing (DISC) systems. In particular, recent years have witnessed a rapid growth in the popularity of fast analytics systems. These systems exemplify a trend where queries that previously involved batch-processing (e.g., running a MapReduce job) on a massive amount of data, are increasingly expected to be answered in near real-time with low latency. This dissertation addresses the problem that existing designs for various components used in the software stack for DISC systems do not meet the requirements demanded by fast analytics applications. In this work, we focus specifically on two components:

1. Key-value storage: Recent work has focused primarily on supporting reads with high throughput and low latency. However, fast analytics applications require that new data entering the system (e.g., new web-pages crawled, currently trending topics) be quickly made available to queries and analysis codes. This means that along with supporting reads efficiently, these systems must also support writes with high throughput, which current systems fail to do. In the first part of this work, we solve this problem by proposing a new key-value storage system – called the WriteBuffer (WB) Tree – that provides up to 30$\times$ higher write performance and similar read performance compared to current high-performance systems.

2. GroupBy-Aggregate: Fast analytics systems require support for fast, incremental aggregation of data for with low-latency access to results. Existing techniques are memory-inefficient and do not support incremental aggregation

efficiently when aggregate data overflows to disk. In the second part of this dissertation, we propose a new data structure called the Compressed Buffer Tree (CBT) to implement memory-efficient in-memory aggregation. We also show how the WB Tree can be modified to support efficient disk-based aggregation.

CHAPTER I

INTRODUCTION

1.1 Problem

Recent years have witnessed a rapid growth in the popularity of fast analytics systems. These systems require real-time processing of high-volume data streams. For example, Twitter's real-time search targets making a tweet searchable within 10s of creation [26], and Google's Percolator [87] decreased the average age of documents in the search index by a factor of 100 by enabling *incremental* updates to the index. The focus on low latency necessitates re-designing the underlying software stack.

In this dissertation, we focus on two important abstractions that have emerged among the primary building blocks used to build systems for data-intensive computing: (a) key-value storage and retrieval – implement a dictionary data structure, allowing arbitrary values to be associated with keys, and queried by key, and (b) GroupBy-Aggregate – given a set of records, partition the records into groups according to some key, and execute a user-defined aggregation function on each group. Key-value storage is used in a variety of domains including: e-commerce [41], photo storage [20] and caching [5]. GroupBy-Aggregate is used both in batch-processing models like MapReduce [39], and Dryad [62], and in stream-processing systems [11, 10, 68].

However, the increasing popularity of fast analytics systems imposes new and unique demands on these systems. In the case of key-value stores, earlier key-value stores mainly targeted two categories of workloads: low-latency workloads that were typically read-intensive, such as those enabled by memcached [5], and latency-insensitive workloads that allowed efficient batch insertion, such as non-realtime analytics or the earlier versions of Google's MapReduce-based indexing. To support fast-analytics workloads, key-value storage systems must ingest incoming data at a high rate as well as allow analysis codes and/or front-ends to query this data. Thus, handling write-heavy interactive workloads is becoming increasingly important for key-value stores. For example, at Yahoo!, typical key-value store workloads have transitioned from being 80-90% reads in 2010 to only 50% reads in 2012 [94]. The first part of the dissertation focuses on write-optimized techniques for key-value stores.

In the case of GroupBy-Aggregate, high-throughput, incremental aggregation is required by fast analytics runtimes. Existing systems typically use hashtables to store intermediate aggregate data. However, existing techniques for aggregation are: (a) memory-inefficient (i.e. memory required per unique key in aggregate data is high) and (b) techniques to overflow aggregate state to secondary storage such as external sorting [39] and hybrid hashing [43] are unsuitable for incremental aggregation. The second part of the dissertation focuses on techniques to solve both problems.

Moreover, this dissertation makes the important insight that the techniques required to build write-optimized key-value storage systems and fast, resource-efficient aggregators are similar. In particular, *buffering*, used in the context of log-structured systems, is an important technique that enables both systems.

1.2 Thesis Statement

Key-value storage systems and aggregators are important components of fast analytics systems. This dissertation unites these problems and claims that buffering is a powerful, common technique that allows key-value stores, as well as aggregators, to be implemented with high performance and efficiency in terms of (bytes of) memory required and disk bandwidth.

1.3 Thesis Overview

Distributed key-value stores such as BigTable [28] and Cassandra [67] use a cluster of machines, each running a single-node data store. The performance of these single-node data stores is a significant factor in deciding overall system performance. The Log-Structured Merge (LSM) Tree has emerged as the data structure of choice in implementing these systems. However, the LSM Tree inefficiently uses available storage bandwidth. In the first part of this dissertation, we apply the technique of buffering to the problem of building a single-node, write-optimized disk-based key-value store. Our key-value store – the WriteBuffer (WB) Tree – uses buffering to perform disk I/O efficiently resulting in up to 30×-higher write performance compared to LSM Tree-based key-value stores, while providing comparable read performance (using 1-2 I/Os per read using 1-2B per key of memory for indexing).

In the second part of this dissertation, we describe the design, implementation and evaluation of the Compressed Buffer Tree (CBT) – a new in-memory implementation of the GroupBy-Aggregate abstraction that not only supports high-throughput aggregation, but also stores the aggregated data efficiently in memory. The CBT uses compression to reduce memory consumption; to prevent overly-frequent compression and decompression from impacting performance, it uses buffering coupled with *lazy aggregation*. The CBT consumes between 21-42% less memory compared to state-of-the-art hashtable-based aggregators. Next, for datasets where the available memory is insufficient, we extend the CBT to spill its contents to secondary storage such as hard disks or SSD.

1.4 Contributions

This dissertation makes the following contributions:

- **Write-Optimized Key-Value Store**: We introduce a new write-optimized data structure called the WB Tree which provides up to 30× higher write performance than a popular LSM Tree implementation: LevelDB.

- **Memory-efficient GroupBy-Aggregate**: We introduce the Compressed Buffer Tree (CBT), a data structure for stand-alone, memory-efficient GroupBy-Aggregate written in C++; the CBT seeks to reduce memory consumption through compression. To reduce the number of compression/decompression operations (for high performance), the CBT leverages the buffer tree data structure.

- **I/O-efficient External GroupBy-Aggregate**: We modify the WriteBuffer Tree to implement an I/O-efficient aggregator. Unlike the CBT, this aggregator supports random access to partially aggregated results at any time during aggregation.

1.5 Organization

The remainder of the dissertation is structured as follows: Chapter 2 explains the trends that motivate this dissertation; in Chapter 3, we describe the WriteBuffer (WB) Tree, a write-optimized key-value store; in Chapter 4, we demonstrate how techniques used in the WB Tree can be applied to the GroupBy-Aggregate problem. In Chapters 5 and 6, we describe our solutions to in-memory and external aggregation, and finally, we discuss related work in Chapter 7, and conclude with future work in Chapter 8.

CHAPTER II

TRENDS

2.1 Growth of Fast Analytics Systems

Recent years have witnessed a strong interest in the development of systems that can perform analysis with low latency on massive amounts of data in a scalable and fault-tolerant fashion. For example, being able to search "tweets" (short 140-character messages sent by users to their "followers") in a real-time fashion on Twitter has exciting possibilities. In many news-worthy events, such as the covert mission to capture Osama Bin Laden in 2011 [3], tweets relating to the event appeared on Twitter much before being covered by conventional news organizations. The desire for the ability to search and analyze tweets shortly after being produced led to the development of Earlybird [26], a system that allows tweets to be made searchable less than 10s after being produced. Social media, including Facebook "like" streams and uploaded Youtube videos, along with real-time stock market data and sensor data remain drivers for this growth.

The need to reflect newly-produced data in search results also motivated the development of Percolator [87], a system used at Google for processing incremental updates to a large dataset (e.g., a web index). Before Percolator, whenever a portion of the Web was recrawled, the indexing system had to run a series of batch (e.g., MapReduce) jobs on the recrawled pages as well as the entire existing repository because of links between old and new pages. Not only does this approach increase the average age of pages in the index, but updating the index in this fashion is wasteful because the time for each update is dependent on the total index size rather than the number of new pages crawled. Percolator solved this problem by incrementally updating large datasets, such as the web index, resulting in the capability of more frequent updates; for the web index, this resulted in a decrease of more than two orders of magnitude in the average document age in the index.

Many systems also make the observation that much of the “big data” that is created is most valuable for a short duration after creation (e.g. monitoring data). Thus, a need to analyze “data in motion”, rather than “data at rest” in a repository has motivated a mainstream resurgence in interest in streaming systems. Among recent systems, Spark Streaming [106] treats streaming computations as a series of batch computations on small time intervals. It does not replicate data for fault tolerance; instead, data is recovered by tracking the lineage of operations used to build it. Muppet [68] aims to provide a MapReduce-like interface to analyze fast data. Continuous Bulk Processing (CBP) [76], Comet [59] and Spark [105] also provide incremental processing by running new MapReduce jobs on new data; they allow the base repository to be accessed randomly while only mapping new data.

2.2 Fast Analytics Systems are Stateful

An important difference between batch-processing paradigms, such as MapReduce and Dryad, and fast analytics systems is that the latter provide expressive ways to maintain and access state, whereas the former usually limit straightforward access to state. For example, Percolator improves indexing performance precisely because it maintains the current index as state, and allows freshly-crawled pages to be integrated into already-stored state; previously, the index and new pages were simply used as read-only input to generate a completely new index.

An important characteristic of these systems is that, because they are stateful, they use a storage service, implicitly or explicitly, to store this state. The storage service can have varying requirements, depending on the specific system, with respect to consistency model, fault tolerance etc. For example, Percolator requires a storage service that provides replication with strong consistency support; for its storage service it extends BigTable with distributed transactions for stronger consistency guarantees. The storage service can be either tightly coupled with the system runtime (e.g., Spark Streaming) or pluggable (Yahoo!'s S4). Structurally, it typically consists of per-node database or tablet servers [28, 97], along with a distribution components that manages replication, consistency and availability etc. Generally, the tablet servers must support queries and inserts with high throughput and low latency.

Secondly, many of these systems include a stateful online aggregation [60] component. To support low-latency queries that need to aggregate stored data, aggregation cannot be performed on-the-fly on stored data; instead, aggregate results are precomputed. For example, Muppet [68] is a stream-processing system that presents a MapReduce-like interface to process streaming data; it allows online aggregation in the form of user-defined *update* functions that maintain summaries of data in queryable structures called *slates*. Spark Streaming also provides capabilities for incremental aggregation over sliding windows [106].

In this work, we specifically consider designs for the storage and aggregation components.

2.3 DRAM is an increasingly precious resource

In this section, we elucidate our reasons for focusing on memory efficiency in this dissertation. Firstly, we explain how the recent rise of fast analytics systems is renewing interest in processing in main memory. There has been increased interest in using memory-based processing for low latency for data center applications. These techniques include caching [5, 86], storage of metadata in memory to limiting the worst-case number of I/Os [72, 18], database joins [25] etc. Many of these systems are limited by memory capacity and memory-efficient designs of these systems can

Table 1: Amazon EC2 proportional resource costs for April 2012.

EC2 Instance type (size)	Percentage of hourly cost		
	CPU	Memory	Storage
Std. (S)	18%	47%	35%
Std. (L)	16%	44%	40%
Hi-Mem. (XL)	17%	69%	14%
Hi-CPU (M)	40%	20%	40%

process larger datasets in available memory.

In Section 2.3.1, we provide a cost model based on analysis of Amazon EC2 prices to demonstrate that memory remains an expensive component to purchase and power. Finally, in Section 2.3.2, we present evidence to show that the gap between CPU and memory capacities in future systems is projected to increase; this trend motivates techniques such as compression that use CPU to save memory.

2.3.1 Cost of DRAM

To construct a cost model for different resources, we use publicly available data from Amazon EC2. By considering the hourly costs of different VM instances on EC2, along with the amount of resources available in each instance, we can estimate the proportional resource costs for each instance. Our analysis of resource costs in Amazon EC2 from April'12 in Table 1 shows that for most EC2 instances, memory costs already dominate. In other words, with Amazon's standard configurations, it is far cheaper to double the CPU capacity than to double the memory capacity. We detail the analysis used in Section 5.5.2.1.

2.3.2 Increasing CPU-memory gap

In additional to the high cost of DRAM, several reports suggest that this cost of memory (vs. compute) is likely to grow in the future. As one example, Lim et al. project that the gap between the number of cores and memory capacity per socket will continue growing (Figure 1), effectively decreasing the amount of memory available

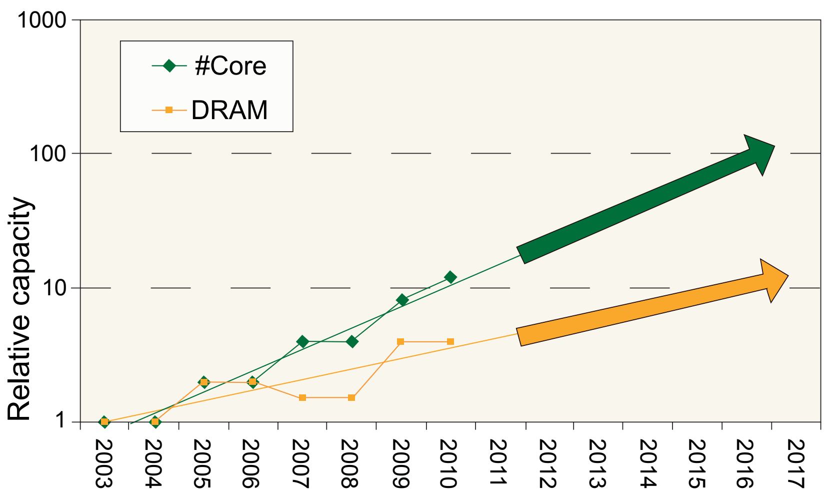


Figure 1: The increasing gap between CPU and memory capacities (reproduced from [88])

per core [74]. As another example, while most of the current Top 10 Supercomputers are in the 0.1-0.5 bytes/flop range [23], predictions for exascale computers include 32-64PB of memory [96, 98], leading to a 100× decrease in the available bytes/flop. This means that techniques that are able to reduce memory usage, at the cost of increased CPU usage without impacting performance are increasingly beneficial.

2.4 Looking Ahead

In Section 2.2, we showed that a key-value storage service is an integral part of a runtime of a fast analytics system. Further, the performance of a distributed key-value store depends significantly on the performance of the per-node store (or the tablet server). In the next chapter, we focus on the design of the per-node store, and show that the key characteristics of a storage service are (a) supporting inserts with high throughput and low latency, (b) supporting gets with high throughput and low latency, and (c) low memory requirements per key. We show that existing solutions fail to meet these requirements and present a new store called the WriteBuffer (WB) Tree.

CHAPTER III

WRITE-OPTIMIZED KEY-VALUE STORE

The WriteBuffer (WB) Tree is a new write-optimized data structure that can be used to implement per-node storage in unordered key-value stores. By providing nearly 30× higher write performance compared to current high-performance key-value stores, while providing comparable read performance (1-2 I/Os per read using 1-2B per key of memory), the WB Tree addresses the needs of a class of increasingly popular write-intensive workloads. The WB Tree provides faster writes than the Log-Structured Merge (LSM) Tree that is used in many current high-performance key-value stores. It achieves this by replacing compactions in LSM Trees, which are I/O-intensive, with light-weight *spills* and *splits*, along with other techniques.

3.1 Introduction

Earlier key-value stores targeted two categories of workloads: low-latency workloads that were typically read-intensive, such as those enabled by memcached, and latency-insensitive workloads that allowed efficient batch insertion, such as non-realtime analytics or the earlier versions of Google's MapReduce-based indexing. However, the increasing demand for real-time results breaks these models. Twitter's real-time search makes a tweet searchable 10s after creation [26]. Google's Percolator [87] and Continuous Bulk Processing (CBP) [76] also seek to perform incremental updates to large datasets (e.g. web search indexes) efficiently (i.e., without having to run a large MapReduce job). To support such functionality, key-value storage systems must ingest incoming data at a high rate as well as allow analysis codes and/or front-ends to query this data. Therefore, handling write-heavy interactive workloads is becoming increasingly important for key-value stores. For example, at Yahoo!, typical key-value store workloads have transitioned from being 80-90% reads in 2010 to only 50% reads in 2012 [94].

In this chapter, we present a new write-optimized data structure, the WriteBuffer (WB) Tree, that provides more than an order higher write performance than other state-of-the-art write-optimized stores while also supporting random access queries. For comparison, to write 64B records, the WB-Tree provides nearly 7× the write throughput of LevelDB [52] along with equal or better read performance.

In previous work, two data structures are popular choices for implementing single-node data stores and databases: B+ Trees are used in systems more similar to conventional databases (e.g. BerkeleyDB [84]). The Log-Structured Merge (LSM) Tree [85], however, has emerged as the data structure of choice in single-node storage for many "NoSQL" systems. Systems in which the per-node storage is provided by an LSM variant include HBase [1], Hyperdex [46], PNUTS [36, 94], BigTable [28], and Cassandra [67].

The B+ Tree is used in when low-latency reads are required. Reads from a B+ Tree typically require a single I/O from disk (by caching the frequently-accessed higher levels in memory). Its drawback is that updates to existing keys are performed by writing the new data *in place* which leads to poor performance due to many small, random writes to disk. The LSM Tree avoids this by performing disk I/O in bulk. It organizes data into multiple, successively larger components (or levels); the first component is in-memory and the rest are on disk. When a component becomes full, data is moved to the succeeding component by performing a *compaction*. Compactions ensure that each component contains at most one copy of any key[1]. Unlike B+ Tree reads, which only check one location on disk, an LSM Tree read might check all components. By protecting components with in-memory filters [52, 94], LSM Trees

[1]Some LSM Tree implementations (e.g. LevelDB) relax this constraint for the first disk-based component for faster inserts.

can provide reads that mostly require only one disk I/O per read.

The drawback of an LSM tree is that its compactions are very I/O-intensive (§3.2.3). Briefly, if $M = \frac{\text{size of component } i+1}{\text{size of component } i}$, a compaction performs $2 \cdot M \cdot B$ bytes of I/O to compact B bytes of data from component i to $i+1$. Since ongoing compactions can stall writes, this can lead to low and bursty write throughput.

WB Trees are a new *unordered* data structure that make two main improvements over LSM Trees. First, WB Trees replace compactions with cheaper primitives called *spills* and *splits*. This relaxes the constraint in LSM Trees that a component can contain at most one copy of each key. This relaxation provides a significant increase in write throughput, as explained in §3.3.2.

The second improvement is a technique called *fast-splitting*. Compactions have two objectives: (a) they ensure that no future compaction becomes *very* expensive (because compactions can block inserts); and (b), they reclaim disk space by deleting outdated records. The nature of the compaction is such that it cannot separate the two objectives. Instead, fast-splitting allows separate mechanisms to be used for (a) and (b). Fast-splitting allows the more expensive but less critical garbage collection to run less frequently. This provides higher and less-bursty write throughput, in exchange for using additional disk space, as explained in §3.3.3.

At a high level, the B+ Tree, LSM Tree and WB Tree approaches can be viewed as occupying a spectrum of increasing write performance as well as increasing degrees of freedom in the location of a given record on disk. A consequence of this is that a naive implementation of random reads in WB Trees can result in very poor performance (owing to more locations to search).

As a solution, after considering various alternatives (§3.3.4), we adopt a technique also used by some LSM Tree implementations [52, 94] and protect each possible location using in-memory Bloom filters. During a read, the filter for each possible location is tested first, and only positive tests result in I/Os. This allows the WB Tree to use just 1-2 I/Os per read, providing similar throughput and latency as LSM-Tree reads, with a memory overhead of about 1-2B/key. Thus, when ordered access to keys is not required, the WB Tree can replace the LSM Tree because it achieves similar read latencies and throughput while offering significantly higher write throughput.

The WB Tree is *unordered*, in that it does not store keys in lexicographical order. Instead, keys are stored in order of hashes of keys. While this decision improves write performance (by avoiding expensive string comparisons) and simplifies index design, the WB Tree only supports random read queries.

More concretely, our contributions are as follows:

- We introduce a new write-optimized data structure called the WB Tree which provides up to 30× higher write performance than LevelDB, a popular LSM Tree implementations.

- We introduce new primitives called *spills* and *splits* to replace compactions, which bias the read-write performance tradeoff towards writes compared to compactions.

- We introduce a new technique called *fast-splitting* that further improves write performance.

- We show that, as with LSM Trees, Bloom filters effectively augment the WB Tree to reduce the number of I/Os required per read to 1-2, and provide read performance equal to that of LSM Trees.

In the remainder of the chapter, the WB Tree is often interchangeably used to refer to both the data structure as well as the key-value store build around the data structure. We provide background including a description of the LSM Tree in Section 3.2. The WB Tree design, along with write and read optimizations, is discussed in Section 3.3. We provide an comparisons with other single-node key-value stores along with a deep-dive into WB Tree performance in Section 3.4.

3.2 Background

3.2.1 Terminology

The tradeoff between read and write performance is a recurring theme in this chapter. The read and write performance of key-value stores depends, to a great extent, on read and write amplification respectively. We define write amplification as:

$$\textit{write amplification} = \frac{\textit{Total I/O performed to write record}}{\textit{size of record}} \tag{1}$$

B+ Tree variants and other in-place update schemes such as external hashing have a write amplification ≥ 1. However, write-optimized stores batch multiple records into a single write leading to an amortized write amplification $\ll 1$.

We use two metrics for read performance: the *worst case number of seeks* and *read amplification*. Read amplification is defined as:

$$\textit{read amplification} = \frac{\textit{Total I/O performed to read record}}{\textit{size of record}} \tag{2}$$

Data structures such as buffer trees [16] also buffer reads allowing read amplification to be $\ll 1$. However, we are only concerned with low-latency reads for which the preferred value for the number of seeks and read amplification is 1.

3.2.2 Write-Optimized Key-Value Stores

In-place update data structures such as variants of the B+ Tree provide low worst-case read latency. By using high fan-outs, B+ Trees require less memory as up to 99% of the data can reside in the leaves [56]. By caching the upper levels of the tree in the page cache, B+ Trees typically require a single I/O for reads. However B+ Trees, like other in-place update structures, suffer from poor write performance especially for small records.

Write-optimized data stores are gaining prominence because, as noted earlier, workloads are increasingly write-heavy, and the relatively high capacity of DRAM in

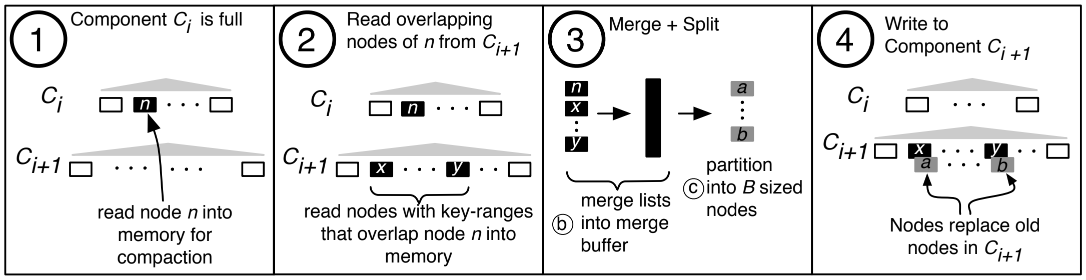

1
Component C_i is full
C_i
n
C_{i+1}
read node n into memory for compaction
2
Read overlapping nodes of n from C_{i+1}
C_i
n
C_{i+1}
x
y
read nodes with key-ranges that overlap node n into memory
3
Merge + Split
n
x
y
a
b
merge lists
(b) into merge buffer
partition
(c) into B sized nodes
4
Write to Component C_{i+1}
C_i
C_{i+1}
x
y
a
b
Nodes replace old nodes in C_{i+1}

Figure 2: Compaction in LSM Tree at component C_i: Triggered when total size of nodes in C_i exceeds a threshold.

modern clusters allows a greater proportion of reads to be satisfied from memory (e.g. using memcached) while writes must be persisted to disk.

Historically, log-structured systems have been used for write-heavy workloads; among these, *insert-ordered* and *key-ordered* log-structured stores may be distinguished. Insert-ordered stores, such as the Log-Structured File System (LFS) [91], FawnDS [15] etc., write data to disk immediately. These have excellent write throughput, but suffer from latency spikes due to garbage collection, have poor scan performance, and require a large amount of memory to support low-latency reads. Key-ordered log-structured stores buffer updates in memory and *sort* them before writing to disk; key-ordered log-structured stores have lower write throughput compared to insert-ordered stores [95]. Examples of key-ordered stores include Buffer Trees [16] and Log-Structured Merge (LSM) Trees [85].

Among these, LSM Trees have typically provided a practical tradeoff between read and write performance. They provide significantly better write performance than in-place update stores, and provide random read performance comparable to B+ Trees with a modest cost in memory. Many state-of-the-art systems including BigTable [28], PNUTS [36, 94], and HBase [1] use variants of the LSM Tree, which we detail next.

Table 2: Comparison of I/O costs for various data structures

Data Structure	Worst-case I/Os required (no caching)	
	`INSERT`	`GET`
B+ Tree	h	h
LSM Tree [85]	$\frac{2(M+1)eh}{p}$	$h \lg \frac{B}{p}$
WB Tree	$\frac{2eh}{p}$ (§3.3.2)	$Mh \lg \frac{B}{Mp}$ (§3.3.4)
Append-to-file	$\frac{e}{p}$	$\frac{Ne}{p}$

$\downarrow$ Increasing `insert` performance

3.2.3 LSM Trees

An LSM Tree consists of multiple tree-like components C_i. The C_0 component is memory-resident and allows in-place updates whereas the remaining components reside on disk and are append-only. The components are ordered by freshness; newest data is present in C_0 and age increases with i. For specificity, we consider an LSM variant that stores each component as a B+ Tree, with the size of each tree node B bytes, and the maximum sizes of the components fixed such that $\frac{\text{size of } C_{i+1}}{\text{size of } C_i} = M$.

For writes into an LSM Tree, records are inserted into C_0 until it fills to capacity. When any component C_i fills to capacity, merges or compactions are performed between components C_i and C_{i+1} as shown in Figure 2. During the compaction, records from some range of keys, represented by node n in the figure, from C_i are read into memory along with records in the same key range from C_{i+1} and merged (steps ①-③). The newly-merged records replace the records for the key range in component C_{i+1} (step ④). Except C_0, the nodes of all other components are not modified in-place. The records are written back as new nodes to disk and the replaced nodes are garbage-collected. By avoiding in-place updates for on-disk components and only performing I/O in large, sequential chunks, LSM Trees achieve significantly better write performance than B+ Trees.

Unfortunately, the compaction operation is extremely I/O-intensive. Recall that each node is sized B; because C_{i+1} has around M times the number of nodes as C_i, the number of nodes in C_{i+1} that contain keys overlapping with keys in node n is also close to M. This means moving B bytes from C_i to C_{i+1} during a compaction requires $(M + 1) \times B$ bytes to be read and the same number written back to disk after merging into C_{i+1}, resulting in high write amplification as shown next.

An inserted record moves progressively from $C_0, C_1, \ldots$ to C_{h-1}, where h is the number of components or "height" of the tree; therefore, the total I/O performed for the record is the sum of the I/O performed at each component. During a compaction from C_{i-1} to C_i, the total I/O performed is $(M + 1) \times B$ bytes read plus an equal amount written. This is amortized across the B/e records in the node being compacted where e is the size of each record. This yields a per-record I/O cost of $(M + 1) \times e$ bytes read plus written. From §3.2.1,

$$\textit{write amplification} \leq \tfrac{1}{e} \sum_{i=1}^{h} \frac{2(M+1)B}{B/e} = 2h(M+1)$$

During an LSM Tree `GET`, the in-memory component C_0 is first searched (recall that the age of records in component C_i increases with i). If the queried key is not found in C_0, then the disk-based components $C_1, C_2, \ldots$ are progressively searched. As a result of the compaction process, the LSM Tree maintains the invariant that there is at most one record for any key in each component. This leads to a worst-case read cost of h seeks. Nodes can be protected with Bloom filters to avoid wasteful I/Os for a modest memory cost [52, 94]. Table 2 shows the I/O requirements (without the page cache) of some data structures. In the table, h is the height of the tree, e is the length of the record, p is the unit size of data movement between disk and memory, B is node size for WB Trees and partition size for LSM Trees, $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$ for LSM Trees or the fan-out for WB Trees, and N is the total number of unique keys.

In this chapter, for discussion, we consider an LSM Tree with partitioned, exponentially-sized levels such as used in LevelDB [52]. We also evaluate a second variant without partitioning in Section 3.4.

3.3 Write Buffer Tree

This section introduces the WB Tree, an *unordered* key-value store optimized for high insert performance while maintaining fast random read access. The tree's key novel elements are its replacement of performing compactions with cheaper spills and splits realized via mechanisms described in §3.3.2. Further, fast-splitting substantially improves write performance, as explained in §3.3.3. The WB Tree includes indexes that limit read amplification to 1-2 I/Os per read (§3.3.4). Finally, garbage collection (§3.3.5), logging and recovery are discussed (§3.3.6).

3.3.1 Overview of the basic WB Tree

The WB Tree exports a simple API: `INSERT(key, value)`, for both insertion and update, `GET(key)`, and `DELETE(key)`. It also supports bulk insertion and deletion for high throughput.

The WB Tree maintains a single root node. The root node is unique in that it consists simply of a single in-memory buffer of size B. Non-root nodes are divided into *leaf* nodes and *internal* nodes. Each non-root node contains one or more *lists* of sorted records on disk; a *list* is similar to an SSTable [28], but does not contain any indexing information. The total size of the lists in a node must be less than or equal to its capacity, B.

The WB Tree uses high fan-out, which means that a large proportion of nodes are leaf nodes. For example, for a fan-out of 256, around 99.2% of the nodes are leaf nodes[2]. A high fan-out helps reduce write amplification by reducing the height of the tree. The write amplification depends on the height of the tree, because as records

[2]Let there be l leaf nodes and let fan-out be $f = 256$, then there are at least $l/128$ nodes in the level above the leaf nodes, $l/128^2$ in the level above that and so on until a single root node. The total number of nodes is therefore $N = l + \lceil l/128 \rceil + \lceil l/128^2 \rceil + \cdots + 1 \leq l \times \frac{1}{1-1/128}$. Therefore, $l/N \geq 0.9921$, i.e. more than 99.2% of the nodes are leaves

progress down the tree, they are read and written at each level.

As mentioned before, the WB Tree uses hashes of the actual keys to route records within the tree, i.e. the hash of the key in each record decides the location of the record in the tree. Therefore, the sub-tree rooted at each node is responsible for a subset of the hash-space. Each node's hash-space is partitioned between the children of the node (i.e. hash-spaces for sibling nodes *do not* overlap).

An empty WB Tree consists of only the root node. To `INSERT` a key-value pair (record) into the WB Tree, the tuple $\langle$`hash, size, record`$\rangle$ is appended to the memory buffer of the root node; *hash* is a hash of the key, and *size* is the size of the record. To `DELETE` a key is to actually `INSERT` the key with a special tombstone value τ, i.e., `DELETE(k) = INSERT(k,`τ`)`. Discussion of the `GET` operation is deferred until §3.3.4.

The two primitives in the WB Tree are *spills* and *splits*. Recall that, in an LSM Tree, when the total size of a component exceeds a threshold, data is *compacted* from that component to the next. In the WB Tree, when the total size of any non-leaf node reaches its capacity of B, it undergoes a *spill*. This operation performs a similar function as the compaction, in that it moves inserted data progressively down the tree. However, it differs in that it simply appends the spilled data to the nodes in the next level of the tree and performs no reading or merging of data that a compaction does. In this manner, a spill performs substantially less I/O than a compaction.

Leaf nodes, instead, *split* when full. Conceptually, a split converts a full leaf node into two half-full leaf nodes, making room to receive further spills from the parent node.

The strict enforcement of a spill or a split when a node reaches its capacity ensures non-bursty insert performance. A long-running spill or split can block insertions, similar to how compactions can block insertions in an LSM Tree. Before we discuss mechanisms for spills and splits (§3.3.2), we describe collapsing, which is used in both

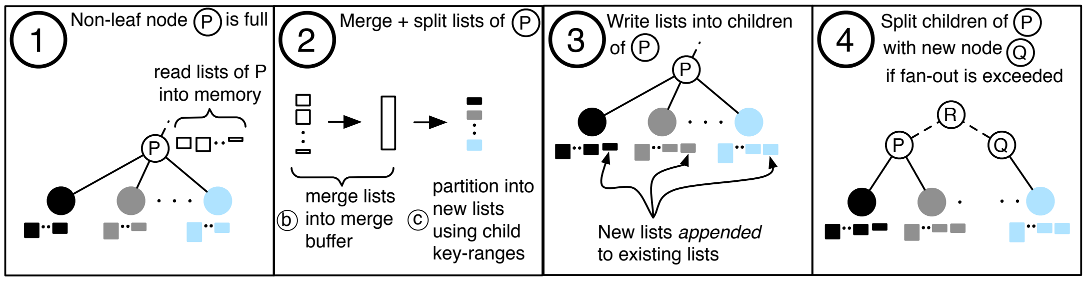

1
Non-leaf node P is full
read lists of P into memory
P
2
Merge + split lists of P
merge lists b into merge buffer
partition into c new lists using child key-ranges
3
Write lists into children of P
P
New lists appended to existing lists
4
Split children of P with new node Q if fan-out is exceeded
R
P
Q

Figure 3: Spilling in a WB Tree at Internal Node P: Triggered when total size of lists in Node P exceeds node size.

operations.

Collapsing Given a sorted list with buffered operations, the list can be collapsed by replacing multiple operations on the same key with a single operation. For example, a `DELETE` appearing after an `INSERT` can be replaced by a `DELETE`, or two `INSERT`s for the same key can be replaced by the later one. Collapsing is important as it allows disk space to be reclaimed from outdated records. Collapsing is intra-list garbage collection; we introduce a new term to differentiate it from the garbage collection of entire lists (§3.3.5).

Collapsing can be performed in a single pass over the list because the list is sorted by hash. However, due to hash collisions, it is possible for colliding keys to be interleaved in the list (e.g. for different keys a and b, the following order might occur: `{`h_a`,INSERT(a,1)}`, `{`h_b`,INSERT(b,3)}`, `{`h_a`,DELETE(a)}` where $h_a = h_b$). These are handled using a separate hashtable for colliding keys whenever a collision is detected. A stable sorting algorithm (we use a fast radix sort because integer hashes are being sorted) ensures that insertion order is preserved for records with the same key.

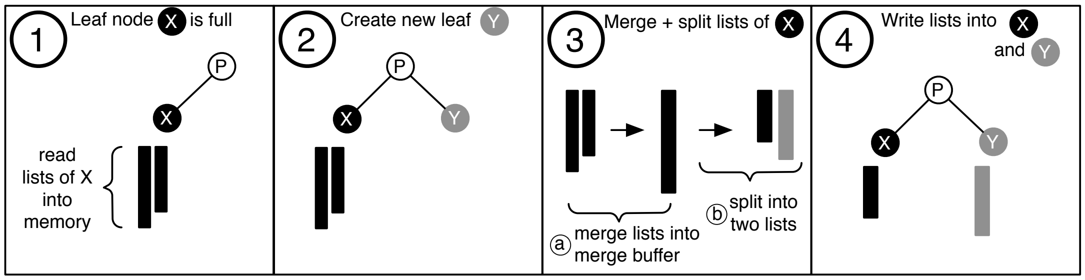


Figure 4: Slow-split operation for leaf node X.

3.3.2 Spills and Splits

3.3.2.1 On Spilling

The spill procedure works differently for root nodes and internal nodes. For the root node, the root buffer is, first, sorted and collapsed. Then, the root node spills into its children by partitioning the in-memory buffer according to the hash-spaces handled by each of its children, and writes each partition as a new list to a child node. If the root is the only node in the tree, it is a leaf and would be split not spilled.

An internal node may contain many lists – one from each spill of its parent. A spill of node P is depicted in Figure 3. The lists of node P (already sorted) are read into memory (step ①). In step ②, the lists are merged into a merge buffer, collapsing the list during the merge. After the merge, the contents of the merge buffer are partitioned according to the key-ranges of P's children and written as new lists in the children in step ③.

After each spill, if the number of children of P is greater (see §3.3.2.2 for when the number of children of a node might increase) than the maximum fan-out allowed, then a new node Q is created and half of P's children are transferred to Q. Node Q is then added as a child to P's parent. If P is the root, then a new root is created, increasing the height of the tree by one (step ④). Node P is empty after a spill, so there is no data buffered in P that requires splitting.

3.3.2.2 On Splitting

For a leaf node, when the total size of all lists reaches the capacity, the node undergoes a split to form a new leaf node, just like a B+ Tree. The new leaf is added as a child to the parent of the split leaf.

One way to perform the split is to read the on-disk lists of the leaf into memory, merge them into a merge buffer, and split the merge buffer into two lists. One list replaces the list being split in the current leaf and the second is added to the new leaf. Both lists are written to disk. This approach, termed a *slow-split*, is shown in Figure 4.

3.3.2.3 Spills and Splits Replace Compactions

Compactions in an LSM Tree and spills in the WB Tree perform the same function, i.e., moving B bytes of data from one level to the next. Compaction, however, requires significantly more I/O. As explained in §3.2.3, during each compaction, to move B bytes from component C_i to C_{i+1}, requires reading $(M+1) \times B$ bytes and writing $(M+1) \times B$ bytes, where $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$. In contrast, both a spill and a split read only B bytes to fetch all the lists from a node into memory and write back B bytes for the new lists in the split leaves (slow-split). Thus, for the WB Tree,

$$\textit{write amplification} \leq \tfrac{1}{e} \sum_{i=1}^{h} \frac{2B}{B/e} = 2h$$

The WB Tree performs a factor of $(M+1)$ less I/O per write. In practice, this replacement yields a 6× improvement in `INSERT` throughput over LevelDB (§3.4.3).

3.3.3 Fast-Splits

As noted earlier, the WB Tree uses a large fan-out to reduce write amplification, leading to a large proportion of nodes being leaves. Because leaf nodes undergo splits and not spills, improving the performance of splits is crucial.

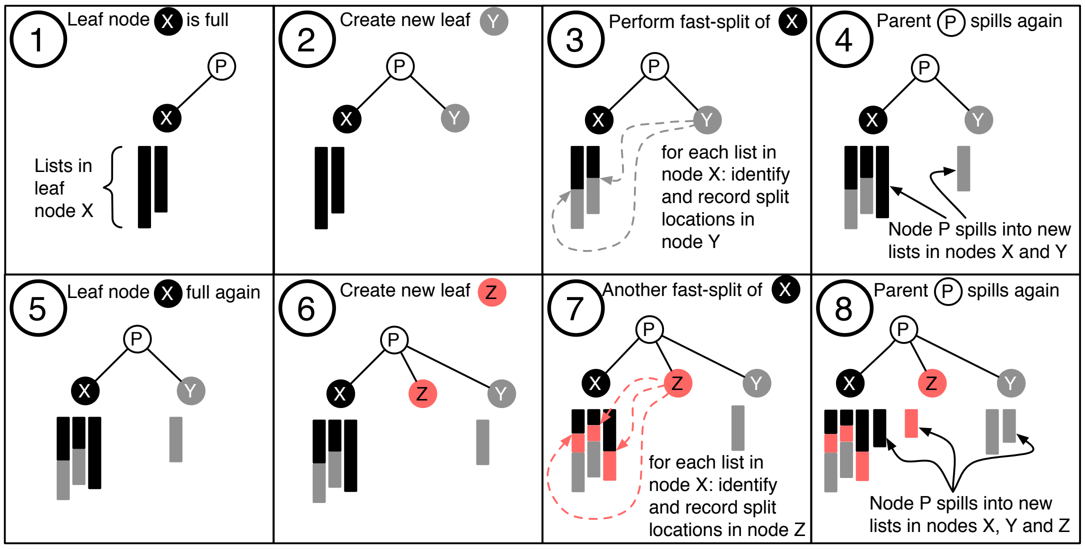

1
Leaf node X is full
P
X
Lists in leaf node X
2
Create new leaf Y
P
X
Y
3
Perform fast-split of X
P
X
Y
for each list in node X: identify and record split locations in node Y
4
Parent P spills again
P
X
Y
Node P spills into new lists in nodes X and Y
5
Leaf node X full again
P
X
Y
6
Create new leaf Z
P
X
Z
Y
7
Another fast-split of X
P
X
Z
Y
for each list in node X: identify and record split locations in node Z
8
Parent P spills again
P
X
Z
Y
Node P spills into new lists in nodes X, Y and Z

Figure 5: Fast-split operation: The figure shows leaf node X splitting two times (steps 3 and 7). There is no data copied during fast-splits, and newly-created leaf nodes only point to the offsets of the partitions in the original files.

Slow-splits, while cheaper than compactions, still incur significant I/O (B bytes to read the lists and B bytes to write back the split halves of the merged list). We provide a simple solution to speed up splitting – *fast-splits*. The basic idea of a fast-split is to avoid reading the leaf from disk; instead, the splitting offset in each list is marked and stored in the new leaf.

Mechanism Figure 5 illustrates the operation of a fast-split on Node X, which is a full leaf node. The median hash in a random list from X is selected as the separator, which is used to split each list in X. Because each list is sorted, finding the median is fast (logarithmic number of seeks) using binary search. Each list in the node is partitioned using the separator value, and one partition assigned to the new leaf (step ③). Unlike slow-split, this process avoids bulk data movement and is fast. When X's parent node P spills for the next time, the new lists are written into the correct nodes (step ④).

Further splits of X are handled similarly. Steps ⑥-⑧ show the creation of new

Table 3: Comparison of splitting schemes

Data spilled	slow-split Leaves	slow-split I/O	slow-split (4) Leaves	slow-split (4) I/O	fast-split (4:1) Leaves	fast-split (4:1) I/O
B	$1 \to 2$	$2B$	1	–	$1 \to 2$	$\log B$
$+B$	$2 \to 4$	$4B$	1	–	$2 \to 4$	$2 \log B$
$+2B$	$4 \to 8$	$8B$	$1 \to 2$	$8B$	$4 \to 8$	$4 \log B$
$+4B$	$8 \to 16$	$16B$	$2 \to 4$	$16B$	$8 \to 16$	$8 \log B$
$+8B$	$16 \to 32$	$32B$	$4 \to 8$	$32B$	$16 \to 32$	$32B$
Total						
$16B$	$63B$		$56B$		$32B$	

leaf Z. Various heuristics can be used to decide how many fast-splits to perform on a node before a slow-split is required. Next, we provide some intuition behind why fast-splitting works.

Intuition Consider what a slow-split accomplishes: (1) when a leaf reaches size B, it converts the multiple lists in the leaf to a single list and collapses the list; and (2) it splits the list into two parts of equal size and assigns one part each to the split leaves. The latter effect crucially allows the parent node to continue spilling into the newly-created leaves. The former effect frees disk space by deleting outdated versions of records during collapsing.

Fast-splitting splits a leaf into two without merging its lists, i.e., it splits the leaf and ensures that insertions do not block, but avoids the more expensive task of reclaiming disk space occupied by outdated records. Instead of the $2B$ bytes of I/O for a slow-split, a fast-split requires only $O(\log B)$ random I/Os. As explained briefly in Section 3.1, fast-splits provide improved write performance by trading off extra disk space. The outcome is superior, consistent and non-bursty write performance.

The advantages of fast-splitting are subtle. It may seem that slow-splitting less frequently might yield the same benefits, but that is not the case. For example, suppose that leaf nodes were $4\times$ the size of other nodes, i.e., $4B$; this would result in

$4\times$ fewer slow-splits of leaves. However, each split would now cost $8B$ bytes of I/O ($4B$ to read and $4B$ to write), which is $4\times$ the cost of slow-splitting a B-sized leaf. Table 3 shows the I/O performed by slow-splits, slow-splits with larger leaves, and fast-splits (4 fast-splits per slow-split). Starting with just 1 leaf, the table shows how each scheme causes leaves to split as data is spilled into the leaves. Using fast-splits performs the least amount of I/O. Using larger leaves for slow-splits requires less I/O, but this causes bursty write performance as `INSERT`s can block, waiting for a large leaf to slow-split.

Along with a favorable choice of system parameters, these factors contribute to a $30\times$ improvement in write performance over LevelDB (§3.4.3). These performance improvements are not free. Compactions perform worse because they enforce the constraint that each component in the LSM Tree, analogous to a level in the WB Tree, can have only one record per key. This bounds the number of possible locations for the key when performing a `GET`. In the WB Tree, this constraint is relaxed, and can lead to significantly higher I/O during `GET`s. Next, we explain how we limit read amplification to achieve read performance in WB Trees that is comparable to that of other key-value stores.

3.3.4 Higher Read Performance with Indexes

We measure read performance using two metrics: (1) worst case number of seeks; and (2) read amplification, which is the total amount of I/O performed to read a record divided by the size of the record. Ideally we should perform reads with a single seek and read amplification of 1 (if record size is less than page size).

Recall from §3.2.3 that in an LSM Tree, a `GET` is performed by successively checking components $C_0, C_1, \ldots, C_{h-1}$, where h is the number of components. Because each component contains at most one version of a key, a `GET` requires at most h seeks in an LSM Tree with no additional indexes.

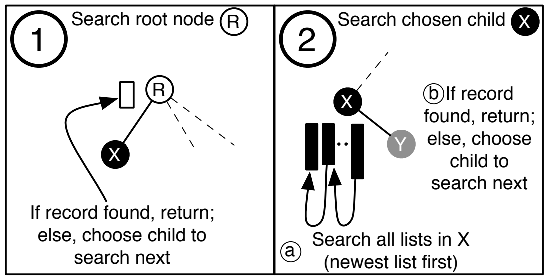

1
Search root node R
R
X
If record found, return; else, choose child to search next
2
Search chosen child X
X
Y
b If record found, return; else, choose child to search next
a Search all lists in X (newest list first)

Figure 6: `GET` operation in an *unindexed* WB Tree.

Table 4: Design alternatives for a List-Selection index

Property	Full-Tree	Per-Node		Per-List
	Hashtable		Perfect Hashing	Bloom filters
Per-key Memory	O(size of key)	O(size of key)	$O(1)$	$O(1)$
Dynamic	Yes	Yes	Requires keys to recompute hash function	Yes
Frequency of updates	For each `INSERT`, spill and split	For each spill and split		
Absent keys return "not found"	Yes	Yes	No	Likely

By contrast, in the WB Tree, each node can contain more than one record per key (a key can occur potentially in each list in a node). In a WB Tree with no indexes, a `GET` is, therefore, performed by starting from the root and proceeding to search along some root-leaf path in the tree (the specific path is dependent on the key). Then, in each node, starting from the last-added list, all the lists have to be searched. Figure 6 shows an unindexed `GET`: In step ①, the root node is searched and if the record is found, it is returned. If not, step ② is invoked recursively until a leaf is reached or the queried key is found. Because lists of all nodes (except the root) are maintained on disk, each `GET` can result in an unacceptably large number of I/Os. Clearly, to achieve our goal of a single I/O per `GET`, an in-memory index is required that maps each key to the list in the tree that contains the most recent version of the key.

3.3.4.1 Index for List Selection

The desired properties of such an index are: (a) it must map a key to the list in the tree that contains the latest record for the key; (b) it must be fast to construct and update, because each spill or split causes multiple updates to the index, and slowing down spills or splits can block insertions; (c) it must be fast to query, because each `GET` potentially queries the index multiple times; and (d) it must be memory-efficient, because a greater number of keys can be indexed in memory, boosting `GET` performance. Design alternatives for such an index are described next and summarized in Table 4.

Full-tree index A straightforward design maintains a dictionary data structure (e.g., hashtable) that maps each key in the tree to the list that contains the most recent record for the key. Unfortunately, this index would require updating not only for each spill and split, but also for each `INSERT` and `DELETE`. `GET`s would have to synchronize with updates to access the index which results in high synchronization overhead.

Per-node indexes reduce synchronization overhead. An index can be maintained by each node that stores the list in the node that contains the latest record for a key for every key in the node. This index only requires updating when a new list is spilled to the node or during a split or spill of the node. Because a node's (hashed) key-space is partitioned among its children, the records for any key can only be contained by nodes on some particular root-leaf path in the tree. In this scheme, a `GET` request would, therefore, have to check the per-node index of each node that occurs on this path.

Hashtables can be used to implement the index, but hashtables are space-inefficient. First, in order to check for collisions, hashtables typically maintain the entire key or a digest (e.g. SHA-1 hash) in memory. Second, closed hashtables (e.g. probing for collision resolution) typically avoid filling up the buckets to capacity to maintain performance, whereas open hashtables (e.g. chaining-based) use extra space for chaining-related data structures. This problem of having to store the key in memory also affects other dictionary data structures such as red-black trees, skip-lists, etc.

Perfect hashing offers a potential solution to having to store keys in memory. Perfect hashing maps the elements from a set S of size n elements to a set of integers with no collisions. Minimal perfect hashing further constrains the size of the set of integers to n. The advantage of this idea is that, because there are no collisions, keys can be stored on disk (instead of memory). However, in order to make a (minimal) perfect hashing scheme dynamic, i.e., supporting insertions and deletions, parts of the hashtable may require rehashing in case of collisions due to newly inserted keys [44]. Rehashing requires keys, which have to be read back into memory using multiple random I/Os. This proves unsustainable for insert-heavy workloads.

Per-list index Another alternative is to extend the idea of a per-node index to a per-list index. Instead of using an index that maps a key to a list, a per-list index stores membership information only. A `GET` would check *all* lists along some root-leaf path in order of age. Recall that each key (more precisely, its hash) maps to some specific root-leaf path in the tree.

Owing to the deficiencies of the first two alternatives, the WB Tree opts for a per-list index. For membership, a per-list index must implement a set data structure for the keys in the list. The data structure must be memory-efficient, without requiring entire keys to be stored in memory. Additionally, it must be fast to construct and query.

Bitmaps using hashes can potentially be used, but become inefficient for lists that sparsely populate the hash-space. Compressed bitmaps are efficient for sparse lists and provide fast queries, but are slow to build. The WB Tree, instead, opts for Bloom filters. Bloom filters are compact (10 bits/key), and fast to build and query. The tradeoff is that a Bloom filter can yield a small percentage of false positives (but never false negatives).

3.3.4.2 Index for List Offsets

Having explained the design of a List-Selection index to select the list that contains the most recent record for a key, we now explain how to find the record within the list itself.

Given that each list in the WB Tree can be large, to maintain low read amplification, for each list, a *List-Offset* index determines the offset within a list at which the record is located. Recall that lists in the WB Tree are sorted by hashes of the keys. This makes it possible to maintain a simple index, called the *first-hash-index* which stores in memory the hash of the key of the first record in each page (e.g., 4kB) of the list.

Searching for a key in a list then proceeds by: Binary search the in-memory first-hash-index to find the page that contains the queried key. Read this page into memory and search sequentially until the target key is found or the end of the page is reached. Binary-searching within the page is not possible because the size of the record is stored with the record on disk.

3.3.5 On Garbage collection

Each list in the WB Tree is backed by a separate file on disk. For an internal node, the files that back lists in that node can be deleted after the node has completed spilling. In the case of leaves that undergo slow-split, the files that back the original lists of the leaf can safely be deleted after the new lists, created by the slow-split, have been written to disk. Fast-splitting complicates garbage collection: the file containing the original list is now pointed to by multiple leaf nodes. We solve this problem by maintaining ref-counts for each file and deleting the file only after no lists reference

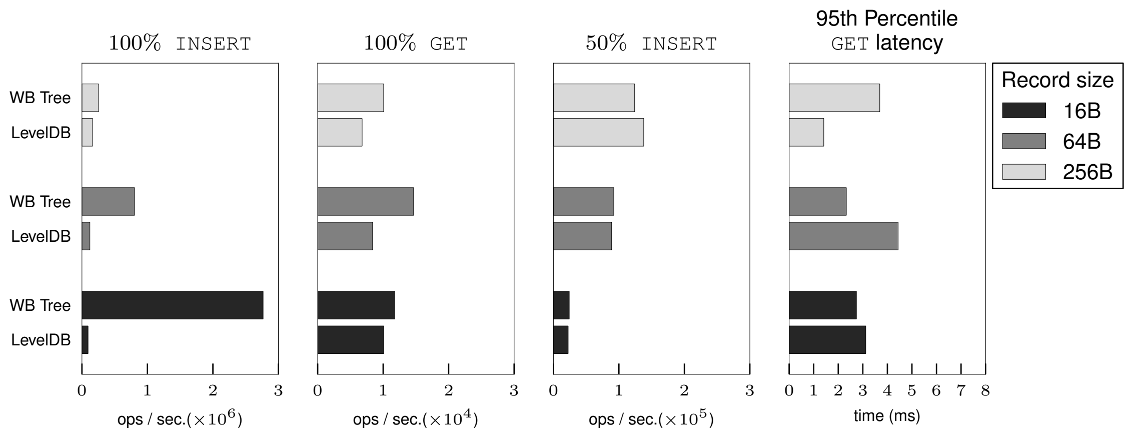


Figure 7: Comparison of the WB Tree with various other key-value stores

it any longer.

3.3.6 Logging and Recovery

The WB Tree uses a write-ahead log to write all `INSERT` and `DELETE` operations to disk before successfully returning to the client. The system also supports synchronous operation, in which `fsync()` is invoked on the log before returning from an `INSERT` or `DELETE` operation. When the root node spills to its children (or splits, forming a new root), its contents are written as lists in the children nodes; the log can be cleared after this. Recovery consists of replaying the contents of the log.

3.4 Evaluation

This section compares the performance of the WB Tree with a variant of an LSM Tree: LevelDB. The experiments use a 12-core server (two 2.66GHz six-core Intel X5650 processors) with 12GB of DDR3 RAM. The disk used is an Intel 520 SSD. We report the median of three runs of each experiment. We use the `jemalloc` [47] memory allocator for all experiments. We use bulk interfaces for insertion. Write-ahead logging is enabled for all systems (`fsync()` is invoked on the log file before returning to the client). The Yahoo! Cloud Serving Benchmark (YCSB) [37] tool is used to generate workload traces, which are replayed in a light-weight workload generator. The dataset used is denoted as (U, R, e, S) where U is the number of unique keys, R is the average number of repeats for each key, e is the record size and S is the total size of the dataset. We use uniformly distributed data for experiments.

3.4.1 Tuning

WB Tree We use a fan-out of 256 and node size of 600MB. As explained in §3.4.3, using relatively high fan-outs and node sizes favors `INSERT` performance.

LevelDB We found that allocating memory to the page cache (instead of a special LevelDB write buffer) improves insert performance. Compression is turned off. The creation of Bloom filters is enabled for fast reads. We use the default values for the overlap factor $M = 10$ and partition size (2MB); the heat-map in Figure 10b shows that, unlike the WB Tree, relatively small values for these parameters provide better insert performance.

3.4.2 Full System Benchmarks

Figure 7 shows the throughput of each key-value store. The datasets used are: D_1: $(2 \times 10^9$, 2, 16B, 42GB), D_2: $(5 \times 10^8$, 2, 64B, 42GB), and D_3: $(10^8$, 2, 256B, 48GB) for 16B, 64B and 256B records respectively. Figure 8 compares the key-value stores in terms of memory use and performance of negative `GET`s. Four important observations stand out:

- `INSERT` throughput in the WB tree is nearly 30× higher than LevelDB for small (16B) records. For 64B records and 256B records, the improvements are 6.6× and 1.5× respectively.
- As the record size increases, the number of `INSERT`s per second achieved by the WB Tree drops as expected. The net amount of data written (= `INSERT`s/sec.

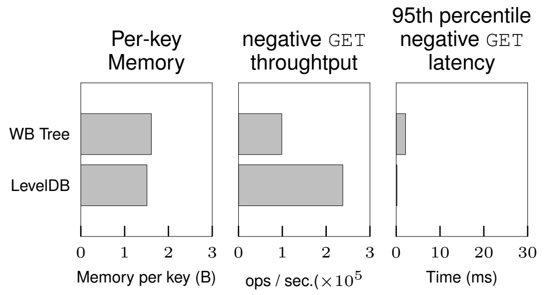


Figure 8: Memory use and negative GET performance of different systems

× record size) actually increases from 45MB/s to 65MB/s. The INSERTs/sec. remains almost constant for LevelDB which seems to indicate that compactions rather than disk bandwidth, are the bottleneck.

- The WB Tree offers equal or slightly higher GET throughput compared to LevelDB, with similar GET latencies.

- For the 50%-INSERT workload, the WB Tree and LevelDB perform similarly. The runtime for this workload is dominated by the GET operations.

Figure 8 shows the memory use of the different systems along with negative GET throughput and latency on dataset D_2. LevelDB and WB Tree require about 1.5B per key. The figure also shows throughput and latency for GET requests for absent keys. LevelDB's latency is lower because it checks fewer Bloom filters. While the mean latency is just 0.2ms (not shown), 95th percentile latency is high.

3.4.3 Write Performance

In this section of the evaluation, we demonstrate the effects of various optimizations on write performance. Figure 9 compares the WB Tree and LevelDB write performance. LevelDB uses default settings for file size (2MB) and overlap factor (10). The Baseline WB Tree uses a node size of 2MB and a fanout of 10. Using spills and splits instead of compactions allows a 6× performance improvement.

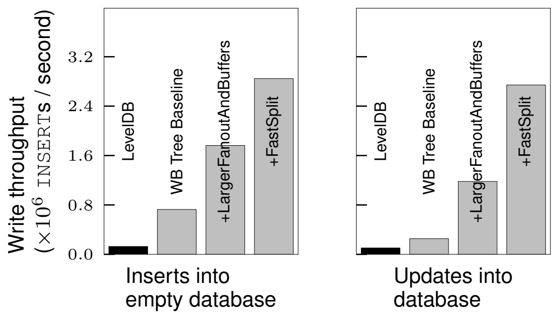
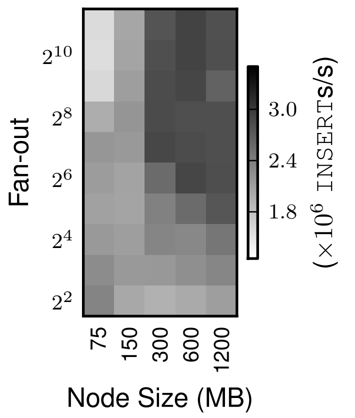
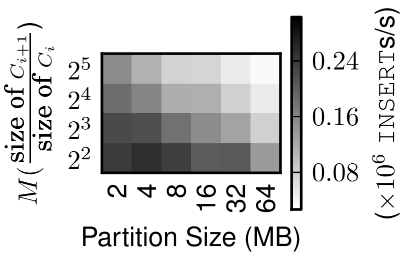


Figure 9: Write performance contributions: Large fan-out/node sizes and fast splitting are both significant.



(a) WB Tree: The upper bound on write amplification is $O(h)$; h is decreased by large fan-outs and node-sizes.



(b) LSM Tree: The upper bound on write amplification is $O(hM)$; h decreases logarithmically with M, favoring small M.

Figure 10: Effect of tree parameters on INSERT performance

Write performance depends on keeping write amplification low. For the WB Tree, reducing the height of the tree reduces write amplification. Figure 9 shows that a careful choice of WB Tree parameters significantly increases write performance.

Figure 10a shows that, generally, increasing the fan-out increases throughput. This is because a high fan-out decreases the height of the tree which leads to lower write amplification. An exception is small node sizes, where high fan-outs lead to mostly small, random writes. Increasing the node size also decreases the height of the tree and generally improves write performance. However, extremely large nodes lead to bursty write performance (not shown).

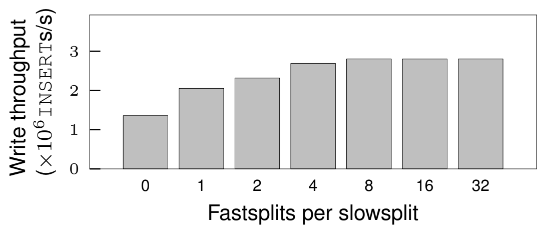


Figure 11: Write throughput increases with an increase in the fastsplit-slowsplit ratio.

Crucially, the same improvements cannot be applied to LSM Trees. Recall from §3.2.3 that the upper bound on write amplification for LSM Trees is $2h(M+1)$ where h is the height of the tree and $M = \frac{\text{size of } C_{i+1}}{\text{size of } C_i}$, is the overlap factor. Increasing the overlap factor reduces the height of the tree, but leads to poorer write performance due to more expensive compactions as shown in Figure 10b. Therefore, using relatively small M and partition sizes works best for LSM Trees.

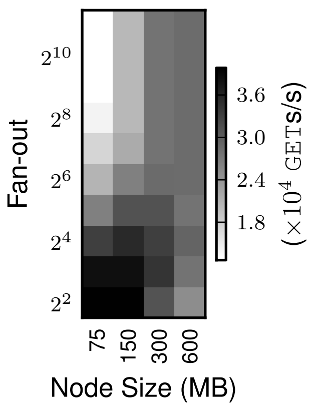
Figure 9 shows that fast-splitting yields a further improvement of nearly 2× in both cases. One possible heuristic to decide when to fast-split is to use a fixed fast-split / slow-split ratio. Figure 11 shows that the write throughput of the WB Tree increases with increasing values of this ratio.

3.4.4 Read Performance

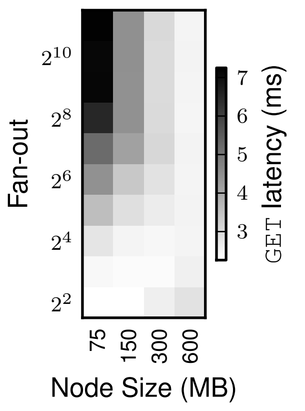
To understand `GET` performance, Figure 12 shows heat-maps for `GET` throughput, 95th percentile `GET` latency and I/Os per `GET`. The following observations can be made:

- Figure 12a and 12b show that the fan-out-node-size combination that maximizes `GET` throughput and minimizes latency is the same, viz. small values of fan-out and node-sizes. Also, the combination that works best for reads, unfortunately, minimizes write throughput (Figure 10a).

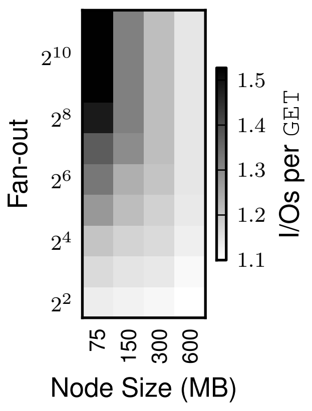
- The trend is partially explained by the heat-map in Figure 12c which shows the number of I/Os per `GET`. Large fan-outs and small node sizes increase the



(a)



(b)



(c)

Figure 12: `GET` performance is helped by low fan-outs and relatively small node sizes.

number of lists per node, which, in turn, incurs more false-positives from the Bloom filters protecting the lists. False positives cause wasted I/Os leading to lower `GET` throughput and higher latencies. An exception is low fan-out with large node sizes: this incurs few I/Os per `GET`, but the large node sizes increase latency from cache misses while searching the List-Offset index.

`GET` performance depends primarily on the number of I/Os performed per `GET` operation. The number of I/Os performed depends on the total number of lists that have to be checked.

3.4.5 Summary

When to use WB Trees The WB Tree's `INSERT` throughput for small records is $5 - 30\times$ higher than LevelDB's, and it has slightly better `GET` throughput with similar latency. The memory cost is a modest 1-2B per key. For large records, the WB Tree provides a more modest improvement of $1.5 - 2\times$ higher `INSERT` throughput.

When to use LSM Trees If a key-ordered store is needed. Also, if memory is insufficient for List-Selection and List-Offset indexes, an LSM Tree will provide higher `GET` throughput.

How to use the WB Tree Figure 10 and Figure 12 show that system parameters trade between `INSERT` and `GET` performance. For high `INSERT` performance, high fan-out and large node sizes must be used; For high `GET` performance, small fan-out and small node sizes must be used. A fast-split / slow-split ratio of 8 or 16 can be used, as higher values can lead to excessive lists in each node which degrades `GET` performance.

3.5 Discussion

Providing ordered access. While many systems require only per-object retrieval, many also benefit from the range query support provided by an ordered store. While the fundamental notion of spills and splits applies naturally to both ordered and unordered stores, extending the design of the WB tree to support ordered access is important future work that will require non-trivial engineering to do well while preserving the structure's high performance.

Bounding worst-case memory per key. Being parsimonious with memory is particularly important when dealing with many small key/value pairs. Here we consider two possible scenarios where the memory used per key can become amplified. We show that the problem is non-existent in the first scenario and provide a solution for the second.

For records smaller than the page size the memory used per key is due, predominantly, to the Bloom filters used in the List-Selection index (the List-Offset index uses only 8 bytes per (4kB) page of records). For each key in a list, the Bloom filter protecting the list requires about 10 bits for a 1% false positive ratio. If a key appears in multiple lists in the WB Tree, then the memory used for that key would be 10 bits *per list*. This *amplification* of memory can occur in two scenarios: (a) repeats in different nodes in the tree, and (b) repeats in different lists within a single node. We consider each case separately.

In the former case, the problem of memory amplification does not arise. For ease of analysis, suppose that copies of all keys in the WB Tree are present in the leaf level (i.e., level 0). As shown by the analysis in Section 3.3.1, with large fan-outs (e.g., 256), the proportion of leaf nodes in the tree is close to 1. This means that the memory used by the non-leaf nodes is small (less than 1%) of that of the leaf nodes.

In the latter case, if the number of lists in a node is l, a key could occur in each of l lists amplifying the memory used by l. To solve this problem, we need an estimate of $\alpha = \frac{\text{number of unique keys in node}}{\text{total number of keys in node}}$. The ratio α provides a measure of memory amplification in the node. If α is 1, then no keys repeat; if $\alpha = \frac{1}{l}$, then all keys repeat l times. If an estimate of α can be maintained, then memory amplification can be bounded by forcing a spill or slow-split on the node when α reaches the desired threshold. To bound the memory amplification to 2, for example, a node is spilled or slow-split whenever α becomes less than 1/2. Next, we provide a method for estimating α.

Suppose that $L_1, L_2, \ldots, L_l$ are sets containing the keys in each list; the number of unique keys in the node is $|L_1 \cup L_2 \cup \cdots \cup L_l|$. Computing the union of the lists is difficult, because when the list corresponding to set L_i is spilled from the parent, all older lists in the node have already been written to disk. To solve this, we propose the use of *cardinality estimators*. K-Minimum Values (KMV) [19] is a cardinality estimator that can estimate the cardinality of a list of elements by inspecting a small fraction of the list. Assuming that a hash function exists that uniformly distributes the elements of the list, intuitively, if there are n elements in the list, the average spacing between the hash values would be $1/n$-th of the range of hash values. KMV uses this idea to maintain a digest of the k-smallest hash values seen in the list; the average distance between these k-successive hashes yields an estimate of the cardinality of the list. For a union of m lists, the digest of each list can simply be merged and truncated to k to obtain a digest for the union. This solution works well for WB Tree lists because the sorted hashes of all keys in a list are already available.

3.6 Summary and Looking Ahead

This chapter presents the WriteBuffer (WB) Tree, a new data structure that forms the basis of a write-optimized, single-node key-value store. State-of-the-art write-optimized key-value stores are typically based on variants of the popular Log-Structured Merge (LSM) Tree. The WB Tree replaces the I/O-heavy primitive in the LSM Tree, the compaction, with new light-weight primitives called spills and splits. Further, a novel technique called fast-splitting is proposed to improve the performance of splits. Using these techniques, the WB Tree's insert throughput is up to 7× higher than LevelDB, a popular LSM Tree implementations, for 64B records. The tradeoff is that unindexed read performance in a WB Tree is worse than unindexed LSM Tree performance. A solution to restore read performance is then proposed: a new set of indexes for the WB Tree allow reads to be performed with 1-2 seeks using less than 2B/key for the index.

In the next chapter, we consider the problem of aggregation. The aggregation service is an important component of fast analytics runtimes. More importantly, the next chapter explains how write-optimized techniques can also be used to make aggregation fast and resource-efficient.

CHAPTER IV

ON THE APPLICATION OF WRITE-OPTIMIZED TECHNIQUES TO GROUPBY-AGGREGATE

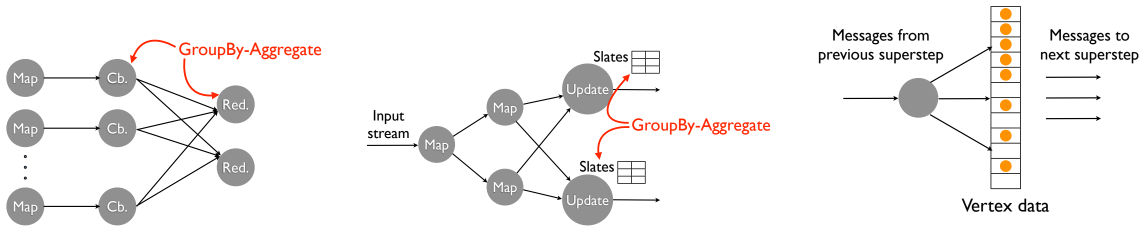
This chapter discusses the GroupBy-Aggregate abstraction and utilization of write-optimized data structures to implement the abstraction. In Section 4.1, we first define the GroupBy-Aggregate abstraction, and discuss its use in different analytics systems. Next, we outline the interface that must be supported by a stand-alone GroupBy-Aggregate implementation. Finally, we discuss desirable properties for an implementation. In Section 4.2, we provide an overview of existing implementations of GroupBy-Aggregate and discuss how these do not support the requirements of fast analytics systems. Finally, in Section 4.3, we describe our approach, which uses write-optimized data structures to implement aggregators.

4.1 GroupBy-Aggregate

4.1.1 Definition

In the GroupBy-Aggregate abstraction, given a set of records, the records are partitioned into groups according to some key, and a user-defined aggregation function is executed on each group. In a canonical example, the number of instances of each word in a text corpus can be calculated by grouping (e.g., by sorting) the text, and counting the number of instances of each word. In this dissertation, we refer to the abstraction as *aggregation* and the implementation as an *aggregator*.

In this work, we consider an aggregator as a stand-alone component that can be used to construct runtimes for different programming paradigms, and, hopefully, be



(a) MapReduce [39] (b) Muppet [68] (c) Pregel [78]

Figure 13: GroupBy-Aggregate in different programming models

re-used across different paradigms. In previous work, aggregators are typically tightly-coupled with the entire runtime system. For example, Hadoop [2], an implementation of MapReduce, uses sort-based aggregation in its combiners and reducers, which is not easily separable or replaceable by other aggregators.

4.1.2 Using Aggregators to Construct Runtimes

An aggregator implements the GroupBy-Aggregate abstraction on a single node. Runtimes for many popular parallel-processing paradigms, such as MapReduce and recent fast analytics systems, can be constructed using one or more aggregators on each node in the cluster. Yu et al. describe how GroupBy-Aggregate lies at the heart of MapReduce, Dryad and parallel databases [104]. Figure 13 shows how GroupBy-Aggregate is used in MapReduce, Muppet, a recent fast analytics systems, and Pregel, a system for graph analytics.

A MapReduce runtime can be built using aggregators to act as combiners and reducers on each node. Each intermediate key-value pair generated by a mapper is inserted into a combiner. Once the map task completes, the combiner is FINALIZEd and the aggregated key-value pairs can be transferred over the network to the reducers.

Muppet [68] is a stream-processing system that presents a MapReduce-like API. Mappers distributed over the network convert incoming streams of data into one or more output streams. Updaters parse streams and maintain summarized aggregate data in memory using *slates* which may be queried in a real-time fashion. For example, an incoming stream of tweets is converted by mappers into streams of tuples comprising tweeter IDs and hashtags from the tweets. Updaters, in turn, process these streams to update a count of tweets per hashtag. Updaters essentially implement the GroupBy-Aggregate operation.

Pregel [78] is a system for distributed graph analytics inspired by the Bulk Synchronous Parallel model [101]. Pregel distributes partitions of the graph among the nodes in a cluster. Programs are expressed as computation to be executed on each vertex in the graph. Programs run in iterations or supersteps, in each of which a vertex can receive messages from other vertices from the previous iteration, change its state, and send messages to other vertices to be delivered in the following iteration. The Pregel runtime on each node is, therefore, responsible for grouping messages received by the node by vertex, possibly aggregating the messages and executing the *vertex-program*. This can be viewed as executing a GroupBy-Aggregate operation during each iteration.

4.1.3 Classification of Aggregators

Yu et al. [104] introduce a taxonomy of aggregators based on three criteria: (a) full or partial aggregation: the former aggregates all possible data, while the latter typically only uses bounded resources and performs best-effort aggregation; (b) accumulator or iterator-based: in the former, accumulators are maintained per key, into which records can be merged; in the latter, all records are first grouped by key and a single aggregation pass is performed by iteratively accessing the records; (c) hash or sort-based grouping. In this work, we consider only full aggregation. The methods we introduce are accumulator-based and use a combination of sorting and hashing, as described in Chapters 5 and 6.

We also consider a further, orthogonal criterion to classify aggregators: (a) In-memory, and (b) External. The former supports very high rates of aggregation, relative to the latter, but is naturally limited by the available memory in the amount of data that it can aggregate (recall that an aggregator only uses the available memory on a single node; distributed aggregation can be performed using multiple aggregators).

Finally, additionally, we also require that the aggregator support *incremental aggregation*. For example, in the case of MapReduce, aggregation occurs either before the shuffle phase (in combiners) or during reduction; in each case, the aggregated results are not re-used after aggregation has completed. But, in recent fast analytics systems, aggregation occurs continuously and queries for aggregated data can occur at any time. We formalize these requirements, next.

4.1.4 Aggregator Requirements

- Fast, incremental aggregation: This corresponds to a high rate of INSERTs and fast finalization.

  Rationale: For batch-processing systems, fast aggregation reduces the time of execution; for online aggregators, faster aggregation on each node reduces the number of nodes required to process incoming events.

- Memory Efficiency: Reduce the amount of memory required per key.

  Rationale: This is most crucial for in-memory aggregators which can handle larger datasets in a given amount of memory, or reduce the memory (and cost) required for a specific dataset. For example, hashtables that use chaining for collision resolution are memory-inefficient as chaining-related data structures, such as pointers, consume memory.

- Efficient I/O: Reduce the amount of I/O performed per record being aggregated.

  Rationale: This is important for external aggregators; since the I/O bandwidth available per node is typically the bottleneck, efficient I/O use improves overall aggregation throughput. For example, using a B+Tree to store aggregate data leads to a high number of random writes to disk.

- Efficient access to aggregated state: permit efficient random and iterative access to aggregation state in memory and on disk.

  Rationale: While MapReduce only requires iterative access, models that support low-latency queries require random access by key.

Next, we specify the API that an aggregator must support.

Aggregator Interfaces The interfaces supported by both types of aggregators are similar:

- INSERT: add a record to be aggregated.
- FINALIZE: prepare aggregated records for reading.
- GET: fetch the current aggregated value of the key.
- REGISTER: register a user-defined aggregation function with the aggregator.

GETs must only be invoked after FINALIZE. Any INSERTs after the last FINALIZE are not guaranteed to be reflected in a following GET. All aggregated records can also be accessed iteratively.

4.2 Overview of Current Solutions

In-memory aggregators typically use hash-based aggregation. For example, Phoenix++ [99] and Metis [79], two shared-memory MapReduce libraries, use specialized hash tables to hold intermediate aggregation state. Pregel, a distributed graph-processing library, also uses hash tables for storing vertex data on each node. For hash-based aggregation, a hash table containing accumulators is maintained in memory and records are hashed by key to the respective accumulators.

External aggregators use sort and hash-based aggregators. In the former, the records to be aggregated are first sorted by key. As records with the same key now appear in sequence, a single pass is sufficient for aggregation. If the aggregated data is too large for available memory, sort-based aggregators use external sorting algorithms, many of which are provably I/O-optimal; for hash-based aggregators, partitioning schemes similar to hash-joins [42] and hybrid hash-joins [43] are used. Traditional implementations of MapReduce uses sort-based aggregation [39, 2]. Databases use sort and hash-based methods for performing aggregation [30, 55].

The idea of online aggregation in databases [60] grew from the high latency associated with queries which performed on-the-fly aggregation of large datasets. An important component of designing systems to support continuous querying is *online data structures* [49]. Vitter discusses a variety of online data structures for external memory [103] including B-trees, hash tables, buffer trees, and spatial data structures. For scenarios where low latency is required, recent fast analytics system (e.g. Muppet and Storm) summarize state in in-memory hash tables.

4.3 Our Solution: Write-Optimized Data Structures

In this work, we propose that write-optimized data structures, used to implement single-node key-value stores in Chapter 3, can be used to implement aggregators.

It may seem, at first glance, that aggregators and single-node key-value stores are quite different with respect to function and implementation. In fact, there are many similarities, which are clear to see if both systems are viewed as black boxes:

- In both cases, records are added into the black box (using INSERT and PUT for aggregators and key-value stores respectively).
- With respect to how two records with the same key are treated, key-value stores may be viewed as a special case of aggregation. During aggregation, two records with the same key combine using a user-specified function to produce a single

record. In key-value stores, the newer record replaces the older record with the same key (since we do not consider delta-update stores).

- External aggregators and key-value stores both require methods to efficiently "overflow" from memory (DRAM) if available memory is insufficient. This refers to managing available memory effectively, as well as performing I/O efficiently.

- External, online aggregators and key-value stores both require random access (using the GET operation) to be supported.

Having outlined the reasons supporting our proposal to use write-optimized data structures to support aggregation, we further propose that variants of the buffer tree are ideally suited for certain types of aggregation that we focus on in this work.

In particular, for in-memory, batch-mode aggregation, we introduce a novel variant of the buffer tree called the Compressed Buffer Tree. Since batch-mode aggregation does not require random access to be supported, buffer trees provide the highest aggregation throughput among alternative write-optimized data structures (Chapter 3). For external, online aggregation, we use a slightly-modified WB Tree. The use of buffering, as before, allows high throughput aggregation. Online aggregation requires random access, which is provided by the WB Tree.

In the following chapters, we present two aggregators:

1. For in-memory, batch-mode aggregation, we present a new data structure known as the Compressed Buffer Tree (CBT) that uses compression for memory efficiency in Chapter 5. While the use of buffer trees for aggregation does not require a big leap of imagination, the novelty in our work is the application of buffering to the reduction of performance overhead from memory compression, which is used to reduce memory use.

2. For external, online aggregation, we present an aggregator based on the WB Tree in Chapter 6. The novelty in this work, lies in the application of the buffer

tree to online scenarios, made possible through the use of the WB Tree which allows low latency reads.

CHAPTER V

COMPRESSED BUFFER TREES

The rapid growth of fast analytics systems, that require data processing in memory, makes memory capacity an increasingly-precious resource. This chapter introduces a new compressed data structure called a *Compressed Buffer Tree* (CBT). Using a combination of techniques including buffering, compression, and serialization, CBTs improve the memory efficiency and performance of the GroupBy-Aggregate abstraction that forms the basis of not only batch-processing models like MapReduce, but recent fast analytics systems too. For streaming workloads, aggregation using the CBT uses 21-42% less memory than using Google SparseHash with up to 16% better throughput. The CBT is also compared to batch-mode aggregators in MapReduce runtimes such as Phoenix++ and Metis and consumes 4× and 5× less memory with 1.5-2× and 3-4× more performance respectively.

5.1 Introduction

This chapter presents the design, implementation, and evaluation of a new data structure called the *Compressed Buffer Tree* (CBT). The CBT implements in-memory GroupBy-Aggregate – given a set of records, partition the records into groups according to some key, and execute a user-defined aggregation function on each group – using lesser memory and providing higher performance than current alternatives.

The CBT focuses on *in-memory* GroupBy-Aggregate (called *aggregation* henceforth) because of the recent need for performing aggregation with not just high throughput, but low latency as well. This trend is driven by *fast analytics* systems such as Muppet [68], Storm [9] and Spark [105], that seek to provide real-time estimates of aggregate data such as customer check-ins at various retailers or trending topics in tweets. For these systems, the conventional model of batch-aggregating disk-based data does not yield the necessary interactive performance. Instead, fast and incremental aggregation on summarized data in memory is required.

The CBT focuses on performing *memory-efficient* aggregation because, unfortunately, recent hardware trends indicate that memory capacity is growing slowly relative to compute capacity [73]. This trend, seen in the context of heavy demand for memory capacity for fast analytics, naturally motivates the development of techniques that can trade off additional CPU use in return for reduced memory consumption.

Memory compression is used by the CBT to exploit this trend, as in previous systems [12, 90]. The CBT efficiently maintains intermediate aggregate data in compressed form, trading off extra compute work for reduced memory capacity use. Direct application of memory compression to existing methods for in-memory aggregation does not work. Consider hashing, a popular technique for fast in-memory aggregation [29, 68]: To maintain, for example, a running count of real-time events, each new event is hashed using its unique key to a hashtable bucket and the stored counter is incremented. Compressing buckets of the hashtable in memory proves ineffective; compressing individual buckets makes compression inefficient as each bucket may be small, and compressing many buckets together as a block causes frequent decompression and compression whenever a bucket in the block is accessed. Hashtables, therefore, are not ideal for aggregating compressed data.

The CBT solves this problem with the insight that accessing compressed data is similar to accessing data on disk. In both cases, (a) there is a high static cost associated with access (i.e. decompression or reading from disk), and (b) an access returns more data than requested (i.e. the block that was compressed or an entire disk page). The well-known External Memory model [13] is developed around precisely these constraints. Techniques based on this model, exploit these constraints by organizing data on disk such that all the data in the block that is returned from an access is “useful” data. This allows the static cost to be amortized across all the data in the block. Most importantly, techniques developed for this model are analogously applicable to compressed memory.

In particular, the CBT leverages the *buffer tree* data structure [16] as the store for compressed aggregate data. The CBT intelligently organizes the aggregate data into compressed buffers in memory such that data that is compressed together also tends to be accessed together. This allows the cost of decompression and compression to be amortized across all the data in the buffer. High compression efficiency is maintained by always compressing *large* buffers. This allows aggregation to be both extremely fast and memory-efficient.

Concretely,

- We introduce the Compressed Buffer Tree (CBT), a data structure for standalone, memory-efficient aggregation written in C++; the CBT seeks to reduce memory consumption through compression. To reduce the number of compression/decompression operations (for high performance), the CBT leverages the buffer tree data structure. The novelty of this work lies in two insights: (a) the buffer tree, originally designed to minimize I/O, can also be used to reduce the number of compression/decompression operations, and (b) the buffer tree can be used to support the GroupBy-Aggregate abstraction, hitherto only implemented using sorting or hashing [104, 55]. This abstraction is a building block for many data-processing models.

- The CBT includes optimizations to specifically improve aggregation performance (e.g. multi-buffering, pipelined execution) and reduce memory (e.g. serialization, column-wise compression), which were not the focus of the original buffer tree work.

- For stream-based workloads prevalent in fast analytics, the CBT uses 21-42% less memory than an aggregator based on Google's `sparsehash` [54], a memory-efficient hashtable implementation, with up to 16% higher aggregation throughput. The CBT can also be used as a batch-mode aggregator. Compared to state-of-the-art batch-mode aggregators from the Phoenix++ [89] and Metis [79] MapReduce runtimes, the CBT uses 4× and 5× less memory, achieving 1.5-2× and 3-4× higher throughput, respectively. Our contribution can be viewed as either allowing existing aggregation workloads to use less memory, or accommodating larger workloads on the same hardware.

We organize the chapter as follows: the rationale behind the choice of the Buffer Tree is explained in Section 5.2. CBT design is detailed in Section 5.3 and implementation details and various optimizations are described in Section 5.4. Aggregator comparisons and an in-depth CBT factor analysis are presented in Section 5.5.

5.2 Aggregator Data Structures

In this section, we first discuss the applicability of the well-known External Memory (EM) model to compressed memory. Next, we explain why write-optimized data structures can be used to implement GroupBy-Aggregate. Finally, we explain the rationale behind the choice of the buffer tree as the basis for a memory-efficient aggregator.

5.2.1 The External Memory (EM) Model

The External Memory model [13] is a popular framework for the comparison of external data structures and algorithms. It models the memory hierarchy as consisting of two levels (e.g. DRAM and disk). The CPU is connected to the first level (sized M) which is connected, in turn, to a large second level. Both levels are organized in blocks of size B and, transfers between levels happen in units of blocks. Note that

Table 5: External Memory (EM) Model analogs for compressed memory

Term	DRAM/Hard disk	Uncompressed/Compressed DRAM
Cache	DRAM	DRAM
Disk	Hard drive	Compressed DRAM
Read	Page read	Block decompress
Write	Page write	Block compress
I/O	Page read/write	Block compress/decompress

the model can be used to describe various combinations of levels: hardware last-level cache (LLC)/DRAM or DRAM/hard disk etc. In this section, we will use the (capitalized) terms *Cache* and *Disk* to indicate the two levels, and the terms *Read* and *Write* to indicate the movement of a block of data from *Disk* to *Cache* and *Cache* to *Disk* respectively.

5.2.2 EM Model for Compressed Memory

Our goal is to improve the memory efficiency of GroupBy-Aggregate by maintaining aggregate data in compressed form. Data access necessitates decompression, so the organization of data plays a critical role in determining the frequency of compression, which affects overall aggregation performance. Our first contribution is to study techniques for this organization of data in the EM model; the EM model models the memory hierarchy as consisting of two levels – *Cache* and *Disk* – that allow bulk transfers between the levels. More precisely, compressed memory is modeled as *Disk* and uncompressed memory as *Cache*. Therefore, a *Read* is a decompression operation and a *Write* is a compression operation. Table 5 shows the analogs for uncompressed/compressed memory compared to the conventional DRAM/disk hierarchy.

The EM model requires that data movement between *Disk* and *Cache* are in blocks of size B. For compressed memory, Figure 14 shows that compression efficiency is high only when moderately large blocks are compressed together. This holds true for many

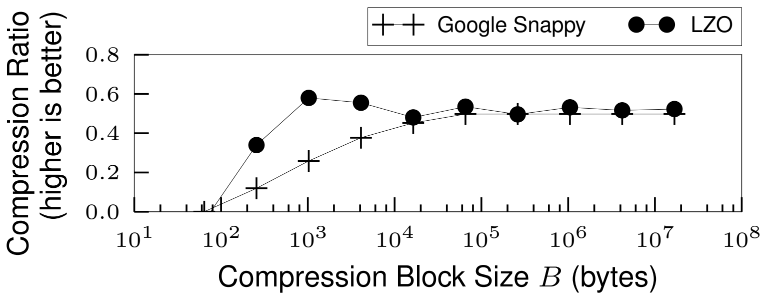


Figure 14: Efficient compression requires large block sizes. Dataset used: Snapshot of Wikipedia compressed using Snappy [53] and LZO [4] compression libraries.

Table 6: Comparison of I/O costs for various data structures in the External Memory model

Data Structure	*Reads* per `get`	*I/O*s per `insert`
B-Tree	$O(\log_B N)$	$O(\log_B N)$
Hashing [48]	$O(1)$	$O(1)$
LSM Tree [85]	$O(\log N \log_B N)$	$O(\frac{1}{B} \log N)^a$
COLA [21]	$O(\log^2 N)$	$O(\frac{1}{B} \log N)^a$
Buffer tree [16]	$O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})^a$	$O(\frac{1}{B} \log_{\frac{M}{B}} \frac{N}{B})^a$
Append-to-file	$O(\frac{N}{B})$	$O(\frac{1}{B})^a$

Increasing `insert` performance ↓

^{a}amortized

popular compression algorithms based on Lempel-Ziv techniques [107, 108], because these work by building a dictionary of frequently-occurring strings and encoding the strings with (the much shorter) references to the dictionary. Only large blocks allow long, repeating runs to be captured.

5.2.3 Data Structure Alternatives

Next, we discuss alternatives for a data structure for storing data on *Disk*. We consider data structures that support two operations: 1) `insert(key, value)` – insert the key, if absent, and associate the value with the key, and 2) `get(key)` – fetch the current value associated with the key. The performance of `insert` and `get` is measured in terms of *I/O*s required per operation. Previous work offers data structures that provide a spectrum of tradeoffs between `insert` and `get` performance under the EM model. Table 6 summarizes these in increasing order of `insert` performance. In the table, B is the block size for data movement between *Disk* and *Cache* (see §5.2.1), M is the size of the *Cache* and N is the number of unique keys.

At one end of the spectrum are B-trees and hashing [48]. Both methods provide low latency `get`s, but modify data on *Disk* in-place. This means that each `insert` operation requires one *Read* and one *Write*. At the other end are unordered log-structured stores (append-to-file in Table 6) such as the Log-Structured File System [91]. They are called *unordered* because they do not order keys in any way (e.g. sort) before *Writing* to *Disk*. Typically, a B-sized buffer in *Cache* is maintained and new key-value pairs copied into it. When full, it is *Written* to a log on *Disk*. This amortizes the cost of the *Write* across all key-value pairs in the block providing the optimal cost of $O(\frac{1}{B})$ *Write*s per `insert`. However, `get`s need to *Read* the entire log in the worst case.

Ordered log-structured stores form the middle of the spectrum. These buffer multiple key-value pairs in the *Cache* and *order* them before writing to *Disk*. Ordered log-structured stores offer much higher `insert` performance compared to in-place update structures like B-trees, and also provide the crucial advantage that key-value pairs with the same key tend to cluster together in the data structure allowing periodic merge operations to be performed efficiently. Examples of ordered log-structured stores include Log-Structured Merge (LSM) Trees [85], Cache-Oblivious Lookahead Arrays (COLA) [21] and buffer trees [16]. Similar to unordered stores, ordered stores amortize the cost of *Writes* across multiple key-value pair aggregations, but, unlike them, periodically merge records belonging to the same key.

LSM Trees maintain multiple B-Trees organized in levels with exponentially increasing size. When a tree at one level becomes full, it flushes to the tree at the next level. COLA are similar to LSM Trees, but use arrays instead of B-trees at each level for slightly slower reads. Both LSM Trees and COLA, with their emphasis on maintaining low latency for *Reads*, aim to serve as online data structures and are used as data stores in key-value databases [28]. The buffer tree has a slightly different goal. It targets applications that *do not* require low latency queries, and only provides good amortized query performance, as shown in Table 6. In return, buffer trees make better use of available *Cache* space, to improve on the `insert` performance of LSM Trees and COLA.

Briefly, the buffer tree maintains an (a, b)-tree with each node augmented by a nearly *Cache*-sized buffer. Inserts and queries are simply copied into the root buffer. When any node fills up, it is emptied by pushing the contents of the node down to the next level of the tree. During the emptying process, multiple `insert`s with the same key can be coalesced.

We leave proofs regarding the performance bounds to related work. Having outlined the tradeoffs in `insert` and `get` performance, we look at what properties are required from a data structure to support GroupBy-Aggregate efficiently.

5.2.4 GroupBy-Aggregate in the EM Model

To understand the requirements of GroupBy-Aggregate, consider implementing GroupBy-Aggregate using a hashtable on *Disk* to store aggregate data. Recall that contents of the hashtable on *Disk* can only be accessed in blocks of size B. To aggregate a new key-value pair (k, v), the current value V_0 associated with k (if any) must be fetched using a `get`; the value v is then aggregated with V_0 to produce V_1, and (k, V_1) `insert`ed. Thus, each aggregate requires a `get` and an `insert`. The problem is that if each key-value pair is much smaller than a block, the remaining data in the block that has been read is wasted. We term this type of *read-modify-update* aggregation as *eager* aggregation, since each new key-value is immediately aggregated into the current value.

An alternative is *lazy* aggregation, where up-to-date aggregates of all keys are not always maintained. Eager aggregation assumes the worst: that aggregated values for any keys may be required at any point during aggregation. In lazy aggregation, on the other hand, for a new key-value pair (k, v) that has to be aggregated, aggregation is simply *scheduled* by `insert`ing the pair. Of course, this requires a lazy aggregator to be *finalized*, where all scheduled-but-not-executed aggregations are performed, before the aggregated values can be accessed.

Lazy aggregation works well in scenarios where the accessing of aggregated values happens at well-defined intervals (e.g. MapReduce and databases where the aggregated results are only accessed at the end of the job and query respectively) or when the ratio of aggregations to accesses is high. For example, lazy aggregation works well for the following scenario: Find the top-100 retweeted tweets, updated every 30 seconds, by monitoring a stream of tweets; the incoming rate of all tweets is high, possibly millions per second, and the aggregated data which includes the top-100 tweets is only accessed once (therefore requiring a finalize) every 30 seconds.

To contrast the requirements of a lazy aggregator with an eager aggregator, it is immediately clear that, for the former, the `get` corresponding to accessing the current aggregated value can be avoided. Therefore, a data structure used to support a lazy aggregator only needs to support fast `insert`s. There is one other requirement: For fast finalization, the data structure *must* be ordered. To see why, consider the case of append-to-file from §5.2.3. Append-to-file (unordered log) supports very fast `insert`s, but at finalization time, the `insert`ed values for a given key can be distributed over the entire log on *Disk*. To collect these, the entire log essentially has to be sorted, which leads to poor finalization performance. Instead, an ordered data structure clusters `insert`ed values with the same key together, allowing periodic partial aggregation. This improves finalization performance. For example, for LSM and buffer trees, buffered values in the higher levels must only be flushed down to the lowest level during finalization, which is significantly faster than sorting the entire log.

Taking these requirements into account, along with conclusions from §5.2.3, it is clear that buffer trees provide the solution we seek: low-cost `insert`s for fast scheduling of aggregation and ordering for efficient finalization. Using this insight, our data structure, the *Compressed Buffer Tree* (CBT), uses the buffer tree to organize aggregate data in compressed memory.

5.3 Compressed Buffer Trees

The CBT is a data structure for maintaining key-value pairs in compressed memory. It allows new key-value pairs to be inserted for aggregation; it merges inserted key-value pairs with existing key-value pairs in compressed memory while reducing the number of compression/decompression operations required. We adopt the CBT for use as a fast, memory-efficient aggregator of key-value pairs that can be used to implement a generic GroupBy-Aggregate operation. This aggregator can be used in a variety of data-processing runtimes including MapReduce and fast analytics systems.

5.3.1 Partial Aggregation Objects (PAOs)

The CBT uses a simple data structure called a Partial Aggregation Object (PAO) to represent intermediate partial results. Yu et al. [104] hint at a similar abstraction, but we make it explicit to help simplify the description of CBTs.

Prior to aggregation, each new key-value pair is represented using a PAO. During aggregation, PAOs accumulate partial results. PAOs also provide sufficient description of a partial aggregation such that two PAOs p^q and p^r with the same key can be *merged* to form a new PAO, p^s. Because different PAOs with the same key provide no guarantees on the order of aggregation, applications must have a `merge` function that is both commutative and associative. As an example, for a word-counting application, the two PAOs: $\langle \texttt{arge}, 2, f : \texttt{count()} \rangle$ and $\langle \texttt{arge}, 5, f : \texttt{count()} \rangle$, can be merged since they share the same key "`arge`", invoking the function associated with the PAOs `count()`, and resulting in $\langle \texttt{arge}, 7, f : \texttt{count}\rangle$. The CBT is programmed by the user by specifying the structure of the PAO (members and function).

5.3.2 The CBT API

The CBT API consists of two mutating operations: `insert(PAO p)` – schedule a PAO for aggregation, and `finalize()` – produce final aggregate values for all keys. Aggregated PAOs can then be read using an iterator, but random key lookups are not currently supported (we briefly address this in Section 5.6).

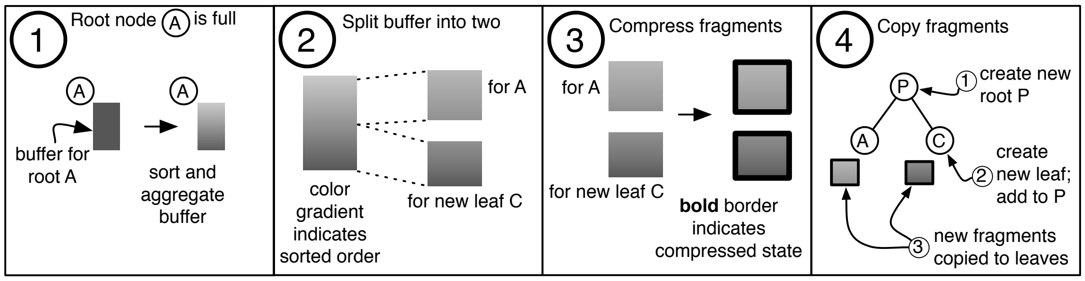
5.3.3 CBT Operation

Like the buffer tree, the CBT uses an (a, b)-tree [61] with each node augmented by a large memory buffer. An (a, b)-tree has all of its leaf nodes at the same depth and all internal nodes have between a and b children, where $2a \leq b$. The entire CBT resides in memory. We term all nodes except the root and leaf nodes "internal nodes." The root is uncompressed, and the buffers of all other nodes are compressed.

Inserting PAOs into the CBT When a PAO is `insert`ed into the CBT, the tuple $\langle$`hash, size, serialized PAO`$\rangle$ is appended to the root node's buffer; `hash` is a hash of the key, and `size` is the size of the serialized PAO. We use the hash value of the key, instead of the key itself, because string comparisons are expensive (excessively so for long keys). Handling hash collisions is explained later. PAOs have to be serialized into the root buffer because PAOs, in general, can contain pointers, dynamic data structures such as C++ vectors etc., and since the eventual goal is to compress the buffer, the PAOs have to serialized into a contiguous chunk of memory. PAOs are copied into the root node until it becomes full. The procedure to empty a full node (Algorithm 1) is summarized next.

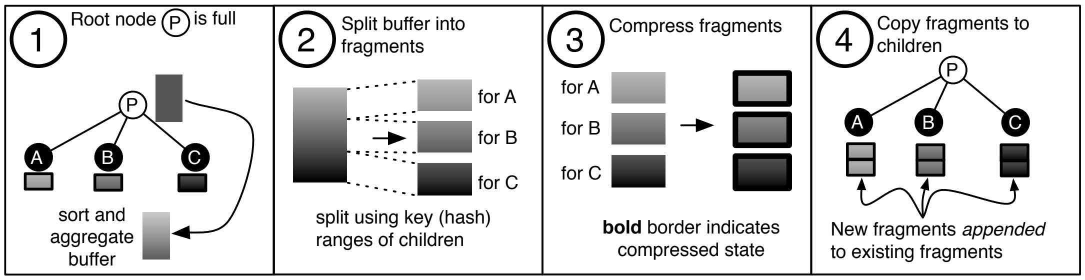
```
# @param N Node which must be emptied
def empty(N):
    if N is the root:
        if N is a leaf:
            # When the tree has only one node,
            # it is both the root and a leaf
            splitRoot() # see Figure 15
        else:
            spillRoot() # see Figure 16
    else:
        if N is a leaf:
            splitLeaf() # see Figure 17
        else:
            spillNode() # see Figure 18
```

Algorithm 1: Emptying a node



1
Root node A is full
A
A
buffer for root A
sort and aggregate buffer
2
Split buffer into two
for A
color gradient indicates sorted order
for new leaf C
3
Compress fragments
for A
for new leaf C
bold border indicates compressed state
4
Copy fragments
P
1 create new root P
A
C
create 2 new leaf; add to P
3 new fragments copied to leaves

Figure 15: Split process for a root node A: The root buffer is sorted and aggregated; the buffer is then split into two and each part compressed. One split is copied into a new node C. The new root P is created as the parent of A and C.



1
Root node P is full
P
A
B
C
sort and aggregate buffer
2
Split buffer into fragments
for A
for B
for C
split using key (hash) ranges of children
3
Compress fragments
for A
for B
for C
bold border indicates compressed state
4
Copy fragments to children
P
A
B
C
New fragments *appended* to existing fragments

Figure 16: Spill process for root node P: The root buffer is sorted and aggregated; the buffer is then split into fragments according to hash ranges of children, and each fragment is compressed and copied into the respective children node.

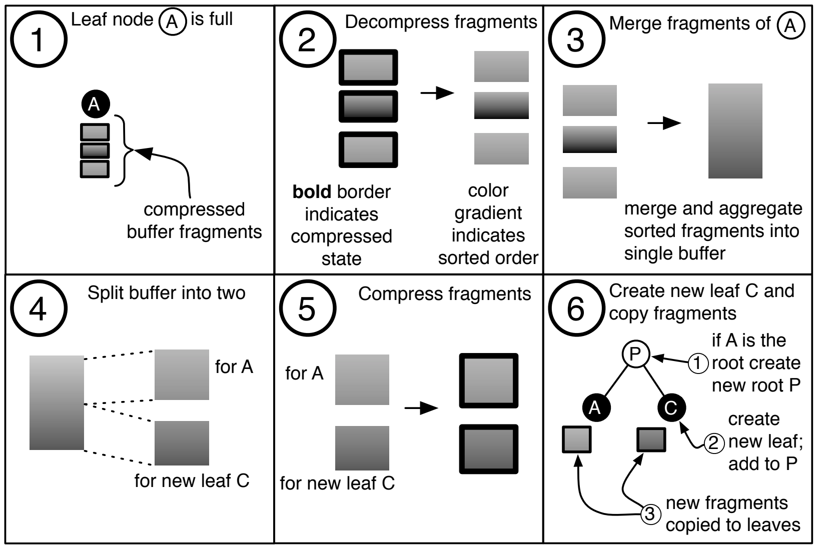
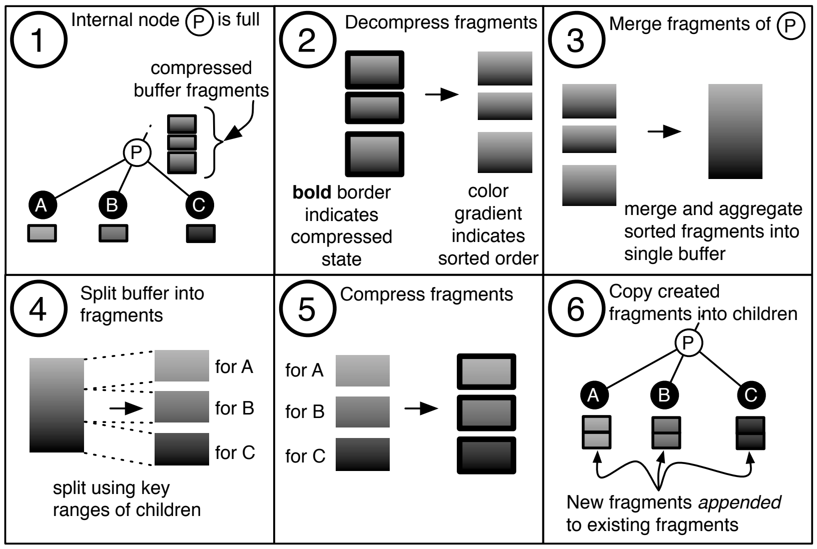

1
Leaf node A is full
A
compressed buffer fragments
2
Decompress fragments
bold border indicates compressed state
color gradient indicates sorted order
3
Merge fragments of A
merge and aggregate sorted fragments into single buffer
4
Split buffer into two
for A
for new leaf C
5
Compress fragments
for A
for new leaf C
6
Create new leaf C and copy fragments
P
A
C
1 if A is the root create new root P
2 create new leaf; add to P
3 new fragments copied to leaves

Figure 17: Split process for leaf node A: The compressed fragments in A are decompressed, merged (each is sorted), and divided into two splits. Both splits are compressed and one split copied into a new leaf node C.

Emptying the Root Node When the root becomes full, the PAOs in the root buffer are sorted by hash, using a fast radix sort since 32-bit hashes are being sorted. After sorting, since PAOs with the same key appear consecutively in the sorted buffer, an aggregation pass is performed. This aggregates all PAOs with a given key in the buffer into a single PAO for the key. The sorted, aggregated buffer is then handled in two possible ways: if the root is also a leaf (if the tree only contains one node), then it undergoes the *split* procedure (Figure 15), otherwise, it undergoes the *spill* procedure (Figure 16).

Emptying Internal and Leaf Nodes Each root spill copies compressed buffer fragments into its children. Any node can spill until one or more of its children becomes full. A full child must first be emptied, before the parent can spill again. Internal and leaf nodes consist of compressed buffer fragments. During emptying, these compressed fragments are first decompressed; since each decompressed fragment is already sorted, the fragments are then *merged* into a single buffer. During merging, PAOs with the same key are aggregated into a single PAO per key. If the node is a



1
Internal node P is full
compressed buffer fragments
P
A
B
C
2
Decompress fragments
bold border indicates compressed state
color gradient indicates sorted order
3
Merge fragments of P
merge and aggregate sorted fragments into single buffer
4
Split buffer into fragments
for A
for B
for C
split using key ranges of children
5
Compress fragments
for A
for B
for C
6
Copy created fragments into children
P
A
B
C
New fragments *appended* to existing fragments

Figure 18: Spill process for internal node P: The compressed fragments in A are decompressed, merged (each is sorted), and split into fragments according to hash ranges of children. Each compressed fragment is copied into the respective child.

leaf, it is *split* (Figure 17), else *spilled* (Figure 18).

Handling hash collisions Ordering PAOs by hash values instead of the keys improves performance and reduces CPU use, but introduces the possibility of hash collisions. Suppose that we wished to aggregate a list of PAOs sorted by hash. In the event of no collisions, a single accumulator can be maintained and PAOs aggregated into it until the hash value changes. After this, the accumulator is written out to a separate aggregated list, and the process is repeated for the next set of PAOs.

Collisions occur rarely, but when they do they can cause PAOs for colliding keys to occur in interleaved fashion. For example, the following sequence (containing hashes, sizes and PAOs) could occur: `{`h_a`,`s_1`,(a,1)}`, `{`h_b`,`s_2`,(b,3)}`, `{`h_a`,`s_3`,(a,2)}`, `{`h_c`,`s_4`,(c,1)}`, where a and b collide, i.e., $h_a = h_b \neq h_c$. Collisions are handled by maintaining accumulators for all colliding keys, and aggregating PAOs into the appropriate accumulator until the hash value changes. For the above sequence, an accumulator for a is first created, but since $h(a) = h(b)$, but $a \neq b$, a second accumulator is created for b. These accumulators are stored in separate hash table. The second PAO for key a is aggregated into key a's accumulator. When the PAO for key c is read, $h(c) \neq h(a)$, so both accumulators for a and b are written out to the aggregated list, and the process continues.

Implementing `finalize()` Due to buffering, CBTs can have multiple PAOs for the same key buffered at different levels in the tree. When the final aggregated value is required, the CBT is finalized by performing a breadth-first traversal of the tree (starting from the root) and emptying each node using the procedure outlined in Algorithm 1. This causes all PAOs to be pushed down to the leaf nodes in the tree. At the end of this process, any possible duplicate PAOs would have been merged.

5.4 Implementation

We implement the CBT as a stand-alone aggregator in C++, allowing it to be integrated into different parallel-processing runtimes. The CBT can either be used as a library or as a service (uses ZeroMQ [6] for cross-language RPC). In either form, it can implement per-node aggregation in a distributed runtimes for MapReduce or stream-processing systems such as Muppet [68] or Storm.

5.4.1 Performance Optimizations

Asynchronous operations Core CBT operations such as compression, sorting, merging and emptying execute asynchronously. Queues are maintained for each of these operations and nodes are moved between queues. For example, when a non-root node is full, it is inserted into the decompression queue. After decompression, it is queued for merging, where its fragments are merged and aggregated. Finally, the node is queued for emptying. All movement of nodes between queues is by reference only, with no copying of buffers involved. Queues also track dependencies to avoid cases such as a parent node spilling into an emptying child node.

Scaling the number of workers Structuring CBT operations as a pipeline of queues allows minimal synchronization during parallel execution (only required when adding/removing nodes from queues). Multi-core scaling is achieved by simply adding more worker threads per queue. We use a thread-pool for worker threads as work is bursty, especially for workloads with uniformly distributed keys where entire levels of nodes in the tree become full and empty within a short interval of time.

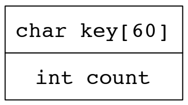
Multiple Root Buffers PAOs are inserted into the CBT by an insertion thread. The insertion thread blocks when the root buffer becomes full and remains blocked until the buffer is emptied into the nodes in the next level. Overall performance depends significantly on minimizing the blocking time for the insertion thread. Using multiple buffers for the root allows insertion to continue while buffers are sorted and aggregated and wait to be emptied. The inserter blocks only when all buffers are full.

Scaling the number of trees Due to the bursty nature of the work, a single tree, though multi-threaded, cannot utilize all CPUs on our node. Therefore, we use multiple CBTs, using hashes of the keys to partition PAOs between trees, such that any given key can occur only in one tree. We find it sufficient to restrict all threads belonging to one instance of the CBT to run on the same processor socket.

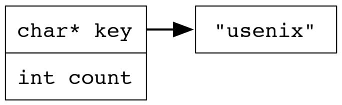
5.4.2 Memory Optimizations

Efficiency through Serialization CBTs improve memory efficiency through compression. As discussed, being able to compress buffers requires that PAOs are serialized into the buffers. Serialization, in fact, also improves memory-efficiency by avoiding many sources of memory overhead that “live” objects require. For example, if an uncompressed in-memory hashtable is used to store the aggregators, there are many sources of overhead:

1. Pointers: Because an intermediate key-value pair is hashed by key, the hashtable



(a) static



(b) dynamic

	Per-entry memory (B)	
Allocator	`unordered_map`	`sparse_hash`
jemalloc [47]	80.01	**58.99**
tcmalloc [51]	78.97	60.35
hoard [22]	70.56	65.14

(c) Per-entry memory consumption for different allocator, hashtable and representation combinations (dataset details in text)

Figure 19: Memory overheads associated with in-memory, hash-based aggregation

implementation stores a pointer to the key, at a cost of 8B (64 bit).

2. Memory allocator: Both keys and values are allocated on the heap, incurring overhead from the user-space memory allocator because: (a) sizes are rounded up to multiples of 8B to prevent external fragmentation (e.g. jemalloc [47] has size classes $8, 16, 32, 48, ..., 128$), (b) each allocation requires metadata; an allocator that handles small objects efficiently (e.g. jemalloc) reduces this overhead.

3. Hashtable implementation: unoccupied hashtable slots waste space to limit the load factor of the hashtable for performance. A memory-efficient implementation such as Sparsehash [54] minimizes this overhead and has a per-entry overhead of just 2.67 bits (at the cost of slightly slower inserts).

Consider the following example: a text dataset with 10^7 unique words of length varying from 6 to 60 characters, distributed binomially ($p = 0.15$, implying a mean length of approx. 10). The intermediate key-value pair can be sized either for the longest word, if known in advance, or can allocate memory dynamically for each word as shown in Figures 19a and 19b respectively. Figure 19c shows the memory used per key-value pair to run word-count on the dataset for the latter representation.

The CBT always stores PAOs in efficient[1] serialized form and avoids "live"-object overheads. "Live" PAOs also use C-style `struct`s instead of C++ objects to avoid virtual table overheads. Compared to hashtables, allocator overhead is minimal because memory is allocated for large buffers, both compressed and uncompressed (typical buffer sizes are hundreds of MB).

Column-Specialized Compression The CBT borrows the idea of organizing data by columns from column-store databases. This enables the use of specialized compression techniques to save memory. Recall that each buffer fragment in the CBT consists of tuples of the form ⟨`hash, size, serialized PAO`⟩; storing tuples column-wise results in three columns, each of which is compressed separately. For example, in each buffer fragment, the hashes appear in sorted form; therefore, we use Delta-encoding to compress the column of hashes. Because many of the serialized PAOs are of similar size, we use Run-Length Encoding to compress the column of PAO sizes. For the column of actual PAOs, we use Snappy [53], a general-purpose compressor, but allow the user to over-ride this with a custom compression algorithm.

5.5 Evaluation

In this section, we evaluate the CBT as an aggregator in streaming and batch modes and compare performance with uncompressed hashtable-based aggregators. Since the CBT trades off additional CPU use for memory savings, we introduce a cost model based on Amazon EC2 resource costs to study whether the saved memory is worth the cost of increased CPU consumption. Next, we perform a factor analysis to understand how CBT parameters and workload characteristics affect performance.

[1]e.g. Google Protocol Buffers (protobufs) serializes integers as *varints* which take one or more bytes allowing smaller values to use fewer bytes instead of the usual 4B

Setup The experiments use a 12-core server (two 2.66GHz six-core Intel X5650 processors) with 48GB of DDR3 RAM. Each processor has 32KB L1 caches, 256KB L2 caches and a 12MB L3 cache. The system runs Linux 2.6.32 with the gcc-4.4 compiler (with -O3 optimization). Each experiment is repeated 12 times and the mean of the last 10 runs is reported with error bars denoting the standard deviation. We use the `jemalloc` [47] memory allocator for all experiments. While `jemalloc` and `tcmalloc` [51] have good multi-threaded performance, the latter shows inflated memory usage as it does not return freed memory to the kernel. Unless specified, we use the CBT with compression enabled using the Snappy [53] general-purpose compressor; a buffer size of 30MB and a fan-out of 64 are also used.

5.5.1 Applications

Recall that user-defined aggregate functions are specified using PAOs (Section 5.3.1). Next, the applications used in the evaluation are described.

Wordcount In this application, we count the number of occurrences of each unique word. We use this application with a series of synthetic datasets to understand the characteristics of the different runtimes and aggregators. The synthetic datasets are characterized by the number of unique keys, a measure of how often the key appears—aggregatability (for an application, we define the *aggregatability* of the dataset as the ratio of the size of the dataset to the aggregated size of the dataset), the average length of the keys and the distribution of key lengths. These datasets use randomly-generated keys, which is the worst-case for compressibility.

N-gram Statistics An n-gram is a continuous sequence of n items from a sequence of text. N-gram counting is useful in applications such as machine translation [80], spelling correction [63] and text classification [27]. This application computes n-gram statistics on 30k ebooks downloaded from Project Gutenberg [7] for different values of n. Each PAO contains a key-value pair: the n-gram and a 32-bit count. Merging PAOs simply adds the count values. The popularity of words in English follows a Zipf distribution, and this application tests the ability of aggregators to handle workloads where the aggregatability and key length is non-uniform across keys.

k-mer counting Counting k-mers, which are substrings of length k in DNA sequence data, is an essential component of many methods in bioinformatics, including genome assembly and for error correction of sequence reads [81]. In this application we count k-mers (for k=25) from Chromosomes 1 & 2 from human genomic DNA available at `ftp.ncbi.nlm.nih.gov/genomes/H_sapiens`. Each PAO consists of the k-mer as the key and a 32-bit count. Merging PAOs simply adds the count values, similar to n-gram. This application tests the ability of aggregators to handle datasets with a large number of unique keys. We use k-mer counting as a representative batch aggregation application.

Nearest Neighbor In this application, we detect similar images using the perceptual hashes [82] of 25 million images from the MIT Tiny Image dataset [100]. Perceptual hashes (PHs) of two images are close (as defined by a similarity metric like Hamming distance) if the images are perceptually similar according to the human visual response. Perceptual hashes are often used in duplicate detection.

This application consists of two MapReduce jobs: the first job clusters images that have the same PH prefix and the second job performs a brute-force nearest-neighbor search among these clusters. For the first job, a PAO consists of a PH-prefix as key and a list of `(image ID, perceptual hash)` tuples, which denote the images in that cluster, as value. Merging two PAOs with the same key combines their image lists. From an input image and its hash (e.g. A, 563), a PAO is created whose key is a prefix of the PH (e.g. 56) and whose value is the image's file name. Therefore, PAOs for images with the same prefix (e.g. 561, 567), which by definition

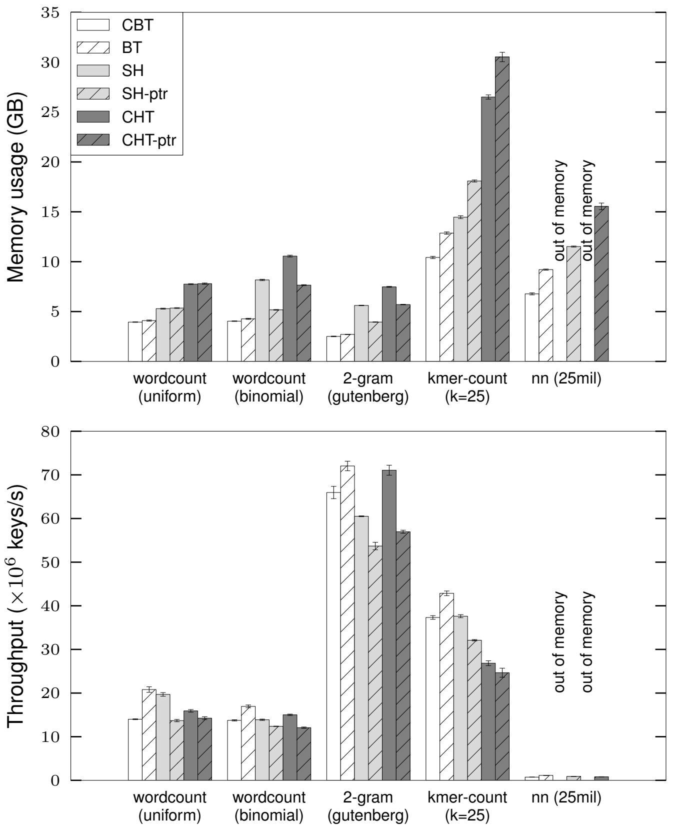


Figure 20: Comparison of various aggregators

are perceptually similar, can be merged by merging the file lists from the PAOs. This job has an expensive reduce function and large PAOs.[2] The second job generates all possible pairs of images from each cluster; the key is the image ID and the value is a `(neighbor ID, distance)` tuple. These are merged by choosing neighbors with the least distance.

[2]This basic method does not find images whose hashes are close by Hamming distance but differ in higher-order bits (e.g. 463 and 563). Therefore, we repeat the process after rotating the PH values for each image (635 and 634 share the same prefix). This works because the Hamming distance is invariant under rotations.

5.5.2 Comparison with Hashtable-based Aggregators

Workload generator To model a stream-based system, we use a multi-threaded workload generator that generates streams of application-specific PAOs that are inserted into the aggregator for grouping and aggregation. To generate the workload, the workload generator can either use an input dataset file or generate randomized data according to input parameters. We run the workload generator on the same machine as the aggregator and link the aggregator libraries into the workload generator. For all experiments, we find that using four generator threads is sufficient to saturate all aggregators.

The workload generator also periodically *finalizes* the aggregators. For hashtable-based aggregators, this has no effect since they always maintain up-to-date aggregated values for all keys. CBTs, however, can have multiple PAOs per key buffered, so finalization causes these to be merged to a single PAO per key.

Hashtable-based Aggregators For comparison with the CBT, we also implemented two other stand-alone hashtable-based aggregators: 1) *SparseHash (SH)*, which permits only serial access but uses Google's `sparse_hash_map`, an extremely memory-efficient hashtable implementation, and 2) *ConcurrentHashTable (CHT)*, which allows concurrent access using the `concurrent_hash_map` from Intel's Threading Building Blocks.

Unlike the CBT, which maintains intermediate key-value pairs in compact, serialized form before being compressed, SH and CHT maintain each (unique) intermediate key-value pair unserialized. We consider two representations for the intermediate key-value pairs: (a) statically sized, where the intermediate key-value pair is stored in-place in a statically-sized C++ `struct`, and (b) dynamically sized, which uses pointers. The former approach (called SH and CHT respectively) requires the `struct` to be sized to the largest key-value pair, but avoids heap allocation overheads, whereas the latter approach (called SH-ptr and CHT-ptr respectively) allocates the exact amount of memory required for each intermediate key-value pair. For the CBT, we also show performance with compression disabled (BT).

A single instance of each aggregator uses a different amount of CPU (CBT uses approx. 4 cores, SparseHash is serial, and ConcurrentHashTable uses all 20 cores). For fair comparison, we use the same amount of CPU overall by using multiple instances for the CBT and SparseHash and partitioning the PAO stream between the instances (e.g. by using a hash function on the PAO key). Therefore, we use 5 instances of the CBT and 20 instances of SparseHash.

First, we compare the aggregators with finalization performed only once at the end of the input dataset. This is equivalent to batch-mode execution. Figure 20 shows these results, and we make the following observations: (a) For all applications, CBT consumes the least amount of memory; (b) the performance of BT is better than the alternatives (except for the application wordcount(uniform) which is a special case with words of exactly the same length); (c) BT always offers better throughput than CBT; (d) CBT consumes significantly less memory than BT if the dataset is compressible: the randomized text and ebook datasets are not highly compressible, but the k-mer and nearest-neighbor are. Many real-world datasets, such as URLs, DNA sequences etc., are compressible; finally, (e) for the hashtable-based aggregators, no intermediate key-value pair representation (static or dynamic) consistently outperforms the other.

As the frequency of finalization increases, hashtables remain unaffected, but CBT performance decreases. Figure 21 shows the effect of finalization frequency on degradation of aggregation performance for two applications: WordCount (Binomial) and 2-gram (Gutenberg). It can be seen that while frequent finalization (once every 5s) can lead to high degradation, more realistic inter-finalize intervals lead to between 10-20% of degradation in aggregation throughput.

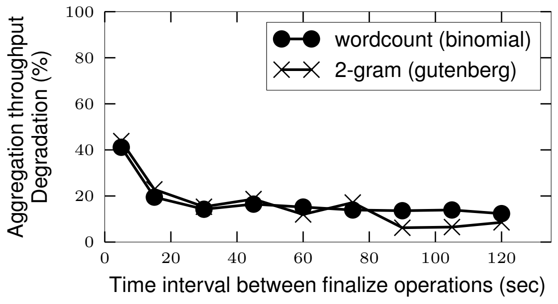


Figure 21: Effect of time interval between finalize operations on aggregation throughput

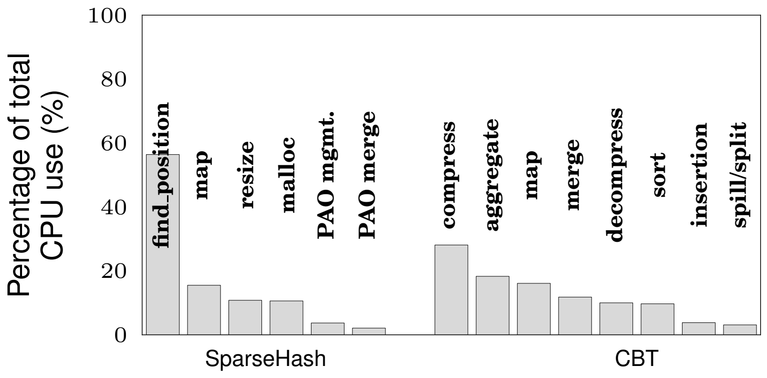


Figure 22: Comparison of CPU use by SparseHash and CBT

Finally, intuitively, it seems that hashtable-based aggregators should always outperform CBTs, given that aggregation using CBTs entails compression, sorting, etc. However, especially for memory-efficient hashtables, the comparison is hard to intuit because aggregation using hashtables involves overheads not present in the CBT. Figure 22 shows the breakup of CPU use for operations involved in aggregation using both hashtables and CBTs. We use 20 instances of SparseHash and 4 instances of the CBT for aggregation resulting in equal total CPU use. For SparseHash, `find_position` finds an unassigned index in a closed hashtable, `map` consists of map-side processing (tokenizing of keys etc.), `resize` resizes the hashtable when full causing rehashing of keys, `malloc` totals overheads involving memory allocation, `PAO mgmt.` measures the overhead of creation of PAOs for insertion into hashtable, and `PAO merge` measures the merging of PAOs with the same key. For CBT, each bar refers to the respective operation performed on buffer fragments in CBT nodes.

For clarity, we briefly explain how SparseHash works: The hashtable is implemented as an array of fixed-sized "groups". Each hashtable operation (e.g. insert, find) is passed on to the group responsible for the index returned by the hash of the key. Each group consists of a bitmap, that stores whether a particular index in the group is assigned or not, and a vector that stores values for assigned indices **only**. This allows the hashtable to use less memory (2.67 bits) for each entry, wasting little memory for unassigned indices. The savings in memory result in slower inserts, as indicated by the high CPU use of `find_position`. SparseHash uses quadratic probing for collision resolution; during an insert, multiple string comparisons might be performed along the probe sequence. Like other hashtables, SparseHash also requires resizing when the load factor (for a closed hashtable, the proportion of occupied indices) becomes too high. Sparsehash also incurs higher allocator overhead than regular hashtables, as unassigned indexes are not pre-allocated and have to be allocated on inserts.

Table 7: Dataset parameters for applications

Application	Unique keys ($\times 10^6$)	Avg. repeats
wordcount (uniform)	100	10
wordcount (binomial)	100	10
2-gram (gutenberg)	105	80.5
k-mer (genomic)	394	6.1
nn (25mil)	116	5.3

5.5.2.1 Cost model

Estimating the cost-benefit of using the CBT requires a cost model for the resources it consumes. The model must account for the total cost of operation including purchase and energy costs. Instead of synthesizing a model from component costs, inspired by an idea in a blog post [75], we analyze the publicly available pricing of different Amazon EC2 instances, which have varying amounts of CPU, memory and disk resources, to estimate the unit cost of each kind of resource as priced by Amazon as alluded to in Section 5.1. We describe the derivation next.

Let A be an $m \times 3$ matrix with each row containing the number of units of CPU, memory and storage available in a type of instance. There is one row for each of the m different types of EC2 instances (e.g. Standard Small, Hi-memory Large etc.). Let b be an $m \times 1$ matrix representing the hourly rates of each kind of instance, and let $x = [c, m, d]^\top$ be the per-unit-resource rates for CPU, memory and disk respectively. Solving for x in $A.x = b$ using a least squares solution that minimizes $\|b - A.x\|^2$ yields the per-unit-resource costs. Using prices from April 2012, this yields hourly costs of 1.51¢ per ECU (Elastic Compute Unit), 1.93¢ per GB of RAM and 0.018¢ per GB of storage. Table 8 shows the corresponding costs predicted by our model along with the benefits of using the CBT over its alternatives.

Table 8: Application costs using Amazon EC2 cost model

Application	Total cost (¢)						Savings
	CBT	BT	SH	SH-ptr	CHT	CHT-ptr	
wordcount (uniform)	0.43	**0.28**	<u>0.31</u>	0.43	0.50	0.57	9.6%
wordcount (binomial)	0.41	**0.31**	0.51	<u>0.46</u>	0.64	0.65	32.6%
2-gram (gutenberg)	0.53	**0.43**	0.70	<u>0.67</u>	0.86	0.92	35.8%
k-mer (genomic)	**0.53**	0.56	<u>0.68</u>	0.94	1.50	1.84	22%
nn (25mil)	**0.15**	0.18	-	<u>0.25</u>	-	0.33	40%

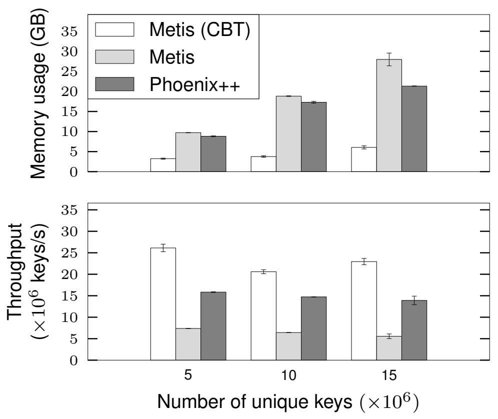


Figure 23: Comparison of CBT with aggregators in Metis and Phoenix++

Batch-mode comparison with MapReduce Aggregators The CBT can also be used as a batch-mode aggregator. For comparison, we replace the default aggregator in the Metis [79] MapReduce framework, which uses a hash table with a B+ Tree in each entry to store intermediate data, with the CBT. We use a set of synthetic datasets with an increasing number of unique keys to compare the growth in memory of the independent aggregator pipeline with the partitioned aggregator pipeline.

The number of mappers (and aggregators) for (default) Metis and Phoenix++ is set to 24, which is equal to the number of logical cores on our system. Metis and Phoenix++ performance scales linearly up to 24 logical cores, and both systems are able to utilize all available cores. For Metis-CBT, we use only 5 instances of the CBT, with each instance configured with 16 worker threads, as this is able to saturate all cores on the system. Figure 23 shows that Metis with CBT outperforms Metis

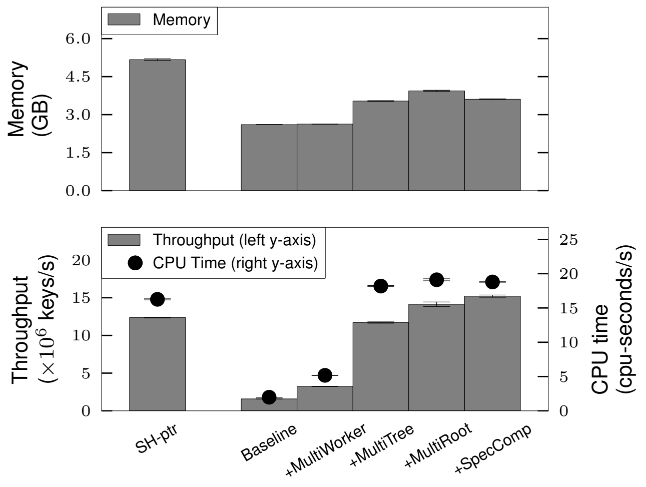


Figure 24: Effects of various design optimizations on CBT performance and memory consumption

and Phoenix++, while requiring up to 5× and 4× less memory. In this experiment, all systems sort final results by count and display the top 5 words. The synthetic datasets consist of an increasing number of unique keys, with average key length 16B, and each key repeating 1000 times. Recall that simply compressing the hashtables used in Metis or Phoenix++ will not yield the same benefits, since each hashtable entry is typically too small to yield significant memory savings.

5.5.3 CBT Factor Analysis

In this section, we use microbenchmarks to evaluate the performance of the CBT for varying workload characteristics and the effect of different CBT parameters. We first evaluate the impact of CBT design features and configuration parameters on aggregation performance and memory consumption. Finally, we evaluate the effects of workload characteristics.

5.5.3.1 Design features

We analyze the performance of the CBT by considering its features in isolation. For each feature, we show its incremental impact on aggregation throughput and memory consumption in Figure 24. Gains/losses are relative to previous version and not to the baseline. For reference, the memory use and throughput of the SparseHash-dynamic (SH-ptr) aggregator is also shown. A synthetic dataset with 100 million unique keys, each repeating 10 times, with binomially ($p = 0.15$) distributed key-lengths between 6 and 60B, is used.

The baseline CBT consists of a single instance of the CBT. It has 4 worker threads: one each for compression, sorting, aggregation and emptying. The basic CBT consumes about 50% less memory compared to SH-ptr. It avoids overheads associated with allocating small objects on the heap (by always allocating large buffers) and avoids storing pointers for each PAO (by serializing the PAO into the buffer) in addition to compression. Aggregation throughput of the baseline CBT is about 87% less than that of SH-ptr.

By increasing the number of worker threads to 16, CBT parallelism increases (overall CPU use increases 2.6×) and improves throughput by 1.5×. Due to the burstiness of the work generated by the CBT, adding more CBT instances allows full use of the multi-core platform. CPU use increases 3.5×, boosting aggregation throughput by 3.6×. Per-instance static overhead increases the memory use by 1.3×.

Using multiple root buffers ($n = 4$ in this case) allows insertion to continue even when the root is being emptied. "MultiRoot" increases throughput by nearly 21% at the cost of 11% additional memory (increasing the number of buffers beyond 4 did not increase performance enough to justify the increase in memory use). Finally, specialized compression for each column, "SpecComp", which uses delta encoding for the hashes-column, run-length encoding for the sizes-column and Snappy for the PAOs, improves the effectiveness of compression, reducing memory use by a further 8%.

Next we consider how the performance and memory consumption of the CBT depend on system parameters such as the node buffer size and the tree fan-out.

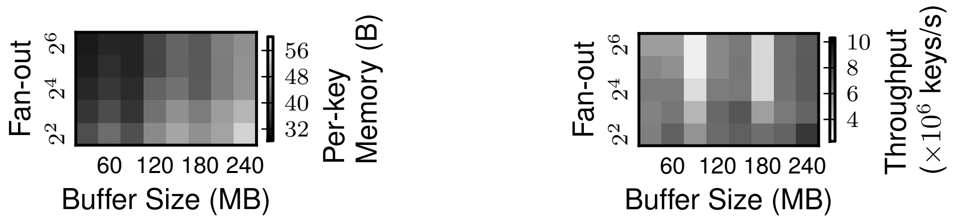


Figure 25: Effects of tree fan-out and buffer size on CBT performance and memory consumption

5.5.3.2 CBT Parameters

Variation with buffer size and fan-out of CBT Although the CBT avoids many overheads associated with hashtable-based aggregation, it incurs memory overhead when serialized PAOs with the same key occur at multiple levels of the tree as a result of lazy aggregation.

This overhead depends on certain CBT parameters: we provide intuition for the dependence of both memory consumption and aggregation performance on these parameters. Figure 25 shows heat-maps for memory use and aggregation throughput for variations in buffer size and fan-out of the tree. Darker colors imply lower memory overheads and higher aggregation performance respectively and vice versa.

In general, increasing buffer size increases aggregation performance, but also increases memory overhead. The reason for this is that larger buffers allow less frequent spilling (improving performance), but provide a greater opportunity for keys to repeat (increasing memory overhead).

The trend with fan-out is similar: increasing fan-outs decrease (colors lighten) memory overhead as well as aggregation throughput. This is because an increase in fan-out results in a decrease in the height of the tree. Shallower trees provide less opportunity for keys to repeat at different levels of the tree (decreasing memory overhead), but also lead to more frequent spilling (decreasing performance).

Therefore, adjusting the buffer size and fan out when configuring the CBT allows

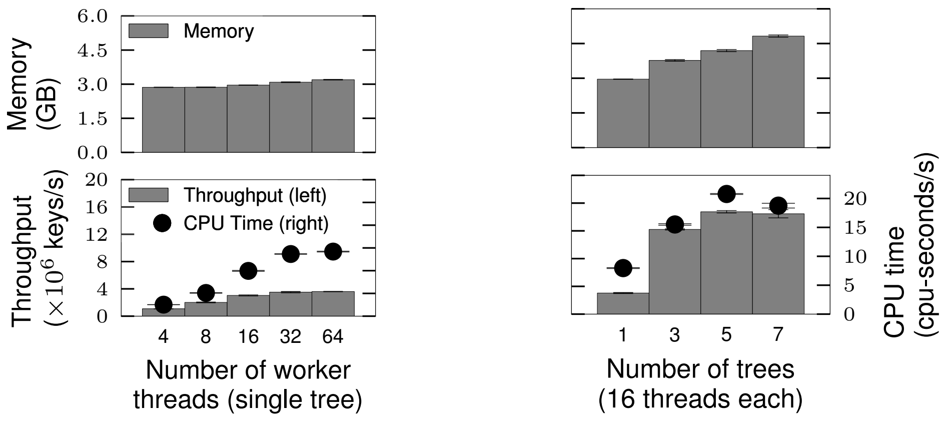


Figure 26: Scaling aggregation throughput

the user to trade off throughput for memory efficiency, or vice versa, as desired.

Scalability Figure 26 shows how the memory use and throughput of the CBT scales with an increasing number of worker threads (left), and by partitioning keys across an increasing number of trees (right). In the rightmost graph, each tree uses 16 worker threads. The striking throughput improvement from using multiple trees arises because they help smooth out the otherwise extremely-bursty workload imposed upon the CBT; without additional trees, the CBT alternates between periods of idle and periods of maximum CPU use. With the additional trees, the cores are occupied nearly full-time. Partitioning keys across more trees adds additional root buffers, increasing memory use modestly.

5.5.3.3 Workload Properties

Here we evaluate how workload properties affect the performance of the CBT. We consider the number of unique keys, the aggregatability (for wordcount in the following experiments), the size of the PAO, and distribution of key length. Figure 27 shows the per-key memory consumption and aggregation throughput.

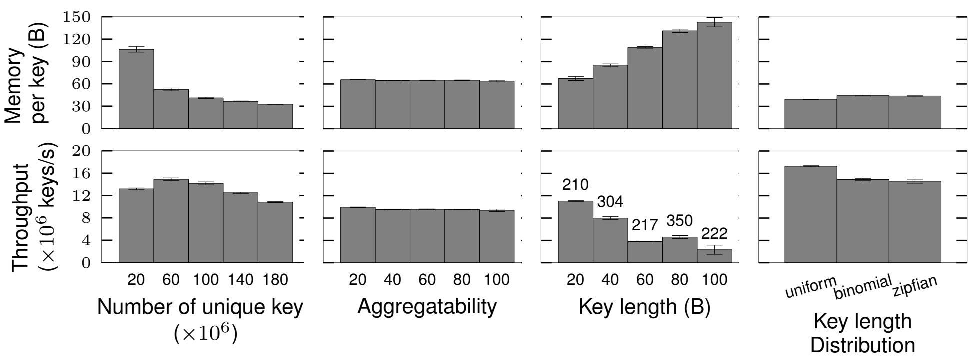


Figure 27: Evaluating the CBT with microbenchmarks

Aggregated data size We use synthetic datasets with an increasing number of unique keys while holding other properties constant. Figure 27 shows that the static memory use of the CBT causes per-key use to be high for (relatively) small number of keys, but this cost gets amortized with increasing number of unique keys.

Aggregatability We examine memory efficiency as a function of increasing aggregatability by using a progressively larger number of total keys with a fixed number of unique keys. With hashtable-based aggregation, increased aggregatability does not require more memory since a single PAO is maintained per key (and wordcount's PAOs do not grow with aggregation). Figure 27 shows that even for a CBT, despite buffering, per-key memory use does not increase with aggregatability.

PAO size Datasets with increasing key size are used, with the rest of the parameters remaining constant. The per-key memory use increases with increasing key-size, as expected. While the throughput appears to drop with increasing key-size in Figure 27, the throughput in terms of MB/s shown above the bars, does not share this trend.

Key length distribution We use datasets with 100 million unique keys, each occurring 10 times, with varying distributions on key length: uniform, binomial and Zipf; the average length of each word is around 10B. The throughput for binomial and Zipf are similar, with uniform being marginally higher owing to better cache use (each cache line fits 4 serialized PAOs) which improves performance during sorting and aggregation.

5.6 Conclusion

This chapter introduced the design and implementation of a new structure for memory-efficient, high-throughput aggregation: the Compressed Buffer Tree (CBT). The CBT is a stand-alone aggregator that can be used in different data-parallel frameworks. Our evaluation results show that when used as an aggregator for streaming data, the CBT uses up to 42% less memory than an aggregator based on Google SparseHash, an extremely memory-efficient hashtable implementation, while achieving equal or better throughput. CBTs can also be used in MapReduce runtimes. Substituting the default aggregator in the Metis MapReduce framework with the CBT, enables it to operate using 4-5x less memory and run 1.5-4x faster than default Metis and Phoenix++, another modern in-memory MapReduce implementation.

The primary goal of the CBT is to allow efficient aggregation on compressed data. Hashtables offer the *read-modify-update* paradigm for aggregation where up-to-date aggregates are always maintained for all keys. This paradigm makes it nearly impossible to keep data compressed since it may be accessed at any time. The CBT is built on the realization that aggregation need not be eager and aggregates can be lazily computed. With this relaxation, the CBT maintains aggregate data in compressed form (increasing memory efficiency), and limits the number of decompression and compression operations (increasing aggregation throughput).

CHAPTER VI

EXTERNAL, LAZY AGGREGATION USING THE WB TREE

In the past decade, performing real-time analytics on large, continuously-updated datasets has become essential. This chapter focuses, on a variant of GroupBy-Aggregate, where the memory available on each node in the cluster is insufficient to store the aggregate data on that node. In Chapter 5, we have shown how memory-efficient data structures increase the amount of data that can be aggregated on a single node. However, for large datasets, it may be necessary to offload aggregate state to secondary storage. In this chapter, we demonstrate how the WriteBuffer (WB) Tree, presented in Chapter 3, can be used to implement a high performance external aggregator.

6.1 Introduction

The requirements of an external aggregator are: (a) fast, incremental aggregation, (b) efficient access to aggregate state. We briefly discuss these requirements next: For high-throughput aggregation, the aggregator must perform I/O efficiently. For example, an external sort-based aggregator is more likely to use a variant of merge sort rather than insertion sort, because the former performs bulk I/O instead of inefficient, random I/O. The aggregator must also support incremental aggregation efficiently. To incorporate new, arriving records into the aggregate corpus, a sort-based aggregator, for example, requires the entire corpus to be merged with the new data. A hash-based aggregator, on the other hand, allows only the new keys to be updated. Finally, the aggregator must support access to aggregate data at any time.

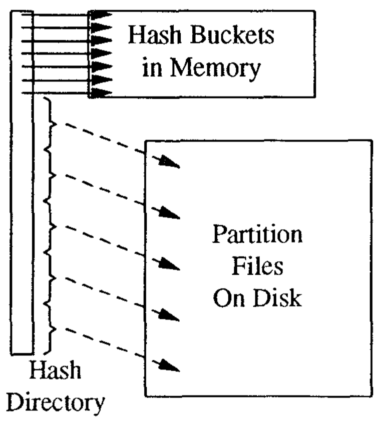


Figure 28: Hybrid hashing (reproduced from [55])

For example, sort-based aggregation does not support access to aggregate data when sorting is underway. We consider previous appraches to external aggregation next.

6.2 Previous approaches

External Sorting and Hybrid Hashing When available memory is insufficient to perform aggregation, two methods have previously been used [55] for batch aggregation: (1) external sorting, and (2) hybrid hashing. In the case of sorting, the input data is first written to disk as sorted runs; next, these sorted runs are merged into progressively larger runs until a single, sorted run remains, which is the output.

In the case of hashing, a method similar to the hybrid hash-join [43] is used. In this method, the input is partitioned into F partitions. Each partition is uses a separate hashtable in memory for aggregation. If a memory overflow occurs, one of the partitions is written to disk. Each time an overflow occurs, another partition is written to disk (all F partitions may eventually be written to disk). The number of disk partitions is typically chosen such that each partition may be expected to fit in memory [43]. If one of the partitions is larger than the memory available, then this partition must be further partitioned recursively. This method is used by recent systems like One Pass MapReduce [69].

External sorting and hybrid hashing work well if the data has to be aggregated in a batch. But *incremental aggregation* is inefficient, as the entire aggregate state, along with the new records, has to either be sorted and aggregated or run through an additional hybrid-hashing pass. This is similar to the problem that motivated the creation of Percolator [87]; in that case, recall that the authors propose techniques to incorporate newly crawled webpages into the web-index without having to rebuild the entire index. Another problem with external sorting and hybrid hashing is that these methods do not permit access to partially-aggregated data when an aggregation pass is in progress.

A possible solution is to use a disk-based data structure to store accumulators for each key. For example, an external hashtable [48], or B+ Tree can be used. Aggregation could then performed using a read-modify-write approach, where the current accumulator for a given key is read into memory, the new record aggregated into the accumulator, and the accumulator written back to disk. Clearly, this approach will suffer from poor performance as each aggregate operation requires two I/Os.

6.3 Solution: Use Write-Optimized Data Structures

To solve the problem of poor aggregation performance, we recognize that aggregation performance essentially depends on the write performance of the data structure (this was also the case with the Compressed Buffer Tree (CBT) in Chapter 5). The data structure for aggregate data can be viewed as a black box into which records meant to be aggregated are inserted. The mechanism of aggregation (e.g. eager or lazy aggregation) is internal to the black box. As before, we also require that records that have the same key "cluster together" in the data structure for fast finalization (finalization is the process that enforces that there is only one record per key in the data structure). While a purely log-based data structure provides extremely high write performance, finalization is inefficient.

As shown in Chapter 3, the WB Tree is a data structure that satisfies both constraints. For external aggregation, the WriteBuffer (WB) Tree is used to store partially-aggregated data. Instead of performing collapsing as explained in Section 3.3.1, where two records with the same key are replaced by the newer record, this process is modified to perform an aggregation pass. A merge function is registered before-hand with the aggregator, and is invoked on pairs of records with the same key, and the result is written in place.

6.3.1 Supporting Random Access

The WB Tree supports random access, but only tracks the last inserted record for each key. In the case of aggregation, the aggregate value computed from all inserted records is required. To support this, we make a simple modification to the WB Tree: Recall that in the original WB Tree, during a GET, the nodes along some root-leaf path (depending on the queried key) in the tree are searched. Each list in each of these nodes is searched. The search continues until either the queried key is found along the path or the last level in the leaf returns a negative. To allow the tree to act as an aggregator, instead of stopping the search at the first occurrence of the key, all the levels in the tree are searched. All records for the key found are then aggregated on-the-fly and the computed result returned.

6.4 Evaluation

This section compares the WB Tree-based aggregator with an external sort-based aggregator that uses Nsort [83]; Nsort is a highly-optimized sorter used by winners in many categories of the Sort Benchmark [8]. The sort-based aggregator only supports efficient batch aggregation. We also compare the WB Tree-based aggregator to a external hashtable-based aggregator using Kyoto Cabinet [66]'s HashDB, which is a disk-based hashtable that uses chaining to resolve collisions.

The experiments use a 12-core server (two 2.66GHz six-core Intel X5650 processors) with 12GB of DDR3 RAM. The disk used is an Intel 520 SSD. We report the median of three runs of each experiment. We use the `jemalloc` [47] memory allocator for all experiments. We use bulk interfaces for insertion.

6.4.1 Tuning

WB Tree We use a fan-out of 256 and node size of 600MB. As explained in §3.4.3, using relatively high fan-outs and node sizes favors `INSERT` performance, and therefore, aggregation throughput.

Kyoto HashDB For HashDB, we disable compression; we enable TLINEAR option (uses linked lists instead of binary trees for collision chaining) to reduce the memory footprint of each record. The number of hash buckets is set to greater than the number of unique keys in our benchmarks (to keep the load factor low and eliminate possible rehashing). Finally, as recommended [66], 6GB is allocated to the internal memory-mapped region.

6.4.2 Datasets

The Yahoo! Cloud Serving Benchmark (YCSB) [37] tool is used to generate workload traces, which are replayed in a light-weight workload generator. The dataset used is denoted as (U, R, e, S) where U is the number of unique keys, R is the average number of repeats for each key, e is the record size and S is the total size of the dataset. We use uniformly distributed data for experiments. The datasets used are: D_1 (red): (10^8, 20, 16B, 21GB), D_2 (blue): (2.5×10^8, 20, 16B, 21GB), and D_3 (green): (5×10^8, 4, 16B, 21GB) and D_4 (black): (10^9, 2, 16B, 21GB).

6.4.3 Aggregation Throughput

In this experiment, we compare the aggregation throughput of different aggregators. Recall that only WB Tree and the external hashtable support incremental aggregation

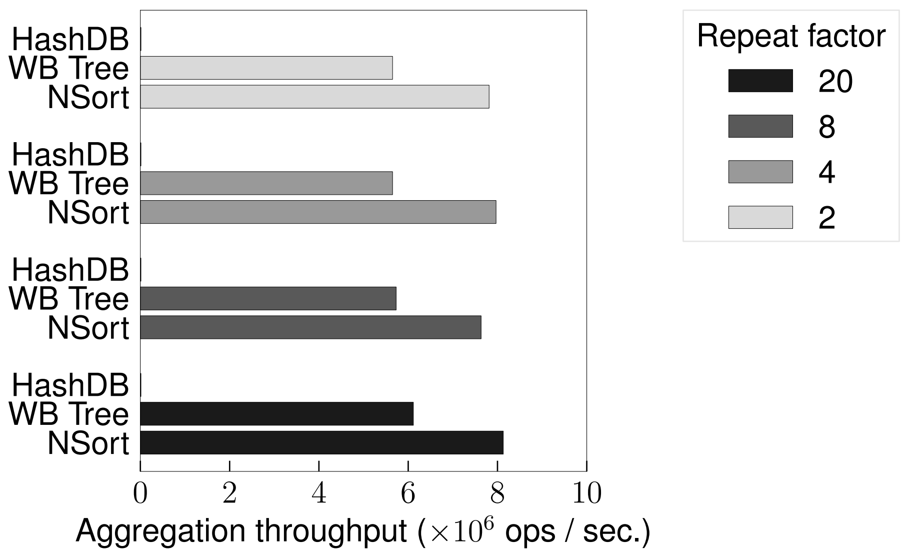


Figure 29: Comparison of batch aggregation throughput

efficiently. Sort-based aggregation is fast when all records are available before-hand. The following may be concluded from Figure 29:

- Using HashDB (and other in-place modify structures) to store accumulators and aggregating eagerly using a read-modify-write approach leads to poor aggregation throughput due to random I/Os.

- WB Tree-based aggregation achieves about 75% the performance of sort-based aggregation, but the latter does not support incremental aggregation.

6.4.4 Query Throughput

Because they are batch aggregators, sort-based aggregators also do not allow partially aggregated values to be queried while aggregation is underway. However, both HashDB and the WB Tree, which performs some aggregation on-the-fly, support random queries. Some important observations can be made from Figure 30:

- HashDB provides excellent query throughput compared to the WB Tree, because HashDB always maintains up-to-date aggregates for all keys. The WB Tree, instead, performs some aggregation on-the-fly.

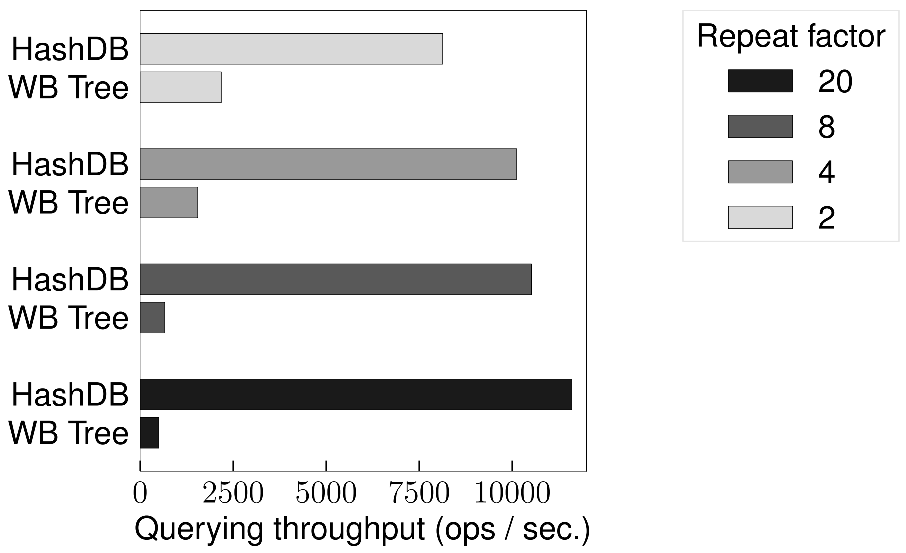


Figure 30: Comparison of aggregator query performance

- The throughput of HashDB decreases with an increase in the size of the hashtable, due to a lower number of hits in the page cache.
- The throughput of WB Tree decreases with an increase in the repeat factor, because higher repeat factors increase the potential amount of on-the-fly aggregation.

6.5 Conclusion

This chapter describes how the WriteBuffer (WB) Tree, introduced in Chapter 3, can be modified to serve as a high-throughput external aggregator. Stand-alone aggregators, when used in fast analytics runtimes, are required to support fast, incremental aggregation, along with random access to the currently aggregated values for all keys. Existing solutions are either designed for batch aggregation (i.e., incremental aggregation is inefficient and random access not supported when aggregation is underway) or support incremental aggregation at the cost of a loss in aggregation throughput. The WB Tree solves this problem by supporting incremental aggregation at nearly 75% the throughput of high-performance batch aggregators.

CHAPTER VII

RELATED WORK

We first discuss related work in write-optimized data structures in Section 7.1, which may be used to design high throughput aggregations structures as well as improve the write performance of key-value stores. Next, we discuss approaches to improve read performance in key-value stores in Section 7.2. Finally, in Section 7.3, we discuss previous approaches to GroupBy-Aggregate implementations.

7.1 Write-Optimized Stores

A wide range of data structures have been proposed for use in key-value stores. The External Memory model [13] is a popular framework for the comparison of external data structures. It models the memory hierarchy as consisting of two levels (e.g. DRAM and disk). The CPU is connected to the first level (sized M) which is connected, in turn, to a large second level. Both levels are organized in blocks of size B and, transfers between levels happen in units of blocks. Various data structures can be viewed on a spectrum of read-write tradeoffs.

At one end of the spectrum are B-trees and external hashtables [48]. Both methods provide low latency reads, but modify data on disk in-place. This means that each write operation requires two I/Os. At the other end are unordered log-structured stores such as the Log-Structured File System [91] and FAWNDS [15]. They are called *unordered* because they do not order keys in any way (e.g. sort) before writing to disk. Typically, a buffer is maintained in memory and newly inserted key-value pairs are copied into it. When full, it is written to a log on disk. This amortizes the cost of the write across all key-value pairs in the block providing the optimal cost of $O(\frac{1}{B})$ I/Os per insert. However, gets need to read the entire log in the worst case.

Ordered log-structured stores form the middle of the spectrum. These buffer multiple key-value pairs in memory and *order* them before writing to disk. Ordered log-structured stores offer much higher insert performance compared to in-place update structures like B-trees, and also provide the crucial advantage that key-value pairs with the same key tend to cluster together in the data structure allowing periodic merge operations to be performed efficiently. Examples of ordered log-structured stores include Log-Structured Merge (LSM) Trees [85], Cache-Oblivious Lookahead Arrays (COLA) [21] and buffer trees [16]. Similar to unordered stores, ordered stores amortize the cost of disk writes across multiple key-value pair inserts, but, unlike them, periodically merge records belonging to the same key.

LSM Trees maintain multiple B-Trees (components) organized in levels with exponentially increasing size. When a tree at one level becomes full, it flushes to the tree at the next level. COLA are similar to LSM Trees, but use arrays instead of B-trees at each level for slightly slower reads. Both LSM Trees and COLA, with their emphasis on maintaining low latency for gets, aim to serve as online data structures and are used as data stores in key-value databases [28]. Many real-world systems including BigTable [28], bLSM [94], LevelDB [52], HBase [1] are variants of the LSM tree. BigTable writes sorted runs to disk as *minor compactions* and periodically performs a *major compaction* by merging multiple sorted runs and rewriting them as a single sorted run. This technique is also used by HBase and Cassandra (pre 1.0). FD-Trees are LSM Trees that optimize for SSDs [71] by noting that writes with good locality have similar performance as sequential writes.

The Sorted Array Merge Tree (SAMT), used in GTSSL [97], uses exponentially-sized levels and writes *multiple* possibly-overlapping ranges from one component to the next before having to perform a compaction. The SAMT has asymptotically faster writes than LSM Trees. GTSSL develops techniques to adapt to changing read-write ratios and adapting to hybrid disk-flash systems.

The buffer tree has a slightly different goal. It targets applications that *do not* require low latency queries, and only provides good amortized query performance, as shown in Table 6. In return, buffer trees make better use of available memory, to improve on the insert performance of LSM Trees and COLA. Briefly, the buffer tree maintains an (a, b)-tree with each node augmented by a memory-sized buffer. Inserts and queries are simply copied into the root buffer. When any node fills up, it is emptied by pushing the contents of the node down to the next level of the tree. During the emptying process, multiple inserts with the same key can be coalesced.

The WriteBuffer (WB) Tree is an *ordered* key-value store in the sense that buffered records in memory are ordered before being spilled to disk and records with the same key will eventually merge. However, the WB-Tree is a *hash-ordered* store, and not a *key-ordered* store. We clarify this, as *ordered* is sometimes conflated with *key-ordered* in the literature. The WB Tree is closest to buffer trees, but differs from them by buffering only writes, and not read requests. Moreover, WB Trees introduce the idea of fast-splitting for faster writes.

Many key-value stores either implicitly (e.g. cached layers of a B+ Tree) or explicitly (e.g., HashCache [18], bLSM [94]) use a memory-based index to improve read performance. Reducing the memory consumed by the index has been a focus in recent work. SILT [72] is a multi-store key-value store that focuses on memory-efficiency of the index and read performance. It uses three different key-value stores with varying memory efficiency and write-friendliness and moves input data between them, and achieves a lower memory use per key and I/Os required per read compared to other SSD-based key-value stores such as SkimpyStash [40], BufferHash [14] and HashCache [18].

7.2 Read Performance

B+Trees and disk-based hashtables typically offer the best read performance. By using high fan-outs, which results in a large proportion of nodes being leaves, B+ Trees cache higher levels of the tree in memory allowing them to perform reads with a single I/O.

Since LSM Trees consist of multiple memory and disk-based components, each component may have to be checked during a read. To improve performance, datastores typically (a) cache frequently accessed data in memory, (b) protect components with Bloom filter to prevent wasteful accesses (e.g. LevelDB, Cassandra, bLSM), or (c) use fractional cascading [31], where partial results from searching one component are used to speed up searching following components (e.g. Cache-Oblivious Lookahead Arrays (COLA) [21]).

Multi-level data structures, such as the LSM Tree, can use additional indexes that map each stored key to the level in the tree that stores the key. This index can be implemented using hashtables, but even a memory-efficient hashtable like Sparsehash [54] is space-inefficient for this function as it has to store entire keys in memory (for collision resolution). Read requests in the WB Tree are handled similar to an LSM Tree by sequentially checking each possible location, and protecting each location with a Bloom filter to avoid wasteful I/Os.

SILT [72] includes an immutable index that uses minimal perfect hashing that maps n keys to the consecutive integers $0 \ldots n-1$ with no collisions; this does not require the keys to be present in memory for non-mutating accesses. However, for dynamic perfect hashing, keys are required to be present in memory to allow rehashing parts of the hashtable in case of inserts that may cause collisions [44]. The WB Tree, instead, constructs the List-Selection index by using Bloom filters to protect each list. We do not use memory-efficient indexes from SILT in the WB Tree as these are read-only, and rebuilding costs impacts write throughput.

7.3 GroupBy-Aggregate

7.3.1 Aggregation in Databases

Graefe [55] and Chaudhuri [30] survey sort and hash-based aggregation in database queries and propose a number of techniques:

- When possible, use hashing to organize the aggregation columns in memory, storing one aggregate value for each hash entry.
- When memory is insufficient, the use of sorting or hybrid hashing [43] is recommended, which then allows aggregation in a single sequential pass.

SQL GROUP BY and aggregate operators produce zero or one-dimensional aggregates; The CUBE SQL operator was proposed as an N-dimensional generalization of GroupBy-Aggregate by Gray et al. [57]. The operator treats each of the N aggregation attributes as a dimension in N-space. Sort and hash-based methods for efficiently implementing the Cube are also discussed. Iceberg-CUBE extended the CUBE operation to include user-specified condition to filter out groups [24].

Kotidis et al. propose R-trees to support Cube operations with storage reduction and faster updates [65]. Similar to our work, the authors focus on using bulk, incremental updates, contending that neither record-at-a-time updates nor re-computation of the Cube are viable alternatives.

There is also a significant amount of previous work in online aggregation in databases [60] that returns early results and on-the-fly modification of queries. Database systems can be dichotomized into Online Transaction Processing (OLTP) systems and Online Analytical Processing (OLAP) systems [34]. OLTP systems primarily store operational data with high integrity and support relatively simple queries, while OLAP systems support complex queries, involving aggregations, with interactive performance on large quantities of data. This involves maintaining auxiliary data including join and bitmap-indices to speed up processing. Even within OLAP systems, Relational OLAP (ROLAP) systems attempt to further speed up aggregations by storing pre-computed aggregates in summary tables [92].

7.3.2 Aggregation in Batch-Processing Systems

Yu et al. [104] discuss the generality of the GroupBy-Aggregate operation across a range of batch-processing programming models including MapReduce, Dryad [62] and parallel databases.

MapReduce [39] and Hadoop [2] use sort-based aggregation. Other work that has considered the optimization of the aggregator include Tenzing [29], a SQL query execution engine based on MapReduce that uses hashtables to support SQL aggregation operations, and One-pass MapReduce [69]. Phoenix++ and its predecessors [99, 89], Metis [79], Tiled-MapReduce [32] and MATE [64] are shared-memory MapReduce implementations that also focus on the design of aggregation data structures.

7.3.3 Incremental Aggregation in Stream-Processing Systems

Several proposed modifications [45, 105] to MapReduce allow incremental aggregation by allowing read access to existing aggregate state while mapping new data. For example, to implement pagerank, the current pageranks of pages in the index can be accessed randomly while mapping newly-crawled pages that link to existing pages. CBP [76] and Comet [59] also provide bulk incremental processing by periodically executing MapReduce jobs on new data. MapReduce Online [35] allows application stages to be pipelined, and allows incremental aggregation.

Stream-processing systems are designed to support *continuous queries* on streams of input data, such as monitoring data. Pioneering streaming systems such as Aurora [11], Borealis [10], STREAM [17], and SPADE [50] include primitives for supporting grouping and aggregation of events. For example, SPADE includes aggregation operators to compute moving averages over event data.

Among recent stream-processing systems, Spark Streaming [106] allows computing aggregates (e.g. averages) over sliding windows. Muppet [68], which provides a MapReduce-like interface to operate on streams, and Twitter's Storm [9], also support interactive queries by pre-computing aggregates. These systems allow fully user-defined aggregate functions to be set up as aggregators that are updated in near real-time with input data streams.

7.4 Compression for Memory Efficiency

Although the CBT applies conventional compression techniques to save memory, the grouping and aggregation occurs on decompressed data. A natural alternative to this approach is to use specialized compressed data structures which allow specific operations to be performed on compressed data. This study of compressed or succinct data structures is advancing rapidly; for example, both Grossi et al. and Välimäki et al. show how compressed suffix trees help in practice for real-world problems by reducing memory use [58, 102]. While we believe this is a promising approach, to our knowledge, no compressed data structure provides a suitable interface to implement a generic GroupBy-Aggregate operation.

There is a significant body of work involving compression in databases. Chen et al. compress the contents of databases, and derive optimal plans for queries involving compressed attributes [33], and Li et al. consider aggregation algorithms in a compressed Multi-dimensional OLAP databases [70]. Rose [93] is a database storage engine that uses LSM Trees; Rose uses compression to reduce the amount of sequential I/O required by LSM Tree merges. The Compressed Buffer Tree (CBT) uses compression in a similar manner to reduce the memory consumption of the in-memory aggregator.

Memory Compression A number of previous approaches integrate compression into the memory hierarchy in a manner that is transparent to applications [12].

Among software-only approaches, swap compression [90] involves setting aside a partition in the main memory to store evicted pages in compressed form and gain performance by avoiding reads from storage. The CBT is similar to these approaches in the sense that it saves memory by storing memory contents in compressed form, transparently to (in our case, MapReduce) applications. The CBT can benefit from hardware support for memory compression if available.

CHAPTER VIII

CONCLUSION AND FUTURE WORK

8.1 Conclusion

This dissertation has demonstrated that buffering is a powerful technique that can be used to construct write-optimized key-value stores and resource-efficient aggregators. The rapid growth in popularity of fast analytics systems places unique demands on components, such as key-value stores and aggregators, used to compose these systems.

Existing designs for key-value stores, optimizing for WORM (Write Once Read Many) workloads, focus primarily on read performance (i.e., low latency and high read throughput). For key-value stores that use single-node stores based on in-place-update data structures such as the B-Tree, write performance is limited by the random I/O throughput. For this reason, single-node stores are typically based on variants of write-optimized data structures such as the LSM Tree or Cache-Oblivious Lookahead Arrays (COLA). Even for these data structures, write throughput is limited by the need to perform expensive compactions. The WB Tree replaces compactions with cheaper spill and split primitives, providing up to 30× higher write performance compared to LSM Trees, while providing similar read performance.

State-of-the-art aggregators rely on hash or sort-based aggregation. This results in two problems: (a) hashtables are memory-inefficient, and (b) when the available memory is insufficient, the respective methods to overflow to disk (i.e., using external sorting and hybrid hashing) are unsuitable for incremental aggregation. For in-memory aggregation, the Compressed Buffer Tree (CBT) data structure provides up to 50% higher aggregation throughput while requiring between 21-42% less memory.

8.2 Future Work

In addition to future work discussed in Chapters 3 and 5, we briefly discuss some other applications for the techniques described in this dissertation.

8.3 Evolving Memory Hierarchies

Stacked DRAM Die-stacking is an emerging technology that promises to provide high-bandwidth and low-latency memory access, providing relief from the "memory wall" problem. However, not all of main memory can be practically provided by stacked DRAM, leading to a heterogeneous memory space consisting of both on-chip stacked DRAM and conventional off-chip DRAM [77]. We believe that techniques for memory-efficient indexes explored here will prove useful.

Non-volatile memories Further, non-volatile memories, such as Phase Change Memory (PCM), are a promising solution to insufficient DRAM capacities that possess unique characteristics such as non-volatility, fast random read access, and improved lifetimes compared to NAND flash. Since PCM devices exhibit asymmetry in read and write performance (reads latencies are similar to DRAM, while writes are slower), structures such as the WB Tree which batch writes, while allowing random read access, can be leveraged to exploit the capabilities of PCM in analytics software stacks. The ability to move data from DRAM to PCM in a log-structured fashion, as performed by write-optimized data structures, enables the masking of individually slower writes, and also enables wear-leveling.

8.4 Accelerating GroupBy-Aggregate

An important idea in this work is that I/O-efficient data structures are useful technique for transferring data in memory hierarchies. This is especially true for hierarchies that can be modeled using the external model of Aggarwal and Vitter [13].

Based on initial results, we believe that this technique is also applicable to offloading GroupBy-Aggregate execution to accelerators.

Offloading GroupBy-Aggregate to the GPU is challenging for two reasons: (1) GPU execution works best on large blocks, but the intermediate key-value pairs are usually quite small; (2) naive batching of intermediate key-value pairs could result in a large number of transfers to and from the GPU leading to unacceptably high overhead. Using write-optimized data structures, such as the buffer tree, solves the problem because they enable batching of intermediate key-value pairs during copying of data between GPU and CPU memory. This cost of data movement is analogous to reading and writing buffers to disk in the setting of external data structures.

8.5 Multi-dimensional GroupBy-Aggregate

In this dissertation, we show how write-optimized data structures can be used to fast and efficient GroupBy-Aggregate. We believe that a promising future direction would be extend our results for multi-dimensional GroupBy-Aggregate. Spatial data structures to support the CUBE SQL operator [57], which is d-dimensional GroupBy-Aggregate, remains an active area of research [38], but to our knowledge, the application of *buffered* variants of these data structures, or other multi-dimensional write-optimized data structures has not been explored.

REFERENCES

[1] `hbase.apache.org`.

[2] "Hadoop." `hadoop.apache.org`.

[3] "Improving twitter search with real-time human computation." `blog.twitter.com/2013/improving-twitter-search-real-time-human-computation`.

[4] "LZO." `oberhumer.com/opensource/lzo`.

[5] "Memcached – a distributed memory object caching system." `memcached.org`.

[6] "ØMQ (zeroMQ)." `zeromq.org`.

[7] "Project Gutenberg." `www.gutenberg.org`.

[8] "The sort benchmark home page." `sortbenchmark.org`.

[9] "Storm." `storm-project.net`.

[10] Abadi, D. J., Ahmad, Y., Balazinska, M., Cherniack, M., Hwang, J.-h., Lindner, W., Maskey, A. S., Rasin, E., Ryvkina, E., Tatbul, N., Xing, Y., and Zdonik, S., "The Design of the Borealis Stream Processing Engine," in *In Conference on Innovative Data Systems Research (CIDR)*, pp. 277–289, 2005.

[11] Abadi, D. J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., and Zdonik, S., "Aurora: A New Model and Architecture for Data Stream Management," *The VLDB Journal*, vol. 12, pp. 120–139, Aug. 2003.

[12] Abali, B., Franke, H., Shen, X., Poff, D. E., and Smith, T. B., "Performance of Hardware Compressed Main Memory," in *Proceedings of the 7th International Symposium on High-Performance Computer Architecture*, HPCA '01, (Washington, DC, USA), pp. 73–81, IEEE Computer Society, 2001.

[13] Aggarwal, A. and Vitter, Jeffrey, S., "The Input/Output Complexity of Sorting and Related Problems," *Communications of the ACM*, vol. 31, pp. 1116–1127, Sept. 1988.

[14] Anand, A., Muthukrishnan, C., Kappes, S., Akella, A., and Nath, S., "Cheap and large cams for high performance data-intensive networked systems," in *Proceedings of the 7th USENIX conference on Networked systems design and implementation*, NSDI'10, (Berkeley, CA, USA), pp. 29–29, USENIX Association, 2010.

[15] Andersen, D. G., Franklin, J., Kaminsky, M., Phanishayee, A., Tan, L., and Vasudevan, V., "FAWN: a fast array of wimpy nodes," in *SOSP '09: Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, (New York, NY, USA), pp. 1–14, ACM, 2009.

[16] Arge, L., "The Buffer Tree: A Technique for Designing Batched External Data Structures," *Algorithmica*, vol. 37, no. 1, pp. 1–24, 2003.

[17] Babu, S. and Widom, J., "Continuous Queries over Data Streams," *ACM SIGMOD Record*, vol. 30, pp. 109–120, Sept. 2001.

[18] Badam, A., Park, K., Pai, V. S., and Peterson, L. L., "Hashcache: cache storage for the next billion," in *Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, NSDI'09, (Berkeley, CA, USA), pp. 123–136, USENIX Association, 2009.

[19] Bar-Yossef, Z., Jayram, T. S., Kumar, R., Sivakumar, D., and Trevisan, L., "Counting Distinct Elements in a Data Stream," in *Proceedings of the 6th International Workshop on Randomization and Approximation Techniques*, RANDOM '02, (London, UK, UK), pp. 1–10, Springer-Verlag, 2002.

[20] Beaver, D., Kumar, S., Li, H. C., Sobel, J., and Vajgel, P., "Finding a needle in haystack: facebook's photo storage," in *Proceedings of the 9th USENIX conference on Operating systems design and implementation*, OSDI'10, (Berkeley, CA, USA), pp. 1–8, USENIX Association, 2010.

[21] Bender, M. A., Farach-Colton, M., Fineman, J. T., Fogel, Y. R., Kuszmaul, B. C., and Nelson, J., "Cache-Oblivious Streaming B-Trees," in *Proceedings of the Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '07, (New York, NY, USA), pp. 81–92, ACM, 2007.

[22] Berger, E. D., McKinley, K. S., Blumofe, R. D., and Wilson, P. R., "Hoard: A Scalable Memory Allocator for Multithreaded Applications," in *Proceedings of the Ninth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-IX, (New York, NY, USA), pp. 117–128, ACM, 2000.

[23] Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Karp, S., Keckler, S., Klein, D., Lucas, R., Richards, M., Scarpelli, A., Scott, S., Snavely, A., Sterling, T., Williams, R. S., Yelick, K., Bergman, K., Borkar, S., Campbell, D., Carlson, W., Dally, W., Denneau, M., Franzon, P., Harrod, W., Hiller, J., Keckler, S., Klein, D., Kogge, P., Williams, R. S., and Yelick, K., "Exascale computing study: Technology challenges in achieving exascale systems," 2008.

[24] Beyer, K. and Ramakrishnan, R., "Bottom-up Computation of Sparse and Iceberg CUBE," in *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data*, SIGMOD '99, (New York, NY, USA), pp. 359–370, ACM, 1999.

[25] Blanas, S. and Patel, J. M., "Memory Footprint Matters: Efficient Equi-Join Algorithms for Main Memory Data Processing," in *Proceedings of the 4th ACM Symposium on Cloud Computing*, SoCC '13, (New York, NY, USA), ACM, 2013.

[26] Busch, M., Gade, K., Larson, B., Lok, P., Luckenbill, S., and Lin, J., "Earlybird: Real-Time Search at Twitter," in *Proceedings of the 2012 IEEE 28th International Conference on Data Engineering*, ICDE '12, (Washington, DC, USA), pp. 1360–1369, IEEE Computer Society, 2012.

[27] Cavnar, W. B. and Trenkle, J. M., "N-Gram-Based Text Categorization," in *In Proceedings of SDAIR-94, 3rd Annual Symposium on Document Analysis and Information Retrieval*, pp. 161–175, 1994.

[28] Chang, F., Dean, J., Ghemawat, S., Hsieh, W. C., Wallach, D. A., Burrows, M., Chandra, T., Fikes, A., and Gruber, R. E., "Bigtable: A Distributed Storage System for Structured Data," *ACM Transactions on Computer Systems*, vol. 26, pp. 4:1–4:26, June 2008.

[29] Chattopadhyay, B., Lin, L., Liu, W., Mittal, S., Aragonda, P., Lychagina, V., Kwon, Y., and Wong, M., "Tenzing: A SQL Implementation on the Mapreduce Framework," in *Proceedings of the VLDB Endowment*, vol. 4, pp. 1318–1327, 2011.

[30] Chaudhuri, S., "An Overview of Query Optimization in Relational Systems," in *Proceedings of the seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '98, (New York, NY, USA), pp. 34–43, ACM, 1998.

[31] Chazelle, B. and Guibas, L. J., "Fractional Cascading: A Data Structuring Technique with Geometric Applications," in *Automata, Languages and Programming*, vol. 194 of *Lecture Notes in Computer Science*, pp. 90–100, Springer Berlin Heidelberg, 1985.

[32] Chen, R., Chen, H., and Zang, B., "Tiled-Mapreduce: Optimizing Resource Usages of Data-Parallel Applications on Multicore with Tiling," in *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, (New York, NY, USA), pp. 523–534, ACM, 2010.

[33] Chen, Z., Gehrke, J., and Korn, F., "Query Optimization in Compressed Database Systems," in *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, SIGMOD '01, (New York, NY, USA), pp. 271–282, ACM, 2001.

[34] Codd, E. F., Codd, S. B., and Clynch, S. T. Computerworld, (New York, NY, USA), ACM, July 1993.

[35] Condie, T., Conway, N., Alvaro, P., Hellerstein, J. M., Gerth, J., Talbot, J., Elmeleegy, K., and Sears, R., "Online aggregation and continuous query support in mapreduce," in *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 1115–1118, ACM, 2010.

[36] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R., "PNUTS: Yahoo!'s Hosted Data Serving Platform," *Proceedings of the VLDB Endowment*, vol. 1, pp. 1277–1288, Aug. 2008.

[37] Cooper, B. F., Silberstein, A., Tam, E., Ramakrishnan, R., and Sears, R., "Benchmarking Cloud Serving Systems with YCSB," in *Proceedings of the 1st ACM Symposium on Cloud computing*, SoCC '10, (New York, NY, USA), pp. 143–154, ACM, 2010.

[38] Cuzzocrea, A. and Leung, C. K., "Efficiently compressing olap data cubes via r-tree based recursive partitions," in *Proceedings of the 20th international conference on Foundations of Intelligent Systems*, ISMIS'12, (Berlin, Heidelberg), pp. 455–465, Springer-Verlag, 2012.

[39] Dean, J. and Ghemawat, S., "Mapreduce: Simplified Data Processing on Large Clusters," in *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2004.

[40] Debnath, B., Sengupta, S., and Li, J., "Skimpystash: Ram space skimpy key-value store on flash-based storage," in *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 25–36, ACM, 2011.

[41] DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., and Vogels, W., "Dynamo: amazon's highly available key-value store," in *Proceedings of twenty-first ACM SIGOPS Symposium on Operating Systems principles*, SOSP '07, (New York, NY, USA), pp. 205–220, ACM, 2007.

[42] DeWitt, D. J. and Gerber, R. H., "Multiprocessor Hash-Based Join Algorithms," in *Proceedings of the 11th International Conference on Very Large Data Bases*, VLDB '85, pp. 151–164, VLDB Endowment, 1985.

[43] DeWitt, D. J., Katz, R. H., Olken, F., Shapiro, L. D., Stonebraker, M. R., and Wood, D. A., "Implementation Techniques for Main Memory Database Systems," vol. 14, pp. 1–8, June 1984.

[44] Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Meyer auf der Heide, F., Rohnert, H., and Tarjan, R. E., "Dynamic Perfect Hashing: Upper and Lower Bounds," *SIAM J. Comput.*, vol. 23, pp. 738–761, Aug. 1994.

[45] Ekanayake, J., Li, H., Zhang, B., Gunarathne, T., Bae, S.-H., Qiu, J., and Fox, G., "Twister: a runtime for iterative mapreduce," in *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, (New York, NY, USA), pp. 810–818, ACM, 2010.

[46] Escriva, R., Wong, B., and Sirer, E. G., "HyperDex: A Distributed, Searchable Key-Value Store," *SIGCOMM Computer Communication Review*, vol. 42, pp. 25–36, Aug. 2012.

[47] Evans, J., "A Scalable Concurrent `malloc(3)` Implementation for FreeBSD,"

[48] Fagin, R., Nievergelt, J., Pippenger, N., and Strong, H. R., "Extendible Hashing - a Fast Access Method for Dynamic Files," *ACM Trans. Database Syst.*, vol. 4, pp. 315–344, Sept. 1979.

[49] Fiat, A. and Woeginger, G. J., eds., *Developments from a June 1996 seminar on Online algorithms: the state of the art*, (London, UK, UK), Springer-Verlag, 1998.

[50] Gedik, B., Andrade, H., Wu, K.-L., Yu, P. S., and Doo, M., "SPADE: the System S Declarative Stream Processing Engine," in *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, (New York, NY, USA), pp. 1123–1134, ACM, 2008.

[51] Ghemawat, S. and Menage, P., "Tcmalloc: Thread-caching malloc." `goog-perftools.sourceforge.net/doc/tcmalloc.html`.

[52] Google, "Leveldb." `code.google.com/p/leveldb`.

[53] Google, "Snappy." `code.google.com/p/snappy`.

[54] Google, "Sparsehash." `code.google.com/p/sparsehash`.

[55] Graefe, G., "Query Evaluation Techniques for Large Databases," *ACM Computing Surveys*, vol. 25, pp. 73–169, June 1993.

[56] Graefe, G., "Write-Optimized B-trees," in *Proceedings of the Thirtieth International Conference on Very Large Data Bases - Volume 30*, VLDB '04, pp. 672–683, VLDB Endowment, 2004.

[57] Gray, J., Chaudhuri, S., Bosworth, A., Layman, A., Reichart, D., Venkatrao, M., Pellow, F., and Pirahesh, H., "Data Cube: A Relational Aggregation Operator Generalizing Group-By, Cross-Tab, and Sub-Totals," *Data Mining and Knowledge Discovery*, vol. 1, pp. 29–53, Jan. 1997.

[58] Grossi, R. and Vitter, J. S., "Compressed suffix arrays and suffix trees with applications to text indexing and string matching (extended abstract)," in *Proceedings of the Thirty-Second Annual ACM Symposium on Theory of Computing*, STOC '00, (New York, NY, USA), pp. 397–406, ACM, 2000.

[59] He, B., Yang, M., Guo, Z., Chen, R., Su, B., Lin, W., and Zhou, L., "Comet: Batched Stream Processing for Data Intensive Distributed Computing," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 63–74, ACM, 2010.

[60] Hellerstein, J. M., Haas, P. J., and Wang, H. J., "Online Aggregation," in *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, (New York, NY, USA), pp. 171–182, ACM, 1997.

[61] Huddleston, S. and Mehlhorn, K., "A New Data Structure for Representing Sorted Lists," *Acta Informatica*, vol. 17, no. 2, pp. 157–184, 1982.

[62] Isard, M., Budiu, M., Yu, Y., Birrell, A., and Fetterly, D., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," in *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, (New York, NY, USA), pp. 59–72, ACM, 2007.

[63] Islam, A. and Inkpen, D., "Real-Word Spelling Correction using Google Web IT 3-grams," in *Proceedings of the 2009 Conference on Empirical Methods in Natural Language Processing*, vol. 3 of *EMNLP '09*, (Stroudsburg, PA, USA), pp. 1241–1249, Association for Computational Linguistics, 2009.

[64] Jiang, W., Ravi, V. T., and Agrawal, G., "A Map-Reduce System with an Alternate API for Multi-core Environments," in *Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing*, CCGRID '10, (Washington, DC, USA), pp. 84–93, IEEE Computer Society, 2010.

[65] Kotidis, Y. and Roussopoulos, N., "An Alternative Storage Organization for ROLAP Aggregate Views Based on Cubetrees," in *Proceedings of the 1998 ACM SIGMOD International Conference on Management of Data*, SIGMOD '98, (New York, NY, USA), pp. 249–258, ACM, 1998.

[66] Labs, F., "Kyoto cabinet." `fallabs.com/kyotocabinet`.

[67] Lakshman, A. and Malik, P., "Cassandra: A Decentralized Structured Storage System," *SIGOPS Operating Systems Review*, vol. 44, pp. 35–40, Apr. 2010.

[68] Lam, W., Liu, L., Prasad, S., Rajaraman, A., Vacheri, Z., and Doan, A., "Muppet: MapReduce-Style Processing of Fast Data," *Proc. VLDB Endow.*, vol. 5, pp. 1814–1825, Aug. 2012.

[69] Li, B., Mazur, E., Diao, Y., McGregor, A., and Shenoy, P., "A Platform for Scalable One-Pass Analytics using MapReduce," in *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, (New York, NY, USA), pp. 985–996, ACM, 2011.

[70] Li, J., Rotem, D., and Srivastava, J., "Aggregation Algorithms for Very Large Compressed Data Warehouses," in *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, (San Francisco, CA, USA), pp. 651–662, Morgan Kaufmann Publishers Inc., 1999.

[71] Li, Y., He, B., Yang, R. J., Luo, Q., and Yi, K., "Tree Indexing on Solid State Drives," *Proceedings of the VLDB Endowment*, vol. 3, pp. 1195–1206, Sept. 2010.

[72] Lim, H., Fan, B., Andersen, D. G., and Kaminsky, M., "Silt: a memory-efficient, high-performance key-value store," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 1–13, ACM, 2011.

[73] Lim, K., Chang, J., Mudge, T., Ranganathan, P., Reinhardt, S. K., and Wenisch, T. F., "Disaggregated Memory for Expansion and Sharing in Blade Servers," in *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, (New York, NY, USA), pp. 267–278, ACM, 2009.

[74] Lim, K. T.-M., *Disaggregated memory architectures for blade servers.* PhD thesis, Ann Arbor, MI, USA, 2010. AAI3406380.

[75] Liu, H. `huanliu.wordpress.com/2011/01/24/the-true-cost-of-an-ecu/`.

[76] Logothetis, D., Olston, C., Reed, B., Webb, K. C., and Yocum, K., "Stateful bulk processing for incremental analytics," in *Proceedings of the 1st ACM Symposium on Cloud Computing*, SoCC '10, (New York, NY, USA), pp. 51–62, ACM, 2010.

[77] Loh, G. H., Jayasena, N., Chung, J., Reinhardt, S. K., O'Connor, J. M., and McGrath, K., "Challenges in heterogeneous die-stacked and off-chip memory systems," in *Proceedings of 3rd Workshop on SoCs, Heterogeneous Architectures and Workloads (SHAW-3)*, HPCA '12, 2012.

[78] Malewicz, G., Austern, M. H., Bik, A. J., Dehnert, J. C., Horn, I., Leiser, N., and Czajkowski, G., "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[79] Mao, Y., Morris, R., and Kaashoek, F., "Optimizing Mapreduce for Multicore Architectures," Tech. Rep. MIT-CSAIL-TR-2010-020, MIT, 2010.

[80] Mariòo, J. B., Banchs, R. E., Crego, J. M., de Gispert, A., Lambert, P., Fonollosa, J. A. R., and Costa-jussà, M. R., “N-gram-based Machine Translation,” *Comput. Linguist.*, vol. 32, pp. 527–549, Dec. 2006.

[81] Melsted, P. and Pritchard, J., “Efficient Counting of k-mers in DNA Sequences using a Bloom Filter,” *BMC Bioinformatics*, vol. 12, no. 1, pp. 1–7, 2011.

[82] Monga, V. and Evans, B. L., “Robust Perceptual Image Hashing Using Feature Points,” in *PROC. IEEE Conference on Image Processing*, pp. 677–680, 2004.

[83] Nyberg, C., Koester, C., and Gray, J., “Nsort: A Parallel Sorting Program for NUMA and SMP Machines,” 1997.

[84] Olson, M. A., Bostic, K., and Seltzer, M., “Berkeley DB,” in *Proceedings of the 1999 USENIX Annual Technical Conference*, ATC ’99, (Berkeley, CA, USA), pp. 43–43, USENIX Association, 1999.

[85] O’Neil, P., Cheng, E., Gawlick, D., and O’Neil, E., “The Log-Structured Merge-Tree (LSM-Tree),” *Acta Informatica*, vol. 33, pp. 351–385, June 1996.

[86] Ousterhout, J., Agrawal, P., Erickson, D., Kozyrakis, C., Leverich, J., Mazières, D., Mitra, S., Narayanan, A., Ongaro, D., Parulkar, G., Rosenblum, M., Rumble, S. M., Stratmann, E., and Stutsman, R., “The Case for Ramcloud,” *Communications of the ACM*, vol. 54, pp. 121–130, July 2011.

[87] Peng, D. and Dabek, F., “Large-scale Incremental Processing using Distributed Transactions and Notifications,” in *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI’10, (Berkeley, CA, USA), pp. 1–15, USENIX Association, 2010.

[88] Qureshi, M. K., Gurumurthi, S., and Rajendran, B., *Phase Change Memory: From Devices to Systems.* Morgan & Claypool Publishers, 1st ed., 2011.

[89] Ranger, C., Raghuraman, R., Penmetsa, A., Bradski, G., and Kozyrakis, C., “Evaluating MapReduce for Multi-core and Multiprocessor Systems,” in *Proceedings of the 2007 IEEE 13th International Symposium on High Performance Computer Architecture*, HPCA ’07, (Washington, DC, USA), pp. 13–24, IEEE Computer Society, 2007.

[90] Rizzo, L., “A Very Fast Algorithm for RAM Compression,” *SIGOPS Oper. Syst. Rev.*, vol. 31, pp. 36–45, Apr. 1997.

[91] Rosenblum, M. and Ousterhout, J. K., “The Design and Implementation of a Log-Structured File System,” *ACM Trans. Comput. Syst.*, vol. 10, pp. 26–52, Feb. 1992.

[92] Roussopoulos, N., “View Indexing in Relational Databases,” *ACM Transactions on Database Systems*, vol. 7, pp. 258–290, June 1982.

[93] Sears, R., Callaghan, M., and Brewer, E., “Rose: Compressed, Log-Structured Replication,” *Proceedings of the VLDB Endowment*, vol. 1, pp. 526–537, Aug. 2008.

[94] Sears, R. and Ramakrishnan, R., “bLSM: a General Purpose Log Structured Merge Tree,” in *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, SIGMOD ’12, (New York, NY, USA), pp. 217–228, ACM, 2012.

[95] Seltzer, M., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., and Padmanabhan, V., “File system logging versus clustering: a performance comparison,” in *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON’95, (Berkeley, CA, USA), pp. 21–21, USENIX Association, 1995.

[96] Shalf, J., Dosanjh, S., and Morrison, J., “Exascale computing technology challenges,” in *Proceedings of the 9th international conference on High performance computing for computational science*, VECPAR’10, (Berlin, Heidelberg), pp. 1–25, Springer-Verlag, 2011.

[97] Spillane, R. P., Shetty, P. J., Zadok, E., Dixit, S., and Archak, S., “An Efficient Multi-tier Tablet Server Storage Architecture,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC ’11, (New York, NY, USA), pp. 1:1–1:14, ACM, 2011.

[98] Stevens, R. and White, A., “Architectures and Technology for Extreme-Scale Computing,” Tech. Rep. CMU-PDL-09-107, ASCR Scientific Grand Challenges Workshop Series, December 2009.

[99] Talbot, J., Yoo, R. M., and Kozyrakis, C., “Phoenix++: Modular MapReduce for Shared-Memory Systems,” in *Proceedings of the Second International Workshop on MapReduce and its Applications*, MapReduce ’11, (New York, NY, USA), pp. 9–16, ACM, 2011.

[100] Torralba, A., Fergus, R., and Freeman, W., “80 Million Tiny Images: A Large Data Set for Nonparametric Object and Scene Recognition,” *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, pp. 1958 –1970, Nov. 2008.

[101] Valiant, L. G., “A Bridging Model for Parallel Computation,” *Communications of the ACM*, vol. 33, pp. 103–111, Aug. 1990.

[102] Välimäki, N., Mäkinen, V., Gerlach, W., and Dixit, K., “Engineering a compressed suffix tree implementation,” *J. Exp. Algorithmics*, vol. 14, pp. 2:4.2–2:4.23, Jan. 2010.

[103] Vitter, J. S., "Online data structures in external memory," in *Algorithms and Data Structures* (Dehne, F., Sack, J.-R., Gupta, A., and Tamassia, R., eds.), vol. 1663 of *Lecture Notes in Computer Science*, pp. 352–366, Springer Berlin Heidelberg, 1999.

[104] Yu, Y., Gunda, P. K., and Isard, M., "Distributed Aggregation for Data-Parallel Computing: Interfaces and Implementations," in *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, ACM, 2009.

[105] Zaharia, M., Chowdhury, M., Franklin, M. J., Shenker, S., and Stoica, I., "Spark: Cluster Computing with Working Sets," in *Proceedings of the 2nd USENIX conference on Hot Topics in Cloud Computing*, HotCloud'10, (Berkeley, CA, USA), pp. 10–10, USENIX Association, 2010.

[106] Zaharia, M., Das, T., Li, H., Hunter, T., Shenker, S., and Stoica, I., "Discretized Streams: Fault-Tolerant Streaming Computation at Scale," in *Proceedings of the ACM SIGOPS 24th Symposium on Operating Systems Principles*, SOSP'13, 2013.

[107] Ziv, J. and Lempel, A., "A Universal Algorithm for Sequential Data Compression," *IEEE Trans. Inf. Theor.*, vol. 23, pp. 337–343, Sept. 1977.

[108] Ziv, J. and Lempel, A., "Compression of Individual Sequences Via Variable-Rate Coding," *Information Theory, IEEE Transactions on*, vol. 24, no. 5, pp. 530–536, 1978.

VITA

Hrishikesh Amur joined the School of Computer Science at Georgia Tech. in 2007 and worked on his doctoral dissertation under the able guidance of Dr. Karsten Schwan. His research interests broadly include operating systems and distributed systems. In particular, his recent work concerns systems and algorithmic techniques for resource efficiency in storage systems. He has enjoyed summer internships at Intel Labs Pittsburgh, the Parallel Data Lab at CMU, and IBM Research at Austin during his PhD. He was selected as a Key Scientific Challenges Scholar by Yahoo! in 2010. He received his undergraduate degree in Computer Engineering from the National Institute of Technology Karnataka.