Efficient, Scalable, and Versatile Application and System Transaction Management for Direct Storage Layers

A Dissertation Presented

by

Richard Paul Spillane

to

The Graduate School

in Partial Fulfillment of the

Requirements

for the Degree of

Doctor of Philosophy

in

Computer Science

Stony Brook University

Technical Report FSL-02-12

May 2012

Stony Brook University

The Graduate School

Richard Paul Spillane

We, the dissertation committee for the above candidate for the
Doctor of Philosophy degree, hereby recommend
acceptance of this dissertation.

Erez Zadok–Dissertation Advisor
Associate Professor, Computer Science Department

Robert Johnson–Chairperson of Defense
Assistant Professor, Computer Science Department

Donald Porter–Third Inside member
Assistant Professor, Computer Science Department

Dr. Margo I. Seltzer–Outside member
Herchel Smith Professor, Computer Science, Harvard University

This dissertation is accepted by the Graduate School

Charles Taber
Interim Dean of the Graduate School

Abstract of the Dissertation

Efficient, Scalable, and Versatile Application and System Transaction Management for Direct Storage Layers

by

Richard Paul Spillane

Doctor of Philosophy

in

Computer Science

Stony Brook University

2012

A good storage system provides efficient, flexible, and expressive abstractions that allow for more concise and non-specific code to be written at the application layer. However, because I/O operations can differ dramatically in performance, a variety of storage system designs have evolved differently to handle specific kinds of workloads and provide different sets of abstractions. For example, to overcome the gap between random and sequential I/O, databases, file systems, NoSQL database engines, and transaction managers have all come to rely on different on-storage data structures. Therefore, they exhibit different transaction manager designs, offer different or incompatible abstractions, and perform well only for a subset of the universe of important workloads.

Researchers have worked to coordinate and unify the various and different storage system designs used in operating systems. They have done so by porting useful abstractions from one system to another, such as by adding transactions to file systems. They have also done so by increasing the access and efficiency of important interfaces, such as by adding a write-ordering system call. These approaches are useful and valid, but rarely result in major changes to the canonical storage stack because the need for various storage systems to have different data structures for specific workloads ultimately remains.

In this thesis, we find that the discrepancy and resulting complexity between various storage systems can be reduced by reducing the difference in their underlying data structures and transactional designs. This thesis explores two different designs of a consolidated storage system: one that extends file systems with transactions, and one that extends a widely used database data structure with better support for scaling out to multiple devices, and transactions. Both efforts result in contributions to file systems and database design. Based in part on lessons learned from these experiences, we determined that to significantly reduce system complexity and unnecessary overhead, we must create transactional data structures with support for a wider range of workloads. We have designed a generalization of the log-structured merge-tree (LSM-tree) and coupled it with two

novel extensions aimed to improve performance. Our system can perform *sequential* or file system workloads 2–5× faster than existing LSM-trees because of algorithmic differences and works at 1–2× the speed of unmodified LSM-trees for *random* workloads because of transactional logging differences. Our system has comparable performance to a pass-through FUSE file system and superior performance and flexibility compared to the storage engine layers of the widely used Cassandra and HBase systems; moreover, like log-structured file systems and databases, it eliminates unnecessary I/O writes, writing only once when performing I/O-bound asynchronous transactional workloads.

It is our thesis that by extending the LSM-tree, we can create a viable and new alternative to the traditional read-optimized file system design that efficiently performs both random database and sequential file system workloads. A more flexible storage system can decrease demand for a plethora of specialized storage designs and consequently improve overall system design simplicity while supporting more powerful abstractions such as efficient file and key-value storage and transactions.

To God:
for his creation,
to Sandra:
the bedrock of my house, and
to Mom, Dad, Sean, and my family and friends:
for raising, loving, and teaching me.

Table of Contents

List of Figures

List of Tables

Acknowledgments

This work was made possible by the unending patience and tireless devotion of my wife Jianing Guo, who always pushed me to go farther, to make things done, and to keep to my commitments both to her and others.

Professor Erez Zadok provided encouragement, and a belief in my ideas and abilities when even I was unsure at times. His confidence in me gave me the confidence necessary to pursue some of the most technically challenging and important research I've ever explored, making this work possible. Professor Zadok is a rare type of advisor that gives his students sufficient rope to hang themselves with. Over the years he has given me countless opportunities to manage students and resources, to become a better team leader, a better project manager, a better scientist, and all in all, a better researcher, by simply letting me learn from my own mistakes in a forgiving environment.

My father, mother, and brother have been critical in supporting me, up to when they dropped me off at Stony Brook as a freshman, and throughout my years as a graduate student researcher. Its been tough, both personally and professionally, and when I need support, or a home to go to, my family was always there. My father would always ask about my research, and my mother would always actually listen. My brother who is also a computer scientist originally sparked the interest I have in me now that made me go beyond just playing video games.

Pradeep J. Shetty has been a tireless thinker and coder on this work. He challenges me to inspire him with the work, and because of my conversations with him, the work is well compared to other research projects, and its contributions are clearly delineated.

Charles P. Wright has been a close friend and mentor for my six years in the lab, even after he left he had a presence. I learned about transactions, and much of basic research from him directly. He has helped me with intern opportunities, he has acted as a sounding board, and we are friends. I thank him for everything he has done for me, and for all the help he has given me.

Sachin Gaikwad offered his independent mindedness and clear analysis and judgement of Valor's benefits and drawbacks. He was a fierce coder, and could hack the kernel quickly. I appreciated the nights he spent working with me on Valor.

Sagar Dixit worked with me side by side on algorithmic issues, figuring out how to avoid placing cursors in all levels to perform a fault in the virtual address space of a file. He carefully wrote difficult data-structure code, and fought bug after bug. Without his assistance, the work we did on external page caching, and tablet serving would not have been possible.

Shrikar Archak helped implement several novel kernel policies that allowed the kernel to notify applications when they were inducing memory pressure, and gave them an opportunity to flush, without risking resource deadlock. He also helped implement the trapping code to reroute system calls back to a user-level handler. Finally he was instrumental in testing and debugging the journal and recovery routines in the tablet server and database code.

Ramya Edara wrote the directory locking and per-process lock maintenance code in the kernel for Valor. Without her assistance we would not have been able to test concurrency as accurately in our evaluation, or have locking policies and permissions in place for security reasons.

Gopalan Suryanaryana helped with the implementation of `libvalor` and specifically with code to cleanup file descriptors. He helped me analyze the effects of using system transactions on the `etc` directory and its files. He was the first student I worked with as a researcher, and taught me much more than I taught him, and I thank him.

Manjunath Chinni expected the highest standard from me, and looked to me for leadership, and

in so doing, pushed me to promote the Valor project and drive it forward.

Vasily Tarasov has helped me countless times. He has been at the ready right before some of my most important deadlines to help me run benchmarks when I was undermanned. He is a good friend, and a reliable colleague.

Saumitra Bhanage was the master of Filebench, and helped set up countless benchmarks. He stayed up long nights before one of my toughest deadlines, and hacked through several critical bugs. Without his help, my research would have been a paper deadline behind.

Justin Seyster has been a close friend and confidant. We have laughed and joked in the lab, and he also wrote me some code just recently so he gets in here.

I thank Russel Sears, and Michael Bender, two co-authors whose papers I have read, and who I have worked with on past papers. Without their research, and their trust, I would not know much of what I needed to do my own research and make my own discoveries.

Leif Walsh, Karthikeyan Srinaivasan, and Zhichao Li have all written code for me, or helped me through a deadline. Without their help, some of my camera ready copies and submissions would not have been up to par.

I would also like to thank all the anonymous reviewers out there who have taken the time to anonymously read my papers, and anonymously educate me via trial by fire.

Finally I would like to thank my committee Dr. Margo Seltzer, Dr. Robert Johnson, Dr. Donald Porter, and my adviser, again, Dr. Erez Zadok.

Thank you everyone, education is a team sport, and we all help each other through it all, and we're all much better off for it.

Chapter 1

Introduction

The file system is the part of the operating system that provides a way for applications to easily read and write to a storage device. The typical abstraction provided to applications is the POSIX API [54], which can be used to manipulate files using routines such as `open`, `close`, `read`, and `write`.

The file system is optimized for workloads that read and write sequentially to large files. For other workloads, there are multiple other data structures underlying multiple software packages. To efficiently perform a non-file system workload, the user must first select the most efficient software package for that workload, and install it.

Maintaining multiple separate storage abstractions and implementations increases the complexity of the operating system, and introduces new overheads between storage layers that effect application performance dramatically. Consolidating these storage designs into one universal storage design would significantly reduce performance overheads, security vulnerabilities, and system instability, while greatly simplifying application level code.

Though this may not be possible, we attempt to consolidate at least two highly popular workloads into a single storage system: key-value storage and file system workloads. Key-value storage is a method of creating and utilizing large dictionaries of tuples on a storage device. We have found in our survey of related work that key-value storage is a powerful abstraction that is effective as a backend for implementing both file systems and database storage engines.

We also focus on obtaining efficient transactions for use with querying, modifying, and updating multiple key-value pairs in a single compound operation to simplify concurrent application-level programs manipulating storage. File tagging, meta-data management, and other workloads that mix more complex and randomly accessed data workloads (structured data) with more sequential workloads can be transactionally and efficiently performed without explicit specialized interfaces and implementations. By giving applications a single transactional interface to key-value storage that works efficiently for file workloads, applications can avoid carefully checking that other malicious applications are not observing them when performing sensitive operations. Package installation or large system-wide modifications can be performed without fear of leaving partial changes to the operating system in case of an error.

By providing a storage system that efficiently performs transactional key-value storage for

workloads ranging from large sequences of tuples (files) to small tuples (database entries) we reduce the plurality that complicates storage programming in the operating system.

In this work we explore a variety of possible transactional file system designs, with a focus on maximizing performance. For example, if a regular file system can perform a sequential write at the storage device's maximum measured sequential write throughput, then we strove for a transactional file system design that could achieve the same performance for that workload. If a database data structure, such as the log-structured merge-tree (LSM-tree) [95], is capable of randomly inserting small tuples at the theoretical optimal throughput [11, 18] for a target query throughput, then we strove for a transactional file system design that could achieve that level of performance as well.

We explored extensions to the operating system that enable existing file systems to perform application-level transactions through an extended POSIX API. We also designed a high-insertion throughput database with improved insertion and point-query performance that allowed transactions. Ultimately we generalized that design with algorithmic optimizations for partially and fully sequential insertions to better suit it for file system workloads. In this way we pushed toward a consolidated storage system that has the best properties of a transactional database and a highly efficient file system to simplify system storage.

Our exploration resulted in several new storage system designs that answered some of our research questions about storage system design. We enumerate the most important of these lessons. It also resulted in several new extensions and algorithms for key-value storage systems, such as sequentially inserting LSM-trees, multi-tier storage with LSM-trees, transactional file system support for existing file systems, and in related research with others [12], quotient filters.

The lessons we extracted from these experiences include:

- **Avoiding typical transaction overheads**: Performing sequential asynchronous over-writes and appends within a traditional transactional architecture is costly in comparison to a file system such as Ext3.

- **The transactional variation that fits your problem**: Ext3 was able to carefully optimize its transactional architecture for its specific needs, avoiding most overheads, and comparing favorably to Ext2, its non-journaling predecessor. This would not have been possible with a general approach, but only by building a carefully considered custom design and implementation.

- **Avoiding memory copy overheads for secure shared caches**: Caches used for file system or storage system data and meta-data play a large role in the performance of in-RAM workloads, and even partially or largely out-of-RAM workloads. If processes can directly address these caches within their own address space they perform much more efficiently than by sending and receiving messages through a shared memory pipe or socket.

- **Fast point queries are useful for file systems**: Most meta-data operations within a file system such as Ext3 are single element queries and can be heavily optimized and can perform well in certain write-optimized data structures.

- **Log-structured storage systems allow for a simpler cache implementation**: Normally, database systems must implement a separate user-level stand-alone page cache to enforce important reliability guarnatees. Log-structured transactional architectures can use `mmap` to

efficiently reuse the kernel's page cache and can therefore avoid having a separate and slower user-level stand-alone implementation.

- **Sequential and Multi-tier LSM-trees: a promising avenue for file system design**: LSM-trees obey or exploit many of the lessons learned about transactional storage architecture, and if improved for file system workloads, could serve as a promising alternative to existing extent-based file system designs.

The rest of this thesis is organized as follows. We begin by outlining our goals and research questions that motivated us to push toward a consolidation of transactional database and efficient file system designs in Chapter 2. Next, we explain background material related to our transactional file system extensions and generalized LSM-tree design in Chapter 3. In Chapter 4 we describe our transactional file system extensions. In Chapter 5 we describe our LSM-tree design and extensions. We further extend the LSM-tree to better support partially and fully sequential workloads in Chapter 6. Finally we conclude in Chapter 7 and discuss future work in Chapter 8. A glossary is provided in Appendix A for terms the reader may be unfamiliar with.

Chapter 2

Motivation

There are three key ideas that motivate our search for a transactional file system that is efficient for a broader range of workloads, spanning from small (64B or smaller) tuples to larger tuples (sequences of 4KB tuples or larger). These are:

1. Abstractions (Key-value Storage and Transactions)
2. Performance
3. Ease of Implementation

Section	Project Name	Summary
Section 3.3.1	SchemaFS	A prototyping framework and file system adaptation for Berkeley DB 4.4 that provides a transactional file system interface
Section 3.3.2	LSMFS	A framework for efficient user-level file systems with safe shared-memory caches and an example file system based on LSM-trees that stores meta-data in an LSM-tree, and data in an object store
Section 4.2	Valor	A set of operating system extensions explicitly designed to provide system transactions to existing file systems
Section 5.3	GTSSLv1	A database for heterogeneous storage and high update volume
Section 6.1	VT-trees and GTSSLv2	A file system design supporting system transactions and key-value storage, representing a culmination of lessons learned from previous trial designs

Table 2.1: ***Overview of trial transactional file system designs:*** *A brief high-level outline of the five trial storage and file system designs undertaken in this dissertation. The top two rows in the table includes shorter design vignettes. The bottom three rows refer to entire design sections.*

We describe our search for a more flexible transactional file system by explaining the purpose, design, related work, evaluation, and impact of several trial storage system designs. A brief outline of the five trial designs developed through the course of our research is shown in Table 2.1.

We discuss two of our trial designs, SchemaFS and LSMFS, in Chapter 3. Valor is a full transactional file system design and is designed to operate on top of any existing file system and provide transactions to applications without incurring performance overheads on non-transactional applications. GTSSLv1 is a database optimized for high insertion, update, and delete throughputs while still supporting efficient scans and optimal point queries. GTSSLv1 provides asynchronous transactions that are more efficient than existing approaches. We generalize the data structure GTSSLv1 is based upon to more efficiently execute sequential workloads.

These designs were explored as part of a search for a consolidated file system design that could offer both key-value storage and system transactions. Our earlier designs informed our later designs, and some designs were more important for our ultimate design for a transactional key-value storage file system than others. Specifically, Valor and GTSSLv1 had a larger impact on GTSSLv2's design than SchemaFS and LSMFS. Valor is described in Chapter 4, and GTSSLv1 is described in Chapter 5.

In this chapter, in Sections 2.1–2.2 we discuss what each of the Abstractions, Performance, and Implementation Simplicity factors means and show why these factors motivated our search through the various trial designs in Table 2.1. We also show how a file system design can score high or low for each factor. In Section 2.1 we introduce what we feel are the most important abstractions, and specifically, *system transactions*, which allow grouping of simpler storage operations into more complex operations and *key-value storage* which enables efficient storage and analysis of complex data on storage devices.

Motivating Factors These factors were chosen based on our experience with transactional and non-transactional file system design and our interpretations of the various works related to transactional file system, database, and storage design. These factors were chosen to be as orthogonal and as complete as possible: in our experience it would be difficult to compare a file system design to other file systems without knowing its relation to all factors (abstractions, performance, and implementation simplicity). Furthermore, in our experience a file system design can have a high or low score for one of the factors, without necessarily having a high or low score for one of the other factors. These factors make it possible to compare different storage system designs, but they say nothing about whether certain designs actually exist. For example, in practice it is difficult to have a system with flexible key-value storage including files, transactions, a simple implementation, and also good performance. The goal of our research has been to move in this direction: to find a consolidated storage design which maximizes all three of our factors. The designs we arrived at can be applied to many storage contexts, and the lessons we have learned from this motivate our generalization of the LSM-tree.

2.1 Important Abstractions: System Transactions and Key-value Storage

Abstractions define what additional logic applications must include to store and retrieve data from the file system. The measure of a good abstraction is to reduce the complexity of applications, while permitting them to provide their intended features. An example file system with no abstractions might be an interface for applications to send and receive messages using the protocol

natively understood by the storage or memory technology being used. An example file system with a robust set of abstractions might be one that behaves more like a database than a typical file system. The de facto abstraction in file systems today is a hierarchy of names, where each name is associated with a large array of bytes. In this work, we focus primarily on providing *system transactions*, an abstraction that makes it easy for applications to perform complex storage operations by composing together simpler storage operations. These composed operations behave in a reliable and predictable manner in the presence of concurrency and despite the possibility of software or hardware faults. By expressing their application's storage logic in terms of system transactions, programmers can remove a large amount of storage-related security and crash-recovery code from their applications.

2.1.1 Transactions

A transactional file system allows applications to group multiple storage operations into a single operation that behaves in a reliable and predictable manner. Such a group of operations is a *transaction*. When the storage operations are POSIX file system operations, then these transactions are called *system transactions*. Both transactions and system transactions obey the same basic four rules. These rules are commonly referred to with the four letters *ACID*. The rules are briefly explained here for convenience. However more extensive and precise descriptions can be found elsewhere [43].

- **Atomicity** Either every operation in the group will be performed as intended, or none of the operations will have any effect on the system.
- **Consistency** If every transaction transitions the storage system from one valid state to another, the storage system will always be in a valid state, despite transactions running in parallel or detectable faults.
- **Isolation** Transactions can be programmed as if they are the only operation occurring on the storage system until the transaction commits. Isolation and Consistency are related, and there are various definitions of Isolation (levels [43]) that can increase performance in contentious scenarios.
- **Durability** When a commit operation completes, enough data has been written to disk to ensure that the transaction will not be undone, even in the face of a detectable fault.

Transactions, and in the case of file systems, system transactions are excellent abstractions because all the ACID properties in concert make it possible for developers to remove a great deal of complexity and code from their applications. In one of our trial designs that provided applications with system transactions, called Amino [151] (see Appendix A), we were able to reduce the complexity of a mail inbox management application, `mail.local`, by a factor of six. We measured complexity using lines of code and McCabe's cyclomatic complexity metric [78]. To see how the ACID properties can reduce an application's complexity, consider each of the following four properties:

- **Reducing Complexity with Atomicity** If an error or a detectable fault occurs, the developer does not have to program any logic to clean up or undo partially applied operations performed

before the error was detected. Instead, the transaction can be aborted, and atomicity ensures the failed operation had no effect on the state of the storage system.

- **Reducing Complexity with Consistency** If every transaction transitions the storage system from one valid state to another, it is sufficient to execute the system's recovery or abort algorithm to ensure the storage system is in a valid state after a system or application crash. Consequently, fewer post-crash checks need to be implemented by the application developer [19, 39] and it is easier to reason about the reliability and fault tolerance of the storage system.

- **Reducing Complexity with Isolation** The developer does not have to acquire locks on storage resources or spend time debugging complex race and deadlock scenarios. An application cannot accidentally make itself vulnerable to malicious alteration of data half-way through a transaction [106, 151].

- **Reducing Complexity with Durability** The developer does not have to identify which caches or devices to flush [8, 93, 152], and the system can often group together many slow flushes across unrelated applications running at the same time into a single flush to the storage medium.

Transactional Model To completely understand what features the application developer can now expect from the storage interface, the resources in the operating or storage system that are protected by the umbrella of the ACID properties need to be clearly delineated. This is for two reasons:

1. Developers can identify what portions of their program can be expressed with system transactions and therefore recognize they do not have to write recovery, locking, or flushing code for that portion.

2. Conversely, developers can identify which caches and interfaces (e.g., network connections, user interfaces, etc.) require explicit error handling.

A list of which resources and operations are considered part of a system transaction is called a *transactional model*. In Chapter 4 we describe a system that supports system transactions on files, directories, and other file system operations which can alter, create, or delete files.

In Figure 2.1 we see an example of a simple application of a system transaction system call. The application is easily able to undo the effects of the `open` system call, as well as the `write`, if an error occurs while logging in `log_create_in_application_log()`, by just calling `sys_txn_abort()`. The `log_create_in_application_log()` routine is an application-level routine, not part of the transactional system, and is an example POSIX operation. Some of the variable names are just specific to this example only (e.g., `handle_t`, `configs`).

Notably, this transactional model does not include processes, network connections, signals, memory, and many other operating system objects and operations on these objects. In this work we focus on the task of designing a transactional file system that is expressive, fast, and sufficiently elegant in design, and this is already an open research question which we address. Additional efforts to extend the system transaction model to other areas not within the file system is outside the scope of this dissertation. Such efforts either incur upwards of $2\times$ overheads which we avoid

```
int create_config_file(handle_t handle)
{
  int ret = 0;
  ssize_t wret;

  ret = sys_txn_begin();
  assert(!ret);

  int fd = open(configs[handle].name,
                O_RDWR | O_CREAT, 0600);
  if (fd < 0) {
    ret = -errno;
    goto out;
  }

  wret = write(fd, configs[handle].contents,
               configs[handle].sz_contents);
  if (wret != configs[handle].sz_contents) {
    ret = -EIO;
    goto out;
  }

  ret = log_create_in_application_log(configs[handle].name,
                                      "create");
  if (ret)
    goto out;

  /* Closes the file and undoes the write and log append */
  ret = sys_txn_commit();
  assert(!ret);
out:
  if (ret)
    sys_txn_abort();
out_txn:
  return ret;
}
```

Figure 2.1: ***Example of a system transaction:*** *An internal application routine to create a configuration file based on information related to* `handle`*. Either both the file creation and log entry are performed, or nothing in the file system is changed. Error cleanup is easily done with a single call to* `sys_txn_abort()`*.*

or reduce [106, 147] or require rewriting too much of the operating system [119]. However, such efforts are complimentary to our goals, as our final proposed design is completely separable from the kernel, and could cooperate with most of these systems.

2.1.2 Key-Value Storage

System transactions are a powerful abstraction and are the focus of our research and this dissertation in large part. However, another abstraction that we consider in the design of our trial file systems and storage systems is a key-value storage interface, which naturally compliments transactions as repeatedly proven in the database field [1, 107, 130]. This form of interface is different from the typical POSIX file system interface in three ways:

1. Key-value stores are typically designed to store at least billions of objects on a single machine. File systems have historically not been optimized for extremely large numbers of files, although more recent file system designs [139,140] are capable of this, at least in theory. XFS benchmarks show that unsuitable file system data-structures make the insertion of a billion objects infeasible in practice (see Section 3.3.2, Figure 6.15).

2. Key-value stores can efficiently store small pieces of information. Typically, file systems make many design decisions that assume a file's contents is at least a page (4KB currently) in size. For example, the size required to store a small file in cache is typically at least a page.

3. Key-value stores support fast lower and upper bound binary (or B-ary) searches for an object, and fast scans of objects belonging to a particular sub-set. This makes feasible more complex queries that perform a series of unpredictable searches.

These advantages make key-value stores an excellent abstraction for applications that operate on large quantities of meta-data or large sets of objects that are inter-related in many different ways. For example, a storage system may want to index video data by automatically generating tags with visual recognition algorithms, and then storing the video and these tags in the file system. Such an application can avoid errors by ensuring that tags and video contents are updated transactionally, and dependencies between the two types of data are consistently maintained across detectable faults. However, storing such tags using small files would prove complex and error-prone for application developers for the reasons listed above. Instead, developers would prefer the use of a key-value store, or perhaps even an abstract query interpreter layer on *top* of a key-value store for their tags, and a file system for their video data. Furthermore, developers would want it all to be protected by system transactions. If a file system provided a key-value store interface in addition to system transactions, this would enable a new class of applications to operate more efficiently, and to be programmed more simply.

Another critical application for a key-value store in the file system is the reduction of countless lines of parsing code for parsing, reading, writing, and maintaining configuration files. These parsers exist because structured data, such as a Makefile or configuration file, is stored in an unstructured format. However, if stored in a key-value store, this code would be redundant, and applications could avoid many potential bugs and redundant code, a good sign that key-value stores are an excellent abstraction for applications. As an example, consider SQCK [44] which converts

Ext2 file system meta-data into a key-value format and is able to perform SQL queries to find inconsistencies. This method is less error-prone than the C querying code in Ext2's FSCK program and is able to find errors not found by FSCK. Storing complex data in a structured format lends itself to cleaner and simpler applications, and that is the point of a useful system abstraction.

2.2 Performance

Performance is how quickly a file system can execute the most common and important workloads. This factor is one of the easiest to quantify for any particular file system design, and is typically a very important goal for any storage system. In this work we develop several file system designs that always strive for good performance, but sometimes must compromise in order to support better abstractions, or a simpler architecture. One important goal of this dissertation is to show how to maintain very good performance, *without* foregoing system transactions, or key-value pair storage.

Many modern applications need to store structured data along with traditional files to help categorize them. Traditional files includes video clips, audio tracks, and text documents. Structured data represents large numbers of interrelated smaller objects. Examples include media tags such as a photograph light conditions and object placement, or an MP3's performance, album cover, etc. File systems handle traditional file-based data efficiently, but struggle with the structured data. Databases manage structured data efficiently, but not for large file-based data. Storage systems supporting both file system and database workloads are architecturally complex [38, 83]. They use separate storage stacks or outside databases, use multiple data structures, and often are inefficient [137].

A system that can efficiently process varying workloads, will be complex if its storage design uses multiple, heterogeneous data structures. No known single data structure is flexible enough to support both database and file system workloads efficiently.

Read-optimized stores using B-trees have predictable performance as the working set increases. However, they hurt performance by randomly writing on each new insert, update, or delete. Log-structured stores using B-trees efficiently insert, update, and delete; but they randomly read data if their workload includes random writes and is larger than RAM. Log-structured merge-trees [95] (LSM-trees) have neither of these problems. Furthermore, LSM-based transactions are inherently log structured: that is why LSMs are widely in many write-optimized databases [6, 66]. Still, current LSMs do not efficiently process sequential insertions, large key-value tuples, or large file-based data.

We extended the LSM to support highly efficient mixes of sequential and random workloads, and any mix of file-system and database workloads. We call this new data structure a *VT-tree*. We discuss VT-trees further in Chapter 6.

2.3 Implementation Simplicity

Adding new abstractions to a file system's interface typically requires new algorithms, data structures, and a general increase in complexity to provide these features. This increased complexity can make it difficult to modify the file system later if the target workload or abstractions have to change. Some of the most daunting systems research questions ask how to extend a widely used

and popular system with just enough modifications to enable some new abstraction or increase performance [19,36,83,86,105,106,125,135]. Therefore, the number and complexity of modifications to other operating system components in order to support a new file system design also contributes to that design's overall architectural and implementation complexity.

Chapter 3

Background

Designing a consolidated storage system that is capable of small to large key-value tuple storage, system transactions and workloads ranging from database to file system workloads, has required the study of many past systems. We surveyed previous work and applied what we learned to building a number of prototype systems. We then took what we learned from those systems, bundled it together with our knowledge of previous work, and embarked upon the designs that are the ultimate topic of this thesis. We break our related work into two parts: the work of others (related work) and our own work (trial designs):

Related work which includes other past systems that have features, research goals, design decisions, or data structure choices that informed our own. Some of these are older systems we have worked on.

Trial designs includes systems that are like related work, but that we were responsible for developing, allowing us to benefit from the experience of working on these systems. The lessons learned from this experience shapes our current transactional and data structure design.

Previous techniques for protecting meta-data writes within the file system have been extensively studied [127]. However, we wanted to extend system support for transactions to applications. This required more powerful transactions that could be indefinitely large, support for deadlock detection for directory and file locks, and changes to the buffer cache implementation that would permit at least a limited form of write ordering. Early in our research and to the best of our knowledge, we showed for the first time how to add support for these features for any underlying file system with a small number of changes to the operating system's caching code. The system we constructed is called Valor and is discussed in Chapter 4. We discuss write-ahead logging algorithms (i.e., WAL algortihms), other logging algorithms used by transactional systems such as Valor, and work related to Valor in Section 3.1.

We found that many traditional file system workloads that would benefit from transactions executed 2–3× slower on a WAL-based transactional file system (see Chapter 4). Furthermore, although Valor's design offered a way to transactionally protect meta-data updates, it did not provide to user-level applications a way to efficiently store key-value pairs that integrates well with its

file system transactional API. Based on these conclusions we decided to pursue a new underlying data structure that would efficiently support sequential file workloads and more complex key-value storage workloads. We used the log-structured merge-tree (LSM-tree). We review LSM-trees in Section 3.2.

Section 3.3 discusses two trial designs and reflects on a few lessons learned from each of these designs; these influenced the ultimate design that we push toward in Chapters 5 and 6. Finally, we outline important transactional and data structure design decisions in Section 3.4 where we introduce Table 3.1 that lists these decisions. We come back to Table 3.1 in subsequent Chapters 4, 5, and 6 where we develop and describe our transactional and data structure design ideas in detail.

3.1 Transactional Storage Systems

Transactions are a useful programming abstraction and have been used in many storage systems. Some systems use transactions only internally for certain types of data and other systems expose transactions to the application layer so they can be used by external applications. Journaling file systems use transactions internally to store meta-data updates to the file system. Databases and transactional file systems allow external applications to use transactions. Alternative approaches to using transactions also exist, such as exposing write-ordering primitives to the application layer.

In Section 3.1.1 we introduce the basic concepts of logging within a transactional storage system. In Section 3.1.2 we discuss journaling file systems, in Section 3.1.3 we discuss databases and their influence on our own storage system designs. We also outline issues arising from different APIs, performance limitations, the impact of soft-updates on transactional storage, and previous work on integrating an LFS journal into database applications in Section 3.1.3.

3.1.1 WAL and Performance Issues

The family of logging algorithms used by most databases is the write-ahead logging family of algorithms (WAL). WAL algorithms write modifications to data to a separate log file before performing those modifications on the data. Writing modifications to a log before performing those modifications on data is called *logging*. Logging algorithms that follow this procedure are called *write-ahead* because they write the log record *ahead* of performing the modification to the data. Databases can recover from partially executed transactions by comparing the contents of the database with the contents of the log, and determining what must be done to restore the database to a consistent state. This procedure is called *recovery*.

The WAL family of algorithms has several well-known variants [43]. In this thesis we refer to two variants:

ARIES-like The ARIES variant of WAL specifies how to reliably undo a transaction (multiple times if necessary) and cooperate with the operating system's and database's caches in an efficient way [85]. We describe ARIES and WAL in Appendix A as well. Traditional databases such as Berkeley DB and InnoDB use ARIES-like write-ahead logging algorithms. The Valor design in Chapter 4 is ARIES-like.

redo-only Redo-only means that this variant need only ensure that transactional operations can be re-executed in case of a crash. Redo-only variants do not have to write as much data

to storage as ARIES-like variants, but are less flexible and typically do not provide larger-than-RAM transactions. Redo-only algorithms are used to improve performance for write-heavy workloads but typically do not allow for many large transactions. Cassandra [31] and HBase [28] use a redo-only approach.

Some of the algorithms in this thesis are not strictly WAL. One popular variant of a non-WAL logging algorithm is the journaling algorithm employed by Ext3 for file appends. We call this variant of logging *Ext3-like* logging and it is used in log-structured copy-on-write databases and file systems [30, 51, 139] as well.

Ext3 uses both a redo-only approach for meta-data writes, as well as the Ext3-like approach for atomic file appends. To see how Ext3, and in a larger sense Redo-only can be useful, consider how Ext3 protects meta-data updates. Ext3 must overwrite meta-data tables on the storage device, but these record updates cannot be only partially executed or the file system's state will become corrupted. To avoid this, Ext3 logs meta-data modifications *ahead* of executing these modifications on the actual meta-data tables. If the file system crashes at any point, Ext3 can simply check to see what pending modifications are in the log, and re-execute those modifications on the meta-data tables, thus overwriting any partial writes on the table that failed to fully complete due to the crash. If the meta-data modifications were not logged ahead of being executed, then there would have been no way to re-execute those modifications during recovery. This is why redo-only is a member of the WAL family of logging algorithms: it must ensure that log records are always written ahead of writes to the actual data. Databases typically use some WAL algorithm and often ARIES.

As mentioned before, Ext3 is not strictly in the WAL family because of how it performs atomic file appends. The Ext3 journaling file system actually ensures that *data* modifications are performed before the log writes are flushed to disk. This is required if Ext3 wants to avoid having junk data at the end of a file after recovery and is the default mode that Ext3 operates in (ordered mode [144]). To see why Ext3 does this, consider the case of a process appending to a file. Ext3 ensures that data appended to the end of a file is on disk before that file's meta-data shows an increased file length. If Ext3 did not do this, it could show junk at the file's end after recovery. If Ext3 did not order data writes before the log is flushed, the file meta-data modification would be logged, but the appended data might not have been written before the file system crashed. In such a scenario, Ext3 would mistakenly replay the logged file meta-data modification, but without the appended data, the end of the file would only show the contents of unused storage space. Ext3 avoids this by requiring that data is written first in its ordered journaling mode. A strictly WAL-based approach that writes log writes ahead of data writes would have to record at least the new data being appended to the end of the file. In our above example, this would mean that newly appended data would be written at least twice (once to the log and once to the end of the file). Cassandra and HBase are examples of strict WAL-based systems that must write twice for newly inserted data [20, 31].

Other systems swap the order of a typical WAL order of log writes and data writes; they include log-structured copy-on-write systems [30, 51], and LSM-tree-based log-structured storage systems such as GTSSLv1 and GTSSLv2 from Chapters 5 and 6. In general, logging algorithms that "point" to extents of data typically ensure that the extent is entirely on the storage device before writing the "pointer" to the log, thus inverting the typical write-ordering relationship required by the WAL family of algorithms. For example, Ext3 effectively points to appended data by increasing the length of the file in the file's meta-data and so it ensures that appended data is on the storage device before initiating the update of the file's meta-data by writing to the log.

A strictly WAL-based approach can also fully recover from a crash as long as all modifications are logged. To support recovery from a crash after modifying data (as opposed to meta-data), the system merely needs to log those modifications before performing them. However, this requires that all modifications are written at least twice, once to the log, and again while being executed. All WAL algorithms need to write a record of the modification before executing it. There are some opportunities to optimize the WAL algorithm; in general, however, a WAL or ARIES-like approach incurs at least a 2–3$\times$ overhead in comparison to a file system performing an asynchronous sequential write to the storage device.

Conversely, the Ext3-like approach avoids multiple writes of data within extents by pointing to extents from the log without having to write those extents to storage more than once. This optimization is only possible if it is not necessary to recover the contents of storage overwritten by the extent. This is the case if that storage was unallocated before being overwritten (there is no point in recovering the contents of unallocated storage). This is the case for log-structured systems or Ext3 file appends.

WAL algorithms that support arbitrarily large transactions that can overwrite data must also gather undo records. Undo records are the opposite of redo records. While a redo record tells the WAL algorithm how to re-execute a failed operation, an undo record tells the WAL algorithm how to un-execute or undo a failed operation. The general-purpose ARIES WAL algorithm logs both undo and redo records. If a WAL algorithm must gather undo records, then it must perform random reads for workloads that are much larger than RAM and perform random inserts, updates, or deletes. Contemporary log-structured approaches do not depend on undo records [20].

We wanted to avoid the double or triple-write overheads of the WAL family of logging algorithms file system and sequential workloads. We wanted to develop an approach that can be extended to support larger-than-RAM transactions. We wanted to support high-throughput insertions, updates, and deletes to facilitate efficient write-only transactions for tasks such as indexing data [20].

We wanted to avoid the double- or triple-write overheads added by the WAL family of logging algorithms for all workloads, while still allowing for larger-than-RAM transactions. This way we could support high-throughput insertions, updates, and deletes and provide efficient write-only transactions for tasks such as indexing data [20]. In Chapter 4 we demonstrate the overheads that the WAL family of algorithms places on sequential asynchronous file system workloads by examining several implementations as well as an idealized mock implementation of ARIES. In Chapter 5 we describe the design of a log-structured database that dynamically switches between the Ext3-like approach for large or asynchronous transactions and the redo-only approach for small, durable, and concurrent transactions. This design can be extended to support larger-than-RAM transactions which we discuss in the implementation of GTSSLv2 in Chapter 6.

3.1.2 Journaling File Systems

Non-transactional applications often resort to post-crash checking programs that must be run to check for errors in the application's storage and fix those errors. Non-journaling file systems operate in a similar way. After a crash, a file system check [81] tool is run to detect and fix errors. There are several problems with this approach: (1) it is difficult to show that the checking tool found all possible errors; (2) it is difficult to predict what the state of storage will be after the checking tool fixes any errors it found; and (3) the tool must typically scan the entire storage system to verify

that all is well. There are other additional problems, such as detecting and recovering from faults while the checking tool is running, or maintaining a largely separate but redundant checking tool source code [44]. However, a separate checking tool is still useful in case of hardware faults, or other kinds of faults not detected by the transactional implementation [143].

Transactional systems avoid many of these problems. As long as each operation transitions the storage system to a valid state, the system will always be in a valid state and will have no formatting errors after recovery is run. Recovery is a well studied and described process in transactional research [43]. If a transaction commits, it is guaranteed its effects will not be undone during recovery and it is easier to predict and limit what updates are jeopardized by a crash. The recovery log is typically much smaller than the entire storage system and just the most recent pending operations need to be replayed from the log after a crash. Transactional systems also safe-guard against crashes that occur during recovery and the code needed to replay operations in a correct manner is hidden within the transactional implementation and does not require a separate source code.

Journaling file systems [19, 39] harness the advantages of transactional systems, but make careful design decisions that avoid most of the performance overheads of using a more general-purpose transactional storage design [43]. Journaling file systems avoid these overheads by introducing a log and using a particular brand of transactions well suited to meta-data accesses: the only kind of operation the file system performs transactionally.

The kind of transaction supported by a journaling file system is a redo-only transaction [39]. By performing redo-only operations and only for meta-data accesses, the file system need only write a meta-data record twice when a record is updated; regular data writes are still written only once. The limitation of redo-only logging is that a partially committed transaction cannot be undone (with redo-only, only complete transactions can be re-done) and so transactions must be logged completely before they can be flushed to the storage system. This requires transactions that can fit entirely within RAM [144]. This restriction is not a problem for journaling file systems that make only small meta-data updates relative to sequential data reads and writes. Journaling file systems use a no-steal no-force caching policy which means that incomplete transactions are not flushed and writing only occurs periodically for safety, or when there is a shortage of memory in the system. The transactional implementation used in journaling file systems is not exposed to applications and so applications must use a separate library or database software package to use transactions.

Journaling is the dominant solution for obtaining an efficient and predictable file system recovery process and is used by major file systems such as Ext3 [19], NTFS [82], XFS [140], ZFS [139], HFS+ [7], and JFS [62]. It is not the only method of recovering a file system from a crash and we discuss one popular variant, soft updates [36], in Section 3.1.3. File-system–specific recovery is an ongoing research area: journaling and journal recovery represent only one way of solving many common file system recovery problems. Journaling is not the final word on efficient file system checking and recovery after a crash.

Like journaling file systems, we want to protect the file system's meta-data information, but we also want to protect other important meta-data maintained by other applications. We want to allow applications the ability to perform complex transactions that consist of many POSIX and tuple operations, not just `inode` updates, for example. We want to permit transactions to be larger than RAM or long-running. Chapter 4 shows how to build a transactional POSIX API on top of existing file systems. The design discussed in Chapter 4 is no more than 34% slower for asynchronous sequential workloads than the general-purpose transactional design used by most databases, ARIES [43]. Chapter 5 shows how to build a transactional API for tuple storage on top

of an LSM-tree database architecture that avoids write-ahead logging overheads (including redo logging) for asynchronous, sequential, or large transactions and permits compound smaller-than-RAM transactions for many tuple operations. In Chapter 6 we show how to extend LSM-trees to support file system workloads and make it possible to support transactional operations on large transactions, sequential workloads, and file operations. In future work we outline how the beneficial features of each of these approaches can be combined.

3.1.3 Database Access APIs and Alternative Storage Regimes

The other common approach to providing a transactional interface to applications is to provide a user-level library to store data in a special page file or B-Tree maintained by the library. Berkeley DB offers a B-Tree, a hash table, and other structures [130]. Stasis offers a page file [120]. These systems require applications to use database-specific APIs to access or store data in these library-controlled page files.

There are two difficulties with using database access APIs. The first difficulty is unwieldy performance for file system and write-heavy random-access workloads. The second difficulty is efficiently interfacing with the POSIX API so that system applications can easily adopt and utilize efficient transactions.

Soft-updates and Featherstitch Soft-updates were developed as an alternative to journaling described in Section 3.1.2. Soft-updates avoid additional writes to a journal but require running garbage collection after a crash to recover potentially lost space. Garbage collection can be run in the background while the file system is in use.

Featherstitch developed by Frost et al. [33] explores the idea of generalizing soft updates by permitting writes that may be smaller than a block and allowing user-level applications to specify write dependencies in addition to the file system. Featherstitching requires non-trivial kernel modifications and support for an SMP kernel configuration is not present in their current implementation.

In Frost et al.'s use-case studies [33], applications that benefit from featherstitch already perform `fsync` in the context of a recovery framework that the application has already implemented in case of crash. In these scenarios write-ordering for atomicity instead of waiting on a forced flush with `fsync` improves the performance of these applications. The authors describe how featherstitching can be used to underlie a journaling implementation, but no direct comparison between featherstitching and an asynchronous transactional system is made either in terms of semantics or performance. Forming dependency cycles in the application layer of Featherstitch is an error and it is left as future work to check for cycles in this layer. Conversely, serializability of transactions is enforced with dead-lock detection in the locking layer of transactional systems, requiring no checking by applications. Support for ACID transactional semantics or more efficient log-structured-like shadow-paging is left as future work.

Integration of LFS Journal for Database Applications Seltzer et al. develop an in-kernel logging system that is used both as part of a log-structured file system and also to act as an in-kernel journal for a database at the user-level [126]. They show with a simulation that logging overheads can be reduced when using an in-kernel logging agent for a database. This performance gain helps

to explain performance differences between Valor and a user-level logging library such as Stasis [120] as discussed in Chapter 4. In this thesis we discuss how to extend the VFS to support transactional POSIX operations on top of any file system and generalizations to the widely used log-structured merge-tree to enable it to better support a variety of workloads including sequential and file system workloads. In this way we address a set of different problems that go beyond the initial placement of a logging component in the kernel, but Seltzer's work buttresses the design decision to place Valor's journal within the kernel for performance reasons.

3.2 Explaining LSM-trees and Background

We discuss transactional and data structure design decisions to enable key-value and transaction support in a file system. To support a variety of workloads, ranging from indexing workloads, to read-modify-update transactions, to file system workloads, we develop a generalization of a widely used contemporary log-structured data structure that is capable of better handling sequential workloads. The log-structured data structure that we generalize is the log-structured merge-tree or the *LSM-tree* [95].

The LSM-tree is used for faster insertion performance by HBase, Cassandra, Big Table, and other systems such as Hypertable [52]. The LSM-tree is not a traditional tree structure exactly, but is similar to a tree. It differs in that an LSM-tree is actually a collection of trees. This collection is treated as a single key-value store. LSM-trees writes sorted and indexed lists of tuples to storage, which are then asynchronously merged into larger sorted and indexed lists using a *minor compaction* or *major compaction*. Minor and major compaction do not differ in functionality, but rather in purpose. A minor compaction is when the merging occurs to bound the number or size of sorted indexed lists on storage. A major compaction operates similarly to a minor compaction except that it is scheduled to run periodically and is intended to ensure that all lists are eventually compacted within a set time-period. For example, some sensitive data may need to be removed within 24 hours of marking it as deleted. However, the data will only be removed during compaction and no minor compaction has yet compacted the lists containing this sensitive data. Without scheduling a compaction of all lists, this sensitive data would continue to persist. Scheduling an additional compaction that occur every 24 hours is an example of a major compaction.

There are many variants of the LSM-tree but we focus on two variants: (1) the COLA [11], used in the popular HBase database [28], and (2) the SAMT which we analyzed and extended in our own database, GTSSLv1, and is also used in Cassandra [1].

So the reader may better understand our discussion on background material surrounding our LSM-tree extensions and generalizations, we introduce and explain the DAM model and B-trees in Section 3.2.1. We explain the LSM-tree's operation in Section 3.2.2 and then discuss background material in Section 3.2.3.

[1] Although we independently discovered the SAMT as described here, we found by inspecting Cassandra's source code that they also use a data structure similar to the SAMT. The exact details of this data structure were only roughly outlined in Cassandra's documentation. However, its asymptotic performance is different from other customarily used LSM-tree implementations and merits more than a cursory outline. We offer a full analysis of the SAMT in Chapter 5. We also generalize and extend the SAMT to the MT-SAMT in Chapter 5 and discuss further generalizations to support efficient sequential workloads in Chapter 6. However, since the SAMT is used in Cassandra, we describe its basic operation here. The reader may better understand our discussion of background material related to LSM-trees given a basic understanding of the SAMT which can serve as a canonical LSM-tree.

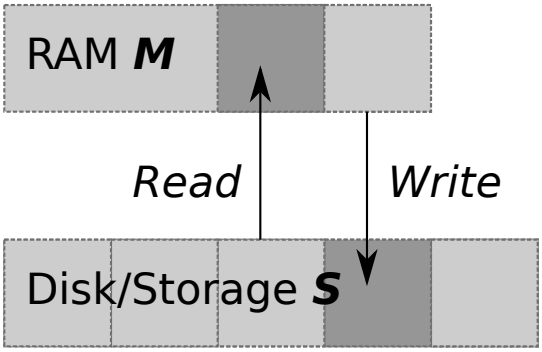

Figure 3.1: ***The DAM model:*** *The DAM model only allows block transfers of size b bytes or B elements from S to M or vice verse.*

3.2.1 The DAM Model and B-trees

We explain the LSM-tree by explaining the SAMT. When a term or object has a name in common use in related work, we make a note of it and also refer to the glossary where all such synonymous terms are listed as well. To explain the SAMT we first introduce a simple explanation of the DAM model, the B-tree, and important sub-components and terminology used in the SAMT and most other LSM-tree implementations.

To understand why LSM-trees are different from other on-disk data structures and why they are organized the way they are, it is necessary to understand the cost model that is assumed when designing the LSM-tree data structure and its variants known as the disk access model or *DAM* model [11]. Figure 3.1 illustrates the DAM model. In the DAM model, there is a RAM M, and a disk or storage device S. Device S is divided into a series of blocks, and it costs 1 to transfer a block from M to S or vice verse. All blocks are the same size B. The convention in related work is to use B to indicate the size of the block in elements, where as b is used to indicate the size of the block in bytes.

SAMTs are composed of Wanna-B-trees. A Wanna-B-tree is a simplified B-tree that is sufficient as a component of a SAMT, but is much simpler to implement. We now explain what a Wanna-B-tree is and introduce important terminology that will be used throughout the rest of this thesis and particularly in Chapters 5 and 6.

A Wanna-B-tree is a limited version of the B-tree. The B-tree is a balanced tree structure which is optimized for lookups on storage devices which can be accurately modeled using the DAM model. The B-tree is composed of index and leaf nodes. In Figure 3.2 we define an *index node* as any list of key-value pairs, where the key is the key type of the tree and the value is an offset pointing to another node in the tree. We define a *leaf node* as any list of key-value pairs, where the key is the key type of the tree, and the value is the associated value of that key.

The simplified B-tree in Figure 3.2, panel ①, makes efficient use of each block transfer of B_{index} index nodes. B-trees hold B_{index} index nodes in a block of size B, and B_{small} tuples in a block of the same size B. By configuring its arity to B_{index} it is able to eliminate the near maximum number of sub-trees that cannot contain the sought-after tuple with each block transfer until it retrieves the block containing the sought-after tuple. One minor optimization not shown in Figure 3.2 or discussed here is that the first index node in a group of index nodes can actually hold one more pointer to a sub-tree of tuples whose keys are less than all keys in the group of index nodes. For

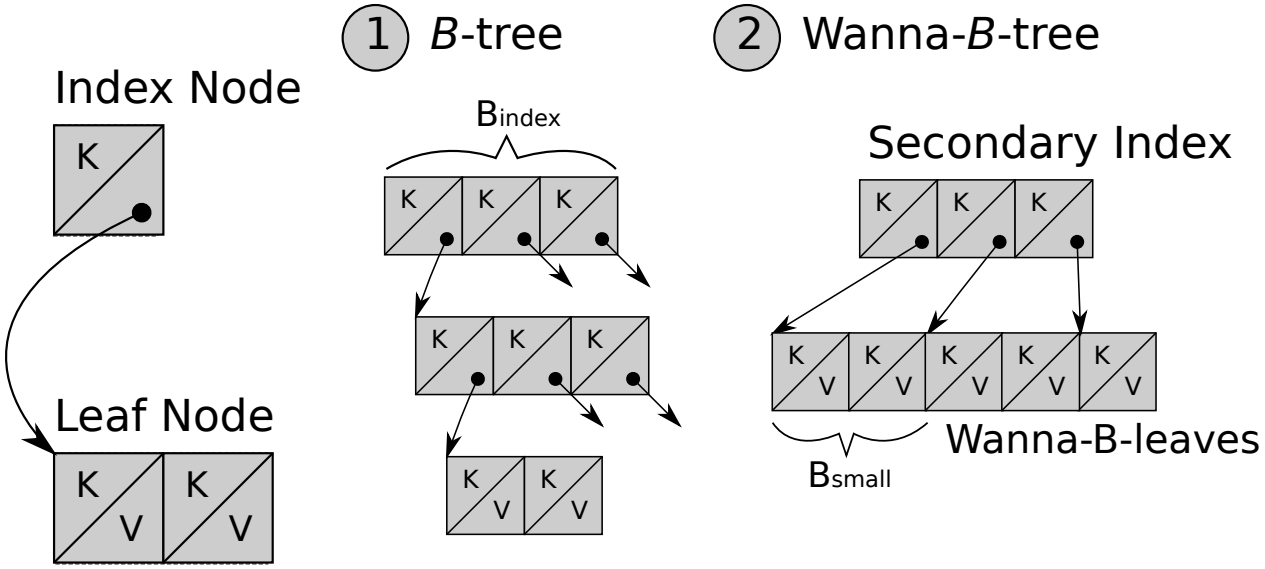

Figure 3.2: ***Definitions:*** *There are two types of nodes,* index *and* leaf *nodes. You can construct a B-tree or a much simpler Wanna-B-tree that only has a single layer of index nodes, called a* secondary index. *The leaf nodes that comprise a Wanna-B-tree are called Wanna-B-leaves. For best performance, Wanna-B-leaves should be contiguously stored together in sorted order on storage, but this is not a requirement.*

large B_{index} this has a negligable impact on the space utilization of index-nodes.

Figure 3.2 shows that Wanna-B-trees use index and leaf nodes as well, and just as for B-trees, Wanna-B-trees hold B_{small} tuples in a leaf node, and B_{index} key-pointer pairs in an index node. Unlike the B-tree, the *Wanna-B-tree* shown in Figure 3.2, panel ②, maintains only the first layer of index nodes above the leaf nodes. We call this single layer of index nodes the *secondary index*. We call the set of leaf nodes pointed at by the secondary index entries in a Wanna-B-tree, Wanna-B-leaves. When a Wanna-B-tree's leaf nodes are written to a storage device in a single contiguous allocation with the secondary index written either separately or after the leaf nodes, this is called an *SSTable* [20]. See Appendix A.

Wanna-B-trees only support insertions in sorted order but allow random lookups. To append we add the elements into the last empty leaf node. If there is no empty leaf node, we create a new one, and append its offset into the last empty index node. If there is no empty index node, then we create one. To perform a lookup on a key K, we perform a binary search on the keys in the index nodes to find the two neighboring key entries K_i and K_{i+1} such that $K_i \leq K < K_{i+1}$. Finally we read in the leaf node pointed at by K_i as this is the leaf-node that must contain the tuple with key K.

It is possible to use B-trees instead of Wanna-B-trees. However, we feel that this adds an unnecessary level of complexity given that the secondary index is always resident in RAM. Still, this may not be true. Systems with constrained RAM configurations (e.g., embedded or older systems) may require a way to avoid thrashing on lookups in the secondary index. Our current implementation of the SAMT does not account for potential thrashing on the secondary index and assumes it is always be resident in RAM. Extending our implementations to use B-trees instead of Wanna-B-trees or applying fractional cascading [11] would avoid this limitation (see Appendix A).

3.2.2 LSM-tree Operation and Minor/Major Compaction

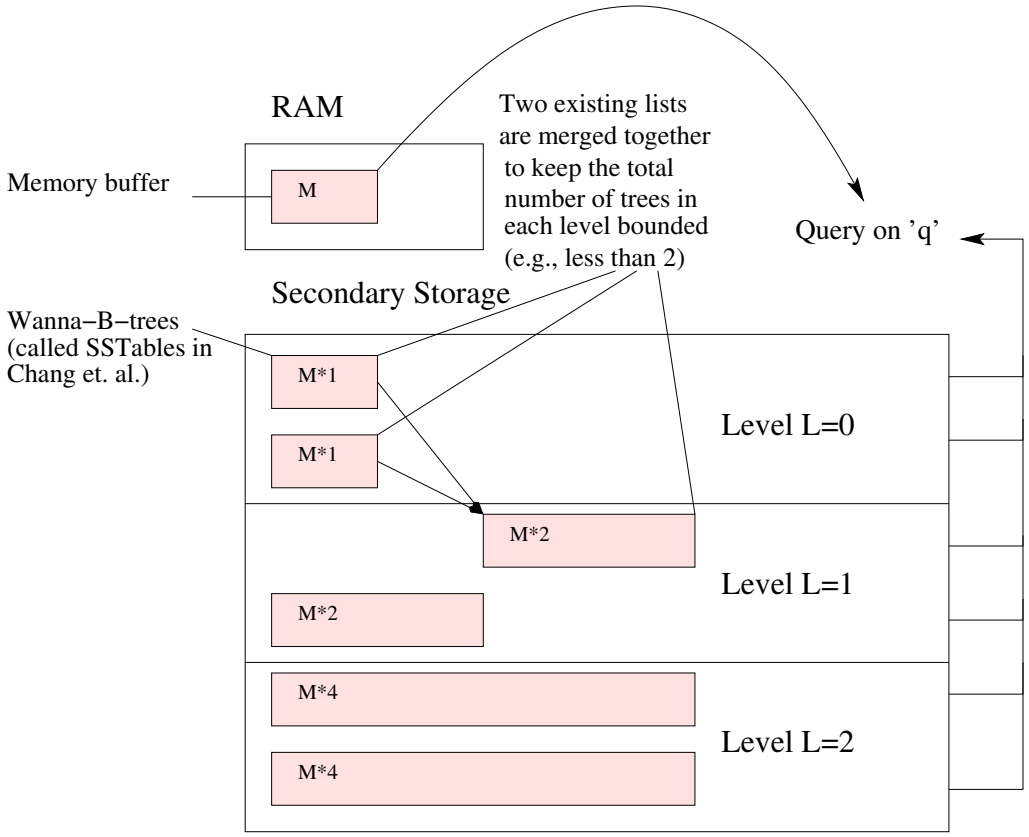

Figure 3.3: Basic LSM-tree Design

The SAMT, like other LSM-trees, is composed of Wanna-B-trees. Figure 3.3 shows a SAMT composed of 6 Wanna-B-trees. These Wanna-B-trees are created when elements are inserted into the SAMT, and are queried when the SAMT is queried. Figure 3.3 shows an example of a query on the element q. First the memory buffer is checked, and if it does not contain the element, each Wanna-B-tree is queried, from most recent to least, until the item is found. If the element is not found in the memory buffer or any Wanna-B-tree, the query returns an error indicating the element does not exist. This kind of query where we search for the most recent tuple belonging to a particular key is called a *point query*.

If we want to read many tuples in series, or find the tuple that comes immediately before or after a queried key value, we perform a query on all Wanna-B-trees and the memory buffer. We remember the location of the tuple after querying it in each Wanna-B-tree or memory buffer, and call this location a *cursor*. We then merge all cursors using the merge algorithm used in merge sort to produce a single stream of all tuples in sorted order that come after the queried key value. This kind of query is called a *scan*.

LSM-trees perform insertions by inserting the new tuple into the memory buffer in RAM. When the memory buffer is full, the LSM-tree serializes the memory buffer to storage by appending each item in sorted order to a new Wanna-B-tree. After serializing the memory buffer, a new Wanna-B-tree with a secondary index has been created and tuples in the memory buffer can be removed

to accept new insertions. The reader may appreciate that point queries and especially scans will take longer to perform when there are a large number of Wanna-B-trees. This is because lookup operations must check an increasing number of Wanna-B-trees as more and more insertions are serialized to storage as Wanna-B-trees.

LSM-trees solve this issue by merging Wanna-B-trees together into larger Wanna-B-trees, and then removing the original smaller trees that were just merged together. This process is called *compaction* in the LSM-tree. Although the compaction algorithms of LSM-tree designs differ, the overall goal is the same: keep the total number of sorted buffers on storage bounded to some function of N, the number of tuples inserted. O'Neil and Sears bound LSM-tree lookups to 2 [95, 121]. Jagadish et al. [57] bound LSM-tree lookups to KJ where J and K are configurable constants. The COLA and HBase bound lookups to $\log_K N$ [11,28]. The SAMT used in Cassandra and GTSSLv1 and GTSSLv2 bound lookups to $K \log_K N$ (for some user-chosen K) [31, 136]. By bounding lookups to a function of N, LSM-trees ensure that regardless of how many tuples are inserted, query times are not indefinitely long.

LSM-trees perform two major types of compaction using the terminology described in Chang et al.'s work [20]. The two types are *minor compaction* and *major compaction*. Minor compaction occurs based on the number of items inserted into the LSM-tree so far, so it is triggered when the LSM-tree has accepted a certain number of items. Major compaction occurs periodically and is intended to occur far less frequently than minor compaction. Major compaction is not required to maintain the asymptotic bounds on the LSM-tree's performance.

Figure 3.3 illustrates what happens when a memory buffer is serialized to storage as a Wanna-B-tree and we trigger minor compaction. The SAMT variant of the LSM-tree ensures bounded queries by making sure the number and size of the Wanna-B-trees lie within certain parameters. The SAMT ensures this by organizing itself into *levels*. Each level holds K Wanna-B-trees: for example in Figure 3.3, $K = 2$. Each level holds Wanna-B-trees that are no larger than $M * K^L$ where M is the size of the memory buffer in tuples and L is the level, where the first level is $L = 0$. The SAMT initiates a minor compaction when the number of Wanna-B-trees at any level is greater than K. If we follow this rule, then on the third serialization of the memory buffer to storage we will perform a minor compaction as there will already be two Wanna-B-trees at level $L = 0$ and we will be making a third. Figure 3.3 illustrates this scenario. Before we can make a third tree at level $L = 0$, we perform a minor compaction by merging the two Wanna-B-trees already at level $L = 0$ into a larger Wanna-B-tree that we place in level $L = 1$ and then remove the two source trees to the merge at level $L = 0$. Now there are no Wanna-B-trees in level $L = 0$ and it is legal to serialize the memory buffer to level $L = 0$ as a new Wanna-B-tree. We discuss in more depth the analysis of several LSM-tree variants in Chapter 5.

The above process works unmodified for insertions of tuples with unique keys. However, if we want to perform updates on a tuple or delete a tuple, then we must specify special actions to be taken during minor and major compactions. The LSM-tree takes care to ensure that there are no more than one tuple with the same key in the memory buffer. When merging multiple or K Wanna-B-trees as part of a minor compaction, the LSM-tree may have to merge multiple tuples with the same key value.

To perform updates, the merge selects the most recent of these tuples with the same key to be included in the new larger output Wanna-B-tree. To perform deletes, we insert a special tuple that has the same key as the tuple we are trying to delete, but with a flag set indicating the tuple is a *tombstone*. When merging K Wanna-B-trees together as part of minor or major compaction, we

treat the tombstone as an update, so if it is the most recent tuple in the merge, it is included in the new larger output Wanna-B-tree and the other tuples are omitted. Different LSM-tree variants treat tombstones differently. The SAMT removes tombstones when it knows that this minor compaction is producing a Wanna-B-tree that will occupy the first slot in the lowest level—so there can be no more tuples potentially with the same key to delete. Other LSM-tree variants only remove tombstones during major compaction, which satisfies the above condition as well.

Periodically, a major compaction is performed. This major compaction merges all Wanna-B-trees belonging to an LSM-tree into one Wanna-B-tree. In addition to some LSM-tree variants relying on major compaction to remove tombstones, there are other reasons to perform periodic compaction of all Wanna-B-trees. One example is performance, another is to ensure that tuples that were deleted are physically removed from the storage device within a set time period (e.g., daily). Major compaction is also an opportune time to perform defragmentation. Although existing LSM-tree variants do not fragment, our generalized LSM-tree that efficiently executes sequential and file system workloads does fragment.

The LSM-tree is a tree data structure, like a B-tree, that can be used on storage devices. The LSM-tree can perform completely random updates, insertions, and deletes asymptotically faster than any data structure that performs one or more random block reads or writes per insertion. However, if it takes more time to perform $logN$ *sequential transfers* of the same tuple than to perform *one* seek and one transfer of the same tuple, then LSM-trees are not as efficient as a more traditional in-place storage data structure like an in-place B-tree. All file systems evaluated in our LSMFS work, as well as in Chapter 4 use on-storage data structures that perform at least one random block I/O when performing random updates for large working sets. The LSM-tree is indeed much more efficient at random updates than these structures as shown in Figure 6.15.

Chapters 5 and 6 introduce extensions and generalizations of the SAMT. We extended the SAMT described in Chapter 5 to support a multi-tier mode, more complex transactions, as well as more efficiently flush to disk when performing large or asynchronous transaction commits. These extensions are not sufficient to make the LSM-tree useful for file system, less uniformly random, or large tuple workloads. To support these workloads, we will need to generalize the LSM-tree to support sequential insertions. Our generalized LSM-tree introduces the idea of *stitching* which avoids copying largely unmodified data during a minor compaction. In order to support efficient support for queries, we must maintain Bloom filters and these Bloom filters cannot be efficiently updated when performing stitching. So we use a new data structure called a *quotient filter* that is like a Bloom filter, but can be efficiently updated while stitching. We discuss these extensions in Chapter 6.

3.2.3 Other Log-structured Database Data Structures

There are other kinds of data structures that are log-structured besides the LSM-tree that are intended for large and small tuples. RethinkDB [148], Write-optimized B-trees [42] and Berkeley DB Java Edition [30] are examples. Each of these log-structured trees uses a very different design, but subscribes to a common theme: making random updates to an on-storage tree more efficient by appending them sequentially in batch and then re-wiring or modifying the tree structure to include the new sequentially appended updates or insertions. We offer a critique of the most common adaptation of this approach in Section 6.4.

Log-structured file systems such as NetApp's write-anywhere file system [51] and ZFS [16]

use a similar approach as log-structured databases, but are designed such that all meta-data for the file system such as page offsets, `inodes`, and other similar information will be resident in RAM when running workloads. Therefore, they can always efficiently re-wire their trees to point to newly appended updates without incurring random I/Os. However, they cannot support updates to datasets much larger than RAM that consist of tuples much smaller than a page without incurring one or more random I/Os for each tuple update. This is because the dataset's keys will not be able to fit within RAM and so re-wiring the tree to point to new insertions will incur evictions and faults of leaf pages at random points in the log. This is explained in more detail in Chapter 6.

The LSM-tree does support high throughput updates, insertions, and deletes regardless of randomness or distribution of writes. Furthermore, its compactions are predictable, de-amortizable, and when taken into account—do not significantly effect update throughput. Finally, scans operate near the disk's sequential read throughput, and point queries operate near the disk's random read throughput.

Unfortunately, if the data inserted is partially sorted, the LSM-tree is not able to spend less write bandwidth by exploiting the fact that the data is already partially sorted. In fact, the LSM-tree does not treat completely sequential or large writes any differently than completely random updates. This is a significant deterrent from using the LSM-tree as the underlying data structure for a file system, because many file system writes are much larger than a page and sequential insertion is important to the performance of many file system workloads.

The LSM-tree is uniquely suited to efficiently processing random updates while maintaining good lookup and scan performance and so it is widely used in NoSQL database implementations. Chang et al. [20] motivate their Big Table architecture with the Web table example. This is a table that receives many random updates, and some significant proportion of their queries are effectively uniformly random. The LSM-tree is uniquely suited to this task.

We limit our comparison of our LSM-tree generalization to the unmodified LSM-tree, or as in Chapter 5, to the LSM-tree implementations utilized by the Cassandra and HBase NoSQL systems. In this way we provide a clear contribution to the LSM-tree structure and transactional file system design.

3.3 Trial Designs

We began Chapter 3 by categorizing related work into *related work* which dealt with past systems that are related to our research, but that we were not directly involved with and *trial designs* that are related systems that we were directly involved in. We now provide a more detailed account of those of our trial designs that shaped our design decisions in a transactional storage system but were not given full chapters in this thesis.

The two systems described here are SchemaFS, a user-level file system based on a popular and efficient database library and LSMFS, a user-level file system that uses LSM-trees to manage meta-data while using an object store to manage data. Experiences from these systems shaped our design decisions that led us to exploring Valor (Chapter 4) and GTSSLv1 (Chapter 5) and ultimately generalizing the LSM-tree as discussed in Chapter 6.

We found that although user-level implementations increase the simplicity of the implementation, they do so at the cost of performance typically. We also found that separating data and meta-data increases the complexity of our transactional implementation in LSMFS and reduces the

flexibility of the file system to handle a variety of workloads.

3.3.1 SchemaFS: Building an FS Using Berkeley DB

KBDBFS is an in-kernel file system we built on a port of the Berkeley Database [130] to the Linux kernel. It was part of a larger project in which we explored uses of a relational database within the kernel. KBDBFS utilized transactions to provide file-system–level consistency, but did not export these same semantics to user-level programs. It became clear to us that unlocking the potential value of a file system built on a database required exporting these transactional semantics to user-level applications.

KBDBFS could not easily export these semantics to user-level applications, because as a standard kernel file system in Linux it used the virtual file system layer (VFS) to re-use common file system code. This reusable component used by many file system implementations maintains caches for `inodes` and directory entry data, or `dentrys`. These caches are not managed by a transaction manager and so would be placed in an inconsistent state if an application aborted a file system operation. To export transactions to user space, KBDBFS would therefore be required to either (1) bypass the VFS layers that require these cached objects and be unable to benefit from re-using common file system code, or (2) alternatively track each transaction's modifications to these objects by integrating the VFS into the KBDFS transaction manager by some means. The first approach requires introducing a large amount of redundant code; the second approach requires major kernel modifications that could significantly reduce performance of non-transactional applications [106], as well as reduce kernel reliabilty.

Based on our experiences with KBDBFS, we chose to prototype a transactional file system, again built on BDB, but in user space. Our prototype, Amino, utilized Linux's process debugging interface, `ptrace` [45]. This allowed us to service file-system–related calls on behalf of other processes, storing all data in an efficient Berkeley DB B-tree schema called *SchemaFS*. Through Amino we demonstrated two main ideas. First, we demonstrated the ability to provide transactional semantics to user-level applications. Second, we showed the benefits that user-level programs gain when they use these transactional semantics: programming model simplification and application-level consistency [151]. Amino and its internal storage interface to BDB—called SchemaFS—is one of our *trial designs*. We discuss it in more detail in Section 3.3.

We extended `ptrace` to reduce context switches and data copies, but Amino's performance was still poor compared to an in-kernel file system for system-call-intensive workloads (such as the configuration phase of a compile). Amino's performance was comparable to Ext3 for meta-data workloads (such as Postmark [60]). For data-intensive workloads, Amino's database layout resulted in significantly lower throughput. Finally, Amino required that a new volume be used to store data, and consequently could not work on top of existing volumes, or on top of existing file systems that may provide a desirable feature or optimization for some workloads.

Amino was a successful project in that it validated the concept of a transactional file system with a user-visible transactional API, but the performance we achieved could not displace traditional file systems. Moreover, one of our primary goals was for transactional and non-transactional programs to have access to the same data through the file system interface. Although Amino provided binary compatibility with existing applications, running programs through a `ptrace` monitor was not as seamless as we had hoped. The `ptrace` monitor had to run in privileged mode to service all processes. It serviced system calls inefficiently due to additional memory copies and context

switches, and it imposed additional overheads from using signal passing to simulate a kernel system call interface for applications [151].

Based on these limitations, we developed *Valor*, which is described in Chapter 4. Valor grants transactional POSIX storage semantics to applications running on any file system. In experimentation, Valor is shown to be more efficient than approaches that rely on a user-level page caching approach such as Stasis [120]. Valor incurs the typical WAL overheads for asynchronous sequential file system workloads, but only for applications that perform these workloads within a transactional context. Valor provides a valuable outline for designing a transaction-capable VFS for file systems. In Chapters 5–6 we develop a data structure that allows for a transactional implementation that can compete more favorably with typical file systems while still efficiently supporting traditional and more contemporary database workloads.

3.3.2 User-Space LSMFS: Building an FS on LSMs in User-Space with Minimal Overhead

Much of the inefficiency of a user-level storage system is due primarily to a special feature reserved to file system caches: caches can be mapped into the address space of every process, and that security can still be maintained with the system-call interface. We explored the performance impact of in-kernel and out-of-kernel file system caches, and compared them to a user-level shared-memory cache that guards against unrestricted access using a user definable trapping mechanism [133] (trampoline). This work illustrates a partial, or hybrid, approach that moves a monolithic kernel in the direction of a more fluid and extensible micro-kernel architecture. File systems that cannot wait for expensive memory copies or context switches [141] can still be developed as user-level daemons by using our approach.

We built a file system based on the LSM-tree data structure in user-level using our external page caching framework. This file system consisted of two components: (1) a cache-oblivious transactional meta-data database based on the cache-oblivious look-ahead array (COLA, see Appendix A) [11], a variant of the LSM-tree, and (2) an object store for data pages. By using an object store to hold data pages we avoided generalizing the LSM-tree for sequential data or large tuples. However, this prevented data from naturally spanning multiple tiers in the manner described in Chapter 5, and also created a dependence on two underlying data structures that increased the complexity of our design and implementation. By using an object store for data pages, we ensured that LSMFS would not behave differently from other file systems except when updating and scanning meta-data. Furthermore, we would have to categorize which tuples are "data" and go in the object store, and which are "meta-data" and go in the LSM-tree. We address these questions by generalizing the LSM-tree in Chapter 6.

Although LSMFS showed us that the LSM-tree was a good candidate for storing file system meta-data, we were interested in further exploring the potential benefit of multi-tiering within the LSM-tree. The transactional implementation in LSMFS was also not capable of handling high numbers of concurrent durable transactions, and performed additional unnecessary copies on commit. We also wanted to find a way to efficiently store file data within the LSM-tree instead of within a separate object store, but this would require extensions to the LSM-tree that we explore further in Chapter 6.

3.4 Putting it Together

To arrive at the generalized LSM-tree data structure used in Chapter 6, we combined the lessons learned from Section 3.3 and Chapters 4–6. In Table 3.1 we lay out these lessons and show how we concluded that we must explore using a new data structure for transactional file systems. The data structure we chose to study and generalize is the log-structured merge-tree (LSM-tree), which is introduced and briefly explained in Section 3.2.

	Type	**Num Writes**	**Log-Struct.**	**Trans-actions**	**Conc-urrent**	**Async**	**Write Order**	**Random**	**Stitch-ing**	**Sequen-tial**
Ext3	FS	1	¬	MD-only	¬	✓	Kernel	R	¬	R,W
SchemaFS	FS	3	¬	Logical	✓	✓	User	R	¬	R
LSMFS	FS	1	✓	MD-only	¬	✓	`mmap`	S,W	¬	R,W

Table 3.1: ***A qualitative comparison of Transactional Storage Designs:*** *We can conclude that we must try new data structures for transactional file systems.*

We can show how we ultimately decided to use LSM-trees, and how we arrived at GTSSLv1 and v2's transactional architecture by using some of the observations listed in Table 3.1. Table 3.1 summarizes the features and properties of the major transactional storage system designs researched within this thesis. This table extracts the technical details from our lessons learned in Section 3.3 for easier comparison across transactional and file storage systems.

Items already described in detail, including Ext3, SchemaFS, and LSMFS are highlighted in **bold** in Table 3.1. Other items not yet comprehensively discussed will be described more thoroughly at the end of that item's respective chapter. For example, Valor will be highlighted in bold and its impact on our ultimate design choices will be discussed at the end of Chapter 4 where Valor is discussed in detail along with a new copy of Table 3.1 with a new entry for Valor. GTSSLv1, and GTSSLv2 will be similarly detailed at the end of Chapters 5 and 6, respectively.

- **Type** denotes whether the system exports a file system (FS) API or a key-value storage (KVS) API.
- **Num Writes** is 1 if the system performs the minimal number of writes to carry out a larger write operation for the majority of its write operations. If it is more than 1, then that is the total number of writes that may be needed to carry out a write operation (e.g., to a log). GTSSLv1 and v2 require only two writes for concurrent, small, durable transactions, and only one write for larger or asynchronous transactions.
- **Log Struct.** denotes whether the system's design is log-structured, thus not having to write data first to the log as it does not overwrite existing data; non-log-structured systems which may overwrite existing data must more tightly control write ordering.
- **Transactions** has the following values:

  MD-only if the storage system uses transactions internally for meta-data logging only.
  Logical if applications can extend the transactional system's functionality. Only systems that we list as Logical support this manner of extensibility.
  POSIX if the transactional model supports only POSIX file operations.
  Single if it can insert, modify, or delete only a single tuple atomically at a time.

Vals if it supports arbitrary transactions comprising one or more tuples, as long the entire transaction fits in RAM. (In future work we are extending this for larger-than-RAM transactions.) All other values (except "Logical") must also fit in RAM, and are much more restrictive.

- **Concurrent** is checked if the storage system can efficiently commit many concurrent durable transactions (e.g., by grouping them into a single write request to a log record, or using NVRAM, etc.)
- **Async** is checked if the system can commit non-durable transactions asynchronously, *without* violating any transaction's isolation or atomicity.
- **Write Order** has the following values:

  Kernel if write ordering in the cache is controlled directly by kernel code.
  User if write ordering is controlled by having user-level code maintain a separate cache and decide when and how to flush it. This obviously carries significant space and time overheads.
  `mmap` if write ordering for the cache is not necessary and consequently a simple and efficient cache implementation based on `mmap` can be derived.

- **Random** refers to the structured tables of a database or key-value (KV) store, or file-system (FS) meta-data. File systems access their meta-data randomly, and so we rate the effectiveness of their data structures for those operations. For key-value stores, we rate their efficiency with random tuple accesses and updates. This column has one or more of the following values:

  R if all read operations are efficient for structured data workloads (e.g., database queries).
  W if all write operations are efficient for structured data workloads.
  S if scans are efficient.
  P if point queries are efficient.

- **Stitching** is checked if the system is able to avoid redundant compactions of a sequence of tuples that is already contiguous enough to facilitate an efficient scan. Stitching systems perform fewer copies for insertions of larger, more sequentially accessed tuples than other LSM-tree–based systems.
- **Sequential** uses the same tokens as *Random* except that they denote the performance of the storage system when processing larger tuples or highly sequential workloads. For file systems this rates their efficiency at performing large sequential file reads and writes. For key-value stores, this rates the store's efficiency at performing insertions of mostly sequentially inserted tuples or very large tuples.

The storage systems listed above are evaluated in Table 3.1 for their suitability as the underlying storage system on a single-node system that supports system transactions and key-value storage. Some systems have additional features, some utilize mechanisms not exposed to the application layer, and some are designed to be portable. For example, Ext3's internal transaction logging mechanism, or directory entry hash tables are not available for applications to use directly. Berkeley DB (upon which SchemaFS is based) uses a user cache to be portable. Regardless, we have found

many of these systems have surprisingly efficient (even if limited) transactional implementations (e.g., Ext3), or provide fairly featureful transactional capabilities and we want to compare and contrast all these systems' features to understand how our own research learns from the lessons listed in Section 3.3.

3.5 Conclusion

In Chapter 2 we explained the three criteria we used to evaluate transactional and key-value storage file system designs: abstractions, performance, and implementation simplicity. We explained that specifically the abstractions we were interested in were system transactions, and key-value storage operations. In this chapter we explained how our search for a transactional key-value storage file system related to other research in storage systems and data structures.

In addition to discussing related work we also highlighted important lessons learned from our own trial designs in Section 3.3. We summarized the effects of these lessons and other related systems' transactional design decisions in Table 3.1. We saw which transactional design decisions were made by Ext3 discussed in related work in Section 3.1.2, and SchemaFS and LSMFS discussed in Sections 3.3.1 and 3.3.2.

Now that we have framed our research from the perspective of related work and our past experience with other systems, we describe in detail our Valor design in Chapter 4, and subsequent designs thereafter in Chapters 5 and 6.

Chapter 4

Valor: Enabling System Transactions with Lightweight Kernel Extensions

In the past, application developers seeking to utilize a transactional interface for files typically had to choose from two undesirable options: (1) modify complex file system code in the kernel or (2) use a user-level solution, which incurs unnecessary overheads. Previous in-kernel designs either had the luxury of designing around transactions from the beginning [119] or limited themselves to supporting only one primary file system [147]. Previous user-level approaches were implemented as libraries (e.g., Berkeley DB [130] and Stasis [120]) and did not support interaction through the VFS [63] with other non-transactional processes. These libraries also introduce a redundant page cache and provide no support to non-transactional processes.

This chapter presents Valor, the design of a transactional file interface that requires modifications to neither existing file systems nor applications, yet guarantees atomicity and isolation for standard file accesses using the kernel's own page cache. We have fully implemented the locking and data logging portions of the Valor system. Directory logging and recovery are partially implemented and are not evaluated in this thesis. Our experiments are limited to benchmarks which have negligible meta-data and directory overhead; our testing shows that for these benchmarks, directory logging overhead is negligible. We have come back to the Valor design repeatedly when considering how to support transactional file access in a VFS-based operating system like Unix. Valor's design has been useful and is presented here in full. When discussing Valor's directory logging design, we remind the reader that this component is only partially implemented. When discussing Valor's experimental results, we remind the reader that we have implemented recovery for data, but not for meta-data. The experiments in this chapter are for evaluations of data logging and data recovery only. We describe techniques and methods for more generally supporting efficient transactional operations on small complex data including file system meta-data in subsequent Chapters 5 and 6.

Enforcing the ACID properties often requires many OS changes, including a unified cache manager [50] and support for logging and recovery. Despite the complexity of supporting ACID semantics on file operations [111], Microsoft [147] and others [35, 151] have shown significant interest in transactional file systems. Their interest is not surprising: developers are constantly

reimplementing file cleanup and ad hoc locking mechanisms that are unnecessary in a transactional file system. A transactional file system does not eliminate the need for locking and recovery, but hides it inside the transactional implementation. Defending against TOCTTOU (time of check till time of use) security attacks also becomes easier [108, 109], because sensitive operations are easily isolated from an intruder's operations. The number of programs running on a standard system continues to grow along with the cost of administration. In Linux, the CUPS printing service, the Gnome desktop environment, and other services all store configuration information in files that can become corrupted when multiple writers access them or if the system crashes unexpectedly. Despite the existence of database interfaces, many programs still use flat text configuration files for their simplicity, generality, and because a large collection of existing tools can access these simple configuration files. For example, Gnome stores over 400 configuration files in a user's home directory. A transactional file interface is useful to all such applications.

To provide ACID guarantees, a file interface must be able to mediate all access to the transactional file system. This forces the designer of a transactional file system to put a large database-like runtime environment either in the kernel or in a kernel-like interceptor, since the kernel typically services file-system system calls. This environment must employ abortable logging and recovery mechanisms that are linked into the kernel code. The caches in the VFS layer effected by the transaction must also be rolled back including [151] its stale inodes, dentries, and other in-kernel data structures. The situation can be simplified drastically if one abandons the requirement that the backing store for file operations must be able to interact with other transaction-oblivious processes (e.g., `grep`), and by duplicating the functionality of the page cache in user space. This concession is often made by transactional libraries such as Berkeley DB [130] and Stasis [120]: they provide a transactional interface to a separate set of store files and they do not solve the complex problems of rewinding the operating system's page cache and stale in-memory structures after a process aborts. Systems such as QuickSilver [119] and TxF [147] address this trade-off between the completeness and implementation size by redesigning a specific file system around proper support for transactional file operations. In this chapter we show that such a redesign is unnecessary, and that every file system can provide a transactional interface without requiring specialized modifications. We describe our system, which uses a seamless approach to provide transactional semantics using a dynamically loaded kernel module and only minor modifications to existing kernel code. Our technique keeps kernel complexity low yet offers a full-fledged transactional file interface without introducing unnecessary overheads for non-transactional processes.

We call our file interface *Valor*. Valor relies on improved locking and write ordering semantics that we added to the kernel. Through a kernel module, it also provides a simple in-kernel logging subsystem optimized for writing data. Valor's kernel modifications are small and easily separable from other kernel components; thus introducing negligible kernel complexity. Processes can use Valor's logging and locking interfaces to provide ACID transactions using seven new system calls. Because Valor enforces locking in the kernel, it can protect operations that a transactional process performs from any other process in the system. Valor aborts a process's transaction if the process crashes. Valor supports large and long-lived transactions. This is not possible for `ext3`, `XFS`, or any other journaling file system: these systems can only abort the entire file system journal and only if there is a hardware I/O error or the entire system crashes. These systems' transactions must always remain in RAM until they commit (see Section 4.1).

Another advantage of Valor is that it is implemented on top of an unmodified file system. This results in negligible overheads for processes not using transactions: they simply access the under-

lying file system, using the Valor kernel modifications only to acquire locks. Using tried-and-true file systems also provides good performance compared to systems that completely replace the file system with a database. Valor runs with a statistically indistinguishable overhead on top of `ext3` under typical loads when providing a transactional interface to a number of sensitive configuration files. Valor is designed from the beginning to run well without durability. File system semantics accept this as the default, offering `fsync(2)` [46] as the accepted means to block until data is safely written to disk. Valor has an analogous function to provide durable commits. This makes sense in a file-system setting where the default policy is for writes to be serviced asynchronously by the file system. For non-durable data-only transactions, Valor's overhead on top of an idealized ARIES implementation is only 35% (see Section 4.3).

The rest of this chapter is organized as follows. We detail Valor's design in Section 4.2 and evaluate its performance in Section 4.3. We conclude and propose future work in Section 4.4.

4.1 Background

The most common approach for transactions on stable storage is using a relational database, such as an SQL server (e.g., MySQL [90]) or an embedded database library (e.g., Berkeley DB [130]); but they have also long been a desired programming paradigm for file systems. By providing a layer of abstraction for concurrency, error handling, and recovery, transactions enable simpler, more robust programs. Valor's design was informed by two previous file systems we developed using Berkeley DB: KBDBFS and Amino [151]. Next we discuss journaling file systems' relationship to our work, and we follow with discussions on database file systems and APIs.

4.1.1 Database on an LFS Journal and Database FSes

Another transaction system which modified an existing OS was Seltzer's log-structured file system, modified to support transaction processing [126]. Seltzer et al.'s simulations of transactions embedded in the file system showed that file system transactions can perform as well as a DBMS in disk-bound configurations [124]. They later implemented a transaction processing (TP) system in a log-structured file system (LFS), and compared it to a user-space TP system running over LFS and a read-optimized file system [126]. Although Seltzer's work discusses the performance of a user-level database relying on an in-kernel transaction manager, it does not discuss at all how to achieve POSIX system transactions. In Section 4.2 we discuss interception mechanisms, directory locking, deadlock detection, locking permissions and policy to prevent denial of service attacks, and multi-process transactions (e.g., bash). To support this level of locking, as well as process inheritance of locks, transaction information must be maintained with the process block such as transaction ID, and a list of held locks. Valor performs transactional operations concurrently with other updates to the operating system's and file system's caches. Valor provides data recovery (although directory recovery is not yet implemented). We detail Valor's locking design within the operating system and VFS layer as well as its recovery approach in Section 4.2.

Microsoft's TxF [83, 147] and QuickSilver's [119] database file systems leverage the early incorporation of transactions support into the OS. TxF exploits the transaction manager which was already present in Windows. TxF uses multiple file versions to isolate transactional readers from transactional writers. TxF works only with NTFS and relies on specific NTFS modifications and

how NTFS interacts with the Windows kernel. QuickSilver is a distributed OS developed by IBM Research that makes use of transactional IPC [119]. QuickSilver was designed from the ground up using a microkernel architecture and IPC. To fully integrate transactions into the OS, QuickSilver requires a departure from traditional APIs and requires each OS component to provide specific roll-back and commit support. We wanted to allow existing applications and OS components to remain largely unmodified, and yet allow them to be augmented with simple begin, commit, and abort calls for file system operations. We wanted to provide transactions without requiring fundamental changes to the OS, and without restricting support to a particular file system, so that applications can use the file system most suited to their work load on any standard OS. Lastly, we did not want to incur any overheads on non-transactional processes.

LOCUS [87] details locking protocols for a distributed storage system that replicates files across nodes on the same network. TABS [132] details a logging protocol for a distributed storage system that allows processes to coordinate distributed two-phase transactions that manipulate objects across nodes. LOCUS and TABS focus on sketching out workable protocols that can be used in future implementations. LOCUS describes a protocol for performing locking in a distributed storage system. TABS describes a protocol for transaction logging in a distributed storage system. Neither system details how to add transactional file system support to a VFS-based operating system. For example, LOCUS's locking protocols do not deal with directories and provide no details on how transactions interact with non-volatile storage. TABS describes the design of a two-phase WAL-based transactional page store, but provides no experimental performance. TABS offers new discussion on the interaction between transactional and non-transactional processes or file system objects. TABS does not address write-ordering concerns within a complex operating system used by legacy applications, non-transactional applications, or transactional applications. Neither LOCUS nor TABS provide robust deadlock detection: they rely on time-outs only. Our work in Valor shows that achieving good performance for data recovery and meta-data isolation is non-trivial. Valor shows how to integrate these features into a complex modern operating system.

Inversion File System [94], OdeFS [37], iFS [100], and DBFS [88] are database file systems implemented as user-level NFS servers [77]. As they are NFS servers (which predate NFSv4's locking and callback capabilities [128]), the NFS client's cache can serve requests without consulting the NFS server's database; this could allow a client application to write to a portion of the file system that has since been locked by another application, violating the client application's isolation. They do not address the problem of supporting efficient transactions on the local disk.

4.1.2 Database Access APIs

We discussed in Section 3.1.3 how one common approach to providing a transaction interface to applications is to provide a user-level library. This library stores data in a special page file or set of files maintained by the library. Berkeley DB offers a B-Tree, a hash table, and other structures [130]. Stasis offers a page file [120]. These systems require applications to use database-specific APIs to access or store data in these library-controlled page files.

Berkeley DB Berkeley DB is a user library that provides applications with an API to transactionally update key-value pairs in an on-disk B-Tree. We discuss Berkeley DB's relative performance in depth in Section 4.3. We benchmark BDB through Valor's file system extensions. Relying on BDB to perform file system operations, for which it was not originally designed or optimized, can result

in large overheads for large serial writes or large transactions (256MB or more). This is because BDB is being used to provide a file interface, which is used by applications with different workloads than applications that typically use a database. If the regular BDB interface is used, though, transaction-oblivious processes cannot interact with transactional applications, as the former use the file system interface directly.

Stasis Stasis provides applications a transactional interface to a page file. Stasis requires that applications specify their own hooks to be used by the database to determine efficient undo and redo operations. Stasis supports nested transactions [43] alongside write-ahead logging and LSN-Free pages [120] to improve performance. Stasis does not require applications to use a B-Tree on disk and exposes the page file directly. Like BDB, Stasis requires applications to be coded against its API to read and write transactionally. Like BDB, Stasis does not provide a transactional interface on top of an existing file system which already contains data. Also like BDB, Stasis implements its own private, yet redundant page cache which is less efficient than cooperating with the kernel's page cache (see Section 4.3).

Reflecting on our experience with KBDBFS and Amino, we have come to the conclusion that adapting the file system interface to support ACID transactions does indeed have value and that the two most valuable properties that the database provided to us were the logging and the locking infrastructure. Therefore, in Valor we provide two key kernel facilities: (1) extended mandatory locking and (2) simple write ordering. Extended mandatory locking lets Valor provide the isolation that in our previous prototypes was provided by the database's locking facility. Simple write ordering lets Valor's logging facility use the kernel's page cache to buffer dirty pages and log pages which reduces redundancy, improves performance, and makes it easier to support transactions on top of existing file systems.

4.2 Design and Implementation

The design of Valor prioritizes (1) a low complexity kernel design, (2) a versatile interface that makes use of transactions optional, and (3) performance. Our approach achieves low complexity by exporting a minimal set of system calls. Functionality exposed by these system calls would be difficult to implement efficiently in user-space.

Valor allows applications to perform file-system operations within isolated and atomic transactions. If desired, Valor can ensure a transaction is durable: if the transaction completes, the results are guaranteed to be safe on disk if the underlying file system uses journaling or some other mechanism (e.g., soft-updates) to ensure safe meta-data writes. We now turn to Valor's *transactional model*, which specifies the scope of these guarantees and what processes must do to ensure these guarantees.

Transactional Model Valor's transactional guarantees extend to the individual inodes and pages of directories and regular files for reads and writes. Valor enforces a bare minimum of kernel-side locking for all processes and relies on transactional applications to perform their own additional locking as necessary. Transactional applications are expected to use a user-level library to perform any additional locking not already performed by Valor's kernel-side components; this user-level library will correctly acquire and release locks on behalf of transactional applications. Locking is

mandatory (i.e., not advisory), so even though transactional applications must use the Valor library to acquire and release locks correctly, these locks are fully enforced and respected by all processes.

Valor's transactional model is a function of what kinds of locks it acquires for what operations. These policies can be changed by an implementer but we specify here a default transactional model. In Section 4.2.2 we describe how Valor can be configured to prevent misuse of locks. Valor supports transactional isolation of all of the `inode`'s fields except those that are updated by a random write (e.g., `mtime`) or a random read (e.g., `atime`). Appending to a file and increasing its length is protected by transactional isolation. Valor supports transactional isolation of testing for the existence of, removing, and listing directory entries. Page overwrites to a file are protected as well. Processes can spawn children that can partake in the parent's transaction according to policies that can be controlled by the application developer or system administrator as we discuss in further detail in Section 4.2.2. Increasing the configurability of Valor's transactional model is a subject of future work. Maintaining the concurrency of random file accesses without sacrificing full isolation of all `inode` fields, such as `mtime` and `atime`, is possible with lock types [43, 132]; extending Valor to support additional types of locking is also subject of future work.

The bare minimum of kernel locking as performed by Valor is locking an `inode` when files are accessed, and locking a path of directory `inode`s to a directory's `inode` when a directory is accessed. Valor read-locks a regular file's `inode` when the file is written to, and write-locks the `inode` when the file is appended to. Meta-data which is updated by a non-appending write is not protected by isolation (e.g., `mtime`). Valor read-locks the `inode`s of directories leading to a directory, and then either read or write-locks the directory being operated upon depending on whether the application is modifying the directory.

Valor's implementation attempts to avoid deadlocking when performing broad modifications to a sub-tree within the file system, by allowing processes to recursively lock all of a directory's descendants during a rename. In this way, a process can reserve a sub-tree within a file system before making extensive changes to it. Subsequent designs such as LSMFS and GTSSLv2 do not follow this practice as it increases the cost of performing a rename. Futhermore, GTSSLv2 has better support for snapshots. Snapshots can be used to provide private copies of a sub-tree which a process can modify extensively; the process can then merge the changes back into the most recent version of the file system. In such a scenario, handling conflicts during a merge would require application-specific conflict handlers [61] and is currently a subject of future work.

To illustrate Valor's locking policies, both inside and outside the kernel, we describe a brief example. Consider two processes A and B. Process B is transactional but A is not. Process B begins transaction T_B which creates a directory /USR/SHARE/DOCS/. Then process A tries to create /USR/SHARE/DOCS/FILE.TXT. Then process B performs some other work, removes /USR/SHARE/DOCS/, and finally commits. Process A is not transactional and releases its locks before each of its system calls end. However, process A still acquires locks on resources before utilizing them and will attempt to acquire a read-lock on /, USR, and SHARE even though it is not transactional: A has to wait on B. In this way, B's isolation is enforced in that process A is not able to observe B's intermediate state by successfully creating FILE.TXT and therefore determining that A created SHARE within its transaction. After B's transaction commits, A will fail to create FILE.TXT as SHARE will not exist.

We now turn to the concepts underlying Valor's architecture. These concepts are implemented as components of Valor's system and illustrated in Figure 4.1.

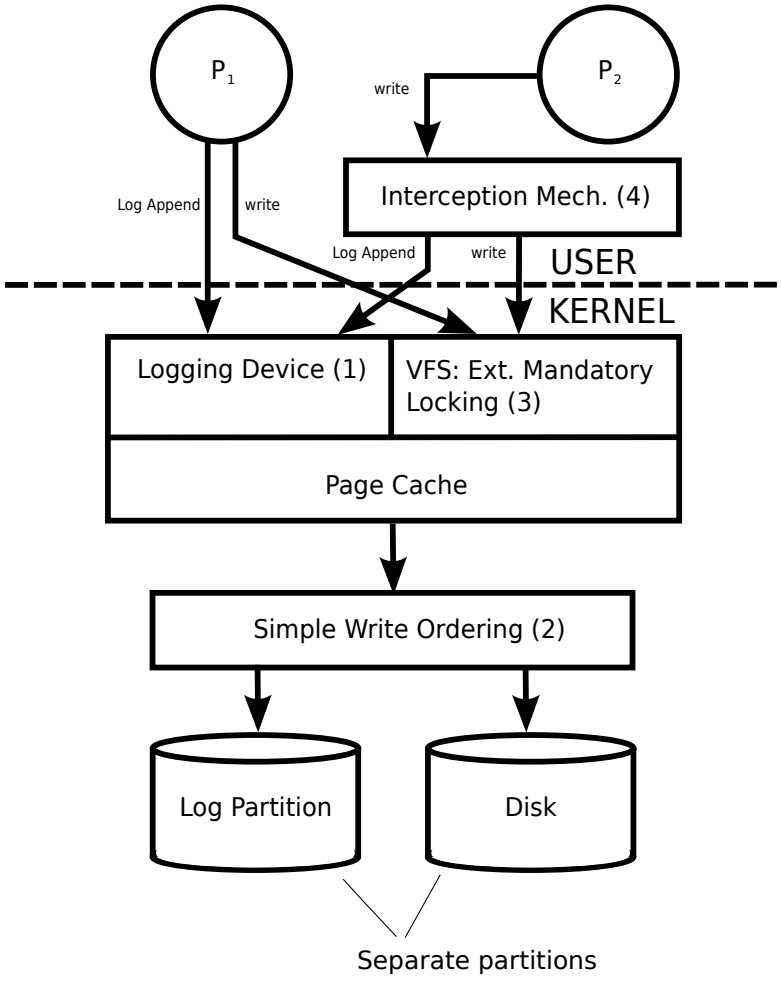

Figure 4.1: Valor Architecture

1. Logging Device In order to guarantee that a sequence of modifications to the file system completes as a unit, Valor must be able to undo partial changes left behind by a transaction that was interrupted by either a system crash or a process crash. This means that Valor must store some amount of auxiliary data, because an unmodified file system can only be relied upon to atomically update a single sector and does not provide a mechanism for determining the state before an incomplete write. Common mechanisms for storing this auxiliary data include either a *log* [43] or a copy-on-write approach [51]. Valor does not modify the existing file system, so it uses an undo-redo log stored on a separate partition called the *log partition*.

2. Simple Write Ordering Valor relies on the fact that even if a write to the file system fails to complete, the auxiliary information has already been written to the log. Valor can use that information to undo the partial write. In short, Valor needs to ensure that writes to the log partition occur before writes to other file systems. This requirement is a special case of *write ordering*, in which the page cache can control the order in which its writes reach the disk. We discuss our implementation in Section 4.2.1, which we call *simple write ordering* both because it is a special case and because it operates specifically at page granularity. Simple write ordering is not a general-purpose write-ordering system [33] for applications and was designed to be simple and direct. It is a component of Valor which is intended to help provide ACID transactions to applications. The log partition and disk components in Figure 4.1 must be on separate partitions, although those partitions may reside on the same physical device.

3. Extended Mandatory Locking Isolation gives a process the illusion that there are no other concurrently executing processes accessing the same files, directories, or inodes. Transactional processes implement this by first acquiring a lock before reading or writing to a page in a file, a file's inode, or a directory. However, an OS with a POSIX interface and pre-existing applications must support processes that do not use transactions. These *transaction-oblivious* processes do not normally acquire locks before reading from or writing to files or directories. *Extended mandatory locking* ensures that all processes, even transaction-oblivious processes, acquire locks before accessing these resources. See Section 4.2.2. Existing mandatory locking in the Linux kernel includes a waits-for-graph and cycle detection which is performed each time a lock is acquired. These locks are blocking (unless they deadlock). Extended mandatory locking inherits these features for Valor's own deadlock detection.

4. Interception Mechanism New applications can use special APIs to access the transaction functionality that Valor provides; however, pre-existing applications must be made to run correctly if they are executed inside a transaction. This could occur if, for example, a Valor-aware application starts a transaction and launches a standard shell utility. Valor modifies the standard POSIX system calls used by unmodified applications to perform the locking necessary for proper isolation. Section 4.2.3 describes our modifications.

The above four Valor components provide the necessary infrastructure for the seven Valor system calls. Processes that desire transactional semantics must use the Valor system calls to log their writes and acquire locks on files. We now discuss the Valor system calls and then provide a short example to illustrate Valor's basic operation.

`Log Begin` begins a transaction. This must be called before all other operations within the transaction.

`Log Append` logs an *undo-redo record*, which stores the information allowing a subsequent operation to be undone or redone. This must be called before every operation within the transaction. See Section 4.2.1.

`Log Resolve` ends a transaction. In case of an error, a process may voluntarily *abort* a transaction. The *abort* operation undoes partial changes made during that transaction. Conversely, if a process wants to end the transaction and ensure that changes made during a transaction are all done as an atomic unit, it can *commit* the transaction. Whether a `log resolve` is a commit or an abort depends on a flag that is passed to the call.

`Transaction Sync` flushes a transaction to disk. A process may call `Transaction Sync` to ensure that changes made in its committed transactions are on disk and can never be undone. This is the only sanctioned way to achieve durability in Valor. `O_DIRECT`, `O_SYNC`, and `fsync` [46] have no useful effect within a transaction for the same reason that nested transactions cannot be durable: the parent transaction has yet to commit [43].

`Lock`, `Lock Permit`, `Lock Policy` Our `Lock` system call locks a page range in a file, an entire directory (and the path to that directory), or an entire file with a shared or exclusive lock. This is implemented as a modified `fcntl`. Since Valor operates with extended mandatory locking, the `fcntl` is no longer advisory. These routines provide Valor's support for transactional isolation. `Lock Permit` and `Lock Policy` are required for security and interprocess transactions, respectively. See Section 4.2.2.

Cooperating with the Kernel Page Cache As illustrated in Figure 4.1, the kernel's page cache is central to Valor, and one of Valor's key contributions is its close cooperation with the page cache. In systems that do not support transactions, the `write(2)` system call initiates an asynchronous write which is later flushed to disk by the kernel page cache's dirty-page write-back thread. In Linux, this thread is called `pdflush` [17]. If an application requires durability in this scenario, it must explicitly call `fsync(2)`. Not mandating durability by default is an important optimization which allows `pdflush` to economize on disk seeks by grouping writes together.

Therefore most system tools that modify the file system are optimized for a buffered `write(2)` call: shell scripts, build scripts, make processes, and more slow down if they block on durable commits. Further most file system operations can be easily retried, as they are generated by a tool (e.g., system update, program installation) that can easily be re-run, but which wreak havoc if only partially performed. Existing installation and update tools use ad-hoc techniques to avoid this calamity, but providing a reliable single system that all such tools can use is one of Valor's goals as a transactional file system.

Databases, despite introducing transaction semantics, achieve similar economies through *No-Force* page caches. These caches write auxiliary log records only when a transaction commits, and then only as one large serial write, and use threads similar to `pdflush` to flush data pages asynchronously [43]. Valor is also No-Force, but can further reduce the cost of committing a transaction by writing nothing—neither log pages nor data pages—until `pdflush` activates. Valor's default transactions are analogous to a No-Force, non-durable transaction.

Valor's simple write ordering scheme facilitates this optimization by guaranteeing that writes to the log partition always occur before the corresponding data writes. In the absence of simple write

ordering, Valor would be forced to implement a redundant page cache, as many other systems do.

This is because in the absence of a system-supported write-ordering mechanism, modifications to the file system page cache must be intercepted by a secondary cache, one which has been programmed to not write back until given the signal that the corresponding log writes have hit the disk. Without introducing a secondary cache that would intercept every file system write, the log would have to be synced before every single write to the cache to ensure no write in the cache is ever flushed without its corresponding log record having been flushed first. This would ruin the file system's ability to perform efficient asynchronous sequential writes.

Valor implements simple write ordering in terms of existing Linux `fsync` semantics. Until recently the `fsync` command on Ext3 and other popular file systems (e.g., XFS, and ReiserFS) would return when the writes were scheduled, but before they had hit the disk platter. This introduced a short race where applications running on top of Valor and the other systems we evaluated (MySQL, Berkeley DB, Stasis, and ext3) could crash irrecoverably. All Linux kernel versions prior to 2.6.30 suffered from this problem. Linux 2.6.30 forced `fsync` to perform a journal flush in Ext3 which ensures a disk-platter flush but only after improving `fsync` latency in Linux. Linux 2.6.30 included fixes for a host of I/O performance bugs for Ext3 and the block layer, including proper prioritization of meta-data I/O in the block scheduler [25]. On systems that do not support proper sync within the operating system we know of three workarounds: disabling write cache, switching to an operating system that provides correct synchronous writes, such as MacOS X, Linux, or Windows, or powering the hard disk with an uninterruptible power supply.

Simple write ordering is also a general purpose primitive; user-level transactional storage systems often support non-durable atomic transactions. Although such systems avoid synchronously writing the log to disk at commit, they must still ensure that when pages are written back, that log writes precede all else. Without any sort of write-ordering mechanism, they are forced to `fsync(2)` the log, to ensure that all page writes are preceded by their respective log writes. The intent of calling `fsync(2)` by a user-level database performing a commit of an asynchronous transaction is to order the log write before the page write. However, the file system will *also* flush the write (by the definition of `fsync(2)`). Instead, simple write ordering can coordinate with the page cache, ensuring that log writes happen before page writes but that writes occur only when there is pressure on the kernel's page cache. This avoids redundant disk writes and cache flushes while maintaining write ordering for database applications.

Alternative approaches to "soften" `fsync(2)` by not requiring an immediate write to the disk platter include speculating [93] or a comprehensive write-ordering system [33]. Nightingale et al. describe a system that makes `fsync(2)` calls asynchronous and forces a flush only when a user observes the result of an operation through an I/O operation such as using the `tty` device. Such a system would have to comprehensively detect any communication between processes and the user: a notoriously challenging open problem [68]. Nightingale et al.'s system is not effective for removing the overhead incurred by `fsync(2)` calls that always occur immediately before a user is directly or indirectly notified of the sync. A comprehensive write ordering system with an application interface must pervade the entire storage stack down to the block layer. Frost et al.'s write ordering system maintains an additional logging and graph-management system [33].

Like these more complex systems, simple write ordering is a suitable solution for database, source-control, logging, and transactional applications and requires only a small amount of code to change in the kernel's page write-back code. In other work [134], we configured a transactional page-cache library called Stasis [120] to use Valor to schedule log write-back by modifying a few

lines of code, significantly reducing the cost of short asynchronous transactions without sacrificing recoverability and atomicity. Simple write ordering is not as comprehensive as Frost et al.'s work in terms of strictly providing write-ordering primitives. However, in concert with a database implementation, or a transactional interface such as Valor, simple write ordering provides a sufficient amount of cooperation with the kernel's page cache to facilitate efficient asynchronous and atomic storage operations.

Since Valor's simple write ordering mechanism always guarantees that log writes hit the disk before other pages, it is possible that an entire transaction can commit to the log, while pending writes to the file system for that transaction are not yet written back. Valor reports to a process blocking on this event (for durability) that the transaction committed to disk. If the system crashes before this transaction's pending writes hit the disk, Valor must be able to complete the disk writes during recovery to fulfill its durability guarantee. Similar to database systems that also perform this optimization, Valor uses redo-undo logging and includes sufficient information in the log entries to *redo* the writes, allowing the transaction to be completed during recovery.

Valor supports large transactions that may not fit entirely in memory. This means that some memory pages that were dirtied during an incomplete transaction may be flushed to disk to relieve memory pressure. If the system crashes in this scenario, Valor must be able to rollback these flushes during recovery to fulfill its atomicity guarantee. Valor writes *undo* records describing the original state of each affected page to the log when flushing in this way. A page cache that supports flushing dirty pages from uncommitted transactions is known as a *Steal* cache; XFS [140], ZFS [139], and other journaling file systems are No-Steal, which limits their transaction size [144] (see Section 4.1). Valor's solution is a variant of the ARIES transaction recovery algorithm [85].

An Example Figure 4.2 illustrates Valor's write-back mechanism. A process P initially calls the `Lock` system call to acquire access to two data pages in a file, then calls the `Log Append` system call on them, generating the two 'L's in the figure, and then calls `write(2)` to update the data contained in the pages, generating the two 'P's in the figure. Finally, it commits the transaction and quits. The processes did not call `transaction sync`. On the left hand side, the figure shows the state of the system before P commits the transaction; because of Valor's non-durable No-Force logging scheme, data pages and corresponding undo/redo log entries both reside in the page cache. On the right hand side, the process has committed and exited; simple write ordering ensures that the log entries are safely resident on disk if any data pages were written out by `pdflush`.

We now discuss each of Valor's four architectural components in detail. Section 4.2.1 discusses the logging, simple write ordering, and recovery components of Valor. Section 4.2.2 discusses Valor's extended mandatory locking mechanism, and Section 4.2.3 explains Valor's interception mechanism.

4.2.1 The Logging Interface

The design of Valor maintains two logs. A *general-purpose log* records information on directory operations, like adding and removing entries from a directory, and inode operations, like appends or truncations. A *page-value log* records modifications to individual pages in regular files [26]. Currently our Valor implementation supports only accesses to the *page-value log*.

Before writing to a page in a regular file (*dirtying* the page), and before adding or removing a name from a directory, the process must call `Log Append` to prepare the associated undo-redo

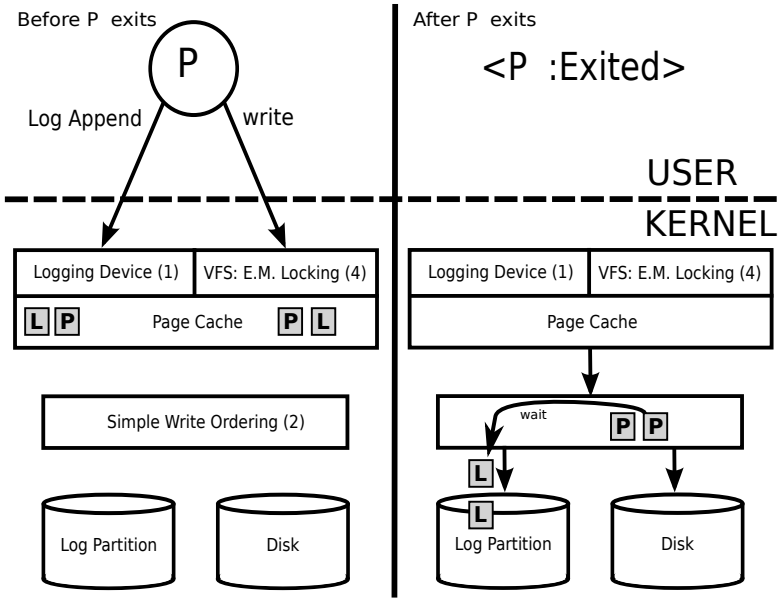

Figure 4.2: Valor Example

record. We refer to this undo-redo record as a *log record*. We found during our implementation that maintaining page alignment within the page-value log is important for performance and so we separated meta-data operations so that they would take place on a separate log.

Since the bulk of file system I/O is from dirtying pages and not directory operations, we run experiments in Section 4.3 that consequently perform negligible meta-data I/O. Valor and Stasis do not log meta-data at all in our experiments. Ext3 and Berkeley DB incur minimal overhead from logging meta-data as our experiments are data intensive. Our recovery experiment only demonstrates the time taken to perform recovery on data-intensive workloads. Valor manages its logs by keeping track of the state of each transaction, and tracking which log records belong to which transactions.

In-Memory Data Structures

There are three states a transaction goes through during the course of its life: (1) *in-flight*, in which the application has called `Log Begin` but has not yet called `Log Resolve`; (2) *landed*, in which the application has called `Log Resolve` but the transaction is not yet safe to deallocate; and (3) *freeing*, in which the transaction is ready to be deallocated. Landed is distinct from freeing because if an application does not require durability, `Log Resolve` causes neither the log nor the data from the transaction to be flushed to disk (see above, *Cooperating with the Kernel Page Cache*).

Valor tracks a transaction by allocating a *commit set* for that transaction. A commit set consists of a unique *transaction ID* and a list of log records. As depicted in Figure 4.3, Valor maintains separate lists of in-flight, landed, and freeing commit sets. It also uses a radix tree to track free on-disk log records.

Life Cycle of a Transaction When a process calls `Log Begin`, it gets a transaction ID by allocating a new log record, called a *commit record*. Valor then creates an in-memory commit set

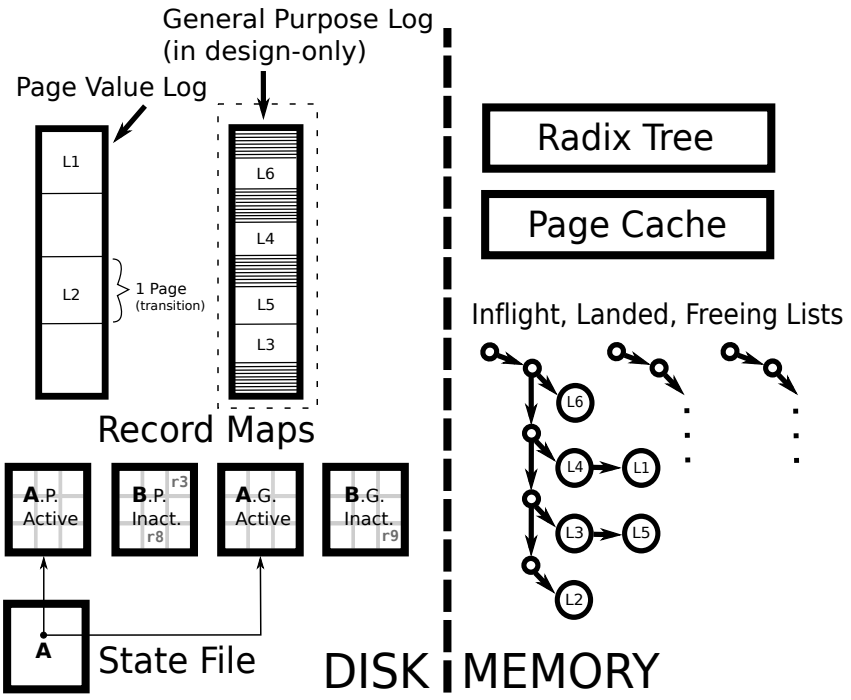

Figure 4.3: Valor Log Layout

and moves it onto the in-flight list. During the lifetime of the transaction, whenever the process calls `Log Append`, Valor adds new log records to the commit set. When the process calls `Log Resolve`, Valor moves its commit set to the landed list and marks it as *committed* or *aborted* depending on the flag passed in by the process. If the transaction is committed, Valor writes a *magic value* to the commit record allocated during `Log Begin`. Because the record has not been written to since the last time the system was in a guaranteed recoverable state, and the record is currently unallocated, this is not an overwrite of a record in the log. If the system crashes and the log is complete, the value of this log record dictates whether the transaction should be recovered or aborted.

Because Valor relies on transition logging, it must enforce that log records are written for each page update. Although it can permit one update to a page without forcing a log record to disk, it cannot permit two or more. The transaction can commit asynchronously and write to other pages, but until the transaction is durably committed to storage, the page remains exclusively locked. Valor keeps a flag in each page in the kernel's page cache. This flag can read *available* or *unavailable*; between the time Valor flushes the page's log record to the log and the time the file system writes the dirty page back to disk, it is marked as unavailable, and processes which try to call `Log Append` to add new log records wait until it becomes available, thus preserving our simple write ordering constraint. This un/available flag is separate from the write locking system that Valor uses to enforce isolation. For hot file-system pages (e.g., those containing global counters), this could result in bursty write behavior. One possible remedy which is a subject of future work is to borrow Ext3's solution: when writing to an *unavailable* page, Valor can create a copy. The original copy remains read-only and is freed after the flush completes. The new copy is used for new reads and writes and is not flushed until the next `pdflush`, maintaining the simple write ordering.

We modified `pdflush` to maintain Valor's in-memory data structures and to obey simple write

ordering by flushing the log's super block before all other super blocks. When `pdflush` runs, it (1) moves commit sets which have been written back to disk to the freeing list, (2) marks all page log records in the in-flight and landed lists as unavailable, (3) atomically transitions the disk state to commit landed transactions to disk, and (4) iterates through the freeing list to deallocate transactions which have been safely written back to disk. The `pdflush` routine in Linux asks the underlying file system to safely write dirty `inodes` to disk. In the case of Ext3, Ext3 initiates a journal flush, which forces the data to the disk platter.

Soft vs. Hard Deallocations Valor deallocates log records in two situations: (1) when a `Log Append` fails to allocate a new log record, and (2) when `pdflush` runs. *Soft deallocation* waits for `pdflush` to naturally write back pages and moves a commit set to the freeing list to be deallocated once all of its log records have had their changes written back. *Hard deallocation* explicitly flushes a landed commit set's dirty pages and directory modifications so it can immediately deallocate it.

On-Disk Data Structures

Figure 4.3 shows the page-value log (implemented) and general-purpose log (in design-only). Valor maintains two *record map* files to act as superblocks for the log files, and to store which log records belong to which transactions. One of these record map files corresponds to the general-purpose log, and the other to the page-value log. For a given log, there are exactly the same number of entries in the record map as there are log records in the log. The five fields of a record map entry are:

Transaction ID The transaction (commit set) this log record belongs to.
Log Sequence Number (LSN) Indicates when this log record was allocated.
inode Inode of the file whose page was modified.
netid Serial number of the device the inode resides on.
offset Offset of the page that was modified.

During recovery, we use the contents of the record map entry to properly recover the contents of a page.

General-purpose log records contain directory path names for recovery of original directory listings in case of a crash. Page value log records contain a specially encoded page to store both the undo and the redo record. The state file is part of the mechanism employed by Valor to ensure atomicity and is discussed further when we describe Valor's log device state transition procedure (LDST).

Transition Value Logging Although the undo-redo record of an update to a page could be stored as the value of the page before the update and the value after, Valor instead makes a reasonable optimization in which it stores only the XOR of the value of the page before and after the update. This is called a *transition page*. Transition pages can be applied to either recover or abort the on-disk image. A pitfall of this technique is that idempotency is lost [43]; Valor avoids this problem by recording the location and value of the first bit of each sector in the log record that differed between the undo and redo image. Although log records are always page-sized, this information must be stored on a per-sector basis as the disk may only write part of the page. (Because meta-data is stored in a separate map, transition pages in the log are all sector-aligned.) If a transaction updates

the same page multiple times, Valor forces each `Log Append` call to wait on the Page Available flag which is set by the simple write ordering component operating within `pdflush`. If it does not have to wait, the call may update the log record's page directly, incurring no I/O. However, if the call must wait, then a new log record must be made to ensure recoverability. Since Valor uses the un/available flag to ensure every page write has a transition record, and since other processes wait on the transaction's write lock on that page before they can make unlogged updates, it will always be the case that if the i^{th} version of a page is written back to disk, then either the i^{th} or $i + 1^{th}$ transition record for that page will be in the log file and no log record for that page more recent then version $i + 1$ will be in the log record.

LDST: Log Device State Transition

Valor's in-memory data structures are a reflection of Valor's on-disk state; however, as commit sets and log records are added, Valor's on-disk state becomes stale until the next time `pdflush` runs. We ensure that `pdflush` performs an atomic transition of Valor's on-disk state to reflect the current in-memory state, thus making it no longer stale. To represent the previous and next state of Valor's on-disk files, we have a *stable* and *unstable* record map for each log file. The stable record maps serve as an authoritative source for recovery in the event of a crash. The unstable record maps are updated during Valor's normal operation, but are subject to corruption in the event of crashes. The purpose of Valor's LDST is to make the unstable record map consistent, and then safely and atomically relabel the stable record maps as unstable and vice versa. This is similar to the scheme employed by LFS [116, 126].

The core atomic operation of the LDST is a pointer update, in which Valor updates the state file. This file is a pointer to the pair of record maps that is currently stable. Because it is sector-aligned and less than one sector in size, a write to it is atomic. All other steps ensure that the record maps are accurate at the point in time where the pointer is updated. The steps are as follows:

1. Quiesce (block) all readers and writers to any on-disk file in the Valor log partition.
2. Flush the inodes of the page-value and general-purpose log files. This flushes all new log records to disk. Log records can only have been added, so a crash at this point has no effect as the stable records map does not point to any of the new entries.
3. Flush the inodes of the unstable page-value and general-purpose record map files.
4. Write the names of the newly stable record maps to the state file.
5. Flush the inode of the state file. The up-to-date record map is now stable, and Valor now recovers from it in case of a system crash.
6. Copy the contents of the stable (previously unstable) record map over the contents of the unstable (previously stable) record map, bringing it up to date.
7. Un-quiesce (unblock) readers and writers.
8. Free all freeing log records.

Any transaction that wants to durably commit must participate in an LDST. However, multiple transactions that update different pages can participate in the same LDST.

Atomicity The atomicity of transactions in Valor follows from two important constraints that Valor ensures that the OS obeys: (1) that writes to the log partition and data partitions obey simple write ordering and (2) that the LDST is atomic. At mount time, Valor runs recovery (Section 4.2.1) to ensure that the log is ready and fully describes the on-disk system state when it is finished mounting. Thereafter, all proper transactional writes are preceded by `Log Append` calls. No writes go to disk until `pdflush` is called or Valor's `Transaction Sync` is called. Simple write ordering ensures that in both cases, the log records are written before the in-place updates, so no update can reach the disk unless its corresponding log record has already been written. Log records themselves are written atomically and safely because writes to the log's backing store are only made during an LDST. Since an LDST is atomic, the state of the entire system advances forward atomically as well.

Performing Recovery

System Crash Recovery During the `mount` operation, the logging device checks to see if there are any outstanding log records by checking if there are any allocated log records in the stable record map, and, if so, runs recovery. During `umount`, the Logging Device flushes all committed transactions to disk and aborts all remaining transactions. Valor can perform recovery easily by reading the state file to determine which record map for each log is stable, and reconstructing the commit sets from these record maps. A log sequence number (LSN) stored in the record map allows Valor to read in reverse order the events captured within the log and play them forward or back based on whether the write needs to be completed to satisfy durability or rolled back to satisfy atomicity. Recovery finds all record map entries and makes a commit set for each of them which is by default marked as aborted. While traversing through record map entries if it finds a record map entry with a magic value (written asynchronously during `Log Resolve`) indicating that this transaction was committed, it marks that set committed. Finally all commit sets are deallocated and an LDST is performed. The system can then come on line.

Process Crash Recovery Recovery handles the case of a system crash, something handled by all journaling file systems. However, Valor also supports user-process transactions and, by extension, user-process recovery. When a process calls the `do_exit` process clean-up routine in the kernel, its `task_struct` is checked to see if a transaction was in-flight. If so, Valor moves the commit set for the transaction onto the landed list and marks the commit set as aborted.

4.2.2 Ensuring Isolation

Extended mandatory locking is a derivation of mandatory locking, a system already present in Linux and Solaris [47, 80]. Mandatory locks are shared for reads but exclusive for writes and must be acquired by all processes that read from or write to a file. Valor adds these additional features: (1) a locking permission bit for owner, group, and all (LPerm), (2) a lock policy system call for specifying how locks are distributed upon `exit`, and (3) the ability to lock a directory (and the requirement to acquire this lock for directory operations). System calls performed by non-transactional processes that write to a file, inode, or directory object acquire the appropriate lock before performing the operation and then release the lock upon returning from the call.

Valor requires *all* processes to take locks via its extended mandatory locking mechanism. By ensuring that even non-transactional system calls acquire and release locks, we guarantee that transactional applications can retain higher degrees of isolation. All processes, even non-transactional processes, obey at least degree 1 isolation. In this environment, then, by the degrees of isolation theorem [43], transactional processes that obey higher degrees of isolation (up to degree 3) can do so.

Valor supports inter-process transactions by implementing inter-process locking. Processes may specify (1) if their locks can be recursively acquired by their children, and (2) if a child's locks are released or instead given to its parent when the child exits. These specifications are propagated to the Extended Mandatory Locking system with the `Lock Policy` system call.

Valor prevents misuse of locks by allowing a process to acquire a lock only under one of two circumstances: (1) if the process has permission to acquire a lock on the file according to the LPerm of the file, or (2) if the process has read access or write access, depending on the type of the lock. Only the owner of a file can change the LPerm, but changes to the LPerm take effect regardless of transactions' isolation semantics. Deadlock is prevented using a deadlock-detection algorithm. If a lock would create a circular dependency, then an error is returned. Transaction-aware processes can then recover gracefully. Transaction-oblivious processes should check the status of the failed system call and return an error so that they can be aborted. We have successfully booted, used, and shutdown a previous version of the Valor system with extended mandatory locking and the standard legacy programs. We observed no deadlocks and major system tools and services did not hang. We believe that Linux as it is, is already close to an operating system that could work in an extended mandatory locking environment. A related issue is the locking of frequently accessed file-system objects or pages. The default Valor behavior is to provide degree 1 isolation, which prevents another transaction from accessing the page while another transaction is writing to it. For transaction-oblivious processes, because each individual system call is treated as a transaction, these locks are short lived. For transaction-aware processes, an appropriate level of isolation can be chosen (e.g., degree 2—no lost updates) to maximize concurrency and still provide the required isolation properties.

4.2.3 Application Interception

Valor supports applications that are aware of transactions but need to invoke sub-processes that are not transaction-aware within a transaction. Such a subprocess is wrapped in a transaction that begins when it first performs a file operation and ends when it exits. This is useful for a transactional process that forks sub-processes (e.g., `grep`) to do work within a transaction. During system calls, Valor checks a flag in the process to determine whether to behave transactionally or not. In particular, when a process is `fork`ed, it can specify if its child is transaction-oblivious. If so, the child has its Transaction ID set to that of the parent and its in-flight state set to Oblivious. When the process performs any system call that constitutes a read or a write on a file, inode, or directory object, the in-flight state is checked, and an appropriate `Log Append` call is made with the Transaction ID of the process. There is currently no support for `mmap` calls made by non-transaction-aware sub-processes. Support could be added by instrumenting the kernel's page-fault handling routine, but this is a subject of future work.

4.3 Evaluation

Valor provides atomicity, isolation, and durability, but these properties come at a cost: writes between the log device and other disks must be ordered, transactional writes incur additional reads and writes, and in-memory data structures must be maintained. Additionally, Valor is designed to provide these features while only requiring minor changes to a standard kernel's design. In this section we evaluate the performance of data-intensive workloads running on Valor and also compare it to Stasis and BDB. Since Valor only implements data logging, we limit ourselves to experiments with a negligible meta-data overhead. Studying the performance of a traditional logging architecture like that used by Valor for meta-data workloads is a subject of future work. In Chapters 5 and 6 we adopt a design that supports arbitrary dictionary relationships and hosts a transactional file system on top of this database, similar to how SchemaFS is hosted on top of the more general-purpose Berkeley DB.

Section 4.3.1 describes our experimental setup. Section 4.3.2 analyzes a benchmark based on an idealized ARIES transaction logger to derive a lower bound on overhead. Section 4.3.3 evaluates Valor's performance for a serial file overwrite. Section 4.3.4 analyzes Valor's concurrent performance. Finally, Section 4.3.5 measures Valor's recovery time for a data-intensive workload that performs no meta-data logging. All benchmarks test scalability of sequential asynchronous writes in a transactional file system.

4.3.1 Experimental Setup

We used four identical machines, each with a 2.8GHz Xeon CPU and 1GB of RAM for benchmarking. Each machine was equipped with six Maxtor DiamondMax 10 7,200 RPM 250GB SATA disks and ran CentOS 5.2 with the latest updates as of September 6, 2008. We compared the results running on separate machines and found that for a four-hour set of different micro-benchmarks, performance results were within 3.6% of each other. We show experiments that were run on the same machine within the same figure. Results shown in different figures may have been run on different machines. To ensure a cold cache and an equivalent block layout on disk, we ran each iteration of the relevant benchmark on a newly formatted file system with as few services running as possible. We ran all tests at least five times and computed 95% confidence intervals for the mean elapsed, system, user, and wait times using the Student's-t distribution. In each case, unless otherwise noted, the half widths of the intervals were less than 5% of the mean. Wait time is elapsed time less system and user time and mostly measures time performing I/O, though it can also be affected by process scheduling. We benchmarked Valor on the modified Valor kernel and all other systems on a stable unmodified 2.6.25 Linux kernel.

Comparison to Berkeley DB and Stasis The most similar technologies to Valor are Stasis and Berkeley DB (BDB): two user level logging libraries that provide transactional semantics on top of a page store for transactions with atomicity and isolation and with optional durability. Valor, Stasis, and BDB were all configured to store their logs on a separate disk from their data, a standard configuration for systems with more than one disk [43]. The logs used by Valor and Stasis were set to 128MB. Berkeley DB's log buffer was set to 128MB. Because we emphasized non-durable transactions during the design of Valor, we configured Stasis and BDB to also use non-durable

transactions. This configuration required modifying the source code of Stasis to open its log without `O_SYNC` mode. Similarly, we configured BDB's environment with `DB_TXN_NOSYNC`. Berkeley DB was configured with a 512MB cache, half the size of the machine's available physical RAM. The `ext3` file system performs writes asynchronously by default. For file-system workloads it is important to be able to perform efficient asynchronous serial writes, so we focused on non-durable transactions performing asynchronous serial writes.

For our BDB benchmark, we used the same database schema as Amino [151]. Each page of data that the file system writes is stored as a key-value pair. The key is the file ID (an identifier similar to an inode number) and page offset and the value is the data to be stored. We used the B-Tree access method as this is the suitable choice for a large file system, and allows for sequentially accessing adjacent pages. We refer to the size of each value stored in the database as the *schema page size*. In conjunction with the schema page size, BDB's *database page size* form an important pair of tuning parameters. The BDB Manual [130] stipulates when either a key or a value cannot be stored directly in the page of the leaf node. When a key or a value cannot fit in the page of the leaf node, they are stored in an *overflow page*. Overflow pages reduce BDB performance, so ideally a data set would always fit within the leaf pages. Each key and value are stored separately in the leaf node and each are allotted $page\_size/(minimum\_keys * 2)$ bytes, where $page\_size$ is the database page size (e.g., 4KB, 8KB, ..., 64KB) and $minimum\_keys$ is by default 2. In the case of an 8KB database page size, a key would be forced into an overflow page if it were larger than 2KB. Similarly, a value would be forced into an overflow page if it were larger than 2KB.

When BDB allocates an overflow page, either only one key or only one value may be stored in the overflow page; if the item's size exceeds the database page size, then it consumes space in units of the database page size (i.e., it cannot use a fraction of an overflow page). Additional complexity is introduced because the overflow page meta-data requires some space. Thus, to achieve the best possible performance, it is critical to select an appropriate and compatible schema and database page sizes. For example, a naive approach that selects 4,096 bytes for both the schema and database page sizes results in every key-value pair being stored across two overflow pages. The first overflow page contains meta-data and the first 4,073 bytes of the schema page, the second overflow page contains meta-data and the remaining 23 bytes of the schema page. As two key-values can never share an overflow page, this second overflow page is less than 1% utilized; wasting both disk space and bandwidth. Similarly, if one increases the database page size to 8,192 bytes (while keeping the schema page size at 4,096 bytes), two values are still unable to fit into a single database page and an overflow page is required. However, after storing the meta-data and schema page, the remaining 50% of the database page is wasted. If however, the schema page size was set to 4,073 bytes; then the key-value pair precisely fits into a single overflow page without wasting any additional space in overflow pages. An alternative design is to increase the database page size so that more than two key-value pairs can fit into each database page. For example, if the database page size is set to 16,384 bytes; then three 4,096-byte schema pages can fit into the leaf node. However, this is only 75% space efficient as the fourth key must be stored in a separate page. Using the maximum configurable database page size of 65,536, fifteen 4,096-byte schema pages fit within a single leaf, which is 93.75% space efficient. We benchmarked each of these strategies, and selected the fastest: a 64KB database page size with a 4KB byte schema page size.

Berkeley DB Expectations Since Berkeley DB's performance was not sufficiently close to the expected theoretical upper bound on performance for our benchmarks, we performed further bench-

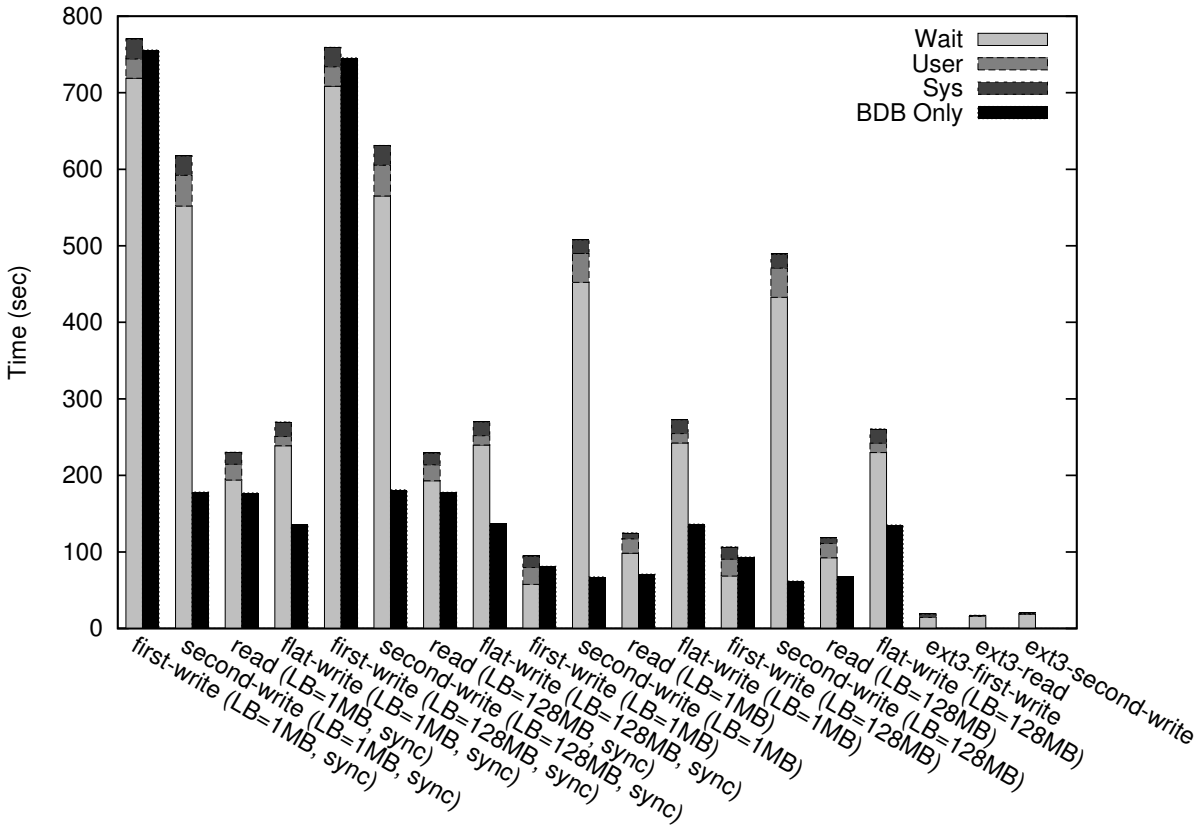

Figure 4.4: Berkeley DB Micro-benchmarks

marking. We expected that Berkeley DB would perform 2–3× slower than Ext3 for asynchronous sequential workloads such as those run in Section 4.3. This is for reasons discussed in Section 3.1.1: we have to write the redo record in addition to the actual page write (2×) and if the write is an overwrite, then we have to sequentially read the page (negligible overhead) and write an additional undo record (3×). We found that SchemaFS running on Berkeley DB would sometimes operate with overheads of 5× for asynchronous sequential workloads—and up to 25× overheads for durable workloads, or asynchronous but large sequential workloads. We also experimented with two alternate independent implementations of a sequential benchmark for Berkeley DB as well as a variety of caching, log-buffer, and durability configurations that we expected would have the largest impact on performance.

Figure 4.4 shows the results of a micro-benchmark intended to determine how much of Berkeley DB's overheads could be credited to our implementation of SchemaFS on top of Berkeley DB. A third party skilled in Berkeley DB programming provided us with a separate benchmark called FLAT-WRITE. This benchmark writes pages with sequentially increasing IDs directly to Berkeley DB. The other Berkeley DB-based benchmarks are implemented using the SchemaFS interface hosted on top of Berkeley DB. Both FLAT-WRITE and the SchemaFS benchmarks use an `<inode,page-offset>` tuple as the key. Both implementations carefully implement the Berkeley DB comparison routine. The SchemaFS system has a sequential read routine based on database cursors that was used to test the contiguity of data pages. The FLAT-WRITE implementation's code was reviewed for correctness.

The SchemaFS-based benchmarks are FIRST-WRITE, SECOND-WRITE, and READ. The FIRST-WRITE benchmark performs a sequential 512MB write using newly allocated keys (like FLAT-WRITE), except it enables asynchronous transactions once when starting the BDB environment, where as FLAT-WRITE passes in a flag into each transaction commit to disable a synchronous commit. The SECOND-WRITE benchmark does a 512MB *overwrite*, re-using the same keys used in FIRST-WRITE, and so should incur reads of the original page and will have to write larger undo-records. The READ benchmark performs a read of the entire 512MB file in SchemaFS. For all reads we disabled `atime` as implemented in SchemaFS. We additionally list EXT3-FIRST-WRITE, EXT3-SECOND-WRITE, and EXT3-READ to compare these sequential writes to the underlying Ext3 file system. The BDB-ONLY measurement counts time spent waiting on BDB calls to return, as opposed to waiting for the test benchmark to exit successfully. For example, the BDB-ONLY measurement may not include any of the time spent by Berkeley DB flushing dirty pages to storage. The LB=1MB configuration indicates the benchmark ran with a 1MB log-buffer, and the LB=128MB configuration indicates the benchmark ran with a 128MB log-buffer.

We found that performance of Berkeley DB was highest when performing asynchronous FIRST-WRITE or FLAT-WRITE as it did not have to gather undo images and write undo records for newly created tuples. Since FLAT-WRITE disables transaction commit on each write, it executes with asynchronous performance as well. We also note that SECOND-WRITE is $5\times$ slower than FIRST-WRITE in asynchronous mode when counting total elapsed time. This difference is because the environment must read in the undo entries, perform undo-redo record writes to the log, and then ultimately write-back the log before it can completely shutdown. Furthermore, we note that the leaf-node and parent-node allocation mechanism used by a B-tree implementation can severely effect its performance for sequential workloads. Typically, file systems that utilize B-trees actually allocate extents in the leaves of their B-tree instead of storing one page per leaf-node like SchemaFS does.

We point out in the following benchmarks when we are performing an overwrite. Regardless, even when waiting only on BDB calls to finish, both FLAT-WRITE and FIRST-WRITE, the fastest asynchronous configurations, take $4.3\times$ longer to complete than EXT3-FIRST-WRITE, even when accounting for time taken by Ext3 to sync to disk.

To account for total time spent performing a transaction, the page cache is written back to disk using `sync` (and simple write-ordering for Valor runs). Since Valor gathers all undo records immediately and submits its transition records to the log-buffer before performing its actual write, all transactional overheads are accounted for in each run, but can be batched together as all benchmarks run with asynchronous (ACI) transactions.

To fully account for transactional overheads incurred by Berkeley DB calls, we measure elapsed time of each benchmark, including time to shutdown its environment.

4.3.2 Mock ARIES Lower Bound

Figure 4.5 compares Valor's performance against a mock ARIES transaction system to see how close Valor comes to the ideal performance for its chosen logging system. We configured a separate logging block device with `ext2`, in order to avoid overhead from unnecessary journaling in the file system. We configured the data block device with `ext3`, since journaled file systems are in common use for file storage. We benchmarked a 2GB file overwrite under three mock systems. MT-ownoread performed the overwrite by writing zeros to the `ext2` device to simulate logging, and then

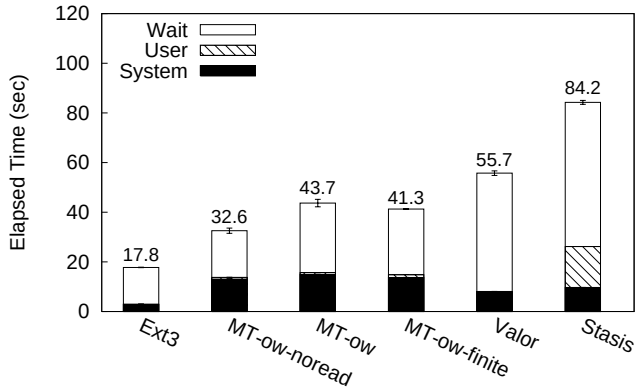

Figure 4.5: Valor's and Stasis's performance relative to the mock ARIES lower bound

writing zeros to the `ext3` device to simulate write back of dirty pages. Valor hosts its journal on `ext2` and its data on `ext3` and this is why our Mock benchmark uses a similar configuration. On one hand we might run Mock ARIES on two `ext2` devices to obtain an even tighter lower bound. On the other hand, we might run the data partition of the Mock ARIES benchmark on `ext3` to obtain a lower bound for a transactional file system: this permits non-transactional operations to utilize `ext3`'s journal, like Valor does. We chose the Mock configuration that more closely matches how Valor behaves by using `ext3` to hold the data partition. Either way in practice the overhead of `ext3` in ordered mode (no data journaling) to `ext2` for asynchronous benchmarks is negligible [122]. We use `ext3` in the default ordered mode.

MT-ow differs from MT-ow-noread in that it copies a pre-existing 2GB data file to the log to simulate time spent reading in the before image. MT-ow-finite differs from the other mock systems in that it uses a 128MB log, forcing it to break its operation into a series of 128MB copies into the log file and writes to the data file. A transaction manager based on the ARIES design must do at least as much I/O as MT-ow-finite. Valor's overhead on top of MT-ow-finite is 35%. Stasis's is 104%. The cost of MT-ow reading the before images as measured by the overhead of MT-ow on MT-ow-noread is 34%. Although MT-ow-finite performs more logical seeks than MT-ow, it is still 6% faster due to repeatedly writing within the same set of physical tracks. In this experiment, we varied the physical location of the tracks by changing the partitioning of the disks; we observed up to a 9% performance difference and concluded that the primary source of difference in execution time was due to performing large writes instead of repeated small flushed writes. Stasis's overhead is more than Valor's overhead due to maintaining a redundant page cache in user space.

4.3.3 Serial Overwrite

In this benchmark we measure the time it takes for a process to transactionally overwrite an existing file. File transfers are an important workload for a file system. See Figure 4.6. Providing transactional protection to large file overwrites demonstrates Valor's ability to scale with larger workloads and handle typical file system operations. Since there is data on the disk already, all systems but `ext3` must log the contents of the page before overwriting it. The transactional systems use a transaction size of 16 pages. The primary observation from these results is that each system scales linearly with respect to the amount of data it must write. Valor runs 2.75 times longer than `ext3`, spending the majority of that overhead writing Log Records to the Log Device. Stasis

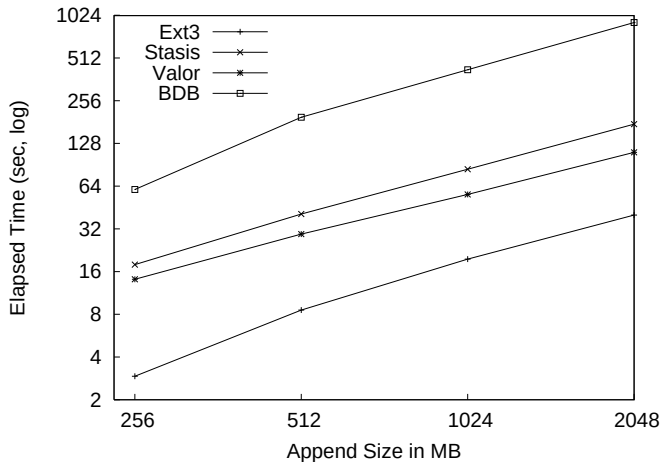

Figure 4.6: Asynchronous serial overwrite of files of varying sizes

runs 1.75 times slower than Valor. It spends additional time allocating pages in user space for its own page cache, and doing additional memory copies for its writes to both its log and its store file. For the 512MB overwrite of Valor and Stasis, and the 256MB overwrite of Stasis, the half-widths were 11%, 7%, and 23%, respectively. The asynchronous nature of the benchmark implies that performance depends in part on how quickly the buffer cache is filled by the write. Other processes executing at the same time can then seriously effect the performance of these benchmarks. This is why we see such a high variance in our measurements of Valor and Stasis in this serial write benchmark. BDB's on-disk B-Tree format stores one file page per leaf-node; this is complex and sensitive to the block allocation policies of Berkeley DB. Valor and Stasis use a simple page-based layout. For example, Valor writes to the underlying file system which allocates an extent that can then be populated with the sequential write without introducing any fragmentation. Valor's page-logging is page aligned: this is an important optimization that gave Valor a 2–3$\times$ speedup. Thanks to these allocation and page alignment policies, Valor runs 8.22$\times$ faster than BDB for this data-heavy workload.

4.3.4 Concurrent Writers

To measure concurrency, we ran varying numbers of processes that would each serially overwrite an independent file concurrently. Each process wrote 1GB of data to its own file. Since the files are independent, the writers need not wait on any locks to write. We ran the benchmark with 2, 4, 8, and 16 processes running concurrently. Figure 4.7 illustrates the results of our benchmark. For low numbers of processes (2, 4, and 8) BDB had half-widths of 35%, 6%, and %5 because of the high variance introduced by BDB's user space page cache. Stasis and BDB run at 2.7 and 7.5 times the elapsed time of `ext3`. For the 2, 4, 8, and 16 process cases, Valor's elapsed time is 3.0, 2.6, 2.4, and 2.3 times that of `ext3`. What is notable is that these times converge on lower factors of `ext3` for high numbers of concurrent writers. The transactional systems must perform a serial write to a log followed by a random seek and a write for each process. BDB and Stasis must maintain their page caches, and BDB must maintain B-Tree structures on disk and in memory. For small numbers of processes, the additional I/O of writing to Valor's log widens the gap between transactional systems

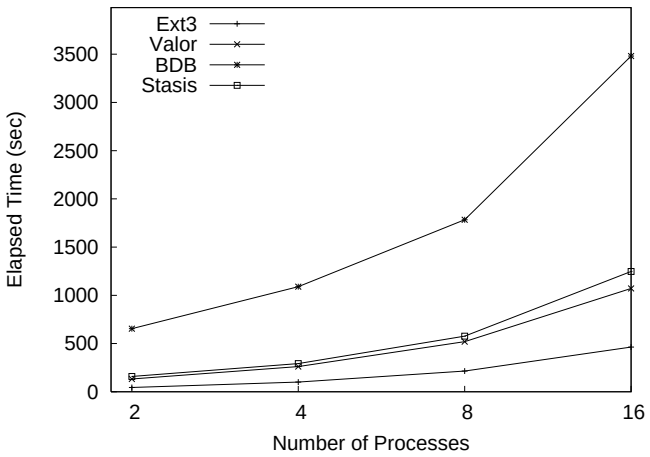

Figure 4.7: Execution times for multiple concurrent processes accessing different files

and `ext3`, but as the number of processes and therefore the number of files being written to at once increases, the rate of seeks overtakes the cost of an extra log serial write for each data write, and maintenance of on-disk or in-memory structures for BDB and Stasis.

4.3.5 Recovery

One of the main goals of a journaling file system is to avoid a lengthy `fsck` on boot [49]. Therefore it is important to ensure Valor can recover from a crash in a finite amount of time with respect to the disk. Although with our current Valor implementation we cannot show that Valor's meta-data recovery time is efficient in comparison to an `fsck`, we can evaluate Valor's data-logging recovery time and show that it grows linearly with the amount of data that is written into the log.

Valor's ability to recover a file after a crash is based on its logging an equivalent amount of data during operation. The amount of total data that Valor must recover cannot exceed the length of Valor's log, which was 128MB in all our benchmarks. Valor's recovery process consists of: (1) reading a page from the log, (2) reading the original page on disk, (3) determining whether to roll forward or back, and (4) writing to the original page if necessary. To see how long Valor took to recover for a typical amount of uncommitted data, we tested the recovery of 8MB, 16MB and 32MB of uncommitted data. In the first trial, two processes were appending to separate files when they crashed, and their writes had to be rolled back by recovery. In the second trial, three processes were appending to separate files. Process crash was simulated by simply calling `exit(2)` and not committing the transaction. Valor first reads the Record Map to reconstitute the in-memory state at the time of crash, then plays each record forward or back in reverse Log Sequence Number (LSN) order. Figure 4.8 illustrates our recovery results. Label `2/8-rec` in the figure shows elapsed time taken by recovery to recover 8MB of data in the case of 2 process crash. We see that although the amount of time spent recovering is proportional to the amount of uncommitted data for both the 2 and 3 process case, that recovering 3 processes takes more time than for 2 because of additional seeking back and forth between pages on disk associated with log records for 3 uncommitted transactions instead of 2. 2/32-rec is 2.31 times slower than 2/16-rec and 2/16-rec is 1.46 times slower than 2/8-rec due to varying size of recoverable data. Similarly, 3/32-rec is 2.04 times slower than

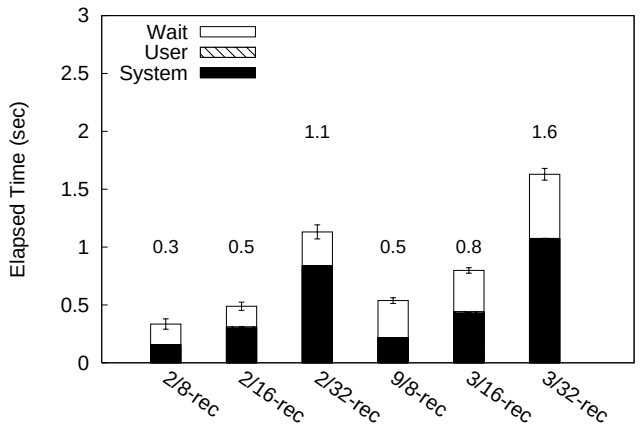

Figure 4.8: ***Recovery benchmark:*** *Time spent recovering from a crash for varying amounts of uncommitted data and varying number of processes*

3/16-rec and 3/16-rec is 1.5 times slower than 3/8-rec. Keeping the amount of recoverable data same we see that 3 processes have 44%, 63%, and 60% overhead compared to 2 process with recoverable data of 8MB, 16MB, 32MB, respectively. In the worst case, Valor recovery can become a random read of 128MB of log data, followed by another random read of 128MB of on-disk data, and finally 128MB of random writes to roll back on-disk data.

Valor does no logging for read-only transactions (e.g., `getdents`, `read`) because they do not modify the file system. Valor only acquires a read lock on the pages being read, and, because it calls directly down into the file system to service the read request, there is no overhead.

Systems which use an additional layer of software to translate file system operations into database operations and back again introduce additional overhead. This is why Valor achieves good performance with respect to other database-based user level file system implementations that provide transactional semantics. These alternative APIs can perform well in practice, but only if applications use their interface, and constrain their workloads to reads and writes that perform well in a standard database rather than a file system. Our system does not have these restrictions.

4.4 Conclusions

Valor performs asynchronous sequential over-writes 2× faster than Stasis. There are more sophisticated user-level transactional libraries that use B-tree values to store page data directly in B-tree leaf nodes; these can perform 8× slower than Valor's page-logging when used on top of Ext3's asynchronous write throughput. Recovery of page data is efficient, taking 1.6 seconds to recover three 32MB appends. For applications that modify a small number of files, or for any larger files, Valor is within 35% overhead of an idealized ARIES implementation; Valor is still 3.6× slower than an unmodified non-transactional Ext3. These results test only data recovery; meta-data recovery from the general purpose log was not tested. Recovery of meta-data could take much longer. For more meta-data–intensive workloads we turn to a more efficient design in Chapter 5. However, even with these data-logging-only results, it is clear that asynchronous sequential file system work-

loads will not fair well on a traditional transactional implementation that must gather undo records. In Chapter 5 we introduce a new transactional database design: it is based on a data structure and transactional architecture that are more suited for workloads consisting of small random writes with complex queries such as a meta-data–intensive file system workload.

Applications can benefit greatly from having a POSIX-compliant transactional API that minimizes the number of modifications needed to applications. Such applications can become smaller, faster, more reliable, and more secure—as we have demonstrated in this and prior work. However, adding transaction support to existing OSs is difficult to achieve simply and efficiently, as we had explored ourselves in several prototypes.

The primary contribution introduced in this chapter is our design of Valor, which was informed by our previous attempts in the forms of KBDBFS and Amino. Valor runs in the kernel cooperating with the kernel's page cache, and runs more efficiently: Valor's performance comes close to the theoretical lower bound for a log-based transaction manager, and scales much better than Amino, BDB, and Stasis.

Unlike KBDBFS, however, Valor integrates seamlessly with the Linux kernel, by utilizing its existing facilities. Valor required less than 100 LoC changes to `pdflush` and another 300 LoC to simply wrap system calls; the rest of Valor is a standalone kernel module which adds less than 4,000 LoC to the stackable file system template Valor was based on.

	Type	**Num Writes**	**Log-Struct.**	**Trans-actions**	**Conc-urrent**	**Async**	**Write Order**	**Random**	**Stitch-ing**	**Sequen-tial**
Ext3	FS	1	¬	MD-only	¬	✓	Kernel	R	¬	R,W
SchemaFS*	FS	3	¬	Logical	✓	✓	User	R	¬	R
LSMFS	FS	1	✓	MD-only	¬	✓	`mmap`	S,W	¬	R,W
Valor	FS	2	¬	POSIX	¬	✓	Kernel	R	¬	R

Table 4.1: ***A qualitative comparison of Transactional Storage Designs:*** *We can conclude that although Valor provides a general-purpose option for adding transactions to file systems from the VFS, it incurs 3–4× overheads for sequential file system workloads.*

Table 4.1 adds the Valor design to our table of transactional design decisions initially introduced as Table 3.1 in Section 3.4. Valor was designed as a file system, and does not support efficient key-value storage and retrieval. Valor writes data twice: once for the transition log entry, and again for the actual page (if it is an overwrite). Valor is not log-structured, and uses a variant of the general-purpose ARIES algorithm which permits it to run on top of and provide transactions for existing storage systems but requires additional writes. Valor provides transactions for POSIX operations, and Valor's current implementation provides logging for data only. Valor uses a page locking scheme so that it can perform transition logging, which prevents it from efficiently performing highly concurrent small durable transactions. Valor provides asynchronous transactions and controls write ordering directly from the kernel. Valor is efficient for all read operations, and inefficient for sequential and random write operations. Valor does not provide a gradual performance trade-off for workloads ranging from sequential to random access patterns.

Valor is modular and can operate on top of existing file systems. However, this modularity comes at a performance cost of a 3.6× overhead on top of an unmodified non-transactional file system (Ext3). If we want to perform asynchronous file system workloads with less overhead compared to standard file systems, we will have to move in the direction of a log-structured architecture.

If we want to better support more meta-data–intensive workloads, not just logging them but also indexing them efficiently, then we will have to explore more sophisticated key-value storage data structures like those used in contemporary databases.

In Chapter 5 we explore our transactional log-structured database architecture based on the log-structured merge-tree. This architecture can avoid double writes for asynchronous transactions, better support complex workloads such as file system meta-data workloads, and provide a flexible multi-tier architecture for incorporating various storage device technologies.

Chapter 5

GTSSLv1: A Fast-inserting Multi-tier Database for Tablet Serving

In Chapter 4, we have seen that a partially in-kernel and user-level approach for adding new abstractions such as system transactions can be nearly as efficient as a completely in-kernel approach, but considerably easier to debug, and require few changes to the kernel itself. However, the logging algorithm that we had to use for a modular transactional system was too slow. Valor could extend transactional POSIX semantics to any underlying file system but it had to impose a 2–3× overhead on asynchronous sequential write performance. Furthermore, Valor did not provide effective key-value storage. We intend to add support for key-value storage to the file system, and to provide efficient execution of workloads for database and more sequential file system workloads.

In this chapter we introduce GTSSLv1, a log-structured merge tree which operates efficiently on multiple heterogeneous devices, and a carefully designed and implemented cache, transaction manager, journal, recovery system, and data structure. GTSSLv1 was also designed to be the primary storage management system for a single node. GTSSLv1 was designed as a *tablet server*, a database program that runs on almost every node of a large structured data store such as Big Table [20]. Tablet servers typically require high throughput indexing and support for complex queries. In Chapter 6 we discuss extending GTSSLv1 to better support sequential file system workloads. However, first we must describe how GTSSLv1's design can handle tablet server workloads so that it is clear to the reader how GTSSLv1 and GTSSLv2 can handle tablet serving and other database workloads. We do not discuss message passing or networking between such nodes, but only the database software that runs on these nodes, and compare to existing similar software such as Cassandra [66] and HBase [6].

We found that for database workloads, the GTSSLv1 architecture provides 2× faster insertions for asynchronous transactional workloads than existing popular LSM-tree databases such as Cassandra and HBase. This performance difference was due to architectural differences in how GTSSLv1 provides support for transactions. For random read and write workloads, GTSSLv1 performs between 3–5× faster than Cassandra and HBase. This performance difference was mainly due to implementation differences between Cassandra and HBase, and GTSSLv1. Cassandra and HBase are implemented in Java for easier development. We found out that even for workloads

consisting of the relatively standard pair size of 1KB [24], these Java based systems became CPU-bound due to caching overheads. GTSSLv1 provides details on expanding the database architecture used by Cassandra and HBase to support multi-tier configurations where Flash SSD, magnetic disk, RAM, and other devices are all leveraged to improve performance of workloads. When we insert a Flash SSD into a traditional RAM+HDD storage stack, GTSSLv1's insertion throughput increased by 33%, and our lookup throughput increased by 7.4×. This allows the bulk of colder data to reside on inexpensive media, while most hot data automatically benefits from faster devices. We explain the design of GTSSLv1 in this chapter, showing how to create an efficient transactional database for indexing workloads, traditional database workloads, and multi-tier support. We show how existing implementations of the BigTable architecture must be wary of implementation overheads or will be significantly hampered when randomly accessing even moderately sized tuples (i.e., 1KB or smaller tuples).

The way systems like Big Table, HBase, and Cassandra interact with underlying storage devices is a critical part of their overall architecture. Cluster systems comprise many individual machines that manage storage devices. These machines are managed by a networking layer that distributes to them queries, insertions, and updates. Each machine runs database or database-like software that is responsible for reading and writing to the machine's directly attached storage. Figure 5.1 shows these individual machines, called *tablet servers*, that are members of the larger cluster. We call the portion of this database software that communicates with storage, the *Tablet Server Storage Layer* (TSSL). For example, Hadoop HBase [6], a popular cluster technology, includes a database, networking, and an easier-to-program abstraction above their TSSL (*logical layer*).

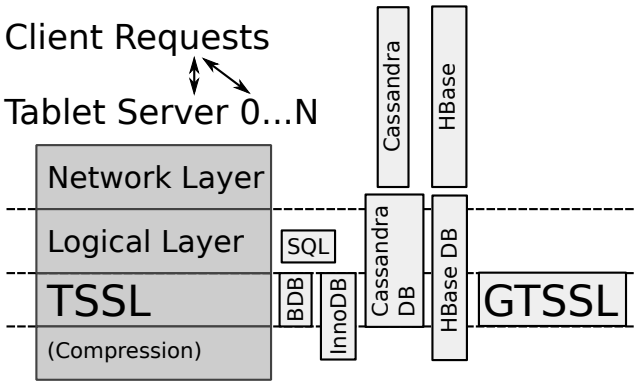

Figure 5.1: ***Location of the Tablet Server Storage Layer:*** *Different storage and communication technologies are used at different layers in a typical database cluster architecture.*

The performance and feature set of the TSSL running on each node affects the entire cluster significantly. If performance characteristics of the TSSL are not well understood, it is difficult to profile and optimize performance. If the TSSL does not support a critical feature (e.g., transactions), then some programming paradigms can be difficult to implement efficiently across the entire cluster (e.g., distributed and consistent transactions).

It takes time to develop the software researchers use to analyze their data. The programming model and abstractions they have available to them directly affect how much time development takes. This is why many supercomputing/HPC researchers have come to rely upon structured data clusters that provide a database interface, and therefore an efficient TSSL [2, 104].

Still, some supercomputing researchers develop custom cluster designs for a particular application that avoids logical layer overheads (such as SQL) when necessary [27, 74, 91]. These researchers still want to understand the performance characteristics of the components they alter or replace, and *especially* if those components are storage-performance bottlenecks. Cooper compared the performance of 5-node Cassandra and HBase clusters [24] to a shared SQL cluster. Pavlo compares the efficiency of more powerful parallel DBMS clusters running on high-end hardware [2, 102] to widely used NoSQL cluster software. These research efforts show an emphasis on understanding single-node storage performance for understanding overall cluster performance.

One of the most important components to optimize in the cluster is the TSSL. This is because affordable storage continues to be orders of magnitude slower than any other component, such as the CPU, RAM, or even a fast local network. Any software that uses storage heavily, such as major scientific applications, transactional systems [85, 135], databases [97], file systems [65, 122, 123, 131, 142], and more, are designed around the large difference in performance between RAM and storage devices. Using a flexible cluster architecture that can scale well with an increasing number of nodes is an excellent way to prorate these storage and computing costs [27]. Complimentary to this effort is increasing the *efficiency* of each of these nodes' TSSLs to decrease overall computing costs.

In this work we present a new, highly scalable, and efficient TSSL architecture called the *General Tablet Server Storage Layer* or GTSSLv1. GTSSLv1 employs significantly improved compaction algorithms that we adapted to multi-tier storage architectures. GTSSLv1 supports general-purpose transactions of multiple tuples. GTSSLv1 avoids a 2× overhead for asynchronous I/O-bound transactional workloads that other LSM-tree–based databases such as HBase, Big Table, and Cassandra pay. We describe and analyze the compaction algorithms used by HBase and Cassandra. This is important for understanding what workloads these various systems are best suited for. We identify the importance of CPU efficiency for small tuple workloads. We show that an LSM-tree–based database is sensitive to implementation choices for small tuple workloads and that Java-based implementations suffer a 5× overhead in comparison to GTSSLv1, a comparable C++ implementation. By focusing on a single node we were better able to understand the performance and design implications of these existing TSSLs, and how they interact with Flash SSD devices when using small as well as large data items. In summary, we provide a comprehensive overview of the performance issues applicable to all LSM-tree–based tablet servers or databases.

We introduce the standard TSSL architecture and terminology in Section 5.1. We theoretically analyze existing TSSL compaction techniques in Section 5.2. In Section 5.3 we introduce GTSSLv1's design and compare it to existing TSSLs. Our evaluation, in Section 5.4, compares the performance of GTSSLv1 to Cassandra and HBase, and GTSSLv1's transactional performance to Berkeley DB and MySQL's InnoDB. We discuss related work in Section 5.5. We conclude and discuss future work in Section 5.6.

5.1 Background

As shown in Figure 5.1, the data stored in the TSSL is accessed through a high-level logical interface. In the past, that interface has been SQL. However, HBase and Cassandra utilize a logical interface other than SQL. As outlined by Chang et al. [20], HBase, Cassandra, and Big Table organize structured data as several large tables. These tables are accessed using a protocol that

groups columns in the table into *column families*. Columns with similar compression properties and columns that are frequently accessed together are typically placed in the same column family. Figure 5.1 shows that the logical layer is responsible for implementing the column-based protocol used by clients, using an underlying TSSL.

The TSSL provides an API that allows for atomic writes, as well as lookups and range queries across multiple *trees* that efficiently store variable length key-value pairs or *pairs* on storage. These trees are the on-storage equivalent of column families. Although TSSLs transactionally manage a set of tuple trees to be operated on by the logical layer, TSSL design is typically different from traditional database design. Both the TSSL and a traditional database perform transactional reads, updates, and insertions into multiple trees, where each tree is typically optimized for storage access (e.g., a B+-tree [23]). In this sense, and as shown in Figure 5.1, the TSSL is very similar to an embedded database API such as Berkeley DB (BDB) [130], or a DBMS storage engine such as MySQL's InnoDB. However, unlike traditional database storage engines, the majority of insertions and lookups handled by the TSSL are *decoupled*. To understand what decoupling is, consider a large clustered system: one process may be inserting a large amount of data gathered from sensors, a large corpus, or the Web, while many other processes perform lookups on what data is currently available (e.g., search). Furthermore, most of these insertions are simple updates, and do not involve large numbers of dependencies across multiple tables. This leads to two important differences from traditional database storage engine requirements: (1) most insertions do not depend on a lookup, not even to check for duplicates, and (2) their transactions typically need only provide atomic insertions, rather than support multiple read and write operations in full isolation. We call the relaxation of condition (1) *decoupling*, and it permits the use of efficient *write-optimized* tree data-structures that support higher insertion throughputs such as the LSM-tree. These kinds of data-structures are different from traditional B+-trees. We call the relaxation of condition (2) *micro-transactions*; it enables using a simple, non-indexed, redo-only journal. Thus, the TSSL need not support a mix of asynchronous and durable transactions. The GTSSLv1 and GTSSLv2 systems do not support only micro-transactions, but Cassandra and HBase are limited to supporting only these simpler kinds of transactions. In Section 5.3 we show how to extend an LSM-tree-based TSSL to support more complex transactions.

5.2 TSSL Compaction Analysis

GTSSLv1 was designed to scale to a multi-tier storage hierarchy, and much of how compaction works must be re-thought. We analyze and compare the existing compaction methods employed by HBase and Cassandra, and then in Section 5.3, we introduce what extensions are necessary in a multi-tier regime.

We analyzed the compaction performance of Cassandra, HBase, and our GTSSLv1 using the *Disk-Access Model* (DAM) for cost and using similar techniques as those found in Bender et al.'s work on the R-fanout cache-oblivious look-ahead array (R-COLA) [11]. We introduced the DAM model in Section 3.2.1.

HBase Analysis HBase [28] is a variation of the R-fanout Cache-Oblivious Look-ahead Array (R-COLA) [11]. The R-COLA can dynamically trade off insert throughput for lower query latency by increasing R. The value of R can go from 2 to B. If $R = B$ then the R-COLA is as efficient

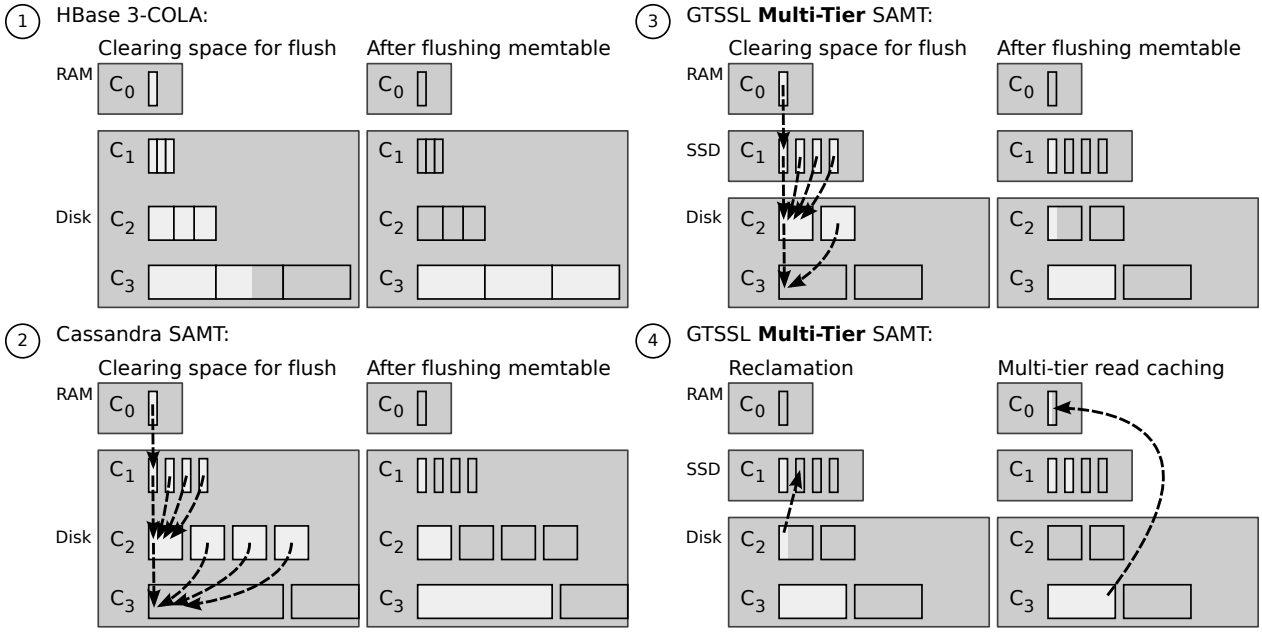

Figure 5.2: ***LSM-tree and MT-SAMT analysis:*** *In panel ①, HBase merges C_0, C_1, C_2, and half of C_3 back into C_3, like a 3-COLA would. In panel ②, Cassandra merges buffers in quartets to create space for a flushing memtable. In panels ③ and ④, GTSSLv1 merges, and then promotes the resulting Wanna-B-tree up into a higher tier. Subsequent reads will also be cached into the higher tier via re-insertion.*

as a B-tree for queries, and only *random* insertions. Typically $R = 3$ or $R = 4$. HBase uses $R = 3$ or a *3-COLA*. In this analysis, N is the number of tuples that have been inserted into a data structure. Figure 5.2, panel ①, shows that the R-COLA consists of $\lceil \log_R N \rceil$ arrays of exponentially increasing size, stored contiguously (C_0 through C_3). In this example, $R = 3$. C_1 through C_3 on storage can be thought of as three Wanna-B-trees, and C_0 in RAM can be thought of as the memtable. When the memtable is serialized to disk (and turned into an Wanna-B-tree), the R-COLA checks to see if level 0 is full. If not, it performs a merging compaction on level 0, on all adjacent subsequent arrays that are also full, and on the first non-full level, into that same level. In Figure 5.2's example, C_0 through C_3 are merged into C_3; after the merge, the original contents of C_3 have been written twice to C_3. Each level can tolerate $R - 1$ merges before it too must be included in the merge into the level beneath it. This means that every pair is written $R - 1$ times to each level.

Bender et al. provide a full analysis of the R-COLA; in sum, the amortized cost of insertion is $\frac{(R-1)\log_R N}{B}$, and the cost of lookup is $\log_R N$. This is because every pair is eventually merged into each level of the R-COLA; however, it will be repeatedly merged into the same level $R - 1$ times. So for N total pairs inserted, each pair would have been written $R - 1$ times to $\log_R N$ levels. As all pairs are written serially, we pay 1 in the DAM for every B pair written, and so we get $\frac{(R-1)\log_R N}{B}$ amortized insertion cost. A lookup operation must perform 1 random read transfer in each of $\log_R N$ levels for a total cost of $\log_R N$. Bender et al. use fractional cascading [21] to ensure only 1 read per level. Practical implementations and all TSSL architectures, however, simply use small secondary indexes in RAM.

By increasing R, one can decrease lookup costs in exchange for more frequent merging during insertion. HBase sets $R = 3$ by default, and uses the R-COLA compaction method. HBase adds additional thresholds that can be configured. For example, HBase performs major compactions when the number of levels exceeds 7.

SAMT Analysis The R-COLA used by HBase has faster lookups and slower insertions by increasing R. GTSSLv1 and Cassandra, however, can both be configured to provide faster *insertions* and slower lookups by organizing compactions differently. We call the structure adopted by Cassandra's TSSL and GTSSLv1, the *Sorted Array Merge Tree* (SAMT). As shown in Figure 5.2, panel ②, rather than storing one list per level, the SAMT stores K lists, or *slots* on each level. The memtable can be flushed K times before a compaction must be performed. At this time, only the slots in C_1 are merged into a slot in C_2. In the example depicted, we must perform a cascade of compactions: the slots in C_2 are merged into a slot in C_3, so that the slots in C_1 can be merged into a slot in C_2, so that the memtable in C_0 can be serialized to a slot in C_1. As every element visits each level once, and merges are done serially, we perform $\frac{\log_K N}{B}$ disk transfers per insertion. Because there are K slots per level, and $\log_K N$ levels, we perform $K \log_K N$ disk transfers per lookup. The cost of lookup with the SAMT is the same for $K = 2$ and $K = 4$, but $K = 4$ provides faster insertions. So $K = 4$ is a good default, and is used by both GTSSLv1 and Cassandra.

Comparison Although the HBase 3-COLA method permits more aggressive merging during insertion to decrease lookup latency by increasing R, it is unable to favor insertions beyond its default configuration. This permits faster scan performance on disk, but for 64B or larger keys, random lookup performance is already optimal for the default configuration. This is because for the vast majority of lookups, Bloom filters [15] on each Wanna-B-tree avoid all $\log_R N$ Wanna-B-trees except the one containing the desired pair. Furthermore, on Flash SSD the 3-COLA is less optimal, as even the seeking incurred from scanning is mitigated by the Flash SSD's obliviousness toward random and serial reads. We found that 256KB random reads operate at 98% the throughput of 64MB random reads on our Flash SSD. This means that the amount of read-ahead buffer required to merge multiple lists at disk throughput can be small for Flash SSD, and that the initial cost of placing cursors is the primary difference between scanning within a SAMT or a COLA. The cost of placing a cursor in a SAMT compared to a 2-COLA is $K \log_K N / \log_2 N$ times more expensve, and K times if comparing an R-COLA for $R = K$. For reasonable values of K (e.g., $K < 10$), a SAMT will never take more than $3\times$ more seeks to place a cursor than a 2-COLA, but will perform inserts $3\times$ faster. Where the cost of scan can be mitigated by advancing the cursor K times to effectively diminish the overhead of placing cusrors in the SAMT close to 2, the cost of insertion can never be improved upon by the COLA for any insertion workload.

Conversely, the SAMT can be configured to further favor insertions by increasing K, while maintaining lookup performance on Flash SSD and disk by using Bloom filters, and maintaining scan performance on Flash SSD. Although Bloom filters defray the cost of unnecessary lookups in Wanna-B-trees, as the number of filters increases, the total effectiveness of the approach decreases. When performing a lookup in the SAMT with a Bloom filter on each Wanna-B-tree, the probability of having to perform an unnecessary lookup in some Wanna-B-tree is $1 - (1 - f)^{N_B}$ where N_B is the number of Bloom filters, and f is the false positive rate of each filter. This probability is roughly equal to $f * N_B$ for reasonably small values of f. In our evaluation, Bloom filters remain

effective as long as the number of Wanna-B-trees for each tree/column-family is less than 40.

5.3 Design and Implementation

We studied existing TSSLs (Cassandra and HBase) as well as existing DBMS storage engines (Berkeley DB and InnoDB). This guided GTSSLv1's design. GTSSLv1 utilizes several novel extensions to the SAMT (discussed in Section 5.2). As shown in Figure 5.2 panels ③ and ④, GTSSLv1 supports storage device specific optimizations at each tier. GTSSLv1 intelligently migrates recently written *and read* data between tiers to improve both insertion and lookup throughput and permit effective caching in storage tiers larger than RAM.

TSSL efficiency is critical to overall cluster efficiency. GTSSLv1 extends the scan cache (described in Section 5.1) and buffer cache architecture used by existing TSSLs. GTSSLv1 completely avoids the need to maintain a buffer cache while avoiding common `mmap` overheads; GTSSLv1 further aggressively exploits Bloom filters so they have equal or more space in RAM than the scan cache.

Although Web-service MapReduce workloads do not typically require more than atomic insertions [20], parallel DBMS architectures and many scientific workloads require more substantial transactional semantics. GTSSLv1 introduces a light-weight transactional architecture that allows clients to commit transactions as either durable or non-durable. Durable transactions fully exploit group-commit as in other TSSL architectures. However, GTSSLv1 also allows non-durable transactions, and these can avoid writing to the journal completely for heavy insertion workloads without compromising recoverability. In addition, GTSSLv1 provides the necessary infrastructure to support transactions that can perform multiple reads and writes atomically and with full degree three isolation.

We discuss how we improved the SAMT structure so that it could operate in a multi-tier way that best exploits the capabilities of different storage devices in Section 5.3.1. We detail our caching architecture and design decisions in Section 5.3.2. We discuss GTSSLv1's transactional extensions to the typical TSSL in Section 5.3.3.

5.3.1 SAMT Multi-Tier Extensions

As discussed in the introduction of this chapter and in Section 5.1, GTSSLv1 is a TSSL and is comparable in functionality to a database storage engine such as MySQL's InnoDB or Berkeley DB. GTSSLv1 allows applications to create tuple trees that can store key-value tuples. GTSSLv1 uses an extended SAMT data structure called the *Multi-Tier-SAMT* (MT-SAMT) as its tuple tree. So an installation of GTSSLv1 would use an instance of an MT-SAMT for each tuple tree. In this way, the MT-SAMT is to GTSSLv1 what the B-tree is to a traditional database storage engine. The MT-SAMT extends the SAMT merging method in three ways. (1) Client reads can be optionally re-inserted to keep recently read (hot) data in faster tiers (e.g., a Flash SSD). (2) Lists of recently inserted data are automatically promoted into faster tiers if they fit. (3) Different tiers can have different values of K (the number of slots in each level; see Section 5.2). Our implementation also includes support for full deletion, and variable-length keys and values. Deletes are processed by inserting tombstones as described in Section 3.2.2.

One way of looking at tombstones is as though they are *messages*. These messages are processed during the merge by combining them with the tuples they are supposed to affect or convey their message to. In this sense, deletes or tombstones are one kind of message that deliver their message (deletion) when finding matching tuples during the merge, but there are others [20]. For example, a deduplicating system may maintain a schema where the key is a checksum and the value is an offset to a content-addressed block [153]. Since many files may point to the same block, we cannot remove this block when deleting a file as it will remove the block for other files as well. Instead, we add a reference count and introduce a new type of message similar to how a tombstone works. This new message is called an *increment*, or conversely, a *decrement*. With increments, we can maintain reference counts which are updated during a merge, and combined during lookup. When merging a tuple with an increment, we add the value of the increment to the tuple's reference count, and when that count exceeds some threshold (e.g., goes below zero) we perform some action (e.g., deleting the reference block).

Another example of a message is a timestamp. Rather than deleting items immediately we can add a *timestamp* value to the delete message or tombstone. During the merge we do not actually perform the delete operation on matching tuples, unless the current time is past the timestamp listed in the tombstone. This way we can mark tuples for deletion but delete them only after a period of time. This can be used to obey data retention policies or regulatory compliance laws, for example. The MT-SAMT design does not preclude extensions for increments, timestamps, or other common types of messages that more complex than tombstones.

Re-Insertion Caching Whenever a pair is inserted, updated, deleted, or read, the C_0 (fastest) cache is updated. The cache is configured to hold a preset number of pairs. When a pair is inserted or updated, it is marked DIRTY, and the number of pairs in the cache is increased. Similarly, after a key is read into the C_0 cache, it is marked as RD_CACHED, and the number of pairs is increased. Once a pre-set limit is met, the cache *evicts* into the MTSAMT structure using the merging process depicted in Figure 5.2 panel ③. By including RD_CACHED pairs in this eviction as regular updates, we can answer future reads from C_1 rather than a slower lower level. However, if the key-value pairs are large, this can consume additional write bandwidth. This feature is desirable when the working-set is too large for C_0 (RAM) but small enough to fit in a fast-enough device residing at one of the next several levels (e.g., C_1 and C_2 on Flash SSD). Alternatively, this feature can be disabled for workloads where saving the cost of reading an average pair is not worth the additional insertion overhead, such as when we are not in a multi-tier scenario. All RD_CACHED values are not written to the output Wanna-B-tree during a major compaction. During a minor compaction, a RD_CACHED pair is treated like any normal insertion and is eliminated during the merge if a more recent tuple has the same key. At any point a tier can relieve space pressure by re-compacting its Wanna-B-trees and removing any RD_CACHED pairs. This would be analogous to a cache eviction. Since RD_CACHED pairs are treated as updates, no additional space is used by inserting RD_CACHED pairs.

When scanning through tuple trees, if read caching is enabled, the scanner inserts scanned values into the cache, and marks them as RD_CACHED. We have found that randomly reading larger tuples (>4096KB) can make effective use of a Flash SSD tier. However for smaller tuples (<64B) the time taken to warm the Flash SSD tier with reads is dominated by the slower random read throughput of the magnetic disk in the tier below. By allowing scans to cache read tuples, applications can exploit application-specific locality to pre-fetch pairs within the same or adjacent

rows whose contents are likely to be later read.

Evictions of read-cached pairs can clear out a Flash SSD cache if those same pairs are not intelligently brought back into the higher tier which they were evicted from after a cross-tier merging compaction. In Figure 5.2 panel ④, we see evicted pairs being copied back into the tier they were evicted from. This is called *reclamation*, and it allows Wanna-B-trees, including read-cached pairs, that were evicted to magnetic disks (or other lower-tier devices) to be automatically copied back into the Flash SSD tier if they can fit.

Space Management and Reclamation We designed the MTSAMT so that more frequently accessed lists would be located at higher levels, or at C_i for the smallest i possible. After a merge, the resulting list may be smaller than the slot it was merged into because of resolved deletes and updates. If the resulting list can fit into one of the higher (and faster) slots from which it was merged (that are now clear), then it is moved upward, along with any other slots at the same level that can also fit. This process is called *reclamation* and requires that the total amount of pairs in bytes that can be reclaimed must fit into half the size of the level they were evicted from. By only reclaiming into half the level, a sufficient amount of space is reserved for merging compactions at that level to retain the same asymptotic insertion throughput. In the example in Figure 5.2, the result of the merging compaction in panel ③ is small enough to fit into the two (half of four) available slots in C_1, and specifically in this example requires only one slot. If multiple slots were required, the Wanna-B-tree would be broken up into several smaller Wanna-B-trees. This is possible because unlike Cassandra and HBase, GTSSLv1 manages blocks in the underlying storage device directly, rather than treating Wanna-B-trees as entire files on the file system, which allows for this kind of optimization. Reclamation across levels within the same tier is very inexpensive, as this requires merely *moving* Wanna-B-tree blocks by adjusting pointers to the block, rather than *copying* them across devices. If these rules are obeyed, then partially filled slots are guaranteed to always move upward, eliminating the possibility that small lists of pairs remain stuck in lower and slower levels.

We optimized our MTSAMT implementation for throughput. Our design considers space on storage with high latency and high read-write throughput characteristics (e.g., disk) to be cheaper than other hardware (e.g., RAM or Flash SSD). GTSSLv1 can operate optimally until $1/2$ of total storage is consumed; after that, performance degrades gradually until the entire volume is full, save a small amount of reserve space (usually 5% of the storage device). (Such space-time tradeoffs are common in storage systems [81], such as HBase [28], Cassandra [66], and even Flash SSD devices [55], as we elaborate further below.) At this point, only deletes and updates are accepted. These operations are processed by performing the equivalent of a major compaction: if there is not enough space to perform a merging compaction into the first free slot, then an in-place compaction of all levels in the MTSAMT is performed using the GTSSLv1's reserve space. As tuples are deleted, space is reclaimed, freeing it for more merging compactions that intersperse major compactions until $1/2$ of total storage is again free; at that point, only merging compactions need be performed, regaining the original optimal insertion throughput.

Chang et al. do not discuss out of space management in Big Table [20] except to say that a major compaction is performed in those situations; they also do not indicate the amount of overhead required to perform a major compaction. Cassandra simply requires that half of the device remain free at all times [66] and ceases to operate if half of the device is not free, arguing that disk storage is cheap. It is not uncommon for write-optimized systems, such as modern Flash SSD firmware, to require a large amount of storage to remain free for compaction. High performance Flash SSD

devices build these space overheads (among other factors) into their total cost [34]. Even commodity Flash SSD performs far better when the partition actually uses no more than 60% of the total storage capacity [55]. To exploit decoupling, compaction-based systems such as GTSSLv1 have some overhead to maintain optimal insertion throughput in the steady state, without this space their throughput will degrade. We believe that GTSSLv1's gradual degradation of performance beyond 50% space utilization is a sufficient compromise.

5.3.2 Committing and Stacked Caching

We showed how the MTSAMT extends the typical SAMT to operate efficiently in a multi-tier environment. In addition to efficient compaction, reclamation, and caching as discussed above, the efficiency of the memtable or C_0 (Section 5.1) as well as how efficiently it can be serialized to storage as an Wanna-B-tree is also extremely important. As we evaluate in Section 5.4, the architecture of the transaction manager and caching infrastructure is the most important determiner of insertion throughput for small key-value pairs ($<$ 1KB). GTSSLv1's architecture is mindful of cache efficiency, while supporting new transactional features (asynchronous commits) and complex multi-operation transactions.

Cache Stacking The transactional design of GTSSLv1 is implemented in terms of GTSSLv1's concise cache-stacking feature. Like other TSSLs, GTSSLv1 maintains a memtable to store key-value pairs. GTSSLv1 uses a red-black tree with an LRU implementation, and DIRTY flags for each pair. An instance of this cache for caching pairs in a particular column family or tree is called a *scan cache*. Unlike other TSSL architectures, this scan cache can be stacked on top of another cache holding pairs from the same tree or MTSAMT. In this scenario the cache on top or the *upper cache* evicts into the *lower cache* when it becomes full by locking the lower cache and moving its pairs down into the lower cache. This feature simplifies much of GTSSLv1's transactional design, which we explore further in Section 5.3.3. In addition to the memtable cache, like other TSSLs, GTSSLv1 requires a buffer cache, but as we discuss in the next paragraph, we do not need to fully implement a user-level buffer cache as traditional DBMSes typically do.

Buffer Caching TSSLs utilized in cloud-based data stores such as Cassandra, HBase, or GTSSLv1 never overwrite data during the serialization of a memtable to storage, and therefore need not pin buffer-cache pages, greatly simplifying their designs. We offload to the Linux kernel all caching of pages read from storage, by `mmap`ing all storage in 1GB *slabs*. This simplifies our design as we avoid implementing a buffer cache. 64-bit machines' address spaces are sufficient and the cost of a random read I/O far exceeds the time spent on a TLB miss. Cassandra's default mode is to use `mmap` within the Java API to also perform buffer caching. However, serial writes to a mapping incur reads as the underlying Linux kernel always reads the page into the cache, even on a write fault. This can cause overheads on serial writes of up to 40% in our experiments. Other TSSL architectures such as Cassandra do not address this issue. To avoid this problem, we `pwrite` during merges, compactions, and serializations. The `pwrite` call is the same as `write` but takes an additional offset parameter so that an additional `lseek` call is not required. We then invalidate only the affected mapping using `msync` with `MS_INVALIDATE`. As the original slots are in place during the merge, reads can continue while a merge takes place, until the original list must be deallocated.

Once deallocated, reads can now be directed to the newly created slot. The result is that the only cache which must be manually maintained for write-ordering purposes is the journal cache. The journal cannot be written to disk until the `mmap`ed buffer cache has been flushed to disk so we use a journal cache for recent appends to the journal. When this cache fills, we flush the `mmap`ed buffer cache and then write the journal cache to the end of the journal.

All TSSLs that employ `mmap`, even without additionally optimizing for serial writes like GTSSLv1, typically avoid read overheads incurred by a user-space buffer cache. On the other hand, traditional DBMSes cannot use `mmap` as provided by commodity OSes. This is because standard kernels (e.g., Linux) have no portable method of pinning dirty pages in the system page cache. Without this, or some other write-ordering mechanism, traditional DBMSes that require overwrites (e.g., due to using B+-trees), will violate write-ordering and break their recoverability. Therefore they are forced to rely on complex page cache implementations based on `malloc` [43, 120, 135] or use complex kernel-communication mechanisms [133–135].

5.3.3 Transactional Support

Pavlo et al. [102] and Abouzeid et. al [2] use traditional parallel DBMS architectures for clustered structured data workloads, but these still rely on distributed transaction support. GTSSLv1's transactional architecture permits for atomic durable insertions, batched insertions for higher insertion-throughput, and larger transactions that can be either asynchronous or durable. This lets the same TSSL architecture to be used in a cluster operating under either consistency model.

We described MTSAMT's design and operation and its associated cache or memtable (C_0). As mentioned before, each MTSAMT corresponds to a tree or column family in a cloud storage center. GTSSLv1 operates on multiple MTSAMTs to support row insertions across multiple column families, and more complex multi-operation transactions as required by stronger consistency models. Applications interact with the MTSAMTs through a transactional API: `begin`, `commit_durable`, and `commit_async`.

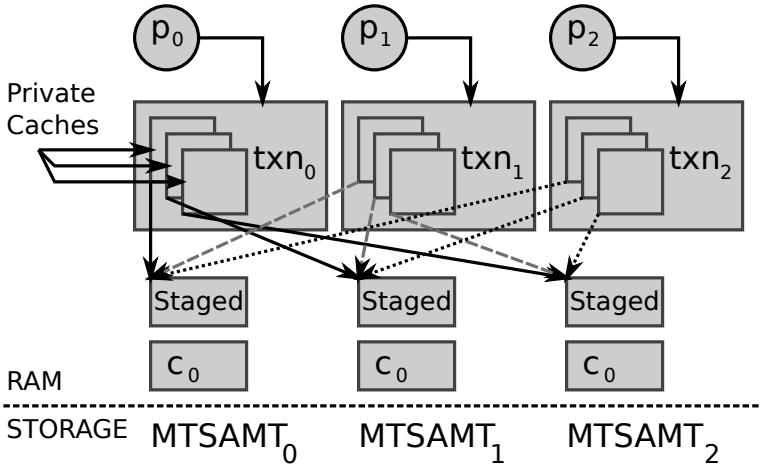

Figure 5.3: ***Private caches of a transaction:*** *Three processes, $p_0 ... p_2$, each maintain an ongoing transaction that has modified all 3 MTSAMTs so far.*

GTSSLv1's transaction manager (TM) manages all transactions for all threads. As shown in Figure 5.3, the TM maintains a stacked scan cache (Section 5.3.2) called the *staged cache* on top of

each tree's C_0 (also a scan cache). When an application begins a transaction with `begin`, the TM creates a handler for that transaction, and gives the application a reference to it. At any time, when a thread modifies a tree, a new scan cache is created if one does not already exist, and is stacked on top of that tree's staged cache. The new scan cache is placed in that transaction's handler. This new scan cache is called a *private cache*. In Figure 5.3 we see three handlers, each in use by three separate threads P_0 through P_2. Each thread has modified each of the three trees (MTSAMT_0 through MTSAMT_2).

Transactions managed by GTSSLv1's TM are in one of three states: (1) they are uncommitted and still exist only with the handler's private caches; (2) they are committed either durably or asynchronously and are in either the staged cache or C_0 of the trees they effect; or (3) they are entirely written to disk. Transactions begin in state (1), move to state (2) when committed by a thread, and when GTSSLv1 performs a snapshot of the system, they move to state (3) and are atomically written to storage as part of taking the snapshot.

Durable and asynchronous transactions can both be committed. We commit transactions durably by moving their transaction to state (2), and then *scheduling and waiting* for the system to perform a snapshot. While the system is writing a snapshot to storage, the staged cache is left unlocked so other threads can commit (similar to EXT3 [19]). A group commit of durable transactions occurs when multiple threads commit to the staged cache while the current snapshot is being written, and subsequently wait on the next snapshot together as a group before returning from `commit`. Asynchronous transactions can safely commit to the staged cache and return immediately from `commit`. After a snapshot the staged cache and the C_0 cache swap roles: the staged cache becomes the C_0 cache.

Next we discuss how we efficiently record snapshots in the journal, and how we eventually remove or garbage-collect snapshots by truncating the journal.

Snapshot, Truncate, and Recovery Unlike other BigTable based cluster TSSL architectures, GTSSLv1 manages blocks directly, not using separate files for each Wanna-B-tree. GTSSLv1 uses a large flat file that is zeroed out before use. A block allocator manages the flat file on each storage device. Every block allocator uses a bitmap to track which blocks are in use. The block size used is 128MB to prevent excessive fragmentation, but the OS page cache still uses 4KB pages for reads into the buffer cache.

Each tree (column family) maintains a cluster of offsets and meta-data information that points to the location of all Wanna-B-tree block offsets, secondary index block offsets, and Bloom filter block offsets. This cluster is called the *header*. When a snapshot is performed, all data referred to by all headers, including blocks containing Wanna-B-tree information, and the bitmaps, are flushed to storage using `msync`. Afterward, the append-only cache of the journal is flushed, recording all headers to the journal within a single atomic transaction. During recovery, the most recent set of headers are read back into RAM, and we recover the state of the system at the time that header was committed to the journal.

Traditional TSSLs implement a limited transaction feature-set that only allows for atomic insertion. Chang et al. [20] outline a basic architecture that implements this. Their architecture always appends insertions to the journal durably before adding them to the memtable. Cassandra and HBase implement this transactional architecture as well. By contrast, Pavlo et al. [102] and Abouzeid et al. [2] make the case for distributed transactions in database clusters. GTSSLv1's architecture does not exclude distributed transactions, and is as fast as traditional TSSLs like Cas-

sandra or HBase, or a factor of 2 faster when all three systems use asynchronous commits. One important feature of GTSSLv1 is that high-insertion throughput workloads that can tolerate partial durability (e.g., snapshotting every 3–5 seconds) need not write the majority of data into the journal. Although Cassandra and HBase support this feature for many of their use cases as well, they only delay writing to the journal, rather than avoid it. GTSSLv1 can avoid this write because if the C_0 cache evicts its memtable as an Wanna-B-tree between snapshots, the cache will be marked clean, and only the header need be serialized to the journal, avoiding double writing. This design improves GTSSLv1's performance over other TSSLs. The transaction must fit into RAM in GTSSLv1. Our implementation of GTSSLv2 overcomes this limitation as discussed in Section 6.2.

5.4 Evaluation

We evaluated GTSSLv1, Cassandra, and HBase along with some traditional DBMSes for various workloads. However, we focus here on the four most important properties relevant to this work: (1) the multi-tier capabilities of GTSSLv1, (2) the flexibility and efficiency of their compaction methods, (3) the efficiency of their serialization and caching designs for smaller key-value pairs, and (4) the transactional performance of GTSSLv1 and potentially other TSSLs with respect to traditional DBMSes for processing distributed transactions in a cluster. As laid out in Sections 5.2 and 5.3, we believe these are key areas where GTSSLv1 improves on the performance of existing TSSL architectures.

In Section 5.4.2, using micro-benchmarks we show that GTSSLv1's multi-tier capability provides better insertion throughput and caching behavior. We then evaluate the insertion and lookup throughput of these systems by configuring them in write-optimized and read-optimized modes in Section 5.4.3. We compare transaction throughput of GTSSLv1 with that of Berkeley DB and MySQL (using InnoDB) in Section 5.4.4. We then compare insert and lookup performance of GTSSLv1 against Cassandra and HBase using real-world deduplication index workload in Section 5.4.5.

5.4.1 Experimental Setup

Our evaluation ran on three identically configured machines running Linux CentOS 5.4. The experiments in each figure were run on the same machine; results shown in different figures were run on different, identically configured machines. The client machines each have a quad-core Xeon CPU running at 2.4GHz with 8MB of cache, and 24GB of RAM; the machines were booted with either 4.84GB, or 0.95GB of RAM to test out-of-RAM performance, and we noted with each test how much RAM was actually used. Each machine has two 146.1GB 15KRPM SAS disks (one used as system disk), a 159.4GB Intel X-25M Flash SSD (2^{nd} generation), and two 249.5GB 10KRPM SATA disks. Our tests used pre-allocated and zeroed out files for all configurations. We cleared all caches on each machine before running any benchmark. To minimize internal Flash SSD firmware interference due to physical media degradation and caching, we focus on long-running throughput benchmarks in this evaluation. Therefore, we reset all Flash SSD wear-leveling tables prior to evaluation (using the TRIM command), and we also confined all tests utilizing Flash SSD to a 90GB partition of the 159.4GB disk, or 58% of the disk. In tests involving HBase and Cassandra, we configured both systems to run directly on top of the file system. This is the default behavior for

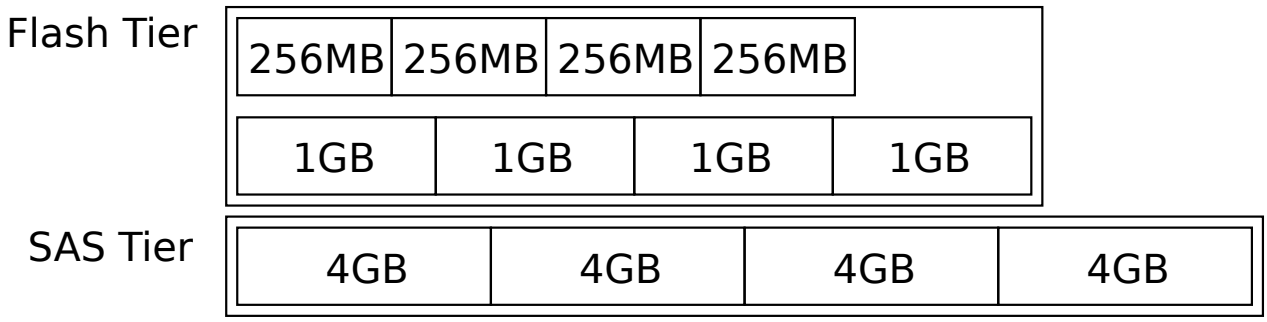

Figure 5.4: ***Configuration of re-insertion caching benchmark:*** *Storage is divided into two tiers, Flash and SAS (magnetic disk).*

Cassandra, but HBase had to be specially configured as it typically runs on top of HDFS. We gave both systems 3GB of JVM heap, and we used the remaining 1.84GB as a file cache. We configured GTSSLv1 to use upwards of 3GB for non-file cache information, including secondary indexes and Bloom filters for each slot in each tree, and the tuple cache (C_0) for each tree. This memory is allocated in GTSSLv1's heap. GTSSLv1 often used much less than 3GB, depending on the size of the pairs, but never more. We disabled compression for all systems because measurements of its effectiveness and for which data-sets are orthogonal to efficient TSSL operation. To prevent swapping heap contents when the file cache was under memory pressure due to MMAP faults, we set the SWAPPINESS parameter to zero for all systems and monitored swap-ins and swap-outs to ensure no swapping took place. All tests, except the multi-tier storage tests in Section 5.4.2 were run on the Intel X25-M Flash SSD described above.

5.4.2 Multi-Tier Storage

We modified the SAMT compaction method so that multiple tiers in a multi-tier storage hierarchy would be naturally used for faster insertion throughputs and better caching behaviors. Here we explore the effectiveness of caching a working set that is too large to fit in RAM, but small enough to fit in our Flash SSD.

Configuration We used only the Flash SSD in this test. In this test we also use the SAS disk. We configured GTSSLv1 with the first tier as RAM, the second on Flash SSD, and the third on the SAS disk. As shown in Figure 5.4, the Flash SSD tier holds two levels, each with a maximum of 4 Wanna-B-trees (slots): the maximum Wanna-B-tree size on the first level is 256MB, and on the second level it is 1GB. The SAS tier holds one level, with a maximum of 4 Wanna-B-trees, each no larger than 4GB. We booted the machine with 900MB of RAM. The size of available cache is 256MB, the size of the page cache is approximately 500MB after deducting expected space for secondary indexes and Bloom filters. The size of each pair was 4KB including the key and value. Since re-insertion caching was flooding the page cache with writes, we found little of it was available for read caching. We randomly inserted 16GB of random 4KB pairs and then performed random queries on a randomly selected subset of these pairs. The subset was 1GB large and is called the hot-set.

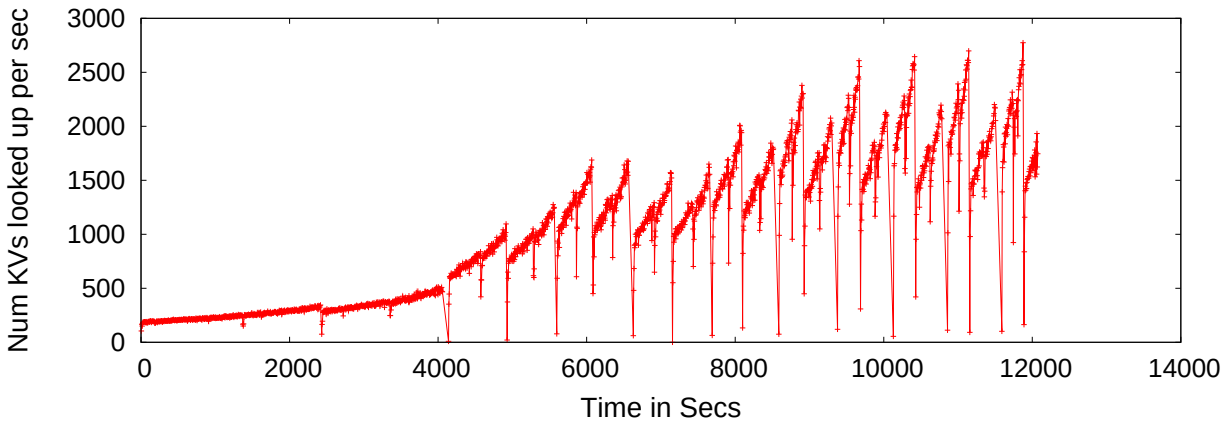

Figure 5.5: ***Multi-tier results:*** *Initially throughput is disk-bound, but as the hot-set is populated, it becomes Flash SSD-bound, and is periodically evicted and reclaimed.*

Results As shown in Figure 5.5, initial lookup throughput was 243.8 lookups/s, which corresponds to the random read throughput of the disk, 251 reads/s. Pairs are read into the scan cache (C_0), and once 256MB have been read, as described in Section 5.3.1, data in C_0 is flushed into the Flash SSD to facilitate multi-tier read caching. This corresponds to the 20 sudden drops in lookup throughput. Once the entire 1GB of hot-set has been evicted into the Flash SSD tier, subsequent reads, even from the Flash SSD, are re-inserted. These reads cause the contents of the Flash SSD to flush into the SAS tier. Since at most 1GB of the 4GB being merged into the SAS tier is unique, we will omit at least 3GB of tuples while writing into the SAS tier. Since the resulting output from the merge is small enough to fit into a single slot at the lowest level of the Flash SSD tier, it is reclaimed back into the Flash SSD tier via a copy.

The mean lookup throughput is 1,815.93 lookups/s, a 7.4× speedup over the disk read throughput, and 48% of the Flash SSD random read throughput. The sudden drops in lookup throughput are due to evictions, now being caused by reads which actually result in writes. Latency spikes are a common problem with compaction based TSSLs. In Figure 5.5 we see low points in the instantaneous lookup throughput. These points are an artifact of our benchmarking methedology. When we measure the instantaneous throughput of the workload, we begin timing and then repeatedly check to see if five seconds have elapsed after every insert. Since some insertions can cause long pauses (when minor compaction is performed), samples of instantaneous throughput immediately before a minor compaction will show as low insertion throughput.

For hotsets that will be queried over a long period of time, read-caching for random reads from lower storage tiers can be beneficial, as we have shown above. Additionally, caching of recently inserted values in higher tiers is an automatic benefit of the MTSAMT. We re-ran the above experiment with a data-set of 16GB of randomly inserted 1KB keys with read caching disabled. After all values were inserted, we searched for each pair from most recently inserted to least. Figure 5.6 shows our results. Insertion of the pairs was 33% faster for the RAM-SSD-SAS configuration as the more frequent merging compactions of the higher tiers took place on a Flash SSD device, and merges across tiers did not have to read and write to the same device at once. The pairs were inserted randomly, but the Wanna-B-trees on storage are sorted, so we see a series of random reads within each Wanna-B-tree. After insertions, the scan cache of 256MB was full; there were three

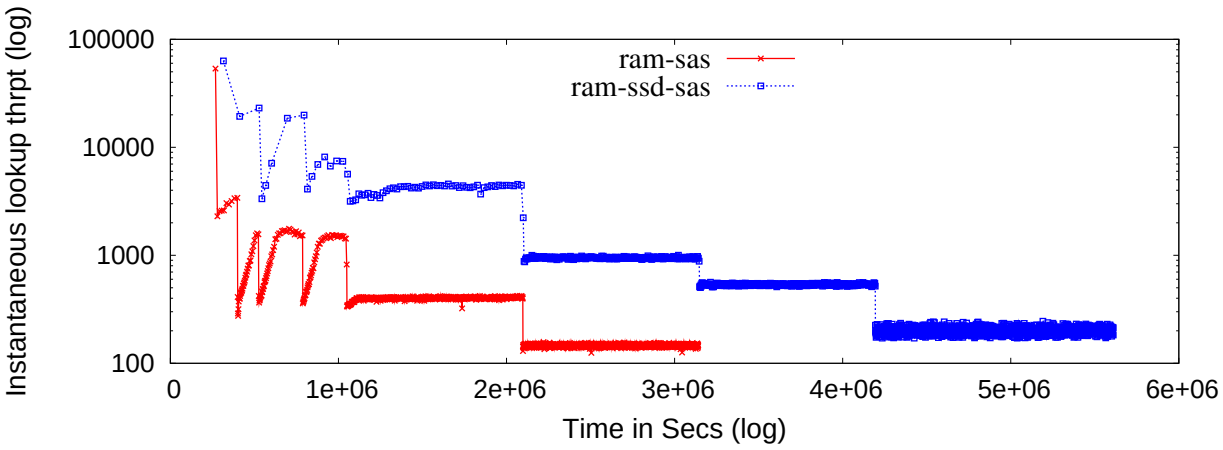

Figure 5.6: ***Multi-tier insertion caching:*** *Caching allows for lookups of recently inserted data to happen rapidly, while still allowing for very large data-sets that can be cheaply stored mainly on disk.*

256MB Wanna-B-trees, and three 1GB Wanna-B-trees in the first tier, and three 4GB Wanna-B-trees in the second tier. For RAM-SSD-SAS only the three 4GB Wanna-B-trees were on SAS, for RAM-SAS they all were. Although each Wanna-B-tree is guarded by an in-RAM bloom filter, false positives can cause lookups to check these tables regardless. Furthermore the ratio of buffer cache to Wanna-B-tree size shrinks exponentially as the test performs lookups on lower levels. This causes the stair-step pattern seen in Figure 5.6. Initial spikes in lookup throughput occur as the buffer cache is extremely effective for the 256MB Wanna-B-trees, but mixing cache hits with the faster cache-populating Flash SSD (14,118 lookups/s) provides higher lookup throughput than with the SAS (1,385 lookups/s). Total lookup throughput of the first 3,000,000 pairs, or the first 27% of the data-set was 2,338 lookups/s for RAM-SSD-SAS, and 316 lookups/s for RAM-SAS, a 7.4× performance improvement.

5.4.3 Read-Write Trade-off

We evaluated the performance of Cassandra, HBase, and GTSSLv1 when inserting 1KB pairs into four trees, to exercise multi-tree transactions. 1KB is the pair size used by YCSB [24]. For each system we varied its configuration to favor either reads or writes. HBase supported only one optimal configuration, so it was not varied. Cassandra and GTSSLv1 can trade off lookup for insertion performance by increasing K (see Section 5.2). Our configuration named BALANCED sets $K = 4$, the default; configuration MEDIUM sets $K = 8$; configuration FAST sets $K = 80$. We measured insertion throughput and lookup separately to minimize interference, but both tests utilized ten writers or readers.

Configuration In addition to the configuration parameters listed in Section 5.4.1, to utilize four trees in Cassandra and HBase, we configured four column families. We computed the on-disk footprint of one of Cassandra's pairs based on its SIZE routine in its TSSL sources (which we analyzed manually), and we reduced the size of the 1KB key accordingly so that each on-disk tuple would actually be 1KB large. We did this to eliminate any overhead from tracking column

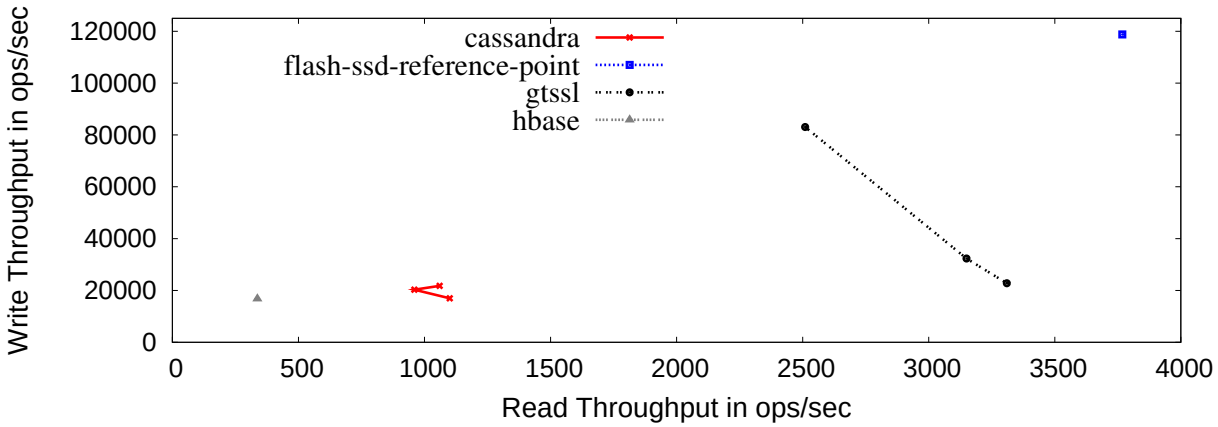

Figure 5.7: ***Read-write Optimization Efficiency:*** *Cassandra has comparable insertion performance to GTSSLv1 when both systems retain as much lookup throughput as possible. GTSSLv1 reaches much further into the trade-off space. HBase is already optimally configured, and cannot further specialize for insertions.*

membership in each pair. We did the same for HBase, and used tuples with no column membership fields for GTSSLv1, while also accounting for the four byte size field used for variable length values. This minimized differences in performance across implementations due to different feature sets that require more or less meta-data to be stored with the tuple on disk. Overall, we aimed to configure all systems as uniformly as possible, to isolate only the TSSL layer, and to configure Cassandra and HBase in the best possible light.

Results Figure 5.7 is a parametric function, where each point represents a run, and the parameter varied is the system configuration. The x-axis measures that configuration's insertion (write) throughput, and the y-axis measures its random lookup (read) throughput. The maximum lookup throughput of each structure cannot exceed the random read performance of the drive; similarly, the maximum insertion throughput cannot exceed the serial write throughput of the drive. These two numbers are shown as one point at FLASH SSD REFERENCE POINT. Systems are better overall the closer they get to this upper-right-hand-side corner reference point.

Both HBase and Cassandra utilize Bloom filtering, but Bloom filtering is a new feature for HBase that was recently added. HBase caches these Bloom filters in an LRU cache. So although HBase can swap in different Bloom filters, for uniform or Zipfian lookup distributions, HBase has to page in Bloom filter data pages to perform lookups, causing a 10× slowdown compared to Cassandra and GTSSLv1. However, if we perform a major compaction (which can take upwards of an hour) we notice that with 4KB blocks, HBase lookups can be as high as 910 lookups/s; for the same block size before major compaction, however, lookup throughput is only 200 lookups/s, lower than with the default 64KB blocksize. Performing major compactions with high frequency is not possible as it starves clients.

For the BALANCED configuration, Cassandra and GTSSLv1 have similar insertion throughputs of 16,970 ops/s and 22,780 ops/s, respectively. However, GTSSLv1 has a 3× higher lookup throughput than Cassandra, and a 10× higher than HBase. GTSSLv1 utilizes aggressive Bloom filtering to reduce the number of lookups to effectively zero for any slot that does not contain the

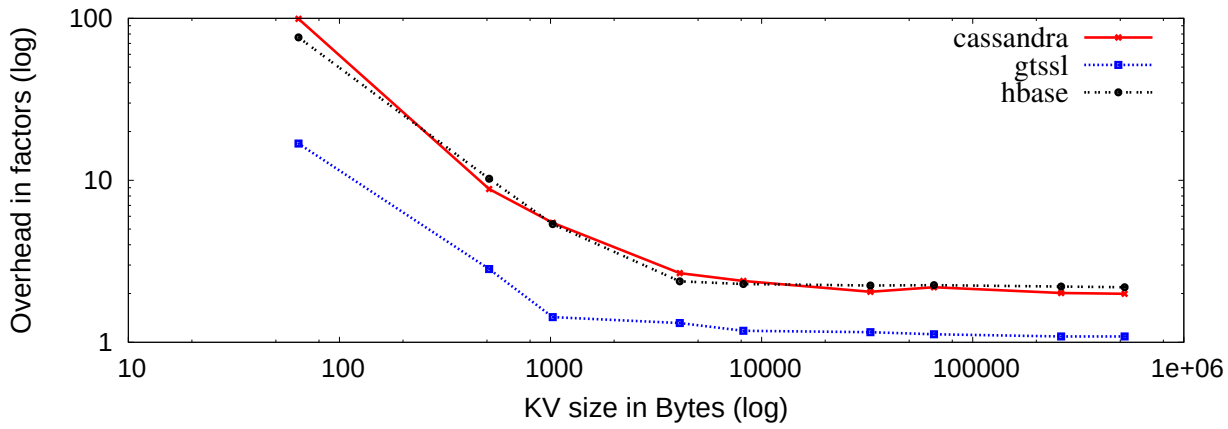

Figure 5.8: ***Single-item Transaction Throughput:*** *Neither Cassandra nor HBase improve beyond an overhead of 2.0× for large pairs, or 4x for small pairs when compared to GTSSLv1.*

sought-after key. The random read throughput of the Flash SSD drive tested here is 3,768 reads/s, closely matching the performance of GTSSLv1. Cassandra uses 256KB blocks instead of 4KB blocks, but uses the meta-data to read in only the page within the 256KB block containing the key. We observed that block read rates were at the maximum bandwidth of the disk, but Cassandra requires three I/Os per lookup [31] when memory is limited, resulting in a lookup throughput that is only $1/3$ the random read throughput of the Flash SSD.

For the more write-optimized configurations, GTSSLv1 increased its available bandwidth for insertions considerably: for MIDDLE, GTSSLv1 achieved 32,240 ops/s and 3,150 ops/s, whereas Cassandra reached only 20,306 ops/s and 960 ops/s, respectively. We expected a considerable increase in insertion throughput and sustained lookup performance for both Cassandra and GTSSLv1 as they both use variants of the SAMT. However, Cassandra's performance could not be improved beyond 21,780 ops/s for the FAST configuration, whereas GTSSLv1 achieved 83,050 ops/s. GTSSLv1's insertion throughput was higher thanks to its more efficient serialization of memtables to Wanna-B-trees on storage. To focus on the exact cause of these performance differences, we configured all three systems (HBase, Cassandra, and GTSSLv1) to perform insertions but no compaction of any sort. We explore those results next.

Cassandra and HBase limiting factors To identify the key performance bottlenecks for a TSSL, we ran an insertion throughput test, where each system was configured to insert sizes of pairs varying from 64B to 512KB as rapidly as possible, using ten parallel threads. Cassandra, HBase, and GTSSLv1 were all configured to commit asynchronously, but still maintain atomicity and consistency (the FAST configuration). Furthermore, Cassandra's compaction thresholds were both set to 80 (larger than the number of Wanna-B-trees created by the test); HBase's compaction (and compaction time-outs) were simply disabled, leaving both systems to insert freely with *no compactions* during this test. The ideal throughput for this workload is the serial append bandwidth of the Flash SSD (110MB/s), divided by the size of the pair used in that run. Figure 5.8 shows these results. Each point represents an entire run of a system. The y-axis represents how many times slower a system is compared to the ideal, and the x-axis represents the size of the pair used for that run. All three systems have the same curve shape: a steep CPU-bound portion ranging from 64B to 1KB,

and a shallower I/O-bound portion from 1KB to 512KB.

For the I/O-bound portion, HBase and Cassandra both perform at best 2.0× worse than the ideal, whereas GTSSLv1 performs 1.1× worse than the ideal, so GTSSLv1 is 2× faster than Cassandra and HBase in the I/O-bound portion. Cassandra and HBase both log writes into their log on commit, even if the commit is asynchronous, whereas GTSSLv1 behaves more like a file-system and avoids writing into the log if the memtable can be populated and flushed to disk before the next flush to the journal. This allows GTSSLv1 to avoid the double-write to disk that Cassandra and HBase perform, a significant savings for I/O-bound insertion-heavy workloads that can tolerate a 5-second asynchronous commit.

For the CPU-bound portion, we see that GTSSLv1 is a constant factor of 4× faster than both HBase and Cassandra, and additionally that HBase and Cassandra have very similar performance: the ratio of Cassandra's overhead to HBase's is always within a factor of 0.86 and 1.3 for all runs. When running Cassandra and its journal entirely in RAM, their insertion throughput of the 64B pair improved by only 50%, dropping from 99.2× to 66.1×, which is still 4× slower than GTSSLv1 which was *not* running in RAM. The meager change in performance for running entirely in RAM further confirms that these workloads were CPU-bound for smaller pairs (< 1KB).

We wanted to understand the performance difference between GTSSL and Cassandra in the memory-bound range, because nothing in GTSSL's architecture could easily explain it. This led us to an extensive analysis of the various in-memory data structures used by Cassandra and GTSSL, which we show in Appendix B. We discovered that Cassandra's Java-based skip-list implementation is approximately three times slower than our C++ red-black tree implementation. So, the factor of three difference we see in performance in the memory bound regime is largely due to this implementation artifact.

5.4.4 Cross-Tree Transactions

We designed GTSSLv1 to efficiently process small transactions, as well as high-throughput insertion workloads. We evaluated GTSSLv1's transaction throughput when processing many small and large transactions. We ran two tests: (1) TXN-SIZE and (2) GROUP-COMMIT. In TXN-SIZE, the number of executing threads is fixed at one, and each commit is synchronous. Each run of the benchmark performs a transaction that inserts four pairs, each into a separate tree. Each run uses a different size for the four pairs, which is either 32B, 256B, or 4096B. In GROUP-COMMIT, we performed parallel transactions across 512 threads. Each transaction inserted a different 1KB tuple into four separate trees, so each transaction inserted 4KB at a time. We wanted to test performance of parallel transactions on our 4-core machine.

Configuration We configured three systems for comparison in this test: GTSSLv1, MySQL (using InnoDB), and Berkeley DB (BDB). We configured each system identically to have 1GB of cache. We did not include HBase or Cassandra in these results as they do not implement asynchronous transactions. We configured BDB as favorably as possible through a process of reconfiguration and testing: 1GB of cache and 32MB of log buffer. We verified that BDB never swapped or thrashed during these tests. We configured BDB with a leaf-node size of 4096B. We configured InnoDB favorably with a 1GB of cache and 32MB of log buffer. We configured GTSSLv1 with 1GB of cache (four 256MB caches). All systems ran on Flash SSD.

	32B	256B	4KB
GTSSLv1	8,203	3,140	804
Berkeley DB	683	294	129
MySQL	732	375	

Table 5.1: Performance of databases performing asynchronous transactions.

Results GTSSLv1 outperformed MySQL and BDB on the whole by a factor of about 6–8×. We inserted 1,220MB of transactions (9,994,240 transactions of four 32-byte insertions). For 32-byte insertions, overall insertion performance for BDB, MySQL, and GTSSLv1 is 683, 732, and 8,203 commits/s, respectively. For 256 byte insertions it is 294, 375, and 3,140 commits/s, respectively. At 4KB insertions, MySQL does not permit 4K columns, and so we omit this result. However, GTSSLv1 and BDB each have throughputs of 804 and 129, respectively. GTSSLv1 is 6.23× faster than BDB. MySQL, BDB, and GTSSLv1 each attained *initial* insertion throughputs of 2,281, 5,474, and 9,908 transactions/s, respectively, when just updating their own journals. MySQL and BDB begin converging on their B-Tree insertion throughput as they write-back their updates. GTSSLv1, on the other hand, avoids random writes entirely, and pays only merging overheads periodically due to merging compactions, so final insertion throughputs are quite different.

We were surprised by the high durable commit throughput of GTSSLv1, and expected performance to be approximately no more than the random write throughput of the device, or 5,000 writes/sec. So, we ran a micro-benchmark on Ext3. We simulated performing synchronous 32B *sequential* transactions by appending 32B, and then syncing the file, and repeating. If transactions are submitted serially, the current transaction must wait for the disk to sync its write before the next transaction can proceed. For comparison, we ran the same test on a magnetic disk and found that its write throughput was 300 commits/sec; on the Flash SSD, however, the same test scored a surprisingly high 15,000 commits/sec, even higher than the random write throughput of the Flash SSD. We concluded that due to differences between the memory technologies employed on both devices, that the Flash SSD is capable of much higher synchronous append throughput when compared to a magnetic disk.

GTSSLv1 is able to keep the total amount written per-commit small—as it must only flush the dirty pairs in its C_0 cache plus book-keeping data for the flush (111 bytes). This additional amount written per transaction gives direct synchronous append an advantage of 66% over GTSSLv1; however, as GTSSLv1 logs only redo information, its records require no reads to be performed to log undo information, and its records are smaller. This means that as BDB and MySQL must routinely perform random I/Os as they interact with a larger-than-RAM B+-tree, GTSSLv1 need only perform mostly serial I/Os, which is why GTSSLv1 and other TSSL architectures are better suited for high insertion-throughput workloads when there is sufficient drive or storage space for a log-structured solution, and transactions are no larger than RAM.

In the GROUP-COMMIT test, we tested GTSSLv1 for synchronous transaction commit throughput. When testing peak Flash SSD-bandwidth GROUP-COMMIT throughput, we found that GTSSLv1 could perform 26,368 commits/s for transactions, updating four trees with 1KB values, at a bandwidth of 103MB/s, which is near the optimal bandwidth of the Flash SSD: 110MB/sec.

Figure 5.9: ***Deduplication insertion and lookup performance:*** *Displayed are the throughput of the Cassandra and HBase TSSLs, and GTSSLv1. Both y-axes are in ops/sec.*

5.4.5 Deduplication

The previous tests were all micro-benchmarks. To evaluate the performance of Cassandra, HBase, and GTSSLv1 when processing a real-world workload, we built a deduplication index. We checksummed every 4KB block of every file in a research lab network of 82 users, with a total of 1.6TB. This generated over 1 billion hashes. We measured the time taken to insert these hashes with ten parallel insertion threads for all systems. We then measured the time to perform random lookups on these hashes.

Configuration We generated the deduplication hashes by chunking all files in our corpus into 4KB chunks, which were hashed with SHA256. We appended these 32B hashes to a file in the order they were chunked (depth-first traversal of the corpus file systems). To test lookups, we shuffled the hashes in advance into a separate lookup list. During insertion and lookup, we traversed the hashes serially, minimizing the overhead due to the benchmark itself, during evaluation of each system.

Results As seen in Figure 5.9, we found that performance is analogous to the 64B case in Section 5.4.3, which used randomly generated 64B numbers instead of a stream of 32B hashes. Cassandra, HBase, and GTSSLv1 were able to perform 22,920 ops/s, 23,140 ops/s, and 109,511 ops/s, respectively. For lookup performance they scored 967 ops/s, 398 ops/s, and 2,673 ops/s, respectively. As we have seen earlier, the performance gap between Cassandra and HBase compared to GTSSLv1 are due to CPU and I/O inefficiency, as the workload is comparable to a small-pair workload, as discussed above. Real-world workloads can often have pairs of 1KB or smaller in size, such as this deduplication workload. An efficient TSSL can provide up to 5$\times$ performance improvement without any changes to other layers in the cluster architecture. In our experiments we found that improvements for CPU-bound workloads were due mostly to basing GTSSLv1 on C++ instead of Java.

Evaluation summary TSSL architectures have traditionally optimized for IO-bound workloads for pairs 1KB or larger on traditional magnetic disks. For 1KB pairs, GTSSLv1 has a demonstrably

more flexible compaction method. For the read-optimized configuration, GTSSLv1 lookups are near optimal: 88% the maximum random-read throughput of the Flash SSD, yet our insertions are still 34% faster than Cassandra and 14% faster than HBase. For the write-optimized configuration, GTSSLv1 achieves 76% of the maximum write throughput of the Flash SSD, yet our lookups are 2.3× and 7.2× faster than Cassandra and HBase, respectively. This performance difference was due to Cassandra and HBase being CPU-bound for pairs 1KB or smaller. When we varied the pair size, we discovered that for smaller pairs, even when performing no compaction and no operations other than flushing pairs to storage, all TSSLs became CPU-bound, but GTSSLv1 was still 5× faster than the others.

For larger pairs, all TSSLs eventually became I/O-bound. GTSSLv1 achieved 91% of the maximum serial write throughput of the Flash SSD. Cassandra and HBase achieved only 50% of the maximum Flash SSD throughput, due to double-writing insertions even when transactions were asynchronous. Cassandra's and HBase's designs were geared for ease of development and were written in Java; but as modern Flash SSD's get faster, the tuple size at which workloads become CPU-bound increases. We observed CPU-bound effects for 1KB tuples in our experiments. Developers of LSM-tree-based databases will have to carefully consider the underlying runtime environment their databases run on top of and can no longer assume that most workloads will be strictly I/O-bound.

GTSSLv1's design explicitly incorporates Flash SSD into a multi-tier hierarchy. When we insert a Flash SSD into a traditional RAM+HDD storage stack, GTSSLv1's insertion throughput increased by 33%, and our lookup throughput increased by 7.4×. This allows the bulk of colder data to reside on inexpensive media, while most hot data automatically benefits from faster devices.

Supporting distributed transactions in clusters does not require a different TSSL layer. GTSSLv1's transactions are light-weight yet versatile, and achieve 10.7× and 8.3× faster insertion throughputs than BDB and MySQL InnoDB, respectively.

5.5 Related Work

We discuss cluster evaluation (1), multi-tier and hierarchical systems (2–4), followed by alternative data-structures for managing trees or column families in a TSSL architecture (5–7).

(1) Cluster Evaluation Super computing researchers recognize the need to alter out-of-the-box cluster systems, but there is little research on the performance of individual layers in these cluster systems, and how they interact with the underlying hardware. Pavlo et al. measured the performance of a Hadoop HBase system against widely used parallel DBMSes [102]. Cooper et al. compared Hadoop HBase to Cassandra [66] and a cluster of MySQL servers (similar to HadoopDB and Perlman and Burns' Turbulence Database Cluster). The authors of HadoopDB include a similar whole-system evaluation in their paper [2]. We evaluate the performance bottlenecks of a single node's storage interaction, and provide a prototype architecture that alleviates those bottlenecks.

(2) Multi-tier storage Flash SSD is becoming popular [48]. Solaris ZFS can use intermediate SSDs to improve performance [70]. ZFS uses an Flash SSD as a DBMS log to speed transaction performance, or as a cache to decrease read latency. But this provides only temporary relief: when

the DBMS ultimately writes to its on-disk tree, it bottlenecks on B-tree throughput. ZFS has no explicit support for very large indexes or trees, nor does it utilize its three-tier architecture to improve indexing performance. GTSSLv1, conversely, uses a compaction method whose performance is bound by disk bandwidth, and can sustain high-throughput insertions across Flash SSD flushes to lower tiers with lower latencies. Others used Flash SSD's to replace swap devices. FlashVM uses an in-RAM log-structured index for large pages [118]. FASS implements this in Linux [58]. Payer [103] describes using a Flash SSD hybrid disk. Conquest [149] uses persistent RAM to hold all small file system structures; larger ones go to disk. These systems use key-value pairs with small keys that can fit entirely in RAM. GTSSLv1 is more general and can store large amounts of highly granular structured data on any combination of RAM and storage devices.

(3) Hierarchical Storage Management HSM systems provide disk backup and save disk space by moving old files to slower disks or tapes. Migrated files are accessible via search software or by replacing migrated files with links to their new location [53, 101]. HSMs use multilevel storage hierarchies to reduce overall costs, but pay a large performance penalty to retrieve migrated files. GTSSLv1, however, was designed for always-online access as it must operate as a TSSL within a cluster, and focuses on maximum performance across all storage tiers.

(4) Multi-level caching Multi-level caching systems address out-of-sync multiple RAM caches that are often of the same speed and are all volatile: L2 vs. RAM [29], database cache vs. file system page cache [40], or located on different networked machines [73, 129]. These are not easily applicable to general-purpose multi-tier structure data storage due to large performance disparities among the storage devices at the top and bottom of the hierarchy.

(5) Write-optimized trees The *COLA* maintains $\mathbf{O}(\log(N))$ cache lines for N key-value pairs. The amortized asymptotic cost of insertion, deletion, or updates into a COLA is $\mathbf{O}(\log(N)/B)$ for N inserted elements [11]. With fractional cascading, queries require $\mathbf{O}(\log(N))$ random reads [21]. Fractional cascading is a technique that can be used to accelerate overall query performance when a query comprises multiple queries to component data structures and these component queries are performed in a predictable order. See Appendix A for a more detailed description of fractional cascading. GTSSLv1's SAMT has identical asymptotic insertion, deletion, and update performance; however, lookup with SAMT is $\mathbf{O}\left(\log^2(N)\right)$. In practice GTSSLv1's secondary indexes easily fit in RAM though, and so lookup is actually equivalent for trees several TBs large. Furthermore, as we show in our evaluation, GTSSLv1's Bloom filters permit 10–30$\times$ faster lookups for datasets on Flash SSD than what the COLA (used by HBase) can afford. *Log-Structured Merge* (LSM) trees [95] use an in-RAM cache and two on-disk B-Trees that are R and R^2 times larger than cache, where $\frac{1}{R} + R + R^2$ is the size of the tree. LSM tree insertions are asymptotically faster than B-Trees: $\mathbf{O}\left(\frac{\sqrt{N}\log N}{B}\right)$ [121] compared to $\mathbf{O}\left(\log_{B+1} N\right)$, but asymptotically slower than GTSSLv1's SAMT. LSM tree query times are more comparable to B-Tree's times. Rose is a variant of an LSM tree that compresses columns to improve disk write throughput [121].

Jagadish et al. describe a variant of the LSM-tree that uses an approach similar to the SAMT [57]. In their scheme, K trees are held at N levels, like in the SAMT, but these trees are merged into a single large B tree after N levels. The SAMT is never required to merge into a single B tree.

According to the authors, the analysis provided does not obtain a meaningful comparison to traditional LSM-trees. In this chapter, Section 5.2, we compare the DAM to the HBase/COLA style of traditional LSM-tree merging, and the Cassandra/SAMT style of LSM-tree merging. Jagadish et al. do not implement concurrency or recovery mechanisms for their data structure. Both GTSSLv1 and GTSSLv2 implement these features and explore in detail many complexities in supporting transactions, efficient delete. In Chapter 6 we detail our support for efficient sequential insertion.

Anvil [75] is a library of storage components for assembling custom 2-tier systems and focuses on development time and modularity. Anvil describes a 2-COLA based structure and compares performance with traditional DBMSes in TPC-C performance. GTSSLv1's uses the multi-tier MTSAMT structure, and is designed for high-throughput insertion and lookups as a component of a cluster node. We evaluate against existing industry standard write-optimized systems and not random-write-bound MySQL InnoDB. Data Domain's deduplicating SegStore uses Bloom filters [15] to avoid lookups to its on-disk hash table, boosting throughput to 12,000 inserts/s. GTSSLv1 solves a different problem: the base insertion throughput to an on-disk structured data store (e.g., Data Domain's Segment Index, for which insertion is a bottleneck). GTSSLv1 is complimentary to, and could significantly improve the performance of similar deduplication technology.

(6) Log-structured data storage Log-structured file systems [117] append dirtied blocks to a growing log that must be compacted when full. Goetz's log-structured B-trees [42] and FlashDB [92] operate similarly to WAFL [51] by rippling up changes to leaf pointers. Goetz uses fence-keys to avoid expensive rippling, and uses tree-walks during scans to eliminate leaf pointers. FAWN [4] is a distributed 2-tier key-value store designed for energy savings. It uses a secondary index in RAM and hash tables in Flash SSD. FAWN claims that compression (orthogonal to this work) allows large indexes to fit into 2GB of RAM. By contrast, GTSSLv1 has been tested with 1–2TB size indexes on a single node. Log-structured systems assume that the entire index fits in RAM, and must read in out-of-RAM portions before updating them. This assumption breaks down for smaller (64B) pairs where the size of the index is fairly large; then, compaction methods employed by modern TSSLs become vital.

(7) Flash SSD-optimized trees Flash SSD has high throughput writes and low latency reads, ideal for write-optimized structured data storage. FD-Trees's authors admit similarity to LSM trees [72]. Their writes are worse than an LSM-tree for 8GB workloads; their read performance, however, matches a B-tree. GTSSLv1's insertions are asymptotically faster than LSM trees. LA-Tree [3] is another Flash SSD-optimized tree, similar to a Buffer Tree [10]. LA-Trees and FlashDB can adaptively reorganize the tree to improve read performance. Buffer Trees have asymptotic bound equal to COLA. However, it is not clear or discussed how to efficiently extend Buffer Trees, LA-Trees, or FD-Trees for multiple storage tiers or transactions as GTSSLv1 does.

5.6 Conclusions

We introduced GTSSLv1, an efficient tablet server storage architecture that is capable of exploiting Flash SSD and other storage devices using a novel multi-tier compaction algorithm. Our multi-tier extensions have 33% faster insertions and a 7.4× faster lookup throughput than traditional RAM+HDD tiers—while storing 75% of the data (i.e., colder data) on cheaper magnetic disks. For

IO-bound asynchronous transactional workloads, GTSSLv1 achieves 2× faster throughput than Cassandra and HBase by avoiding multiple writes to a journal and the Wanna-B-tree. GTSSLv1 is able to dynamically switch between logging redo records to a journal for smaller concurrent ARIES workloads and writing directly to the Wanna-B-tree for larger asynchronous transaction workloads.

We also found that CPU-boundedness is an important factor for small tuple insertion efficiency. We were surprised to find that current implementations of Cassandra and HBase were 2.3× and 7.2× slower for lookups than GTSSLv1 respectively, and 3.8× slower for insertions in CPU-bound workloads (e.g., 64B tuples). This was due primarily to differences in the choice of underlying implementation, GTSSLv1 used C++ and Cassandra and HBase use Java. However, this implementation decision had far reaching consequences both on performance of smaller tuples, and even for read-write flexibility. Cassandra and HBase were unable to effectively trade-off read throughput for write throughput, even though their design and data structure allow for this, as their overall throughput was already bottlenecking on runtime overheads for 1KB tuples. Future implementations should take heed and avoid CPU overheads as well as IO overheads when profiling and optimizing.

We have shown how the existing TSSL layer can be extended to support more versatile transactions capable of performing multiple reads and writes in full isolation without compromising performance. GTSSLv1 achieved 10.7× and 8.3× faster insertion throughputs than BDB and MySQL's InnoDB, respectively.

Newer storage technologies such as Flash SSD do not penalize random writes of 1KB or less. Our theoretical analysis of existing TSSL compaction methods indicates that although Cassandra is better suited for such workloads than HBase, true multi-tier support is required to leverage modern Flash SSD's as part of the TSSL storage hierarchy.

Our performance evaluation of existing TSSL architectures show that, faced with increasingly faster random I/O from Flash SSD's, CPU and memory efficiency are paramount for increasingly more complex and granular data. Integrating modern storage devices into a TSSL requires a more general approach to storage than currently available, and one that operates in a generic fashion across multiple tiers. GTSSLv1 offers just that.

Despite GTSSLv1's effective performance for random workloads in comparison to Cassandra and HBase, when performing sequential insertions it still operates at $1/5$ of the disk's write throughput. In comparison to Ext3, for sequential insertions, GTSSLv1 would perform 5× slower. If we intend to use LSM-trees as an underlying data structure for a file system design, we will have to improve performance considerably.

Table 5.2 now lists Cassandra and GTSSLv1 in comparison to other storage systems in terms of the transactional design decisions table, Table 3.1 first introduced in Section 3.4. Cassandra and GTSSLv1 are comparable in purpose: they both target key-value storage workloads, and for concurrent small transaction workloads they both must write twice. GTSSLv1 provides value-logging for tuple updates and general ACID transaction support for transactions that fit within RAM. Both systems are log-structured and provide some support for asynchronous transactions, though Cassandra still writes twice for asynchronous transactions and cannot blend synchronous and asynchronous transactions together dynamically. Both systems perform like LSM-tree–based systems in that predecessor queries are less efficient than for a strict B-tree–based system, but point queries, scans, and all random writes are efficient. However, both systems are inefficient for sequential write workloads.

If we can maintain the efficiency of GTSSL while extending it with several key optimizations—

	Type	Num Writes	Log-Struct.	Trans-actions	Conc-urrent	Async	Write Order	Random	Stitch-ing	Sequen-tial
Ext3	FS	1	¬	MD-only	¬	✓	Kernel	R	¬	R,W
SchemaFS*	FS	3	¬	Logical	✓	✓	User	R	¬	R
Valor	FS	2	¬	POSIX	¬	✓	Kernel	R	¬	R
LSMFS	FS	1	✓	MD-only	¬	✓	`mmap`	S,W	¬	R,W
Cassandra	KVS	2	✓	Single	✓	✓	`mmap`	P,S,W	¬	R
GTSSLv1	KVS	1–2	✓	Vals	✓	✓	`mmap`	P,S,W	¬	R

Table 5.2: ***A qualitative comparison of Transactional Storage Designs:*** *We can conclude that although we have great flexibility in terms of transactional performance and random access workloads, we still execute sequential file system workloads inefficiently.*

namely support for sequential insertions that do not re-copy on merge—then we believe we would have the basis for an efficient transactional file system. Next, in Chapter 6, we discuss our extensions to the LSM-tree to support efficient sequential insertion.

Chapter 6

GTSSLv2: An Extensible Architecture for Data and Meta-Data Intensive System Transactions

We pointed out in the conclusion of Chapter 5, Section 5.6, that LSM-trees are not as efficient when inserting sequential workloads as a typical file system or B-tree. Figure 6.1 illustrates the problem that LSM-trees face when performing sequential workloads. In Figure 6.1, panel ① we see a typical merge in a SAMT, where the number of slots per level is set to $K = 2$. If we consider the path of an element el as it is copied from one level into the next level below, as illustrated in panel ②, we see that over the course of N insertions, el is copied $\log_K N$ times.

This causes a problem when there is a subsequence of A els that is sufficiently large enough that copying it $\log_K N$ times takes more time than seeking to some reserved location and writing it once. If it is faster to insert A els by seeking and writing once, then inserting this sequence into an LSM tree will take longer than inserting it into a B-tree or existing traditional file system.

In the DAM model, it costs $\frac{A}{B_{small}}$ storage transfers to sequentially read or write A tuples or els. The cost of reading and writing A els $\log_K N$ times compared to seeking once $(+1)$ and writing A els sequentially once is:

$$2 * \frac{A \log_K N}{B_{small}} \geq 1 + \frac{A}{B_{small}}$$

We solve for A to determine the length of a sequential insertion at which the cost of randomly writing is as good or better than insertion into an LSM-tree:

$$A \geq \frac{B_{small}}{2 \log_K N - 1}$$

As a practical example, lets assume that we have 5 levels. The first level is 128MB large and $K = 4$. Then $\log_K N = 5$. If our disk transfer rate is 100MB/sec, and time to seek is 8ms, then we can read 819KB in the time it takes to perform a single seek. If we take a $2\times$ overhead for sequential reads and writes, then we can assume that in our DAM model, the block size in bytes

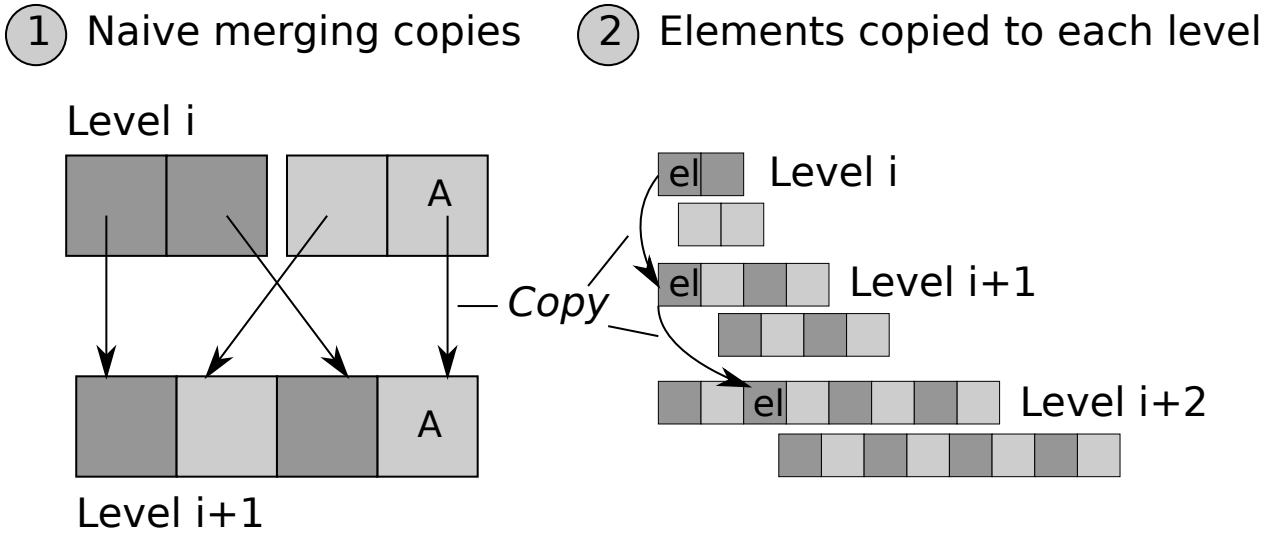

Figure 6.1: ***Elements Copy During Merge:*** *We perform "copies" of a sequence of els across multiple levels, rather than writing them just once, like a more traditional storage system would.*

$b = 819$KB. Then, solving for A we see that for 91KB sequences of els or larger, seeking and writing once beats insertion into an LSM. For this basic example, we have shown that a traditional random write will beat a SAMT for a sufficiently large A.

We extend the multi-tier sorted-array merge-tree (MT-SAMT) database discussed in Chapter 5 to efficiently store sequential objects by generalizing LSM-trees to support *stitching*. The stitching optimization is designed to avoid the problem depicted in Figure 6.1, while still preserving the good insertion, update, and delete throughput of the SAMT. Typically an LSM-tree destroys the sets of Wanna-B-leaves that are merged together at some point after the merge completes. The intuition behind stitching is to *not* destroy these sets of Wanna-B-leaves after the merge so that we can avoid copying large sequences of els by leaving these els in their original location. When stitching we populate the secondary index with entries that point both to tuples within a newly created output set of Wanna-B-leaves, as well as to runs of sequential tuples that were left in place in older sets of Wanna-B-leaves.

The purpose of generalizing LSM-trees to support stitching is to make the LSM-tree a viable data structure for file system workloads. This will require avoiding scans when re-hashing the Bloom filters of the LSM-tree, and will require methods for handling updates and deletes during merges. Our prototype implementation has support for high-throughput sequential writes, and variable-throughput random appends, depending on the desired scan performance . It permits a trade-off between scan and random append throughput, but always out-performs existing LSM-tree designs for sequential insertion, and never performs worse for completely random insertions.

Existing file systems do not efficiently support a key-value storage abstraction. An example of a file system that comes close is BeFS [38], a file system based on B-trees with support for non-hierarchical style tagging. BeFS introduces several system calls to perform tagging, but because our implementation also supports system transactions, this baroque interface is not required. By supporting both system transactions and key-value storage, applications are free to perform their own tagging, as transactions ensure that inconsistencies do not occur between tags and tagged files. Beyond tagging, existing file systems are not well suited for many important and popular workloads

because they do not efficiently support key-value storage. We consider that the data managed by a contemporary application can roughly be divided into three categories:

Small Structured meta-data or a large collection of objects all related to each other or other data such as a collection of tags on media, data gathered from sensors such as position, or the frame index of a movie.

Large Runs of sequentially read or written data, such as a backup copy, a large media file, or a collection of randomly updated objects that are always read sequentially.

Medium Items that are neither particularly small nor large, and cost too much to write repeatedly or can cause fragmentation if space is allocated for them too naively.

Today's file systems are capable of only efficiently handling large objects, and to some extent, medium sized objects. This is by design. File system designers leave more complex workloads that are likely to be highly application-specific to database libraries [1,107,130] that can be linked by the application when needed. However similar to system transactions, the key-value storage abstraction is a well understood and widely used abstraction, and there are only a few very successful and widely used datastructures used in databases today. The major datastructures in use are the B-tree [1,23,107,130,140], the log-structured merge-tree (LSM-tree) [6,11,20,66,95], and the copy-on-write LFS tree [42,67,115,139].

The approach that this thesis explores is to extend the log-structured merge-tree design detailed in Chapter 5 to efficiently support sequential and file system workloads by introducing *stitching* and using quotient filters [12]. We extend the transactional architecture of GTSSLv1 to use *VT-trees*, our generalized version of the SAMT that supports sequential insertions. Our new transactional database is called GTSSlv2. We build a FUSE-based prototype file system on top of this database called SimpleFS to evaluate performance for a file system workload.

We begin by discussing specific arguments for stitching-based LSM-trees in the general sense, the stitching algorithm, and the properties of the VT-tree in Section 6.1. We then discuss our specific implementation of stitching and GTSSLv2 in Section 6.2. We evaluate VT-trees in comparison to traditional LSM-trees as well as perform a simple system benchmark in Section 6.3. Next we discuss related work in Section 6.4. Finally we conclude in Section 6.5.

6.1 Stitching and Related Arguments

In this Section we introduce the VT-tree. The VT-tree is an LSM-tree that supports stitching and uses quotient filters. The intuition behind the VT-tree is to delay writing for as long as possible so that better decisions can be made for both allocation and placement of the write. We explain the VT-tree, and its relationship to the LSM-tree and copy-on-write LFS trees by informally arguing several assertions:

1. Copy-on-write LFS trees (e.g., LFS [115]) and merge trees (e.g., LSM-Tree [95]) are actually two extreme configurations of the same data structure. LSM-trees never fragment and are only compacted for performance reasons by using minor and major compactions. LFS does not perform minor compactions, and must perform periodic cleaning to defragment for performance reasons.

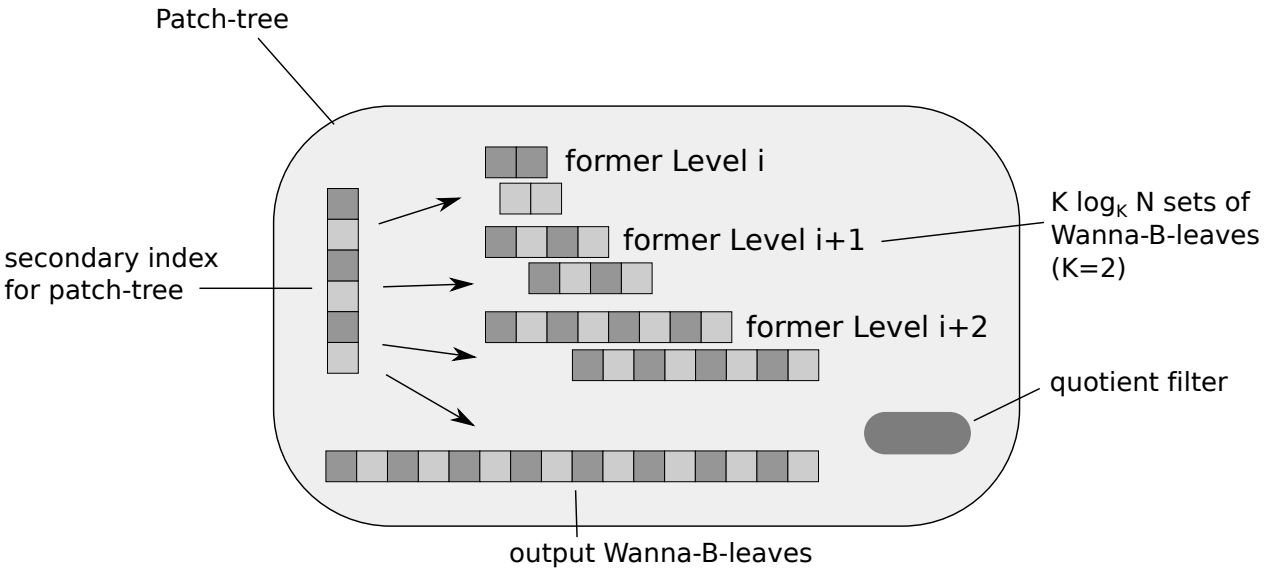

Figure 6.2: ***Patch-trees:*** *A Patch-tree is a Wanna-B-tree that may be fragmented. A patch-tree is a composite structure consisting of any Wanna-B-leaves containing stitched tuples, output Wanna-B-leaves for any tuples compacted when creating the patch-tree, and a secondary index that turns these sets of Wanna-B-leaves into a fragmented Wanna-B-tree.*

2. Sometimes it is better to leave a sequence of data items in place rather than copying them when performing a merge. This is the basis of stitching and is an extension to the MT-SAMT structure discussed in Chapter 5, and was briefly discussed in the beginning of Chapter 6. We show that the amount of stitching that should take place within a sequence of items depends on the number of times a sequence is read, the storage characteristics (including seek time and bandwidth for reads and writes), and the number and size of the elements in the sequence.

3. If using quotient filters and they fit into RAM, we can almost always perform point queries with one I/O, regardless of the amount of stitching.

4. If we always choose to stitch, due to fragmentation, scan performance will eventually hit a performance cliff and rapidly degrade. We show why in Section 6.1.1. This may be an acceptable configuration for workloads that perform mostly random writes and point queries.

These arguments are outlined in this Section. In our arguments and explanations we use the DAM model as described in Section 3.2. We now turn to the structure of the VT-tree and how it performs stitching and explain its properties.

SAMT + Stitching and the Patch-tree When the SAMT is generalized to support stitching, we call it a *VT-tree*. Both the SAMT and the VT-tree are composed of $K \log_K N$ Wanna-B-trees, but Wanna-B-trees within a VT-tree can be fragmented. In other words, the Wanna-B-leaves need not be contiguously allocated in a VT-tree but could be stored in physically separate locations on the storage device. As Figure 6.2 shows, patch-trees consist of multiple sets of Wanna-B-leaves, a secondary index that ties these sets of Wanna-B-leaves together into a potentially fragmented Wanna-B-tree, and a quotient filter. So just like a SAMT is a composition of Wanna-B-trees and

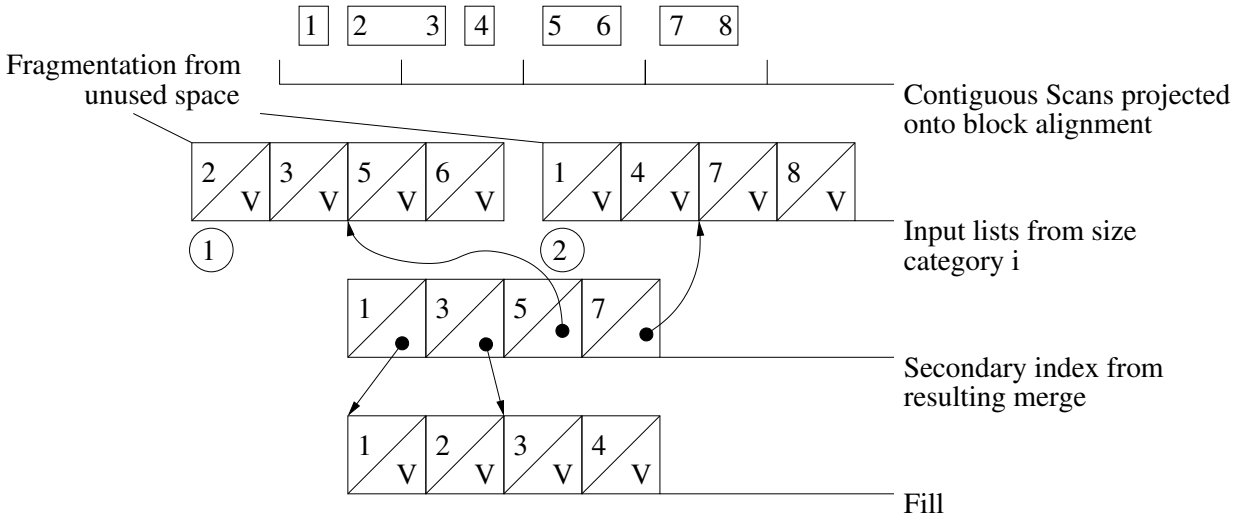

Figure 6.3: Stitching

their Bloom filters, a VT-tree is a composition of *fragmented* Wanna-B-trees and their quotient filters.

VT-trees perform queries the same way SAMTs and other LSM-trees do—as described in Section 3.2—by first querying the memory buffer, and then the most- to least-recent patch-trees. Insertions are performed the same way up until a minor or major compaction is performed. The difference lies in how patch-trees are merged together during compaction.

Figure 6.3 illustrates how a VT-tree performs a merge of two lists into a larger list without copying every single tuple as a naive LSM-tree would. Figure 6.3 executes the algorithm shown in Figure 6.4. Some tuples which were adjacent to each other in the input sets of Wanna-B-leaves will be adjacent in the output set of Wanna-B-leaves. The top of Figure 6.3 illustrates this effect: elements which were adjacent in the input and output sets of Wanna-B-leaves are outlined together in a box (e.g., 2 and 3, 5 and 6, or 7 and 8), and if an element is interleaved between two other elements during the merge, then it is outlined in its own box (e.g., 1 and 4). Each box of tuples is called a *contiguous scan-unit*. The same illustration at the top of Figure 6.3 shows which leaf-nodes the tuples would be written to in the output set of Wanna-B-leaves from the merge. In the example depicted in Figure 6.3, B_{small} is 2 tuples, so a leaf-node holds 2 tuples. In practice, b, the block size in bytes, would be sufficiently large that a block is at least 4KB large.

Figure 6.3 depicts a secondary index that is indexing the result of merging the two input lists ① and ②. To perform stitching, the VT-tree constructs a new secondary index that points to the input sets of Wanna-B-leaves and the output set of Wanna-B-leaves as shown in Figure 6.3. If a contiguous scan-unit coincides with an output block (e.g., 5 and 6, and 7 and 8), then it is an aligned leaf node and we can avoid copying it by pointing at it directly from the new secondary index. If a scan-unit does not coincide with an output block (e.g., the scan-unit is too small like 1 or 4, or is out of alignment like 2 and 3) or it should not be stitched for some other reason, then it is copied into the *output Wanna-B-leaves* (depicted in Figure 6.3), and the secondary index is made to point to the new block written to the fill. Figure 6.3 shows how the VT-tree did not have to copy tuples 5 and 6 or 7 and 8 during the merge. For large sequential insertions, the vast majority of the tuples can be broken into scans that coincide with blocks in the output list. This allows the VT-tree to

```
// Takes an array of patch-trees that need to be merged
// Returns a tuple of the new secondary index and Wanna-B-leaves
(secondary_index, wanna_B_leaves) stitching_merge(patch_tree []ps):
  // An iterator that merges together the patch-trees in 'ps'.
  // The iterator produces tuples, but is smart enough to know when
  // a tuple is at the beginning of a leaf node, and can get the
  // secondary index entry pointing to it in that case.
  patch_tree::merge_iterator mi(ps)

  // Our new secondary index and Wanna-B-leaves
  secondary_index new_si
  wanna_B_leaves new_leaves

  while (mi.has_next()):
    if (mi.leaf_node_aligned() && mi.can_stitch()):
      // We are at a tuple that's at the beginning of a leaf node
      // and that leaf node can be stitched, so we copy over the
      // secondary index entry pointing to that leaf node
      new_si.append(mi.secondary_index_entry())
    else:
      // We cannot stitch, either we are not 'leaf-node-aligned',
      // or that leaf node cannot be stitched for some reason. We
      // create a new secondary index entry pointing to the output
      // Wanna-B-leaves since that is where we will put the tuple
      new_si.append(new secondary_index_entry(
          mi.current_tuple().key,
          new_leaves.get_location()))

      // Always put at least the current tuple into the output leaves
      new_leaves.append(mi.current_tuple())
      mi.next()

      while (mi.has_next() && !mi.leaf_node_aligned()):
        // Put the following tuples in the output Wanna-B-leaves as
        // long as we are iterating in the current leaf node. When
        // we get to the beginning of the next leaf node check again
        // to see if we can stitch
        new_leaves.append(mi.current_tuple())
  return (new_si, new_leaves);
```

Figure 6.4: The stitching algorithm is written here in pseudo-code. It attempts to stitch on every available opportunity, and when it cannot stitch, it copies the tuples into a new leaf node that contains at least one tuple in a new set of Wanna-B-leaves. Nevertheless, when stitching or creating a new leaf node, a new secondary index entry is created. The algorithm returns the new secondary index and the set of Wanna-B-leaves.

leave these blocks in place, and only copy some of the blocks at the beginning and the end to the fill. This is how the VT-tree is able to avoid repeatedly copying tuples in sequential workloads.

6.1.1 Patch-tree Characteristics

We now outline some basic characteristics of the patch-tree which we rely upon. We show when we should stitch and when we should copy. We show how the decision to stitch or copy relates to the workload, the characteristics of the storage device, and the number of times a sequence of items is expected to be read. The list of properties of stitching that we discuss includes:

Limits on the secondary index overhead The overhead on the space consumed by the secondary index in RAM will not be more than double due to stitching.

Linkage between LSM and LFS LSM-trees can behave like an LFS system in that they can avoid repeated writes of tuples and rely on an asynchronous cleaning method to recover scan performance and space lost from fragmentation. LSM-trees that rely on cleaning are still fundamentally different from copy-on-write based LFS trees described in related work.

Stitching predicates The application can specify a routine to indicate what data is likely to be accessed together. The VT-tree can use this hinting routine during minor compactions to speed up insertions without harming read performance. The routine tells the VT-tree what tuples need to be relocated together physically on the storage device, and which tuples can stay far apart without impacting read performance. We call this routine a *stitching predicate*.

Limits on the secondary index overhead Since we only support stitching complete, intact, and aligned leaf nodes, we can show that a secondary index resulting from a merge that uses stitching will never be more than twice as large as a secondary index resulting from a typical LSM-tree merge. The stitching algorithm shown in Figure 6.4 greedily checks every tuple to see if it is at the beginning of a leaf node; if it is, and that leaf node can be stitched, then we stitch it. We can see that at any point this algorithm will either be copying over an existing secondary index entry from one of the K patch-trees being merged into the new secondary index, or it will create a new secondary index entry pointing to at least one but no more than B_{small} tuples. If it is stitching, it will create a secondary index entry that points to a full leaf node. Conversely, if it is copying, it could create a secondary index entry that points to a leaf node with just one tuple. If we conservatively assume that leaf nodes with a single tuple in them are wasted, then every other secondary index entry will point to a fully utilized leaf node, and the remaining entries will point to nothing and be wasted. This is why we conclude that for F secondary index entries in the output secondary index, generated during a merge of K patch-trees:

$$F - 1 \leq 2 * \frac{N}{B_{small}} \tag{6.1}$$

Equation 6.1 says that if we always stitch at least one secondary index entry, and the minimum number of secondary index entries we could attain in a typical non-stitching merge is N/B_{small}, then we will not create more than twice that many secondary index entries when stitching. It is possible for both the first and last secondary index entries to point to a wasted leaf-node so we subtract 1 from F.

Linkage between LSM and LFS LSM-trees and copy-on-write LFS systems are both log-structured, but utilize different approaches for solving two problems. The first problem is efficiently finding, updating, and writing tuples when the size of the working set is much larger than the size of RAM. The second problem is how both systems optimize for future sequential scans of tuples. Our intent in exploring the linkage between LSM-trees and copy-on-write LFS trees is to better understand the differences between how LSM-trees and copy-on-write LFS trees solve these two problems so that the best features of both can be used for any workload as appropriate.

LSM-trees solve the first problem of organizing tuples for rather large working sets by using $\mathbf{O}(\log N)$ B-trees or Wanna-B-trees of exponentially [11, 28, 31, 136] or quadratically [95, 121] increasing size as discussed in Section 3.2. LFS copy-on-write trees typically maintain a single copy-on-write B-tree that stores the keys in a leaf node separately from the values as shown in Figure 6.16. This keys-only leaf node stores the keys with pointers to their corresponding values in the values-only leaf node. When a tuple is updated, the new tuple value is written to the end of the log at position $logend$, then the keys-only leaf node that contains the key for that tuple is faulted in if it is not already resident in RAM; an updated version of the keys-only leaf node is appended with the newly written value with the key's pointer set to $logend$. We discuss issues with this approach in Section 6.4. Our LFS-like configuration of the VT-tree still solves this first problem the LSM-tree-way, by maintaining exponentially increasing Wanna-B-trees.

Solving the second problem, or optimizing for future sequential scans of tuples, is more complicated. LSM-trees automatically optimize for sequential scans of tuples by sorting them immediately upon insertion using minor compaction (discussed in Section 3.2). On the other hand, copy-on-write LFS trees do not typically perform any kind of sorting up front, but instead append new writes to a log that is later cleaned by an asynchronous background cleaning process; this process restores sufficient physical locality to ensure good performance. Thus, LSM-trees perform up-front compaction via minor compaction, and copy-on-write LFS trees perform background compaction via cleaning.

VT-trees do not always have to stitch at every opportunity. We can configure them to not stitch unless a relatively long sequence of leaf-nodes can be stitched. By compacting sequences of tuples that might have otherwise been stitched as tiny fragments, the VT-tree is able to perform any desired amount of defragmentation during a minor compaction. In this way it effectively performs cleaning up-front, rather than in a background asynchronous cleaning thread like the kind used for cleaning in a copy-on-write LFS tree. Whatever cleaning the VT-tree is not performed up-front, it could still be performed later on by an asynchronous cleaning thread (e.g., during idle time).

To relate LSM-trees to a copy-on-write LFS system, we utilize the notion of a *scan-unit*. A scan-unit is a series of els in a patch-tree which are accessed sequentially and in their entirety. For example, we could define a simple schema for holding files where the key is `<inode,page offset>`, and the primary sortin ordering is by `inode`, and the secondary sort ordering is by `page offset`. We assume in this schema that files are always read sequentially and in their entirety. In this schema, sequentially reading a file would be equivalent to scanning all els from inclusive `<inode=`*i*`,page offset=`*0*`>`, to exclusive `<inode=`*i+1*`, page offset=`*0*`>`. So then in this schema, two els belong to the same scan-unit if their keys have the same `inode` value. Ideally we want els in the same scan-unit to be sorted in sequential order, but we do not care if different scans are not physically sorted on the storage device. This is because we do not expect readers to sequentially read first file a and then immediately sequentially read file b, just because file a's `inode` number is the highest `inode` number less than file b's `inode` number. In other

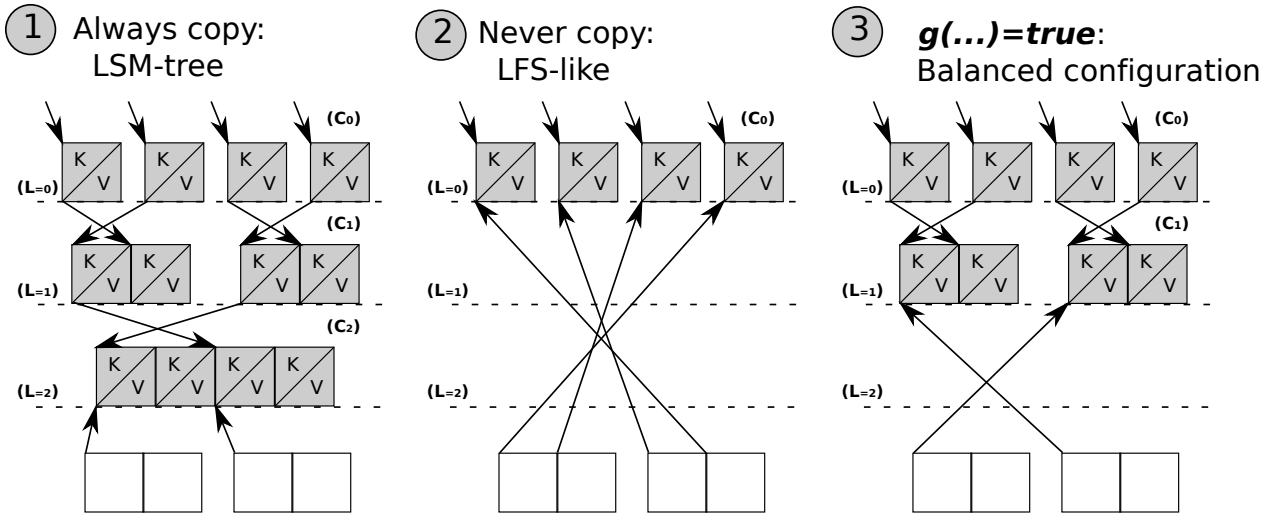

Figure 6.5: ***Linkage between an LFS and an LSM-tree:*** *Three examples of how the decision of when to stitch during a minor compaction affects the performance of subsequent scans or a cleaning operation.*

words the actual inode numbers of files do not imply the ordering in which their respective files would be read.

Figure 6.5 depicts three separate scenarios that follow along with our example of files as scans. In all three scenarios we have two files with different `inodes` and two blocks each. Each scenario shows how we trade off insertion performance by paying for more up-front compaction during minor compactions in exchange for better scan performance at some later time. Panel ① shows the behavior of the VT-tree is LSM-tree-like when the VT-tree copies on every minor compaction. The opposite configuration is shown in panel ② where tuples are only written once and are always stitched from then on. Panel ③ shows a balanced configuration where we do not spend unnecessary additional sequential I/O but do some up-front compaction during a minor compaction to eliminate unnecessary seeking.

In all scenarios we first insert the four blocks belonging to the two files, and then we subsequently insert additional tuples not shown in Figure 6.5. These additional tuples cause minor compactions on the four tuples that are shown, first at (C_1) where the four tuples are (or are not in ②) sequentially merged and written physically together into level $L = 1$. Then, just in ①, more insertions cause a second minor compaction at (C_2) where the elements are sequentially merged and written physically together into level $L = 2$. In this example, the VT-tree is configured to have two Wanna-B-trees per level so $K = 2$.

When items are read, an upward arrow indicates the placement of the cursor (a seek); as seen in panel ① in Figure 6.5, all the stitched items in separate sets of Wanna-B-leaves required a seek to scan: this is the penalty of stitching. The files are read randomly, so the two scan requests are shown separately on the bottom of each panel. In the case of ① and ②, scans of both files are satisfied with the minimal number of seeks, two. However, panel ① pays an additional unnecessary copy of the two files from $L = 1$ to $L = 2$ that panel ③ avoids. In the full stitching case shown in panel ②, we saved four sequential writes at minor compaction (C_1), but only at the cost of two additional random reads during the scan of the two files. In the general case, the LFS-like configuration would

require a random read for every block scanned when reading a file.

We now see how a VT-tree can dynamically shift from acting like a typical LSM-tree to acting like a copy-on-write LFS tree in terms of optimizing physical layout for future scans. Although the VT-tree still organizes tuples into exponentially increasing Wanna-B-trees, the physical location of these tuples in their sets of Wanna-B-leaves can range from completely random (panel ②) to fully contiguous (panel ①) depending on the predicate used to determine when to stitch. If a good predicate $g(\ldots)$ is chosen, as in the case of panel ③, unnecessary sequential writes can be avoided for sufficiently sequential scans without adding additional seek overheads when performing reads. We now discuss several predicates that can be used to determine whether to stitch or not during a minor compaction.

The stitching predicate: optimizing for future scans We consider two predicates that can be used to decide when to stitch. First a simple *stitching threshold* where $g(\ldots) = f$ for $f \geq 0$ and secondly a schema-specific *same-scan* function which returns `true` if two tuples belong to the same scan-unit.

In Figure 6.3 we showed an example where stitching was performed only if the scan-unit was leaf-node aligned. In that example, this requirement prevented tuples 2 and 3 from being stitched. We introduce an additional constraint on stitching called the *stitching threshold* which we represent with the variable f.

When performing a merge, we stitch a sequence of els iff they are all from the same Wanna-B-leaves, are leaf-node aligned, and the size of the stitch is greater than $f * B_{small}$ tuples. By requiring a stitching threshold, we can trade-off insertion performance for scan performance for schemas like the file schema.

The problem with the static stitching threshold function is that we may have files of different sizes. Consider the case where we have randomly written small files, and randomly written to a few large files. Before a compaction, we will have to seek to and merge from multiple sets of Wanna-B-leaves to sequentially read a large file. We will not have wasted any time on sequentially writing small files into a single sorted set of Wanna-B-leaves. So before compaction, reading large files will be slow because we will have to merge from many sources, but writing small files will be cheap because we will not have done any unnecessary writes yet.

After compaction, we will have fewer Wanna-B-trees. Because we are performing stitching, however, we could still have many sets of Wanna-B-leaves. How compaction effects performance depends on the stitching threshold. For large files a larger stitching threshold will be better, and for small files a smaller stitching threshold will be better. To see why, consider our above example with a large stitching threshold. The compaction with a large stitching threshold will sequentially re-write small files' pages so that they are sorted. These small files are smaller than the stitching threshold, so they will be sorted by `inode`, even though they will not be read in order of `inode`. On the other hand, larger files will benefit from the large stitching threshold as randomly written pages will be re-written so that they are physically contiguous and sequential scans of large files will be much faster after compaction.

If we decide instead to use a small stitching threshold, then after compaction small files will be stitched and left in place. However, blocks which were randomly updated in larger files will also be stitched in place. With a smaller stitching threshold we avoided sequentially writing files and sorting them by `inode`, but sequential reads of larger files will have to deal with serious fragmentation which will harm scan performance. What we see is that when the length of a sequential scan

depends on the data, as in the case of sequentially scanning files of potentially different length, then there is no ideal or optimal stitching threshold.

Alternatively we can use a schema-specific `same-scan` function as our stitching predicate instead of a constant threshold. The `same-scan` function returns `true` when two tuples belong to the same scan-unit, and `false` otherwise. In our example file schema, `same-scan` would return `true` if two tuples have the same `inode`. During a minor compaction, when merging patch-trees, we stitch only when `same-scan` return `true`, or in the case of the file schema, when the stitch would not leave two tuples with the same `inode` in different sets of Wanna-B-leaves. We can still use a stitching threshold in addition to `same-scan` to avoid compaction of files that are already much larger than b, e.g., we could always stitch when tuples are in 8MB sequences regardless of whether they belong to the same file.

We can imagine other stitching predicates besides `same-scan` and the stitching threshold f. For instance, Matthews et al. [76] modify an LFS to use a graph to remember previous access patterns; then, based on the sequence in which blocks are accessed as encoded in the graph, the cleaner places these blocks together. The stitching predicate cannot arbitrarily read any leaf nodes, but only those it encounters during the merge of the minor compaction and those it can hold in its RAM. This limits what kinds of cleaning the stitching predicate can be used for, but this does not preclude other background asynchronous cleaners from also operating.

Typically, copy-on-write LFS trees and LFS systems rely solely on cleaning to optimize physical layout for future queries. Cleaning is also used to defragment the store. Although VT-trees do not rely on cleaning for optimizing the layout to improve future queries performance, as VT-trees perform such compaction during minor compactions, VT-trees do rely on cleaning for defragmentation.

Fragmentation occurs when a region of leaf nodes that is the size of the minimum unit of allocation and deallocation is only being partially utilized. The minimum unit of allocation in our implementation is called a *zone*. Zones are 8MB large. So if a zone contains blocks pointed at by secondary index entries, along with blocks of unused leaf nodes, it is fragmented. The zone cannot be deallocated because it is in use; moreover, the unused blocks consume space that cannot be reused.

Patch-trees can be defragmented at any time. Defragmentation can be performed by performing a major compaction and setting the stitching threshold to be fairly large. This forces all tuples to be copied into an output set of Wanna-B-leaves, leaving the sets of Wanna-B-leaves that were formerly used by the merged patch-trees completely unused. While the merge proceeds, when a zone has been iterated across and all of its tuples have been moved, it can be deallocated. This defragmentation algorithm is simple; exploring alternatives in a comprehensive manner is a subject of future work.

6.1.2 Finding an Alternative to Bloom Filters

Many large storage systems employ data structures that give fast answers to approximate membership queries (AMQs). The Bloom filter [15] is a well-known example of an AMQ. An AMQ data structure supports the following dictionary operations on a set of keys: insert, lookup, and optionally delete. For a key in the set, lookup returns "present." For a key not in the set, lookup returns "absent" with a probability of at least $1 - \epsilon$, where ϵ is a tunable false-positive rate. There is a tradeoff between ϵ and the space consumption.

Within the context of the LSM-tree, Bloom filters are particularly useful. The weakness of the LSM-tree is that it must perform searches in multiple Wanna-B-trees to find a tuple when performing a point query. However, if the tuple is not within a Wanna-B-tree, that Wanna-B-tree's Bloom filter can probably quickly eliminate that I/O by returning "absent." When the Bloom filter false positive rate is sufficiently high, and all Bloom filters are resident in RAM, an LSM-tree can perform point queries at the storage device's random read throughput [136].

The problem is that if we naively use Bloom filters while stitching, then we will forfeit most of our performance gains for sequential workloads. This is because current LSM-tree implementations rely on being able to scan all the keys during a merge so they can rehash each key into the new larger Bloom filter for the output SSTable resulting from this merge operation. In our tests, reading tuple keys to rehash them into the new larger Bloom filter limited the benefit of stitching to only a 5MB/sec throughput increase over a standard SAMT without stitching. If stitching optimizations are to avoid I/O (including reads), then we have to populate the new patch-tree's Bloom filter without reading the keys that cannot fit in RAM.

We could avoid reading tuples by simply not attempting to merge the Bloom filters. In this case we would simply have many small Bloom filters, and we would have to check them all when performing lookups. Current LSM-trees need to check only $\mathbf{O}(\log) N$ Bloom filters; if we perform no merging in order to avoid rehashing tuples, however, then we would have to check more—$\mathbf{O}(N)$—Bloom filters. The expected number of false positives would increase dramatically and this has a negative performance impact on point queries as we show in Section 6.3.

More importantly, existing LSM-tree implementations also rely on rehashing tuples in order to process deletes. Current LSM-trees delete tuples from their Bloom filters by simply not including the deleted items when creating the new larger Bloom filter on a merge. If they elect to not merge their Bloom filters, they would never have an opportunity to create a new, larger Bloom filter that does not have the deleted tuples in it. Therefore, point query performance would also suffer for workloads that delete tuples due to Bloom filters aging.

If we want to use filters to ensure that point queries can be performed with a single I/O, then we will have to use an alternative AMQ structure besides the Bloom filter. In other work we developed Quotient Filters (QFs) [12], which we utilize here. QFs are a replacement for Bloom filters (BF) and also function as an AMQ. QFs are faster than Bloom filters, but consume 10–25% more space than similarly configured BFs. QFs offer two benefits we use in our VT-trees. (1) We can efficiently merge two QFs together without ruining their false positive rates (not possible with BFs). (2) we can efficiently tolerate a small number of duplicates which can also be deleted. Quotient filters are comparable in space efficiency to Bloom filters but can be merged efficiently entirely within RAM without having to re-insert the original keys.

The QF stores p-bit fingerprints of elements. The QF is a compact hash table similar to that described by Cleary [22]. The hash table employs *quotienting*, a technique suggested by Knuth [64, Section 6.4, exercise 13], in which the fingerprint is partitioned into the q most significant bits (the quotient) and the r least significant bits (the remainder). The remainder is stored in the bucket indexed by the quotient. Figure 6.6 illustrates the basic structure of a quotient filter. In this example, $0 \leq q \leq 8$ and the values $\hat{a} \ldots \hat{f} \leq 8$. If we consider an example insertion of d, we first hash d to generate its p-bit fingerprint. We will actually store the fingerprint and not d itself. d hashes to the fingerprint $2 * 8 + \hat{d}$, so $\hat{d}$ is stored at position 2.

We perform a lookup by hashing the value to its fingerprint, and then separating the fingerprint into the bucket q and remainder r. If we find the remainder r in bucket q, then we know that the

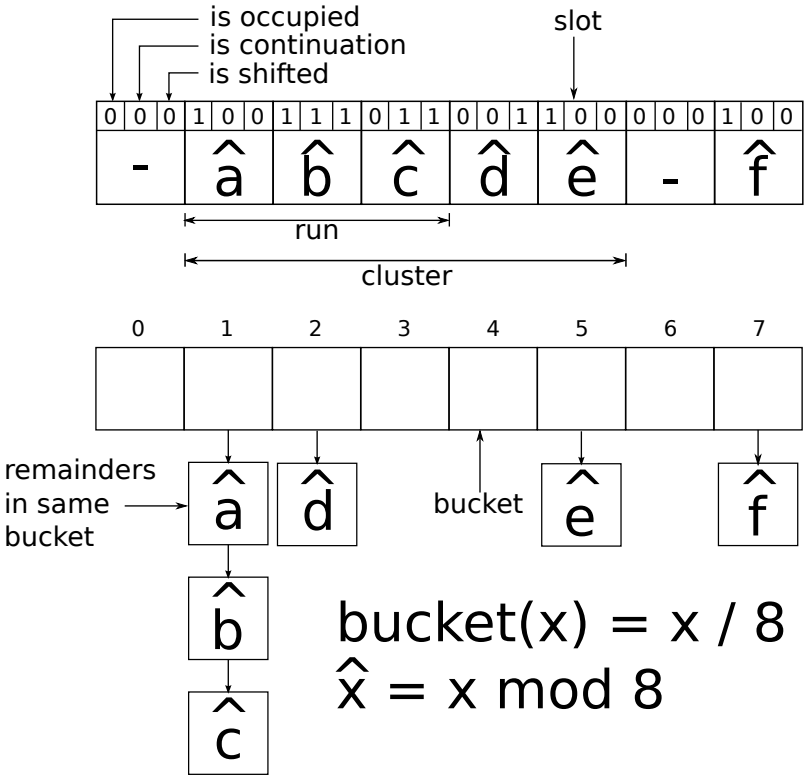

Figure 6.6: Quotient Filter

element was inserted. For example, if we query d, we check to see if $\hat{d}$ is stored at position 2, and find it is, so we report that d may have been inserted. However, we must consider the case where two different fingerprints have different remainders but the same quotient. In this case both insertions try to store different remainders in the same bucket. In Figure 6.6 we see that this occurs when a hashes to $1 * 8 + \hat{a}$ and b hashes to $1 * 8 + \hat{b}$. The quotient of both a and b is 1, but their remainders are different: $\hat{a}$ and $\hat{b}$, respectively. We cannot naively store both of these remainders in the same bucket. When two fingerprints try to store different remainders in the same bucket, we call this a *soft collision*.

To handle soft collisions, we use a linear probing method and an additional 3 meta-data bits per bucket. We adopt an encoding scheme that uses the 3 meta-data bits and a linear probing technique to convert portions of the quotient filter into an open hashtable format. Figure 6.6 illustrates how the conceptual open hashtable format that stores fingerprints of $a \ldots f$ is encoded as a quotient filter using linear probing and our 3-bit scheme.

As shown in Figure 6.6, we store remainders belonging to the same bucket in a *run*. We store runs that are physically contiguous in the quotient filter as a *cluster*. When we want to perform a lookup, we first decode the cluster. We then search through the cluster to find the appropriate run. Finally, we search through the run to see if the remainder we are looking for is there. If it is found, then we return that the element was inserted into the quotient filter. If we were to query d in Figure 6.6, then we would first hash d to $2*8+\hat{d}$, obtaining 2 as our quotient. We now must decode the cluster containing d's run. We know that the cluster containing d must begin at or also contain whatever is in slot 2 of the quotient filter. We need a way of knowing if the element that occupies

slot 2 also occupies bucket 2, or if instead it was shifted. To determine this we use *is_shifted* as one of our meta-data bits and set it to 1 if a fingerprint's remainder had to be stored in a slot not equal to its quotient. We can scan to the left until we find a slot where *is_shifted* is 0 and the next slot to the left is empty, to find the beginning of a cluster. We do this in our lookup of d and find the beginning of the cluster potentially containing d's fingerprint is at slot 1, where $\hat{a}$ is currently stored.

Next, we need to determine which buckets contain which elements and which buckets are empty. We do this by using the *is_occupied* and *is_continuation* bits. Starting from the beginning of the cluster at slot 1 where $\hat{a}$ is, we scan to the right. Every time is_occupied is 1 we know that there is at least one remainder stored in the bucket list in the open hashtable representation corresponding to that slot. Every time we see is_continuation set to 0, we know that we are reading the beginning of a new run. Since runs can be shifted away from the slot they are associated with, we must keep count of runs. For our example lookup of d, we know that it is stored 1 bucket away from where a's fingerprint is stored, so we scan to the right until we are in the second run and slot 2's is_occupied bit is set. Once we are in the second run and slot 2's is_occupied bit is set, we know that this run must contain $\hat{d}$ or else d's fingerprint was never inserted. We scan through the run of length 1 that contains d's fingerprint at slot 4 and find $\hat{d}$, so we know that d's fingerprint was inserted and the lookup returns `true`.

Quotient filters are fully functional hashtables and perfectly reconstruct the values they store. However, since we do not store actual keys (e.g., d) but rather store their fingerprints (e.g., $2*8+\hat{d}$), it is possible that different keys will hash to the same fingerprint. A false positive occurs when two keys hash to the same fingerprint. A careful analysis of quotient filters shows that a quotient filter and Bloom filter with the same false positive rate and supporting the same number of insertions have comparable space efficiency. More details (e.g., how to perform insertions and deletes and false positive analyses) are available in other work [12].

All the values stored in a QF are actually stored in sorted order. This is because after a stored value is reconstructed, its most significant bits are determined by its offset in the QF. So, if we reconstruct stored fingerprints from left to right, we obtain a list of sorted integers.

We can merge two QFs by converting them back into a list of sorted integers, merging these integers, and then re-hashing this new sorted list back into a larger QF. Since we are inserting elements in sorted order, and they are uniformly distributed across the length of the new QF, we can efficiently *append* them and insert them into the new QF without having to perform random reads. This is because newly inserted items will be stored after previous inserted items, and their uniform distribution (as they were randomly hashed fingerprints originally) makes it highly improbable that there are clusters much longer than length $logN$. This is exactly how we merge QFs in the VT-tree.

Now that we have a Bloom filter-like data structure which can be merged, we can create new larger Wanna-B-trees without having to scan any tuples from the storage device, and so we can perform minor compactions without incurring I/O when workloads are dominantly sequential. When merging patch-trees, we first merge their quotient filters, including any duplicate fingerprints. Then, during the merge of the Wanna-B-trees, if any tuples are removed due to processing updates or deletes according to the method in Section 3.2, they are also removed from the merged quotient filter. After the merge completes, we are left with a quotient filter that only has a fingerprint of x if tuple x is in the resulting merged patch-tree.

6.2 Stitching Implementation

The implementation of VT-trees within the GTSSL architecture described in Chapter 5 makes several simplifying assumptions. We describe the implementation of VT-trees in Section 6.2.1 and then discuss how VT-trees are placed within the GTSSL architecture in Section 6.2.2. Finally we describe the schema of our FUSE-based file system SimpleFS in Section 6.2.3 and how it uses VT-trees within the GTSSL architecture to perform file operations.

6.2.1 VT-tree Implementation

VT-trees are implemented as a modification of the SAMT introduced in Section 3.2, extended in Chapter 5, and generalized in Chapter 6. There are two primary ways of accessing the SAMT:

Receiving an eviction which happens when the red-black tree cache is full, and must be evicted into the SAMT.

Performing a query which happens when a client performs a query on the SAMT structure for tuples not found in the red-black tree.

We receive an eviction by performing a minor compaction if necessary, and then serializing the contents of the cache (e.g., red-black tree) into a new Wanna-B-tree, while simultaneously building a secondary index and populating a quotient filter. These two items (new Wanna-B-tree and quotient filter) are combined along with any other Wanna-B-trees pointed at by the secondary index into a new patch-tree which is then added to the SAMT. We shoed an example of this in Figure 6.2 where a patch-tree is shown being composed of 6 sets of merged Wanna-B-leaves, a new set of Wanna-B-leaves for the output, the secondary index tying the Wanna-B-leaves together, and a quotient filter.

Creating patch-trees is different than merging together and creating Wanna-B-trees in a regular LSM-tree. The primary difference lies in how we perform a minor compaction, or how we merge together patch-trees into a new patch-tree. When merging together multiple patch-trees into a new patch-tree, we first merge the quotient filter as described in Section 6.1.2, and then merge the patch-trees.

Merging the patch-trees requires having a cursor on each patch-tree: we merge the cursors and produce the output patch-tree. A cursor on a patch-tree is actually equivalent to a cursor on a Wanna-B-tree as a patch-tree is a fragmented Wanna-B-tree with a quotient filter. A patch-tree cursor consists of two parts: (1) a secondary index iterator, and (2) a leaf-node iterator. We initialize a patch-tree cursor by starting the secondary index iterator at the first secondary index entry of the patch-tree, and then placing the leaf-node iterator at the beginning of the leaf-node pointed to by that secondary index entry. We iterate through the tuples in the leaf-node until we reach the end, and then we iterate the secondary index iterator so it points to the next leaf-node, and reset the leaf-node iterator to the beginning of that next leaf-node. In this way merging K patch-trees is equivalent to having K patch-tree iterators, which means a merge of K patch-trees involves K quotient filters and K Wanna-B-trees.

We must be careful when merging patch-trees to not read leaf-nodes that are being stitched as this will accidentally cause I/O, even if we do not mutate the value of that leaf-node. Because of this, we perform two kinds of operations when merging patch-trees into a new patch-tree:

Load tuple which loads a single tuple into the leaf-node currently being filled in the output Wanna-B-leaves being created by the merge operation.

Stitch which copies the current secondary index entry into the new secondary index being created by the merge, stitching the leaf-node pointed at by that secondary index entry.

We perform the actual merge by executing the algorithm listed in Figure 6.4. We merge multiple patch-trees into a new patch-tree by stitching whenever possible. It is only possible to stitch when the merge operation is currently processing a tuple that starts at the beginning of a block pointed to by a secondary index entry. In this case, the secondary index entry has the same key value as the tuple, so we avoid retrieving the key from the block itself, and simply use the copy in the secondary index entry. When we cannot stitch, we load tuples until we are able to stitch again. This is how we avoid reading leaf-nodes when stitching.

We attempt to stitch forward in the `stitch_next` method depicted in Figure 6.7. We construct an iterator that merges multiple patch-tree iterators. This iterator is called a `merge_iterator`. The routine is called only if the current tuple in the merge is at the beginning of a leaf-node pointed at by a secondary index entry. In this case, the `merge_iterator` is at a *stitch point*. Each time we stitch one secondary index entry, we stitch the tuples within the leaf-node pointed at by that secondary index entry into the new secondary index for the patch-tree currently being constructed from the merge. To implement a stitching threshold as discussed in Section 6.1.1, we do not want to stitch leaf-nodes if there are not enough of them contiguously laid out in sequential order. We require that there be at least `stitch_thresh` leaf-nodes in a row to stitch them.

This routine returns two pieces of information: (1) whether or not we can stitch the next `stitch_scan_cnt` secondary index entries, and (2) `stitch_scan_cnt`. For example, if `stitch_next` returns `(0,10)`, then the first tuple element 0 tells us that we can stitch, and the number of secondary index entries that we can stitch is 10. If, on the other hand, `stitch_next` returns `(-EINVAL,10)`, then we must load each tuple for the next 10 leaf-nodes into the output set of Wanna-B-leaves and create new secondary index entries to point to these new leaf-nodes in the output set of those Wanna-B-leaves. In this way `stitch_next` determines on behalf of the caller how many secondary index entries to stitch, or how many entries it must process by copying before it can attempt to stitch again.

Currently our implementation always tries to stitch as many secondary index entries as we can. However, we cannot always stitch. There may not be enough blocks to overcome the stitching threshold, or the blocks may contain tuples that need to be removed during the merge (e.g., *delete* or tombstone tuples, see Section 5.3). In these cases we report that these secondary index entries cannot be stitched, but must be processed by copying their tuples into the output Wanna-B-leaves.

As shown in Figure 6.7, `stitch_next` begins by verifying that we are at a stitch point. Next, `min_vec_cursor` returns a pointer to the iterator being merged by the `merge_iterator` with the smallest key according to our comparison method `bt->Merge`. This pointer is stored in `min`. Next, we create `second_min` which is explicitly initialized with an invalid pointer value `cs.end()`. We then compare `min` in a loop with the other iterators to find the second-smallest iterator that is not finished iterating, and store it in `second_min`. If all other iterators are finished, `second_min` continues to be initialized with the invalid pointer value `cs.end()`. Next, we dereference `second_min` and set `kv` to `NULL` if `second_min` is not initialized or to the key `second_min` points to if it is.

```
tuple<int,int> merge_iterator::stitch_next()
{
  // We can only stitch from a valid stitch point
  assert(stitch_point());
  auto min = min_vec_cursor();

  // Get the value of the next smallest iterator's key
  auto second_min = cs.end();
  for (auto i = cs.begin(); i != cs.end(); ++i)
    // i is not min and is not ended so it has a valid
    // value, and second_min either hasn't been set yet,
    // or i is smaller then second_min
    if (i != min && !(*i)->is_end() &&
        (second_min == cs.end() ||
         (bt->Merge((*i)->get_key(), (*second_min)->get_key()) <= 0)))
      second_min = i;

  const dbt_buff *kv;
  if (second_min != cs.end())
    kv = (*second_min)->get_kv();
  else
    kv = NULL;

  // stitch_check_scan(NULL,...) means scan to the end
  int stitch_scan_cnt = (*min)->stitch_check_scan(kv, redacting);

  if (stitch_scan_cnt <= stitch_thresh);
    return make_tuple(-EINVAL,stitch_scan_cnt);
  else
    return make_tuple(0,stitch_scan_cnt);
}
```

Figure 6.7: An internal routine to the VT-tree enabled SAMT which makes appropriate modifications to a merge iterator when it is possible to stitch one or more secondary indexes during a merge.

```
uint64_t patch_tree::iterator::stitch_check_scan(
    const dbt *kv, bool r) const
{
  assert(parent->ith_source_block(index_curr) == p);
  auto end = parent->sis.end();
  auto i = index_curr;
  if (i.equals(end))
    return 0;
  auto i_after = index_curr.iterate_forward(1);
  while (!i.equals(end) &&
      !i_after.equals(end) &&
      (!kv || parent->bt.Merge(kv, i_after.get_dbt())) >= 0) &&
      !(r && ((ptr *)(i.get_dbt()->get_value()))->is_redacting())) {
    i.next();
    i_after.next();
  }

  return i.distance(index_curr);
}
```

Figure 6.8: An internal routine to the VT-tree allows SAMT to ensure that a series of secondary index entries from a patch-tree is eligible for being stitched across.

We now try to count how many secondary index entries we can stitch by calling `stitch_check_scan`, whose implementation is depicted in Figure 6.8. The `stitch_check_scan` routine verifies that we are at a stitch point by checking that the secondary index iterator of the patch-tree iterator and the leaf-node iterator are pointing at the same location. Next, we initialize and then iterate `i` until it is at the secondary index that is furthest from `index_curr` but whose key is not greater than or equal to `kv`'s key according to the `bt.Merge` comparison routine. If we are supposed to elide certain tuples then `r` is set to `true` and we verify that the leaf-node pointed at by the secondary index entry has no tuples within it that need to be elided. If it does have tuples that may have to be removed during the merge, then we cannot stitch that leaf-node, and so we stop the stitch at the current position. Once we have stitched forward as far as possible without violating any of the above invariants, we return the distance we successfully stitched (which could be zero).

We can force the VT-tree to act like a regular SAMT by never stitching and always loading tuples into the output Wanna-B-leaves constructed during the merge of patch-trees. If we do this then the other sets of Wanna-B-leaves will not be pointed at by any new secondary index entry created during the merge, and those unused sets of Wanna-B-leaves will be automatically removed and deallocated during the merge. We can compare traditional LSM-trees to VT-trees by simply increasing the stitching threshold to be sufficiently large enough that we always copy tuples into the output Wanna-B-leaves. This is how we evaluate VT-trees in Section 6.3.

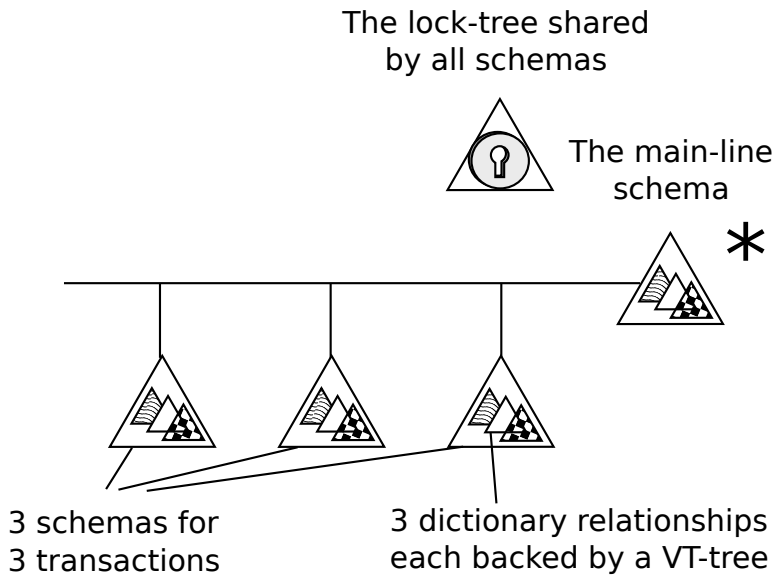

Figure 6.9: ***Multiple transactions as multiple schemas:*** *Multiple transactions are represented as multiple schemas and a main-line schema. All transactions and the main-line schema share the lock-tree to avoid conflicting with each other.*

6.2.2 VT-trees within GTSSL

VT-trees sit underneath the exact same red-black tree cache used in GTSSLv1. In our current implementation of the GTSSLv2 architecture, we support transactions that are larger than RAM by giving each transaction a separate set of VT-trees for their new insertions. As shown in Figure 6.9, the system groups VT-trees into a *schema*. The schema is the set of all dictionary relationships the user has created. There are multiple instances of the schema, one for each running transaction, and a *main-line* schema. These schemas all have the same number of VT-trees, are compatible, and can be duplicated and merged together. Transactions begin by creating a new schema, modifying it, and then merging it back into the *main-line*. This method of managing transactions is analogous to source control systems [41]. This method of managing transactions is also similar to the Primebase XT transaction manager for MySQL [79]. The difference between how GTSSLv2 manages transactions and how Primebase XT manages transactions is that GTSSLv2 transactions do not have undo images, are indexed as LSM-trees, and can be queried. These differences are necessary to eliminate undo images; this is important for supporting high throughput random write workloads (see Section 3.1.1).

The set of locks on individual tuples and ranges of tuples is stored separately in RAM and shared by all transactional and non-transactional processes. Although transactions can grow to be larger than RAM, their set of locks currently cannot in our implementation. The lock-tree in RAM uses ranges of locks and heuristic methods that can be customized by applications to pre-lock ranges of the key-space within a particular dictionary relationship. This permits applications to interact with more tuples than the lock-tree can hold locks for in RAM by using one range lock to protect many tuple updates. Large, complex transactions that cannot group tuple accesses into range locks, may have to abort if their sets of locks cannot be held within RAM within the lock-tree. As future work, we are investigating storing locks within a VT-tree as well so that the set of locks can grow to be

larger than RAM.

Transactions begin by creating a new schema, a set of VT-trees. The transaction makes modifications to the schema by first acquiring appropriate locks within the lock-tree shared by all schemas in RAM. Then they insert modifications into the schema's red-black tree until the red-black tree is full and it must evict into the schema's VT-tree for that dictionary relationship. Transactions perform reads by acquiring the appropriate locks within the lock-tree in RAM, and then querying for the tuple in the transaction's schema or the main-line schema depending on which schema has the most up to date version of the read tuples.

Transactions commit in two steps. First, the transaction inserts the elements in each of its schema's VT-tree's red-black trees into the corresponding red-black tree for each VT-tree in the main-line schema. If any of the main-line schema's red-black trees must evict, they do so. Both GTSSLv1 and GTSSLv2 perform this first step. However, GTSSLv2 executes a new second step: it updates the main-line schema's VT-trees such that they now include the corresponding patch-trees belonging to the committing transaction (if it has any). These patch-trees contain the transaction's updates that had to be evicted from RAM because the transaction became too large.

The extensions in the implementation of GTSSLv2's transactional architecture permit larger-than-RAM transactions using the above design and implementation, but evaluation of this system is currently a subject of joint research and future work with Pradeep Shetty. See Chapter 8. All experiments in Section 6.3 were run without transactions and operated only on the main-line schema with interaction with the lock-tree disabled.

6.2.3 SimpleFS and System Benchmarking

One of our goals was to test the feasibility of VT-trees to act as the underlying data structure in a file system. Although VT-tree performance for sequential and file system workloads compares very favorably to regular LSM-trees, we wanted to test their performance against native kernel file systems. For that purpose, we developed SimpleFS: a FUSE implementation of the same schema used by LSMFS 3.3.2. The SimpleFS layer is an implementation of a FUSE low-level interface which translates FUSE file system requests into key-value requests. To support file system requests, the SimpleFS layer creates three VT-trees. The three Key-value pair formats for these three VT-trees are shown below. The VT-trees are (1) *nmap* for namespace entries, similar to `dentries`. In *nmap*, the *path-component* of a file and its parent directory's `inode` number forms the key, and the value is the `inode` number of the file. (2) *imap* for storing inode attributes; and (3) *dmap* for the data blocks of all files.

VT-tree	**Format**
nmap	$\langle\{$*parent-inode#*, *path-component*$\}$, *inode#*$\rangle$
imap	$\langle$*inode#*, *inode*$\rangle$
dmap	$\langle\{$*inode#*, *offset*$\}$, *data-block*$\rangle$

6.3 Evaluation

To evalute the effects of stitching on performance of sequential, random insert, and random append workloads we modified the GTSSLv2 compaction layer to support three modes: (1) NO-COMPACT

and (2) STITCHING. When compacting in the STITCHING mode we can set the stitching threshold f to any value from 0 to $2^{64} - 1$. We can force the compaction layer to behave as a traditional LSM-tree implementation by setting $f = 2^{64} - 1$. This is how we obtain our third configuration (3) COPY-ALL. When stitching, if GTSSLv2 must perform a copy, it uses the same code that COPY-ALL would use. By measuring only algorithmic changes on top of the same underlying implementation, we are able to isolate differences in performance to just the efficiency of our stitching algorithm and implementation.

We find that workload differences greatly affect the performance gains made by stitching. In the following sections we discuss three micro-benchmarks. Finally in Section 6.3.7 we discuss a file server system benchmark. Each section first describes the workload, predicts its performance in GTSSLv2's compaction layer, and then we analyze benchmarks for that workload.

6.3.1 Experimental Setup

Our evaluation ran on the same machines that we ran our experiments in Chapter 5. Our tests used pre-allocated and zeroed out files for all configurations. We cleared all caches on each machine before running any benchmark. To minimize internal Flash SSD firmware interference due to physical media degradation and caching, we focus on long-running throughput benchmarks in this evaluation. Therefore, we reset all Flash SSD wear-leveling tables prior to evaluation (using the TRIM command), and we also confined all tests utilizing Flash SSD to a 90GB partition of the 159.4GB disk, or 58% of the disk. This is near the ideal partition size to maximize random write throughput of the device [55]. In all tests, the GTSSLv2 compaction layer was configured to use an 8MB key-value cache (std::map [89]) for sorting randomly inserted keys. Sequential insertions were also placed in this cache for fairness. Once the 8MB cache fills, we evict it into the compaction layer and write its contents asynchronously with `pwrite` (and `msync` with `MS_INVALIDATE` to re-synchronize the page cache). Quotient Filters are enabled in all runs, as are secondary indexes. Unlike GTSSLv1, GTSSLv2's secondary indexes are stored in a file `mmap` and may need to be faulted in during a run. This will cause some read I/O for stitching workloads as we will see in the subsequent sections. All tests were run on the Intel X25-M Flash SSD

Storage Device Performance In the following experiments, we refer to the random and sequential read and write throughput of our underlying Flash SSD, the Intel X-25M.

	read	**write**
random	3,000 IOps	2,100 IOps
sequential	245 MB/sec	110 MB/sec

Table 6.1: Performance of our Intel X-25M Flash SSD

In Table 6.1, we see what random and sequential read and write throughput the Flash SSD is capable of. When measuring random-write throughput, we noticed initially a high write-throughput (roughly 5,000 IOps) that dropped considerably and became highly variatic once the underlying FTL began performing compactions in parallel with our random writes. These tests were performed using large `mmaps` of the entire disk (90GB), while the machine was only booted with 3GB of RAM. Random writes would first have to fetch the page before writing to it. Sequential read

and write throughput were measured with `dd`, either writing zeros to the entire 90GB volume, or reading the entire volume into `/dev/null`. All datasets written are 10GB large, so for example, if randomly reading 4KB pages from the dataset, with 3GB of RAM, we'd expect to see a factor $1.3\times$ speed up on random reads, so 3,900 lookups per second instead of 3,000.

Configuration For each workload, we performed three benchmarks: INSERT, SCAN, and POINT-QUERY. For each workload we first run the INSERT benchmark resulting in a populated VT-tree, and then run the SCAN and POINT-QUERY benchmarks on this tree. During insertion, we randomly insert 4KB tuples consisting of an 8B key, a 4087B value, and a 1B flag into an 8MB in-RAM red-black tree, which is then evicted into the VT-tree data structure. Key generation depends on the workload we are performing. For SEQUENTIAL-INSERTION we pick a random number from 1 to $2^{64} - 1$ and use that as our key, and then sequentially insert 16,384 of the following numbers after (e.g., we pick 42, then insert 42, 43, ..., 16,426). For RANDOM-INSERT we just insert random numbers. For RANDOM-APPEND we simulated the effect of randomly selecting a range of tuples, and then just appending to the end of this range. This would be analogous to randomly selecting files and appending to them (e.g., a `/var/mail` workload [32]). For benchmarks running this workload, there are 2,040 "files" or ranges, and we append to each of them 1,280 times, resulting in 10GB of random appends.

The details of how we perform each benchmark follow. To perform the INSERT benchmark on a workload, we filled a red-black tree with 8MB of 4KB tuples (2,040 tuples), and then evicted these tuples into the VT-tree data structure directly. We repeated this 1,280 times, for a total of 10GB of insertions. The only difference between workloads is how the keys are generated for these 4KB tuples. We bypassed the transactional and standard caching layers completely so as only to measure the VT-tree's performance with various merging and stitching configurations. Note the only cache available is the 8MB red-black tree, and the remaining 3GB serve only as a file or page cache. To perform the SCAN benchmark, we randomly select the beginning of an inserted sequence of `seq` tuples, and then scan all `seq` tuples in that sequnce. To perform the POINT-QUERY benchmark, we randomly select a tuple from all of those inserted, including those within a sequence of tuples, and retrieve its value.

We never perform point queries on tuples that do not exist, so our QFs can only help us hone in on the patch-tree in the VT-tree with the query result, and cannot completely avoid work from a point query. This is done by generating random numbers with a 64-bit hash function instead of a linear congruence.

To randomly insert numbers, and guarantee existing lookups, we hash the numbers from $0...N$, to generate random numbers. Then, to perform queries, we randomly select a number from $0...N$, hash it, and can know that we have inserted it while performing the lookup. We used the 128-bit Murmur hash [9].

6.3.2 RANDOM-APPEND

The random-append scan workload randomly appends new elements to a number of *scan-units* as discussed in Section 6.1.1. One of the intuitions of stitching is that it is not necessary to perform all the work that a normal LSM-tree performs to obtain good scan performance. Figure 6.5 shows in panel ③ a scan-unit of 8 blocks on the bottom, the upward arrows indicating seeks to four positions to perform the scan. If the cost of sequentially transferring two blocks is equivalent to the cost of

a single seek, then we may find it sufficient to group blocks belonging to the same scan-unit into groups of two. This would avoid additional copies during merge, but guarantee a scan throughput that is within a factor of two of optimal. We can set the stitching threshold $f = 1$ to ensure that sequences of length one are copied with the next or previous tuple during a merge such that we rarely have a stitch of length one. Consequently, scans will spend roughly an equal amount of time transferring as seeking.

Configuration The random-local scan workload performs *local updates*, or updates that will only dirty a small number of blocks out of all the blocks belonging to a scan-unit. Appending to a scan-unit is an instance of a local update, because the only block dirtied is the block at the end of the scan-unit, and the blocks being newly appended. Our synthetic random-local scan update workload appends newly inserted blocks to the end of the scan-unit.

The workload operates as follows. First, a series of 2,040 random numbers are selected. Next, in the first pass, we insert these random numbers adding an offset `off` to each random number. Initially `off` is 0, so the first pass simply inserts the random numbers. Afterward, we increment `off` by 1, and perform a second pass, inserting the 2,040 random numbers we generated, this time incremented by 1 (`off`). We repeat this 1,280 times, incrementing `off` each time. Since there are 2,040 4KB pages in an 8MB eviction, we insert a total of 10GB of 4KB tuples in this manner.

Since we generate 64-bit random numbers, it is highly probable that the 2,040 numbers we generate will be further than 1,280 apart, and so we should rarely have overlap. In this way, we randomly append to 2,040 scans, 1,280 times. It will not cause a fault if there are any overlaps or duplicates, so we do not check.

We expect that the majority of the blocks in a scan-unit will not be dirtied by interleaving blocks, because blocks from two separate patch trees being merged together from the same scan-unit will be ordered such that the more recently created patch-tree will have blocks that come after those in the older patch-tree. This way the merge can stitch the blocks if they are already greater than the stitching threshold.

This means that increasing the stitching threshold will slow down insertion throughput, but should increase scan throughput by decreasing the number of seeks necessary to read a stream of blocks. Based on Table 6.1, in the time it takes to perform a random read IOp on the Flash SSD, we could read 85KB at full sequential read throughput. Naively we can say that setting the stitching threshold such that blocks larger than 85KB are stitched will grant us a scan throughput no worse than half the Flash SSD scan throughput, or 122MB/sec (half because we spend half our time seeking, and half our time transferring). For a 1MB block, we would expect to see a 226MB/sec read throughput.

Figure 6.10 shows the result of randomly appending tuples to 2,040 scans. The y-axis shows the number of evictions performed by the benchmark until it reaches 1,280 and ends, and the x-axis shows the time that each eviction occurred. The second y-axis shows the block read throughput. When we perform a merging compaction, portions of a scan-unit are placed together. Once the aggregated scan-unit has become sufficiently large that it is not less than the stitching threshold f, it is no longer copied during compaction. Therefore we can trade insertion throughput for scan throughput: by increasing the stitching threshold thus forcing tuples to accumulate together across merging compactions to ensure a higher locality of tuples when scanning. We have chosen several key stitching threshold, $f = 0$, $f = 3$, $f = 15$, and $f = 16,383$, and we expect to perform 1, 2, 3, 4, or more copying merges respectively as $K = 4$. We see respectively decreasing

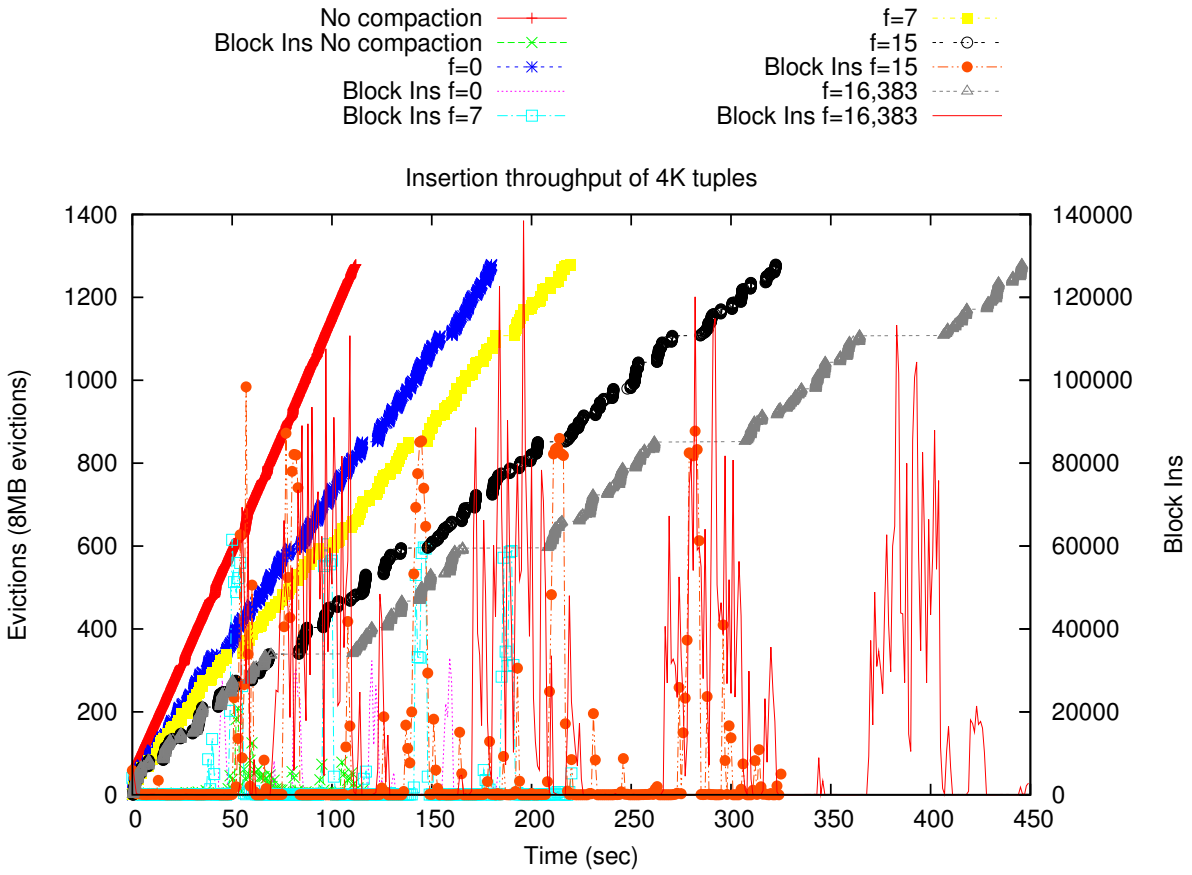

Figure 6.10: Random Append and Stitching

throughputs of 57MB/sec, 47MB/sec, 31MB/sec, and 23MB/sec as we spend more time in each merge performing copying merges. The amount of Block Ins decreases accordingly as well, with $f = 16,383$ performing the most Block Ins during compaction, and decreasing Block Ins bursts as f decreases. Since appends are random, the stitching code must perform more work to update the secondary indexes as scans accumulate together across merging compactions, so we are not able to reach the NO-COMPACT throughput of 90MB/sec, which is only bound by time spent serializing tuples into the page cache.

As depicted in Figure 6.11, scan throughput was 16MB/sec, 63MB/sec, 81MB/sec, and 131MB/sec for the stitching thresholds of $f = 0$ through $f = 16,383$ accordingly. We insert into VT-trees with the lower stitching thresholds more quickly by avoiding copies during merging compactions. This improvement is paid for with lower scan throughput due to loss of locality.

6.3.3 SEQUENTIAL-INSERTION

We measured the result of sequentially inserting groups of 16,384 4KB tuples according to the SEQUENTIAL-INSERTION workload. The NO-COMPACT configuration performs no merging compactions of patch-trees, and so there are 1,280 patch-trees upon completion of the benchmark. All other configurations perform stitching, with either a set threshold of $f = 0$, $f = 15$, or $f = 16,383$. As long as we stitched less than the `seq`, we performed no I/Os on compaction, as can be seen by

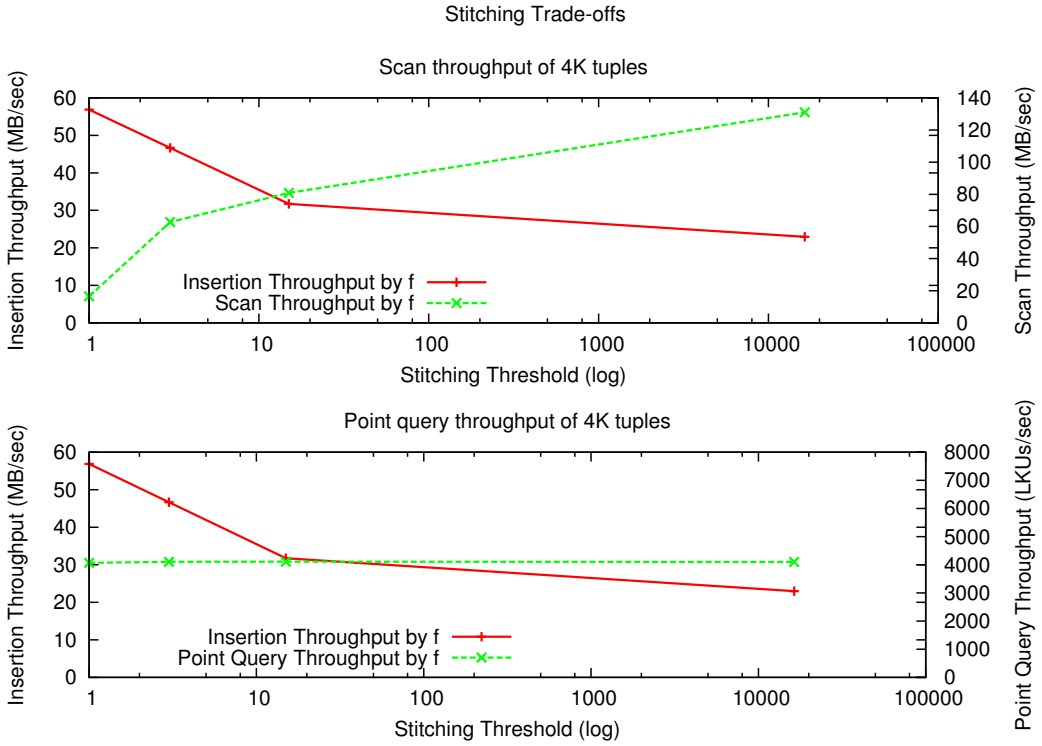

Figure 6.11: Stitching Trade-offs

the minimal levels of block ins measured by `vmstat` on the second y-axis. Since the sequentiality of the insertions was always higher than the stitching threshold up to 15, both $f = 0$ and $f = 15$ perform at the same throughput, 85MB/sec and 87MB/sec, respectively. In our `vmstat` logs all stitching configurations were CPU-bound (25 us on a 4-core machine), but primarily by our code serializing tuples to the underlying page cache. This is evident as NO-COMPACT, which performs no compactions or merges, and only serializes to the underlying page cache, has a closely matching throughput of 91MB/sec, while the underlying Flash SSD throughput is 110MB/sec. The fully compacting configuration of $f = 16,383$ performs insertions at 28MB/sec, or less than $1/3$ the throughput of our stitching configurations for sequential insertions.

For scan throughput for the SCAN benchmark on the SEQUENTIAL-INSERTION workload, we measured 152MB/sec, 158MB/sec, 149MB/sec, and 25MB/sec, respectively, for $f = 0$, $f = 15$, $f = 16,383$, and NO-COMPACT. For compacted VT-trees, scan throughput was between 60–64% of the Flash SSD's sequential read throughput. The difference in performance is a combination of having to read eviction data in 8–32MB chunks depending on f, and having to set cursors in each patch-tree before performing a scan. The latter component is exacerbated in the NO-COMPACT run where throughput is only 25MB from having to set 1,280 cursors before performing a scan, instead of less than $4 * log_4(1,280 * 2,040) = 40$ (the bound on the resulting number of lists after compaction with any stitching threshold).

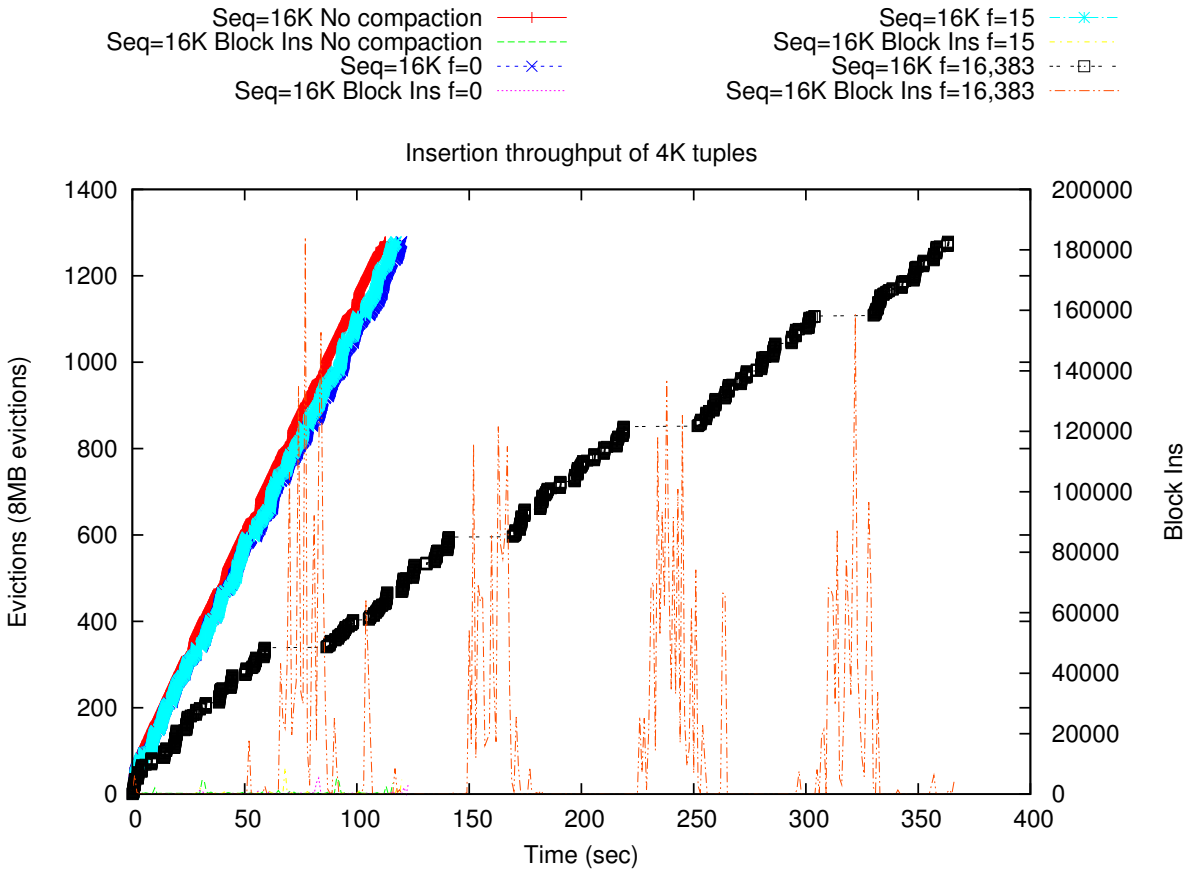

Figure 6.12: Sequential Insertion Stitching

6.3.4 RANDOM-INSERTION

The RANDOM-APPEND workload is less obviously sequential than SEQUENTIAL-INSERTION, but there are still hot spots and cold spots. The hot spots are the ends of the scan-units being randomly appended to, and the cold spots are the bulk of the scan-units which remain unmodified after each append.

With the RANDOM-INSERTION workload, there are no cold spots, and the VT-tree is unable to significantly benefit from stitching. As seen in Figure 6.13, benchmarks running this workload experienced no change in insertion throughput for lower stitching thresholds (except $f = 0$ which can always stitch 4KB tuples). This is because it is highly probable that two patch-trees of random numbers will interleave completely, making stitching configurations always choose to copy on the merge anyway. When performing random insertions, all configurations with a stitching threshold exceeding the size of the sequential cluster inserted performed at nearly the same throughput, 24MB/sec, the same throughput as the $f = 16,383$ configuration, as all configurations copied on every merge.

Configurations that had more sequential insertions performed better than those with less, even if both stitched completely. The $f = 0$ configuration for sequentially inserted clusters of length 1 is 50% slower than the $f = 0$ configuration for clusters of length 16. The overhead from shorter merges dominates the I/O overhead of longer merges, so for this 10GB workload, the $f = 0$ configuration for clusters of length 1 seems almost linear.

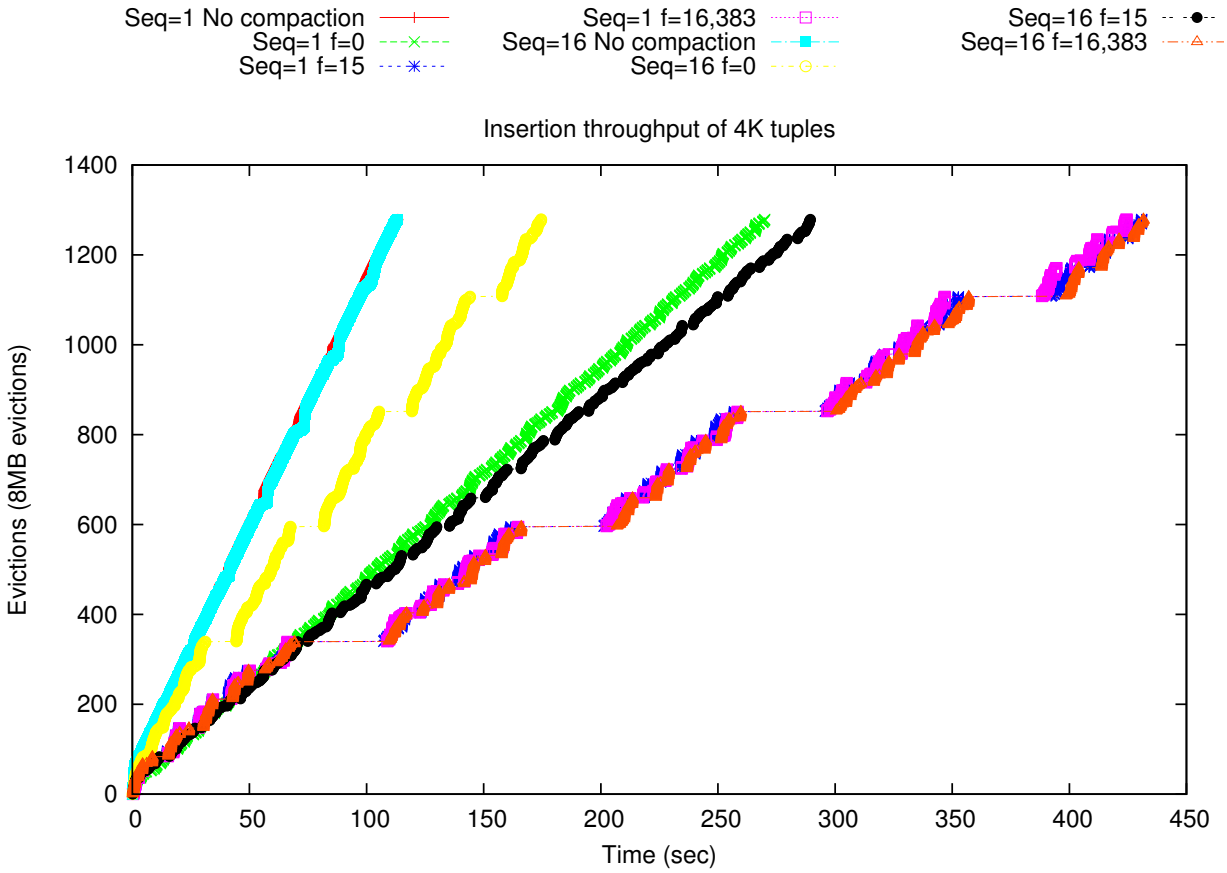

Figure 6.13: Random Insertion Stitching

6.3.5 Point Queries

As depicted in Figure 6.11, in all compacting benchmarks run, we found point query throughput was always between 3,900 and 4,000 IOps, or 1.3 the random read throughput of the Flash SSD because of the page cache answering 30% of the read IOps. This was also true in other workloads besides RANDOM-APPEND. In all NO-COMPACT benchmarks, lookup throughput was much worse, always 1,400 IOps, due to having to consult 1,280 QFs since we did not merge filters. This would be equivalent to lookup throughput if we were using Bloom filters and not performing merges. The configured FP-rate for each QF was $1/1,024$, so with 1,280 QFs to check in NO-COMPACT, we can expect an additional 1.25 random read IOps, for a total expected number of 2.25 random read IOps/lookup, giving us an expected lookup throughput of 1,777, in addition to CPU overheads and having to perform lookups in a fragmented secondary index, lookup throughput drops to 1,400. This issue will become greatly exacerbated for datasets larger than 10GB. Therefore, we concluded that QFs are necessary to maintain high point-query throughput while stitching.

6.3.6 Tuples of 64B in size

In addition to experiments run on 4KB tuples, we also conducted identical experiments on 64B tuples. In these workloads, insertions are primarily CPU-bound, and yet we find that stitching still helps considerably. The reason is that not only are I/O overheads avoided by stitching, but so are CPU overheads.

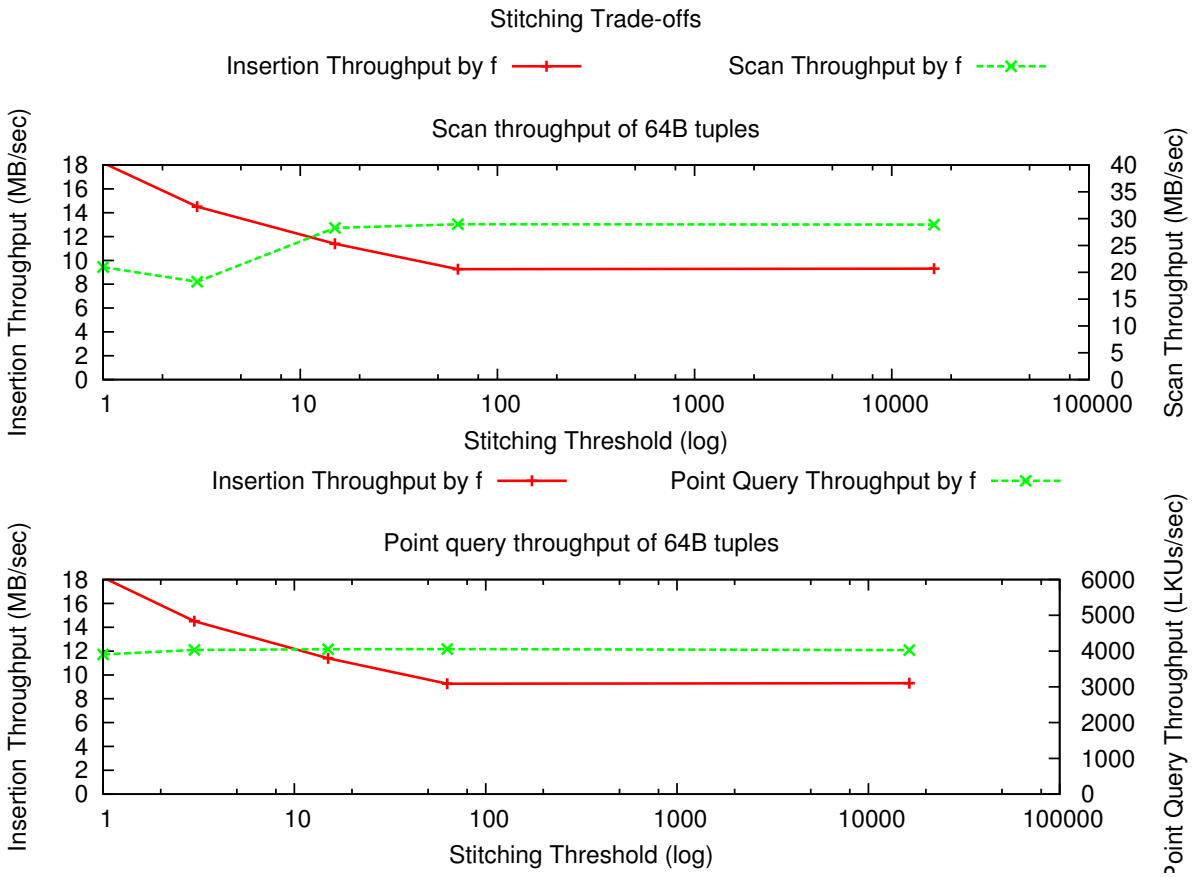

Figure 6.14: Random Append of 64B-tuple Stitching

Configuration In this RANDOM-APPEND workload, we inserted tuples in sequences of 64, otherwise the optimal strategy is to always copy on merge. It is natural that small tuples require longer sequences to benefit from stitching. This is a benefit of stitching: very small tuple insertions are automatically handled in an optimal fashion with copying on every merge. Stitching only optimizes sequential workloads. Small levels of sequentiality (e.g., the aggregate sequence after multiple compactions is still smaller than the stitching threshold) will be handled as if randomly inserted.

Figure 6.14 shows the stitching trade-offs for 64B tuples instead of 4KB tuples. Stitching thresholds $f = 0$, $f = 3$, $f = 15$, $f = 16,383$ each were measured at 18.2MB/sec, 14.5MB/sec, 11.4MB/sec, and 9.3MB/sec. All insertion throughputs are well below the disk saturated insertion throughput of 110MB/sec, and our `vmstat` logs show that 100% of the CPU was spent in user-level code: the benchmark was CPU-bound. Still, stitching provides a $2\times$ performance improvement over the $f = 16,383$ configuration as even the CPU time spent scanning tuples can be greatly reduced by stitching. Scanning with compaction for $f = 16,383$ compared to $f = 0$ is 38% faster at 28.9MB/sec compared to 21.0MB/sec.

6.3.7 Filebench Fileserver

SimpleFS uses FUSE to support POSIX operations. Using FUSE requires two additional context switches and buffer copies than running on a native file system. This results in around 2–3$\times$ overhead compared to the native file system performance [110]. However, serial reads on FUSE are comparable and even some times better than native file systems. This is due to caching and read-

Ext4	FUSE-Ext4	SimpleFS	FUSE-XFS
2,411	604	617	897

Table 6.2: Filebench file server workload results in ops/sec

ahead performed at both the FUSE kernel component and the lower native file system [110]. The FUSE kernel module caches read pages, but writes are immediately sent to the FUSE server running in user space. Supporting write-back cache in FUSE kernel is complex as the entity responsible for accepting or rejecting the writes is not the FUSE kernel, but the FUSE server instead. To exclude FUSE overhead, we compare SimpleFS's performance with FUSE-Ext4 and FUSE-XFS, a pass-through FUSE mounted on Ext4 and XFS, respectively. We also evaluated Ext4 to measure FUSE overhead by comparing it against FUSE-Ext4. We use Filebench [32] for evaluating these systems.

Configuration SimpleFS creates an *fs-schema* consisting of three schemas—*nmap*, *imap*, *dmap*—as SAMTs (with stitching), as described in Section 6.2.3. We configured *nmap*, *imap*, and *dmap* to have RAM buffers of sizes 6MB, 12MB and 512MB, respectively. We set the stitching threshold to 1MB for all the runs. Fileserver workload performs a sequence of creates, deletes, appends, reads, writes and stat operations on a directory tree. We configured the mean size of the file to 100KB, mean append size to 16KB, directory width to 20, and number of files to 100K. Filebench pre-allocates 80% of the files and randomly selects a file for each of the above operations. We ran the benchmark for 10 minutes with 10 threads and I/O size of 4KB for all file systems on Flash SSD.

As seen in Table 6.2, for file server workload, SimpleFS performs comparable to FUSE-Ext4. FUSE-XFS has 45% better throughput than FUSE-Ext4 and SimpleFS. File server workload has a good mix of meta data operations and, large sequential and random reads and writes. By looking at ops/sec for each operation, we noticed that SimpleFS performs comparable or superior to FUSE-Ext4 and FUSE-XFS for meta-data operations such as `open`, `create`, `stat` and `delete` operations. These meta data operations are similar to random database workloads consisting of small tuples. VT-trees can process these operations at a very high speed. File server workload also includes appending 16KB to a random file and also randomly reading the whole file. SimpleFS is $2\times$ slower here because appends causes the data of the file to be spread on disk. Compactions of on-disk lists in GTSSLv1 brings these data together over the life time of the file. If the file is read before a compaction is triggered, it causes the file to be read from multiple places on disk, resulting in lower throughput. The frequency of compaction is determined by the insertion rate and, can also be triggered periodically which would help improve SimpleFS's performance for whole file reads.

6.4 Related Work

We discuss work related to the VT-tree, a component of GTSSLv2. Sections 6.4.1 discusses several categories of widely used tree data-structures for key-value storage on a storage device and makes several important arguments for why LSM-trees are superior for workloads where the number of keys is far larger than what RAM can hold. The efficiency of LSM-trees for this type of workload is a key motivating factor in generalizing the LSM-tree to support sequential insertions. Section 6.4.2 discusses related work for quotient filters and other kinds of write-optimized systems and databases.

6.4.1 Adding Stitching to the Log-structured Merge-tree

SimpleFS is a file system implemented as a FUSE driver on top of a user-level database. The SimpleFS database is based on a transaction manager and journal that maintain a number of log-structured merge-trees. Typically a database maintains B-trees. However, we choose log-structured merge-trees because of their much higher insert and update throughputs, their comparable performance for point queries and scans, and their acceptable performance for small scans and lower bound queries.

We claim that extending the log-structured merge-tree to support stitching is an effective way to support file system workloads, and that it constitutes a novel extension to log-structured merge-tree algorithms. To explain why this claim is true, we categorize background material in this section into three categories:

In-place trees B-trees [23], their derivations and implementations [96, 112, 140], and hash tables [19]

Copy-on-write LFS trees Log-structured file systems [117], copy-on-write log-structured B-trees [42, 67, 92], and their derivations and implementations [51, 139]

Merge trees Log-structured merge-trees [95], and their derivations and implementations [3, 10, 11, 72, 75, 121]

In-place trees In-place trees typically have the best random read performance and extremely competitive serial read performance. However, they must read in whatever blocks they update, and do not perform as well as other data structures for random-write patterns or workloads. In practice, for insert/update-heavy workloads where the majority of operations result in cache misses, in-place trees and hash tables will create an I/O bottleneck if the workload is not already CPU-bound.

Figure 6.15) shows the insertion throughputs of the LSM-tree-based file system LSMFS as the configuration `exp-pc` in comparison to the hash-table–based file system `ext3` and the B-tree–based file systems `xfs` and `reiserfs`. In the benchmark depicted in Figure 6.15, we created 1,000,000 to 16,000,000 files with randomly generated names in the same directory, and then performed 100,000 random lookups within that directory. Directory operations are backed by the tree data-structure (i.e., LSM-tree, hashtable, or B-tree) of the file system. At 4 million inserted items, LSMFS was 62× faster than `ext3`, 150× faster than `reiserfs`, and 188× faster than `xfs`. For 8 million insertions, `reiserfs` was 163× slower than `exp-pc`, and `xfs` was 262× slower than `exp-pc` (LSMFS). Runs of 16 million took more than 20 hours for traditional file systems to complete. The `exp-pc` configuration inserted 16 million random keys in 422 seconds. Afterrward, it performed 100,000 lookups in 676 seconds. The large performance gap is due to the latency difference between performing random insertions into a log-structured merge-tree and random insertions into an in-place B-tree that converts each random insertion into a random write. Other in-place B-tree designs fair similarly.

Compared to a traditional B-tree, a write-optimized MT-SAMT or other LSM sacrafices a small drop in predecessor query performance for a large increase in random-write throughput. In fact, an MT-SAMT or LSM can actually be configured to have identical asymptotic performance to a B-tree for both insertion and lookup, as long as the items are inserted in random order, and the secondary indexes are all resident in RAM or fractional cascading is used [11]. Furthermore,

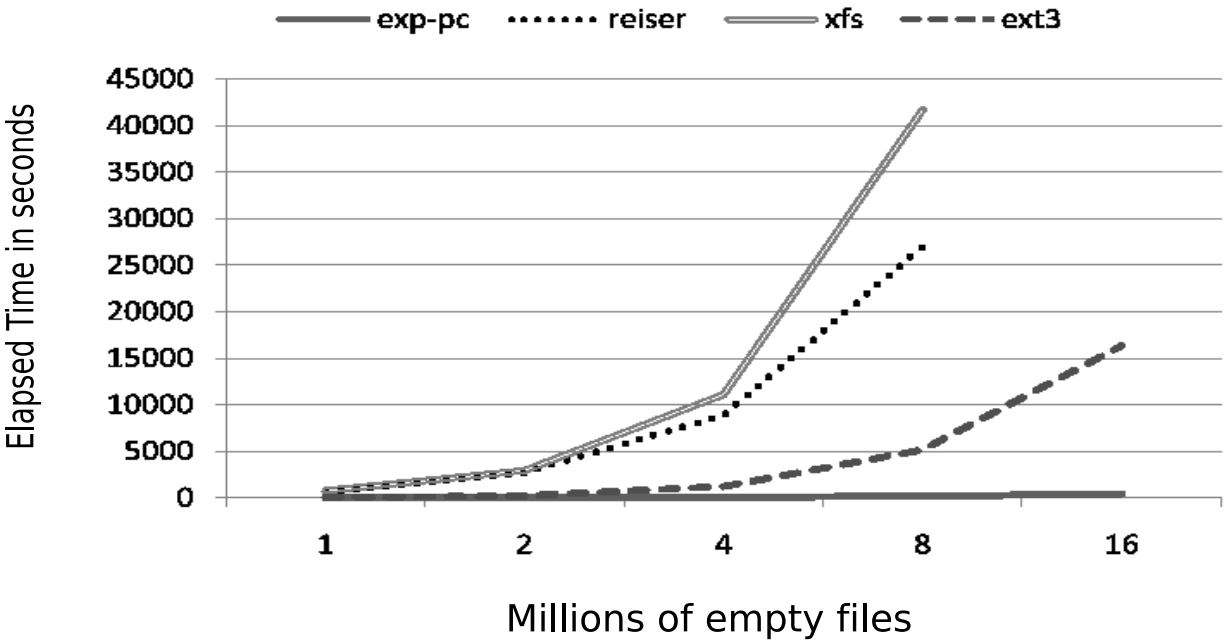

Figure 6.15: Out-of-Cache object creation and lookup.

LSM-trees can dynamically shift between being read- to write-optimized while running. Then, for random workloads, an MT-SAMT or LSM can be thought of as a more dynamic and configurable B-tree. For sequential workloads, a B-tree will probably outperform an MT-SAMT or LSM. This issue is addressed directly with stitching, which we expect will alleviate most of the penalties LSM trees experience when accepting sequential insertions.

Copy-on-write LFS trees Copy-on-write LFS trees append new data, even if that data is being randomly inserted into the data set. However, during append, the tree must acquire other related information to complete the update, if all the keys of the working set cannot be resident in RAM, the acquisition of this key data causes reads in current designs. An example is shown in Figure 6.16.

In Figure 6.16 we see a copy-on-write LFS tree performing an update to a random value. The index entry pointing to the original block location must be updated as the updated value is in a new location (the tail of the journal). The index block must be read before it can be updated and this request incurs a random read request. In the case of a log-structured file system, the amount of RAM required to index all blocks typically fits into RAM and so no index nodes need be faulted in on random updates. On the other hand, copy-on-write LFS trees such as the log-structured B-tree [42] are designed to handle database workloads where the working set could be much larger than RAM, and so these systems cannot expect all keys to fit into RAM.

The copy-on-write LFS database may rely on the Flash storage technology's rapid random read throughput to quickly fault index nodes on random writes. We ran an experiment to compare the performance of an LSM-tree to the performance of a mock copy-on-write LFS tree on both SSD and magnetic disk. The mock copy-on-write LFS tree is like a regular copy-on-write LFS with several major assumptions intended to idealize its performance and simplify its implementation.

We assume that mock-LFS uses all of its RAM during insertion to store parent nodes; normally

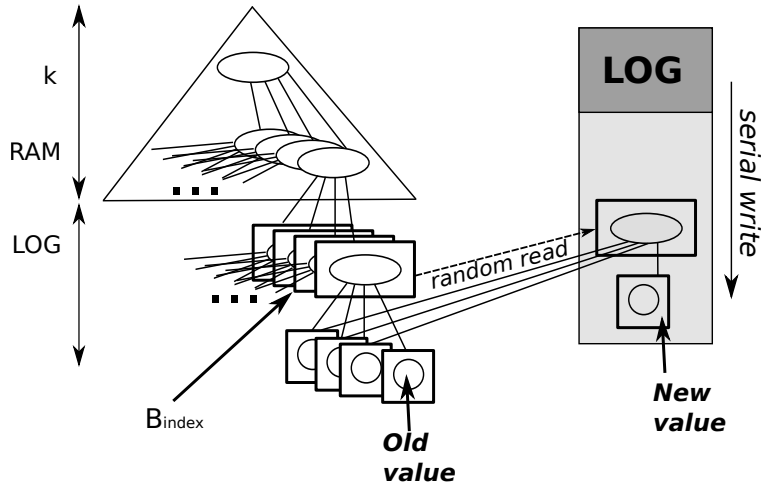

Figure 6.16: ***Illustration of copy-on-write LFS operation:*** *When the keys do not fit into RAM, the parent node immediately above the original value on storage must be faulted into RAM so that one of its pointers can be updated to point to the new value appended to the log.*

on a random insert, $log_2(N)$ nodes from leaf to root would have to be updated to point to newly allocated nodes in the output log. If we assume that mock-LFS caches 2^k parent nodes in RAM, then it need only write $log_2(N) - k$ parent nodes, and the new key-value pair. During insertion, we assume that mock-LFS caches only the left and right pointers, along with the key (no value) in each parent node to maximize cache efficiency. Furthermore, we assume that the cache is always clean. Since we are inserting random keys, then we assume that mock-LFS never needs to rebalance and presume that it does not flush its cache out until all insertions complete.

Copy-on-write LFS trees must be balanced for best performance according to their workload. The parameter that must be balanced is the arity of the tree. When using larger arities, the performance of sequential insertions and random insertions where the set of keys fits into RAM is much slower, but random insertions when the set of keys does *not* fit into RAM can be much faster. The reason for this is that for each random insert, when the keys do not fit into RAM, we would have to copy B_{index} mostly unmodified key-pointer pairs to the log on each write in addition to the updated value itself as depicted in Figure 6.16.

When using smaller arities, the performance of sequential insertions and random insertions where the set of keys fits into RAM is much faster, but random insertions when the set of keys does not fit into RAM can be much slower. The reason smaller arities can slow down random inserts when the set of keys does not fit into RAM is that more than one level of parent nodes may be out of RAM. Therefore, we may have to randomly fault in more than one parent node per random insert.

In our experiment we balanced the mock LFS tree to the best of our ability [13, 145]. We used an arity of 2 so that random insertions would be as fast as possible before the set of keys exceeded RAM in size. After repeated insertions cause the set of keys to be larger than what can fit into RAM, our mock LFS configuration faults in no more than two parent nodes per random write. Therefore, the difference in performance between LSM-trees and mock LFS could at best be halved if a larger arity is used, but the difference in measured performance between LSM-trees and the mock LFS tree is far more than a factor of 2.

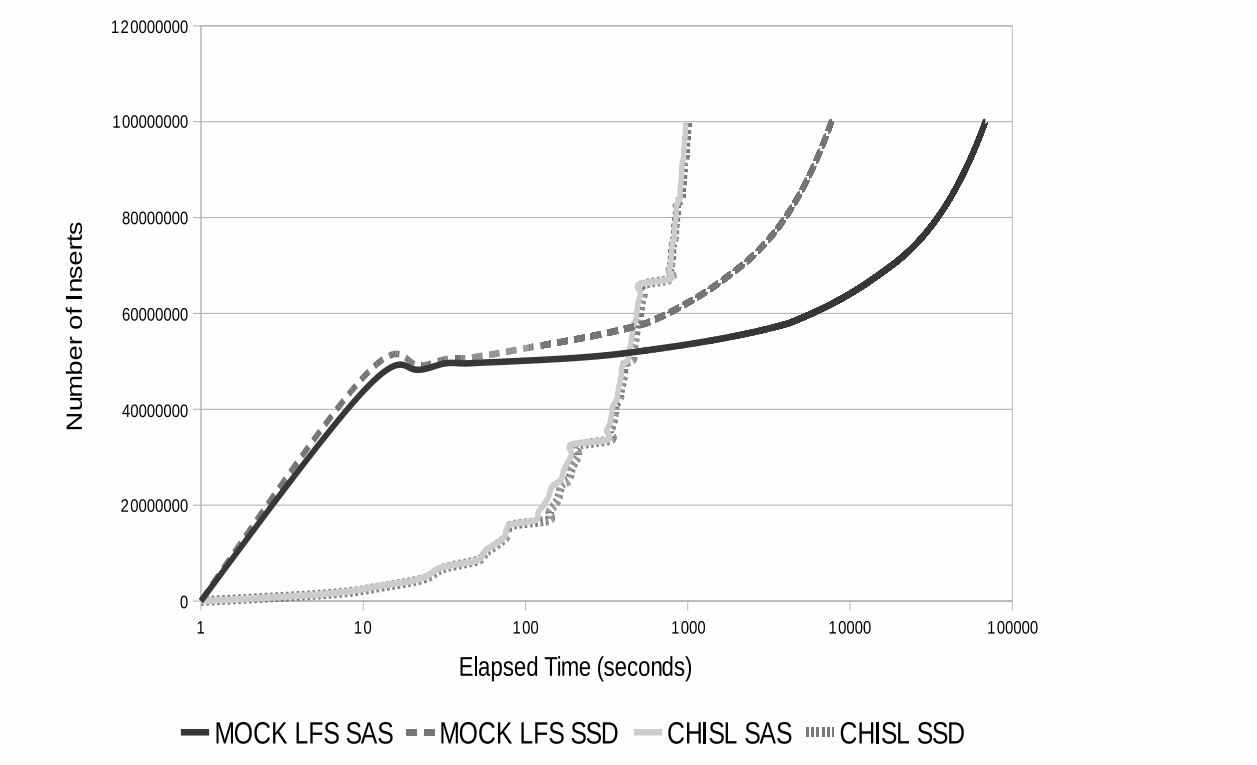

Figure 6.17: ***Copy-on-write LFS performance when thrashing:*** *Once the size of the index exceeds the size of RAM, becoming random read-bound slows LFS insertion to random read throughput.*

We inserted 100 million elements in random order into mock-LFS and the LSM-tree represented as CHISL. For both mock-LFS and CHISL we ran the test on two different configurations: RAM and SSD, and RAM and SAS (magnetic disk). Initially mock-LFS was able to rapidly insert the first 33 million (2^{25}) key-value pairs into its in-RAM cache and sequentially write them to the log. However, once the size of the index exceeded RAM, both mock-lfs-ssd and mock-lfs-sas began performing at least one random read per insertion to find the parent of the node to insert into, so it can append a new node and parent in its output log. The mock-lfs-sas configuration is indeed much slower than the mock-lfs-ssd configuration. After 5,000 seconds, however, even the mock-lfs-ssd was only able to reach an average throughput of 5,407 inserts/sec, averaged from 5,000 seconds until the end of the benchmark. The LSM-tree's insertion throughput on SSD is represented by CHISL-ssd and is 136,000 random inserts/sec in this benchmark. The LSM-tree has comparable throughput for both SSD and SAS. This is because, even though the devices have substantially different random read performance, they have comparable sequential read and write performance: random insertion into an LSM-tree depends only on sequential read and write performance. Twigg independently makes the same argument we make here, and runs a similar simulation (written in OCAML [56]) with similar results.

Merge trees Log-structured merge-trees can be used to store key-value pairs at high insertion throughputs. Because of their high insertion and update throughputs, they form the basic building block of contemporary cluster database designs as explained in Section 5.1. We introduce and comprehensively explain LSM-trees in Section 3.2. Section 5.2 described the COLA [11], our formalized SAMT analysis of the compaction algorithm used by Cassandra [66], Section 5.3 described the multi-tier SAMT (MT-SAMT) which we started with as the basis of the generalized LSM-tree, the VT-tree, which we discuss in Chapter 6.

Merge trees never perform a random read when doing an insert or complete update of a data item. This is possible because data items are written to storage in large sorted lists that are periodically compacted using a minor compaction which issues only sequential reads and never faults in pages randomly. The merge tree can then perform queries by creating a view on multiple sorted lists. Essentially, randomly inserted items are never allowed to stray too far apart. Therefore, a complex indexing framework to track these items is not required, and need not be updated on append as with the case of a copy-on-write LFS tree.

For typical configurations, Merge trees typically perform random reads more slowly than either In-place or copy-on-write LFS trees, and typically perform scans more slowly than In-place trees. However, if queries are typically either point queries or scans that are always larger than several dozen blocks, Bloom filters [15] can be used with great effect to avoid most overheads in lookup, while maintaining an order of magnitude or more increase in insertions, updates, and deletes. In the worst case scenario, a merge tree will perform a short scan approximately $10\times$ slower than an in-place tree or a copy-on-write LFS tree.

In Table 6.3, we list tree data structures that are well suited for storing data on disk or Flash SSD. All of these data structures are analyzed using the DAM model, which is introduced and discussed in Section 3.2.1 and applied in Section 5.2 on the SAMT and COLA.

	Insert Delete Update	Lower bound small scan	Large scan	Multi-tier
SAMT	$\lceil \frac{\log_K N}{B} \rceil$	$\lceil K \log_K^2 N \rceil$	$\lceil K \log_K^2 N + \frac{A}{B} \rceil$	No
MT-SAMT	$\lceil \frac{\log_K N}{B} \rceil$	$\lceil K \log_K^2 N \rceil$	$\lceil K \log_K^2 N + \frac{A}{B} \rceil$	Yes
VTREE	Depends on Input	$\lceil \log_K^2 N \rceil$	$\lceil \log_K^2 N + \frac{A}{B} \rceil$	Yes
K-COLA/LSM	$\frac{(K-1)\log_K N}{B}$	$\lceil \log_K N \rceil$	$\lceil \log_K N + \frac{A}{B} \rceil$	No
Log-str. FS/B-tree	Depends on Input	$\mathbf{O}(\log_K N)$	Depends on Input	No
B-Tree	Depends on Input	$\lceil \log_B N \rceil$	$\lceil \log_B N + \frac{A}{B} \rceil + S$	No

Table 6.3: ***Comparison of main storage data structures:*** *We evaluated each framework according to our four criteria. Each row represents a framework, and each column represents a criterion. The VT-tree (SAMT+stitching) performs A insertions either in $\frac{A}{B}$ if sequential and batched, or $A\frac{\log_K A}{B}$ if not sequential or not batched. LFS inserts are $\frac{A}{B}$ if previous insertions have been sequential and batched, otherwise LFS inserts of A elements are A due to forced random reads. Insertion into a B-tree is $\frac{A}{B}$ if insertions are batched and sorted, or otherwise are $\log_B N$.*

6.4.2 Quotient Filters and other Write-optimized Approaches

(1) Quotient Filters with LSM-trees Quotient Filters and their related work are discussed more thoroughly in our companion paper [12]. We discovered quotient filters, and we found a first reference to a similar structure in a paper by Cleary in 1984 [22]. In the Cleary paper, 5 bits are used instead of 3, and more importantly, applications of the unique merging and re-hashing capabilities of QFs are not discussed. Due to the inefficient handling of duplicates in QFs, QFs are most useful when they are merged within the context of a log-structured merge-tree, as this allows ample opportunity to remove duplicates and apply deletes. Our application of QFs to LSM-trees is to the best of our knowledge unique.

Existing LSM-tree algorithms, designs, and implementations [20, 28, 31] do not address the problem of re-hashing keys into a larger Bloom filter during Wanna-B-tree creation. Existing implementations simply scan the keys while merging, which we have shown introduces unacceptable IO overheads, and makes the stitching optimization far less useful.

(2) Write-Optimized Databases: Work related to GTSSLv1 is also related to GTSSLv2, we treat with many other variants of write-optimized databases there as well, in Section 5.5. We summarize some important log-structured database approaches in the following text.

BDB Java Edition (BDBJE) [30] is a log-based, no-overwrite transactional storage engine. In BDBJE, each key-value pair gets stored in a separate leaf node of a B-Tree. It uses a log to store the dirty key-value pair. It is unclear from the white paper [30] how BDBJE handles completely random insertions while keeping its number of internal nodes (INs and BINs) bounded within RAM without re-balancing, and consequently, randomly reading its leaf nodes, or performing a minor compaction (similar to that of LSM-trees) to re-integrate newly created leaf node entries with existing leaf nodes. The paper does not provide details on this reason for a kind of compaction, what compaction would be used, or how BDBJE operates under this access pattern. It is, however, stated in the BDBJE FAQ [98] that if all keys (contained within BINs) are not resident in RAM, then every operation in a completely random write pattern requires faulting in a likely evicted BIN.

This causes random reads on almost every insertion or other random write. This effect is not unique to BDBJE, but is a side-effect of all log-structured B-trees or red-black trees [146]. We emulated this effect with a highly idealized stochastic benchmark in Section 6.4.1 where we show that even if compaction overhead is not counted, just randomly reading keys on each random insertion will be far costlier than using an LSM-tree approach for large working set sizes, even on Flash SSD devices.

On the other hand, it is clear how LSM-trees achieve optimal insertion throughput for the lookup throughput they provide, and minor compactions are predictable in length and exact time of occurring, regardless of workload or access pattern. Bounds on amortized insertion time are guaranteed for both the COLA [11, 28] and SAMT [31, 136] variations on the LSM-tree. De-amortization can increase predictability of latency without sacrificing insertion bounds [11].

(3) Relation to LFS Threading and other Cleaning Cleaning in the log-structured file system is a difficult problem that ties together out-of-space two concerns: the efficiency of various compaction algorithms, and when to schedule cleaning. In our work we introduce a third concern: efficient storage and retrieval of tuples when a randomly written-to working set's keys cannot fit within RAM. This is an important workload for indexing systems such as BigTable [20], deduplication systems [153], and any workload where write operations do not depend on the result of a random read [11, 136].

In Section 3.2, we describe two forms of compaction: minor compaction which is performed regularly to ensure the LSM-tree remains efficient for querying; and major compaction which is performed periodically and can be likened to compaction in an LFS. GTSSLv1 treats major compactions as a minor compaction that involves all lists or SSTables. In either case, during a minor compaction, we avoid the overheads typically associated with LSM-trees while performing sequential workloads, by utilizing stitching.

The concept of stitching is reminiscent of *threading* in log-structured file systems [115]. A log-structured file system can avoid copying old data blocks to the beginning of a newly compacted log by leaving them in place and allocating from the blocks in-between. The primary difference between stitching and threading is that stitching occurs when merging two or more SSTables in an LSM-tree. There are many tuples in each block pointed to by a secondary index entry in a stitching LSM-tree or VT-tree. As shown in Figure 6.3, some blocks of tuples *must* be copied into a new larger pair of blocks in order to bound the total number of secondary index entries. Further complications arise when performing deletes. As discussed in more detail in Chapter 5, deletes are performed during a major compaction. When stitching, deletes that are omitted reduce the size of the resulting block as it contains fewer tuples. Therefore, blocks that must have their deletes removed are copied during a merge, to bound the total number of secondary index entries. Bounding the number of secondary index entries is vital to maintaining the LSM-tree's good insertion throughput.

Log-structured file systems can rely on storing meta-data required for efficient retrieval of data within RAM. Indeed this increase in size of RAM for caching is a precept of Rosenblum and Ousterhout's paper [115]. Conversely, LSM-trees expect to hold $\mathbf{O}(\log) N$ of their keys in RAM. It is not obvious how a log-structured or copy-on-write B-tree would keep the size of its secondary index bounded in the face of completely random writes without performing random reads, or performing some form of merging similar to how LSM-trees operate.

The copy-on-write approach (used by WAFL [51] and ZFS [16]) is shown by Twigg et al. to

be as slow as the storage device's random read throughput when performing random insertions and the index of keys no longer resides in RAM [146]. In the DAM model, each random write would cost $\mathbf{O}(1)$ for a copy-on-write LFS tree. Conversely, and as shown in Chapter 5, a random write to a VT-tree would cost $\frac{\lceil \log_K N \rceil}{B_{small}}$. We also provide an explanation and idealized experimental analysis of this effect in Section 6.4.

Since the LSM-tree packs new random insertions into newly allocated zones (or clusters in LFS terminology), it makes full use of the storage device's sequential bandwidth. It is the LSM-tree's repeated rounds of merging that allow an LSM-tree to bound the total number of lists and size of its secondary indexes when performing random writes. Copy-on-write LFS trees bound the size of their secondary indexes by maintaining keys in sorted order and faulting in blocks of keys as needed. This has to happen all the time when performing random writes and the set of keys does not fit into RAM. In this way, an LSM-tree is different from a log-structured or copy-on-write B-tree and it can better support random write workloads [11,20,145]. We have extended the LSM-tree to support more sequential workloads by understanding how to perform stitching, as introduced in Chapter 6.

Stitching alone is not sufficient to maintain good point query performance. LSM-trees use Bloom filters to avoid performing I/Os while searching through SSTables that do not contain the key. This way they can usually spend just one I/O searching through the list that does have the key, or no I/Os if the key does not exist. However, an LSM-tree that uses a Bloom filter would still have to scan every block—even those it stitches—to recover the keys within each block (that could not fit in RAM) to re-hash them into a new Bloom filter. It was this fundamental limitation of Bloom filters that led us to a joint research on cascading quotient filters (QF) [12]. Using QFs, we can reconstruct the QF belonging to an SSTable resulting from a merge of other SSTables without having to scan any blocks. To obtain the QF of an SSTable resulting from a merge, we simply merge the two QFs from the SSTables being merged together—that are already in RAM—to create the QF for the resulting SSTable. This merging capability of QFs and the concept of merging them alongside the regular compactions of the LSM-tree, allows us to continue to avoid I/Os when stitching, even when re-hashing filters to maintain good point query performance.

These difficulties, along with determining when blocks can and cannot be stitched due to updates, deletes, or interleaving are not addressed in Rosenblum or Ousterhout's work. In their work, Bloom filters were not required, primarily because the log-structured file-system maintained all keys in RAM and merging was not required to bound the number of lists or size of secondary indexes.

Alternative LFS Cleaning Approaches In addition to the Sprite LFS cleaning approaches such as threading, there are several other approaches for reducing the cost of compaction in an LFS. Matthews et al. [76] utilize a graph to maintain a cache of recent read access patterns. This graph is consulted when cleaning to decide which blocks to place together when moving them out of fragmented segments and into new compacted segments. Additionally, Matthews et al. explore the application of dynamically switching cleaning methods from a standard cleaner to a "hole-plugging" cleaner that utilizes free (but fragmented) space within existing segments to perform allocations.

Other approaches to reducing the impact of compaction focus on identifying idle times when running background compaction. These have a minimal impact on important workloads executing

on the storage system [14].

Optimizing on-disk layout based on recent popular access patterns and identifying idle times to run cleaning with minimal impact are important lessons to any log-structured system that wants to minimize the impact of compaction on file system performance. LSM-tree based file systems and storage systems can capitalize on these techniques by scheduling large minor or major compactions to occur while the system is idle. Our generalized stitching LSM-tree discussed in Chapter 6 could be modified to physically place groups of tuples frequently accessed together based on expected read access patterns even while secondary index entries pointing to these tuples remain in sorted order. Making such modifications to exploiting these kinds of techniques in the context of LSM-tree based storage systems is a topic of future work.

6.5 Conclusions

We found that stitching configurations perform at least as well as traditional LSM-trees. However, when there was any sequentially (e.g., random appends vs. random insertions), stitching configurations performed upwards of 2.5× better than the standard LSM-tree algorithm which performed copies on every merge. In addition to being able to trade off locality for insertion throughput for random appends, sequential insertions are 3.1× faster with stitching with no loss in scan performance. For CPU-bound workloads, such as inserting 64B tuples, stitching permits workloads with some sequentiality to perform 2× faster than under a traditional LSM-tree. Our prototype SimpleFS system benchmark shows that stitching LSM-trees can perform comparably to existing native kernel-level file systems within the overheads imposed by the FUSE framework.

The culmination of lessons learned in previous trial designs discussed in Chapters 4–5 is that the choice of data structure, and the decision to host the database and transaction manager at the user-level, are critical to efficiently and simply supporting the desired abstractions of system transactions and key-value storage for system processes.

Our main goal was to build a simple storage system that supports both file system and database workloads and, provides efficient system transactions to applications. With our novel extension to LSM-tree, GTSSLv1 can perform efficiently for highly sequential workloads (file-based data access), as well as highly random workloads (structured data access, databases), and anything in between. We showed that our sequential optimization, stitching, is a must feature in LSM-trees in order to support sequential and file system workloads. We further showed that stitching can only avoid read IOs if quotient filters are employed to permit merging of filters entirely within RAM.

Table 6.4 now lists GTSSLv2 in comparison to other storage systems in terms of transactional design decisions. GTSSLv2, like GTSSLv1, can switch between concurrent small durable transactions and larger or asynchronous transactions in the same workload; spending either one or two writes when necessary. GTSSLv2 is log-structured and supports value logging for generic ACID transactions. Unlike GTSSLv1, GTSSLv2's current implementation has been extended to support larger-than-RAM transactions. Details as to how this is accomplished is part of joint research with Pradeep Shetty and is discussed in future work in Chapter 8. GTSSLv2 uses `mmap` for page caching, and `write` with `msync` and `MS_INVALIDATE` for efficient sequential writes. Unlike GTSSLv1, GTSSLv2 is able to perform sequential writes at 90MB/sec or 81% of the disks sequential write throughput, and is CPU-bound meaning that its data structure is not the bottle-neck, but implementation efficiency. For this reason, GTSSLv2 is able to perform efficient sequential writes.

	Type	Num Writes	Log-Struct.	Trans-actions	Conc-urrent	Async	Write Order	Random	Stitch-ing	Sequen-tial
Ext3	FS	1	¬	MD-only	¬	✓	Kernel	R	¬	R,W
SchemaFS*	FS	3	¬	Logical	✓	✓	User	R	¬	R
Valor	FS	2	¬	POSIX	¬	✓	Kernel	R	¬	R
LSMFS	FS	1	✓	MD-only	¬	✓	`mmap`	S,W	¬	R,W
Cassandra	KVS	2	✓	Single	✓	✓	`mmap`	P,S,W	¬	R
GTSSLv1	KVS	1–2	✓	Vals	✓	✓	`mmap`	P,S,W	¬	R
GTSSLv2	KVS	1–2	✓	Vals	✓	✓	`mmap`	P,S,W	✓	R,W

Table 6.4: ***A qualitative comparison of Transactional Storage Designs:*** *VT-trees permit sufficient flexibility to process both random and sequential access workloads. The log-structured nature of VT-trees makes them easy to integrate into a log-structured architecture.*

Chapter 7

Conclusions

In this thesis we described several designs, and focused on two in specific: (1) a transactional extension to the operating system support layer for file systems, and (2) flexible, scalable, and transactional, extensions to a widely used database design intended for high-throughput insertions, updates, and deletes. We have learned lessons along the way about how a consolidated storage system that supports transactional key-value storage for small to large tuples and files might look. These are enumerated and explained in Section 7.1.

We finally summarize the conclusions of this thesis in Section 7.2.

7.1 List of Lessons Learned

Once Written, Twice Nearby In Chapter 4 we discussed in detail the design, implementation, and evaluation of a transactional VFS layer—called Valor—designed to allow any file system to support application level system transactions. Like Stasis [120] and Berkeley DB [130], we chose to use undo and redo records. We decided to closely follow a traditional ARIES [85] journaling architecture.

This design allowed us to provide transactions to lower file systems, even without modifying them. However, we suffered significant performance overheads. We found that these overheads were not unique to the Valor implementation, but similar overheads or worse could be observed by running the same workloads on other transactional storage and database libraries such as Stasis and Berkeley DB.

We concluded that if we want a modular solution that will work with most underlying file systems, we will have to pay for at least two writes (a redo record and the actual write), and possibly additional reads and processing depending on what restrictions we place on transaction sizes. After Valor, we decided to pursue a more specialized solution that would perform fewer writes by customizing both the transactional design and the underlying data structures used.

Journaling: A Special Case An WAL transactional system typically supports small and large durable transactions. Berkeley DB and Valor also support asynchronous transactions or non-

durable (but still atomic) transactions. There are some important optimizations typically used by these systems, such as group commit, which make the WAL architecture well-suited to applications which run small durable transactions, but may have to run transactions larger than RAM. However, if some transactional features were omitted, the transactional system could have been designed differently to allow enhanced performance for specific workloads.

The Cedar file system [39] introduces the concept of journaling in combination with a fast file system design to maintain the file system's meta data consistency after a crash or other failure. Cedar is an example of a file system which does not use the standard ARIES variant of WAL. Like Cedar, Ext3 [19] uses redo-only logging and in its default `ordered` mode always writes data blocks before meta-data blocks. Ext3 never needs to roll back a transaction because it will never partially commit a transaction. Partial transaction commit typically occurs because of memory pressure. However, all transactions in Ext3 are very small, and are composed of `inode` updates which can be arbitrarily separated into multiple transactions if necessary. Since Ext3 also does not allow applications to transactionally write to multiple data blocks, there is no source of memory pressure that would force Ext3 to partially commit a transaction.

Ext3 also exploits the notion that it must only provide atomic updates of `inode` meta-data, and not even guarantee durability. This allows Ext3 to group together multiple log writes for even single-threaded workloads, and to saturate a commodity storage device's read and write throughput with minimal overhead from journaling [122].

We learned by designing Valor, and by studying other journaled and log-structured transaction systems [16, 51, 114, 125], that one size does not fit all [138], and that by understanding what transactional semantics are desired, significant overheads like those present in Valor and other WAL-based systems can be avoided, much like how Ext3 is able to perform closely to or surpass its non-journaled predecessor, Ext2. Through the course of our research we observed however, that some of the most compelling reasons to support system transactions require transactions much larger than RAM. One commonly cited example is a system-wide package installation or upgrade [106, 135, 151]. Similarly, many smaller transactions that require durability can benefit from a more ARIES-like approach.

We discovered that if the storage system is *log-structured*, it can naturally support both durable, asynchronous, large, and small transactions in the same workload. By log-structured, we mean that the storage system can refer to its previously flushed transactional logs to perform queries. A log-structured system with the right transactional architecture can be both ARIES-like and Ext3-like when necessary. Implementation details for this type of transactional architecture and its integration with a log-structured merge-tree were discussed in Chapter 5.

Keep your Caches Close, Keep your Shared Caches Closer Another advantage to Ext3's limited transactional architecture is its easier implementation in the kernel. Conversely, we have found that porting large sophisticated storage systems into the kernel can be extremely cumbersome and difficult [59, 151]. So one major influence in our storage research has been taking into account implementation difficulty and complexity, and engineering solutions that may not have to completely reside in the kernel. When designing Valor, we followed this lesson by not putting all transactional functionality into the kernel.

Because of the complexity of these more featureful storage systems, when implementing a transactional storage system for file system and database workloads, we did so in user-space. In past related research we either utilized `ptrace` primitives, or more recently used FUSE. We have con-

ducted several research projects that were based on or discussed FUSE including Story Book [134] and LSMFS [133]. We found that FUSE can inhibit application performance dramatically when every VFS request it receives from a system call must be sent to the user-level server daemon. We explored two different ways of avoiding this round-trip-time: (1) by not using FUSE and instead finding another way to secure inter-process storage cache sharing, and (2) by finding a way to overcome cache-incoherency issues that arise when enabling FUSE's in-kernel caches.

In Section 3.3.2, we discussed an alternative we explored [133] to using FUSE for a user-level file system. Essentially we enabled applications to directly access a shared memory cache instead of using the NFS-like relay mechanism employed by FUSE. This allowed our workloads to perform as efficiently as kernel-level file systems for in-cache workloads, but required a separate VFS implementation, a system call trampoline mechanism. Furthermore, we had to map large shared user-level file system caches into each process' address space. Although this did not threaten security because we required a kernel context-switch to access these address ranges, it increased the complexity of the kernel and required per-thread stack maintenance in each address space.

We found that if we could leave FUSE's read-caching enabled, it performed comparably well to native file systems for read-only benchmarks. Unfortunately, when read caching is enabled, many VFS read-related requests will never be forwarded to the user-level server or daemon. For some developers of FUSE-based file systems, not receiving every read request is not an option. For example, a file system that logs every read access for security monitoring reasons will not meet its specifications while FUSE read caching is enabled. FUSE also does not support write-back caching. We have found in experimentation that FUSE can benefit from batching together multiple write requests.

If in-RAM or in-cache performance is important, an efficient user-level implementation will work as long as the caches accessed by applications can be immediately accessed using shared memory, either within the kernel address range, or some other protected address range. FUSE remains a promising path toward more simply shared inter-application caches, but currently it does not support a write-back cache. Most importantly, developers of FUSE-based file systems must be able to work around not receiving every read request.

Get to the Point In our work on LSMFS [133], we measured the file creation throughput of LSMFS against several in-kernel file systems. These results were discussed in the "User-Space LSMFS" vignette, Section 3.3.2. At the time that research was conducted we developed a file system based on a database schema not too unlike that used in SchemaFS from Section 3.3.1. Since page ranges were at least one page large, the secondary indexes of these tables could be expected to fit into RAM. Other meta-data operations such as inode and path lookup, file creation, existence checking, and read and write operations can also be performed with strictly point queries or in-RAM index queries. Listing directories requires placing a database cursor, and incrementing it multiple times, which is generally more expensive operation than a single point query.

Zhu, Li, and Patterson [154] found that you can actually perform unique creations rapidly by using a write-optimized index and a Bloom filter. Newly created unique items will be falsely reported as already existing by the Bloom filter $K\%$ of the time, where K is the false positive rate. All the others can be efficiently inserted into the write-optimized index. We discovered that with LSMFS, where we use an LSM-tree as our write-optimized index for `inodes` and `dentries`, we could create files orders of magnitude faster than existing file systems (this is discussed in detail in the vignette below). Later we discovered it was possible to perform point queries in an

LSM-tree at the disk's random read throughput (discussed in Chapter 5). The lesson here was that many common meta-data operations can be performed orders of magnitude faster than existing file systems without violating POSIX semantics, or using asynchronous error reporting buffers.

Log-structured and `mmap` When developing LSMFS and later GTSSLv1 we learned that new data writes do not overwrite existing pages. On the other hand, writes to the journal must be held until related data writes are flushed. For example, a journal entry may point to a list of flushed tuples, although the list can be flushed at any time, the journal entry pointing to it cannot be flushed until the list is entirely on the storage device. This implies that log-structured systems can place their data caches in a large `mmap`, and must only buffer log writes, that are sequential and for which a simpler buffering implementation is obtained. This allows these systems to benefit from a zero-copy page cache managed by the kernel, and the kernel is free to flush this cache when necessary.

We found upon analyzing Cassandra's source code that this optimization is already in use there. However, although we used `mmap` in LSMFS for page caching, we incurred additional overheads on write operations. When writing to an `mmap`ed region, the kernel does not know if you will only partially write, and then read the remainder of a single page, and so it faults-in whole pages, even if they will be over-written by the process. GTSSLv1 avoided this by using `write` and `msync` with the `MS_INVALIDATE` flag to invalidate `mmap`ed regions after calling `write`.

Sequential and Multi-tier LSM-trees: a promising avenue for file system design In addition to a flexible transactional architecture that allows for high-throughput asynchronous updates and reads and group commit of small durable transactions, the data structure underlying the storage system should exhibit certain properties.

First, we have found that a log-structured data structure permits the greatest flexibility in allowing transactions both larger and smaller than RAM, as well as durable and asynchronous transactions.

Second, we have found that supporting both random index updates and insertions, as well as maintaining a high scan throughput, can be achieved with a log-structured merge tree (LSM). This data structure is widely used in NoSQL type databases, but the data structure itself can be used with any transactional paradigm. One of the more popular variants of the LSM-tree is the COLA [11] which is used by HBase [28] for compaction. Within the disk-access model (DAM model), it is not possible to find another data structure that inserts as quickly as the COLA, but can query any faster, or vice verse [18]. It should be noted that it is also not possible to find a data structure which can randomly insert *and* query faster than the B-tree [18]. The LSM-tree trades-off query time for increased insertion throughput. In practice, LSM-tree queries can be sped up dramatically with the aid of Bloom filters and appropriate caching. Since LSM-trees are log-structured, we can support both ARIES (database) and Ext3-style (file system) workloads as summarized in Section 3.4 and detailed in Chapter 5.

Third, we have found with LSMFS and as further discussed in our vignette below, that supporting larger tuple insertions, or sequential tuple workloads is vitally important for many common file system operations and workloads. As we will discuss in detail in Chapter 6, LSM-trees are inefficient at performing these kinds of operations. Our generalized LSM-tree can efficiently perform sequential file system-like workloads, and those results are also discussed in Chapter 6.

We showed in Chapter 6 that our generalized LSM-tree can serve as a new basic underlying file system data structure by supporting both random and sequential workloads, as well as a file system file server workload, and we hope that the contents of this thesis will help guide future research in studying the application of LSM-trees to scalable, transactional file system design.

7.2 Summary

Searching for a consolidated, transactional, and workload-flexible storage system involves optimizing the data structures, design, and implementation to best meet those goals. Trying to retrofit existing file system designs to support new abstractions and paradigms is a low-hanging fruit for researchers, but ultimately leads to performance issues, scalability issues, and most importantly awkward, or limited abstractions. We began with an idea of what we wanted, and then after attempting to make it fit in as a component of existing systems, and suffering from the requisite performance and engineering difficulties, we ultimately realized a much better solution was to start again, with lessons learned from our research to inform a design which would still be efficient, and offer both system transactions and a key-value storage interface that would perform well for a range of workloads from structured to sequential data.

Research in storage systems must pay closer attention to the underlying data structures that are relied upon. A thorough understanding of the theoretical limitations of underlying data structures from the perspective of a limited RAM and a less limited storage device can help to realize a storage system that is more flexible for a variety of workloads, and is more future-proof even as workloads and demands on storage change. Such a system if designed from the ground up to support widely used and highly desirable abstractions, such as transactions and key-value storage, can be simpler, more efficient, and more scalable overall.

Chapter 8

Future Work

Beyond what is described and shown in Chapters 4–6, there are many interesting avenues for this research. At the heart of our design is the idea of linking together many trees of the same schema to provide snapshots, transactions, stitching, and more. This idea can be extended to network scenarios, and the algorithms employed can be extended for multiple storage tiers. We believe the abstraction of merging many trees together is one that will make powerful transactional and key-value storage semantics not only possible on a single node, but on a network of nodes. We discuss some of these ideas below.

8.1 External Cache Management

As operating systems become increasingly more complex, they must distribute operating components into modules, and rely on message passing between these modules to simplify the architecture as well as to better exploit machines with an increasing number of cores, NUMA regions, and multiple buses. File systems can no longer be developed as a main-memory data-structure that periodically flushes its contents to a storage device. The file system must assume that multiple client caches will exist that will be read and updated independently of each other. GTSSL already supports this workload where the client cache corresponds 1-to-1 to a transaction. It would be beneficial to explore tying these independent transaction caches with affinity to a particular core or memory region.

8.2 Opt-in Distributed Transactions with Asynchronous Resolution

In a larger context, the algorithms and techniques within this thesis could be extended to multiple nodes where each client maintains its own transactional state. Commits would happen periodically as map-reduce processes that perform a merge of all committed lists, and notify clients when a transaction is rejected and must be restarted. Rejections could be avoided by reserving ranges of

the key-space with the commit process so that rejection favors those who reserved the range in the key-space in which the conflict occurs. This requires that transactions that want to guarantee commit of transactions do so by contacting a lock-tree (which can also be distributed in a more task-specific manner). The transactions that wish to forgo this guarantee do not suffer performance loss. Thus, guarantee of transaction commit is opt-in.

8.3 Multi-tier Deployments for Client-side Compaction

Currently database clusters are distributed across many machines, but the clients are expected to do little or not partake in the operation of the database. One possible avenue of future work is to have clients compact insertions locally into lists, and then rather than sending each request across the Internet, send large compacted lists of insertions across the network. As the transactional model used allows applications to update their client-side independently, queries on data created by the client can be answered locally by the client itself. If a lock reservation system is used, the client need not worry about abort of its transaction if it contacts the lock management system maintained by the server before performing updates and modifications to the key-spaces it intends to modify or read.

The view manager, the module responsible in SBFS for unioning multiple read-only snapshots into a single consistent view for a transaction, and for maintaining pessimistic locks, can naturally be shelled out to a separate server. In this way multiple nodes could define a single view manager that would maintain locks across ranges in the keyspace of various tables, and tell nodes where various snapshots are located so that tables distributed across multiple machines can be viewed as a single table. We view this as an elegant and compelling architecture to extend SBFS DB to a multi-tier deployment where different tiers are on different machines connected by a network.

8.4 Multi-tier Range Query Histograms

In joint work with Bender et al. [12], we showed how a Bloom filter could be replaced with another data-structure suitable for use outside of RAM, the quotient filter. We also showed in this work how it is necessary to support stitching alongside efficient filtering. We now explain the potential benefit of a multi-tier range query histogram, a sort of "Bloom filter" for range queries, which could be extended to an out-of-RAM context as well, allowing it to be incorporated into a multi-tier regime as well. The basic idea is to use an X-fast (or Y-fast) trie [150] to separate keys into roughly uniformly distributed clusters, whose most-significant bits are then stored in cascade filters to maintain a compact representation. Insertion into this data-structure would probably be faster than into a single cascade filter, but at the cost of added complexity to the architecture. Furthermore, such a system would be incapable of obviating range queries on ranges where the start and end of the range have the same most-significant bits as a key inserted into the VT-tree.

8.5 Multi-node Deployment

In our Chisl work [136] we explored the performance of a tablet server storage layer on a single node, and found potential causes of slowness, and identified definite bottlenecks, while providing

architectural alternatives for transaction management and multi-tier support. Benchmarking these extensions to verify that efficiency is maintained in the distributed case is a natural extension of experimentation on this project.

8.6 B-tree-like Out-of-space management

Typically when an LSM-tree nears the capacity of the storage device, it begins to perform in-place merging, and non-optimal in-place merging. Essentially, there is enough space for only the lowest level, and it consumes most of the free space. RAM merges in place with this massive sequence on every flush. Cassandra simply requires half the disk be free. Flash drives handle out-of-space scenarios more gracefully, but still performance can drop by a factor of $5\times$ when performing random insertions on an Intel X-25 Mv2 that is 90% full compared to 55% full. In the DAM model, if the RAM buffer is B times smaller than the massive single list on disk, an in-place merge is equivalent to randomly writing each el in the RAM buffer. However, if the els are sequential, and could be written in $\frac{A}{B}$ time for A els, this is a huge penalty.

When a SAMT+stitching tree (VT-tree) runs out of space, a naive approach is to perform inefficient in-place merges as described above, where very small trees are merged into very large trees on disk. By utilizing stitching, we intend to explore if the merge can be performed in name only. This means that we perform a stitching merge, and only modify in-place blocks that need actually be modified. If the changes are not randomly spread across the tree, we expect the vast majority of the merge will be effectively free of charge as existing elements will merely be stitched back in place. However, because space is tight, the resulting patch-tree will need to be immediately cleaned, if not during the merge itself. This will result in random writes as the inserted els will not have time to coalesce with future els into larger blocks. In this way, the VT-tree begins to perform random writes similar to a B-tree when it is low on space. This is an interesting aspect of the research, and may demonstrate a linkage between B-trees, LSM-trees, and an LFS.

8.7 GTSSLv3: Independent SAMT Partitions for Partitionable Workloads and Multi-Tiering with Stitching

Work on GTSSLv2 consisted primarily of extending LSM-trees to support stitching, and then evaluating the performance benefit of this extension. We also are currently undergoing work on extending the transactional architecture of GTSSLv2 to support multiple VT-trees that can be used in parallel, and then placed into a single VT-tree which is then asynchronously re-shaped using merging compactions. This is joint work with Pradeep Shetty.

By permitting multiple VT-trees to be used, we can begin transactions by creating a new VT-tree, and maintain locks in a separate lock-tree with deadlock detection. When a transaction commits, we simply place the patch-trees of the committing VT-tree into the most recent VT-tree shared by non-transactional applications. After the commit, we create a new common VT-tree that non-transactional applications commit into.

This form of "snap-shotting as transactions" also solves several incoherency problems with how FUSE limits use of its read cache, and also can potentially parallelize partionable workloads in a natural and transparent way.

Bibliography

[1] M. AB. MySQL Reference Manual, March 2007. `www.mysql.com/doc/en/index.html`.

[2] Azza Abouzeid, Kamil Bajda-Pawlikowski, Daniel Abadi, Avi Silberschatz, and Alexander Rasin. Hadoopdb: an architectural hybrid of mapreduce and dbms technologies for analytical workloads. *Proc. VLDB Endow.*, 2:922–933, August 2009.

[3] D. Agrawal, D. Ganesan, R. Sitaraman, Y. Diao, and S. Singh. Lazy-adaptive tree: an optimized index structure for flash devices. *Proc. VLDB Endow.*, 2(1):361–372, 2009.

[4] D. G. Andersen, J. Franklin, M. Kaminsky, A. Phanishayee, L. Tan, and V. Vasudevan. FAWN: A Fast Array of Wimpy Nodes. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles (SOSP '2009)*, pages 1–14. ACM SIGOPS, October 2009.

[5] Eric Anderson and Joseph Tucek. Efficiency matters! *SIGOPS Oper. Syst. Rev.*, 44:40–45, March 2010.

[6] The Apache Foundation. Hadoop, January 2010. `http://hadoop.apache.org`.

[7] Apple Computer, Inc. HFS Plus Volume Format, March 2004. `http://developer.apple.com/technotes/tn/tn1150.html`.

[8] Apple, Inc. *Mac OS X Reference Library*, 2009.

[9] A. Appleby. Murmur Hash, January 2012. `http://sites.google.com/site/murmurhash`.

[10] L. Arge. The buffer tree: A new technique for optimal i/o-algorithms (extended abstract). In *University of Aarhus*, pages 334–345. Springer-Verlag, 1995.

[11] M. A. Bender, M. Farach-Colton, J. T. Fineman, Y. R. Fogel, B. C. Kuszmaul, and J.Nelson. Cache-oblivious streaming b-trees. In *SPAA '07: Proceedings of the nineteenth annual ACM symposium on Parallel algorithms and architectures*, pages 81–92, New York, NY, USA, 2007. ACM.

[12] M. A. Bender, M. Farach-Colton, R. Johnson, B. C. Kuszmaul, D. Medjedovic, P. Montes, P. Shetty, R. P. Spillane, and E. Zadok. Don't thrash: How to cache your hash on flash. In *HotStorage '11: Proceedings of the 3rd USENIX Workshop on Hot Topics in Storage*, June 2011.

[13] P. A. Bernstein. Scaling Out without Partitioning: A Novel Transactional Record Manager for Shared Raw Flash. In *Proceedings of the HPTS 2009 Workshop*, 2009. `www.hpts.ws/session2/bernstein.pdf`.

[14] Trevor Blackwell, Jeffrey Harris, and Margo Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *Proceedings of the USENIX 1995 Technical Conference Proceedings*, TCON'95, pages 23–23, Berkeley, CA, USA, 1995. USENIX Association.

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

[16] J. Bonwick and B. Moore. ZFS: The Last Word in File Systems. `http://hub.opensolaris.org/bin/view/Community+Group+zfs/docs`, 2010.

[17] D. P. Bovet and M. Cesati. *Understanding the LINUX KERNEL*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebatopol, CA 95472, 2005.

[18] Gerth Stolting Brodal and Rolf Fagerberg. Lower bounds for external memory dictionaries. In *Proceedings of the fourteenth annual ACM-SIAM symposium on Discrete algorithms*, SODA '03, pages 546–554, Philadelphia, PA, USA, 2003. Society for Industrial and Applied Mathematics.

[19] M. Cao, T. Y. Tso, B. Pulavarty, S. Bhattacharya, A. Dilger, and A. Tomas. State of the Art: Where we are with the Ext3 filesystem. In *Proceedings of the Linux Symposium*, Ottawa, ON, Canada, July 2005.

[20] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: a distributed storage system for structured data. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, pages 15–15, Berkeley, CA, USA, 2006. USENIX Association.

[21] B. Chazelle and L. J. Guibas. Fractional cascading: A data structuring technique with geometric applications. In *Proceedings of the 12th Colloquium on Automata, Languages and Programming*, pages 90–100, London, UK, 1985. Springer-Verlag.

[22] J. G. Cleary. Compact hash tables using bidirectional linear probing. *IEEE T. Comput.*, 33(9):828–834, 1984.

[23] D. Comer. The Ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[24] Brian F. Cooper, Adam Silberstein, Erwin Tam, Raghu Ramakrishnan, and Russell Sears. Benchmarking cloud serving systems with ycsb. In *Proceedings of the 1st ACM symposium on Cloud computing*, SoCC '10, pages 143–154, New York, NY, USA, 2010. ACM.

[25] J. Corbet. Solving the Ext3 latency problem. `http://lwn.net/Articles/328363/`.

[26] B. Cornell, P. A. Dinda, and F. E. Bustamante. Wayback: A User-level Versioning File System for Linux. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 19–28, Boston, MA, June 2004. USENIX Association.

[27] Ewa Deelman, Gurmeet Singh, Miron Livny, Bruce Berriman, and John Good. The cost of doing science on the cloud: the montage example. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, SC '08, pages 50:1–50:12, Piscataway, NJ, USA, 2008. IEEE Press.

[28] Bruno Dumon. Visualizing hbase flushes and compaction. http://outerthought.org/blog/465-ot.html, February 2011.

[29] E. D. Demaine. Cache-Oblivious Algorithms and Data Structures, 1999.

[30] BDB Java Edition. Bdb java edition white paper. `www.oracle.com/technetwork/database/berkeleydb/learnmore/bdb-je-architecture-whitepaper-366830.pdfi`, September 2006.

[31] J. Ellis. Re: Worst case iops to read a row, April 2010. `http://cassandra-user-incubator-apache-org.3065146.n2.nabble.com/Worst-case-iops-to-read-a-row-td4874216.html`.

[32] Filebench. `http://filebench.sourceforge.net`.

[33] Christopher Frost, Mike Mammarella, Eddie Kohler, Andrew de los Reyes, Shant Hovsepian, Andrew Matsuoka, and Lei Zhang. Generalized file system dependencies. In *Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, SOSP '07, pages 307–320, New York, NY, USA, 2007. ACM.

[34] FusionIO. IODrive octal datasheet. `www.fusionio.com/data-sheets/iodrive-octal/`.

[35] E. Gal and S. Toledo. A transactional flash file system for microcontrollers. In *Proceedings of the Annual USENIX Technical Conference*, pages 89–104, Anaheim, CA, April 2005. USENIX Association.

[36] G. R. Ganger, M. Kirk McKusick, C. A. N. Soules, and Y. N. Patt. Soft updates: a solution to the metadata update problem in file systems. *ACM Trans. Comput. Syst.*, 18(2):127–153, 2000.

[37] N. H. Gehani, H. V. Jagadish, and W. D. Roome. OdeFS: A File System Interface to an Object-Oriented Database. In *Proceedings of the Twentieth International Conference on Very Large Databases*, pages 249–260, Santiago, Chile, September 1994. Springer-Verlag Heidelberg.

[38] D. Giampaolo. *Practical File System Design with the Be File System*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1998.

[39] D. K. Gifford, R. M. Needham, and M. D. Schroeder. The Cedar File System. *Communications of the ACM*, 31(3):288–298, 1988.

[40] B. S. Gill. On Multi-level Exclusive Caching: Offline Optimality and Why promotions are better than demotions. In *FAST '08: Proccedings of the 6th conference on File and storage technologies*, Berkeley, CA, USA, 2008. USENIX Association.

[41] git. http://git-scm.com.

[42] G. Graefe. Write-optimized b-trees. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, pages 672–683. VLDB Endowment, 2004.

[43] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, San Mateo, CA, 1993.

[44] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: a declarative file system checker. In *Proceedings of the 8th USENIX conference on Operating systems design and implementation*, OSDI'08, pages 131–146, Berkeley, CA, USA, 2008. USENIX Association.

[45] M. Haardt and M. Coleman. *ptrace(2)*. Linux Programmer's Manual, Section 2, November 1999.

[46] M. Haardt and M. Coleman. *fsync(2)*. Linux Programmer's Manual, Section 2, 2001.

[47] M. Haardt and M. Coleman. *fcntl(2)*. Linux Programmer's Manual, Section 2, 2005.

[48] F. Hady. Integrating NAND Flash into the Storage Hierarchy ... Research or Product Design?, 2009. http://csl.cse.psu.edu/wish2009_invitetalk1.html.

[49] R. Hagmann. Reimplementing the Cedar file system using logging and group commit. In *Proceedings of the 11th ACM Symposium on Operating Systems Principles (SOSP '87)*, pages 155–162, Austin, TX, October 1987. ACM Press.

[50] J. S. Heidemann and G. J. Popek. Performance of cache coherence in stackable filing. In *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 3–6, Copper Mountain Resort, CO, December 1995. ACM SIGOPS.

[51] D. Hitz, J. Lau, and M. Malcolm. File System Design for an NFS File Server Appliance. In *Proceedings of the USENIX Winter Technical Conference*, pages 235–245, San Francisco, CA, January 1994. USENIX Association.

[52] Hypertable. Hypertable. www.hypertable.org, 2011.

[53] IBM. Hierarchical Storage Management. www.ibm.com/servers/eserver/iseries/hsmcomp/, 2004.

[54] IEEE/ANSI. Information Technology–Portable Operating System Interface (POSIX)–Part 1: System Application: Program Interface (API) [C Language]. Technical Report STD-1003.1, ISO/IEC, 1996.

[55] Intel Inc. Over-provisioning an intel ssd. Technical Report 324441-001, Intel Inc., October 2010. cache-www.intel.com/cd/00/00/45/95/459555_459555.pdf.

[56] Objective Caml. http://caml.inria.fr/index.en.html.

[57] H. V. Jagadish, P. P. S. Narayan, S. Seshadri, S. Sudarshan, and Rama Kanneganti. Incremental organization for data recording and warehousing. In *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 16–25, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[58] D. Jung, J. Kim, S. Park, J. Kang, and J. Lee. FASS: A Flash-Aware Swap System. In *Proc. of International Workshop on Software Support for Portable Storage (IWSSPS)*, 2005.

[59] A. Kashyap, J. Dave, M. Zubair, C. P. Wright, and E. Zadok. Using the Berkeley Database in the Linux Kernel. `www.fsl.cs.sunysb.edu/project-kbdb.html`, 2004.

[60] J. Katcher. PostMark: a new filesystem benchmark. Technical Report TR3022, Network Appliance, 1997. `www.netapp.com/tech_library/3022.html`.

[61] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the Coda file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 213–225, Asilomar Conference Center, Pacific Grove, CA, October 1991. ACM Press.

[62] D. Kleikamp and S. Best. How the Journaled File System handles the on-disk layout, May 2000. `www-106.ibm.com/developerworks/library/l-jfslayout/`.

[63] S. R. Kleiman. Vnodes: An architecture for multiple file system types in Sun UNIX. In *Proceedings of the Summer USENIX Technical Conference*, pages 238–247, Atlanta, GA, June 1986. USENIX Association.

[64] D. E. Knuth. *The Art of Computer Programming: Sorting and Searching*, volume 3. Addison Wesley, 1973.

[65] R. Kothiyal, V. Tarasov, P. Sehgal, and E. Zadok. Energy and performance evaluation of lossless file data compression on server systems. In *Proceedings of the Second ACM Israeli Experimental Systems Conference (SYSTOR '09)*, Haifa, Israel, May 2009. ACM.

[66] Avinash Lakshman and Prashant Malik. Cassandra: structured storage system on a p2p network. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, PODC '09, pages 5–5, New York, NY, USA, 2009. ACM.

[67] C. Lamb. The design and implementation of a transactional data manager. In *Sun 2004 Worldwide Java Developer Conference*. Sun, 2004. `www.oracle.com/technetwork/database/berkeleydb/transactional-data-manager-129520.pdf`.

[68] Butler W. Lampson. A note on the confinement problem. *Commun. ACM*, 16:613–615, October 1973.

[69] LevelDB, January 2012. `http://code.google.com/p/leveldb`.

[70] A. Leventhal. Flash storage memory. *Communications of the ACM*, 51(7):47–51, 2008.

[71] J. Levon and P. Elie. Oprofile: A system profiler for linux. `http://oprofile.sourceforge.net`, September 2004.

[72] Y. Li, B. He, Q. Luo, and K. Yi. Tree indexing on flash disks. In *ICDE '09: Proceedings of the 2009 IEEE International Conference on Data Engineering*, pages 1303–1306, Washington, DC, USA, 2009. IEEE Computer Society.

[73] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in thor. In *SIGMOD '96: Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329, New York, NY, USA, 1996. ACM.

[74] Wei Lu, Jared Jackson, and Roger Barga. Azureblast: a case study of developing science applications on the cloud. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 413–420, New York, NY, USA, 2010. ACM.

[75] M. Mammarella, S. Hovsepian, and E. Kohler. Modular data storage with anvil. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 147–160, New York, NY, USA, 2009. ACM.

[76] Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello, Randolph Y. Wang, and Thomas E. Anderson. Improving the performance of log-structured file systems with adaptive methods. *SIGOPS Oper. Syst. Rev.*, 31:238–251, October 1997.

[77] D. Maziéres. A toolkit for user-level file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 261–274, Boston, MA, June 2001. USENIX Association.

[78] T. J. McCabe and C. W. Butler. Design complexity measurement and testing. *Communications of the ACM*, 32(12):1415–1425, December 1989.

[79] P. McCullagh. The primebase xt transactional engine. `www.primebase.org/download/pbxt_white_paper.pdf`, 2006.

[80] R. McDougall and J. Mauro. *Solaris Internals: Solaris 10 and OpenSolaris Kernel Architecture, Second Edition*. Prentice Hall, Upper Saddle River, New Jersey, 2006.

[81] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A fast file system for UNIX. *ACM Transactions on Computer Systems*, 2(3):181–197, August 1984.

[82] Microsoft Corporation. Advantages of Using NTFS. `http://technet.microsoft.com/en-us/library/cc976817.aspx`.

[83] Microsoft Corporation. Microsoft MSDN WinFS Documentation. `http://msdn.microsoft.com/data/winfs/`, October 2004.

[84] M. Milenkovic, S. T. Jones, F. Levine, and E. Pineda. Using performance inspector tools. In *Proceedings of the 2009 Linux Symposium*, pages 215–224, Ottawa, Canada, June 2009. IBM, Inc., Linux Symposium.

[85] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: a transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[86] D. Morozhnikov. FUSE ISO File System, January 2006. http://fuse.sourceforge.net/wiki/index.php/FuseIso.

[87] Erik T. Mueller, Johanna D. Moore, and Gerald J. Popek. A nested transaction mechanism for locus. In *Proceedings of the ninth ACM symposium on Operating systems principles*, SOSP '83, pages 71–89, New York, NY, USA, 1983. ACM.

[88] N. Murphy, M. Tonkelowitz, and M. Vernal. The Design and Implementation of the Database File System. www.eecs.harvard.edu/~vernal/learn/cs261r/index.shtml, January 2002.

[89] David R. Musser and Atul Saini. *The STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1995.

[90] MySQL AB. MySQL: The World's Most Popular Open Source Database. www.mysql.org, July 2005.

[91] Beomseok Nam, Henrique Andrade, and Alan Sussman. Multiple range query optimization with distributed cache indexing. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[92] S. Nath and A. Kansal. FlashDB: Dynamic Self-tuning Database for NAND Flash. Technical Report MSR-TR-2006-168, Microsoft Research, November 2006.

[93] E. B. Nightingale, K. Veeraraghavan, P. M. Chen, and J. Flinn. Rethink the sync. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, pages 1–14, Seattle, WA, November 2006. ACM SIGOPS.

[94] M. A. Olson. The Design and Implementation of the Inversion File System. In *Proceedings of the Winter 1993 USENIX Technical Conference*, pages 205–217, San Diego, CA, January 1993. USENIX.

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

[96] Oracle. Btrfs, 2008. http://oss.oracle.com/projects/btrfs/.

[97] Oracle. Database administrator's reference. http://download.oracle.com/docs/cd/B19306_01/server.102/b15658/tuning.htm, March 2009.

[98] Oracle. Bdb java edition faq. www.oracle.com/technetwork/database/berkeleydb/je-faq-096044.html#33, September 2011.

[99] Oracle. Bdb java edition faq. www.oracle.com/technetwork/database/berkeleydb/je-faq-096044.html#38, September 2011.

[100] Oracle Corporation. Oracle Internet File System Archive Documentation. http://otn.oracle.com/documentation/ifs_arch.html, October 2000.

[101] R. Orlandic. Effective management of hierarchical storage using two levels of data clustering. *Mass Storage Systems, IEEE Symposium on*, 0:270, 2003.

[102] Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel J. Abadi, David J. DeWitt, Samuel Madden, and Michael Stonebraker. A comparison of approaches to large-scale data analysis. In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 165–178, New York, NY, USA, 2009. ACM.

[103] H. Payer, M. A. A. Sanvido, Z. Z. Bandic, and C. M. Kirsch. Combo Drive: Optimizing Cost and Performance in a Heterogeneous Storage Device. In *Proc. Workshop on Integrating Solid-state Memory into the Storage Hierarchy (WISH)*, 2009. `http://csl.cse.psu.edu/wish2009_papers/Payer.pdf`.

[104] Eric Perlman, Randal Burns, Yi Li, and Charles Meneveau. Data exploration of turbulence simulations using a database cluster. In *Proceedings of the 2007 ACM/IEEE conference on Supercomputing*, SC '07, pages 23:1–23:11, New York, NY, USA, 2007. ACM.

[105] Z. Peterson and R. Burns. Ext3cow: a time-shifting file system for regulatory compliance. *Trans. Storage*, 1(2):190–212, 2005.

[106] Donald E. Porter, Owen S. Hofmann, Christopher J. Rossbach, Alexander Benn, and Emmett Witchel. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09)*, pages 161–176. ACM, 2009.

[107] PostgreSQL Global Development Team. PostgreSQL. `www.postgresql.org`, 2011.

[108] Calton Pu, Jim Johnson, Rogério de Lemos, Andreas Reuter, David Taylor, and Irfan Zakiuddin. 06121 report: Break out session on guaranteed execution. In *Atomicity: A Unifying Concept in Computer Science*, 2006.

[109] Calton Pu and Jinpeng Wei. A methodical defense against tocttou attacks: The edgi approach. In *Proceedings of the International Symposium on Secure Software Engineering (ISSSE'06)*, pages 399–409, March 2006.

[110] Aditya Rajgarhia and Ashish Gehani. Performance and extension of user space file systems. In *25th Symposium On Applied Computing*. ACM, March 2010.

[111] V. K. Reddy and D. Janakiram. Cohesion Analysis in Linux Kernel. *apsec*, 0:461–466, 2006.

[112] H. Reiser. ReiserFS v.3 Whitepaper. `http://web.archive.org/web/20031015041320/http://namesys.com/`.

[113] H. Reiser. ReiserFS. `www.namesys.com/`, October 2004.

[114] M. Rosenblum. *The Design and Implementation of a Log-structured File System*. PhD thesis, Electrical Engineering and Computer Sciences, Computer Science Division, University of California, 1992.

[115] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-structured File System. In *ACM Transactions on Computer Systems (TOCS)*, pages 26–52. ACM, 1992.

[116] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. *ACM Transactions on Computer Systems*, 10(1):26–52, 1992.

[117] M. Rosenblum and J. K. Ousterhout. The design and implementation of a log-structured file system. In *Proceedings of 13th ACM Symposium on Operating Systems Principles*, pages 1–15, Asilomar Conference Center, Pacific Grove, CA, October 1991. Association for Computing Machinery SIGOPS.

[118] M. Saxena and M. M. Swift. Flashvm: Revisiting the virtual memory hierarchy, 2009.

[119] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles (SOSP '91)*, pages 239–253, Pacific Grove, CA, October 1991. ACM Press.

[120] R. Sears and E. Brewer. Stasis: Flexible Transactional Storage. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, Seattle, WA, November 2006. ACM SIGOPS.

[121] R. Sears, M. Callaghan, and E. Brewer. Rose: Compressed, log-structured replication. In *Proceedings of the VLDB Endowment*, volume 1, Auckland, New Zealand, 2008.

[122] P. Sehgal, V. Tarasov, and E. Zadok. Evaluating performance and energy in file system server workloads extensions. In *Proceedings of the Eighth USENIX Conference on File and Storage Technologies (FAST '10)*, pages 253–266, San Jose, CA, February 2010. USENIX Association.

[123] P. Sehgal, V. Tarasov, and E. Zadok. Optimizing Energy and Performance for Server-Class File System Workloads. *ACM Transactions on Storage (TOS)*, 6(3), September 2010.

[124] M. Seltzer and M. Stonebraker. Transaction Support in Read Optimized and Write Optimized File Systems. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 174–185, Brisbane, Australia, August 1990. Morgan Kaufmann.

[125] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[126] M. I. Seltzer. Transaction Support in a Log-Structured File System. In *Proceedings of the Ninth International Conference on Data Engineering*, pages 503–510, Vienna, Austria, April 1993.

[127] M. I. Seltzer, G. R. Ganger, M. K. McKusick, K. A. Smith, C. A. N. Soules, and C. A. Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *Proceedings of the Annual USENIX Technical Conference*, pages 71–84, San Diego, CA, June 2000. USENIX Association.

[128] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. NFS Version 4 Protocol. Technical Report RFC 3530, Network Working Group, April 2003.

[129] L. Shrira, B. Liskov, M. Castro, and A. Adya. How to scale transactional storage systems. In *EW 7: Proceedings of the 7th workshop on ACM SIGOPS European workshop*, pages 121–127, New York, NY, USA, 1996. ACM.

[130] Sleepycat Software, Inc. *Berkeley DB Reference Guide*, 4.3.27 edition, December 2004. `www.oracle.com/technology/documentation/berkeley-db/db/api_c/frame.html`.

[131] Keith A. Smith. File system benchmarks. `www.eecs.harvard.edu/~keith/usenix96/`, 1996.

[132] A. Z. Spector, J. Butcher, D. S. Daniels, D. J. Duchamp, J. L. Eppinger, C. E. Fineman, A. Heddaya, and P. M. Schwarz. Support for distributed transactions in the tabs prototype. *IEEE Trans. Softw. Eng.*, 11:520–530, June 1985.

[133] R. Spillane, S. Dixit, S. Archak, S. Bhanage, and E. Zadok. Exporting kernel page caching for efficient user-level I/O. In *Proceedings of the 26th International IEEE Symposium on Mass Storage Systems and Technologies*, Incline Village, Nevada, May 2010. IEEE.

[134] R. Spillane, R. Sears, C. Yalamanchili, S. Gaikwad, M. Chinni, and E. Zadok. Story Book: An Efficient Extensible Provenance Framework. In *Proceedings of the first USENIX workshop on the Theory and Practice of Provenance (TAPP '09)*, San Francisco, CA, February 2009. USENIX Association.

[135] R. P. Spillane, S. Gaikwad, E. Zadok, C. P. Wright, and M. Chinni. Enabling transactional file access via lightweight kernel extensions. In *Proceedings of the Seventh USENIX Conference on File and Storage Technologies (FAST '09)*, pages 29–42, San Francisco, CA, February 2009. USENIX Association.

[136] R. P. Spillane, P. J. Shetty, E. Zadok, S. Archak, and S. Dixit. An efficient multi-tier tablet server storage architecture. In *Proceedings of the 2nd ACM Symposium on Cloud Computing (SOCC'11)*, Cascais, Portugal, October 2011.

[137] M. Stonebraker. One Size Fits All: An Idea Whose Time has Come and Gone. In *In Proceedings of the International Conference on Data Engineering (ICDE)*, pages 2–11, 2005.

[138] Michael Stonebraker and Ugur Cetintemel. One size fits all: An idea whose time has come and gone. In *Proceedings of the 21st International Conference on Data Engineering*, ICDE '05, pages 2–11, Washington, DC, USA, 2005. IEEE Computer Society.

[139] Sun Microsystems, Inc. Solaris ZFS file storage solution. *Solaris 10 Data Sheets*, 2004. `www.sun.com/software/solaris/ds/zfs.jsp`.

[140] A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck. Scalability in the XFS file system. In *Proceedings of the Annual USENIX Technical Conference*, pages 1–14, San Diego, CA, January 1996.

[141] M. Szeredi. Filesystem in Userspace. http://fuse.sourceforge.net, February 2005.

[142] A. Traeger, N. Joukov, C. P. Wright, and E. Zadok. A Nine Year Study of File System and Storage Benchmarking. *ACM Transactions on Storage (TOS)*, 4(2):25–80, May 2008.

[143] T. Ts'o. Planned Extensions to the Linux Ext2/Ext3 Filesystem. In *Proceedings of the Annual USENIX Technical Conference, FREENIX Track*, pages 235–243, Monterey, CA, June 2002. USENIX Association.

[144] Stephen Tweedie. Ext3, journaling filesystem. In *Ottawa Linux Symposium*, July 2000. http://olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html.

[145] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[146] Andy Twigg, Andrew Byde, Grzegorz Milos, Tim Moreton, John Wilkes, and Tom Wilkie. Stratified b-trees and versioned dictionaries. In *Proceedings of the 3rd USENIX conference on Hot topics in storage and file systems*, HotStorage'11, pages 10–10, Berkeley, CA, USA, 2011. USENIX Association.

[147] S. Verma. Transactional NTFS (TxF). http://msdn2.microsoft.com/en-us/library/aa365456.aspx, 2006.

[148] L. Walsh, V. Akhmechet, and M. Glukhovsky. Rethinkdb - rethinking database storage. www.rethinkdb.com/papers/whitepaper.pdf, 2009.

[149] A. Wang, G. Kuenning, P. Reiher, and G. Popek. The conquest file system: Better performance through a disk/persistent-ram hybrid design. *Trans. Storage*, 2(3):309–348, 2006.

[150] Dan E. Willard. Log-logarithmic worst-case range queries are possible in space theta(n). *Inf. Process. Lett.*, 17(2):81–84, 1983.

[151] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID Semantics to the File System. *ACM Transactions on Storage (TOS)*, 3(2):1–42, June 2007.

[152] Peter Zaitsev. True fsync in linux (on ide). Technical report, MySQL AB, Senior Support Engineer, March 2004. lkml.org/lkml/2004/3/17/188.

[153] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the 6th USENIX Conference on File and Storage Technologies*. USENIX Association, 2008.

[154] B. Zhu, K. Li, and H. Patterson. Avoiding the Disk Bottleneck in the Data Domain Deduplication File System. In *Proceedings of the Sixth USENIX Conference on File and Storage Technologies (FAST '08)*, San Jose, California, USA, 2008.

Appendix A

Glossary

An alphabetical glossary of terms is provided.

`1ffs` is a custom *1-file file system* we built for use with user-level file systems such as LSMFS. It is a simple file system that exports only a single file, but can respond to swapping and flush events within the kernel and runs on top of the disk device [133]. In this way, a user-level database or file system can intelligently respond to memory pressure without having to pre-allocate a fixed cache size.

ACI is an acronym referring to (A)tomic, (C)onsistent, (I)solated, but not necessarily (D)urable. This kind of transaction is commonly used in high-insertion throughput and file system workloads where individual operations are either more expendable, or are easily repeated [43]. See "Transaction."

ACID is an acronym referring to (A)tomic, (C)onsistent, (I)solated, and (D)urable [43]. See "Transaction."

ARIES is an acronym which stands for (A)lgorithm for (R)ecovery and (I)solation (E)xploiting (S)emantics. It is the de facto write-ahead logging algorithm used for transactions and recovery in databases [85]. ARIES works well for many concurrent transactions which read several random tuples, update their contents, and then durably commit these changes during a transaction commit. ARIES can be between 2–3$\times$ slower than a typical file system for more sequential file system workloads. ARIES gathers undo records and can be orders of magnitude slower for transactions which only perform random updates, insertions, or deletes compared to a log-structured approach.

Amino is a trial design [151] built on top of SchemaFS which provides support for system transactions, or transactions that perform POSIX operations on the Amino file system. See *transactions*, *SchemaFS*, and *POSIX*.

Berkeley DB is a database library [130] designed to be portable across many operating systems. The library provides support for an extensible logging infrastructure, shared memory cache, and transactional B-tree, hash table, and array on-disk data structures.

Bloom Filter is a space-efficient data structure used as an approximate membership query, or a negative-cache for lookups [15]. It consists of k hash functions and a bit-array. To insert it takes any key and translates it into k random bit-offsets within this bit array, and sets those offsets to 1. To perform a "may-contain" query it takes any key and translates it into k random bit-offsets within the bit array, and returns true if all offsets are 1. If may-contain returns true, there is a probability approaching $1/2^k$ that the Bloom Filter does *not* contain the sought after key. This is the false positive rate of the Bloom filter. This is only true if the Bloom filter is optimally configured and contains no more keys than it was originally configured to hold.

BtrFS is a Linux file system designed for better handling of snap-shots and long-term fragmentation [96].

COLA is a variant of the log-structured merge-tree. The COLA or cache-oblivious look-ahead array [11] (without fractional cascading) is independently discovered and used in actual LSM-tree implementations (e.g., Acunu [145], Anvil [75], and HBase [28]). See *fractional cascading*. The COLA paper provides an excellent analysis of the LSM-tree within the DAM model, and shows that its configuration does not depend on the ideal block size of the device.

Consistent State is any point in the single predictable timeline that all transactions and other operations can be placed on. If an error occurs at some point in this timeline, recovery will restore the storage system to how it was at the last point in the timeline immediately before the error [43].

Defragmentation is a process for removing fragmentation. Fragmentation occurs when a block that is the size of the minimum unit of allocation and deallocation is only being partially utilized. The block cannot be deallocated because it still is being utilized, but the unutilized portions cannot be deallocated because they are smaller than the minimum unit of deallocation. Defragmentation copies the utilized portion of the block to another location, leaving the block completely unutilized, and it can then be deallocated. Since defragmentation incurs additional reads and writes not directly related to insert, delete, update, or query operations, it is typically considered as overhead. See Chapter 6.

Ext3 is the default Linux file system [19]. It uses a combination of journaling, hash tables to quickly access directory entries, and traditional file system structures.

Fractional Cascading is a technique that can be used to avoid repeated searches within a data structure [21]. Some data structures are composed of multiple searchable sub-data structures. When searching through the data structure, one must search through some or all of the sub-data structures. Fractional cascading builds a secondary index for each of these sub-data structures. If sub-data structure A is sought through before sub-data structure B, then we embed B's secondary index into A such that when the block that may contain the sought after element in A is retrieved, with it we also retrieve the location of the constant number of blocks that may contain the element in B. In this way we can confine our search through B to those specific blocks. In practice, the secondary indexes of both A and B will easily fit in RAM, and so B's secondary index is not embedded into A as it will already be in RAM anyway and does not need to "piggy-back" on block retrievals from A.

FUSE is a plugin for the Linux kernel that developers can use to write (F)ile systems in (U)ser (S)pac(E) [86].

Indirect block is an internal or parent node in a file radix tree [19]. A file radix tree is a high arity tree (e.g., 512-ary) that contains offsets (e.g., 32-bit or 64-bit), and where the leaf nodes are blocks of file data (e.g., 512B or 4096B blocks). File trees permit easily maintained sparse files, more flexible allocation strategies, and more flexible defragmentation strategies. They are complex and must typically be maintained with a journal. See "Journaling."

Journal(ling) is a transactional log used by file systems that only intend to perform internal meta-data transactions. Typically on `inodes`, indirect blocks, and other internal file system structures. Journaling is how file systems use the journal: first, they write a meta-data operation command to the journal, and then second they perform the operation. If an error occurs, they can re-try the operation by reading back the command from the journal, and re-trying the operation until it succeeds [19, 39]. See "Recovery."

LSMFS is a file system developed on top of a COLA implementation with basic transactional support, but used only internally for meta-data. It runs on top of `1ffs`, and uses exported page caching for user-level file systems [133].

Memtable is a buffer or in-RAM list of sorted tuples. In GTSSLv1 and v2, it is a red-black tree. In other systems it is typically a skip list. Once the memtable is full, it is serialized to storage as an SSTable or list [20]. If stitching is used, it is serialized as a patch-tree. See Section 3.2.

Patch-tree is a compound of lists of sorted tuples or source-trees, along with a single secondary index, and a single quotient filter. It effectively replaces the SSTable in a stitching LSM-tree. See Chapter 6 and specifically Section 6.1.

POSIX is a standard interface for interacting with the underlying operating system. It allows for manipulation of files, processes, memory, networking, and user and permission management [54].

Quotient Filter or QF is similar to the Bloom filter, but supports duplicates, deletes, and its contents can be efficiently re-hashed into a larger QF, which is less than a preset size [12].

Recovery is executing an algorithm that reads commands written to a journal or transactional log, and either re-tries them, or reverses them in order to restore the storage system to a consistent state [43]. See "consistent state."

ReiserFS is a Linux file system designed for better handling of small files and meta-data operations [113].

SchemaFS is a POSIX API for system transactions built on top of Berkeley DB 4.4. It is used in conjunction with Amino [151], see *Amino* and *Berkeley DB*.

Structured data is data whose access patterns can be difficult to predict, and could be based on application-specific algorithms. Structured data includes database tables, file system meta-data, tablets used in cloud data centers, large astronomical images, large graphs, indexes, and much more [20].

Source-tree is a list of sorted tuples on storage. It is equivalent to an SSTable or list in a traditional LSM-tree, but without a Bloom filter or secondary index. When two patch-trees are merged together, their constituent lists of sorted tuples are combined together into the new patch-tree, along with the fill, which is also a new list of sorted tuples. During combination a new secondary index and quotient filter are constructed. The secondary index and QF for each of the old patch trees are discarded. The sorted lists of tuples contributed by the old patch trees are source-trees. After merging, source-trees could be partially or completely deleted. See Chapter 6 and specifically Section 6.1.

SSTable is a serialized memtable. It is an on-storage list of sorted tuples, along with (optionally) a serialized (but potentially cached) secondary-index, and a serialized (but potentially cached) Bloom filter [20]. If the term SSTable is used to describe a serialized list of tuples in a stitching LSM-tree, it refers to a patch-tree. See Section 3.2. See Wanna-B-tree.

Transaction a grouping of operations into a compound operation. This compound operation cannot be partially applied (atomic), can be placed in a single predictable timeline with all other operations (consistent), can be performed as if it were the only operation occurring at that time (isolated), and once success is confirmed, the operation cannot be revoked at a later time (durable). Transactions may exhibit some or all of these properties (atomic, consistent, isolated, durable). In this thesis we deal with transactions that are atomic, consistent, and isolated, and optionally, durable [43]. See "ACID."

VFS is the virtual file system and refers to using indirect calls into specific file system implementations using a pre-defined standard set of routines which can be overridden by each specific file system implementation. The VFS also refers to code common to most or all specific file system implementations, which is implemented in terms of these pre-defined standard routines which perform indirect calls. By organizing the storage component of the operating system in this way, code common to all specific file system implementations can be refactored into a single, maintainable code base, and only code that differs due to differences in file system formats need be separately maintained.In this way an operating system can efficiently mount and interact with multiple file systems [63].

VT-tree is our variant of the SAMT which utilizes the stitching generalization for LSM-trees and uses quotient filters to support efficient point queries, see Chapter 6.

Wanna-B-tree is a simplified B-tree that supports only appending sorted tuples and performing lookups. It is composed of *zones* which can be individually deallocated without having to destroy the entire Wanna-B-tree. It consists of a secondary index, and a set of leaf nodes containing tuples that are called Wanna-B-leaves. See Chapter 3. See *zones*.

Write-ahead Logging (WAL) is a family of logging algorithms for database logging and recovery. ARIES is one variant of a WAL algorithm. WAL algorithms are at least $2\times$ slower than a log-structured approach for large, sequential, or asynchronous transactions. WAL algorithms do not support larger-than-RAM transactions when not gathering undo records and performing redo-only logging [43, 85].

XFS is a Linux file system developed by SGI which is designed for better handling of long-term fragmentation [140].

Zones are the unit of allocation and deallocation in the GTSSLv1 and GTSSLv2 systems. After performing stitching on a VT-tree, a zone may no longer have used tuples within it; at this point the zone can be deallocated. If it has a mix of used and unused tuples, then it wastes space. It can be defragmented at any time by copying the used tuples out during a minor compaction and can then be deallocated.

Appendix B

Java Database Cache Analysis

We evaluated the performance of just Cassandra's primary caching implementation, the Java skip list, and compared it to the primary caching implementation used by GTSSLv1, the STL red-black tree. We performed further profiling of Cassandra by observing CPU time usage, L2 cache misses, and counting out-of-core requests (using CPU counters).

When the database or key-value store is CPU-bound, Java generational garbage collectors often have difficulty in efficiently managing a large cache that contains objects that are mostly created once, and then infrequently used there after [99]. We used the garbage collection settings that Cassandra specified in their configuration. We suspected that the bottleneck to Cassandra's performance may lie within the skip list implementation that Cassandra uses as its primary cache. When randomly inserting into a skip list on a machine with 16GB of RAM, and with Java configured to have an 8GB heap, we noticed that insertion throughput would steadily drop to 20,000 inserts per second. However our above benchmarks ran with a 3GB heap, and under these circumstances, insertion throughput did not drop so low, but still much lower than an STL red-black tree.

To analyze the skip list's performance on the same system, we ran a benchmark that compares a skip list to an STL red-black tree. Figures B.1, B.2, B.3, and B.4 depict a benchmark where random 64B tuples are entered into a C++ red-black tree [89], C++ skip list [69], Java red-black tree, and a Java skip list, respectively. Garbage collection events that were collected while running the Java benchmarks are also shown. The benchmark runs in three epochs, as the per-epoch performance for Java is not entirely consistent, it is useful to run multiple epochs to see the disparity. In each epoch, the cache (whether it is a red-black tree or skip list) is cleared, and a new set of random elements are inserted. We inserted 18.3 million tuples, or 1.1GB of tuples. We attempted to insert more, but Java crashed with an out-of-memory exception with our configured heap size (5GB) on our system. We used the same system configuration as for the varying key-value and read-write trade-off benchmarks above. We found that the average throughput of insertions into the red-black tree was 341,000 inserts/second, and for the Java red-black tree was 223,000 inserts/second. For the C++ skip list we measured 270,000 inserts/second, and for the Java skip list we measured 109,000 inserts/second. The C++ red-black tree sustains a 1.5× faster insertion throughput over the Java red-black tree. The C++ skip list sustains a 2.5× faster insertion throughput over the Java skip list. GTSSLv1 uses the STL red-black tree, and Java uses the measured skip list. We found the

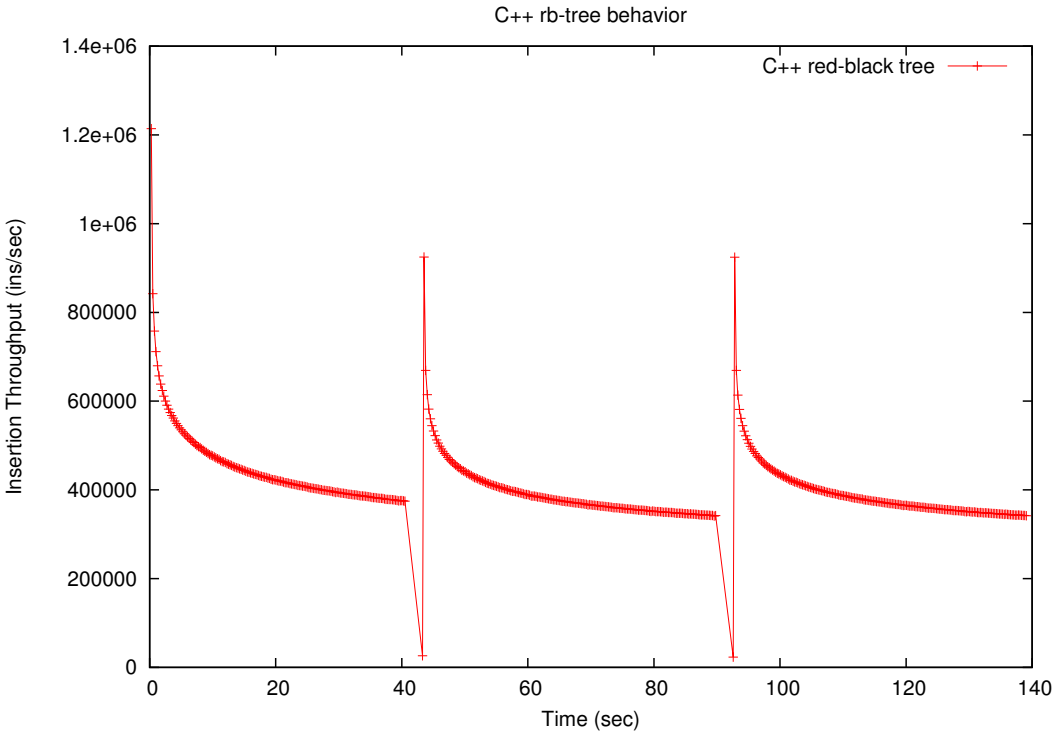

Figure B.1: ***Multi-eviction C++ red-black tree Micro-benchmark***

C++ red-black tree was overall 3.1× faster than the Java skip list. We also noted the consistent interference of Java's garbage collection, sometimes running for 8–10 seconds, decreased overall insertion throughput. Total time spent garbage collecting for the JAVA SKIP LIST configuration was 209.9 seconds, or 42% of the total runtime of the benchmark. Garbage collection causes significant performance differences in memory-access intensive applications, such as a database cache.

We configured OProfile [71] to profile for both timer interrupts, and instructions which required out-of-core requests (`OFFCORE_REQUESTS`) [1]. Off-core requests includes data read requests, code read requests, uncached memory requests, L1D writebacks, and offcore RFO requests.

Table B.1 shows the performance counters measured by OProfile when running the first 10% of the 64B data point of the key-value benchmark from Figure 5.8. The top group of rows corre-

[1] We performed further profiling of Cassandra using tools that could perform more in-depth profiling of Java Just-in-Time (JIT) compiled code as well as utilize other CPU counters besides the timer interrupt. First we attempted to use Performance Inspector [84]. Performance Inspector provides Java Just-in-Time (JIT) profiling, and most importantly, can profile using CPU timer interrupts, or other kinds of CPU counter events, such as L2 cache misses. Unfortunately, despite claiming to support Ubuntu 10.04, Performance Inspector relies on the JVMI interface to the underlying JVM to perform profiling, which has since been deprecated after Java 1.5, and Cassandra 0.7 requires at least Java 1.6 which no longer supports JVMI. OProfile [71] *does* use the JVMTI interface, which is the preferred way of converting text addresses into Java symbols for Java 1.5 and later we also learned that OProfile can be configured to profile for arbitrary CPU events.

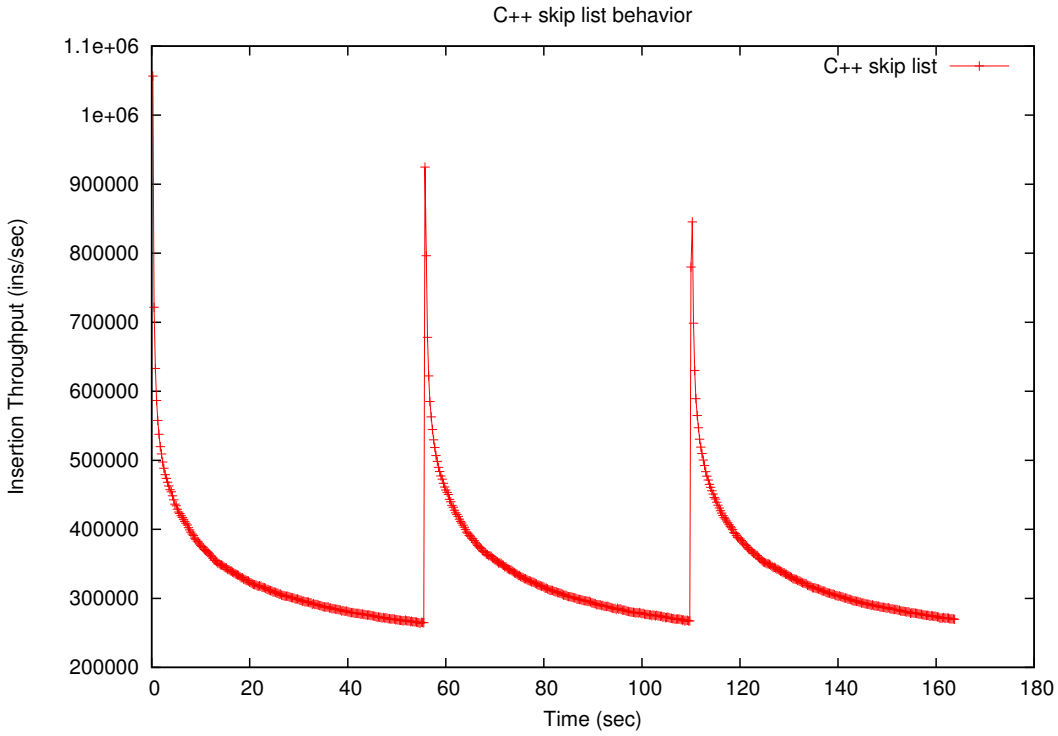

Figure B.2: ***Multi-eviction C++ skip list Micro-benchmark***

sponds to Cassandra, and the bottom group of rows corresponds to GTSSLv1. The total number of clock samples taken during the GTSSLv1 and Cassandra runs were 4,408,685 and 80,556,391, respectively; the numbers of LLC misses were 6,498 and 154,348 respectively. Cassandra performed 521.9 clock samples per LLC miss, where as GTSSLv1 performed 678.5 clock samples per LLC miss. This indicates that irrespective of running time, GTSSLv1 had fewer LLC misses per operation than Cassandra. Similarly, GTSSLv1 performed 61.6 clock samples per off-core request, whereas Cassandra performed only 25.5 clock samples per off-core request. We see that 32.6% of Cassandra's LLC misses occurred within `libjvm.so` (the JVM), whereas 33.6% of GTSSLv1's LLC misses occurred within its "Merge" routine, which is responsible for performing comparisons during cache accesses, and merges. In addition to LLC misses from within the JVM, Cassandra's skip list (findPredecessor, doGet, and doPut), and calls to binary search account for 26.5% of its LLC misses. Cassandra spends 55.3% of its time and 42.7% of its LLC misses within the kernel, the JVM, and the benchmark program itself. Conversely, GTSSLv1 spends 43.2% of its time and only 19.2% of its LLC misses within the kernel and `libstdc++`. We can see that Cassandra performs 2.2× more of its LLC misses within the kernel, the JVM, or the benchmark program, and not JIT-ed database code. The primary source of LLC misses in Cassandra is `libjvm.so`. This indicates that Cassandra performs 32.6% of its LLC misses while interpreting non-JIT-ed code, or manipulating memory within the JVM. GTSSLv1 directly manipulates memory through `mmap`ed

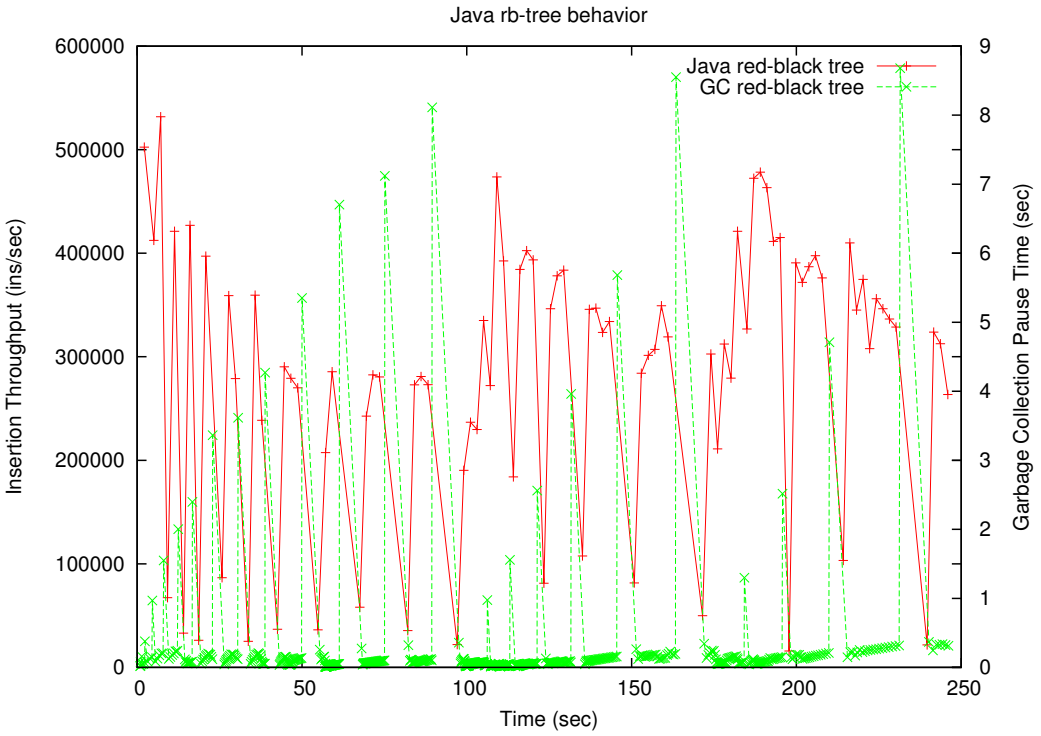

Figure B.3: ***Multi-eviction Java red-black tree Micro-benchmark***

regions, and from optimized machine code running directly on top of the kernel.

Java both compiles and interprets code when running an executable. Java also provides array-bounds checking, built-in synchronization primitives, garbage collection, runtime type-cast checking, and reflection facilities. These features contribute to more easily developed code, but also contribute to performance losses for large complex systems such as Cassandra and HBase. We found it difficult to carefully pin-point a single bottleneck in Cassandra or HBase's performance because we were not able to find one specific bottleneck. We did find that the JVM consumed up to 25% of CPU time, which did not include overheads from the Java skip list implementation, or other code which was just-in-time compiled.

We found that major components, such as the data structure used for caching tuples, can be up to 3× slower than an equivalent in C++. Ultimately, differences in design and implementation make a direct comparison of performance difficult, but we can see that GTSSLv1 performance is at worst comparable to existing, widely used key-value stores; and for these specific workloads that are so heavily CPU-bound, basing the LSM-tree database implementation on C++ as GTSSLv1 does is a much faster approach than basing the implementation on Java. These results corroborate the reported inefficient use of CPU and RAM in HBase by others as well [5].

Future TSSL architectures must seriously consider CPU efficiency as the cost of a random write drops significantly for 1KB block sizes on Flash SSD.

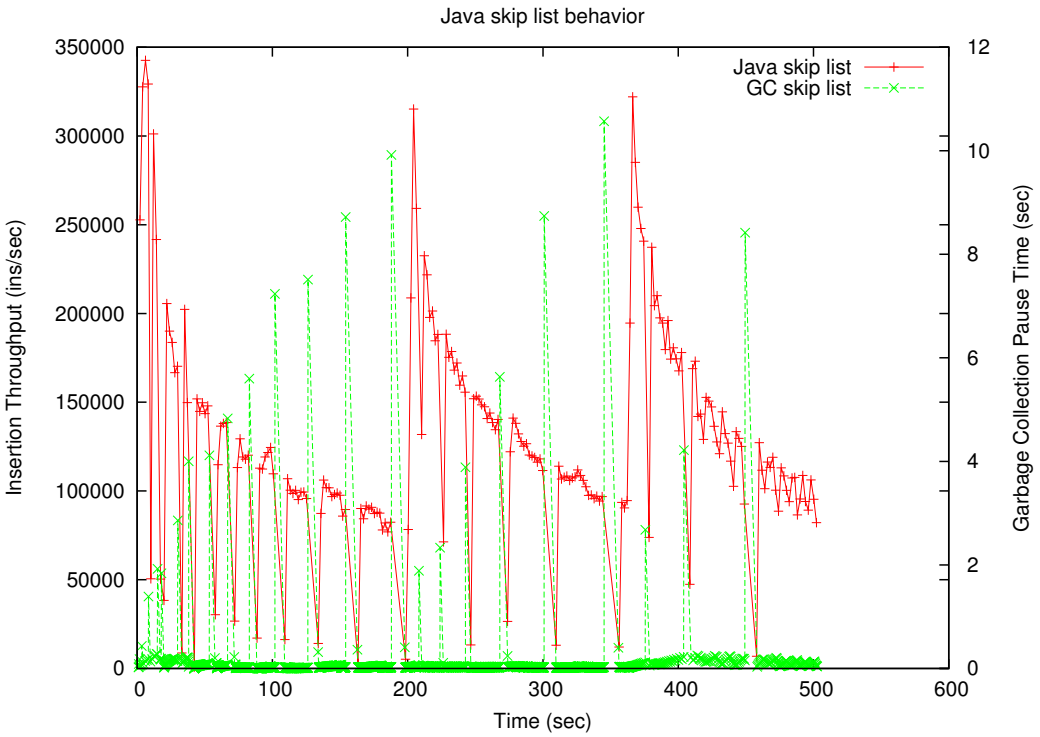

Figure B.4: ***Multi-eviction Java skip list Micro-benchmark***

Symbol	CLK	Perc.	L2 Miss.	Perc.	OFFCORE	Perc.
Cassandra						
vmlinux	19,071,280	23.7%	5,937	3.8%	831,463	26.3%
libjvm.so	17,982,060	22.3%	50,369	32.6%	454,741	14.4%
Bench.java	7,476,035	9.3%	9,729	6.3%	424,231	13.4%
findPredecessor	2,142,821	2.7%	12,401	8.0%	54,929	1.7%
MD5.implCompress	1,971,985	2.5%	257	0.2%	34,672	1.1%
doGet	1,713,567	2.1%	12,061	7.8%	42,016	1.3%
findPredecessor	1,260,313	1.6%	7,606	4.9%	33,118	1.0%
SSTableReader.getPosition	807,889	1.0%	1,251	0.8%	13,652	0.4%
StorageProxy.mutate	693,905	0.9%	1,096	0.7%	41,643	1.3%
indexedBinarySearch	669,771	0.8%	5,158	3.3%	15,891	0.5%
doPut	617,549	0.8%	3,783	2.5%	18,368	0.6%
GTSSLv1						
vmlinux	1,655,251	37.4%	456	7.0%	30,745	42.7%
Merge	451,879	10.2%	2,185	33.6%	10,380	14.4%
libcrypto.so	375,372	8.5%	1	0.0%	1,125	1.6%
nolock_insert	275,922	6.2%	1,097	16.9%	6,268	8.7%
libstdc++	255,283	5.8%	793	12.2%	3,300	4.6%
bloom::insert_key	79,923	1.8%	302	4.6%	2,700	3.8%

Table B.1: Performance counters of cpu-time, L2 cache misses, and off-core requests.