SCALABLE LOW-LATENCY INDEXES FOR A KEY-VALUE STORE

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Ankita Arvind Kejriwal
March 2017

This dissertation is online at: http://purl.stanford.edu/hv866jd5663

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

John Ousterhout, Primary Adviser

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Hector Garcia-Molina

I certify that I have read this dissertation and that, in my opinion, it is fully adequate in scope and quality as a dissertation for the degree of Doctor of Philosophy.

Keith Winstein

Approved for the Stanford University Committee on Graduate Studies.

Patricia J. Gumport, Vice Provost for Graduate Education

This signature page was generated electronically upon submission of this dissertation in electronic format. An original signed hard copy of the signature page is on file in University Archives.

Abstract

Many large-scale key-value storage systems sacrifice features like secondary indexing and/or strong consistency in favor of scalability or performance. This limits the ease and efficiency of application development on such systems. Implementing secondary indexing in a large-scale memory based system is challenging because the goals for low latency, high scalability, strong consistency and high availability often conflict with each other.

This dissertation shows how a large-scale key-value storage system can be extended to provide secondary indexes while meeting those goals. The resulting architecture is called Scalable Low-Latency Indexes for a Key-Value Store, or SLIK. It extends a standard key-value store to enable multiple secondary keys for each object and allows lookups and range queries on these keys via secondary indexes. SLIK allows indexes to be partitioned and distributed independently of the data in tables in order to ensure scalability. Locating objects and corresponding index entries on different servers can lead to potential consistency issues. However, SLIK provides strong consistency guarantees using a lightweight ordered write approach. While SLIK stores indexes in DRAM to enable low latency, it ensures that the index information is durable and quickly recovered using backups in case of crashes.

This design was implemented in RAMCloud, a distributed in-memory key-value storage system. This implementation performs indexed reads in 11–13 µs and writes in 30–37 µs, which is approximately twice the latency of basic non-indexed reads and writes in RAMCloud. It supports indexes spanning thousands of nodes, and yields linear scalability for throughput.

To my parents.

Acknowledgements

If I were a better writer, and if I had many more pages to write, I might have managed to express the importance of the many people in my life and my gratitude for their support. Alas, I am only the writer I am and what follows must suffice.

First and foremost, I would like to thank my advisor, John Ousterhout, for being everything I could ever ask for in an advisor. Some professors are great researchers, some are great advisors, and some are great teachers - I am lucky to have found all three in John. Every interaction with John has been an absolute pleasure: discussing high level ideas, designing distributed systems algorithms on the whiteboard, debugging pesky performance issues, learning how to write good systems software, or talking about life in general. John's maxims, distilled from these interactions, will always stay with me. For example, he pushed to "dig one level deeper" when debugging or analyzing performance. He often said that "if you dont know what the problem was, you havent fixed it" - an idea that served well when I found non-deterministic bugs. John has supported and encouraged me throughout my graduate years. He helped me overcome the hesitancy in asking questions (which is common in young graduate students, but nonetheless impacts how quickly we learn). He taught me to think critically, to never accept an idea simply because it came from someone with authority. He pushed me to engineer around what appeared to be unreasonable constraints in order to stretch my thinking. John has played a major role in shaping me in my personal and professional life. When I grow up, I want to be him.

At Stanford, I've had the opportunity to interact with and learn from some of the best minds in Computer Science. They have influenced the way I think and I am thankful for my interactions with every one of them. In particular, I would like to thank Hector Garcia-Molina and Keith Winstein for serving on my reading committee, and Mendel Rosenblum for serving on my oral exam committee. Hector provided different perspectives on many distributed systems problems and helped me fill the gaps in my knowledge. Keith improved

my clarity of thought in technical ideas by encouraging me to challenge my assumptions. He also provided a really useful bit of career advice: he said I should ask myself "in five years from now, what do I want to have accomplished in the past five years?" This simple twist on the common question "where do you see yourself in five years?" is going to help me with my career decisions for a long time to come. Mendel always encouraged me to step back and look at the big picture and to think about the eventual goal of a project (or a group of projects), no matter how far that might be.

Over the long course of my PhD, I spent most of my time with a fantastic group of students in Gates 444. Our lab had students in two big waves, and I was placed squarely between them. However, being the awkward middle child turned out be a blessing in disguise as I was able to closely interact with both these generations. The 1st generation – Diego Ongaro, Steve Rumble, and Ryan Stutsman – taught me about distributed systems in general, and RAMCloud in particular. I enjoyed seeing their journeys take them different places and accomplish wonderful things. The 2nd generation – Jonathan Ellithorpe, Collin Lee, Behnam Montazeri, Seo Jin Park, Henry Qin, and Stephen Yang – have been my fellow troublemakers and comrades-in-arms for most of my PhD. Together, we've explored new projects, brainstormed countless ideas, and battled Git and GDB. They have all come a long way since we first met and I can't wait to see where they go from here. I'm also happy to have had the chance to work with Greg Hill, William Sheu, and Jacqueline Speiser – while we didn't have too much time together, I appreciate that I had the opportunity to get to know them. I would like to give a special shout-out to Arjun Gopalan, Ashish Gupta, Zhihao Jia and Stephen Yang, who have contributed directly to my dissertation work. I'm proud to call myself a labmate to all of them.

Navigating graduate school would not have been as easy without the help of the wonderful staff in the Computer Science Department. I would like to especially thank Sue George, Jay Subramanian, Verna Wong and Jam Kiattinant. Additionally, the staff at Bechtel International Center helped demystify and manage the paperwork that comes with being an international student.

At Stanford I have met many wonderful people, some of whom have become my closest friends. Doing a PhD can be a crazy adventure, filled with ups and downs; my friends have made this the best adventure I could've asked for. They have been my home away from home. We've shared frustrations and advice; enjoyed discussing Computer Science problems, technology and politics; saved the world playing Pandemic, built Seven Wonders;

waltzed and swinged the night away; and so much more. No matter where life takes us, I hope we will always remain friends.

I would also like to thank my extended family for always supporting me. My *nani* and *nana* (grandparents) were always proud of me, even when I didn't feel the same way. I wish they were here today.

Every day, I am thankful for having Kiran and Arvind Kejriwal as my mom and dad. They have supported me and provided guidance whenever I needed it. My dad instilled excitement about computers in me. He ensured I knew that I could do whatever I wanted in life, but I now know quite how proud he was when I chose to follow in his footsteps. My mom has always been there for me. When I was a kid, she stayed up with me into the wee hours of the night to keep me company as I finished reading that last chapter of a book or wrote another program. Even now that I live across the world (12 timezones apart), she's always just a phone call away, whether I need advice or just want to chat. My parents have been my co-conspirators in my adventures with science, math, dancing, swimming, programming, and anything else that I've wanted to do. Everyday they surprise me with the their openness to new ideas and acceptance of whoever I want to be. They have given me more love than any one person can ever hope to receive.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Over the last decade, main-memory-based data storage systems arose to meet the needs of large-scale web applications. These systems have scaled to span hundreds or thousands of servers, with unprecedented overall performance. Some examples include Aerospike [1], H-Store [24], RAMCloud [34] and Redis [13]. However, in order to achieve their scalability, most large-scale storage systems have accepted compromises in their feature sets and consistency models. In particular, many of these systems are simple key-value stores with no secondary indexes. The lack of secondary indexes makes it difficult to implement applications that need to make range queries or retrieve data by keys other than the primary key.

Indexing has been studied extensively in the context of traditional databases. However, its design for a low-latency large-scale main-memory storage system presents several unique design issues (given below). These are further challenging due to the inherent tension between some of them.

- **Low Latency:** The latency for indexed operations should be as low as possible. This requirement means that the system should harness low latency networks and store index data in DRAM. Further, the system should leave out complex mechanisms wherever possible in favor of lightweight methods that minimize overhead.

- **Scalability:** A large-scale data store must support tables so large that their objects and indexes need to span many servers. The total throughput of an index should increase linearly with the number of servers it spans. This objective is at odds with low latency, as contacting more servers (even if done in parallel) increases latency.

Ideally, a system should offer nearly constant latency irrespective of the number of servers an index spans.

- **Consistency:** The system should provide clients with strong consistency guarantees, similar to what a centralized system might provide. For instance, when an indexed object is written, the update to that object and all of its indexes must appear atomic, even in the face of concurrent accesses and server crashes. However, providing consistency when information is distributed, traditionally requires locks or algorithms that impact latency or scalability. Further, as data and indexes become sharded over more and more nodes, it becomes increasingly complex and expensive to manage metadata and maintain consistency between data and the corresponding indexes.

- **Durability and Availability:** Even though all the data (including indexes) is stored in DRAM to enable low latency, it should be durable (i.e., it must survive server crashes). Further, the system must also be continuously available. This requirement means that after crashes, the indexing system should recover indexes as quickly as the underlying storage system recovers objects.

- **Dealing With Large Scale Operations:** To maximize scalability, large-scale long-running operations must not block other operations. For example, large range lookups should not block other lookups or writes on that table. Schema changes such as adding or removing indexes, and splitting or migrating index partitions should be accomplished without taking the system offline.

In this dissertation, I show how to overcome these challenges and how a large-scale key-value store can be extended to provide secondary indexes. The resulting architecture, SLIK (Scalable, Low-latency Indexes for a Key-value store), combines several attractive features. First, it stores all data in DRAM and employs simple mechanisms to enable ultra low latency, while recovering quickly from crashes to ensure durability and high availability. Second, it scales to support high performance even with indexes that span hundreds of servers, while providing strong consistency guarantees. Third, it enables live index split and migration, background index creation and deletion, and non-blocking range lookups to ensure that large scale operations do no impact other operations. Finally, it uses main memory efficiently when storing secondary index structures.

To demonstrate the practicality of the design, SLIK was implemented in RAMCloud [34, 11], a low-latency distributed key-value store. This implementation of SLIK supports extremely low latency indexing and is highly scalable:

- SLIK performs index lookups in 11–13 µs, which is only 2× the latency of non-indexed reads in RAMCloud.
- SLIK performs durably replicated writes of indexed objects in 30–36 µs, which is also about 2× the latency of non-indexed durable writes in RAMCloud.
- The latency provided by SLIK is 5–90× faster than H-Store, a state-of-the-art in-memory database.
- As an index is partitioned among more and more servers, the throughput of index lookup in SLIK grows linearly while the latency remains nearly constant.

Overall, SLIK demonstrates that large-scale storage systems need not forgo the benefits of secondary indexes.

1.1 Contributions

The main contribution of this dissertation is the design and implementation of SLIK, which provides low-latency, scalable, consistent, durable and available secondary indexing in a memory-based key-value store. The implementation of SLIK in RAMCloud is available freely and open source [11]. Here is a summary of some of the interesting design contributions:

- SLIK uses a *multi-key-value* data model where each object can have multiple secondary keys in addition to the primary key and an uninterpreted data blob. This approach reduces parsing overheads for both clients and servers to improve latency.
- It achieves high scalability by partitioning indexes such that the index entries can be distributed independently from the corresponding objects, rather than colocating them (which is the more commonly used approach today).
- As a result of the partitioning scheme above, indexed operations are distributed, which creates potential consistency problems between indexes and objects. SLIK provides

clients with consistent behavior using a novel lightweight mechanism that avoids the complexity and overhead imposed by most distributed transaction implementations. It utilizes an ordered-write approach for updating indexed objects and uses objects as ground truth to determine liveness of index entries.

- SLIK performs long-running large-scale operations without blocking normal operations. For example, SLIK uses a logging approach for index migration, which allows updates to an index as it is being migrated.

- Finally, it implements secondary indexes using an efficient B+ Tree algorithm. Each tree node is kept compact by mapping secondary keys to the primary key hashes of the corresponding objects. SLIK further uses objects of the underlying key-value store to represent these nodes, and leverages the existing recovery mechanisms of the key-value store to recover indexes.

This dissertation also aims to explain the various design issues that have to be tackled while building an indexing system, and the approaches that can be taken to achieve the desired properties.

1.2 Dissertation Outline

The rest of this dissertation is organized as follows. Chapter 2 introduces the evolution and current landscape of data storage systems and positions SLIK with respect to other systems. Additionally, it provides an overview of RAMCloud, the underlying key-value store for my implementation of SLIK. It also motivates the need for secondary indexing. Chapter 3 describes the interface provided by SLIK to application developers. The next few chapters explain the various design issues encountered while building an indexing system, some possible approaches to tackle each issue, and the details of the approach chosen in SLIK: Chapter 4 explores various ways to partition indexes and how they impact scalability; Chapter 5 dives into the details for providing strong consistency even in the face of failures; Chapter 6 describes the logical structure of the indexes; and Chapter 7 explains how an in-memory indexing system can achieve durability and availability as well as how this affects the physical layout of the indexes. These design decisions come together to provide indexing, as summarized in Chapter 8 with the help of internal Remote Procedure Calls (RPCs) involved in implementing the index API. Chapter 9 benchmarks SLIK's implementation in

RAMCloud to evaluate how well SLIK meets its latency and scalability goals. While the discussion of related work is woven throughout the dissertation, Chapter 10 summarizes the related work. Finally, Chapter 11 concludes the dissertation.

Chapter 2

Background

Traditionally, relational databases, like MySQL, ruled the data storage world. While they operated at small scale (often on single machines or servers), they offered data models that made it easy for developers to build meaningful, complex applications. Their data models include features like secondary indexing, aggregation, and sort, which enable powerful queries. They also include features to support updates beyond simple writes and removes, such as transactions, foreign key checks, and triggers.

With the advent of very large web applications, the SQL based databases were no longer sufficient. These databases could not provide the needed scalability as they had been originally designed to operate at much smaller scale. Most of their features (like the ones mentioned above) require access to many objects, often in different tables, making it important for these accesses to be efficient. Given that they were designed to operate on data that fit on a single server, they weren't optimized for large-scale data that spanned hundreds or thousands of servers. As a result, some applications later started using these databases by partitioning their data across multiple database instances. However, this meant that the applications could take advantage of the database features (like transactions) only within a partition and not across partitions.

Thus, many NoSQL systems emerged to support the massive scale of web applications. However, in order to achieve this, many sacrificed higher level data models and strong consistency guarantees offered by relational databases (Figure 2.1 illustrates this trend). Datastores missing these features impose significant challenges on the application developer and the type of applications she can develop. The lack of consistency places the burden on the application developer to ensure correct behavior at the application level. The lack of

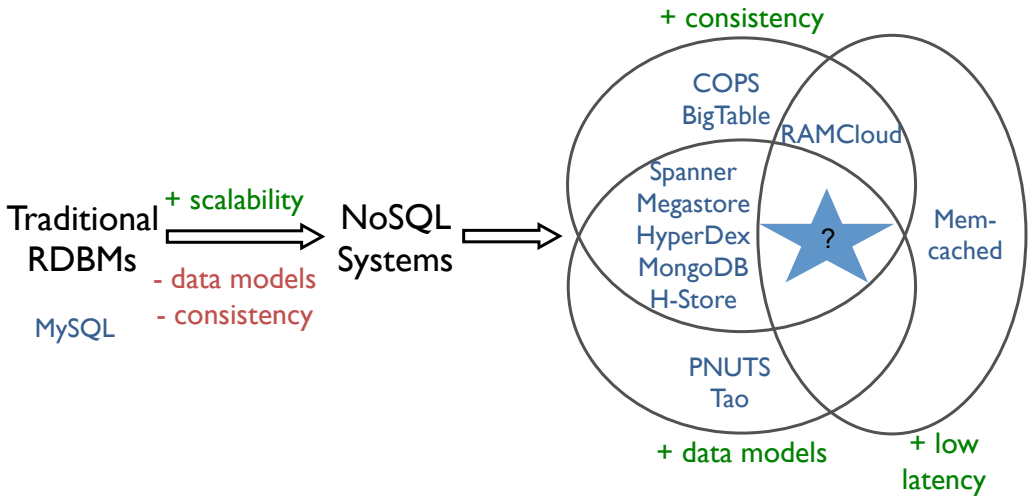


Figure 2.1: Evolution of storage systems from traditional databases to the current landscape. The blue star illustrates an unsolved problem: SLIK is a step towards filling that void.

secondary indexes forces the application developer to find creative ways to store and query data, which may limit the kinds of queries that can be done efficiently.

The community is now trying to add back some of the higher-level features. Figure 2.1 shows a few examples (the space is populated with many systems and the figure shows only a few of them).

Some systems offer stronger consistency but support only simple data models. For example, COPS [29] provides Causal+ consistency but has a simple key-value data model. RAMCloud [34] provides linearizability (the strongest consistency level) but has a key-value data model which supports writes, reads, scans, conditional operations and multi operations. BigTable [17] also provides strong consistency but stores data in a multi-dimensional sorted map which supports writes, reads and scans.

Some other systems provide weaker consistency guarantees but have richer data models. For example, Tao [16] is eventually consistent (weakest consistency level) and provides a graph database API (with nodes and associations between the nodes). PNUTS [18] has relaxed consistency but offers a basic relational data model.

Most current systems offer stronger consistency guarantees to some extent and one or more features of higher level data models. For example, HyperDex [21] is linearizable (the strongest consistency level) and supports a key-value data model with rich data types.

MongoDB [7] is also linearizable and offers document oriented storage with JSON-style docs. Megastore [15], Spanner [19] and H-Store [24] have different levels of strong consistency and their data models share characteristics with relational databases. Megastore allows the client to pick the level of consistency desired, and offers fully serializable ACID semantics within fine-grained partitions of data (in contrast, relational databases provide the same semantics across the entire data). Spanner is externally consistent, and offers semi-relational tables with general purpose transactions and a richer query language (this move towards higher level data models was in part motivated by the wide adoption of Megastore within Google). H-Store is strictly consistent and supports a row based relational model.

However, providing this feature set at scale often comes at the cost of latency. The operations in most of these systems require fairly high latency, often in the order of milliseconds or even seconds within a single datacenter. Some systems have been designed to provide low-latency access to data. For example, Memcached [6] is a distributed in-memory cache with a key-value data model. Applications often use it in conjunction with many MySQL instances: this improves the overall performance of the system by providing faster access to most recent data. RAMCloud (described earlier) is a distributed in-memory storage system (which durably stores data) and provides remote access to objects in 5 to 15 microseconds.

I wanted to see if it is possible to get the best of all worlds - scalability, low latency, and a rich data model - and to identify the limits if it is not possible. No current datastore offers all of these features, which left this an important unsolved problem (the star in Figure 2.1). I suspect that filling this void will enable a new class of applications or enable an easier way of programming the current applications. I decided to approach the problem by taking an existing scalable system with a weak data model and implementing a better data model.

This led me to design SLIK: Scalable Low-Latency Indexes for a Key-Value Store. I implemented SLIK using RAMCloud as the underlying storage system. RAMCloud is a distributed in-memory key-value storage system. It is designed for large-scale applications, operates at ultra low latency, and provides strong consistency. While SLIK was designed in the context of RAMCloud, I have attempted to make the design decisions that would be appropriate for any key-value-store developer trying to introduce secondary indexing to their system. Over the rest of this dissertation, I discuss the design decisions independently of RAMCloud where possible. I also explicitly point out how RAMCloud influenced some of our decisions and implementation details.

Next, this chapter provides an overview of RAMCloud, which will be helpful in understanding the implementation-specific details in the later chapters. More details about RAMCloud can be found in the paper that covers all the basics [34], its wiki [12], and its source code [11]. Then it presents a short historical context for the design and development of RAMCloud and SLIK. It ends with some additional background on the key-value data model and a general motivation for secondary indexing.

2.1 RAMCloud Overview

RAMCloud is a datacenter storage system with two main properties: low latency and large scale. In order to get the lowest possible latency, RAMCloud uses DRAM as the primary storage medium: all data is present in DRAM at all times. Further, we carefully designed the RAMCloud software to be efficient enough to exploit DRAM's latency advantage. In order to support large scale data (beyond the capacity of a single machine), RAMCloud aggregates the memories of thousands of servers into a single coherent key-value store.

RAMCloud uses secondary storage (like disks or flash) only to hold redundant copies for durability. When a server crashes, the data that was present in the DRAM of that server is recovered in the DRAM of other servers using a fast crash recovery mechanism.

The RAMCloud implementation is open-source and available freely [11]. The current implementation achieves end-to-end times of 4.7 µs for reading small objects and 13.5 µs for writing small objects in our test cluster of 80 nodes.

2.1.1 Data Model

The original data model of RAMCloud was a simple *key-value* store consisting of any number of objects. Each object has a variable-length key and a variable-length uninterpreted value blob. Objects are grouped into *tables*, which act as namespaces for sets of keys. A table may span more than one server: each partition of the table, stored on a separate server, is called a *tablet*.

In order to approximate even distribution of data across tablets, RAMCloud relies on hashing the keys of the objects. A table is represented as objects within a 64-bit hash space. To partition a table, its objects are divided into tablets that form contiguous, non-overlapping subsets of the hash space. The key hash of an object, along with the identifier for the table, determines which tablet (and thus, which master) the object belongs to.

`createTable(tableName)` ⟶ `tableId` Create a new table named `tableName` if it does not exist and return the identifier for the table.
`dropTable(tableName)` ⟶ `status` Delete the specified table. All the objects in the table are also deleted.
`getTableId(tableName)` ⟶ `tableId` Return the identifier for a table given its name.
`write(tableId, key, value)` ⟶ `status` Create or overwrite the object identified by *tableId* and *key*.
`remove(tableId, key)` ⟶ `status` Remove the specified object.
`read(tableId, key)` ⟶ `value, status` Get the value of the specified object.

Table 2.1: Summary of the core API provided by RAMCloud to client applications.

Table 2.1 shows the basic operations supported by the original version of RAMCloud. Additionally, RAMCloud also supports enumeration (scan all objects within a given table), `multiWrite`/`multiRemove`/`multiRead` (batched versions of the corresponding operations without any atomicity guarantees), and conditional versions of write/remove/read.

2.1.2 System Structure

Figure 2.2 shows the building blocks of a RAMCloud system. Each storage server is composed of two components. A *master* module handles read and write requests from the clients. It manages the main memory of the server in a log-structured fashion to store one

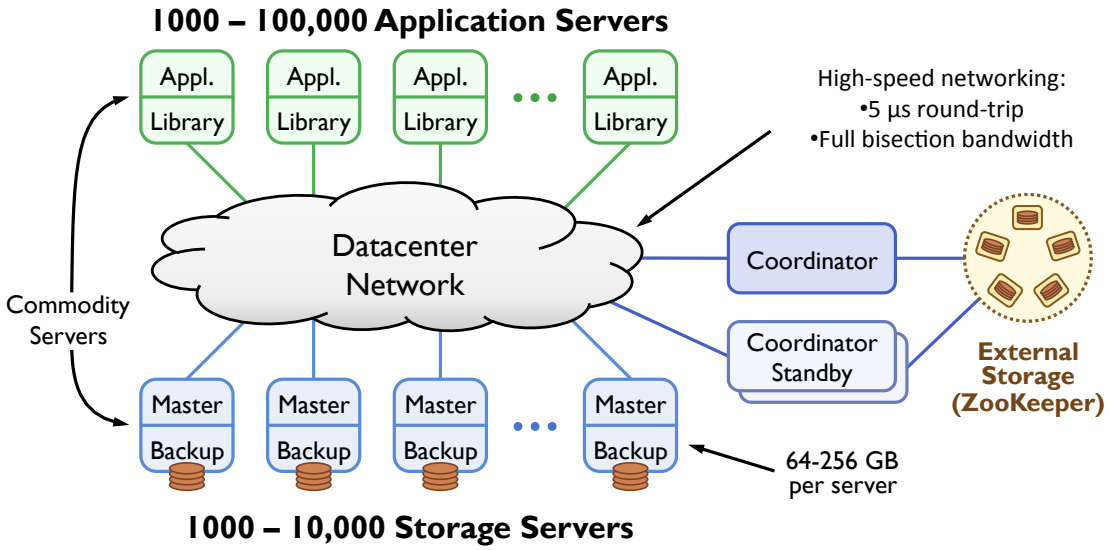


Figure 2.2: RAMCloud cluster architecture.

copy of all the objects in tables. A *backup* module uses local disk or flash memory to store multiple backup copies of log information. A backup is read only during the recovery of a crashed master or when restarting a cluster.

The masters and backups are managed by a central *coordinator*. The coordinator handles configuration-related issues, like server membership and information about which storage servers handle which table partitions. It is not normally involved in operations other than those querying or modifying configuration information.

The coordinator is a highly reliable and available system (with active and standby instances). While it appears as a single service to the rest of the cluster, in reality it is composed of multiple servers to avoid a single point of failure for the entire system. At any given time, only one of these servers is active: it acts as the coordinator and interacts with the rest of the cluster. The rest of the servers are standbys that actively follow the active coordinator's state. In case the active coordinator fails, one of the standbys is ready to take its place. A separate highly available configuration storage server is used to store consistent replicas of the active coordinator's state. It also determines a new active coordinator in case the current active coordinator fails. RAMCloud's coordinator is designed to support various external services. While the default is Zookeeper [23], the LogCabin [5] implementation of Raft [32] can also be used.

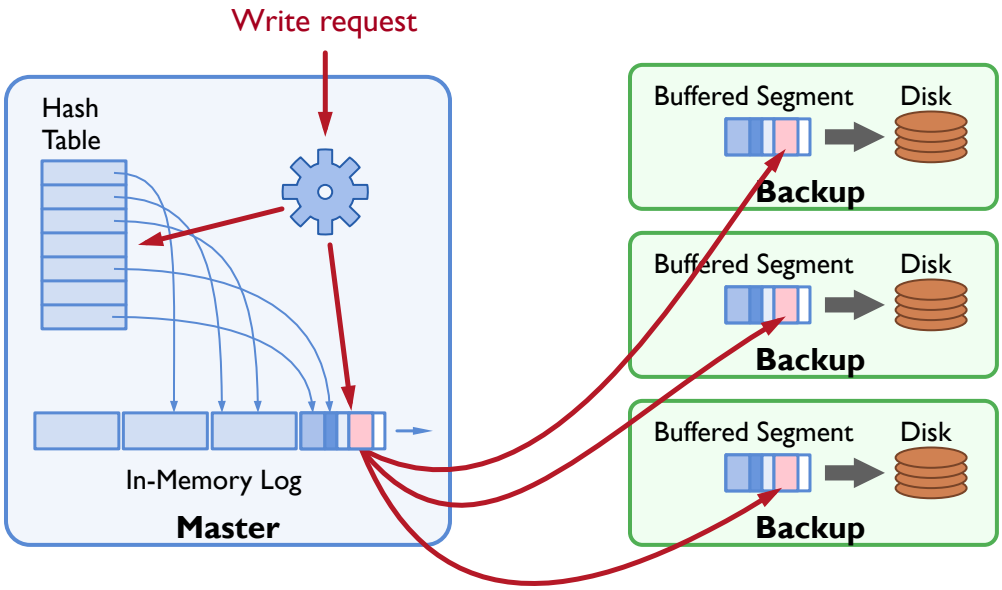


Figure 2.3: A master primarily consists of a hash table and an in-memory log. When it receives a write request, it updates its hash table and in-memory log. It then forwards the new data to multiple backups so that they can store it in secondary storage (disk or flash) for durability. The backups write that data in their non-volatile staging memory buffers and respond back to the master, which in turn responds back to the client. The data buffered by the backups is eventually written to secondary storage in large batches.

2.1.3 Log Structured Storage

RAMCloud uses a unified log-structured approach to manage objects in the main memory of masters as well as secondary storage on backups [37]. The log-structured approach allows in-memory storage to achieve a high (80–90%) memory utilization while still offering high performance.

With a log-structured approach, each master's memory is organized into a log and a hash table. The log is divided into small segments that store log entries containing objects. Log entries may also contain additional metadata and tombstones which indicate the removal of corresponding objects. The hash table is used to locate the objects in memory. It contains one entry for each live object stored on the master and allows objects to be located quickly given a table identifier and key.

When a master receives a write request (Figure 2.3), it appends the new object to its in-memory log and adds an entry to the hash table. It then forwards that log entry to several

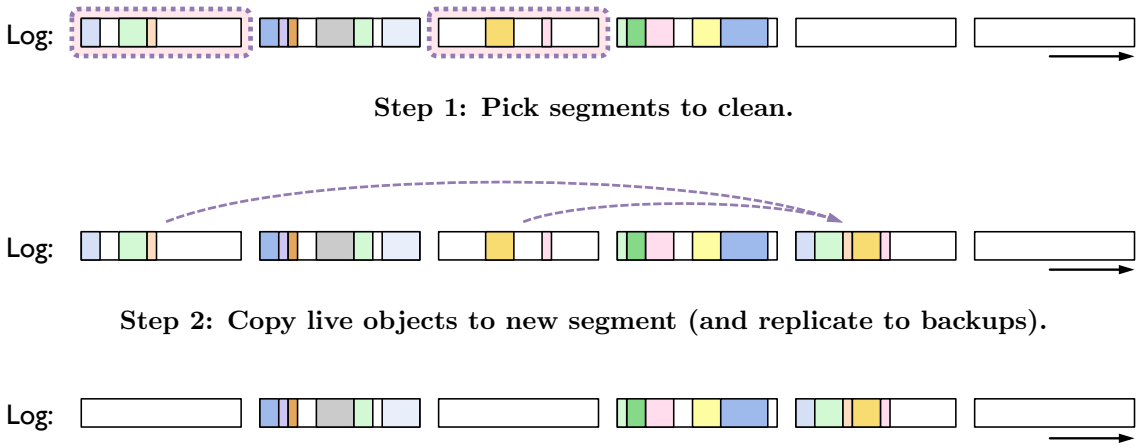


Figure 2.4: Illustration of the basic idea behind RAMCloud's log cleaning mechanism.

backup servers. The backups buffer this information in memory and return to the master. The master completes its request and returns to the client once all of the backups have acknowledged receipt of the log data. In the background, backups write the accumulated buffered data to disk or flash then delete the buffer from memory. The buffers are kept small and flushed regularly to disks (even if they do not fill up). This minimizes the amount of data lost in case of power failures. The loss can be prevented all together in a couple of ways. Many existing data centers already provide per-server or per-rack battery backups that extend power briefly after outages. The extended time with power can be utilized to flush any buffered data. Alternatively, the backups can buffer data in small NVRAM modules, which persist information even in case of power failures.

In order to avoid losing the buffered data in case of power failures, the buffers are kept small and flushed regularly to disks (even if they do not fill up). Additionally, many existing data centers already provide per-server or per-rack battery backups that extend power briefly after outages. Alternatively, the backups can buffer data in small NVRAM modules, which persist information even in case of power failures.

Given that the data in RAMCloud is organized using an append-only log, it needs a mechanism to reclaim the free space that accumulates in segments when objects are deleted or overwritten. Further this free space may be fragmented (as objects created at different

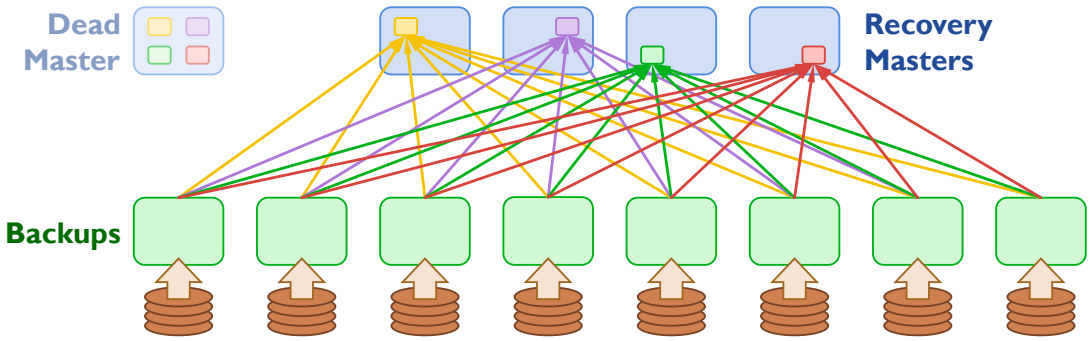


Figure 2.5: RAMCloud utilizes the scale of the cluster to enable fast crash recovery. Each master's data is scattered across many available backups to remove the disk bottleneck during recovery. A crashed master's data is recovered by multiple recovery masters to remove network and CPU bottlenecks. When a master crashes, the relevant data from all the backups is read by multiple recovery masters to recover the lost data.

times get deleted or overwritten). This fragmented space needs to be coalesced in memory and on backups so that new segments can be created to store more data. RAMCloud accomplishes this with a log cleaner [37]. The basic idea behind the log cleaner is shown in Figure 2.4: the cleaner selects segments to clean; reads the live data from the segments and rewrites them at the head of the log; it then deletes the cleaned segments along with their backup copies. The cleaner runs concurrently with normal operations and the cleaning barely affects the normal operations.

2.1.4 Crash Recovery

When a master crashes, the objects that had been present in its DRAM must be recovered. RAMCloud achieves this using a fast crash recovery mechanism [33]. The objects are recovered by one or more other servers, called recovery masters. At its core, the recovery happens by reading the log segment replicas from backups back into DRAM of the recovery master(s) and replaying the log to identify the current version of each live object and reconstruct the hash table.

The key to fast recovery is to take advantage of the massive resources of the cluster by combining the disk bandwidth, network bandwidth, and CPU cycles of many of machines in a highly parallel and tightly pipelined fashion (Figure 2.5). If each master's data had been replicated to a small number of backups, then disk bandwidth would become a bottleneck

during recovery. Hence, the master scatters segment replicas across all the available backups during writes. If all the data is recovered on one recovery master, then the network and the CPU of that machine become the bottleneck. Hence, the data of the crashed master is partitioned and each partition is recovered on a separate recovery master.

2.2 Towards Higher Level Data Models in RAMCloud

Our group started developing RAMCloud in 2009. We wanted to see how far we can push the boundary: what is the lowest latency we can provide in a highly scalable, strongly consistent storage system? In a few years we built RAMCloud and it achieved our goals. However, it only supported a key-value data model and simple operations like reads and writes. I wanted to ask: can I enable higher level data models while retaining the low latency and scalability properties of RAMCloud? After investigating other databases, I determined that if we added multi-object transactions and secondary indexing to RAMCloud, we could offer most of the features of traditional databases. Introducing these features is an important step to help us determine the extent to which we can recreate the high level features typical of traditional databases while maintaining the performance and scalability of a key-value stores. Over the next few years, we added linearizability and multi object transactions via RIFL [27] as well as secondary indexing via SLIK [25]. This dissertation is dedicated to a detailed discussion of SLIK.

2.3 The Need for Secondary Indexing

Secondary indexing is important because it allows application developers to query the data in the storage system in more meaningful ways. It allows data to be queried based on its various attributes, rather than just the primary key, as is the case with most standard key-value stores.

Figure 2.6 shows a toy example for a table of data that might be stored in a key-value store. It contains the records of books in a library as objects. The *primary key* of this table is the ISBN: this is the key that uniquely identifies the objects in the table. Key-value stores allow objects to be accessed via their primary key. So, using a standard read (supported by all key-value stores), a client can query for a book with a given ISBN to retrieve information about that book. However, the client might want to find “The Little Prince”, or all books

ISBN (Primary Key)	Title	Author	Year	Location
038533348X	Cat's Cradle	Kurt Vonnegut	1963	3409.102
0486284727	The Time Machine	H. G. Wells	1895	8948.389
0156012197	The Little Prince	Antoine de Saint-Exupéry	1943	5699.212
081297736X	A Man Without a Country	Kurt Vonnegut	2005	3409.121

Figure 2.6: A simplified example of a table that stores records of books in a library. Each row is an object, uniquely identified by the ISBN.

by the author "Kurt Vonnegut". The client might even want to find all books published between the years 1940 and 1970. To get the answers to these queries, the client could simply scan the table to read in all the objects and parse them to find the relevant objects. But this is not efficient with real tables (as opposed to our toy example) which can have millions of records.

To enable access to an object via attributes other than its primary key (as in the example queries above), we treat these attributes as secondary keys for that object. Further, we need additional data structures that organize information in a way that allows efficient lookups via these keys. These additional data structures are indexes on the secondary keys, and can be simply referred to as secondary indexes.

Chapter 3

SLIK Application Interface

This chapter discusses the indexing interface viewed by an application using SLIK. The goal of indexing is to enable the application to find relevant objects based on their secondary keys. An application may want to perform range lookups (for example, `find objects where ‘‘p’’ < given secondary key < ‘‘t’’`) as well as point lookups (for example, `find objects where given secondary key = ‘‘lily’’`).

In order to enable such lookups, SLIK needs to maintain index structures that map from secondary keys to corresponding objects. This means the clients and servers must agree on where the secondary keys are located in the object. Section 3.1 discusses how this is achieved and the resulting data model.

While later chapters are dedicated to the system-side design for various components required to provide indexing, Section 3.2 in this chapter shows the top-level Application Programming Interface (API) exposed by SLIK. Some of the operations described in this section can be large-scale long-running operations. These are discussed in further detail in the next section.

3.1 Object Format and Data Model

In order to have secondary indexes, clients and servers must agree on where the secondary keys are located in the object. In traditional databases, the server knows the schema for the table, and the exact structure of each record. Hence, a database server understands where the keys are located and finding them is not a problem. However, typically a key-value store treats the objects mostly as uninterpreted blob of data; only the clients understand a more

Figure 3.1: Schematic of a traditional key-value object format.

```
{KeyName0:Val0, KeyName1:Val1, KeyName2:{KeyName3:Val3, KeyName4:Val4}}
```

Figure 3.2: Schematic of an example for a JSON object format.

detailed structure of the objects. Specifically, as shown in Figure 3.1, each key-value object consists of a single primary key (to identify the object) and an uninterpreted value blob. This is because key-value stores are designed to favor simplicity and to provide flexibility in case of evolving schemas (as is the case with many web applications).

A commonly used approach is to store the secondary keys (to be indexed) as part of the object's value. The value blob is no longer uninterpreted as the server parses the entire object to find the secondary keys. To enable this, the servers and clients have to agree on a specific format for object values. For example, an application might store objects in a JSON format, as shown in Figure 3.2. Here, each index is associated with a particular named field. Several storage systems use this approach, including CouchDB [2] and MongoDB [7]. However, this introduces overheads: when a client writes an object, the server has to parse the entire object to find the secondary keys to be indexed before it can complete the operation. Thus, systems that use formats like JSON do not provide the lowest possible latency.

Given our objective of lowest possible latency in SLIK, I chose an object structure that directly identifies all the secondary keys as shown in Figure 3.3. Consequently, there is no parsing required to carry out index operations. I call this a *multi-key-value* format: an object consists of one or more variable-length keys, plus a variable-length uninterpreted value blob. The first key is the primary key: along with the table identifier, this uniquely identifies an object. The rest of the keys are for secondary indexes. These need not be

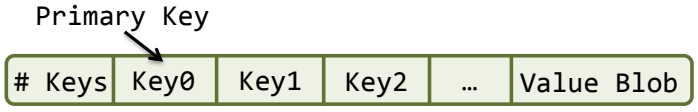


Figure 3.3: Schematic of multi-key-value object format.

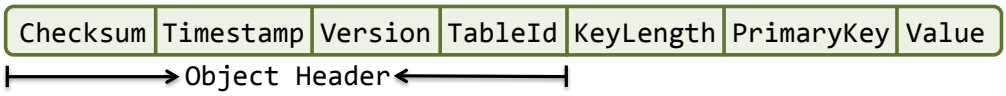


Figure 3.4: Detailed representation of RAMCloud's key-value object format before the introduction of secondary keys and indexing with SLIK.

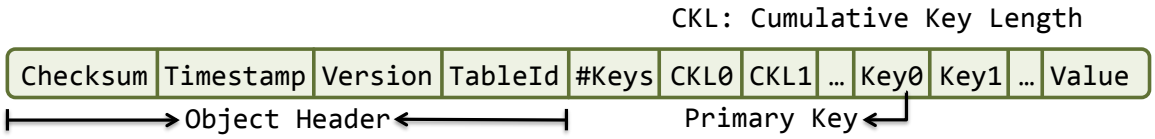


Figure 3.5: Detailed representation of RAMCloud's multi-key-value object format after implementing secondary keys and indexing with SLIK.

unique within the table. As with a key-value store, the value is an uninterpreted blob of data: a server never parses the value, it just stores and returns it as-is.

Each of the secondary keys is identified by its position in the object and can have an index corresponding to it. Each key can be of a different type. The type of the key specifies the ordering function for the corresponding index (for example, an integer key can be numerically ordered, or a string key can lexicographically ordered). An index is uniquely identified by the combination of the table identifier (`tableId`) and index identifier (`indexId`). For example, an index identified by `tableId` t and `indexId` n indexes the nth key in all the objects of table with `tableId` t.

SLIK's data model also offers flexibility as the objects in a table do not have to follow a strict schema. For example: different objects can have different numbers of secondary keys; an object can have secondary keys that are not indexed; an object can be missing a secondary key corresponding to an index on the table.

An actual object contains more information than the simplified object formats shown earlier. Figure 3.4 shows a more detailed view of the basic key-value format of objects in RAMCloud. The object header includes various pieces of metadata, such as `tableId` (which identifies the table the object belongs to), version (which is used to disambiguate different incarnations of objects with the same key), timestamp (which is the creation time of the object, primarily used by the cleaner to order live objects in order to improve cleaning performance), and a CRC32C checksum (which covers everything but this field, including

the keys and the value in order to detect corrupt data). The key length specifies the length of the variable length primary key. The rest of the bytes form the primary key (up to the length specified earlier) and the value blob.

The multi-key-value object must additionally specify secondary keys. Figure 3.5 shows a detailed view of the multi-key-value format from the implementation of SLIK in RAMCloud. It retains the object header from the basic key-value object (Figure 3.4). The number of keys (including the primary) is identified explicitly in the object. Since each object is required to have a primary key, this number is never less than one. Each object contains a blob of data which is composed of one or more variable-length keys and a variable-length value. To find a given key within this blob, we need its offset within this blob as well as its length: these are calculated using the cumulative key lengths (shown as CKL in the figure) specified in the object.

A cumulative key length is the total length of all keys up to and including the key corresponding to the given `indexId`. The length of each key can be calculated using the cumulative lengths: $Length_{key[i]} = CumulativeLength[i] - CumulativeLength[i-1]$. If an object doesn't contain a key corresponding to a certain id, then the length of that key should be 0. This means if key i is absent from an object, then $CumulativeLength[i] = CumulativeLength[i-1]$. Using cumulative lengths means that the offset for a key can be determined directly by the cumulative length of the previous key, and the length can be determined by a simple subtraction of two cumulative lengths. If an alternative implementation instead stored the length of the keys in the object, then computing the length would be trivial, but computing the offset would require adding the lengths of all the previous keys. Hence, the SLIK implementation saves a little bit of latency every time an object is parsed.

3.2 API

Table 3.1 summarizes the Application Programming Interface (API) of SLIK visible to client applications. A client invokes `createIndex` and `dropIndex` to create or delete an index (with `indexId`) on an existing data table (identified by `tableId`). Each index corresponds to a single table and a single secondary key within that table. The next subsection covers index creation and deletion in detail. Further, as an index gets large or highly loaded, it can be split into multiple partitions, each stored on a separate server. The final metadata

related operation, `splitAndMigrateIndexlet` enables this by splitting an index partition and migrating one of the resulting partitions to a different server. I will revisit this operation while discussing partitioning and configuration management in Chapter 4.

To write or overwrite an indexed object, a client sends `write` request to the data server that stores the tablet containing the object. Similarly, to remove an indexed object, a client sends `remove` request to the data server that stores the tablet containing the object. The server then updates (writes or overwrites or removes) the object as well as any secondary indexes associated with this object.

To find objects that fall within a given index key range, the client uses a streaming approach. It fetches objects one at a time in sorted index order. This mechanism is implemented with an `IndexLookup` class. Constructing an object of this class starts the query and `getNext` allows the client to iterate over the results. Using this streaming approach helps scalability. An alternative would have been to collect and return all the objects at once. However, this requires the client application to have enough space to store all the objects that will be returned in response to a lookup. As the number of objects returned increases, the client needs to have more and more space freely available for this purpose. For a very large query, the amount of data returned might exceed the storage capacity of the client. Hence, this alternative would not have been scalable.

3.3 Index Creation and Deletion

Creating an index typically requires modifying all objects in the table. This may happen for two reasons. First, the information to be indexed might be absent from the objects if new attributes or metrics are added. For example, a social networking site with a table for all the users (with one object corresponding to each user) might want to add a new metric to determine user influence: the number of times their profile is viewed. Second, the information to be indexed might be present in the value portion of the objects. The value of an object is often used to store all the information that is unlikely to be used for finding objects. Later if it appears that some attribute is useful for querying, then the application developer might want to move this information to an indexed secondary key. In both of these cases, the objects need to be reformatted to include the corresponding secondary keys – this process requires rewriting the objects. Rewriting all the objects in a a table can take a considerable amount of time, during which normal operations cannot be serviced.

`createIndex(tableId, indexId, indexType)` ⟶ `status` Create a new index for an existing table.
`dropIndex(tableId, indexId)` ⟶ `status` Delete the specified index. Secondary keys in existing objects are not affected.
`splitAndMigrateIndexlet( tableId, indexId, splitKey, newServerId)` ⟶ `status` For the index identified by `(tableId, indexId)`, split the index partition containing the key splitKey at that key. Migrate the second of the two resulting partitions to server identified by newServerId.
`write(tableId, keys, value)` ⟶ `status` Create or overwrite the object. Update secondary indexes both to insert new secondary keys and to remove old ones (if this was an overwrite).
`remove(tableId, primaryKey)` ⟶ `status` Remove the specified object. Update secondary indexes to remove the corresponding secondary keys.
`IndexLookup(tableId, indexId, firstKey, lastKey, flags)` Initiate the process of fetching objects whose keys (for index `indexId`) fall in the given range. `flags` provide additional parameters (for example, whether the end points of the range should be included in the search). This constructs an `IndexLookup` object.
`IndexLookup::getNext()` ⟶ `object` Return the next object in index order as per parameters specified earlier in `IndexLookup`.

Table 3.1: Summary of the core API provided by SLIK to client applications for managing indexes and secondary keys.

In order to allow the system to function normally during index creation, SLIK allows indexes to be inconsistent with objects (in a way that is temporary and invisible to clients). This makes it possible to have an index and yet have objects that do not have keys for that index. With SLIK, an empty new index is created: it is then populated in the background while other operations on the table can continue to be serviced normally. For example, reads (based on the primary key) or lookups based on other indexes can be serviced as before. Additionally, objects can be written into the table. These writes update the new index as well as the previously existing ones.

The index is populated by client-level code: it scans the table, reading each object and then rewriting it. Rewriting an object populates the indexes, including the newly created one, with entries corresponding to that object. Before rewriting the object, the client can restructure the object if the schema has changed. So, once all of the objects have been read and rewritten, the index is complete.

The index population operation is idempotent: if it is interrupted by a crash, it can be restarted from the beginning. When the client-level code scans the table next time, it will read and rewrite some objects that had already been rewritten earlier – this process does not change the objects in the table or the entries in the indexes. The code will also rewrite the rest of the objects (that had not been rewritten earlier), which will complete the population of the new index.

Index deletion behaves similarly to index creation. The index itself can be deleted while leaving all the objects untouched. Then, a follow-on step can scan the objects and rewrite them after restructuring them. The object can be restructured to remove the secondary key altogether (if that information is no longer needed) or move it to the value (if that information is needed but doesn't have to be indexed).

A recurring feature in SLIK is that it permits temporary inconsistencies in its implementation, while maintaining consistent behavior for applications. Index creation and deletion share this feature (as described above).

Chapter 4

Index Partitioning

To be usable in any large-scale storage system, a secondary indexing system must support tables so large that neither their objects nor their indexes fit on a single server. In an extreme case, an application might have a single table whose data and indexes span thousands of servers. Thus, it must be possible to split indexes into multiple index partitions, or *indexlets*, each of which can be stored on a different server.

An index should perform well even if it spans many servers: it should provide nearly constant and low latency irrespective of the number of servers it spans. Additionally, the total throughput of an index should increase linearly with the number of partitions.

To design an indexing architecture that achieves these scalability goals, I considered three alternative approaches to index partitioning.

4.1 Colocation Approach

One approach is to colocate index entries and objects, so that all of the indexing information for a given object is stored on the same server as that object. In this approach, one of the keys is used to partition the table's objects (and corresponding index entries) among servers. The partitioning key can be either the primary key or a given secondary key. As a result, each server stores a table partition (tablet), plus one index partition (indexlet), for each of that table's indexes. The indexlet stores only index entries for the tablet on the same server. Figure 4.1 illustrates this approach with an example. This approach is used widely by many modern storage systems, including Cassandra [26] and H-Store [24], and the local indexes in Espresso [36] and Megastore [15].

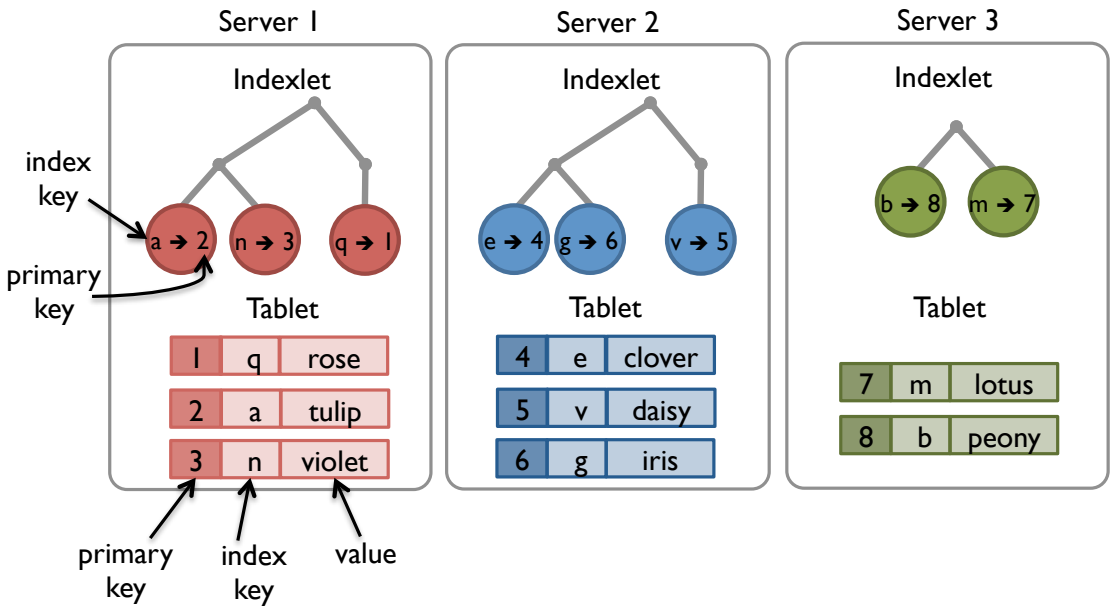


Figure 4.1: **Colocation Approach:** In this approach, indexes for a table are partitioned so that the index entries for each object are on the same server as the object. This example assumes that the table is partitioned by the primary key. Colors are used to distinguish objects (and secondary index keys) that belong to different tablets.

Tablet & Indexlet	**Master**
For objects with *primary key* < 4	Server 1
For objects with $4 \leq$ *primary key* < 7	Server 2
For objects with $7 \leq$ *primary key*	Server 3

Figure 4.2: Colocation Approach: Example of metadata for table and index partitions showing their placement in the cluster. This metadata corresponds to the example in Figure 4.1.

With this approach, only the server holding the relevant partition has to be involved in a write even if the object contains indexed secondary keys. To write an indexed object, the client issues a write RPC to the server that owns this object. The server then writes the object in its tablet as well as all the corresponding index entries in the indexlets located on the same server.

To perform an index lookup on a key other than the primary key, the client has to issue RPCs to all the servers holding partitions for this table and its indexes. Each server scans its indexlet, then returns the matching objects from its local tablet.

This approach enables low latency for lookups at small scale as it requires only a single set of parallel RPCs. For example, in the limit of a single partition (for the table and index), colocation requires a single RPC. However, as the scale gets larger, the performance for lookups with this approach degrades: the latency for lookup increases linearly with the number of servers across which a table (along with its indexes) is sharded. The latency increases because there is no particular association between index ranges and partitions, causing each index lookup operation to contact every server storing an indexlet for the table. Moreover, the total lookup throughput of an index does not scale with the number of partitions, as each server must be involved in every index lookup.

4.2 Independent Partitioning

The second approach is to partition each index and table independently, so that index entries are not necessarily located on the same servers as the corresponding objects. This allows each index to be partitioned according to the sort order for that index. Each resulting indexlet and tablet can be placed on any server in the cluster. Figure 4.3 illustrates this with an example.

With this approach, an indexed object write involves multiple servers: the object itself is written by the server that hosts the relevant tablet, and the index entry for each secondary key is written by the server that hosts the relevant indexlet. It is possible for a client to issue parallel RPCs to the data server and the relevant index servers to complete the write/insertion. There are two different ways in which a write could proceed. The first way is for the client to coordinate the operation by issuing parallel RPCs to data server and the index servers. The other way is for the client to issue the write RPC to the data server which then coordinates the operation by writing the object and issuing RPCs to the index

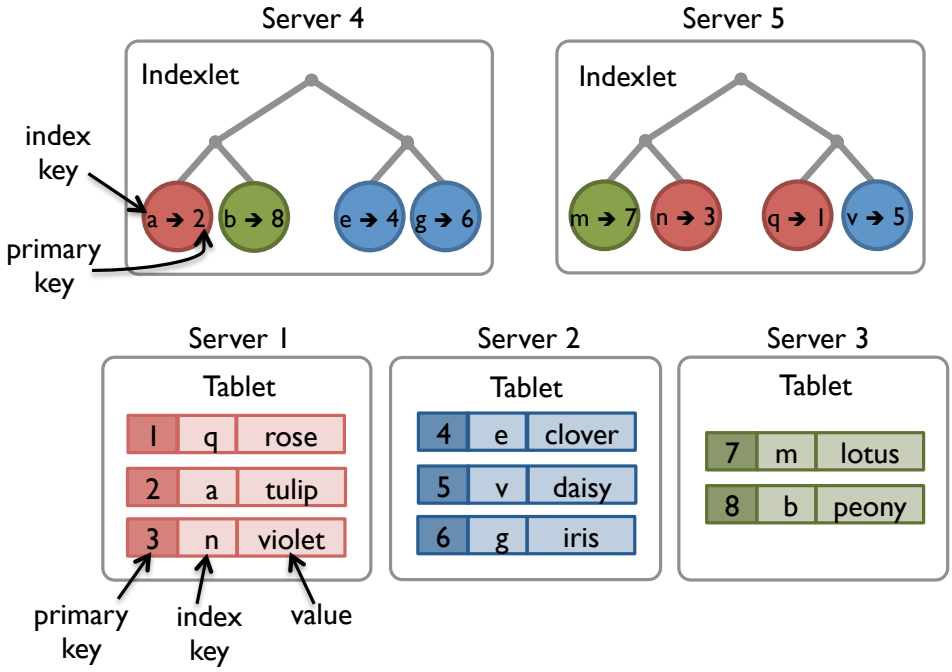


Figure 4.3: **Independent Partitioning:** In this approach, indexes are partitioned so that each indexlet contains all the keys in a particular range. This example assumes that the table is partitioned by the primary key. Colors are used to distinguish objects (and secondary index keys) that belong to different tablets.

Tablet	**Master**
For objects with *primary key* < 4	Server 1
For objects with $4 \leq$ *primary key* < 7	Server 2
For objects with $7 \leq$ *primary key*	Server 3

Indexlet	**Master**
For objects with *secondary key* $< h$	Server 4
For objects with $h \leq$ *secondary key*	Server 5

Figure 4.4: Independent Partitioning: Example of metadata for table and index partitions showing their placement in the cluster. This metadata corresponds to the example in Figure 4.3.

servers. The exact mechanism can affect consistency, and is discussed in further detail in the next chapter.

To perform an index lookup, the client has to issue two sequential RPCs. First, it issues an RPC to the server holding the relevant indexlet. A small index range query can be processed entirely by a single index server. The server returns information to identify objects. The client then issues parallel RPCs to the data servers to retrieve these objects.

At small scale, independent partitioning results in higher lookup latency than the colocation approach due to the need for two sequential RPCs. For example, in the limit of a single partition (for the table and index), independent partitioning results in twice the latency of the colocation approach. However, as the scale gets larger, independent partitioning offers dramatically better performance than colocation. Performing two sequential RPCs results in a constant latency even as number of partitions is increased, and this latency is lower than doing a large number of parallel RPCs. Moreover, the total lookup throughput scales linearly with the addition of servers because different indexlets process different queries independently.

4.3 Global Secondary Indexing

The third approach partitions data as in independent partitioning (discussed in the previous section), but fully or partially replicates table data in each index. Any data that may be accessed via an index needs to be duplicated in that index. Figure 4.5 illustrates this with an example. This approach is used by the global indexes in DynamoDB [3] and Phoenix [10] on HBase [4].

The mechanism to write an object with this approach is similar to the mechanism with independent partitioning. However, because the index entries store a copy of the data, it is possible that the consistency mechanisms required for this approach may not be the same as the previous approach (this will be further clarified in the next chapter).

To perform a lookup, a client issues an RPC to the server storing the relevant index partition. The server then finds the matching entries and returns the attributes stored in those entires (i.e., the *projected* attributes).

Global indexing enables better lookup latency than the independent partitioning approach while still providing the same scalability benefit. Global indexes do not require a two-step mechanism for lookups because the table data is replicated with the indexes.

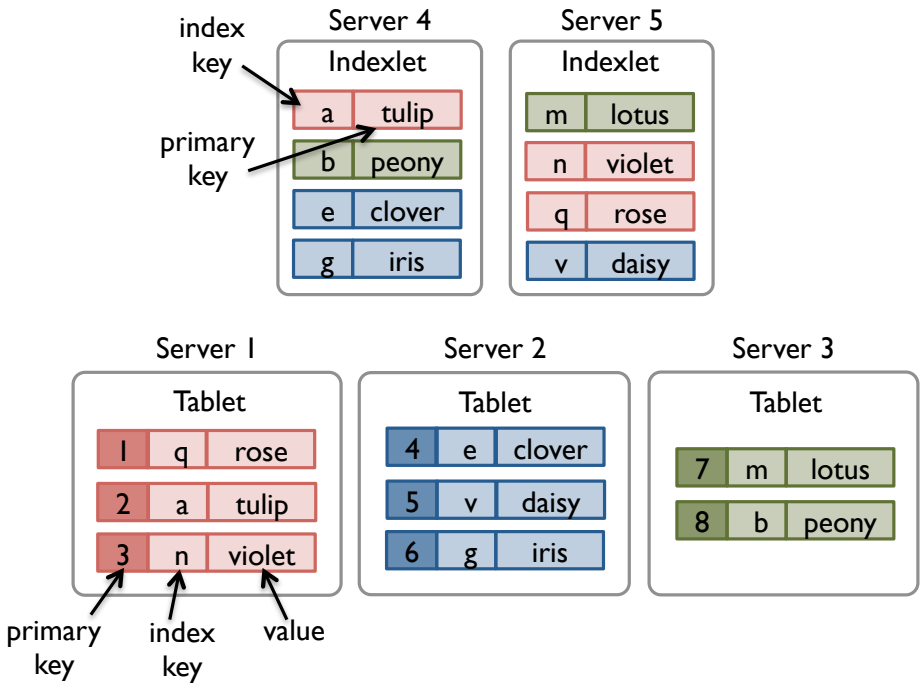


Figure 4.5: **Global Indexing:** In this approach, each index entry contains a full or partial copy of the corresponding object. Further, indexes can be partitioned independently of each other and the table. Thus, each indexlet can hold a contiguous range of keys. Colors are used to distinguish objects (and secondary index keys) that belong to different tablets.

Tablet	**Master**
For objects with *primary key* < 4	Server 1
For objects with $4 \leq$ *primary key* < 7	Server 2
For objects with $7 \leq$ *primary key*	Server 3

Indexlet	**Master**
For objects with *secondary key* $< h$	Server 4
For objects with $h \leq$ *secondary key*	Server 5

Figure 4.6: Global Indexing: Example of metadata for table and index partitions showing their placement in the cluster. This metadata corresponds to the example in Figure 4.5. The metadata in this example is the same as that for independent partitioning approach (as shown by the metadata in Figure 4.4 for the example in Figure 4.3). This is because global indexing allows indexes to be partitioned independently of each other and the table, just like the independent partitioning approach.

This allows global indexes to achieve lower latency than the independent partitioning approach. Additionally, global indexes are scalable because they are partitioned using the same mechanism as independent partitioning.

These benefits come at the cost of increased memory footprint. As an index lookup can return only those attributes of the object that are projected and stored with that index, a substantial amount of data might be duplicated to enable meaningful lookups. While this might be acceptable for disk or flash based systems, it may not be for memory-based systems because the storage medium is more expensive.

4.3.1 Usage in Other Systems

Some systems that use global secondary indexing, use it in addition to the colocation approach: they have two sets of indexes, which are used for different purposes. The first set is colocated with the objects and provides strong consistency. The second set is global and provides weak consistency in favor of lower latency.

Further, some systems organize the global indexes using the hash of the index key rather than the key itself. However, this makes it inefficient to perform range queries on the index as querying a range would require a scan over the entire index.

4.4 The Right Approach for SLIK

To yield scalable performance, SLIK uses the independent partitioning approach and partitions each index based on its key ranges, even though many modern datastores use the colocation approach and global indexes. This chapter explained why the independent partitioning scheme should lead to better scalability than the colocation approach. The experiments in Chapter 9 validate this reasoning. Additionally, while global indexing allows good scalability, it substantially increases the memory footprint which makes it not viable for a memory-based system like SLIK.

4.5 Identifying Objects With Our Approach

Index entries need a way to identify the objects they refer to while providing the mapping from secondary keys to the objects that contain those keys. If the objects were located on the same server as the indexlet, then the index entries could have mapped the indexed keys

directly to the location of the objects (using memory addresses for example). However, because SLIK uses an independent partitioning approach, objects and the corresponding index entries are likely to be located on separate servers. Hence, index entries need a different way to identify the objects they refer to. A straightforward way is to use the primary key of the corresponding object. However, primary keys are variable length byte arrays, which can potentially be large (many KBs). So instead SLIK identifies an object with a 64-bit hash value computed from its primary key. Primary key hashes have the advantage of being shorter and fixed in size. A compressed form of the key, such as a hash, works just as well as using the entire key, as it finds the right server and does not miss any objects. It may occasionally select extra objects due to hash collisions, but these extra objects get pruned out as a by-product of the consistency algorithm (Chapter 5).

4.6 Metadata and Coordination

Clients and servers need a way to identify destination servers for the RPCs used to implement various index operations. Specifically, they need metadata that lists the boundaries of tablets that compose each table and the mapping from these tablets to their host servers. Similarly, they need metadata that lists the boundaries of indexlets that compose each index for each table and the mapping from these indexlets to their host servers. This metadata is updated when a new table or index is created or dropped, a server crashes or recovers data from another crashed server, and when a tablet or indexlet is split or migrated to another node. Any underlying key-value store already manages and disseminates the metadata for tables. If an indexing system used the colocation approach, no additional work would be needed because index entries are colocated with the objects they refer to. However, given that SLIK partitions indexes independently from the tables, it needs to additionally manage the metadata for indexes and disseminate that information to clients.

RAMCloud has a coordinator that manages and disseminates the metadata related to cluster configuration, including the mapping from tablets to host servers [34]. SLIK uses the same coordination service for indexing metadata like the mapping from indexlets to host servers: the RAMCloud coordinator was modified to additionally store and disseminate this information. When a client or server accesses a table or an index corresponding to a table for the first time, it queries the coordinator for the configuration of the table and all the indexes for that table. It also caches this configuration locally to identify the destination

servers for RPCs in the future. If the cached configuration becomes stale, the client library detects this when it sends a query to a server that no longer stores the desired tablet or indexlet. The client then flushes the local configuration for the corresponding table or index from its cache and fetches up-to-date information from the coordinator. The coordinator only stores and disseminates the metadata: it does not take part in any lookup or write operations.

4.7 Indexlet Reconfiguration

Indexlets need to be reconfigurable, that is, we should be able to migrate an indexlet from one server to another, or split an indexlet if it gets too large and migrate one of the resulting partitions. This requires moving an entire indexlet or a large chunk of an indexlet from one server to another. Moving such a large amount data between servers can take a significant amount of time.

A straightforward approach to ensure consistency during reconfiguration is to lock the indexlet, copy the relevant part to another server, and then unlock. However, as a result, any other operations accessing the indexlet are blocked for the entire duration of this operation. The blocked operations include index lookups that would involve the locked indexlet and any object updates (writes/overwrites/deletes) where either the old or the new object value contains a secondary key indexed by the locked indexlet. As indexlets get larger, so does the amount of data blocked. This means that a concurrent operation has a higher probability of trying to access the locked data and hence being stalled until the reconfiguration is complete. This degrades performance at large scale.

SLIK uses a different approach (see Figure 4.7): it copies the relevant portion of the indexlet to another server while allowing other operations on it to proceed concurrently. SLIK uses a log to keep track of the mutations that have occurred since it started the copying process. Since RAMCloud uses a log structure to organize the objects in its memory, we can simply use that log for migration in our implementation of SLIK in RAMCloud. Once the migration process has copied over all the data including the mutations from the log, it acquires a lock for a short duration, while copying over the last mutation(s) at the head of the log (if any). This is similar to approaches used in the past for applications such as virtual machine migration [31] and process migration [39].

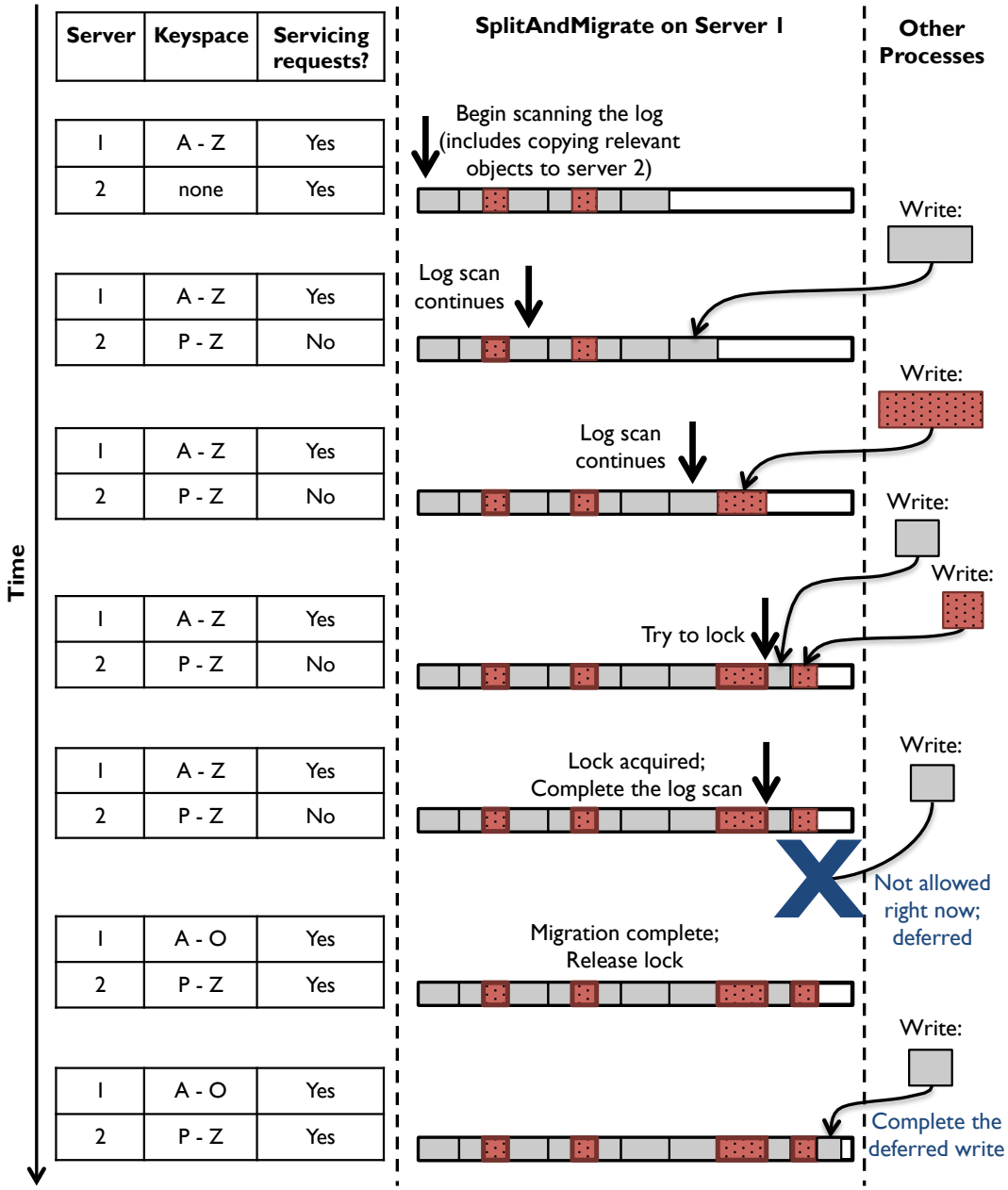


Figure 4.7: Timeline for splitting an indexlet and migrating one of the resulting partitions to a different server. In this example, server 1 hosts an indexlet for a given secondary key in the range A to Z. This indexlet is to be split such that the indexlet for range P to Z is migrated to server 2. The leftmost panel shows the top-level metadata; the central panel shows the log on server 1 and the splitAndMigrate process; the rightmost panel shows concurrent writes to this log by clients or other servers. Within the log, the information corresponding to the indexlet to be migrated is shown in red and the rest is shown in grey.

Chapter 5

Consistency

As discussed in the previous chapter, indexed object writes and index lookups are distributed operations because objects and corresponding index entries may be stored on different servers. This creates potential consistency problems between the indexes and objects.

The consistency problems could temporarily affect client requests or permanently corrupt data. For example, it might be possible for a client to read an object using its primary key but be unable to find the same object using a secondary key. This could happen if the object is written successfully but the index entry insertion is delayed (due to lost network packets or slow servers), or if a server crashed after writing the object but before inserting the index entry. This inconsistency could be temporary if the index entry is eventually written or permanent if it never gets written. Another example is that it might be possible for a client to lookup objects with a particular value for a secondary key, but to find an object that contains a different value for that secondary key. This could happen in many different ways as well. One possibility is that two clients were concurrently writing objects with the same primary key and different secondary keys and values; certain interleaving of the steps could result in one version of the object on the data server but the index entries for the other version on the index servers.

Many large scale storage systems permit inconsistencies in order to simplify their implementations or improve performance. For example, CouchDB [2], PNUTS [18], the global indexes for Espresso [36] and Megastore [15], and Tao [16] use various levels of relaxed consistency. For example, Espresso is timeline-consistent, CouchDB and Tao are eventually consistent. This forces application programmers to build their own mechanisms to ensure higher levels of consistency.

Our goal is to provide strong consistency, similar to the consistency expected from a centralized system, while still ensuring scalability of the distributed system. In the rest of this chapter, I first discuss the properties required for strong consistency and how SLIK ensures these properties. Then I describe one situation under which SLIK accepts more relaxed consistency in favor of scalability and offer alternatives that can be implemented if this relaxation is unacceptable for a system. Finally, I discuss additional consistency issues that can arise if servers crash during operations and how SLIK ensures consistency in the face of crashes.

5.1 Basic Consistency

SLIK provides clients with the same behavior as if indexes and objects were on the same server with locks to control access. More concretely, SLIK guarantees the following consistency properties:

1. If an object contains a given secondary key, then an index lookup with that key will return the object.

2. If an object is returned by an index lookup, then the object contains a secondary key for that index within the range specified in the query.

Figure 5.1 illustrates these properties with an example. There is a caveat to these properties when performing range queries: this is discussed further in Section 5.1.3.

These consistency properties guarantee that indexed object writes, overwrites and deletes as well as point lookups are linearizable. Linearizability is a safety property concerning the behavior of operations in a concurrent system. A collection of operations is linearizable if, to a client, each operation appears to occur instantaneously and exactly once at some point in time between its invocation and its completion. This means it must not be possible for any client of the system, either the one initiating an operation or other clients operating concurrently, to observe contradictory behavior.

5.1.1 Using Transactions

A straightforward approach is to wrap indexed object updates in transactions. Transactions are primitives supported by many storage systems that allow operations within a transaction to appear to have completed atomically. This means that a client can write the object and

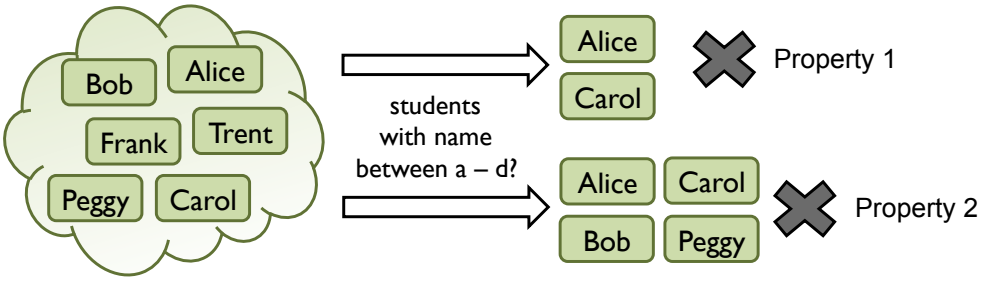


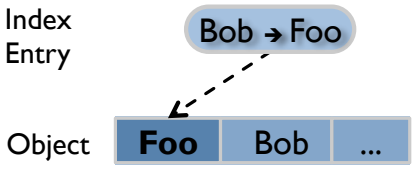
Figure 5.1: **Consistency Properties:** The first property ensures that the client does not miss an object, and the second ensures that the client does not get an extraneous object. Together these properties guarantee that the client views a consistent state of data.

the index entries within a transaction to ensure that the object and its index entries are updated atomically.

If transactions are used to implement indexed operations, the performance and scalability of indexed operations is directly dependent on that of the transactions it uses. For example, if the transactions implement an optimistic concurrency control (OCC) algorithm, the latency is comparatively low in an unloaded system but degrades quickly in a loaded system as the number of conflicts increase causing the transactions to abort. I observed this when we experimented with implementing indexed operations using the transactions in RAMCloud (which use an OCC protocol similar to Sinfonia [14] built on linearizability enabled by RIFL [27]).

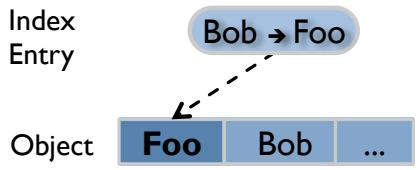
SLIK does not use transactions for a few reasons. First, many large scale storage systems don't support transactions, so an algorithm that doesn't use transactions is likely to be more widely applicable. Second, I was concerned that distributed transactions can be fairly complex to implement, and can result in scalability or performance bottlenecks if not designed and implemented well. However, it is entirely possible for another system to provide consistency for indexed operations using transactions and still use all the other design elements of SLIK. If this approach is used, I recommend using low latency and scalable transactions (for example, protocols described in Sinfonia [14] and SNOW [30] amongst others).

Step 1: Add index entry

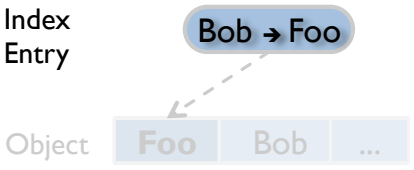


Step 2: Write object

Figure 5.2: Step by step illustration of the mechanism to ensure consistency when a new indexed object is written. The rectangular box shows an object, and the rounded box shows its index entry.



Starting state



Step 1: Remove object

Step 2: Remove index entry

Figure 5.3: Step by step illustration of the mechanism to ensure consistency during an indexed object remove. The rectangular box shows an object, and the rounded box shows its index entry. The index entry and object that has been removed are shown in a lighter color to indicate their absence.

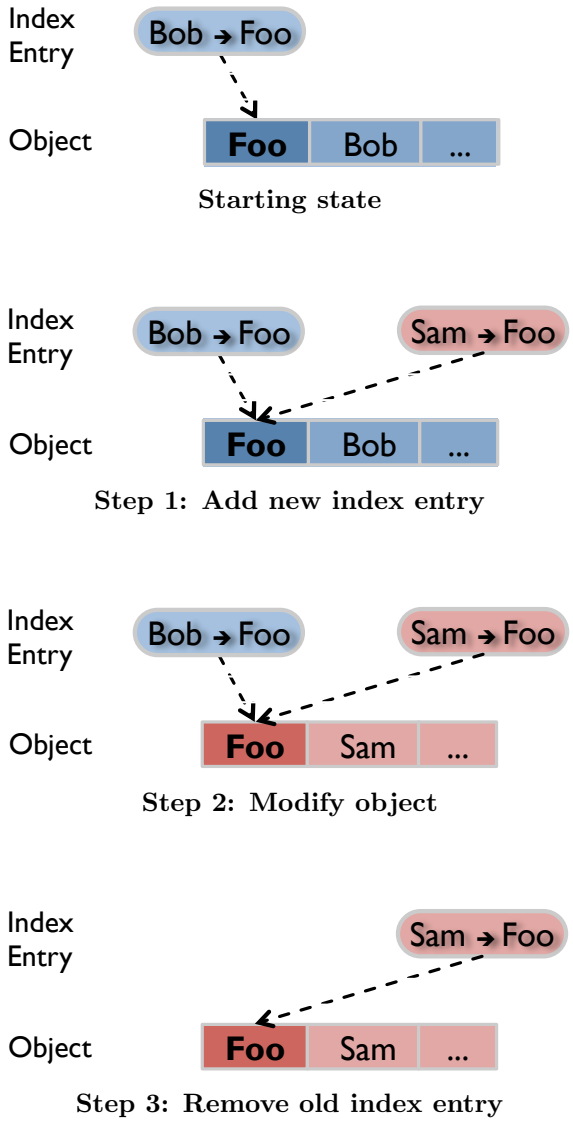


Figure 5.4: Step by step illustration of the mechanism to ensure consistency during an indexed object overwrite. In each step, the box at the bottom shows an object as it is modified from a version shown in blue to a different version shown in red. The rounded boxes above show the index entry (or entries) that exist at each step.

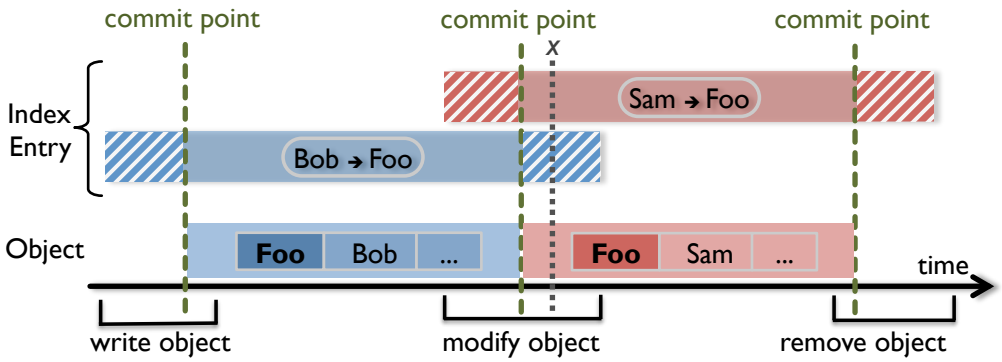


Figure 5.5: **The mechanism to ensure consistency:** The ordered write approach ensures that if an object exists, then the corresponding index entries exist. Index entries serve as hints; each object serves as the ground truth to determine the liveness of its index entires. Writing an object serves as the commit point. The box at the bottom shows an object as it is created, modified and removed (Foo is the object's primary key; the secondary key is changed from Bob to Sam when the object is modified). The boxes above show corresponding index entries, where the solid portion indicates a live entry. At point x, there are two index entries pointing to the object, but the stale entry (for Bob) will be filtered out during lookups.

5.1.2 A Lightweight Mechanism

I designed a simple lightweight mechanism that ensures the consistency properties stated earlier. It guarantees the first property by using an ordered write approach. It guarantees the second property by treating index entries as hints and using objects as ground truth to determine the liveness of index entries. This mechanism is explained in detail below and by Figures 5.2, 5.3, 5.4 and 5.5. Figure 5.2 shows, step by step, how an indexed object is written; Figure 5.3 shows how it is removed; and Figure 5.4 shows how it is overwritten. Figure 5.5 summarizes the algorithm for all indexed operations.

SLIK uses an *ordered write approach* to ensure that the lifespan of each index entry spans that of the corresponding object. Specifically, when a data server receives a write request, it first issues requests (to the server(s) with relevant indexlets) for creating index entries corresponding to each of the secondary keys in the new object's data. Then it writes the object and replicates it durably. At this point, it can respond back to the client (which acknowledges that the operation has completed). If this was an overwrite, it issues requests (again, to the server(s) with relevant indexlets) for removal of old index entries.

Finally, it releases the lock on the object being overwritten. When a data server receives a remove request, it acquires the same lock as above, removes the object and replicates this removal durably, and responds back to the client. Then it issues requests to the relevant index servers to remove old index entires and finally releases the lock. This means that if an object exists, then the index entries corresponding to it are guaranteed to exist – thus ensuring the first of the two consistency properties.

However, now it is possible for a client to find index entries that refer to objects that have not yet been written or no longer exist – this would violate the second consistency property. To solve this, I observe that the information in an object is the *truth* and index entries pointing to it can be viewed as *hints*. If an index lookup finds inappropriate objects, they can be filtered out by checking the actual index key in the objects. Specifically, to perform an index lookup, the client library first queries the index server(s) responsible for the requested secondary key or key range. These servers identify the matching objects by returning a hash of the primary key for each matching object (Section 4.5 discusses the use of primary key hashes in detail). The client library then uses these primary key hashes to fetch the corresponding objects from relevant data servers. Some of these objects may not exist, or they may be inconsistent with the index (see Figure 5.2 step 1, or Figure 5.3 step 1, or Figure 5.4 steps 1 and 2, or Figure 5.5 point x). The SLIK client library detects these inappropriate objects by rechecking the actual index key present in each object. Only objects with keys in the desired range are returned to the application.

Modifying an object effectively serves as a commit point – any index entries corresponding to the current data are now live, and any old entries pointing to it are now dead. Figure 5.5 illustrates the implicit commit points during write, overwrite and delete.

The SLIK approach permits temporary inconsistencies in internal data structures but masks them to provide the client applications with a consistent view of data. This results in a relatively simple and efficient implementation, while giving client applications the consistent behavior defined by the two properties above.

5.1.3 Caveat With Large Range Queries

There is a caveat to the consistency properties guaranteed by SLIK (discussed earlier) during range queries. If an object is overwritten concurrently during a range query, such that the secondary keys for both the versions fall within the lookup range, then either the old or new or both or neither version(s) of the object may be returned. Returning either

the old or the new version of the object is strongly consistent behavior because the update can be considered to have completed either before or after the lookup. However, returning both or neither versions results in non-linearizable behavior.

These discrepancies arise with certain interleaving of range queries with object updates. Let us consider an example. Say we have a table with one index corresponding to that table. The indexlet for range A to K is located on server 1 and for range L to Z is located on server 2. Now, a client (say, client X) performs a lookup on the index for range "A to U". Concurrently, another client (say, client Y) modifies an object such that its secondary key changes from Bob to Sam, as shown earlier in Figure 5.4. Client X first finds the primary key hashes from the indexlet on server 1, and reads the corresponding objects. If this happens during or before step 1 of the update, then client X finds the old version of the object. Then client X finds the primary key hashes from the indexlet on server 2, and reads the corresponding objects. If this happens during or after step 2 of the update, then client X also finds the new version of the object. By the end of this operation, this client has received the same object twice, with different secondary key values. If the update had been done such that Sam changed to Bob, then it is possible that client X would miss the index entry for Bob while getting the primary key hashes from server 1 (if this happened before step 1 of the update) and miss the index entry for Sam while getting the primary key hashes from server 2 (if this happened after step 2 of the update). In this case, the client performing the index lookup could miss the object altogether.

I considered a few approaches to remove this caveat to enforce strong consistency under all situations. One possibility is to lock the entire range being queried. However, this would block other write operations on this index, and by extension, on the table associated with this index. As this approach stalls other concurrent operations for a long period, it impacts scalability. Another possibility is to use Multi Version Concurrency Control (MVCC) to return a consistent snapshot of data (which may include older versions of some objects). With MVCC, multiple versions of each object are stored (only one of which is the latest version). It allows queries to be performed at a particular timestamp such that the objects returned form a consistent snapshot of the data at the given timestamp. However, MVCC is a fairly complex mechanism and adds overheads to normal operations. Given that RAMCloud (the underlying storage system for my implementation of SLIK) does not provide MVCC, my implementation of SLIK leaves out MVCC and accepts the caveat in consistency. However, if SLIK was to be implemented on another storage system that

provided MVCC, it could be beneficial to leverage that mechanism and close the caveat.

With this caveat, the consistency properties stated earlier in this chapter can be revised by adding the following exceptions:

1. During an range lookup, if an object is concurrently overwritten such that its secondary key changes from x to y where both x and y are within the range queried and $y < x$, then either one version of this object may be returned by the lookup or the object may not be returned at all.

2. During an range lookup, if an object is concurrently overwritten such that its secondary key changes from x to y where both x and y are within the range queried and $y > x$, then either one version of this object may be returned by the lookup or both the versions may be returned.

5.2 Consistency after Crashes

SLIK must ensure strong consistency even if servers or clients crash in the middle of an operation. The algorithm described earlier in this chapter ensures strong consistency when data is accessed concurrently. It also provides consistent behavior in case of crashes to a large extent. However, crashes can lead to additional complications related to consistency. This section describes the potential problems caused by the crash of each component and describes how they are handled.

5.2.1 Data Server Crash

Writing or overwriting an indexed object requires two or three separate updates: first to the index entries, then to the object, and again to the index entries in case of overwrite (as discussed earlier in Section 5.1.2). If one update completes but not the other, the system will be inconsistent.

These inconsistencies can arise in two different ways. First, a data server could crash after sending an RPC to insert the new index entry but before updating the object. If this index entry gets written, then it is extraneous as it does not refer to an object with that secondary key. This is illustrated by the state after step 1 in Figure 5.2 for writes and Figure 5.4 for overwrites. Such an extraneous index entry indicates the presence of an object with a given secondary key even though such an object does not exist. While

this does not lead to incorrect responses to the client (due to the consistency mechanism discussed in Section 5.1.2), it does cause the system to be internally inconsistent. When the client retries (because it never received a response that the operation was completed), the entire operation will be retried. The index entry written previously stays unnecessary. Second, the data server could crash after updating an object but before the old index entry has been removed. In this case, the old entry no longer refers to an object. This is illustrated by the state after step 1 in Figure 5.3 for removes and by the state after step 2 in Figure 5.4 for overwrites. If the server crashes before sending a response, the client will retry the operation and the old index entry will get removed. However, if the server crashed after sending a response, the client will not retry and the old entry will stay extraneous.

However, these inconsistencies are benign. The orphan entries do not affect correctness as they get filtered out during lookups by the consistency algorithm in the previous section. The only problem is that these entries waste memory. Let us assume (as a reasonable upper bound) that the mean time to failure for a server is about 4 months [22]. If 10 objects owned by this server are being written (or overwritten) at the time of its crash, and each object has a 100 B indexed key, then the total amount of garbage accumulated by this server will be less than 3 KB per year. I did not consider this to be an important drawback because it doesn't affect correctness and the wasted space is negligible as the garbage entries accumulate slowly.

Another system implementing SLIK's basic consistency mechanism might care about removing these inconsistent entries. If so, they can be garbage collected by a background process. Occasionally, this process scans the indexes and sends the entries to relevant data servers. For each index entry, the data server acquires a lock that prevents concurrent accesses to the corresponding object. It then checks whether the object exists. If the object does not exist, the data server sends a request for removing the index entry to the index server. If the tablet corresponding to an entry is being recovered, the collector skips that entry as it can be removed during the next scan.

5.2.2 Index Server Crash

Another consistency issue arises if an index server crashes while updating an indexlet. This can cause inconsistencies in the internal B+ tree structure corresponding to that indexlet. Inserting or removing an entry in the B+ tree can cause nodes of the B+ tree to be split or joined. Further, changes can propagate up the B+ tree to maintain its balance. This

requires updates to multiple objects that encapsulate the nodes being modified. By default, the multi-object updates are not atomic. This means if a server crashes in the middle of updating a set of nodes then the resulting B+ Tree will be in an inconsistent state.

In order to maintain consistency within the B+ tree across crashes, the changes to all the affected nodes need to occur atomically. SLIK uses a multi-object update mechanism implemented using the log-structured memory or transaction log of the underlying key-value store. This ensures that after a crash, either all or none of the updates will be visible.

This imposes an additional requirement on the underlying key-value store to allow atomic multi-object updates. If a system does not already support this feature, it will have to be implemented in order to support indexing with SLIK. For example, while implementing SLIK in RAMCloud, this mechanism had to be implemented in RAMCloud because it was not already provided by RAMCloud.

5.2.3 Client Crash

SLIK has been designed such that a client crash does not affect consistency. This is because SLIK does not rely on clients for correctness: any operations that have consistency issues (like write) are managed by servers. Consequently, a client crash does not require any recovery actions other than closing network connections between the servers and the crashed client. This is typically handled by low-level networking protocols and the underlying key-value store.

Chapter 6

Index Structure

The way indexing information is organized within each indexlet affects the types of requests that can be efficiently supported by that indexlet. In order to service a client's index lookup request, the relevant index server(s) have to find the index entries corresponding to a given secondary key or key range. Our goal is to support point lookups (for example, `find entries with key = ‘‘pam’’`) as well as range lookups (for example, `find entries with ‘‘p’’ < key < ‘‘t’’`). Further, when a client writes or removes an object containing one or more indexed secondary keys, the relevant index server(s) have to insert or delete the index entries corresponding to the keys in the object. The structure of the indexlet affects the time required to find an entry and the time to insert an entry, thus affecting the end-to-end latency for lookups and writes respectively.

One way to structure the index information is to use a hash table. Each entry in the hash table could map from the hash of the indexed key to the corresponding object. This results in a simple implementation and provides fast access times (O(1) for lookups and insertions if there are no collisions). However, a hash table can only support point lookups, not range lookups.

Hence, SLIK uses a tree structure to logically organize the information in an indexlet. This enables range queries apart from point lookups (with access times in O(log n) for lookups and insertions, where n is the total number of entries in the indexlet). In particular, SLIK uses a specific kind of tree, called a B+ tree, where the data is stored only in leaf nodes and the leaf nodes are linked in a list. This enables more efficient range searches: once we find the first key in the range, we can simply do a linear scan via the linked leaf nodes until we hit the end of the range.

It is possible that other structures might also be well suited to storing the indexlets, and this can be of interest to investigate further. SLIK is designed such that the structure of the tree can be changed without changing other design decisions; and its implementation is modular to allow the tree structure to be modified in the future without impacting any other implementation.

Chapter 7

Durability and Availability

While SLIK stores indexes in DRAM to service requests with lowest possible latency, our goal is that the indexes must be as durable and available as the object data in the underlying key-value store. This means that if a server crashes, the indexing system should recover lost index data in about the same amount of time that the underlying storage system recovers lost table data.

7.1 Possible Approaches

There are two reasonable approaches to ensure durability and availability after crashes. I call these the rebuild approach and the backup approach. With the rebuild approach, a crashed indexlet is reconstructed using object data from the corresponding table. This approach has the advantage of not imposing any overheads during normal write operations. However, it does not meet our crash recovery goal as the time to recover the indexlets is longer than the amount of time it takes the underlying key-value store to recover tablets. On the other hand, with the backup approach, a crashed indexlet is reconstructed from the backup copies of that indexlet. This approach increases the latency for normal write operations, but meets our goal for fast crash recovery.

7.1.1 Rebuild Approach

The first approach originates from the observation that the information in indexes is redundant: all the information stored in index entries is also present in the corresponding objects.

Thus, indexes don't need to be persistent as they can be reconstructed from objects in the tables. With the rebuild approach, indexes exist only in DRAM and are volatile.

When a server crashes, each indexlet that was present on that server is recovered on another server (*recovery master*) by using object data from the corresponding table. Each server storing objects for the table parses all the objects in its memory to find objects with secondary keys that belong to the crashed indexlet. Then it sends the relevant data to the recovery master. Finally, the recovery master reconstructs the indexlet structure using all data received from various servers.

This approach is attractive for two reasons. First, it is simple: indexlets can be managed without worrying about durability. Second, it offers high write performance: there is no need to replicate index entries or copy them to nonvolatile storage such as disk or flash.

However, the rebuild approach does not allow fast crash recovery. In the first step, data servers scan their memories in parallel (to find the pertinent objects). Assuming the servers have 250 GB memory with a memory bandwidth of 50 GB/s, this requires at least 5 s. As the memory sizes increase, this time will also increase. In the second step, the new owner of the indexlet reconstructs the indexlet. For a relatively small 500 MB indexlet with 50 B index entries, this step requires 20 s as the time to insert each entry is about 2 µs (verified by our tests). This time for crash recovery is very long. For our implementation with RAMCloud as the underlying key-value store, the index recovery time would be much slower than the recovery time for objects (RAMCloud's goal is to recover in 1–2 seconds).

7.1.2 Backup Approach

A different approach is to store copies of the index entries in nonvolatile storage on backups and use those copies for recovery on crashes. When an index server crashes, the indexlets on that server can be recovered by reading in the appropriate backup copies.

However, ensuring durability with this approach impacts the performance of normal operations like updating indexed objects. Whenever an object is updated, in addition to updating its index entries in DRAM on index servers, each of these entries must be replicated to secondary storage on backups. This additional step makes the index updates slower, thus increasing the end-to-end latency of indexed object (over)writes by about 10 µs.

Further, implementing a mechanism for backup and recovery of indexes can add a lot of complexity. Instead, we can leverage the persistence and replication mechanisms already

implemented by the underlying store for its object data. Each indexlet B+ tree is represented as a *backing table* in the underlying store. The backing table is just like any other table, except that it is not visible to clients and has a single tablet. This is discussed further in Section 7.2. With this mechanism, index crash recovery consists primarily of recovering the corresponding backing table. This is handled by the underlying key-value store. In RAMCloud, this recovery takes about 4–8 seconds for a 500 MB partition. Once the backing table becomes available, the indexlet is fully functional; there is no need to reconstruct a B+ tree or to scan objects to extract index keys. Thus, this approach allows indexes to be recovered just as quickly as objects in the underlying key-value store.

7.1.3 The Right Approach for SLIK

SLIK uses the backup approach because it achieves our crash recovery goal. As the backup approach uses the persistence mechanisms employed by the underlying key-value store and does not require any additional work to reconstruct the B+ tree, it allows lost index data to be recovered in about the same amount of time required by the underlying store to recover table data. This means that if any given server crashes, the time to recover the data that was on that server doesn't take any longer just because the system offers indexing.

7.1.4 A Case for Choosing the Other Approach

For a system that can tolerate longer recovery times, the rebuild approach might be a better choice. The main drawback of the rebuild approach is that crash recovery takes 25 s as opposed to 4–8 s. However, as a tradeoff, the write throughput and latency are much better. The throughput is higher because, compared to the backup approach, the rebuild approach requires lesser backup I/O, which is ultimately the limiting factor for throughput. The latency is lower because the index entries don't have to be replicated. I estimate writes would require about 20 µs with rebuild as opposed to about 30 µs with backup. This is particularly useful for systems with write-heavy workloads, or in systems where a lower write latency is more critical than lower crash recovery time. There could be different reasons that might make higher recovery times tolerable for a system. For example, a system that has online replicas (multiple copies in DRAM which are all used to respond to client requests) can continue servicing requests even during crash recovery, so crash recovery time does not affect availability.

Given that one of the goals of SLIK is to ensure index recovery that is no slower than object recovery, the backup approach remains the best option for SLIK. If a server crashes, it will not take any longer to recover because it served indexes apart from objects. For other systems it would be important to weigh the benefits of shorter recovery time versus lower write latency and select the best approach based on the systems' requirements.

7.2 Storing Indexes With Backing Tables

SLIK represents indexlet B+ trees as backing tables in the underlying key-value store in order to simplify the backup and recovery for index (as discussed earlier in this chapter). Each node of the B+ tree is represented with one object in the backing table. Pointers between nodes of the B+ tree are represented as keys in the key-value store. Traversing to a child node requires a lookup in the key-value store since each node in the B+ tree is encapsulated by a separate object in the backing table in the key-value store.

The use of backing tables has two additional advantages. First, it easily permits variable size nodes in index B+ trees. Many B+ tree implementations (such as MySQL/InnoDB [8]) allocate fixed size B+ tree nodes. This results in internal fragmentation when the index keys are of variable length (for example, strings, which are used commonly). This internal fragmentation increases memory footprint and replication overheads because the entire node must be replicated (including the unused bytes). Since key-value stores naturally permit variable-size objects, the nodes in SLIK's B+ trees can also be of variable size, which eliminates internal fragmentation and simplifies allocation. An earlier implementation of SLIK B+ trees used the Panthema STX B+ Tree open source package [9] which allocates fixed size nodes. The current implementation uses objects in the underlying store (RAMCloud) to allocate variable size nodes.

The second advantage of using the backing tables is simpler memory management. Since a single server may store both indexlets and tablets, the server's DRAM must be shared by these different purposes. If indexes were stored using an independent mechanism (e.g. by using malloc) there would need to be an additional mechanism to divide memory between the tables and indexes. Furthermore, the mechanism would need to allow for the division to adjust over time. For example, the table on a server might get dropped and it might recover indexlets from crashed servers, thus increasing the amount of memory allocated to index information. Using tables for indexlets means that the underlying storage system essentially

views indexes as tables and its memory management mechanism handles the memory for both indexes and tables. This also allows memory to move back and forth between various table and index partitions on a server automatically as needs change.

RAMCloud uses a log-structured approach to memory management [37], described in Section 2.1.3. With the implementation of SLIK, RAMCloud uses the same log for tablets and indexlets on a given server. The log-structured-memory allows the system to achieve high memory utilization (80–90%) while still offering high performance. This is not the case with most memory allocators [37]. Further, the log can be leveraged to simplify implementation of a few operations. For example, SLIK uses this log to implement atomicity for multi-node updates as discussed in Section 5.2. SLIK also uses the log to keep track of the mutations during an index split and/or migration as discussed in Section 4.7.

7.3 Implementation Details for Recovery With Our Approach

Our implementation of SLIK leverages RAMCloud's crash recovery mechanism [33] to ensure durability and availability of indexes. Index crash recovery first uses the existing table recovery mechanisms to recover the backing table for an indexlet. The underlying crash recovery mechanism coordinates how and when recovery occurs. This includes: handling detection of crashed servers; selecting servers that will recover data; initiating recovery; and retrying the recovery if a recovery master crashes during recovery. Once the backing table is recovered, SLIK needs to be able to interpret the table as indexlet. The only information SLIK needs in order to do so is the primary key for the object encapsulating the root node of the indexlet B+ Tree. In order to eliminate the need for this step, SLIK always assigns a fixed string (which is hard-coded in the system) as the primary key for the object that encapsulates the root of any indexlet B+ tree. So once a backing table has recovered, it knows exactly where to find the root node for the B+ tree. After all the index data has been recovered, SLIK updates the relevant metadata: the top-level index metadata on the coordinator is updated to specify the new owner for the recovered indexlet, and the metadata on the recovery master is updated to indicate that the recovery is complete and it can now handle requests for the recovered indexlet.

While RAMCloud can recover crashed tablets of any size within 4–8 seconds, indexlets should be no larger than 500 MB in size to ensure fast indexlet recovery. This is because when a tablet is larger 500 MB, RAMCloud splits the tablet during recovery and assigns

each sub-tablet to a different recovery master, so that all of the data can be recovered quickly (Section 2.1.4). Such splitting cannot be used for indexlet backing tables because the B+ tree structure requires all of the objects in the backing table to be present on a single server. A split during recovery is determined by diving the primary key hash space in half: this translates to the hash of the node ids for the indexlet B+ tree, which would essentially lead to random distribution of nodes in one split or another, thus rendering the B+ tree useless. While the implementation of RAMCloud's crash recovery has been modified to ensure that a backing table is not split during recovery, having a large backing table will increase the amount of time needed to recover it. So if an indexlet starts getting larger than 500 MB, it can be split using the algorithm described in Section 4.7 to split the indexlet based on the B+ tree structure.

If a client issues another operation while data is being recovered, this operation is deferred during recovery and handled internally once recovery completes via RAMCloud's retry mechanism. The client's request is processed as soon as the recovery is complete and then the client receives a response.

7.4 Summary

To ensure durability and availability, SLIK replicates copies of index entries to backups and uses them to recover from crashes. Although this incurs a cost in write latency, the improved availability makes this trade-off worthwhile. Further, SLIK uses the underlying key-value store to represent the indexlet B+ Trees. This allows SLIK to leverage the existing mechanisms of crash recovery and memory management.

Chapter 8

Implementation of Index Operations

The core design decisions together inform the system-side implementation of the SLIK API (Table 3.1 in Chapter 3). This chapter walks through the basic index operations (lookups and writes) to summarize the various steps involved. I use the internal RPCs (shown in Table 8.1) to make this description precise.

For an *IndexLookup* operation (from the API shown in Table 3.1), the SLIK client library acts as overall manager. Figure 8.1 illustrates the basic algorithm. A lookup requires a two-step approach because the index entries and objects are located on independent servers due to the partitioning scheme chosen to enable scalability (Chapter 4). First, the client issues `lookupKeys` to the appropriate index servers. The client identifies the appropriate index servers using the configuration information about index structures which includes a mapping from indexlets to their host servers (Section 4.6). Each index server finds the relevant entries in the B+ Tree corresponding to that indexlet (Chapter 6 and Section 7.2) and returns the matching primary key hashes (Section 4.5) to the client. Then the client issues `readHashes` in parallel to the relevant data servers to fetch the actual objects using the primary key hashes.

In order to make *IndexLookup* efficient in cases where the range queried is large, SLIK uses a concurrent and pipelined approach with multiple RPCs in flight simultaneously. If the number of hashes that match on a single server is very large, then returning all those hashes in a single response may not be a good idea for multiple reasons: first, it does not maximize the amount of work that can be parallelized, and second, the client could run

out of memory. So the client library code specifies the maximum number of hashes that can be returned in a single RPC (the current implementation sets this to 1000). If an index server has more matching hashes than can be returned in one RPC, then its response indicates that not all hashes have been returned and provides the next secondary key and primary key hash to be returned. This gives the client the information necessary to issue the next `lookupKeys` request to that server for the remaining hashes. Thus, the client may have to send multiple rounds of `lookupKeys` RPCs to get all the matching hashes. However, the client starts by sending only the first `lookupKeys` RPC. As soon as it receives a response to that RPC, it sends parallel `readHashes` RPCs to the relevant data servers to fetch objects. Each `readHashes` RPC requests many objects rather than one object at a time in order to increase the efficiency of the overall lookup. However, all objects to be returned by each data server may not fit in a single RPC; so the client may send multiple rounds of RPCs to each server until it receives all the objects from that server. As it receives the objects from various data servers, it prunes extraneous entries (as per the consistency algorithm in Chapter 5) and merges the rest such that the objects are sorted in index order to return to the client. The next round of `lookupKeys` RPC is pipelined with current round of `readHashes`; and the next round of `readHashes` starts as this round completes. Having many parallel and pipelined RPCs in flight can lead to complicated code and failure scenarios that might be hard to reason about: to keep this manageable, SLIK implements this code using a rules-based approach [38].

The *write* operation (from the API shown in Table 3.1) is managed by the data server that stores the tablet containing the object (also referred to as the master for this object). While the lookups can be managed by clients, writes are managed by servers because crashes during writes can affect consistency and it is more feasible to manage recovery for servers than it is for clients (Section 5.2). Figure 8.2 illustrates the basic algorithm. As with lookups, writes also require separate RPCs to update the object and its index entries due to the partitioning scheme chosen for scalability (Chapter 4). These RPCs are sent in a particular order to ensure consistency (Chapter 5). A client initiates the write or overwrite of an indexed object by sending a *write* request to the master. The master synchronously issues `entryInsert` RPCs to relevant index servers to add new index entries. The index servers insert the entries in the corresponding indexlet B+ Trees (Chapter 6, Section 7.2) and replicate them to backups (Chapter 7). Once all the index servers have completed this step, the master then modifies the object locally and replicates it to backups. At this point,

the master returns a response to the client, then asynchronously issues `entryRemove` RPCs to relevant index servers to remove old index entries. If the object is a new one that did not previously exist, then the index removal step is skipped.

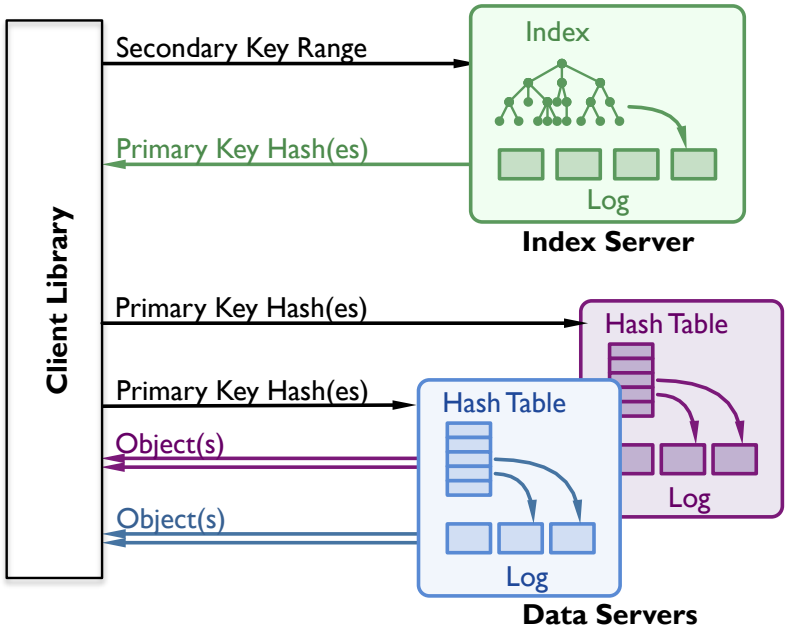


Figure 8.1: Simplified summary of the steps required to complete an index lookup operation.

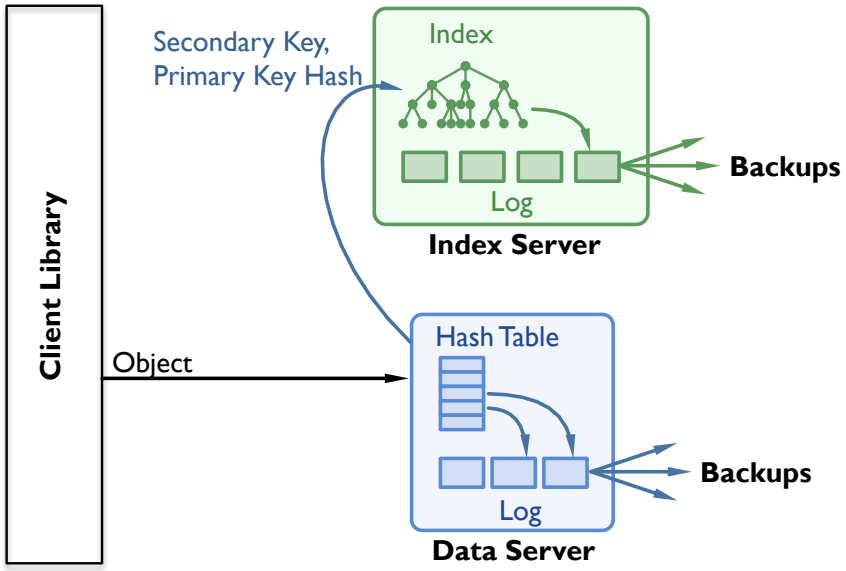


Figure 8.2: Simplified summary of the steps required to complete a write operation.

`lookupKeys(tableId, indexId, firstKey, lastKey, maxNumHashes)` `⟶ pKHashes, nextKey, nextHash` Returns primary key hashes (`pKHashes`) from the index entries corresponding to the given key range (`firstKey` to `lastKey`). In the case of a point lookup, `firstKey` and `lastKey` are the same. The primary key hashes are returned in the index sort order. That is, if the secondary key of object x is greater than the secondary key of object y, then the primary key hash for object x appears before that for object y in the `pKHashes` returned. `maxNumHashes` limits the number of hashes that can be returned. `nextKey` and `nextHash` specify the starting point for next `lookupKeys` to be issued by the client to this server.
`readHashes(tableId, pKHashes)` `⟶ objects, numHashes, numObjects` Returns objects in table (identified by `tableId`) whose primary key hashes match one of the hashes in `pKHashes`. The objects are returned in the same order as the corresponding primary key hashes in `pKHashes`. `numHashes` indicates the number of hashes corresponding to which objects have been returned. This allows the client to send follow up `readHashes` RPC to this server if not all hashes were handled by this response. `numObjects` indicates the actual number of objects returned, which helps the client parse the response.
`entryInsert(tableId, indexId, key, pKHash)` `⟶ status` Adds a new entry to the given index. This entry maps the secondary key (`key`) to a primary key hash (`pKHash`). Replicates the update durably before returning.
`entryRemove(tableId, indexId, key, pKHash)` `⟶ status` Removes the given entry in the given index. Replicates the update durably before returning.

Table 8.1: Summary of the core RPCs used internally by SLIK to implement the basic index operations: lookup, write and remove from Table 3.1. The RPCs for creating and deleting an index and reconfiguring index partitions are omitted here.

Chapter 9

Evaluation

This chapter evaluates the implementation of SLIK in RAMCloud to answer the following questions:

- *Does SLIK provide low latency?*
  Is it efficient enough to perform index operations at low latencies? Are the latencies comparable to other RAMCloud operations? How does the latency compare to other state-of-the-art systems?

- *Is SLIK scalable?*
  Does the choice of independent partitioning enable better scalability than the colocation approach? Does SLIK's performance scale as the number of servers increases? How does its scalability fare compared to other systems?

- *How does SLIK impact tail latency of operations?*
  How does the tail latency for indexed operations with SLIK compare to the tail latency for non-indexed operations in RAMCloud?

- *How does the throughput increase with the size of range queried?*
  How does the throughput for range queries increase as the number of objects returned by the query increases?

I chose H-Store [24] as the system for comparison with SLIK. I made this choice because H-Store and VoltDB (which is H-Store's commercial sibling) are in-memory database systems that are widely adopted.

In the early evaluations, I also benchmarked HyperDex [21], but eventually decided to drop HyperDex from the benchmarks. I had chosen HyperDex because it appeared to be one of the fastest existing large-scale storage systems with indexing based on the reported benchmarks [21] (which shows it achieves 12–13× lower latency and 2–3× higher throughput than popular systems like MongoDB and Cassandra). However, it turned out to be so much slower than SLIK that the comparisons were not interesting. HyperDex was designed to use disk for storage, which could explain its high latency. In order to improve its performance, we switched HyperDex to use RAM disk (with RAM disk, the software treats a block of DRAM as if it were a disk drive). This allowed HyperDex to take advantage of the lower latencies offered by DRAM even though the system's code is written to access disk. However, SLIK still significantly outperformed HyperDex in a variety of performance measures. Since H-Store performed more comparably to SLIK, I elected to retain H-Store and remove HyperDex from subsequent analysis. This allowed us to simplify the benchmarking process.

9.1 Summary of Results

Here are the key results from the evaluation:

- *SLIK provides low latency indexing:*
  - It performs index lookup operations in 11–13 µs (depending on the index size).
  - It performs indexed write and overwrite operations in 30–37 µs (depending on the index size).
  - Its performance degrades slightly with additional secondary indexes: writing an object with 10 secondary keys takes only 50% longer than writing an object with one secondary key.
  - SLIK is 15–91× faster than H-Store for lookups and 5–31× faster for overwrites (depending on H-Store's configuration).
- *SLIK indexes are highly scalable:*
  - As the number of index partitions increases, independent partitioning achieves lower latency and higher throughput than colocation.
  - The throughput of index lookup in SLIK increases linearly with the number of indexlets.

- Lookup latency in SLIK remains nearly constant even with increasing number of indexlets.
- Throughput of index lookup in H-Store increases sub-linearly with the number of partitions, while its latency does not remain constant.

9.2 Common Experimental Setup

We ran all experiments on an 80-node cluster of identical commodity servers. Table 9.1 shows the hardware configuration.

It was difficult to evaluate H-Store fairly because its behavior is controlled by a large number of tunable parameters, which result in significant variations in performance for different workloads. We were not able to identify a single setting of parameters that produced optimal performance across all workloads. In some cases, parameter settings that produced good performance for some workloads resulted in crashes for other workloads. With assistance from the H-Store developers [35], H-Store was tuned for each test to achieve optimum performance. For example, one of the parameters, `txn_incoming_delay`, controls the amount of time (in ms) the Transaction Queue Manager will wait before letting a distributed transaction acquire a lock on a partition. The default value is 5 ms. We reduced it to 0 ms for experiments where the servers were not loaded (to get the best latency), and 1 ms for the experiments where they were loaded (any lower resulted in many aborts and crashes). Further, H-Store uses the indexlet/tablet colocation approach to partitioning: the user must choose one column to use for partitioning, and the ideal choice varies from workload to workload. We benchmarked H-Store with multiple data partitioning schemes where applicable. These measures help ensure that H-Store is set up for its optimum performance for all benchmarks. As a result, however, the results probably overstate H-Store's performance, since in practice it may not be feasible to change the parameters for each type of query.

SLIK does not have tuning parameters like the ones mentioned for H-Store, so it is simply run as-is for all the experiments. RAMCloud allows the user to specify the number of replicas for each object, i.e., the number of backup servers which store copies for each object on secondary storage. For these experiments we use the default value of three, which means that each object (and thus each index entry) is stored in the flash drives of three different servers, apart from its primary copy in DRAM of the master.

CPU	Xeon X3470 (4×2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2× Crucial M4 SSDs CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 InfiniBand HCA
Switches	Mellanox MSX6036 (4× FDR) and InfiniScale IV (4× QDR)

Note: All nodes ran Linux 2.6.32 and were connected to a two-level InfiniBand fabric with full bisection bandwidth. The InfiniBand fabric supports 32 Gbps bandwidth, but PCI Express limits the nodes to about 24 Gbps.

Table 9.1: Hardware configuration of the 80 node cluster used for benchmarking.

9.3 Does SLIK Provide Low Latency?

We first evaluate the latency of basic index operations (lookups, writes, and overwrites) using a table with a single secondary index. We then evaluate the latency of object overwrites as the number of secondary indexes increases. We don't evaluate the latency of lookups in this case as it is independent of the number of indexes (lookups only access a single index at a time). In both of these cases, we ensure that the table and each index have a single partition. The behavior with increasing number of partitions is measured separately, in Section 9.4.

For these benchmarks, we used the default configuration of SLIK, and configured H-Store to achieve the lowest possible latency for the particular benchmark. By default, SLIK provides 3-way distributed replication of objects and index entries to durable backups. However, H-Store does not support replication. We also disabled the feature that enables durability in H-Store so that the cost of syncing to disk on every write does not impact our estimate of H-Store operation latency. This ensures fairness because RAMCloud's durability mechanism masks the latency of flushing data to disk on every write (as described in Section 2.1). Further, for SLIK, the table, the index and the backups are located on separate servers. In contrast, H-Store is run on a single server to prevent it from partitioning its data and index. If we allocated the same number of total servers as SLIK, H-Store would create that many partitions for all data, with one partition (for both table and index) on each server. Running H-Store on a single server is the only way we could find to ensure a single partition. In summary, in these benchmarks, H-Store executes all reads and writes

locally and no data needs to be transferred to other servers. Finally, as mentioned earlier, H-Store's parameter `txn_incoming_delay` is set to 0 ms to optimize for latency in an unloaded system.

9.3.1 Basic Latency

Figures 9.1, 9.2, and 9.3 graph the latency for single-object index operations. For this experiment, we have a single table (with a single tablet) which contains a varying number of objects. Each object in the table has a 30 B primary key, 30 B secondary key and a 100 B value. The secondary key has an index (with a single indexlet) corresponding to it. A single client sends a single request at a time to ensure an unloaded system for measuring latency.

SLIK Lookup:

The median time for an index lookup that returns a single object is about 11 µs for a small index and 13.1 µs for an index with one million entries. Of this time, about 9.2 µs is accounted for by basic RPC times. An index lookup issues RPCs in two sequential steps as discussed in Chapters 4 and 8. The first RPC is sent to the appropriate index server to find the primary key hashes for the objects that fall within the queried range. The second set of RPCs is sent to the appropriate data servers to read the objects based on the primary key hashes. Given that the time for a single read in RAMCloud (for example when reading an object based on its primary key) is about 4.6 µs, the minimum time required for an index lookup should be about 9.2 µs. The remaining 1.8–3.9 µs is the actual time to query the indexlet on the index server. This includes traversing the indexlet B+ tree to find the correct entry, which takes longer as the number of entries in the indexlet increases. If SLIK stored the B+ trees directly in memory such that links between nodes are represented by memory addresses, this time would be slightly lower (Section 7.2).

SLIK Write:

The median time for writes ranges from 29.6 µs to 36.3 µs depending on index size. Of this time, about 28 µs is accounted for by basic RPC times. Writing an indexed object requires two sequential writes as discussed in Chapters 5 and 8. The first RPC is sent to the appropriate data master, which in turn sends an RPC to the appropriate index server

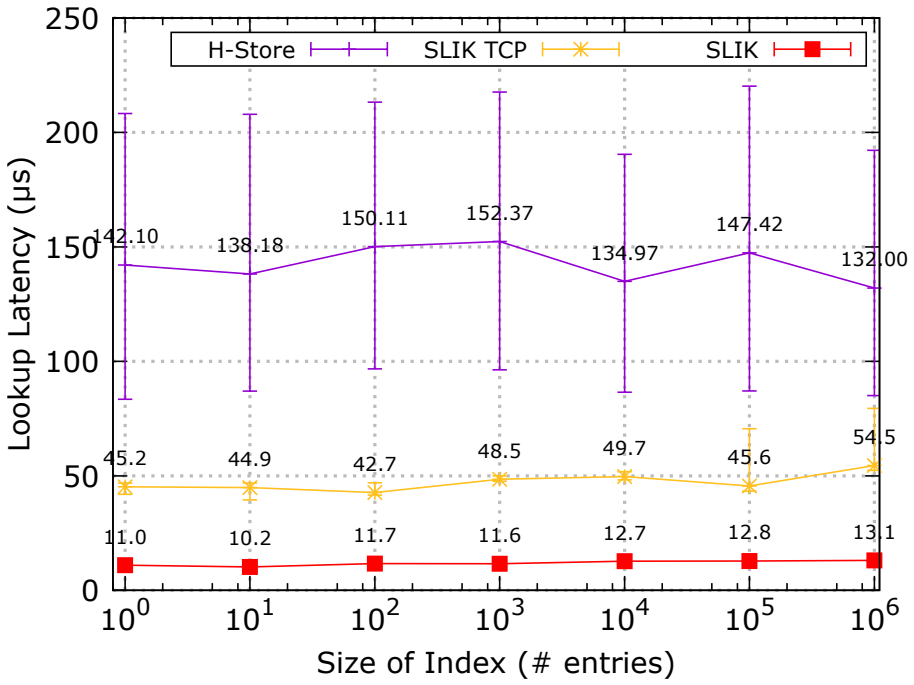


Figure 9.1: **Latency of lookups as index size increases**.
Setup: Graphs the latency to read a single object using a secondary key as the number of objects in the table and correspondingly the number of entries in the index increases. A single table is used, where each object has a 30 B primary key, a 30 B secondary key, and a 100 B value. The secondary key has an index (with a single partition) corresponding to it. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements.
Observations: The lookup latency in SLIK is about 2× the read latency in RAMCloud without indexes (which is about 4.6 µs). The latency increases slightly as the number of objects in the index increases. SLIK is about 10× faster than H-Store. Even when SLIK is run with TCP over InfiniBand (without kernel bypass), it still outperforms H-Store by 2–3×.

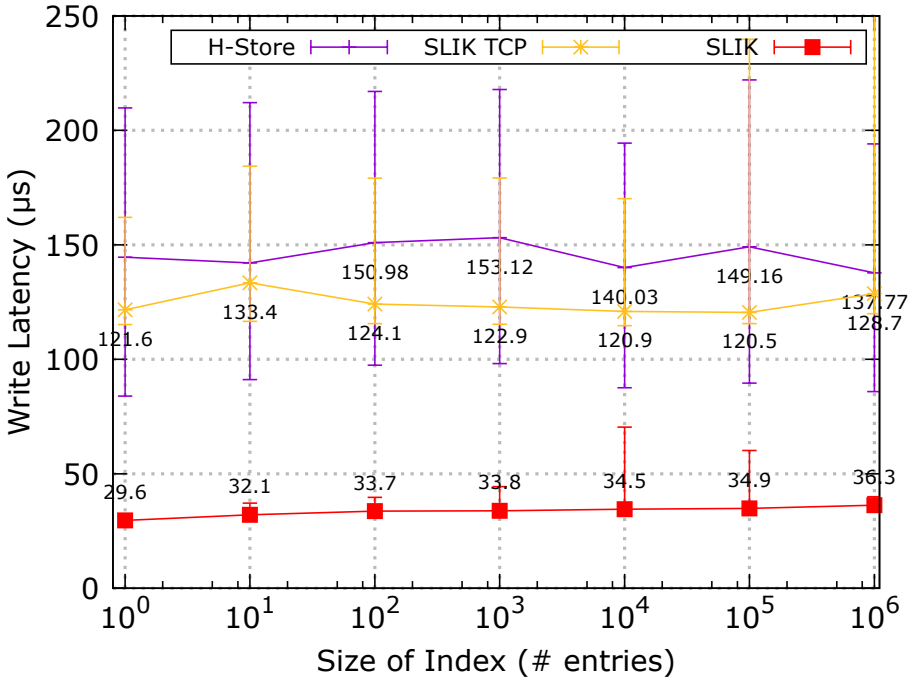


Figure 9.2: **Latency of write as index size increases**.
Setup: Graphs the latency to write a new indexed object as the number of objects in the table increases and correspondingly the number of entries in the index increases. A single table is used, where each object has a 30 B primary key, a 30 B secondary key, and a 100 B value. The secondary key has an index (with a single partition) corresponding to it. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements.
Observations: The write latency in SLIK is about 2× the write latency in RAMCloud for unindexed objects (which is about 14 µs). The latency increases slightly along the x axis (a difference of 6.5 µs between latency for an index with one entry vs a million entries). SLIK writes are about 5× faster than H-Store. When SLIK is run with TCP over InfiniBand (without kernel bypass), it achieves only slightly faster latency than H-Store, but unlike H-Store, it does so while providing three-way replication to backups.

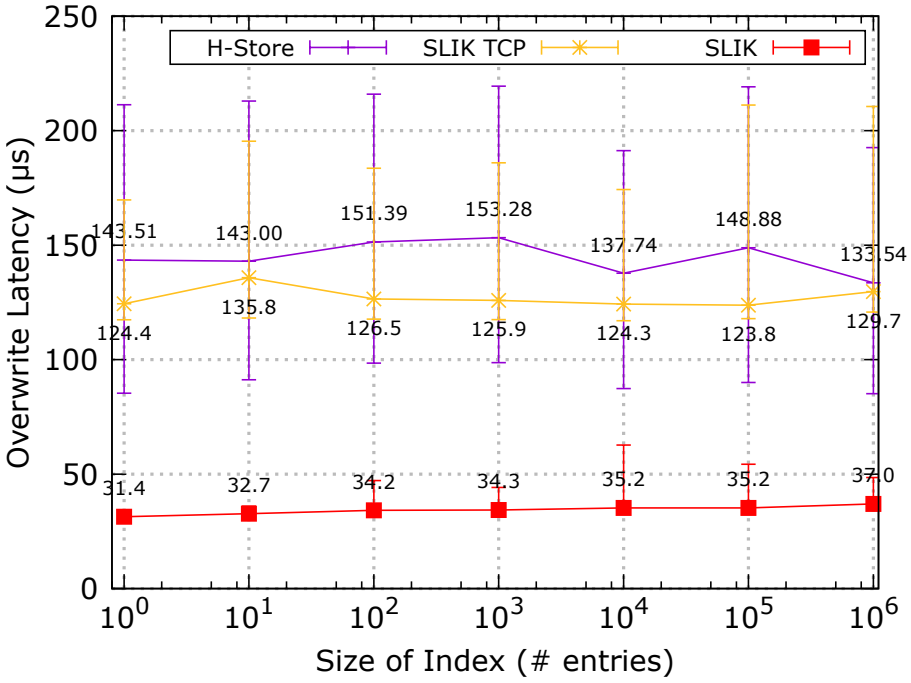


Figure 9.3: **Latency of overwrite as index size increases**.
Setup: Graphs the latency to overwrite an existing indexed object as the number of objects in the table increases and correspondingly the number of entries in the index increases. The setup is the same as that in the previous figure that evaluates basic write latency (Figure 9.2).
Observations: The overwrite latency in SLIK is about 2× the overwrite latency in RAMCloud for unindexed objects (which is about 14 µs). The overwrite latency is comparable to write latency, and the observations are similar to that in Figure 9.2.

(for each indexed secondary key in that object). Once the master receives acknowledgement that the index servers have inserted the index entries, it writes the object. Each of these writes includes replication to secondary storage on backups to ensure durability and provide fast crash recovery as discussed in Chapter 7. Given that the time for a single durable write (i.e., write of a non-indexed object) is about 14 µs, the minimum time required for an indexed object write should be about 28 µs.

This latency would have been lower if SLIK accepted higher crash recovery times and used the rebuild approach for recovering from crashes (refer Chapter 7). In that case, the indexed object write would still require two sequential writes, but one of these writes (i.e., write of the index entry) would not require replication to backups. This would reduce the 28 µs lower bound on indexed write latency by about 10 µs.

The rest of the time (beyond the minimum of 28 µs) is required to traverse the index B+ Tree to find the the location for insertion, and this time increases with the size of the B+ Tree. Further, modifications can sometimes trigger a rebalancing in the B+ Tree which requires the objects corresponding to all the affected nodes to be modified and rewritten. As mentioned earlier with lookups, if SLIK stored the B+ Trees directly in memory, this time would be lower than our current implementation, which stores nodes in RAMCloud objects.

SLIK Overwrite:

The median time for overwrites ranges from 31.4 µs to 37.0 µs, depending on index size. This is still the cost for doing two sequential durable writes (the first to the index and the second to the object). While overwrites have to perform the extra step of removing old index entries, this is handled asynchronously in the background after the overwrite RPC returns to the client, as discussed in Chapter 5.

Comparison:

Figures 9.1, 9.2, and 9.3 also compare SLIK's performance to H-Store. They show that SLIK is more than 10× faster than H-Store for lookups and about 5× faster for writes and overwrites.

However, SLIK is designed to minimize overheads so that it can harness the benefits of low-latency networks and kernel bypass (via InfiniBand). This brings up the question: is SLIK's superior performance solely due to the use of kernel bypass? To answer this question,

we intentionally crippled SLIK and ran it with TCP over InfiniBand, which is similar to the configuration we used for H-Store. Even in this configuration, SLIK outperforms H-Store, making it likely that SLIK's overall design and implementation, rather than kernel bypass, leads to faster operations. Lookups in SLIK remain more than twice as fast as H-Store. Writes and overwrites are only slightly faster. However, unlike H-Store, SLIK achieves this latency while providing 3-way replication of all index data: H-Store's latency would probably increase significantly if it also provided replication.

Although implementing kernel bypass in H-Store was beyond the scope of our work, we can use the difference between the latencies of SLIK implementation with and without TCP to estimate the overhead of using TCP rather than using kernel bypass (via InfiniBand). For instance, Figure 9.1 shows that the difference in latency for SLIK with and without TCP is about 35 µs. That is, the overhead for using TCP rather than kernel bypass is 35 µs. If we adjust the lookup latency for H-Store to theoretically give it the benefit of using kernel bypass, its lookup latency for the smallest possible table is about 150 µs - 35 µs = 115 µs. This is still 10× slower than the 11 µs latency for SLIK. This shows H-Store has additional latency that is unrelated to the network, so it is unlikely that it would approach SLIK's performance even with faster networks.

9.3.2 Impact of Multiple Secondary Indexes on Overwrite Latency

Figure 9.4 graphs the latency for overwriting an object as the number of indexed secondary keys in that object increases. The measurements were done with a single client accessing a single table with a million objects, where each object has a 30 B primary key, a varying number of 30 B secondary keys, and a 100 B value. Each secondary key has an index corresponding to it. For this experiment, each secondary index in SLIK has a single partition and is located on a different server.

SLIK:

As the number of indexed secondary keys in an object increases, we expect the latency for (over)writes to increase because each additional key adds an extra durable write operation to update the corresponding index. However, SLIK parallelizes the updates to different indexes so the total latency should not be significantly higher than the time required to (over)write an object with a single indexed key.

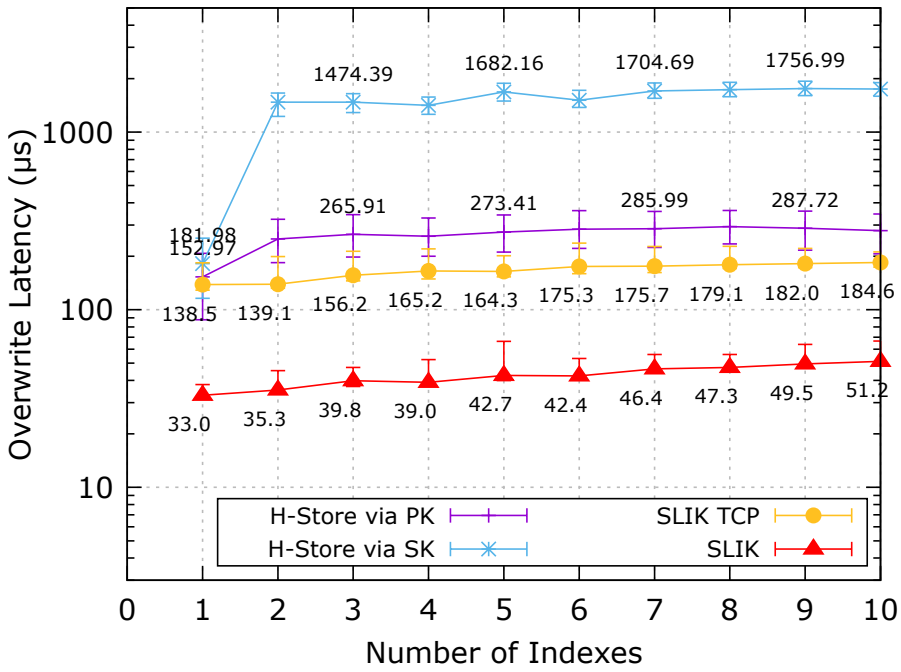


Figure 9.4: **Latency of overwrites as the number of secondary indexes increases.** *Setup:* A single table is used, where each object has a 30 B primary key, x 30 B secondary keys, and a 100 B value. Each secondary key has an index corresponding to it. For SLIK, each index has a single indexlet, and all the indexlets are located on separate servers. For H-Store's line *via Pk*, the table was partitioned by the primary key and for the line *via SK*, it was partitioned by the first secondary key. In both the cases, overwrites were done by querying via the primary key. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements. The y axis uses a log scale.
Observations: SLIK's latency increases slightly along the x axis as more indexes are added. H-Store's latency is significantly affected depending on whether the data is partitioned by the same key that is used for querying (*via Pk*) or by a different key (*via SK*). H-Store's latency does not increase significantly along the x axis. SLIK outperforms H-Store irrespective of way H-Store is configured.

The latency increases slowly for tables with more secondary indexes: overwrites take 33.0 µs with 1 secondary index and 51.2 µs with 10 secondary indexes (about a 50% increase). This increase is due to the cost of initiating additional RPCs to insert index entries: the RPCs execute in parallel, but they are initiated sequentially by a single thread.

Comparison:

SLIK performs performs better with no tuning than a tuned version of H-store. Additionally, unlike H-Store, it does so while providing durability and replication.

For each data point, SLIK and H-Store are both allocated the same number of servers as the number of indexes. H-Store partitions all the data and indexes across these servers and the key that is used for partitioning can be specified while setting up the table. We tried two different configurations:

1. partitioning based on the primary key (shown in the graph by the line *via PK*), and
2. partitioning based on the first secondary key (shown in the graph by the line *via SK*).

In both cases, updates are performed by querying via the primary key (i.e., the primary key is used to locate the object to be updated, as is the case by default in SLIK).

In the first case, given a query, H-Store knows exactly which server to contact in order to find the object being overwritten. In the second case, H-Store has to contact all the servers to find the object. Therefore, the latency for overwrites performed using the same key that is used to partition all data (*via PK*) is lower than the latency for overwrites performed using a key that was not used for partitioning (*via SK*). In both cases, once H-Store finds the correct object, it can complete the overwrite on a single server. This is because all the index entries corresponding to an object are stored on the same server as that object. Consequently, the latency for both the H-Store lines does not increase significantly along the x-axis.

If an application developer knows *a priori* the types of queries that will be most critical, she can partition the tables accordingly to get comparatively better performance than what other configurations would yield. However, even with this additional tuning for H-Store and in spite of the latency increase for increasing indexes in SLIK, SLIK's worst performance is better than that of H-Store.

9.4 Is SLIK Scalable?

Next, we evaluate whether the design and implementation of SLIK offer scalable performance as the number of servers increases. First, to evaluate the design, we compare the scalability of the colocation and independent partitioning approaches while keeping everything else the same (Section 9.4.1). Second, to evaluate the implementation, we compare the scalability of SLIK with H-Store (Section 9.4.2). Given our choice of independent partitioning, we expect a linear increase in throughput as the number of servers increases, since there are no interactions or dependencies between indexlet servers. We also expect minimal impact on latencies as the number of indexlets increases.

9.4.1 Independent Partitioning vs Colocation

We evaluate our design choice of independent partitioning by comparing its scalability to the colocation approach. For independent partitioning, we used our implementation of SLIK in RAMCloud. For the colocation approach, we just changed the partitioning code (in the SLIK implementation) to use colocation instead.

We use two experiments to determine scalability. One experiment measures the end-to-end throughput of index lookup as the number of indexlets increases. The throughput is measured using multiple clients, each issuing multiple lookups in parallel. The number of clients performing lookups and the number of concurrent lookups per client is varied to achieve the highest throughput for each system; the highest value is shown in the graph. This experiment uses a single table where each object has a 30 B primary key, a 30 B secondary key and a 100 B value. The index corresponding to the secondary key is divided into a varying number of indexlets, and the table is divided into the same number of tablets. Each indexlet and tablet is stored on a different server. Each request chooses a random key uniformly distributed across indexlets and returns a matching object. The colocation setup is partitioned based on the key used for lookups, which is its best configuration for this use case.

Another experiment measures the end-to-end latency of index lookup as the number of indexlets increases. The setup for this experiment is the same as the previous one, except that a single client is used (instead of many), which issues one request at a time in order to expose the latency for each operation.

The results of these experiments, shown in Figures 9.5 and 9.6 respectively, confirm

that independent partitioning outperforms colocation at large scale. Figure 9.5 shows that with independent partitioning, the total lookup throughput increases with the addition of servers, whereas with colocation it does not. Figure 9.6 shows that as the scale gets larger, the latency for independent partitioning remains almost constant while that for the colocation approach increases.

While independent partitioning performs well for large numbers of indexlets and servers, the colocation approach performs better at small scale. In the limit of one partition, the colocation approach would only have to send one RPC to one server, while independent partitioning would have to send two sequential RPCs (first to the index then the object in table). If we have a small system (where all objects and indexes are located on a small number of servers) and most clients are querying for large ranges that return large numbers of objects, colocation offers better performance. As the scale gets larger, the cost of doing these two sequential RPCs becomes less than the cost of doing a large number of parallel RPCs with the colocation approach.

9.4.2 System Scalability

In order to determine whether SLIK's independent partitioning performs better than the approach employed in current systems, we evaluate how SLIK performs at large scale compared to H-Store. We use two experiments, one to evaluate the throughput and another to evaluate the latency, with the setup described in the previous subsection. Figure 9.7 shows that the end-to-end throughput of index lookup in SLIK increases linearly as the number of indexlets is increased, while the throughput for H-Store increases sub-linearly. Figure 9.8 shows that increasing the number of indexlets has minimum impact on SLIK's index lookup latency, while the latency for H-Store increases.

Given that H-Store uses the colocation approach, we would expect the scalability numbers to be different depending on whether the queries are based on the same key that is used for partitioning or any other key. In the first case, we expect H-Store to essentially demonstrate scalable behavior because the lookup can contact a single index server (similar to independent partitioning from the previous set of experiments). In the second case, we expect it to demonstrate non-scalable behavior because each index lookup must contact all index servers (similar to colocation approach from the previous set of experiments). Given that in this experiment we configured H-Store to query based on the same attribute used for partitioning, it is unclear why the performance isn't similar to that of SLIK.

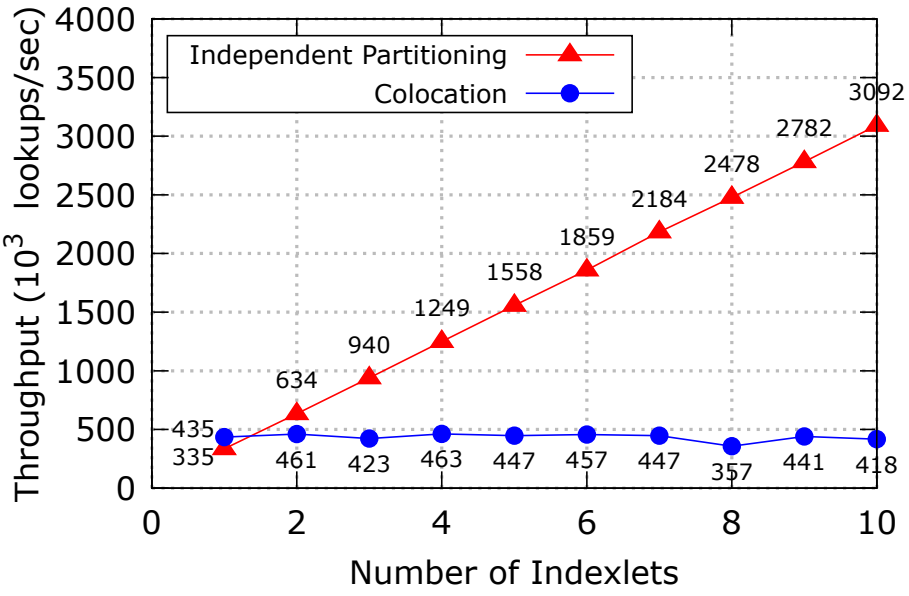


Figure 9.5: **Total index lookup throughput with increasing partitions.**
Setup: Graphs the total index lookup throughput for the two partitioning approaches when a single index is divided into multiple indexlets on different servers and queried via multiple clients. The number of clients and the number of concurrent lookups per client is varied to achieve the maximum throughput for each point on the graph.
Observations: The colocation approach provides higher throughput than independent partitioning in the limit of one index partition (by about 30%). Independent partitioning provides higher throughput with two or more partitions, and this advantage increases with the number of partitions.

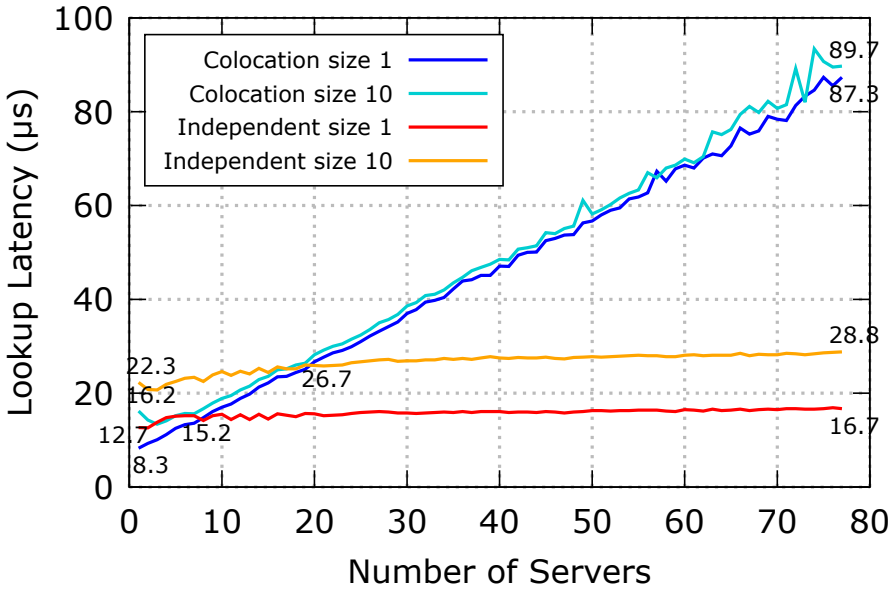


Figure 9.6: **Index lookup latency with increasing partitions.**
Setup: Graphs the latency for index lookup in the two partitioning approaches when a single index is divided into multiple indexlets on different servers and queried by a single client issuing a single request at a time. The size refers to the number of objects returned by a lookup.
Observations: For a small number of servers, independent partitioning has higher latency than colocation. However, as the number of servers increases, the latency for the colocation approach increases in proportion to the number of servers, while latency for the independent approach is nearly constant. As query size increases, the latency does not increase for colocation but it does for independent partitioning.

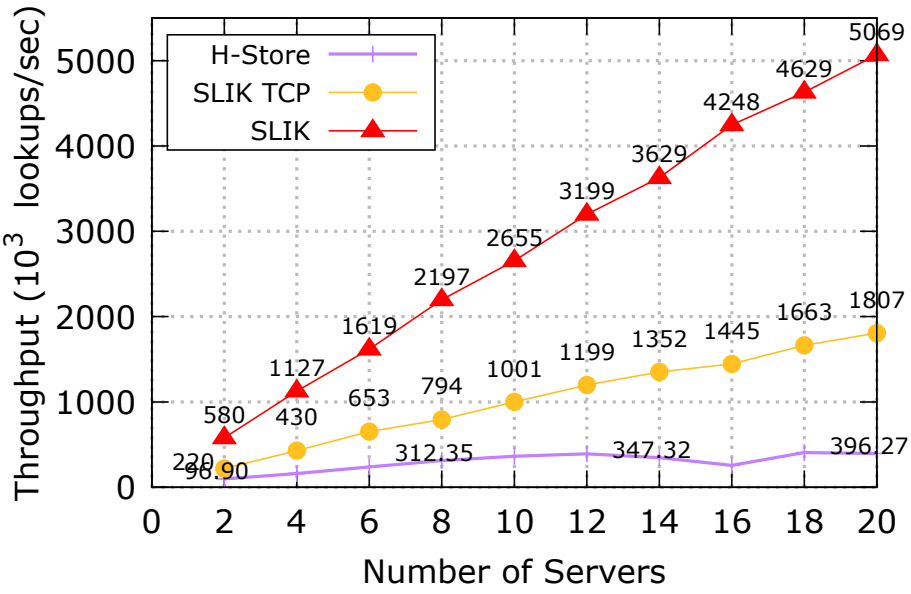


Figure 9.7: **Total index lookup throughput with increasing partitions.**
Setup: Graphs the total index lookup throughput for SLIK and H-Store when a single index is divided into multiple indexlets on different servers and queried via multiple clients. The number of clients and the number of concurrent lookups per client is varied to achieve the maximum throughput for each point on the graph.
Observations: Total throughput is much higher for SLIK than H-Store and it scales with the number of servers. With H-Store, additional servers provide diminishing returns.

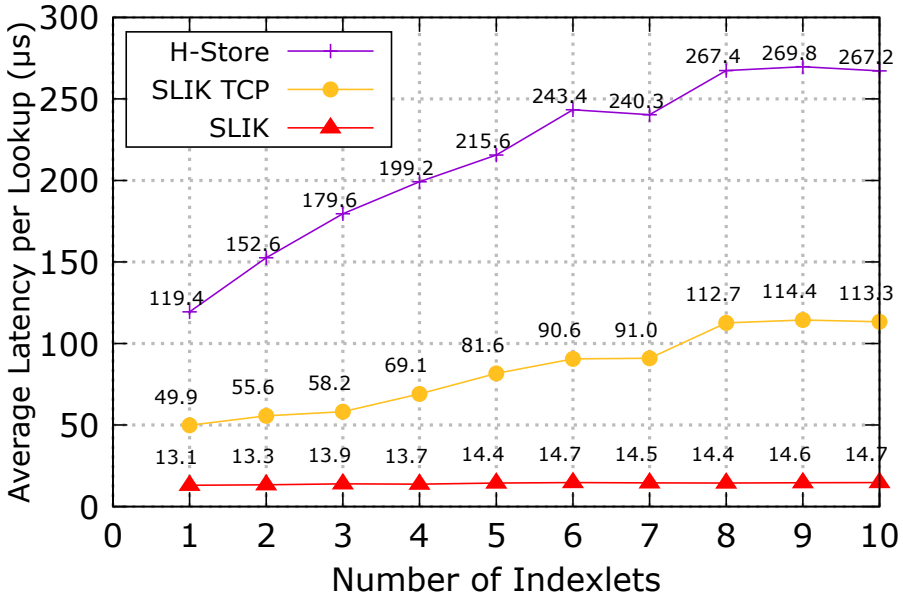


Figure 9.8: **Index lookup latency with increasing partitions.**
Setup: Graphs the latency in SLIK and H-Store for index lookup when a single index is divided into multiple indexlets on different servers and queried by a single client issuing a single request at a time.
Observations: Latency per lookup is nearly independent of the number of indexlets for SLIK but degrades somewhat with increasing index partitioning when implemented with TCP. H-Store has latency that rapidly increases with higher number of partitions before eventually plateauing.

9.5 How Does SLIK Impact Tail Latency of Operations?

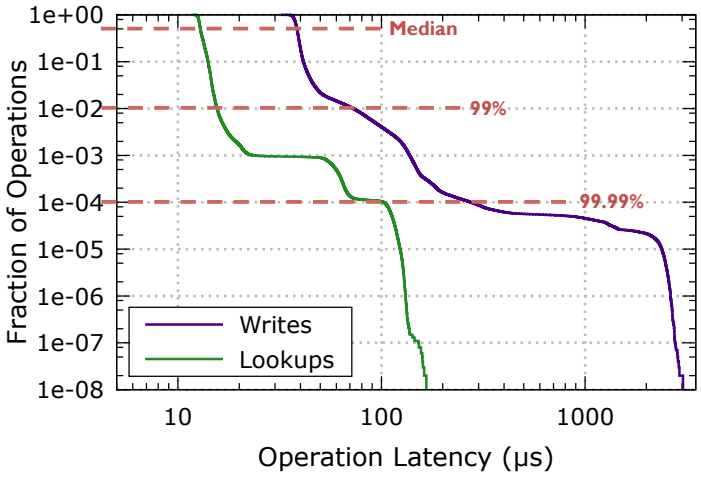
For some applications, the tail latency is more important than median latency. To investigate how the slowest operations in SLIK are distributed, Figure 9.9a graphs the reverse CDFs of latencies for looking up an object based on its secondary key and for writing an object with an indexed secondary key. A single client performs 100 million lookups and overwrites on a table with a million objects. Each object has a 30 B primary key, 30 B secondary key and 100 B value. The secondary key has an index corresponding to it (with a single partition).

Figure 9.9b graphs the reverse CDFs of latencies in RAMCloud for reading an object based on its primary key and writing an object (such that indexes are not involved). As before, a single client performs 100 million reads and overwrites on a table with a million objects. Each object has a 30 B primary key and 100 B value.

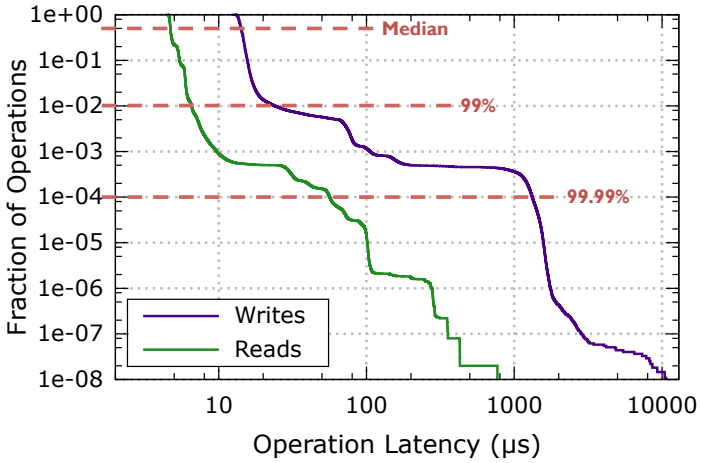
The index lookup operations have a median latency of about 13 µs, and write operations have a median latency of about 36 µs. Further, the tail latency for indexed operations in SLIK follow the same pattern as the tail latency for the corresponding operations in RAMCloud. This indicates that indexing does not modify the behavior of the operations at tail. Therefore, applications that are impacted by a high tail latency are unlikely to be affected due to the addition of secondary indexing any more than they already were in RAMCloud without indexes.

9.6 How Does Throughput Increase With the Size of Range Queried?

Figure 9.10 graphs the throughput for index lookup as a function of the total number of objects returned in that lookup. The setup is the same as in the previous experiment. The figure shows that the total throughput increases as the size of lookup is increased and peaks at about 2 M objects/s. The increase in throughput is expected, because SLIK tightly pipelines the requests to the index servers and data servers, and parallelizes the requests to data servers (Chapter 8). Further, the throughput stabilizes at around 1.7 M objects/s. We are not sure why the throughput drops before stabilizing, but we do expect the throughput to saturate because at some point the size gets large enough that waiting for RPCs to be sent and received is no longer a bottleneck.



(a) *Setup:* A single client performs 100 million index lookup and indexed object write operations in SLIK. A single table (with one partition) is used, where each object has a 30 B primary key, 30 B secondary key and 100 B value. The secondary key has an index (with one partition) corresponding to it. The index lookup fetches an object based on that secondary key and the write updates an indexed object.



(b) *Setup:* A single client performs 100 million read and write operations in RAMCloud. A single table (with one partition) is used, where each object has a 30 B primary key and 100 B value. The read fetches an object based on its primary key and the write updates an object.

Figure 9.9: **Tail latency distribution for basic single-object operations.** The graphs are shown as reverse CDFs on a log scale. A point (x, y) indicates that a fraction y of the 100 M operations took at least xµs to complete.
Observations: Very few operations take more than 10× the median time and the addition of secondary indexes does not drastically change the proportion of slow operations.

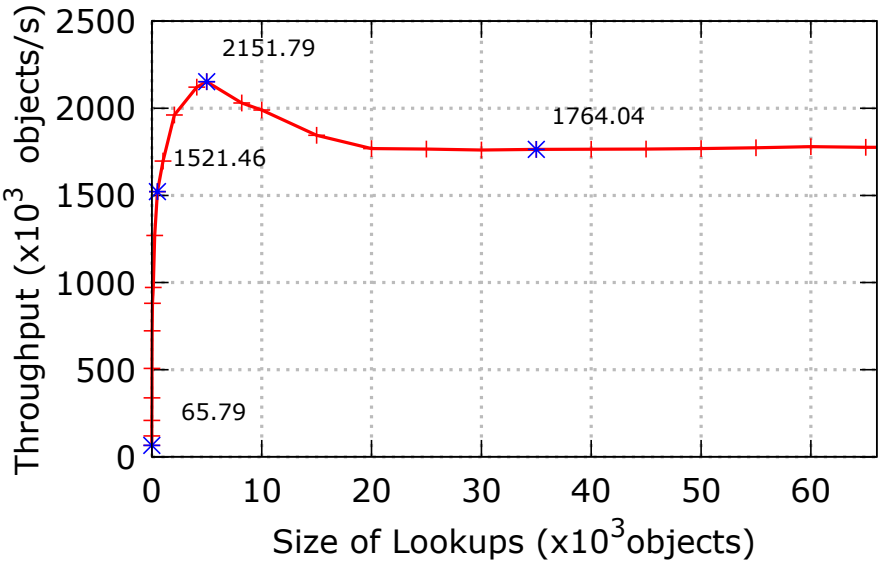


Figure 9.10: **Index lookup throughput with increasing size of lookup.**
Setup: Graphs the throughput of index lookup measured by a single client as a function of the total number of objects returned for that lookup. A single table is used, where each object has a 30 B primary key, 30 B secondary key and 100 B value.
Observations: The throughout increases with the size of the lookup up to about 8000 objects and then declines slightly and saturates.

Chapter 10

Related Work

With the advent of web applications, data storage systems have evolved to support massive data sets (Chapter 2). Modern data storage systems are improving every day and there is a lot of interesting work in the area of providing higher level data models at large scale. Much of the discussion about these systems and their design is interwoven throughout this dissertation. This chapter summarizes some of that discussion.

While most data stores used today support large amounts of data, they still make trade-offs between various desirable features: higher level data models (like indexing), strong consistency, scalability, low latency and even durability. For example, MICA [28] is a scalable in-memory key-value store optimized for high throughput; however it does not ensure data durability and doesn't support indexes. FaRM [20] is a main memory distributed computing platform that offers low latency and high throughput by exploiting RDMA; however it does not support secondary indexing. Many other systems provide higher level data models but have weak consistency guarantees: PNUTS [18] has relaxed consistency; CouchDB [2] and Tao [16] are eventually consistent.

Of the systems that support secondary indexes, it is interesting to note that many provide two types of indexes: local indexes (i.e., separate indexes for data on each server, which is essentially indexes partitioned using the colocation approach) and global indexes. Examples include DynamoDB [3], Phoenix [10] on HBase [4] and Cassandra [26] (Cassandra does not have explicit global indexes, but provides materialized views to achieve the same effect). The local indexes in these systems offer high consistency, but require higher latency at large scale as each request needs to contact all the servers (as described in Section 4). The global secondary indexes in these systems are only eventually consistent. Moreover,

they can return only those attributes of the object data that have been projected onto that index by the developer and stored with it.

No widely known systems provide secondary indexing at very low latency, irrespective of the consistency guarantees. SLIK pushes the boundary to provide better latency while still providing strong consistency. Figure 10.1 shows some examples of large scale data stores to illustrate current landscape in this area. All of these systems have different high level data models. Of them, HBase and Espresso have the simplest data models, but both provide secondary indexes. The latency numbers for most systems are for operations within a datacenter, except Megastore and Spanner which provide cross-datacenter replication. Additionally, the latency numbers are based on published benchmarks, except H-Store and HyperDex, for which I used my own measurements. The systems closest to the x-axis have the weakest consistency while the ones farthest from the x-axis have the strongest consistency. Cassandra and Megastore span a large area along the y-axis because they allow the client to specify the consistency level.

SLIK's most unique aspect is its combination of low latency and consistency at large scale.

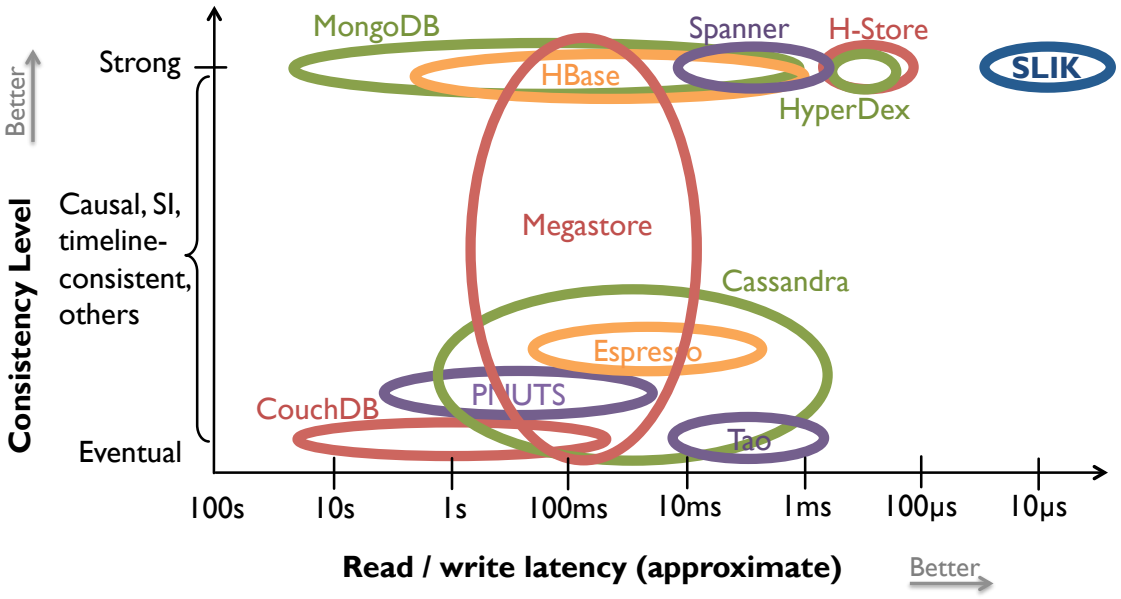


Figure 10.1: The landscape for large scale datacenter storage systems. The systems that provide stronger consistency guarantees are shown higher along the y-axis, and the systems with lower average latencies are shown to the right along the x-axis.

Chapter 11

Conclusion

This dissertation started with the hypothesis that key-value stores can provide scalable, low-latency, strongly consistent indexes. I conclude that this is indeed possible, as demonstrated by our implementation of SLIK in RAMCloud. Over the course of this dissertation, I introduced various aspects of an indexing system and the approaches that can be taken to achieve the desired properties.

In this chapter, I summarize the key aspects of SLIK's design by describing how they achieve the goals stated at the beginning of this thesis. I also share some of the lessons I learnt while building SLIK and other components of RAMCloud. Finally, I end with some possible directions in which this work can be extended in the future.

11.1 Summary

SLIK demonstrates that large-scale storage systems can offer secondary indexes with several desirable properties:

- **Low Latency:** SLIK harnesses low latency networks and stores index data in DRAM. SLIK makes various design decisions that allow it to utilize the network's and DRAM's latency advantage by leaving out complex mechanisms wherever possible in favor of lightweight methods that minimize overhead. These design decisions range from major choices like ensuring consistency via an ordered write approach using objects as ground truth (Section 5.1.2) to the smallest details like using cumulative lengths for keys in the object format to reduce computation while parsing an object (Section 3.1).

- **Scalability:** The total throughput of SLIK indexes increases linearly with the number of servers it spans, while the latency remains nearly constant. It achieves this by partitioning indexes such that the index entries can be distributed independently from the corresponding objects (Section 4.2).

- **Dealing With Large Scale Operations:** To maximize scalability, large-scale long-running operations such as adding / removing indexes (Section 3.3), splitting / migrating index partitions (Section 4.7), and range lookups (Section 5.1.3) are performed such that they do not block other operations.

- **Consistency:** SLIK provides strong consistency guarantees, similar to the consistency properties in a centralized system (Section 5.1). It does so by using a novel lightweight mechanism (Section 5.1.2) that avoids the complexity and overhead imposed by most distributed transaction implementations. This mechanism utilizes an ordered-write approach for updating indexed objects and uses objects as ground truth to determine liveness of index entries. One area where SLIK fails to meet its goal of strong consistency is during large range lookups: if an object is modified concurrently during a range lookup such both the old and the new versions of the object fall within the queried range, then either or neither or both versions of this object may be returned. SLIK accepts this caveat to strong consistency to ensure scalability (Section 5.1.3).

- **Durability and Availability:** While SLIK stores data in DRAM to allow low latency, it ensures that data is durable and survives server crashes. SLIK uses objects of the underlying key-value store to represent the index B+ Tree nodes, and leverages the existing recovery mechanisms of the key-value store to quickly recover indexes (Section 7.1.2).

11.2 Lessons Learned

Achieving low latency in systems software requires leaving out complex mechanisms in favor of lightweight approaches and reducing overheads where possible. I have found that it is often convenient and sometimes even necessary to add layers or levels of indirection. This includes layering to enable cleaner software abstractions; or even using an index structure to enable access to objects based on a given key. However, that comes at the cost of latency.

While many features can be added in, providing low latency requires leaving things out. Latency has to factor in while making various design and implementation choices. In SLIK, this is visible from bigger design decisions, like the consistency mechanism (Section 5.1.2), to small implementation details, like the choice of using cumulative lengths in the object format (Section 3.1).

In order to achieve scalability, the number of servers that need to be contacted in order to complete a request should scale with the size of request rather than the size of data (or the number of servers spanned by the data). In retrospect, this is obvious. It is similar to the principle we learn while designing simple algorithms in introductory Computer Science classes: the amount of work done should scale with the size of the request rather than the amount of data. In SLIK, this idea manifests itself in the scheme used to partition indexes (Section 4.2).

Further, to maximize scalability, other operations should not be blocked by a few long-running operations. If an operation requires exclusive access to some data set, it might lock that data set for the duration of its execution. This blocks other operations from accessing any subset of this data. As the size of the locked data set increases, the likelihood that another operation wants to access some subset of that data also increases. Similarly, as a data set is locked for a longer period of time, it becomes more likely that another operation will be executed on the some subset of that data within that period. Thus, the number of other operations blocked (ob) by a given operation is likely proportional to the size of the data set (s) and the amount of time this data is blocked (t). This means we must minimize $ob \propto s * t$ in order to ensure scalability. Hence, SLIK performs long-running bulk operations such as index creation/deletion (Section 3.3), migration (Section 4.7), and range queries (Section 5.1.3) using different techniques that involve locking only a small amount of data for short periods of time. Such minimal locking of data allows other operations to proceed without being impacted by a few long running operations.

A recurring feature in SLIK is that it permits temporary inconsistencies in its implementation, while maintaining consistent behavior for applications. This feature helps reduce the latency and complexity in the system: it plays a key role in the basic consistency algorithm (Section 5.1.2) which enables strong consistency for basic index operations without the overheads of more complex mechanisms like transactions. This feature also helps retain scalability by allowing long running operations like index creation and deletion (Section 3.3) to proceed without blocking other operations.

11.3 Future Work

This work focused on balancing the tradeoffs between scalability, low latency, consistency and durability. However, it leaves a number of interesting problems unanswered. This section discusses some of them.

The current interface for reconfiguration of indexlets using `splitAndMigrateIndexlet` leaves a lot of room for improvement as it requires heavy involvement by clients. It requires a client to explicitly split an indexlet when it gets too large; it requires the client to specify the identifier for the server that should host one of the resulting partitions; and it requires the client to have information about which key will yield a good split location. The first and second issues can be fixed by a server-driven or coordinator-driven process that automatically initiates a split when an indexlet starts getting too large and determines the new host server based on metadata already available to the servers. The third issue can be handled by adding functionality into the B+ Tree to traverse the nodes and automatically determine a good split point for the given tree. Providing automated index reconfiguration would allow application developers to focus on developing application specific features rather than having to ensure that the underlying system is load balanced.

It is also of interest to consider how the system could be optimized for varying needs of different applications. In particular, it would be valuable to explore different levels of consistency. If we know the expected workloads a priori, it might be possible to implement weaker consistency without impacting the correctness of the final application. For example, consider a social networking site that uses a table to keep track of the number of times (secondary key) the profile for each username (primary key) is viewed that day. Every second it reads the objects with the most views to display a summary of trending users. This application might be able to tolerate a small discrepancy in counts as it is interested only in overall trends. Having weaker consistency (when it does not affect correctness) is useful because consistency trades off with performance: as we enable stronger consistency, the latency increases or the throughput decreases. For example, if SLIK did not have to provide strong consistency, a write request could have simultaneously contacted the index servers and data server to update index entries and the object. This would result in weaker consistency but halve the latency (because now the write becomes a one-step process rather than a two-step process). As another example, transactions that use optimistic concurrency control mechanisms lead to lower overall throughput in case of conflicts in order to ensure

strong consistency. Building a system that offers the application developers an option to tune the level of consistency (like Megastore [15]) might lead to the best of both worlds: strong consistency when needed, better performance when not.

Further, enabling a couple of features that provide a richer data model and query language could better meet the needs of diverse applications. One useful feature is to allow multiple values for the same key. For example, a table storing records of books might have a secondary key for the genre(s) a book belongs to. While a book might have just one genre (like "gardening"), another might have more (like "mystery", "thriller" and "fantasy"). A lookup for all books with "mystery" genre should return the second book, as should a lookup for all books with "thriller" or "fantasy" genre. Providing this feature would require rethinking the object format of SLIK. For example, a more flexible format like JSON might be a better fit, but would impact performance by increasing parsing overheads. Apart from that change, most of the other design and implementation should remain unchanged; hence, SLIK's utility could be improved without significantly sacrificing performance.

Another useful feature is to provide compound indexing that enables multi-attribute lookups. For example, from a books table, one might want to find all books with genre = "mystery" and 1500 < publication year < 2000. One implementation approach is for the client library to concurrently perform the first step of lookup, `lookupKeys`, for each secondary key. It can then take an intersection of the primary key hashes returned from these queries and perform the second step of lookup `readHashes` on that intersection. However, given that the hashes are returned in sort order of the respective secondary keys, this approach would require a client to gather all the hashes from step 1 before performing step 2. This may not work well at large scale as the client may not have the capacity to store all the hashes and compute the intersection locally. Another possibility is to slightly modify the server side code for lookups. Here, the client performs lookup on one of the keys as usual. After `readHashes`, the clients already parse the matching objects to check that the given secondary key falls within the specified lookup range (in order to prune extraneous entries as per the consistency algorithm). They could additionally perform the same check for other keys that are a part of the query and return only the objects where all the keys fall within the respective ranges. To reduce the amount of wasted work, the original lookup should be performed on the key that will match the least number of objects. However, there is currently no clearly optimal solution for efficiently performing multi-attribute lookups, and it is therefore of interest to investigate in the future.

While I built SLIK such that its design is applicable to most key-value stores, it is currently implemented only in RAMCloud. When implementing SLIK in another system, different requirements might mean that some of the alternative approaches discussed are better suited than the ones chosen for SLIK. Further, differences in the underlying key-value store can lead to different index implementations. For instance, if a key-value store did not support fast crash recovery, it might be better to use the rebuild approach for recovering indexes. If a key-value store implemented Multi Version Concurrency Control (MVCC), it might be useful to leverage that and enable stronger consistency for range queries. It will be interesting to see how the ideas presented in this dissertation hold up and what new ideas emerge when implementing SLIK in a different underlying key-value store for a different set of requirements.

11.4 Final Thoughts

SLIK shows that modern scalable storage systems need not sacrifice the powerful programming model offered by traditional relational databases. Furthermore, when implemented using DRAM-based storage and state-of-the-art networking, storage systems can provide unprecedented performance. SLIK is an important first step on the path to a high-function, low-latency, large-scale storage system.

Bibliography

[1] Aerospike. http://www.aerospike.com/. 1

[2] CouchDB. http://couchdb.apache.org/. 18, 34, 79

[3] DynamoDB. http://aws.amazon.com/documentation/dynamodb/. 28, 79

[4] HBase. http://hbase.apache.org/. 28, 79

[5] LogCabin. http://github.com/logcabin. 11

[6] Memcached. http://memcached.org/. 8

[7] MongoDB. http://www.mongodb.org/. 8, 18

[8] MySQL InnoDB Storage Engine. http://dev.mysql.com/doc/refman/5.5/en/innodb-storage-engine.html. 50

[9] Panthema STX B+ Tree. http://panthema.net/2007/stx-btree/. 50

[10] Phoenix. http://phoenix.apache.org/. 28, 79

[11] RAMCloud Git Repository. https://github.com/PlatformLab/RAMCloud.git. 3, 9

[12] RAMCloud Online Wiki. http://ramcloud.stanford.edu. 9

[13] Redis. http://www.redis.io/. 1

[14] Marcos K Aguilera, Arif Merchant, Mehul Shah, Alistair Veitch, and Christos Karamanolis. Sinfonia: A New Paradigm for Building Scalable Distributed Systems. In *ACM SIGOPS Operating Systems Review*, volume 41, pages 159–174, 2007. 36

[15] Jason Baker, Chris Bond, James C Corbett, JJ Furman, Andrey Khorlin, James Larson, Jean-Michel Leon, Yawei Li, Alexander Lloyd, and Vadim Yushprakh. Megastore: Providing Scalable, Highly Available Storage for Interactive Services. In *Proceedings of the Conference on Innovative Data System Research*, pages 223–234, 2011. 8, 24, 34, 86

[16] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry C Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. TAO: Facebook's Distributed Data Store for the Social Graph. In *USENIX Annual Technical Conference*, pages 49–60, 2013. 7, 34, 79

[17] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C Hsieh, Deborah A Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, and Robert E Gruber. Bigtable: A Distributed Storage System for Structured Data. *ACM Transactions on Computer Systems*, 26(2):4, 2008. 7

[18] Brian F Cooper, Raghu Ramakrishnan, Utkarsh Srivastava, Adam Silberstein, Philip Bohannon, Hans-Arno Jacobsen, Nick Puz, Daniel Weaver, and Ramana Yerneni. PNUTS: Yahoo!'s Hosted Data Serving Platform. *Proceedings of the VLDB Endowment*, 1(2):1277–1288, 2008. 7, 34, 79

[19] James C Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, JJ Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolig, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. Spanner: Google's Globally Distributed Database. *ACM Transactions on Computer Systems*, 31(3):8, 2013. 8

[20] Aleksandar Dragojević, Dushyanth Narayanan, Orion Hodson, and Miguel Castro. FaRM: Fast Remote Memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, pages 401–414, 2014. 79

[21] Robert Escriva, Bernard Wong, and Emin Gün Sirer. HyperDex: A Distributed, Searchable Key-Value Store. *ACM SIGCOMM Computer Communication Review*, 42(4):25–36, 2012. 7, 59

[22] Daniel Ford, François Labelle, Florentina I Popovici, Murray Stokely, Van-Anh Truong, Luiz Barroso, Carrie Grimes, and Sean Quinlan. Availability in Globally Distributed Storage Systems. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, pages 61–74, 2010. 43

[23] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. ZooKeeper: Wait-free Coordination for Internet-scale Systems. In *USENIX Annual Technical Conference*, page 9, 2010. 11

[24] Robert Kallman, Hideaki Kimura, Jonathan Natkins, Andrew Pavlo, Alexander Rasin, Stanley Zdonik, Evan P C Jones, Samuel Madden, Michael Stonebraker, Yang Zhang, John Hugg, and Daniel J Abadi. H-Store: A High-Performance, Distributed Main Memory Transaction Processing System. *Proceedings of the VLDB Endowment*, 1(2):1496–1499, 2008. 1, 8, 24, 58

[25] Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang, and John Ousterhout. SLIK: Scalable Low-Latency Indexes for a Key-Value Store. In *2016 USENIX Annual Technical Conference*, pages 57–70, 2016. 15

[26] Avinash Lakshman and Prashant Malik. Cassandra: A Decentralized Structured Storage System. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010. 24, 79

[27] Collin Lee, Seo Jin Park, Ankita Kejriwal, Satoshi Matsushita, and John Ousterhout. Implementing Linearizability at Large Scale and Low Latency. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 71–86, 2015. 15, 36

[28] Hyeontaek Lim, Dongsu Han, David G Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, page 36, 2014. 79

[29] Wyatt Lloyd, Michael J Freedman, Michael Kaminsky, and David G Andersen. Don't Settle for Eventual: Scalable Causal Consistency for Wide-Area Storage with COPS. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 401–416, 2011. 7

[30] Haonan Lu, Christopher Hodsdon, Khiem Ngo, Shuai Mu, and Wyatt Lloyd. The SNOW Theorem and Latency-Optimal Read-Only Transactions. In *Proceedings of the*

12th USENIX Symposium on Operating Systems Design and Implementation, page 135, 2016. 36

[31] Michael Nelson, Beng-Hong Lim, and Greg Hutchins. Fast Transparent Migration for Virtual Machines. In *USENIX Annual Technical Conference*, pages 391–394, 2005. 32

[32] Diego Ongaro and John Ousterhout. In Search of an Understandable Consensus Algorithm. In *2014 USENIX Annual Technical Conference*, pages 305–319, 2014. 11

[33] Diego Ongaro, Stephen M Rumble, Ryan Stutsman, John Ousterhout, and Mendel Rosenblum. Fast Crash Recovery in RAMCloud. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 29–41, 2011. 14, 51

[34] John Ousterhout, Arjun Gopalan, Ashish Gupta, Ankita Kejriwal, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Henry Qin, Mendel Rosenblum, Stephen Rumble, Ryan Stutsman, and Stephen Yang. The RAMCloud Storage System. *ACM Transactions on Computer Systems*, 33(3):7:1–7:55, 2015. 1, 3, 7, 9, 31

[35] Andy Pavlo. Personal communications, March 17 2015. 60

[36] Lin Qiao, Kapil Surlaker, Shirshanka Das, Tom Quiggle, Bob Schulman, Bhaskar Ghosh, Antony Curtis, Oliver Seeliger, Zhen Zhang, Aditya Auradar, Chris Beaver, Gregory Brandt, Mihir Gandhi, Kishore Gopalakrishna, Wai Ip, Swaroop Jgadish, Shi Lu, Alexander Pachev, Aditya Ramesh, Abraham Sebastian, Rupa Shanbhag, Subbu Subramaniam, Yun Sun, Sajid Topiwala, Cuong Tran, Jemiah Westerman, and David Zhang. On Brewing Fresh Espresso: LinkedIn's Distributed Data Serving Platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1135–1146, 2013. 24, 34

[37] Stephen M Rumble, Ankita Kejriwal, and John Ousterhout. Log-structured Memory for DRAM-based Storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies*, pages 1–16, 2014. 12, 14, 51

[38] Ryan Stutsman, Collin Lee, and John Ousterhout. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *2015 USENIX Annual Technical Conference*, pages 17–30, 2015. 54

[39] Edward Zayas. Attacking the Process Migration Bottleneck. *ACM SIGOPS Operating Systems Review*, 21(5):13–24, 1987. 32