SLIK: Scalable Low-Latency Indexes for a Key-Value Store

Ankita Kejriwal, Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang and John Ousterhout

Stanford University

Abstract

Many large-scale key-value storage systems sacrifice features like secondary indexing and/or 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, consistency and high availability often conflict with each other. This paper shows how a large-scale key-value storage system can be extended to provide secondary indexes while meeting those goals. The architecture, called SLIK, enables multiple secondary indexes for each table. SLIK represents index B+ trees using objects in the underlying key-value store. It allows indexes to be partitioned and distributed independently of the data in tables while providing reasonable consistency guarantees using a lightweight ordered write approach. Our implementation of this design on RAMCloud (a main memory key-value store) performs indexed reads in 11 μs and writes in 30 μs. The architecture supports indexes spanning thousands of nodes, and provides linear scalability for throughput.

1 Introduction

Over the last decade, a new class of storage systems has arisen to meet the needs of large-scale web applications. Various main-memory-based data storage systems such as Aerospike [1], H-Store [19], RAMCloud [24] and Redis [8] have scaled to span hundreds or thousands of servers, with unprecedented overall performance. 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 and/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 system should harness low latency networks, store index data in DRAM, and leave out complex mechanisms wherever possible in favor of lightweight methods that add minimal 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 provide nearly constant latency irrespective of the number of servers an index spans.
- Consistency: The system should provide clients with the same strong consistency as a centralized system. 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.
- Availability: The system must also be continuously available; this creates challenges around crash recovery and requires that schema changes such as adding and removing indexes be accomplished without taking the system offline.

In this paper, we 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 scales to provide high performance even with indexes that span hundreds of servers while providing strong consistency guarantees. Second, its mechanisms are simple enough to provide extraordinarily low latency when used with a low-latency key-value store. Third, it provides fast crash recovery, live index split and migration and other features that ensure a high level of availability. Finally, it uses main memory judiciously while storing secondary index structures.

SLIK uses several interesting approaches to achieve the desired properties:

- Its data model is a multi-key-value store, 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.
- SLIK achieves high scalability by distributing index entries independently from their objects rather than colocating them (which is the more commonly used approach today).
- However, the resulting indexed operations are now distributed, which creates potential consistency problems between indexes and objects. SLIK provides clients with a consistent behavior using a novel lightweight mechanism that avoids the complexity and overhead of distributed transactions. It uses an ordered-write approach for updating indexed objects and uses objects as ground truth to determine liveness of index entries.
- SLIK performs long-running bulk operations such as index creation/deletion and migration in the background, 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, SLIK 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.

To demonstrate the practicality of SLIK, we implemented it in RAMCloud [24], a low-latency distributed key-value store. The resulting system provides extremely low latency indexing and is highly scalable:

- SLIK performs index lookups in 11–13 µs, which is only 2x the base latency of non-indexed reads.
- SLIK performs durable updates of indexed objects in 30–36 µs, which is also about 2x the base latency of non-indexed durable updates.
- The throughput of an index increases linearly as it is partitioned among more and more servers.
- SLIK's latency is 5–90x lower than H-Store, a state-of-the-art in-memory database.

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

2 The SLIK Design

In this section we describe the general architecture of SLIK, which could be used with any underlying key-value store. In the next section we will discuss features that are specific to our implementation in RAMCloud.

2.1 Data Model

In order to have secondary indexes, clients and servers must agree on where the secondary keys are located in the object. A traditional key-value store does not provide this information, as each object only contains a single key and a value. One commonly used approach is to store the keys as part of the object's value. In this case, the servers and clients must agree on a specific format for object values, such as JSON. Here, each index is associated with a particular named field, and the server parses the object value to find the secondary keys. Several storage systems use this approach, including CouchDB [2] and MongoDB [5]. However, this approach introduces additional overhead for the server to parse objects.

Given our objective of lowest possible latency in SLIK, we chose an object structure that directly identifies all the secondary keys. Consequently, there is no parsing required to carry out index operations. We 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 unique within the table. Each of the secondary keys can have an index corresponding to it. Each key can be of a different type with a corresponding ordering function (for example, numerical or lexicographic).

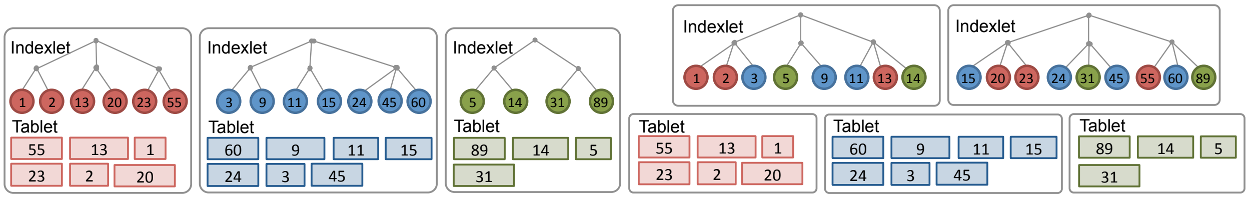
The object format can be managed automatically by client side libraries, so that applications do not have to be aware of how the information is stored in object values and secondary keys.

2.2 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.

A scalable index performs well even if it spans many servers. The index 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 goals, we considered three alternative approaches to index partitioning.

One approach is to colocate index entries and objects, so that all of the indexing information for a given object



(a) Indexlet/Tablet Colocation

(b) Independent Partitioning

Figure 1: Two approaches to index partitioning assuming that a table is partitioned by its primary key. In (a) indexes for the table are partitioned so that the index entry for each object is on the same server as the object. In (b) indexes for the table are partitioned so that each indexlet contains all the keys in a particular range. Rectangles represent objects, and the number in each rectangle is the value of the secondary key for that object (primary keys and object values are omitted). Circles represent index nodes; the number in each circle is the value of the secondary key. Colors distinguish objects (and secondary index keys) that belong to different tablets.

is stored on the same server as that object. In this approach, one of the keys (either the primary key or a specified secondary key) is used to partition the table's objects among servers, as shown in Figure 1(a). We call this ***Colocation Approach***, in which each server stores a table partition (or tablet) plus one indexlet for each of that table's indexes. The indexlet stores only index entries for the tablet on the same server. This approach is used widely by many modern storage systems, including Cassandra [20] and H-Store [19], and the local indexes in Espresso [26] and Megastore [11].

To perform a lookup using an index, a client issues parallel Remote Procedure Calls (RPCs) to all the servers holding partitions for this table. Each server scans its local indexlet, then returns the matching objects from its local tablet.

A 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, as shown in Figure 1(b). We call this ***Independent Partitioning***. With this approach, a small index range query can be processed entirely by a single index server.

With independent partitioning, a client performs index lookups in two steps. In the first step, the client issues an RPC to the server holding the relevant indexlet. This can typically be processed entirely by a single index server. If the queried range spans multiple indexlets, then each of the corresponding servers is contacted. This RPC returns information to identify the matching objects. The client then contacts the relevant data servers to retrieve these objects.

At small scale, the colocation approach provides lower latency. For example, in the limit of a single server, it requires only a single RPC, whereas independent partitioning requires two RPCs. However, as the number of servers increases, the performance of the colocation approach degrades. Each request must contact more and more servers, so the lookup cost increases linearly with the number of servers across which a table is sharded. On the other hand, independent partitioning provides dramatically better performance. Executing two sequential RPCs results in a constant latency (even as the number of partitions is increased), and this latency is lower than executing a large number of parallel RPCs. Moreover, with independent partitioning, the total lookup throughput increases with the addition of servers. This is not the case with the colocation approach, as each server must be involved in every index lookup. While many modern datastores use the colocation approach, our experiments in Section 4 show that the independent partitioning scheme provides better scalability.

A third approach is to use independent partitioning, but also replicate part or all of the table's data with each index. Any data that may be accessed via the index needs to be duplicated in this index. This approach is used by the global indexes in DynamoDB [3] and Phoenix [7] on HBase [4].

Global indexes combine some important benefits of the two approaches above. They enable low latency lookups as a lookup requires only a single RPC for small range queries. They are also scalable as the indexes are partitioned independently of the data table.

These benefits are at the cost of increased memory footprint: an index lookup can return only those attributes of the object that have been duplicated and stored with that index. This results in substantial data duplication, which might be acceptable for disk-based systems, but not for a memory-based system like SLIK.

We use the independent partitioning approach in SLIK. This enables high scalability while using memory efficiently.

2.3 Consistency during normal operations

As discussed in the previous section, 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.

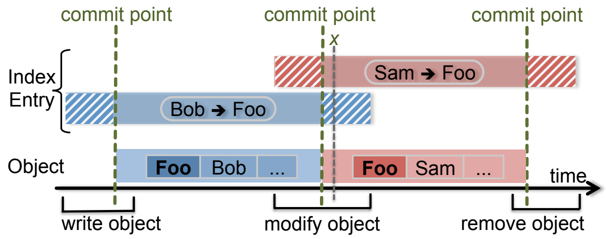


Figure 2: The ordered write approach ensures that if an object exists, then the corresponding index entries exist. Each object provides the ground truth to determine the liveness of its index entires. Writing an object serves as the commit point. The box at the bottom shows an object as it is created, modified and removed (Foo is the objects primary key; the secondary key is changed from Bob to Foo 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.

Existing storage systems have generally dealt with index consistency in two ways. Many large scale storage systems simply permit inconsistencies, in order to simplify their implementations or improve performance. For example, CouchDB [2], PNUTS [13], the global indexes for Espresso [26] and Megastore [11], and Tao [12] use relaxed consistency. This forces application programmers to build their own mechanisms to ensure consistency. The second approach, typical of smaller-scale systems, is to wrap updates in transactions that ensure atomicity. However, we were concerned about the implementation complexity and potential scalability problems of using transactions for this purpose.

Our goal is to build a scalable distributed system with the consistency expected from a centralized system. 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 this object contains a secondary key for that index within the specified range.

We want to provide this consistency while imposing miminal performance overheads. We designed a simple lighweight mechanism that ensures the consistency properties stated above without requiring transactions. 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 illustrated in Figure 2.

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. Finally, it asynchronously issues requests (again, to the server(s) with relevant indexlets) for removing old index entries, if this was an overwrite. 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, we observe that the information in an object is the *truth* and index entries pointing to it can be viewed as *hints*. During index range queries, the client first queries the indexlet server(s) responsible for the requested range. These servers identify the matching objects by returning a hash of the primary key for each matching object (Section 2.6 discusses the use of primary key hashes in detail). The client then uses these primary key hashes to fetch all of the corresponding objects. Some of these objects may not exist, or they may be inconsistent with the index (see Figure 2, 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.

Writing 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. This ensures the second of the two consistency properties.

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.

2.4 Metadata and Coordination

The metadata about the mapping from indexlets to their host servers needs to be managed using a persistent coordination service. This metadata is updated when a new index is created or dropped, an index server crashes or recovers data from another crashed server, and when an indexlet is split or migrated to another node. The co-ordination service only stores and disseminates the metadata: it does not take part in any lookup or write operations.

2.5 SLIK API

Tables 1 and 2 summarize the API of SLIK: Table 1 shows the operations visible to client applications, and

Table 2 shows the internal RPCs used to implement them.

A client invokes `createIndex` and `dropIndex` to create or delete an index on an existing data table (identified by `tableId`). The index is identified by `indexId`, such that the *n*th key in the object is indexed by index with `indexId` *n*.

When a client starts an index lookup, the SLIK API library acts as overall manager. It first issues `lookupKeys` to the appropriate index servers. Each index server identifies the primary key hashes for the objects in the secondary key range and returns them in index order. Then the client issues `readHashes` in parallel to the relevant data servers to fetch the actual objects using the primary key hashes. The objects returned from each data server are also in index order (as the order was specified by the key hashes in the query). For large range queries, SLIK uses a concurrent and pipelined approach with multiple RPCs in flight simultaneously. It is implemented using a rules-based approach [29]. As it receives the responses from various servers, it prunes extraneous entries (as per the consistency algorithm in Section 2.3) and collates results from different RPCs, so that the objects are returned to the client in index order.

We used a streaming approach (with an iterator API) rather than an approach that collects and returns all the objects at once. This allows index range queries to return very large result sets, which might not all fit in client memory at once.

To write an indexed object, a client sends a `write` request to the data server that stores the object. The data server synchronously issues `entryInsert` requests to relevant index servers to add new index entries, then modifies the object locally and durably replicates it. At this point, the data server returns a response to the client, then asynchronously issues `entryRemove` requests to relevant index servers. If the object is new (it did not previously exist), then the index removal step is skipped.

`createIndex(tableId, indexId, indexType)` Create a new index for an existing table.
`dropIndex(tableId, indexId)` Delete the specified index. Secondary keys in existing objects are not affected.

`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 iterator object.

`IndexLookup::getNext()` Get the next object in index order as per parameters specified earlier in `IndexLookup`, or wait until such an object is available.
`write(tableId, keys, value)` Create or overwrite the object. Update secondary indexes both to insert new secondary keys and to remove old ones (if this was an overwrite).

Table 1: A summary of the core API provided by SLIK to client applications for managing indexes and secondary keys.

`lookupKeys(tableId, indexId,`
`firstKey, lastKey, flags)`
Returns primary key hashes for all entries in the given index in the given range. `flags` provide additional parameters (for example, whether the end points of the range should be included in the search).

`readHashes(tableId, pKHashes)` Returns objects in table (`tableId`) whose primary key hash matches one of the hashes in `pKHashes`.
`entryInsert(tableId, indexId, key, pKHash)` 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)` Removes the given entry in the given index. Replicates the update durably before returning.

Table 2: A summary of the core RPCs used internally by SLIK to implement the `IndexLookup` and `write` operations in Table 1. Additional operations for managing indexlet ownership are omitted here.

2.6 Index Storage and Durability

SLIK uses a B+ tree to represent each indexlet, so that range queries can be supported. The B+ tree nodes map secondary keys to the corresponding objects. However, as SLIK uses the independent partitioning approach, index entries need a way to identify the objects they refer to. A straightforward way is for an index entry to map its secondary key to the primary key of the corresponding object. However, primary keys are variable length byte arrays, which can potentially be large (many KBs), so SLIK indexes identify an object instead 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, but these extra objects get pruned out as a by-product of the consistency algorithm.

SLIK keeps these B+ trees entirely in DRAM in order to provide the lowest possible latency. However, index information must be as durable and available as the objects in the key-value store (for example, it must survive server crashes).

One approach for achieving index durability is to rebuild indexlets from table data. To recover an indexlet with the ***rebuild approach***, each server storing objects of the corresponding table reads all the objects in its memory to find keys that belong to the crashed indexlet. Then the server that is the new owner of this indexlet reconstructs the indexlet using the table data. This approach is attractive for two reasons. First, it is simple: indexlets can be managed without worrying about durability. Second, it offers high 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: our tests show that it will take 25 seconds or more to recover a 500 MB partition, and the time (for the same sized partition) will increase as server memory sizes increase.

Our goal for crash recovery is to recover lost index data in about the same amount of time that the underlying storage system needs to recover lost table data. To achieve this, SLIK represents each indexlet B+ tree with a *backing table* in the underlying key-value store; the backing table is just like any other table, except that it is not visible to clients and has a single tablet. Each node in the B+ tree is represented with one object in the backing table. This ***backup approach*** allows indexlets to leverage the persistence and replication mechanisms the underlying key-value store uses for its object data.

With this approach, index crash recovery consists primarily of recovering the corresponding backing table. This is handled by the underlying key-value store. 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.

The backup approach to index recovery does have two disadvantages. First, since each node in the B+ tree is a separate object in the key-value store, traversing a pointer from a node to one of its children requires a lookup in the key-value store (pointers between nodes are represented as keys in the key-value store). This is slightly more expensive than dereferencing a virtual memory address, which would be the case if the B+ tree nodes were not stored using objects. Second, the backup approach requires an object to be written durably during each index update, whereas the rebuild approach would not require this step. This durable write affects the performance of index updates (as shown by the measurements in Section 4).

However, the backup approach has another major advantage of permitting variable-size nodes in index B+ trees. Many B+ tree implementations (such as MySQL/InnoDB [6]) allocate fixed size B+ tree nodes. This results in internal fragmentation when the index keys are of variable length (as with commonly used strings). 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.

2.7 Consistency after Crashes

SLIK must handle additional consistency issues that may arise due to server or client crashes.

2.7.1 Server Crash

A server crash can create two consistency issues. First, if a server crashes after inserting an index entry but before updating the object (or after updating an object but before removing old index entries), the crash may leave behind extraneous index entries that will not be deleted by normal operations.

These entries can be garbage collected by a background process. Occassionally, 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 an `entryRemove` request to the index server. If the table partition corresponding to an entry is being recovered, the collector simply skips that entry: it will be removed during the next scan.

We chose to exclude this garbage collector from our implementation as it would have added complexity for little benefit. The orphan entries do not affect correctness as they get filtered out during lookups by the consistency algorithm in the previous section. Further, the wasted space from these entries would be negligible. Assuming conservatively a mean time to failure for servers of about 4 months [18], 10 indexed object writes (or overwrites) in progress at the time of a crash, and 100 B for each index entry, the total amount of garbage accumulated will be less than 3 KB per server per year.

The second consistency issues arises if an indexlet server crashes while inserting or removing an entry, it can cause consistency issues in the internal B+ tree structure. Index insertions and deletions may cause nodes of the B+ tree to be split or joined, which requires updates to multiple nodes (and the objects that encapsulate each node). In order to maintain the consistency of the index across server crashes, multi-object updates must 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.

2.7.2 Client Crash

SLIK has been designed such that a client crash does not affect consistency: all 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.

2.8 Large Scale Operations vs. Scalability

To maximize scalability, large-scale long-running operations must not block other operations. The number of other operations blocked by a given operation is likely to be proportional to the size of the data set blocked.

This means that an operation may hold a lock on a small amount of data for a comparatively long amount of time while a lock on a larger set of data needs to be held for a short amount of time. Hence, SLIK performs long-running bulk operations such as index creation/deletion and migration in the background, without blocking normal operations.

2.8.1 Index Creation

When a new index is created for an existing table in SLIK, it needs to be populated with the index key information from the table's objects. This requires a scan of the entire table, which could take a considerable amount of time for a large table. Furthermore, objects in the table may need to be reformatted to include the corresponding secondary keys, which requires rewriting the objects in the table.

One approach is to lock the table for the duration of table scanning and object rewriting. However, this is not scalable: as tables get larger and larger, the lock must be held for a longer and longer time period, during which period normal requests cannot be serviced.

In order to allow the system to function normally even during schema changes, SLIK populates a new index in the background, without locking the table. The new index should not be used for lookups until index creation is complete. However, the table is locked only long enough to create an empty indexlet. Once the lock is released, other operations on the table can be serviced while the index is being populated. For example, lookups on other indexes (or the primary key) can be serviced. Additionally, objects can be written into the table. These writes will update the new index as well as existing ones.

To populate the new index with entries corresponding to the objects already in the underlying table, client-level code scans this table, reading each object and then rewriting it. Before rewriting the object, the client can restructure the object if the schema has changed. The act of rewriting the object creates an entry in the new index corresponding to this object. So, once all of the objects have been scanned 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.

Index deletion behaves similarly to index creation. The index can be deleted while leaving all of the secondary keys present in objects. Then, the objects can be scanned and a follow-up step can remove the keys.

Index creation and deletion represent additional situations where SLIK permits temporary inconsistencies in its implementation, but those internal inconsistencies do not result in inconsistent behavior for applications.

2.8.2 Live Index Split and Migration

Indexlets need to be reconfigurable – we should be able to split one if it gets too large, or migrate one from one server to another. This requires moving index data in bulk from one server to another, which could take a significant amount of time. In the case of migration, the entire indexlet is moved; for splitting, a part of the indexlet is moved.

A straightforward approach would be to lock the indexlet, copy the relevant part to another server, and then unlock. However, this blocks out users from accessing this indexlet and any objects in the data table indexed by it, for the entire duration of this operation.

SLIK uses a different approach: it allows other operations to proceed concurrently on an indexlet that is being copied to another server. SLIK keeps track of the mutations that have occured since the copying started (in a log), and transfers these over as well. A lock is then required only for a short duration of time, while copying over the last mutation. This is similar to approaches used in the past for applications such as virtual machine migration [22] and process migration [30].

3 Implementation

To help us better understand SLIK's design decisions, we implemented it on RAMCloud [24]. RAMCloud is a distributed in-memory key-value storage system and has some important properties that make it a good platform for implementing SLIK. RAMCloud is designed for large-scale applications: this helps us understand if SLIK's architecture can be used for such applications as well. Further, RAMCloud is designed to operate at lowest possible latency by keeping all data in DRAM and using high performance networking: this allows to see whether SLIK's design is efficient enough to operate in ultra-low latency environments. Finally, RAMCloud is open-source and available freely [9]. This has allowed us to make SLIK available freely in open-source format since the inception of the project.

In the previous section, we described the implementation-independent design and architecture of SLIK. In this section, we describe how SLIK was implemented in the context of RAMCloud.

3.1 Overview of RAMCloud

RAMCloud [24] is a storage system that aggregates the memories of thousands of servers into a single coherent key-value store (Figure 3). It offers remote read times of 4.7 μs and write times of 13.5 μs for small objects.

Each storage server contains 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 the objects in tables [27].

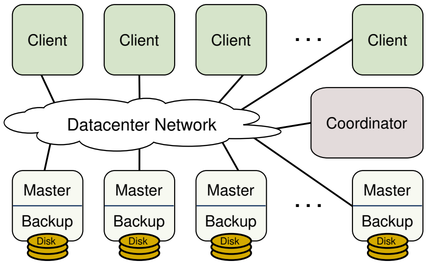


Figure 3: RAMCloud cluster architecture.

A *backup* module uses local disk or flash memory to store backup copies of log information for crash recovery [23]. Each master's log is divided into small segments, and the master scatters multiple replicas of each segment across the available backups. This allows a master's data to be reconstructed within 1-2 s after a crash. RAMCloud uses a small amount of non-volatile memory on each backup, which allows it to declare writes durable as soon as updates have been received by backups, without waiting for disk I/O.

The masters and backups are managed by a central *coordinator* that handles configuration-related issues. The coordinator is a highly reliable and available system (with active and standby instances), but is not normally involved in operations other than those querying or modifying configuration information.

3.2 Implementing Coordination Service for Secondary Indexing

We modified the RAMCloud coordinator to also store and disseminate the metadata about index structures and the servers on which indexlets are stored. When a client starts, it queries the coordinator for the configuration and caches it locally. If this cached configuration becomes stale, the client library discovers this when it sends a query to a server that no longer stores the desired infomation. The client then flushes the local configuration for that table from its cache and fetches up-to-date information from the coordinator (this is described in further detail in [24]).

3.3 Recovering Indexes after Crashes

SLIK stores each indexlet in a RAMCloud table (Section 2.6). Since RAMCloud can recover lost tablets within 1-2 seconds after server crash [23], this ensures that indexes can also be recovered quickly. However, RAMCloud can achieve 1-2 *s* crash recovery only for tablets that are smaller than 500 MB in size. For tablets that are larger than this, RAMCloud will split the tablet during recovery and assign each sub-tablet to a different server, so all of the lost data can be recovered quickly. However, such splitting cannot be used for indexlet backing tables as the B+ tree structure requires all of the objects in the backing table to be present on a single server. Thus, to ensure fast indexlet recovery, SLIK ensures that indexlets are no larger than 500 MB in size. It does this by carrying out live splitting and migration of indexlets that grow beyond the threshold.

CPU	Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)
RAM	24 GB DDR3 at 800 MHz
Flash Disks	2x Crucial M4 SSDs CT128M4SSD2 (128 GB)
NIC	Mellanox ConnectX-2 InfiniBand HCA
Switch	Mellanox SX6036 (4X FDR)

Table 3: The server hardware configuration used for benchmarking. All nodes ran Linux 3.16.0 and were connected to a two-level InfiniBand switching fabric.

3.4 Using Log Structured Memory

Our implementation leverages RAMCloud's log-structured approach of storing data to simplify its implementation. First, it uses this log to implement atomicity for multi-node updates discussed in Section 2.7. Second, it uses the log to keep track of the mutations during an index split and/or migration discussed in Section 2.8.2. More concretely, it migrates the relevant data from an indexlet by scanning the log on that server. When it reaches the head of the log, it locks the log head to migrate the last changes (if any).

4 Evaluation

We evaluated the RAMCloud implementation of SLIK to answer the following questions:

- Does SLIK provide low latency? Is it efficient enough to perform index operations at latencies comparable to other RAMCloud operations?
- Is SLIK scalable? Does its performance scale as the number of servers increases?
- How does the scalability of independent partitioning compare to that of colocation?
- How does the performance of indexing with SLIK compare to other state-of-the-art systems?

We chose H-Store [19] for comparison with SLIK because H-Store and VoltDB (which is H-Store's commercial sibling) are in-memory database systems that are becoming widely adopted. We tuned H-Store for each test to get best performance with assistance from H-Store developers [25]. H-Store uses the indexlet/tablet colocation approach to partitioning, so a column can be specified such that all data gets partitioned according to that column. We evaluated H-Store with multiple data partitioning schemes where applicable.

We ran all experiments on an 80-node cluster of identical commodity servers (see Table 3).

4.1 Latency

We first evaluate the latency of basic index operations (lookups and overwrites) using a table with a single

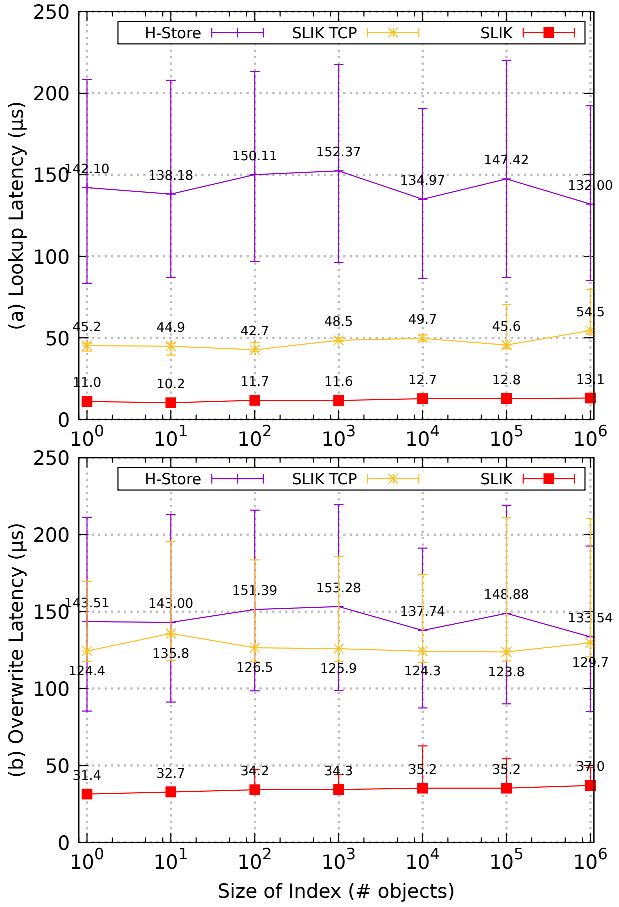


Figure 4: Latency of basic operations as index size increases: (a) read a single object using a secondary key; (b) overwrite an existing object. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements.

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.

4.1.1 Basic Latency

Figure 4 graphs the latency for single-object index operations on a log scale. The measurements were done with a single client accessing a single table, where each object 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.

SLIK lookup: The median time for an index lookup that returns a single object is about 11 µ*s* for a small index and 13.1 µ*s* for an index with a million entries. An index lookup issues two RPCs that read data sequentially (as discussed in Section 2.2) – the time for a non-indexed read in RAMCloud is about 5 µ*s*, making the minimum time required for an index lookup to be 10 µ*s*. The rest of the time is accounted for by the B+ tree lookup time.

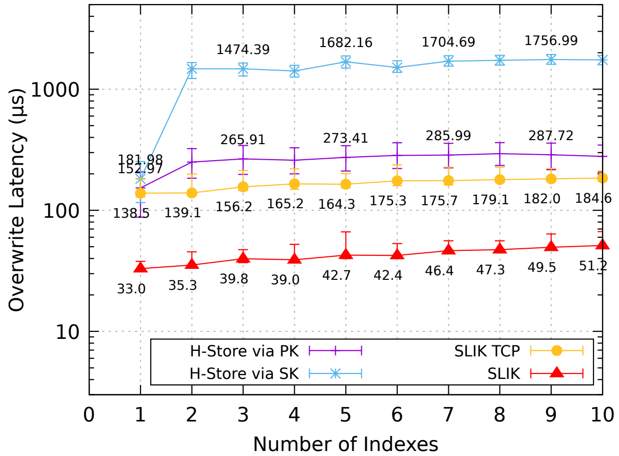


Figure 5: Latency of overwrites as a function of the number of secondary indexes. Each data point displays the 10th percentile, median, and 90th percentile latencies over 1000 measurements. For H-Store's line *via Pk*, it was partitioned by the primary key and for the line *via Sk*, it was partitioned by the first secondary key. In both the cases, overwrites were done by querying via the primary key. The *y* axis uses a log scale.

SLIK overwrite: The median time for `overwrite` ranges from 31.4 µ*s* to 37 µ*s* depending on index size. This is the total cost for doing two sequential durable writes: the first to the index and the second to the object (as discussed in Section 2.3). The removal of old index entries is handled in the background after the overwrite RPC returns to the client.

Comparison: In this benchmark, H-Store is run on a single server so that it uses a single partition for its data and index. It executes all reads and writes locally and no data needs to be transferred to other servers. SLIK provides 3-way distributed replication of objects and index entries to durable backups, whereas H-Store does not perform replication and the durability is disabled. SLIK is about 10x faster than H-Store for lookups and about 4x faster for overwrites.

SLIK is designed to introduce minimal overheads so that it can harness the benefits of low-latency networks and kernel bypass (via InfiniBand). However, we also performed this benchmark by running SLIK with the same network as H-Store: TCP over the InfiniBand network (without kernel bypass). Even in this configuration, SLIK is considerably faster than H-Store for lookups. For overwrites, SLIK provides similar latency as H-Store, but it does so while providing 3-way distributed replication of all data.

4.1.2 Impact of Multiple Secondary Indexes on Overwrite Latency

Figure 5 graphs the latency for overwriting an object as the number of secondary indexes increases. The measurements were done with a single client accessing a single table with 1M objects, where each object has a

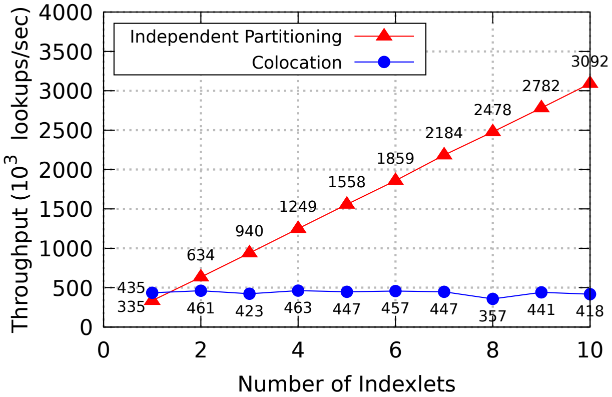


Figure 6: Total index lookup throughput when a single index is divided into multiple indexlets on different servers and queried via multiple clients.

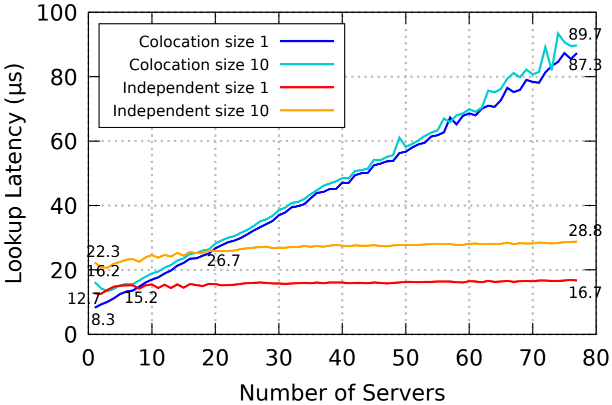


Figure 7: Latency for index lookup when a single index is divided into multiple indexlets on different servers. The size refers to the number of objects returned by a lookup.

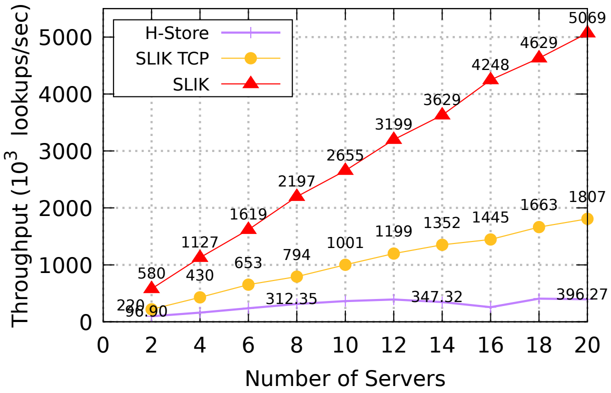


Figure 8: Total index lookup throughput when a single index is divided into multiple indexlets on different servers and queried via multiple clients.

30 *B* primary key, a varying number of 30 *B* secondary keys, and a 100 *B* value. For SLIK, each secondary index has a single partition and is located on a different server.

SLIK: The latency increases moderately for tables with more secondary indexes: overwrites take 32.4 µ*s* with 1 secondary index and 49.8 µ*s* with 10 secondary indexes (about a 50% increase). There is an increase because each of the indexes is stored on a separate server and all the servers must be contacted during writes.

Comparison: SLIK performs better, out of the box without any tuning, while providing durability and replication, than a tuned version of H-Store without durability or 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. For the line *via PK*, the partitioning is done based on the primary key and for the line *via Sk*, the partitioning is done via the first secondary key. The performance while updating using the same key that is used to partition all data (line *via PK*) is lower than the latency for updates done using a key that was not used for partitioning (line *via Sk*).

4.2 Scalability

One of our goals is to provide scalable performance as the number of servers increases. 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 increase.

To test this hypothesis, we evaluated scalability along two parameters. The first measures the end-to-end throughput of index lookup as the number of indexlets increases. This experiment uses a single table where each object has a 30 *B* primary key, 30 *B* secondary key and 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 is stored on a different server. For each data point, the number of clients performing lookups and the number of concurrent lookups per client is varied to achieve the maximum throughput for each system. Each request chooses a random key uniformly distributed across indexlets and returns a matching object. H-Store is partitioned based on the key used for lookups, which is its best configuration for this use case.

The second 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.

We first ran these experiments to compare the scalability of the colocation and independent partitioning approaches while keeping everything else the same (Figures 6, 7). We also ran these experiments to evaluate the scalability of SLIK and compare performance with

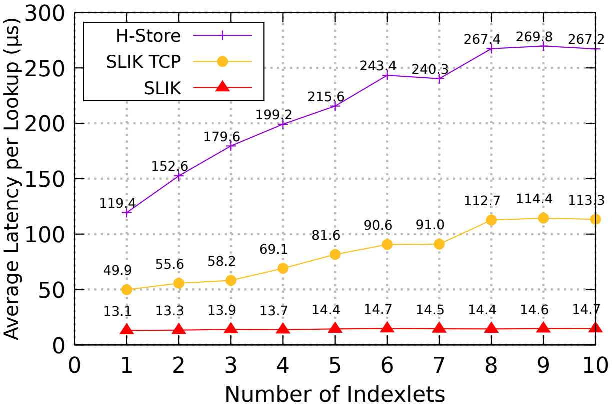


Figure 9: Latency for index lookup when a single index is divided into multiple indexlets on different servers.

H-Store (Figures 8, 9).

4.2.1 Independent Partitioning vs Colocation

We first compare the scalability of independent partitioning with the colocation approach using the setups described earlier. For independent partitioning, we used our implementation of SLIK in RAMCloud. For colocation approach, we used the same implementation and changed the partitioning code to use the colocation approach instead.

These figures confirm that independent partitioning performs better at scale. Figure 6 shows that with independent partitioning, the total lookup throughput increases with the addition of servers, whereas with colocation it does not. Figure 7 shows that as the scale gets larger, the cost of doing two sequential RPCs with independent partitioning is lower than a large number of parallel RPCs with colocation.

4.2.2 System Scalability

We then evaluate how SLIK performs at large scale and also compare against H-Store. Figure 8 graphs the end-to-end throughput of index lookup in SLIK and shows that it scales linearly as the number of indexlets is increased. The throughput for H-Store increases sublinearly with the number of partitions. Figure 9 shows that SLIK's index lookup latency has minimal impact as the number of indexlets is increased, while the latency for H-Store increases because each index lookup must contact all indexlet servers.

4.3 Miscellaneous Benchmarks

4.3.1 Tail Latency

Figure 10 graphs the reverse CDFs for single-object lookup and write operations. A single client performed 100 million reads and overwrites on a table with a million objects (where each object has a 30 *B* primary key, 30 *B* secondary key and 100 *B* value) and there is an index corresponding to the secondary key. The index lookup operations have a median latency of about 15 µ*s*, and `write` operations have a median latency of about 36 µ*s*.

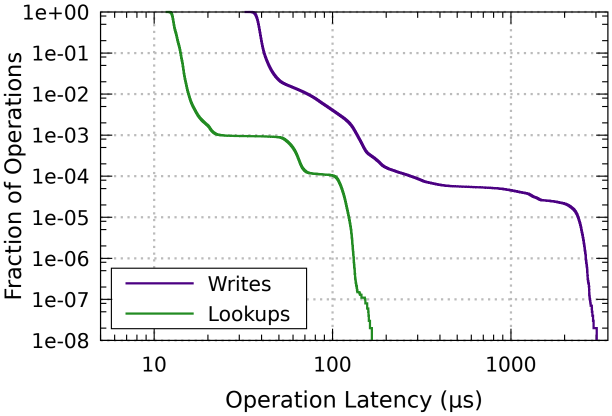


Figure 10: Tail latency distribution for index lookup and `write` operations in SLIK, shown as a reverse CDF with a log scale. A point (x,y) indicates that yth fraction of the 100 M operations measured take at least xµs to complete.

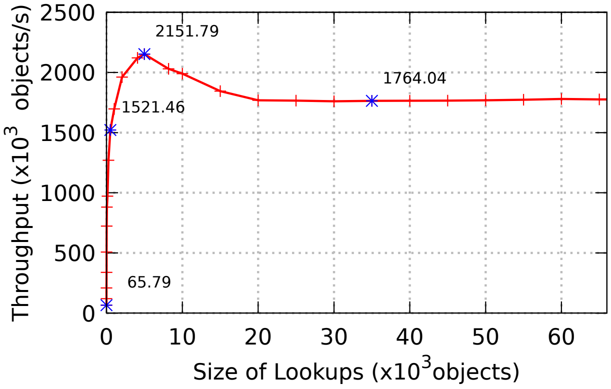


Figure 11: Throughput of index lookup measured by a single client as a function of the total number of objects returned for that lookup.

4.3.2 Range Lookups

Figure 11 graphs the throughput for index lookup as the total number of objects returned in that lookup increases. The experimental setup is the same as the previous experiment. The total throughput increases as the size of lookup is increased, it peaks at about 2*M* objects/*s*, and stabilizes at around 1.7*M* objects/*s*.

5 Related Work

Data storage systems make tradeoffs between various goals: providing higher level data models (like indexing), consistency, durability, scalability and low latency.

Some systems give up certain features in order to optimize for others. For example, MICA [21] is a scalable in-memory key-value store optimized for high throughput; however it does not provide data durability. FaRM [16] is a main memory distributed computing platform that provides low latency and high throughput by exploiting RDMA; however it does not support secondary indexing.

Some systems support consistent and durable secondary indexes but have higher latency than SLIK. Cassandra [20], DynamoDB [3] and Phoenix [7] on HBase [4] provide local secondary indexes which are partitioned using the colocation approach. While these indexes provide high consistency, they also have higher latency as each request needs to contact all the servers (as described in Section 2.2). STI-BT [15] extends an existing key-value store to provide scalable and consistent indexing, and F1 [28] extends Spanner [14] to provide a distributed relational database; however they also have similarly high latencies.

Some of the systems above like DynamoDB [3] and Phoenix [7] on HBase [4] also provide global secondary indexes, but they are only eventually consistent. Moreover, a query on an index can return only those attributes of the object data that have been projected onto that index by the developer and stored with it.

Many other systems provide weak consistency guarantees, while still having latencies comparable to systems above: CouchDB [2] is eventually consistent; PNUTS [13], Espresso [26] and Tao [12] have weak consistency guarantees.

RAMP [10] proposes a new consistency model for transactions called Read Atomic Isolation which can be used to enable strong consistency between object and index updates in a distributed storage system. It proposes three algorithms that offer different trade-offs between speed and the amount of metadata required. The fastest version of RAMP requires two serialized round-trips for writes, which is the same as SLIK but requires a comparatively large amount of metadata that needs to be stored and transported over the network.

H-Store [19] is a main-memory distributed storage system that also provides consistent indexing at a large scale. It partitions data based on a specified attribute (which can be a primary key or a secondary key), which helps the queries based on the partitioning column benefit from the data locality. However, all queries using other attributes need to contact all the partitions to fetch the result, which adversely impacts its performance.

HyperDex [17] is a disk-based large-scale storage system that supports consistent indexing. It partitions data using a novel hyperspace hashing scheme by mapping objects' attributes into a multidimentional space. As the number of attributes increase, the number of hyperspaces increases dramatically. HyperDex alleviates this by partitioning tables with many attributes into multiple lower-dimensional hyperspaces called subspaces. HyperDex also replicates the entire contents of objects in each index. This means that while HyperDex provides an efficient mechanism for search, it uses more storage space for the extra copies of objects. While this is acceptable for disk based systems, it would be very expensive for main-memory based systems.

We have compared approaches taken by other systems and discussed their tradeoffs with the approaches adopted by SLIK in Section 2. Further, in Section 4 we compared SLIK performance with H-Store [19] and found that SLIK outperformed it by a large factor. We also benchmarked HyperDex [17]. However, we omit these benchmarks due to space constraints and because it was hard to quantify how much of its poorer performace was due to its use of disk for storage.

SLIK's most unique aspect is its combination of low latency and consistency at large scale; other systems sacrifice at least one of these.

6 Conclusion

We have shown that it is possible to have durable and consistent secondary indexes in a key-value storage system at extremely low latency and large scale. We made design decisions by considering tradeoffs between various approaches or by developing new algorithms where acceptable solutions did not exist. This design provides secondary indexing that provides better scalability and latency than existing systems, without any tuning for specific use cases.

Modern scalable storage systems need not sacrifice the powerful programming model provided 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 step on the path to a high-function, low-latency, large-scale storage system.

7 Acknowledgments

This work was supported by C-FAR (one of six centers of STARnet, a Semiconductor Research Corporation program, sponsored by MARCO and DARPA), and by grants from Emulex, Facebook, Google, Huawei, Inventec, NEC, NetApp, Samsung, and VMware.

We would like to thank Hector Garcia-Molina and Keith Winstein for their guidance. We would also like to thank Jonathan Ellithorpe, Greg Hill, Collin Lee, Behnam Montazeri, Diego Ongaro, Seo Jin Park, Mendel Rosenblum, Steve Rumble, William Sheu, Ryan Stutsman, and Henry Qin for interesting discussions and feedback at various points in the project. Finally, we would like to thank the anonymous reviewers and our shepherd, Sriram Rao, for their helpful comments.

References

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

[2] CouchDB. http://couchdb.apache.org/.

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

[4] HBase. http://hbase.apache.org/.

[5] MongoDB. http://www.mongodb.org/.

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

[7] Phoenix. http://phoenix.apache.org/.

[8] Redis. http://www.redis.io/.

[9] RAMCloud Git Repository, 2015. https://github.com/PlatformLab/RAMCloud.git.

[10] Bailis, P., Fekete, A., Hellerstein, J. M., Ghodsi, A., and Stoica, I. Scalable atomic visibility with RAMP transactions. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data* (2014), ACM, pp. 27–38.

[11] Baker, J., Bond, C., Corbett, J. C., Furman, J., Khorlin, A., Larson, J., Leon, J.-M., Li, Y., Lloyd, A., and Yushprakh, V. Megastore: Providing scalable, highly available storage for interactive services. In *CIDR* (2011), vol. 11, pp. 223–234.

[12] Bronson, N., Amsden, Z., Cabrera, G., Chakka, P., Dimov, P., Ding, H., Ferris, J., Giardullo, A., Kulkarni, S., Li, H. C., et al. TAO: Facebook's distributed data store for the social graph. In *USENIX Annual Technical Conference* (2013), pp. 49–60.

[13] Cooper, B. F., Ramakrishnan, R., Srivastava, U., Silberstein, A., Bohannon, P., Jacobsen, H.-A., Puz, N., Weaver, D., and Yerneni, R. PNUTS: Yahoo!'s hosted data serving platform. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1277–1288.

[14] Corbett, J. C., Dean, J., Epstein, M., Fikes, A., Frost, C., Furman, J., Ghemawat, S., Gubarev, A., Heiser, C., Hochschild, P., et al. Spanner: Googles globally distributed database. *ACM Transactions on Computer Systems (TOCS) 31*, 3 (2013), 8.

[15] Diegues, N., and Romano, P. Sti-bt: A scalable transactional index. In *Distributed Computing Systems (ICDCS), 2014 IEEE 34th International Conference on* (2014), IEEE, pp. 104–113.

[16] Dragojević, A., Narayanan, D., Hodson, O., and Castro, M. Farm: Fast remote memory. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation, NSDI* (2014), vol. 14.

[17] Escriva, R., Wong, B., and Sirer, E. G. HyperDex: A distributed, searchable key-value store. *ACM SIGCOMM Computer Communication Review 42*, 4 (2012), 25–36.

[18] Ford, D., Labelle, F., Popovici, F. I., Stokely, M., Truong, V.-A., Barroso, L., Grimes, C., and Quinlan, S. Availability in Globally Distributed Storage Systems. In *OSDI* (2010), pp. 61–74.

[19] Kallman, R., Kimura, H., Natkins, J., Pavlo, A., Rasin, A., Zdonik, S., Jones, E. P. C., Madden, S., Stonebraker, M., Zhang, Y., Hugg, J., and Abadi, D. J. H-Store: a high-performance, distributed main memory transaction processing system. *Proceedings of the VLDB Endowment 1*, 2 (2008), 1496–1499.

[20] Lakshman, A., and Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Operating Systems Review 44*, 2 (2010), 35–40.

[21] Lim, H., Han, D., Andersen, D. G., and Kaminsky, M. Mica: A holistic approach to fast in-memory key-value storage. *management 15*, 32 (2014), 36.

[22] Nelson, M., Lim, B.-H., Hutchins, G., et al. Fast transparent migration for virtual machines. In *USENIX Annual Technical Conference, General Track* (2005), pp. 391–394.

[23] Ongaro, D., Rumble, S. M., Stutsman, R., Ousterhout, J., and Rosenblum, M. Fast crash recovery in RAMCloud. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (2011), ACM, pp. 29–41.

[24] Ousterhout, J., Gopalan, A., Gupta, A., Kejriwal, A., Lee, C., Montazeri, B., Ongaro, D., Park, S. J., Qin, H., Rosenblum, M., Rumble, S., Stutsman, R., and Yang, S. The RAMCloud Storage System. *ACM Trans. Comput. Syst. 33*, 3 (Aug. 2015), 7:1–7:55.

[25] Pavlo, A. Personal communications, March 17 2015.

[26] Qiao, L., Surlaker, K., Das, S., Quiggle, T., Schulman, B., Ghosh, B., Curtis, A., Seeliger, O., Zhang, Z., Auradar, A., Beaver, C., Brandt, G., Gandhi, M., Gopalakrishna, K., Ip, W., Jgadish, S., Lu, S., Pachev, A., Ramesh, A., Sebastian, A., Shanbhag, R., Subramaniam, S., Sun, Y., Topiwala, S., Tran, C., Westerman, J., and Zhang, D. On brewing fresh Espresso: LinkedIn's distributed data serving platform. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (2013), ACM, pp. 1135–1146.

[27] Rumble, S. M., Kejriwal, A., and Ousterhout, J. Log-structured memory for DRAM-based storage. In *Proceedings of the 12th USENIX Conference on File and Storage Technologies* (2014), pp. 1–16.

[28] Shute, J., Oancea, M., Ellner, S., Handy, B., Rollins, E., Samwel, B., Vingralek, R., Whipkey, C., Chen, X., Jegerlehner, B., et al. F1: The fault-tolerant distributed RDBMS supporting google's ad business. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data* (2012), ACM, pp. 777–778.

[29] Stutsman, R., Lee, C., and Ousterhout, J. Experience with Rules-Based Programming for Distributed, Concurrent, Fault-Tolerant Code. In *2015 USENIX Annual Technical Conference (USENIX ATC 15)* (Santa Clara, CA, July 2015), USENIX Association, pp. 17–30.

[30] Zayas, E. Attacking the process migration bottleneck. *ACM SIGOPS Operating Systems Review 21*, 5 (1987), 13–24.