Scalable Low-Latency Indexes for a Key-Value Store

Ankita Kejriwal

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

Stanford University
Conjecture

Can a key value store support strongly consistent secondary indexes while operating at low latency and large scale?
Summary of Results

- **Scalable Low-latency Indexes for a Key-value Store: SLIK**
  - Enables multiple secondary keys for each object
  - Allows lookups and range queries on these keys

- **Key design features:**
  - **Scalability** using independent partitioning
  - **Strong consistency** using an ordered write approach

- **Implemented in RAMCloud**
  - Low-latency, DRAM-based, distributed key-value store

- **Performance:**
  - **Scalability:** Linear throughput increase with increasing number of partitions
  - **Low-latency:** 11-13 µs indexed reads, 29-37 µs durable writes/overwrites
  - **Latency:** approximately 2x non-indexed reads and writes
Talk Outline

- Motivation
- Design
- Performance
- Related Work
- Summary
Motivation

Traditional RDBMs

MySQL
Motivation

Traditional RDBMs → NoSQL Systems

MySQL

+ scalability

- data models

- consistency
Motivation

Traditional RDBMs → NoSQL Systems

- MySQL
  - + scalability
  - - data models
  - - consistency

NoSQL Systems →

+ consistency

H-Base
Espresso
RAMCloud
Yesquel
Spanner
Megastore
HyperDex
MongoDB
H-Store
PNUTS
Tao

+ low latency

+ data models

Memcached
Motivation

Traditional RDBMs → NoSQL Systems

+ scalability
- data models
- consistency

MySQL

MySQL

+ consistency
- data models
- low latency

H-Base
Espresso

RAMCloud

Yesquel
Spanner
Megastore
HyperDex
MongoDB
H-Store

PNUTS
Tao

+ consistency
+ data models
+ low latency

Memcached
Talk Outline

- Motivation
- Design
- Performance
- Related Work
- Summary
Design

- Data model
- Scalability
- Strong consistency
- Storage
- Durability
- Availability
Design

- Data model
- Scalability
- Strong consistency
- Storage
- Durability
- Availability
Design

- Data model
- **Scalability**
- **Strong consistency**
- Storage
- **Durability**
- Availability
Scalability

- Nearly constant low latency irrespective of the server span
- Linear increase in throughput with the server span
Index Partitioning: Colocation

- Colocate index entries and objects
- One of the keys used to partition the objects and indexes
Index Partitioning: Colocation

- Colocate index entries and objects
- One of the keys used to partition the objects and indexes
- No association between index partitions and index key ranges

**Metadata:**
- tablet & indexlet w/ pk 1 to 3: S 1
- tablet & indexlet w/ pk 4 to 6: S 2
- tablet & indexlet w/ pk >= 7: S 3
Index Partitioning: Colocation

Client query: objects with index key between m - q
Index Partitioning: Colocation

Client query: objects with index key between m - q

Server 1

Indexlet

Tablet

a \rightarrow 2
n \rightarrow 3
q \rightarrow 1

1 q rose
2 a tulip
3 n violet

Server 2

Indexlet

Tablet

e \rightarrow 4
g \rightarrow 6
v \rightarrow 5

4 e clover
5 v daily
6 g iris

Server 3

Indexlet

Tablet

b \rightarrow 8
m \rightarrow 7

7 m lily
8 b dahlia
Index Partitioning: Colocation

Client query: objects with index key between m - q

Server 1

Indexlet

Tablet

1 q rose
2 a tulip
3 n violet

Server 2

Indexlet

Tablet

4 e clover
5 v daily
6 g iris

Server 3

Indexlet

Tablet

b 8 dahlia

m 7 lily

135x479 Index Partitioning: Colocation
155x268 Indexlet
155x161 Tablet
163x137 1
186x137 q
219x137 rose
163x112 2
188x112 a
219x112 tulip
163x89 3
187x89 n
216x89 violet
145x193 2
185x193 è
241x192 q
250x192 è
308x161 Tablet
316x125 Tablet
340x113 v
372x113 daily
316x89 6
340x89 g
376x89 iris
389x192 è
407x192 v
434x192 è
301x193 4
311x193 è
320x193 g
337x193 è
308x268 Indexlet
336x295 Server 2
349x295 Server 3
351x28 Client query: objects with index key between m - q
362x28 SLIK
Index Partitioning: Colocation

Client query: objects with index key between m - q

Server 1
- Indexlet
  - Tablet
    - 1: q, rose
    - 2: a, tulip
    - 3: n, violet

Server 2
- Indexlet
  - Tablet
    - 4: e, clover
    - 5: v, daily
    - 6: g, iris

Server 3
- Indexlet
  - Tablet
    - 7: m, lily
    - 8: b, dahlia

Not Scalable!
Index Partitioning: Independent

- Partition each index and table independently
- Partition each index according to sort order for that index
Index Partitioning: Independent

- Partition each index and table independently
- Partition each index according to sort order for that index

**Metadata:**
- tablet w/ pk 1 to 3: S 1
- tablet w/ pk 4 to 6: S 2
- tablet w/ pk >= 7: S 3
- indexlet w/ sk a to g: S 4
- indexlet w/ sk >= h: S 5
Index Partitioning: Independent

Client query: objects with index key between m - q
Index Partitioning: Independent

Client query: objects with index key between m - q
Index Partitioning: Independent

Client query: objects with index key between m - q

Server 1

Tablet
1  q  rose
2  a  tulip
3  n  violet

Server 2

Tablet
4  e  clover
5  v  daily
6  g  iris

Server 3

Tablet
7  m  lily
8  b  dahlia

Server 4

Indexlet
a  ➔  2
b  ➔  8
e  ➔  4
g  ➔  6

Server 5

Indexlet
m  ➔  7
n  ➔  3
q  ➔  1
v  ➔  5

SLIK
Index Partitioning: Independent

Client query: objects with index key between m - q

Server 1

1. q rose
2. a tulip
3. n violet

Server 2

4. e clover
5. v daily
6. g iris

Server 3

m lily
v daily
b dahlia

Server 4

Indexlet

a \rightarrow 2
b \rightarrow 8
e \rightarrow 4
g \rightarrow 6

Server 5

Indexlet

m \rightarrow 7
n \rightarrow 3
q \rightarrow 1
v \rightarrow 5

Scalable!
Scalability

- Nearly constant low latency irrespective of the server span
- Linear increase in throughput with the server span
**Scalability**

- Nearly constant low latency irrespective of the server span
- Linear increase in throughput with the server span
- **Solution:** Use independent partitioning
- **But:** indexed object writes: distributed operations
- Potential consistency issues between indexes and objects
Design

- Data model
- Scalability
- Strong consistency
- Storage
- Durability
- Availability
Design

- Data model
- Scalability
- Strong consistency
- Storage
- Durability
- Availability
Consistency Properties

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

● If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

Students with name between a – d?

- Bob
- Alice
- Frank
- Trent
- Peggy
- Carol

- Alice
- Carol
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

Bob  Alice  Trent  Carol  Peggy  Frank
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

- students with name between a – d?
Consistency Properties

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

Students with name between a – d?

Bob  Alice
Frank  Trent
Peggy  Carol

Alice  Bob  Carol

😊
Consistency properties:

- If an object contains a given secondary key, then an index lookup with that key will return the object.
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

Solution:

- Longer index lifespan (via ordered writes)
- Object data is ground truth and index entries serve as hints
Consistency

**Consistency properties:**

- If an object contains a given secondary key, then an index lookup with that key will return the object
- If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range

**Solution:**

- Longer index lifespan (via ordered writes)
- Object data is ground truth and index entries serve as hints
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range

- **Solution:**
  - Longer index lifespan (via ordered writes)
  - Object data is ground truth and index entries serve as hints

Object

Index Entry

| Bob → Foo | Sam → Foo |

1. Add new index entry
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range

- **Solution:**
  - Longer index lifespan (via ordered writes)
  - Object data is ground truth and index entries serve as hints

1. Add new index entry
2. Modify object
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range

- **Solution:**
  - Longer index lifespan (via ordered writes)
  - Object data is ground truth and index entries serve as hints

1. Add new index entry
2. Modify object
3. Remove old index entry
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object.
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

- **Solution:**
  - Longer index lifespan (via ordered writes).
  - Object data is ground truth and index entries serve as hints.

![Diagram showing object and index entry changes over time](image-url)
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object.
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

- **Solution:**
  - Longer index lifespan (via ordered writes).
  - Object data is ground truth and index entries serve as hints.

![Diagram of consistency and object indexing](image)
Consistency

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object.
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

- **Solution:**
  - Longer index lifespan (via ordered writes).
  - Object data is ground truth and index entries serve as hints.

---

[Diagram showing object and index entries with commit points and operations (write, modify, remove).]
**Consistency**

- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object.
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range.

- **Solution:**
  - Longer index lifespan (via ordered writes).
  - Object data is ground truth and index entries serve as hints.

---

![Diagram](https://via.placeholder.com/150)

- **Object:**
  - **Write object:**
  - **Modify object:**
  - **Remove object:**

- **Index Entry:**
  - **Bob → Foo**
  - **Sam → Foo**

- **Commit Point:**
  - Time
Talk Outline

- Motivation
- Design
- **Performance**
- Related Work
- Summary
Performance: Questions

- Does SLIK provide low latency?
- Does SLIK provide scalability?
- How does the performance of indexing with SLIK compare to other state-of-the-art systems?
Performance: Systems for Comparison

- **H-Store:**
  - Main memory database
  - Data (and indexes) partitioned based on specified attribute
  - Many parameters for tuning
    - Got assistance from developers to tune for each test
    - Examples: txn_incoming_delay, partitioning column

- **HyperDex:**
  - Spaces containing objects
  - Data (and indexes) partitioned using hyperspace hashing
  - Each index contains all object data
  - Designed to use disk for storage
## Hardware

<table>
<thead>
<tr>
<th>Component</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Xeon X3470 (4x2.93 GHz cores, 3.6 GHz Turbo)</td>
</tr>
<tr>
<td>RAM</td>
<td>24 GB DDR3 at 800 MHz</td>
</tr>
<tr>
<td>Flash Disks</td>
<td>2x Crucial M4 SSDs</td>
</tr>
<tr>
<td></td>
<td>CT128M4SSD2 (128 GB)</td>
</tr>
<tr>
<td>NIC</td>
<td>Mellanox ConnectX-2 InfiniBand HCA</td>
</tr>
<tr>
<td>Switch</td>
<td>Mellanox SX6036 (4X FDR)</td>
</tr>
</tbody>
</table>
Latency

Experiments:
1. Lookups: table with single secondary index
2. Overwrites: table with single secondary index
3. Overwrites: varying number of secondary indexes

Configuration:
- Single client
- Single partition for table and (each) index
- Object: 30 B pk, 30 B sk, 100 B value
- SLIK: Three-way replication to durable backups
- H-Store: No replication, durability disabled, single server
Lookup Latency

H-Store SK Partitioned

H-Store PK Partitioned

HyperDex

SLIK tcp

SLIK

0 50 100 150 200 250

0 10 2 10 3 10 4 10 5 10 6

(a) Lookup Latency (µs)

(b) Overwrite Latency (µs)
Lookup Latency

- **H-Store**
- **SLIK TCP**
- **SLIK**

![Graph showing Lookup Latency vs Size of Index](image-url)
Overwrite Latency

![Graph showing Overwrite Latency](image)

- **H-Store**: Overwrite Latency values range from 124.4 µs to 153.28 µs.
- **SLIK TCP**: Overwrite Latency values range from 124.3 µs to 137.74 µs.
- **SLIK**: Overwrite Latency values range from 31.4 µs to 37.0 µs.

**Size of Index (# objects)**

- The x-axis represents the size of the index, ranging from $10^0$ to $10^6$.
- The y-axis represents the overwrite latency in microseconds (µs), ranging from 0 to 250 µs.
Multiple Secondary Indexes

![Graph showing Overwrite Latency (µs) vs. Number of Indexes for different methods: H-Store via PK, H-Store via SK, SLIK TCP, SLIK. The graph indicates that the Overwrite Latency increases with the number of indexes for all methods, with SLIK TCP having the highest latency, followed by SLIK, H-Store via SK, and H-Store via PK.](image-url)
Multiple Secondary Indexes

![Graph showing Overwrite Latency (µs) vs. Number of Indexes for different methods: H-Store via PK, H-Store via SK, SLIK TCP, SLIK. Each method has a line with markers indicating data points.](image-url)
Multiple Secondary Indexes

The diagram shows the overwrite latency (in microseconds) as a function of the number of indexes. The latency is measured for different storage systems and index access types:

- **H-Store via PK**
- **H-Store via SK**
- **SLIK TCP**
- **SLIK**

The x-axis represents the number of indexes, ranging from 0 to 10, and the y-axis represents the overwrite latency in microseconds, ranging from 10 to 1000 (in a logarithmic scale).

The data points for each category are marked with different symbols and line styles, allowing for a clear comparison of performance across different access methods and storage systems.
Scalability

**Compare:** (a) partitioning approaches (b) systems

**Experiments:**

1. Lookup throughput with increasing number of partitions
2. Lookup latency with increasing number of partitions

**Configuration:**

- Single table with one secondary index
- Table and index partitioned across servers
- Object: 30 B pk, 30 B sk, 100 B value
- Throughput experiments: Loaded system
- Latency experiments: Unloaded system
Scalability: Throughput

Throughput (10^3 lookups/sec) vs Number of Indexlets

- Independent Partitioning
- Colocation

Slide 58
Scalability: Throughput

Throughput (10^3 lookups/sec)

Number of Servers

H-Store
SLIK TCP
SLIK

<table>
<thead>
<tr>
<th>Number of Servers</th>
<th>H-Store</th>
<th>SLIK TCP</th>
<th>SLIK</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>580</td>
<td>430</td>
<td>220</td>
</tr>
<tr>
<td>4</td>
<td>1127</td>
<td>653</td>
<td>560</td>
</tr>
<tr>
<td>6</td>
<td>1619</td>
<td>794</td>
<td>960</td>
</tr>
<tr>
<td>8</td>
<td>2197</td>
<td>1001</td>
<td>1352</td>
</tr>
<tr>
<td>10</td>
<td>2655</td>
<td>1199</td>
<td>1445</td>
</tr>
<tr>
<td>12</td>
<td>3199</td>
<td>1352</td>
<td>1663</td>
</tr>
<tr>
<td>14</td>
<td>3629</td>
<td>1445</td>
<td>1807</td>
</tr>
<tr>
<td>16</td>
<td>4248</td>
<td>1663</td>
<td>1996</td>
</tr>
<tr>
<td>18</td>
<td>4629</td>
<td>1807</td>
<td>5069</td>
</tr>
</tbody>
</table>
Scalability: Latency

- Colocation size 1
- Colocation size 10
- Independent size 1
- Independent size 10

Lookup Latency (µs) vs. Number of Servers

Slide 60
Scalability: Latency

![Graph showing the relationship between the number of indexlets and average latency per lookup for H-Store, SLIK TCP, and SLIK.]
Talk Outline

- Motivation
- Design
- Performance
- Related Work
- Summary
Related Work

Data storage system

- **Data model** (spectrum from key-value to relational)
- **Consistency** (spectrum from eventual to strong)
- **Performance**: latency and/or throughput
Current Web Scale Datastores

- **Strong**: Causal, SI, “Define your own”
- **Eventual**: Better

Consistency Level

Read / write latency (approx)

100s 10s 1s 100ms 10ms 1ms 100µs 10µs
Current Web Scale Datastores

- **Consistency Level**
  - Strong (Better)
  - Causal, SI, “Define your own” (Eventual)

- **Read / write latency (approx)**
  - 100s
  - 10s
  - 1s
  - 100ms
  - 10ms
  - 1ms
  - 100µs
  - 10µs

- **Datastores**
  - Espresso
  - PNUTS
  - CouchDB
  - Tao

SLIK
Current Web Scale Datastores

Consistency Level

- Strong
- Causal, SI, “Define your own”
- Eventual

Read / write latency (approx)

- Better

MongoDB
- Spanner
- H-Store
- HyperDex
- Espresso
- PNUTS
- CouchDB
- Tao

SLIK
Current Web Scale Datastores

Consistency Level

Strong

Causal, SI, “Define your own”

Eventual

Read / write latency (approx)

Better
Current Web Scale Datastores

Consistency Level

Eventual

Causal, Si, “Define your own”

Strong

Better

Read / write latency (approx)

Better

MongoDB

Spanner

H-Store

H-Base

MegaStore

Cassandra

Espresso

PNUTS

CouchDB

Tao

SLIK
Talk Outline

- Motivation
- Design
- Performance
- Related Work
- Summary
Conjecture

Can a key value store support **strongly consistent secondary indexes**
while operating at **low latency** and **large scale**?
A key value store can support strongly consistent secondary indexes while operating at low latency and large scale.
A key value store can support strongly consistent secondary indexes while operating at low latency and large scale.

By using ordered writes and treating indexes as hints

Lookups and range queries on secondary keys
A key value store can support strongly consistent secondary indexes while operating at low latency and large scale.

By using ordered writes and treating indexes as hints and by using independent partitioning, we get:

- Linear throughput increase
- Minimal impact on latency as the scale increases
A key value store can support strongly consistent secondary indexes while operating at low latency and large scale.

By using approaches that have minimal overheads we get:
- 11-13 µs lookups and 30-37 µs (over)writes

Lookups and range queries on secondary keys

By using ordered writes and treating indexes as hints

By using independent partitioning we get:
- linear throughput increase and minimal impact on latency as the scale increases
A key value store can support strongly consistent secondary indexes while operating at low latency and large scale.

By using approaches that have minimal overheads we get:
- 11-13 $\mu$s lookups and
- 30-37 $\mu$s (over)writes

Lookups and range queries on secondary keys

By using ordered writes and treating indexes as hints

By using independent partitioning we get: linear throughput increase and minimal impact on latency as the scale increases
Thank you!

Code available free and open source: github.com/PlatformLab/RAMCloud

My papers and other information at: stanford.edu/~ankitak
I can be reached at: ankitak@cs.stanford.edu