Scalable Low-Latency Indexes for a Key-Value Store

Ankita Kejriwal

With Arjun Gopalan, Ashish Gupta, Greg Hill, Zhihao Jia, Stephen Yang
and John Ousterhout
A key value store can 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**

- **Performance:**
  - Linear throughput increase with increasing number of partitions
  - 11-13 µs indexed reads
  - 29-37 µs durable writes/overwrites of objects with one indexed attribute
  - Latency approx. 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

+ scalability
- data models
- consistency

MySQL

MySQL + consistency
+ data models

H-Base
Espresso

RAMCloud

Yesquel
Spanner
Megastore
HyperDex
MongoDB
H-Store
PNUTS
Tao

+ consistency

+ low latency

+ data models

Memcached
Motivation

Traditional RDBMSs

- MySQL

+ scalability
- data models
- consistency

NoSQL Systems

+ consistency
- data models
- low latency

RAMCloud

+ consistency
- data models
- low latency

H-Base
Espresso

Yesquel
Memcached

Spanner
Megastore
HyperDex
PNUTS
Tao

H-Store
MongoDB
H-Store
Megastore
HyperDex
PNUTS
Tao

SLIK
Talk Outline

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

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

- Scalability
- Strong consistency
Design

- Scalability
- Strong consistency
Design

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

- **Strong consistency**
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
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
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
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 m 7
    - 7 m lily
    - 8 b dahlia

SLIK
Index Partitioning: Colocation

Client query: objects with index key between m - q

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

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

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

Server 3
- Tablet 7: m lily
- Tablet 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
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

Scalable!
Index Partitioning: Lookup Latency

[Graph showing the relationship between number of servers and lookup latency for different colocation and independent sizes.]

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

Values at specific numbers of servers:
- Colocation size 1: 8.3 at 10 servers, 16.7 at 20 servers, 22.3 at 30 servers, 26.7 at 40 servers, 28.8 at 50 servers
- Colocation size 10: 87.3 at 10 servers, 89.7 at 20 servers
- Independent size 1: 15.2 at 10 servers, 16.7 at 20 servers
- Independent size 10: 16.7 at 10 servers

Lookup Latency (μs) vs. Number of Servers
Index Partitioning: Lookup Throughput

![Graph showing the relationship between the number of indexlets and throughput for independent partitioning and colocation.](image)

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

<table>
<thead>
<tr>
<th>Number of Indexlets</th>
<th>Independent Partitioning</th>
<th>Colocation</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>435</td>
<td>335</td>
</tr>
<tr>
<td>1</td>
<td>634</td>
<td>461</td>
</tr>
<tr>
<td>2</td>
<td>940</td>
<td>423</td>
</tr>
<tr>
<td>3</td>
<td>1249</td>
<td>463</td>
</tr>
<tr>
<td>4</td>
<td>1558</td>
<td>447</td>
</tr>
<tr>
<td>5</td>
<td>1859</td>
<td>457</td>
</tr>
<tr>
<td>6</td>
<td>2184</td>
<td>447</td>
</tr>
<tr>
<td>7</td>
<td>2478</td>
<td>357</td>
</tr>
<tr>
<td>8</td>
<td>2782</td>
<td>441</td>
</tr>
<tr>
<td>9</td>
<td></td>
<td>418</td>
</tr>
<tr>
<td>10</td>
<td>3092</td>
<td></td>
</tr>
</tbody>
</table>
Design

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

- **Strong consistency**
Design

- **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

- **Strong consistency**
Design

● **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

● **Strong consistency**
Design

- **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

- **Strong consistency**
  - With minimal performance overheads
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
Frank
Peggy
Alice
Trent
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.
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?

- Alice
- Bob
- Carol
- Peggy

- Trent
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

- **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

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
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.

![Diagram showing consistency](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
Design

- **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

- **Strong consistency**
  - With minimal performance overheads
  - **Solution:** Ordered write approach + treat indexes as hints
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></th>
<th>Description</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</td>
<td>2x Crucial M4 SSDs</td>
</tr>
<tr>
<td>Disks</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
- 179.53
- 160.16
- 195.06
- 202.07
- 192.46
- 207.10
- 209.37

#### H-Store PK Partitioned
- 987.25
- 939.84
- 961.17
- 1019.82
- 1048.86
- 1010.92
- 968.72

#### HyperDex
- 648
- 667
- 671
- 667
- 660
- 657
- 661

#### SLIK tcp
- 124.4
- 135.8
- 126.5
- 125.9
- 124.3
- 123.8
- 129.7

#### SLIK
- 31.4
- 32.7
- 34.2
- 34.3
- 35.2
- 35.2
- 37.0

---

**Diagram:**
- **H-Store**
- **SLIK TCP**
- **SLIK**

---

**Table:**

<table>
<thead>
<tr>
<th>Size of Index (# objects)</th>
<th>H-Store SK Partitioned</th>
<th>H-Store PK Partitioned</th>
<th>HyperDex</th>
<th>SLIK tcp</th>
<th>SLIK</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>179.53</td>
<td>987.25</td>
<td>648</td>
<td>124.4</td>
<td>31.4</td>
</tr>
<tr>
<td>100</td>
<td>160.16</td>
<td>939.84</td>
<td>667</td>
<td>135.8</td>
<td>32.7</td>
</tr>
<tr>
<td>1,000</td>
<td>195.06</td>
<td>961.17</td>
<td>671</td>
<td>126.5</td>
<td>34.2</td>
</tr>
<tr>
<td>10,000</td>
<td>202.07</td>
<td>1019.82</td>
<td>667</td>
<td>125.9</td>
<td>34.3</td>
</tr>
<tr>
<td>100,000</td>
<td>192.46</td>
<td>1048.86</td>
<td>660</td>
<td>124.3</td>
<td>35.2</td>
</tr>
<tr>
<td>1,000,000</td>
<td>207.10</td>
<td>1010.92</td>
<td>657</td>
<td>123.8</td>
<td>35.2</td>
</tr>
<tr>
<td>10,000,000</td>
<td>209.37</td>
<td>968.72</td>
<td>661</td>
<td>129.7</td>
<td>37.0</td>
</tr>
</tbody>
</table>
Lookup Latency

![Graph showing Lookup Latency vs Size of Index (# objects) for different systems: H-Store, SLIK TCP, SLIK. The x-axis represents the size of the index in logarithmic scale, ranging from $10^0$ to $10^6$, and the y-axis represents lookup latency in microseconds, ranging from 0 to 250. The graph compares the performance of H-Store, SLIK TCP, and SLIK across different sizes of the index, with H-Store showing a steady increase in latency, SLIK TCP showing a more fluctuating pattern, and SLIK showing a consistent performance.](image-url)
Overwrite Latency

![Graph showing Overwrite Latency](image)

- **H-Store**
  - Overwrite Latency (µs): 143.51, 143.00, 151.39, 153.28, 137.74, 148.88, 133.54

- **SLIK TCP**
  - Overwrite Latency (µs): 124.4, 135.8, 126.5, 125.9, 124.3, 123.8, 129.7

- **SLIK**
  - Overwrite Latency (µs): 31.4, 32.7, 34.2, 34.3, 35.2, 35.2, 37.0

**Size of Index (# objects)**

- **10^0**
- **10^1**
- **10^2**
- **10^3**
- **10^4**
- **10^5**
- **10^6**
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, and SLIK.](slide57.png)
Multiple Secondary Indexes

![Graph showing Overwrite Latency (µs) vs Number of Indexes for H-Store via PK, H-Store via SK, SLIK TCP, and SLIK.](image)
### Multiple Secondary Indexes

<table>
<thead>
<tr>
<th>Number of Indexes</th>
<th>Overwrite Latency (µs)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>H-Store via PK</td>
</tr>
<tr>
<td>1</td>
<td>181.98</td>
</tr>
<tr>
<td>2</td>
<td>265.91</td>
</tr>
<tr>
<td>3</td>
<td>273.41</td>
</tr>
<tr>
<td>4</td>
<td>285.99</td>
</tr>
<tr>
<td>5</td>
<td>287.72</td>
</tr>
<tr>
<td>6</td>
<td>288.72</td>
</tr>
<tr>
<td>7</td>
<td>287.72</td>
</tr>
<tr>
<td>8</td>
<td>288.72</td>
</tr>
<tr>
<td>9</td>
<td>288.72</td>
</tr>
<tr>
<td>10</td>
<td>288.72</td>
</tr>
</tbody>
</table>

![Graph showing overwrite latency for different numbers of indexes and storage methods.](chart.png)
**Scalability**

**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 experiment: Loaded system
- Latency experiment: Unloaded system
### Scalability: Lookup Throughput

**Throughput (10^3 lookups/sec)**

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

**Number of Servers**

- 0
- 2
- 4
- 6
- 8
- 10
- 12
- 14
- 16
- 18
- 20

- Throughput values for H-Store:
  - 220
  - 580
  - 1127
  - 1619
  - 2197
  - 2655
  - 3199
  - 3629
  - 4248
  - 4629
  - 5069

- Throughput values for SLIK TCP:
  - 312
  - 354
  - 396
  - 430
  - 463
  - 496
  - 529
  - 562
  - 595
  - 628
  - 661

- Throughput values for SLIK:
  - 396
  - 430
  - 463
  - 496
  - 529
  - 562
  - 595
  - 628
  - 661
  - 694
  - 727

The diagram illustrates the scalability of lookup throughput for H-Store, SLIK TCP, and SLIK as the number of servers increases.
Scalability: Lookup Latency

![Graph showing scalability and lookup latency for H-Store, SLIK TCP, and SLIK. The graph plots average latency per lookup in microseconds against the number of indexlets. Key values for different indexlet counts include:

- **H-Store**:
  - 1 indexlet: 49.9 µs
  - 2 indexlets: 55.6 µs
  - 3 indexlets: 58.2 µs
  - 4 indexlets: 69.1 µs
  - 5 indexlets: 81.6 µs
  - 6 indexlets: 90.6 µs
  - 7 indexlets: 91.0 µs
  - 8 indexlets: 112.7 µs
  - 9 indexlets: 114.4 µs
  - 10 indexlets: 113.3 µs

- **SLIK TCP**:
  - 1 indexlet: 119.4 µs
  - 2 indexlets: 152.6 µs
  - 3 indexlets: 179.6 µs
  - 4 indexlets: 199.2 µs
  - 5 indexlets: 215.6 µs
  - 6 indexlets: 243.4 µs
  - 7 indexlets: 240.3 µs
  - 8 indexlets: 267.4 µs
  - 9 indexlets: 269.8 µs
  - 10 indexlets: 267.2 µs

- **SLIK**:
  - 1 indexlet: 13.1 µs
  - 2 indexlets: 13.3 µs
  - 3 indexlets: 13.9 µs
  - 4 indexlets: 13.7 µs
  - 5 indexlets: 14.4 µs
  - 6 indexlets: 14.7 µs
  - 7 indexlets: 14.5 µs
  - 8 indexlets: 14.4 µs
  - 9 indexlets: 14.6 µs
  - 10 indexlets: 14.7 µs

The graph illustrates that H-Store and SLIK TCP have higher latency compared to SLIK as the number of indexlets increases, with SLIK TCP showing the highest latency among the three. SLIK maintains a consistently lower latency across all indexlet counts.](image-url)
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**

**Consistency Level**

**Read / write latency (approx)**

- Better
Current Web Scale Datastores

Consistency Level

Eventual

Causal, SI, “Define your own”

Strong

Read / write latency (approx)

Better
Current Web Scale Datastores

Read / write latency (approx)

Consistency Level

Better

Strong

Causal, SI, “Define your own”

Eventual

Better

MongoDB

Spanner

H-Store

H-Base

MegaStore

Cassandra

Espresso

PNUTS

CouchDB

Tao

SLIK
Current Web Scale Datastores

Consistency Level

Strong

Causal, SI, “Define your own”

Eventual

Read / write latency (approx)

Better
Talk Outline

- Motivation
- Design
- Performance
- Related Work
- Summary
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

By using ordered writes and treating indexes as hints

Lookups and range queries on secondary keys

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: http://stanford.edu/~ankitak

I can be reached at: ankitak@cs.stanford.edu