Scalable Low-Latency Indexes for a Key-Value Store

Ankita Kejriwal

With Arjun Gopalan, Ashish Gupta, Zhihao Jia, Stephen Yang

and John Ousterhout

PLATFORMLAB

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

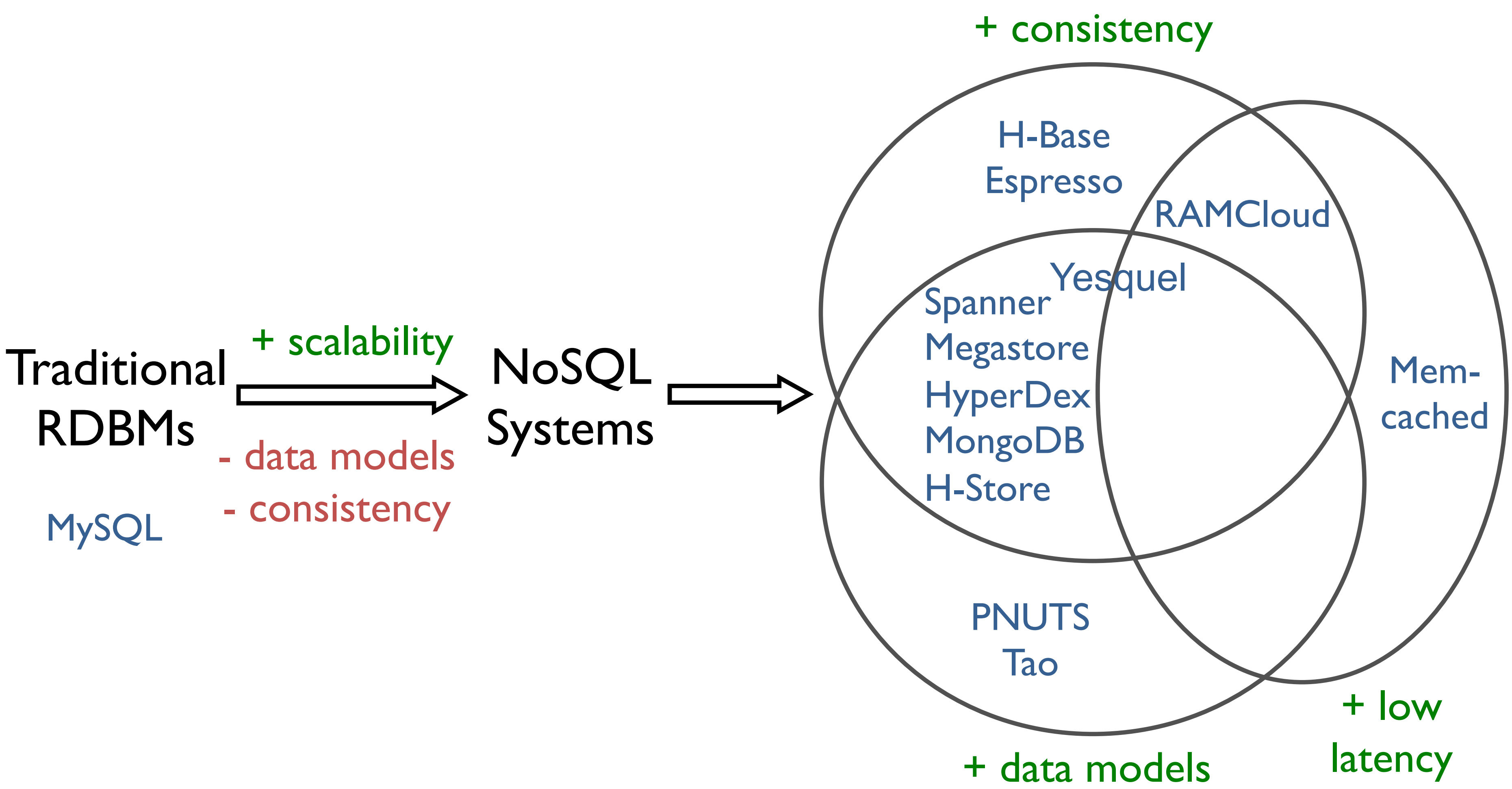
Traditional RDBMs → NoSQL Systems

+ scalability

- data models
- consistency

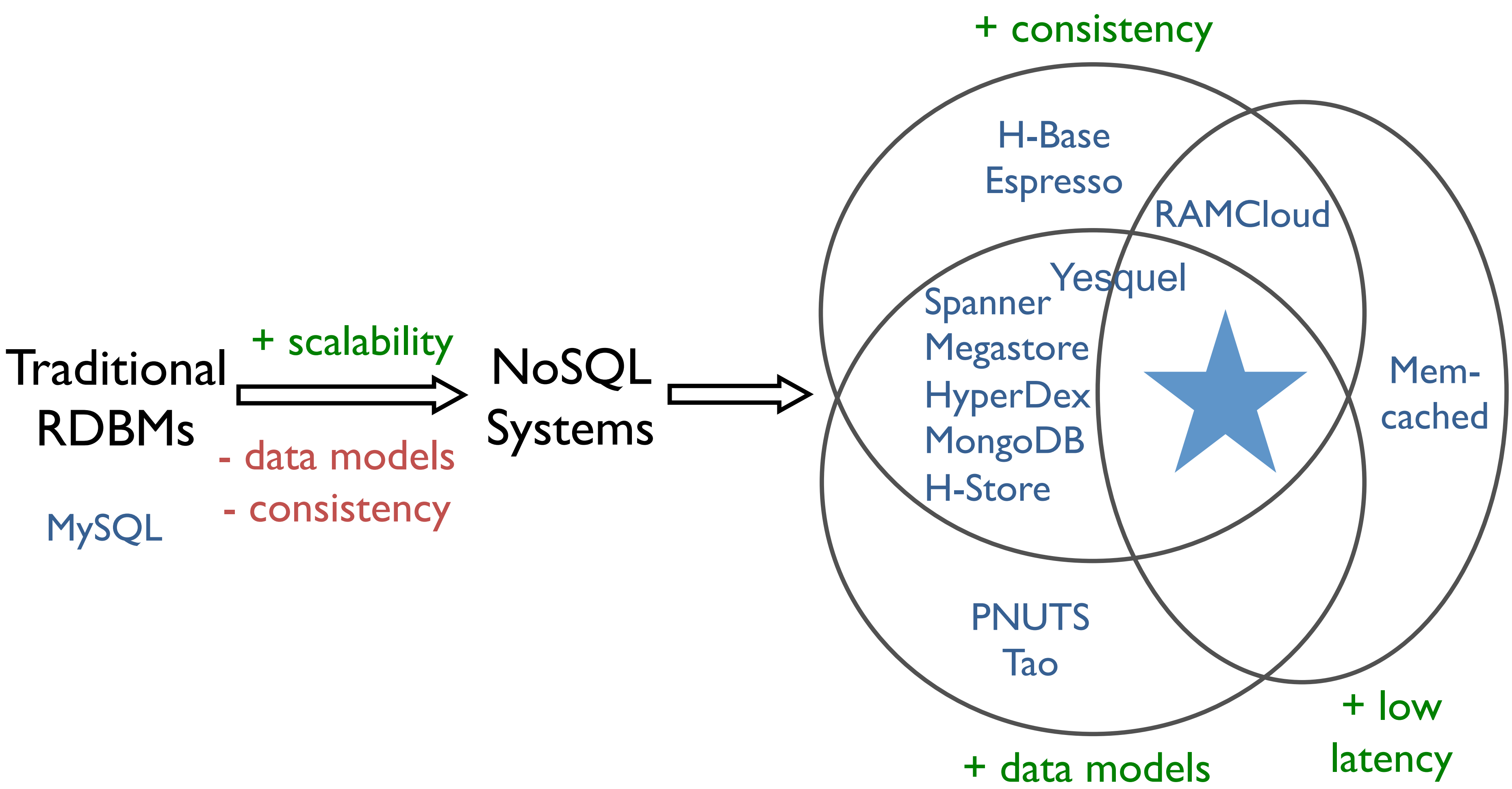
MySQL

Motivation



Traditional RDBMs
MySQL
+ scalability
- data models
- consistency
NoSQL Systems
+ consistency
H-Base
Espresso
RAMCloud
Yesquel
Spanner
Megastore
HyperDex
MongoDB
H-Store
Mem-cached
PNUTS
Tao
+ low latency
+ data models

Motivation



Traditional RDBMs
MySQL
+ scalability
- data models
- consistency
NoSQL Systems
+ consistency
H-Base
Espresso
RAMCloud
Yesquel
Spanner
Megastore
HyperDex
MongoDB
H-Store
Mem-cached
PNUTS
Tao
+ low latency
+ data models

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

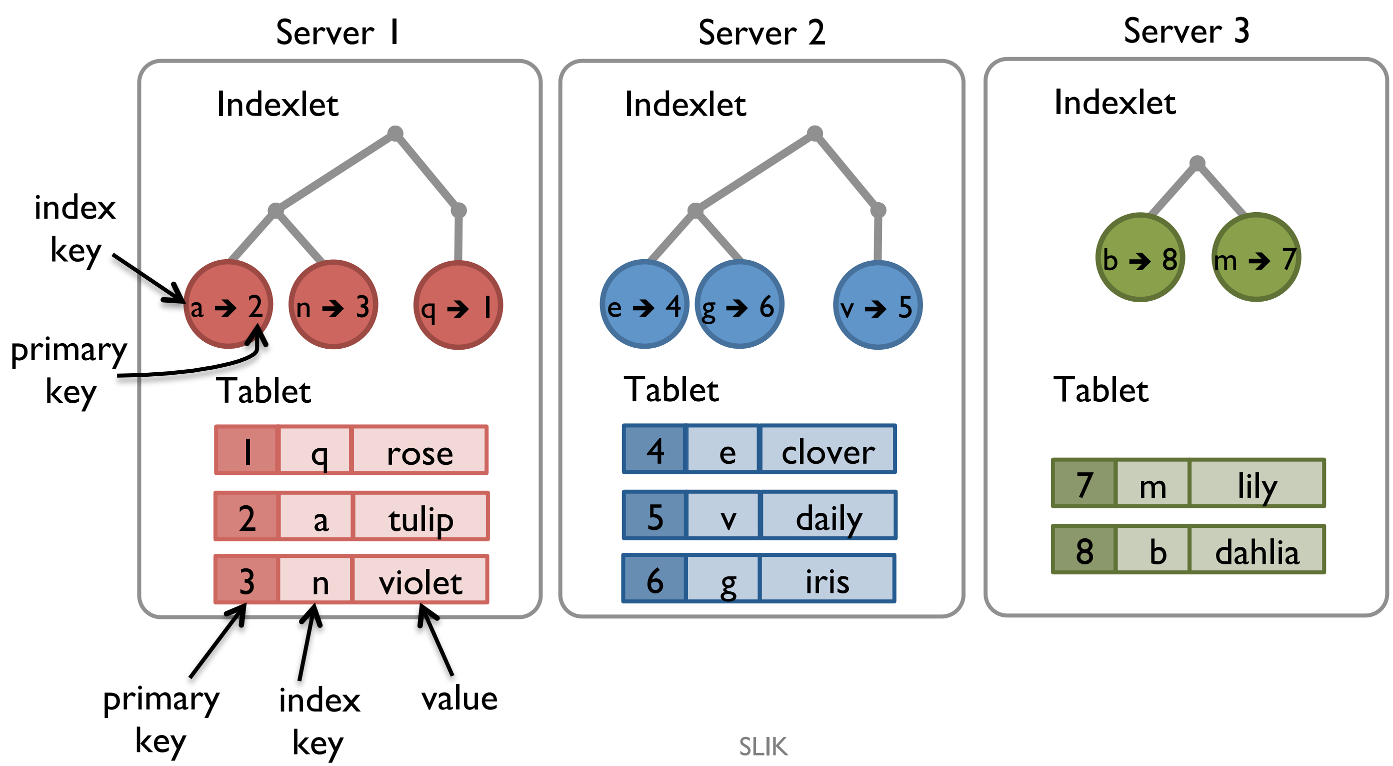
- Data model
- **Scalability**
- **Strong consistency**
- Storage
- Durability
- Availability

Scalability

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

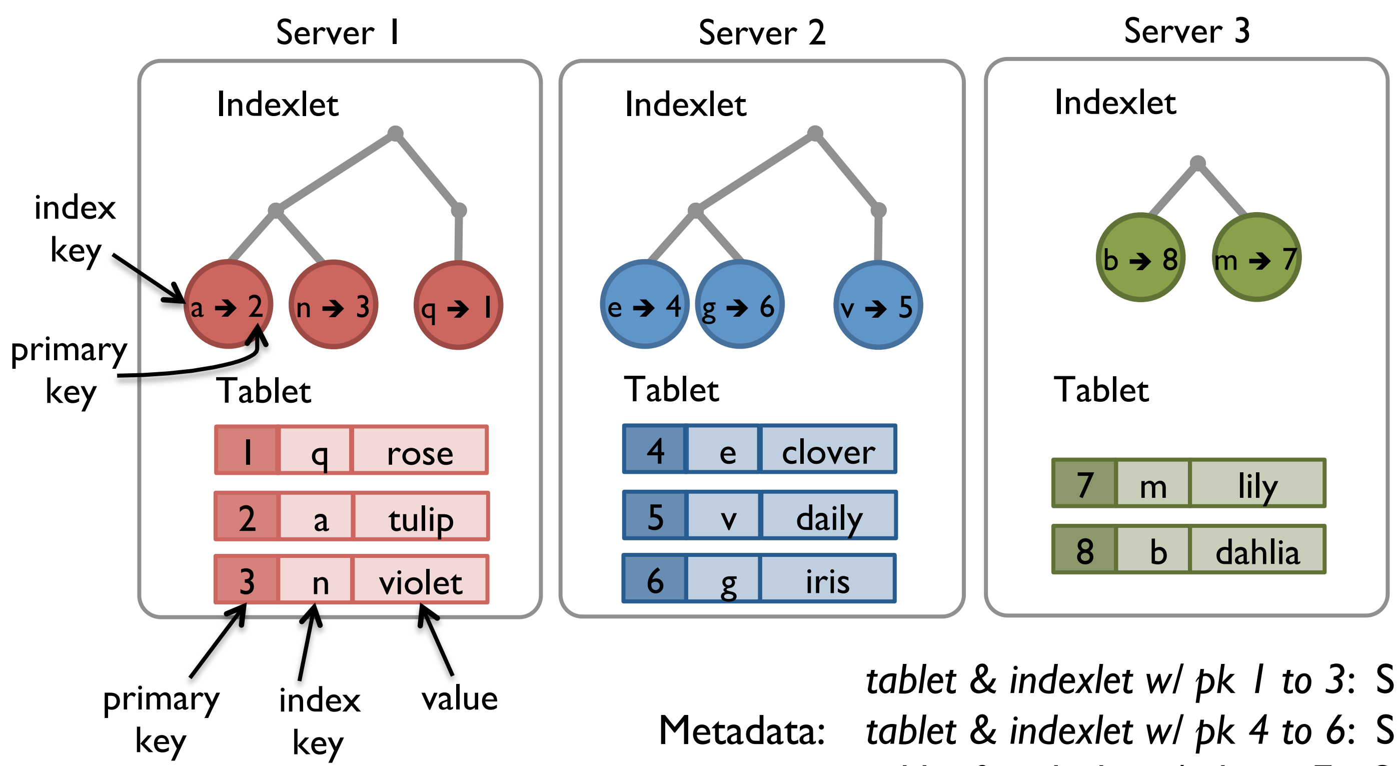
Index Partitioning: Colocation

- Colocate index entries and objects
- One of the keys used to partition the objects and indexes



Index Partitioning: Colocation

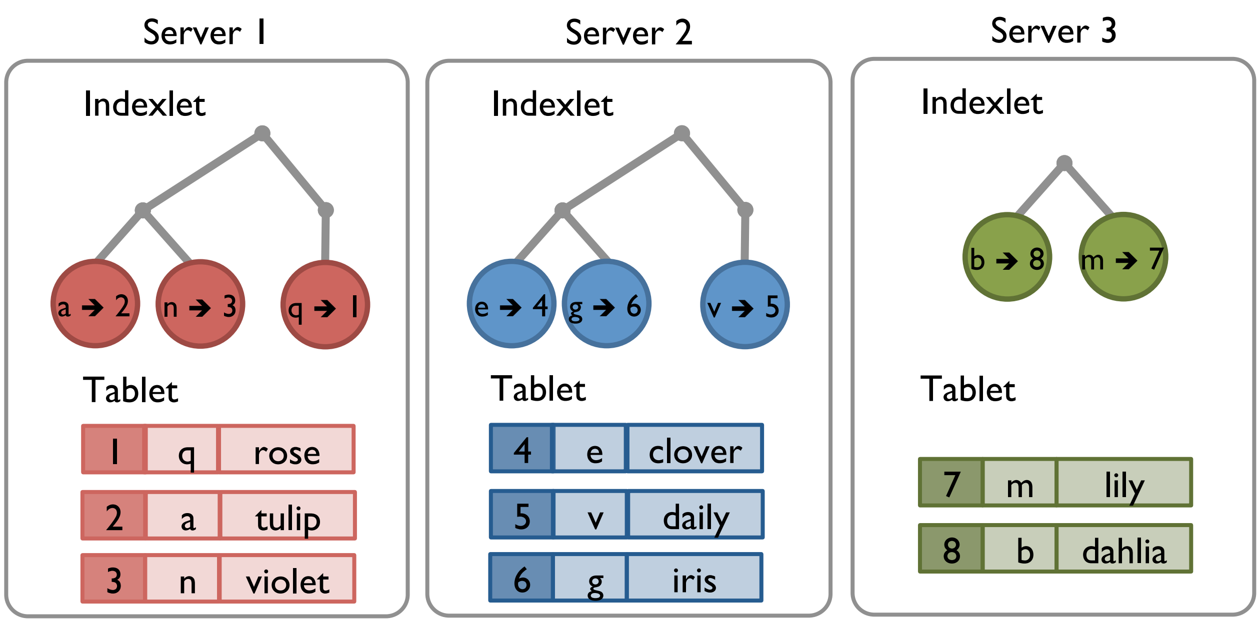
- Colocate index entries and objects
- One of the keys used to partition the objects and indexes
- No association between index partitions and index key ranges



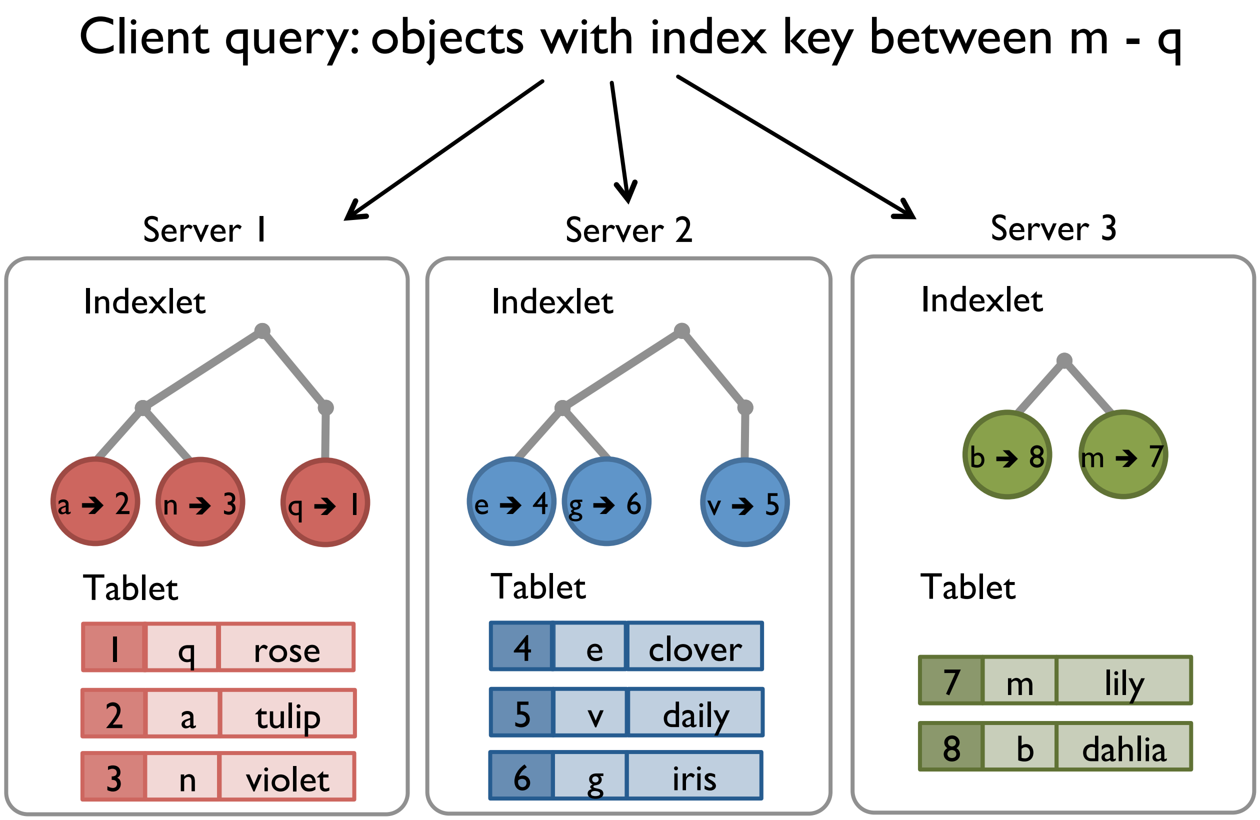
Metadata:
tablet & indexlet w/ pk 1 to 3: S 1
tablet & indexlet w/ pk 4 to 6: S 2
tablet & indexlet w/ pk >= 7: S 3

Index Partitioning: Colocation

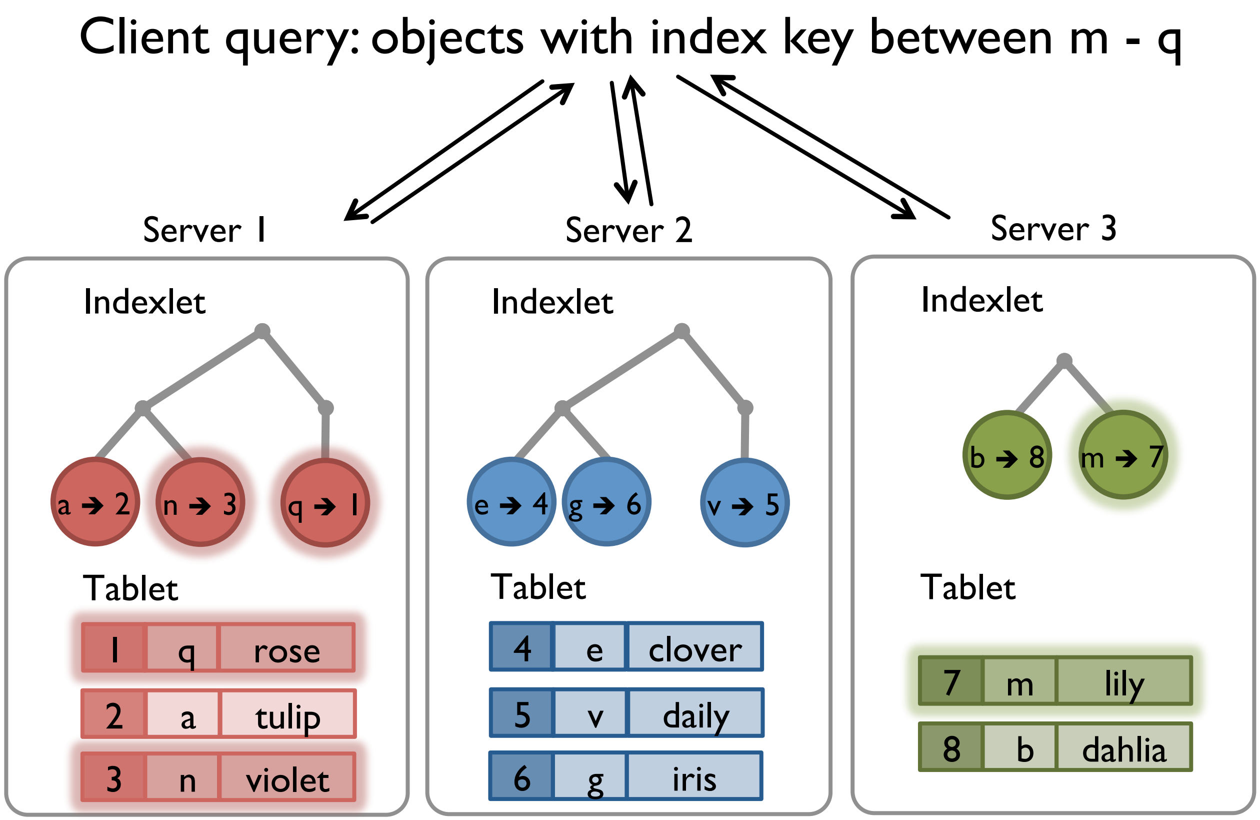
Client query: objects with index key between m - q



Index Partitioning: Colocation

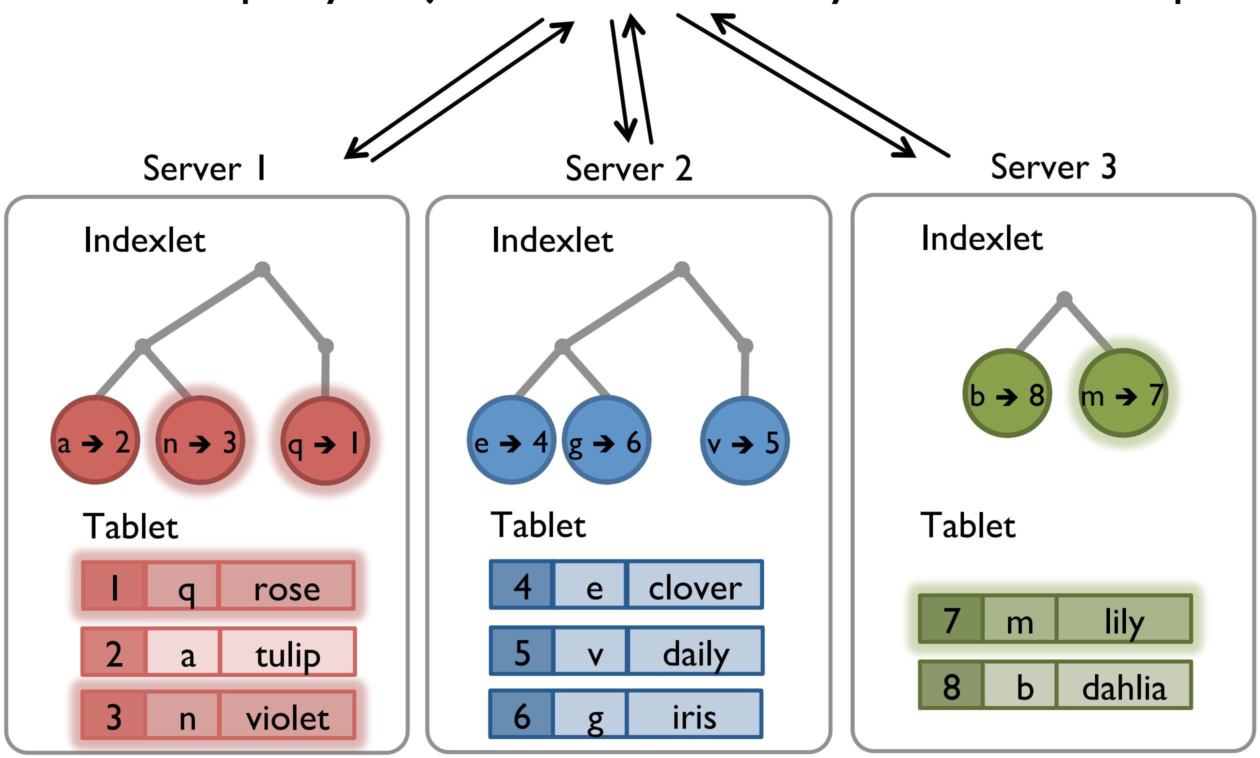


Index Partitioning: Colocation



Index Partitioning: Colocation

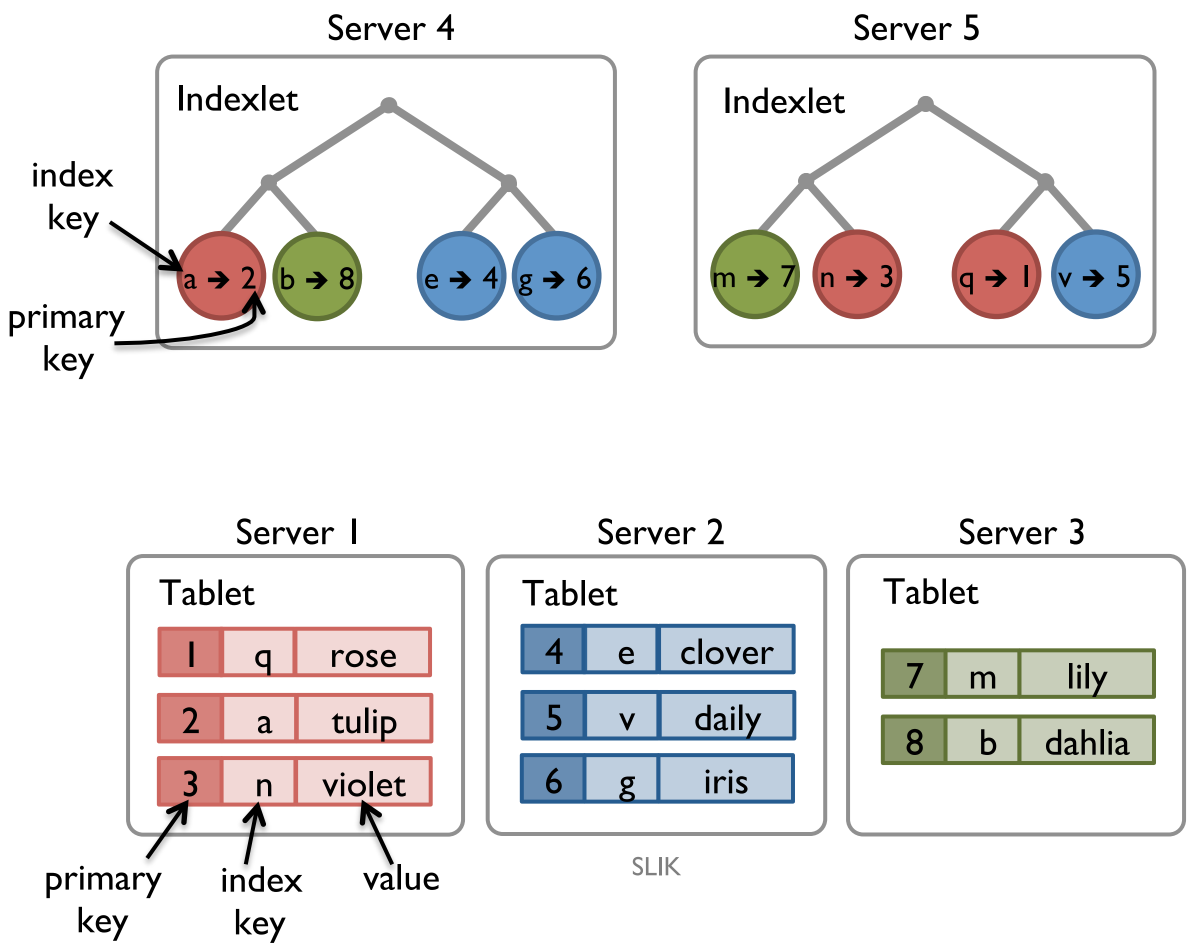
Client query: objects with index key between m - q



Not Scalable!

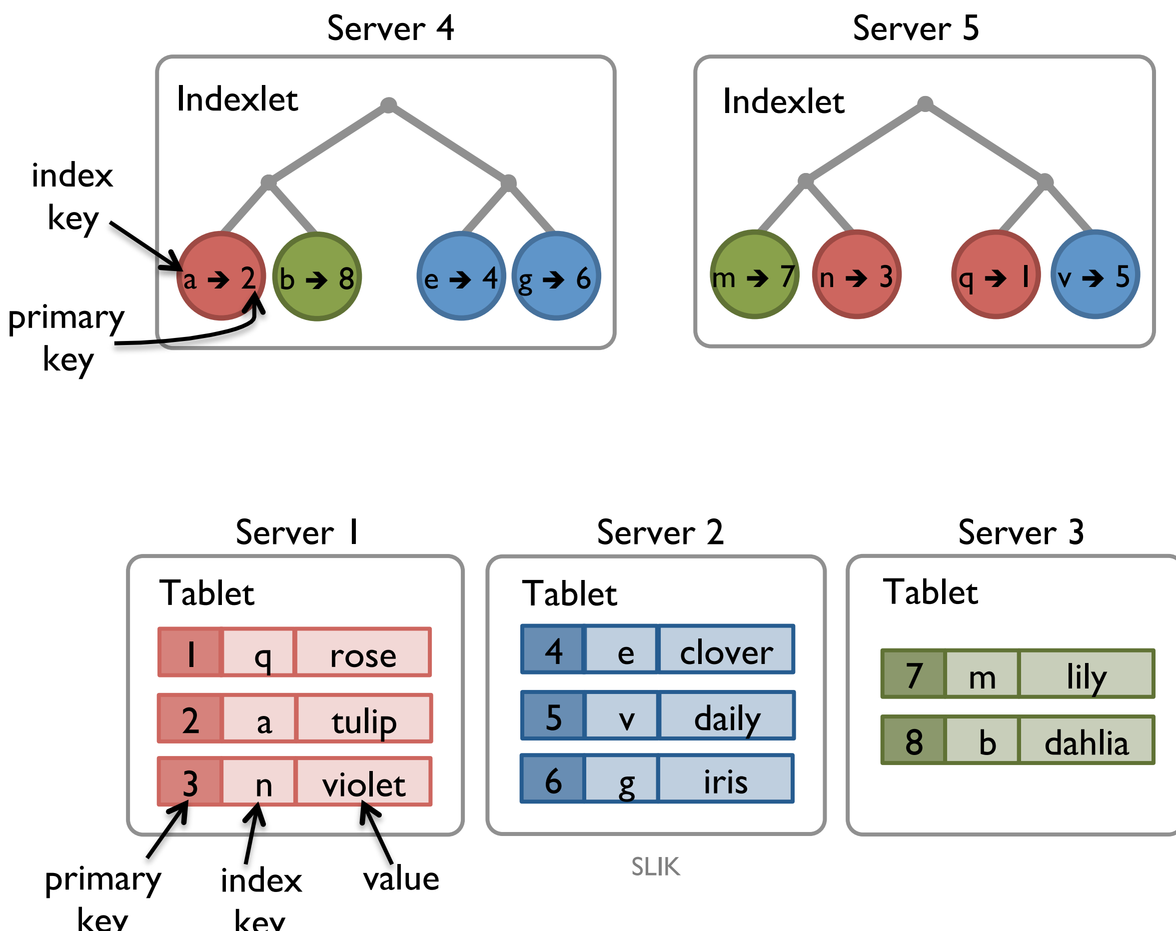
Index Partitioning: Independent

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



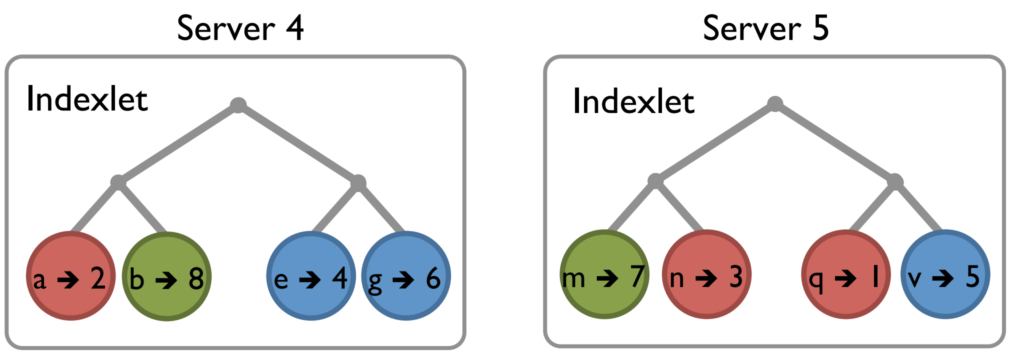
Index Partitioning: Independent

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

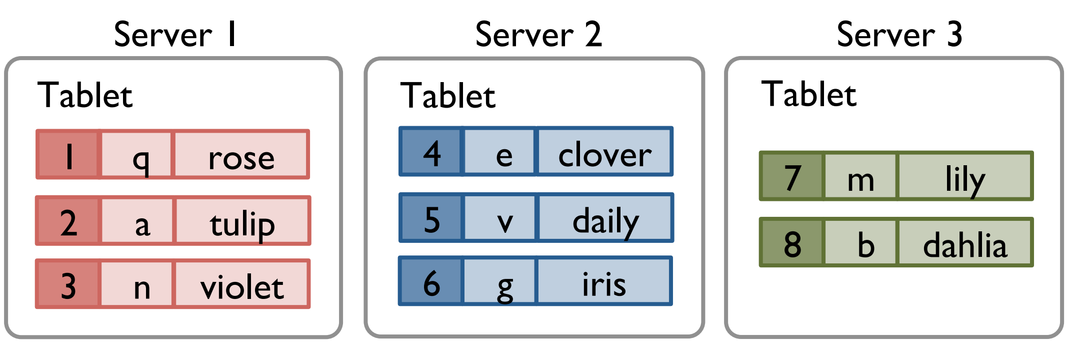


Metadata:
tablet w/ pk 1 to 3: S 1
tablet w/ pk 4 to 6: S 2
tablet w/ pk >= 7: S 3
indexlet w/ sk a to g: S 4
indexlet w/ sk >= h: S 5

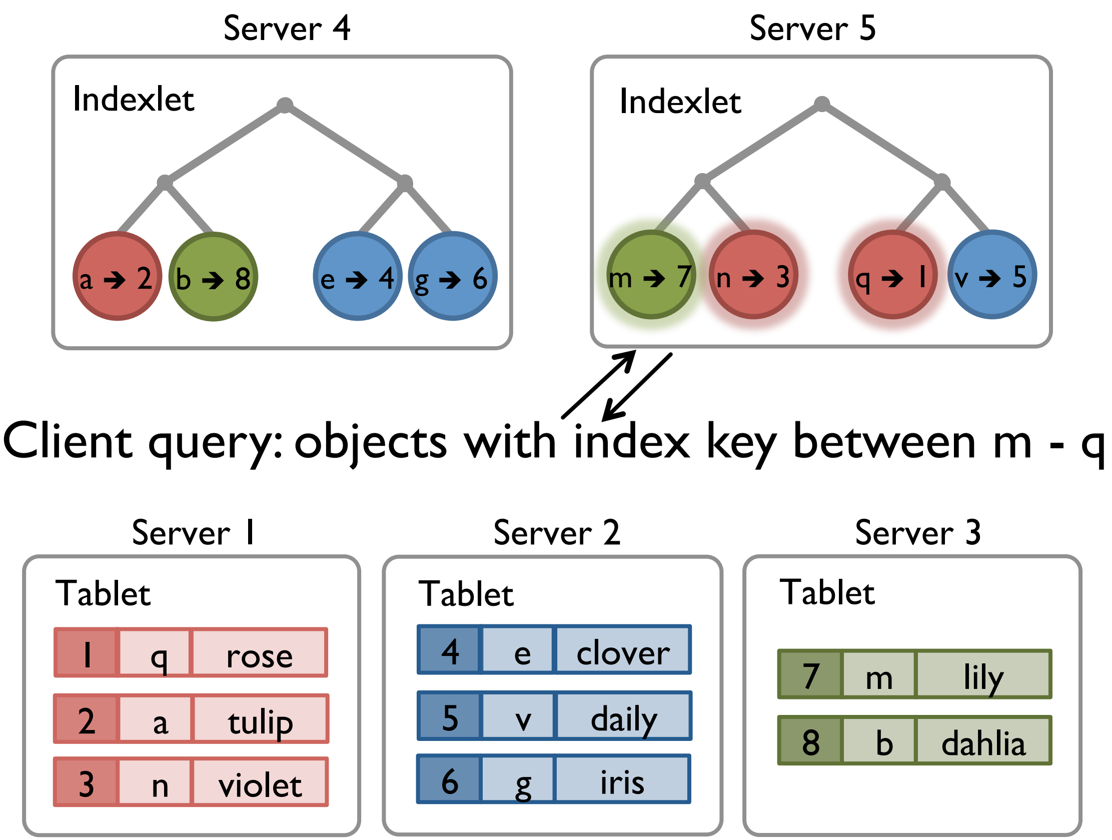
Index Partitioning: Independent



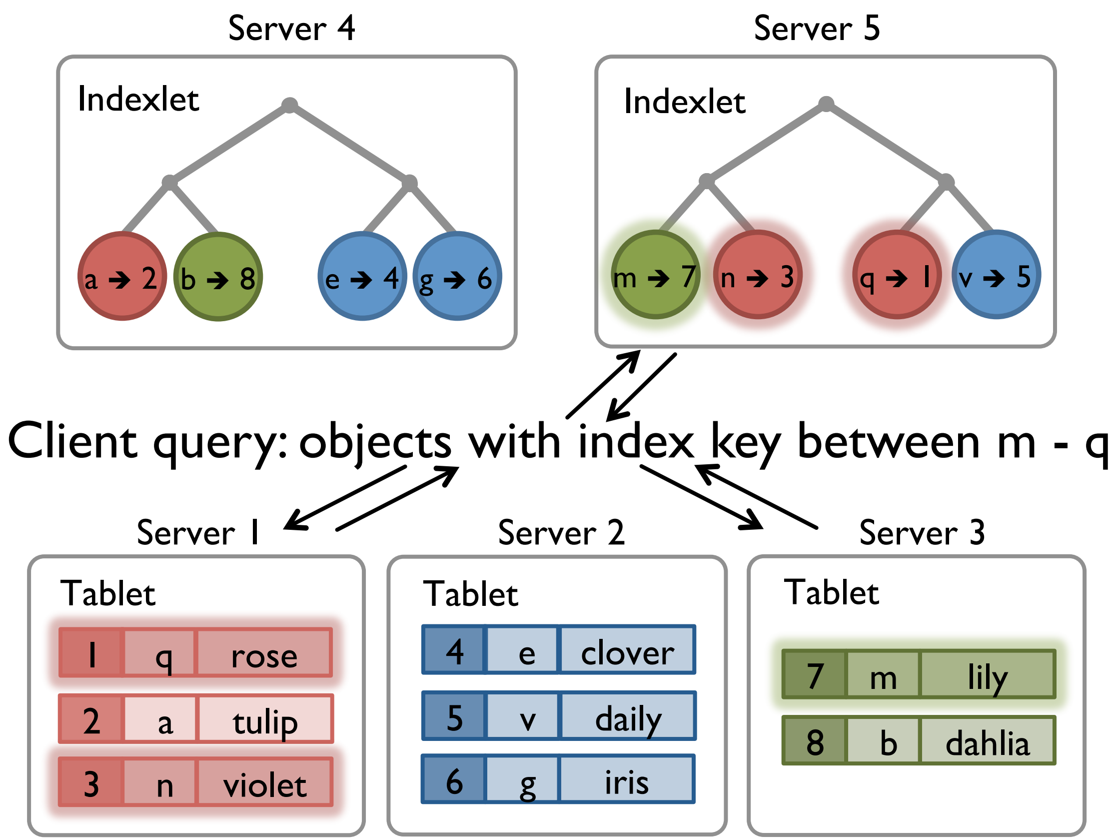
Client query: objects with index key between m - q



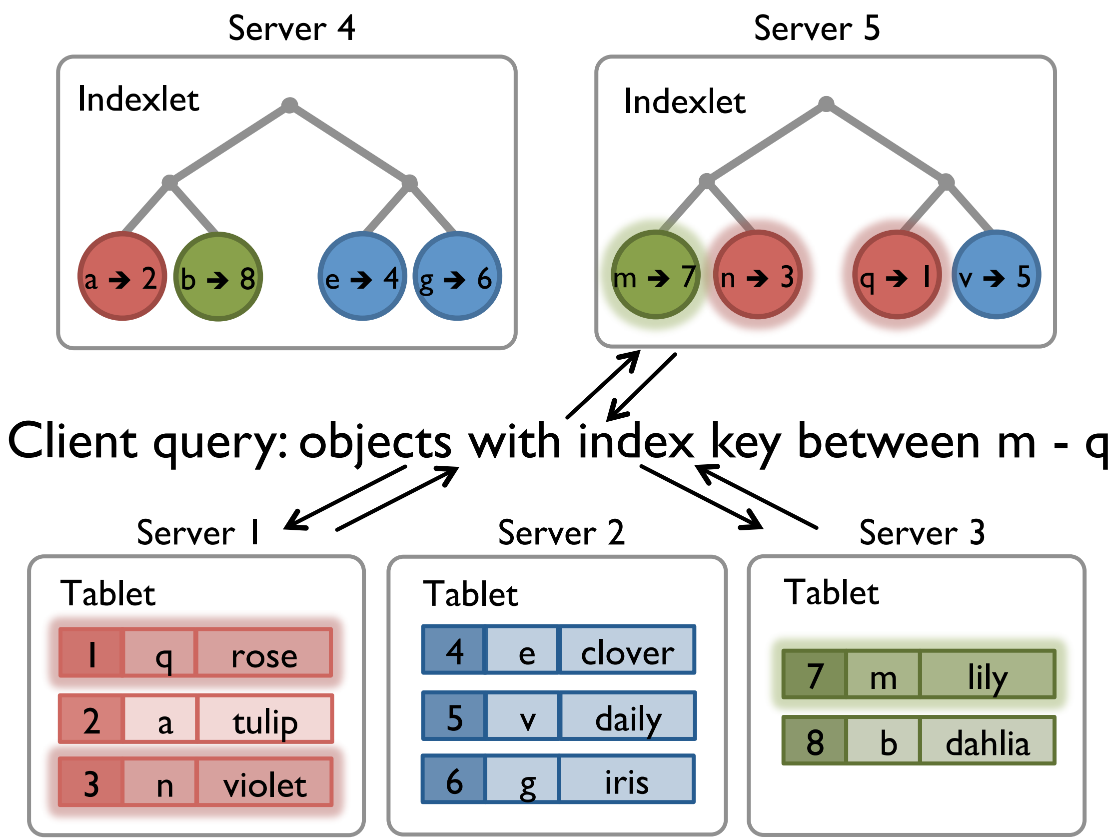
Index Partitioning: Independent



Index Partitioning: Independent



Index Partitioning: Independent



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

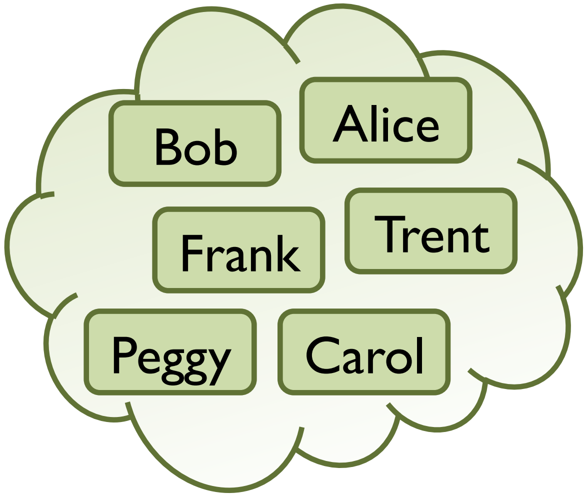
- Data model
- **Scalability**
- **Strong consistency**
- Storage
- Durability
- Availability

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

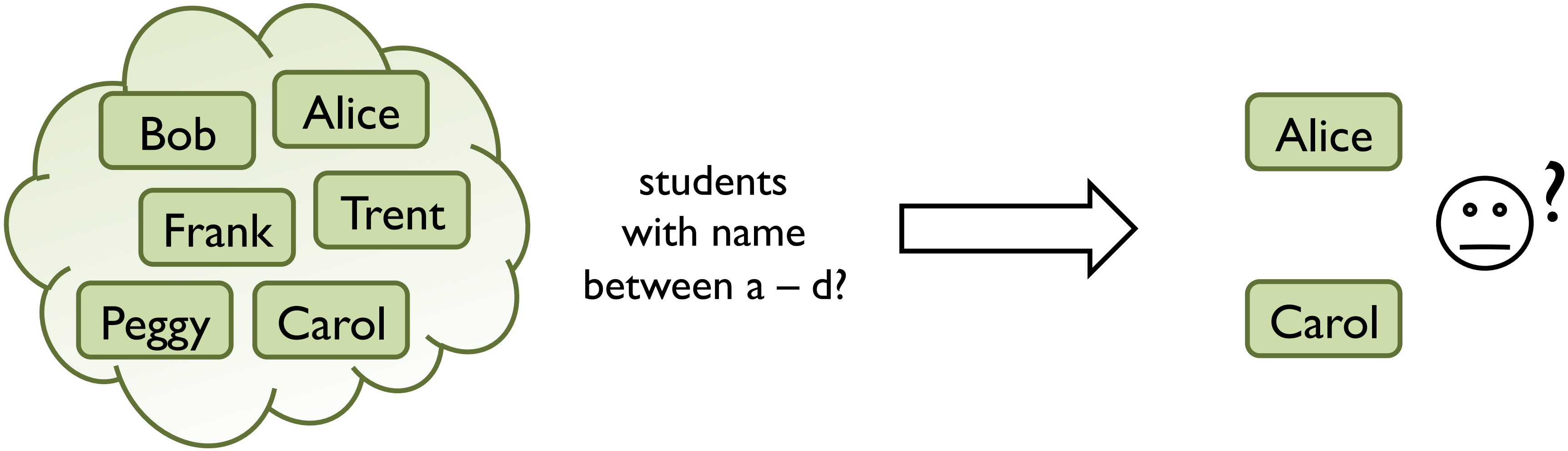
Consistency Properties

- 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

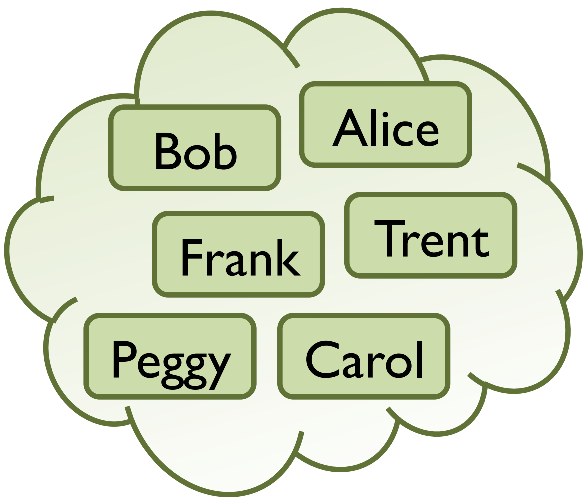


Consistency Properties

- 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

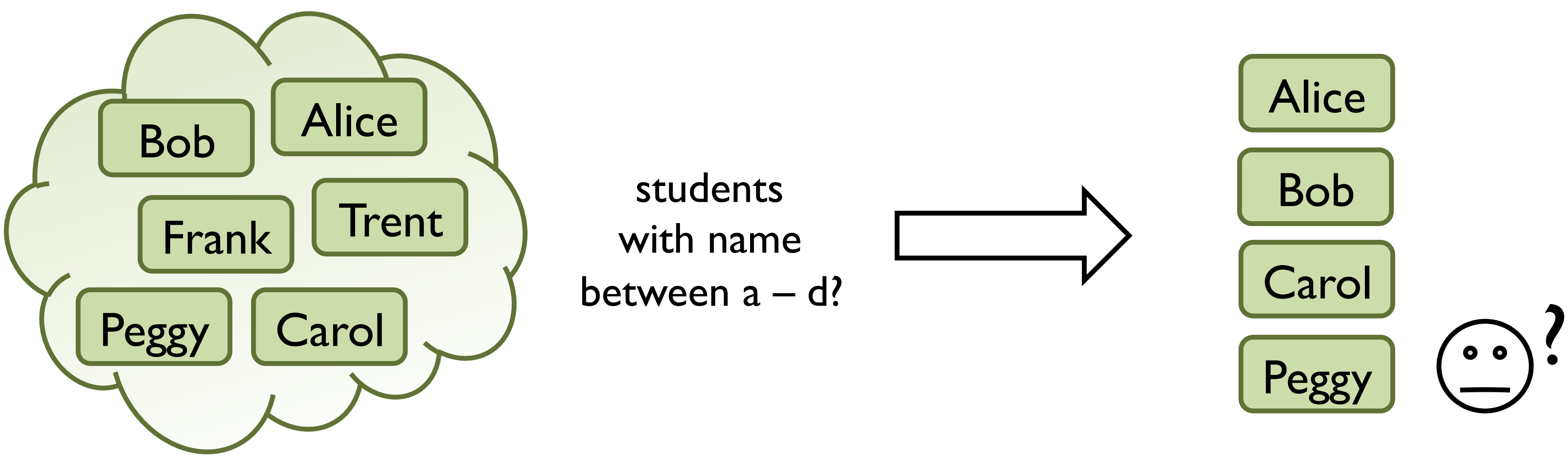
Consistency Properties

- 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



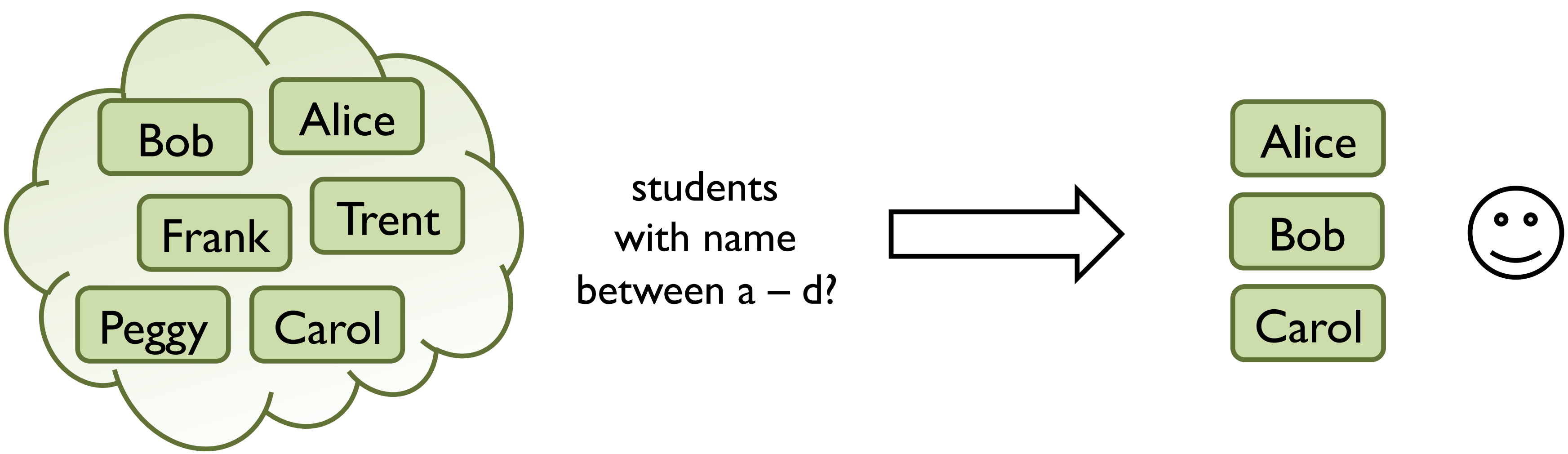
Consistency Properties

- 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



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

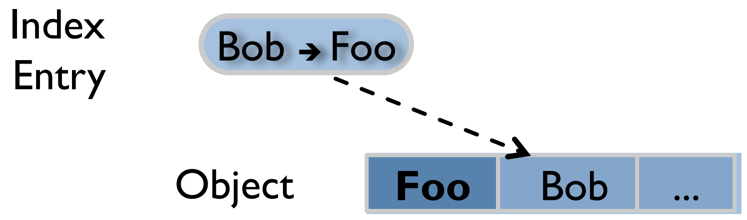


Consistency

- **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

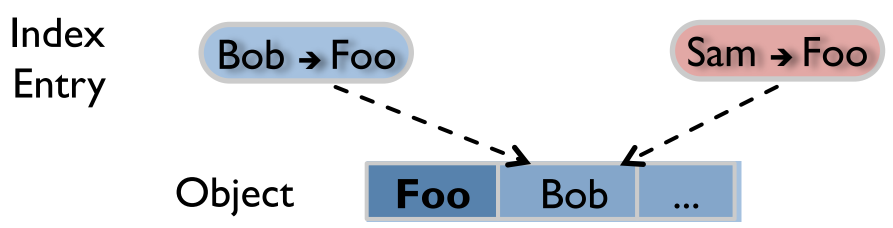
Consistency

- **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

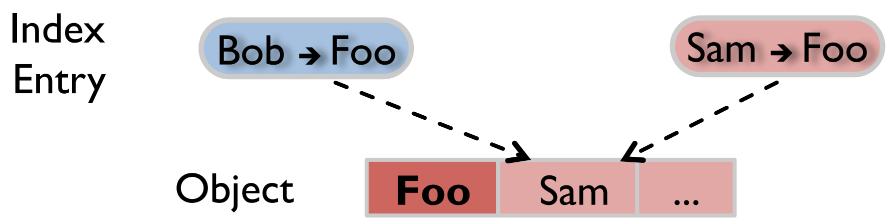
- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range
- **Solution:**
  - Longer index lifespan (via ordered writes)
  - Object data is ground truth and index entries serve as hints



1. Add new index entry

Consistency

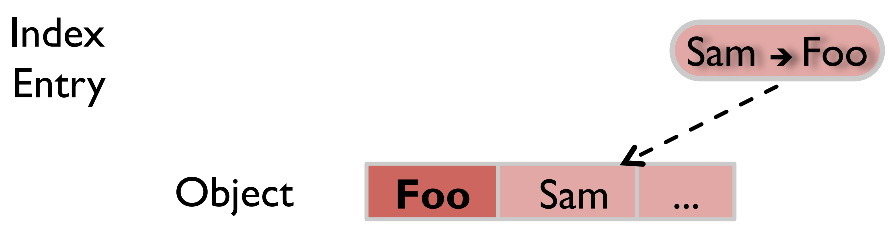
- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range
- **Solution:**
  - Longer index lifespan (via ordered writes)
  - Object data is ground truth and index entries serve as hints



1. Add new index entry
2. Modify object

Consistency

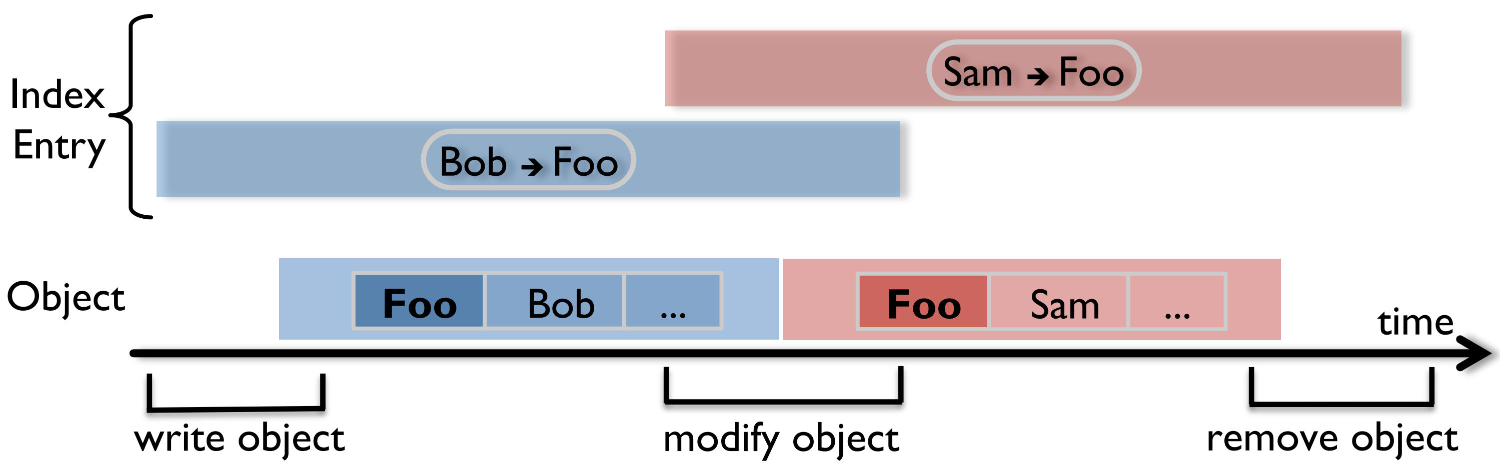
- **Consistency properties:**
  - If an object contains a given secondary key, then an index lookup with that key will return the object
  - If an object is returned by index lookup, then this object contains a secondary key for that index within the specified range
- **Solution:**
  - Longer index lifespan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

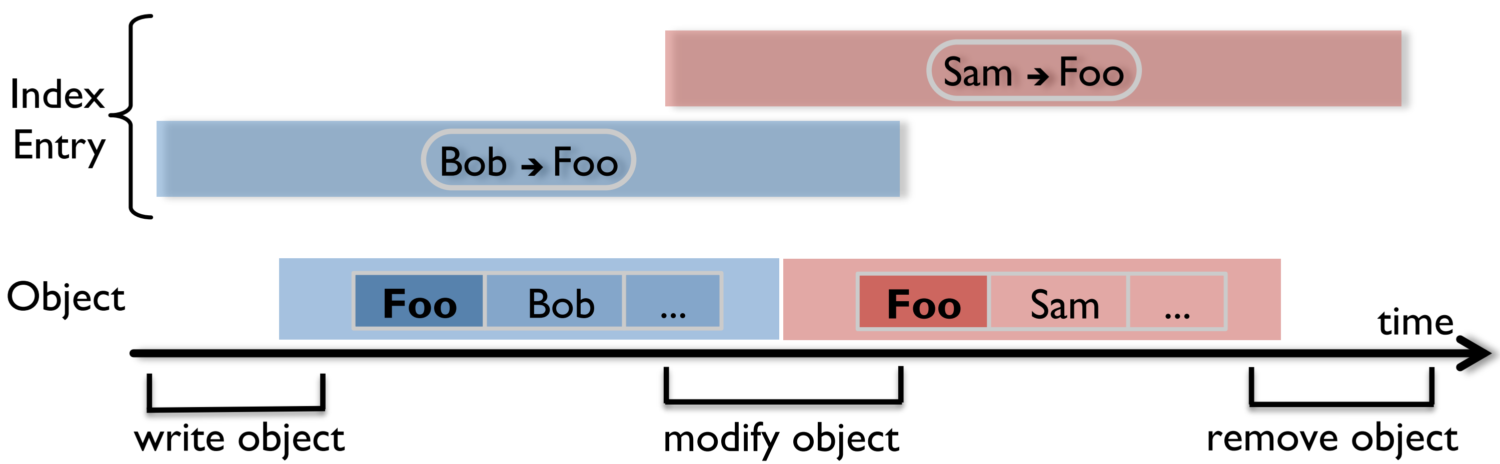
Consistency

- **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



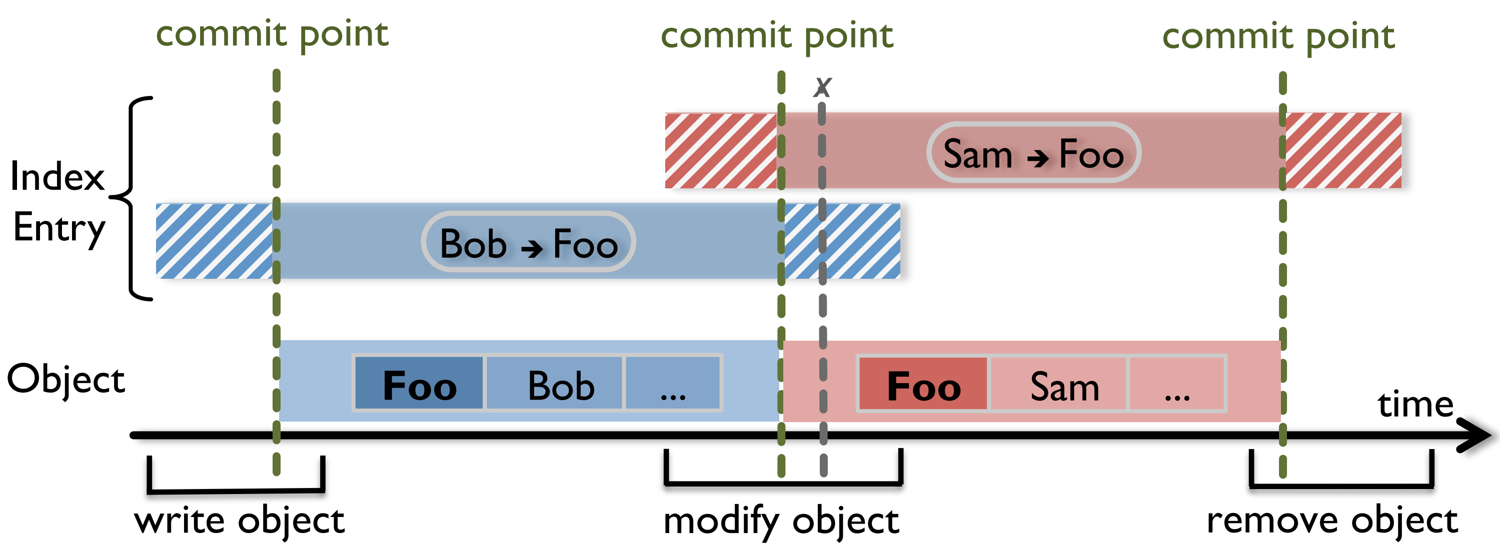
Consistency

- **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



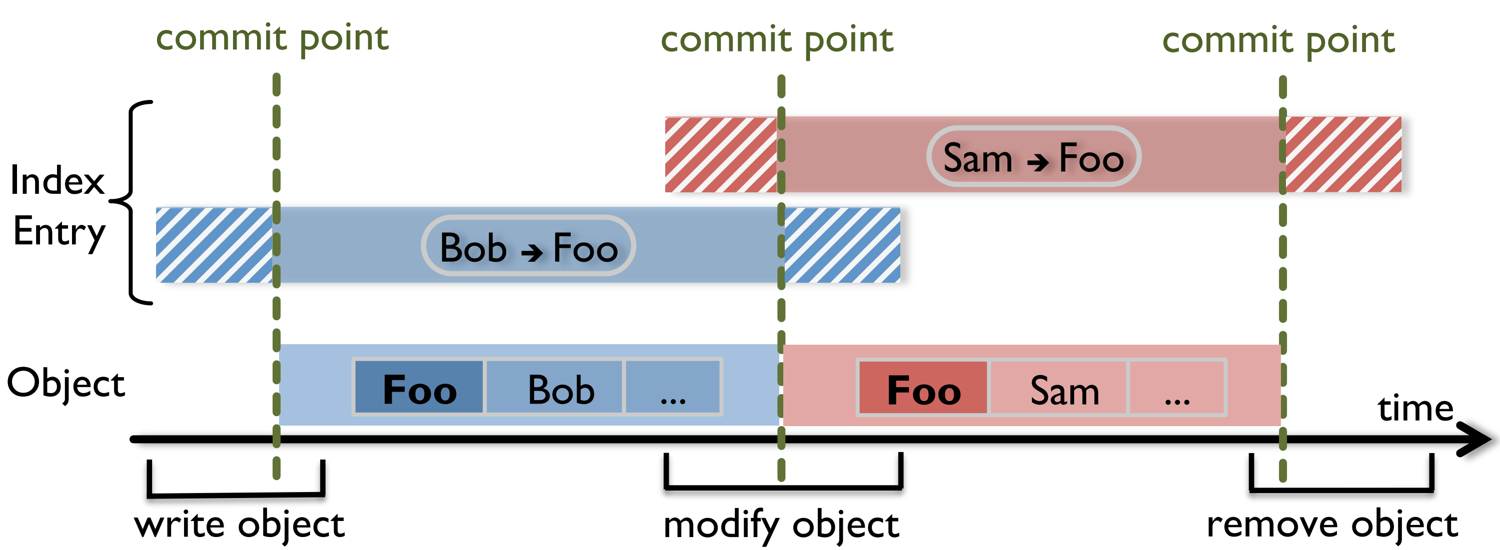
Consistency

- **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



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



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

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)

Latency

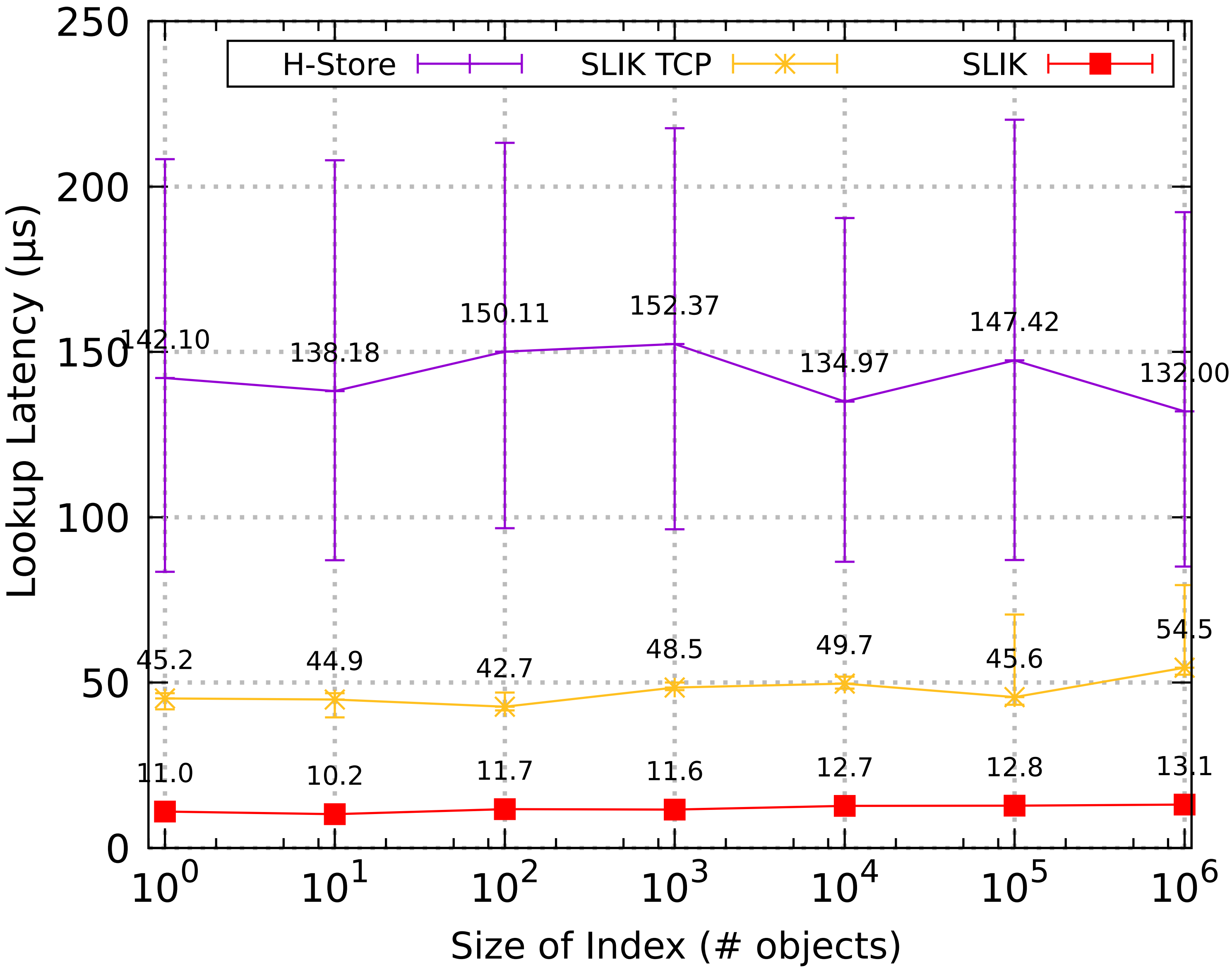
Experiments:

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

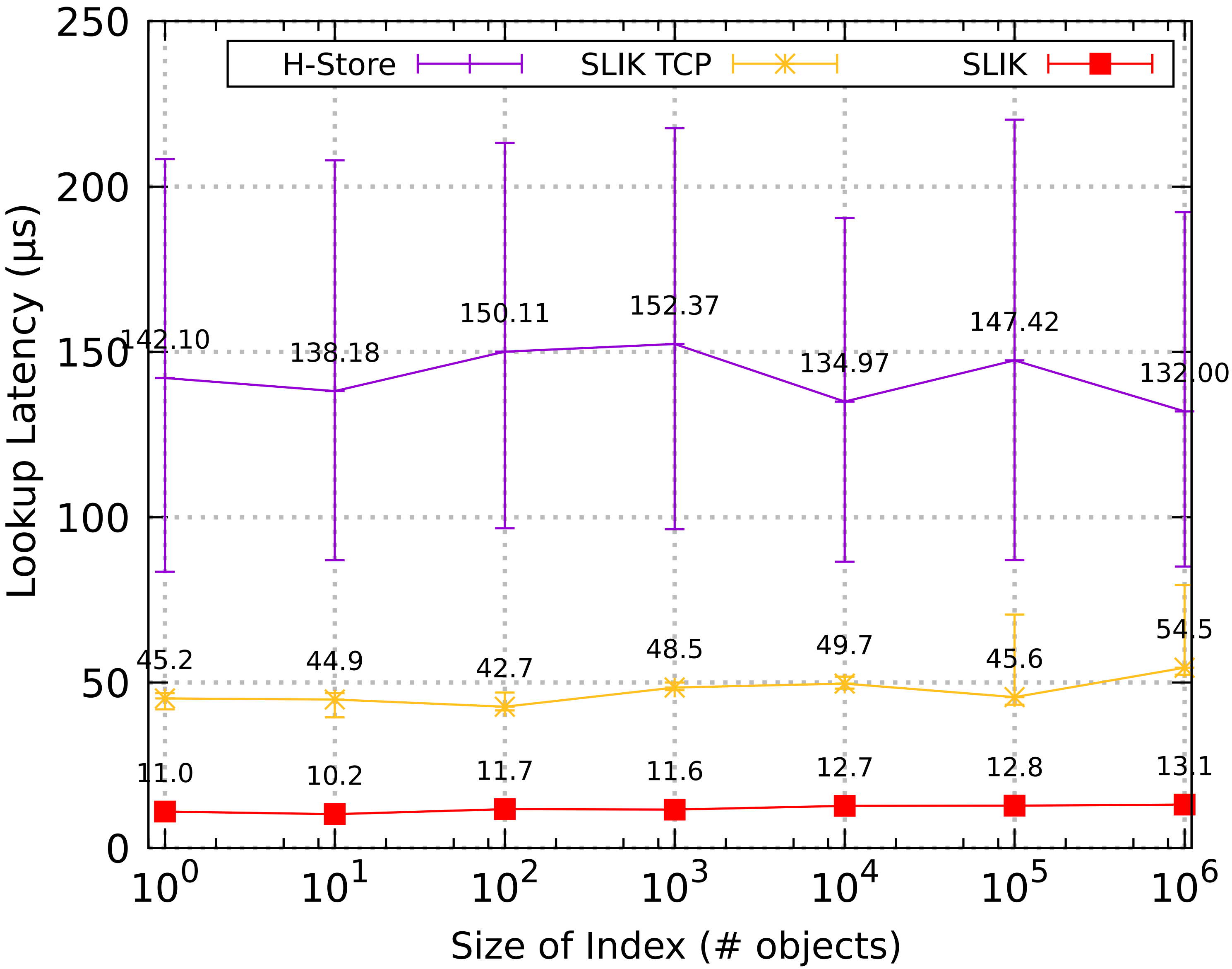
Configuration:

- 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

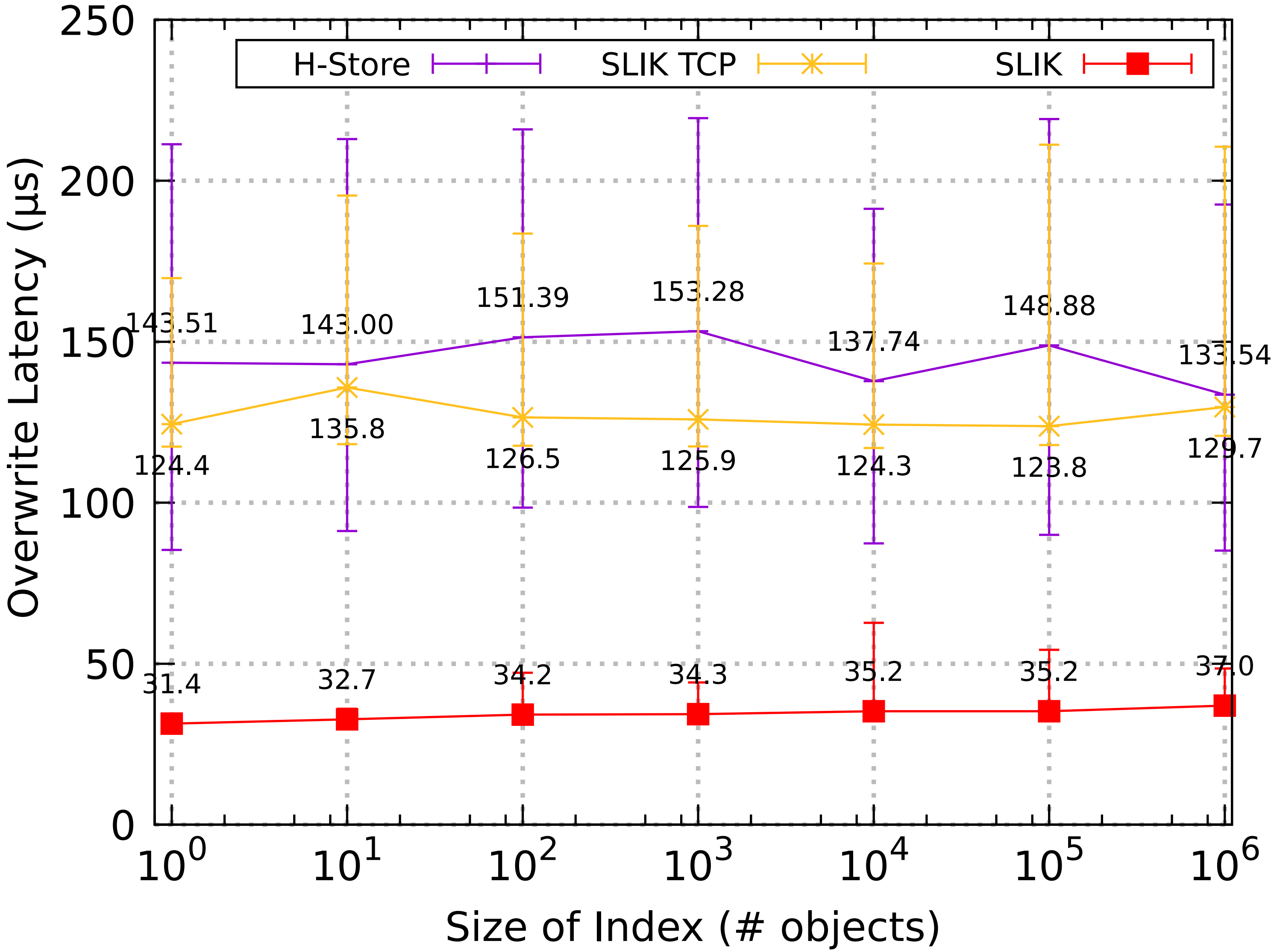
Lookup Latency



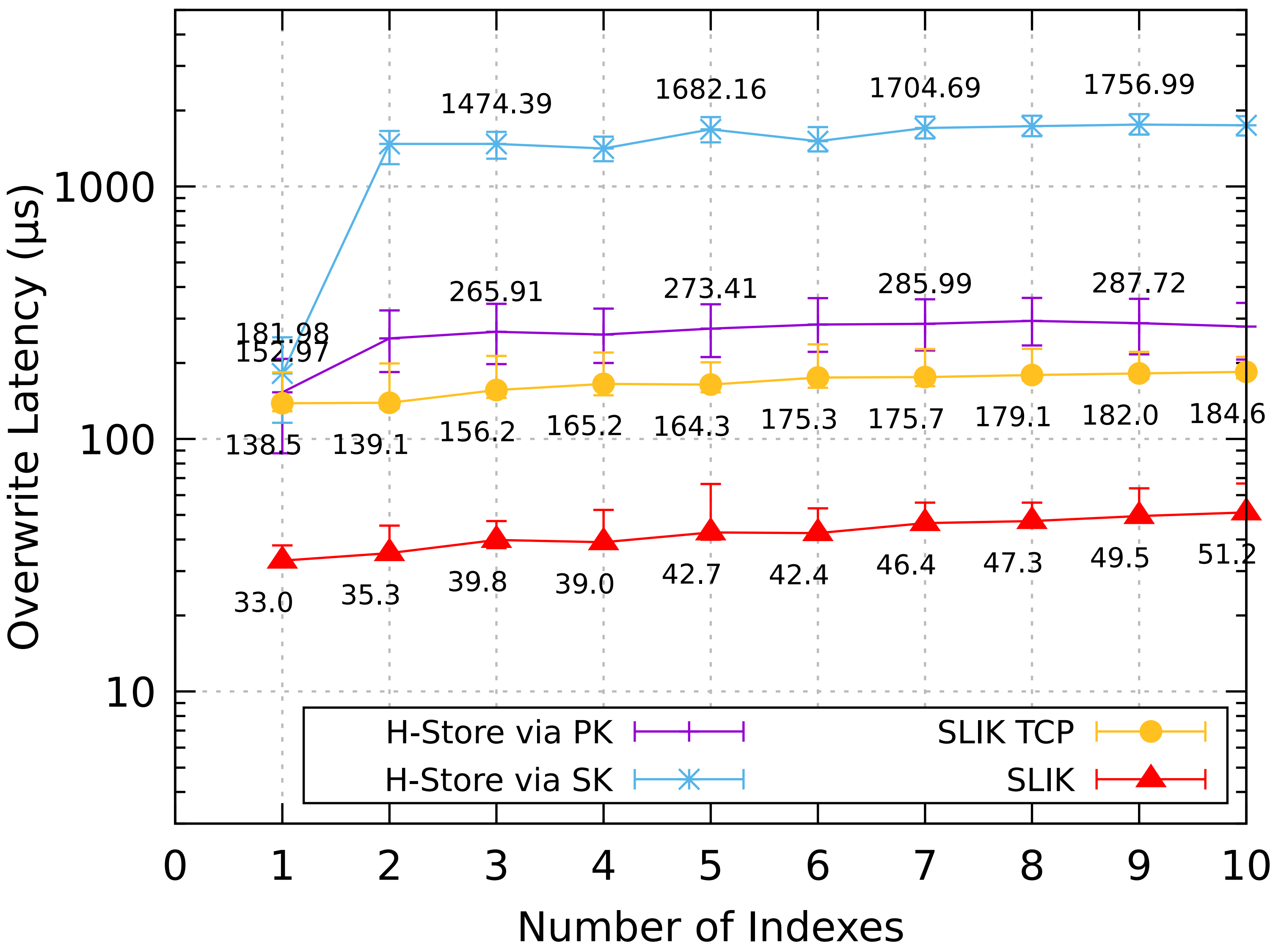
Lookup Latency



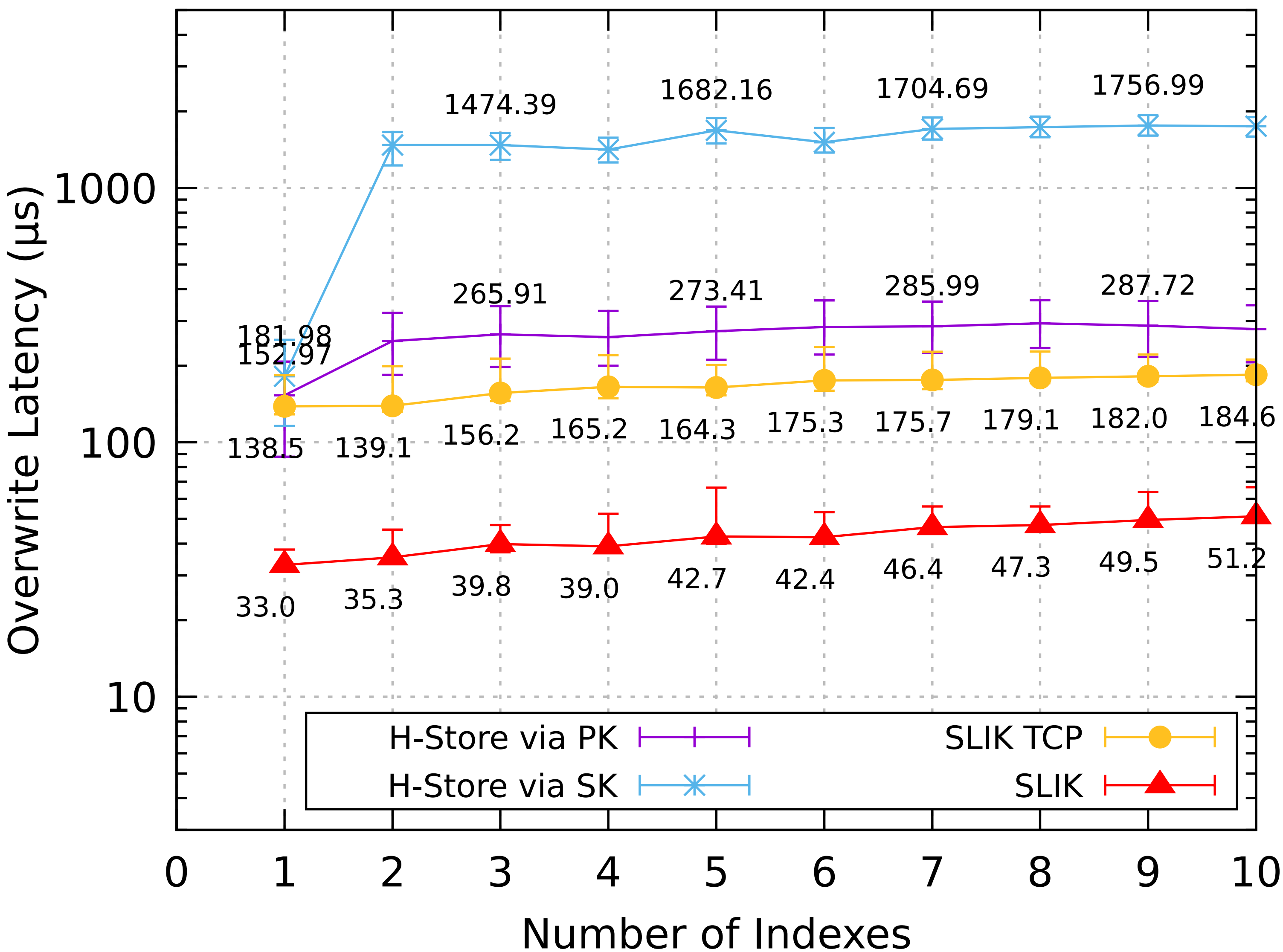
Overwrite Latency



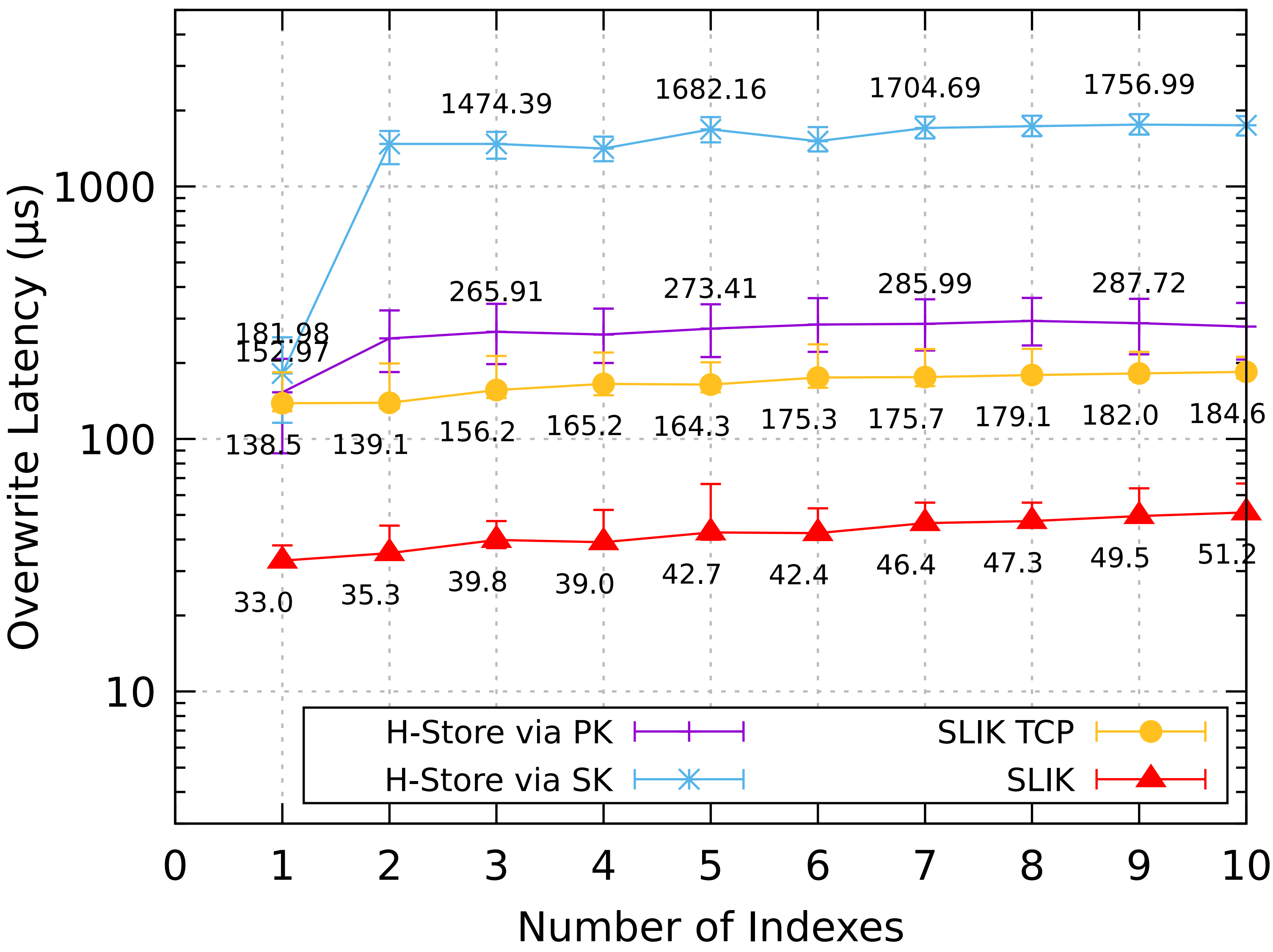
Multiple Secondary Indexes



Multiple Secondary Indexes



Multiple Secondary Indexes



Scalability

Compare: (a) partitioning approaches (b) systems

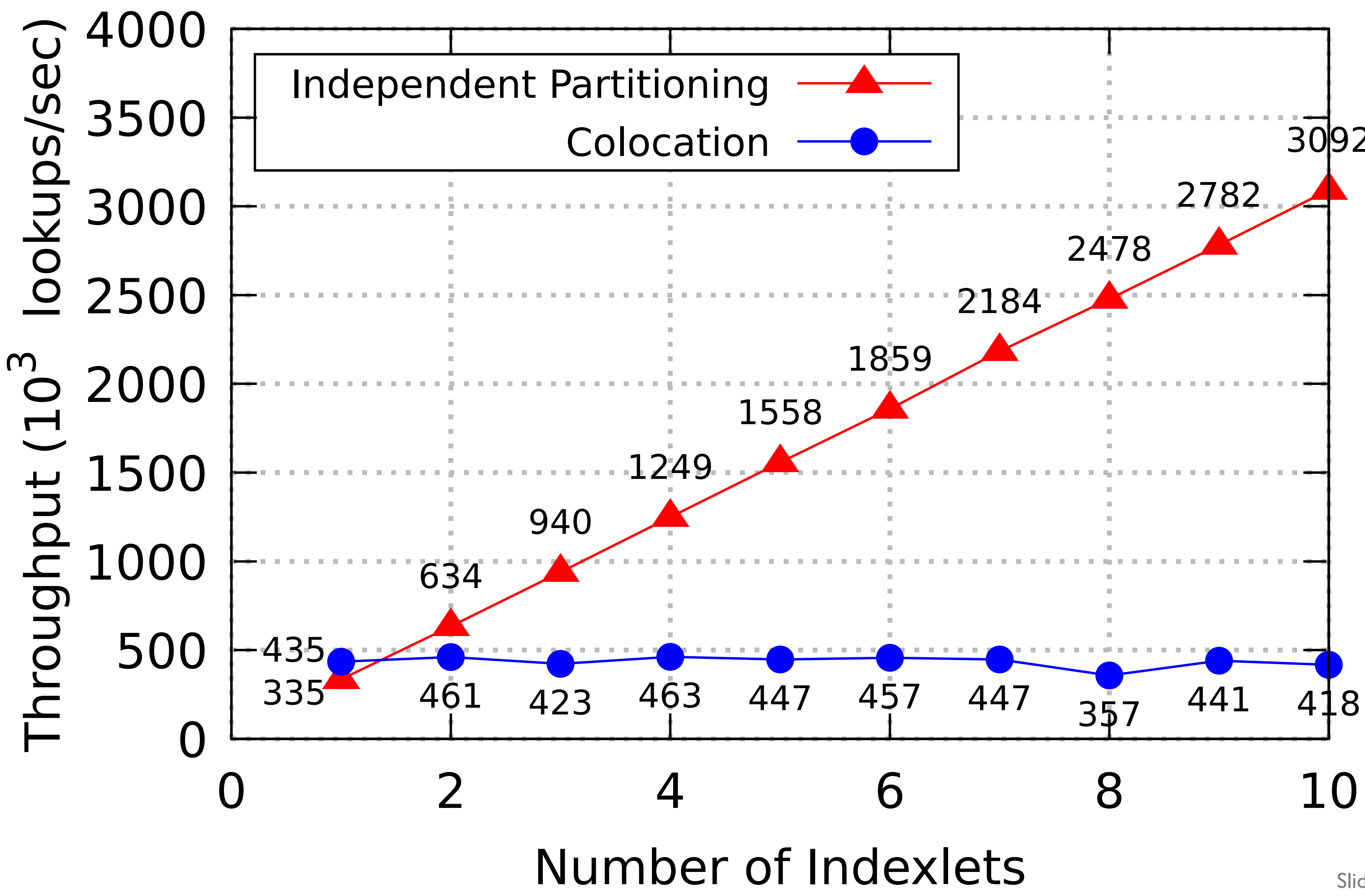
Experiments:

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

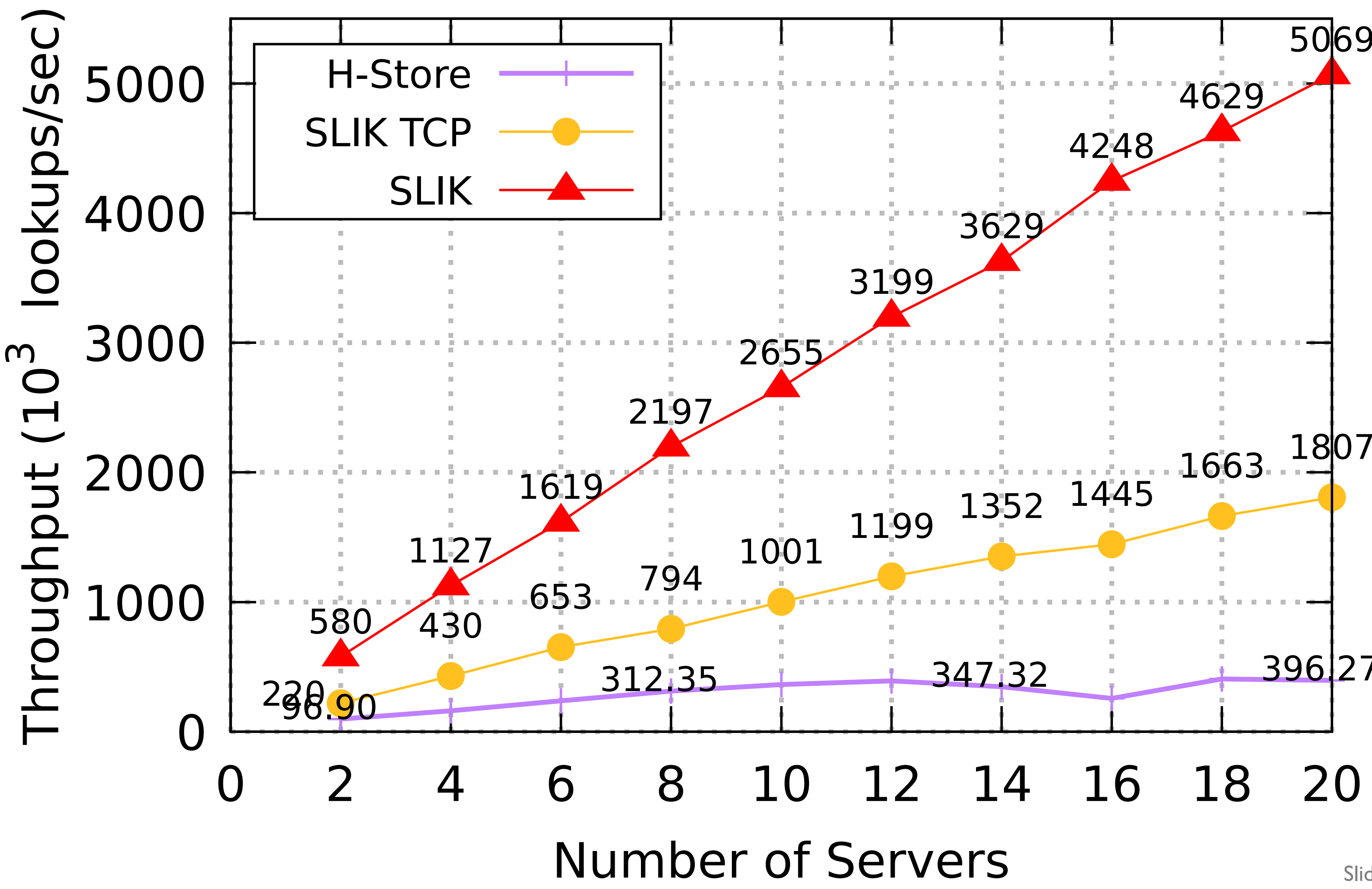
Configuration:

- 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

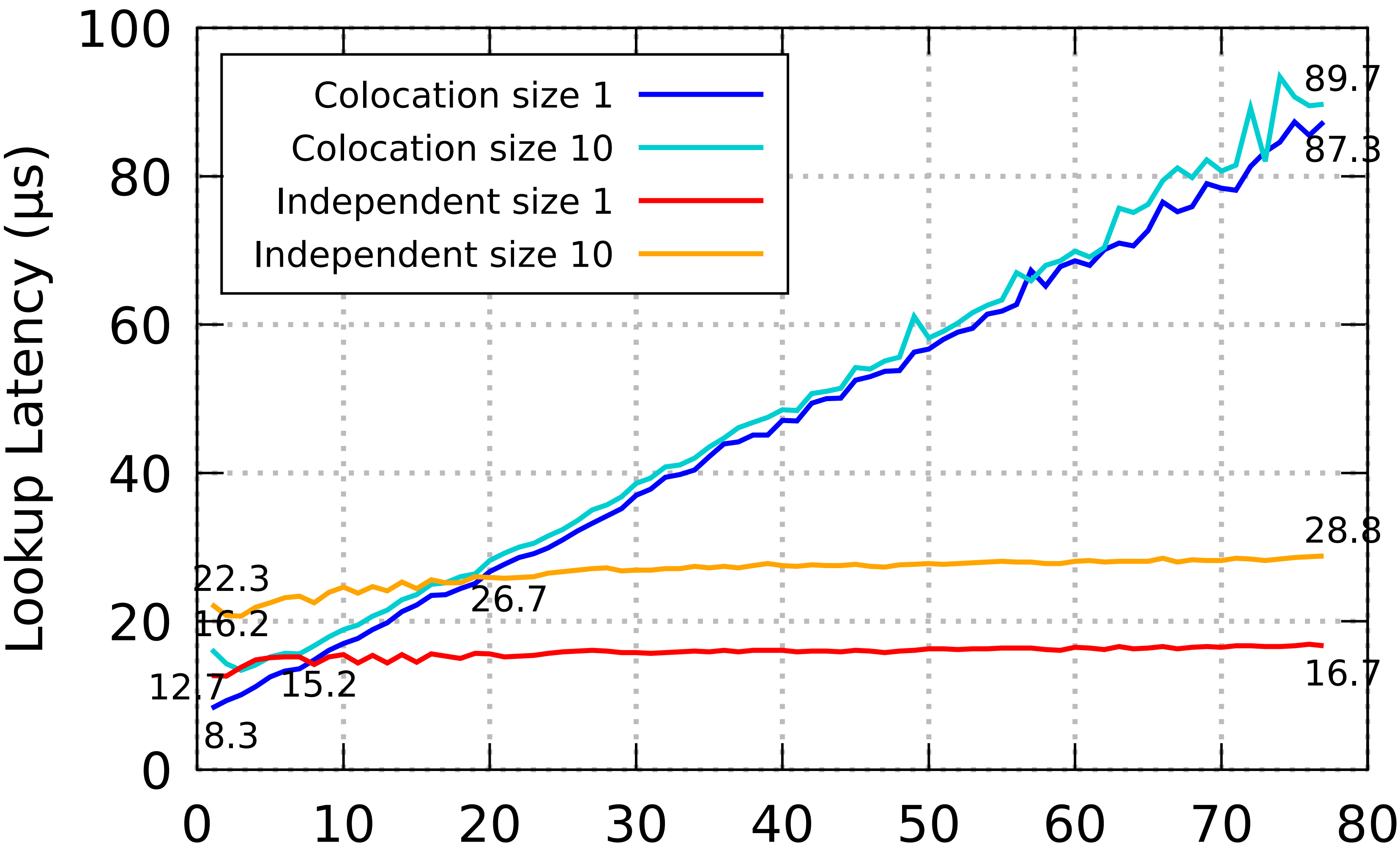
Scalability: Throughput



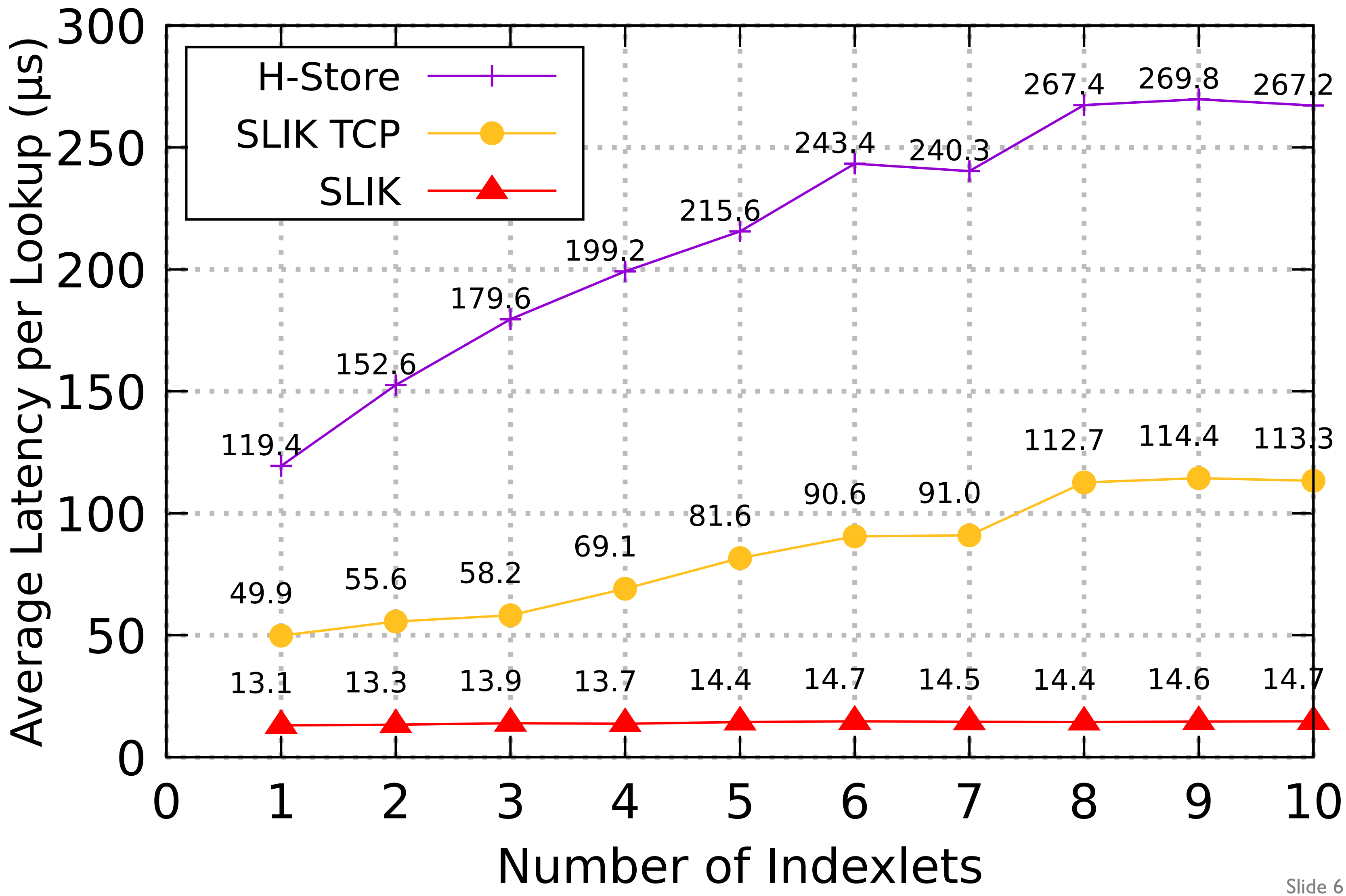
Scalability: Throughput



Scalability: Latency



Scalability: Latency



Talk Outline

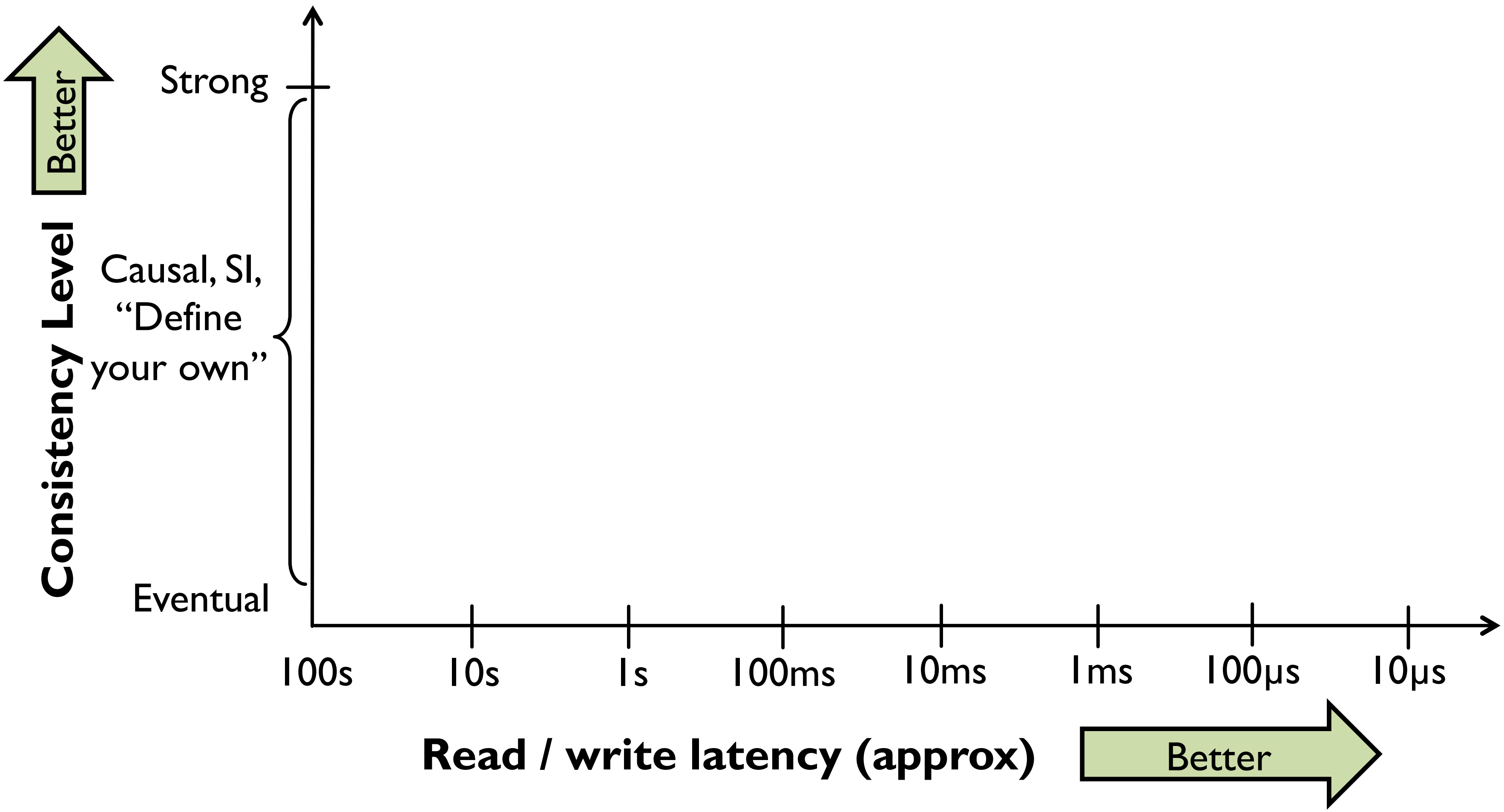
- Motivation
- Design
- Performance
- **Related Work**
- Summary

Related Work

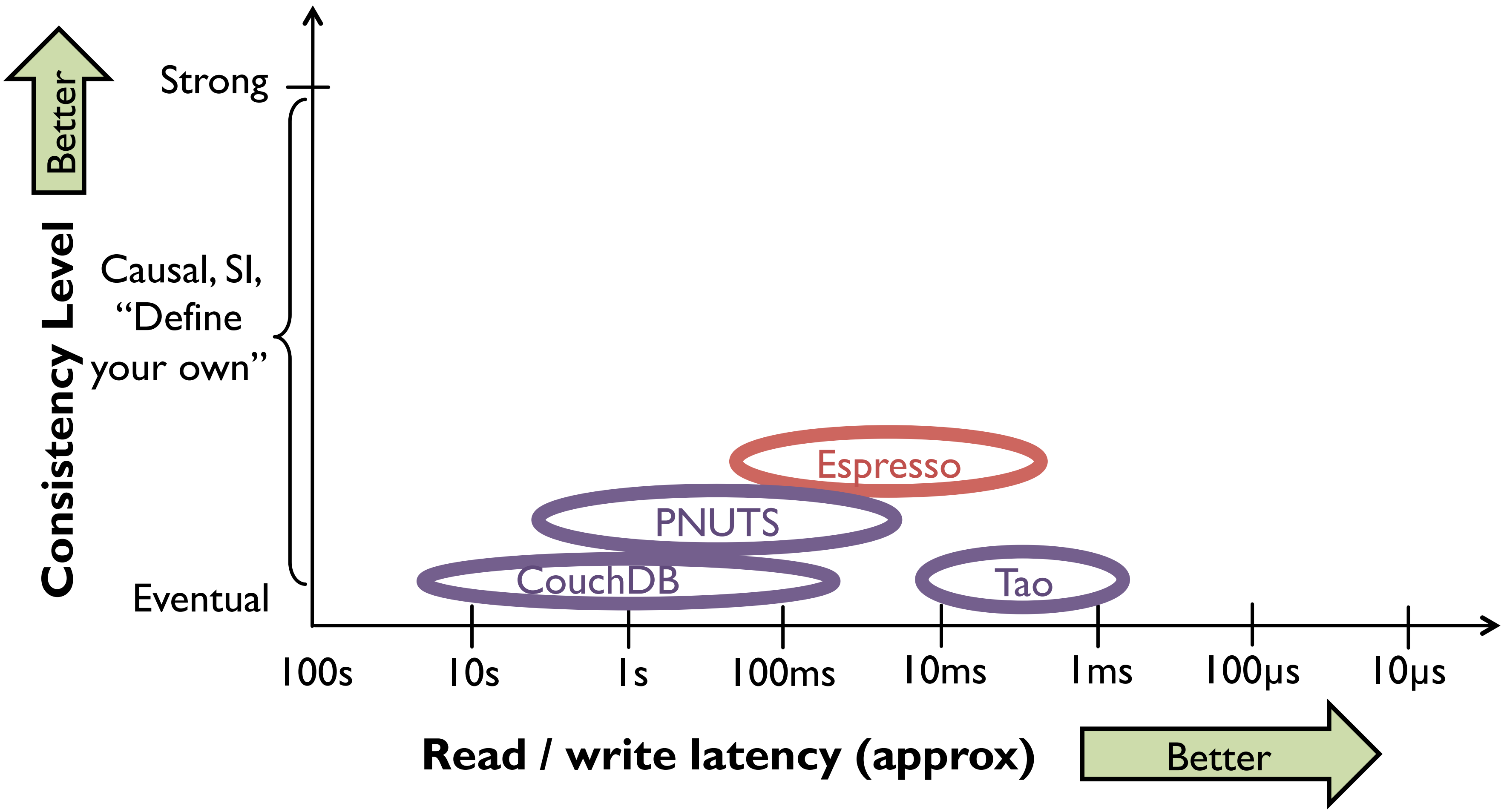
Data storage system

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

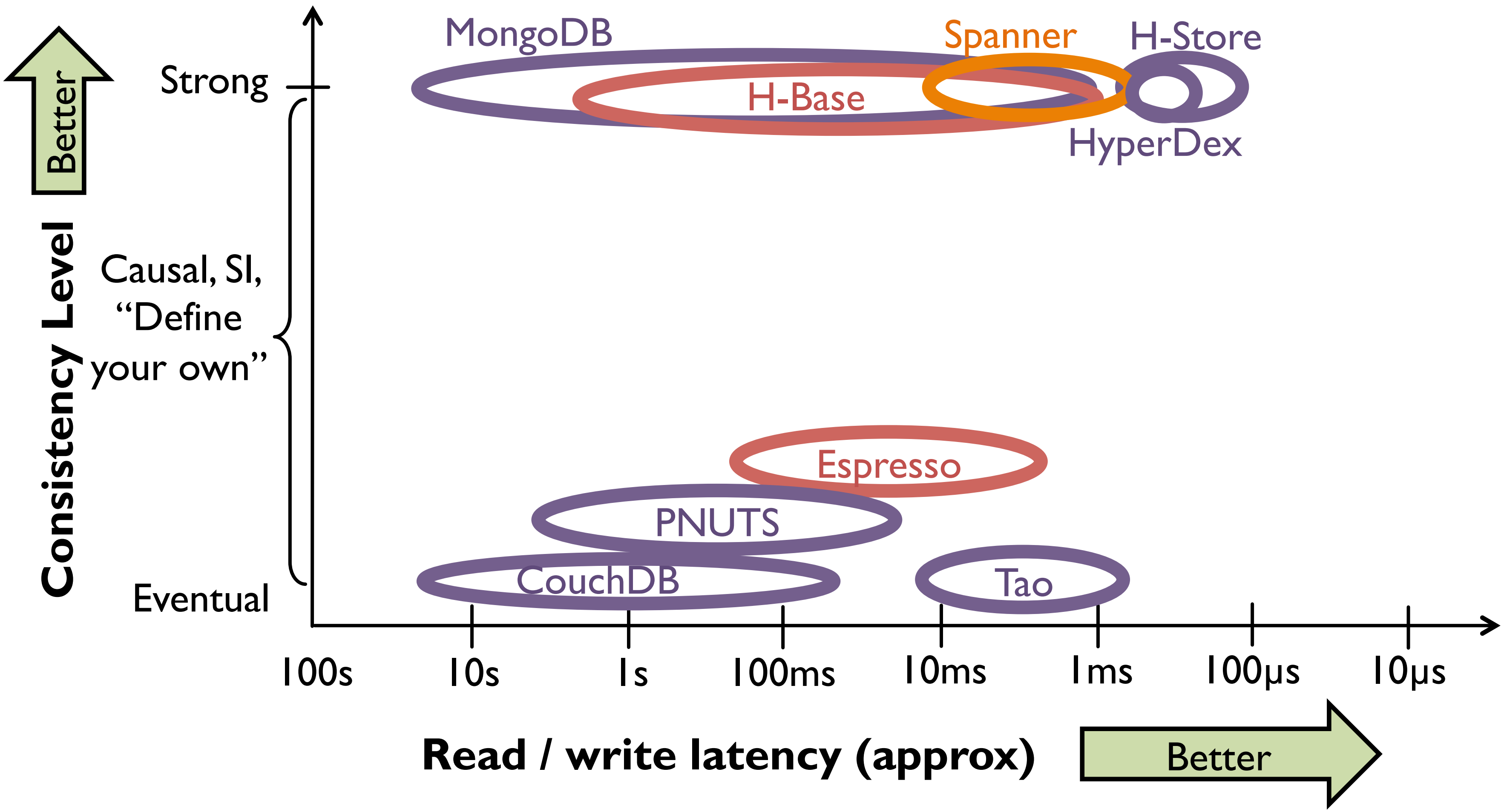
Current Web Scale Datastores



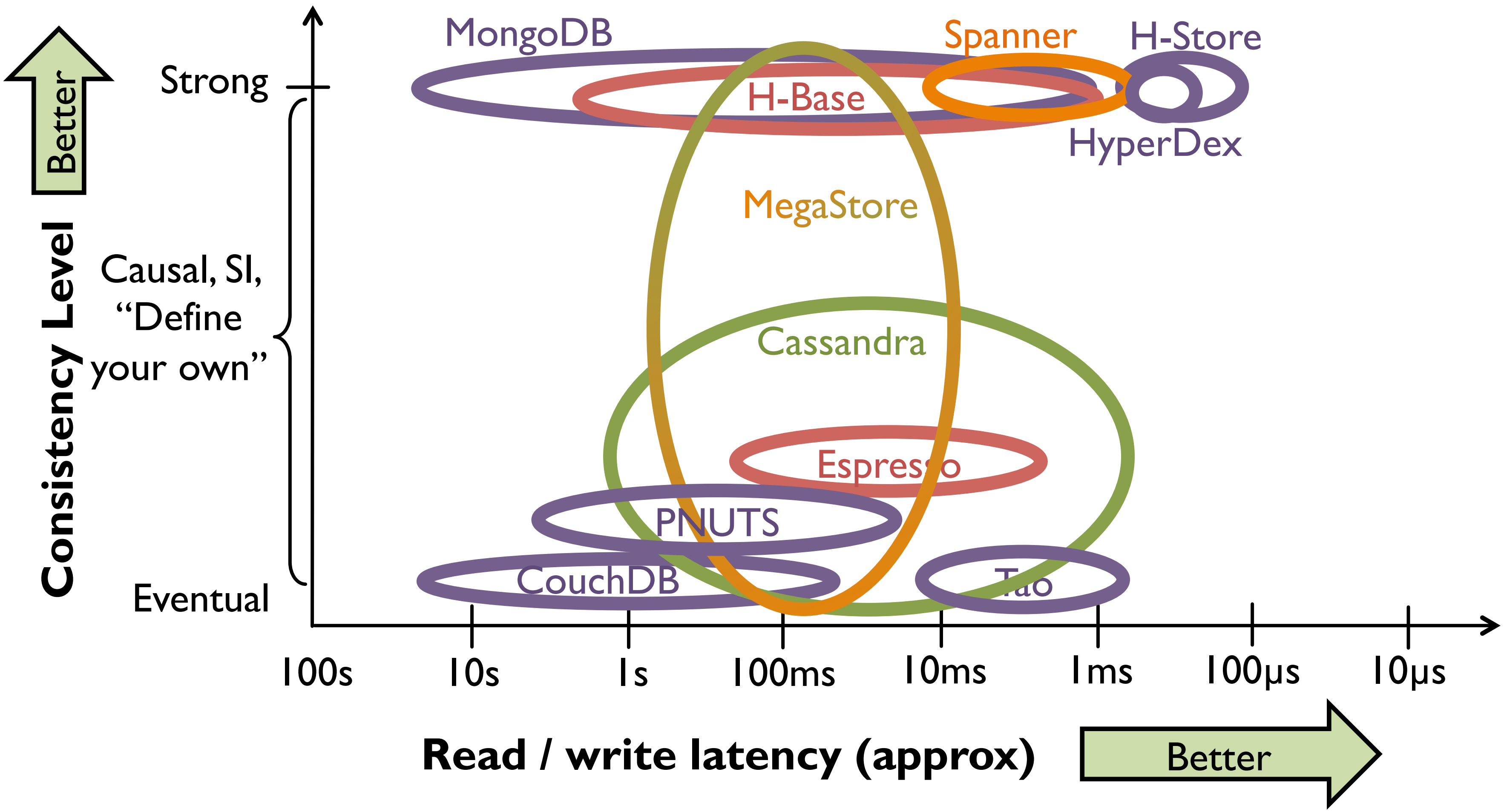
Current Web Scale Datastores



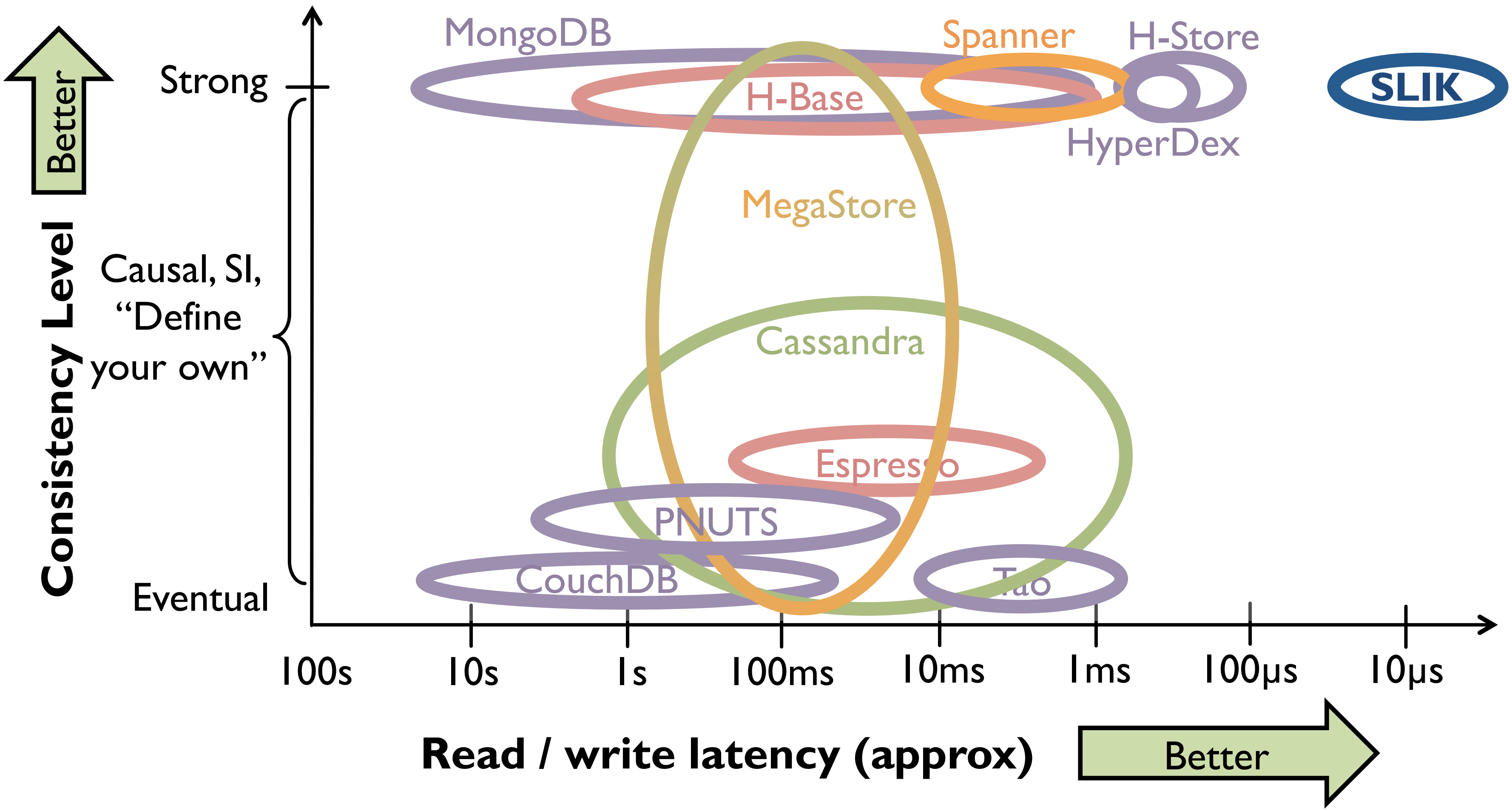
Current Web Scale Datastores



Current Web Scale Datastores



Current Web Scale Datastores



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**?

Summary

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**.

Summary

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**.

Summary

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 independent partitioning we get:
linear throughput increase and
minimal impact on latency as the scale increases

Summary

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

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

Summary

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

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

PLATFORMLAB Stanford University