SLIK: 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)

PlatformLab

Hypothesis

A key value store can support
highly consistent secondary indexes
while operating at low latency and large scale.

Introduction

- **SLIK:**

  Scalable Low-latency Indexes for a Key-value Store
  - Enables multiple secondary keys for each object
  - Allows lookups and range queries on these keys
- **Key design features:**
  - **Scalability** using independent partitioning
  - **Consistency with minimal performance overheads** using an ordered write approach
- **Performance**
  - 11-13 µs indexed reads
  - 29-37 µs writes/overwrites of objects with one indexed attribute
  - Linear throughput increase with increasing number of partitions
- **Feedback welcome!**

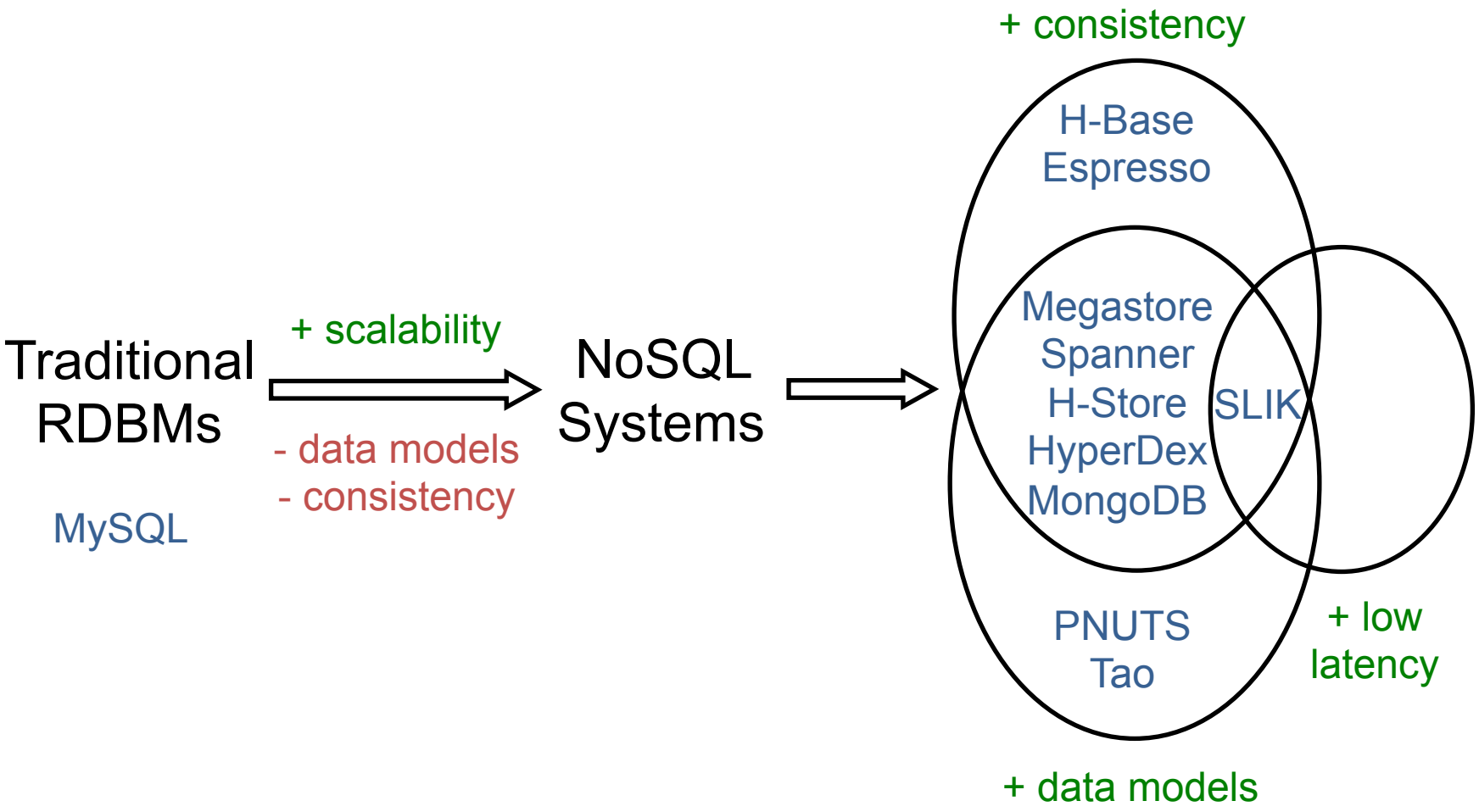
Talk Outline

- Motivation
- Data Model and API
- Design
- Performance
- Related Work
- Summary

Talk Outline

- **Motivation**
- Data Model and API
- Design
- Performance
- Related Work
- Summary

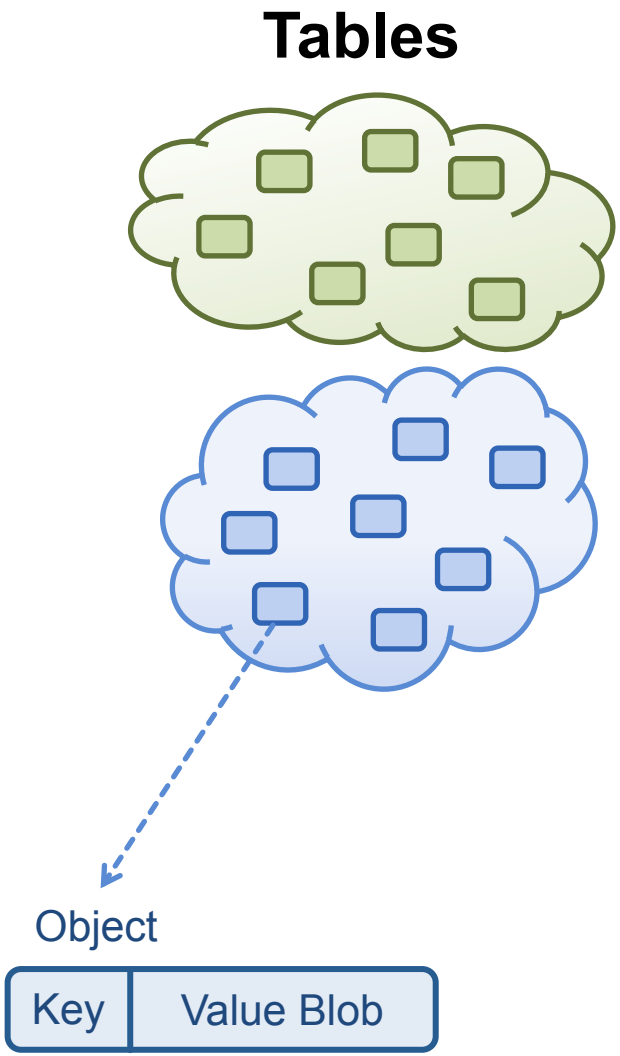
Motivation


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

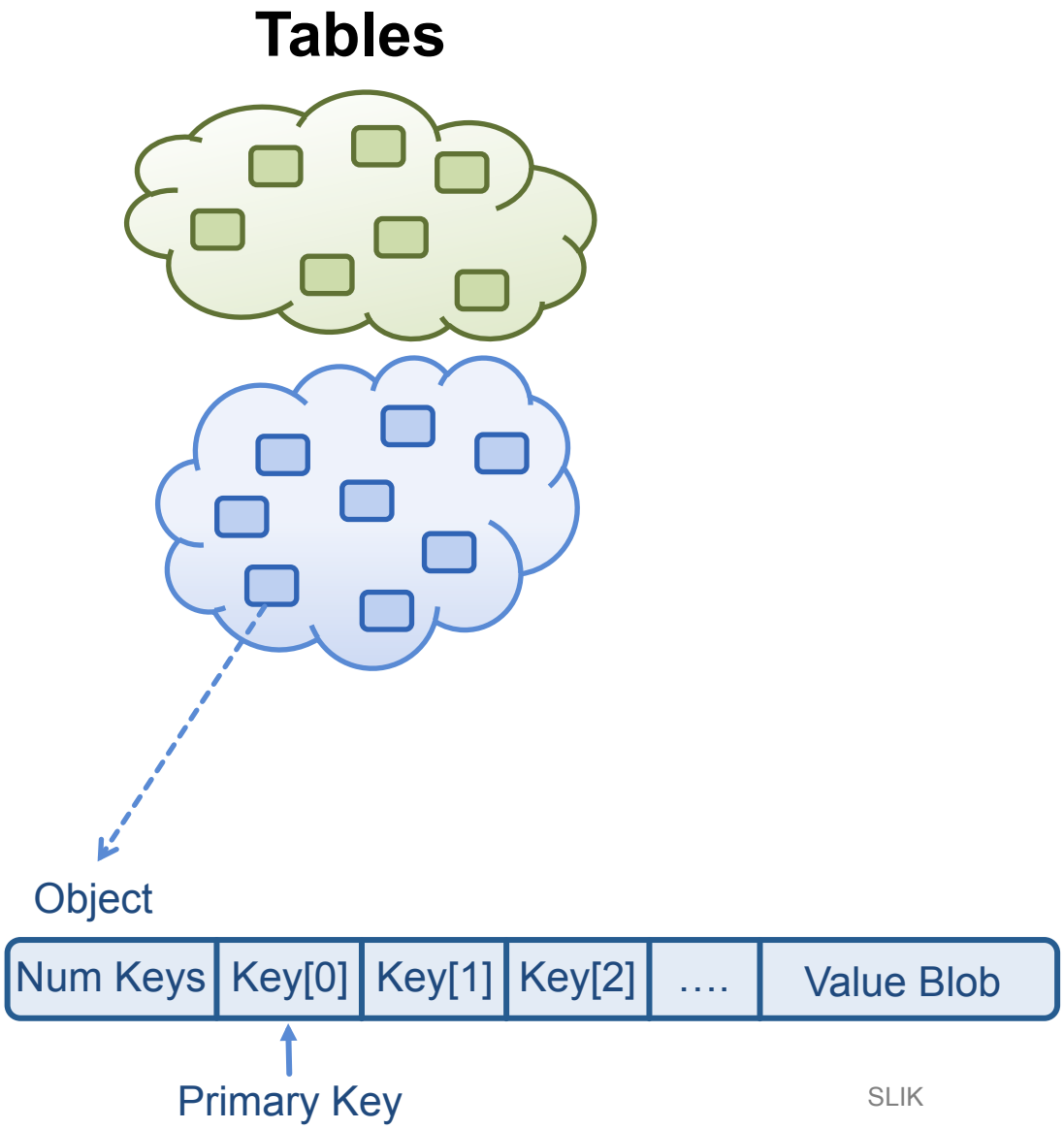
Talk Outline

- Motivation
- **Data Model and API**
- Design
- Performance
- Related Work
- Summary

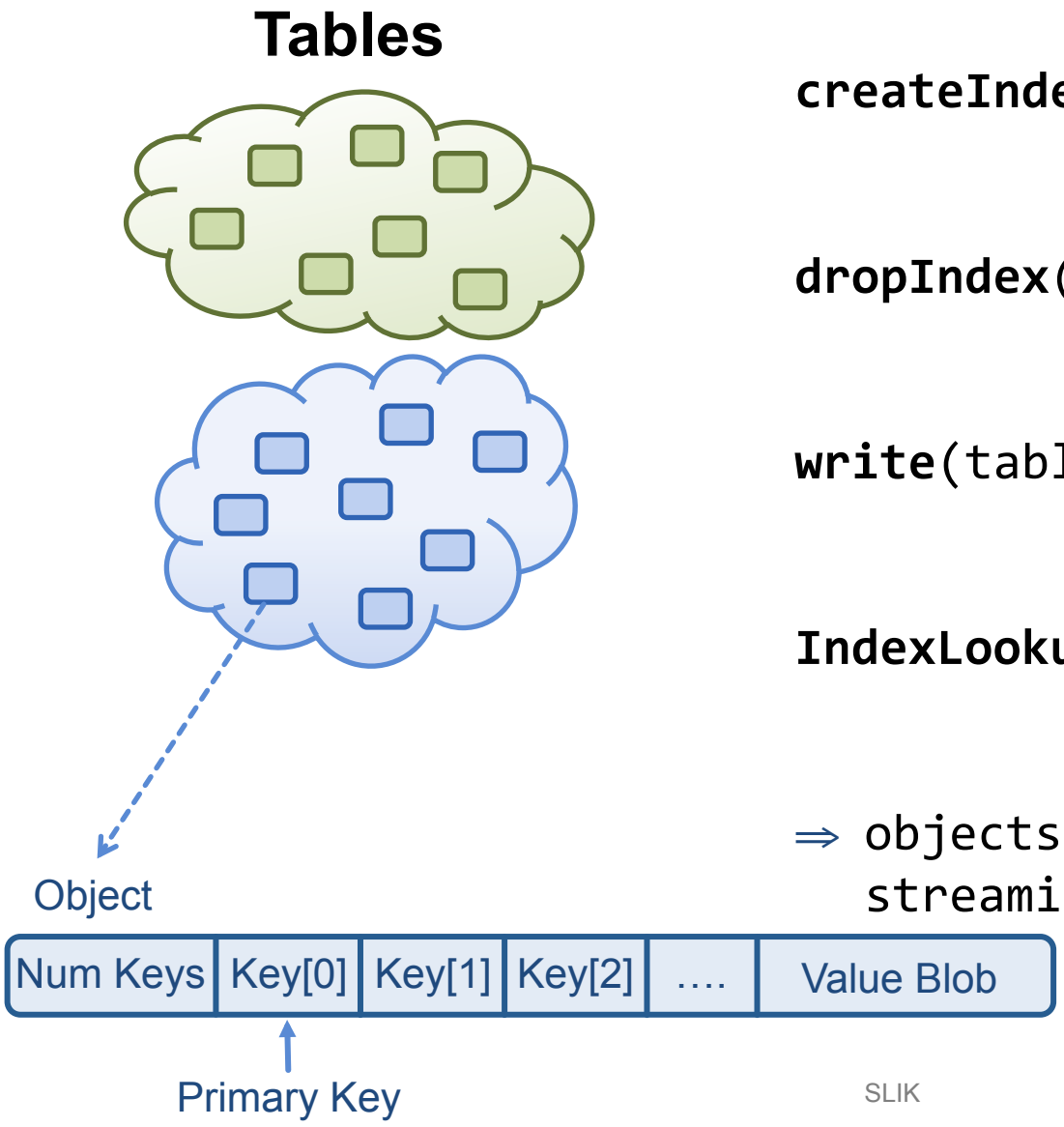
Object Format



Object Format



Object Format and API




```
createIndex(tableId, indexId,
            indexType)
dropIndex(tableId, indexId)

write(tableId, keys, value)

IndexLookup(tableId, indexId,
            keyRange)
```

⇒ objects in a sorted order via streaming interface

Talk Outline

- Motivation
- Data Model and API
- **Design**
- Performance
- Related Work
- Summary

Design Goals

- **Scalable distributed system**
- **Consistency expected from a centralized system** (with minimal performance overheads)

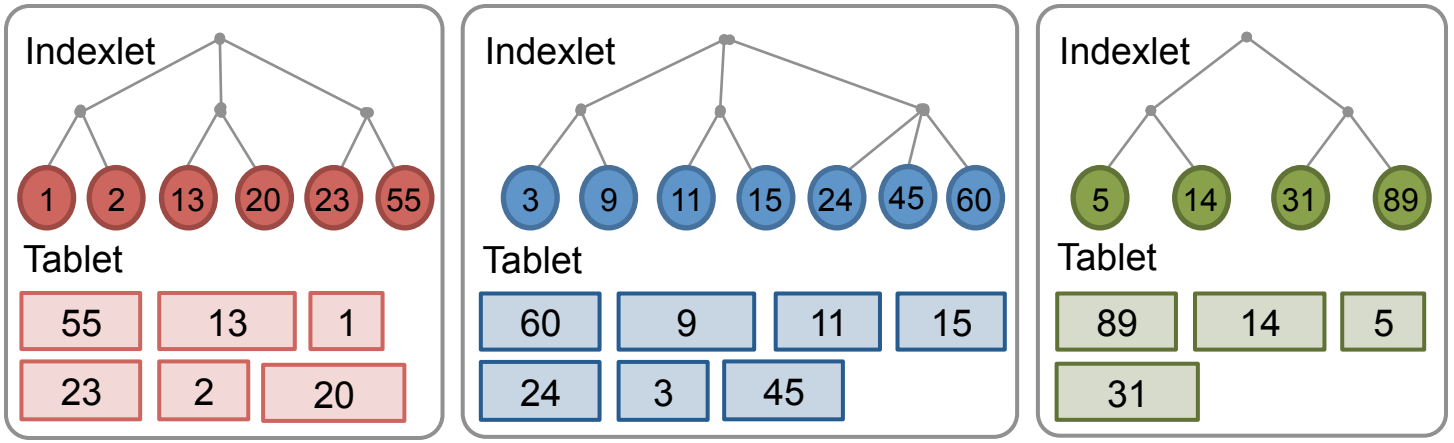
Design Goals

- **Scalable distributed system**
- **Consistency expected from a centralized system** (with minimal performance overheads)

Index Partitioning

Colocation Approach

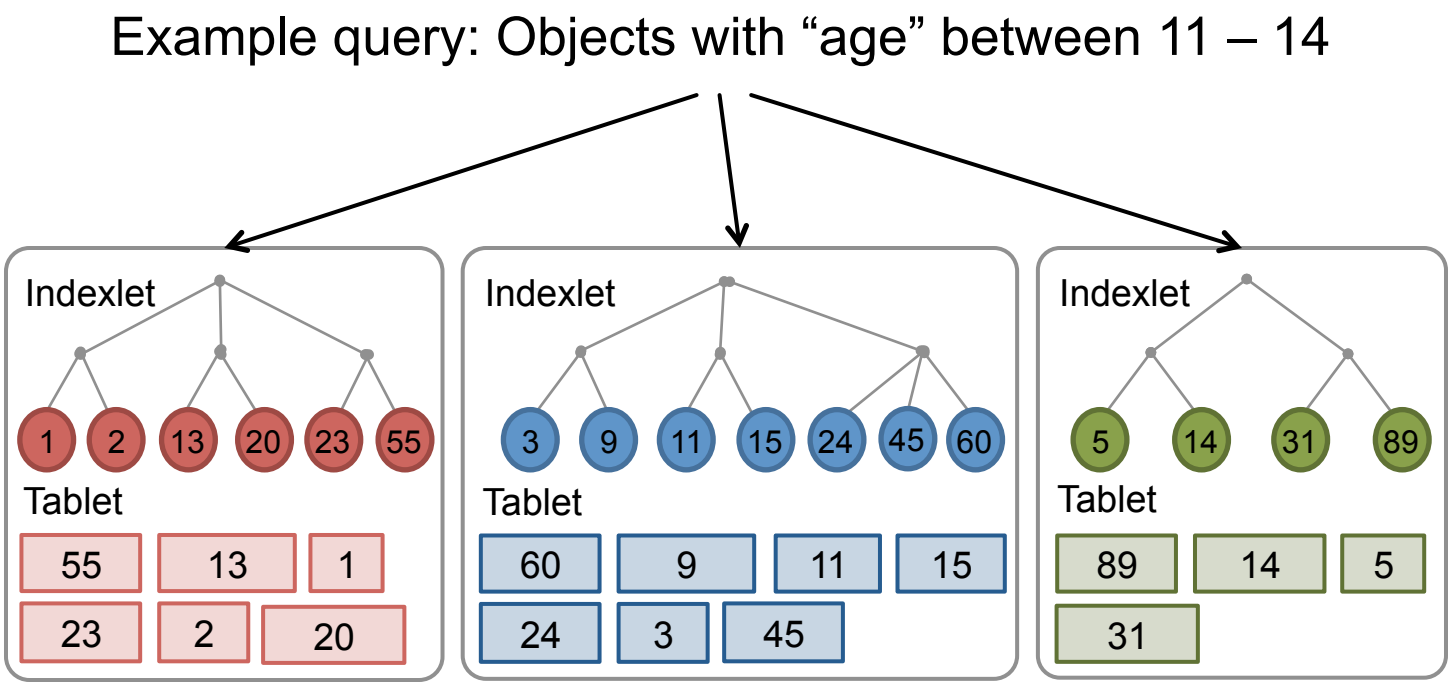
- Colocate index entries and objects
- One of the keys used to partition the table's objects and indexes
- No particular association between index partitions and index key ranges



Index Partitioning

Colocation Approach

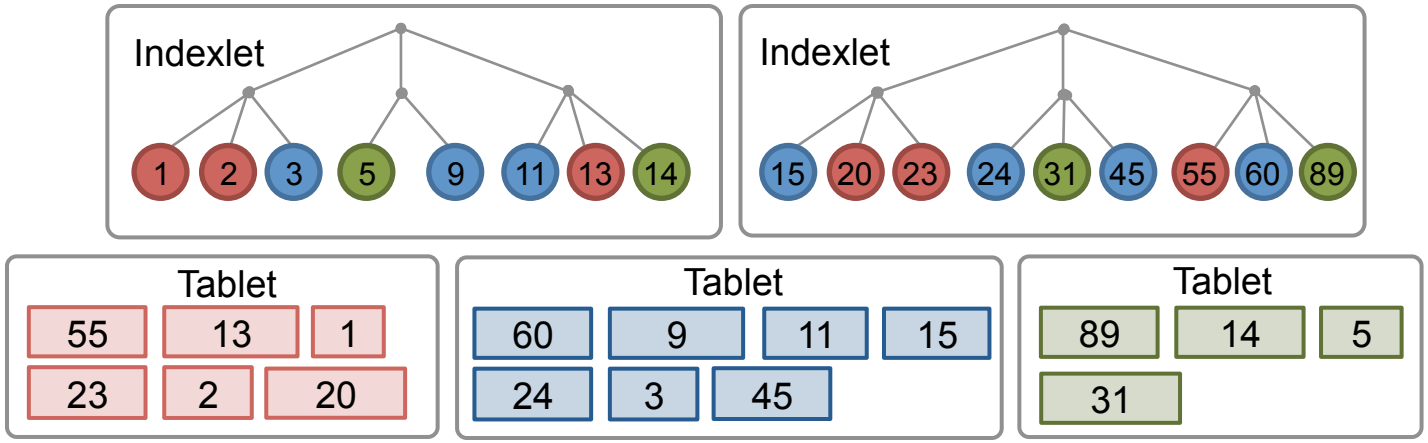
- Colocate index entries and objects
- One of the keys used to partition the table's objects and indexes
- No particular association between index partitions and index key ranges



Index Partitioning

Independent Partitioning

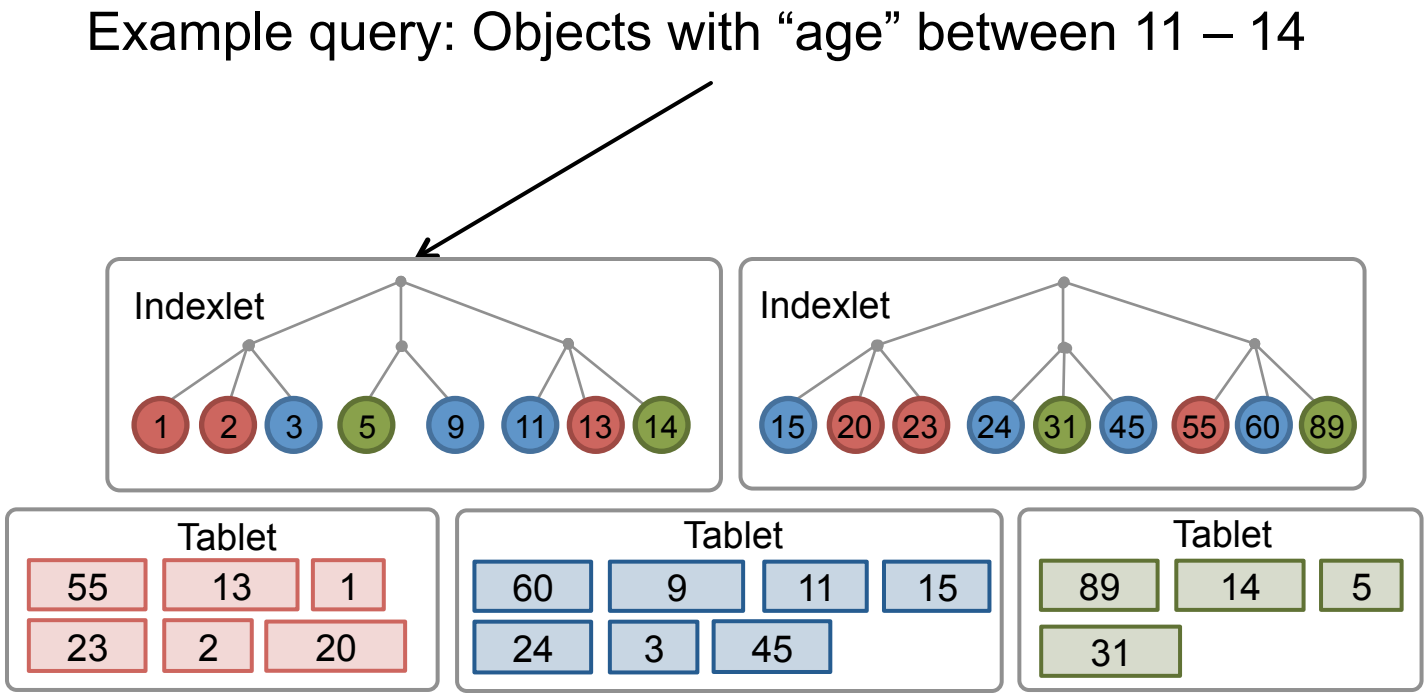
- Partition each index and table independently
- Partition each index according to sort order for that index



Index Partitioning

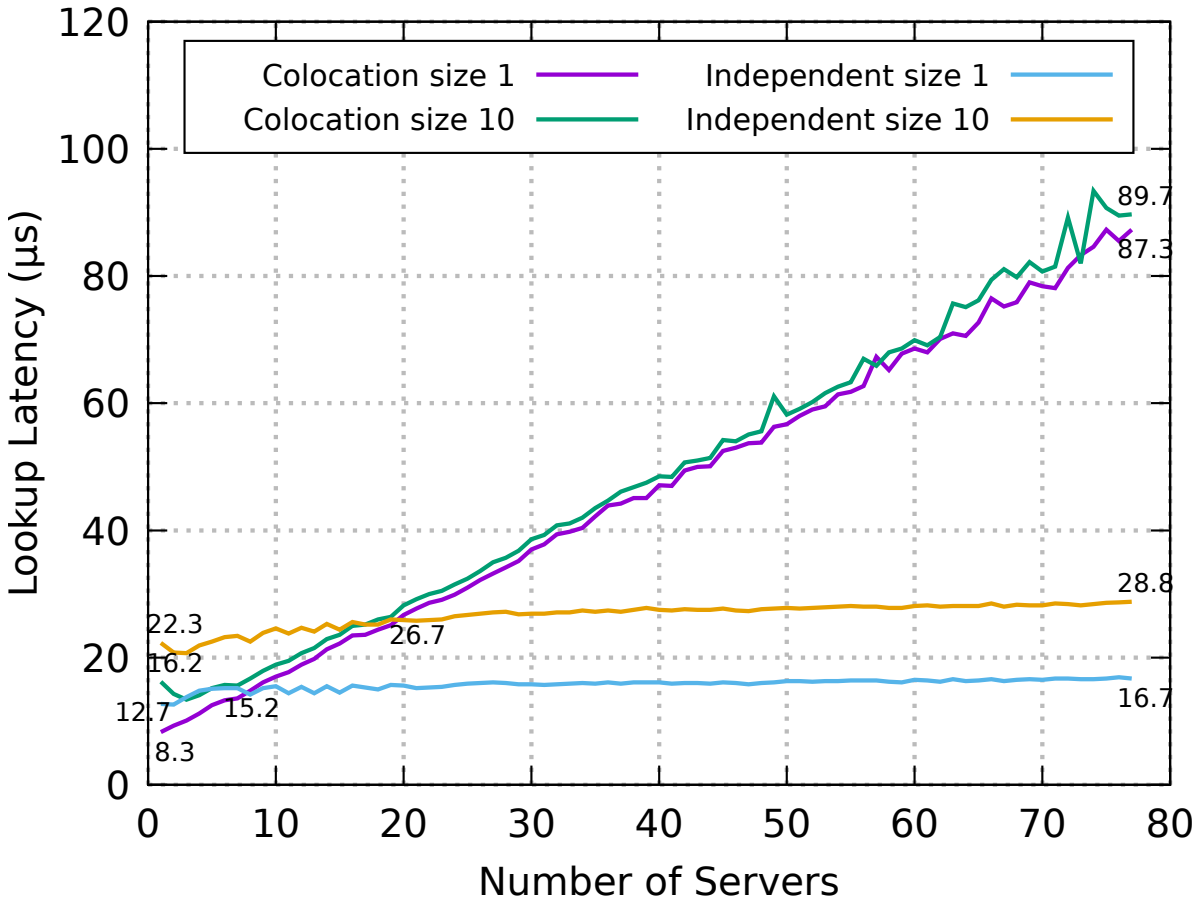
Independent Partitioning

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



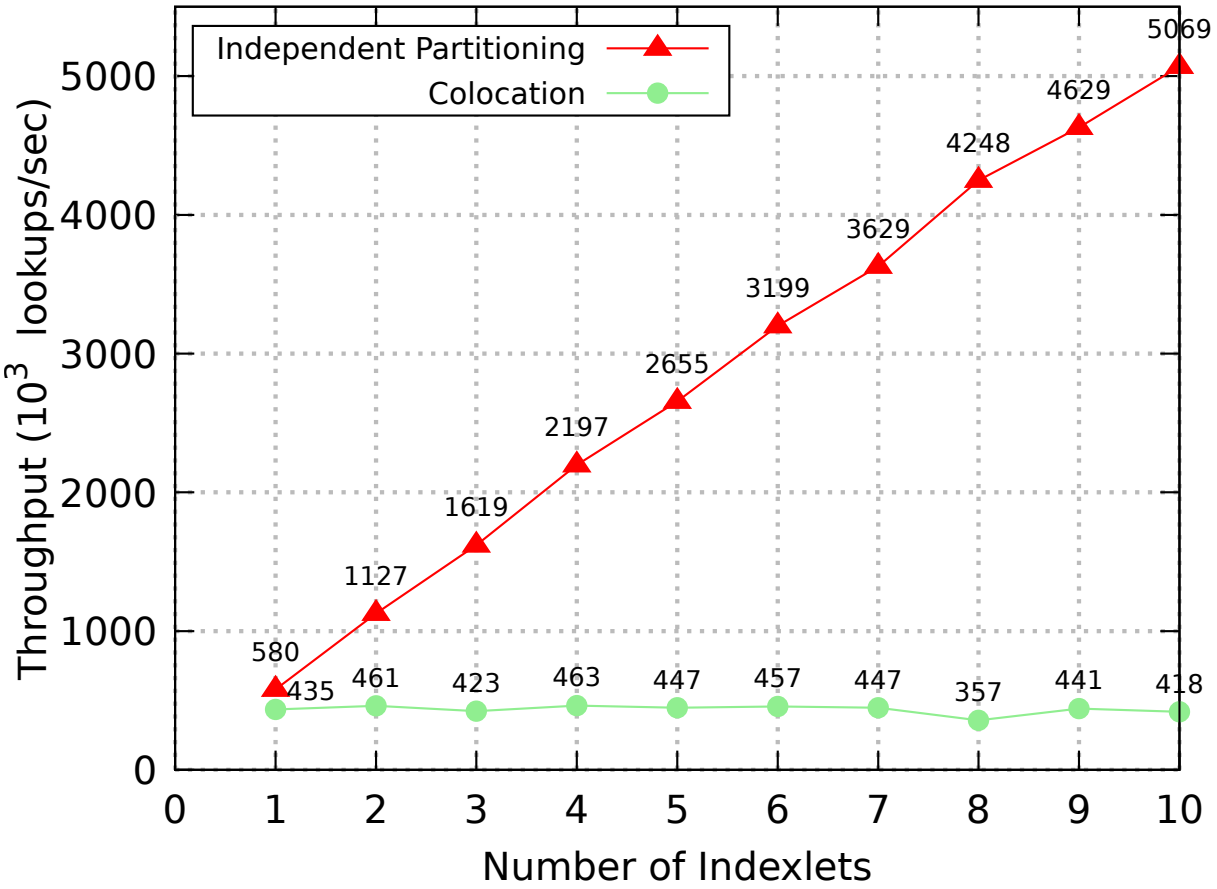
Scalable!

Index Partitioning



Latency for IndexLookup: single table with one index with varying num indexlets
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Index Partitioning



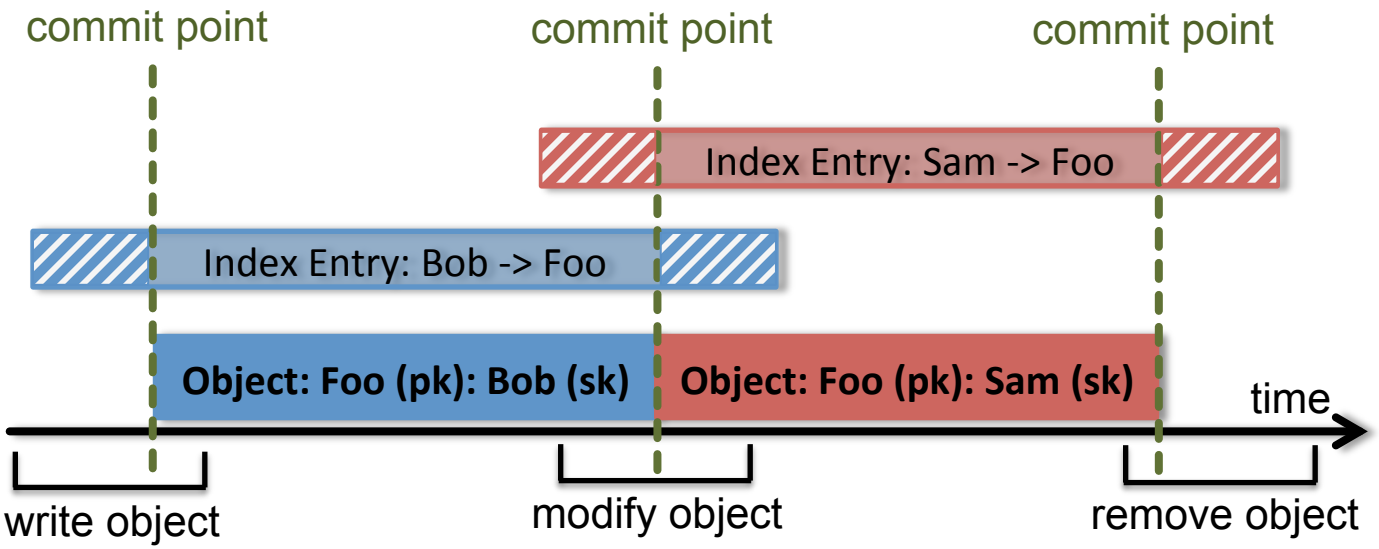
Throughput for IndexLookup: single table with one index with varying num indexlets
Queried via multiple clients
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Design Goals

- **Scalable distributed system:**
  - Use independent partitioning
  - But: indexed object writes: distributed operations
- **Consistency expected from a centralized system** (with minimal performance overheads):
  - 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



Talk Outline

- Motivation
- Data Model and API
- Design
- **Performance**
- Related Work
- Summary

Performance

Implemented SLIK in RAMCloud

- Distributed in-memory key-value storage system
- Designed for large-scale applications
- Optimized to operate at lowest possible latency

Performance

Questions:

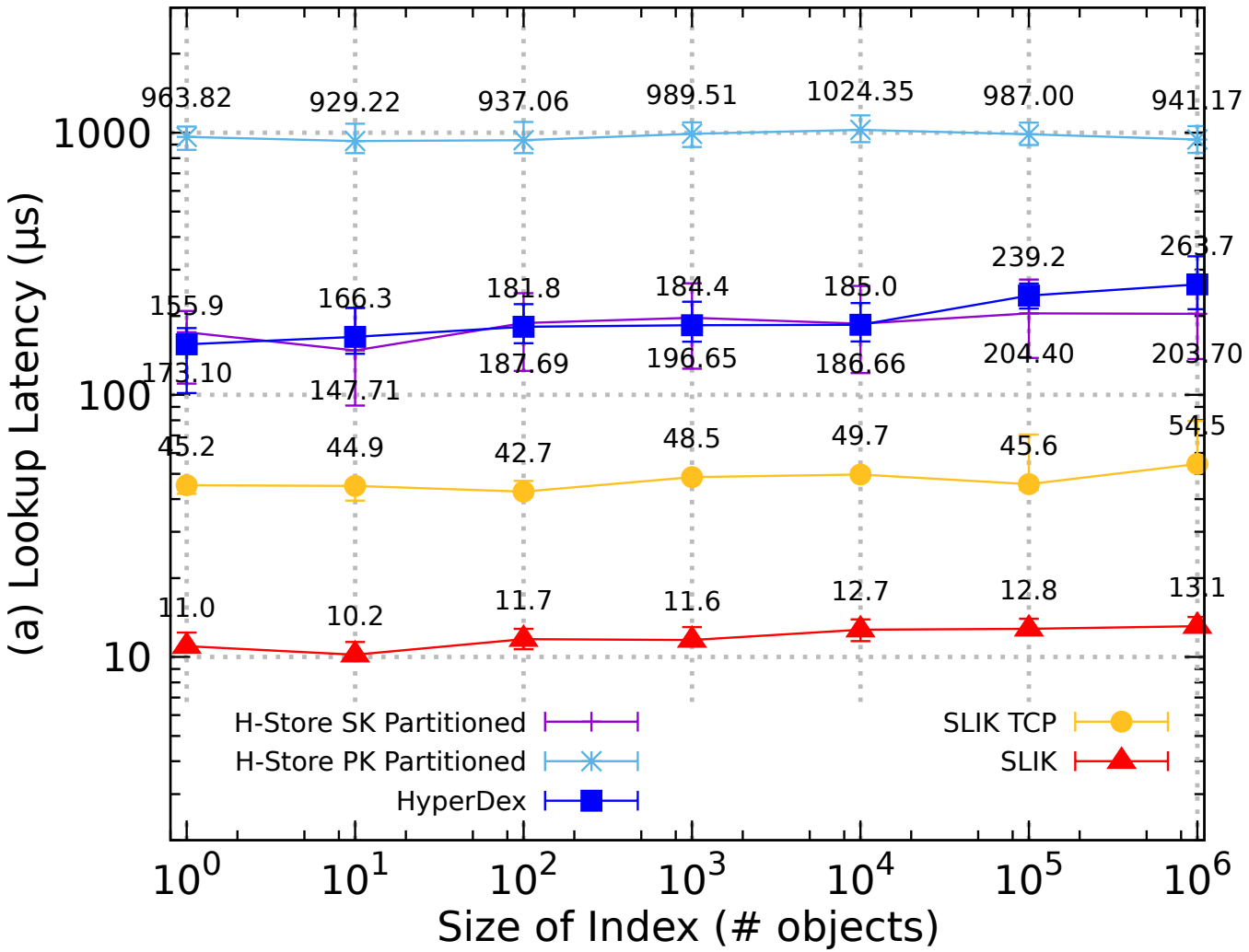
- Does SLIK meet the low latency goal?
- Does SLIK meet the scalability goal?
- How does the performance of indexing with SLIK compare to other state-of-the-art systems?

Performance

Systems we compared:

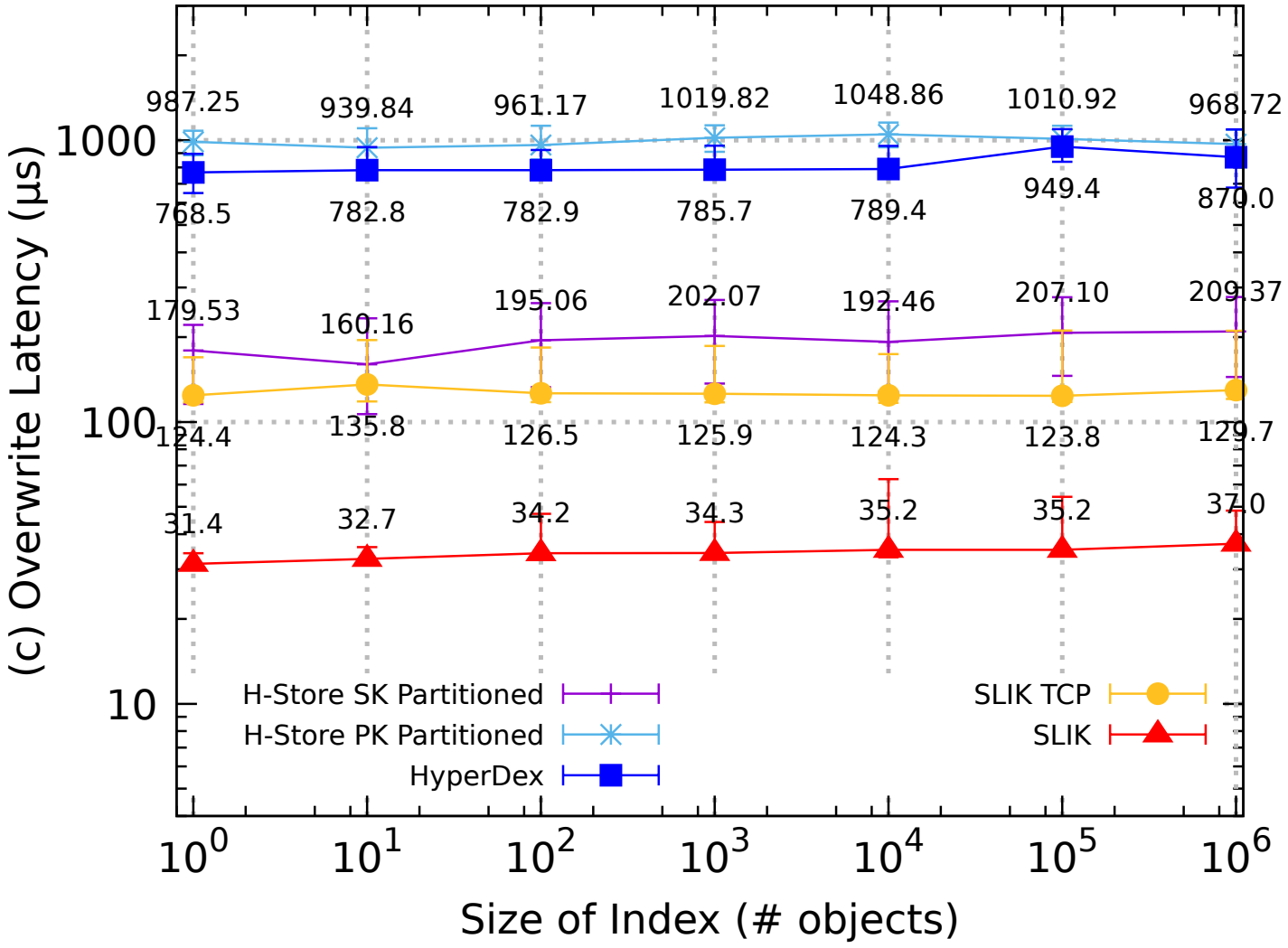
- **HyperDex:**
  - Spaces containing objects
  - Objects have primary key and multiple attributes
  - Data (and indexes) partitioned using hyperspace hashing
  - Each index contains all object data
- **H-Store:**
  - Main memory database
  - SQL+ACID
  - Data (and indexes) partitioned based on specified attribute
  - Many parameters for tuning
    - Got assistance from developers to tune for each test
    - Examples: txn_incoming_delay, partitioning column

Lookup Latency



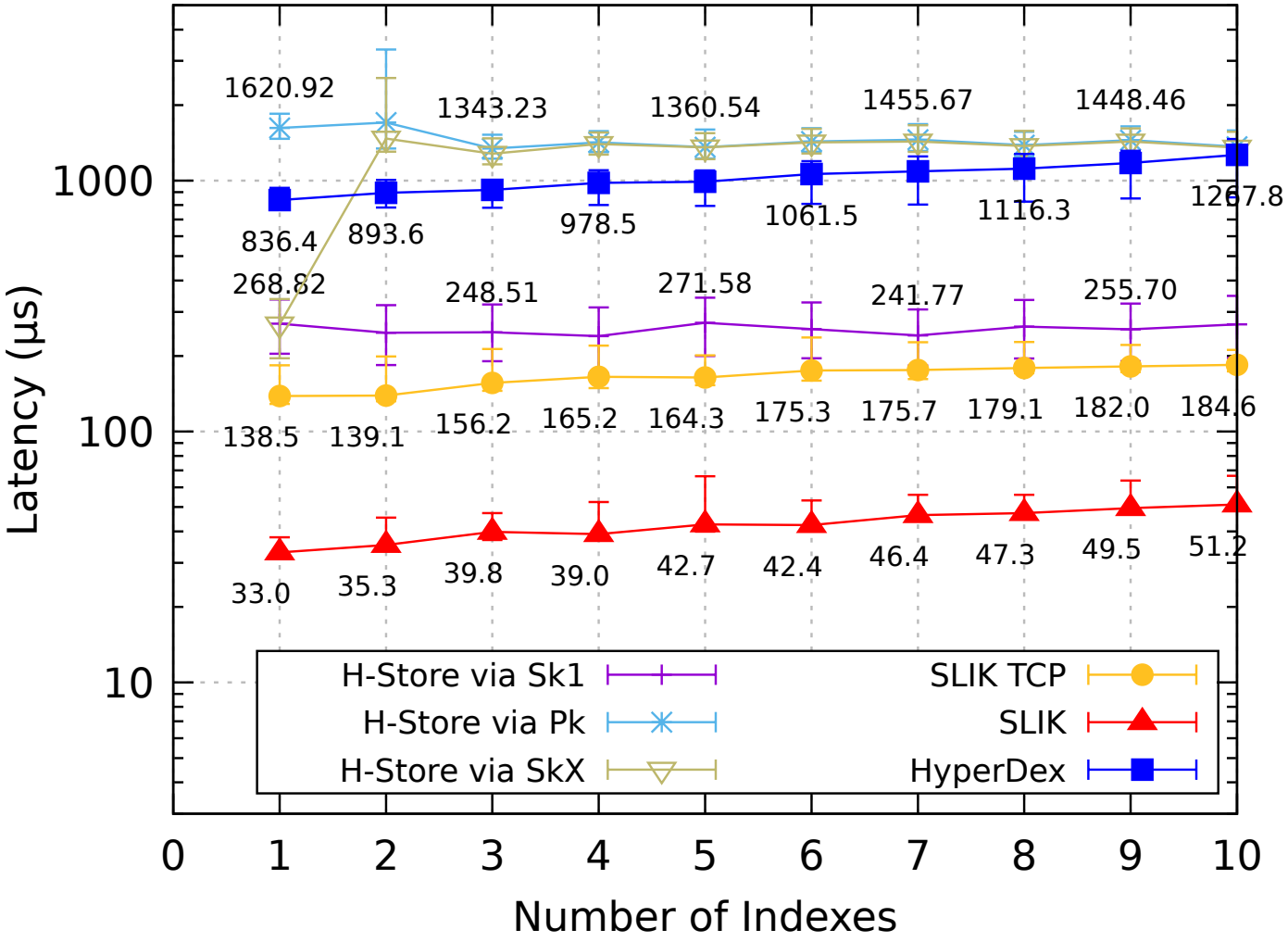
Single table with one index having a single partition;
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Overwrite Latency



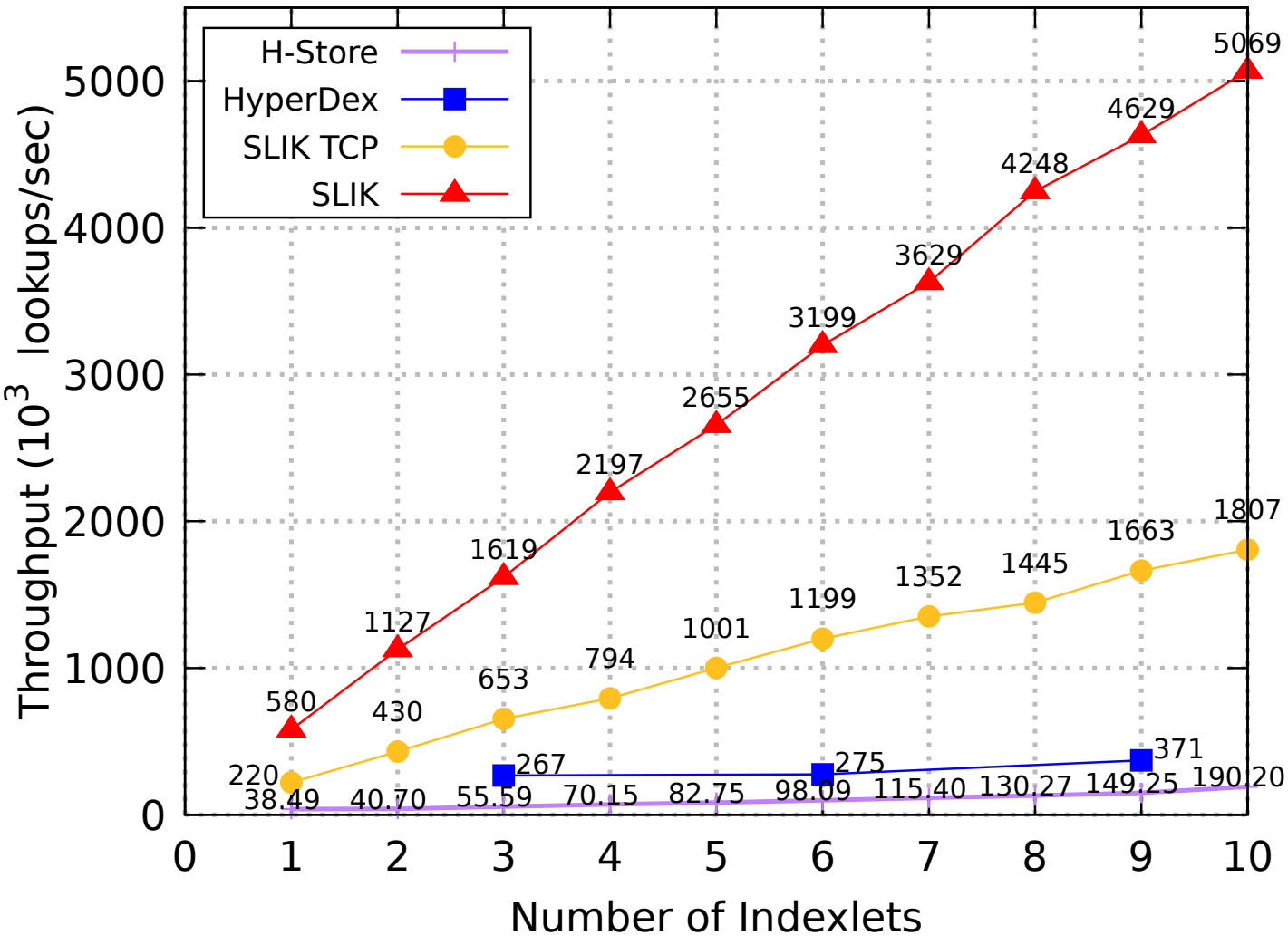
Single table with one index having a single partition;
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Multiple Secondary Indexes



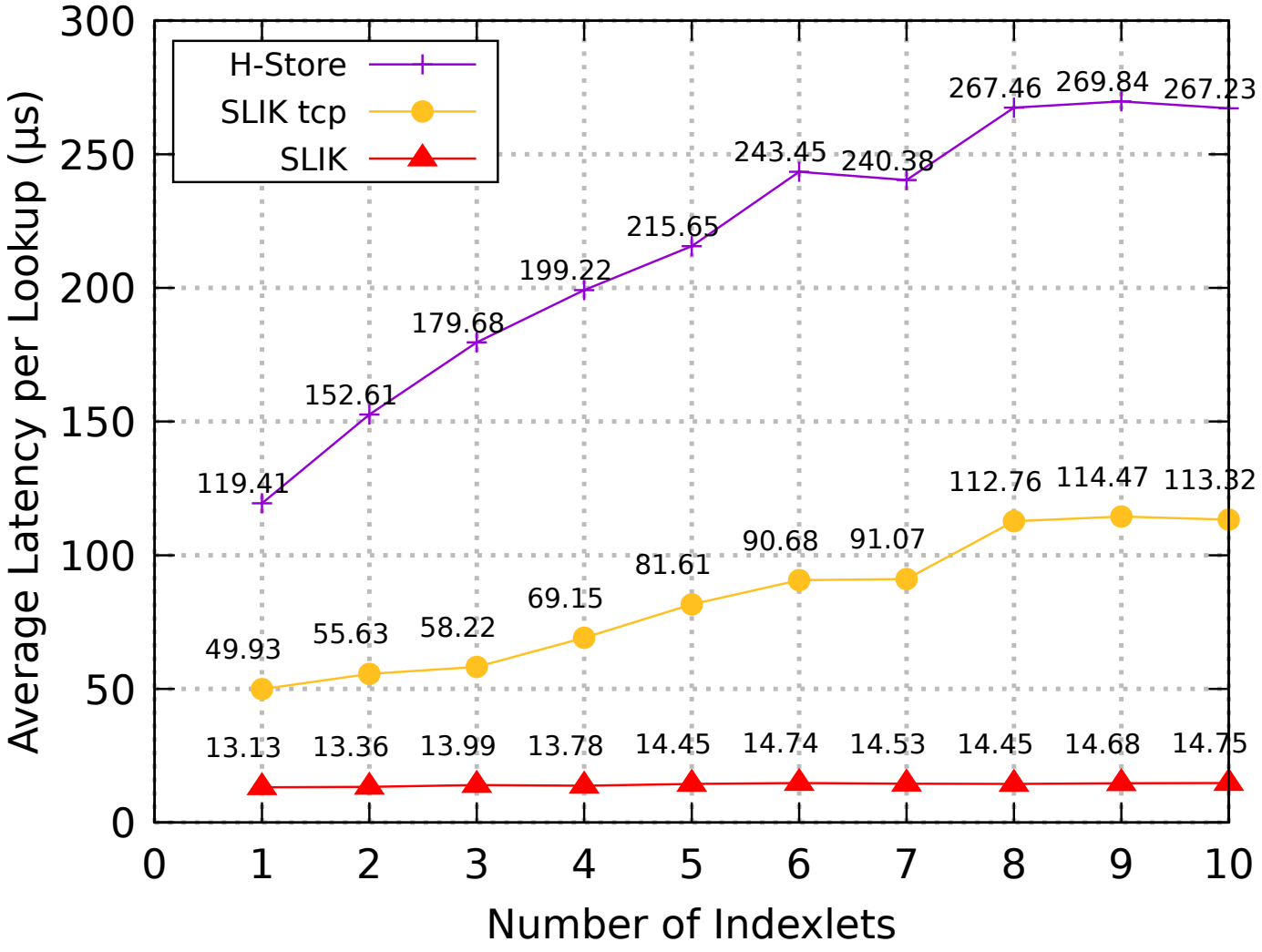
Single table with varying num indexes, each having a single partition;
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Scalability



Single table with one index having varying num partitions;
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Scalability



Single table with one index having varying num partitions;
Each object: pk 30 bytes, sk 30 bytes, val 100 bytes

Talk Outline

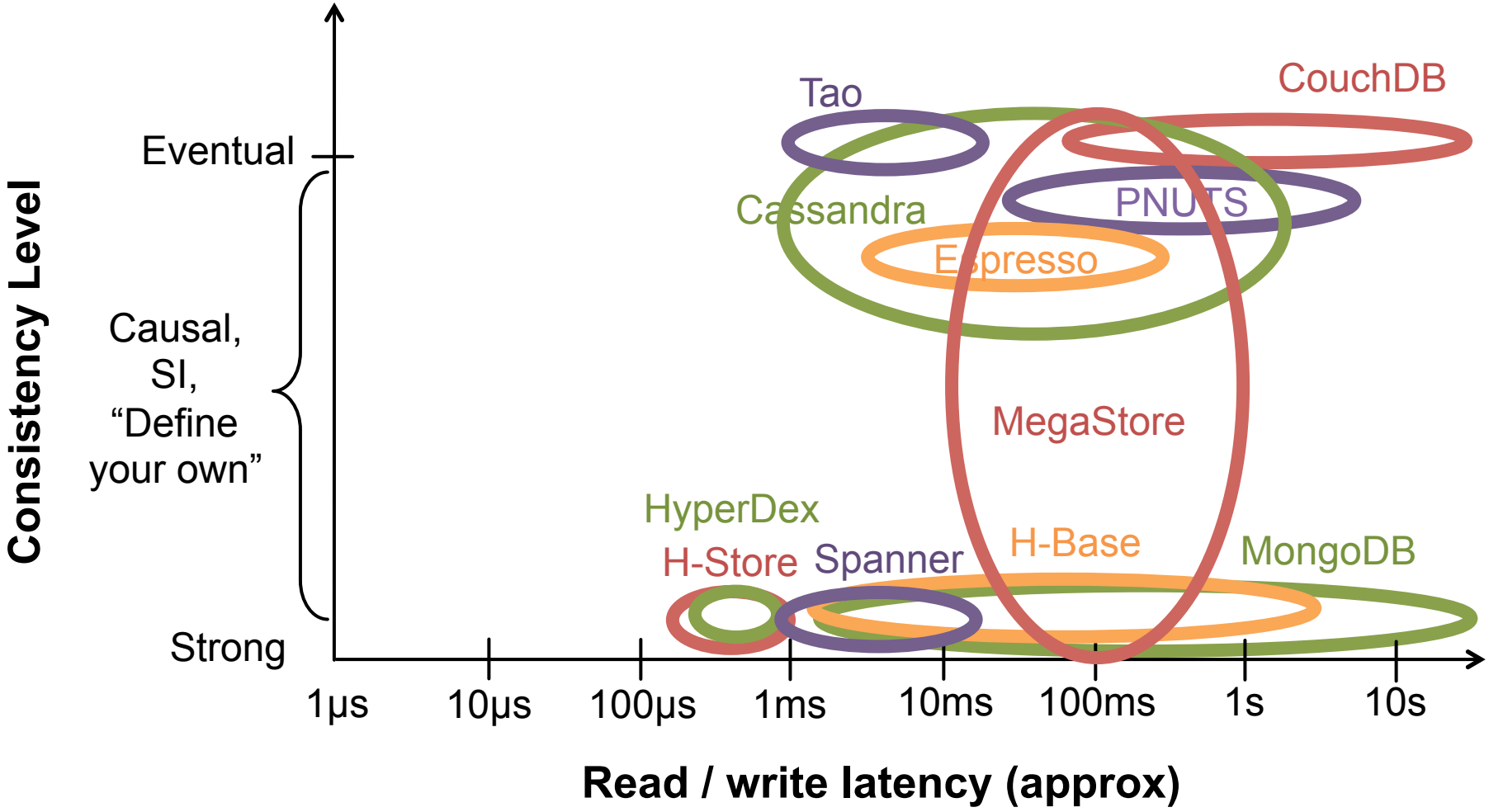
- Motivation
- Data Model and API
- Design
- Performance
- **Related Work**
- Summary

Related Work

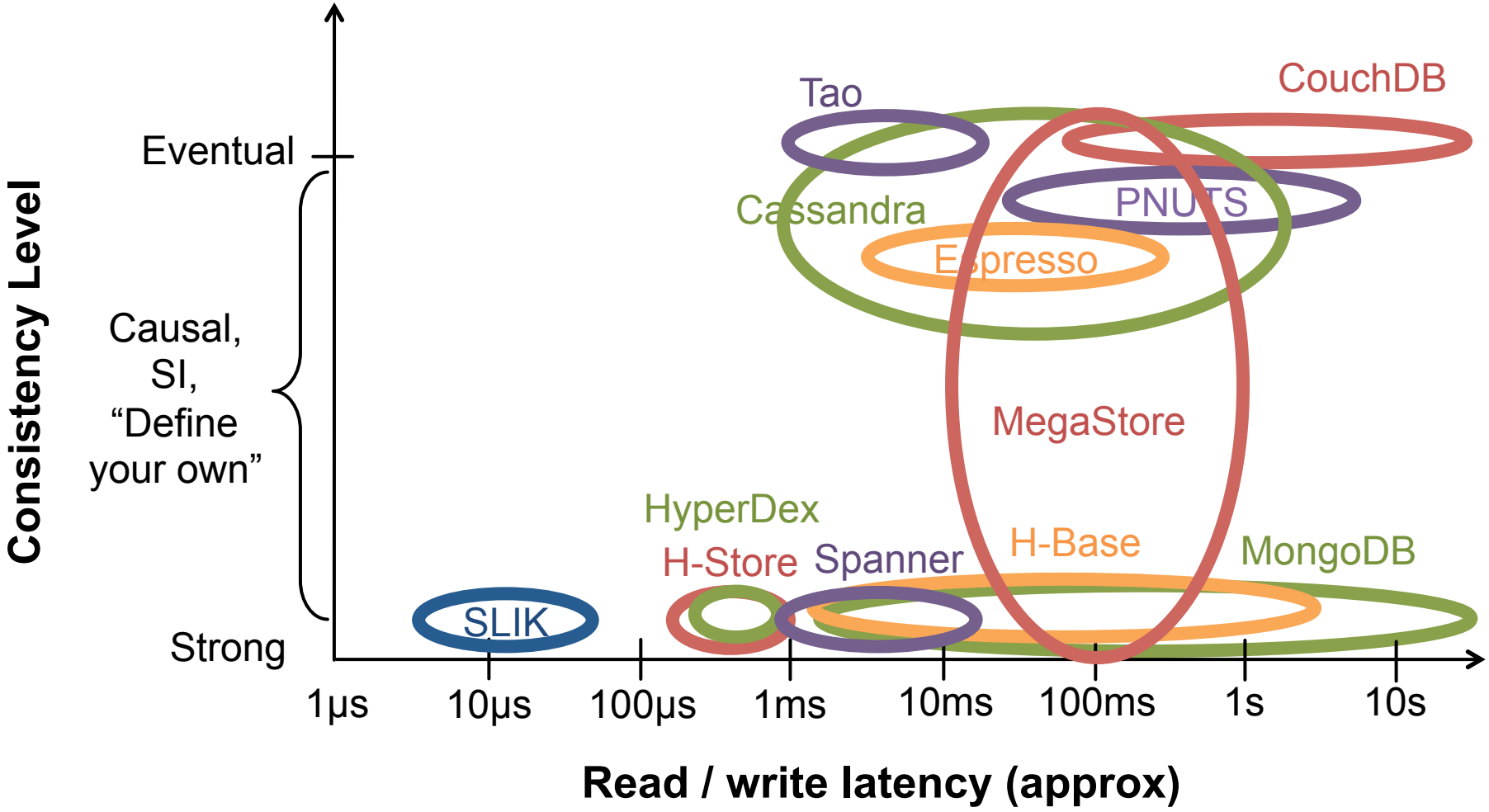
Data storage system

- **Scale** (spectrum from local machine to datacenter)
- **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



Talk Outline

- Motivation
- Data Model and API
- Design
- Performance
- Related Work
- **Summary**

Summary

Lookups and range queries on secondary keys

Ordered writes and indexes as hints

A key value store can support

highly consistent secondary indexes

while operating at low latency and large scale.

11-13 µs indexed reads
29-37 µs writes/overwrites of indexed objects

With increasing number of partitions:

- Linear throughput increase
- Minimal latency impact

Thank you!

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

Code available open source: github.com/PlatformLab/RAMCloud
Papers and other information at: ramcloud.stanford.edu