Fast Scans on Key-Value Stores

By David Shen and Beatrice Tanaga

About the Authors

This work was done in 2017 at ETH Zürich, a university in Switzerland.

Markus Pilman
Snowflake Computing

Kevin Bocksrocker
Microsoft

Lucas Braun
Oracle Labs

Renato Marroquin
PhD Student, ETH Zürich

Donald Kossmann
Microsoft Research, previously Professor at ETH Zürich

Table of Contents

Table of Contents

Introduction to KVS

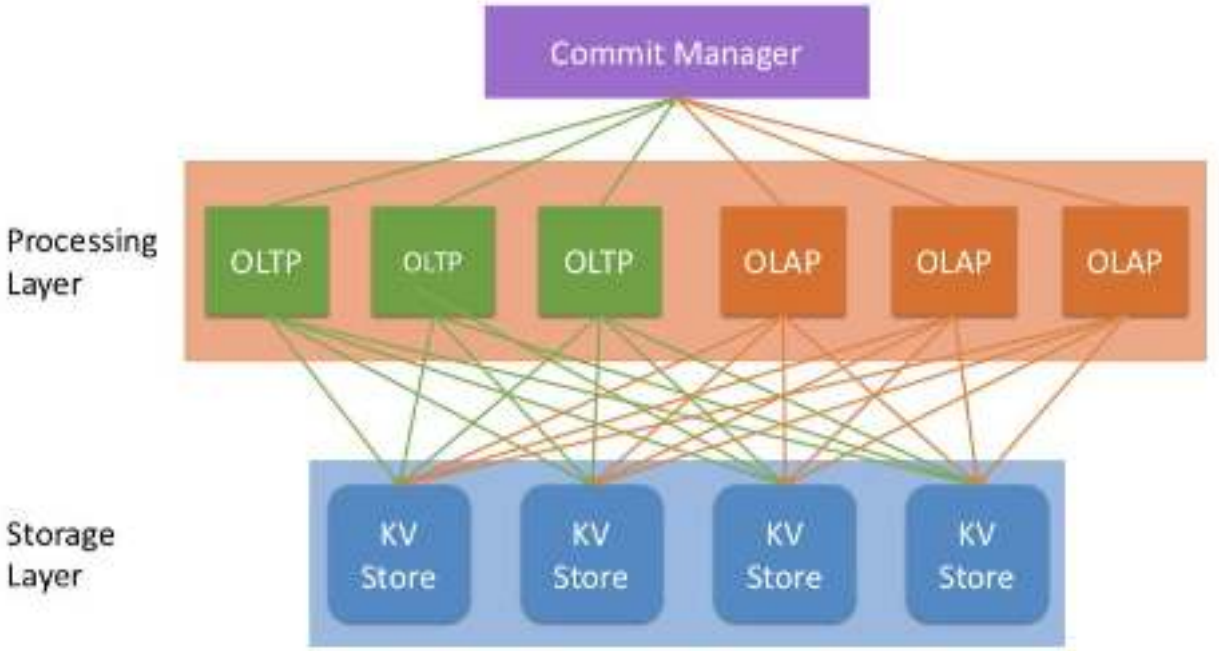
What is a Key-Value Store?

- A type of **storage engine** that stores values indexed by a key.

RamCloud

State of the Art KVS

- Processing layer is responsible for **synchronization**
- Snapshot Isolation is used for synchronization in the commit manager.
- A form of Multi-Version Concurrency Control (a way for the system to provide concurrent accesses to the DB)

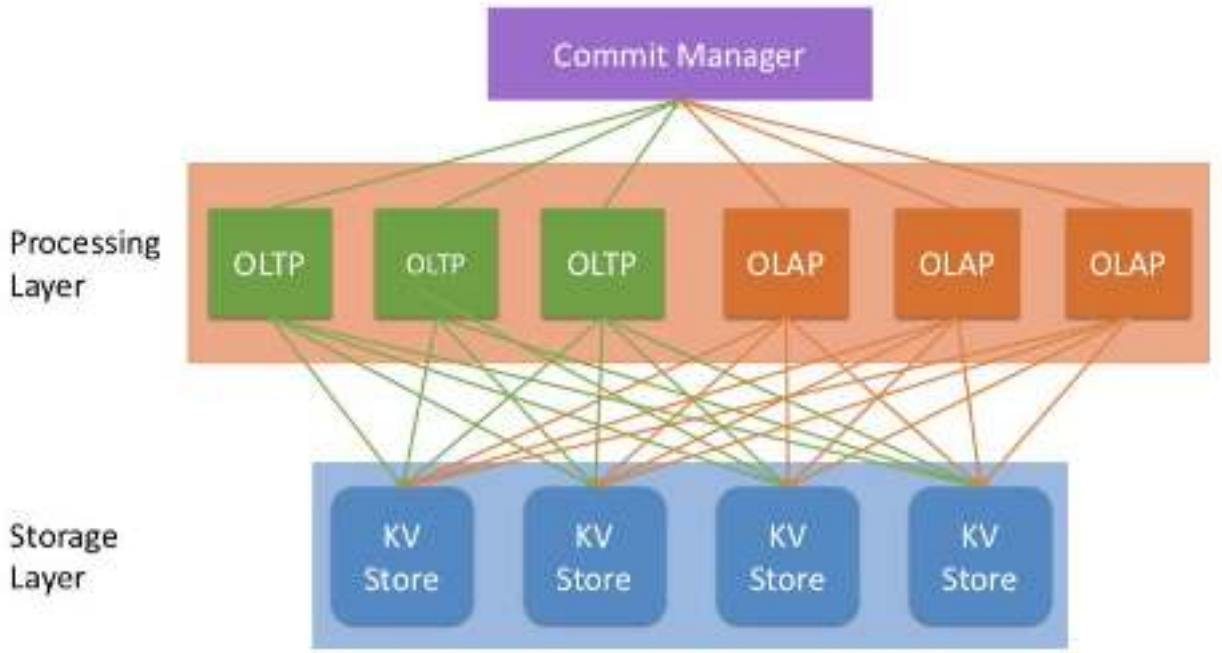


SQL-over-NoSQL Architecture

Advantages of this Design

- Elasticity
- Scalability
- Sustains high throughput for get/put (OLTP) workloads
- Can decouple the storage component and allow it to be part of a completely isolated offering (eg. DynamoDB as part of AWS).

Any downsides?



SQL-over-NoSQL Architecture

Downsides

Issues:

- Access patterns of OLTP and OLAP workloads are different
- OLAP workloads requires reading large, or at times all, portions of the data
- Conflict of interest: Get/Put workloads require sparse indexes, while scans require spatial locality.

Key-Value Store	Scan Time
Cassandra [28]	19 minutes
RAMCloud [34]	46 seconds
HBase [20]	36 seconds
RocksDB [16]	2.4 seconds
Kudu [19]	1.8 seconds
MemSQL [31]	780 milliseconds
TellStore-Row	*197 milliseconds*
TellStore-Log	*133 milliseconds*
TellStore-Column	*84 milliseconds*

```
SELECT max(B) FROM main_table
```

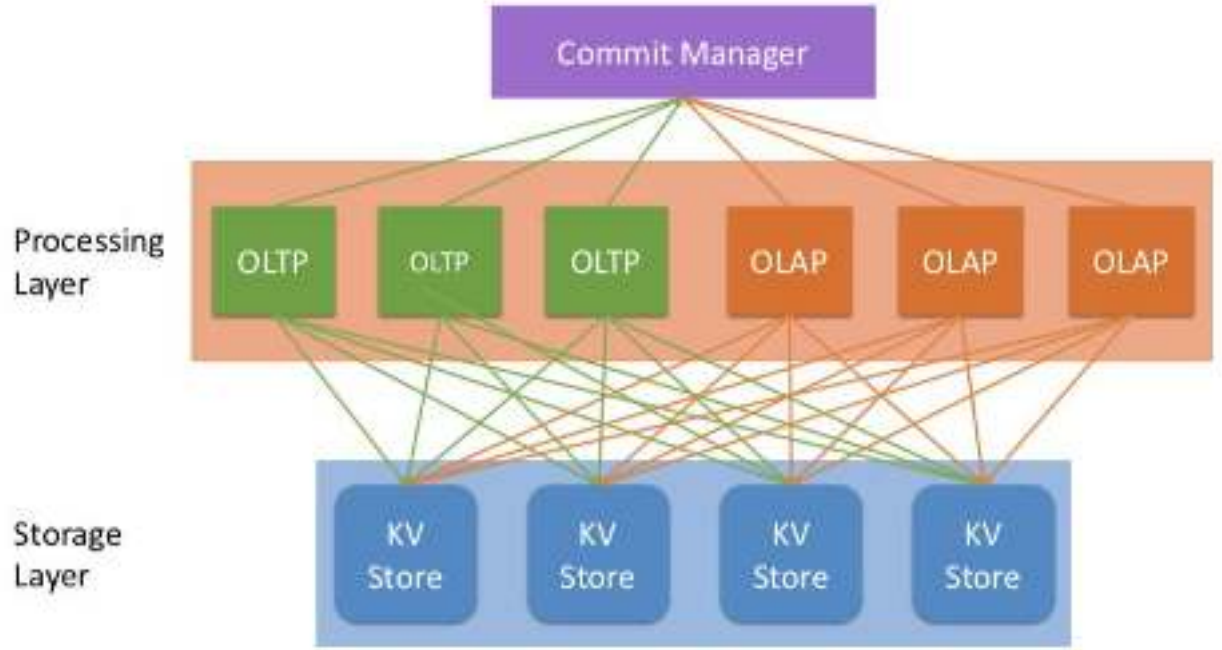

Commit Manager
Processing Layer
OLTP
OLTP
OLTP
OLAP
OLAP
OLAP
Storage Layer
KV Store
KV Store
KV Store
KV Store

Table of Contents

SQL-over-NoSQL Architecture cont.

- What are the distributed KVS requirements to efficiently implement the SQL-over-NoSQL architecture?

1. Scans
2. Versioning
3. Batching and Asynchronous Communication

SQL-over-NoSQL Architecture cont.

- What are the distributed KVS requirements to efficiently implement the SQL-over-NoSQL architecture?

1. Scans
   - KVS must be able to support efficient scan operations in addition to get/put requests
   - Selections, projections and simple aggregates should be supported
     - This allows only the relevant data to be fetched from storage layer and brought up to processing layer
   - Supported shared scans is also a big plus

2. Versioning

3. Batching and Asynchronous Communication

SQL-over-NoSQL Architecture cont.

- What are the distributed KVS requirements to efficiently implement the SQL-over-NoSQL architecture?

1. Scans

2. Versioning
   - Different versions of each record should be maintained in order to return the right version depending on the timestamp of the transaction
   - This is important of Multi-Version Concurrency Control
   - Garbage collection should be able to reclaim storage occupied by old record versions

3. Batching and Asynchronous Communication

SQL-over-NoSQL Architecture cont.

- What are the distributed KVS requirements to efficiently implement the SQL-over-NoSQL architecture?

1. Scans

2. Versioning

3. Batching and Asynchronous Communication
   - OLTP processing nodes should be able to batch several requests to the storage layer
   - Cost of round-trip messages from processing to storage layer can be **amortized** for multiple concurrent transactions
   - Batch requests should also be executed asynchronously

Difficulties of Building the Optimal KVS

- The conditions for the three requirements conflict one another
- As a result, more KVS today, with the exception of Kudu, are specifically designed for get/put requests
- There are many **locality conflicts** that surface when we try to integrate scans into these systems

Locality Conflicts

SCAN vs..

1. Get/Put
2. Versioning
3. Batching

Locality Conflicts

SCAN vs..

1. Get/Put
   - Systems designed for analytical queries normally use columnar layouts to increase locality
     - This is so queries can access the same set of memory locations repetitively over a short period of time
   - Existing KVS typically favor row-store layouts for processing get/put requests without the need to materialize records
2. Versioning
3. Batching

Locality Conflicts

SCAN vs..

1. Get/Put

2. Versioning
   - The presence of irrelevant versions of records reduces locality and slows down scan operations
   - Checking the relevance of a record version as part of a scan can be very costly
3. Batching

Locality Conflicts

SCAN vs..

1. Get/Put

2. Versioning

3. Batching
   - It is not ideal to batch scan operations with get/put requests
   - OLTP workloads require constant and predictable get/put response times
     - Scan operations are highly variable in terms of latency, depending on the selectivity of the predicates and the number of columns that needs processing during a query

Table of Contents

Design Space

The main decisions when designing a storage engine:

- How to **store** data?
- How to **access** data?
- How to **update** data?

Design Space

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Design Space - how to read

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Design Space - how to read

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Design Space - how to read

Measurements:

storage - how efficient is the storage in terms of fragmentation

concurrency - how well is it supported

versioning - how costly is it to implement (performance wise)

GC - how costly is it to implement (performance wise)

get/put - efficiency for get/put operations

scan - efficiency for scan operations

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Garbage Collection

A form of memory management

“Garbage” - useless/obsolete data that is occupying valuable memory space

Advantages:

- Ease of development
- Mitigates certain bugs and memory leaks

Disadvantages:

- Impacts performance

Why do we need to consider GC when designing a key-value store (why can’t we just manually manage our memory)?

Design Space - how to read

Measurements:

storage - how efficient is the storage in terms of fragmentation

concurrency - how well is it supported

versioning - how costly is it to implement (performance wise)

GC - how costly is it to implement (performance wise)

get/put - efficiency for get/put operations

scan - efficiency for scan operations

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

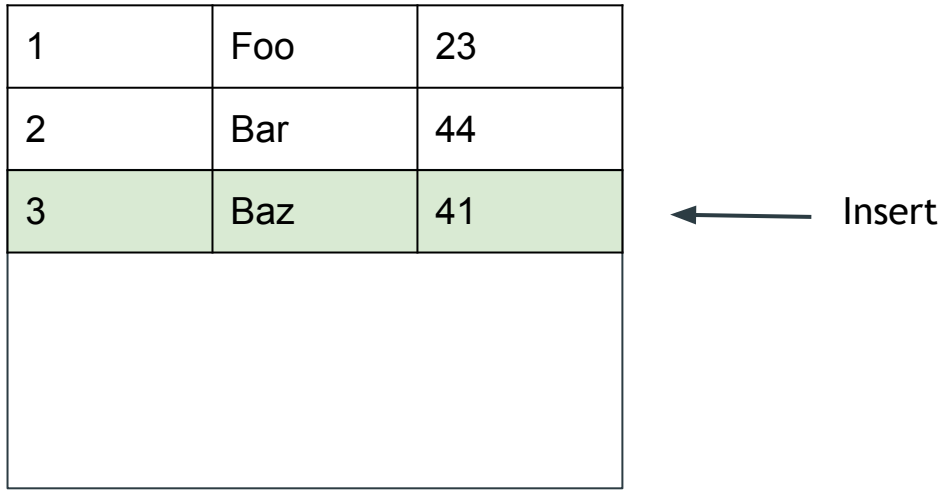
1	Foo	23
2	Bar	44

Page

Records are fixed-size

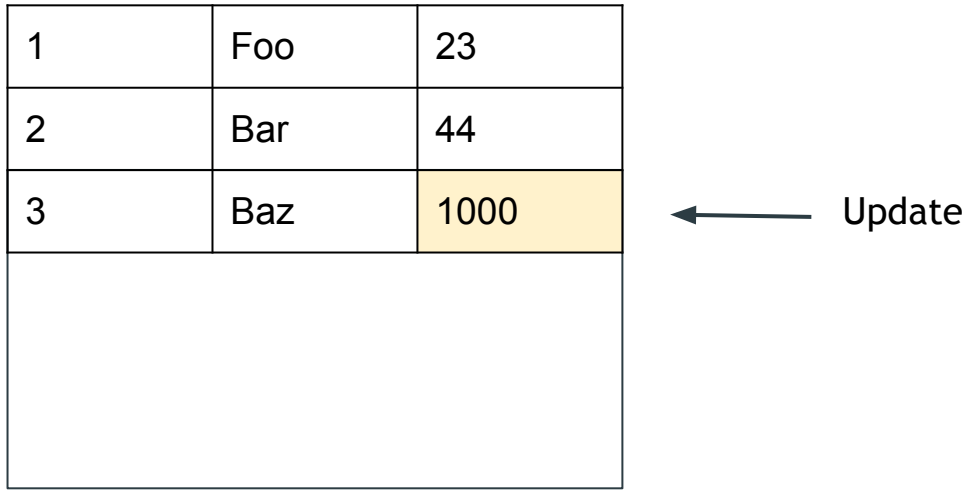
Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	



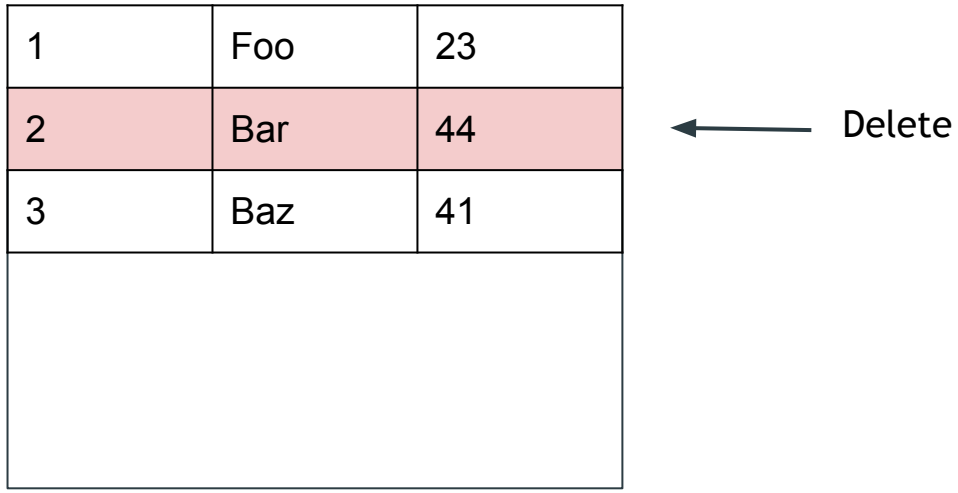
Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	



Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	



Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

1	Foo	23
3	Baz	41

Page

Update: update-in-place

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

1. Why is storage an advantage of update-in-place (assuming fixed record sizes)?
2. Why are versioning and concurrency disadvantages?

Measurements:

storage - how efficient is the storage in terms of fragmentation

concurrency - how well is it supported

versioning - how costly is it to implement

GC - how costly is it to implement

get/put - efficiency for get/put operations

scan - efficiency for scan operations

Update: log-structured

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

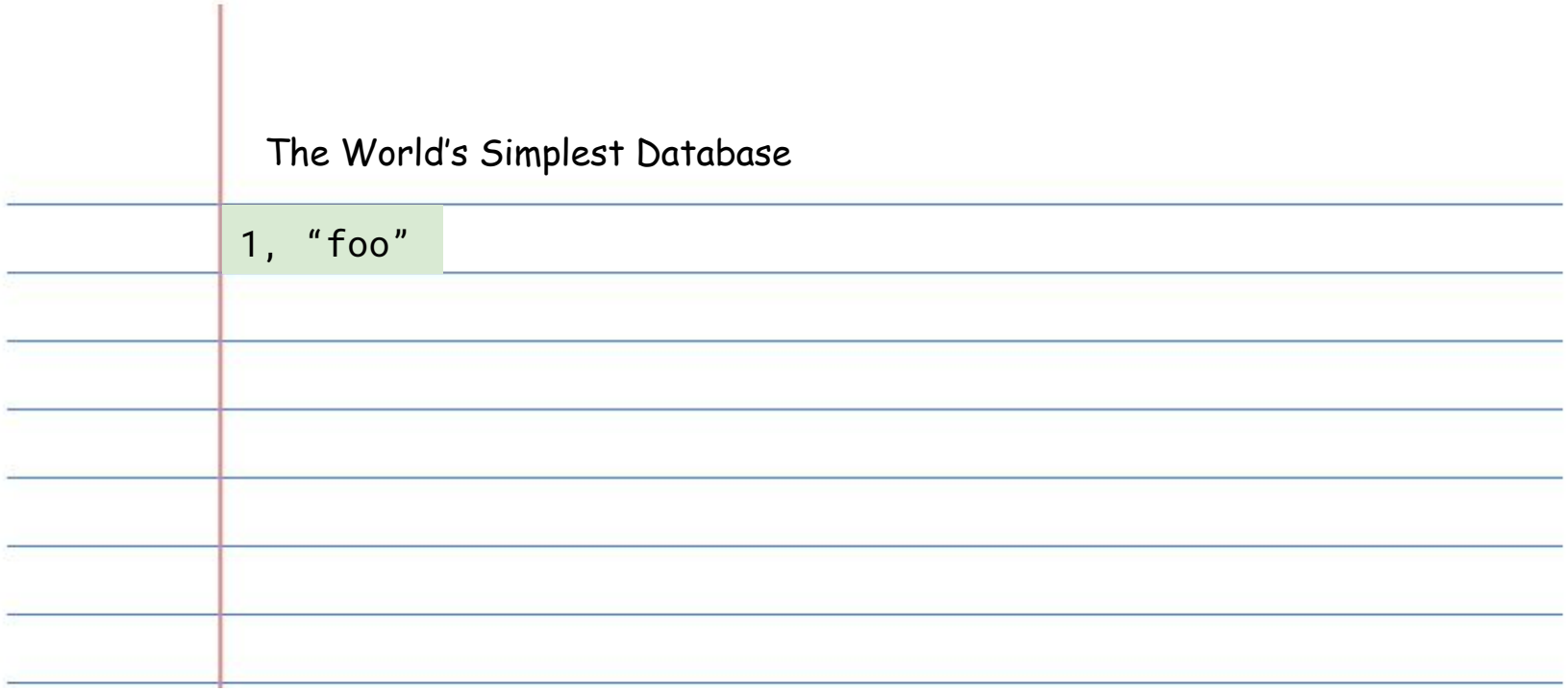
No overwriting necessary, just append to a log.

Introducing

Two methods:
`get(k)`
`put(k,v)`

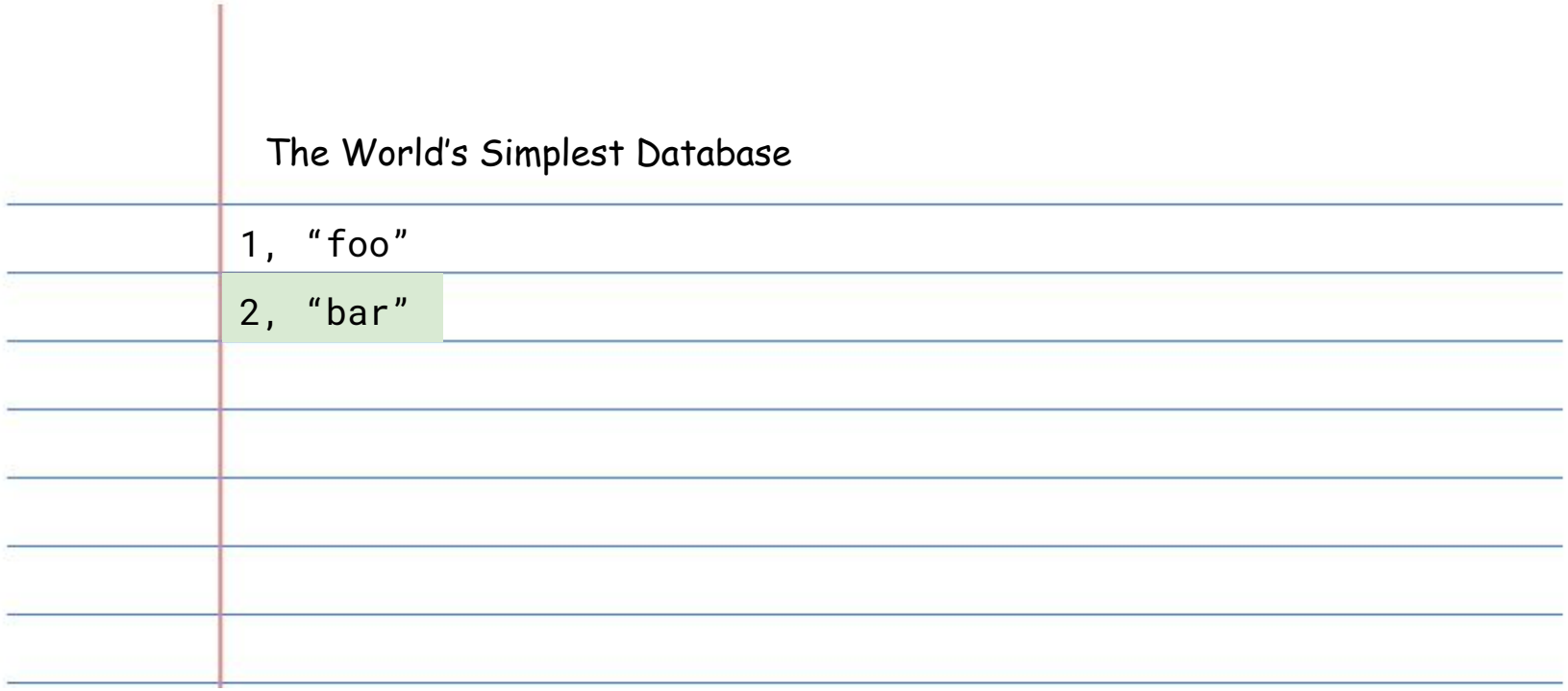
Insert 1

```
put(1, "foo")
```



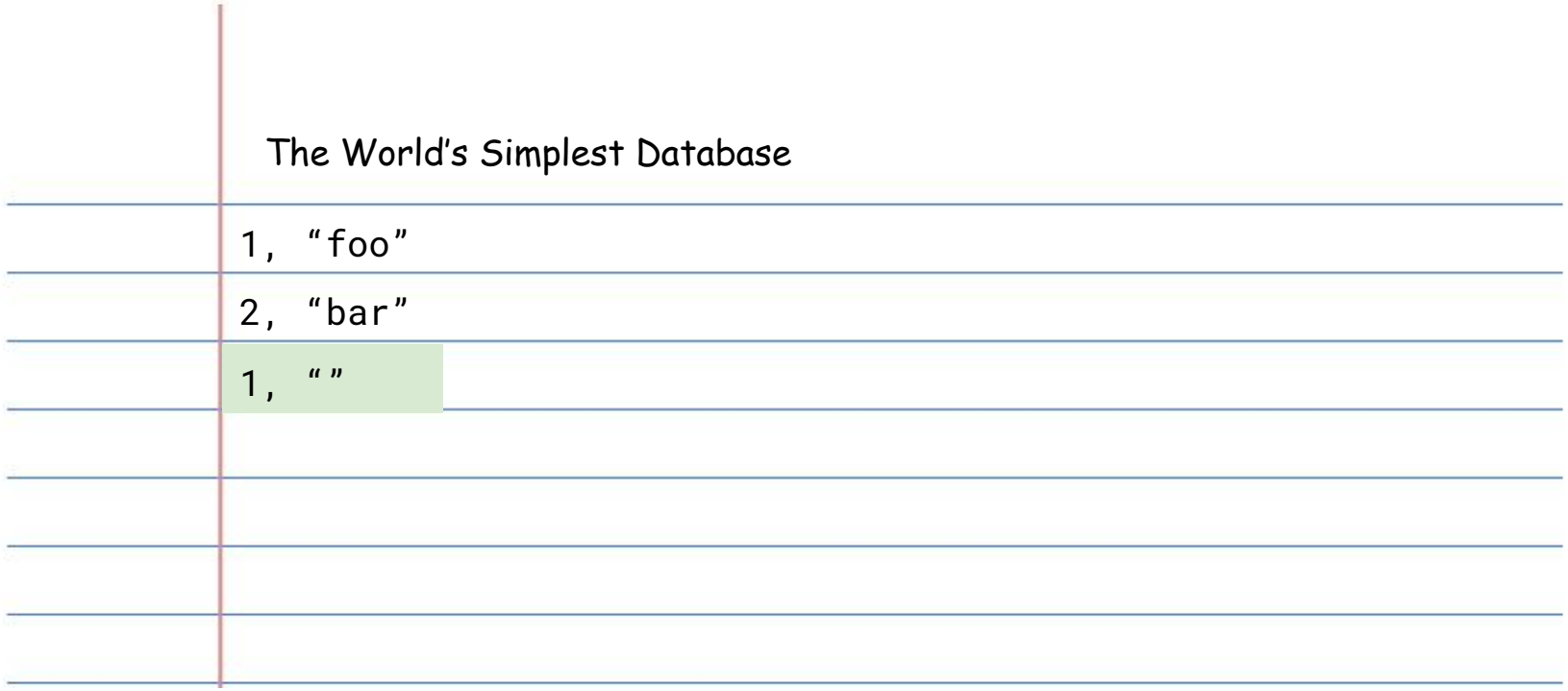
Insert 2

```
put(2, “bar”)
```



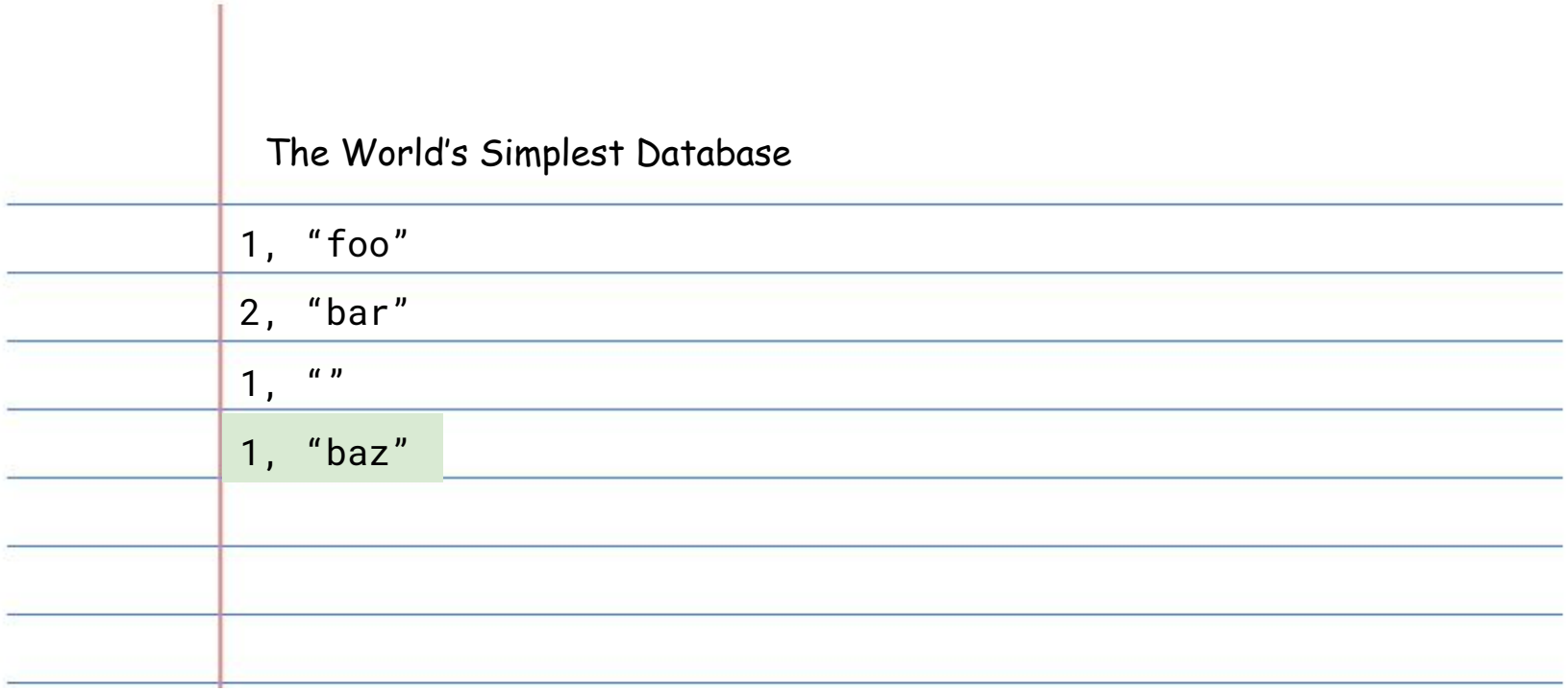
Delete 1

```
put(1, "")
```



Update 1

```
put(1, “baz”)
```



Get 1

```
get(1) =
```

Get 1

```
get(1) =
```

Get 1

```
get(1) =
```

Get 1

```
get(1) = “baz”
```

Update: log-structured

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

Measurements:

storage - how efficient is the storage in terms of fragmentation

concurrency - how well is it supported

versioning - how costly is it to implement

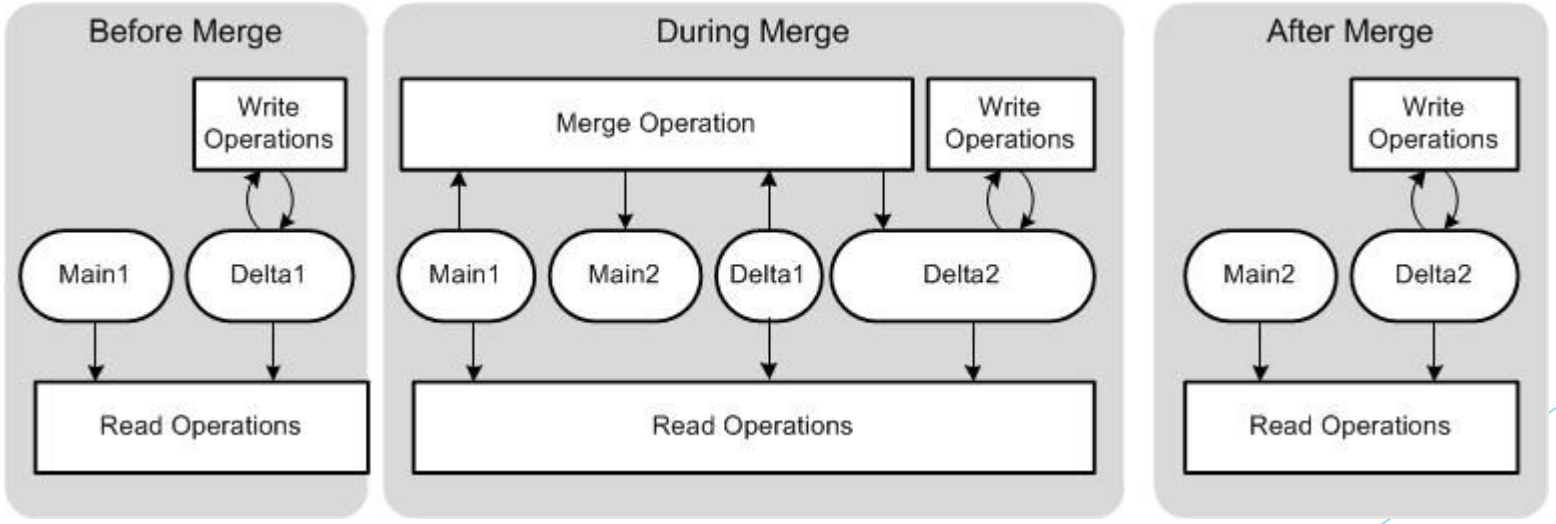
GC - how costly is it to implement

get/put - efficiency for get/put operations

scan - efficiency for scan operations

Update: delta-main

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	



Update: delta-main

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	

This approach tries to combine the advantages of the log-structured approach (fast get/put) with the advantages of update-in-place (fast scans).

Layout

Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

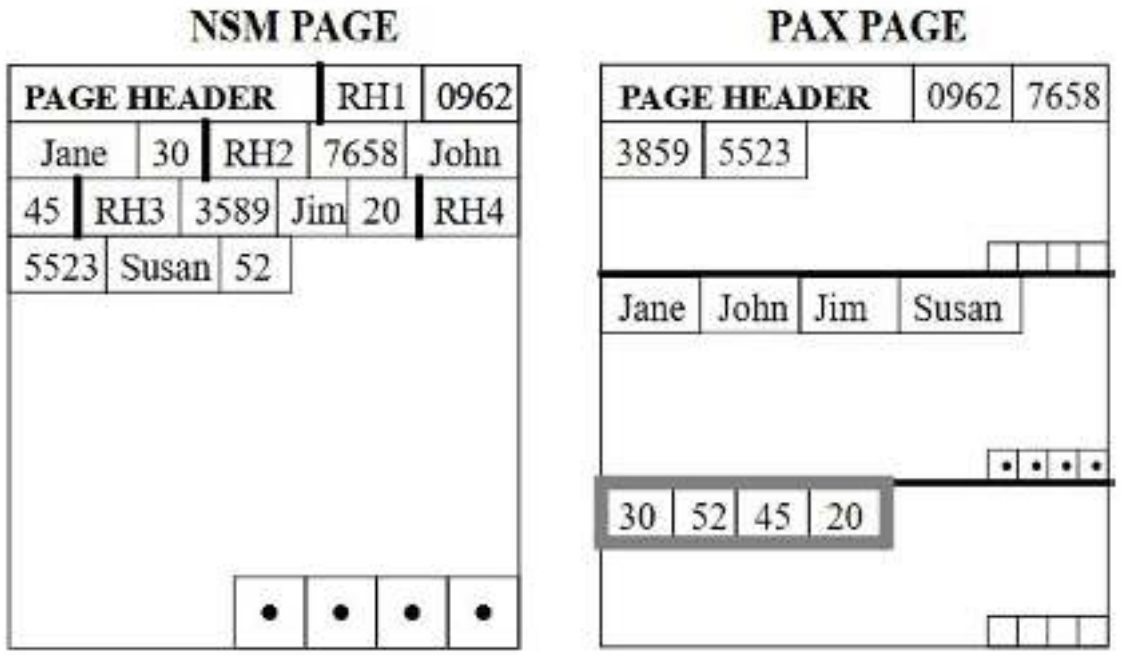
Table 2: Design Tradeoffs

PAX - Partition Attributes Across Paradigm

- Variant of column-major ordering
- PAX is a compromise between the pure column and row-major designs.

PAX - Partition Attributes Across Paradigm

- Stores a set of records in every page, but within the page, all records are stored in a column-major representation.



https://www.pdl.cmu.edu/PDL-FTP/Database/pax.pdf

Versions

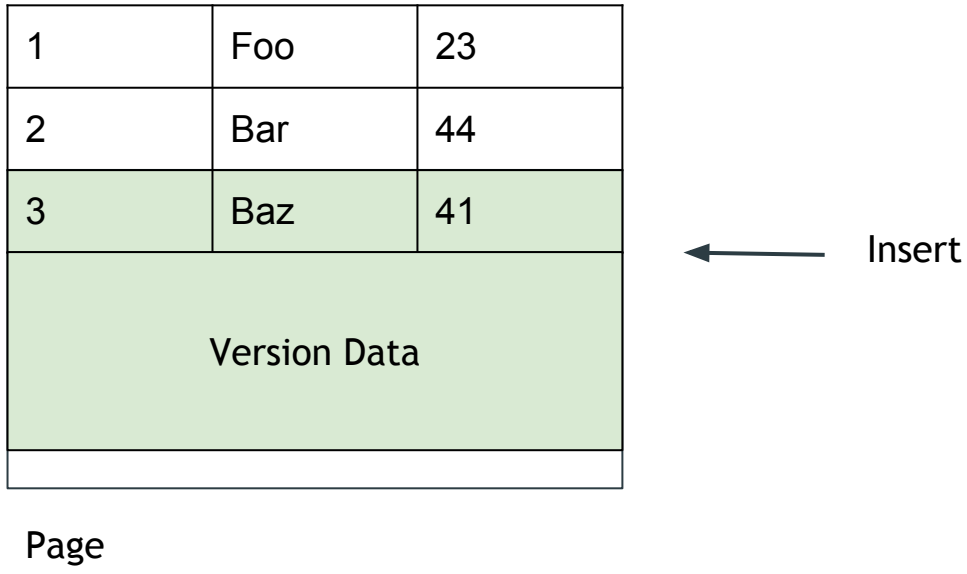
Dimension	*Approach*	*Advantages*	*Disadvantages*
Update	**update-in-place**	storage	versioning, concurrency
	log-structured	storage, concurrency	GC
	delta-main	compromise	
Layout	**column (PAX)**	scan	get/put
	row	get/put	scan
Versions	**clustered**	get/put	GC
	chained	GC	scan

Table 2: Design Tradeoffs

Versions - clustered

Versions	**clustered**	get/put	GC
	chained	GC	scan

- Stores all versions of a record in the same location



Versions - chained

Versions	**clustered**	get/put	GC
	chained	GC	scan

- Chain the versions in a linked list

Garbage Collection - 2 strategies

Periodic garbage collection in a separate dedicated thread/process

Piggy-back garbage collection with (shared) scans

Design Space

(update-in-place vs. log-structured vs. delta-main)
×(row-major vs. column-major / PAX)
×(clustered-versions vs. chained-versions)
×(periodic vs. piggy-backed garbage collection)

Table of Contents

TellStore-Log

Update: Log-structured

Layout: Row-major

Versions: Chained Versions

Garbage collection: Piggy-backed

TellStore-Log

Update: Log-structured

Layout: Row-major

Versions: Chained Versions

Garbage collection: Piggy-backed

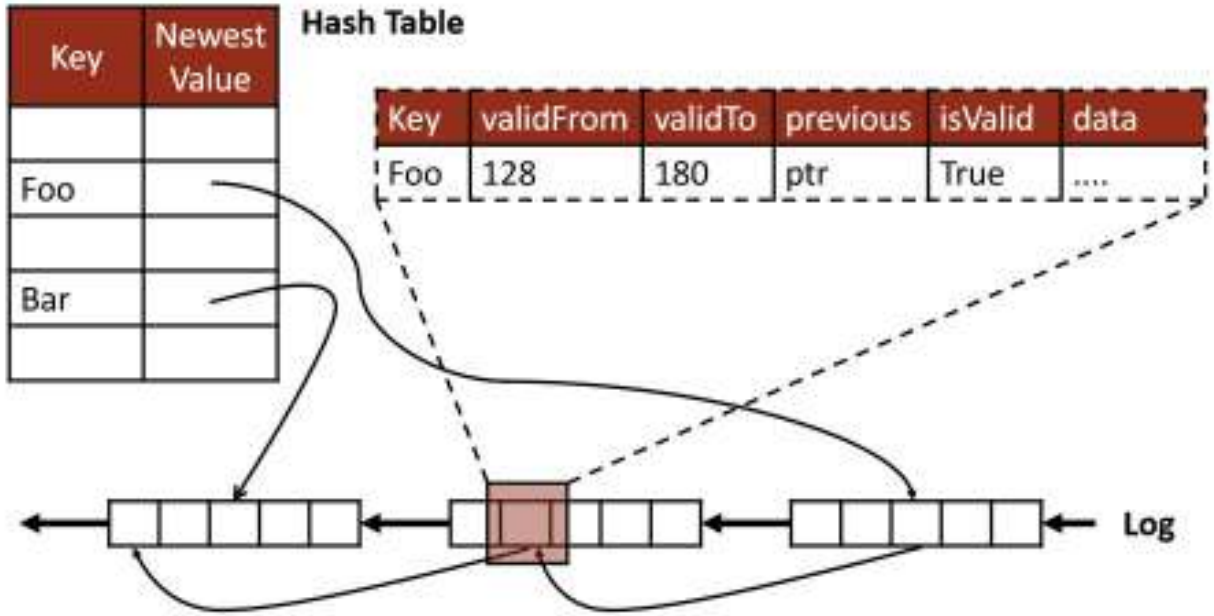


Figure 2: Data Structures of TellStore-Log

TellStore-Log

Questions for discussion:

- Why did the paper use a hash index to implement this log-structured storage (as opposed to eg. a tree index)?
- Why does TellStore-Log have good scan performance compared to other log-structured designs (such as RAMCloud)?

TellStore-Col

Update: Delta-main

Layout: Column-major

Versions: Clustered Versions

Garbage collection: Periodic

TellStore-Col

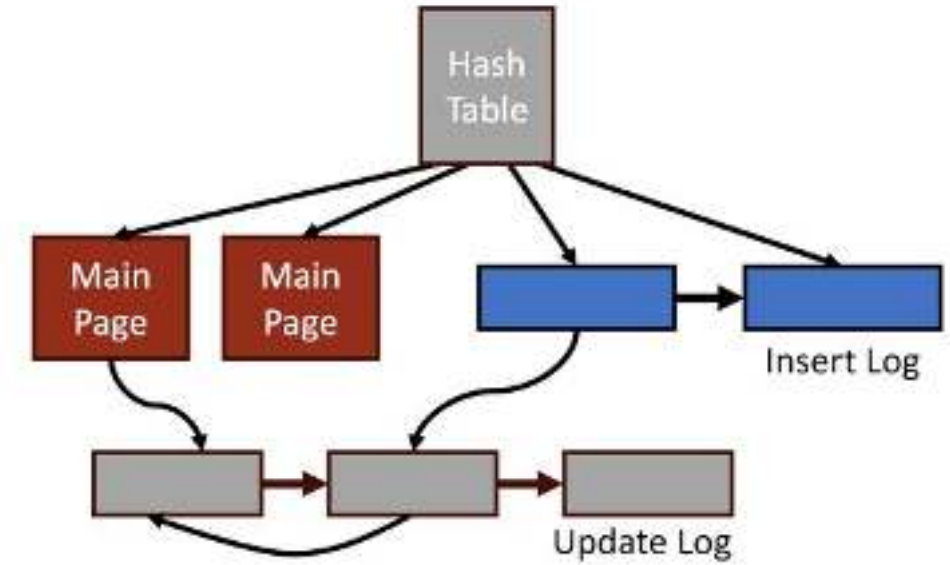


Figure 3: Delta-Main Approach

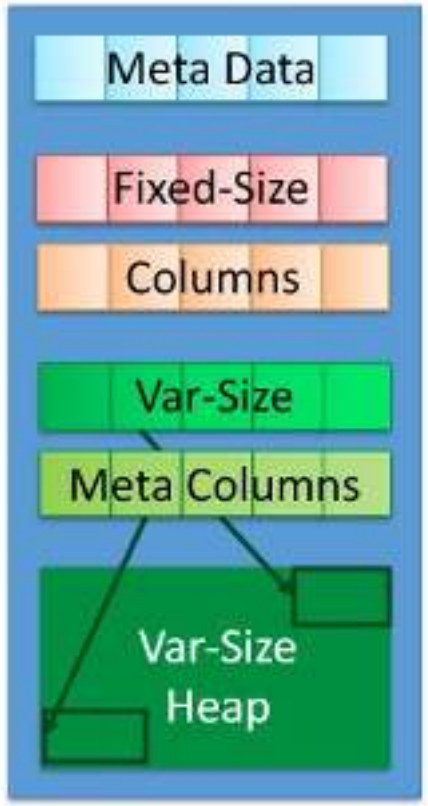


Figure 4: ColumnMap

TellStore-Col

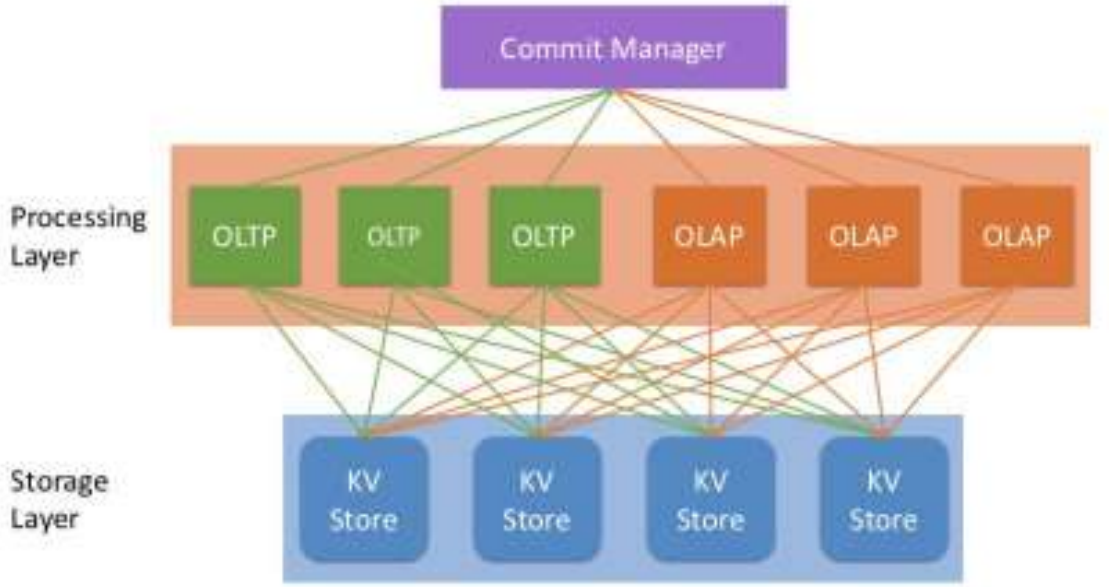
Questions for discussion:

- Why is the “delta” split into two logs - insert log and update log?
  - (Hint: A different log is used depending on whether the key exists or not)
- What are the advantages of TellStore-Col vs TellStore-Log?

Table of Contents

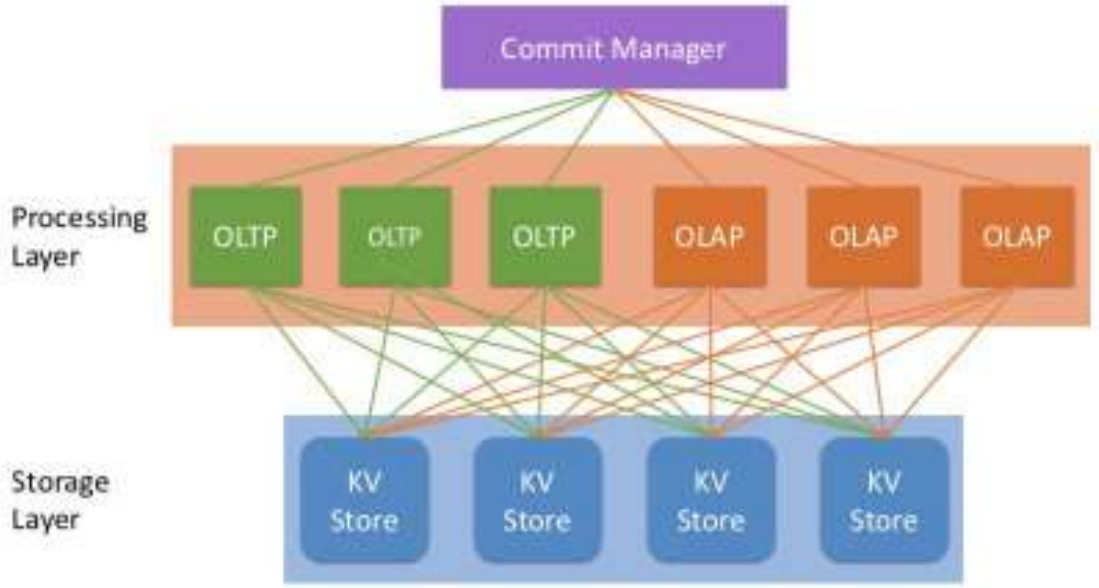
Implementation

- This section describes TellStore as a distributed, in-memory KVS, that can use any of the storage engines described above
  - Communication model
  - Thread model
  - Data indexing
  - Predicate pushdowns



Implementation - Communication Model

- Batching + Async Communication
- A processing instance should not be idle while waiting for a storage request to complete
  - otherwise wasted CPU and network bandwidth



Batching requests from the processing layer and responses from storage cuts down on the messaging rate, which would otherwise be a bottleneck.

Implementation - Thread Model

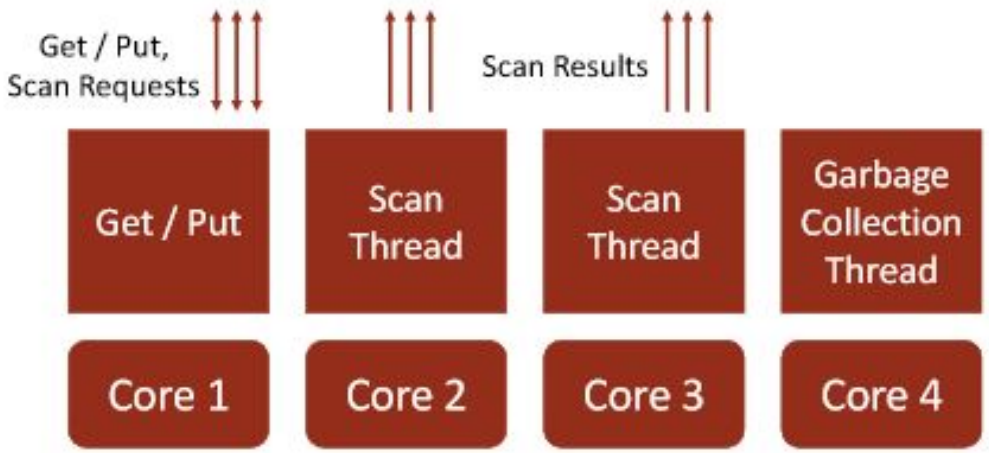
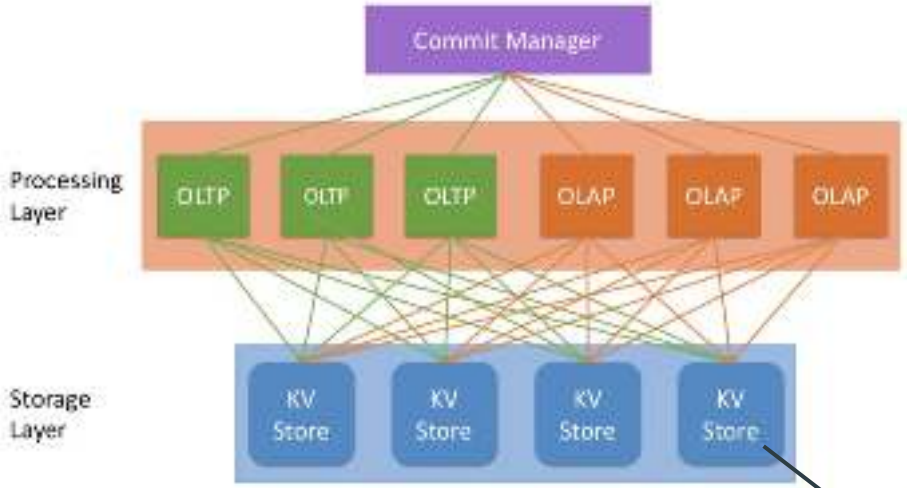


Figure 5: TellStore Thread Model

Why this thread model? In other words, what is the benefit of having separate threads for scans and gets/puts?

Implementation - Data Indexing



What about indexing across KV stores? How do we know which KV Store contains the key we want? Simple if we only have a few instances, but what if we have many KV Stores?

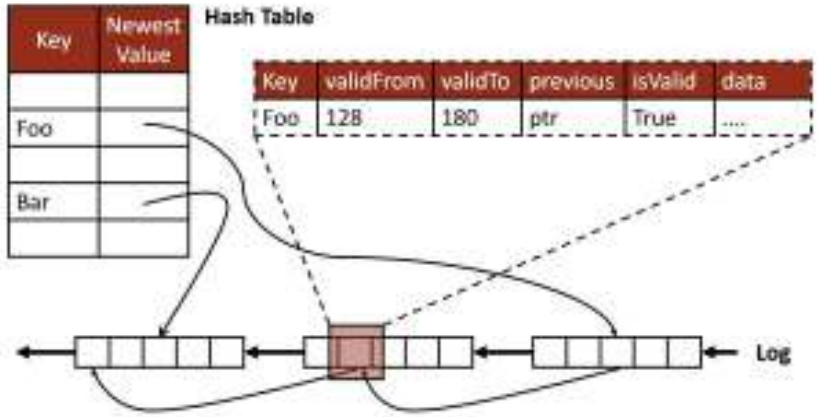
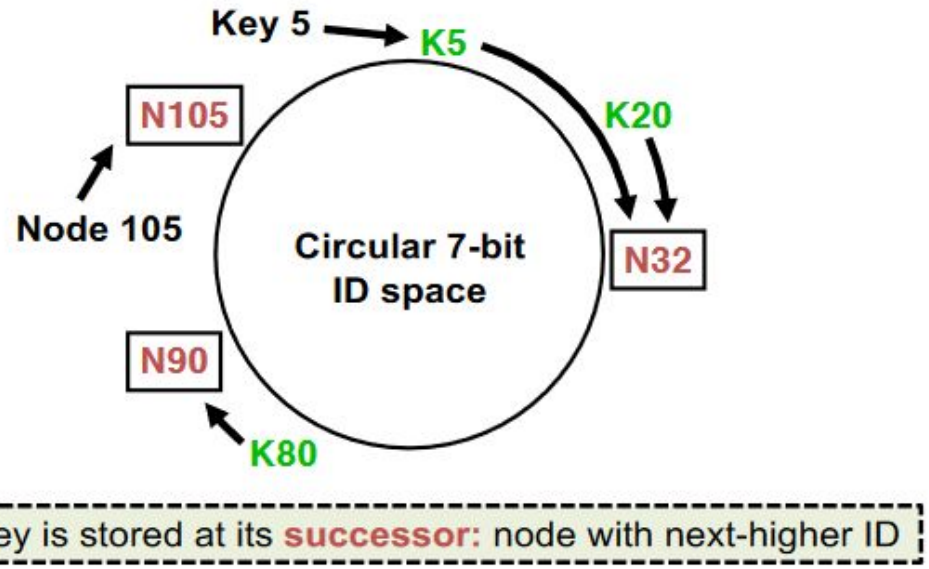


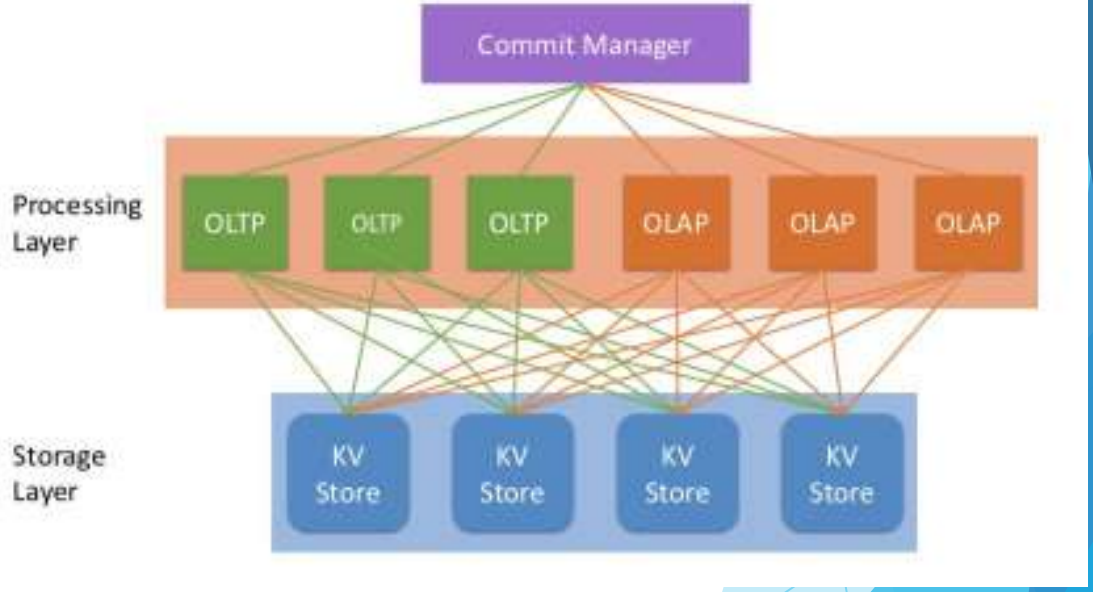
Figure 2: Data Structures of TellStore-Log

Implementation - DHT

Consistent hashing [Karger '97]



Key is stored at its **successor:** node with next-higher ID



Source: https://www.cs.princeton.edu/courses/archive/fall16/cos418/docs/L10-dhts.pdf

Implementation - Predicate Pushdowns

Pushing down predicates allows for less data movement. Imagine:

```
SELECT * FROM A,B WHERE A.id=B.id AND A.name="John";
```

Implementation - Predicate Pushdowns

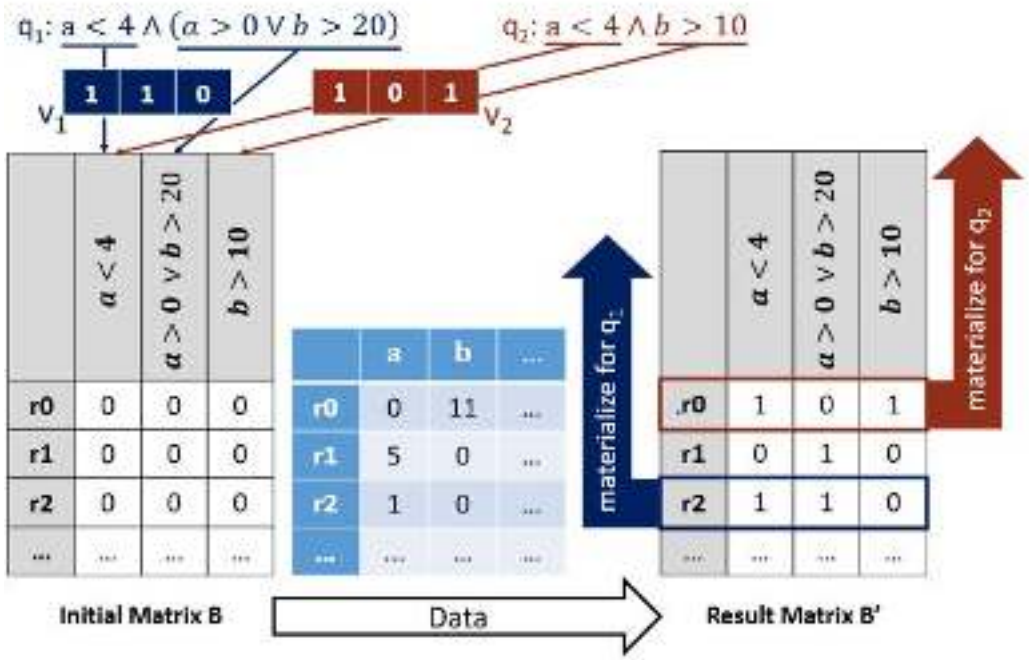


Figure 6: Predicate Evaluation and Result Materialization

- TellStore requires all selection predicates of scans to be in conjunctive normal form (CNF).
- CNF - a conjunction of one or more clauses, where a clause is a disjunction of literals; otherwise put, it is an AND of ORs.

$$(A \vee B) \wedge C$$

$$(A \wedge B) \vee C$$

Table of Contents

Experimental Setup

- Test environment
  - All experiments were done on a cluster of 12 machines
  - All KVS benchmarked are Non-uniform Memory Access (NUMA) unaware
  - 3x the number of processing nodes as storage nodes were used to prevent load generation from becoming the bottleneck

- TellStore-Col was the main test subject against all other KVS
- Kudu performances were used as a baseline for all the experiments

- Running the experiment
  - First populate the data, then run the experiment for 7 minutes total
  - First and last minute were factored out due to warm-up and cool-down effects

Experimental Benchmarks

1. Yahoo! Cloud Serving Benchmark (YCSB)
   - Commonly referred to as “Big Data” Benchmark
   - The YCSB benchmark was the designated goal performance for cloud data serving systems, in particular for transaction-processing workloads

2. Extended YCSB → YCSB#
   - Specially crafted to test additional capabilities of a KVS
   - Schema was extended to include variable-size columns and 3 new queries that involve scans
   - Scaling factor dictating the number of tuples in the database was set to 50
     - Corresponds to a test set of 50 million tuples

Side note about NUMA-unaware

When you have multiple physical cores, each core might have their own cache rather than a shared pool of memory and it’s faster to access each core’s local cache rather than the shared memory bus.
An NUMA-Unaware system simply means that the system does not take this access times into account.

YCSB# Queries

Query 1: A simple aggregation on the first floating point column to calculate the maximum value:

```
SELECT max(B) FROM main_table
```

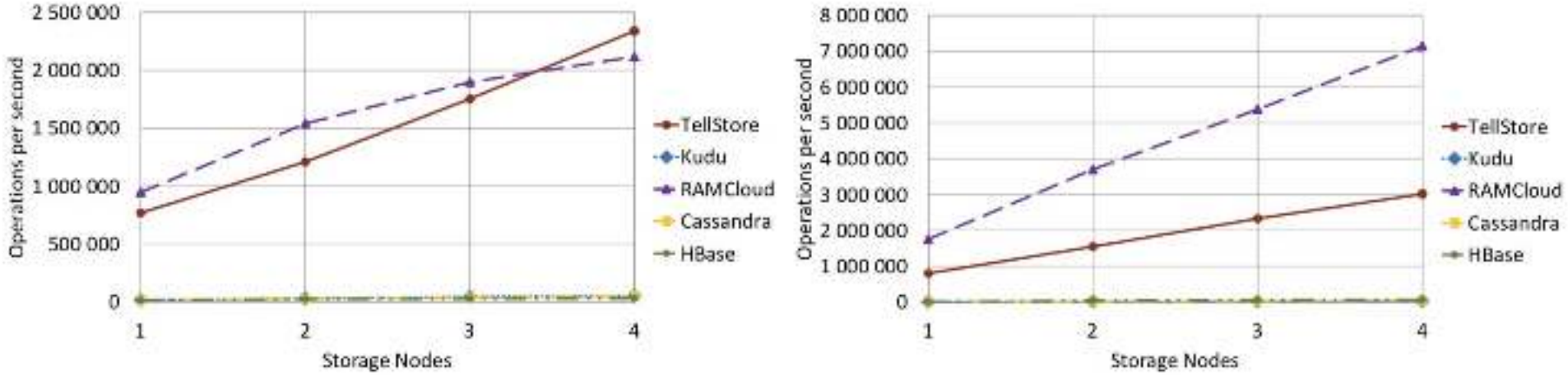
Query 2: The same aggregation as Query 1, but with an additional selection on a second floating point column and selectivity of about 50%:

```
SELECT max(B) FROM main_table
WHERE H > 0 and H < 0.5
```

Query 3: A selection with approximately 10% selectivity:

```
SELECT * FROM main_table
WHERE F > 0 and F < 26
```

Experiment 1: Get/Put Workload



(a) Get/Put Workload

(b) Get-only workload

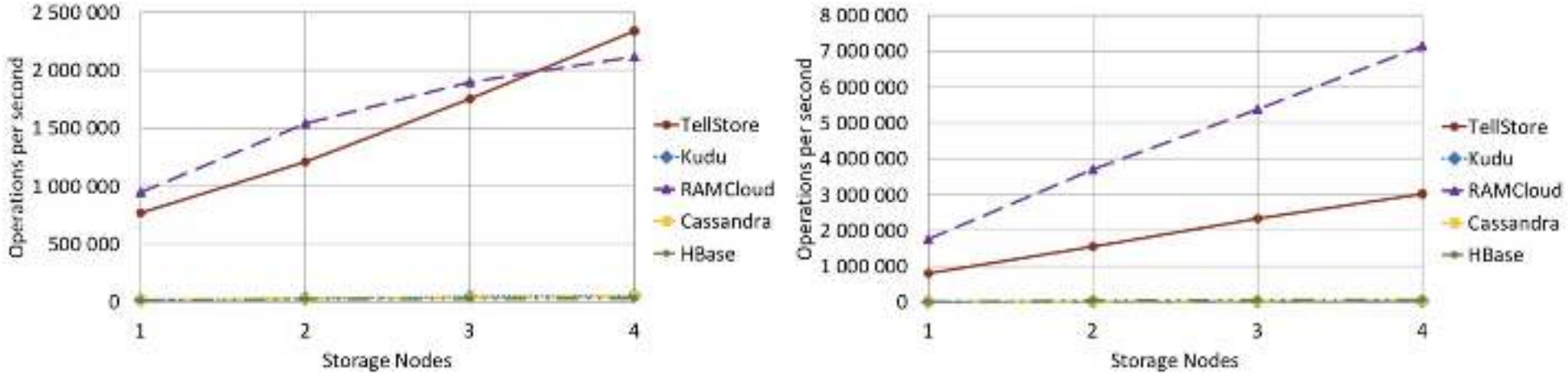
Figure 7: Exp 1, Throughput: YCSB, Various KVS, Vary Storage Nodes

- Tested with traditional YCSB benchmark, no bulk queries
- Get/Put contains 50% update requests consisting of ⅓ inserts, ⅓ updates and ⅓ deletes
- Database size is kept constant with equal amounts of inserts and deletes

Results:

- TellStore performs much better than all others EXCEPT RAMCloud
  - RAMCloud is a distributed in-memory KVS that's highly tuned and specifically designed for these kinds of queries (get/put)
  - RAMCloud is normally seen as the upper bound of the best performance

Experiment 1: Get/Put Workload



(a) Get/Put Workload

(b) Get-only workload

Figure 7: Exp 1, Throughput: YCSB, Various KVS, Vary Storage Nodes

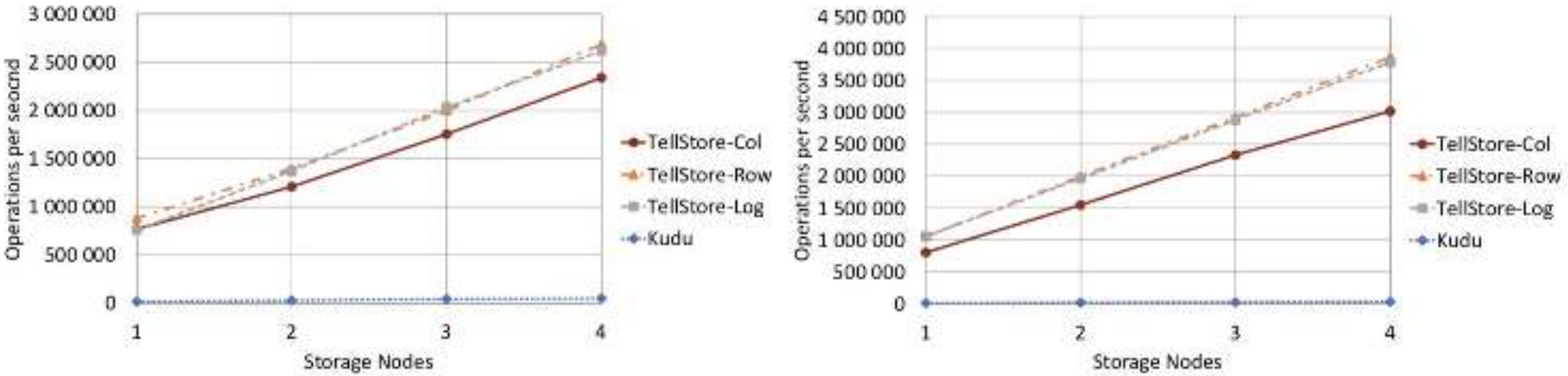
Results:

- Note that all KVS scale linearly with the number of machines
- What does the slope have to do with the system's scalability?

  The steeper the slope, the more "scalable" it is with the number of machines; the more machines you add, the more operations processed

- RAMCloud's scalability issue as shown in (a) might have been due to sub-optimal garbage collection implementation

Experiment 1: Get/Put Workload



(a) Get/Put Workload

(b) Get-only workload

Figure 8: Exp 1, Throughput: YCSB, TellStore Variants and Kudu, Vary Storage Nodes

- TellStore-Log and TellStore-Row outperformed TellStore-Col
- Analysis of TellStore-Col with get/put workload
  - Write-optimized (row-oriented) log in the data helped process Update operations
  - It is more costly to materialize records from columns during Get operations

Experiment 2: Batching

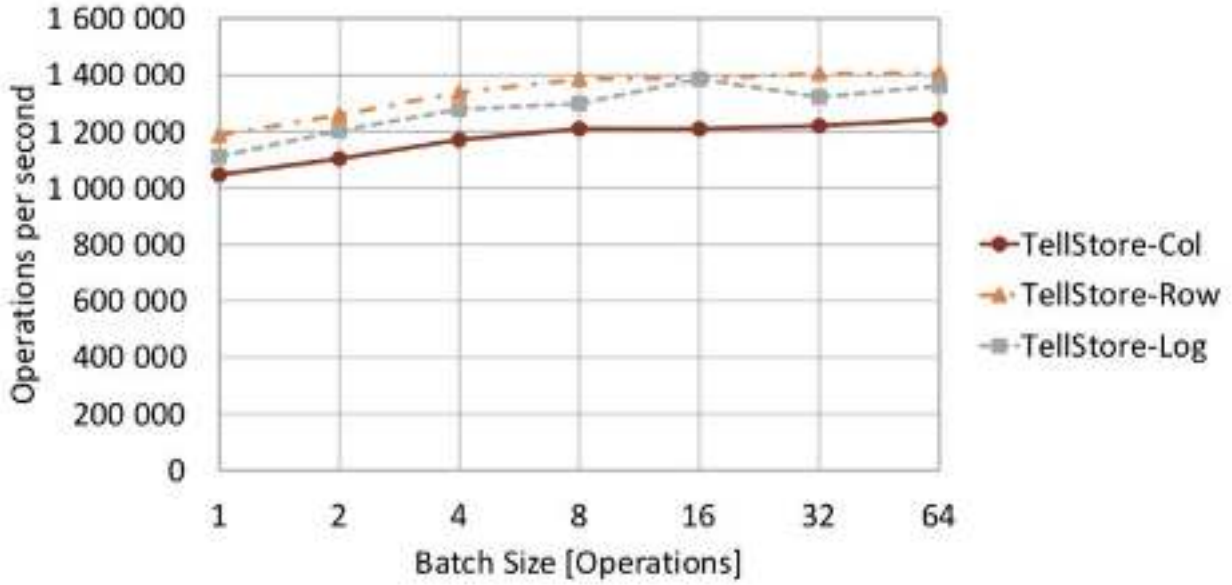


Figure 9: Exp 2, Throughput: YCSB, Vary Batch, 2 Storage Nodes

- As covered in previous sections, processing layer uses batching to improve get/put throughput, but at the cost of increased latency
- The bigger the batch size, the better the throughput for all variants
- Effects however, are not significant enough to say that getting the correct batch size settings is fundamentally important

Experiment 3: Scans

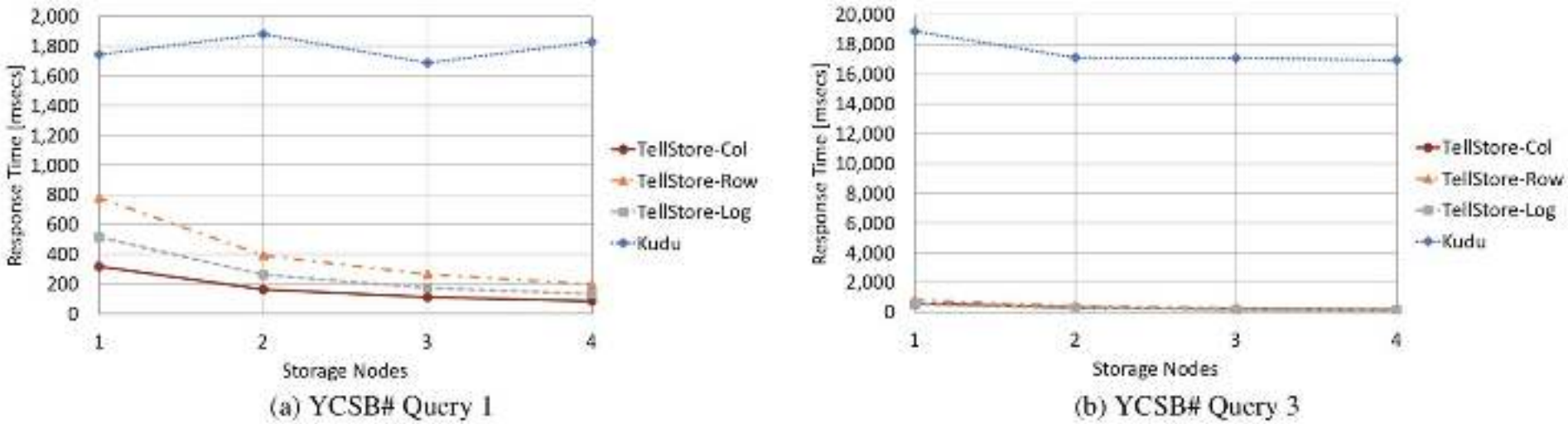
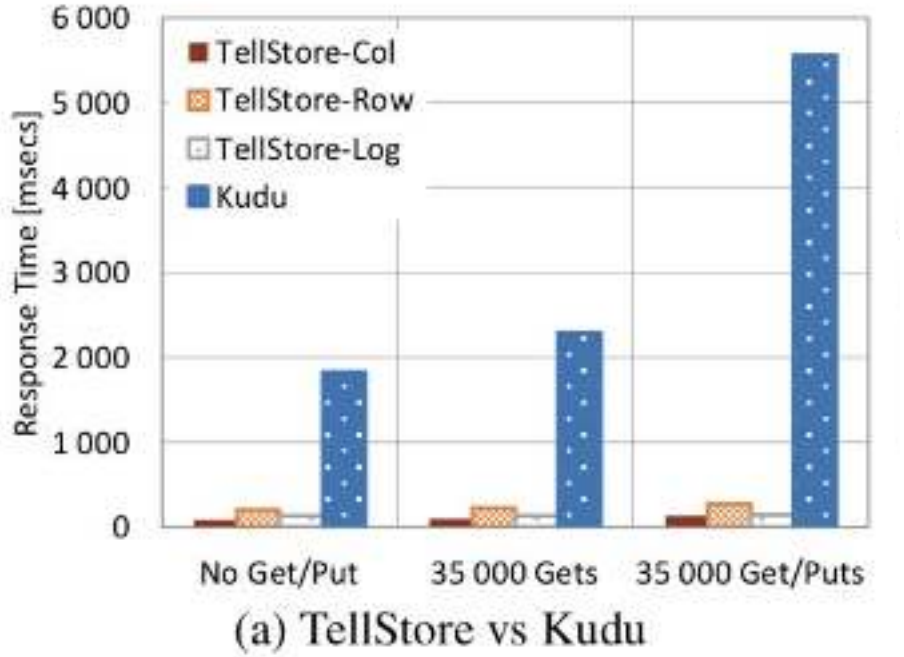


Figure 10: Exp 3, Response Time: YCSB#, Vary Storage Nodes

- YCSB# scan queries were ran in isolation without concurrent get/put workload

- TellStore-Log was slightly faster than TellStore-Col in Query 3
  - This was because Query 3 **has no projections**
  - Read-only workload of Query 3 is the best case for scans in TellStore-Log
    - Scan does not need to perform garbage collection
    - Scan is not affected by data fragmentation

Experiment 4: Hybrid Workloads



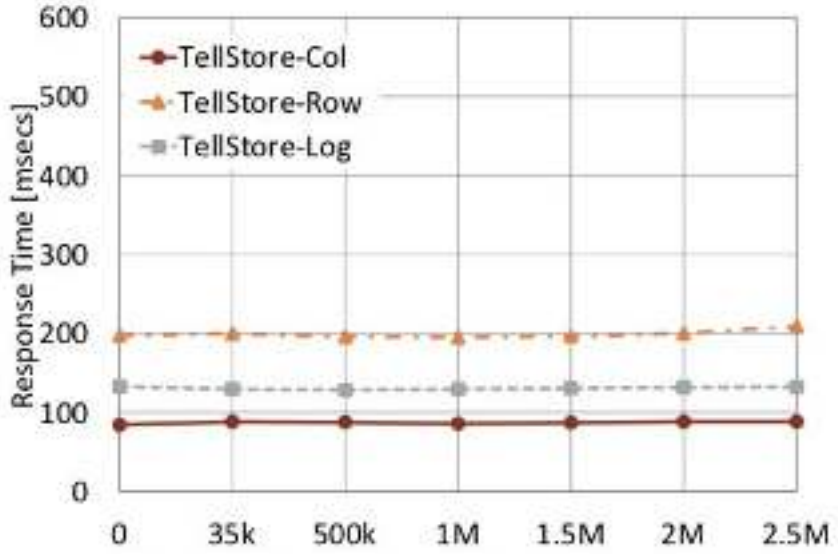
(a) TellStore vs Kudu

- Displays average response time of Query 1 from YCSB#
- Tested on a fixed get/put workload of 35,000 operations per second, because that's Kudu's maximum workload sustainability

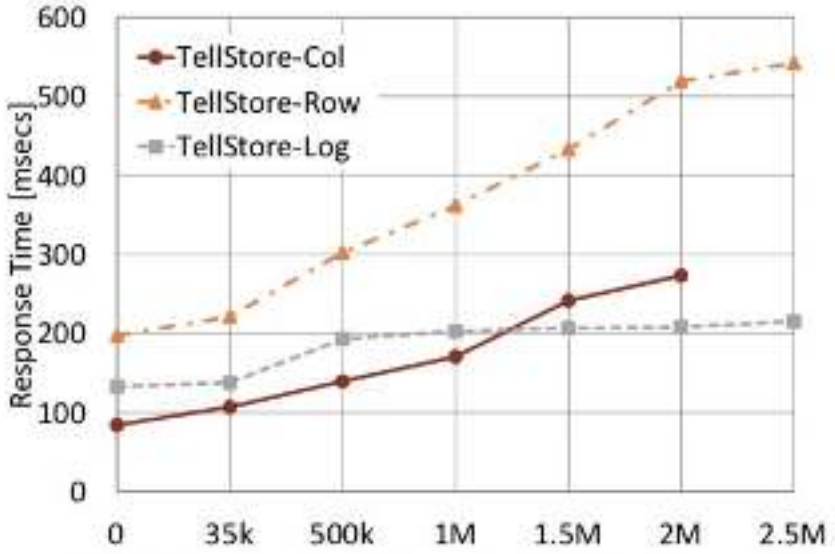
Results:

- Even though Kudu has a columnar design favorable towards Query 1, its overall performance was worse than the TellStore variants
- TellStore-Log does fairly well because in this workload, update requests are kept at a moderate thus fairly high locality was maintained to support scan performance

Experiment 4: Hybrid Workloads



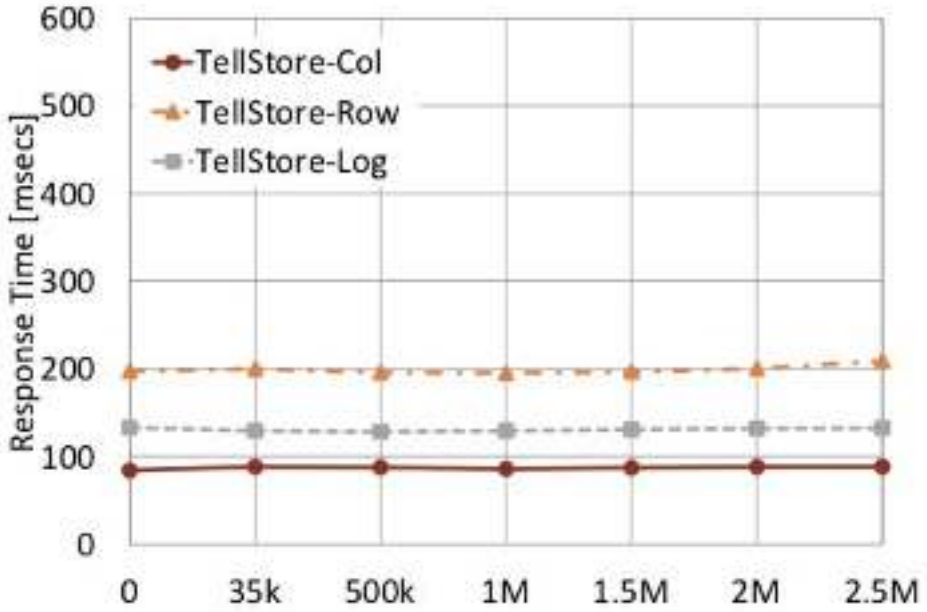
(b) TellStore, Scale Get Requests



(c) TellStore, Scale Get/Put Requests

- Displays response time of Query 1 for the separate TellStore variants when we scale **beyond 35,000 operations per second**

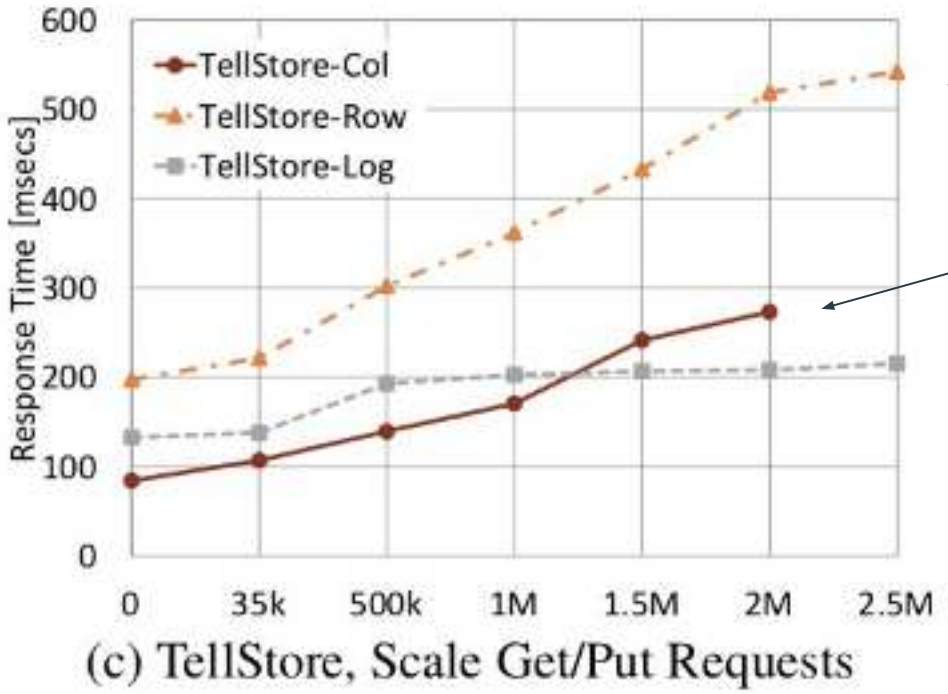
Experiment 4: Hybrid Workloads



(b) TellStore, Scale Get Requests

- Response time for Query 1 is constant and independent of the concurrent get workload, why?
  - Scans and get requests are scheduled on different cores in the storage node
  - Scan performance is not impacted by garbage collection, data stored in delta or stale versions of records

Experiment 4: Hybrid Workloads



(c) TellStore, Scale Get/Put Requests

TellStore-Row and TellStore-Col are heavily impacted by concurrent updates

- Increasing update load results in more costly pointer-chasing to the delta for the look up of the latest version of the record

TellStore-Col also does not have any advantage over TellStore-Row with high update workloads, since almost all records are fetched from the row-store delta

- Scan response time is visibly affected by raising the get/put workload
- TellStore-Log would be the best approach
  - After the sharp increase of the average scan response time at 500,000 concurrent get/put requests per second, garbage collection rewrites the entire log with every scan as every page is affected by an update

Experiment 4: Hybrid Workloads

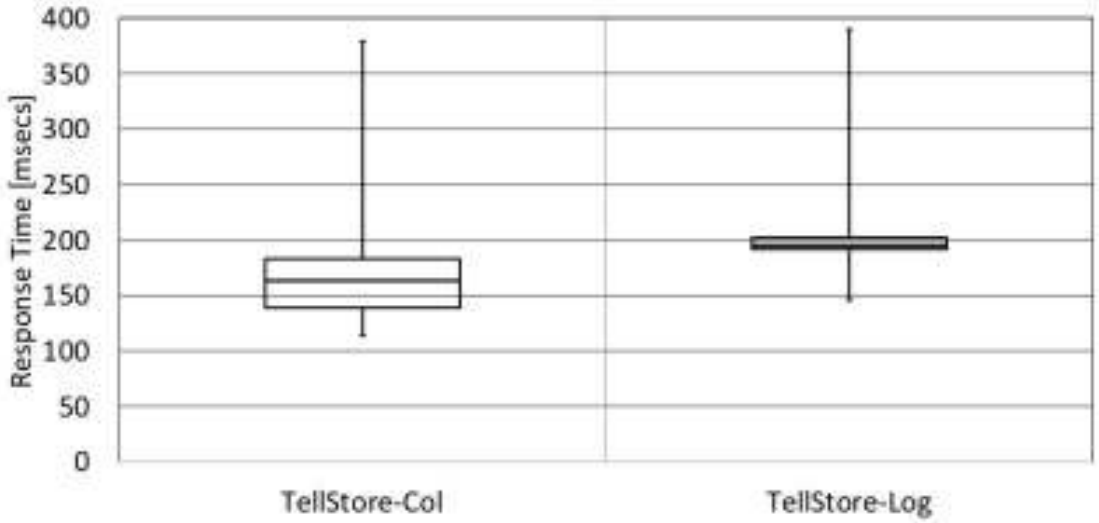


Figure 12: Exp 4, Response Time: YCSB# Query 1
4 Storage Nodes with 1 Mio. concurrent get/put requests

- 1%, 25%, 50%, 75%, and 99% percentile response times of Query 1 with concurrent get/put workload
- Even though TellStore-Col outperforms TellStore-Log at 1 Mio. get/put requests, TellStore-Log is still better overall
  - Scan performance of TellStore-Log is dependent on delta size
    - Before GC, delta size is large so scan is slow
    - After GC, delta size is small so scan is fast
  - Roughly constant response time in TellStore-Log with piggy-backed garbage collection

Experiment 5: Huawei-AIM Benchmark

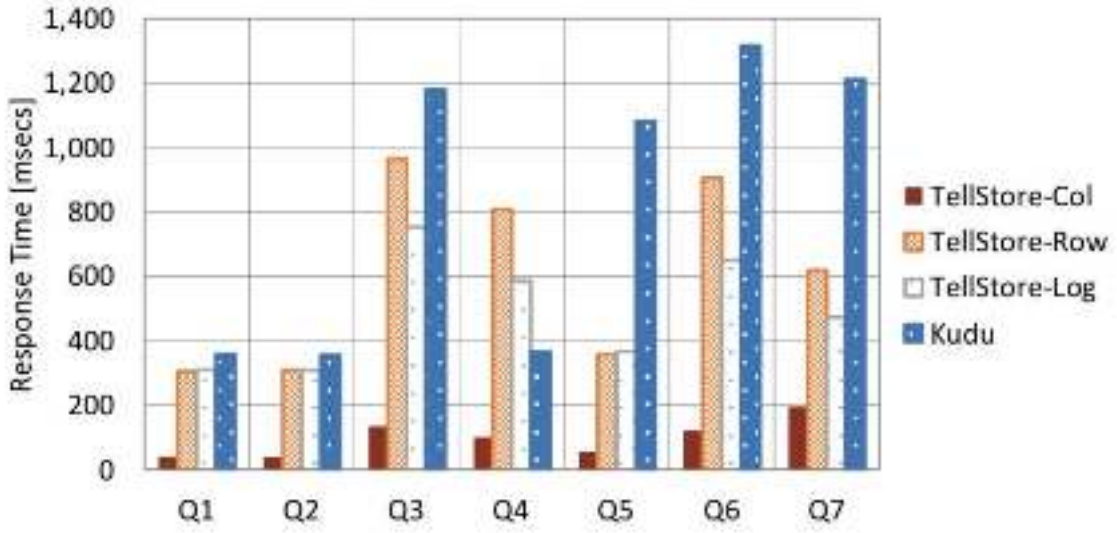


Figure 13: Exp 5, Resp. Time: Huawei-AIM Workload

- This experiments strives to show that TellStore also performs well against more complex, interactive workloads
- Workload is defines 7 analytical queries with concurrent get/put workload of 40,000 operations per second
- There was not much different when scan was ran in isolation, hence these are the results for concurrent workloads

Experiment 5: Huawei-AIM Benchmark

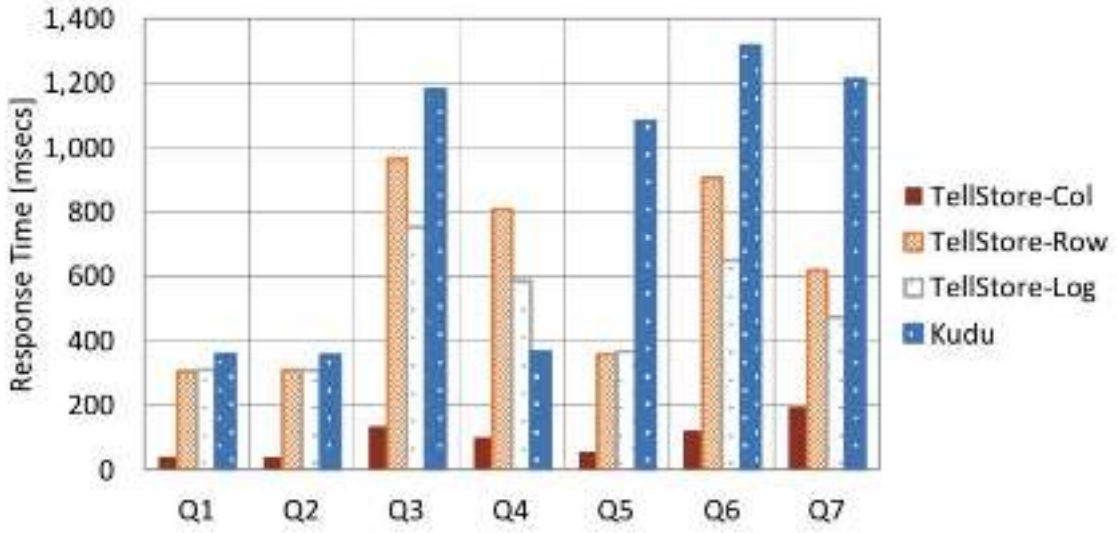


Figure 13: Exp 5, Resp. Time: Huawei-AIM Workload

- TellStore-Col outperformed all others significantly
- Notice that the difference between TellStore-Col and TellStore-Log is the biggest in this experiment, why?
  - This workload was run on a table with more than 500 columns, giving TellStore-Col's columnar layout a bigger advantage

Table of Contents

Related Work

Let's take a look at several other existing KVS:

- Systems that adopted the SQL-over-NoSQL architecture
  - FoundationDB, Hyder, Tell, and AIM
  - FoundationDB, Hyder and Tell were not built to take on analytics; AIM supports read-only analytics and high update rates
- Large-scale Analytical Systems
  - Hadoop, Spark, and DB2/BLUE
  - These systems has significant trouble with or no support for querying live data that is subject to frequent and fine-grained updates
- Systems with good support for hybrid workloads on live data
  - HyPer, HANA, and Hekaton/Apollo
  - TellStore differs from these with its ability to scale out in a distributed system, while the above can only scale up on a single machine
- Document Stores
  - DocumentDB and MongoDB
  - These offer certain scans with secondary indexes which are specifically tuned to document-related use-cases
- TellStore can hopefully be used as a model to further improve these existing KVS systems to achieve a better overall performance

Table of Contents

Conclusion

- It is indeed possible to build a KVS that efficiently supports both **scans for analytics** and **high get/put throughput for OLTP workloads** with TellStore

- The integrated design of TellStore addressed important design questions of a KVS, allowing it to outperform existing ones
- Advanced implementation techniques of TellStore (e.g., piggy-backing garbage collection) helps to mitigate the effects of concurrent update on scan times

- Among the three TellStore variants, we can conclude that TellStore-Col has the best overall performance, followed by TellStore-Log, then TellStore-Row

Table of Contents

Future Research and Advancements

- TellStore currently does not feature **high availability with replication**
  - Replication feature in TellStore is currently being implemented
  - With confirmed observation, replication is orthogonal to all other aspects of a KVS and will not change the main results discovered in this paper
- Getting analytical workloads to run efficiently
  - Having fast scans on KVS only partially solves the issue
  - Explore other analytical queries that can only be efficiently executed in a distributed manner
    - Studies against other distributed query processing systems (e.g., Spark and Presto) using TPC-H benchmark queries
    - TellStore does not have an advantage, since neither Spark nor Presto can efficiently utilize TellStore's scan feature of **shared scans** and **pushing down selections, projections and aggregation to the storage layer**
- Studies against less aggressive variants to carry out garbage collection in TellStore-Col and TellStore-Row
  - Current implementation of garbage collection may result in high contention of memory buss and in-write amplification if data does not fit into main memory