CS 561: Data Systems Architectures

BOSTON UNIVERSITY

class 7

Fast Scans on Key-Value Stores

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/

Fast Scans on Key-Value Stores (KVS)

Key-Value Stores are designed for ***transactional*** workloads (put and get operations)

Analytical workloads require efficient scans and aggregations
(typically offered by column-store systems)

Can we do both in one system?

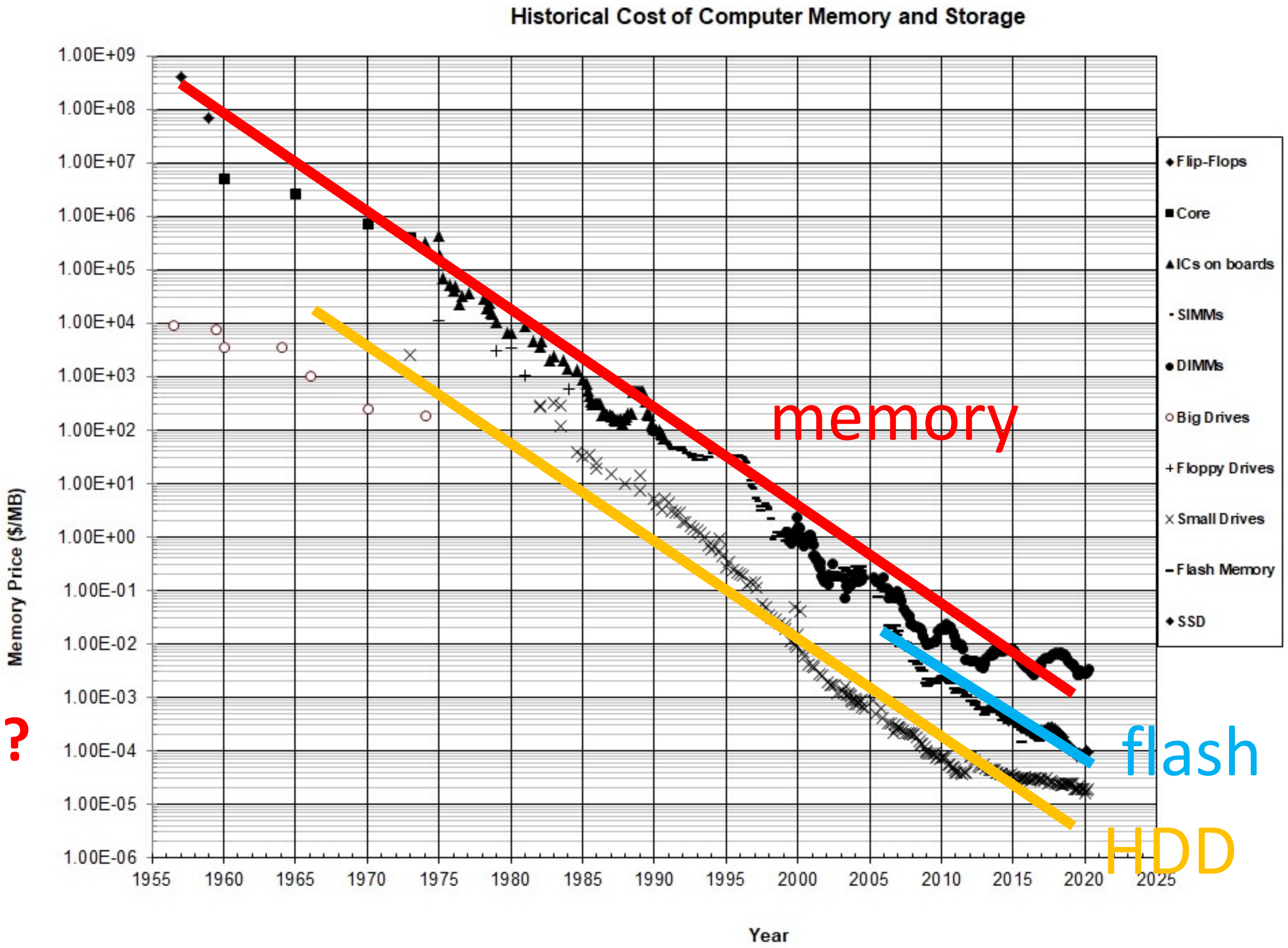
Why combine KVS and analytical systems?

cheaper and cheaper storage

more data ingestion

need for write-optimized data structures

what about analytical queries?



BOSTON UNIVERSITY

Both **transactional** and **analytical** systems

Most organizations maintain both

- ***transactional*** systems (often as key-value stores)
- ***analytical*** systems (often as column-stores)

problems?

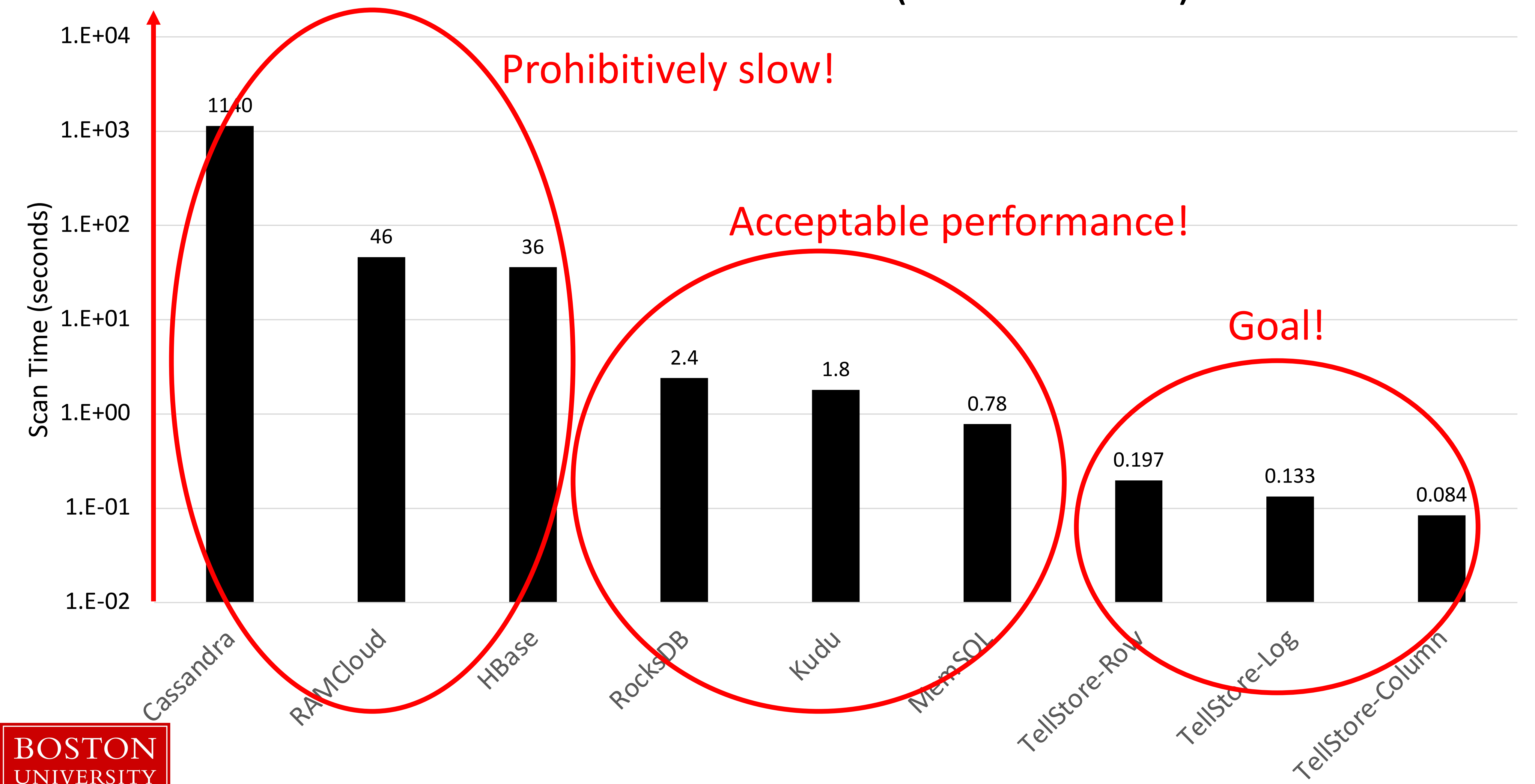
requires additional expertise and management (e.g., two DBAs)

harder to maintain (more systems, more code)

time consuming data integration/transfer

BOSTON UNIVERSITY

Scan Time of 50M records (~4GB of data)



BOSTON UNIVERSITY

Goals of this paper

Bridge the conflicting goals of ***get/put*** and ***scan*** operations

get/put operations need **sparse indexes**

scans require **locality** (relevant data to be packed together)

we will discuss how to compromise, via the design of ***Tellstore***

how to amend the ***SQL-over-NoSQL*** architecture for mixed workloads

BOSTON UNIVERSITY

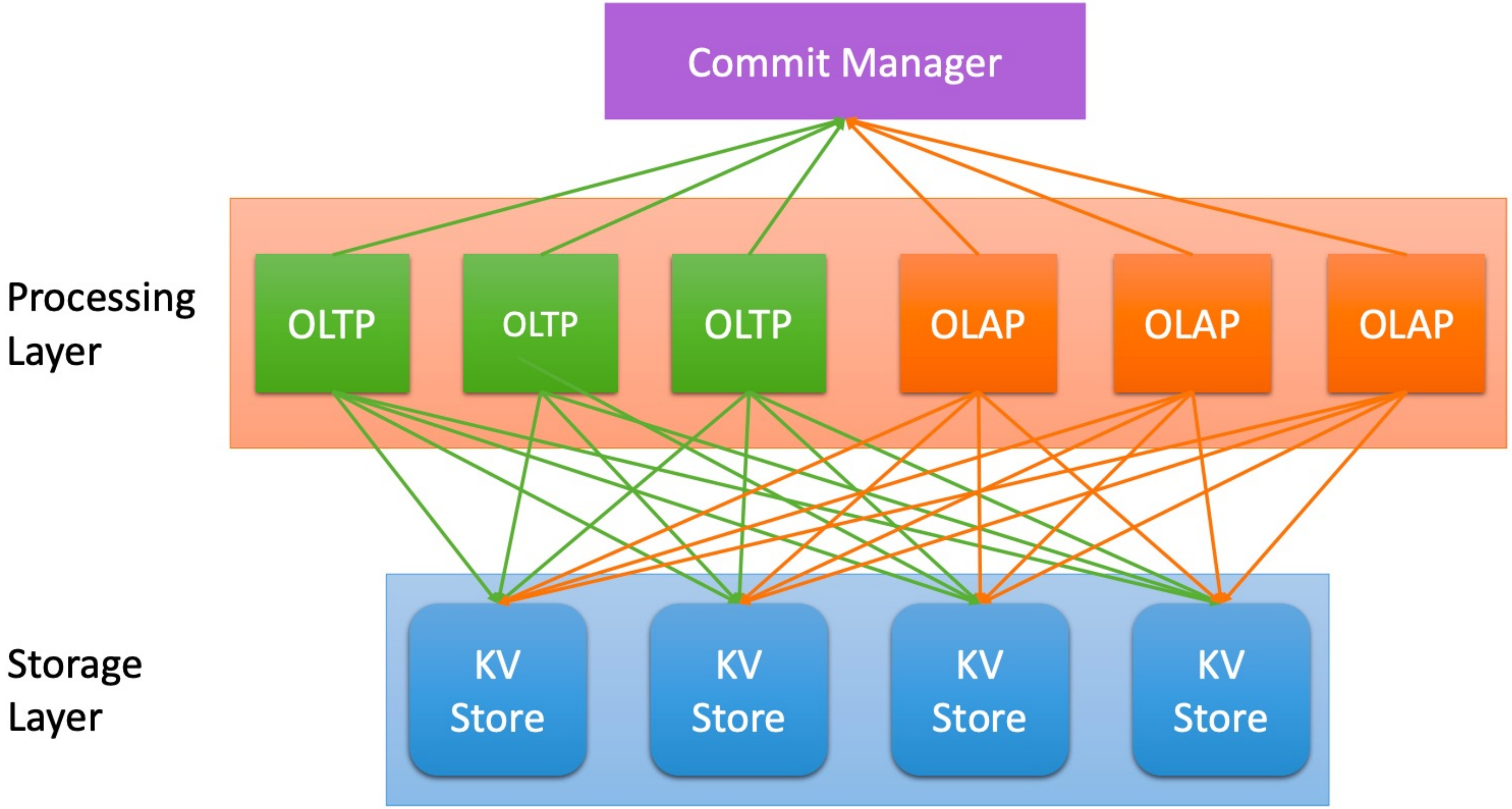
SQL over NoSQL

Elasticity

Snapshot Isolation

Support for:

- Scans
- Versioning
- Batching



BOSTON UNIVERSITY

Scans

selection

projection

(simple) aggregates

shared scans

remember them?

Versioning

multiple versions through timestamps

garbage collection

discarding old versions during scans might be costly

Batching

batch several requests to the storage layer

amortize the network time

BOSTON UNIVERSITY

Challenges

Scans need columnar locality

<u>scans vs. get/put</u>

#1, John, 2/4/88, Boston

2/4/88
2/1/87
7/7/93
4/1/92
3/9/91
9/3/96

get/put need row-wise locality

why?

Challenges

scans vs. get/put

#1, John, 2/4/88, Boston, v1

#1, John, 2/4/88, Cambridge, v2

scans vs. versioning

versioning reduces locality in scans

checking for the latest version in scans needs CPU time

Challenges

<u>scans vs. get/put</u>

<u>scans vs. versioning</u>

<u>scans vs. batching</u>

batching **multiple scans** or **multiple put/get** requests is ok

but …

batching scans and puts/gets is a **bad idea**!

why?

puts/gets need fast predictable performance

scans inherently have high and variable latency

BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

(A) Updates

(B) Layout

(C) Versioning

BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

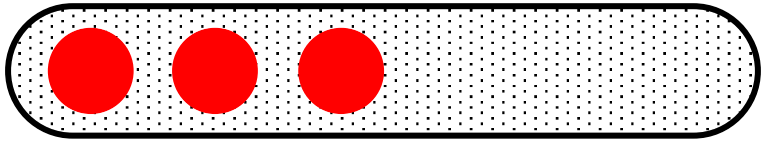
(A) Updates ***in-place***

BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

(A) Updates *in-place* *log-structured*



BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

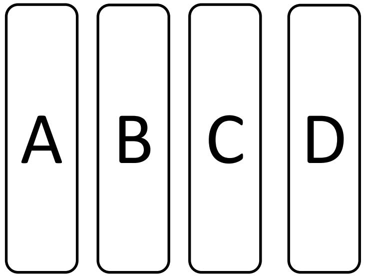
(A) Updates ***in-place*** ***log-structured*** ***delta-main***

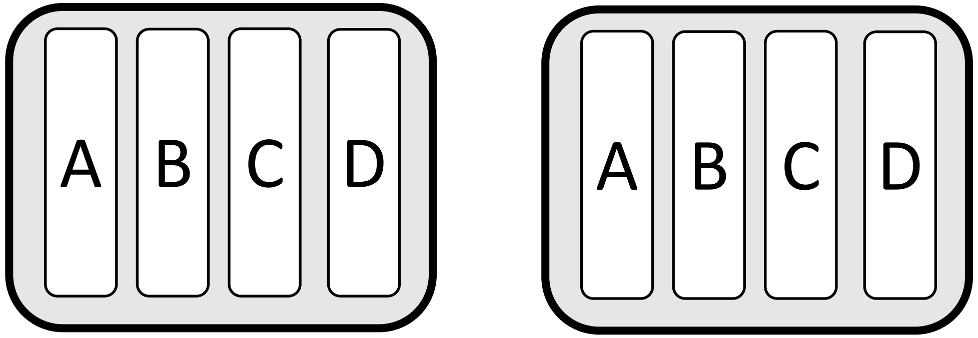
How to design KVS for efficient scans?

Key design decisions

(A) Updates *in-place* *log-structured* *delta-main*

(B) Layout *column* *PAX (columnar per page)*



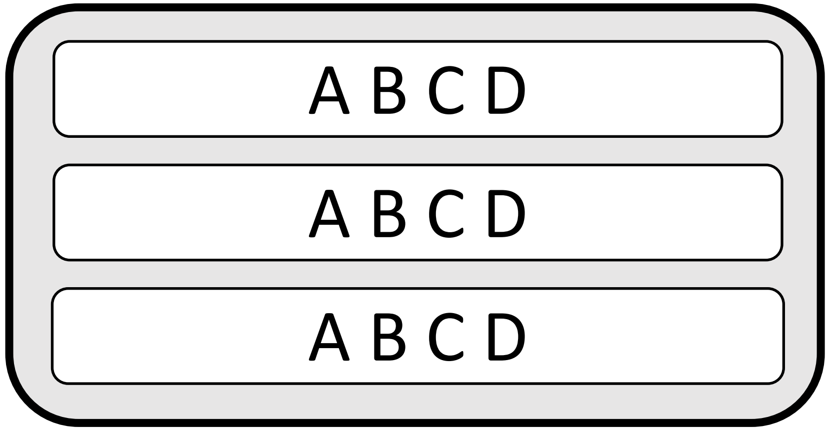


How to design KVS for efficient scans?

Key design decisions

(A) Updates ***in-place*** ***log-structured*** ***delta-main***

(B) Layout ***column (PAX)*** ***row***



BOSTON UNIVERSITY

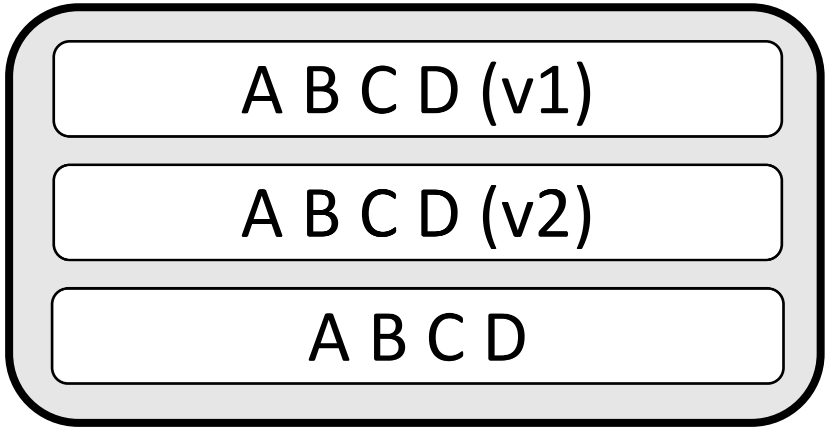
How to design KVS for efficient scans?

Key design decisions

(A) Updates *in-place* *log-structured* *delta-main*

(B) Layout *column (PAX)* *row*

(C) Versioning *clustered*



any other options?

How to design KVS for efficient scans?

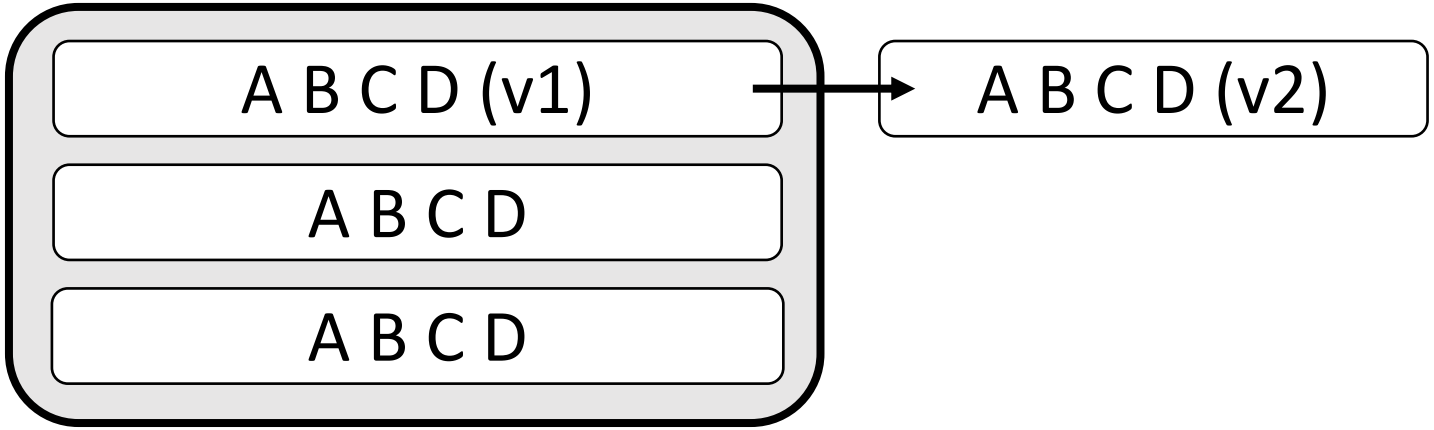
Key design decisions

(A) Updates ***in-place*** ***log-structured*** ***delta-main***

(B) Layout ***column (PAX)*** ***row***

(C) Versioning ***clustered***

chained



BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

(A) Updates ***in-place*** ***log-structured*** ***delta-main***

(B) Layout ***column (PAX)*** ***row***

(C) Versioning ***clustered*** ***chained***

what comes as a result of versioning?

BOSTON UNIVERSITY

Garbage Collection (GC)

(A) Periodic ***separate dedicated thread(s)***

(B) Piggy-backed GC during scans

increases scan time ***but frequently read tables benefit***

avoids re-reading for GC (since data is already accessed)

BOSTON UNIVERSITY

Design Space

Updates	×	Layout	×	Versioning	×	GC
in-place		***column (PAX)***		***clustered***		***periodic***
log-structured		***row***		***chained***		***piggy-backed***
delta-main						

hybrid designs are also valid!
should we consider all possible designs?

Design Space

Updates	×	Layout	×	Versioning	×	GC
in-place		***column (PAX)***		***clustered***		***periodic***
log-structured		***row***		***chained***		***piggy-backed***
delta-main						

some combinations do not make sense:

log-structured & column < delta-main & column

log-structured & clustered < log-structured & chained

note that each combination here represents multiple options

Design Space

Updates	×	Layout	×	Versioning	×	GC
in-place		***column (PAX)***		***clustered***		***periodic***
log-structured		***row***		***chained***		***piggy-backed***
delta-main						

focus on two extremes: ***TellStore-Log***

(1) log-structured & row & chained

(2) delta-main & column & clustered

TellStore-Col

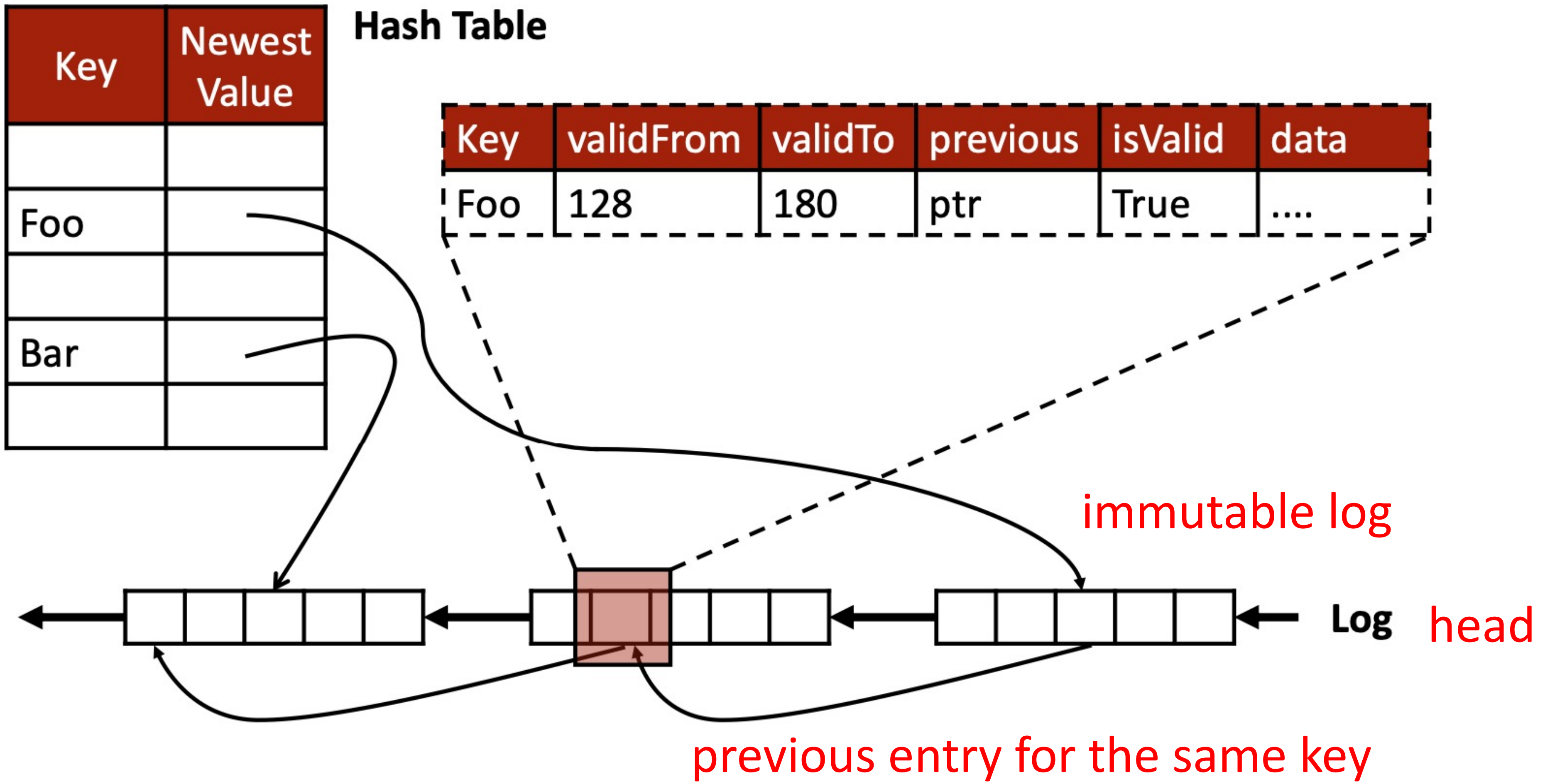
BOSTON UNIVERSITY

TellStore-Log

one log per table (locality for scans)

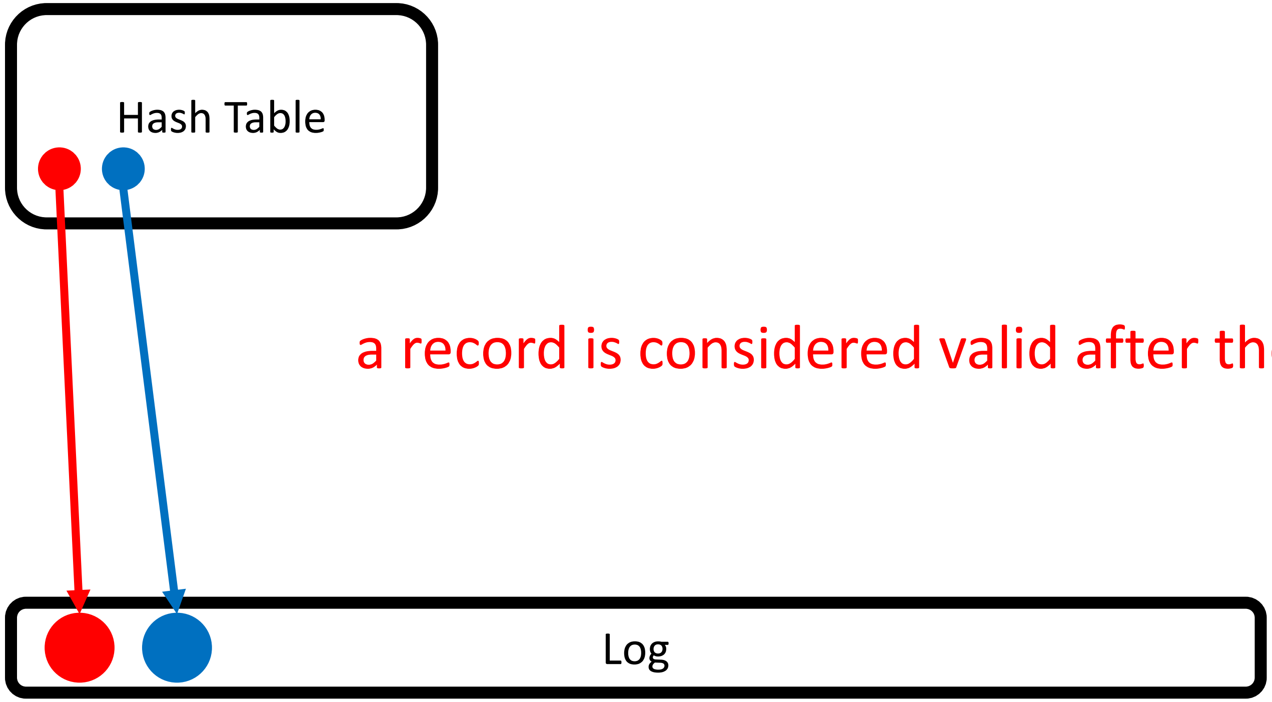
inserts, **updates**, and **deletes** are all **logged**

lock-free hash table



BOSTON UNIVERSITY

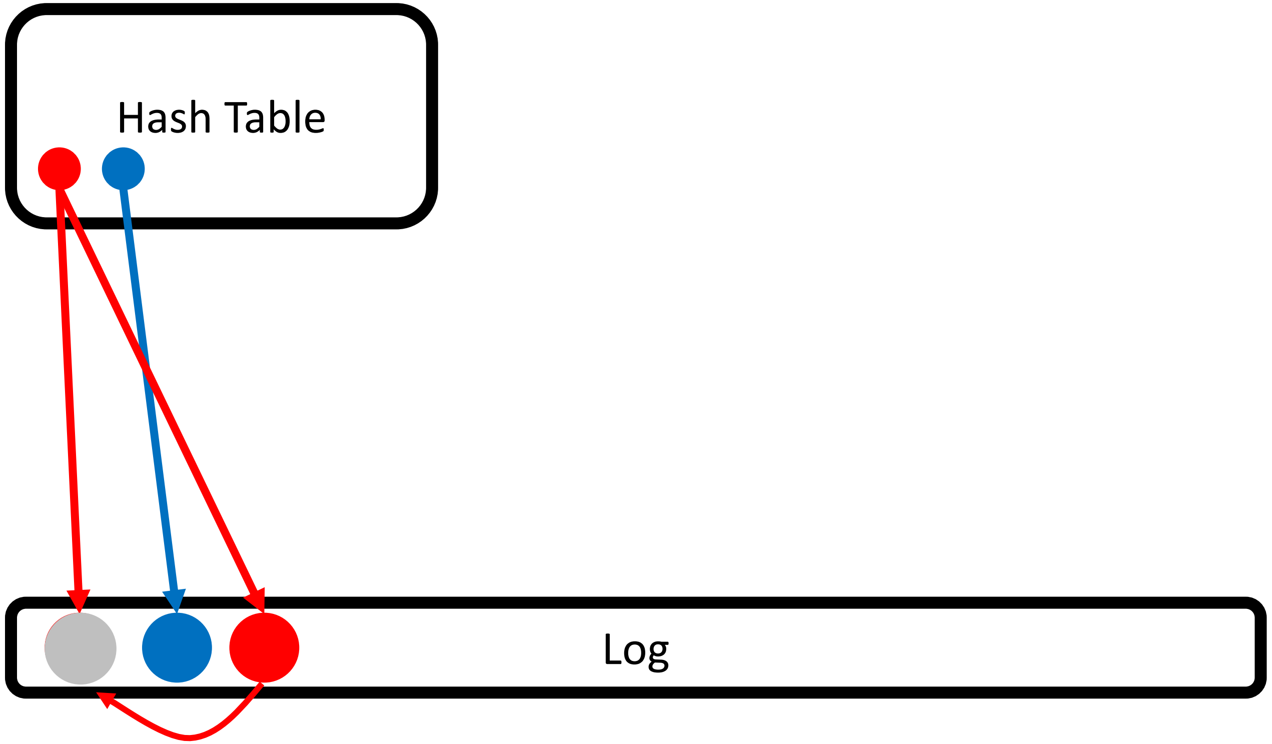
TellStore-Log Insertion



a record is considered valid after the hash table pointer is updated

the log contains ***rows***

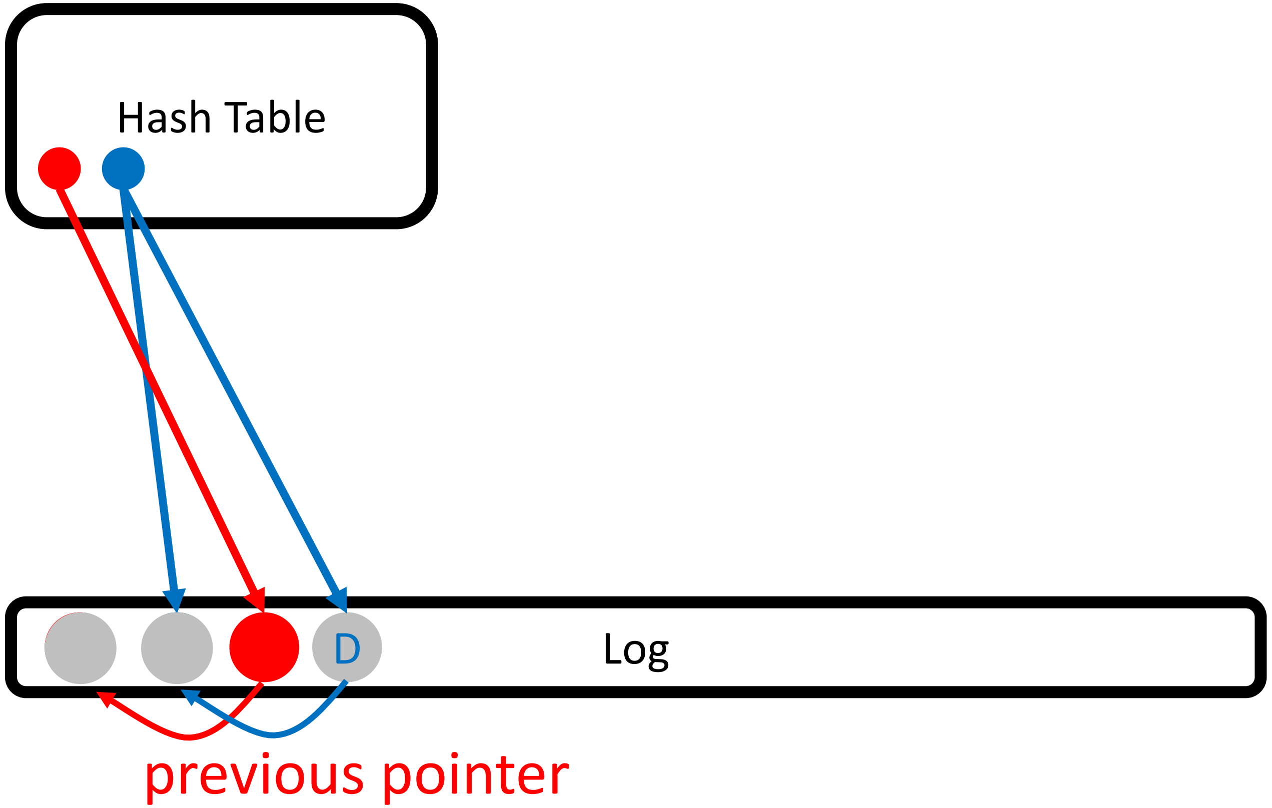
TellStore-Log Update



previous pointer

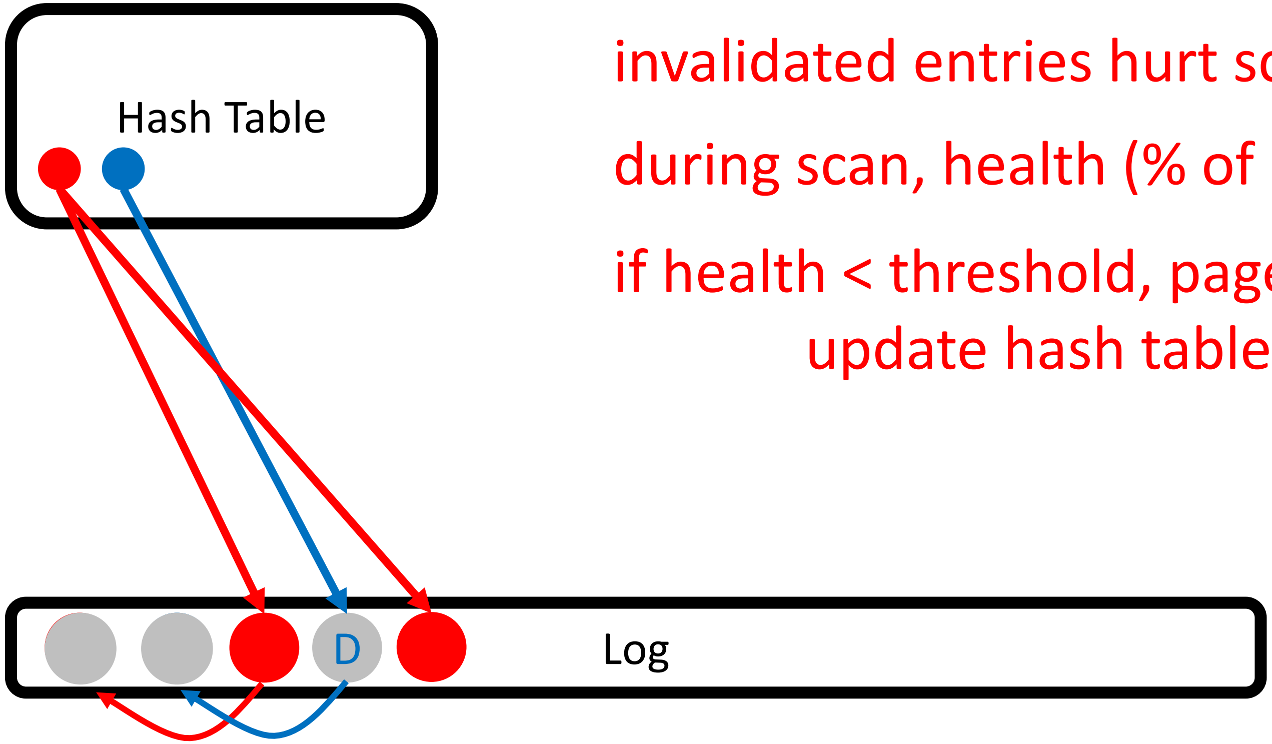
the log contains ***rows***

TellStore-Log Delete



the log contains ***rows***

TellStore-Log Garbage Collection



invalidated entries hurt scan performance

during scan, health (% of invalid entries) per page is calculated

if health < threshold, page is re-written in the head of the log & update hash table & old page is reclaimed

the log contains ***rows***

TellStore-Log in a nutshell

log-structure: efficient puts

hash-table: efficient gets (always points to the latest entry)

snapshot Isolation: high throughput, no locks needed

self-contained log: efficient scans (valid from/to needed)

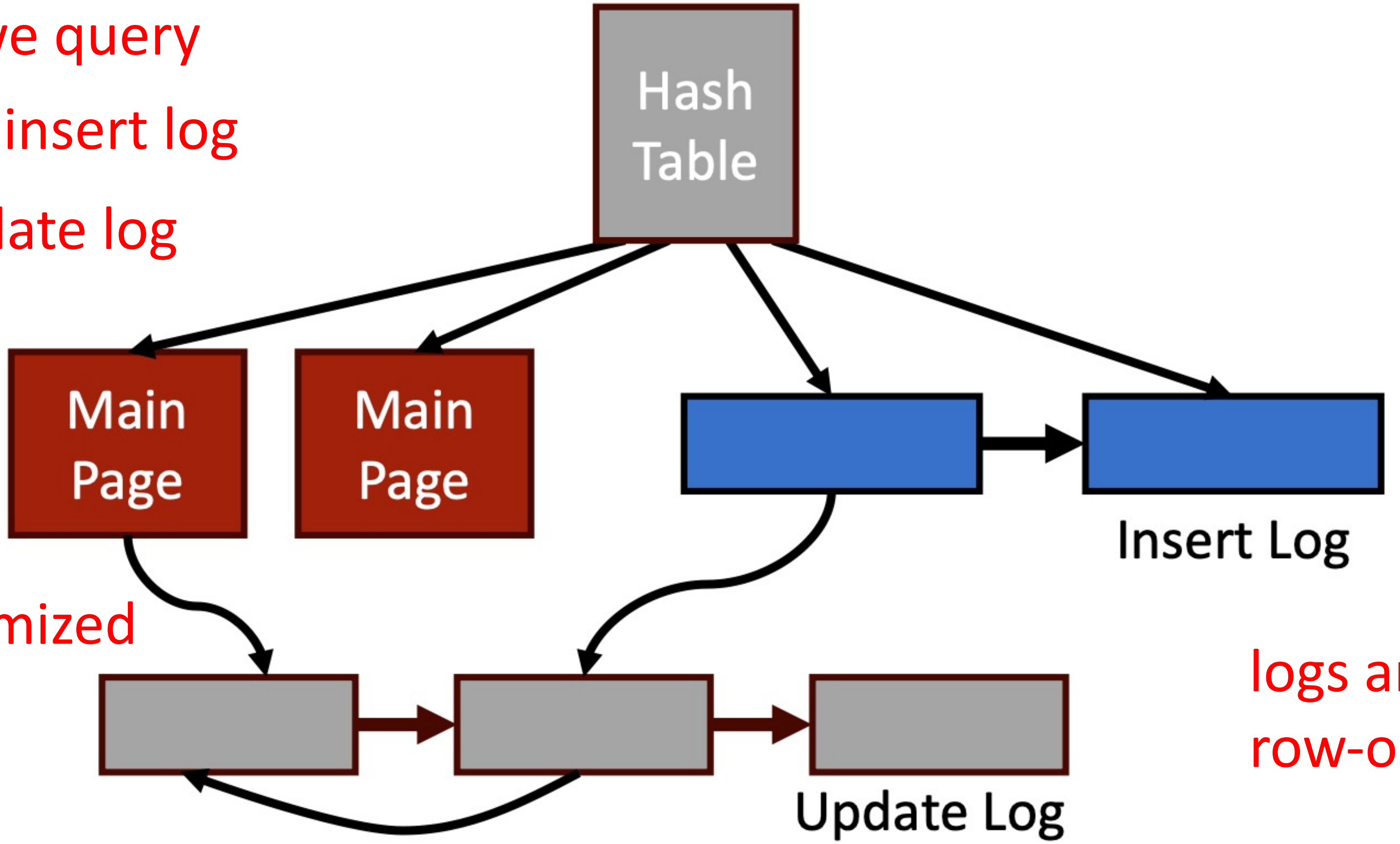
lazy GC: Optimize tables that are scanned four data structures

TellStore-Col

before inserting we query

new entries go to insert log

updates go to update log



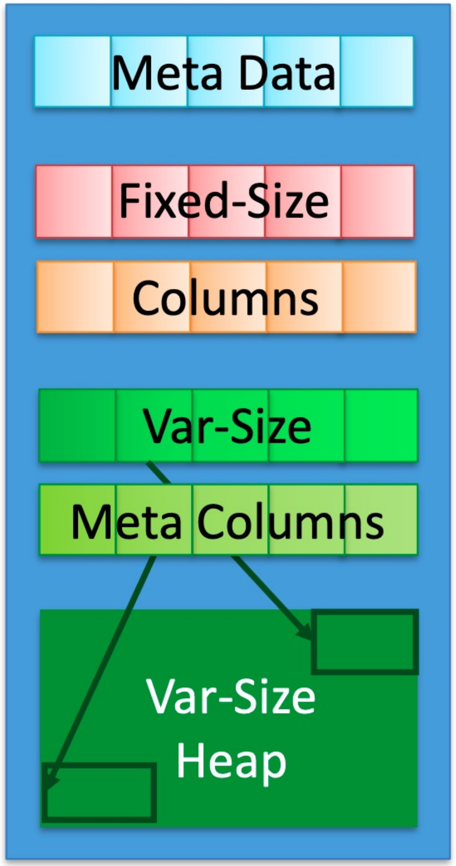
main is read-only

columnar: read-optimized

logs are write-optimized

row-oriented: append-only

TellStore-Col Layout



fixed-size data is stored in columnar format

variable-size data is index in columnar format
but stored in row-wise format

why row-wise?

(1) faster materialization (contiguous copying)

(2) less metadata (one offset for many columns)

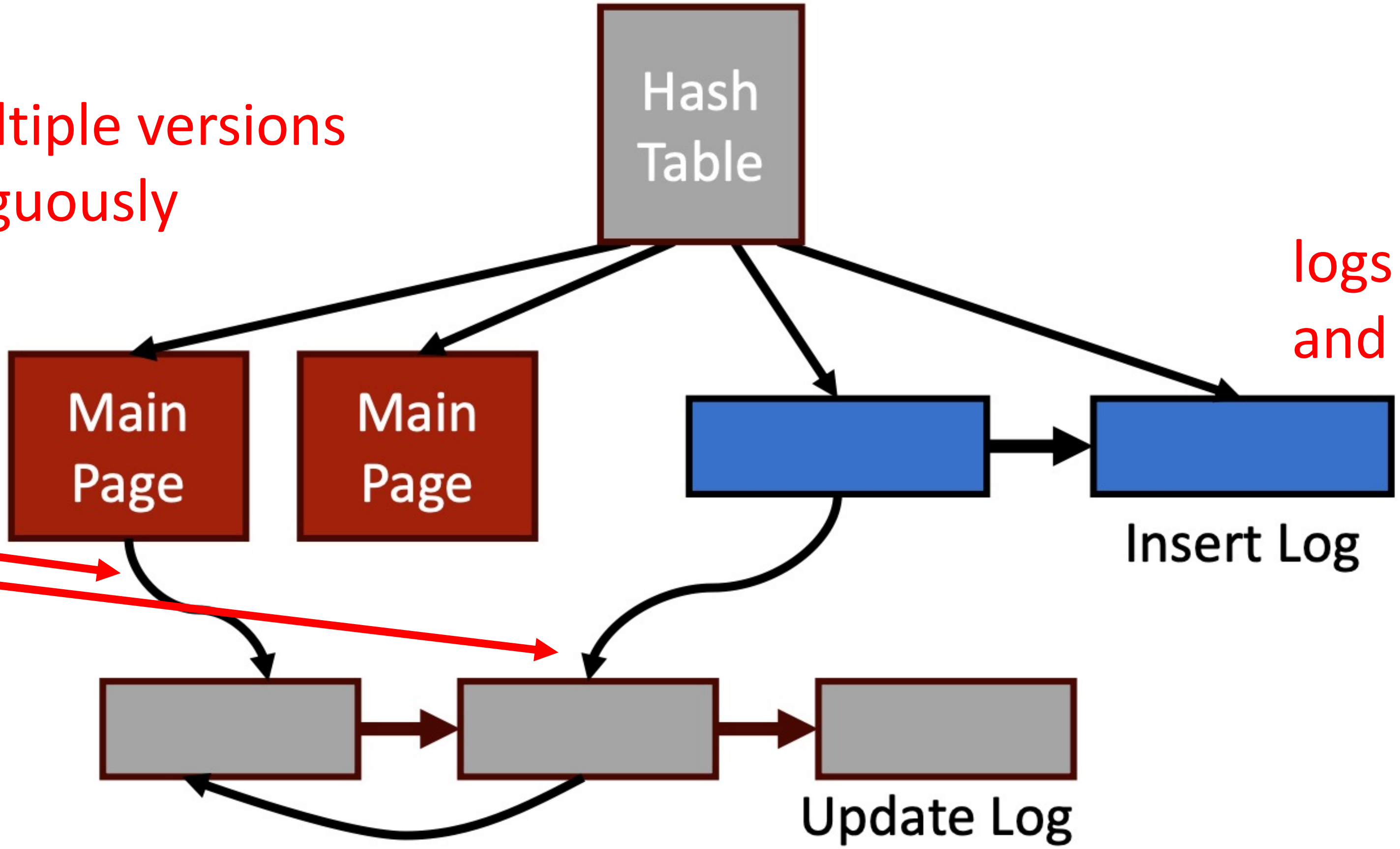
BOSTON UNIVERSITY

TellStore-Col Versioning

in main storage multiple versions are **clustered** contiguously

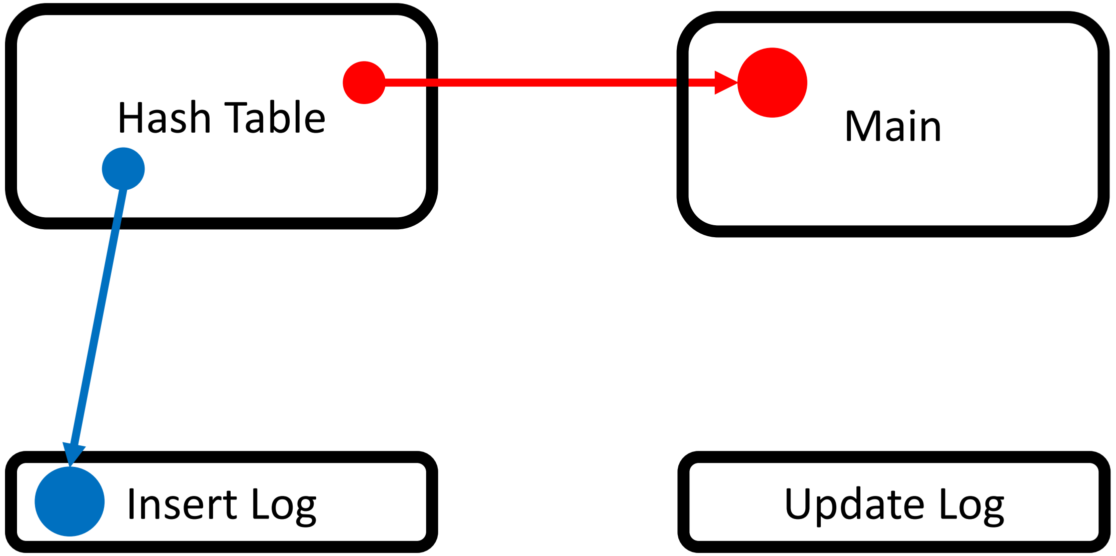
logs work like before and they are row-wise

newest pointers may exist

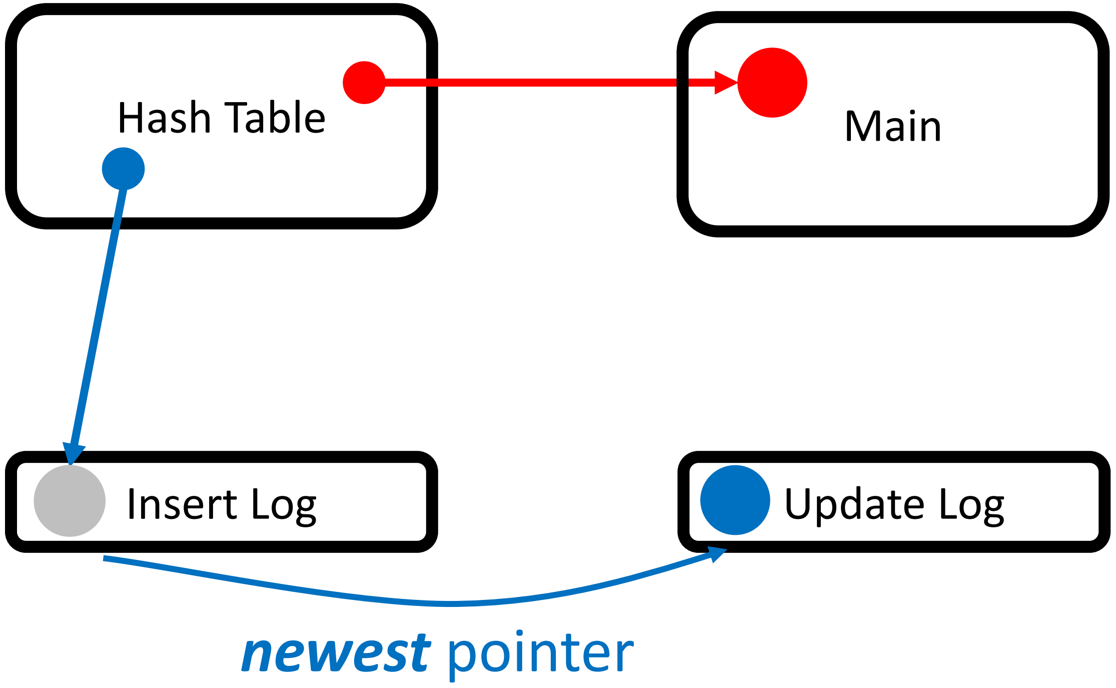


previous pointers may exist only in update log

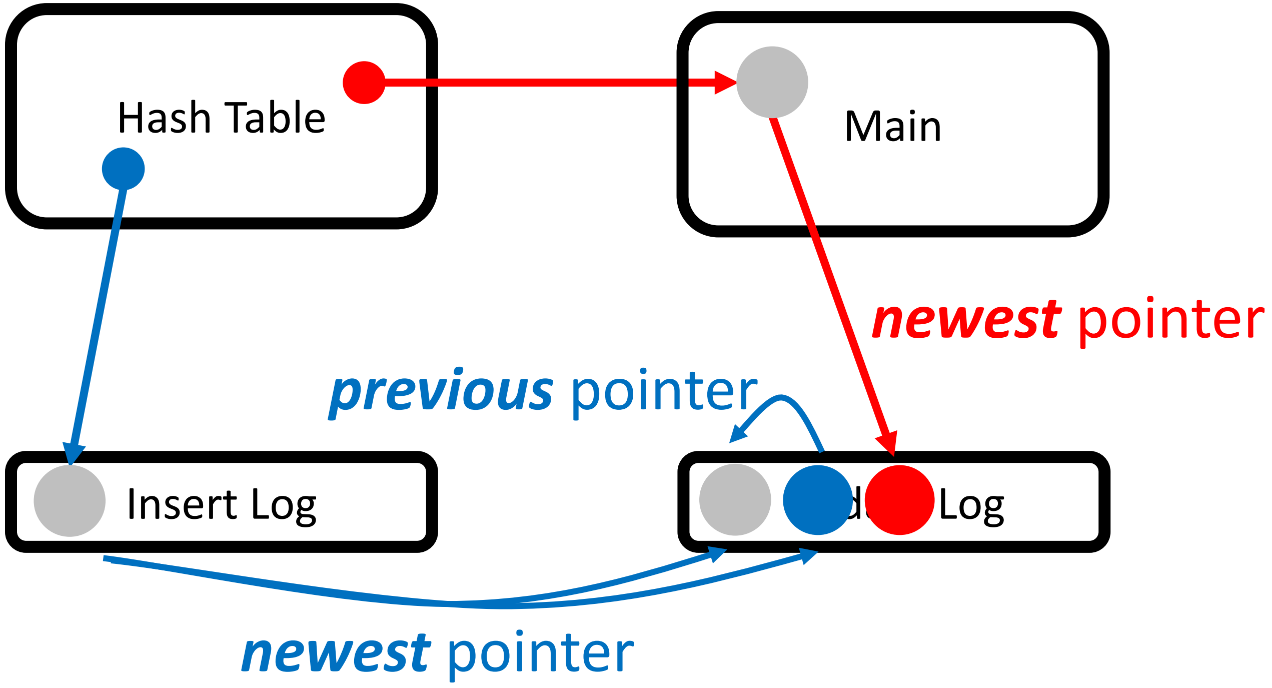
TellStore-Col Insertion



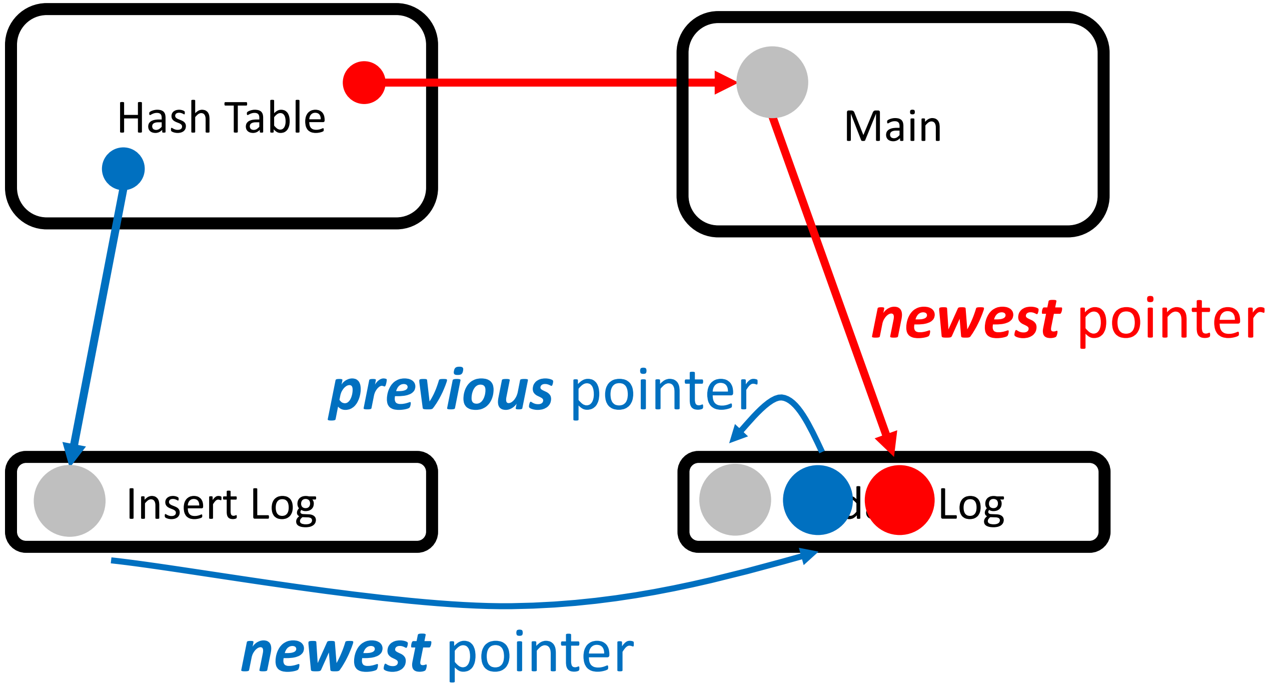
TellStore-Col Update



TellStore-Col Update



TellStore-Col Garbage Collection



dedicated thread
(conversion from row to column)

all main pages with invalid entries

all pages from insert log + update to main

run GC frequently + truncate logs

TellStore-Col in a nutshell

delta-main: compromise between puts and scans

hash-table: efficient gets (always points to the latest entry, may need one more pointer to follow)

PAX layout: minimize disk I/O, maintain locality for scans

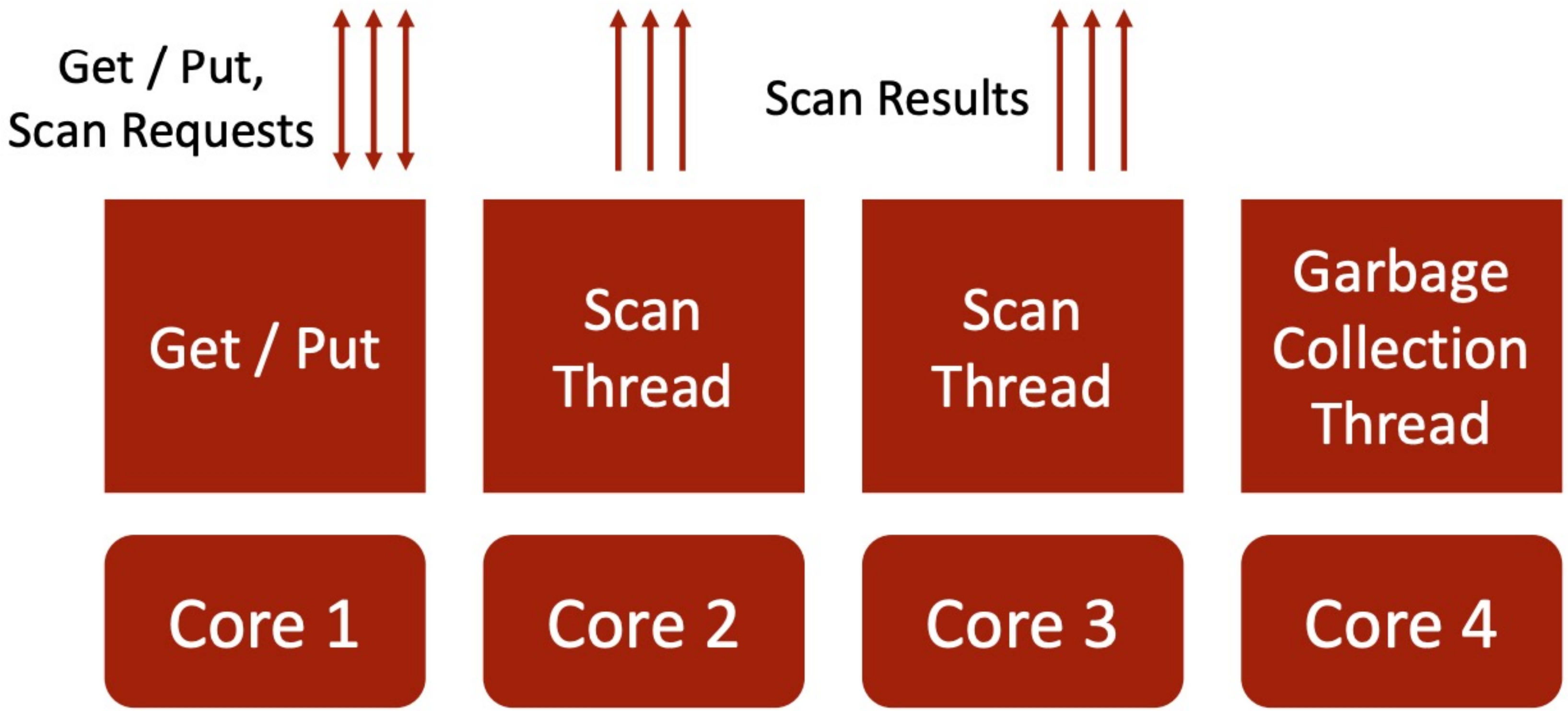
separate insert/update logs: efficient GC

eager GC: improve scans

Implementation Details

scans are assigned to dedicated threads

scan coordinator for shared scans

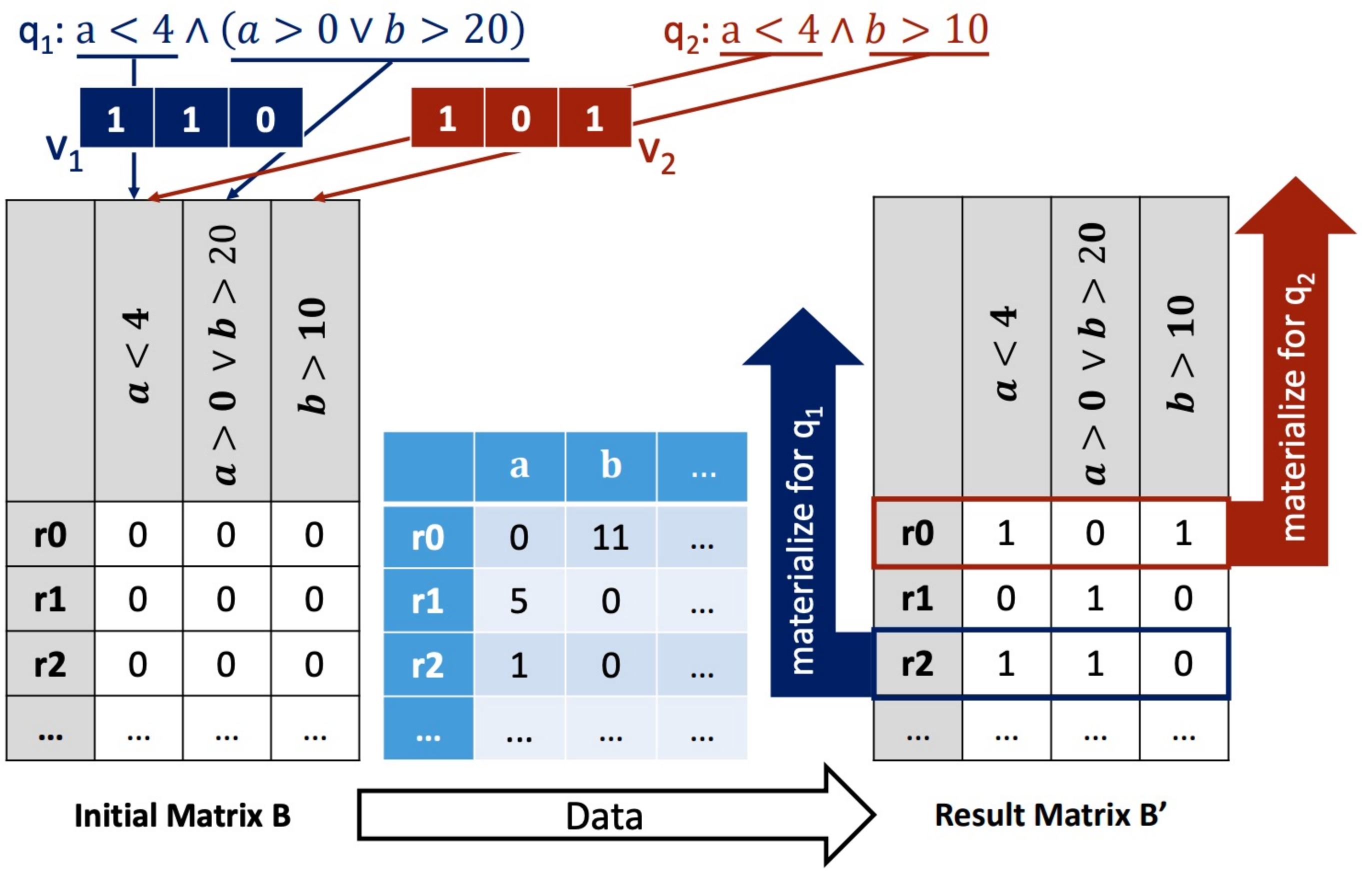


Implementation Details

efficient predicate evaluation via code generation and predicate pushdown

all queries in CNF

reuse work



Yahoo! Cloud Serving Benchmark# (YCSB#)

based on YSCB, a put/get benchmark

main_table (P, A, B, C, D, E, F, G, H, I, J) **P: 8-byte ley** | A-H: 2-bytes, 4-bytes, 8-bytes | I-J: strings 12-16 bytes

- *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
```

Experiments: Transactional Workload

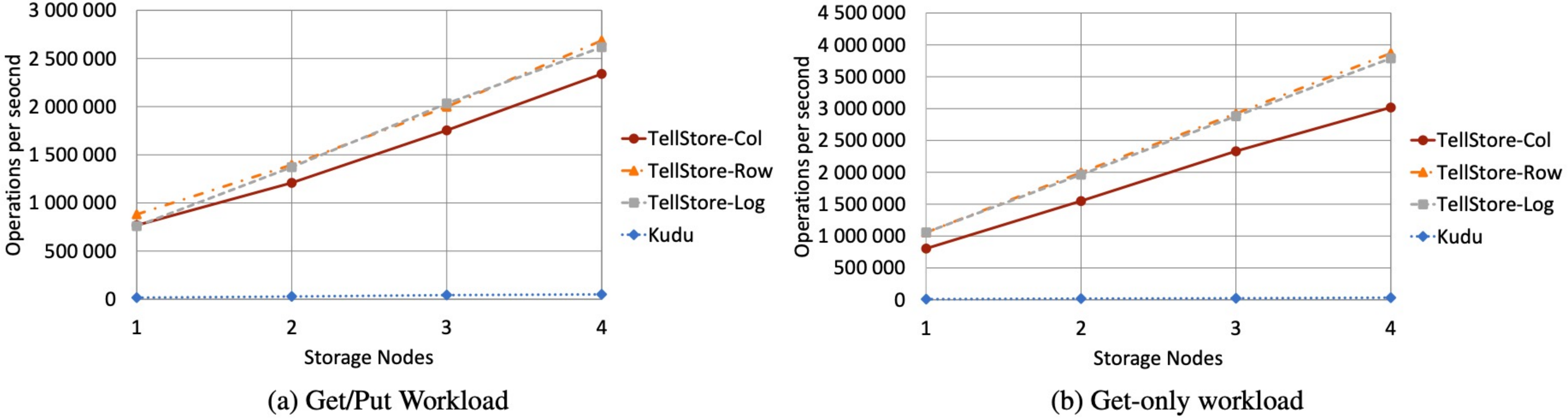


Figure 8: Exp 1, Throughput: YCSB, TellStore Variants and Kudu, Vary Storage Nodes

Kudu is used as it was the most competitive to begin with

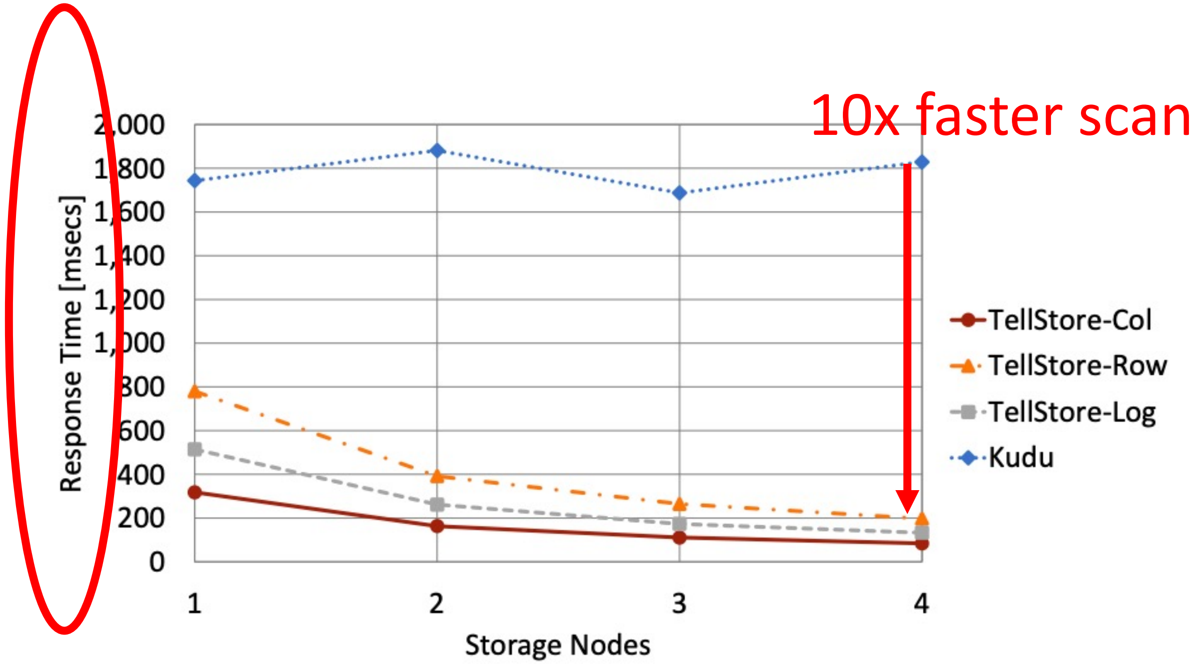
All TellStore approaches are not that far!

BOSTON UNIVERSITY

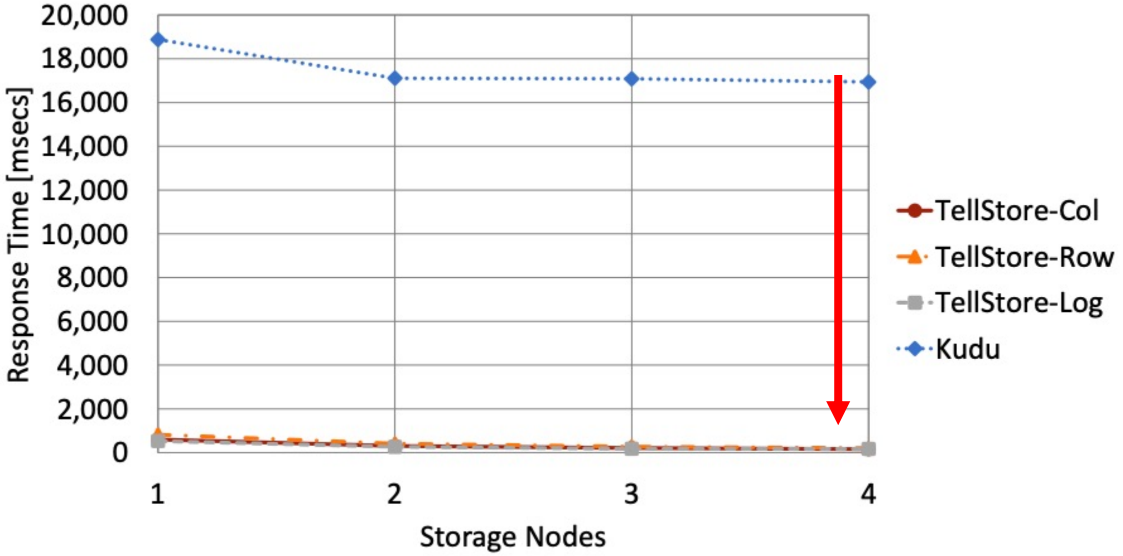
Experiments: Scans

several orders of magnitude faster scans!

10x faster scans!



(a) YCSB# Query 1



(b) YCSB# Query 3

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

Q3 does not have projections, so no benefit from columnar

Experiments: Mixed Workload

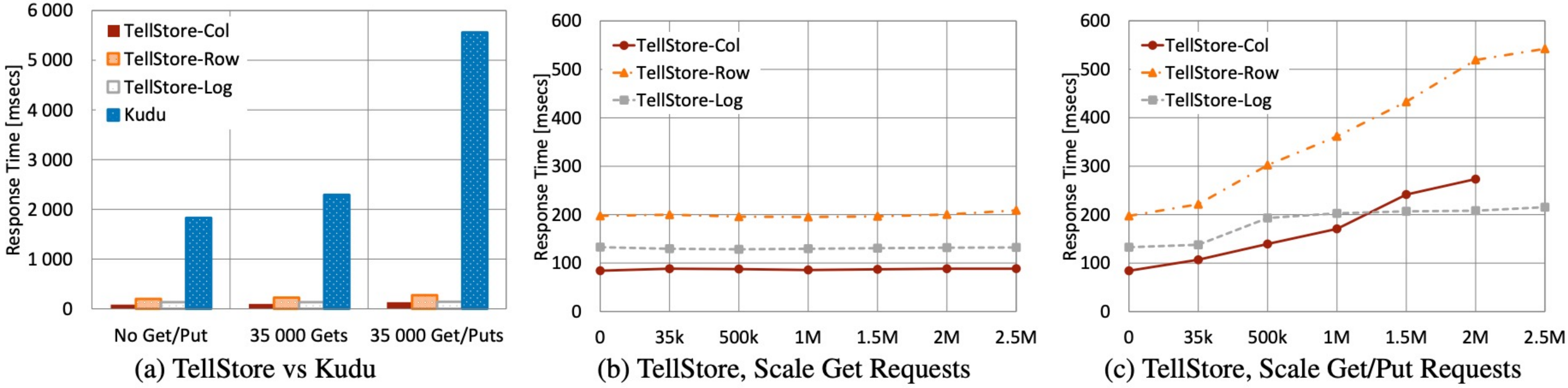


Figure 11: Exp 4, Response Time: YCSB# Query 1, 4 Storage Nodes

Contrary to competition, scan perf. is stable with more gets/puts

In the absence of updates TellStore scales perfectly: scans+gets go to different cores

With 50% updates eventually logging wins

BOSTON UNIVERSITY

Things to remember

KVS vs. Scans: how to compromise, navigate the design space

✓delta-main vs. log-structure

✓chained vs. clustered versions

✓row-major vs. column-major

✓lazy vs. eager GC

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 7

Fast Scans on Key-Value Stores

Prof. Manos Athanassoulis

https://bu-disc.github.io/CS561/