BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 7

Design Tradeoffs in Key-Value Stores

Prof. Manos Athanassoulis

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

Do we have a quiz today … ?

Yes!

what to do now?

A) read the syllabus and the website
B) register to Piazza + Gradescope
C) finish project 0
D) finish project 1
E) finish project proposal (due 2/22)
F) register for the student-presentations (by 3/6)

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?

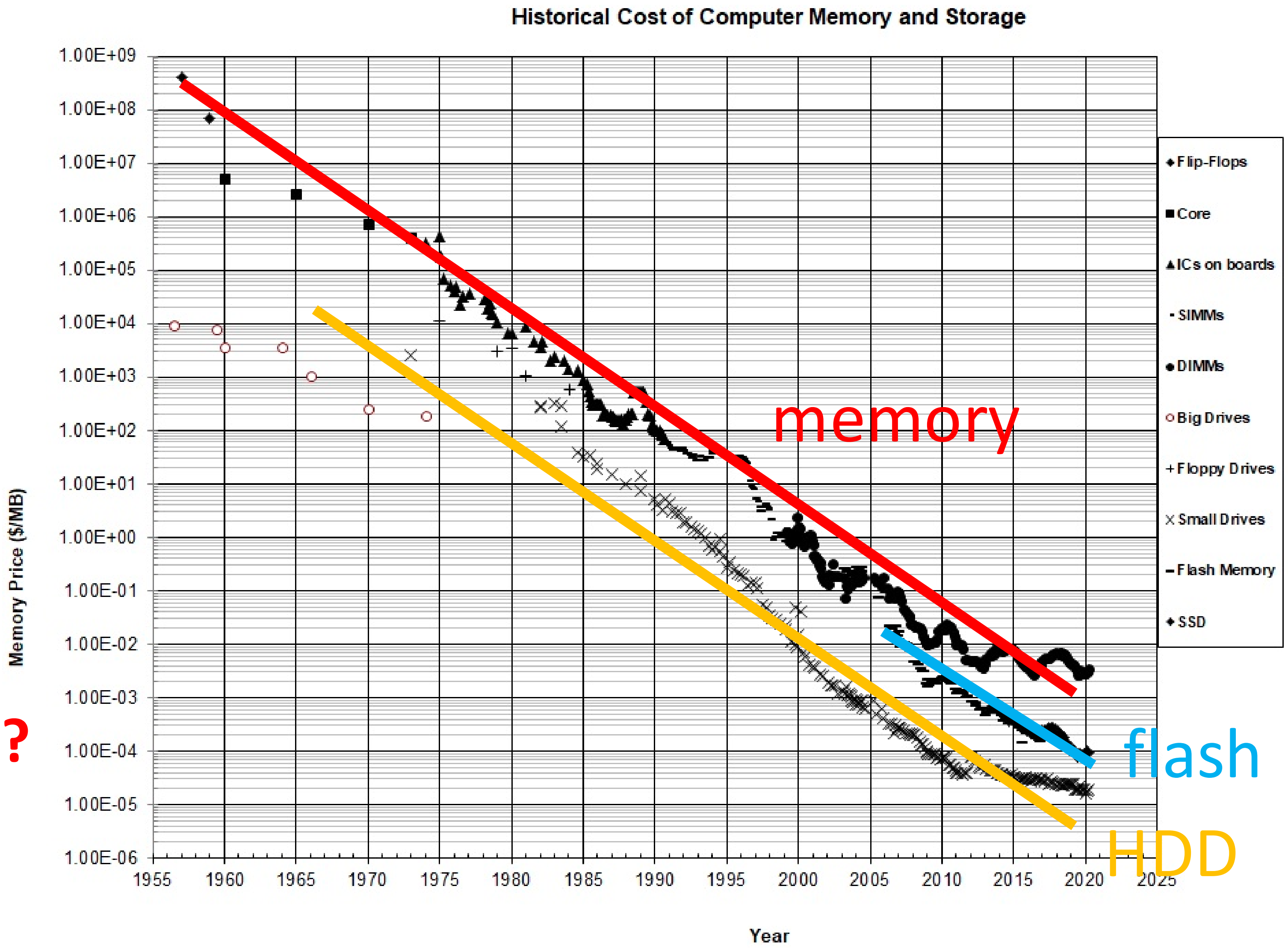
Why combine KVS and analytical systems?

cheaper and cheaper storage

more data ingestion

need for write-optimized data structures

what about analytical queries?



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

Most organizations maintain both

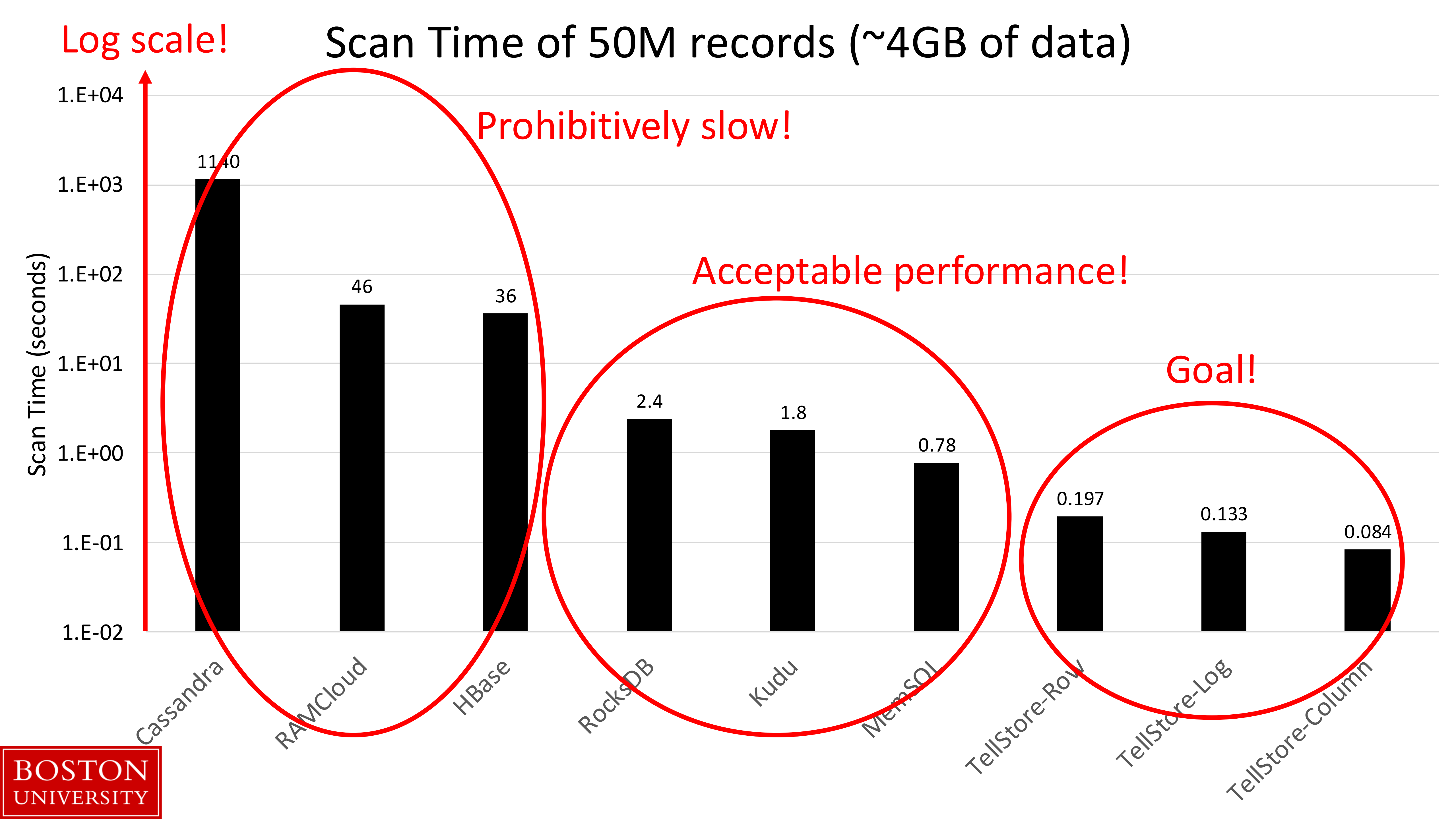
- ***transactional*** systems (often as key-value stores)
- ***analytical*** systems (often as column-stores)

problems?

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

harder to maintain (more systems, more code)

time consuming data integration/transfer



Log scale!
Scan Time of 50M records (~4GB of data)
Prohibitively slow!
Acceptable performance!
Goal!
Scan Time (seconds)
1.E+04
1.E+03
1.E+02
1.E+01
1.E+00
1.E-01
1.E-02
1140
46
36
2.4
1.8
0.78
0.197
0.133
0.084
Cassandra
RAMCloud
HBase
RocksDB
Kudu
MemSQL
TellStore-Row
TellStore-Log
TellStore-Column
BOSTON UNIVERSITY

Goals of this paper

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

get/put operations need **sparse data structures** → locality is not required, access one object

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

SQL over NoSQL

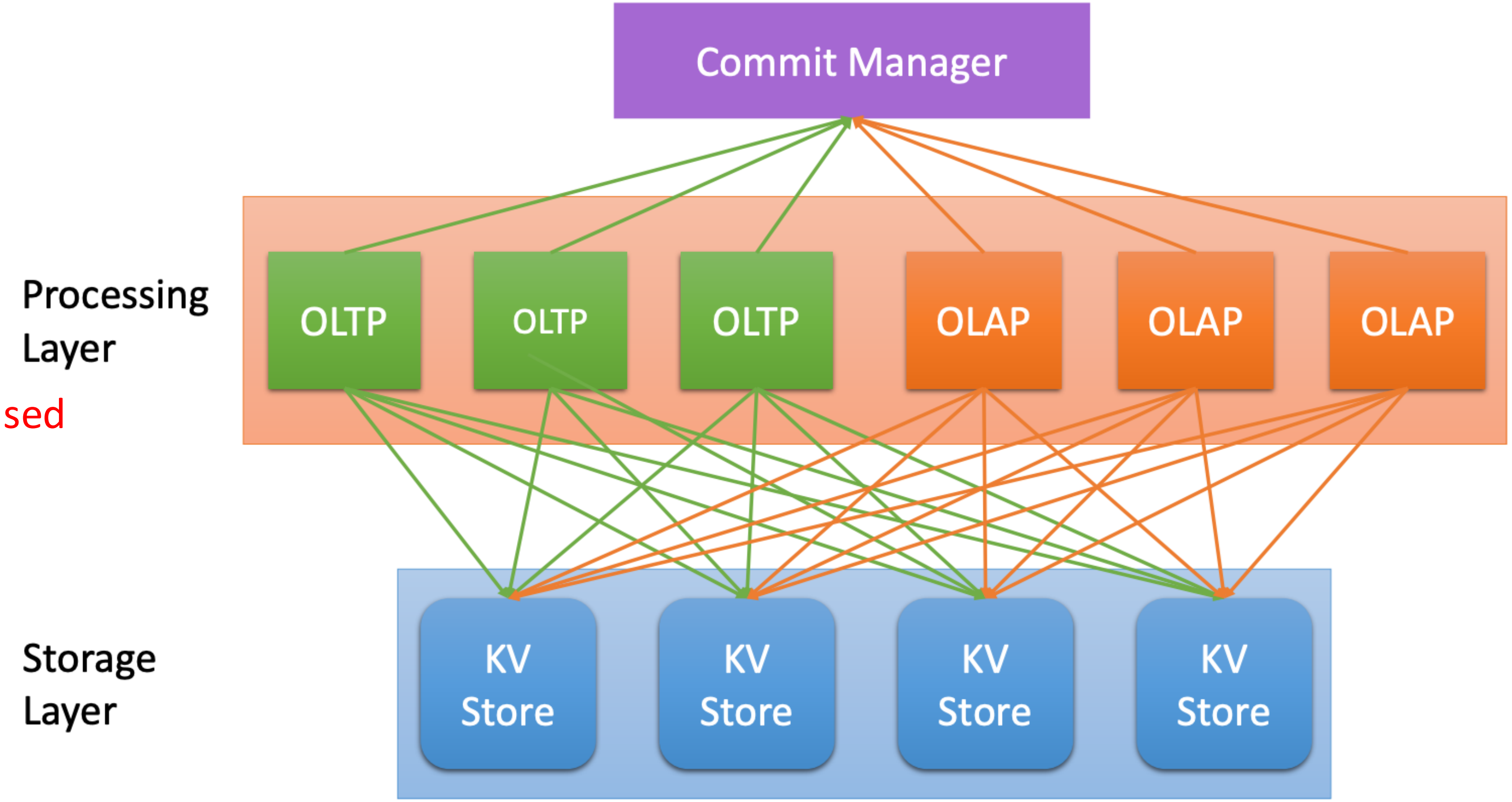
Elasticity

Snapshot Isolation

e.g., MVCC, no locking, timestamp based

Support for:

- Scans
- Versioning
- Batching



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?

BOSTON
UNIVERSITY

Challenges

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

<u>scans vs. versioning</u>

#1, John, 2/4/88, Boston, v1
#1, John, 2/4/88, Cambridge, v2

versioning reduces locality in scans

checking for the latest version in scans needs CPU time

Challenges

scans vs. get/put

scans vs. versioning

scans vs. batching

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

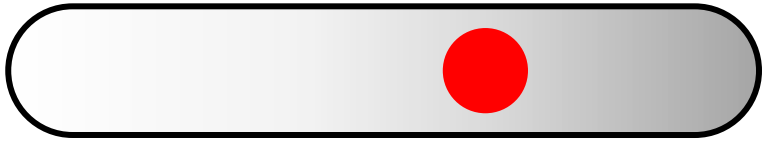
(B) Layout

(C) Versioning

How to design KVS for efficient scans?

Key design decisions

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

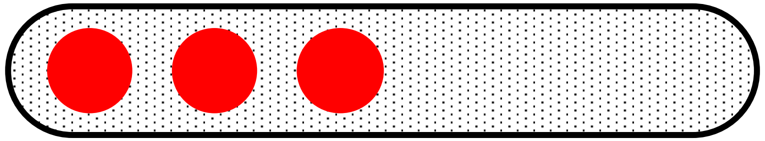


BOSTON UNIVERSITY

How to design KVS for efficient scans?

Key design decisions

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



How to design KVS for efficient scans?

Key design decisions

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

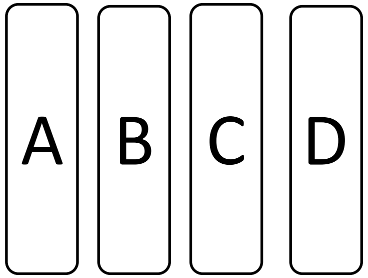
BOSTON UNIVERSITY

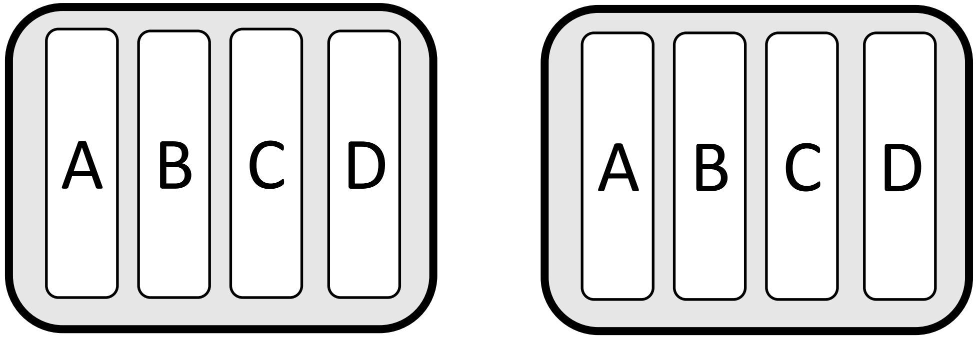
How to design KVS for efficient scans?

Key design decisions

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

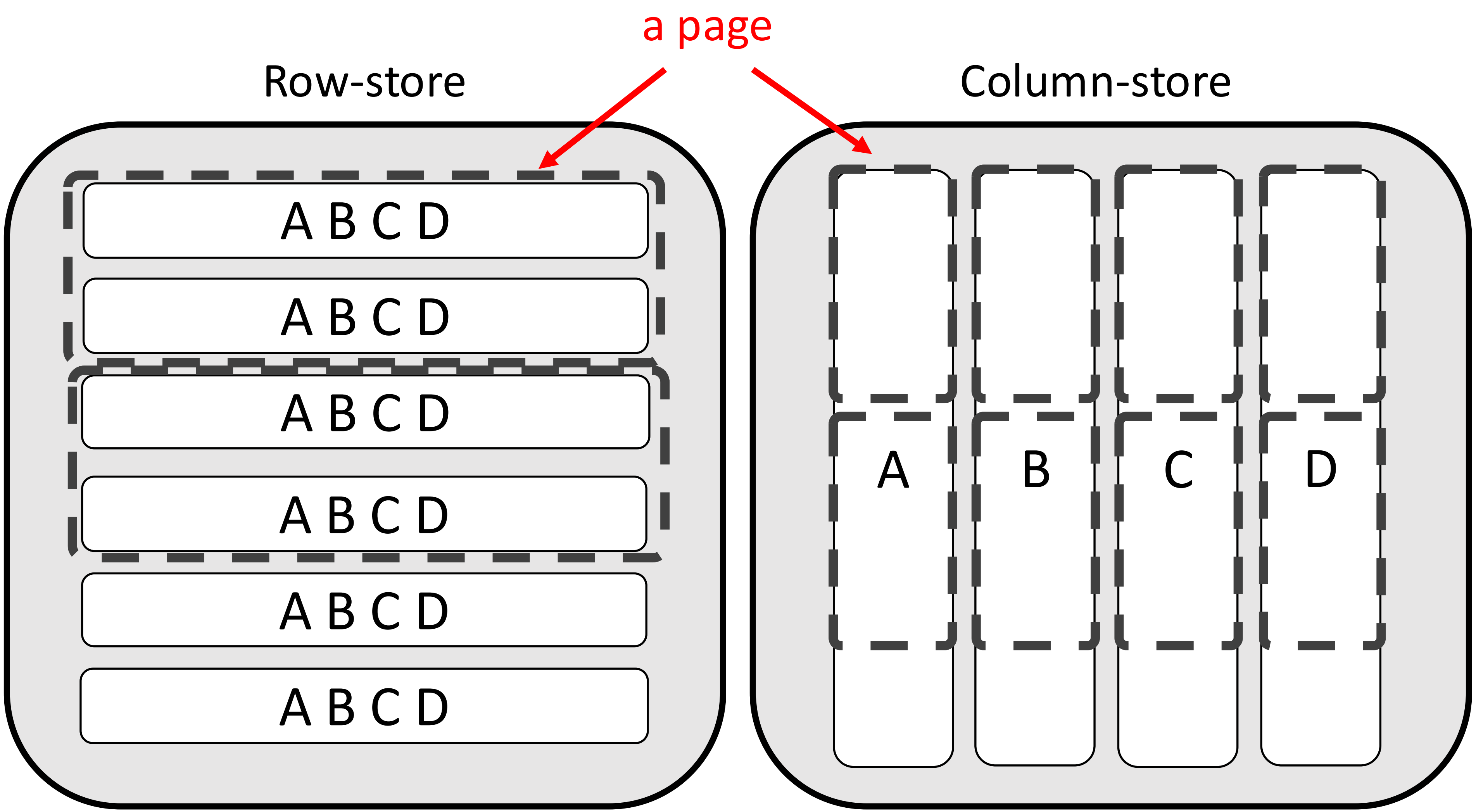
(B) Layout *column* *PAX (columnar per page)*

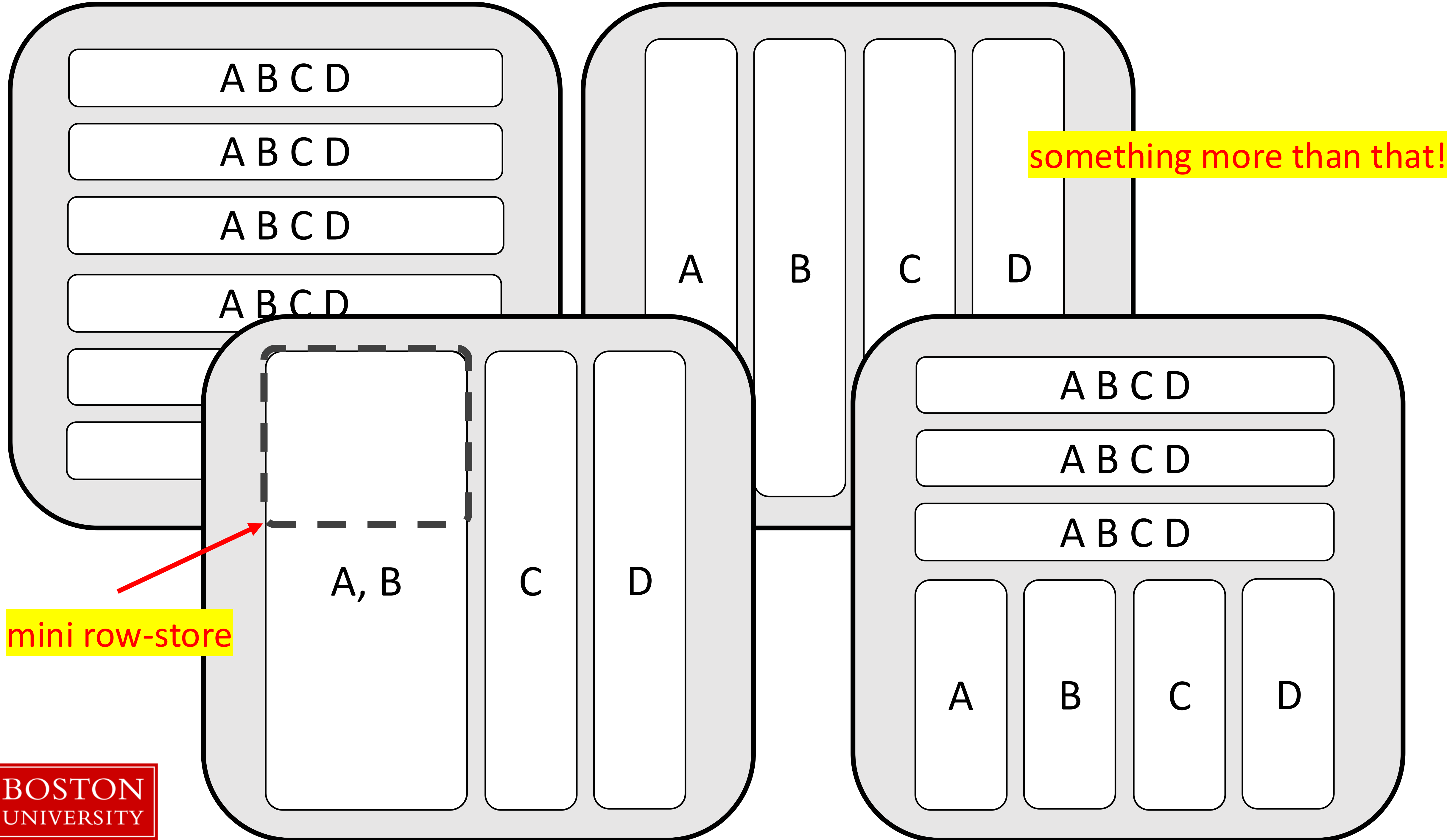




Small detour: page layouts

middle ground?




A B C D
A B C D
A B C D
A B C D
A
B
C
D
something more than that!
A, B
C
D
mini row-store
A B C D
A B C D
A B C D
A
B
C
D

Partition Attributes Across (PAX)

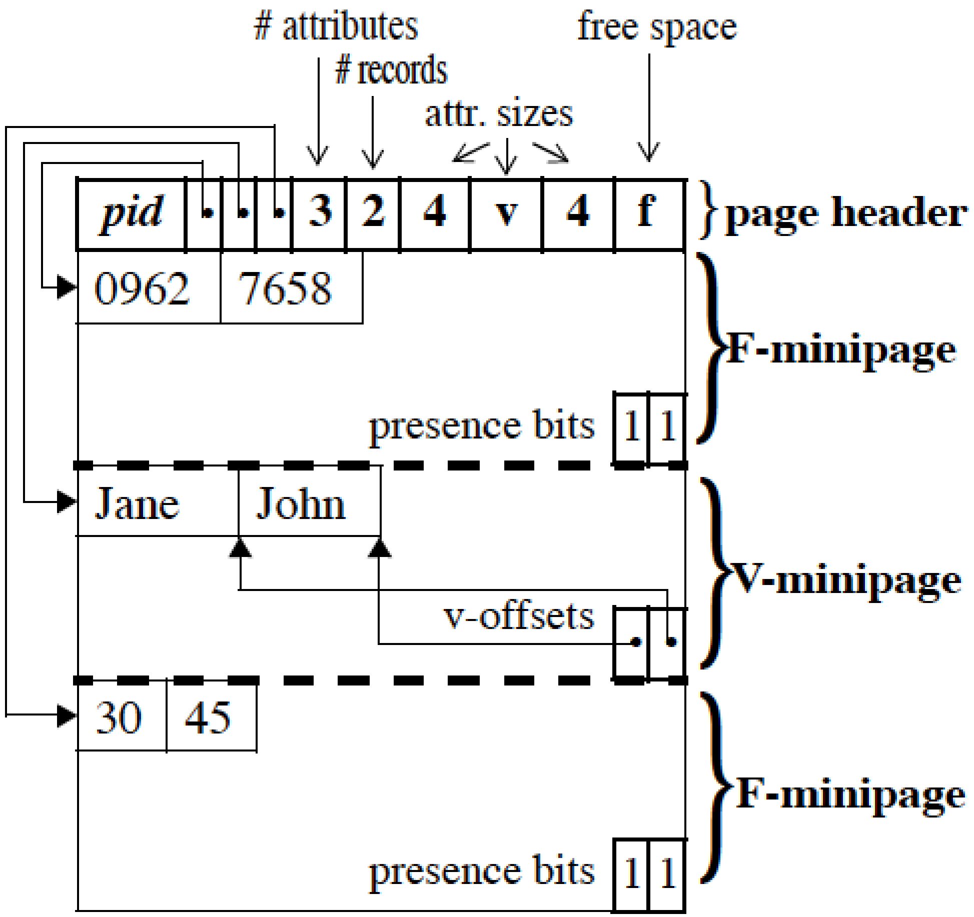
Middle ground?

Decompose a slotted-page internally in mini-pages per attribute

✓ Cache-friendly

- ➢ Brings only relevant attributes to cache

- Compatible with slotted-pages? ✅
- Same update abstraction? ✅
  - – (insert in a page)

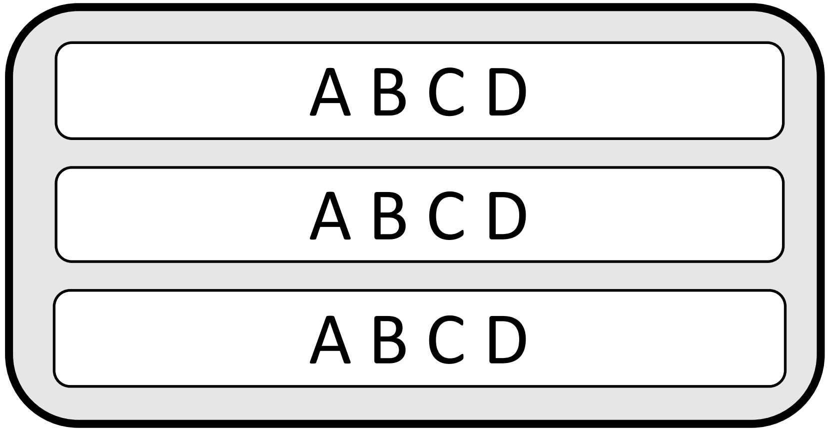


How to design KVS for efficient scans?

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*

A B C D (v1)

A B C D (v2)

A B C D

any other options?

BOSTON UNIVERSITY

How to design KVS for efficient scans?

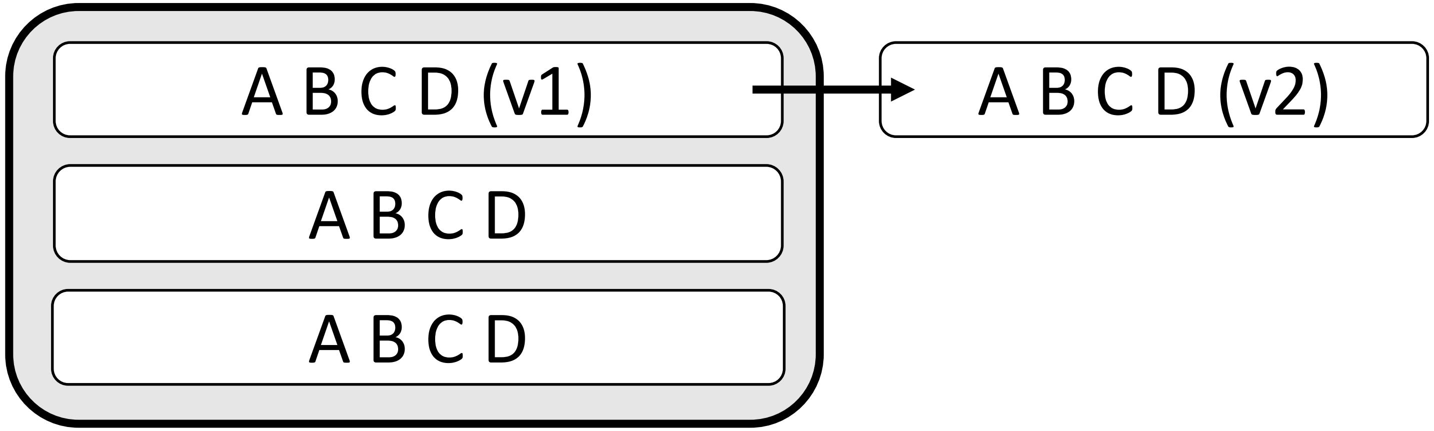
Key design decisions

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

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

(C) Versioning *clustered*

chained



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)

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?

BOSTON UNIVERSITY

Design Space

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

some combinations can be discarded:

log-structured & column ***worse than*** delta-main & column

log-structured & clustered ***worse than*** log-structured & chained

note that each combination here represents multiple options

BOSTON UNIVERSITY

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

BOSTON UNIVERSITY

Design Space

Updates × Layout × Versioning × GC

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

Design Space

Updates × Layout × Versioning × GC

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

focus on two extremes:

(1) log-structured & row & chained — ***TellStore-Log***

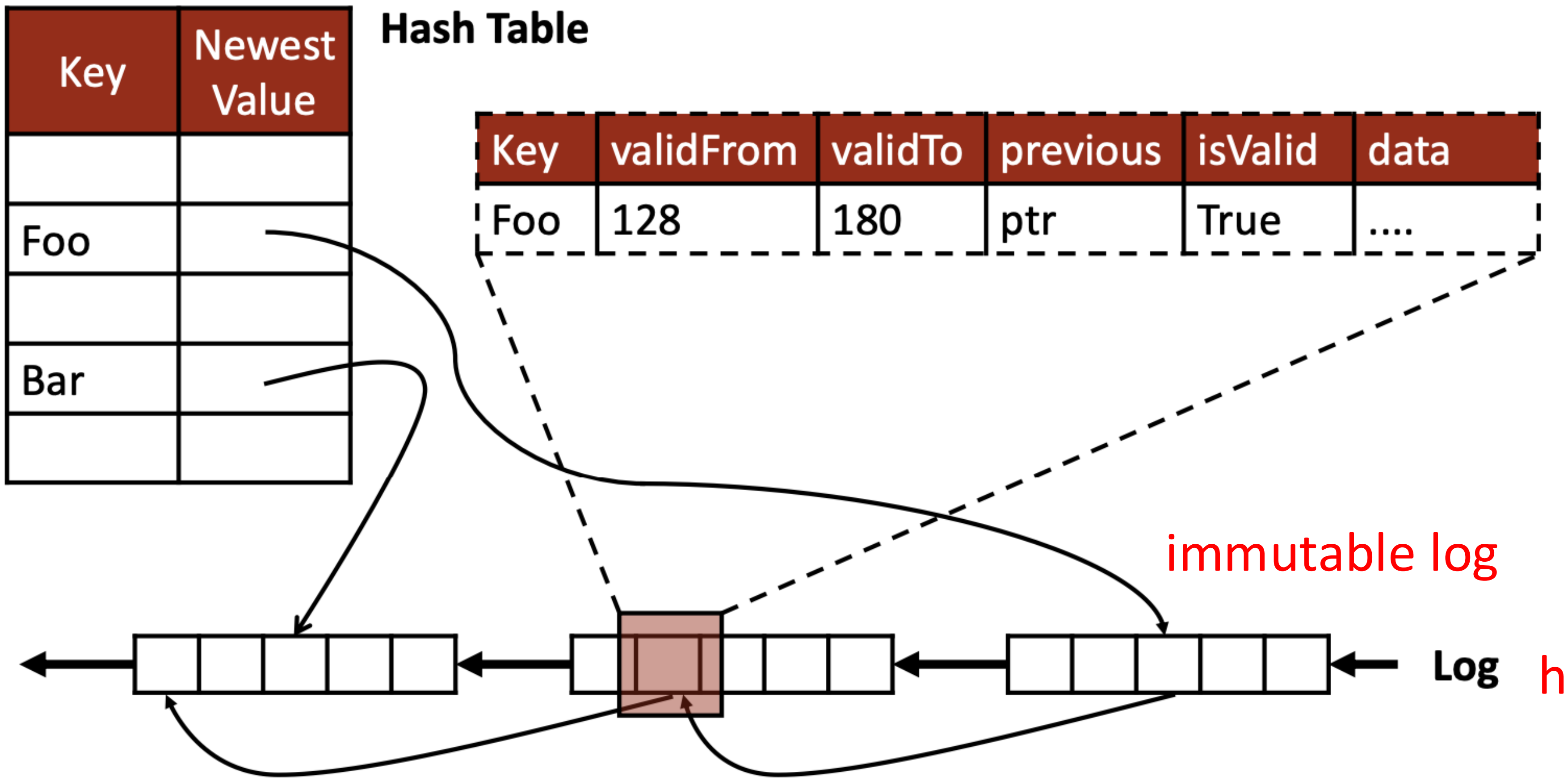
(2) delta-main & column & clustered — ***TellStore-Col***

TellStore-Log

one log per table (locality for scans)

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

lock-free hash table



immutable log

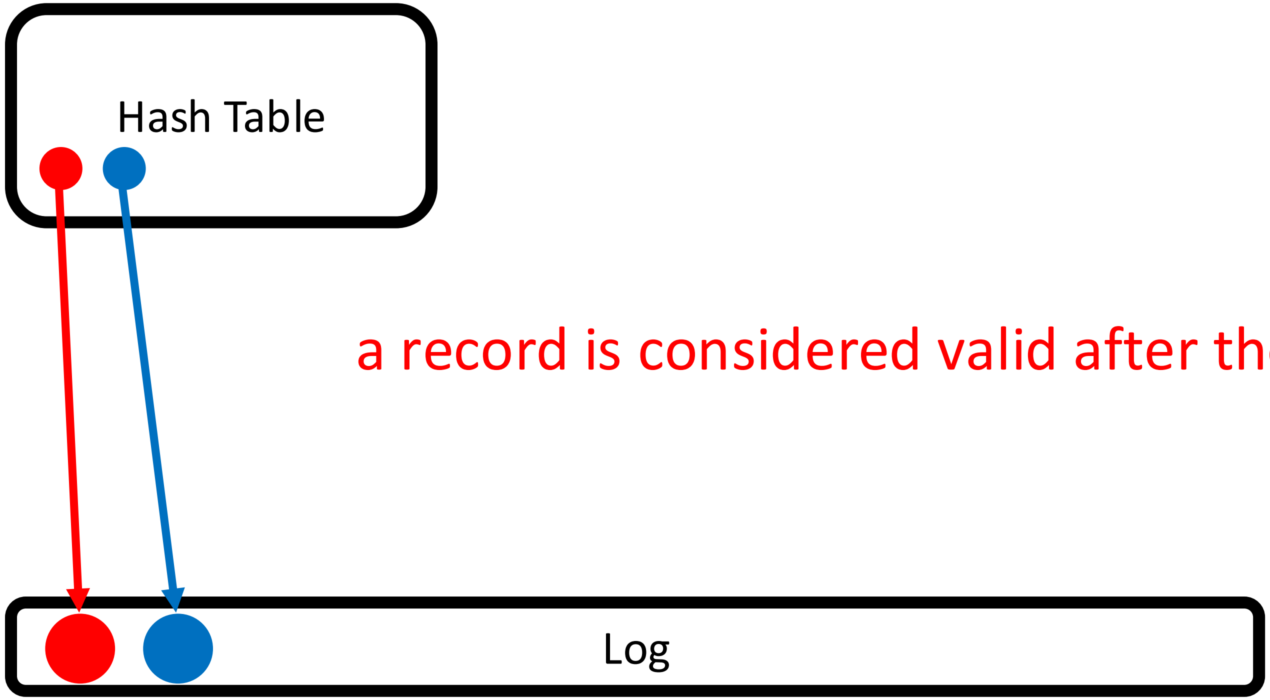
head

previous entry for the same key

why?

BOSTON UNIVERSITY

TellStore-Log Insertion

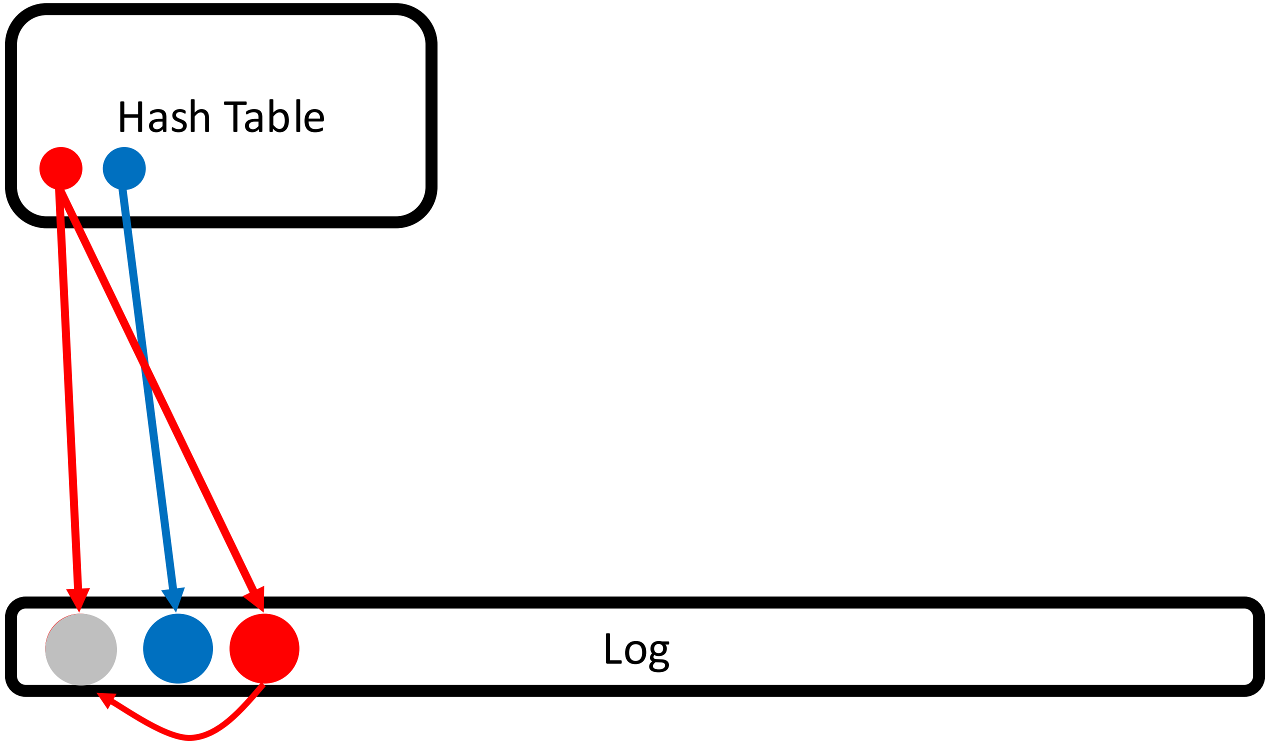


a record is considered valid after the hash table pointer is updated

the log contains ***rows***

BOSTON UNIVERSITY

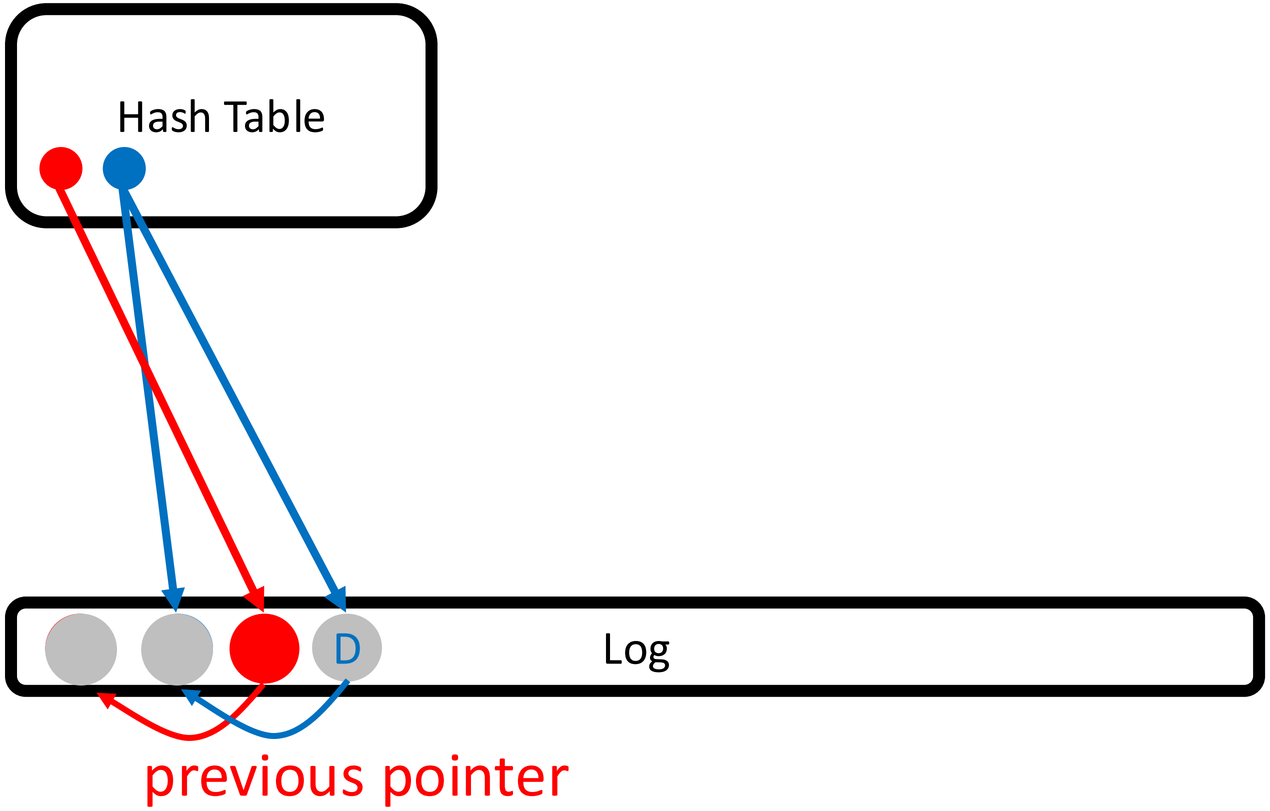
TellStore-Log Update



previous pointer

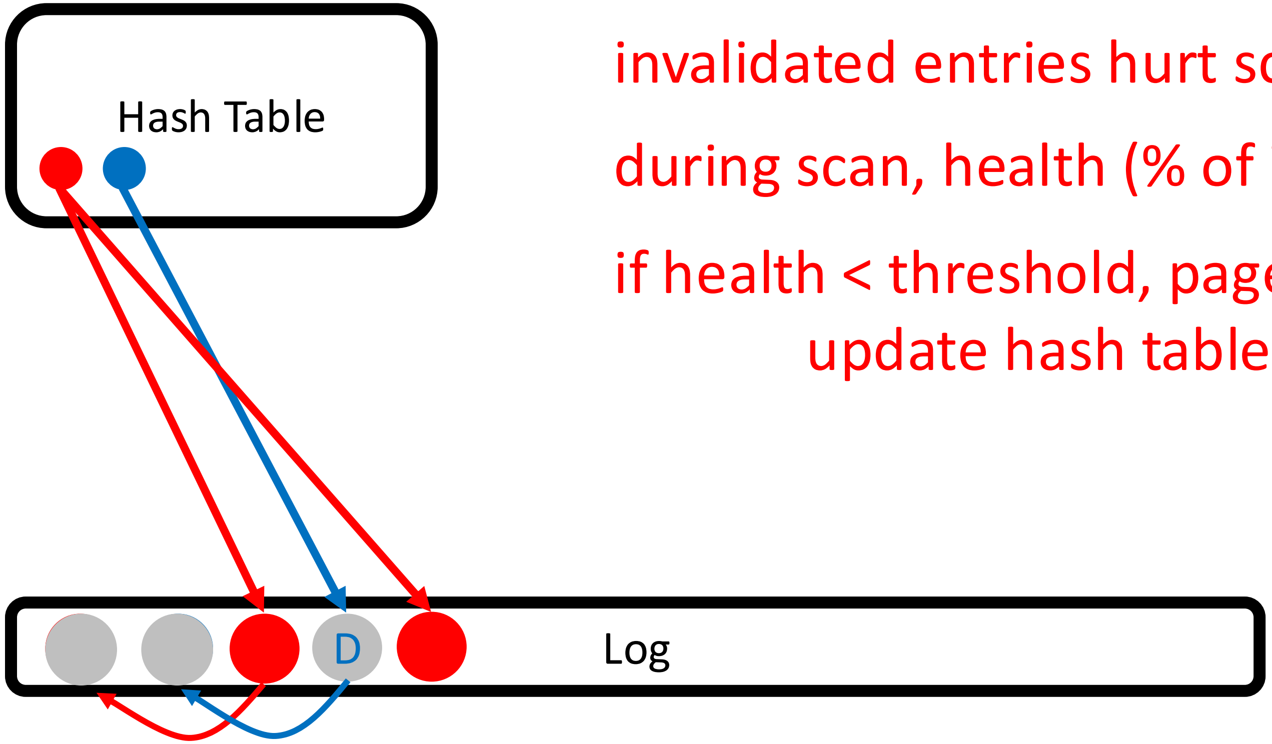
the log contains ***rows***

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***

BOSTON UNIVERSITY

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)

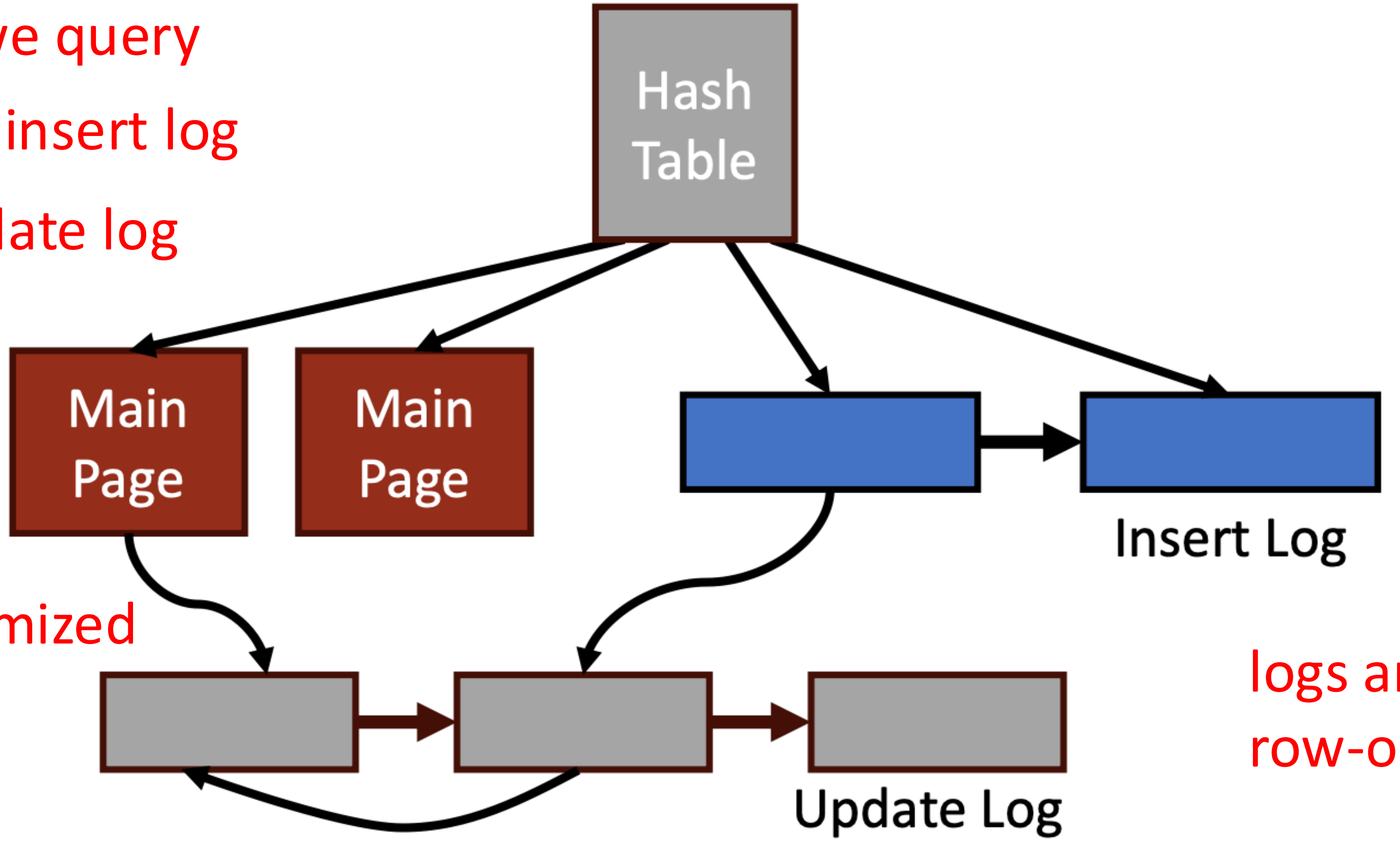
lazy GC: Optimize tables that are scanned four data structures

TellStore-Col

before inserting we query

new entries go to insert log

updates go to update log



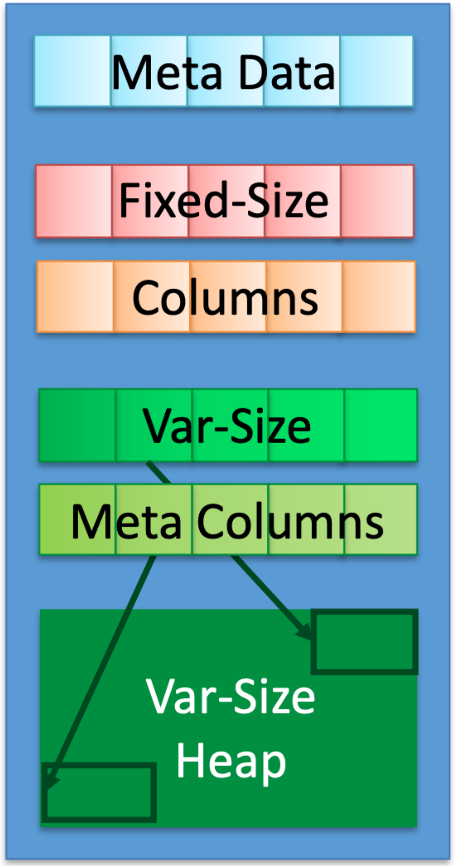
main is read-only

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 indexed in columnar format
but stored in row-wise format

why row-wise?

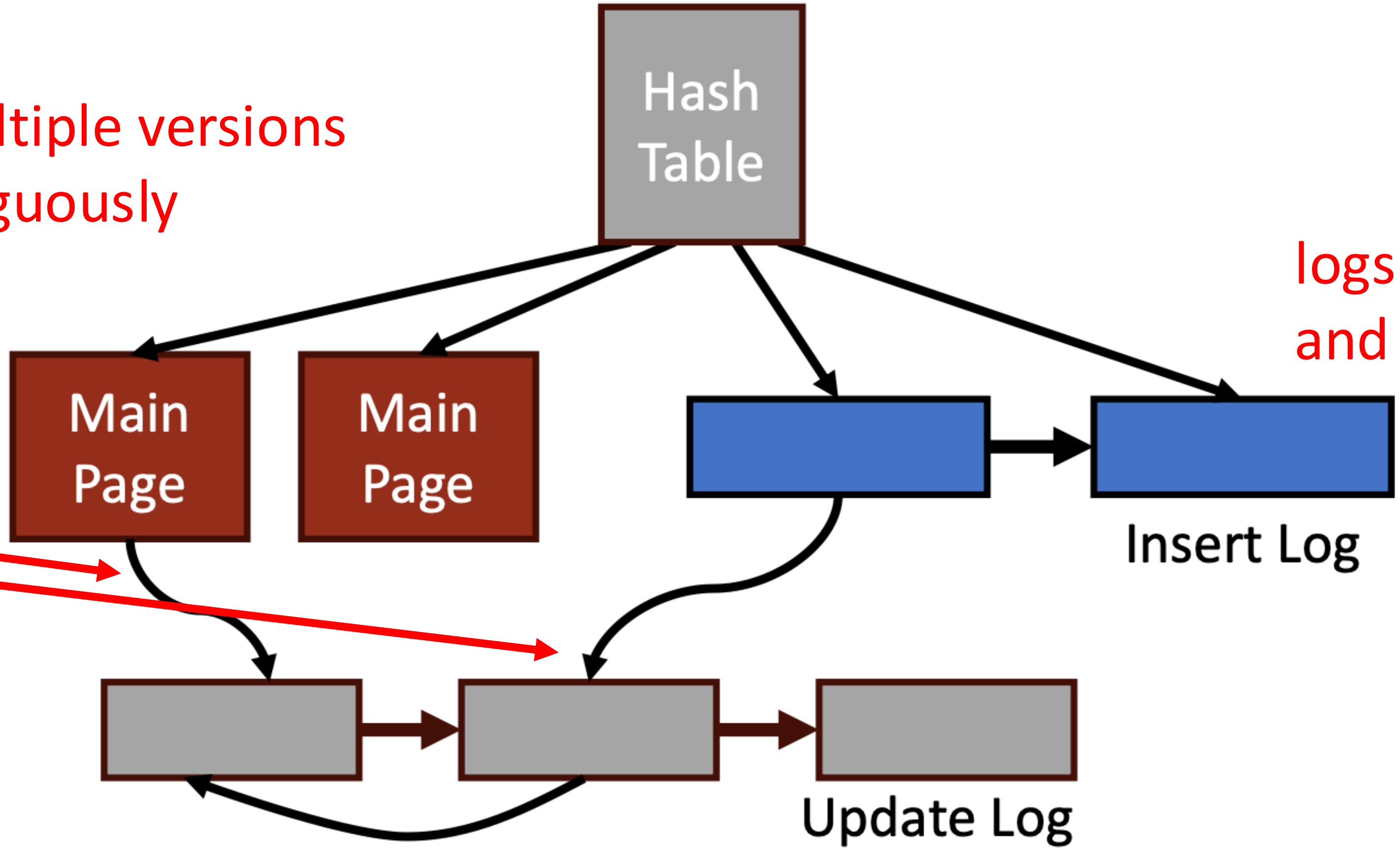
(1) faster materialization (contiguous copying)

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

TellStore-Col Versioning

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

newest pointers
may exist

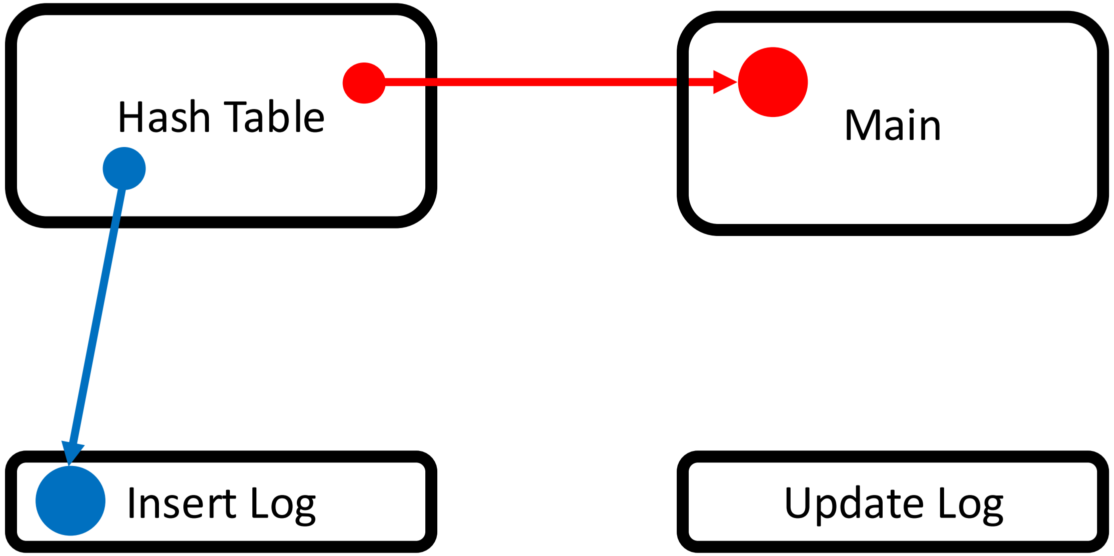


logs work like before
and they are row-wise

previous pointers may
exist only in update log

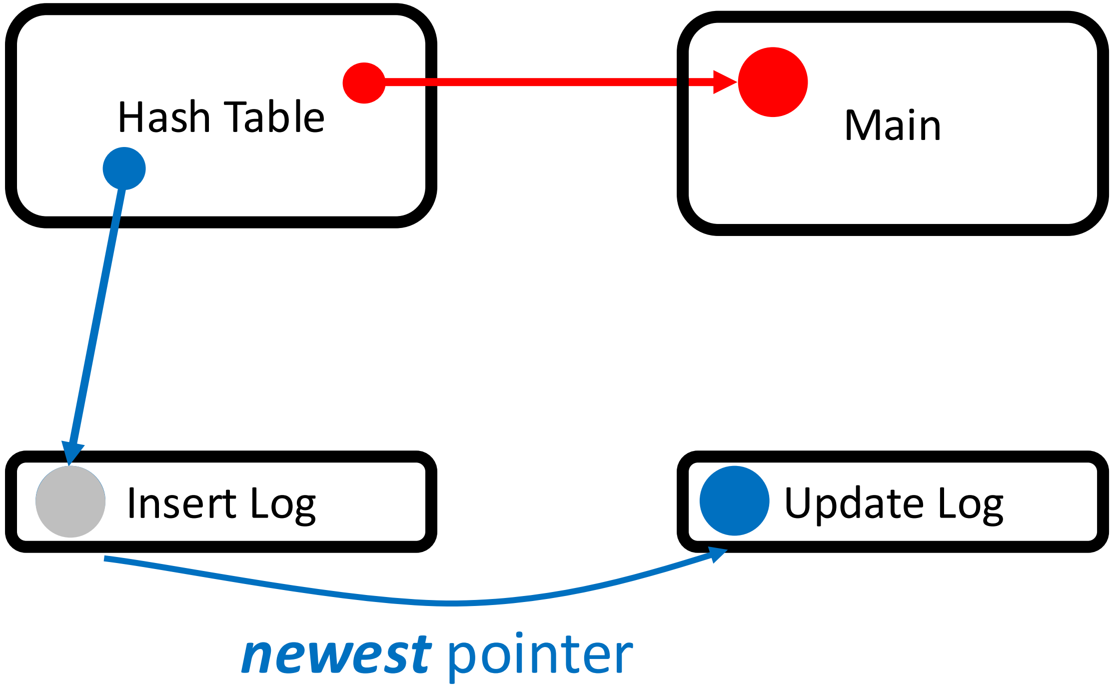
BOSTON
UNIVERSITY

TellStore-Col Insertion

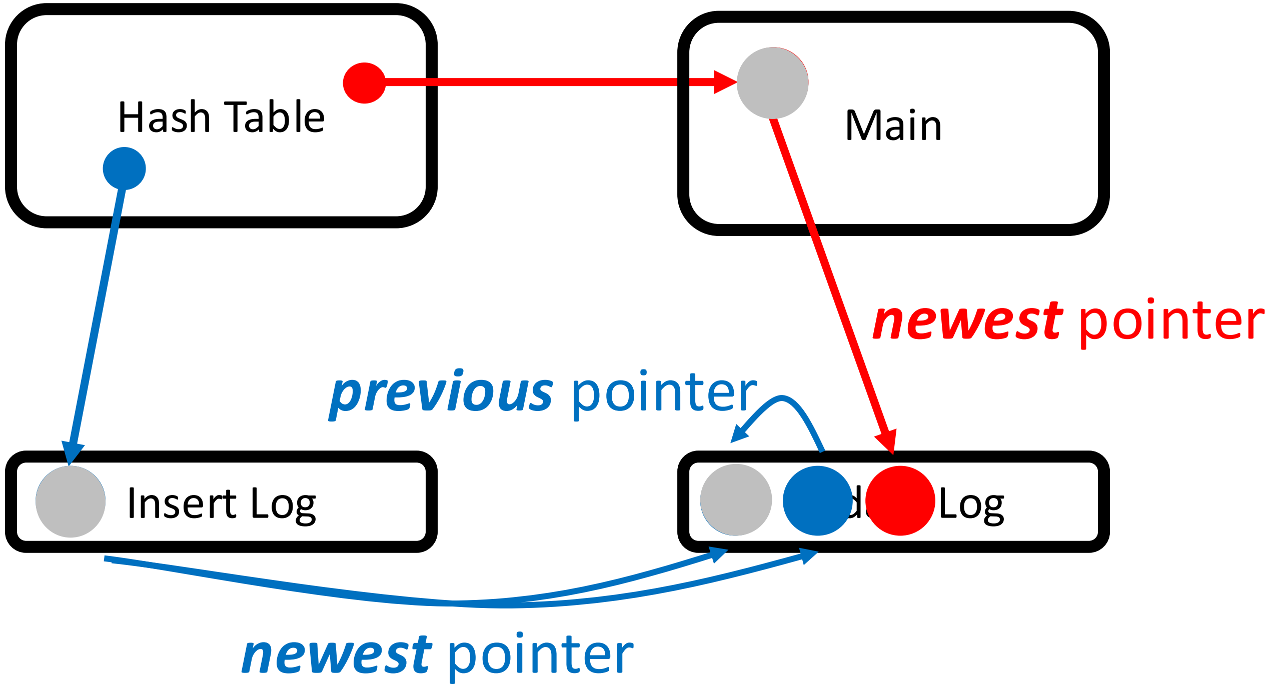


BOSTON UNIVERSITY

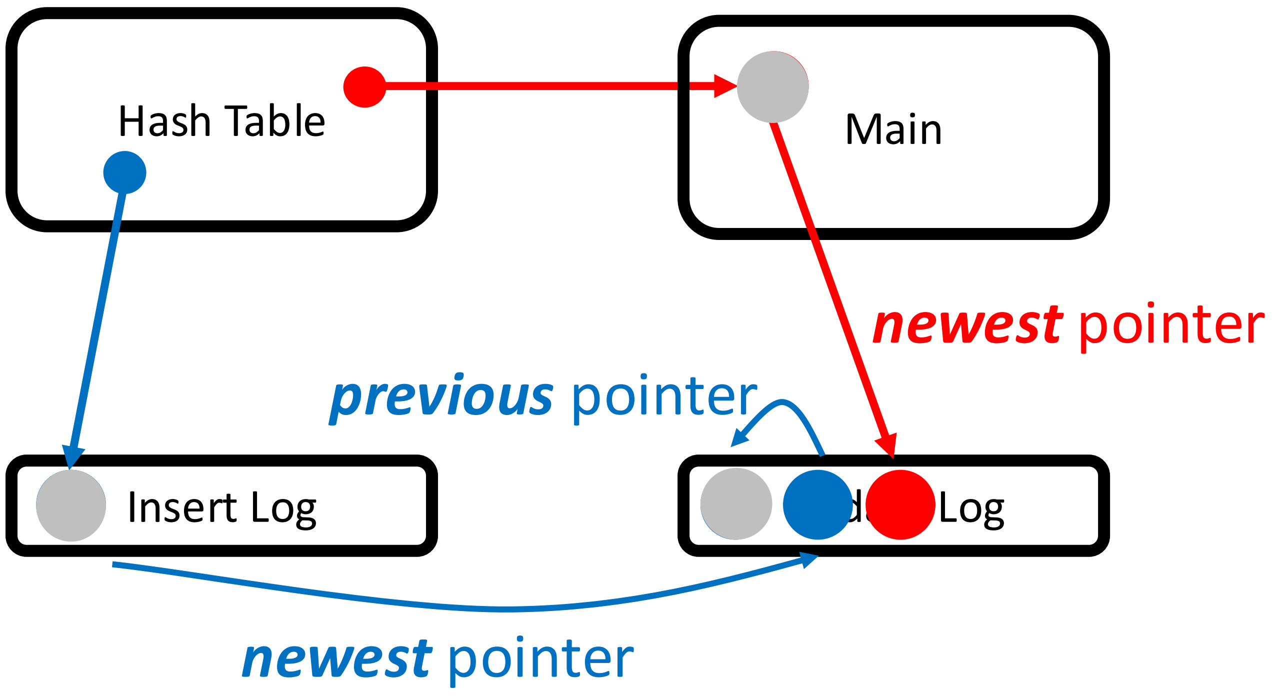
TellStore-Col Update



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

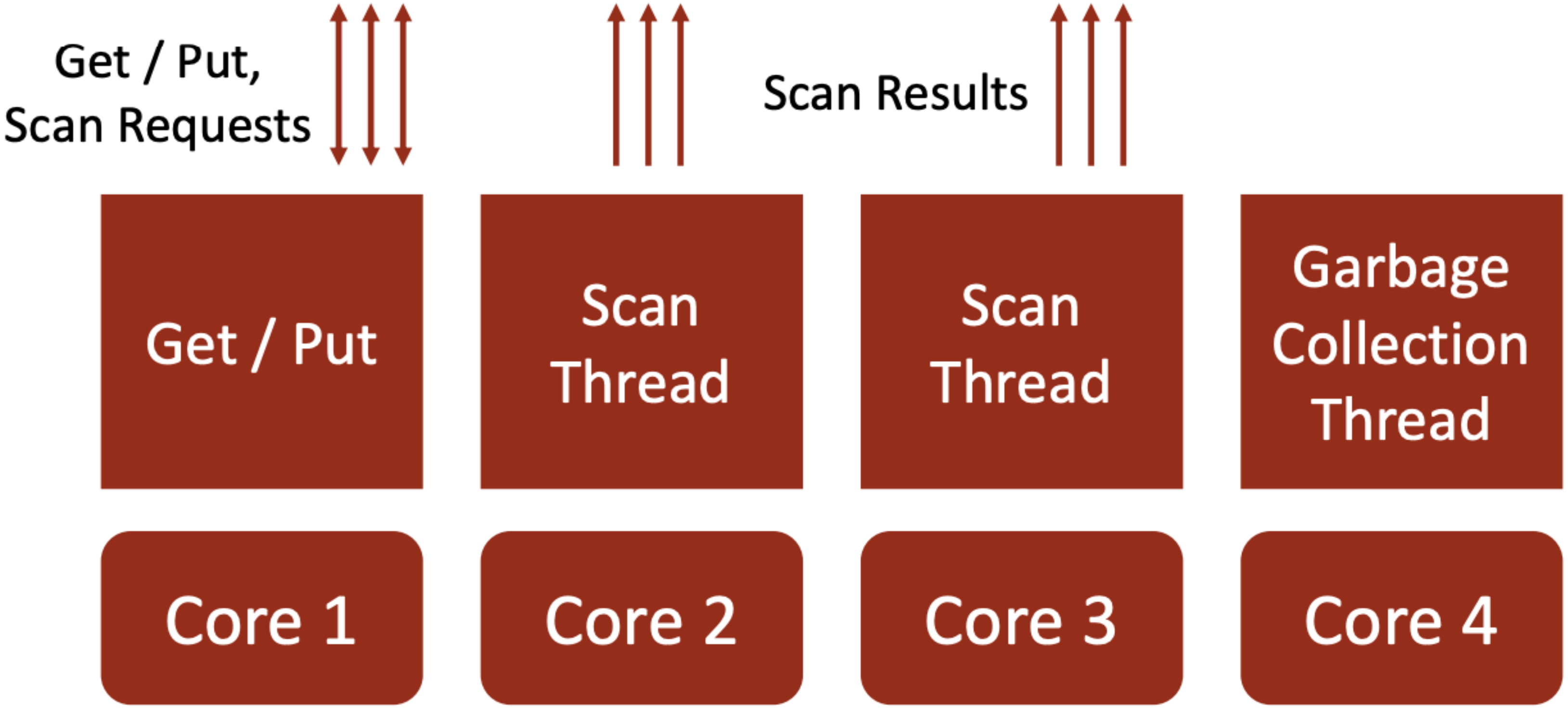
separate insert/update logs: efficient GC

eager GC: improve scans

Implementation Details

scans are assigned to dedicated threads

scan coordinator for shared scans

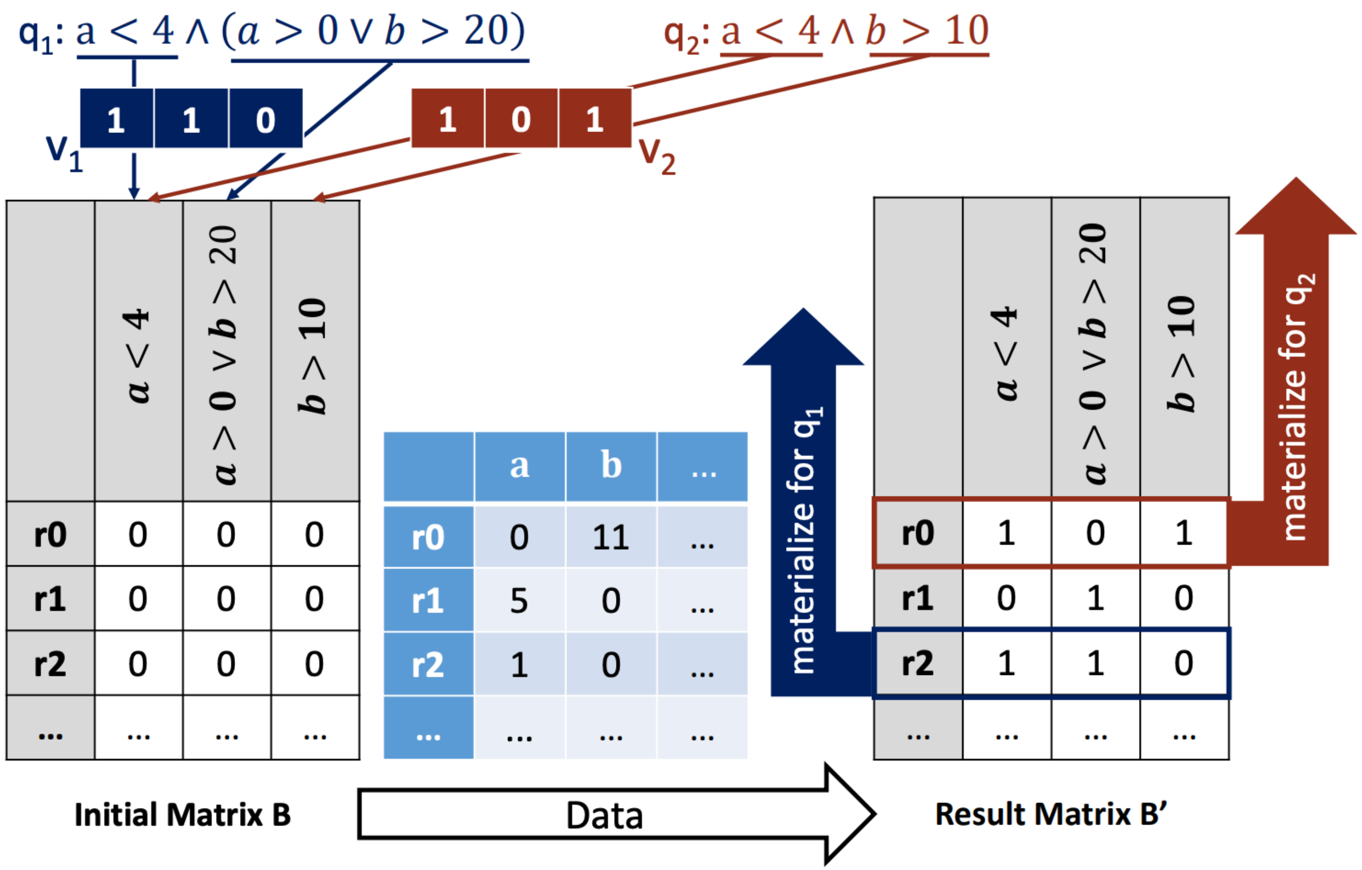


Implementation Details

efficient predicate evaluation via code generation and predicate pushdown

all queries in CNF

reuse work



Yahoo! Cloud Serving Benchmark# (YCSB#)

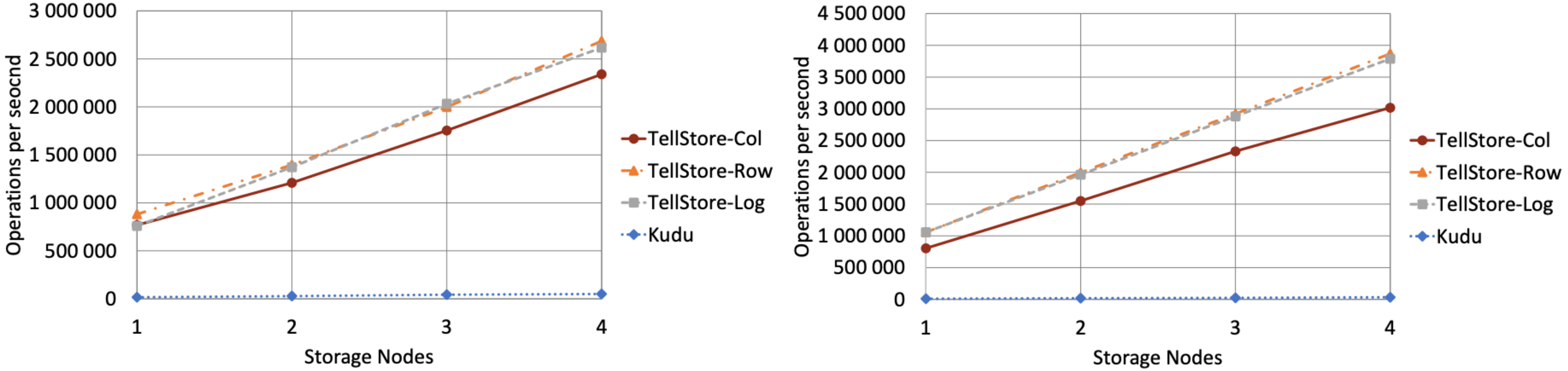
based on YSCB, a put/get benchmark

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
  ```

BOSTON UNIVERSITY

Experiments: Transactional Workload



(a) Get/Put Workload

(b) Get-only workload

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

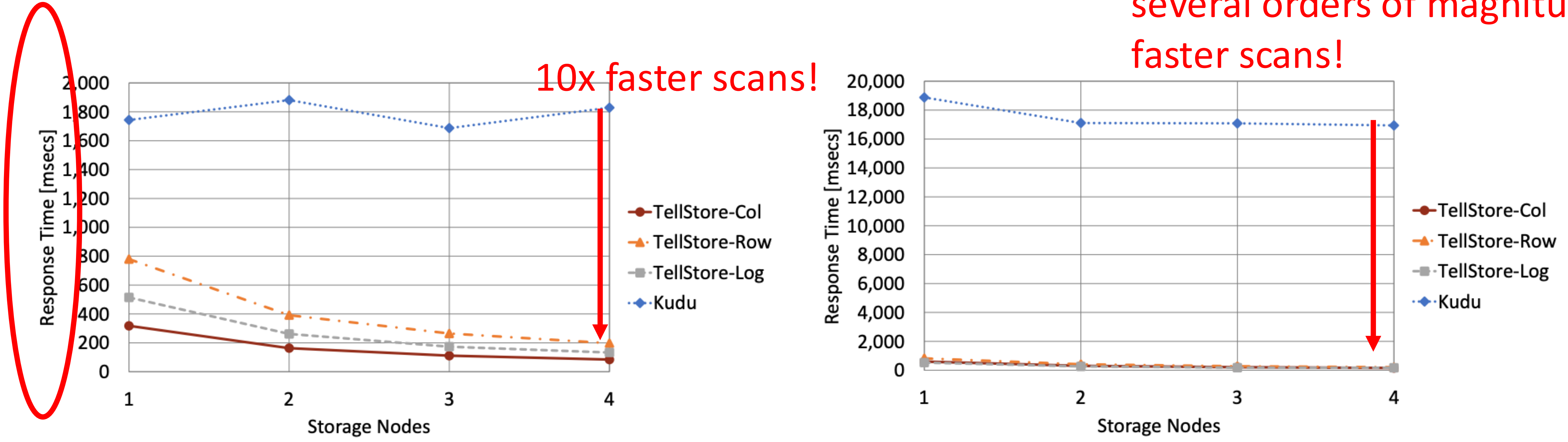
Kudu is used as it was the most competitive to begin with

All TellStore approaches are not that far!

BOSTON UNIVERSITY

Experiments: Scans

several orders of magnitude faster scans!



(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

BOSTON UNIVERSITY

Experiments: Mixed Workload

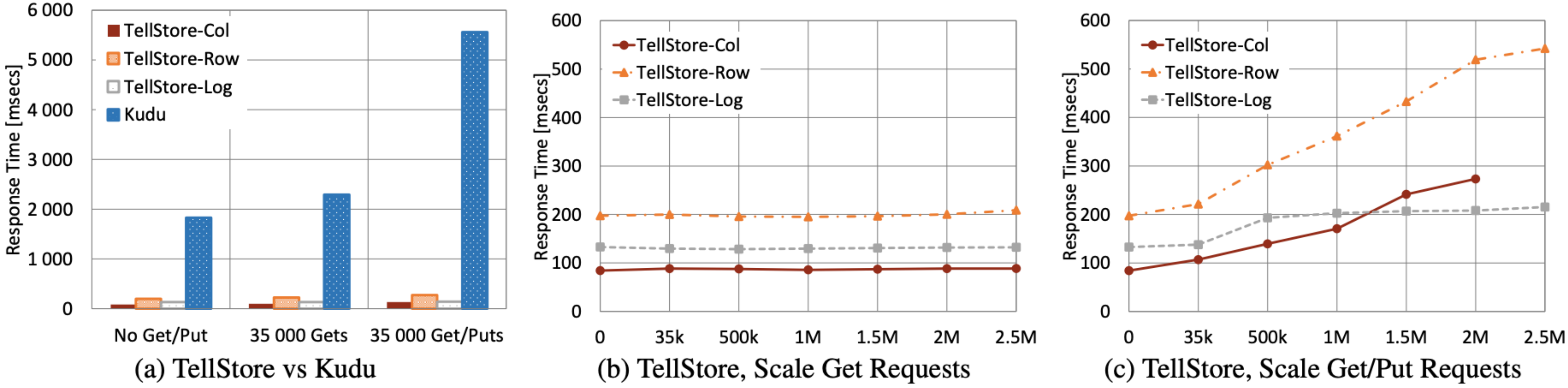


Figure 11: Exp 4, Response Time: YCSB# Query 1, 4 Storage Nodes

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

F2: Designing a Key-Value Store for Large Skewed Workloads

Konstantinos Kanellis*
University of Wisconsin-Madison
kkanellis@cs.wisc.edu

Badrish Chandramouli
Microsoft Research
badrishc@microsoft.com

Shivaram Venkataraman
University of Wisconsin-Madison
shivaram@cs.wisc.edu

A new set of requirements:

- **point** put/get operations (need for high throughput, ideally full in-memory for hot data)
- **larger-than-memory** working set (so full in-memory not possible)
- **high skew** in access patterns (reads/writes)

The target application has **no scans**!

which design should they follow?

Key-value stores

Log-structured Merge (LSM) Trees

- Handle *larger-than-memory* workloads
- Organized in levels; first is in-memory
- Support both *point* & range queries
- Avoid I/O by employing (Bloom) *filters*
- Judicious use of main memory

Point-optimized Stores

- Focus on use-cases like *web caching*
- Large *in-memory* index structures
- *Latch-free* concurrent designs
- Saturate I/O (even for NVMe SSDs)
- Very *high* throughput (>1M ops/sec)

RocksDB

SPLINTERDB

KVell: the Design and Implementation of a Fast Persistent Key-Value Store

Baptiste Lepers
University of Sydney

Karan Gupta
Nutanix

Abstract
Modern block-addressable NVMe SSDs provide much higher bandwidth and similar performance for random and sequential access. Persistent key-value stores (KVs) designed for earlier storage devices, using either Log-Structured Merge (LSM) or B trees, do not take full advantage of these new devices. Logic to avoid random accesses, expensive operations for keeping data sorted on disk, and synchronization

FASTER: A Concurrent Key-Value Store with In-Place Updates

Badrish Chandramouli[†], Guna Prasaad[‡*], Donald Kossmann[†], Justin Levandoski[†], James Hunter[†], Mike Barnett[†]

[†]Microsoft Research [‡]University of Washington

badrishc@microsoft.com, guna@cs.washington.edu, {donaldk, justin.levandoski, jahunter, mbarnett}@microsoft.com

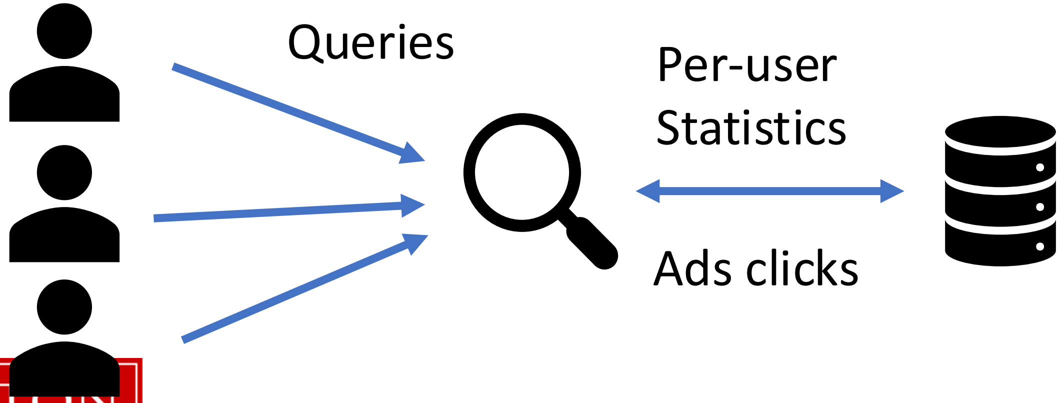
ABSTRACT
Over the last decade, there has been a tremendous growth in data-intensive applications and services in the cloud. Data is created on a variety of edge sources, e.g., devices, browsers, and servers, and processed by cloud applications to gain insights or take decisions. Applications and services either work on collected data, or monitor and process data in real time. These applications are typically update intensive and involve a large amount of state beyond what can fit in main memory. However, they display significant temporal locality

1.1 Challenges and Existing Solutions
State management is an important problem for all these applications and services. It exhibits several unique characteristics:
- ***Large State:*** The amount of state accessed by some applications can be very large, far exceeding the capacity of main memory. For example, a targeted search ads provider may maintain per-user, per-ad and clickthrough-rate statistics for billions of users. Also, it is often cheaper to retain state that is infrequently accessed on secondary storage [18], even when it fits in memory.

Real-world, large *skewed* workloads

- Point queries and high throughput *paramount*
- Working sets *larger* than main-memory – most data rarely accessed or updated
- Total indexed data order of magnitude *larger* than main-memory
- Natural skew in key access pattern – both for *reads* and *writes*
- Memory resources scarce – disk wear is a practical *concern*

Search Engine Workload



Insert / Update statistics (e.g., clicks, statistics)

Millions of active users at any time – *critical* path

Many more *inactive,* but we still need to keep data!

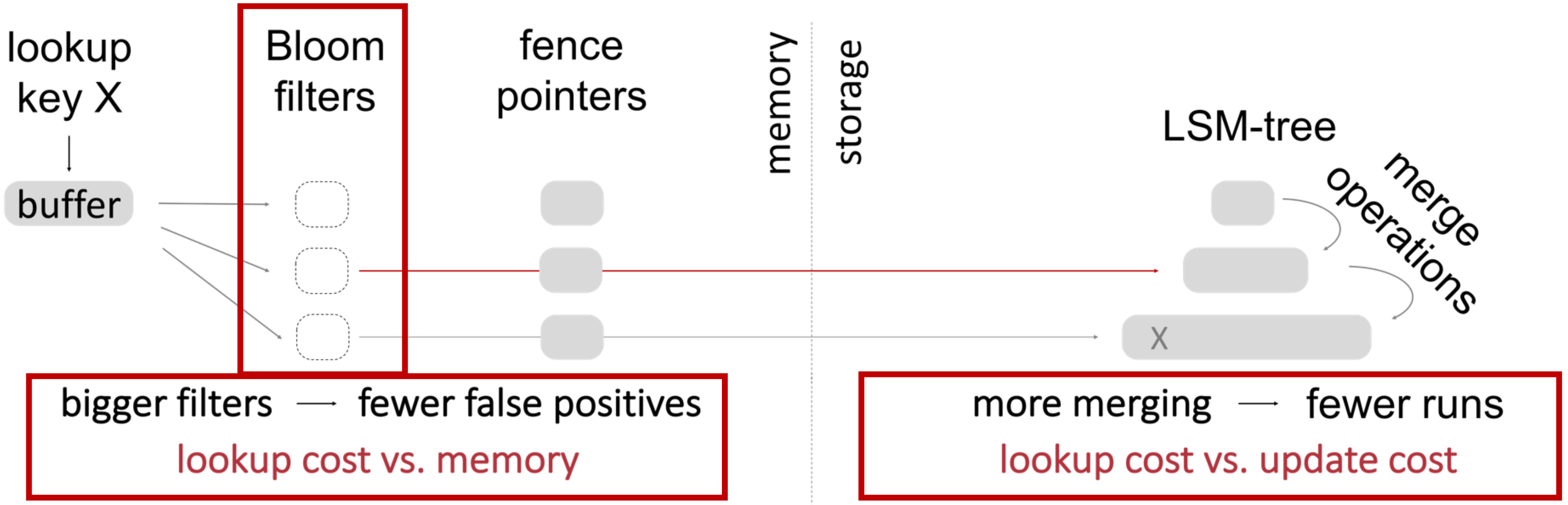
We fetch clicks for active users (during browsing)
We count viewed ads, from active users

Limitations of Existing Systems (1)

Log-structured Merge (LSM) Trees

- \+ Enable and tune (Bloom) *filters*
- \+ Use hash indices
- \+ Efficient compaction policies

- – Filters may no longer fit in memory
- – CPU overhead (10s of filters / query)
- – Need tuning



[1] Dayan et. al. Monkey: Optimal Navigable Key-Value Store (SIGMOD'17)

Limitations of Existing Systems (2)

KVell

- Adjust page-cache size
  – Large in-memory B-tree (19B per key)
  – B-tree continuously paged out to disk

FASTER

- Tune record log in-memory size / hash index
  – Reducing index size increases I/O ops
  – Log compaction “pollutes” in-memory log

BOSTON UNIVERSITY

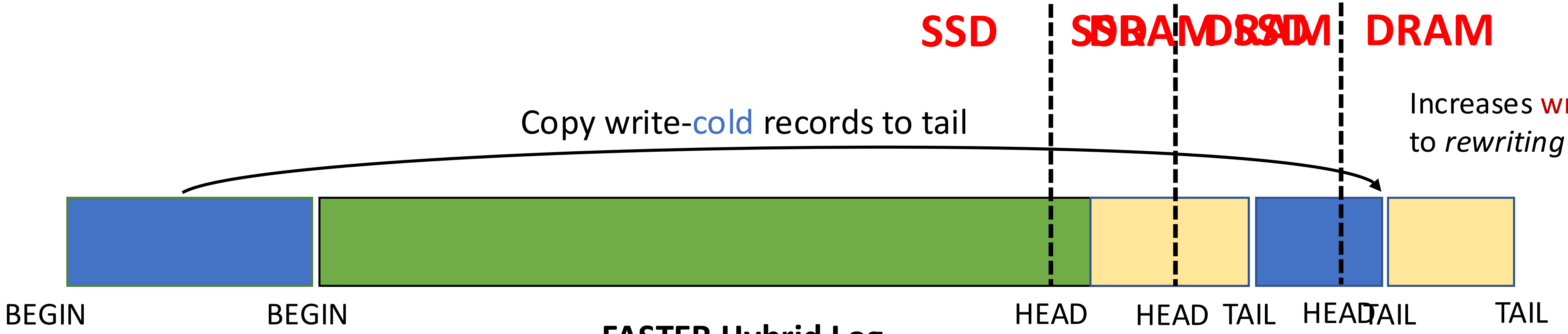
Limitations of Existing Systems (2)

KVell

- Adjust page-cache size
- – Large in-memory B-tree (19B per key)
- – B-tree continuously paged out to disk

FASTER

- Tune record log in-memory size / hash index
- – Reducing index size increases I/O ops
- – Log compaction "pollutes" in-memory log



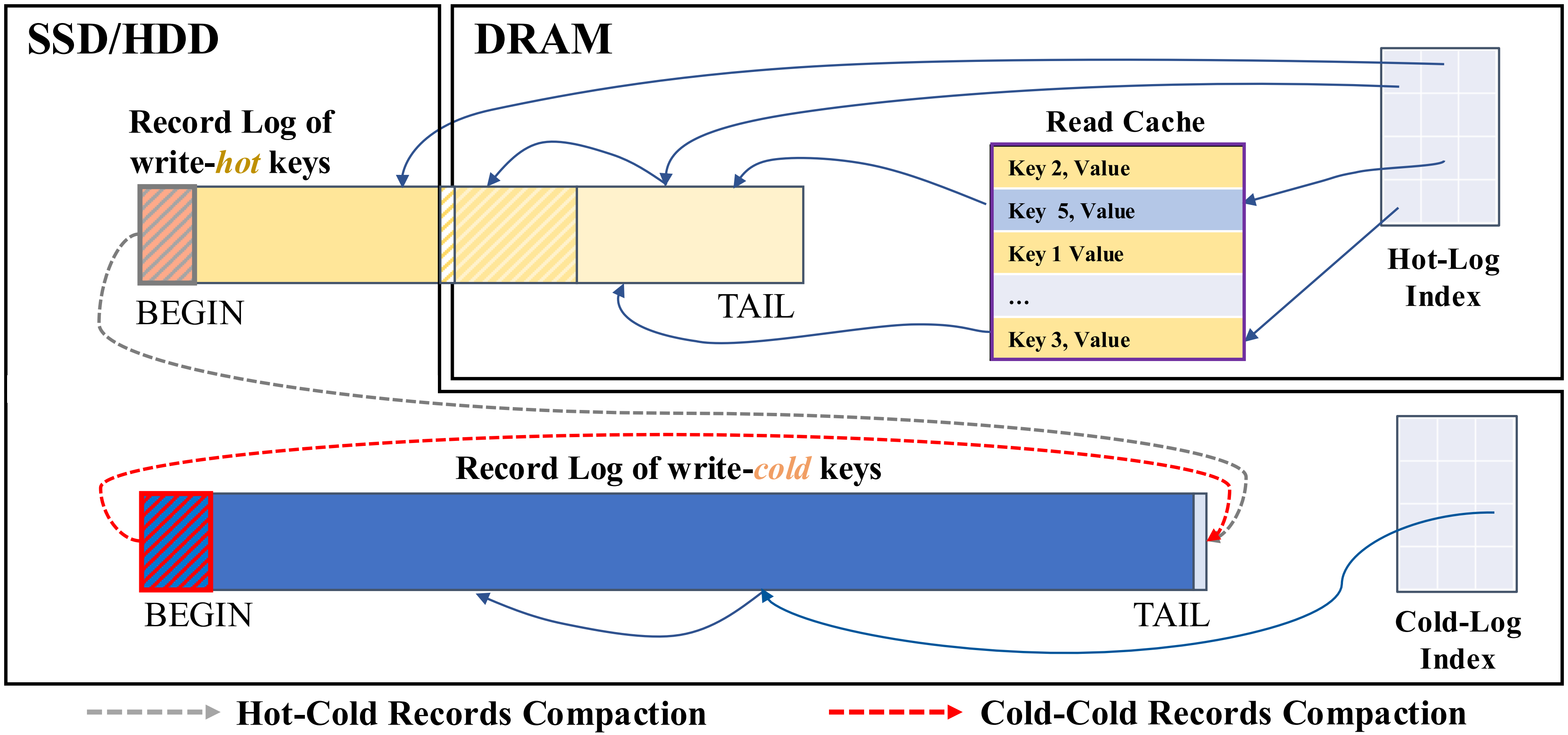
Increases write-amplification due to *rewriting* of records

FASTER Hybrid Log

New records compete with old *compacted* records for a spot at *in-memory* regions

Introducing *F2*

Key Idea: separate management of records across *both* read/write and hot/cold domains



BOSTON UNIVERSITY

Design Space

Updates × Layout × Versioning × GC

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

similar to TellStore-Log, but with **periodic compaction**

compaction aggressively optimized

F2 – Read Cache

Contains *read*-hot records of both hot log and cold log

- Hash index entries can point to *either* read cache entries or (**hot**) record log (single bit in address)

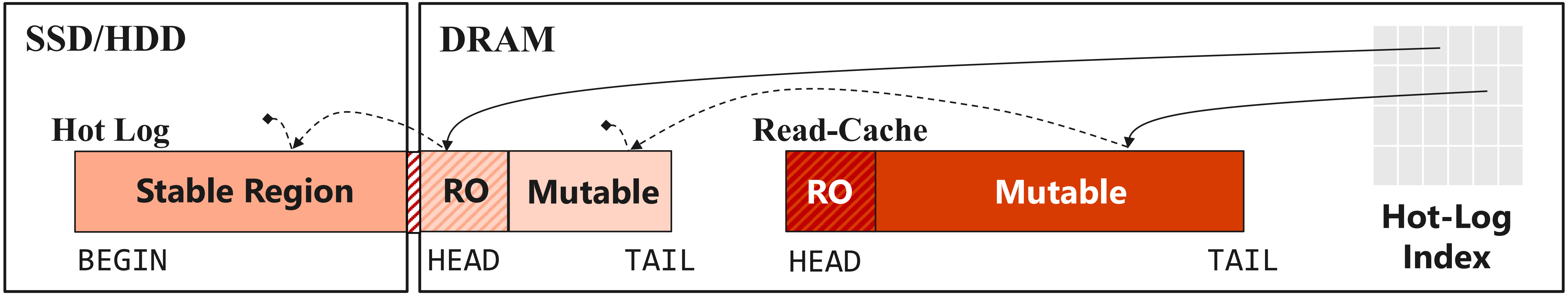
Reads go from **hot**-log index → (optionally) read cache → **hot** log

Reads from **cold**-log are *always* inserted at the tail of the read cache

Upserts and **RMWs** write *directly* to **hot** log tail, eliminating read cache chain

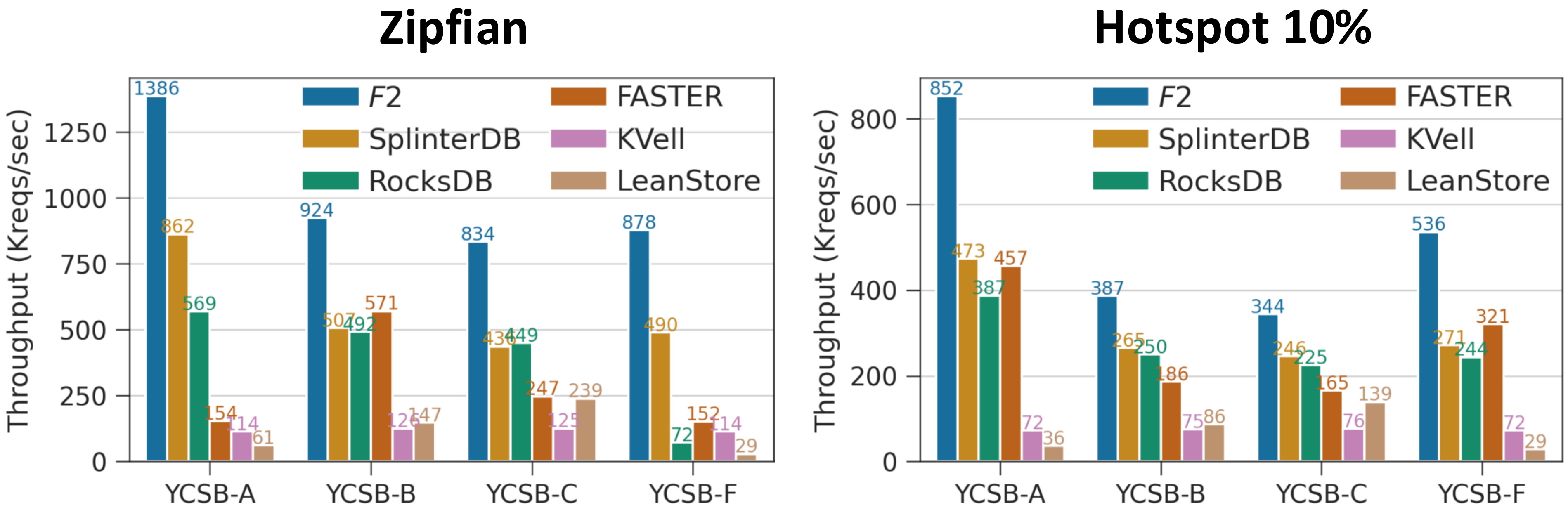
- If record with same key in read cache, it's **invalidated**!

Periodically, read cache is *evicting* in-memory records (HEAD), by altering the hash chains



F2 – Performance Comparison

YCSB: 250M keys, 8B keys, 100B values; **3GiB** mem budget (**10%** of dataset size); 24 threads, NVMe SSD

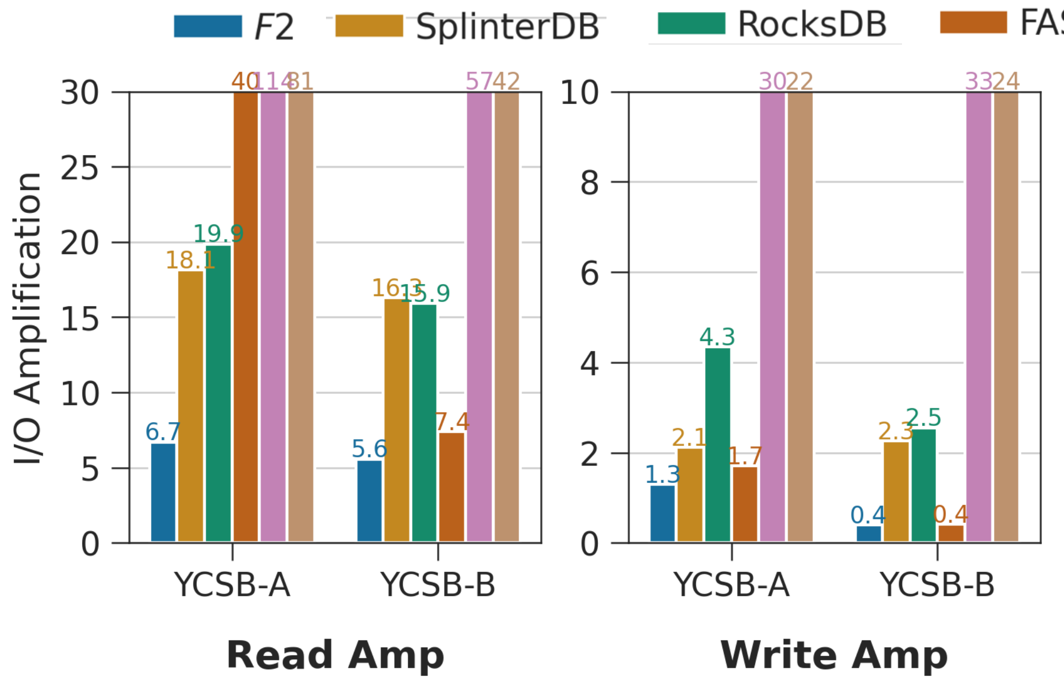


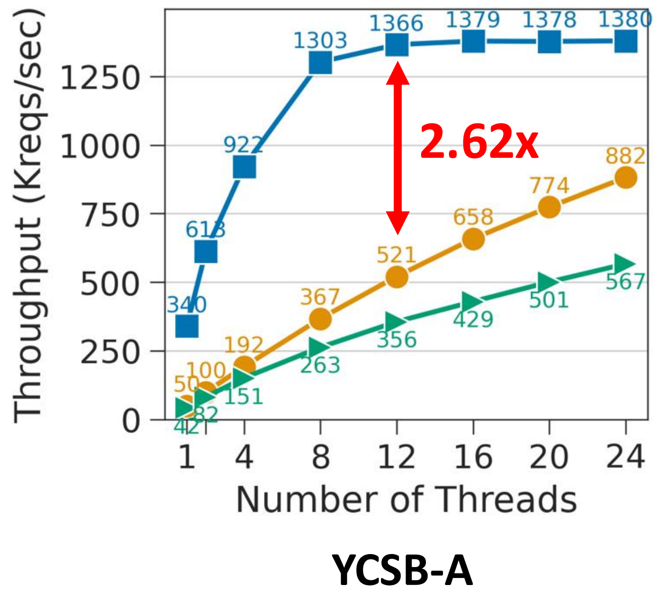
Zipfian: SplinterDB (1.78x), RocksDB (4.61x), FASTER (4.94x)

Hotspot 10%: SplinterDB (1.66x), RocksDB (1.88x), FASTER (1.92x)

F2 – I/O Amplification & Scalability Comparison

YCSB: 250M keys, 8B keys, 100B values; **3GiB** mem budget (**10%** of dataset size); 24 threads, NVMe SSD





F2 achieve less WA than Baselines

F2 saturates at 8-12 threads

→ Varying memory-budget, YCSB skewness; F2 detailed evaluation ←

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 7

Design Tradeoffs in Key-Value Stores

Prof. Manos Athanassoulis

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