BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 9

Key-Value Stores for Concurrent and Point Accesses

Prof. Manos Athanassoulis

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

Key-Value Stores with In-Place Updates

FishStore: Faster Ingestion with Subset Hashing

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

Microsoft Research

Why do we discuss those papers?

Different requirements and workload *than prior approaches*

BOSTON UNIVERSITY

Up to now we focused on **mixed workload**
(inserts, point queries, range queries, updates, deletes)

Lethe

What was the design?

Log-Structured Merge Trees: to support **range** queries

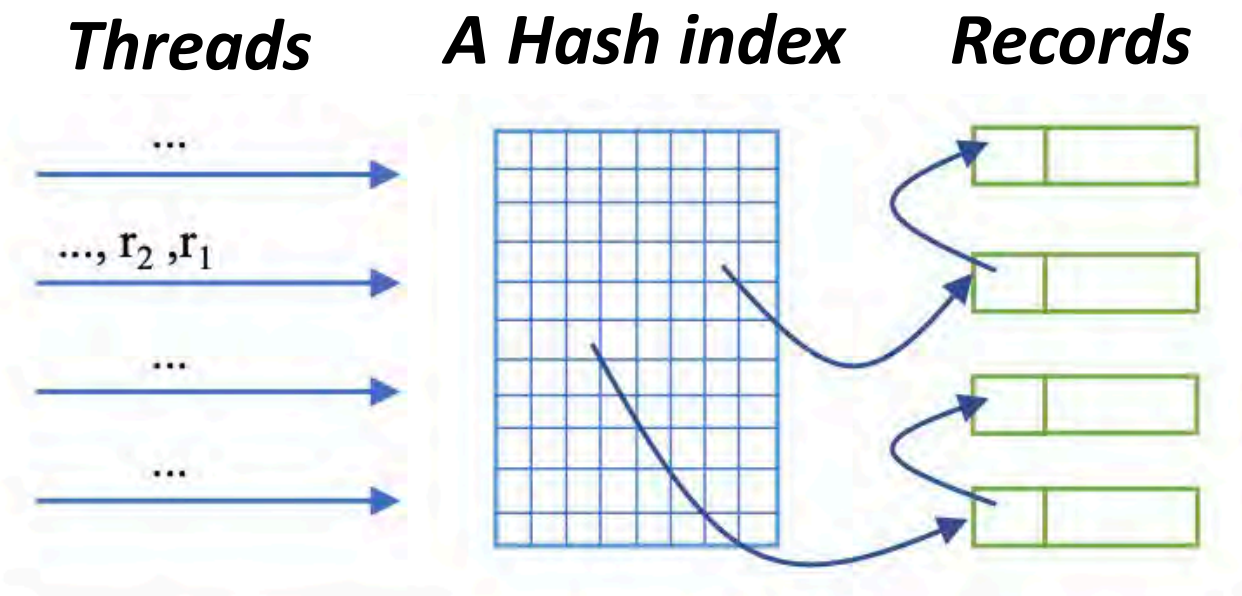
What if we have no range queries?

What if we have intensive updates on existing keys?

What if we have a huge dataset but a small working set?

What if we want to support multiple threads updating at the same time?

What data structure to use?



Threads can access concurrently

No need to spend time sorting

Quick access to any record

Where to store the records?

	Concurrent	*Larger-than-memory*	*In-place updates*
In-Memory	✅		✅
Append-only Log (storage)	✅	✅	
Hybrid Log (mem+storage)	✅	✅	✅

which one to use? all 3!

How to avoid synchronization between threads?

Epochs

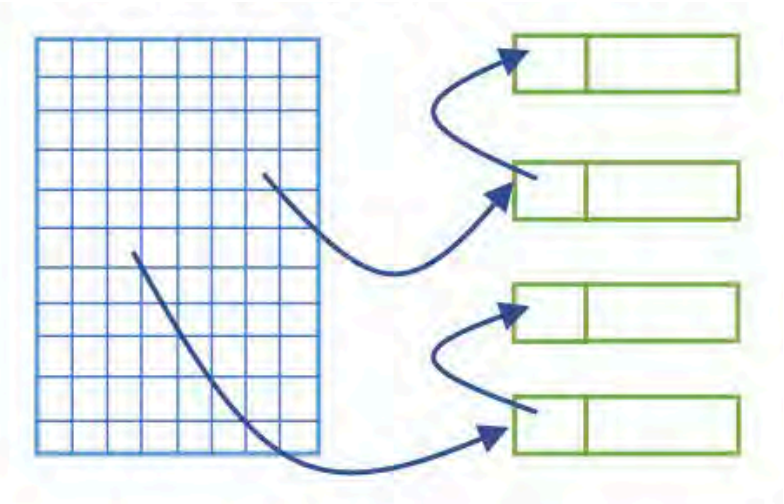
Instead of acquiring locks (which would cause performance problems)

Each thread operates in an epoch e_i

When all threads are past a specific epoch, this is marked as ***safe***

When an epoch becomes ***safe*** then specific actions are ***triggered***

Operating in-memory



Reads: follow the pointer (may have to follow a chain of links)

Updates and Inserts: start by reading,
then either update atomically or insert

Deletes: atomically splice the record from the list

Spilling out of memory

how to handle updates?

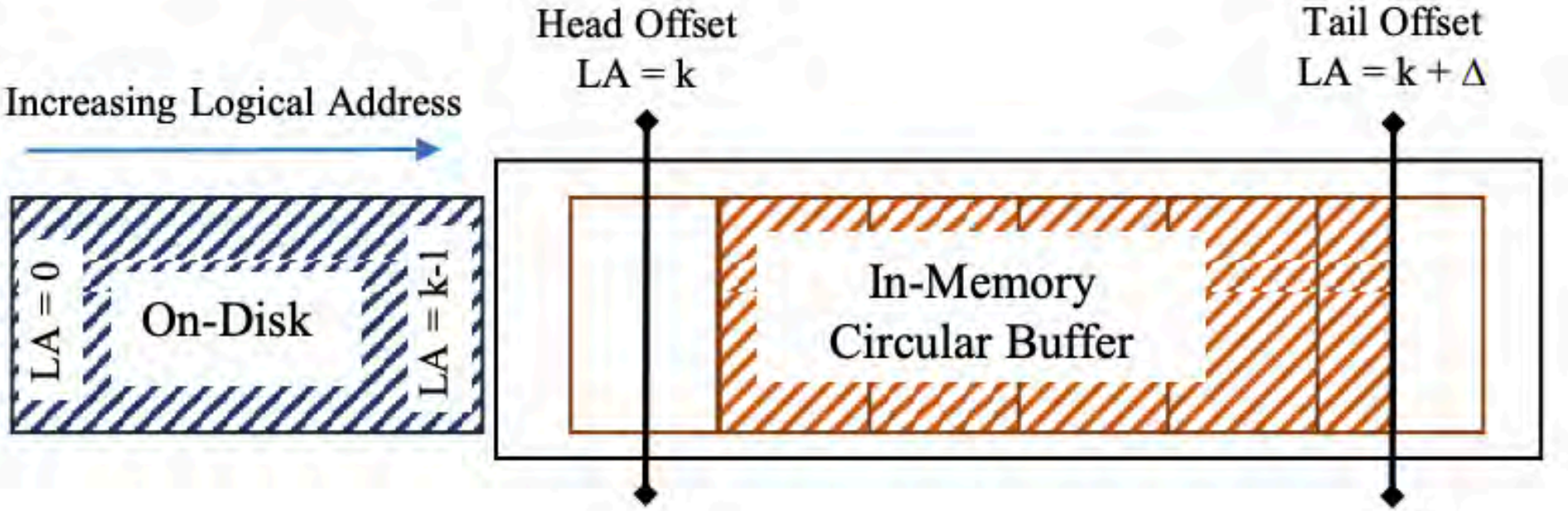
insert + a pointer to the old entry

garbage collection needed

Hash table now stores the ***logical address*** (relevant to the ***tail offset***)

When a page is full, it is marked for flushing to storage

The page is flushed when the epoch (that marked it) becomes safe

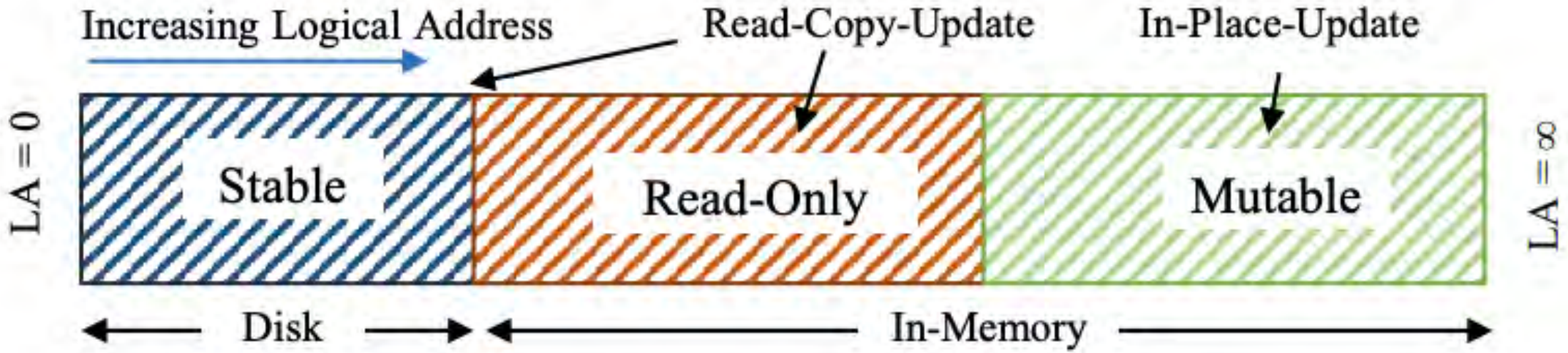


Efficient In-place updates in FASTER

Updating from Stable or Read-Only is a read-modify-write (to mutable)

Further updates are in-place

New inserts go to mutable



Handling multiple threads

Each thread might be in a different epoch

When all agree then it is ok, else, it is fuzzy

In Fuzzy

blind updates are ok

read-modify-write are deferred

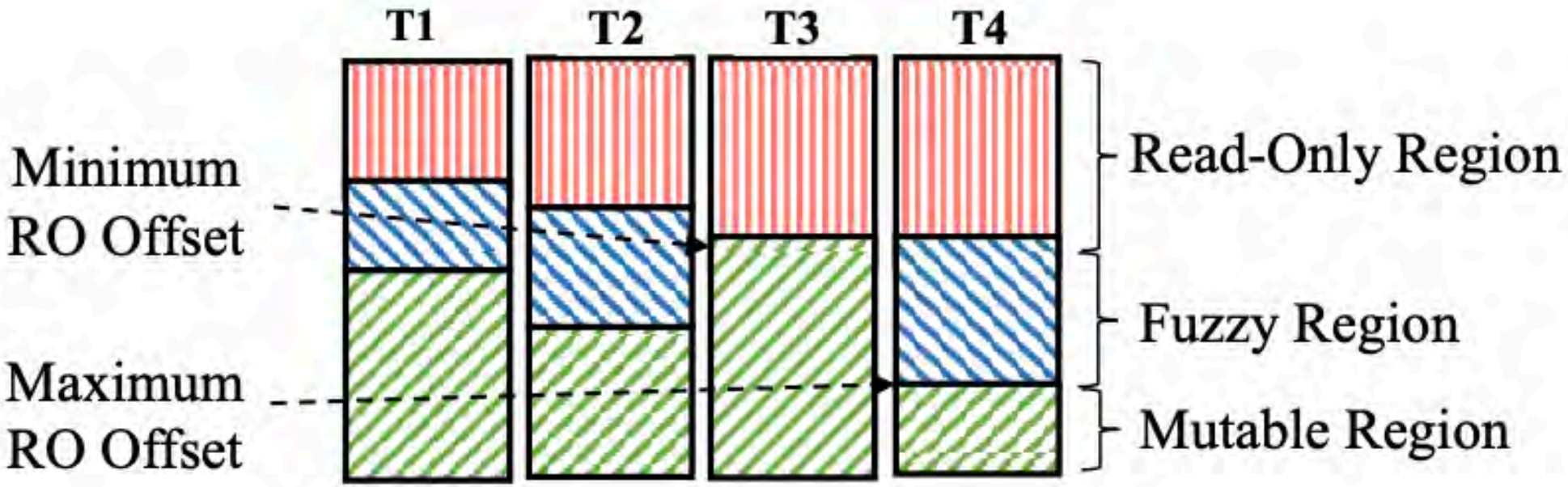


Figure 7: Thread Local View of Hybrid Log Regions

Key-Value Design Depends on Workload!

From LSM to Hash-based logging …

the design depends on the use case!

Range Queries, **Point Queries**, **Working Set Size**, **Update Intensity**

What if we need to know the contents of the value?

In our discussions up to now we focus on the key!

Key	Value

what if we use the value?

what is the value? CSV/JSON

how to use it?

We need to parse it!

JSON/CSV parsing

JSON example:

```
{
   "employee": {
      "name":      "sonoo",
      "salary":     56000,
      "married":    true
   }
}
```

CSV example:

name, salary, married [optional header]

sonoo, 56000, true

Parsing is CPU expensive

But recent approaches can parse 2GB/s/core (selective parsing)!

perfect match for streaming (e.g. telemetry, monitoring)

Faster & Efficient Parsing = FishStore

A new storage layer for **flexible-schema** data.

On-demand indexing over **predicated subset functions** (PSFs).

Group records with the same property in a logical view.

E.g., all records in a population survey whose age > 20.

Faster & Efficient Parsing = FishStore

Fast data ingestion with **minimum effort** parsing & indexing.

Efficient **subset retrieval** over registered properties.

- Fast scan over constructed logical view.
- Support **hybrid scan** over **indexed** and **unindexed** records.

Predicate Subset Functions

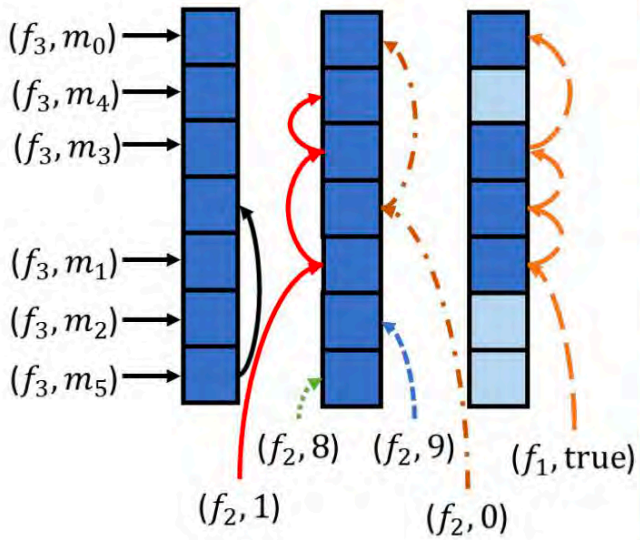
Logically groups records with the **same property**

Given a data source of records in R, a predicated subset function (PSF) is a function $f: R \rightarrow D$ which maps valid records in R, based on a set of fields of interest in R, to a specific value in domain D.

R is the data record collection

D can be a binary value (yes/no) or an arbitrary set of values

PSF Example (Telemetry)



Time	Machine	CPU	MEM
1:00pm	m_0	9.45%	83.52%
1:00pm	m_4	14.67%	57.44%
1:02pm	m_3	10.00%	92.50%
1:03pm	m_5	5.00%	75.32%
1:03pm	m_1	13.45%	90.45%
1:04pm	m_2	93.45%	84.56%
1:05pm	m_5	81.75%	65.03%

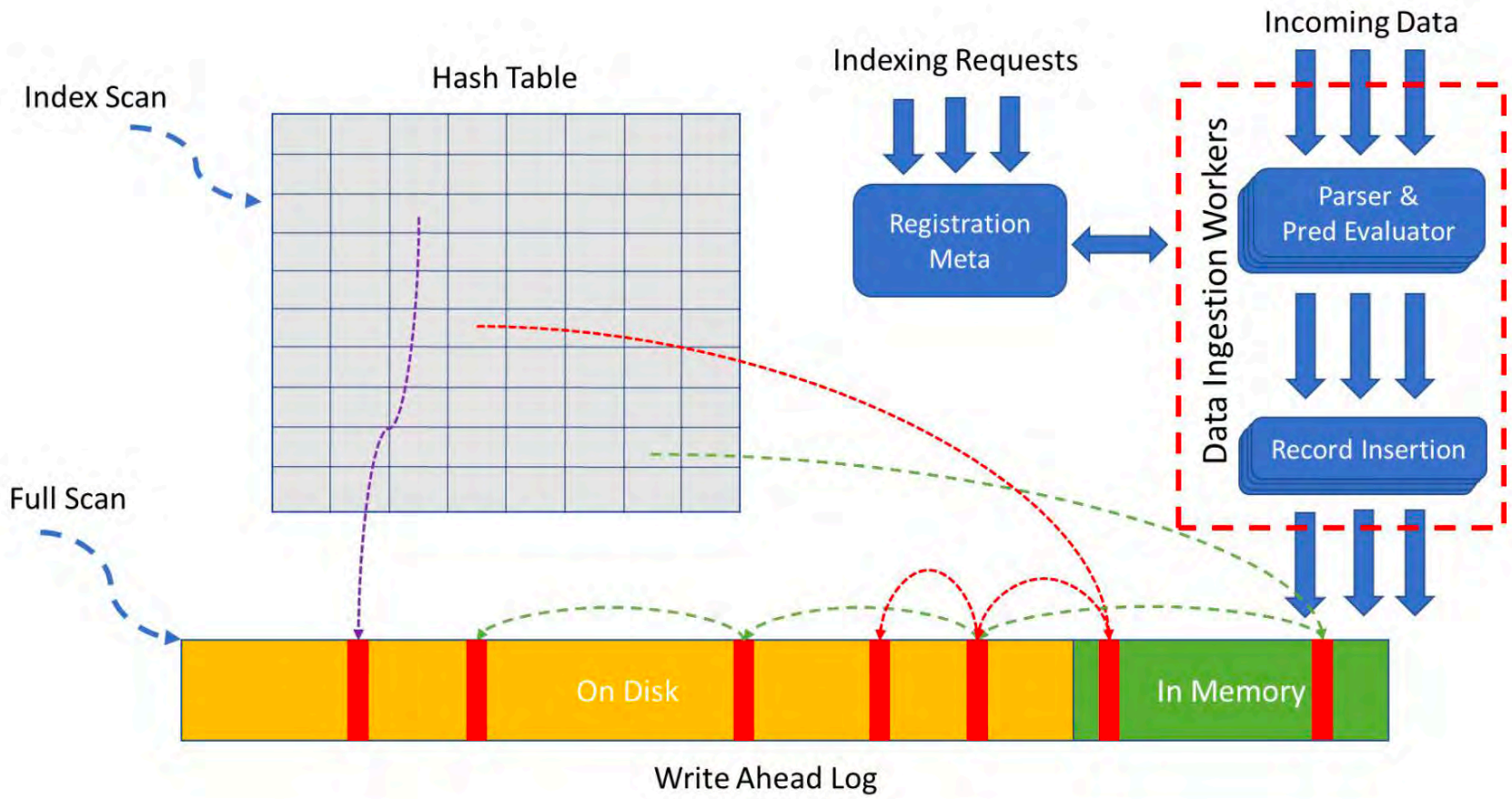
f_1: $\Pi_{\text{CPU}}(r) < 15\%$ & $\Pi_{\text{MEM}}(r) > 75\%$
f_2: $\Pi_{\text{CPU}}(r)$ / 10.0
f_3: $\Pi_{\text{Machine}}(r)$

f_1: Diagnose machines with low CPU, high memory

f_2: Create 10% buckets of CPU range for analysis queries

f_3: Access logs by machine name

FishStore Design



Technical Challenges

Fast concurrent index for PSFs

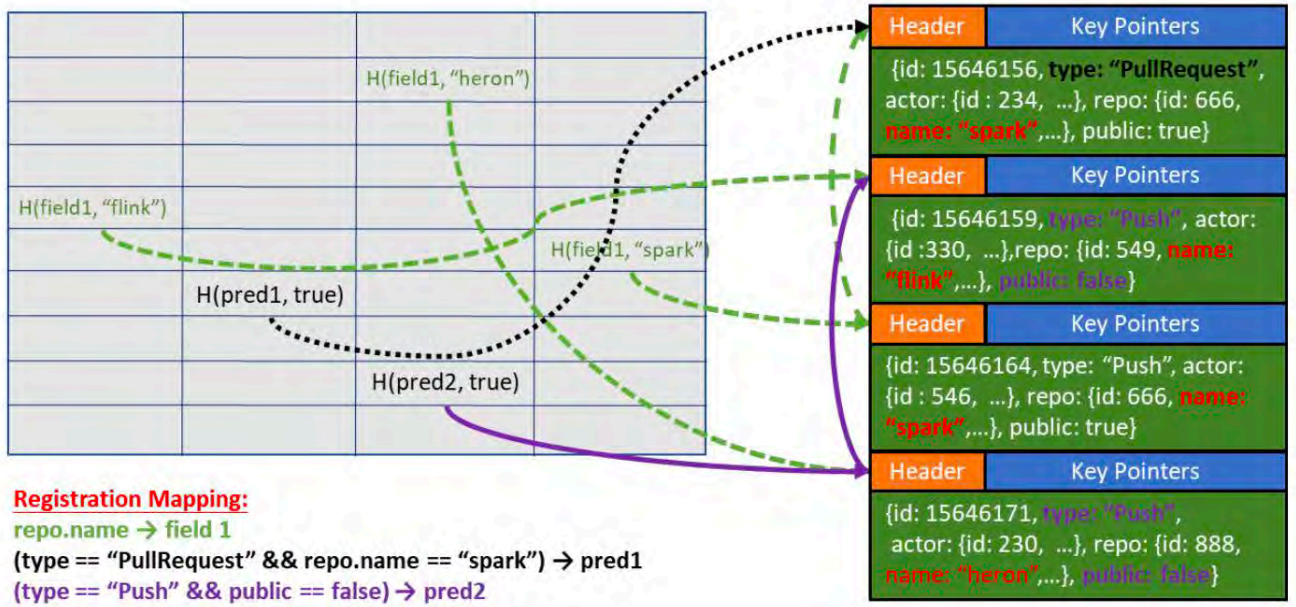
Hybrid Scan (index scan vs sequential scan)

Subset Hash Index

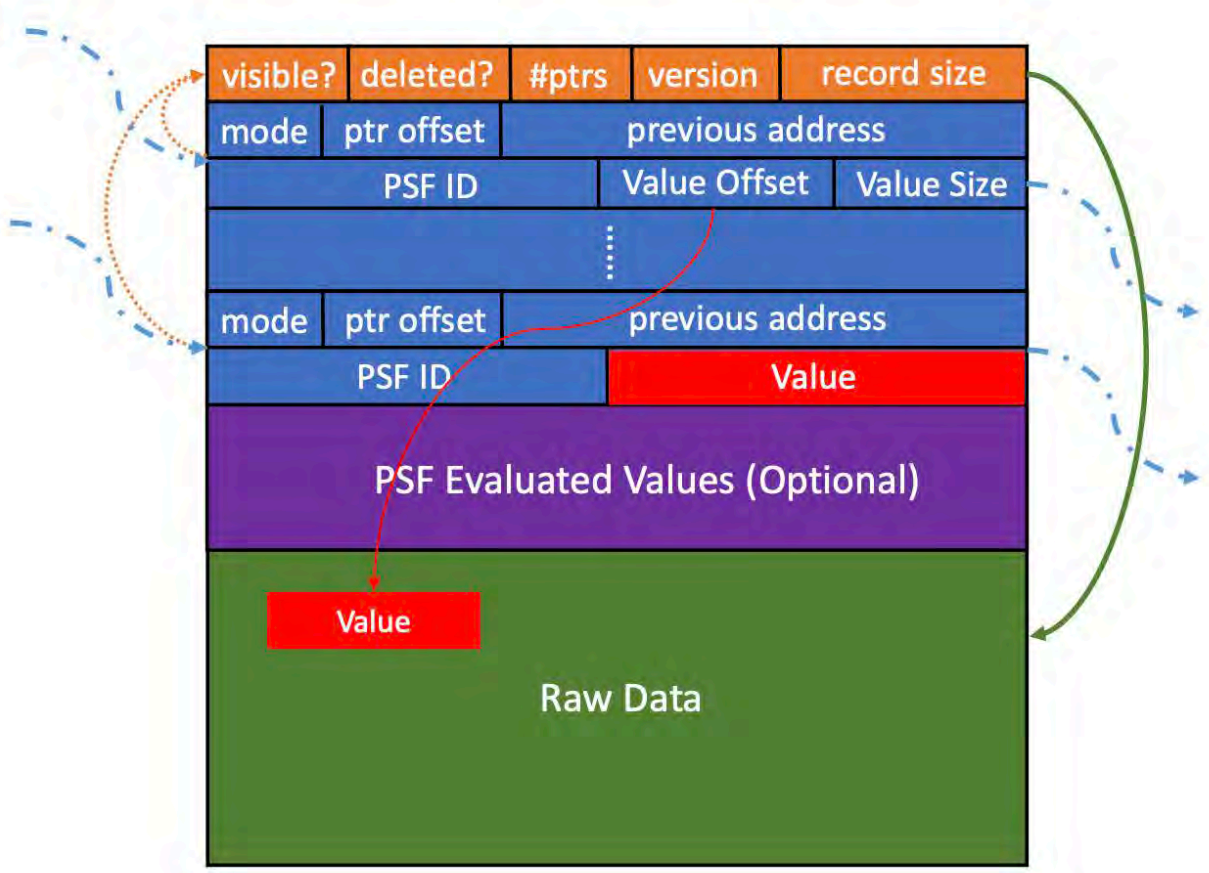
- Hash signature for property (f, v)

$$\mathrm{H}(f, v) = \mathbf{hash}(\mathbf{fid}(f).\mathbf{concat}(v))$$

- Link all records with same property on the same hash chain.



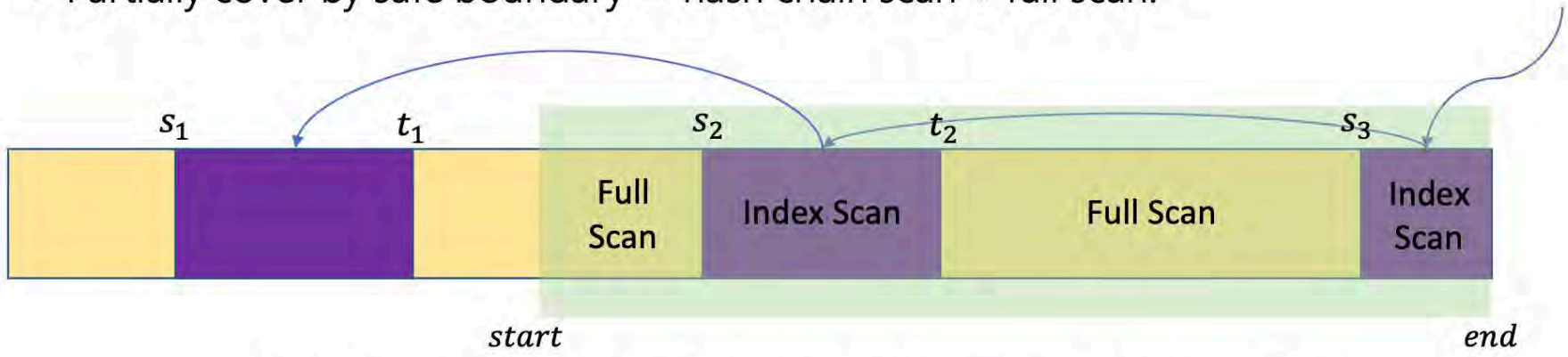
Record Layout



- Fast hash chain traversal
  - Chaining key pointers rather than headers.
- Efficient hash collision check
  - PSF property value embedded in or pointed directly by key pointers.
- Co-locate raw data and index entry.
  - Remove indirect access/IO in record retrieval
- Latch-free index update
  - No forward link restriction.
  - Need to update multiple hash chains for each record.
  - New wait-free index update technique

Subset Retrieval: Hybrid Scan

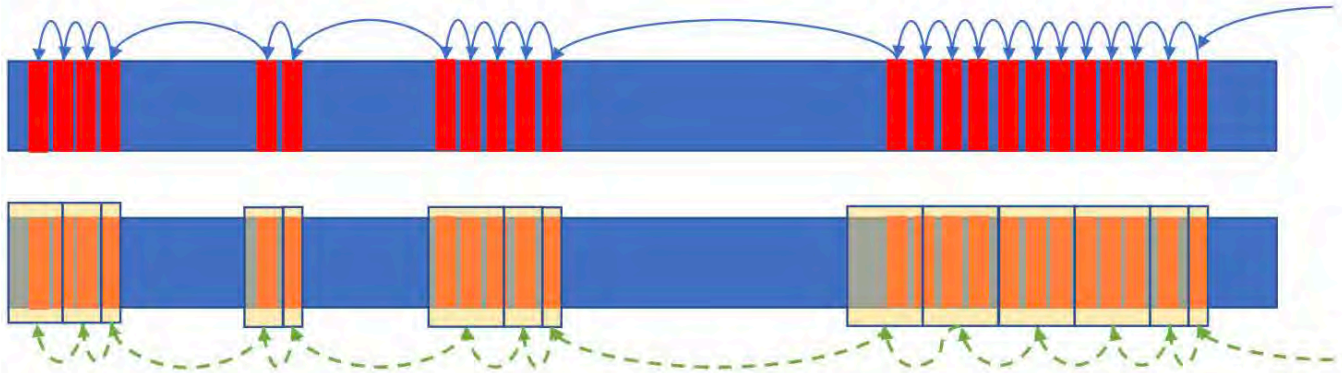
- On demand indexing → Only index a PSF when explicitly registered.
- De/registration service provide safe index boundaries
  - Guarantee all records within the boundary are indexed for a specific PSF.
- Retrieve records with a given property (in a specified address range):
  - No index built → full scan
  - Covered by safe boundary → hash chain scan
  - Partially cover by safe boundary → hash chain scan + full scan.



Index for queried PSF is built between addresses $[s_1, t_1]$, $[s_2, t_2]$, $[s_3, tail]$

Subset Retrieval: Adaptive Prefetching

- When query is selective, random I/O is preferred to save bandwidth.
- When query is NOT selective, large sequential I/O is preferred to save #I/Os.
  - Syscall cost (through the kernel), CPU bounded.
  - SSD Read Latency
- Trying to get the best of both worlds:
  - Observe more locality → More aggressive pre-fetching → Save #I/Os
  - Lose locality → Fall back to random I/O → Save SSD bandwidth



Use-cases

GitHub: GHArchive Sep 2018, 18M records, record size ~3KB

Twitter: 1% twitter samples for 3 days, 9.3M records, record size > 5KB.

Yelp: Yelp review open dataset, 48M records, record size < 1KB

Dataset	Field Projections	Properties of Interest
Github	id, actor.id, repo.id, type	type == "IssuesEvent" && payload.action == "opened" type == "PullRequestEvent" && payload.pull_request.head.repo.language == "C++"
Twitter	id, user.id, in_reply_to_status_id, in_reply_to_user_id, lang	user.lang == "ja" && user.followers_count > 3000 in_reply_to_screen_name = "realDonaldTrump" && possibly_sensitive == true
Yelp	review_id, user_id, business_id, stars	stars > 3 && useful > 5 useful > 10

Evaluation

Key question for FishStore

Do we need both ***fast index*** and ***fast parser***?

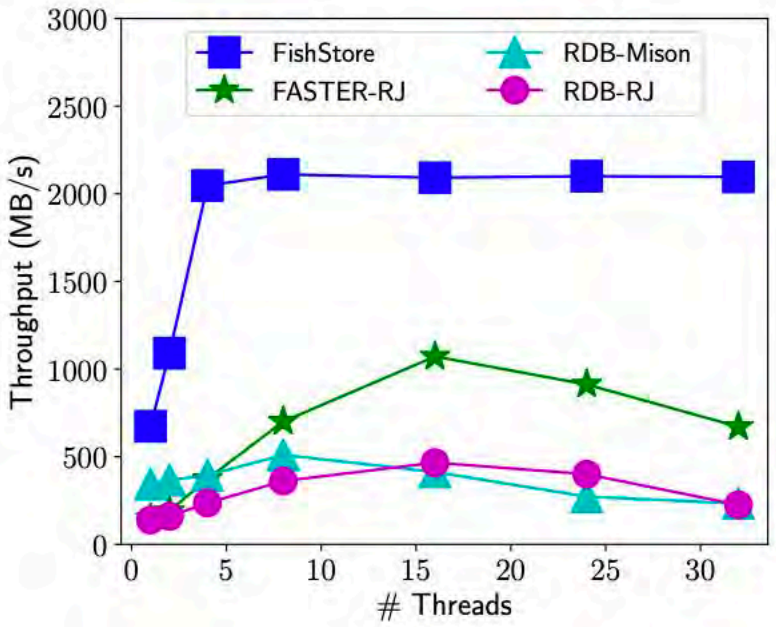
FishStore: FASTER (fast index) + Mison (fast parser)

FishStore-RJ: FASTER (fast index) + RapidJSON (slow parser)

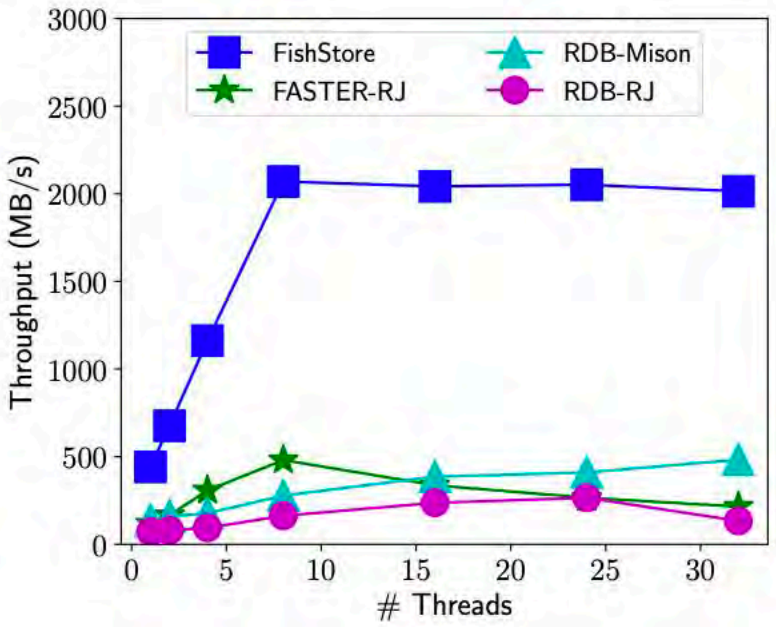
RDB-Mison++: RocksDB (slow index) + Mison (fast parser)

RDB-RJ: RocksDB (slow index) + RapidJSON (slow parser)

Ingestion Throughput (on SSD)



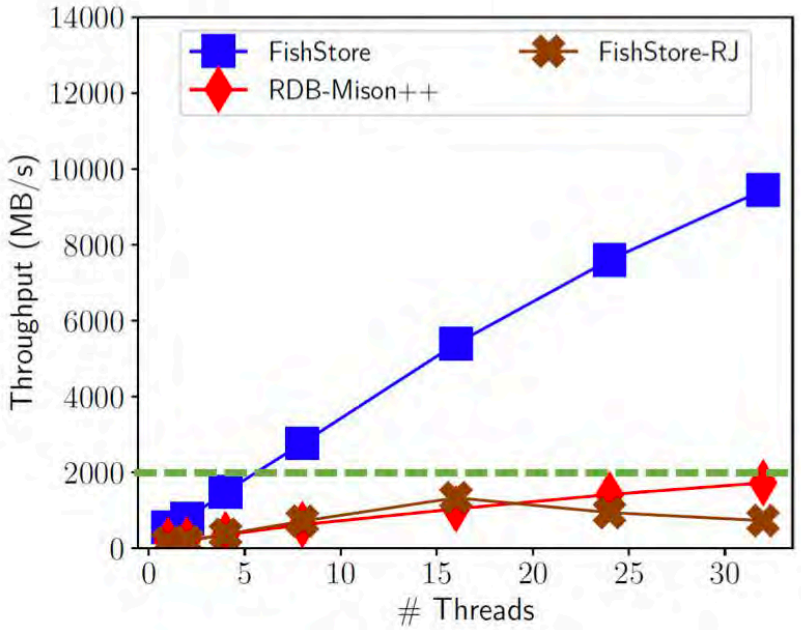
(a) Github



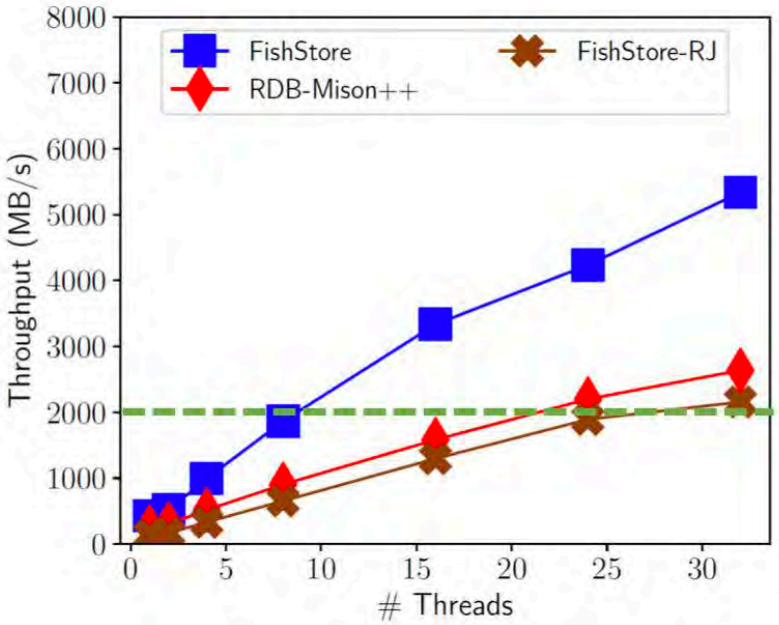
(b) Yelp

Saturate SSD Bandwidth with 8 cores!

Ingestion Throughput (in Memory)



Github

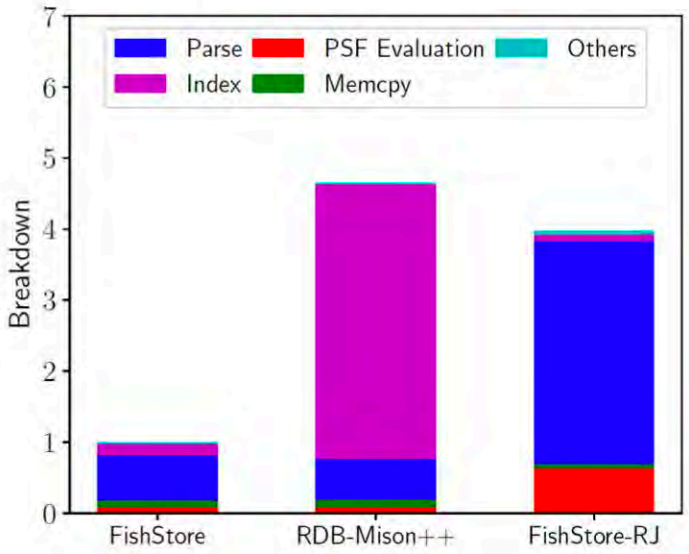


Twitter

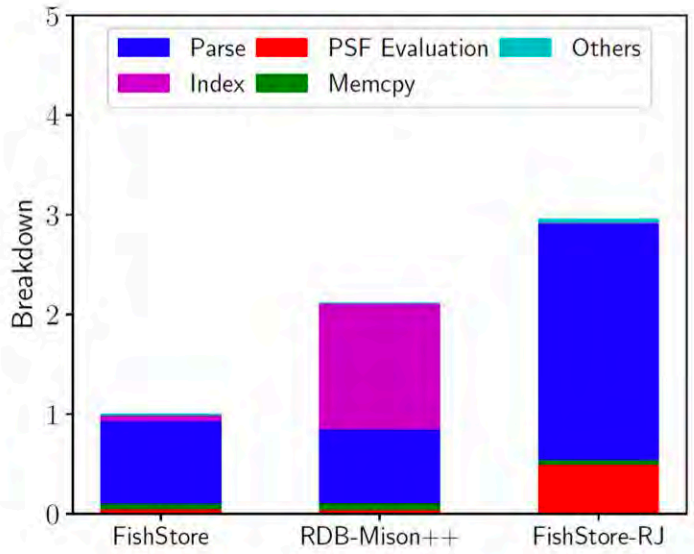
Without SSD achieve 10GB/s of ingesting while parsing!

Ingestion: CPU Breakdown

- Parsing and Indexing can both be system bottleneck.
- FishStore balances the cost.



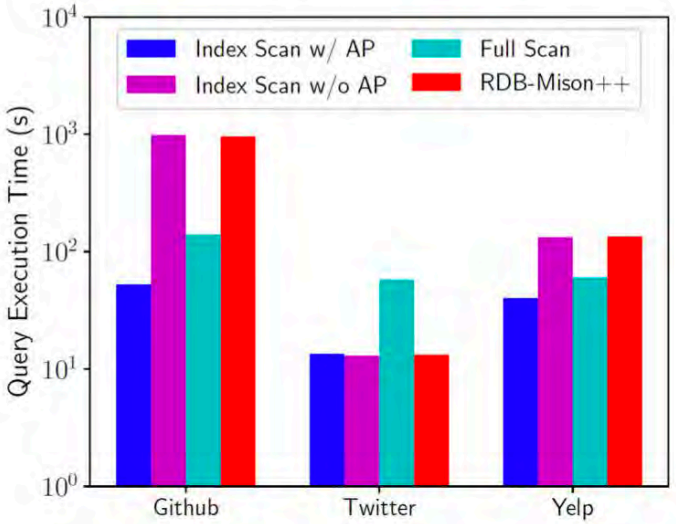
Github



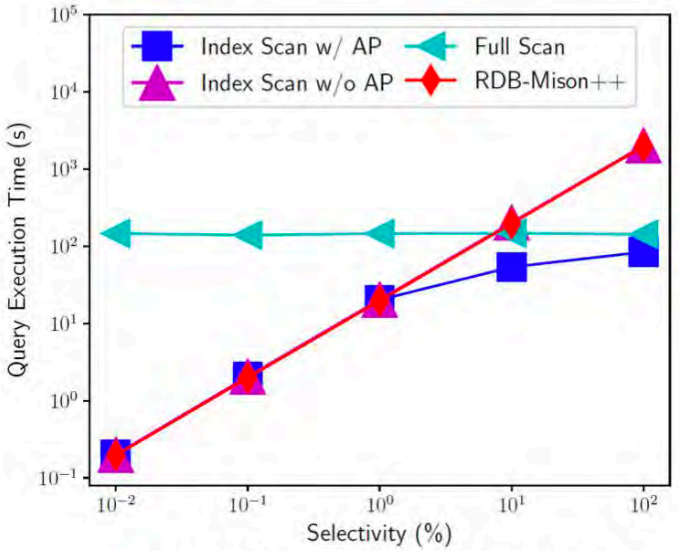
Twitter

Subset Retrieval

Github: `type = "PushEvent"`
Twitter: `user.lang = ja && user.follower_count > 3000`
Yelp: `stars > 3 && useful > 5`



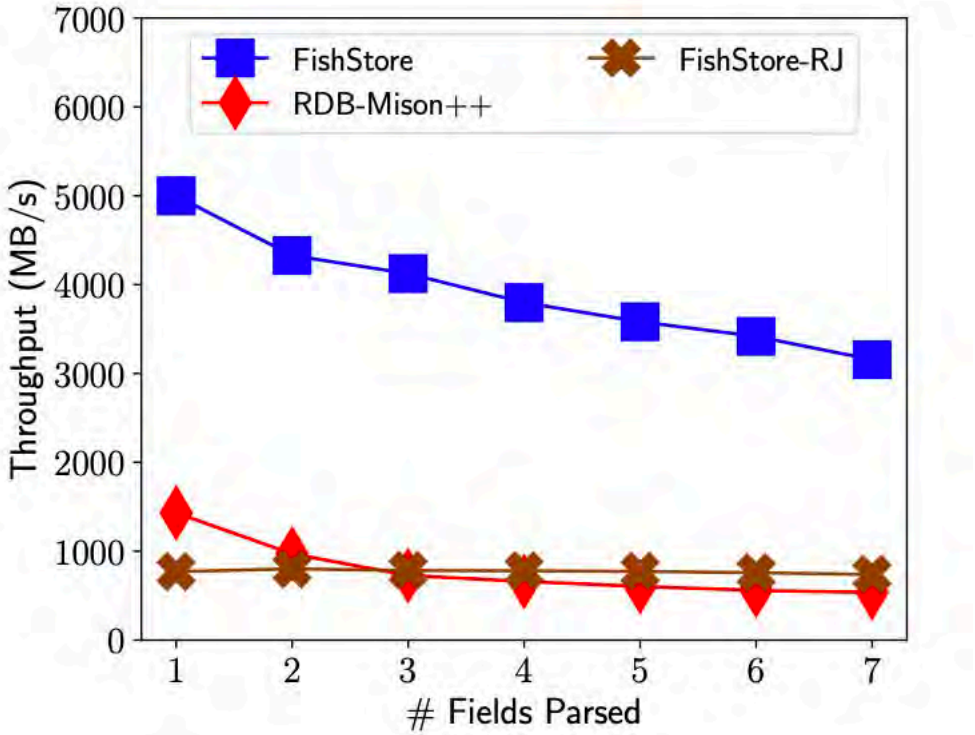
Query Latency



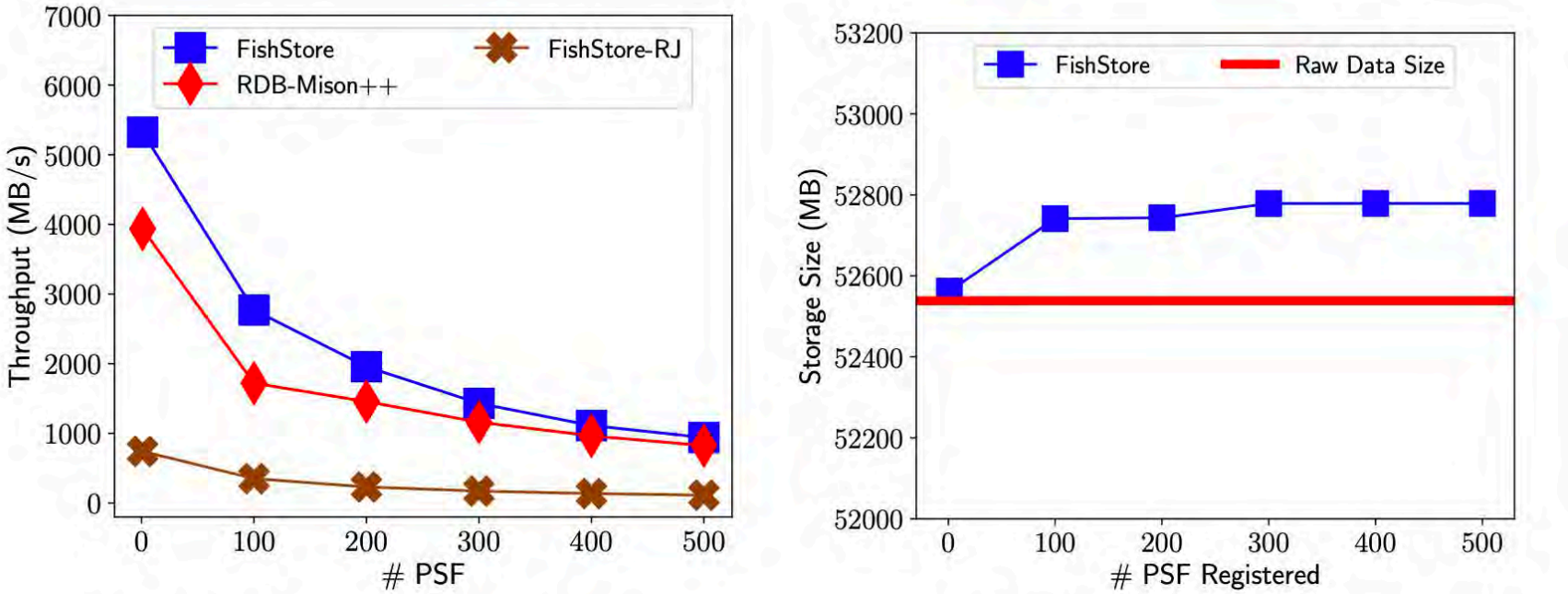
Effect of Selectivity

BOSTON UNIVERSITY

Parsing Overhead



Impact of the number of PSF



(a) Ingestion Throughput (b) Storage Cost

Figure 15: Predicate based PSF Scalability

Conclusion

Data store design depends on **requirements** and **workload!**

FishStore

New storage layer **for flexible-schema** data.

Predicate Subset Function (PSF) group records logically.

Efficient **subset hash index** + **on demand indexing**.

Fast parser + fast index = **minimum effort fast ingestion**.

Hybrid scan with adaptive prefetching.

BOSTON UNIVERSITY

CS 561: Data Systems Architectures

class 9

Key-Value Stores for Concurrent and Point Accesses

Prof. Manos Athanassoulis

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