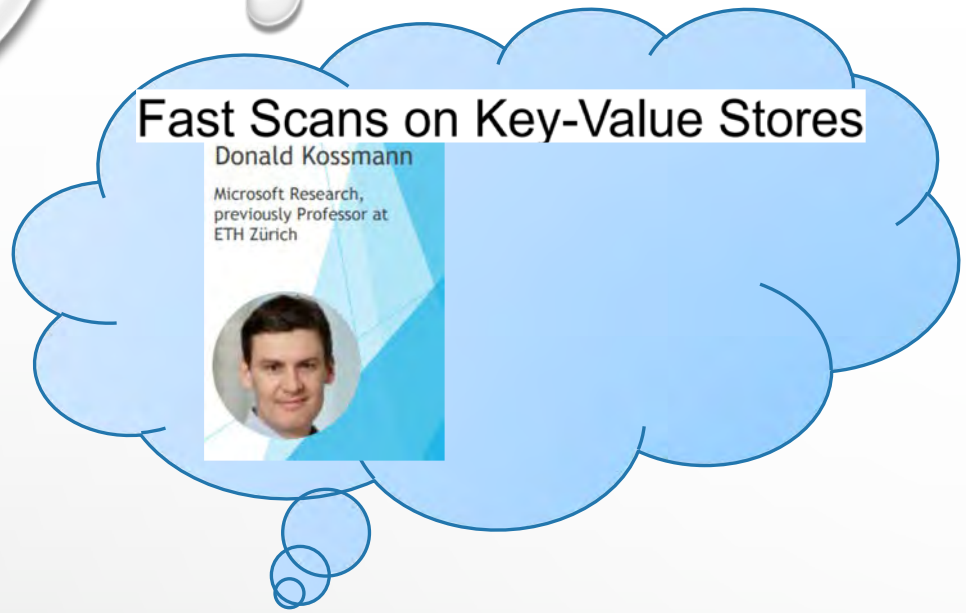
FISHSTORE: FASTER INGESTION WITH SUBSET HASHING

BY BENJAMIN BORDEN

TABLE OF CONTENTS



MEET THE AUTHORS

Donald Kossman
Microsoft Research

Badrish Chandramouli
Microsoft Research

Yinan Li
Microsoft Research

Dong Xie
University of Utah

THE PROBLEM?

- Increased importance of cloud-edge architecture →
  - increased the amount of data being ingested by the cloud
- Previous Industry Goal:
  - Ingest information as fast as possible by dumping it into storage
- Current Industry Goal:
  - The data needs to be immediately ready for analytical queries

THE TYPE OF QUERIES WE CARE ABOUT

- Ad-hoc queries
  - Scan data over time ranges
- Recurring queries
  - Have identical predicates, but are repeated over different time ranges
- Point lookup queries
  - Based on various keys
- Streaming queries
  - Fed parts of the ingested data satisfying custom predicates and based on the query schema

TRADITIONAL APPROACHES

- Ingest the data in raw form
- Load data into warehouse in batch jobs

Why is this bad?

OR

- Parse data and load into database

or

- update secondary range index over every attribute during ingestion

Why is this bad?

TRADITIONAL APPROACHES

- Ingest the data in raw form
- Load data into warehouse in batch jobs

Why is this bad?
This is slow (low throughput)
Has high latency before the data is ready for queries

OR

- Parse data and load into database

or

- update secondary range index over every attribute during ingestion

Why is this bad?
Also very slow (low throughput)

INTRODUCING FASTER

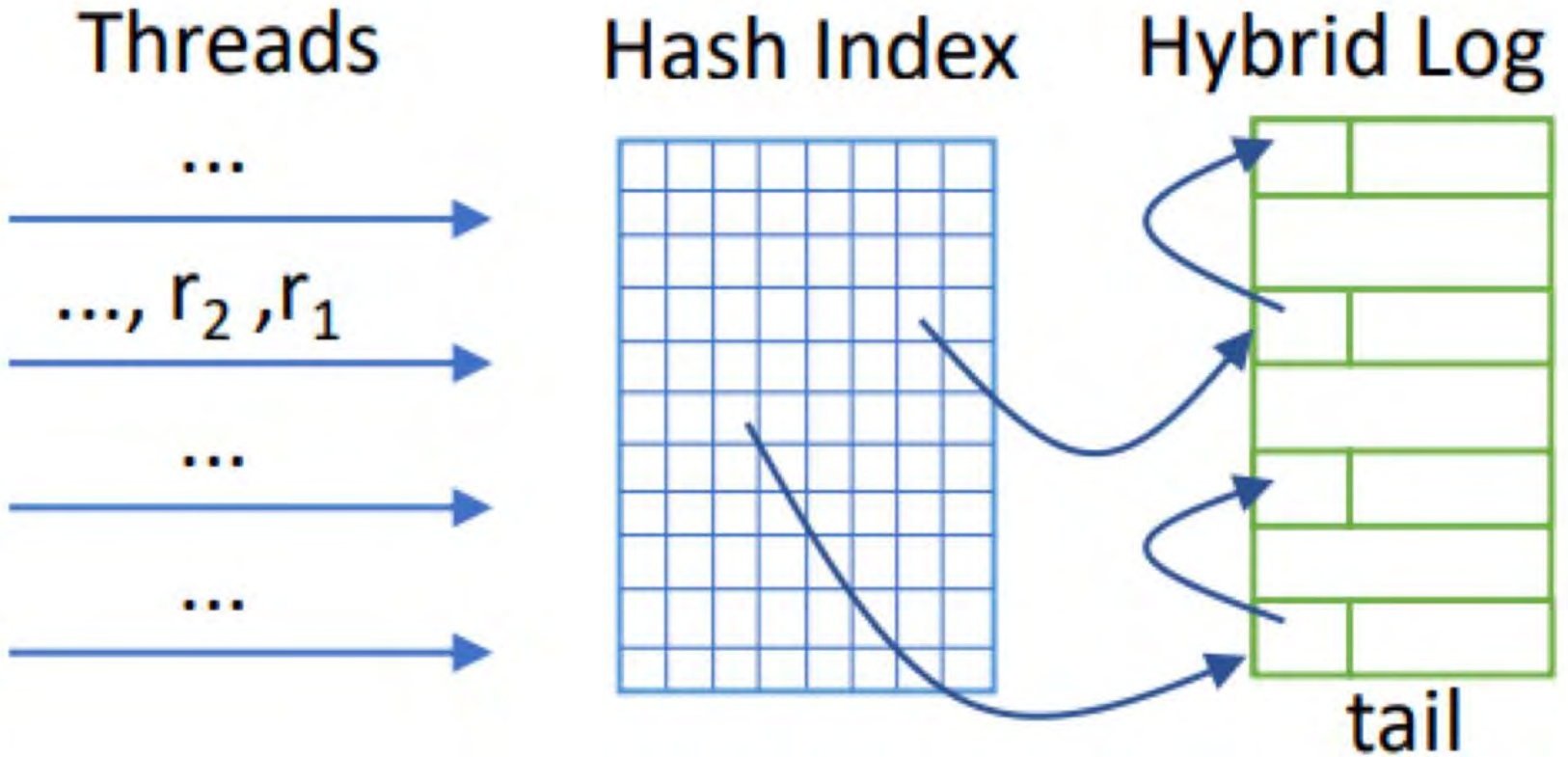
FASTER

- FASTER:
  - a **persistent key-value store**
  - consists of a:
    - lightweight
    - cache optimized
    - concurrent
  - hash index backed by a hybrid log ordered by data-arrival

- Benefits
  - Very low CPU cost
  - More than 150 millions operations per second

BACKGROUND ON FASTER

- FASTER: concurrent latch-free hash key value store with support for larger than memory data
- FASTER has two components:
  - a hash index
  - log structured record store



INTRODUCING FISHSTORE

INTRODUCING FISHSTORE

- Fishstore = [F]aster [I]ngestion with [S]ubset [H]ashing store:
  - a storage layer for flexible schema data
- Borrows some features from faster such as basic index design and epoch model
- Allows for the dynamic registration of predicated subset functions (PSF) →
  - allow for efficient retrieval of subsets of ingested data based on common criteria

- **PROS OF PSFs –**
- **CONS OF PSFs –**

INTRODUCING FISHSTORE

- Fishstore = [F]aster [I]ngestion with [S]ubset [H]ashing store:
  - a storage layer for flexible schema data
- Borrows some features from faster such as basic index design and epoch model
- Allows for the dynamic registration of predicated subset functions (PSF) →
  - allow for efficient retrieval of subsets of ingested data based on common criteria

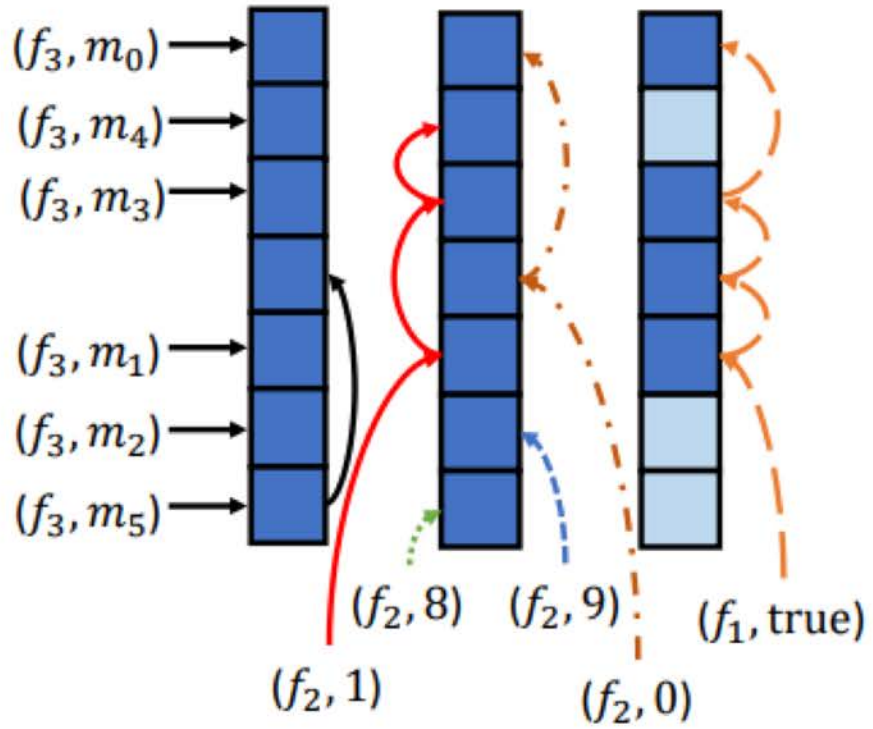
- **PROS OF PSFs – Allows for speedy and customizable retrieval of data**
- **CONS OF PSFs – A large number of PSF leads to throughput penalties**

PREDICATED SUBSET FUNCTION

- Predicated Subset Function (PSFs):
  - groups records with similar properties for later retrieval

- "Given a data source of records in R, a predicated subset function (PSF) is a function f : R → D which maps valid records in R, based on a set of fields of interest in R, to a specific value in domain D”

EXAMPLE: MACHINE 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: $\Pi_{\text{Machine}}(r)$

Figure 1: Machine Telemetry PSF Example

- THE CHALLENGES OF THE FISHSTORE SYSTEM CONSISTS OF TWO MAJOR COMPONENTS:
  - INGESTION AND INDEXING
  - SUBSET RETRIEVAL

INGESTION & INDEXING

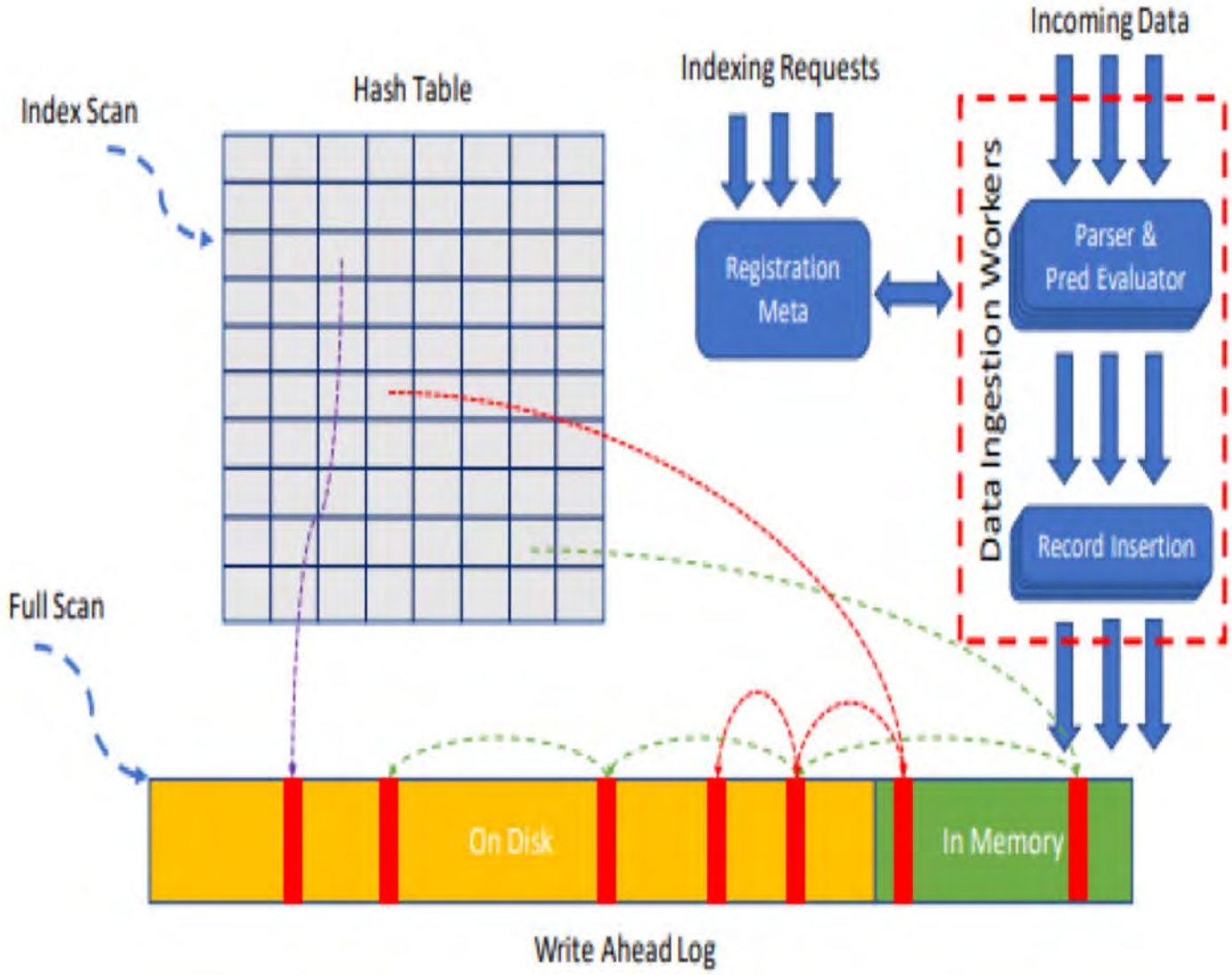
- FISHSTORE ingests data into an immutable log in ingestion order and with a hash index
  - For every PSF and value in the log there is a hash entry which links all matching values in a hash chain
  - Major difference between key-value store and PSF based hash chains:
    - A record can be part of more than one hash chain!

SUBSET RETRIEVAL

- The primary benefit of PSFs is scan optimization
  - If a subset needs to be retrieved which does not have a relevant PSF, a full scan of the data is required
  - Fishstore supports scans for records that match PSF values in which case it retrieves the relevant hash chain.
    - **However FISHSTORE does not recatagorize data which is older than the implementation of the PSF!**
  - When retrieving data that has a relevant PSF but the data is older than the PSF implementation:
    - a full scan must be done over the older data.

SYSTEM ARCHITECTURE OF FISHSTORE

- The architecture of FISHSTORE consists of:
  - A hybrid log for record allocation,
  - A hash index that holds pointers to records on the log,
  - A registration service
  - A set of ingestion workers



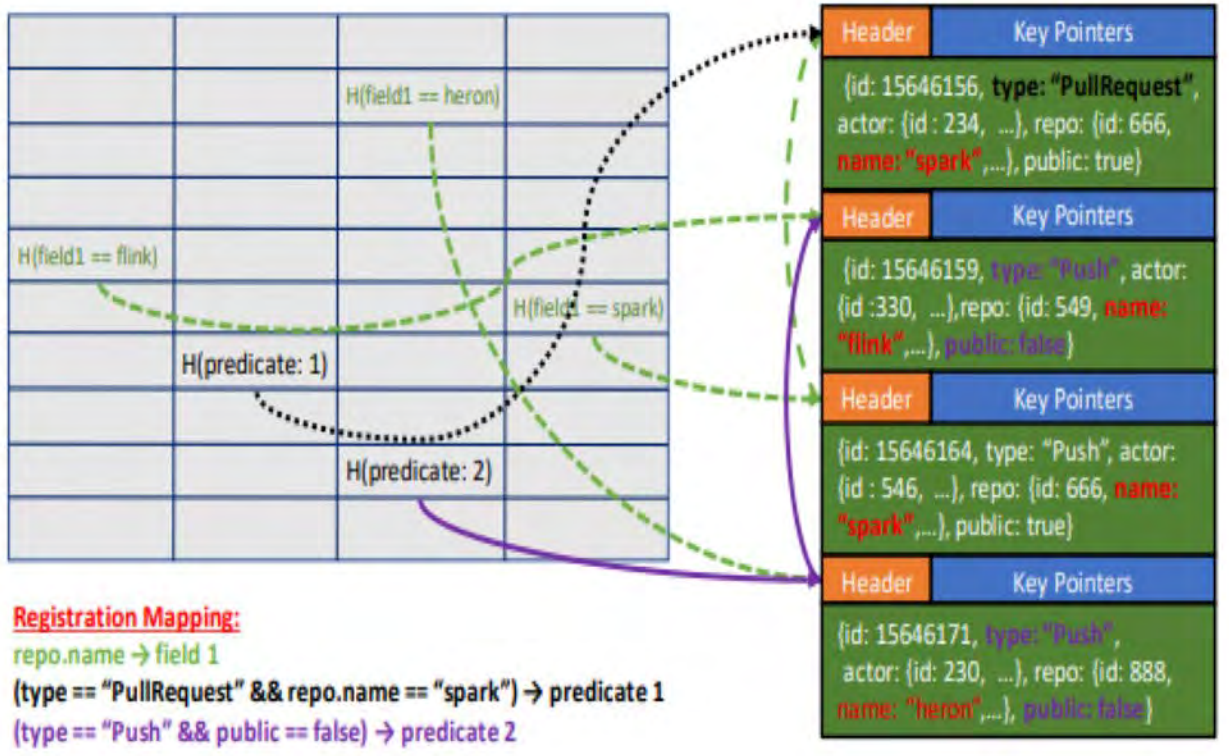
CHALLENGES/SOLUTIONS

- Problem: Designing a fast concurrent index which supports PSFs
  - Solution: Using a subset hash index combined with a carefully designed record layout
- Problem: Acquiring a safety boundary for on-demand indexing.
  - Solution: Utilizing epoch-based threading model to find safe boundaries
- Problem: Data ingestion should be latch-free in order to achieve high throughput
  - Solution: An unusual lock-free technique discussed later
- Problem: Scanning a hash chain imitates random I/Os which damages subset retrieval speeds
  - Solution: Utilizing an adaptive prefetching technique discussed in more detail later

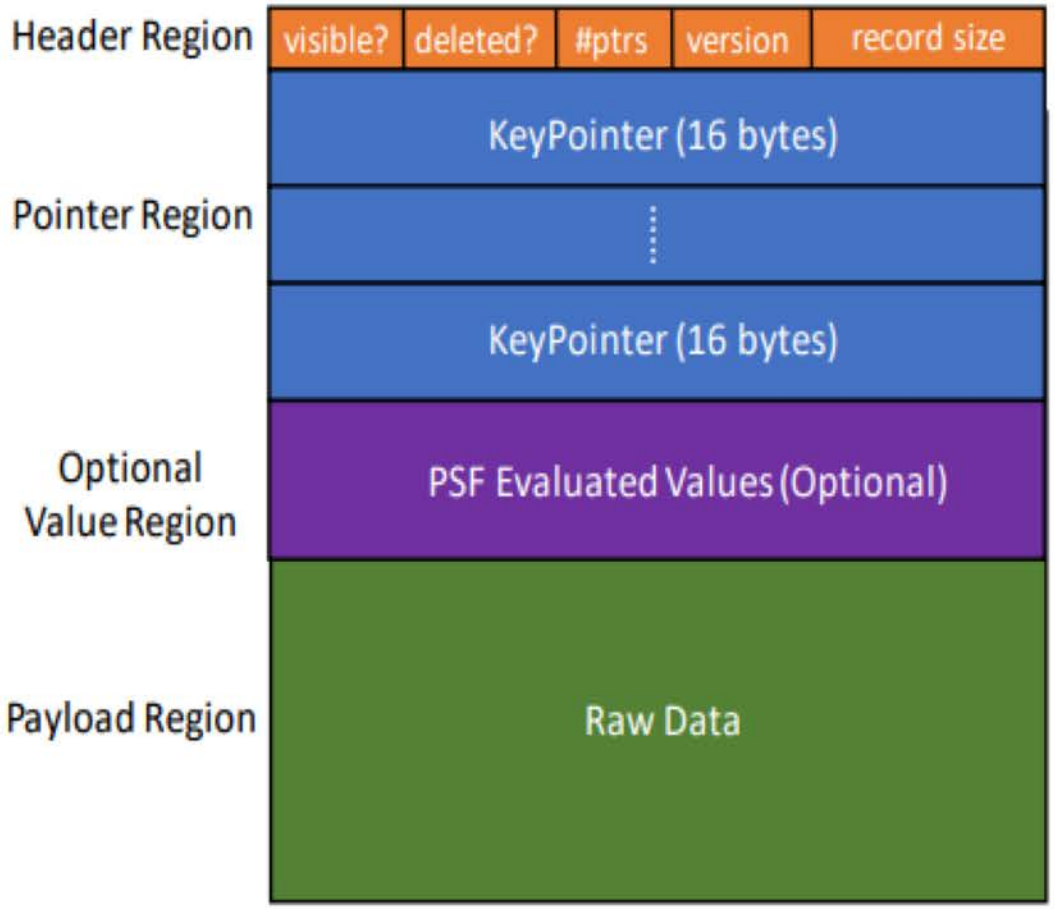
SUBSET HASH INDEX

SUBSET HASH INDEX

- FISHSTORE's fast index called subset hash index:
  - a hash table that indexes hash chains of records based on user defined PSFs
  - To retrieve records with a specific property:
    - calculates the hash signature of the property to locate the hash entry
    - follows the chain until it retrieves all matching entries



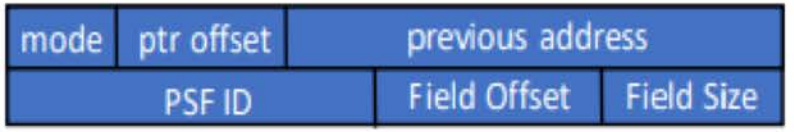
FISHSTORE RECORD LAYOUT



Sample KeyPointer Constructions:

Boolean Domain PSF Key Pointers

Field Projection PSF Key Pointers



General Key Pointers Referring to Value Region

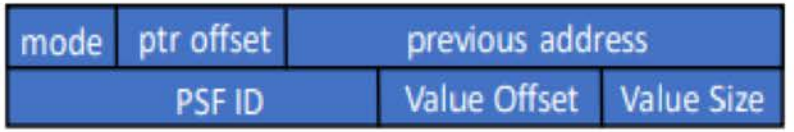


Figure 6: FishStore Record Layout

EPOCHS AND YOU

- Fishstore has an epoch framework where there is a global epoch e which may be incremented by any thread
- Each thread has a local epoch, which they refresh to keep up with the global epoch
- When all threads' epochs are at least at value c, then all threads are aware of all changes up to global epoch c

ON-DEMAND INDEXING

- FISHSTORE keeps two versions of all metadata
- **Rest** state:
  - when there are no index altering requests
- **Prepare** state:
  - when the user pushes an index altering request to the system
- Pending state:
  - Once the inactive metadata is updated, it becomes the active metadata and increments the global epoch to apply the changes.
- **Rest** state:
  - When the epoch becomes safe, FISHSTORE applies the changes to the old metadata

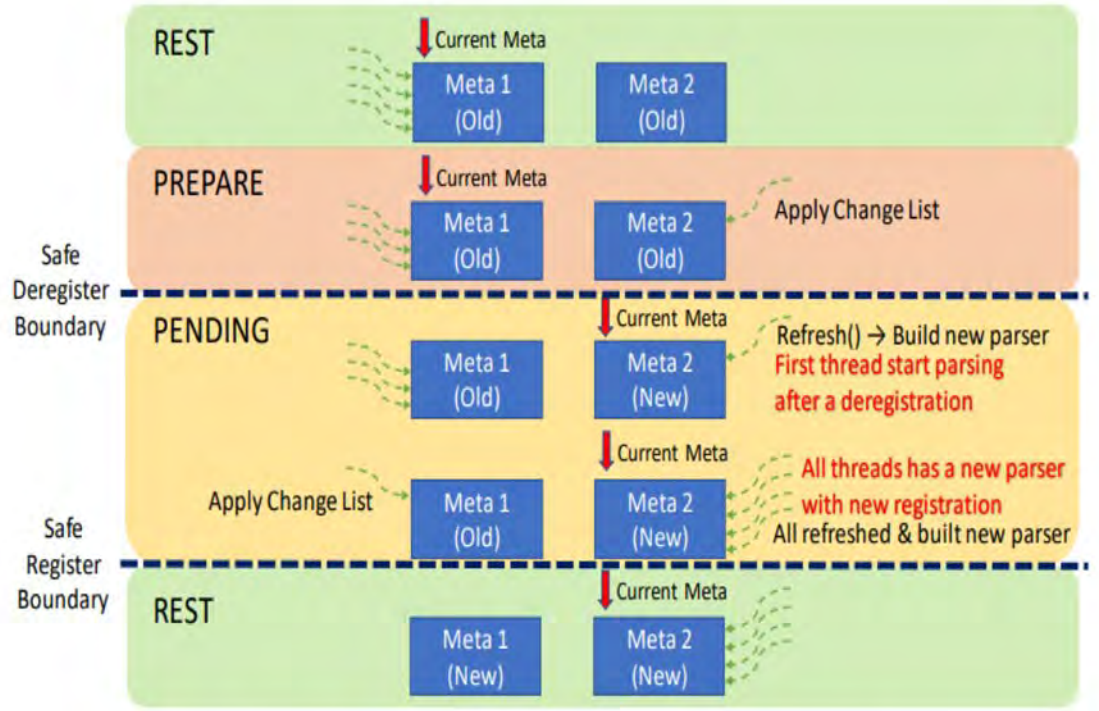


Figure 7: On Demand Indexing

DATA INGESTION IN FISHSTORE

DATA INGESTION

- Fishstore ingests data concurrently with each thread being referred to as an **ingestion worker**
- To complete ingestion, the incoming data moves through four phases:
  - Parsing and PSF evaluation
  - Record space allocation
  - Subset hash index update
  - Record visibility

PARSING AND PSF EVALUATION

- Ingestion worker uses whatever parser which was provided by the user to parse data fields in a batch.
- Whenever a worker detects changes:
  - it recalculates the minimum field set for index building and recreates its thread-local parser
- After parsing out essential fields:
  - the worker annotates the position of a field and evaluates all requested PSFs for each record.

RECORD SPACE ALLOCATION

- After a worker is done parsing the received data →
  - it must determine how much space it requires in fishstore.
- Then the worker allocates space by shifting the tail of the log to accommodate more data.
- Lastly the data is copied over to the payload region and record headers are filled.
- Since we have yet to update the index, the record is inserted with its visibility bit set to false.
  - Any reader that touches an invisible record ignores it.
  - This guarantees the atomicity of record insertion.

INDEX UPDATE & RECORD VISIBILITY

- Once record space allocation is finished, the worker updates all the hash chains of which the record applies.
- To perform this update, we use a compare and swap
- IF THE compare-and-swap fails (which is usually):
  - mark the record as invalid, reallocate space on the log and try again.
- Finally, the record is made atomically visible to readers by setting the visibility bit in the header

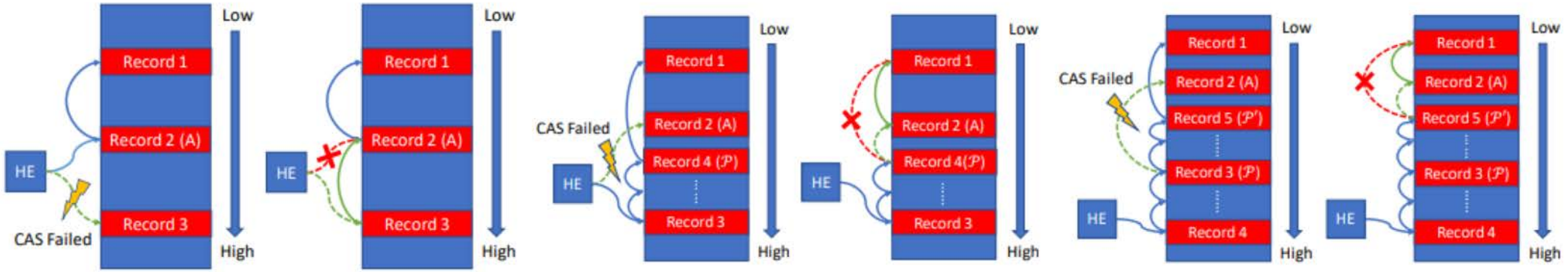


Figure 8: Handling CAS failure.

SUBSET RETRIEVAL IN FISHSTORE

SUBSET RETRIEVAL INTERFACE

- Fishstore retrieves all records satisfying a given property in a continuous range of the log
- However, just because a given property currently has a hash chain associated with it, does not mean all the records which satisfy that property have been indexed. Why?

SUBSET RETRIEVAL INTERFACE

- Fishstore retrieves all records satisfying a given property in a continuous range of the log
- However, just because a given property currently has a hash chain associated with it, does not mean all the records which satisfy that property have been indexed. Why? Because records could have been entered before the PSF was written!
  - Fishstore breaks the request into index scans + full scans based on safe registration and deregistration boundaries for all psfs.
  - Full scans have to parse and evaluate the request against each record encountered

ADAPTIVE PREFETCHING

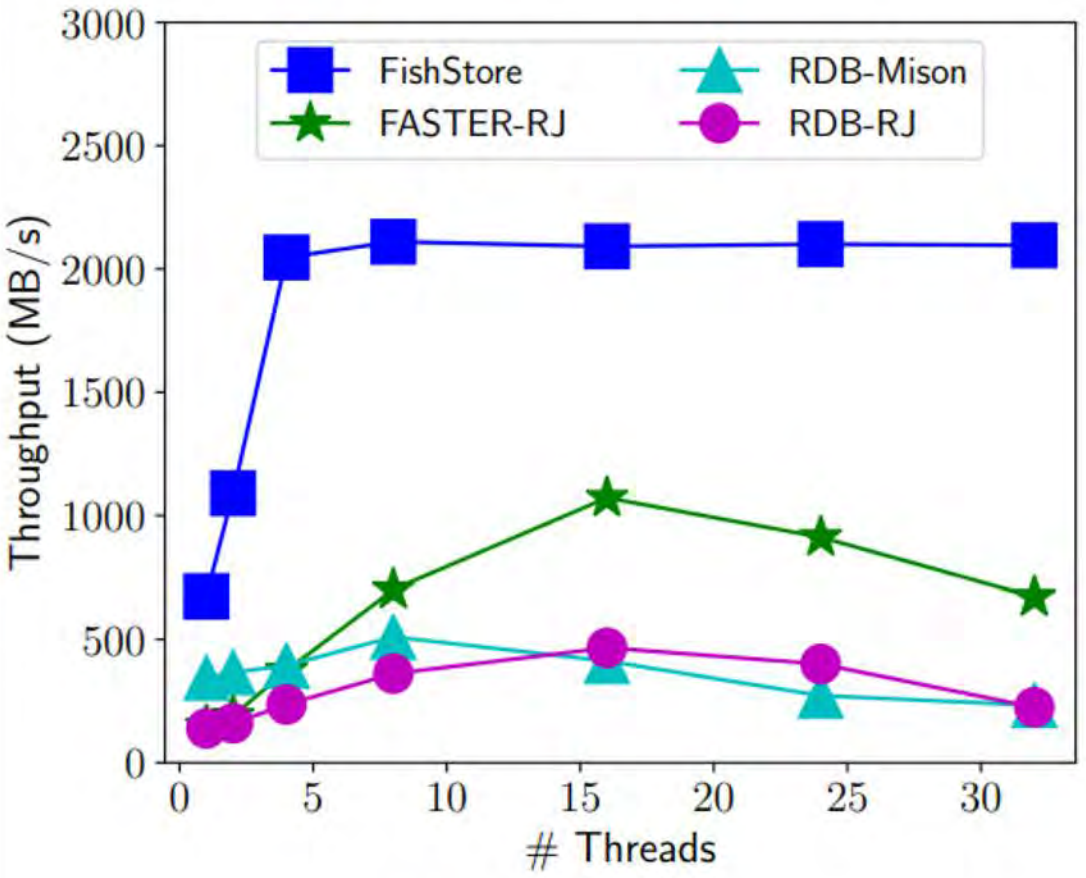
ADAPTIVE PREFETCHING

- When the log is totally in-memory, exploring the hash chain is not resource intensive.
- If the exploration lands on storage:
  - will issue random I/Os even when retrieved records may be continuously located on the log, hurting performance.
- In FISHSTORE, we use adaptive prefetching to detect locality on the hash chain and actively prefetch more data on the log to reduce random I/Os.

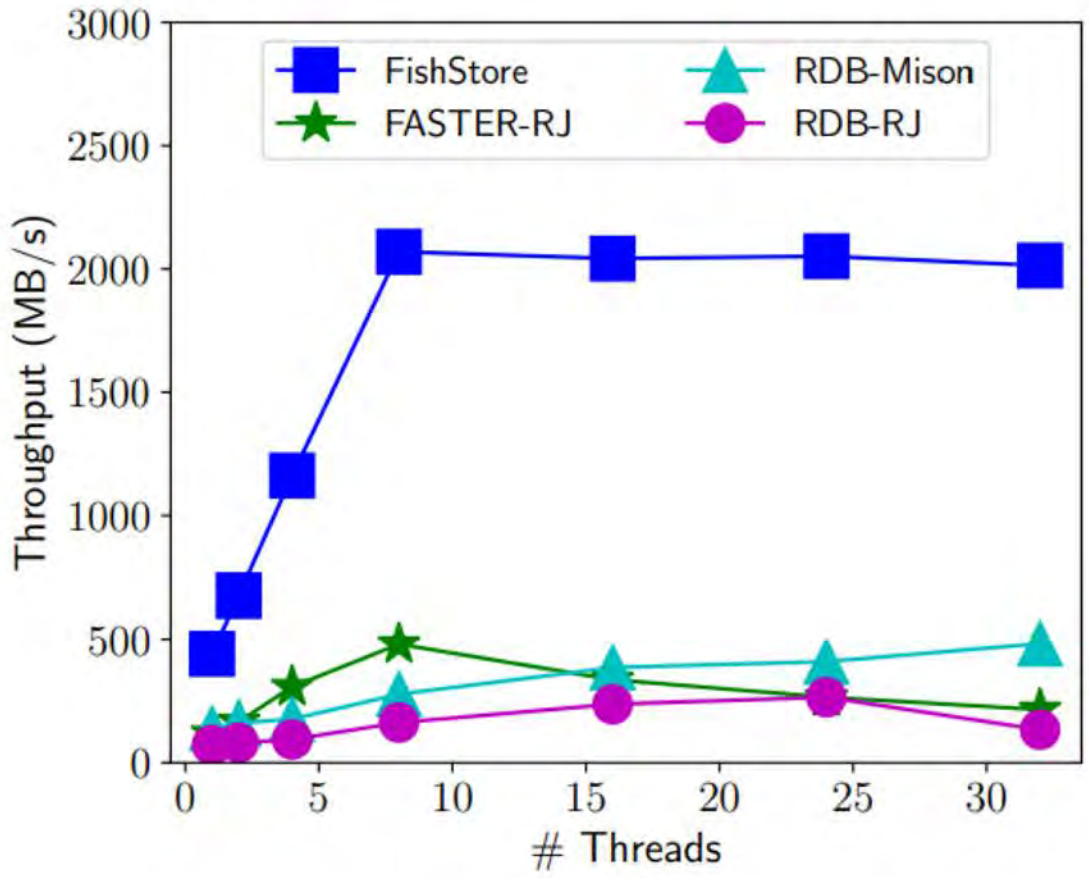
Figure 9: Adaptive Prefetching

EVALUATION OF FISHSTORE ON DATASETS

*HIGHER IS BETTER



(a) Github

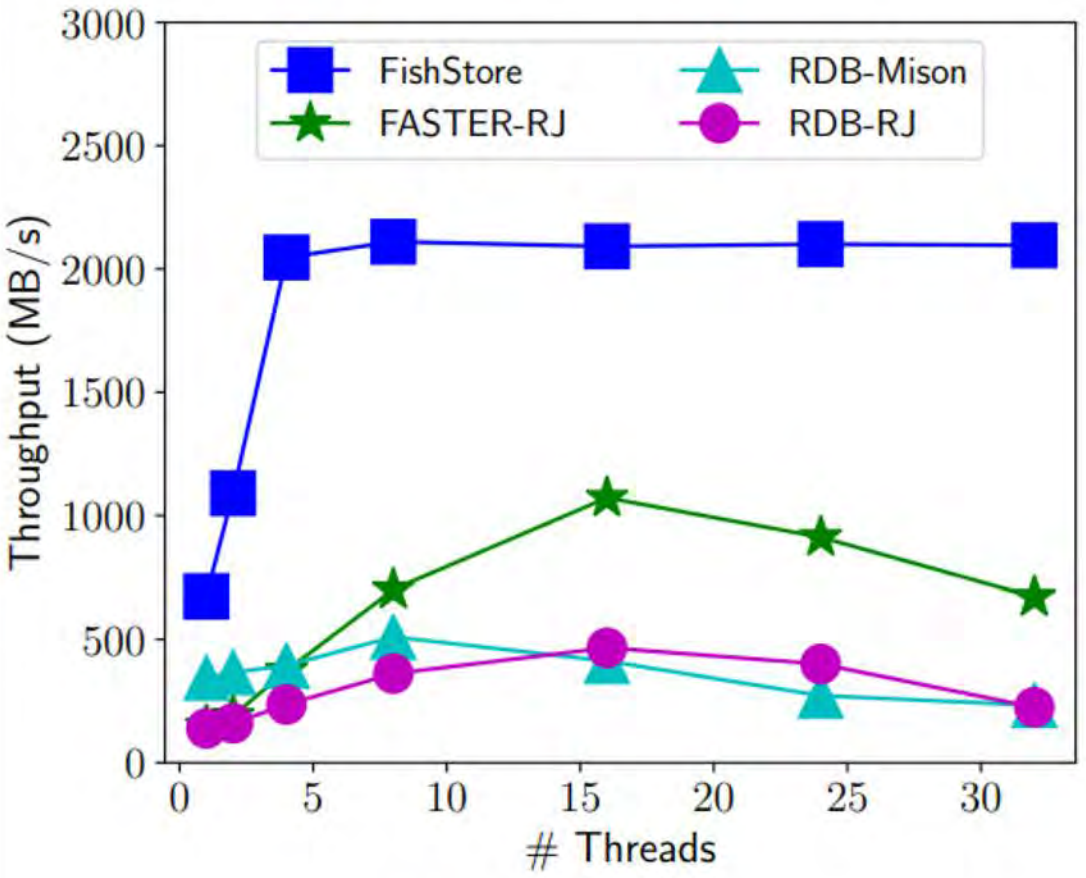


(b) Yelp

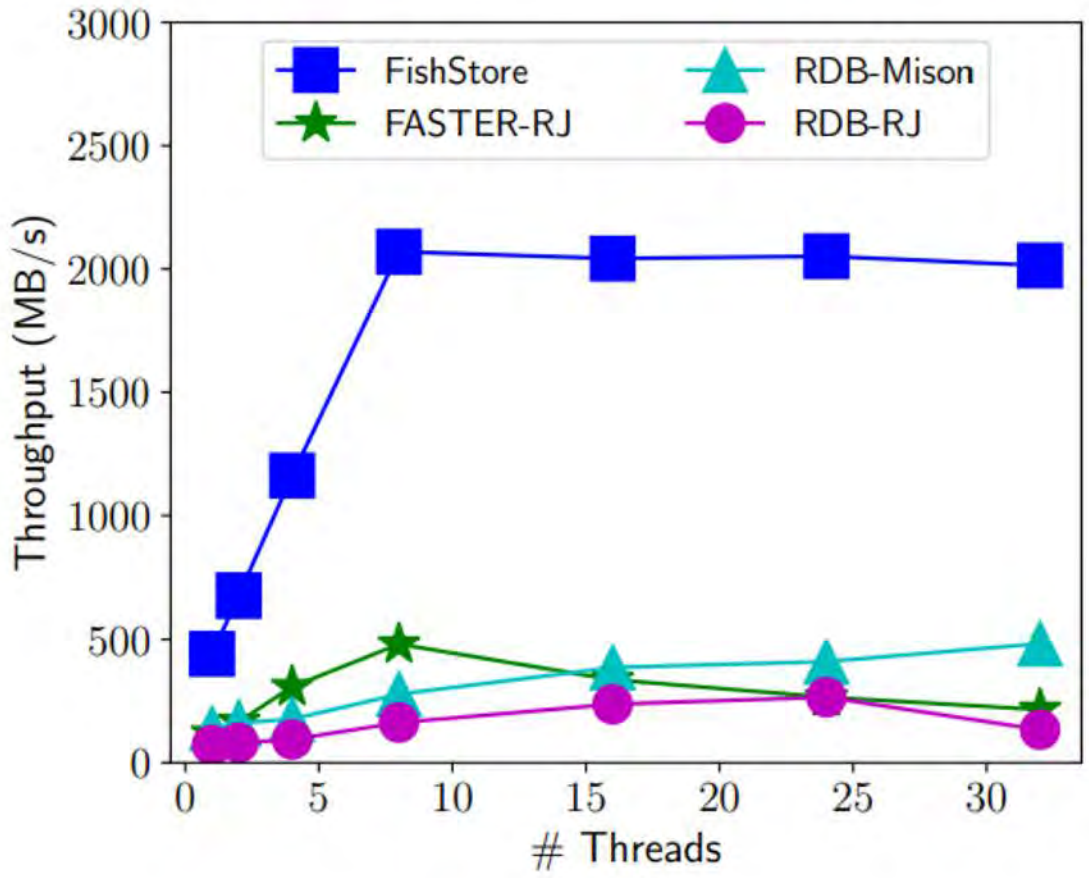
Figure 10: Comparison with Existing Solutions

*HIGHER IS BETTER

FISHSTORE Outperforms Due To Removal Of Indexing And Parsing Bottlebecks!



(a) Github



(b) Yelp

Figure 10: Comparison with Existing Solutions

*HIGHER IS BETTER

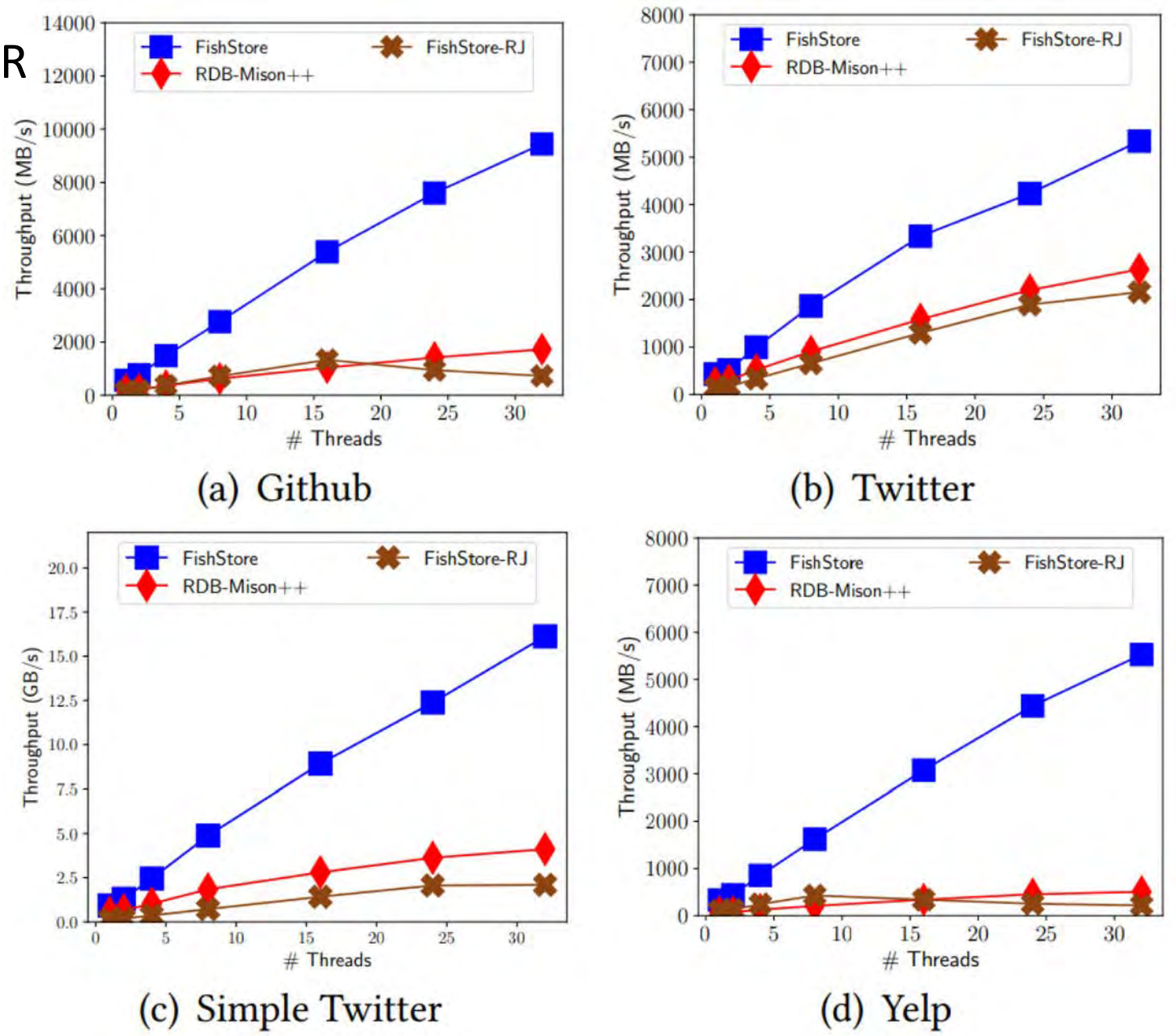


Figure 11: Ingestion Throughput in Main Memory

*HIGHER IS BETTER

FISHSTORE: Throughput ↑ as # of workers ↑!

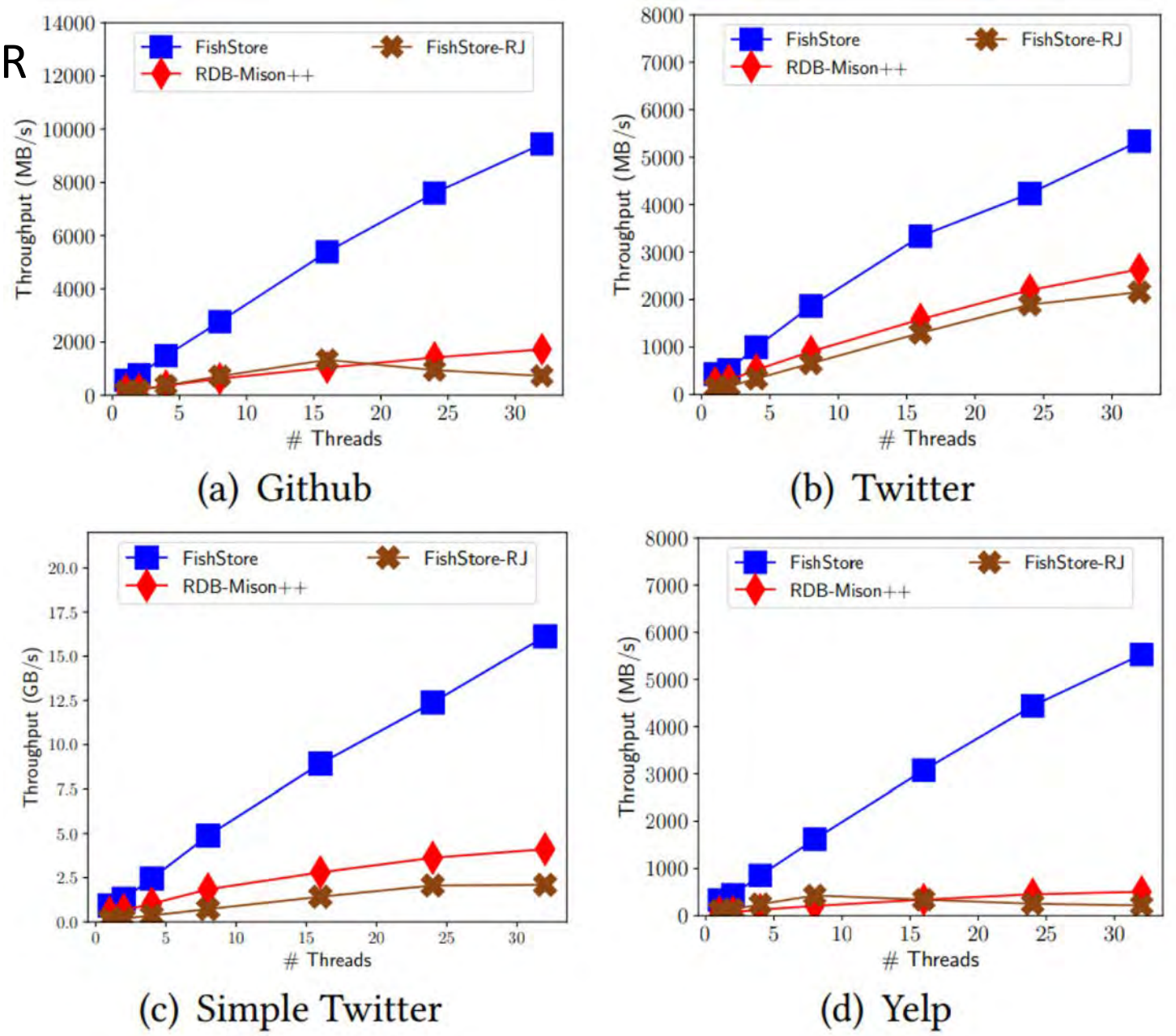
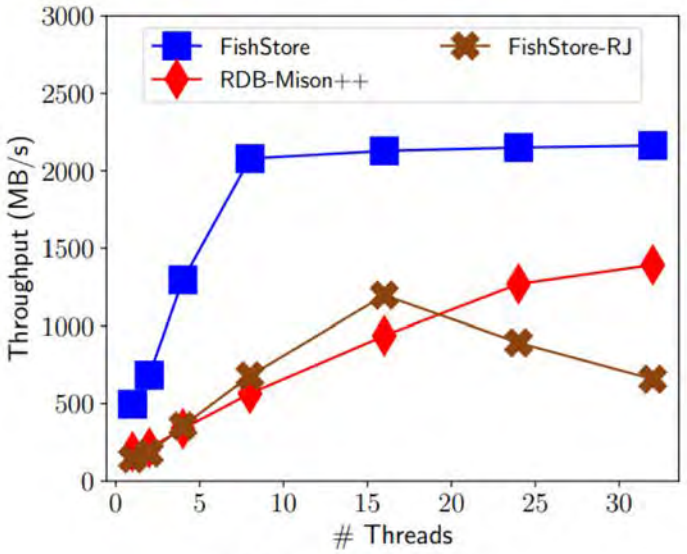
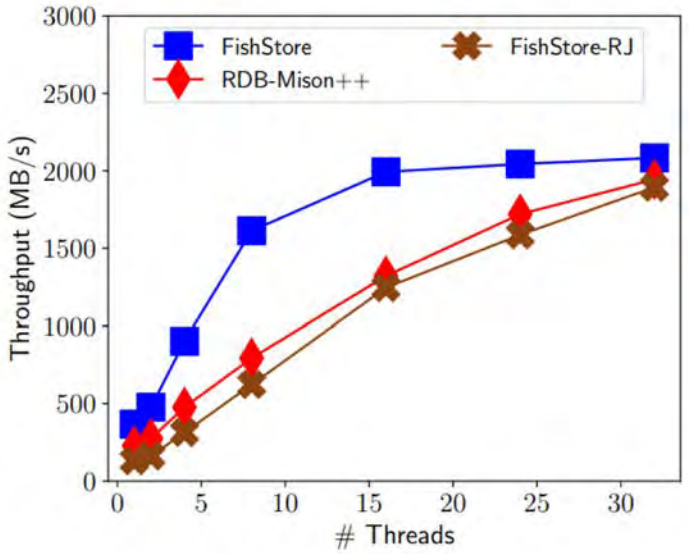


Figure 11: Ingestion Throughput in Main Memory

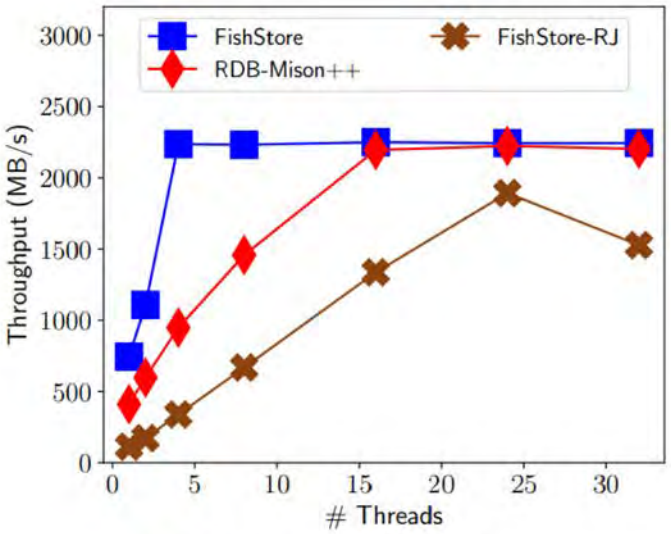
*HIGHER IS BETTER



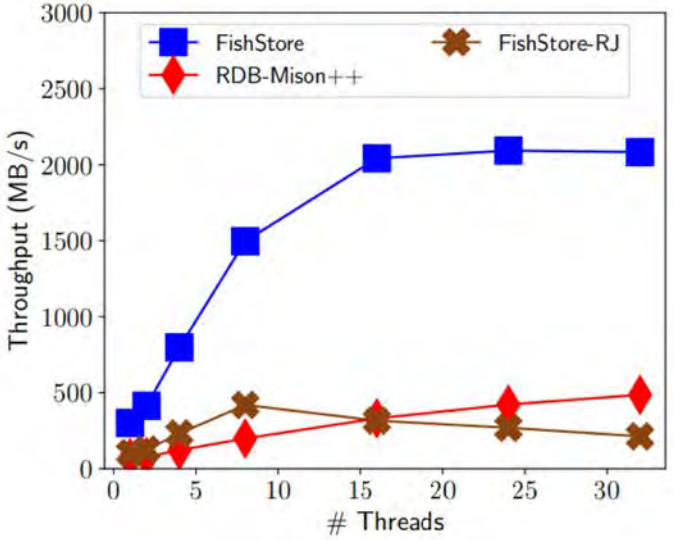
(a) Github



(b) Twitter



(c) Twitter Simple

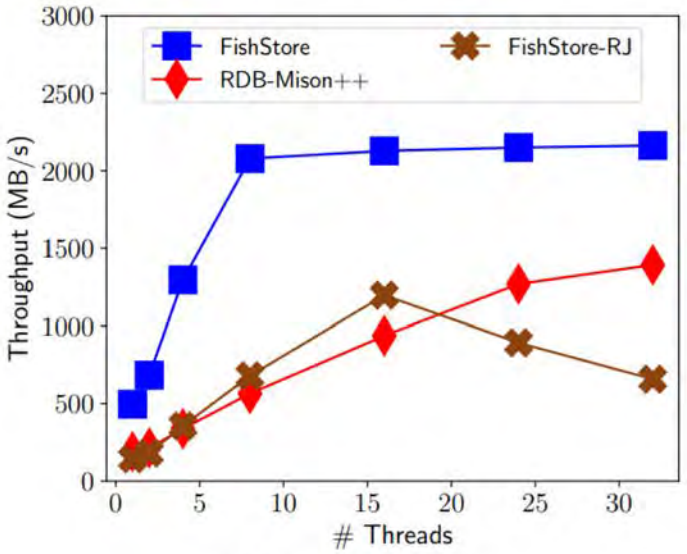


(d) Yelp

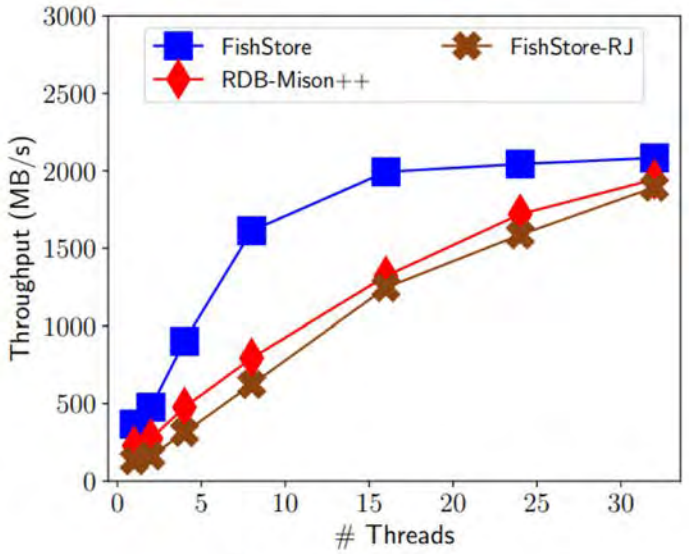
Figure 12: Ingestion Throughput on Disk

*HIGHER IS BETTER

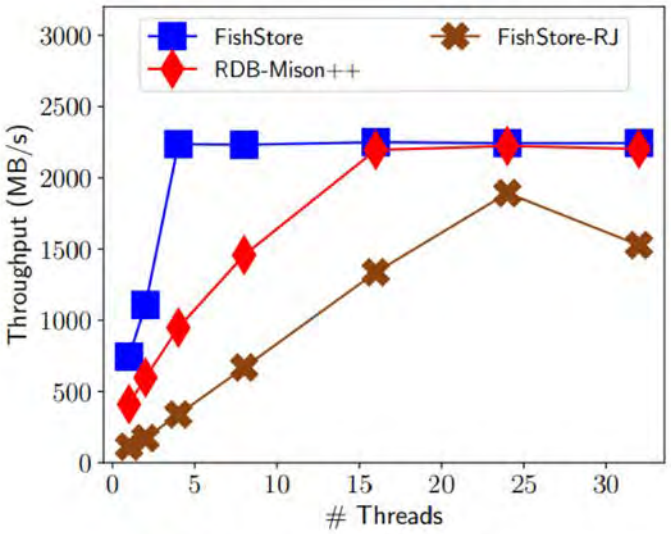
FISHSTORE scales better than other solutions!



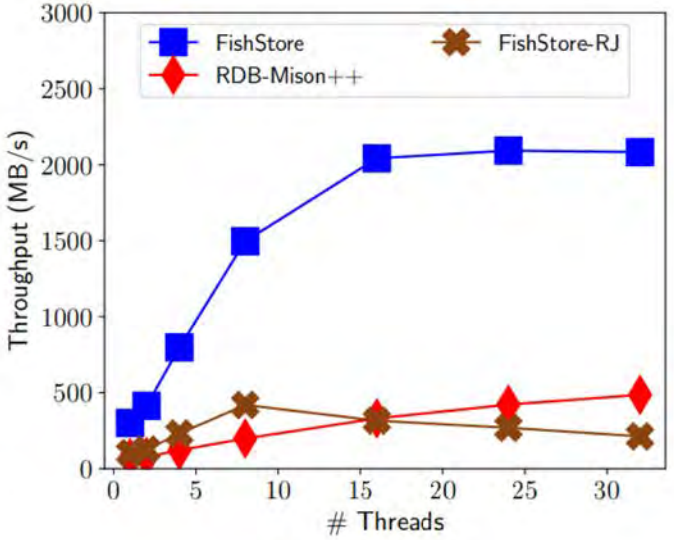
(a) Github



(b) Twitter

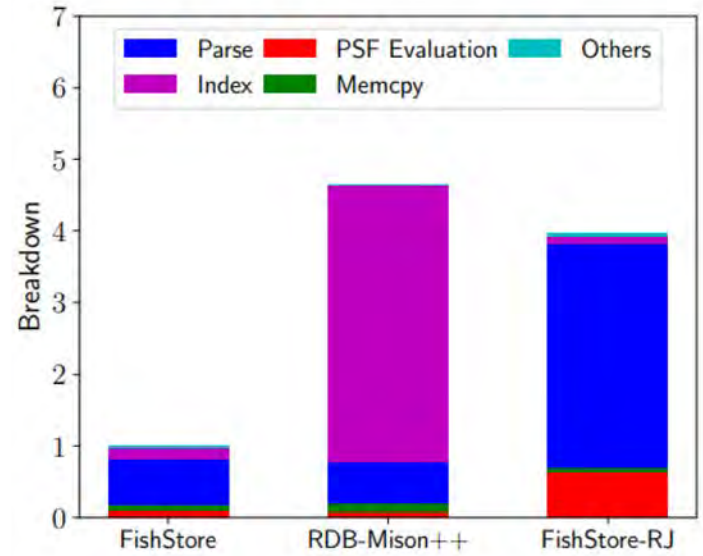


(c) Twitter Simple

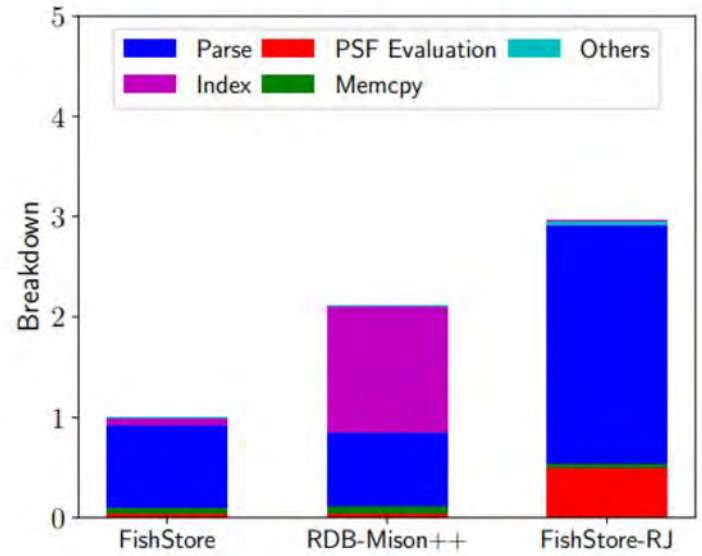


(d) Yelp

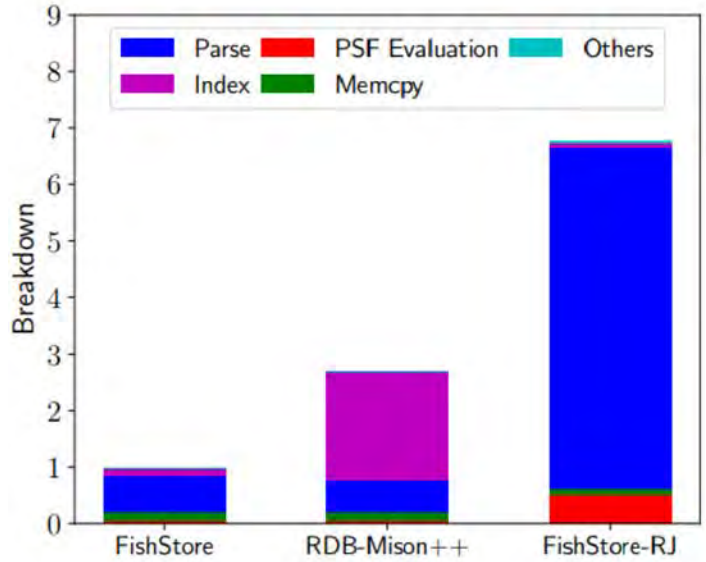
Figure 12: Ingestion Throughput on Disk



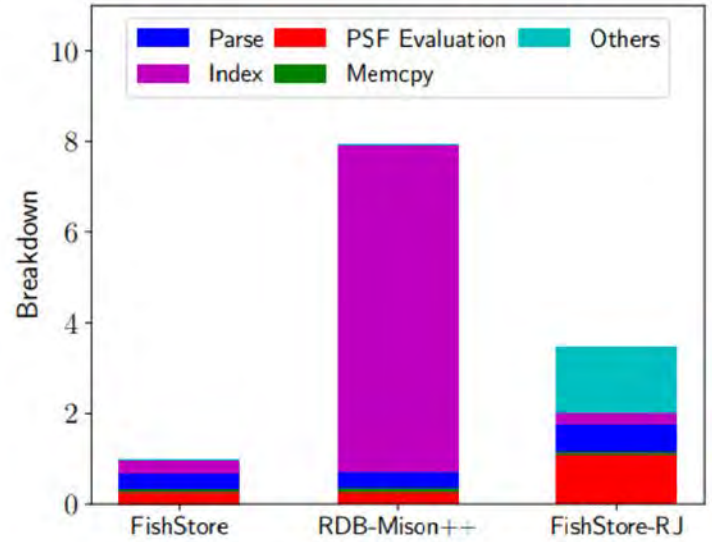
(a) Github



(b) Twitter



(c) Simple Twitter



(d) Yelp

Figure 13: Ingestion in Main Memory: CPU Breakdown

FISHSTORE increases speed with faster parsing and indexing

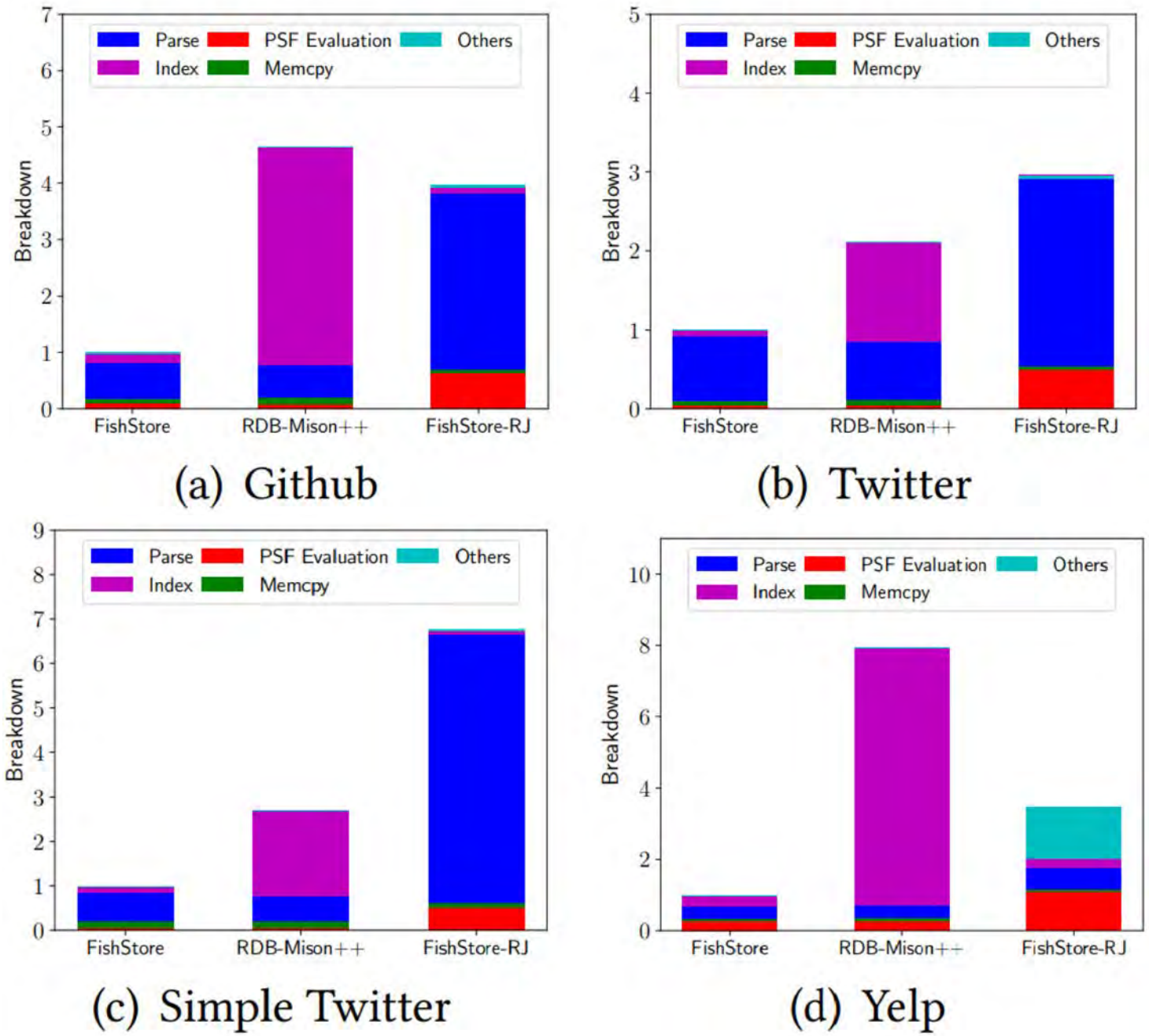
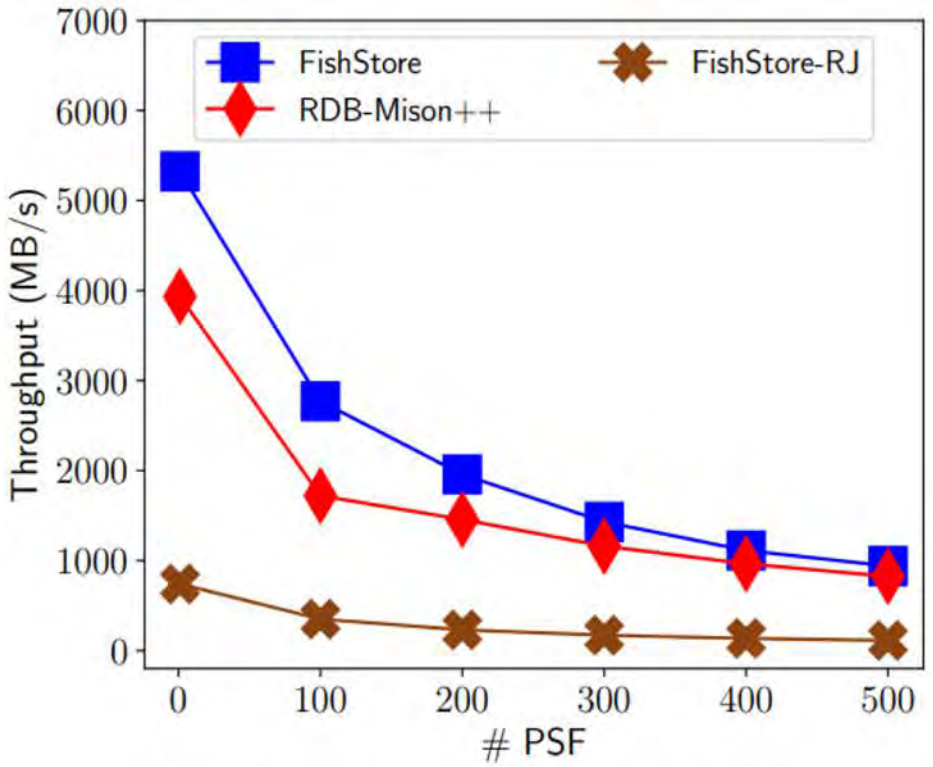
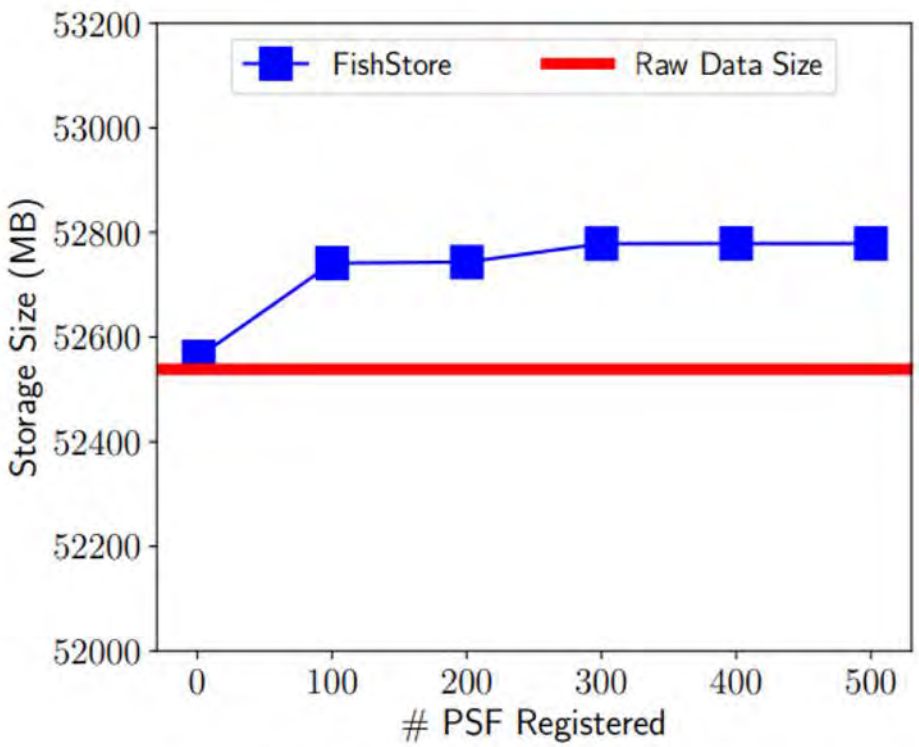


Figure 13: Ingestion in Main Memory: CPU Breakdown

*HIGHER IS BETTER



(a) Ingestion Throughput

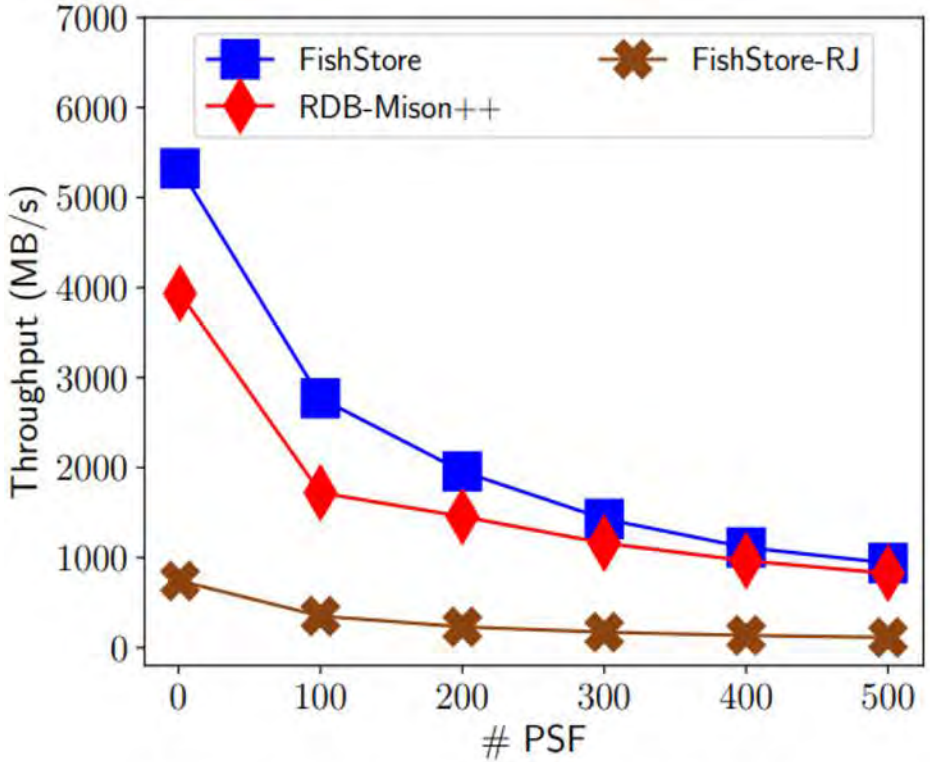


(b) Storage Cost

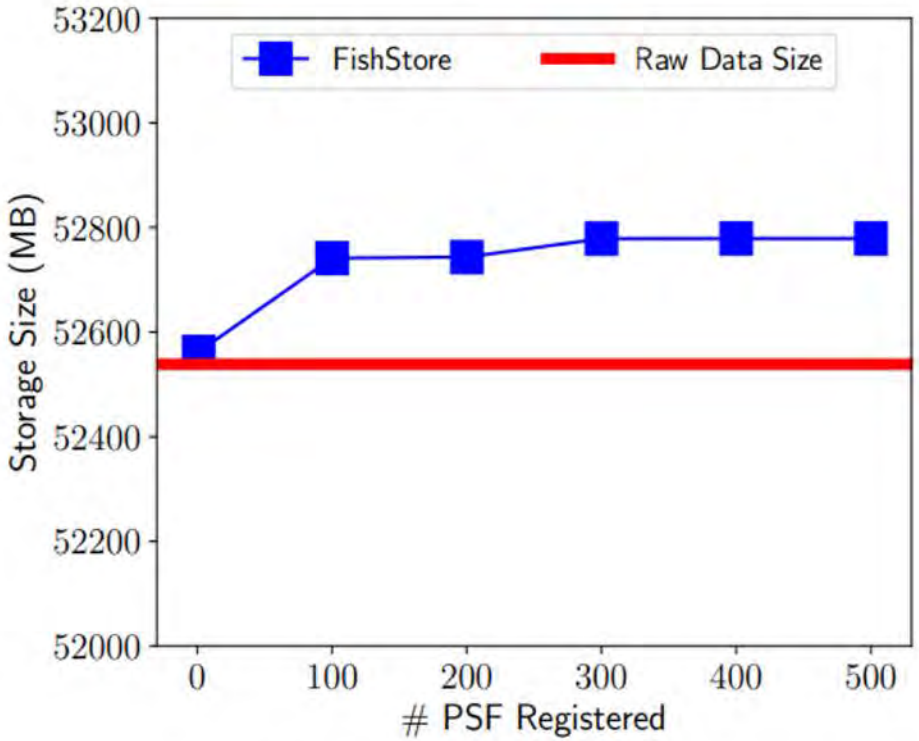
Figure 15: Predicate based PSF Scalability

*HIGHER IS BETTER

Ingestion speed goes ↓ as the # of PSFs ↑!



(a) Ingestion Throughput

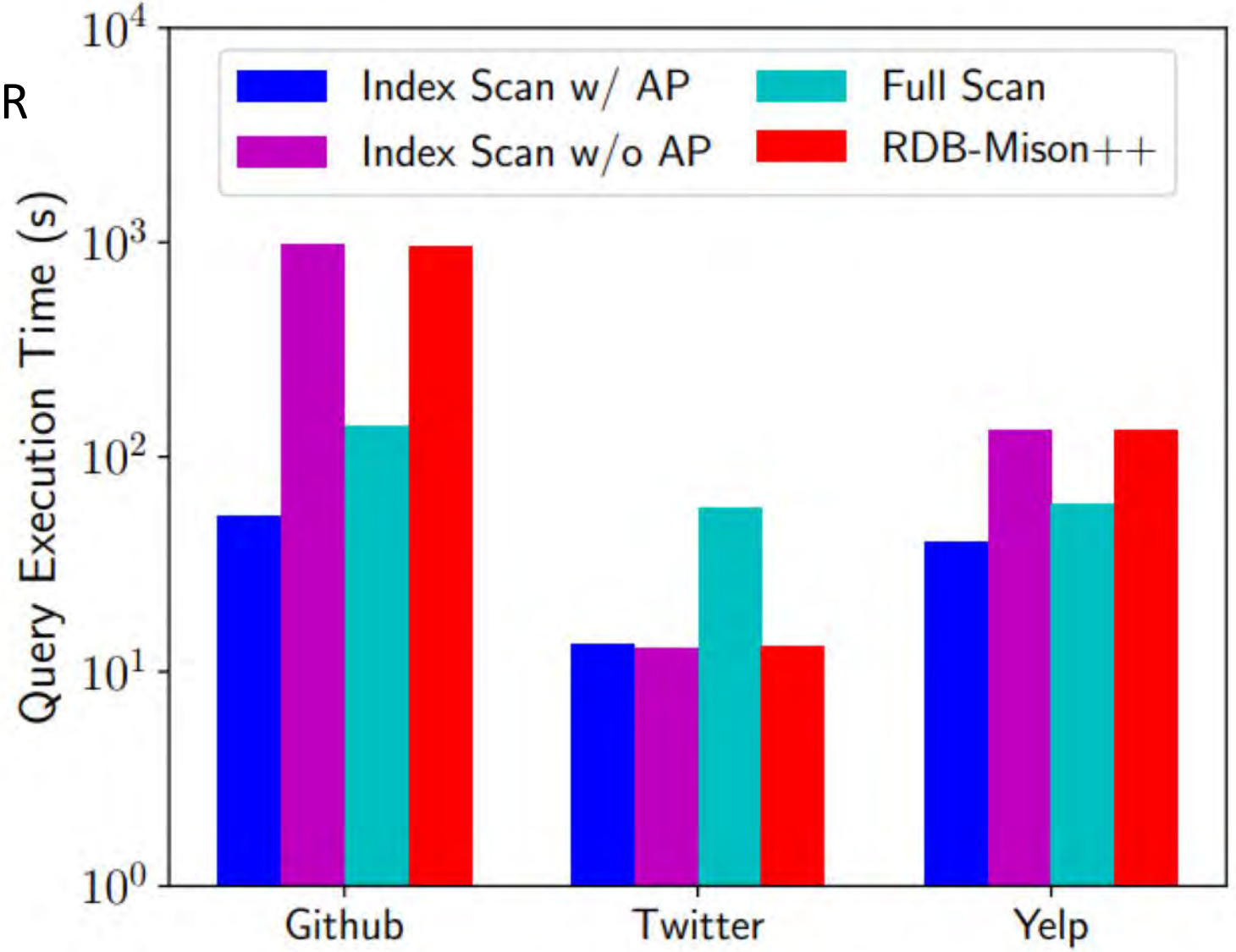


(b) Storage Cost

Figure 15: Predicate based PSF Scalability

FISHSTORE SUBSET RETRIEVAL PERFORMANCE

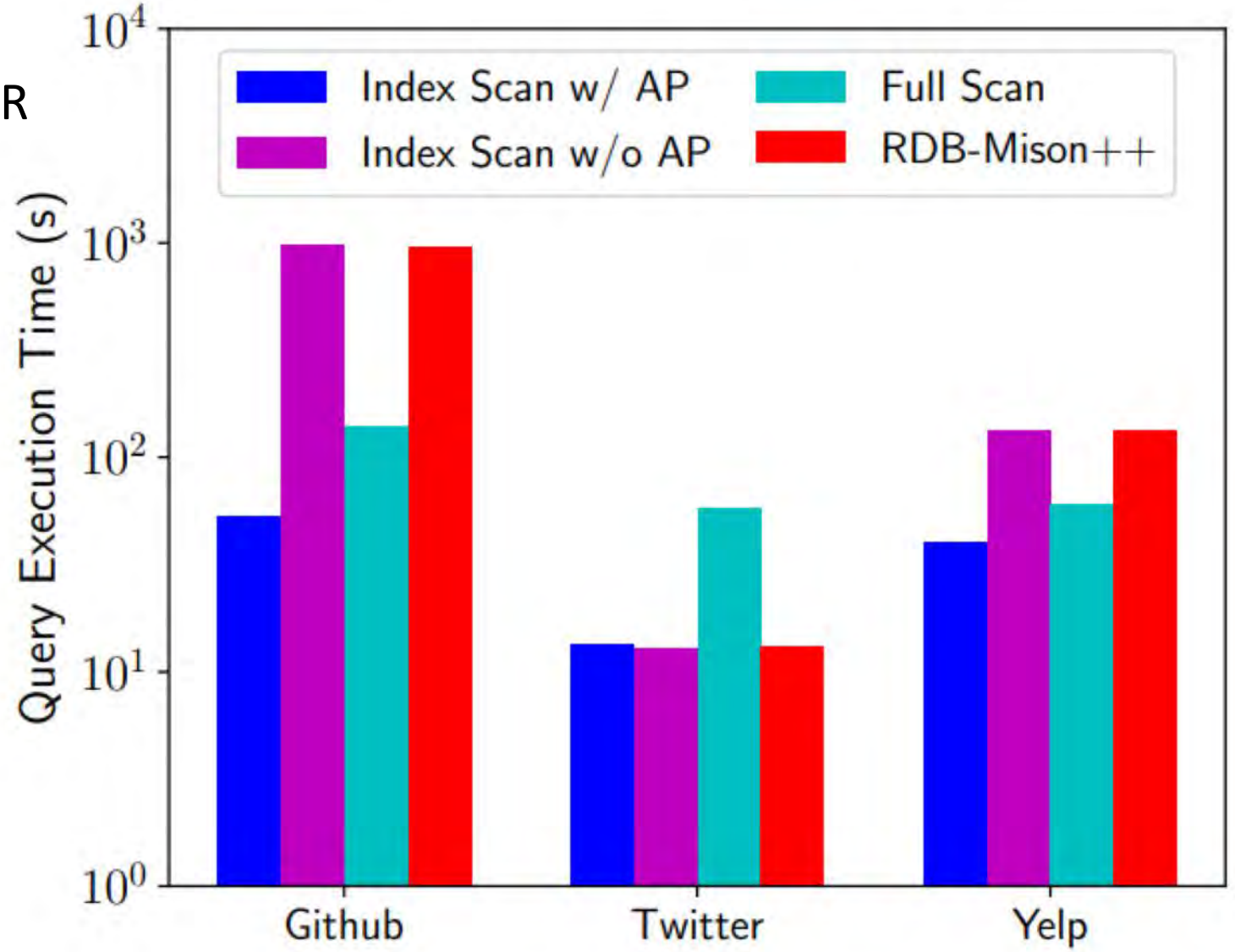
*LOWER IS BETTER



(a) Execution Time Summary

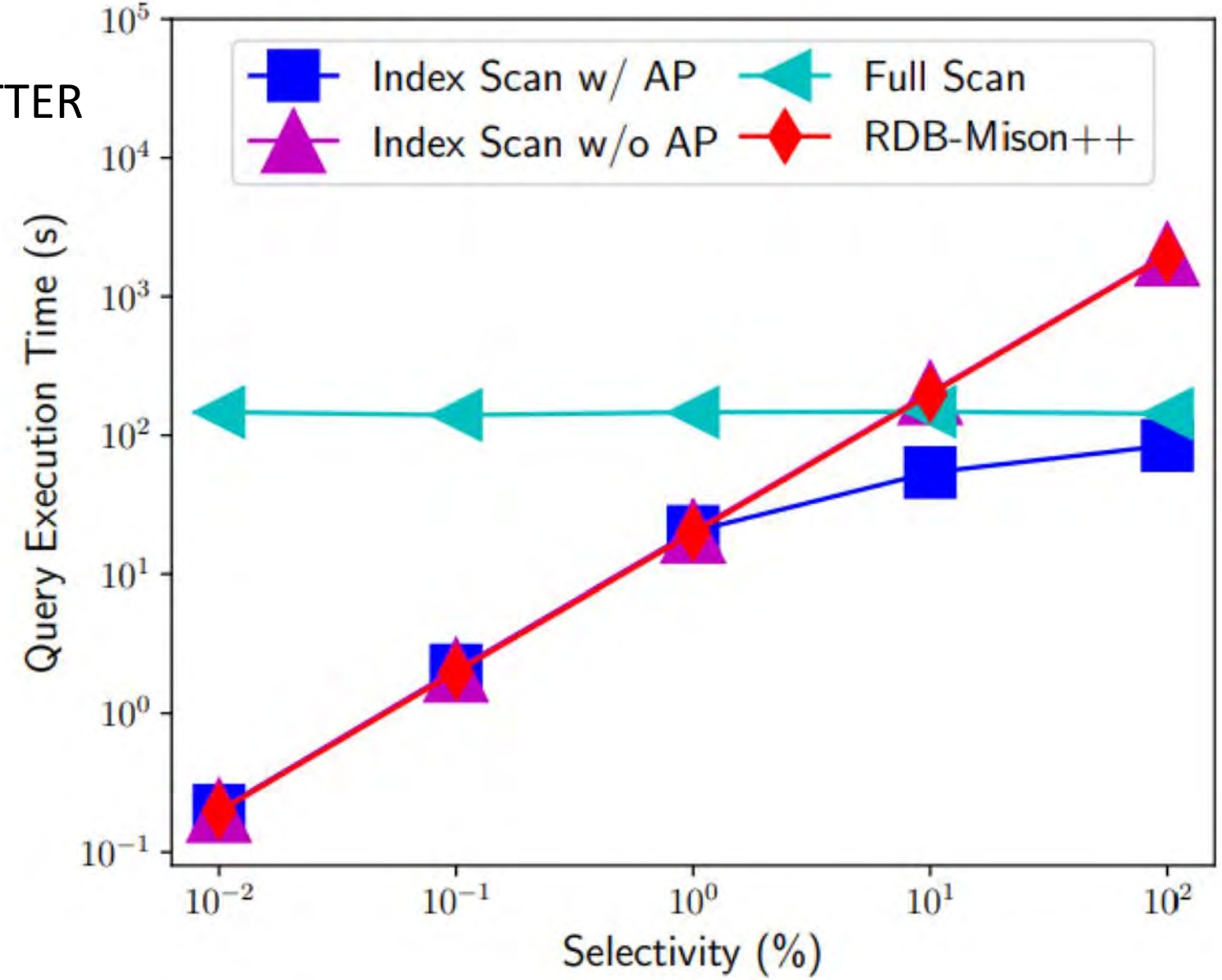
*LOWER IS BETTER

Prefetching good: issue random I/Os for selective queries and okay performance on non-selective ones!



(a) Execution Time Summary

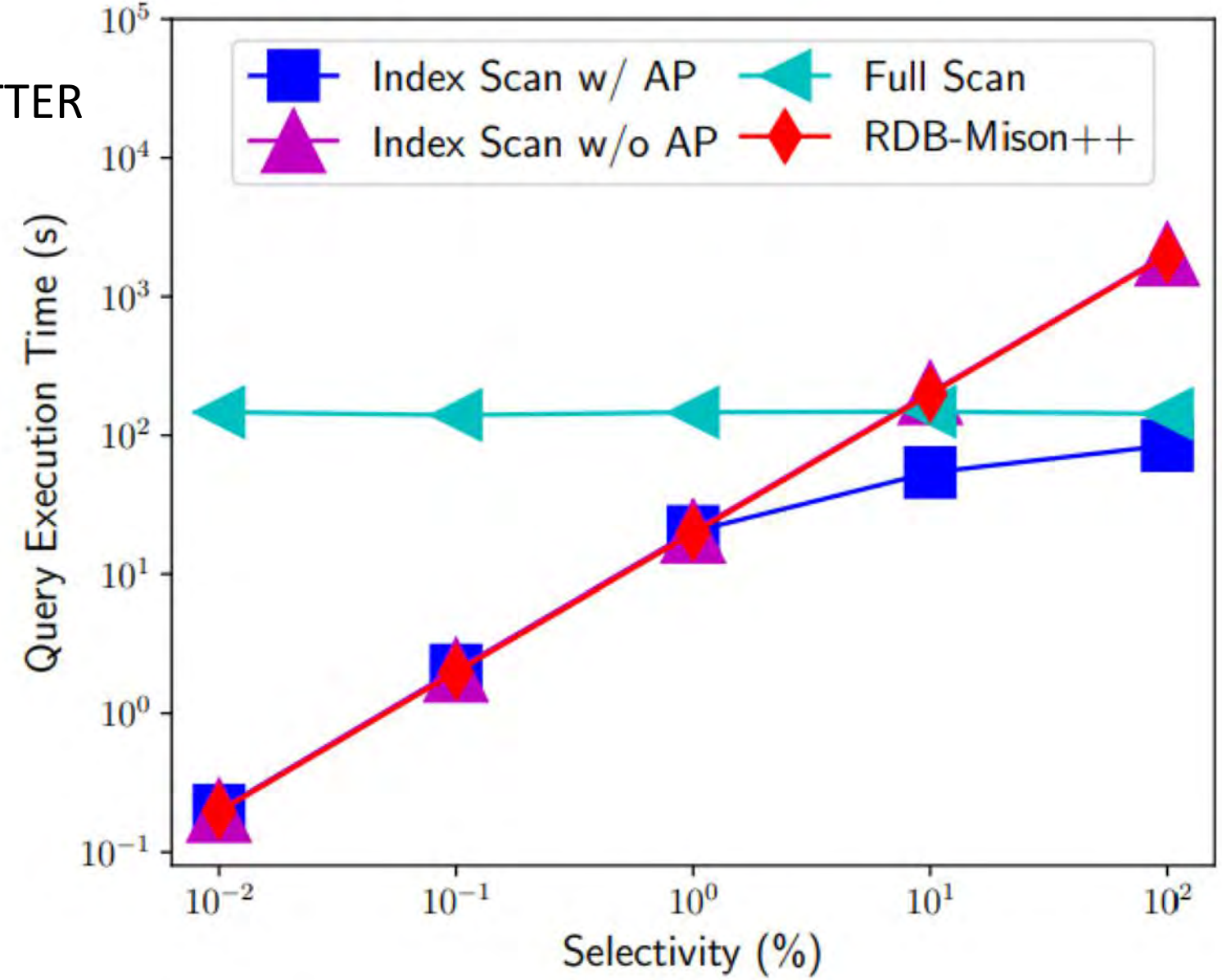
*LOWER IS BETTER



(b) Effect of Selectivity

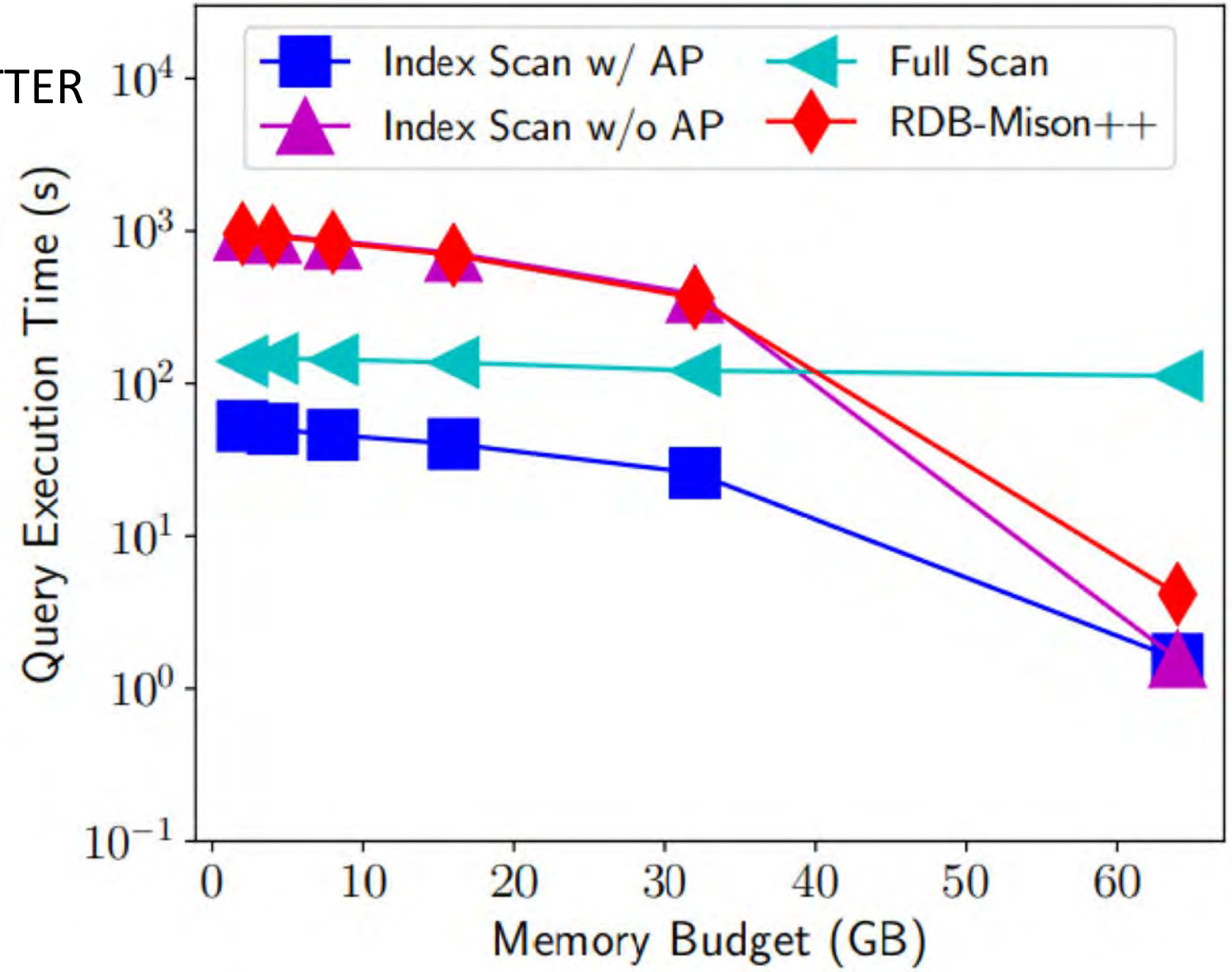
*LOWER IS BETTER

Due to adaptive prefetching, index scan does not become as slow as full scan!



(b) Effect of Selectivity

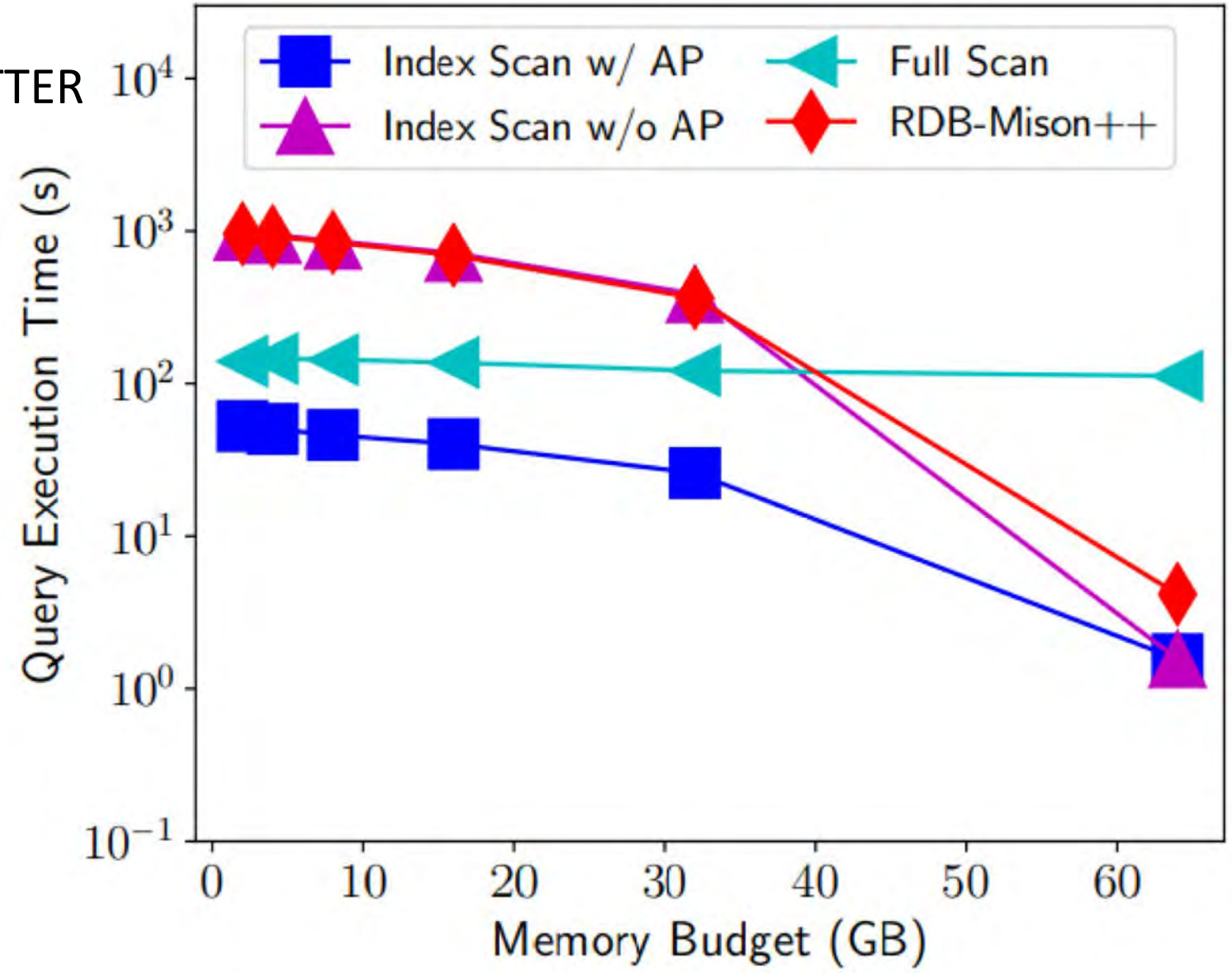
*LOWER IS BETTER



(c) Effect of Memory Budget

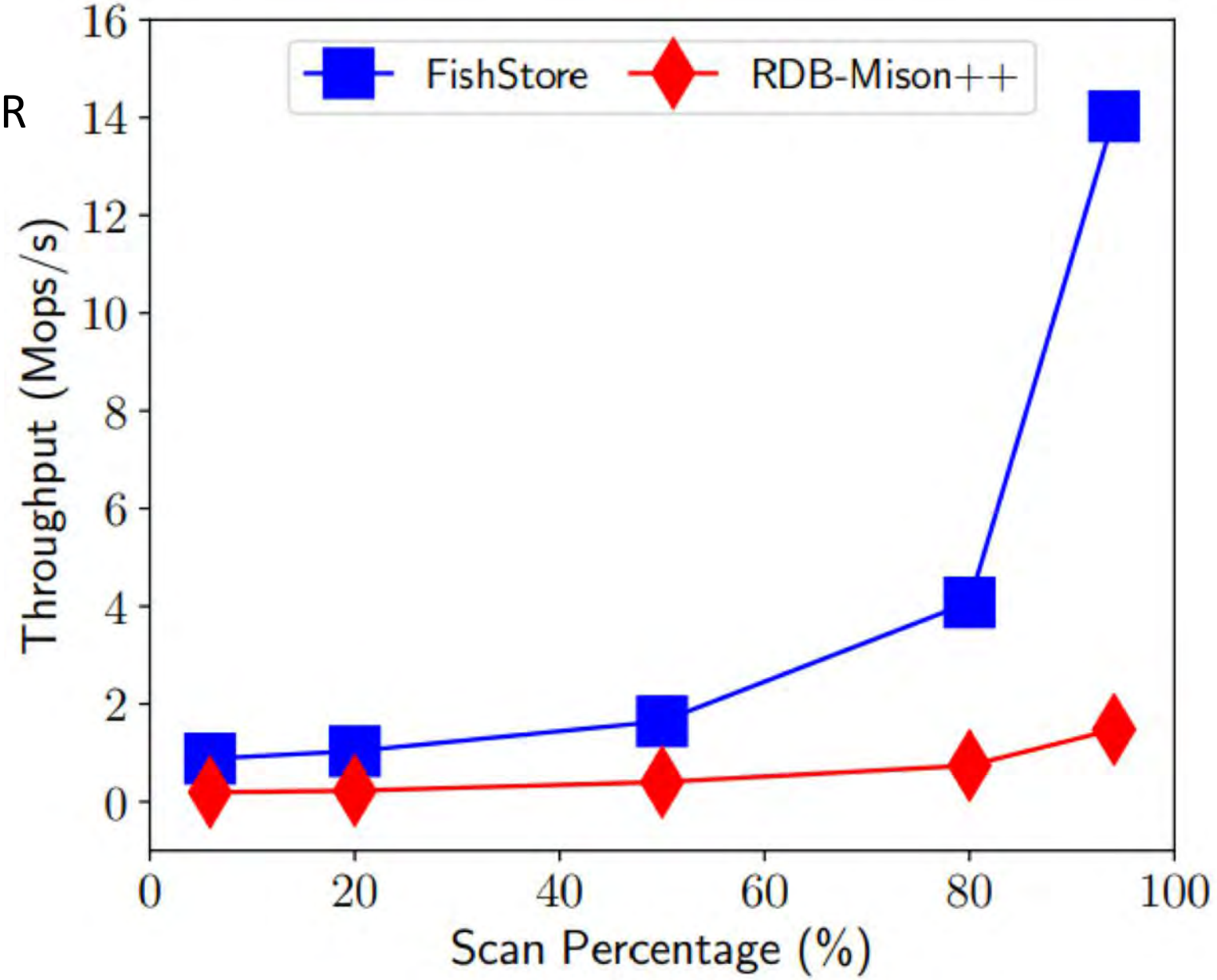
*LOWER IS BETTER

Index scan with and without adaptive prefetching benefit from having more memory!



(c) Effect of Memory Budget

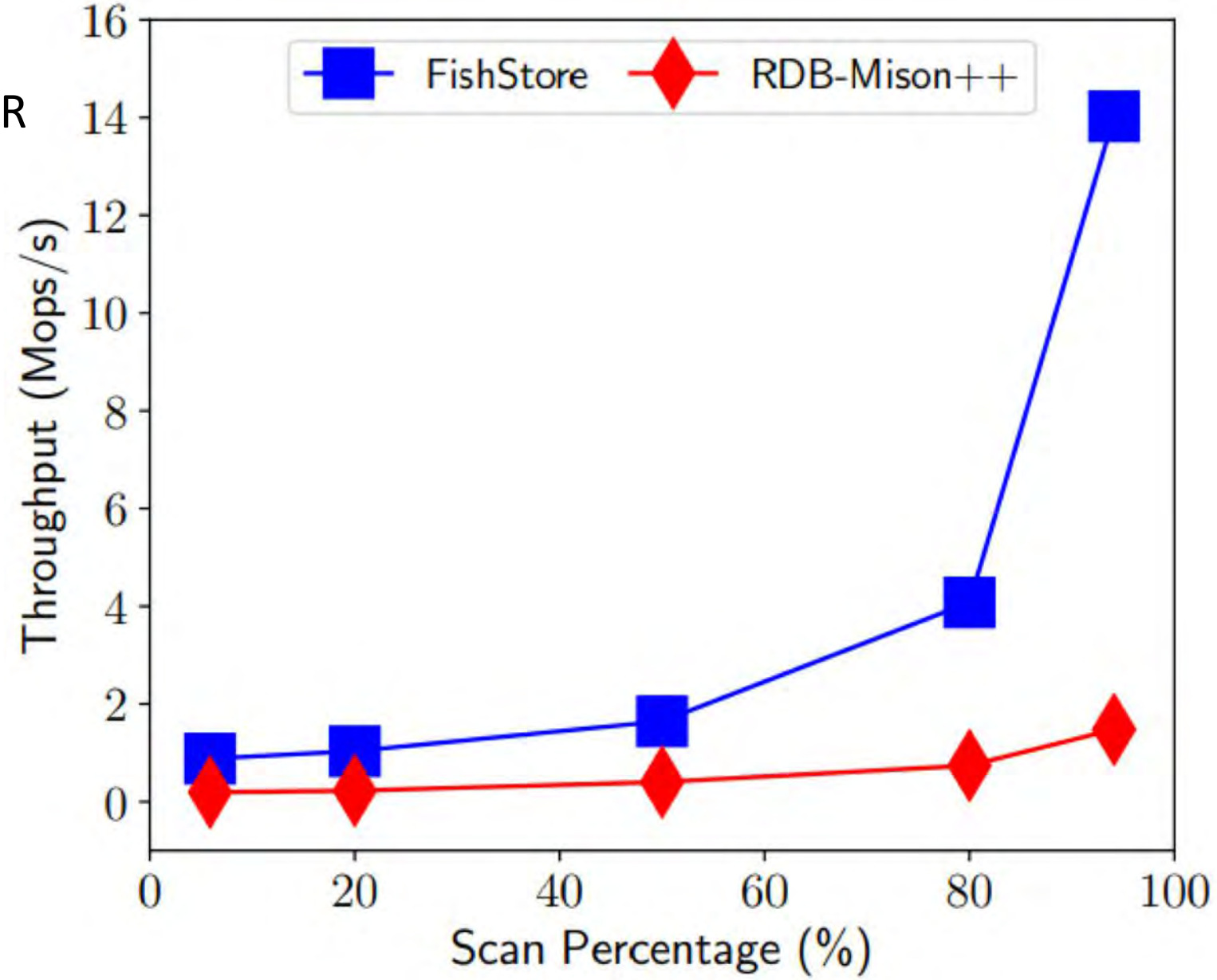
*LOWER IS BETTER



(d) Mixed Workload

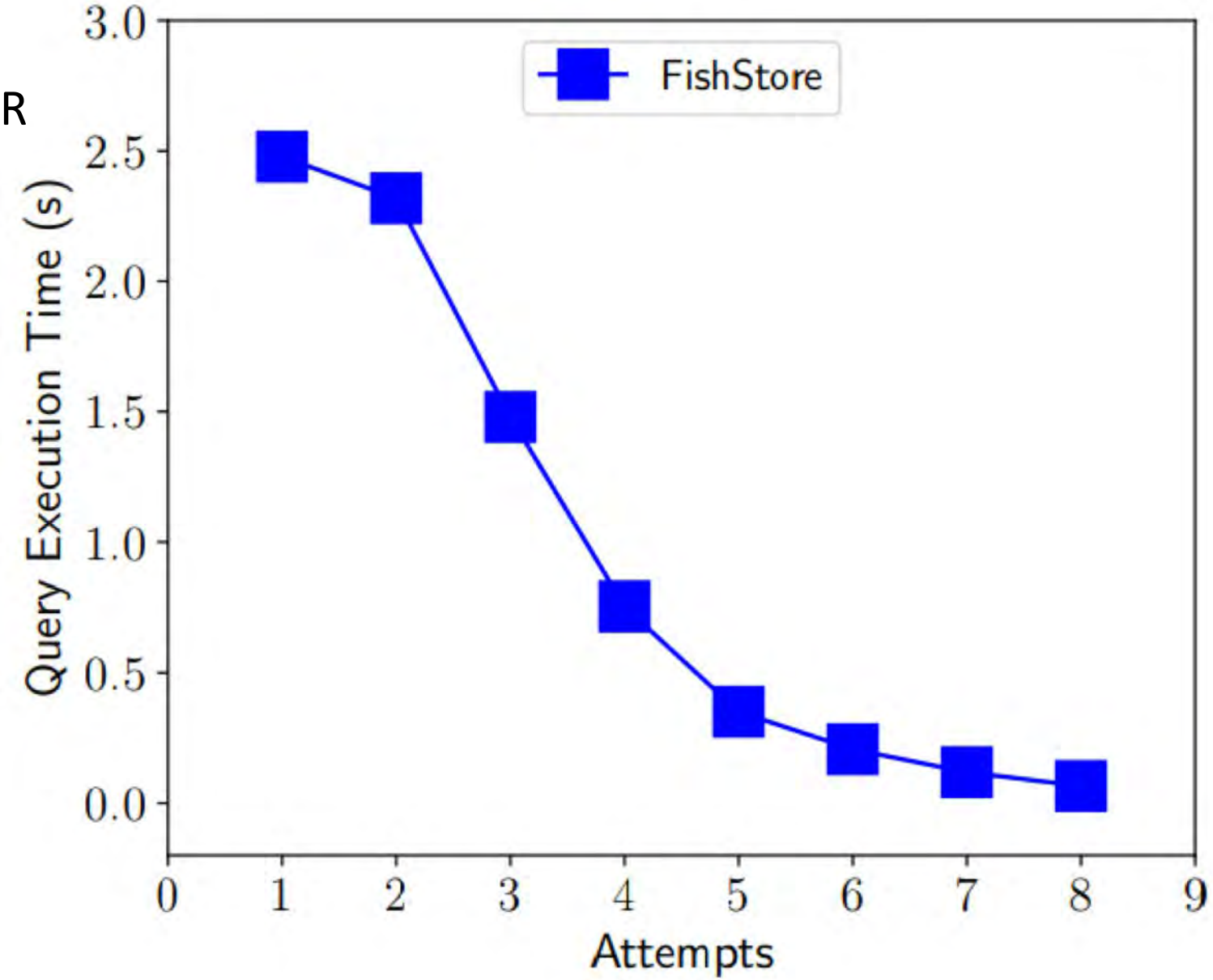
*LOWER IS BETTER

FishStore achieves higher throughput, because short scans are cheaper with no parsing or index update



(d) Mixed Workload

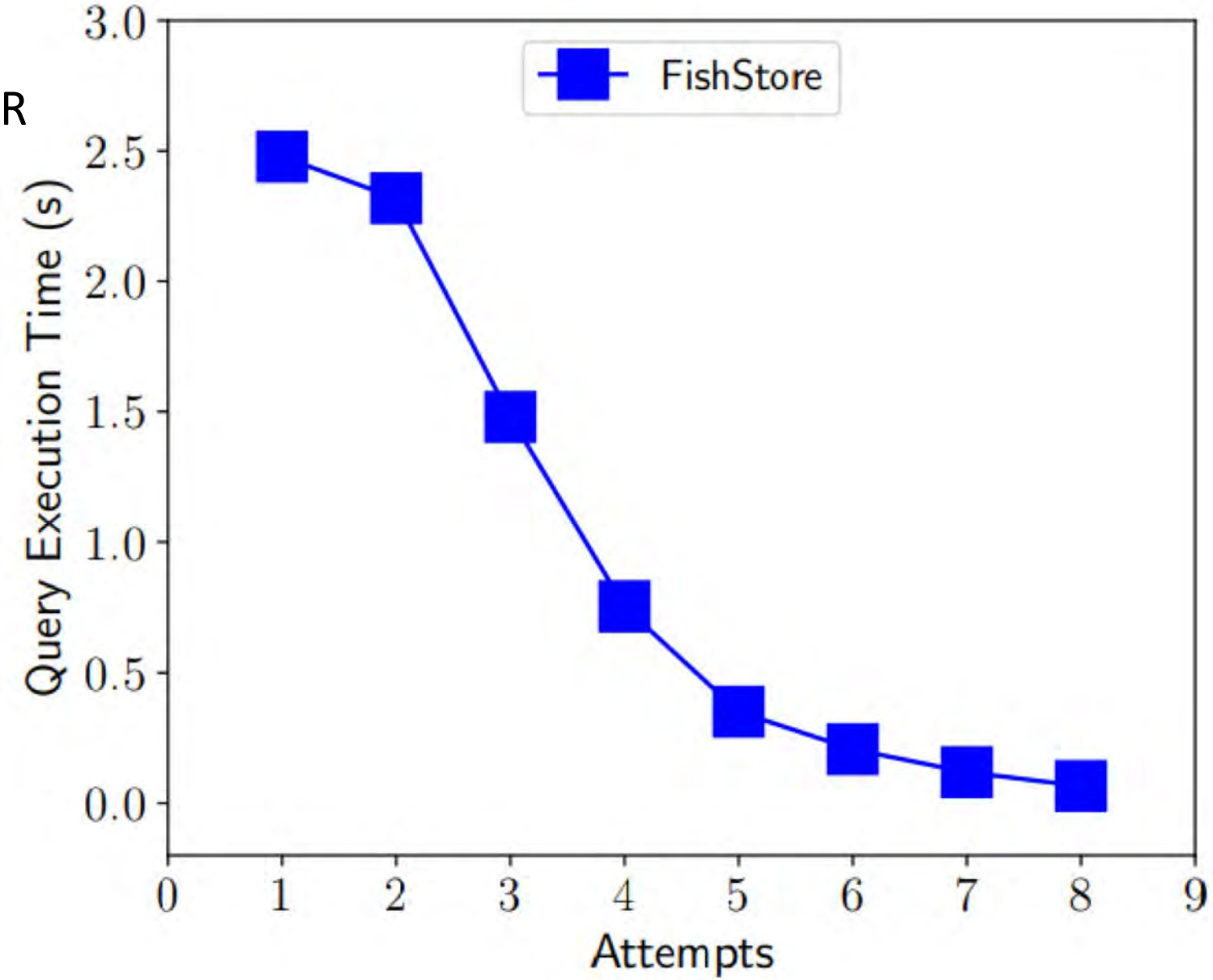
*LOWER IS BETTER



(e) Recurring Query

*LOWER IS BETTER

FISHSTORE performance betters because less of the data requires a full scan!



(e) Recurring Query

CONCLUSION

- Huge volumes of structured and unstructured data are being ingested into the cloud from a variety of data sources which no current methodology can ingest at high throughput with low CPU cost.
- Building on faster, FISHSTORE is a concurrent storage layer for data with flexible schema, based on the notion of hash indexing user chosen, dynamically registered predicated subset functions.