FishStore: Faster Ingestion with Subset Hashing

Dong Xie[§][*] Badrish Chandramouli[†] Yinan Li[†] Donald Kossmann[†]

[†]Microsoft Research [§]University of Utah
dongx@cs.utah.edu,{badrishc,yinali,donaldk}@microsoft.com

ABSTRACT

The last decade has witnessed a huge increase in data being ingested into the cloud, in forms such as JSON, CSV, and binary formats. Traditionally, data is either ingested into storage in raw form, indexed ad-hoc using range indices, or cooked into analytics-friendly columnar formats. None of these solutions is able to handle modern requirements on storage: making the data available immediately for ad-hoc and streaming queries while ingesting at extremely high throughputs. This paper builds on recent advances in parsing and indexing techniques to propose FishStore, a concurrent latch-free storage layer for data with flexible schema, based on multi-chain hash indexing of dynamically registered *predicated subsets* of data. We find predicated subset hashing to be a powerful primitive that supports a broad range of queries on ingested data and admits a high-performance concurrent implementation. Our detailed evaluation on real datasets and queries shows that FishStore can handle a wide range of workloads and can ingest and retrieve data at an order of magnitude lower cost than state-of-the-art alternatives.

ACM Reference Format:
Dong Xie, Badrish Chandramouli, Yinan Li, and Donald Kossmann. 2019. FishStore: Faster Ingestion with Subset Hashing. In *2019 International Conference on Management of Data (SIGMOD '19), June 30-July 5, 2019, Amsterdam, Netherlands.* ACM, New York, NY, USA, 18 pages. https://doi.org/10.1145/3299869.3319896

1 INTRODUCTION

Over the last few years, driven by the increasing importance of the cloud-edge architecture, we have been witnessing a huge increase in data being ingested into the cloud from a variety of data sources. The ingested data takes various forms ranging from JSON (a popular flexible nested data format with high expressive power) to relational-style data in CSV (comma-separated values) format, and binary formats such as Google Protocol Buffers [13] and Apache Thrift [10].

Given the huge ingested data volume, the goal for ingestion has traditionally been to ingest data as fast as possible, saturating storage bandwidth and incurring minimal CPU overhead. These goals usually result in simply dumping raw data on storage. More recently, however, there is an increasing need [17, 33] to make the ingested data available "immediately" for an ever-increasing range of analytic queries:

- *Ad-hoc analysis* queries that scan data over time ranges (e.g., last hour of data). The scan may (1) include complex predicates over possibly nested fields; (2) involve custom logic to select a varying (but usually small) number of records; and (3) access a small number of fields.
- *Recurring queries* that have identical predicates, but are repeated over different time ranges (e.g., execute a report over the last hour of data, repeated every hour).
- *Point lookup queries* that are based on various keys, e.g., join keys in case of streaming joins, that lookup the data, often over a recent window.
- *Streaming queries* that are fed parts of the ingested data satisfying custom predicates and based on the query schema.

1.1 Today's Solutions

The traditional solution is to ingest data in raw form and then make the data available for offline queries using periodic batch jobs that load data into a warehouse, e.g., in an optimized format such as Parquet [9]. This process is highly CPU intensive and slow, incurs high latency before the data is available for ad-hoc or repeated queries, and does not help with point lookups or streaming queries, making it unsuitable for our target applications. Alternatively, we can fully parse records and either load them into a database or update a secondary range index over every (nested) attribute and prefix during ingestion. However, full parsing, database loading, and full secondary index creation are slow. For example, we found that a typical JSON parser can only do full parsing at a speed of around 100MB/sec per CPU core [36].

[*]Work performed during internship at Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

SIGMOD '19, June 30-July 5, 2019, Amsterdam, Netherlands
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-5643-5/19/06…$15.00
https://doi.org/10.1145/3299869.3319896

1.2 New Trends in Parsing and Indexing

Recently, raw parsers such as Mison [36], Sparser [42], and FAD.js [21] have transformed the parsing landscape by achieving speeds of more than 2GB/sec per core. They run on a single thread and exploit batching, SIMD parallelism, and the targeted parsing of a few fields to achieve high throughput. However, we find that simply plugging in a fast parser into today's solutions does not help with ingestion because we have to parse all fields. A modified approach, where only a few fields are indexed, can relieve the parsing bottleneck, but does not improve ingestion because the bottleneck shifts to the heavy range indices such as RocksDB [48] and Bw-Tree [35] used in practice, which incur heavy write amplification [44], random I/Os, and CPU overheads.

Persistent key-value stores such as FASTER [24] have recently been shown to offer unprecedented performance at very low CPU cost – more than 150 millions ops/sec on a modern CPU. FASTER consists of a lightweight cache-optimized concurrent hash index backed by a record-oriented *hybrid log*. The log is ordered by data arrival and incurs no write amplification. A large portion of the log tail is retained in an in-memory circular buffer. While promising, such indices are designed to serve point lookups, inserts, and updates, and as such are insufficient for our target applications.

1.3 Introducing FishStore

In this paper, we advocate a different approach. We introduce a new storage layer for flexible-schema data, called *FishStore*[1], that combines fast parsing with a hash-based primary *subset index*. First, FishStore takes as input a generic *data parser* that exposes the ability to efficiently parse a batch of records and extract a given set of fields from each record in the batch. Second, FishStore allows applications to dynamically register (and deregister) *predicated subset functions* (*PSFs*) over the data. Briefly, PSFs allow applications to identify and efficiently retrieve different subsets of records, and work as follows. Users provide a function $f : R \rightarrow D$ that maps each record $r \in R$ to a value d in domain D, based on a given set of *fields of interest* for the PSF. FishStore allows users to retrieve all records satisfying a given PSF and value.

This paper shows that PSF-based indexing is powerful yet admits an efficient and scalable implementation. For example, it can support point lookups, equi-joins, selection predicates, prefix queries, and predefined range queries over the data.

Example (Machine Telemetry). *Consider the application depicted in Fig. 1, where machines report telemetry data for ingestion (Sec. 2 has the details). An analyst wishes to investigate machines with low CPU and high memory utilization. They register a PSF f_1 that indexes records with CPU usage lower than 15% and memory usage greater than 75%. Records matching this condition are now indexed and available for subsequent analysis. As another example, they may wish to index (or group) the data by machine name using PSF f_2, which allows drilling down into a particular machine's logs.*

Sec. 2 describes PSFs in detail and provides more examples of its use in our target applications involving ad-hoc, recurring, and streaming analysis.

1.4 FishStore Components

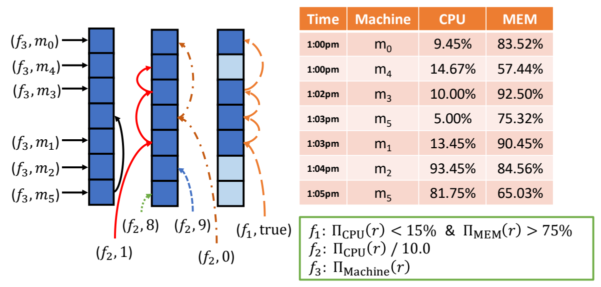
We overview the FishStore system and its challenges in Sec. 4. Briefly, it consists of two major components: (1) ingestion and indexing; and (2) subset retrieval.

Ingestion & Indexing. FishStore ingests data concurrently into an immutable log (in ingestion order) and maintains a hash index. For every active PSF f and non-null value $v \in D$, we create a hash entry (f, v) that links all matching log records for that entry in a hash chain. Based on the registered PSFs, the desired fields are provided to the parser for each data batch. FishStore evaluates the active PSFs and creates or updates hash chains. Unlike hash key-value stores, a record may be part of more than one hash chain, with a variable length record header of pointers to fields and other records. Therefore, we develop a new hash index with latch-free PSF registration (Sec. 5) and latch-free data ingestion with multiple hash chains (Sec. 6).

Subset Retrieval. FishStore supports scans for records matching PSF values (f, v) over a part of the ingested log, and returns the requested fields for matching records. FishStore does not build new indices on older data; therefore, the hash chain for a PSF may not cover the entire log. Hence, FishStore performs an adaptive scan that combines full scans and index lookups. Interestingly, even within the indexed portion of the log, based on selectivity, it may sometimes be preferable to perform a full scan [34]. FishStore performs an adaptive mix of index traversals and full scans to get the highest performance (Sec. 7).

To recap, FishStore combines fast parsing with lightweight dynamic hash indexing to provide an extremely fast and general-purpose storage layer for analytics. PSF registration is similar in concept to dynamically attaching debuggers to the data. Ingestion performance depends on the number of active PSFs and fields of interest. This pattern retains the performance benefits of batched partial parsing, arrival-time-based logging, and hash indexing. Extensive evaluations on real workloads in Sec. 8 show that FishStore can achieve an order of magnitude higher ingestion and retrieval speeds, and can saturate a modern SSD's bandwidth (2GB/sec) using only a few cores (usually less than 8) on one machine, showing that we can use inexpensive CPUs with FishStore. Without the SSD bottleneck, FishStore achieves up to 16.5GB/sec

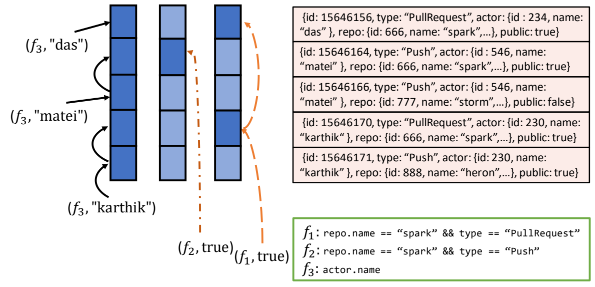
[1]FishStore stands for *F*aster *I*ngestion with *S*ubset *H*ashing Store.



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



{id: 15646156, type: "PullRequest", actor: {id : 234, name: "das" }, repo: {id: 666, name: "spark",...}, public: true}
{id: 15646164, type: "Push", actor: {id : 546, name: "matei" }, repo: {id: 666, name: "spark",...}, public: true}
{id: 15646166, type: "Push", actor: {id : 546, name: "matei" }, repo: {id: 777, name: "storm",...}, public: false}
{id: 15646170, type: "PullRequest", actor: {id: 230, name: "karthik" }, repo: {id: 666, name: "spark",...}, public: true}
{id: 15646171, type: "Push", actor: {id: 230, name: "karthik" }, repo: {id: 888, name: "heron",...}, public: true}

f_1: `repo.name == "spark" && type == "PullRequest"`
f_2: `repo.name == "spark" && type == "Push"`
f_3: `actor.name`

Figure 2: Github Data PSF Example

ingestion speed for a real workload, showing its potential on future disks, RAID, and RDMA storage.

FishStore is row-oriented with record headers, and sits early in the ETL pipeline. Older raw data (or its more refined predicated subsets) may eventually migrate to formats such as Parquet for offline analytics, e.g., using batch jobs. FishStore is compatible with, but orthogonal to, a streaming engine. FishStore serves as a storage layer for data ingestion. Streaming engines process ingested data on-the-fly, typically using transient in-memory structures. They may use FishStore to push down predicates and build shared indices over ingested data. Our work introduces the need for a query processing stack that leverages our interface to optimally execute ad-hoc, recurring, and streaming queries. These extensions are outside the scope of this paper. We cover related work in Sec. 9, and conclude the paper in Sec. 10.

2 FISHSTORE CONCEPT AND EXAMPLES

We describe predicated subset functions and show how indexed PSFs allow FishStore to be used in many applications.

2.1 Predicated Subset Functions

A central concept in FishStore is the notion of a predicated subset function (PSF), which logically groups records with similar properties for later retrieval.

Definition 2.1 (Predicated Subset Function). Given a data source of records in R, a predicated subset function (PSF) is a function $f : R \longrightarrow D$ which maps valid records in R, based on a set of *fields of interest* in R, to a specific value in domain D.

For example, the field projection function $\Pi_C(r)$ is a valid PSF that maps a record r to the value of its field C. If r does not contain field C or its value for field C is `null`, we have $\Pi_C(r) = \texttt{null}$.[2] Given a set of PSFs, a particular record may satisfy (i.e., have a non-`null` value for) several of them. We call these the *properties* of the record:

Definition 2.2 (Record Property). A record $r \in R$ is said to have property (f, v), where f is a PSF mapping R to D and $f(r) = v \in D$.

[2]Note that we do not preclude the use of `null` as a valid indexed value; we can simply map it to a different special value if desired.

As a PSF can be an arbitrary user-defined function, this abstraction covers a large range of applications. With the field projection function Π_C mentioned above, users can logically group records with the same value of field C, which is useful for operations such as joins and lookups. Similarly, if we have a boolean function P that evaluates over a record, we can use $(P, \texttt{true})$ and $(P, \texttt{false})$ to logically group the matching and non-matching records. PSFs and their fields of interest are dynamically registered, enabling ingestion of data with flexible schema.

2.2 Example: Machine Telemetry

Expanding on the machine telemetry example from Sec. 1, Fig. 1 shows a sample dataset and several PSFs. PSFs f_1 and f_2 were covered in Sec. 1. The data may be prepared for analysis by ranges of CPU usage via PSF f_3, which creates buckets for different CPU usage ranges. Fig. 1 depicts hash chains for each property (f, v). The blue boxes to the left represent header entries corresponding to records on the right. Note that different records may satisfy a different number and set of properties; for instance, the second record satisfies only two active properties.

2.3 Running Example: Github Analytics

As a more complex example, consider a dataset of Github activity in JSON format. It consists of activity such as check-ins and pull requests (Fig. 2). Initially, there are no PSFs registered, so the data is ingested without parsing or indexing. Users can prepare the data for various kinds of analyses:

- *Ad-hoc analysis*: Suppose a user wishes to analyze pull request data related to Spark. They register a PSF f_1 that parses the relevant fields (`type`, `repo.name`, etc.) and indexes records matching '`repo.name == "spark" && type == "PullRequest"`' in a chain. Analysts can then run a mix of drill-down queries over the subset, on various portions of the log. Subset retrieval uses a mix of scans (for older data) and index chain traversal (for newer data). Similarly, we could make the data ready for per-repository analysis by grouping records using `repo.name` as the PSF.
- *Recurring queries*: A user may register a PSF for a recurring query, e.g., compute an hourly list of top-k Spark committers over the last day. On registering the PSF f_2 for `true`

values of '`repo.name == "spark" && type == "Push"`', FishStore begins indexing the data, making subsequent executions of the query faster as more data gets indexed.

- *Point lookups*: Suppose a user wishes to join an input stream of tweets with Github commits from the same user in the last minute. We simply register a PSF f_3 that indexes all values of field `actor.name`, to enable such fast lookups aided by the in-memory portion of the log that retains all recent records accessible via the index. Multiple streaming joins may share the same index as well.
- *Streaming queries*: Consider a user who wishes to incrementally compute the average number of repositories created under organization "Yahoo" per hour, per country. They can register the relevant fields and predicate with FishStore, which will then deliver data as it is ingested. The now-schematized data can be fed to a streaming engine for computing query results incrementally.

3 BACKGROUND: INDEXING & PARSING

Before describing FishStore's architecture, we provide a brief background on FASTER and Mison.

3.1 Background on FASTER

This section overviews FASTER [24], our recent open-source[3] concurrent latch-free hash key-value store with support for larger-than-memory data. FASTER caches the hot working set in shared memory, and reports a scalable throughput of more than 150 million ops/sec, making it a good starting point for FishStore's index.

Fig. 3 shows the overall architecture of FASTER. Towards a scalable threading model, it adopts an epoch-based synchronization framework (Sec. 5.3) which facilitates lazy propagation of global changes to all threads via trigger actions. FASTER has two components: a *hash index* and a log structured record store called a *hybrid log*. The index serves as a map from the hash of a key to an address in a logical address space, and is checkpointed regularly. Keys that have the same hash value share a single slot in the index. All reads and updates to the slots are atomic and latch-free. The hybrid log record store defines a single *logical address space* that spans main memory and secondary storage. The system maintains a logical address offset that demarcates addresses on disk from addresses in main memory. Addresses beyond this offset (i.e., at the tail of the log) are mapped directly to a fixed-size in-memory circular buffer. The in-memory tail is further divided into an immutable and mutable region. Each record in the hybrid log contains a fixed-length header, a key, and a value. Records corresponding to keys that share the same slot in the hash index are organized as a reverse linked list: each record contains a previous address in its

[3]Link: https://github.com/Microsoft/FASTER

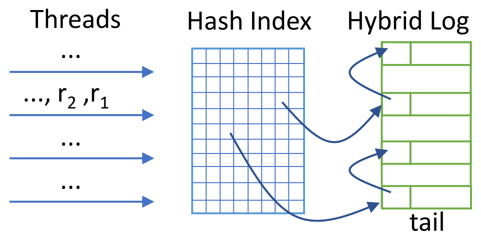


Figure 3: FASTER Index and Log Architecture

header pointing to the previous record mapped to that slot. The hash index points to the tail record of this linked list.

FishStore borrows the basic index design and epoch model of FASTER. It uses a new multi-key organization and insertion technique with variable-sized headers to support PSFs. As FishStore is designed for data ingestion, it uses the hybrid log as append-only, by disabling its mutable region. By leveraging FASTER's techniques for in-place updates in the tail the log, we believe updates can also be supported with modifications to FishStore; this is future work.

3.2 Background on Mison

FishStore exposes a generic parser interface which should support batched parsing of a few fields of interest. For performance, we plug in Mison [36], our fast parser for semi-structured data such as JSON and CSV. Motivated by the observation that applications typically make use of only a few fields, Mison can push down projections into the parser. Users specify the fields of interest when constructing the parser, for optimal performance. Mison follows a two-step approach. It first builds a structural index using SIMD to identify the positions of all fields. Then, it speculates on the schema and directly jumps to the position where it may most likely to find the user-specified fields so as to avoid wasted work parsing irrelevant fields. The parser is used unmodified by FishStore, instantiated by threads as the fields of interest vary over time (Sec. 5.3).

4 FISHSTORE SYSTEM OVERVIEW

FishStore is a storage system for data with flexible schema, that supports fast ingestion with on-demand indexing based on PSFs. We now describe FishStore's interface, provide an overview of the system, and overview the technical challenges addressed in the rest of the paper.

4.1 Operations on FishStore

FishStore supports three kinds of operations: data ingestion, on-demand indexing, and record retrieval.

Data Ingestion. FishStore receives batches of raw records from multiple threads in parallel. Based on the active fields of interest, it uses the user-provided data parser to parse specific fields. It then indexes records based on their properties and inserts them into storage in a latch-free manner

(Sec. 6). FishStore works with data larger than memory, with the most recent data in an (immutable) in-memory circular buffer. As pages are filled and made immutable, FishStore automatically pushes them to storage.

On-demand Indexing. FishStore allows users to register and deregister PSFs over a data source on-demand. Based on the set of active PSFs, FishStore builds a *subset hash index* over the properties defined by a PSF f and a value v in its domain. Specifically, for each property of interest (f, v), FishStore maintains a hash chain that contains all records $r \in R$ such that $f(r) = v$. Thus, a record may be part of more than one hash chain. Further, because only non-`null` PSF values are indexed, we can skip indexing uninteresting records in order to save storage cost. For example, we may only index records that evaluate to `true` for a boolean PSF. All index entries are built right next to the record (in a variable-sized record header) so as to reduce retrieval cost and maximize ingestion speed.

FishStore does not re-index data that has already been ingested into the system. This design implies a need to track the boundaries of an index's existence. When a PSF is registered, FishStore computes a safe log boundary after which all records are guaranteed to be indexed. Symmetrically, FishStore computes a safe log boundary indicating the end of a specific index, when the user deregisters a PSF. We can use these boundaries to identify the available PSF indices over different intervals of the log. One can orthogonally build secondary indices on historical using our techniques or standard indices such as B-trees (Appendix A).

Record Retrieval. FishStore supports retrieving records satisfying a predicate within a range of the log. Two scanning modes are supported, full scan and index scan. Full scan goes through all records and checks if the predicate is satisfied. Index scan uses hash chains to accelerate data retrieval. When records within a log range are partially indexed, FishStore breaks up the request into a combination of full scans and index scans. Note that point lookups naturally fit within this operation, and results can be served from memory if the corresponding portion of the log is in the in-memory immutable circular buffer. PSFs can support pre-defined range queries over fields (within a log range) by building the corresponding hash chains. For arbitrary range queries on older data, one may use post-filtering over pre-defined ranges or build secondary indices as noted earlier.

4.2 System Architecture

Fig. 4 shows the overall architecture of FishStore. It consists of the hybrid log serving as the record allocator, a hash index that holds pointers to records on the log, a registration service, and a set of ingestion workers.

We disable the mutable region of the log and use it in an append-only manner. When a record is ingested, FishStore

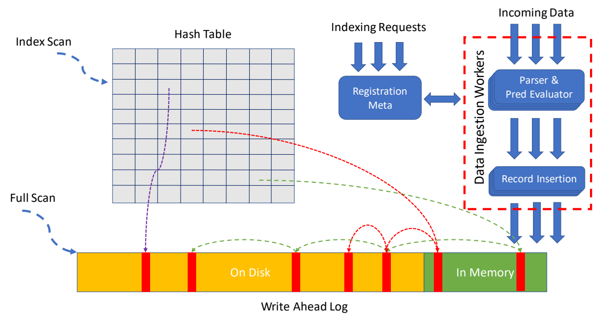


Figure 4: Overall FishStore Architecture

allocates space on the log using an atomic *fetch-and-add* operation on the tail. The tail presents in an in-memory circular buffer of pages, and filled (immutable) pages are flushed to disk. We maintain a unified logical address space across memory and disk, simplifying record indexing and access. The hash table serves as the entry point; each entry in the hash table contains a pointer to the log where records sharing the same (f, v) pair are chained together. The hash table and log hash chains together serve as our index layer. Appendix B discusses our choice of using a hash index in more details.

All indexing requests are reflected in the registration meta-data of FishStore. Through an epoch-based threading-model, indexing request are propagated to all ingestion worker threads. Based on the meta-data, incoming data are parsed and evaluated against user-defined predicates. Based on the results, ingestion workers collaboratively update the hash table and hash chains on the log in a latch-free manner.

A user can issue a subset retrieval scan of any range of the log to FishStore. The scan is satisfied by a combination of full scan and index scan operations. A full scan reads one page at a time, checking each record for the requested property. An index scan starts from the hash table and goes through the hash chain on the log, so as to retrieve all records satisfying the property. Lightweight post-processing is used to eliminate incorrect results due to hash collisions.

Challenges. Retaining high ingestion performance required a careful design that overcomes several challenges, summarized below and detailed in the rest of the paper:

- Designing a fast concurrent index which supports PSFs is non-trivial. FishStore introduces the subset hash index, which combines hashing with a carefully designed record layout, to solve this problem. (See Sec. 5)
- FishStore needs be able to acquire a safety boundary for on-demand indexing. We utilize the epoch-based threading model to help us find safe boundaries within which a specific index is guaranteed to exist. (See Sec. 5.3)
- Data ingestion should be latch-free so as to achieve high throughput on multiple threads. FishStore adopts a novel

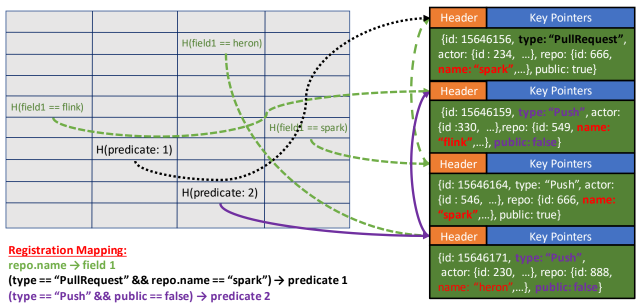

H(field1 == heron)
H(field1 == flink)
H(field1 == spark)
H(predicate: 1)
H(predicate: 2)
Header
Key Pointers
{id: 15646156, type: "PullRequest", actor: {id : 234, ...}, repo: {id: 666, name: "spark",...}, public: true}
Header
Key Pointers
{id: 15646159, type: "Push", actor: {id :330, ...},repo: {id: 549, name: "flink",...}, public: false}
Header
Key Pointers
{id: 15646164, type: "Push", actor: {id : 546, ...}, repo: {id: 666, name: "spark",...}, public: true}
Header
Key Pointers
{id: 15646171, type: "Push", actor: {id: 230, ...}, repo: {id: 888, name: "heron",...}, public: false}
Registration Mapping:
repo.name → field 1
(type == "PullRequest" && repo.name == "spark") → predicate 1
(type == "Push" && public == false) → predicate 2

Figure 5: Subset Hash Index

lock-free technique to update the index with very low cost even during heavy contention. (See Sec. 6)

- Scanning through hash chain on disk involves many random I/Os, which can hurt the performance of subset retrieval. FishStore introduces an adaptive prefetching technique which actively detects locality on disk and effectively reduces the number of I/Os issued. (See Sec. 7)

5 SUBSET HASH INDEX

We introduce a fast index called *subset hash index*, which allows us to link all records with the same property on the same hash chain. Recall that a property is described by a PSF, which is defined as $f : R \longrightarrow D$ (R is the record set), and a value $v \in D$. In other words, our goal is to index all records $r \in R$ such that $f(r) = v$. The subset hash index is organized as a hash table that indexes logical groups of records using hash chains. Each property (i.e., a (f, v) pair) is assigned a hash signature. The hash chain, which chains up all records with a specific property, starts from its corresponding hash entry in the hash table. Therefore, when retrieving records matching a property, we can calculate its hash signature, locate the hash entry, and follow the hash chain to retrieve all qualifying records.

Fig. 5 continues the Github example (Sec. 2). Suppose the user requests indices on distinct values of field '`repo.name`', and `true` values for predicate '`type == "PullRequest" && repo.name == "spark"`', and predicate '`type == "Push" && public == false`'. Using a hash signature H, we assign each property to an entry in the hash table. When the user requests records whose '`repo.name == "spark"`', we calculate $H($`repo.id`, `"spark"`$)$, find the hash entry, and follow the hash chain to retrieve all qualified records (checking for hash collisions). A record can be on several hash chains at the same time: the bottom record satisfies both '`repo.name == "heron"`' and '`type == "Push" && public == false`', and shows up on the hash chains of both properties.

5.1 Hash Signature of Properties

When constructing the subset hash index, each property needs to be assigned a hash signature. This helps distinguish different properties in the hash map and facilitates subset retrieval by providing an entry point in the hash table. As FishStore supports arbitrary properties that can be defined by $f(r) = v$, it is non-trivial to define a universal hash function for each property. To solve this problem, FishStore introduces a *naming service* that assigns deterministic IDs to PSFs. A PSF consists of a list of fields of interest and a function evaluated over these fields. When a new PSF is registered, the naming service assigns it an ID. We build the hash signature of a property (f, v) as $\mathtt{H}(f(r) = v) = \mathtt{Hash}(\mathrm{fid}(f).concat(v))$, where $\mathrm{fid}(f)$ is the ID assigned by the naming service and `Hash` is a hash function. For example, the signature of property (`repo.name`, `"spark"`) is the hash value of the concatenation of the field ID of `repo.name` and the string `"spark"`.

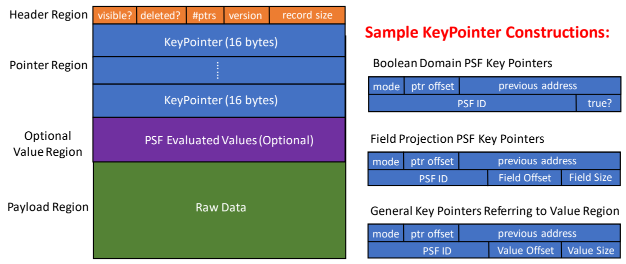


Figure 6: FishStore Record Layout

5.2 FishStore Record Layout

A key difference between the subset hash index and standard secondary indices is that its index entries live right next to the data, even if they are not clustered. With a traditional secondary index design, each index entry includes a pointer to the record allocated in a heap file to avoid duplication. However, this causes a second I/O for index-based record access. FishStore avoids this overhead by collocating records with the index during ingestion. Each record may contain multiple index entries, but only for PSFs satisfied by the record, which limits space overhead. For example, a PSF that indexes `true` values of a selective predicate incurs the header overhead only for records that match the predicate. Due to the typically large flexible-schema payloads (1KB to 11KB – see Sec. 8.1), our layout introduces low storage and scan overhead (see Sec. 8.3). We note that these overheads may be higher for small binary payloads that match many active PSFs. In such cases, and for indexing previously ingested data, we can store key pointers separately, at the cost of an additional cache miss or I/O during lookup (see Appendix A).

Fig. 6 shows the physical layout of a record in FishStore, which contains four regions: header region, pointer region, an optional value region, and the payload region. Header region contains the basic information of the record including if it is invisible or deleted and its checkpoint version number. In addition, the record header also includes the size of record

(`record_size`) and how many properties the record owns (`#ptrs`). Note that the number of properties for a record is also the number of hash chains containing the record. The optional value region may contain evaluated values for some registered PSFs, and is described below. Finally, the payload region contains the raw data in bytes with its length.

The pointer region contains a variable number of fixed-sized key pointers which serve as index entries in the hash chain of the subset hash index. Each key pointer contains an offset that can help navigate back to the record header, and a previous address pointing to the previous entry in the hash chain. Note that it is the key pointers that form the hash chain rather than the records. The hash entry in the hash table and the previous address in a key pointer always point to the corresponding key pointer of a record that has a property with the same hash signature. This design makes hash chain traversals efficient: if key pointers instead pointed to the record header, we would have to check each key pointer in order to find the right hash chain to follow.

Each key pointer contains enough information for a reader exploring the hash chain to check whether the current record satisfies the property or the record was accessed due to a hash collision. A key pointer needs to contain the identification of its corresponding PSF f and a way to access the evaluated value of f over the record. If the evaluated value is small (e.g., a boolean), it may be stored inline. Otherwise, the key pointer contains a pointer to the evaluated value either directly in the payload, or in the optional value region.

We use a few bits (`mode`) to distinguish different classes of key pointers. Fig. 6 shows some example modes. When the PSF domain is boolean, we simply inline the PSF ID along with a bit representing true or false inside the key pointer. If the PSF evaluates to a variable length value, we need to store a pointer to the evaluated value. For instance, if the PSF projects a field, we store the offset and size of the field in the payload, as part of the key pointer. If the value is not available in the payload, we can evaluate it during ingestion and store it in the optional value region. The key pointer would then store the offset and size of this evaluated value.

5.3 On-Demand Indexing

In FishStore, users user can start or stop building indices over a PSF without blocking data ingestion. To achieve this, we take advantage of the epoch framework [24]. In an epoch-based system, we maintain a shared atomic counter E, called the current epoch, that can be incremented by any thread. Each thread T has a thread-local version of E called E_T. Threads refresh their local epoch value in order to catch up the shared global counter E periodically. When the minimum value of all thread-local epoch number is greater than c, we call epoch c safe. This indicates all threads are aware of changes made up to epoch c and we can process actions

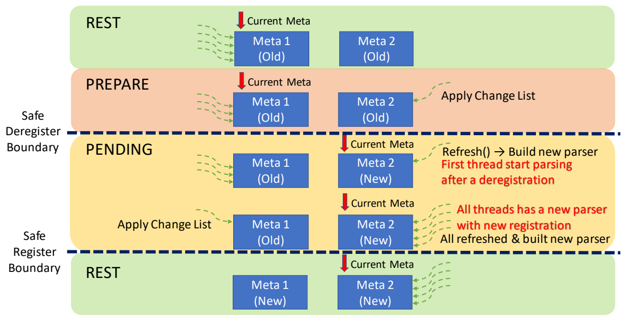
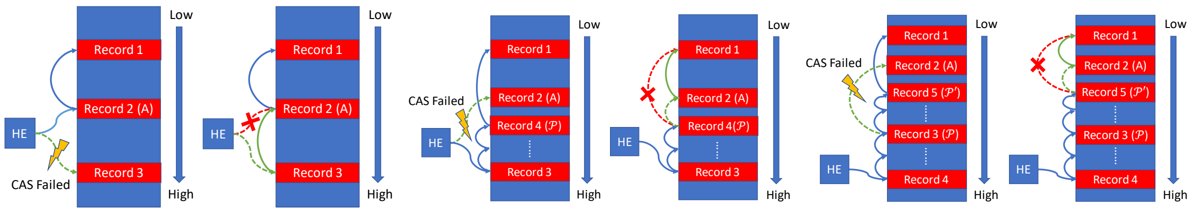


Figure 7: On Demand Indexing

based on the system state in epoch c. The epoch framework provides an interface for users to define a trigger function that automatically evaluates when an epoch becomes safe.

FishStore uses a multi-stage approach to apply changes to index meta-data. Fig. 7 shows the procedure for applying a list of index registration and deregistrations. FishStore keeps two versions of index meta-data; all ingestion workers refer to one of them as the current meta-data. When there are no index altering requests, the two versions are identical, and FishStore is in state REST. When the user pushes a list of index altering requests to FishStore, the system shifts its stage to PREPARE and immediately applies these changes to the inactive meta-data. Then, FishStore sets the current meta-data to the one with applied changes, and increments the global epoch with an action to apply the changes to the unchanged version. Then, the system stage shifts to PENDING, and all threads start to observe that the current meta-data has been changed when they refresh their epoch. This refresh may require the ingestion worker to rebuild its parser so as to parse newly registered fields, and shift its meta-data to the new one. When the epoch for applying index altering requests becomes safe (which also means all ingestion workers have observed and applied changes), FishStore automatically applies all changes to the old meta-data. This makes both meta-data identical as before, and the system shifts its state back to REST.

Since indexing is on demand, FishStore needs to maintain and provide information about which log interval ranges have which subset indices available, so that returning results via the index is sound. Thus, we need to derive the safe boundary for both index registration and deregistration. Note that when the system state shifts from PREPARE to PENDING, no ingestion worker would have yet stopped indexing the deregistered properties. Thus, the tail of the log at that instant can serve as the safe deregistered boundary. Similarly, when the system state shifts from PENDING to REST, all threads would have started indexing the newly registered properties. Hence, the tail of the log at that instant can serve as the safe registration boundary.



(a) Case 1 (b) Case 2 (c) Case 3

Figure 8: Handling CAS failure.

6 DATA INGESTION IN FISHSTORE

FishStore allows concurrent data ingestion; each ingestion thread is called an *ingestion worker* and is registered with the epoch framework. The incoming data goes through four phases before completing ingestion into FishStore: (1) parsing and PSF evaluation; (2) record space allocation; (3) subset hash index update; and (4) record visibility. We next go through these phases in more details.

6.1 Parsing and PSF Evaluation

When incoming data lands at an ingestion worker, it uses the provided parser interface to parse out the active fields of interest in a batch of data. Thus, FishStore can benefit from both batched SIMD-based parsing and the capability of parsers to target a smaller subset of fields efficiently. In FishStore, each data ingestion worker owns a thread-local parser. Parsers typically require users to indicate the set of fields to be parsed when being instantiated. Therefore, whenever a worker detects changes in the fields of interest (due to PSF registration and deregistration), it recalculates the minimum field set for index building and recreates its thread-local parser. Finally, after parsing out essential fields, the worker annotates the position of a field and evaluates all requested PSFs for each record. This provides necessary information for indexing building when the record is actually ingested into FishStore.

6.2 Record Space Allocation

As mentioned in Sec. 4.2, the space for ingested records is allocated on the log. When the data ingestion worker finishes Phase I, it can compute how much space the record requires when ingested into FishStore. For example, a record whose raw data size is s and has k properties, it will requires $8 + 16k + \lceil \frac{s}{8} \rceil \times 8$ bytes when the optional value region is empty according to Fig. 6. Next, the worker shifts the tail of log atomically by the total space that incoming records require. The tail of the log is maintained in an in-memory circular buffer for concurrent read access. We cover the details of concurrent log maintenance in Appendix C. Finally, the worker copies the raw data to the payload region and fills up the record headers. Since we have yet to update the index, the record is inserted with its *visibility bit* in the header set to `false`. Any reader that touches an invisible record ignores it. This guarantees the atomicity of record insertion.

6.3 Index Update & Record Visibility

Once a record is successfully allocated on the log, the ingestion worker starts updating the hash chains of all properties that record has. For high performance, FishStore borrows the latch-free cache-friendly hash table introduced in FASTER [24]. When we wish to update a hash chain, we simply locate its hash entry in the hash table and perform a compare-and-swap (CAS) making it point to the corresponding key pointer of the record. When the CAS fails, we have to mark the record as invalid, reallocate space on the log and try again. If we simply try to CAS again without reallocating the record at the log's tail, there may exist a forward hash link pointing from a low address to a high address. This complicates the structure, as well as operations such as garbage collection, checkpointing, and failure recovery. Further, forward links cause problems when we traverse the hash chain, as we may visit a record in memory after getting into disk. Since disk I/Os in FishStore are asynchronous, we would have to protect I/O threads with the epoch framework to ensure the safety of the in-memory records they may access. However, this is very expensive due to the large number of I/O threads in practice. Thus, we need to ensure that there are no forward hash links in the log.

However, the reallocation solution above does not work well in FishStore due to two reasons: (1) Many records may share the same property, which usually results in a CAS failure. (2) Each record needs to update multiple hash chains; CAS failure on any hash chain will cause reallocation. As a result, this solution can cause substantial write amplification. It wastes disk bandwidth and makes FishStore not scalable on multiple threads (Sec. 8.3). To solve this problem, we propose a novel CAS technique that guarantees zero write amplification and no forward hash links. When a CAS fails, there are two scenarios we have to handle:

Fig. 8(a) shows the easy case where the hash entry happens to point at a lower address than the current key pointer. In this case, it is safe to try the CAS again because we still have a chance to place the record at the head of hash chain without creating a forward hash link. To do this, we simply

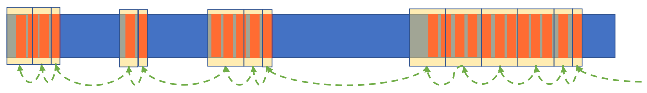


Figure 9: Adaptive Prefetching

change the previous address of the key pointer to where the hash entry pointed and try the CAS again.

When we find that the hash entry points to a higher address than the current record, Fig. 8(b) shows how we handle it. In this case, FishStore will immediately realize that we cannot perform the CAS at the hash entry for the key pointer as it will definitely cause a forward hash link. Instead of trying to swap out the hash entry, we start to find the right place in the hash chain where the key pointer should be. Denote the address of current key pointer as A. We traverse the hash chain starting from the hash entry until we find a key pointer $\mathcal{P}$ whose address is higher than A while its previous address is lower than A. It is clear that the key pointer should be swapped into the hash chain right after $\mathcal{P}$. Now we alter the current key pointer pointing to the previous address of $\mathcal{P}$ and try CAS on $\mathcal{P}$, making it point to A.

If this CAS still fails, we will never fall back to the easy case as the hash entry is already pointing to a higher address. Fig. 8(c) shows how to handle this case. Specifically, we start traversing the hash chain from $\mathcal{P}$ until we find $\mathcal{P}'$ after which the current key pointer should be swapped in, and then repeat the above process until the CAS succeeds. For more details, we outline and analyze the hash chain update algorithm in Appendix D.

Finally, the record is made atomically visible to readers (covered next), by setting the visibility bit in the header. We cover failure recovery of FishStore in Appendix E.

7 SUBSET RETRIEVAL IN FISHSTORE

This section describes the scan interface to FishStore, concurrent log maintenance to aid subset retrieval, and our adaptive prefetching technique to speed up index scans.

7.1 Subset Retrieval Interface

FishStore allows a user to retrieve all records satisfying a given property in a continuous range of the log. Specifically, user provides a property (f, v) and the desired address range in the FishStore log. Note that not all records satisfying a given property will be indexed in the log range because of on-demand indexing. FishStore breaks the request into a combination of index scans and full scans using knowledge of safe registration and deregistration boundaries for all PSFs. Full scans have to parse and evaluate the requested PSF against each record encountered.

Users are also allowed to send an early stop signal to FishStore through a `Touch` function, when they wish to terminate the scan early. This is useful, for example, when the desired record is found or the user wants a limited sample of satisfying records for further analysis.

7.2 Adaptive Prefetching

When the user issues an index scan to FishStore, it searches the corresponding hash entry in the hash table and follows the hash chain to retrieve all qualified records. When the log is totally in-memory, exploring the hash chain is not expensive. However, if the exploration lands on storage, it will issue random I/Os even when retrieved records may actually be continuously located on the log. Issuing too many random I/Os can significantly hurt scan performance, making it worse than a full scan over the log page by page. There are two reasons why issuing random I/Os on the hash chain is significantly slower than sequentially scanning the log: (1) Issuing more I/Os will cause more expensive system calls through the OS kernel. (2) I/Os required to explore a hash chain are small and dependent, i.e., we do not know the next address to request until the previous I/O has completed. As a result, the I/O queue is almost empty most of the time and the SSD's internal parallelism is not utilized.

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. Fig. 9 shows how this technique works. When the index scan hit a record on disk, it checks how many bytes are between the current record and the previous record in the hash chain. If they are close enough, we speculatively retrieve more bytes before the previous record when issuing the next I/O. In other words, if FishStore witnesses locality on the log, it issues a larger I/O request hoping that it will cover more qualified records, thereby reducing the total number of I/O issued. When the index scan is already in speculation and we witness more locality on the log, we can speculate more aggressively by issuing even larger I/O requests. When we notice a loss of locality, we fall back to random I/Os and repeat the process.

The core idea of adaptive prefetching is to trade-off disk bandwidth to reduce the number of I/Os issued. To ensure the effectiveness of our solution, we develop a mathematical model to conduct this trade-off. There are two parameters to configure adaptive prefetching, namely, the threshold to determine locality and the size of prefetching. Generally, we are willing to sacrifice the following amount of disk bandwidth in bytes to save a single random I/O:

$$\Phi = (\text{cost}_{\text{syscall}} + \text{latency}_{\text{rand}}) \times \text{throughput}_{\text{seq}}$$

In the formula above, we calculate the time cost of a random I/O and figure out how many bytes in sequential I/O will cause the same cost. Note that the number of bytes between two entries (i.e., key pointers) on the hash chain also includes the length of a record. As the I/O bandwidth for retrieving records is not wasted, FishStore sets the threshold for adaptive prefetching to be $\tau = \Phi + \text{avg}_{\text{rec\_size}}$.

Table 1: Default Workloads for each Dataset

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`

Twitter Simple	`id, in_reply_to_user_id`	`lang == "en"`

Yelp	`review_id, user_id, business_id,`
	`stars`
		`stars > 3 && useful > 5`
		`useful > 10`

Note that the cost of system call can be determined empirically, while the latency for random I/O and throughput for sequential I/O can be found in disk specifications. Further, the average record size can be actively estimated by profiling incoming data. The number of bytes for prefetching is determined by record size and disk specification as well. It should at least be larger than the average record size so that prefetching can get at least one meaningful record. Note that it is meaningless to retrieve blocks larger than a full disk queue as we would not be able to extract more parallelism from the SSD. Thus, we set several *levels of speculation* between average record size and the full disk queue size, where the gap between each level is exponential. Thus, with continuous locality, FishStore is able to get close performance to sequential scan in just a few steps. Note that FishStore scans are sequential, with SSD parallelism achieved using concurrent scan and ingestion operations. We discuss extensions for parallelizing a single scan operation in Appendix F.

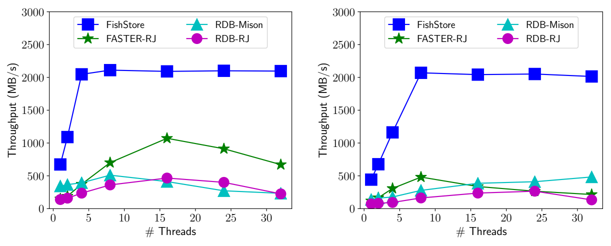
8 EVALUATION

We evaluate FishStore extensively over three real-world datasets, on both ingestion and subset retrieval performance. We first comprehensively evaluate FishStore against existing solutions and our own baselines, both when ingesting into SSD and with in-memory ingest (avoiding the SSD bottleneck). We then evaluate features such as partial parsing and latch-free ingestion. Finally, we evaluate FishStore's subset retrieval performance, including adaptive prefetch scans, read/write workloads, and recurring queries.

8.1 Setup and Workloads

All experiments on FishStore are carried out a Dell PowerEdge R730 machine with 2.3GHz 36-core Intel Xeon Gold 6140 CPUs (we use one socket), running Windows Server 2016. The machine has 512GB of RAM and a 3.2TB FusionIO NVMe SSD that supports around 2GB/sec sequential writes.

Systems. We implemented FishStore in C++ by modifying our open-source FASTER, and used it with unmodified Mison for parsing. In order to understand performance and bottlenecks, we also implemented alternative solutions by combining different storage systems, index types, and parsers:



(a) Github (b) Yelp

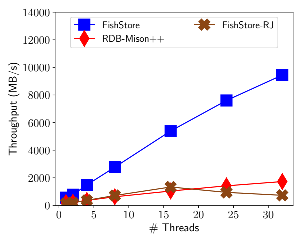
Figure 10: Comparison with Existing Solutions

- `FASTER-RJ` uses RapidJSON [14] to parse the primary key and ingest using this key into the FASTER key-value store.
- `RDB-RJ` and `RDB-Mison` take RapidJSON and Mison respectively to parse only the primary key, and ingest them into RocksDB, which uses an LSM tree designed to handle write-heavy workloads.
- `RDB-Mison++` uses RocksDB as a secondary index to index dynamic PSFs using FishStore's log as primary storage, i.e., we replace FishStore's hash index with RocksDB.
- `FishStore-RJ` replaces Mison with RapidJSON in FishStore, to measure performance without a partial parser.

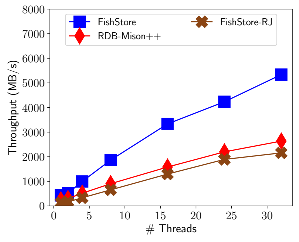
We configured RocksDB (version 5.17.2) to use level-style compaction with 16 compaction threads, and a 1GB write-buffer size (same as the FishStore hash table size). We perform non-synchronous operations on RocksDB, with the WAL disabled. We also tried several prototype systems implementing better LSM trees, including Dostoevsky [28], TRIAD [19], and PebblesDB [44]. We found that even though they achieve slightly better performance compared to vanilla RocksDB on a single ingestion thread, they do not scale well on multiple threads. This led to much lower performance in our evaluations. Therefore, we omit these results and only compare against RocksDB-based systems in this paper.

Datasets. We pick three representative real-world JSON datasets (CSV data experiments are covered in Appendix G):

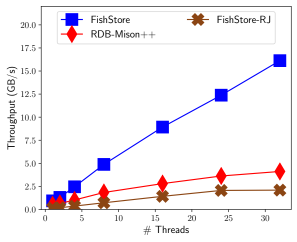
- **Github:** The Github timeline dataset [11] collected in September 2018 includes 18 million records and is 49GB in file size. It features complex JSON structure with a moderate average record size (~ 3KB).



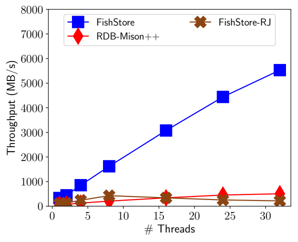
(a) Github



(b) Twitter



(c) Simple Twitter



(d) Yelp

Figure 11: Ingestion Throughput in Main Memory

- **Twitter:** Active 1% sample of tweets crawled through the Twitter API in three days, which is 52GB and contains 9.3 million tweets. Records in this dataset are structurally complex with large average record size (> 5KB) as well.
- **Yelp:** Publicly available Yelp user review data [15]. We duplicate the dataset 8 times to create a 36GB dataset that contains 48 million records. Reviews in this dataset are very small (< 1KB) and have a fixed schema.

Note that each dataset will cause unique behaviors in data ingestion due to the complexity and size of their records. Moreover, people may care about different properties of these dataset in real world applications. In our experiments, we register some field projections and property of interest on each dataset as our default workloads shown by Table 1.

Default Parameters. Unless specified, all evaluations are conducted on 8 threads with a 2GB memory budget, to model a typical lean CPU, fast storage configuration. We preload input datasets into memory for all experiments.

8.2 Compare with Existing Solutions

Before comparing against the baselines mentioned in Sec 8.1, we tried ingesting our dataset in MongoDB [12] and AsterixDB [4], which are open-source storage systems that can ingest schema-free semi-structured data. For the Github dataset, both MongoDB and AsterixDB take more than half an hour to ingest without building any index. This suggests a throughput around 30MB/s which is around 60× lower than the default setting of FishStore, which saturates disk bandwidth. The main reason for the low performance is that these systems spend significant time reorganizing records into their own binary format. Another option is to load the data into formats such as Parquet. We tried loading the Github dataset into Parquet with 8 cores using Apache Spark 2.3.1. This procedure took around 10 minutes; a load throughput of less than 90MB/s, more than 20× slower than FishStore. Furthermore, these results represent offline loading; we expect a lower throughput when creating Parquet batches on-the-fly, as schema inference would be required on every batch.

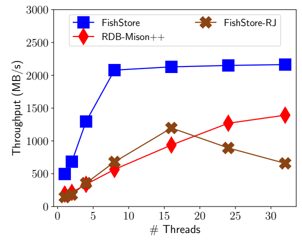
To study how different storage layers and parsers influence performance, we compare the ingestion throughput of FishStore with FASTER-RJ, RDB-RJ, and RDB-Mison. For a fair comparison, we let FishStore also only build the index on one key field projection PSF. We show results on Github and Yelp, and omit Twitter as it is similar to Github. As shown by Fig. 10, FishStore outperforms all other solutions and saturates disk bandwidth with only 8 cores on both datasets, due to the removal of both indexing and parsing bottlenecks. Note that the performance of RDB-Mison drops after 8 threads on Github because the record size of Github is bigger, triggering compaction more aggressively when data arrives fast. In addition, for Yelp dataset, FASTER-RJ's performance drops at 16 threads because FASTER can load data much faster than RocksDB as a key-value store. This forces RapidJSON to parse many short records in a short period of time, which leads to dense memory allocation and performance deterioration.

8.3 Ingestion Performance

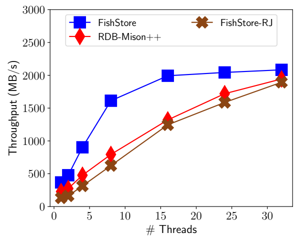
We now evaluate ingestion performance, comparing FishStore against two alternatives, namely RDB-Mison++ and Fishstore-RJ, to show how a full parser or a slower index can significantly impact ingestion performance.

Ingestion Scalability (In-Memory). To understand how FishStore and other solutions perform without the bottleneck of disk, we first conduct experiments that use a null device, which simply discards data to eliminate the disk bandwidth bottleneck. Fig. 11 shows how the solutions scale with an increasing number of worker threads during ingestion.

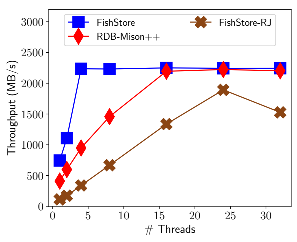
Both RDB-Mison++ and FishStore-RJ do not scale as well as FishStore, but for different reasons. When the records are small, with a reasonable number of registered properties, RDB-Mison++ does scale well as there is much more index update pressure in the system. Fig. 11(d) shows an example of this case, RDB-Mison++ only gets 9.1% of FishStore's throughput with 32 threads. In contrast, FishStore-RJ does not scale for two reasons: (1) compared to Mison, RapidJSON has to parse the whole record; (2) when parsing many records with multiple threads, RapidJSON issues excessive memory allocations for document structure construction, creating a scalability bottleneck. For example, FishStore achieves around 7.7× higher throughput than FishStore-RJ on Simple Twitter workload with 32 cores (shown in Fig. 11(c)),



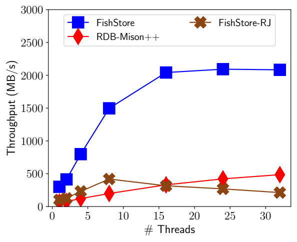
(a) Github



(b) Twitter

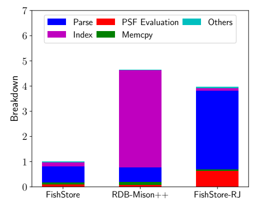


(c) Twitter Simple

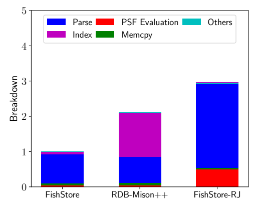


(d) Yelp

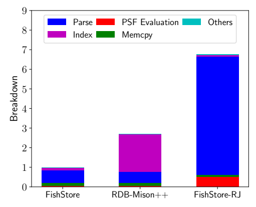
Figure 12: Ingestion Throughput on Disk



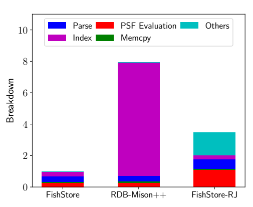
(a) Github



(b) Twitter



(c) Simple Twitter



(d) Yelp

Figure 13: Ingestion in Main Memory: CPU Breakdown

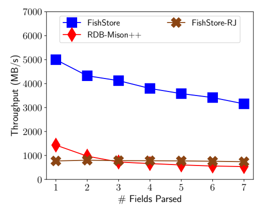


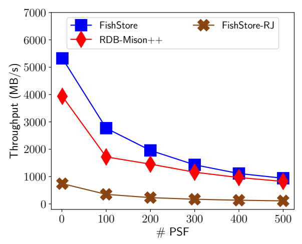
Figure 14: Field Projection PSF Scalability

mainly due to better partial parsing. On the other hand, as we observed in Fig. 11(a), 11(c), and 11(d), the throughput of FishStore-RJ drops after a certain number of threads due to the second reason mentioned above.

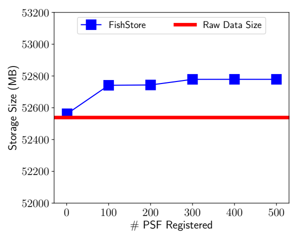
Overall, FishStore is faster than the alternatives and gets more than 16.5GB/s throughput with 32 threads with Twitter Simple, showing its potential on faster storage devices. We also see the gap between FishStore and its alternatives growing larger, which reflects its better scalability.

Ingestion Scalability (Disk). We repeat the experiment above, but enabling our SSD as storage for the ingested data. Fig. 12 shows how different solutions scales against number of threads (i.e., ingestion workers) when ingesting onto disk. As shown in the figure, FishStore hits the SSD bandwidth with less than 8 cores on all the workloads, while both RDB-Mison++ and FishStore-RJ do not scale well for the same reasons discussed earlier.

Ingestion CPU Breakdown. To better understand ingestion performance, we break down the CPU time of FishStore, RDB-Mison++ and FishStore-RJ running all four workloads with 8 threads in memory. In Fig. 13, we normalize the total CPU time of all solutions on each workload by the total CPU time of FishStore on the corresponding workload. This provides us an intuitive comparison of different parts of overheads across different solutions. We see that FishStore combines benefits from a faster parser and index. On Github and Twitter workloads, RDB-Mison++ and FishStore-RJ are slow because of different reasons. RDB-Mison++ incurs significant overhead in updating its index managed by RocksDB, while FishStore-RJ struggles with parsing complex JSON documents. In Simple Twitter workload, the performance of



(a) Ingestion Throughput



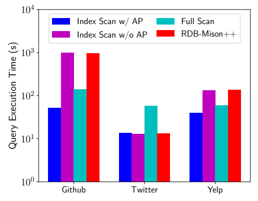
(b) Storage Cost

Figure 15: Predicate based PSF Scalability

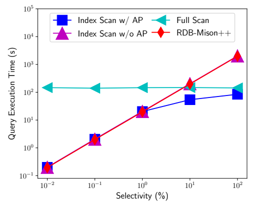
RDB-Mison++ looks decent simply because of a fewer number of registered properties, which leads to low indexing pressure. For Yelp workload, as its records are pretty simple and short, FishStore-RJ does not introduce too much overhead in parsing. However, it spends a non-trivial time in memory allocations for building its internal document structure and evaluating PSFs as it need to scan the document twice to find the location of a parsed out field.

Effect of # PSFs. Three basic factors affect FishStore's throughput: field parsing cost, PSF evaluation cost, and indexing cost. All these factors are influenced by the number of PSFs registered. We perform two sets of experiments to show this effect.

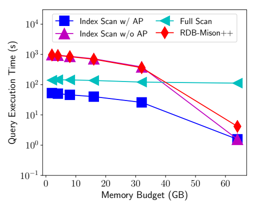
We first register several field projection PSFs, varying the number of them, which increases all three costs, and particularly the parsing cost. As shown in Fig. 14, the ingestion throughput of FishStore and RDB-Mison++ deteriorates as more field are parsed, while FishStore-RJ's performance is not influenced because RapidJSON always parses the entire document. The storage cost of these PSFs (not shown) is similar, following a linear trend up to 1%.



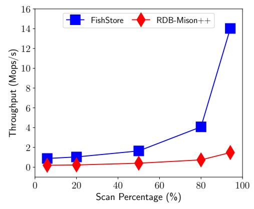
(a) Execution Time Summary



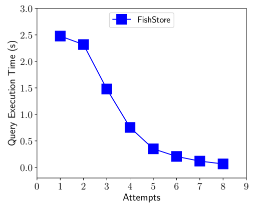
(b) Effect of Selectivity



(c) Effect of Memory Budget



(d) Mixed Workload



(e) Recurring Query

Figure 16: FishStore Subset Retrieval Performance

In the second set of experiments, we aim to show how PSF evaluation and indexing costs alone influence FishStore. In particular, we register a varying number of PSFs over the Twitter dataset: PSF 1-250 are predicate-based PSFs that index disjoint value ranges over the field "user.statuses_count' and PSF 251-500 are similar PSFs but over overlapping ranges. Fig. 15 shows that the ingestion speed drops as the number of PSFs grows, due to higher PSF evaluation and indexing cost. Note that RDB-Mison++ is only 15%-35% slower than FishStore, as each PSF only matches a small fraction of records, leading to much lower index pressure. We also observe that the storage cost is very low at around 0.5%, even with 500 PSFs registered. The storage cost of a predicate-based PSF depends on how many records it matches. As shown in Fig. 15(b), the overlapping ranges of PSFs (PSF 251 to 500) have a slightly higher storage cost than disjoint ranges (PSF 1 to 250) due to higher selectivity.

In practice, our workloads are expected to contain a few field projections and many predicates matching small subsets, which will not incur a high storage cost. For instance, the default Twitter workload only causes around 1.35% storage overhead. As a result, its influence on full scan is negligible, and we omit these results for brevity.

In addition to the results above, we conduct experiments to show the effectiveness of our hash index update algorithm and CSV ingestion support, in Appendix G.

8.4 Subset Retrieval Performance

In this section, we evaluate subset retrieval using FishStore and RDB-Mison++. We also show FishStore's performance over ingest/scan mixed workloads and recurring queries.

Adaptive Prefetching. Fig. 16(a) shows the performance of full scan and index scan with and without adaptive prefetching on different datasets. Specifically, the query for Github dataset is 'type == PushEvent' that touches around 50% of Github records; that for Twitter dataset is 'user.lang = ja && user.follower_count > 3000' which has a selectivity around 1%; query for Yelp data is 'stars > 3 && useful > 5' selecting 2% of all records. We see that RDB-Mison++ is similar to scan without adaptive prefetching. Prefetching helps us to get the best of both worlds: issue random I/Os for selective queries while achieving no worse performance on non-selective ones. This method is particularly effective for non-selective queries such as the Github query, which helps FishStore achieve a 10× speed up compared to raw index scan without adaptive prefetching.

Effect of Selectivity. When the request is non-selective, exploring the hash chain through random I/Os may be slower than a full scan. Fig. 16(b) shows the result of issuing queries with different selectivity on Github data. Full scan is not influenced at all as expected, while index scan without adaptive prefetching and RDB-Mison++ slow down as selectivity grows and finally become slower than doing a full scan. Meanwhile, index scan with adaptive prefetching mediates the cost of random I/Os and tries replacing them with larger and continuous I/Os, and avoids being slower than full scans.

Effects of Memory Budget. With more memory, FishStore can cache more recent data in memory for better scan performance. Fig. 16(c) shows the execution time of 'type == PushEvent' on Github dataset by varying FishStore's memory budget. We see that index scan with and without adaptive prefetching benefit from having more memory. In contrast, a full scan does not benefit much because it also requires field parsing and PSF evaluation over each record, since it was not done during ingestion time. RDB-Mison++ degrades slightly with larger memory due to its more expensive index.

Ingest/Scan Mixed Workload. We emulate an application that performs short scans (e.g., point lookup and hash join) while ingesting data. We generate a workload that combines the ingestion of Github dataset and point lookups on field 'actor.id'. As shown in Fig. 16(d), FishStore achieves higher throughput as the percentage of scans grows, because short scans are cheaper with no parsing or index update. RDB-Mison++ achieves lower throughput, particularly for reads, due to the more complex and secondary index.

Recurring Query. Another interesting application is that user may issue recurring queries against the ingested data. Registering a PSF can accelerate future subset retrievals for the same query on newer data. To evaluate this scenario, we issue a sequence of requests asking for the number of open issues in the past hour, against an active Github data ingestion session. After the second query execution, we register the index on property 'type == "IssuesEvent" &&

`payload.action = "opened"'`. As shown by Fig. 16(e), subset retrieval performance improves over time as FishStore has increasingly more data within the sliding window being indexed, requiring a full scan of lesser data. When the window is completely inside index range, the performance becomes stable and much better than previous attempts.

In Appendix G, we evaluate the generality of adaptive prefetching by studying hash chain locality in real datasets. Subset retrieval in systems such as MongoDB, AsterixDB, and Parquet is also covered in Appendix G..

9 RELATED WORK

Data Formats for Analysis. Today's big data systems are capable of accessing data in many formats, which support complex structures such as arrays and nested objects. These data formats can be categorized into three classes: row-oriented text formats (e.g., CSV, JSON), row-oriented binary formats (e.g., Google Protocol Buffers [13], Apache Thrift [10], Apache Avro [5]), and columnar binary formats (e.g., Apache Parquet [9], Apache ORC [8]). FishStore exposes a generic parser interface, but using FishStore with other formats is an interesting direction for future work.

Parsing for Data Analysis. Many techniques [21, 36, 40, 42] have been developed to accelerate data parsing for analysis. For instance, Mühlbauer et al. [40] exploit SIMD parallelism to accelerate CSV parsing and data deserialization. Mison [36] is a fast JSON parser designed for data analytics applications. It allows applications (e.g., analytical engines) to push down projections and filters of analytical queries into the parser. Sparser [42] is a parsing technique applicable to common formats such as CSV, JSON, and Avro that can filter records even before parsing them, by searching the raw byte stream. These parsing techniques are complementary to FishStore, which can benefit from these innovations.

Query Processing over Raw Data. Our work is closely related to recent work on processing raw data in database systems [16, 17, 20, 26, 32, 40]. NoDB [17, 32] is seminal work that builds structural indices on raw CSV files to locate requested fields. Structural indices differ from FishStore's index in lacking the ability to locate all records that match certain criteria. Amazon Athena [1] and Amazon Redshift Spectrum [3] enable users to query raw data in Amazon S3 using standard SQL. Apache Spark [50] and Apache Drill [6] support access to raw data. Compared to these systems, FishStore takes a further step towards fast query processing: it builds hash indices on demand to quickly locate records in raw data that meet certain conditions.

Storage and Indexing. Systems such as Masstree [39, 49] are pure in-memory indices and cannot be used as a storage layer. Cassandra [29], RocksDB [41, 48], and Bw-Tree [35] are key-value stores that can handle data larger than memory, but expose an ingestion bottleneck. DocumentDB [47] parses raw data to create a secondary range index over every (nested) attribute and prefix during ingestion. Recently, plenty of improvements [19, 23, 27, 28, 43–45, 51] have been made to LSM-tree-based key-value stores, to reduce write amplification and improve scan performance. However, a tree-based index still suffers from issues for our target workloads, as evaluated in this paper and discussed in Appendix B. Systems such as AsterixDB [4] and MongoDB [12] can index and store data, but face bottlenecks due to data re-organization. Smooth Scan [22] enables adaptive access path selection and online reoptimization. Adaptive prefetching differs from Smooth Scan as it operates on a log collocating index entries with data records, which provides a simpler cost model and a unified scan pattern. Crescando [30] adopts a scan-only architecture using shared scans to achieve predictable query performance on unpredictable workloads. The tradeoff between scan and index has been examined previously [25, 34]; we provide an adaptive solution for hash chained records guided by a cost model. Further, existing research [46, 52] performs prefetching using multi-threading and SIMD, which applies between CPU cache and main memory but may not work well on disks.

Streaming and Analytics. Streaming ETL over raw data is supported by systems such as Apache Spark [50], Apache Flink [7], Google Cloud Dataflow [31], and Amazon Kinesis [2]. DataCell [37, 38] and Spark Streaming [18] support streaming and ad-hoc queries over relational data. FishStore focuses on fast ingestion of flexible-schema data while indexing subsets of data on demand, and can serve as a storage layer for streaming pipelines.

10 CONCLUSION

Huge volumes of structured and unstructured data are being ingested into the cloud from a variety of data sources. Traditionally, data is either ingested into storage in raw form, indexed ad-hoc using range indices, or cooked into columnar formats. We find that none of these solutions can ingest data at high throughput with low CPU cost, due to the bottlenecks of parsing and indexing. We build on recent advances in parsing and indexing techniques to propose FishStore, a concurrent storage layer for data with flexible schema, based on the notion of hash indexing dynamically registered *predicated subset functions*. Our detailed evaluation on real datasets and query workloads shows that FishStore can handle a wide range of applications and can ingest, index, and retrieve data at an order of magnitude lower cost than state-of-the-art alternatives used widely today.

REFERENCES

[1] 2018. Amazon Athena. https://aws.amazon.com/athena/. (2018).
[2] 2018. Amazon Kinesis. https://aws.amazon.com/kinesis/. (2018).
[3] 2018. Amazon Redshift. https://aws.amazon.com/redshift/. (2018).
[4] 2018. Apache AsterixDB. https://asterixdb.apache.org/. (2018).
[5] 2018. Apache Avro. https://avro.apache.org/. (2018).
[6] 2018. Apache Drill. https://drill.apache.org/. (2018).
[7] 2018. Apache Flink. https://flink.apache.org/. (2018).
[8] 2018. Apache ORC. https://orc.apache.org/. (2018).
[9] 2018. Apache Parquet. https://parquet.apache.org/. (2018).
[10] 2018. Apache Thrift. https://thrift.apache.org/. (2018).
[11] 2018. Github Archive. https://www.gharchive.org/. (2018).
[12] 2018. MongoDB. https://www.mongodb.com/. (2018).
[13] 2018. Protocol Buffers. https://github.com/protocolbuffers. (2018).
[14] 2018. RapidJSON. http://rapidjson.org/. (2018).
[15] 2018. Yelp Dataset. https://www.yelp.com/dataset/challenge. (2018).
[16] Azza Abouzied, Daniel J. Abadi, and Avi Silberschatz. 2013. Invisible loading: access-driven data transfer from raw files into database systems. In *EDBT*.
[17] Ioannis Alagiannis, Renata Borovica, Miguel Branco, Stratos Idreos, and Anastasia Ailamaki. 2012. NoDB: efficient query execution on raw data files. In *SIGMOD*.
[18] Michael Armbrust, Tathagata Das, Joseph Torres, Burak Yavuz, Shixiong Zhu, Reynold Xin, Ali Ghodsi, Ion Stoica, and Matei Zaharia. 2018. Structured Streaming: A Declarative API for Real-Time Applications in Apache Spark. In *SIGMOD*. 601–613.
[19] Oana Balmau, Diego Didona, Rachid Guerraoui, Willy Zwaenepoel, Huapeng Yuan, Aashray Arora, Karan Gupta, and Pavan Konka. 2017. TRIAD: Creating Synergies Between Memory, Disk and Log in Log Structured Key-Value Stores. In *USENIX ATC*. 363–375.
[20] Spyros Blanas, Kesheng Wu, Surendra Byna, Bin Dong, and Arie Shoshani. 2014. Parallel data analysis directly on scientific file formats. In *SIGMOD*. 385–396.
[21] Daniele Bonetta and Matthias Brantner. 2017. FAD.Js: Fast JSON Data Access Using JIT-based Speculative Optimizations. *PVLDB* 10, 12 (Aug. 2017), 1778–1789.
[22] Renata Borovica-Gajic, Stratos Idreos, Anastasia Ailamaki, Marcin Zukowski, and Campbell Fraser. 2018. Smooth Scan: robust access path selection without cardinality estimation. *VLDB J.* 27, 4 (2018), 521–545.
[23] Edward Bortnikov, Anastasia Braginsky, Eshcar Hillel, Idit Keidar, and Gali Sheffi. 2018. Accordion: Better Memory Organization for LSM Key-Value Stores. *PVLDB* 11, 12 (2018), 1863–1875.
[24] Badrish Chandramouli, Guna Prasaad, Donald Kossmann, Justin J. Levandoski, James Hunter, and Mike Barnett. 2018. FASTER: A Concurrent Key-Value Store with In-Place Updates. In *SIGMOD*. 275–290.
[25] Craig Chasseur and Jignesh M. Patel. 2013. Design and Evaluation of Storage Organizations for Read-Optimized Main Memory Databases. *PVLDB* 6, 13 (2013), 1474–1485.
[26] Yu Cheng and Florin Rusu. 2014. Parallel in-situ data processing with speculative loading. In *SIGMOD*.
[27] Niv Dayan, Manos Athanassoulis, and Stratos Idreos. 2017. Monkey: Optimal Navigable Key-Value Store. In *SIGMOD*. 79–94.
[28] Niv Dayan and Stratos Idreos. 2018. Dostoevsky: Better Space-Time Trade-Offs for LSM-Tree Based Key-Value Stores via Adaptive Removal of Superfluous Merging. In *SIGMOD*. 505–520.
[29] Apache Software Foundation. 2018. Apache Cassandra. http://cassandra.apache.org/. (2018).
[30] Georgios Giannikis, Philipp Unterbrunner, Jeremy Meyer, Gustavo Alonso, Dietmar Fauser, and Donald Kossmann. 2010. Crescando. In *SIGMOD*. 1227–1230.
[31] Google. 2018. Google Cloud Dataflow. https://cloud.google.com/dataflow/. (2018).
[32] Stratos Idreos, Ioannis Alagiannis, Ryan Johnson, and Anastasia Ailamaki. 2011. Here are my Data Files. Here are my Queries. Where are my Results?. In *CIDR*.
[33] Martin L. Kersten, Stratos Idreos, Stefan Manegold, and Erietta Liarou. 2011. The Researcher's Guide to the Data Deluge: Querying a Scientific Database in Just a Few Seconds. *PVLDB* 4, 12 (2011), 1474–1477.
[34] Michael S. Kester, Manos Athanassoulis, and Stratos Idreos. 2017. Access Path Selection in Main-Memory Optimized Data Systems: Should I Scan or Should I Probe?. In *SIGMOD*. ACM, New York, NY, USA, 715–730.
[35] Justin J. Levandoski, David B. Lomet, and Sudipta Sengupta. 2013. The Bw-Tree: A B-tree for New Hardware Platforms. In *ICDE*. IEEE Computer Society, Washington, DC, USA, 302–313.
[36] Yinan Li, Nikos R. Katsipoulakis, Badrish Chandramouli, Jonathan Goldstein, and Donald Kossmann. 2017. Mison: A Fast JSON Parser for Data Analytics. *PVLDB* 10, 10 (2017), 1118–1129.
[37] Erietta Liarou, Romulo Goncalves, and Stratos Idreos. 2009. Exploiting the power of relational databases for efficient stream processing. In *EDBT*. 323–334.
[38] Erietta Liarou, Stratos Idreos, Stefan Manegold, and Martin L. Kersten. 2013. Enhanced stream processing in a DBMS kernel. In *EDBT*. 501–512.
[39] Yandong Mao, Eddie Kohler, and Robert Tappan Morris. 2012. Cache Craftiness for Fast Multicore Key-value Storage. In *EuroSys*. ACM, New York, NY, USA, 183–196.
[40] Tobias Mühlbauer, Wolf Rödiger, Robert Seilbeck, Angelika Reiser, Alfons Kemper, and Thomas Neumann. 2013. Instant Loading for Main Memory Databases. *PVLDB* 6, 14 (2013), 1702–1713.
[41] Patrick O'Neil, Edward Cheng, Dieter Gawlick, and Elizabeth O'Neil. 1996. The Log-structured Merge-tree (LSM-tree). *Acta Inf.* 33, 4 (June 1996), 351–385.
[42] Shoumik Palkar, Firas Abuzaid, Peter Bailis, and Matei Zaharia. 2018. Filter Before You Parse: Faster Analytics on Raw Data with Sparser. *PVLDB* 11, 11 (2018), 1576–1589.
[43] Markus Pilman, Kevin Bocksrocker, Lucas Braun, Renato Marroquin, and Donald Kossmann. 2017. Fast Scans on Key-Value Stores. *PVLDB* 10, 11 (2017), 1526–1537.
[44] Pandian Raju, Rohan Kadekodi, Vijay Chidambaram, and Ittai Abraham. 2017. PebblesDB: Building Key-Value Stores using Fragmented Log-Structured Merge Trees. In *SOSP*. 497–514.
[45] Kai Ren, Qing Zheng, Joy Arulraj, and Garth Gibson. 2017. SlimDB: A Space-Efficient Key-Value Storage Engine For Semi-Sorted Data. *PVLDB* 10, 13 (2017), 2037–2048.
[46] Kenneth A. Ross. 2007. Efficient Hash Probes on Modern Processors. In *ICDE*. 1297–1301.
[47] Dharma Shukla et al. 2015. Schema-Agnostic Indexing with Azure DocumentDB. *PVLDB* 8, 12 (2015), 1668–1679.
[48] Facebook Open Source. 2018. RocksDB. http://rocksdb.org/. (2018).
[49] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *SOSP*. ACM, New York, NY, USA, 18–32.
[50] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*.
[51] Huanchen Zhang, Hyeontaek Lim, Viktor Leis, David G. Andersen, Michael Kaminsky, Kimberly Keeton, and Andrew Pavlo. 2018. SuRF: Practical Range Query Filtering with Fast Succinct Tries. In *SIGMOD*. 323–336.

[52] Jingren Zhou, John Cieslewicz, Kenneth A. Ross, and Mihir Shah. 2005. Improving Database Performance on Simultaneous Multithreading Processors. In *VLDB*. 49–60.

A INDEXING HISTORICAL DATA AND SMALL RECORDS

The subset hash index built by FishStore is collocated with the physical record layout, so as to avoid the indirection between index entries and the actual data. This design is particularly suitable for an ingestion system because it only builds newly registered PSF indices on data ingested later. Interestingly, our design does not preclude building secondary indices over historical data. Thanks to the universal address space constructed on the log, any index structure can be built over data ingested by FishStore, similar to the design of `RDB-Mison++` described in Sec. 8.1. Specifically, if we also choose our subset hash index as the secondary index, we simply need to replace the payload region in the record layout with a pointer to the raw data on FishStore's record log. This design is also applicable in case FishStore is used for ingesting small binary records with many matching PSFs per record, where storing index pointers in the record header may incur overhead.

B HASH- VS. TREE-BASED INDICES

In this section, we discuss our design choice of using a hash index instead of a tree index (such as B-Tree or LSM-Tree) adopted by most databases. The biggest advantage a tree-structured index provides is the ability to answer arbitrary range queries. However, after careful discussions with application developers and extensive evaluations, we chose a hash-based solution due to the following reasons:

(1) A tree-structured index has to store keys in index entry and internal nodes for later navigation. As the amount of indexed data and properties grows, the index size also grows rapidly, making it impossible to be completely in-memory. After spilling onto disk, the performance of insertion and queries over the index drops as well. In contrast, as the hash table adopted by FishStore does not store keys in its hash buckets, its size is not related to data size or number of properties, making it concise in most cases.

(2) The index entries in a tree-structured index are sorted and clustered, which forces it to add an indirection between the index entry and data records. Recall that a record can be referred to by multiple index entries as it may has more than one properties. However, since these index entries represent different properties, they will not be continuous at the leaf level. Duplicating the record next to its index entries is too expensive. Hence, data has to be stored separately and all index entries have to carry a pointer to the data. As index entries may themselves have spilled to disk, retrieving records through the index may cause an additional random I/O.

(3) With the parsing overhead eliminated, the bottleneck shifts elsewhere. Although tree-structured indices provide more flexibility for queries, their performance for inserting a new entry is much slower than a hash table. According to our evaluation, the throughput of a hash table insertion is significantly higher than that of a tree insertion.

(4) FishStore is designed to be highly concurrent and write heavy. Common tree indices for such workloads cause write amplification during log compaction, which introduces overhead and wastes significant disk bandwidth [44]. As a result, the ingestion throughput is limited.

(5) Combined with post-processing, the subset hash index is also able to answer some range queries. If the range is known, it can be registered as a PSF to aid future retrievals. Further, users can build indices over a bucketing function. Then, a range query can be answered by retrieving all records in the covering buckets, with post-filtering.

To evaluate the ingestion performance of FishStore when using a tree-structured index, we built a version of FishStore using RocksDB as its underlying index. Our detailed evaluation in Sec 8 verifies that its performance is significantly worse than the original FishStore.

C CONCURRENT LOG MAINTENANCE

Since FishStore supports data larger than memory, FishStore's log spans across main memory and disk. To facilitate data access on the log, FishStore creates a single logical address space over the log. Specifically, we assign the log a continuous address space starting from 0. Record allocations happen at the tail of the hybrid log, which is mapped to a circular buffer in main memory. When a page is no longer being updated by any ingestion thread (this is identified using epoch protection), it is immutable and flushed to storage.

The in-memory buffer is sized based on available memory; a larger buffer allows more queries to find records in main memory, and benefits from temporal locality of record access by queries. FishStore maintains a *head offset* indicating the boundary between disk and main memory, and a *tail offset* pointing to the next free address. As the tail moves forward on the circular buffer, epoch protection is used to prevent it from overwriting the head, and guarantees the safety of memory access in the presence of concurrent readers (scan queries). To elaborate, an update of the head offset is not considered *safe* (in terms of reclaiming the corresponding region of the circular buffer), until all threads have acknowledged it by refreshing their epochs. Reader threads periodically refresh their epochs during a scan so that writers are not blocked from advancing the tail.

Algorithm 1: Hash Chain Update Algorithm

```
Input: Hash signature $h$ of a property
       key pointer to swap in cur_kpt
Result: Swap cur_kpt into the hash chain of $h$
1  cur_addr ← address of cur_kpt
2  entry ← find_hash_entry($h$)
3  new_entry ← update entry.addr to cur_addr
4  while !(success ← entry.CAS(entry, new_entry)) do
5      if entry.addr < cur_addr then
6          cur_kpt.prev_addr ← entry.addr
7      else
8          break
9      end
10 end
11 if !success then
12     kpt ← get_kpt(entry.addr)
13     repeat
14         while kpt.prev_addr > cur_addr do
15             kpt ← get_kpt(kpt.prev_addr)
16         end
17         cur_kpt.prev_addr ← kpt.prev_addr
18         new_kpt ← update kpt.prev_addr to cur_addr
19     until kpt.CAS(kpt, new_kpt);
20 end
```

D HASH CHAIN UPDATE ALGORITHM

Algorithm 1 summarizes the hash chain update algorithm. As we can see, no successful CAS creates a forward link in the hash chain. Furthermore, the algorithm will always terminate since we can only fall from the easy case into the hard case, and the swapping point will always head to a higher address. A nice property of this algorithm is that it breaks the contention and moves it somewhere else as soon as we realize that the invariant (no forward hash link) is going to break. As a result, even on very high contention, the algorithm is able to break a single contending point to different places on the hash chain. In practice, this method works extremely well and survives contending millions of records on a single chain with 32 threads. Moreover, since we can use the technique independently on all the key pointers of a record, there is no need to do any reallocation upon CAS failures. Note that a newly successful insertion on the hash chain will only happen after $\mathcal{P}$ due to the hash chain invariants. As a result, in all three cases, CAS can only happen between A and $\mathcal{P}$. Combining with the fact that Algorithm 1 will always make progress, we can conclude that it is wait-free. Suppose there are k_1 key pointers between A and $\mathcal{P}$ in case 1, and there are k_2 key pointers if case 1 fails and falls into case 2 and 3. In the worst case, Algorithm 1 will terminate in $O(k_1 + k_2)$.

E FISHSTORE CHECKPOINTING

FishStore automatically persists data from the in-memory circular buffer onto disk. Thus, we can provide a periodic line of persistence to each user, allowing them to continue ingestion after the system crashes. Specifically, we periodically report a batch number and an offset to each ingestion worker, such that all records up to the offset in the batch have been persisted onto disk.

However, after a crash, FishStore will lose its in-memory hash table. As a results, FishStore needs to scan through the log and rebuild its hash table during recovery, which can take time. To address this issue, FishStore allows users to take a periodic "fuzzy checkpoint" of the hash table, similar to the technique used in FASTER [24]. To obtain a fuzzy checkpoint, FishStore simply writes all hash index pages using asynchronous I/Os. Since the hash bucket entries are updated only with atomic CAS instructions, the hash index is always physically consistent. With the help of index checkpointing, recovery time can be reduced by replaying a smaller suffix of the log. Currently, FishStore uses a single thread for recovery. It is possible to parallelize recovery on multiple threads, or mask recovery time using active replicas; these extensions are left as future work.

F PARALLEL SCANS

The adaptive scan described in Sec. 7.2 does not exploit SSD parallelism as it follows a chain of hash pointers. This is suitable for our target applications, where SSD parallelism is achieved using concurrent scan and ingestion operations. We may extend the system to parallelize scans in several ways. First, when adaptive scan chooses to scan pages, we may issue multiple I/Os and process pages in parallel. For hash chain traversals, we may introduce *finger pointers*, additional pointers at the root entry or intermediate records in the hash chain that point to different segments of the hash chain. This incurs additional overhead, but may be used to traverse different sections of the hash chain in parallel during the scan. Alternatively, we may introduce multiple hash entries for the same PSF to traverse in parallel, for example, by concatenating PSF values with a fixed number of constants.

G MORE EVALUATION

Subset Retrieval in Other Systems. As is well-known, Parquet is highly optimized for analytical queries on offline data involving a few columns. In contrast, FishStore is designed for a different workload of fast ingestion of huge data volumes with dynamic subset retrieval support. The raw data (or its more refined predicated subsets) may eventually migrate from FishStore to formats such as Parquet for offline analytics.

For instance, we found that counting the number of push events in the Github dataset takes only 1.4s with Parquet used via Spark. On the other hand, apart from the slow ingestion speed (Sec. 8.2), we found that retrieving records (either all or a non-trivial subset of fields) is slower, primarily due to the higher record reconstruction cost. For example, it takes Parquet more than 286s to retrieve all Github records matching filter `'type == PushEvent'`. Finally, we note that Parquet is also not suitable for highly selective queries such

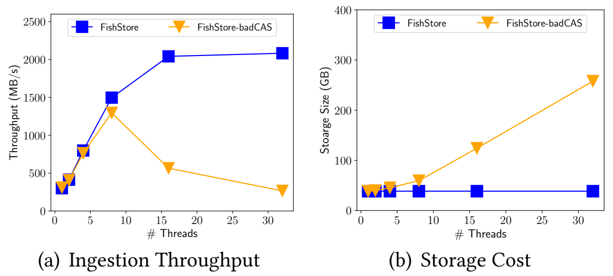


Figure 17: Effect of CAS Technique

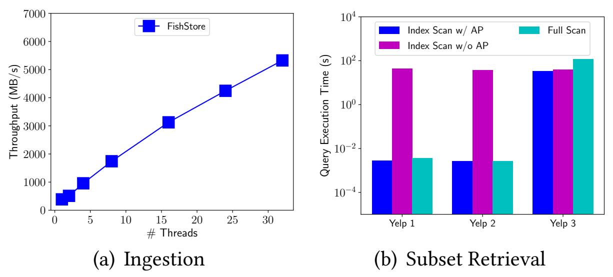


Figure 18: FishStore CSV Support

as point lookups, which hash indices are optimized for. For instance, FishStore takes 30ms to scan 60 records, whereas Parquet takes more than 1s for the same workload. Finally, since we configured MongoDB and AsterixDB for ingestion, we do not build an index. Therefore, subset retrieval falls back to a full scan and takes a long time.

CAS Technique. FishStore's CAS technique of Alg. 1 avoids space and write amplification. Fig. 17 shows its effectiveness by comparing FishStore against another version of FishStore that uses the CAS technique of unmodified FASTER on the Yelp workload. As Fig. 17(a) shows, the original version suffers from significant throughput drop after 8 threads. This is because its index update technique reallocates the record on the log if any of the CAS operations fail during insert. As the number of threads grows, the higher contention results in failed CAS operations and heavy write amplification (as shown by Fig. 17(b)), hurting throughput.

CSV Ingestion. Recall that FishStore exposes a generic parser interface that supports batched parsing of the specified fields of interest. In our examples and evaluations, we focused on JSON as our major supported data type. To verify FishStore's ability to support other data types, we implemented a CSV parser and plugged it into FishStore. We evaluate FishStore against a CSV copy of the Yelp data used in Sec. 8. Fig. 18(a) shows that data ingestion performance in memory grows linearly with the number of threads. Specifically, as with JSON, we hit the 2GB/sec disk bandwidth limit with 8 threads and achieve throughput of up to 5.4GB/sec with 32 cores. Finally, as shown in Fig. 18(b), the adaptive prefetching has effectiveness similar to the trend of its JSON counterpart, which was covered in Sec. 8.4.

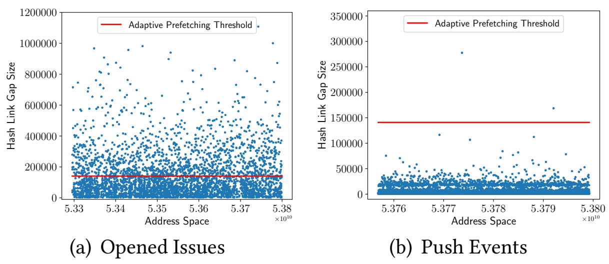


Figure 19: Index Scanning Case Study

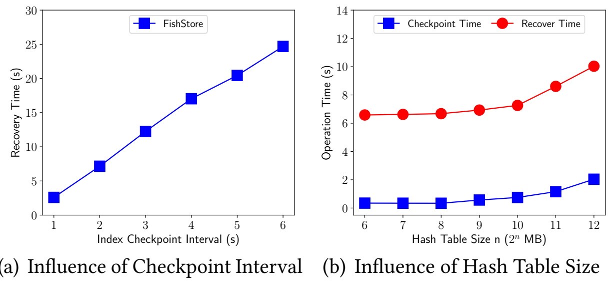


Figure 20: FishStore Checkpoint Performance

Adaptive Prefetching Case Study. To demonstrate the effectiveness of adaptive prefetching and the generality of temporal locality in real world data sets, we plot how records are scattered along the log for two PSF properties on Github Dataset. Fig. 19(a) shows how the hash link gap distribute along address space for query `type=="IssueEvents" && action=="open"`. As we can see, long and short gaps interleave along the log for this query. Specifically, long gaps can be more than 1MB skipping through thousands of records, short ones can link two continuous records. Hence, adaptive prefetching is effective to switch between random I/O and sequential scan back and forth to optimize scan performance.

Fig. 19(b) shows query `type=="PushEvent"` where records on the hash chain are dense on the log with occasional long jumps. Hence, FishStore will first converge to sequential scan, fall back to random IO going over the long gap, and then converge to sequential scan again.

Checkpoint and Recovery. Fig. 20 shows the performance of index checkpointing and crash recovery, described in Appendix E, and how they are influenced by hash table size and checkpoint interval. As shown in Fig. 20(a), the recovery time grows linearly with the checkpoint interval, since FishStore has to load longer suffixes of the log to rebuild its hash table. When the hash table size grows, both checkpoint and recovery time grow as FishStore need to dump or load the whole hash table to or from disk. As described earlier, recovery times may also be controlled by parallelizing recovery or masking it using active replicas.