FishStore: Fast Ingestion and Indexing of Raw Data

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

[†]Microsoft Research [‡]University of Utah

badrishc@microsoft.com, dongx@cs.utah.edu, yinali@microsoft.com, donaldk@microsoft.com

ABSTRACT

The last decade has witnessed a huge increase in data being ingested into the cloud from a variety of data sources. The ingested data takes various 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. We demonstrate FishStore, our open-source concurrent latch-free storage layer for data with flexible schema. FishStore builds on recent advances in parsing and indexing techniques, and is 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 higher performance (by up to an order of magnitude) implementation than current alternatives.

PVLDB Reference Format:
Badrish Chandramouli, Dong Xie, Yinan Li, and Donald Kossmann. FishStore: Fast Ingestion and Indexing of Raw Data. *PVLDB*, 12(12): 1922-1925, 2019.
DOI: https://doi.org/10.14778/3352063.3352100

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 Apache Thrift [2].

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 [10, 8] 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 [1]. 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 [11].

1.2 New Trends in Parsing and Indexing

Recently, raw parsers such as Mison [11] and simdjson [7] 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 [6] that are used in practice, which incur heavy write amplification, random I/Os, and CPU overheads.

[*]Work started during internship at Microsoft Research.

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352100

Persistent key-value stores such as FASTER [9] 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

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 $f : R \to D$ 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. PSF-based indexing (Section 2) 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.

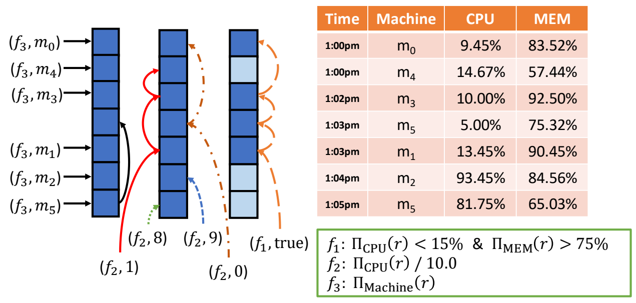
Example (Machine Telemetry) *Consider the application depicted in Figure 1, where machines report telemetry data for ingestion. An analyst wishes to investigate machines with low CPU and high memory utilization. They register a PSF f_1 that indexes records with CPU usage lower than 15% and memory usage 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. The data may be prepared for analysis by ranges of CPU usage via PSF f_3, which creates buckets for different CPU usage ranges.*

1.4 FishStore Components

We overview the FishStore system and its challenges in Section 3. Briefly, it consists of two major components: (1) *Ingestion and 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. (2) *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 uses a novel adaptive scan for high performance.

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

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


(f_3, m_0)
(f_3, m_4)
(f_3, m_3)
(f_3, m_1)
(f_3, m_2)
(f_3, m_5)
$(f_2, 8)$
$(f_2, 9)$
(f_1, true)
$(f_2, 1)$
$(f_2, 0)$
f_1: $\Pi_{\text{CPU}}(r) < 15\%$ & $\Pi_{\text{MEM}}(r) > 75\%$
f_2: $\Pi_{\text{CPU}}(r)$ / 10.0
f_3: $\Pi_{\text{Machine}}(r)$

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%

Figure 1: **Machine Telemetry PSF Example**

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 [12] 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 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 can serve as a storage layer for data ingestion; streaming engines may use FishStore to push down predicates and build shared indices over ingested data.

FishStore is available as open-source software [3], and may be used with open-source parsers such as simdjson using our parser plugin model. Our research paper [12] covers the technical details of FishStore. In this paper, we overview the system and detail our demonstration scenarios.

2. 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 1 (Predicated Subset Function) *For a given data source of records in R, a predicated subset function (PSF) is a function $f : R \to D$ 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 $\Pi_C(r) = \texttt{null}$. 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 (Record Property) *A record $r \in R$ is said to have property (f, v), where f is a PSF mapping R to D and $f(r) = v \in D$.*

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.

Expanding on the machine telemetry example from Section 1, Figure 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. More examples of PSF use are provided in our research paper [12].

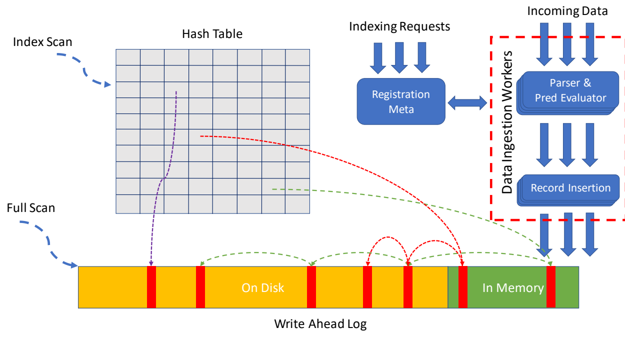


Figure 2: **Overall FishStore Architecture**

3. 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 and overall architecture, and overview the technical challenges, and refer readers to our paper [12] for details.

3.1 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. 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. 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.

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.

3.2 System Architecture

Figure 2 shows the overall architecture of FishStore. It consists of a 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.

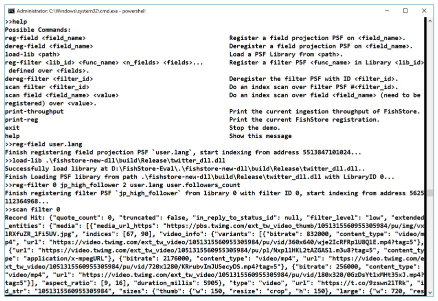
When a record is ingested, FishStore allocates space on the log using an atomic *fetch-and-add* operation on the tail. The tail is present in an in-memory circular buffer of pages, and filled (immutable) pages are flushed to disk. We maintain a unified logical address space across memory and disk, simplifying 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.

All indexing requests are reflected in the registration metadata of FishStore. Through an epoch-based threading model, indexing requests are propagated to 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.

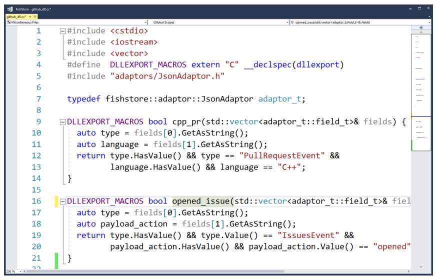
Challenges. Retaining high ingestion performance required a careful design that overcomes several challenges, summarized below and detailed in our research paper [12]:

- 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.
- FishStore needs be able to acquire a safety boundary for on-demand indexing. We utilize the epoch-based thread-



```
Administrator: C:\Windows\system32\cmd.exe - powershell
>>help
Possible Commands:
reg-field <field_name>                                    Register a field projection PSF on <field_name>.
dereg-field <field_name>                                  Deregister a field projection PSF on <field_name>.
load-lib <path>                                           Load a PSF Library from <path>.
reg-filter <lib_id> <func_name> <n_fields> <fields>...    Register a filter PSF <func_name> in Library <lib_id>
 defined over <fields>.
dereg-filter <filter_id>                                  Deregister the filter PSF with ID <filter_id>.
scan filter <filter_id>                                   Do an index scan over Filter PSF #<filter_id>.
scan field <field_name> <value>                           Do an index scan over field <field_name> (need to be
registered) over <value>.
print-throughput                                          Print the current ingestion throughput of FishStore.
print-reg                                                 Print the current FishStore registration.
exit                                                      Stop the demo.
help                                                      Show this message
>>reg-field user.lang
Finish registering field projection PSF `user.lang`, start indexing from address 5513847101024...
>>load-lib .\fishstore-new-dll\build\Release\twitter_dll.dll
Successfully load library at D:\FishStore-Eval\.\fishstore-new-dll\build\Release\twitter_dll.dll..
Finish Loading PSF library from path .\fishstore-new-dll\build\Release\twitter_dll.dll with LibraryID 0...
>>reg-filter 0 jp_high_follower 2 user.lang user.followers_count
Finish registering filter PSF `jp_high_follower` from library 0 with filter ID 0, start indexing from address 5625
112364968...
>>scan filter 0
Record Hit: {"quote_count": 0, "truncated": false, "in_reply_to_status_id": null, "filter_level": "low", "extended
_entities": {"media": [{"media_url_https": "https://pbs.twimg.com/ext_tw_video_thumb/1051315560955305984/pu/img/vx
lRXfuZR_1Fi5UV.jpg", "indices": [67, 90], "video_info": {"variants": [{"bitrate": 832000, "content_type": "video/m
p4", "url": "https://video.twimg.com/ext_tw_video/1051315560955305984/pu/vid/360x640/wje2IcRFRp1U8Q1E.mp4?tag=5"},
 {"url": "https://video.twimg.com/ext_tw_video/1051315560955305984/pu/pl/Nxp1lHKL2tAZGAS1.m3u8?tag=5", "content_ty
pe": "application/x-mpegURL"}, {"bitrate": 2176000, "content_type": "video/mp4", "url": "https://video.twimg.com/e
xt_tw_video/1051315560955305984/pu/vid/720x1280/KRrubvImJUSecyDS.mp4?tag=5"}, {"bitrate": 256000, "content_type":
"video/mp4", "url": "https://video.twimg.com/ext_tw_video/1051315560955305984/pu/vid/180x320/0GzDsYt1xMHt35xJ.mp4?
tag=5"}], "aspect_ratio": [9, 16], "duration_millis": 5905}, "type": "video", "url": "https://t.co/9zswn21TRk", "i
d_str": "1051315560955305984", "sizes": {"thumb": {"w": 150, "resize": "crop", "h": 150}, "large": {"w": 720, "res
```

(a) **Online Ingestion and Retrieval**



FishStore - github_dll.cc*
github_dll.cc*
Miscellaneous Files
(Global Scope)
opened_issue(std::vector<adaptor_t::field_t>& fields)

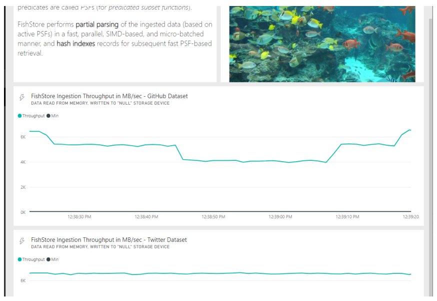
```cpp
#include <cstdio>
#include <iostream>
#include <vector>
#define  DLLEXPORT_MACROS extern "C" __declspec(dllexport)
#include "adaptors/JsonAdaptor.h"

typedef fishstore::adaptor::JsonAdaptor adaptor_t;

DLLEXPORT_MACROS bool cpp_pr(std::vector<adaptor_t::field_t>& fields) {
  auto type = fields[0].GetAsString();
  auto language = fields[1].GetAsString();
  return type.HasValue() && type == "PullRequestEvent" &&
         language.HasValue() && language == "C++";
}

DLLEXPORT_MACROS bool opened_issue(std::vector<adaptor_t::field_t>& fiel
  auto type = fields[0].GetAsString();
  auto payload_action = fields[1].GetAsString();
  return type.HasValue() && type.Value() == "IssuesEvent" &&
         payload_action.HasValue() && payload_action.Value() == "opened"
}
```

(b) **Developing Custom PSFs**



predicates are called PSFs (for *predicated subset functions*).

FishStore performs **partial parsing** of the ingested data (based on active PSFs) in a fast, parallel, SIMD-based, and micro-batched manner, and **hash indexes** records for subsequent fast PSF-based retrieval.

FishStore Ingestion Throughput in MB/sec - GitHub Dataset
DATA READ FROM MEMORY, WRITTEN TO "NULL" STORAGE DEVICE
● Throughput ● Min
6K
4K
2K
0K
12:38:30 PM
12:38:40 PM
12:38:50 PM
12:39:00 PM
12:39:10 PM
12:39:20
FishStore Ingestion Throughput in MB/sec - Twitter Dataset
DATA READ FROM MEMORY, WRITTEN TO "NULL" STORAGE DEVICE
● Throughput ● Min
6K

(c) **Run-time Performance Dashboard**

Figure 3: **Screenshots of Aspects of the System that will be Demonstrated to Visitors**

ing model to help us find safe boundaries within which a specific index is guaranteed to exist.

- Data ingestion should be latch-free so as to achieve high throughput on multiple threads. FishStore adopts a novel lock-free technique to update the index with very low cost even during heavy contention.
- 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.

4. DEMO WALKTHROUGH

We have built an online continuous raw data ingestion application using FishStore, that we will use to demonstrate FishStore. We use two real JSON datasets as input:

- **Github:** The Github timeline dataset [4] 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 ($\sim$ 3KB).
- **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.

In the online demo, a configurable number of threads continuously ingest data from an in-memory cache of the specified dataset (repeating same data when done). The demo can work either with a real local SSD or a "null" device. The latter serves to demonstrate performance without the storage bottleneck. Our console-based interface, depicted in Figure 3a, allows users to perform operations such as:

- Print the current ingestion throughput
- Register a data field for indexing all values
- Dynamically load a specified PSF library (C++)
- Register filter PSFs based on functions in the library
- Perform scans of registered field and filter PSFs
- De-register PSFs

Visitors to the demo can interact with the demo by registering PSFs and viewing the results of scan operations. They may also author new PSFs and load them dynamically, by writing the appropriate C++ code. Figure 3b shows how two queries on the GitHub dataset are authored.

We have also built a rich online performance dashboard (shown in Figure 3c) using Microsoft PowerBI [5], that shows the impact on system performance (e.g., ingestion speed) in real-time, as PSF registration, re-registration, and subset retrieval scans are being performed on the system. The dashboard, along with the demo interface, will serve to demonstrate the power of dynamic PSF-based indexing as well as FishStore's high performance, which will be used as a basis to explain the technical innovations that lead to the performance.

5. CONCLUSION

We demonstrate FishStore, a concurrent storage layer for data with flexible schema, based on the notion of hash indexing dynamically registered *predicated subset functions*. FishStore can handle a wide range of applications and can ingest, index, and retrieve data at up to an order of magnitude lower cost than current alternatives. FishStore is available as open-source software [3], and may be paired with custom or standard data parsers. We welcome contributions from the research community as well.

6. REFERENCES

[1] Apache Parquet. `https://parquet.apache.org/`.
[2] Apache Thrift. `https://thrift.apache.org/`.
[3] FishStore. `https://github.com/microsoft/FishStore`.
[4] GH Archive. `https://www.gharchive.org/`.
[5] Microsoft PowerBI. `https://powerbi.microsoft.com/`.
[6] RocksDB. `http://rocksdb.org/`.
[7] simdjson. `https://github.com/lemire/simdjson`.
[8] I. Alagiannis, R. Borovica, M. Branco, S. Idreos, and A. Ailamaki. NoDB: efficient query execution on raw data files. In *SIGMOD*, 2012.
[9] B. Chandramouli, G. Prasaad, D. Kossmann, J. J. Levandoski, J. Hunter, and M. Barnett. FASTER: A concurrent key-value store with in-place updates. In *SIGMOD*, pages 275–290, 2018.
[10] M. L. Kersten, S. Idreos, S. Manegold, and E. Liarou. The researcher's guide to the data deluge: Querying a scientific database in just a few seconds. *PVLDB*, 4(12):1474–1477, 2011.
[11] Y. Li, N. R. Katsipoulakis, B. Chandramouli, J. Goldstein, and D. Kossmann. Mison: A fast JSON parser for data analytics. *PVLDB*, 10(10):1118–1129, 2017.
[12] D. Xie, B. Chandramouli, Y. Li, and D. Kossmann. FishStore: Faster Ingestion with Subset Hashing. In *SIGMOD*, 2019.