Mach: A Pluggable Metrics Storage Engine for the Age of Observability

Franco Solleza[1], Andrew Crotty[1,2], Suman Karumuri[3], Nesime Tatbul[4,5], Stan Zdonik[1]
[1]Brown University, [2]CMU, [3]Slack Technologies, [4]Intel Labs, [5]MIT
fsolleza@cs.brown.edu,andrewcr@cs.cmu.edu,skarumuri@slack-corp.com,tatbul@csail.mit.edu,sbz@cs.brown.edu

ABSTRACT

Observability is gaining traction as a key capability for understanding the internal behavior of large-scale system deployments. Instrumenting these systems to report quantitative telemetry data called *metrics* enables engineers to monitor and maintain services that operate at an enormous scale so they can respond rapidly to any issues that might arise. To be useful, metrics must be ingested, stored, and queryable in real time, but many existing solutions cannot keep up with the sheer volume of generated data.

This paper describes Mach, a pluggable storage engine we are building specifically to handle high-volume metrics data. Similar to many popular libraries (e.g., Berkeley DB, LevelDB, RocksDB, WiredTiger), Mach provides a simple API to store and retrieve data. Mach's lean, loosely coordinated architecture aggressively leverages the characteristics of metrics data and observability workloads, yielding an order-of-magnitude improvement over existing approaches—especially those marketed as "time series database systems" (TSDBs). In fact, our preliminary results show that Mach can achieve nearly 10× higher write throughput and 3× higher read throughput compared to several widely used alternatives.

1 INTRODUCTION

On the afternoon of May 12, 2020, at 4:45pm PST, the cloud-based business communication platform Slack experienced a total service disruption [22]. For millions of users, this outage lasted for only 48 minutes, but the cascade of events that led to the outage actually began roughly eight hours earlier. The incident prompted an "all hands on deck" response, with engineers from multiple teams poring over petabytes of operational data available from Slack's internal monitoring infrastructure. Through swift action, they were able to fully restore service by 5:33pm PST, but diagnosing and correcting the root cause of the outage under immense time pressure was not an easy task.

This incident is a prime example of the growing importance of observability in large-scale system deployments. Observability enables better understanding of the internal behavior of complex systems in real time by collecting different types of telemetry data, in particular quantitative measurements known as *metrics*. Metrics are used in a variety of ways, including to populate dashboards, generate alerts, and answer ad hoc analytical queries.

Figure 1 shows a subset of a metrics dataset from a server monitoring use case. Each row in the figure represents a sample, with samples from the same source grouped together by color. Samples have three distinct parts: (1) the set of labels or tags identifying the sample's source; (2) a timestamp denoting when the source generated the sample; and (3) one or more numeric values capturing quantitative measurements from the source, such as CPU utilization and temperature.

Labels	Timestamp	Util	Temp
name="CPU", host="host_1", cpu_id="1"	1574260177	0.95	48
name="CPU", host="host_1", cpu_id="1"	1574260216	0.83	32
name="CPU", host="host_1", cpu_id="1"	1574260325	0.93	45
name="CPU", host="host_1", cpu_id="1"	1574260422	0.76	42
name="CPU", host="host_2", v_cpu_id="1"	1574260163	0.03	20
name="CPU", host="host_2", v_cpu_id="1"	1574260173	0.00	19
name="CPU", host="host_2", v_cpu_id="1"	1574260183	0.01	18
name="CPU", host="host_2", v_cpu_id="1"	1574260193	0.05	21
name="CPU", host="lab_1", cpu_id="0", cluster="c_0"	1574260002	1.00	60
name="CPU", host="lab_1", cpu_id="0", cluster="c_0"	1574260012	1.00	60
name="CPU", host="lab_1", cpu_id="0", cluster="c_0"	1574260035	1.00	60

Figure 1: Metrics from a server monitoring use case.

Real-world use cases can produce huge volumes of metrics data from millions or billions of distinct sources. For example, Slack collects metrics from 4 billion unique sources per day at a rate of 12 million samples per second, generating up to 12 TB of compressed data every day [20]. Unfortunately, many off-the-shelf systems cannot scale to support these workloads, leading to a patchwork of brittle, custom-built solutions [20, 21].

To address these shortcomings, we are building a pluggable storage engine called Mach that specifically targets observability use cases. Mach provides a simple API for users to store and retrieve their data, similar to other popular libraries (e.g., Berkeley DB [24], LevelDB [6], RocksDB [9], WiredTiger [14]). However, unlike these storage engines, Mach has a lean, loosely coordinated architecture that is specialized for the unique characteristics of metrics data and observability workloads. These design decisions allow Mach to outperform several widely used alternatives, with nearly 10× higher write throughput and 3× higher read throughput.

2 BACKGROUND

In this section, we discuss the common characteristics of observability workloads, unique aspects of the metrics data model, and shortcomings of existing approaches.

2.1 Workload Characteristics

Metrics span two dimensions: (1) the time dimension, which consists of individual samples from a data source that arrive over time; and (2) the space dimension, which consists of samples from many different data sources. Figure 2 visualizes the example server monitoring dataset in the time and space dimensions. Each dot represents an individual sample (i.e., a row from Figure 1), and each row of dots represents the samples from a single data source.

 12th Annual Conference on Innovative Data Systems Research (CIDR '22). January 9-12, 2022, Chaminade, USA.

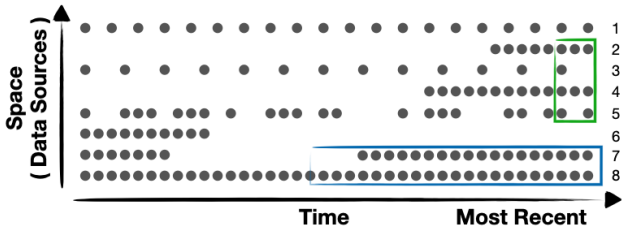


Figure 2: The time and space dimensions of metrics data.

Sources exhibit a variety of write patterns in the time dimension. Some sources produce samples at regular intervals (rows 1, 3), whereas others have bursty or random write patterns (row 5). The space dimension also exhibits churn: sources can become inactive (row 6), start up (rows 2, 4), or stop then restart (row 7).

The inclination for most systems is to consider scalability in the time dimension, since large volumes of data can accumulate over time. However, even if each source emits samples at a relatively low frequency (e.g., once per second), having millions or billions of active sources poses a scalability challenge in the space dimension. Metrics storage engines must therefore consider both of these dimensions in their design.

Read queries are typically biased toward freshness. Analytical queries that look far back in time (the blue box in Figure 2) are much less frequent than queries that look at recent data across many sources (the green box in Figure 2). For example, 95% of queries in Slack's monitoring infrastructure are for metrics data from the past hour, and almost 98% of metrics queries are for metrics data less than 4 hours old.

2.2 Data Model

As mentioned, samples in a metrics dataset consist of three parts: (1) the set of labels or tags identifying the source; (2) a timestamp; and (3) one or more data values. Metrics with a single value are referred to as univariate, whereas metrics comprised of multiple values are called multivariate.

Unfortunately, existing data models fail to neatly capture the unique aspects of metrics data. For example, the relational model cannot easily handle the variable number of labels and values. Storing each one individually would result in many sparsely populated attributes, whereas storing them together as a complex type in a single attribute would preclude common optimizations (e.g., columnar compression). Another alternative is to create a new relation for each data source. However, this approach does not scale beyond a limited number of sources, and certain types of queries become unwieldy (e.g., retrieve all sources with the label `name="CPU"`). Moreover, all of these alternatives require online schema modifications every time a new type of label or value must be added.

Other popular data models incorporate key assumptions that make them similarly ill-suited for metrics data. For instance, nearly all algorithms based on the time series data model assume samples at evenly spaced points in time, but data sources in the observability setting often produce samples at irregular intervals corresponding to real-world events (e.g., an application crashing).

2.3 Existing Approaches

Embedded key-value stores (e.g., Berkeley DB [24], LevelDB [6], RocksDB [9], WiredTiger [14]) are widely used for persisting data in a variety of applications. However, these libraries are designed to be general-purpose, and this flexibility imposes certain limitations (e.g., write amplification, single-threaded writes) that limit their ingestion throughput. Other popular embedded DBMSs (e.g., SQLite [10], DuckDB [27], TileDB [25]) suffer from similar limitations that make them impractical for high-volume data ingestion in an observability setting.

Many storage engines specialized for time series data (e.g., Gorilla [26]/Beringei [2], BTrDB [17], Timon [18], ByteSeries [28]) focus primarily on in-memory processing, whereas Mach is heavily optimized for persistent storage. On the other hand, most full-fledged TSDBs (e.g., InfluxDB [4], ClickHouse [3], TimescaleDB [12], VictoriaMetrics [13]) are geared toward analytical queries rather than ingestion performance. Although TSDBs often provide reasonably fast bulk loading, they are generally not designed for the huge volume of small writes in observability workloads.

Prometheus [8] is the most widely adopted system for metrics data and targets the same use case as Mach. Prometheus optimizes for ingest, as is evident in our preliminary results where it outperforms InfluxDB and RocksDB. However, it does not support multivariate data, incurring unnecessary overhead for many common workloads. As we discuss in Section 3.1, it also suffers from coordination overheads unacceptable for metrics workloads. Heracles [29] redesigns the Prometheus storage engine, gaining much of its performance by operating largely in memory.

Thanos [11], M3 [7], and Monarch [15] are distributed metrics and monitoring systems. They focus primarily on the distributed coordination layer and rely on a lower-level storage engine under the hood. For example, Thanos and M3 are designed to manage many distributed Prometheus instances running in a cloud environment.

3 MACH

We are designing Mach as a pluggable storage engine that specifically targets metrics data. In the following, we begin with a high-level overview of Mach's design and discuss what makes it different from existing approaches. Then, we discuss Mach's API, followed by the corresponding write and read paths.

3.1 System Overview

An overview of Mach's architecture appears in Figure 3. Users interact with Mach through a simple API (Section 3.2) to store and retrieve samples. As stated, Mach is specialized for the observability setting, which requires a storage engine to efficiently handle both massive write volume (i.e., ingestion) and low-latency reads (i.e., querying) in real time.

On the write side (Section 3.3), sources emit samples, routing them through an intermediary (e.g., Kafka [5], Prometheus scrapers) or inserting them directly via a protocol like HTTP. New samples are appended to a buffer that, when full, is compressed and written to persistent storage. Eventually, old or infrequently used segments may be migrated to an OLAP system (e.g., ClickHouse [3]) for analysis or remote storage (e.g., Amazon S3 Glacier [1]) for long-term archival.

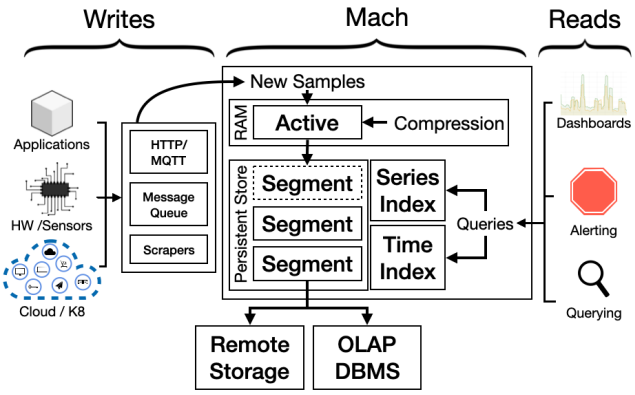


Figure 3: Mach's high-level system architecture.

On the read side (Section 3.4), the system receives a variety of concurrent queries for tasks like populating dashboards, alerting, and ad-hoc analytics. These queries can leverage label and time indexes constructed during ingestion to pinpoint the data sources and time ranges of interest.

Existing storage engines used in observability workloads have many working threads operating in a tightly coordinated fashion. Mach's architecture, on the other hand, takes a fundamentally different approach based on *loose coordination*. More specifically, Mach threads behave independently, keep their own state, and have minimal coordination. We leverage this loosely coordinated design in the following three key ways.

Multiple Independent Active Segments: Most storage engines allow multiple writers for the same data source, coordinating via a mutex and checking to make sure that samples are in order. Consistent with other work [29], we find that mutex acquisition alone comprises roughly 25% of write overhead in Prometheus. Even with multiple writers, these engines eventually write to a single object, and contention can occur at different levels.

In contrast, Mach uses multiple independent writer threads in a loosely coordinated design. Each thread can write samples from multiple sources, but each source is assigned to a single writer thread. To avoid contention, threads maintain all necessary metadata internally and rely on minimal shared state. In this way, Mach behaves like multiple independent storage engines, minimizing coordination overhead in the write path. As shown in our preliminary results (Section 4), this design allows Mach to scale far better than alternatives.

Append-Mostly Fast Path: For flexibility, general-purpose storage engines and TSDBs permit operations like updates and out-of-order inserts. However, for metrics workloads, virtually all samples from a single source are emitted and received in order. The overhead of maintaining data structures like B+trees or LSM-trees is wasted in a workload with millions of small, mostly in-order writes.

Instead, writer threads in Mach have a fast path tailored for append-mostly writes. This fast path is comparable to simply appending to the end of a list and only possible because of Mach's loosely coordinated design. Without needing to worry about concurrent writers, appends incur no synchronization overhead. Furthermore, this list-like behavior allows Mach to execute reads without blocking writes via a low-overhead snapshotting mechanism described in Section 3.4.

Thread-Level Elasticity: Workload burstiness is typically handled in the application layer by scaling up the number of running storage instances as demand increases. This approach adds unnecessary complexity that muddles the actual application logic and often results in poor resource utilization. In fact, systems like Thanos and M3 were built partly to handle this problem.

The core issue is that individual instances of these systems cannot scale in response to bursty workloads, since adding more writer threads simply increases contention. Mach's loose coordination model completely avoids this problem because each thread behaves like an independent storage engine. This approach allows Mach to scale in response to workload changes by adding or reducing writer threads without increasing write contention.

3.2 API

Similar to other popular storage engines, Mach has a simple API to store and retrieve data. Specifically, Mach supports two basic operators, `push` and `get_range`, as well as a `register` function for specifying configuration parameters. The pseudocode in Figure 4 illustrates how an application might use Mach.

`register(config)` registers a new data source with Mach. The configuration describes the samples (e.g., the number of values) and how Mach should handle them (e.g., the precision for the compression scheme described in Section 3.3). `register` returns a 64-bit integer that uniquely identifies the source. Figure 4a shows how an observability application would initialize Mach.

`push(id, ts, values)` pushes a sample from the source identified by `id` to Mach. `ts` is an unsigned 64-bit integer greater than the previous timestamp pushed by this source, and `values` is an n-tuple of floating-point values. Mach guarantees that a sample pushed by a writer thread is immediately available for querying by any calls to `get_range` that occur after the `push` completes. Mach also ensures that samples are durable by periodically calling `fsync` to flush them to persistent storage (see Section 3.3).

The pseudocode for writing samples to Mach with `push` appears in Figure 4b. The application first initializes a writer using the `writer_id` obtained in Figure 4a and then pushes the samples from some input (e.g., a queue or scraper) to the writer.

`get_range(min, max)` returns an iterator over samples from the specified source that fall within the time range between `min` and `max`. `get_range` operates on a snapshot of samples from the source taken as of the time the snapshot was requested. As mentioned, Mach guarantees that all `push` calls completed prior to the snapshot will be visible to `get_range`.

Figure 4c shows pseudocode for querying Mach. The application begins by requesting a snapshot for the data source identified by `id`. Calling `get_range` on the snapshot returns an iterator over samples that fall within the specified time range. The application can either reuse the snapshot object indefinitely for future queries without incurring additional snapshot creation overhead or request a new one to get an updated view of the data.

```
let db = DB::new(writers=10);
let id = db.register(config);
let writer_id = db.writer_id(id);
```

(a) API to initialize Mach with ten writers and register a data source. The user specifies characteristics of samples (e.g., number of values) through the `config` argument.

```
let writer = db.writer(writer_id);
for (id, ts, values) in input {
   writer.push(id, ts, values);
}
```

(b) API to initialize a Mach writer and push samples into Mach from some input (e.g., a queue or scraper). The application may delegate the writer to a separate writer thread.

```
let snap = db.snapshot(id);
for s in snap.get_range(min, max) {
  // process s ...
}
```

(c) API to snapshot samples from a source and query for a range of time. The application may delegate the snapshot to a separate query thread.

Figure 4: Pseudocode demonstrating the usage of Mach's API.

3.3 Write Path

Given the massive volume of metrics data produced in observability use cases, we have designed Mach to prioritize write performance. As shown in Figure 5a, Mach manages loosely coordinated threads in the write path using minimal global state. A concurrent global hash table maps sources to their corresponding samples, and an atomic counter distributes files to each writer thread. To add a new source, Mach updates the global hash table and assigns the source to the appropriate writer thread.

In the figure, the entry for data source *M* is color-coded to show the different components of an entry in the global hash table. The blue and green boxes represent the *active segment* and *active block*, respectively, in the writer thread. These two components comprise the write buffer for new samples.

The yellow and pink boxes are blocks of samples from *M* that the writer thread has already compressed and flushed to secondary storage. Mach stores these blocks in the *block index* so that readers can perform efficient time-based lookups.

The red arrows in Figure 5a show the flow of samples through the append-mostly fast path after calling `push`. When source *M* pushes a sample, the writer thread first looks up the necessary metadata in a thread-local hash table ① and then appends the sample to the active segment ②. For rare cases when the application wants to perform an out-of-order insert, Mach stores the samples in a separate out-of-order buffer that is periodically merged with the rest of the data.

Active Segment: The active segment is an in-memory buffer with a parameterizable fixed size, which defaults to 256 samples. Mach amortizes expensive operations (e.g., compression) in the write path by operating on many samples at a time while still making inserts immediately queryable after a `push` completes. Working with batches of samples also enables many possible optimizations (e.g., SIMD processing).

The active segment stores samples in a columnar fashion, with the current number of samples maintained by an atomic counter. When the active segment fills up, it becomes immutable. The writer thread then takes a snapshot lock to block concurrent readers (see Section 3.4) until it finishes compressing the active segment ③.

Compression: Many TSDBs use some variation of the Gorilla [26] compression scheme, providing little flexibility for use cases not suited to that algorithm. Moreover, they typically compress samples eagerly as soon as they arrive, which adds substantial overhead to the write path.

Rather than compressing individual samples, Mach compresses the entire active segment at once, which amortizes the cost over many samples and enables better compression. Mach also has a modular compression layer that can apply different compression schemes on a case-by-case basis, even at the level of individual columns. For example, CPU temperature readings that vary in a relatively small fixed range might benefit from a more specialized algorithm than measured CPU utilization.

One simple technique that we have found particularly useful in practice involves converting floating-point values to fixed-point integer representations based on a user-specified number of significant digits. The resulting values from this strategy are significantly more intuitive than other lossy compression algorithms that require users to provide an error bound [23, 19].

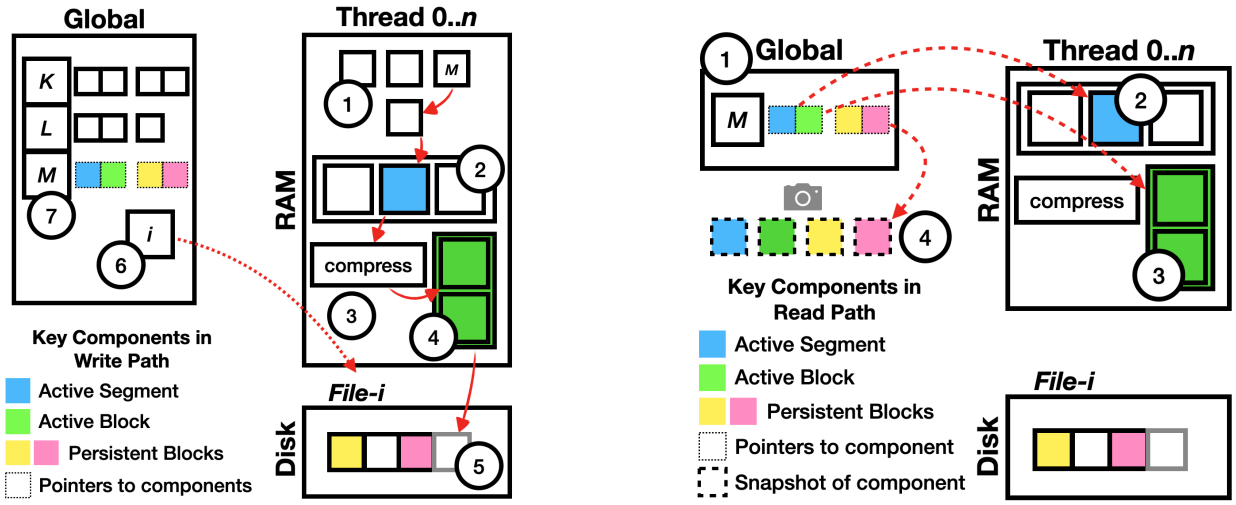
Active Block: The active block ④ is a fixed-size in-memory buffer that corresponds to the OS page size (e.g., 4 KB). As the writer thread adds compressed segments, the active block keeps track of the remaining space using another atomic counter. If not enough space remains in the active block to fit an entire compressed segment, Mach breaks the segment into smaller pieces in an attempt to fill up the block as much as possible and then flushes the block to the corresponding file ⑤.

Block Flushing: As mentioned, each writer thread maintains a private file ⑥ with a name drawn from the global atomic counter shown in Figure 5a, as well as the current offset in the file (i.e., the number of blocks written). After flushing a block to the file, a writer thread adds the block's unique identifier, which consists of the filename and offset, to the block index stored in the global hash table ⑦. Mach periodically calls `fsync` on files to ensure persistence. By default, flushing occurs either after every ten blocks or five seconds, whichever comes first.

When the file reaches a parameterizable size (e.g., 1 GB), Mach performs a final `fsync` and then closes it. The thread then takes the next available filename from the global counter.

3.4 Read Path

Figure 5b shows the read path, which also leverages Mach's loosely coordinated architecture. The read path begins with a snapshot operation executed on the global state ①. Mach looks up the `id` of the data source in the global hash table and then executes a snapshot operation, which returns an iterator over all samples up until the time of snapshot creation. Importantly, this iterator does not block concurrent writer threads.



(a) Write Path (b) Read Path

Figure 5: The read and write paths in Mach. (a) Write Path: When calling push for data source *M*, the write path progressively moves data from an in-memory uncompressed active segment to an in-memory compressed active block, and finally to persistent blocks on disk. The global state (i.e., DB instance in Figure 4a) holds a concurrent hash table that contains pointers to these components. (b) Read Path: When a reader takes a snapshot, it looks up pointers to the active segment, active block, and list of persistent blocks from the global state. It then loads the atomic counters of the active segment and active block, and copies the head of the list of persistent blocks.

Snapshots: As described in Section 3.1, the intuition behind Mach's snapshotting approach is similar to an append-only list. When a writer thread appends a new sample to the end of its active segment, all previous samples in the active segment are considered immutable. To create a snapshot, the reader thread only needs to know the current head and the number of samples in the active segment at the time of the snapshot.

First, the reader thread takes the snapshot lock for the specified source, retrieves pointers to both the active segment ② and active block ③ from the global hash table, and then loads the current values from the atomic counters for each. To complete the snapshotting process, the reader thread then takes a pointer to the head of the block index in the global hash table ④ and releases the snapshot lock.

In many existing systems, a reader will hold locks for the entire duration of the query, which could include expensive disk I/O. In contrast, the critical section of Mach's snapshotting mechanism is short and deterministic. Reads will never block writes in a non-deterministic way; counterintuitively, writes can instead block reads when compressing an active segment.

Identifying Blocks: The reader thread then traverses the active segment, active block, and block index to identify all samples that fall within the time range specified by the `get_range` operation. The block index is currently implemented as a linked list where each node contains several block identifiers. With a linked list, snapshotting is lightweight, since readers do not need to traverse a more complex data structure. Additionally, querying recent samples (i.e., scanning forward from the head of the list) is very efficient, since read queries are biased toward freshness. Although our experiments show that the overhead of traversing the index is small relative to overall query runtime, we plan to investigate alternative data structures in the future.

Reading Blocks: After gathering all of the relevant block identifiers, Mach can then begin scanning these blocks. Since blocks of samples from the same source are written sequentially in time order, a scan involves fast sequential read operations from persistent storage, followed by decompressing the blocks. If blocks are needed by other queries, they will remain available in the OS page cache until being evicted.

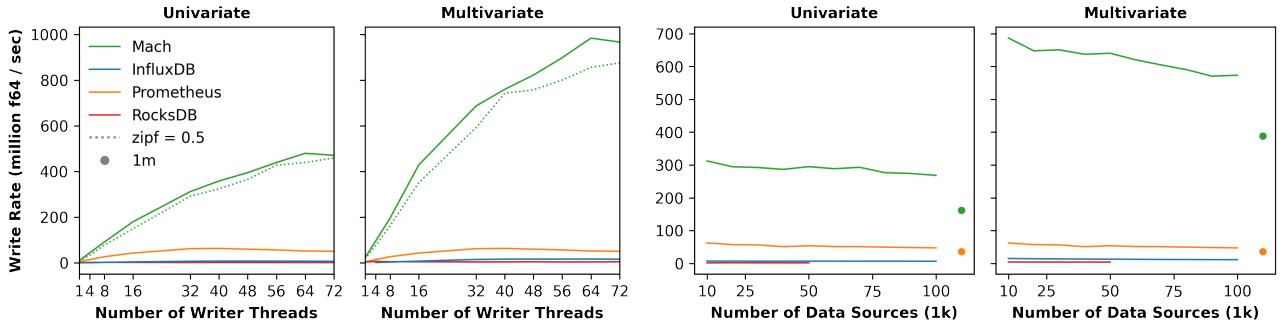
4 PRELIMINARY RESULTS

We implemented a prototype of Mach in Rust. Our preliminary results show: (1) our multiple independent active segments scale better than alternatives and can ingest up to 480M unbatched floating-point values per second on a single node, which is nearly 10× higher write throughput than the closest competitor for comparable univariate workloads; (2) the append-only fast-path can maintain high write throughput when scaling in the space dimension to as many as 1M distinct sources; and (3) Mach provides up to 3× higher read throughput on queries over two different time ranges. In the following, we provide a detailed discussion of these results.

4.1 Setup

Environment: Our experiments were conducted on an Ubuntu 20.04 server with a 2.7 GHz Intel®Xeon®Gold 6150 CPU with 32 cores and hyper-threading, 380 GB RAM, and 3.2 TB Samsung SSDs.

Data: To evaluate the systems, we used server monitoring data collected over three months from the machines managed by the



(a) Varying number of threads

(b) Varying number of data sources

Figure 6: Ingestion Throughput

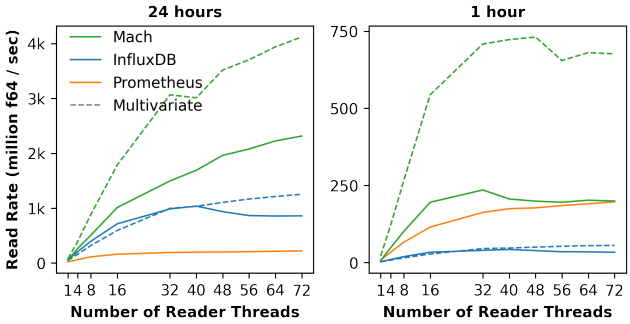


Figure 7: Read Throughput (note the difference in y-axis)

Department of Computer Science at Brown University. The dataset contains 696 data sources, each with 18 metrics and an average of 32K samples, which provides a total of 11.5K univariate and 3.2K multivariate data sources. In our experiments, each data source is randomly chosen (with repetition) from either the univariate or multivariate set.

Comparison Systems: We compare Mach to Prometheus v.2.29, InfluxDB `tsm1`, and RocksDB. For Prometheus and InfluxDB, we extracted and benchmarked their respective storage engines to avoid any overhead. We chose these systems because they persist their data in a deterministic way and are representative of the different types of storage engines that might be used to store metrics data. We excluded systems that rely on distributed execution (e.g., BTrDB [17], Druid [30]) or are primarily designed for in-memory settings (e.g., Gorilla [26]/Beringei [2], Timon [18], Heracles [29]).

The experiments for Prometheus and InfluxDB were written in Go, while all others were written in Rust. All experiments in Section 4.2 flush data to persistent storage, and we disabled write-ahead logging on all systems. For InfluxDB and RocksDB, our ingestion experiments write samples in batches of 1K, as we found that writing single samples yielded extremely low throughput. Note that, in contrast, Mach's `push` operation allows the application to write individual samples without having to worry about batching. Additionally, we set RocksDB's `parallelism=64` and `max_background_jobs=64`.

We report all results as the number of double-precision floating-point values written or read per second (`f64`/sec).

4.2 Write Performance

Scaling with the Number of Writers: In this experiment, we evaluated how well each system scales as the number of writer threads increases. We split 10K sources over the specified number of writer threads. A Zipfian skew of either 0.99 or 0.5 determines the order in which data sources write samples for all systems except Prometheus, as its pull-based approach makes write skew an inappropriate characterization of its ingestion performance. We aborted experiments that ran for longer than 15 minutes.

Figure 6a shows the write throughput of each system. Unsurprisingly, RocksDB exhibited the worst performance, as it was not designed for this workload. InfluxDB outperformed RocksDB by roughly 2–4×, and Prometheus outperformed both InfluxDB (almost 10×) and RocksDB (almost 30×). Of course, Mach significantly outperformed all three of these systems. In particular, we observed that Mach scales much better with the number of writer threads than the comparison points due to its loosely coordinated architecture.

With multivariate data, Mach achieved substantial increases in write throughput because the costs on the write path were amortized over multiple values. We also noticed similar (though much less substantial) improvements for RocksDB and InfluxDB. Since Prometheus only supports univariate data, we include the same results from the previous experiment in the figure as an upper bound of its performance.

Scaling with the Number of Data Sources: In this experiment, we evaluated how well each system scales as the number of data sources increases. We fixed the number of writer threads at 32 and used a Zipfian skew of 0.99 for data source selection.

Figure 6b shows the results. In the univariate case, all systems experienced a gradual decrease in write throughput due to accumulating overheads (e.g., hash table lookups, CPU cache misses). However, Mach always maintained more than a 4× improvement over Prometheus. At 1M distinct data sources, Mach achieved a write throughput of 160M `f64`/sec, more than 2× the peak write throughput achieved by Prometheus. In the multivariate case, Mach's write throughput stayed above 380M `f64`/sec, even for 1M distinct sources. Both InfluxDB and RocksDB timed out.

To model settings with limited compute resources, we reran the same experiment using only a single writer thread. Increasing the number of sources from 10K to 100K resulted in a 35% decrease in Mach's write throughput (from 10M to 6.5M `f64`/sec) and a 46% decrease in write throughput for Prometheus (from 4.9M to 2.7M `f64`/sec). With 10K data sources, Mach's write throughput was 2× higher than Prometheus, increasing to 2.5× at 100K sources.

4.3 Read Performance

In this experiment, we tested read query performance with a varying number of reader threads. We evaluated two different time ranges that reflect common analysis windows in observability settings: the past 1 hour and the past 24 hours. We loaded 10K data sources into InfluxDB, Prometheus, and Mach, and then we executed 100K `get_range` queries per thread. Since not all of the systems batch lookups for multiple data sources, each reader thread only queried a single data source at a time to maintain a consistent baseline. Therefore, the results in Figure 7 are a lower bound on performance for lookups over many data sources.

On the 1-hour workload, Prometheus and Mach each had a read throughput of about 200M `f64`/sec for univariate data. Both outperformed InfluxDB, which had a peak throughput of only 42M `f64`/sec. Since Prometheus and Mach were designed for this workload (i.e., reading more recent data), these results are expected. Similar to ingestion, querying multivariate data increased read throughput. InfluxDB's peak throughput was 56M `f64`/sec, while Mach's was about 700M `f64`/sec. Since these queries access very little of the data, the overall query runtime for all systems was dominated by fixed sources of overhead rather than data scanning. For example, the snapshotting operation consumed the majority of execution time in Mach.

On the 24-hour workload, InfluxDB at 1B `f64`/sec performed close to Mach, except with many concurrent querying threads. The longer time range is advantageous for InfluxDB's read-optimized file format (TSM files). However, InfluxDB and Mach treat multivariate data differently. Specifically, InfluxDB stores multivariate data in a purely columnar format, whereas Mach stores column chunks together similar to PAX [16]. This experiment shows the benefits of Mach's chunked storage approach when requesting several columns, though we expect InfluxDB's strategy to be better when frequently querying only a small subset of the columns.

5 CONCLUSION & FUTURE WORK

This paper presented Mach, a new pluggable storage engine designed to handle high-volume metrics data in an observability setting. We described Mach's loosely coordinated architecture that allows it to fully leverage the unique characteristics of these workloads. Our preliminary results demonstrate that, compared to several widely used alternatives, Mach can achieve nearly 10× higher write throughput and 3× higher read throughput.

In the future, we plan to investigate several natural extensions to Mach. Specifically, we believe that metrics workloads exhibit similar characteristics to other observability data types: logs, events, and traces [20, 21]. For example, integrating log data will require the investigation of fundamental design considerations (e.g., trading off compression speed with size, enabling search over compressed data), whereas supporting traces might necessitate the development of entirely new data models (e.g., graph-based approaches to facilitate root cause analysis). Our ultimate goal is to establish Mach as a unifying storage engine for observability workloads.

Mach's role as a storage engine also opens up opportunities for research in the broader observability ecosystem, and we are investigating several places in the observability stack where Mach could add substantial value. For example, more full-featured systems could build higher-level operators (e.g., filtering, aggregation, interpolation, windowing) on top of Mach's low-level API. More importantly, Mach's excellent performance might even allow us to completely rethink the design of the entire observability stack.

ACKNOWLEDGMENTS

We would like to thank Malte Schwarzkopf, Theo Benson, and Ugur Cetintemel for their helpful feedback. This research was funded in part by NSF grants IIS-1514491 and IIS-1526639.

REFERENCES

[1] Amazon S3 Glacier. https://aws.amazon.com/s3/glacier/.
[2] Beringei. https://github.com/facebookarchive/beringei.
[3] ClickHouse. https://clickhouse.com/.
[4] InfluxDB. https://www.influxdata.com/.
[5] Kafka. https://kafka.apache.org/.
[6] LevelDB. https://github.com/google/leveldb.
[7] M3. https://m3db.io/.
[8] Prometheus. https://prometheus.io/.
[9] RocksDB. http://rocksdb.org/.
[10] SQLite. https://www.sqlite.org/index.html.
[11] Thanos. https://thanos.io/.
[12] TimescaleDB. https://www.timescale.com/.
[13] VictoriaMetrics. https://victoriametrics.com.
[14] WiredTiger. http://source.wiredtiger.com/.
[15] C. Adams, L. Alonso, B. Atkin, J. Banning, S. Bhola, R. Buskens, M. Chen, X. Chen, Y. Chung, Q. Jia, N. Sakharov, G. Talbot, N. Taylor, and A. Tart. Monarch: Google's Planet-Scale In-Memory Time Series Database. *PVLDB*, 13(12):3181–3194, 2020.
[16] A. Ailamaki, D. J. DeWitt, M. D. Hill, and M. Skounakis. Weaving Relations for Cache Performance. In *VLDB*, pages 169–180, 2001.
[17] M. P. Andersen and D. E. Culler. BTrDB: Optimizing Storage System Design for Timeseries Processing. In *FAST*, pages 39–52, 2016.
[18] W. Cao, Y. Gao, F. Li, S. Wang, B. Lin, K. Xu, X. Feng, Y. Wang, Z. Liu, and G. Zhang. Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. In *SIGMOD*, pages 739–753, 2020.
[19] A. Ilkhechi, A. Crotty, A. Galakatos, Y. Mao, G. Fan, X. Shi, and U. Çetintemel. DeepSqueeze: Deep Semantic Compression for Tabular Data. In *SIGMOD*, pages 1733–1746, 2020.
[20] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul. Towards Observability Data Management at Scale. *SIGMOD Rec.*, 49(4):18–23, 2020.
[21] S. Karumuri, F. Solleza, S. Zdonik, and N. Tatbul. Cloud Observability: A MELTing Pot for Petabytes of Heterogenous Time Series. In *CIDR*, 2021.
[22] R. Katkov. All Hands on Deck: What does Slack do when Slack goes down? https://slack.engineering/all-hands-on-deck/, 2020.
[23] P. Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *TVCG*, 20(12):2674–2683, 2014.
[24] M. A. Olson, K. Bostic, and M. I. Seltzer. Berkeley DB. In *USENIX ATC*, pages 183–191, 1999.
[25] S. Papadopoulos, K. Datta, S. Madden, and T. G. Mattson. The TileDB Array Data Storage Manager. *PVLDB*, 10(4):349–360, 2016.
[26] T. Pelkonen, S. Franklin, P. Cavallaro, Q. Huang, J. Meza, J. Teller, and K. Veeraraghavan. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB*, 8(12):1816–1827, 2015.
[27] M. Raasveldt and H. Mühleisen. DuckDB: an Embeddable Analytical Database. In *SIGMOD*, pages 1981–1984, 2019.
[28] X. Shi, Z. Feng, K. Li, Y. Zhou, H. Jin, Y. Jiang, B. He, Z. Ling, and X. Li. ByteSeries: An In-Memory Time Series Database for Large-Scale Monitoring Systems. In *SoCC*, pages 60–73, 2020.
[29] Z. Wang, J. Xue, and Z. Shao. Heracles: An Efficient Storage Model and Data Flushing for Performance Monitoring Timeseries. *PVLDB*, 14(6):1080–1092, 2021.
[30] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli. Druid: A Real-time Analytical Data Store. In *SIGMOD*, pages 157–168, 2014.