Monarch: Google's Planet-Scale In-Memory Time Series Database

Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, Nick Taylor

Google LLC
monarch-paper@google.com

ABSTRACT

Monarch is a globally-distributed in-memory time series database system in Google. Monarch runs as a multi-tenant service and is used mostly to monitor the availability, correctness, performance, load, and other aspects of billion-user-scale applications and systems at Google. Every second, the system ingests terabytes of time series data into memory and serves millions of queries. Monarch has a regionalized architecture for reliability and scalability, and global query and configuration planes that integrate the regions into a unified system. On top of its distributed architecture, Monarch has flexible configuration, an expressive relational data model, and powerful queries. This paper describes the structure of the system and the novel mechanisms that achieve a reliable and flexible unified system on a regionalized distributed architecture. We also share important lessons learned from a decade's experience of developing and running Monarch as a service in Google.

PVLDB Reference Format:
Colin Adams, Luis Alonso, Benjamin Atkin, John Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George Talbot, Adam Tart, Nick Taylor. Monarch: Google's Planet-Scale In-Memory Time Series Database. *PVLDB*, 13(12): 3181-3194, 2020.
DOI: https://doi.org/10.14778/3181-3194

1. INTRODUCTION

Google has massive computer system monitoring requirements. Thousands of teams are running global user facing services (e.g., YouTube, GMail, and Google Maps) or providing hardware and software infrastructure for such services (e.g., Spanner [13], Borg [46], and F1 [40]). These teams need to monitor a continually growing and changing collection of heterogeneous entities (e.g. devices, virtual machines and containers) numbering in the billions and distributed around the globe. Metrics must be collected from each of these entities, stored in time series, and queried to support use cases such as: (1) Detecting and alerting when monitored services are not performing correctly; (2) Displaying dashboards of graphs showing the state and health of the services; and (3) Performing *ad hoc* queries for problem diagnosis and exploration of performance and resource usage.

Borgmon [47] was the initial system at Google responsible for monitoring the behavior of internal applications and infrastructure. Borgmon revolutionized how people think about monitoring and alerting by making collection of metric time series a first-class feature and providing a rich query language for users to customize analysis of monitoring data tailored to their needs. Between 2004 and 2014, Borgmon deployments scaled up significantly due to growth in monitoring traffic, which exposed the following limitations:

- Borgmon's architecture encourages a decentralized operational model where each team sets up and manages their own Borgmon instances. However, this led to non-trivial operational overhead for many teams who do not have the necessary expertise or staffing to run Borgmon reliably. Additionally, users frequently need to examine and correlate monitoring data across application and infrastructure boundaries to troubleshoot issues; this is difficult or impossible to achieve in a world of many isolated Borgmon instances;
- Borgmon's lack of schematization for measurement dimensions and metric values has resulted in semantic ambiguities of queries, limiting the expressiveness of the query language during data analysis;
- Borgmon does not have good support for a distribution (i.e., histogram) value type, which is a powerful data structure that enables sophisticated statistical analysis (e.g., computing the 99th percentile of request latencies across many servers); and
- Borgmon requires users to manually shard the large number of monitored entities of global services across multiple Borgmon instances and set up a query evaluation tree.

With these lessons in mind, Monarch was created as the next-generation large-scale monitoring system at Google. It is designed to scale with continued traffic growth as well as supporting an ever-expanding set of use cases. It provides multi-tenant monitoring as a single unified service for all

Proceedings of the VLDB Endowment, Vol. 13, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3181-3194

teams, minimizing their operational toil. It has a schematized data model facilitating sophisticated queries and comprehensive support of distribution-typed time series. Monarch has been in continuous operation since 2010, collecting, organizing, storing, and querying massive amounts of time series data with rapid growth on a global scale. It presently stores close to a petabyte of compressed time series data in memory, ingests terabytes of data per second, and serves millions of queries per second.

This paper makes the following contributions:

- We present the architecture of Monarch, a multi-tenant, planet-scale in-memory time series database. It is deployed across many geographical regions and supports the monitoring and alerting needs of Google's applications and infrastructure. Monarch ingests and stores monitoring time series data regionally for higher reliability and scalability, is equipped with a global query federation layer to present a global view of geographically distributed data, and provides a global configuration plane for unified control. Monarch stores data in memory to isolate itself from failures at the persistent storage layer for improved availability (it is also backed by log files, for durability, and a long-term repository).

- We describe the novel, type-rich relational data model that underlies Monarch's expressive query language for time series analysis. This allows users to perform a wide variety of operations for rich data analysis while allowing static query analysis and optimizations. The data model supports sophisticated metric value types such as *distribution* for powerful statistical data analysis. To our knowledge, Monarch is the first planet-scale in-memory time series database to support a relational time series data model for monitoring data at the very large scale of petabyte in-memory data storage while serving millions of queries per second.

- We outline Monarch's (1) scalable collection pipeline that provides robust, low-latency data ingestion, automatic load balancing, and collection aggregation for significant efficiency gains; (2) powerful query subsystem that uses an expressive query language, an efficient distributed query execution engine, and a compact indexing subsystem that substantially improves performance and scalability; and (3) global configuration plane that gives users fine-grained control over many aspects of their time series data;

- We present the scale of Monarch and describe the implications of key design decisions on Monarch's scalability. We also share the lessons learned while developing, operating, and evolving Monarch in the hope that they are of interest to readers who are building or operating large-scale monitoring systems.

The rest of the paper is organized as follows. In Section 2 we describe Monarch's system architecture and key components. In Section 3 we explain its data model. We describe Monarch's data collection in Section 4; its query subsystem, including the query language, execution engine, and index in Section 5; and its global configuration system in Section 6. We evaluate Monarch experimentally in Section 7. In Section 8 we compare Monarch to related work. We share lessons learned from developing and operating Monarch in Section 9, and conclude the paper in Section 10.

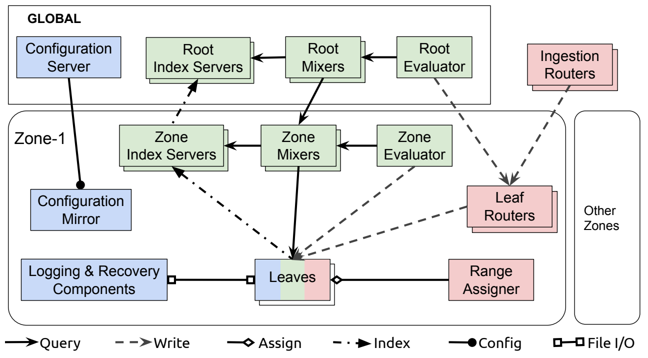


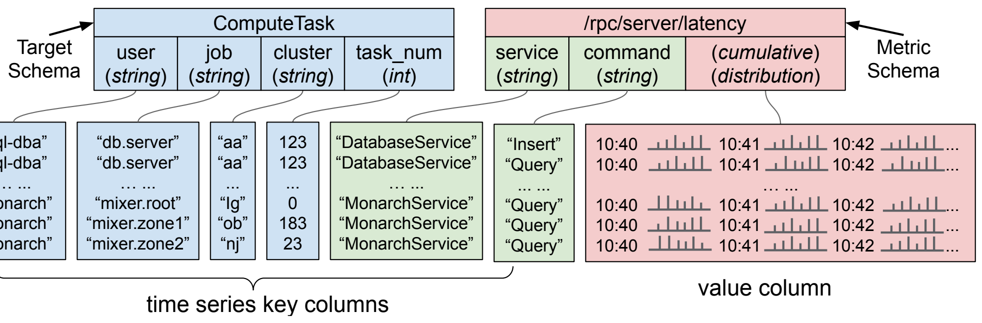
Figure 1: System overview. Components on the left (blue) persist state; those in the middle (green) execute queries; components on the right (red) ingest data. For clarity, some inter-component communications are omitted.

2. SYSTEM OVERVIEW

Monarch's design is determined by its primary usage for monitoring and alerting. First, Monarch readily trades consistency for high availability and partition tolerance [21, 8, 9]. Writing to or reading from a strongly consistent database like Spanner [13] may block for a long time; that is unacceptable for Monarch because it would increase mean-time-to-detection and mean-time-to-mitigation for potential outages. To promptly deliver alerts, Monarch must serve the most recent data in a timely fashion; for that, Monarch drops delayed writes and returns partial data for queries if necessary. In the face of network partitions, Monarch continues to support its users' monitoring and alerting needs, with mechanisms to indicate the underlying data may be incomplete or inconsistent. Second, Monarch must be low dependency on the alerting critical path. To minimize dependencies, Monarch stores monitoring data in memory despite the high cost. Most of Google's storage systems, including Bigtable [10], Colossus ([36], the successor to GFS [20]), Spanner [13], Blobstore [18], and F1 [40], rely on Monarch for reliable monitoring; thus, Monarch cannot use them on the alerting path to avoid a potentially dangerous circular dependency. As a result, non-monitoring applications (e.g., quota services) using Monarch as a global time series database are forced to accept reduced consistency.

The primary organizing principle of Monarch, as shown in Figure 1, is local monitoring in regional *zones* combined with global management and querying. Local monitoring allows Monarch to keep data near where it is collected, reducing transmission costs, latency, and reliability issues, and allowing monitoring within a zone independently of components outside that zone. Global management and querying supports the monitoring of global systems by presenting a unified view of the whole system.

Each Monarch zone is autonomous, and consists of a collection of clusters, i.e., independent failure domains, that are in a strongly network-connected region. Components in a zone are replicated across the clusters for reliability. Monarch stores data in memory and avoids hard dependencies so that each zone can work continuously during transient outages of other zones, global components, and underlying storage systems. Monarch's global components are geographically replicated and interact with zonal components using the closest replica to exploit locality.



Target Schema
ComputeTask
user (string)
job (string)
cluster (string)
task_num (int)
/rpc/server/latency
service (string)
command (string)
(cumulative) (distribution)
Metric Schema
"sql-dba" "sql-dba" … … "monarch" "monarch" "monarch"
"db.server" "db.server" … … "mixer.root" "mixer.zone1" "mixer.zone2"
"aa" "aa" ... "lg" "ob" "nj"
123 123 ... 0 183 23
"DatabaseService" "DatabaseService" … … "MonarchService" "MonarchService" "MonarchService"
"Insert" "Query" … … "Query" "Query" "Query"
10:40 10:41 10:42 ...
time series key columns
value column

Figure 2: Monarch data model example. The top left is a target schema named `ComputeTask` with four key columns. The top right is the schema for a metric named `/rpc/server/latency` with two key columns and one value column. Each row of the bottom table is a time series; its key is the concatenation of all key columns; its value column is named after the last part of its metric name (i.e., `latency`). Each value is an array of timestamped value points (i.e., distributions in this particular example). We omit the *start time* timestamps associated with cumulative time series.

Monarch components can be divided by function into three categories: those holding state, those involved in data ingestion, and those involved in query execution.

The components responsible for holding state are:

- **Leaves** store monitoring data in an in-memory time series store.
- **Recovery logs** store the same monitoring data as the leaves, but on disk. This data ultimately gets rewritten into a long-term time series repository (not discussed due to space constraints).
- A global **configuration server** and its zonal **mirrors** hold configuration data in Spanner [13] databases.

The data ingestion components are:

- **Ingestion routers** that route data to leaf routers in the appropriate Monarch zone, using information in time series keys to determine the routing.
- **Leaf routers** that accept data to be stored in a zone and route it to leaves for storage.
- **Range assigners** that manage the assignment of data to leaves, to balance the load among leaves in a zone.

The components involved in query execution are:

- **Mixers** that partition queries into sub-queries that get routed to and executed by leaves, and merge sub-query results. Queries may be issued at the root level (by root mixers) or at the zone level (by zone mixers). Root-level queries involve both root and zone mixers.
- **Index servers** that index data for each zone and leaf, and guide distributed query execution.
- **Evaluators** that periodically issue standing queries (see Section 5.2) to mixers and write the results back to leaves.

Note that leaves are unique in that they support all three functions. Also, query execution operates at both the zonal and global levels.

3. DATA MODEL

Conceptually, Monarch stores monitoring data as time series in schematized tables. Each table consists of multiple *key columns* that form the time series key, and a *value column* for a history of points of the time series. See Figure 2 for an example. Key columns, also referred to as *fields*, have two sources: *targets* and *metrics*, defined as follows.

3.1 Targets

Monarch uses *targets* to associate each time series with its source entity (or monitored entity), which is, for example, the process or the VM that generates the time series. Each target represents a monitored entity, and conforms to a *target schema* that defines an ordered set of target field names and associated field types. Figure 2 shows a popular target schema named `ComputeTask`; each `ComputeTask` target identifies a running task in a Borg [46] cluster with four fields: `user`, `job`, `cluster`, and `task_num`.

For locality, Monarch stores data close to where the data is generated. Each target schema has one field annotated as *location*; the value of this location field determines the specific Monarch zone to which a time series is routed and stored. For example, the location field of `ComputeTask` is `cluster`; each Borg cluster is mapped to one (usually the closest) Monarch zone. As described in Section 5.3, location fields are also used to optimize query execution.

Within each zone, Monarch stores time series of the same target together in the same leaf because they originate from the same entity and are more likely to be queried together in a join. Monarch also groups targets into disjoint *target ranges* in the form of $[S_{start}, S_{end})$ where S_{start} and S_{end} are the start and end target strings. A *target string* represents a target by concatenating the target schema name and field values in order[1]. For example, in Figure 2, the target string `ComputeTask::sql-dba::db.server::aa::0876` represents the Borg task of a database server. Target ranges are used for lexicographic sharding and load balancing among leaves (see Section 4.2); this allows more efficient aggregation across adjacent targets in queries (see Section 5.3).

[1]The encoding also preserves the lexicographic order of the tuples of target field values, i.e., $S(\langle a_1, a_2, \cdots, a_n\rangle) \leq S(\langle b_1, b_2, \cdots, b_n\rangle) \iff \langle a_1, a_2, \cdots, a_n\rangle \leq \langle b_1, b_2, \cdots, b_n\rangle$, where $S()$ is the string encoding function, and a_i and b_i are the i-th target-field values of targets a and b, respectively.

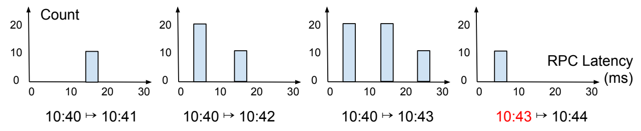
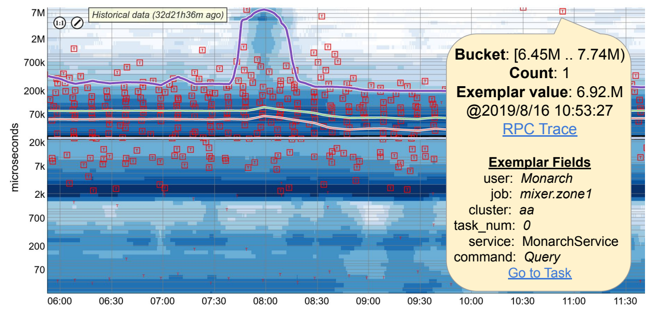


Figure 3: An example cumulative distribution time series for metric `/rpc/server/latency`. There are four points in this time series; each point value is a histogram, whose bucket size is 10ms. Each point has a timestamp and a start timestamp. For example, the 2nd point says that between 10:40 and 10:42, a total of 30 RPCs were served, among which 20 RPCs took 0–10ms and 10 RPCs took 10–20ms. The 4th point has a new start timestamp; between 10:43 and 10:44, 10 RPCs were served and each took 0–10ms.



Bucket: [6.45M .. 7.74M)
Count: 1
Exemplar value: 6.92.M
@2019/8/16 10:53:27

Exemplar Fields
user: *Monarch*
job: *mixer.zone1*
cluster: *aa*
task_num: *0*
service: MonarchService
command: *Query*

Figure 4: A heat map of `/rpc/server/latency`. Clicking an exemplar shows the captured RPC trace.

3.2 Metrics

A metric measures one aspect of a monitored target, such as the number of RPCs a task has served, the memory utilization of a VM, etc. Similar to a target, a metric conforms to a *metric schema*, which defines the time series *value type* and a set of *metric fields*. Metrics are named like files. Figure 2 shows an example metric called `/rpc/server/latency` that measures the latency of RPCs to a server; it has two metric fields that distinguish RPCs by `service` and `command`.

The value type can be `boolean`, `int64`, `double`, `string`, `distribution`, or `tuple` of other types. All of them are standard types except `distribution`, which is a compact type that represents a large number of `double` values. A distribution includes a *histogram* that partitions a set of `double` values into subsets called *buckets* and summarizes values in each bucket using overall statistics such as mean, count, and standard deviation [28]. Bucket boundaries are configurable for trade-off between data granularity (i.e., accuracy) and storage costs: users may specify finer buckets for more popular value ranges. Figure 3 shows an example distribution-typed time series of `/rpc/server/latency` which measures servers' latency in handling RPCs; and it has a fixed bucket size of 10ms. Distribution-typed points of a time series can have different bucket boundaries; interpolation is used in queries that span points with different bucket boundaries. Distributions are an effective feature for summarizing a large number of samples. Mean latency is not enough for system monitoring—we also need other statistics such as 99th and 99.9th percentiles. To get these efficiently, histogram support—aka distribution—is indispensable.

Exemplars. Each bucket in a distribution may contain an *exemplar* of values in that bucket. An exemplar for RPC metrics, such as `/rpc/server/latency`, may be a Dapper RPC trace [41], which is very useful in debugging high RPC latency. Additionally, an exemplar contains information of its originating target and metric field values. The information is kept during distribution aggregation, therefore a user can easily identify problematic tasks via outlier exemplars. Figure 4 shows a heat map of a distribution-typed time series including the exemplar of a slow RPC that may explain the tail latency spike in the middle of the graph.

Metric types. A metric may be a *gauge* or a *cumulative*. For each point of a gauge time series, its value is an instantaneous measurement, e.g., queue length, at the time indicated by the point timestamp. For each point of a cumulative time series, its value is the accumulation of the measured aspect from a *start time* to the time indicated by its timestamp. For example, `/rpc/server/latency` in Figure 3 is a cumulative metric: each point is a latency distribution of *all* RPCs from its start time, i.e., the start time of the RPC server. Cumulative metrics are *robust* in that they still make sense if some points are missing, because each point contains all changes of earlier points sharing the same start time. Cumulative metrics are important to support distributed systems which consist of many servers that may be regularly restarted due to job scheduling [46], where points may go missing during restarts.

4. SCALABLE COLLECTION

To ingest a massive volume of time series data in real time, Monarch uses two divide-and-conquer strategies and one key optimization that aggregates data during collection.

4.1 Data Collection Overview

The right side of Figure 1 gives an overview of Monarch's collection path. The two levels of routers perform two levels of divide-and-conquer: *ingestion routers* regionalize time series data into zones according to *location fields*, and *leaf routers* distribute data across leaves according to the *range assigner*. Recall that each time series is associated with a target and one of the target fields is a location field.

Writing time series data into Monarch follows four steps:

1. A client sends data to one of the nearby ingestion routers, which are distributed across all our clusters. Clients usually use our instrumentation library, which automatically writes data at the frequency necessary to fulfill retention policies (see Section 6.2.2).

2. The ingestion router finds the destination zone based on the value of the target's location field, and forwards the data to a leaf router in the destination zone. The location-to-zone mapping is specified in configuration to ingestion routers and can be updated dynamically.

3. The leaf router forwards the data to the leaves responsible for the target ranges containing the target. Within each zone, time series are sharded lexicographically by their target strings (see Section 4.2). Each leaf router maintains a continuously-updated *range map* that maps each target range to three leaf replicas. Note that leaf routers get updates to the range map from leaves instead of the range assigner. Also, target ranges jointly cover the entire string universe; all new

targets will be picked up automatically without intervention from the assigner. So data collection continues to work if the assigner suffers a transient failure.

4. Each leaf writes data into its in-memory store and recovery logs. The in-memory time series store is highly optimized: it (1) encodes timestamps efficiently and shares timestamp sequences among time series from the same target; (2) handles delta and run-length encoding of time series values of complex types including distribution and tuple; (3) supports fast read, write, and snapshot; (4) operates continuously while processing queries and moving target ranges; and (5) minimizes memory fragmentation and allocation churn. To achieve a balance between CPU and memory [22], the in-memory store performs only light compression such as timestamp sharing and delta encoding. Timestamp sharing is quite effective: one timestamp sequence is shared by around ten time series on average.

Note that leaves do not wait for acknowledgement when writing to the *recovery logs* per range. Leaves write logs to distributed file system instances (i.e., Colossus [18]) in multiple distinct clusters and independently fail over by probing the health of a log. However, the system needs to continue functioning even when *all* Colossus instances are unavailable, hence the best-effort nature of the write to the log. Recovery logs are compacted, rewritten into a format amenable for fast reads (leaves write to logs in a write-optimized format), and merged into the long-term repository by continuously-running background processes whose details we omit from this paper. All log files are also asynchronously replicated across three clusters to increase availability.

Data collection by leaves also triggers updates in the zone and root *index servers* which are used to constrain query fanout (see Section 5.4).

4.2 Intra-zone Load Balancing

As a reminder, a table schema consists of a target schema and a metric schema. The lexicographic sharding of data in a zone uses only the key columns corresponding to the target schema. This greatly reduces ingestion fanout: in a single write message, a target can send one time series point each for hundreds of different metrics; and having all the time series for a target together means that the write message only needs to go to up to three leaf replicas. This not only allows a zone to scale horizontally by adding more leaf nodes, but also restricts most queries to a small subset of leaf nodes. Additionally, commonly used intra-target joins on the query path can be pushed down to the leaf-level, which makes queries cheaper and faster (see Section 5.3).

In addition, we allow heterogeneous replication policies (1 to 3 replicas) for users to trade off between availability and storage cost. Replicas of each target range have the same boundaries, but their data size and induced CPU load may differ because, for example, one user may retain only the first replica at a fine time granularity while another user retains all three replicas at a coarse granularity. Therefore, the range assigner assigns each target range replica individually. Of course, a leaf is never assigned multiple replicas of a single range. Usually, a Monarch zone contains leaves in multiple failure domains (clusters); the assigner assigns the replicas for a range to different failure domains.

Range assigners balance load in ways similar to Slicer [1]. Within each zone, the range assigner splits, merges, and moves ranges between leaves to cope with changes in the CPU load and memory usage imposed by the range on the leaf that stores it. While range assignment is changing, data collection works seamlessly by taking advantage of recovery logs. For example (range splits and merges are similar), the following events occur once the range assigner decided to move a range, say $\boldsymbol{R}$, to reduce the load on the source leaf:

1. The range assigner selects a destination leaf with light load and assigns $\boldsymbol{R}$ to it. The destination leaf starts to collect data for $\boldsymbol{R}$ by informing leaf routers of its new assignment of $\boldsymbol{R}$, storing time series with keys within $\boldsymbol{R}$, and writing recovery logs.

2. After waiting for one second for data logged by the source leaf to reach disks[2], the destination leaf starts to recover older data within $\boldsymbol{R}$, in reverse chronological order (since newer data is more critical), from the recovery logs.

3. Once the destination leaf fully recovers data in $\boldsymbol{R}$, it notifies the range assigner to unassign $\boldsymbol{R}$ from the source leaf. The source leaf then stops collecting data for $\boldsymbol{R}$ and drops the data from its in-memory store.

During this process, both the source and destination leaves are collecting, storing, and logging the same data simultaneously to provide continuous data availability for the range $\boldsymbol{R}$. Note that it is the job of leaves, instead of the range assigner, to keep leaf routers updated about range assignments for two reasons: (1) leaves are the source of truth where data is stored; and (2) it allows the system to degrade gracefully during a transient range assigner failure.

4.3 Collection Aggregation

For some monitoring scenarios, it is prohibitively expensive to store time series data exactly as written by clients. One example is monitoring disk I/O, served by millions of disk servers, where each I/O operation (IOP) is accounted to one of tens of thousands of users in Google. This generates tens of billions of time series, which is very expensive to store naively. However, one may only care about the aggregate IOPs per user across all disk servers in a cluster. *Collection aggregation* solves this problem by aggregating data during ingestion.

Delta time series. We usually recommend clients use cumulative time series for metrics such as disk IOPs because they are resilient to missing points (see Section 3.2). However, aggregating cumulative values with very different start times is meaningless. Therefore, collection aggregation requires originating targets to write *deltas* between adjacent cumulative points instead of cumulative points directly. For example, each disk server could write to Monarch every T_D seconds the per-user IOP counts it served in the past T_D seconds. The *leaf routers* accept the writes and forward all the writes for a user to the same set of leaf replicas. The deltas can be pre-aggregated in the client and the leaf routers, with final aggregation done at the leaves.

[2]Recall that, to withstand file system failures, leaves do not wait for log writes to be acknowledged. The one second wait length is almost always sufficient in practice. Also, the range assigner waits for the recovery from logs to finish before finalizing the range movement.

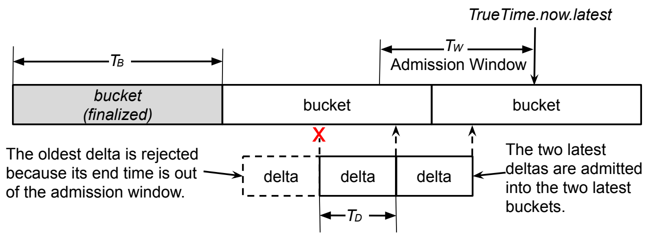

TrueTime.now.latest
T_B
T_W
Admission Window
bucket (finalized)
bucket
bucket
The oldest delta is rejected because its end time is out of the admission window.
delta
delta
delta
The two latest deltas are admitted into the two latest buckets.
T_D

Figure 5: Collection aggregation using buckets and a sliding admission window.

Bucketing. During collection aggregation, leaves put deltas into consecutive time buckets according to the end time of deltas, as illustrated in Figure 5. The bucket length T_B is the period of the output time series, and can be configured by clients. The bucket boundaries are aligned differently among output time series for load-smearing purposes. Deltas within each bucket are aggregated into one point according to a user-selected reducer; e.g., the disk I/O example uses a *sum* reducer that adds up the number of IOPs for a user from all disk servers.

Admission window. In addition, each leaf also maintains a sliding admission window and rejects deltas older than the window length T_W. Therefore, older buckets become immutable and generate finalized points that can be efficiently stored with delta and run-length encoding. The admission window also enables Monarch to recover quickly from network congestion; otherwise, leaves may be flooded by delayed traffic and never catch up to recent data, which is more important for critical alerting. In practice, rejected writes comprise only a negligible fraction of traffic. Once a bucket's end time moves out of the admission window, the bucket is finalized: the aggregated point is written to the in-memory store and the recovery logs.

To handle clock skews, we use TrueTime [13] to timestamp deltas, buckets, and the admission window. To compromise between ingestion traffic volume and time series accuracy, the delta period T_D is set to 10 seconds in practice. The length of the admission window is $T_W = T_D + TT.now.latest - TT.now.earliest$, where TT is TrueTime. The bucket length, $1\text{s} \leq T_B \leq 60\text{s}$, is configured by clients. It takes time $T_B + T_W$ to finalize a bucket, so recovery logs are normally delayed by up to around 70 seconds with a max T_B of 60 seconds. During range movement, T_B is temporarily adjusted to 1 second, since 70 seconds is too long for load balancing, as the leaf may be overloaded in the meantime.

5. SCALABLE QUERIES

To query time series data, Monarch provides an expressive language powered by a distributed engine that localizes query execution using static invariants and a novel index.

5.1 Query Language

A Monarch query is a pipeline of relational-algebra-like table operations, each of which takes zero or more time series tables as input and produces a single table as output. Figure 6 shows a query that returns the table shown in Figure 7: the RPC latency distribution of a set of tasks broken down by build labels (i.e., binary versions). This query can be used to detect abnormal releases causing high RPC latency. Each line in Figure 6 is a table operation.

```
1 { fetch ComputeTask::/rpc/server/latency
2     | filter user=="monarch"
3     | align delta(1h)
4 ; fetch ComputeTask::/build/label
5     | filter user=="monarch" && job=~"mixer.*"
6 } | join
7   | group_by [label], aggregate(latency)
```

Figure 6: An example query of latency distributions broken down by build label. The underlined are table operators. `delta` and `aggregate` are functions. "=~" denotes regular expression matching.

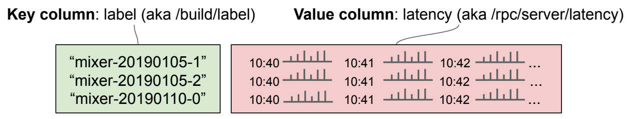

Key column: label (aka /build/label)
Value column: latency (aka /rpc/server/latency)
"mixer-20190105-1"
"mixer-20190105-2"
"mixer-20190110-0"
10:40
10:41
10:42
...

Figure 7: An example output time series table.

The `fetch` operation on Line 1 reads the time series table defined by the named target and metric schema from Figure 2. On Line 4, the `fetch` reads the table for the same target schema and metric `/build/label` whose time series value is a build label string for the target.

The `filter` operation has a predicate that is evaluated for each time series and only passes through those for which the predicate is true. The predicate on Line 2 is a single equality *field predicate* on the `user` field. Predicates can be arbitrarily complex, for example combining field predicates with logical operators as shown on Line 5.

The `align` operation on Line 3 produces a table in which all the time series have timestamps at the same regularly spaced interval from the same start time. The *delta* window operation estimates the latency distribution between the time of each aligned output point and one hour earlier. Having aligned input is important for any operation that combines time series, such as `join` or `group_by`. The `align` can be automatically supplied where needed as it is for `/build/label` (which lacks an explicit `align` operation).

The `join` operation on Line 6 does a natural (inner) join on the key columns of the input tables from the queries separated by the semicolon in the brackets { }. It produces a table with key columns from both inputs and a time series with dual value points: the latency distribution and the build label. The output contains a time series for each pair of input time series whose common key columns match. Left-, right-, and full-outer joins are also supported.

The `group_by` operation on Line 7 makes the key columns for each time series to contain only `label`, the build label. It then combines all the time series with the same key (same build label) by aggregating the distribution values, point by point. Figure 7 shows its results.

The operations in Figure 6 are a subset of the available operations, which also include the ability to choose the top n time series according to a value expression, aggregate values across time as well as across different time series, remap schemas and modify key and value columns, union input tables, and compute time series values with arbitrary expressions such as extracting percentiles from distribution values.

5.2 Query Execution Overview

There are two kinds of queries in the system: *ad hoc queries* and *standing queries*. Ad hoc queries come from users outside of the system. Standing queries are periodic materialized-view queries whose results are stored back into Monarch; teams use them: (1) to condense data for faster subsequent querying and/or cost saving; and (2) to generate alerts. Standing queries can be evaluated by either regional *zone evaluators* or global *root evaluators*. The decision is based on static analysis of the query and the table schemas of the inputs to the query (details in Section 5.3). The majority of standing queries are evaluated by *zone evaluators* which send identical copies of the query to the corresponding zone mixers and write the output to their zone. Such queries are efficient and resilient to network partition. The zone and root evaluators are sharded by hashes of standing queries they process, allowing us to scale to millions of standing queries.

Query tree. As shown in Figure 1, global queries are evaluated in a tree hierarchy of three levels. A *root mixer* receives the query and fans out to *zone mixers*, each of which fans out to *leaves* in that zone. The zonal standing queries are sent directly to zone mixers. To constrain the fanout, root mixers and zone mixers consult the *index servers* for a set of potentially relevant children for the query (see Section 5.4). A leaf or zone is *relevant* if the field hints index indicates that it could have data relevant to the query.

Level analysis. When a node receives a query, it determines the levels at which each query operation runs and sends down only the parts to be executed by the lower levels (details in Section 5.3). In addition, the root of the execution tree performs security and access-control checks and potentially rewrites the query for static optimization. During query execution, lower-level nodes produce and stream the output time series to the higher-level nodes which combine the time series from across their children. Higher-level nodes allocate buffer space for time series from each participating child according to the network latency from that child, and control the streaming rate by a token-based flow control algorithm.

Replica resolution. Since the replication of data is highly configurable, replicas may retain time series with different duration and frequency. Additionally, as the target ranges may be moving (see Section 4.2), some replicas can be in recovery with incomplete data. To choose the leaf with the *best quality* of data in terms of time bounds, density, and completeness, zonal queries go through the *replica resolution* process before processing data. Relevant leaves return the matched targets and their quality summary, and the zone mixer shards the targets into target ranges, selecting for each range a single leaf based on the quality. Each leaf then evaluates the table operations sent to it for the target range assigned to it. Though the range assigner has the target information, replica resolution is done purely from the target data actually on each leaf. This avoids a dependency on the range assigner and avoids overloading it. While processing queries, relevant data may be deleted because of range movements and retention expiration; to prevent that, leaves take a *snapshot* of the input data until queries finish.

User isolation. Monarch runs as a shared service; the resources on the query execution nodes are shared among queries from different users. For user isolation, memory used by queries is tracked locally and across nodes, and queries are cancelled if a user's queries use too much memory. Query threads are put into per-user *cgroups* [45], each of which is assigned a fair share of CPU time.

5.3 Query Pushdown

Monarch pushes down evaluation of a query's table operations as close to the source data as possible. This pushdown uses static invariants on the data layout, derived from the target schema definition, to determine the level at which an operation can be fully completed, with each node in this level providing a disjoint subset of all output time series for the operation. This allows the subsequent operations to start from that level. Query pushdown increases the scale of queries that can be evaluated and reduces query latency because (1) more evaluation at lower levels means more concurrency and evenly distributed load; and (2) full or partial aggregations computed at lower levels substantially decrease the amount of data transferred to higher level nodes.

Pushdown to zone. Recall that data is routed to zones by the value in the *location* target field. Data for a specific location can live only in one zone. If an output time series of an operation only combines input time series from a single zone, the operation can complete at the zone level. For example, a `group_by` where the output time series keys contain the *location* field, and a `join` between two inputs with a common *location* field can both be completed at the zone level. Therefore, the only standing queries issued by the root evaluators are those that either (a) operate on some input data in the *regionless zone* which stores the standing query results with no location field, or (b) aggregate data across zones, for example by either dropping the location field in the input time series or by doing a top n operation across time series in different zones. In practice, this allows up to 95% of standing queries to be fully evaluated at zone level by zone evaluators, greatly increasing tolerance to network partition. Furthermore, this significantly reduces latency by avoiding cross-region writes from root evaluators to leaves.

Pushdown to leaf. As mentioned in Section 4.2, the data is sharded according to *target ranges* across leaves within a zone. Therefore, a leaf has either none or all of the data from a target. Operations within a target complete at the leaf level. For example, a `group_by` that retains all the target fields in the output and a `join` whose inputs have all the target fields can both complete at the leaf level. Intra-target joins are very common in our monitoring workload, such as filtering with slow changing *metadata* time series stored in the same target. In the example query in Figure 6, the `join` completes at the leaf and `/build/label` can be considered as metadata (or a property) of the target (i.e., the running task), which changes only when a new version of the binary is pushed. In addition, since a *target range* contains consecutive targets (i.e., the first several target fields might be identical for these targets), a leaf usually contains multiple targets relevant to the query. Aggregations are pushed down as much as possible, even when they cannot be completed at the leaf level. The leaves aggregate time series across the co-located targets and send these results to the mixers. The `group_by` in the example query is executed at all three levels. No matter how many input time series there are for each node, the node only outputs one time series for each group (i.e., one time series per build label in the example).

Fixed Fields. Some fields can be determined to *fix* to constant values by static analysis on the query and schemas, and they are used to push down more query operations. For example, when fetching time series from a specific *cluster* with a filter operation of `filter cluster == "om"`, a global aggregation can complete at the zone level, because the input time series are stored in only one zone that contains the specific cluster value `om`.

5.4 Field Hints Index

For high scalability, Monarch uses *field hints index (FHI)*, stored in *index servers*, to limit the fanout when sending a query from parent to children, by skipping *irrelevant* children (those without input data to the particular query). An FHI is a concise, continuously-updated index of time series field values from all children. FHIs skip irrelevant children by analyzing *field predicates* from queries, and handle regular expression predicates efficiently without iterating through the exact field values. FHI works with zones with trillions of time series keys and more than 10,000 leaves while keeping the size small enough to fit in memory. False positives are possible in FHI just as in Bloom filters [7]; that is, FHIs may also return irrelevant children. False positives do not affect correctness because irrelevant children are ignored via replica resolution later.

A field hint is an excerpt of a field value. The most common hints are trigrams; for example, `^^m`, `^mo`, `mon`, `ona`, `nar`, `arc`, `rch`, `ch$`, and `h$$` are trigram hints of field value `monarch` where `^` and `$` represent the start and end of text, respectively. A field hint index is essentially a multimap that maps the fingerprint of a field hint to the subset of children containing the hint. A fingerprint is an `int64` generated deterministically from three inputs of a hint: the schema name, the field name, and the excerpt (i.e., trigrams).

When pushing down a query, a root (zone) mixer extracts a set of mandatory field hints from the query, and looks up the root (zone) FHI for the destination zones (leaves). Take the query in Figure 6 for example: its predicate regexp `‘mixer.*’` entails `^^m`, `^mi`, `mix`, `ixe`, and `xer`. Any child matching the predicate must contain all these trigrams. Therefore, only children in `FHI[^^m]` ∩ `FHI[^mi]` ∩ `FHI[mix]` ∩ `FHI[ixe]` ∩ `FHI[xer]` need to be queried.

We minimize the size of FHI to fit it in memory so that Monarch still works during outages of secondary storage systems. Storing FHI in memory also allows fast updates and lookups. FHI trades accuracy for a small index size: (1) It indexes short excerpts to reduce the number of unique hints. For instance, there are at most 26^3 unique trigrams for lowercase letters. Consequently, in the previous example, FHI considers a leaf with target `job:‘mixixer’` relevant although the leaf's target does not match regexp `‘mixer.*’`. (2) FHI treats each field separately. This causes false positives for queries with predicates on multiple fields. For example, a leaf with two targets `user:‘monarch’,job:‘leaf’` and `user:‘foo’,job:‘mixer.root’` is considered by FHI a match for predicate `user==‘monarch’&&job=~‘mixer.*’` (Figure 6) although neither of the two targets actually match.

Despite their small sizes (a few GB or smaller), FHIs reduce query fanout by around 99.5% at zone level and by 80% at root level. FHI also has four additional features:

1. Indexing trigrams allows FHIs to filter queries with regexp-based field predicates. The RE2 library can turn a regexp into a set algebra expression with trigrams and operations (union and intersection) [14]. To match a regexp predicate, Monarch simply looks up its trigrams in FHIs and evaluates the expression.

2. FHIs allow fine-grained tradeoff between index accuracy and size by using different excerpts. For instance, string fields with small character sets (e.g. ISBN) can be configured to use fourgrams and full strings, in addition to trigrams, as excerpts for higher accuracy.

3. Monarch combines static analysis and FHIs to further reduce the fanout of queries with joins: it sends the example query (which contains a leaf-level inner join) only to leaves satisfying *both* of the two filter predicates in Figure 6 (the join will only produce output on such leaves anyway). This technique is similarly applied to queries with nested joins of varying semantics.

4. Metric names are also indexed, by full string, and are treated as values of a reserved “`:metric`” field. Thus, FHIs even help queries without any field predicates.

As illustrated in Figure 1, FHIs are built from bottom up and maintained in index servers. Due to its small size, an FHI need not be stored persistently. It is built (within minutes) from live leaves when an index server starts. A zone index server maintains a long-lived streaming RPC [26] to every leaf in the zone for continuous updates to the zone FHI. A root index server similarly streams updates to the root FHI from every zone. Field hints updates are transported over the network at high priority. Missing updates to the root FHI are thus reliable indicators of zone unavailability, and are used to make global queries resilient to zone unavailability.

Similar Index Within Each Leaf. Field hints index introduced so far resides in index servers and helps each query to locate relevant leaves. Within each leaf, there is a similar index that helps each query to find relevant targets among the large number of targets the leaf is responsible for. To summarize, a query starts from the root, uses root-level FHI in root index servers to find relevant zones, then uses zone-level FHI in zone index servers to find relevant leaves, and finally uses leaf-level FHI in leaves to find relevant targets.

5.5 Reliable Queries

As a monitoring system, it is especially important for Monarch to handle failures gracefully. We already discussed that Monarch zones continue to function even during failures of the file system or global components. Here we discuss how we make queries resilient to zonal and leaf-level failures.

Zone pruning. At the global level, we need to protect global queries from regional failures. Long-term statistics show that almost all (99.998%) successful global queries start to stream results from zones within the first half of their deadlines. This enabled us to enforce a shorter per-zone *soft query deadline* as a simple way of detecting the health of queried zones. A zone is pruned if it is completely unresponsive by the soft query deadline. This gives each zone a chance to return responses but not significantly delay query processing if it suffers from low availability. Users are notified of pruned zones as part of the query results.

Hedged reads. Within a zone, a single query may still fanout to more than 10,000 leaves. To make queries resilient to slow leaf nodes, Monarch reads data from faster replicas. As described in Section 4.2, leaves can contain overlapping but non-identical sets of targets relevant to a query. As we push down operations that can aggregate across all the relevant targets at the leaf (see Section 5.3), there is no trivial output data equivalence across leaves. Even when leaves return the same output time series keys, they might be aggregations from different input data. Therefore, a vanilla hedged read approach does not work.

Monarch constructs the equivalence of input data on the query path with a novel hedged-read approach. As mentioned before, the zone mixer selects a leaf (called the *primary leaf*) to run the query for each target range during replica resolution. The zone mixer also constructs a set of fallback leaves for the responsible ranges of each primary leaf. The zone mixer starts processing time series reads from the primary leaves while tracking their response latencies. If a primary leaf is unresponsive or abnormally slow, the zone mixer replicates the query to the equivalent set of fallback leaves. The query continues in parallel between the primary leaf and the fallback leaves, and the zone mixer extracts and de-duplicates the responses from the faster of the two.

6. CONFIGURATION MANAGEMENT

Due to the nature of running Monarch as a distributed, multi-tenant service, a centralized configuration management system is needed to give users convenient, fine-grained control over their monitoring and distribute configuration throughout the system. Users interact with a single global view of configuration that affects all Monarch zones.

6.1 Configuration Distribution

All configuration modifications are handled by the *configuration server*, as shown in Figure 1, which stores them in a global Spanner database [13]. A configuration element is validated against its dependencies (e.g., for a standing query, the schemas it uses) before being committed.

The configuration server is also responsible for transforming high-level configuration to a form that is more efficiently distributed and cached by other components. For example, leaves only need to be aware of the output schema of a standing query to store its results. Doing this transformation within the configuration system itself ensures consistency across Monarch components and simplifies client code, reducing the risk of a faulty configuration change taking down other components. Dependencies are tracked to keep these transformations up to date.

Configuration state is replicated to *configuration mirrors* within each zone, which are then distributed to other components within the zone, making it highly available even in the face of network partitions. Zonal components such as leaves cache relevant configuration in memory to minimize latency of configuration lookups, which are copied from the configuration mirror at startup with subsequent changes being sent periodically. Normally the cached configuration is up to date, but if the configuration mirror becomes unavailable, zonal components can continue to operate, albeit with stale configuration and our SREs alerted.

6.2 Aspects of Configuration

Predefined configuration is already installed to collect, query, and alert on data for common target and metric schemas, providing basic monitoring to new users with minimal setup. Users can also install their own configuration to utilize the full flexibility of Monarch. The following subsections describe major parts of users' configuration state:

6.2.1 *Schemas*

There are predefined target schemas and metric schemas, such as `ComputeTask` and `/rpc/server/latency` as described in Section 3, that allow data to be collected automatically for common workloads and libraries. Advanced users can define their own custom target schemas, providing the flexibility to monitor many types of entities.

Monarch provides a convenient instrumentation library for users to define schematized metrics in code. The library also periodically sends measurements as time series points to Monarch as configured in Section 6.2.2. Users can conveniently add columns as their monitoring evolves, and the metric schema will be updated automatically. Users can set access controls on their metric namespace to prevent other users from modifying their schemas.

Table 1: Number of Monarch tasks by component, rounded to the third significant digit. Components for logging, recovery, long-term repository, quota management, and other supporting services are omitted.

Component	#Task	Component	#Task
Leaf	144,000	Range assigner	114
Config mirror	2,590	Config server	15
Leaf router	19,700	Ingestion router	9,390
Zone mixer	40,300	Root mixer	1,620
Zone index server	3,390	Root index server	139
Zone evaluator	1,120	Root evaluator	36

6.2.2 *Collection, Aggregation, and Retention*

Users have fine-grained control over data retention policies, i.e., which metrics to collect from which targets and how to retain them. They can control how frequently data is sampled, how long it is retained, what the storage medium is, and how many replicas to store. They can also downsample data after a certain age to reduce storage costs.

To save costs further, users can also configure aggregation of metrics during collection as discussed in Section 4.3.

6.2.3 *Standing Queries*

Users can set up standing queries that are evaluated periodically and whose results are stored back into Monarch (Section 5.2). Users can configure their standing query to execute in a sharded fashion to handle very large inputs. Users can also configure alerts, which are standing queries with a boolean output comparing against user-defined alerting conditions. They also specify how to be notified (e.g., email or page) when alerting conditions are met.

7. EVALUATION

Monarch has many experimental deployments and three production deployments: *internal*, *external*, and *meta*. *Internal* and *external* are for customers inside and outside Google; *meta* runs a proven-stable older version of Monarch and monitors all other Monarch deployments. Below, we only present numbers from the *internal* deployment, which does not contain external customer data. Note that Monarch's scale is not merely a function of the scale of the systems being monitored. In fact, it is significantly more influenced by other factors such as continuous internal optimizations, what aspects are being monitored, how much data is aggregated, etc.

7.1 System Scale

Monarch's internal deployment is in active use by more than 30,000 employees and teams inside Google. It runs in

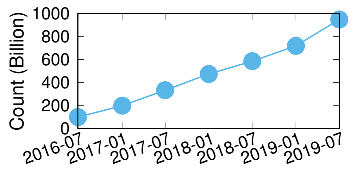


Figure 8: Time series count.

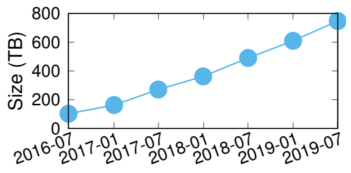


Figure 9: Time series memory size.

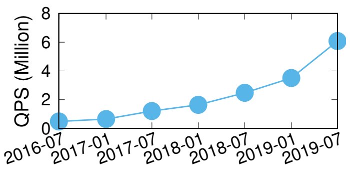


Figure 10: Queries per second.

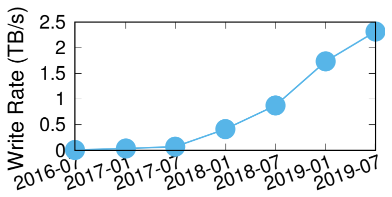


Figure 11: Time series data written per second. The write rate was almost zero around July 2016 because back then data was ingested using a different mechanism, which is not included in this figure. Detailed measurement of the old mechanism is no longer available; its traffic peaked at around 0.4TB/s, gradually diminished, and became negligible around March 2018.

38 zones spread across five continents. It has round 400,000 tasks (the important ones are listed in Table 1), with the vast majority of tasks being leaves because they serve as the in-memory time series data store. Classifying zones by the number of leaves, there are: 5 small zones (< 100 leaves), 16 medium zones (< 1000), 11 large zones ($< 10,000$), and 6 huge zones ($\geq 10,000$). Each zone contains three range assigners, one of which is elected to be the master. Other components in Table 1 (config, router, mixer, index server, and evaluator) appear at both zone and root levels; the root tasks are fewer than the zone counterparts because root tasks distribute work to zone tasks as much as possible.

Monarch's unique architecture and optimizations make it highly scalable. It has sustained fast growth since its inception and is still growing rapidly. Figure 8 and Figure 9 show the number of time series and the bytes they consume in Monarch's internal deployment. As of July 2019, Monarch stored nearly 950 billion time series, consuming around 750TB memory with a highly-optimized data structure. Accommodating such growth rates requires not only high horizontal scalability in key components but also innovative optimizations for collection and query, such as collection aggregation (Section 4.3) and field hints index (Section 5.4).

As shown in Figure 11, Monarch's internal deployment ingested around 2.2 terabytes of data per second in July 2019. Between July 2018 and January 2019, the ingestion rate almost doubled because collection aggregation enabled collection of metrics (e.g., disk I/O) with tens of billions of time series keys. On average, Monarch aggregates 36 input time series into one time series during collection; in extreme cases, over one million input time series into one. Collection aggregation is highly efficient and can aggregate one million typical time series using only a single CPU core. In addition to the obvious RAM savings (fewer time series to store), collection aggregation uses approximately 25% of the CPU of the alternative procedure of writing the raw time series to Monarch, querying via a standing query, and then writing the desired output.

Table 2: Field hints index (FHI) statistics. Children of the root FHI are the 38 zones. Zone FHIs are named after the zone, and their children are leaves. Suppression ratio is the percentage of children skipped by query thanks to FHI. Hit ratio is the percentage of visited children that actually have data. 26 other zones are omitted.

FHI Name	Child Count	Fingerprint Count (k)	Suppr. Ratio	Hit Ratio
root	38	214,468	75.8	45.0
small-zone-1	15	56	99.9	60.5
small-zone-2	56	1,916	99.7	51.8
small-zone-3	96	3,849	99.5	43.8
medium-zone-1	156	6,377	99.4	36.3
medium-zone-2	330	12,186	99.5	32.9
medium-zone-3	691	23,404	99.2	33.4
large-zone-1	1,517	43,584	99.3	26.5
large-zone-2	5,702	159,090	99.2	22.5
large-zone-3	7,420	280,816	99.3	21.6
huge-zone-1	12,764	544,815	99.4	17.8
huge-zone-2	15,475	654,750	99.4	18.4
huge-zone-3	16,681	627,571	99.6	21.4

7.2 Scalable Queries

To evaluate query performance, we present key statistics about query pushdown, field hints index (FHI, Section 5.4), and query latency. We also examine the performance impact of various optimizations using an example query.

7.2.1 Overall Query Performance

Figure 10 shows the query rate of Monarch's internal deployment: it has sustained exponential growth and was serving over six million QPS as of July 2019. Approximately 95% of all queries are standing queries (including alerting queries). This is because users usually set up standing queries (1) to reduce response latency for queries that are known to be exercised frequently and (2) for alerting, whereas they only issue ad hoc non-standing-queries very occasionally. Additionally, the majority of such standing queries are initiated by the zone evaluators (as opposed to the root evaluators) because Monarch aggressively pushes down those standing queries that can be independently evaluated in each zone to the zone evaluators to reduce the overall amount of unnecessary work performed by the root evaluators.

To quantify the query pushdown from zone mixers to leaves, we measured that the overall ratio of output to input time series count at leaves is 23.3%. Put another way, pushdown reduces the volume of data seen by zone mixers by a factor of four.

Besides query pushdown, field hints index is another key enabler for scalable queries. Table 2 shows the statistics of the root and some zone FHIs. The root FHI contains around 170 million fingerprints; it narrows average root query fanout down to $34 \times (1 - 0.758) \approx 9$, among which around $9 \times 0.45 \approx 4$ zones actually have data. Zones vary a lot in their leaf

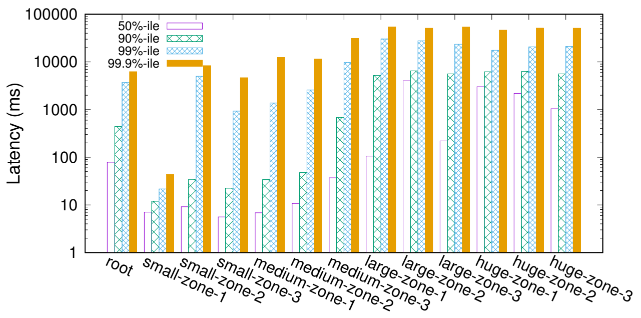


Figure 12: 50, 90, 99, and 99.9 percentile query latency. Root queries include ad hoc queries and root-level standing queries; zone queries are mostly standing queries initiated by zone evaluators. The Y-axis is milliseconds on a log scale.

counts, so do the fingerprint counts in their FHIs. Yet, all zone FHIs have a suppression ratio of 99.2% or higher. FHI's hit ratio ranges from 15.7% to 60.5% across zones. In general, FHIs have higher hit ratio in smaller zones because false positives in field hints are less likely when a zone has fewer targets. FHI is space efficient; on average, a fingerprint occupies only 1.3 bytes of memory. *huge-zone-2* has the largest number of fingerprints (654 million); yet its FHI size is merely 808MB. We achieved this by encoding leaves with small integers and storing integer codes of popular fingerprints in bitsets.

As shown in Figure 12, root queries have a median latency of 79ms, and a 99.9%-ile latency of 6s. The latency difference is due to the number of input time series to a query: a median query involves only 1 time series whereas a 99.9%-ile query involves 12,500. Zones also differ significantly in query latency. In general, smaller zones have faster queries. There are exceptions: noticeably, *large-zone-2* has much higher median query latency than *large-zone-1* and *large-zone-3*. This is because the median number of input time series in *large-zone-2* is more than twice the other two large zones. The 99.9%-ile query latency of the large and huge zones are all around 50s. These are expensive standing queries that fetch 9 to 23 million time series per query. Many of them are queries that aggregate popular metrics (such as the predefined metric `/rpc/server/latency`) across all tasks of each job in a zone; because such metrics tend to be used by many users, we set up automatic standing queries for them to avoid redundant installations from individual users.

7.2.2 Individual Query Performance

Table 3 shows the performance impact of query optimizations on the example query in Figure 6. The query reads approximately 0.3 million input time series. The field hints index suggests 68k leaves to query, out of which 40k leaves contain relevant data matching the query.

As shown in Table 3, the query completes in 6.73 seconds when query pushdown and field hints index are enabled. If we disable partial aggregations (1) only on the leaves and (2) on both the leaves and the zone mixers, the query takes 9.75 seconds and 34.44 seconds to complete, resulting in a **1.4x** and a **5.1x** slowdown respectively. This is because, without the partial aggregations on the leaf and zone mixers, more time series need to be transferred to and be aggregated by higher execution levels with less parallelism (e.g., by only one root mixer vs. concurrently by many leaves).

Table 3: Performance impact on the query shown in Figure 6 with different query features enabled. We measured the latency and query fanout by pushing join and group_by to different levels and disabling field hints.

FHI	Join	Group_by	Latency(s)	#Leaves(k)
Yes	Leaf	Leaf	6.73	68
Yes	Leaf	Zone	9.75	68
Yes	Leaf	Root	34.44	68
Yes	Zone	Zone	242.50	92
Yes	Root	Root	1728.33	92
No	Leaf	Leaf	67.54	141

Additionally, if we perform joins only on (1) the zone mixers and (2) the root mixers, and aggregations on the same and higher levels, the query takes 242.5 seconds and 1728.3 seconds to complete, resulting in a **36.0x** and **256.7x** slowdown, respectively. Moving the execution of joins from lower level to higher level increases the number of time series transferred between levels, because both sides of the joins need to send the input time series to the higher level, some of which would have been filtered by the inner join. In addition, the higher level nodes work on a much larger input set of time series sequentially, which also significantly increases the processing latency. Note that leaf level joins also helped reduce the fanout from 92k to 68k leaves, thanks to the optimization in field hint index that intersects the matching leaves from the predicates on both sides of a leaf level inner join (the third additional feature of FHI in Section 5.4).

Finally, if we execute the query without consulting the field hint index on root and zone index servers and leaves, the query takes 67.54 seconds to complete, resulting in a **10.0x** slowdown. This demonstrates that the field hints index can be very effective in reducing query fanout and improving query latency because (1) field hints index reduces the fanout by eliminating 73k irrelevant leaves; (2) the indexing on leaves also eliminate huge amount of irrelevant targets and time series.

8. RELATED WORK

The explosive growth of time series data drives a proliferation of research [29, 5, 48] on its collection [35], clustering [34, 2], compression [11, 33, 6], modeling [44, 23], mining [17], query [4, 43], search [32, 38], storage [3], and visualization [31]. Much of the recent research focuses on managing time series in constrained hardware of wireless sensor network [11, 33] and the Internet of Things [24]; fewer studies are about cloud-scale time series management systems that store and query data in real-time [37, 30].

There are many open source time series databases [5]; Graphite [16], InfluxDB [27], OpenTSDB [12], Prometheus [39], and tsdb [15] are popular ones. They store data on secondary storage (local or distributed such as HBase [19, 27, 12]); the use of secondary storage makes them less desirable for critical monitoring. They support distributed deployment by scaling horizontally similar to a Monarch zone, but they lack the global configuration management and query aggregation that Monarch provides.

Gorilla [37, 25] is Facebook's in-memory time series database. A Gorilla time series is identified by a string key, as opposed to Monarch's structured data model. Gorilla lacks an expressive query language. Gorilla replicates data across regions for disaster recovery, limiting availability during a network partition. In contrast, Monarch, replicates data in nearby data centers for data locality. Gorilla also does not have an equivalent to Monarch's planet-scale query engine, or the optimizations that power it, such as localization of query execution based on field hints index, and query pushdown. Other Monarch features that Gorilla lacks include: (1) rich data types, such as distribution with exemplars; (2) collection optimizations, including lexicographical sharding and collection aggregation; (3) fine-grained configurations for retention policies; (4) standing and alerting queries.

Monarch's collection aggregation (Section 4.3), which reduces storage cost of cumulative metrics by aggregating time series as they are being ingested, is similar to *in-network aggregation* [42] used in wireless sensor networks.

9. LESSONS LEARNED

Over the past decade of active development and use, Monarch's feature set, architecture and core data structures have been constantly evolving. Key lessons learned include:

- *Lexicographic sharding of time series keys improves ingestion and query scalability*, enabling Monarch zones to scale to tens of thousands of leaves. All metrics from one target can be sent to their destination leaf in a single message. Query operations that aggregate or join data by target can be completed by a single leaf. Aggregations over adjacent targets are also more efficient where adjacent targets are present on the same leaf, limiting query fanout and reducing data transfer between leaves and mixers.

- *Push-based data collection improves system robustness while simplifying system architecture.* Early versions of Monarch discovered monitored entities and "pulled" monitoring data by querying the monitored entity. This required setting up discovery services and proxies, complicating system architecture and negatively impacting overall scalability. Push-based collection, where entities simply send their data to Monarch, eliminates these dependencies.

- *A schematized data model improves robustness and enhances performance.* While requiring slightly more effort to setup than systems like Borgmon [47] that work with unschematized data, operating on structured data allows queries to be validated and optimized *before* execution. In our experience, schemas have not imposed any significant burden on our users compared to Borgmon, thanks to our convenient and flexible configuration management.

- *System scaling is a continuous process.* Index servers, collection aggregation, and sharded standing queries are examples of features that were added after Monarch's initial design to address scaling issues. We continue to refine Monarch's architecture to support better horizontal scaling, and are constantly evolving internal data structures and algorithms to support larger data volumes and new usage patterns.

- *Running Monarch as a multi-tenant service is convenient for users, but challenging for developers.* Users have tremendous flexibility with how they use Monarch, and are isolated from the operational side of the service. The coexistence of vastly differing usage patterns, however, makes ensuring system stability a challenge. Features such as usage accounting, data sanitation, user isolation, and traffic throttling are necessary so that Monarch can provide service-level objectives (SLOs) for availability, correctness, and latency. Optimizations need to work for almost all use cases. Code changes to Monarch must be backwards-compatible to allow graceful live updates with possible rollbacks. We are constantly improving Monarch's multi-tenancy support as Monarch continues to onboard more users that stress many different aspects of the system.

10. CONCLUSION

Monarch is a planet-scale, multi-tenant in-memory time series database that manages trillions of time series. It is deployed across data centers in many different geographical regions. Monarch operates efficiently and reliably at this scale due to its architecture of autonomous regional monitoring sub-systems integrated into a coherent whole by global configuration and query planes. It adopts a novel, type-rich relational time series data model that allows efficient and scalable data storage while powering an expressive query language for rich data analysis. To accommodate this massive scale, Monarch employs a variety of optimization techniques for both data collection and query execution. For data collection, Monarch performs intra-zone load balancing and collection aggregation for improved reliability and efficiency. For query execution, Monarch executes each query in a distributed, hierarchical fashion, performing aggressive filtering and aggregation pushdown for improved performance and throughput, taking advantage of a compact yet powerful distributed index for efficient data pruning.

Since its initial deployment to production, Monarch has sustained years of rapid usage growth. It currently ingests terabytes of data per second, stores close to a petabyte of highly-compressed time series data in memory, and serves millions of queries per second. Monarch is instrumental to serving Google's monitoring and alerting needs at billion-user scale. It is also the foundational infrastructure layer that unlocks many use cases including anomaly detection for alerting, canary analysis for continuous integration and deployment, and automatic task sizing for resource optimizations on Google clusters.

Acknowledgement

Monarch would not have been possible without critical contributions provided by Alex Kehlenbeck, Ben Sigelman and Daniel Sturman; we thank them for their work and leadership. We also thank many engineers who made significant contributions to Monarch over the years, in particular (listed alphabetically): Karsten Behrmann, Dan Bulger, Steven Canfield, Onur Çakmak, Fay Chang, Shishi Chen, Mike Corcoran, Thibaut Coutelou, Lia Davitashvili, Alan Donovan, Mark Goudreau, Leonhard Gruenschloss, Sergey Ivanyuk, Maciej Kowalewski, Rory Kulz, Adam Mckaig, Josh Montana, Oleg Morfiianets, Lajos Nagy, Dustin Pho, Thomas Rast, Gunnar Ritter, Kirill Shevchuk, Dan Tulovsky, Adam Vartanian, Matthew Weaver and numerous others.

Finally, we thank the anonymous VLDB reviewers for their valuable comments.

11. REFERENCES

[1] A. Adya, D. Myers, J. Howell, J. Elson, C. Meek, V. Khemani, S. Fulger, P. Gu, L. Bhuvanagiri, J. Hunter, and et al. Slicer: Auto-sharding for datacenter applications. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 739–753, USA, 2016. USENIX Association.

[2] S. Aghabozorgi, A. S. Shirkhorshidi, and T. Y. Wah. Time-series clustering — a decade review. *Information Systems*, 53:16–38, 2015.

[3] M. P. Andersen and D. E. Culler. BTrDB: Optimizing storage system design for timeseries processing. In *Proceedings of the 14th Usenix Conference on File and Storage Technologies*, pages 39–52. USENIX Association, Feb. 2016.

[4] A. Arasu, S. Babu, and J. Widom. The CQL continuous query language: Semantic foundations and query execution. Technical Report 2003-67, Stanford InfoLab, 2003.

[5] A. Bader, O. Kopp, and M. Falkenthal. Survey and comparison of open source time series databases. In B. Mitschang, D. Nicklas, F. Leymann, H. Schöning, M. Herschel, J. Teubner, T. Härder, O. Kopp, and M. Wieland, editors, *Datenbanksysteme für Business, Technologie und Web (BTW 2017) - Workshopband*, pages 249–268. Gesellschaft für Informatik e.V., 2017.

[6] D. Blalock, S. Madden, and J. Guttag. Sprintz: Time series compression for the internet of things. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 2(3):93:1–93:23, Sept. 2018.

[7] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Communications of the ACM*, 13(7):422–426, July 1970.

[8] E. Brewer. Cap twelve years later: How the" rules" have changed. *Computer*, 45(2):23–29, 2012.

[9] E. Brewer. Spanner, truetime and the cap theorem. Technical report, 2017.

[10] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber. Bigtable: A distributed storage system for structured data. *ACM Transactions on Computer Systems (TOCS)*, 26(2):1–26, 2008.

[11] H. Chen, J. Li, and P. Mohapatra. Race: time series compression with rate adaptivity and error bound for sensor networks. In *2004 IEEE International Conference on Mobile Ad-hoc and Sensor Systems*, pages 124–133, Oct 2004.

[12] B. S. Chris Larsen. OpenTSDB - a distributed, scalable monitoring system. `http://opentsdb.net`.

[13] J. C. Corbett, J. Dean, M. Epstein, A. Fikes, C. Frost, J. J. Furman, S. Ghemawat, A. Gubarev, C. Heiser, P. Hochschild, W. Hsieh, S. Kanthak, E. Kogan, H. Li, A. Lloyd, S. Melnik, D. Mwaura, D. Nagle, S. Quinlan, R. Rao, L. Rolig, Y. Saito, M. Szymaniak, C. Taylor, R. Wang, and D. Woodford. Spanner: Google's globally distributed database. *ACM Transactions on Computer Systems*, 31(3):8:1–8:22, Aug. 2013.

[14] R. Cox. Regular expression matching with a trigram index, 2012. `https://swtch.com/~rsc/regexp/regexp4.html`.

[15] L. Deri, S. Mainardi, and F. Fusco. tsdb: A compressed database for time series. In *Proceedings of the 14th International Workshop on Traffic Monitoring and Analysis*, pages 143–156, Mar. 2012.

[16] J. Dixon. *Monitoring with Graphite: Tracking Dynamic Host and Application Metrics at Scale.* O'Reilly Media, 1 edition, March 2017.

[17] P. Esling and C. Agon. Time-series data mining. *ACM Computing Surveys*, 45(1):12:1–12:34, Dec. 2012.

[18] A. Fikes. Storage architecture and challenges. `https://cloud.google.com/files/storage_architecture_and_challenges.pdf`.

[19] L. George. *HBase: The Definitive Guide.* O'Reilly Media, 1 edition, 2011.

[20] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43. ACM, 2003.

[21] S. Gilbert and N. Lynch. Perspectives on the cap theorem. *Computer*, 45(2):30–36, 2012.

[22] Google. Snappy — a fast compressor/decompressor. `https://github.com/google/snappy`.

[23] K. W. Hipel and A. I. McLeod. *Time series modelling of water resources and environmental systems*, volume 45 of *Developments in Water Science.* Elsevier, 1994.

[24] J. Huang, A. Badam, R. Chandra, and E. B. Nightingale. Weardrive: Fast and energy-efficient storage for wearables. In *Proceedings of the 2015 USENIX Annual Technical Conference*, pages 613–625. USENIX Association, 2015.

[25] F. Inc. Beringei: a high performance, in memory time series storage engine, 2016. `https://github.com/facebookarchive/beringei`.

[26] G. Inc. gRPC: Bidirectional streaming RPC, 2017. `https://grpc.io/docs/guides/concepts/`.

[27] InfluxData. InfluxDB — open source time series, metrics, and analytics database. `http://influxdata.com`.

[28] Y. E. Ioannidis. Universality of serial histograms. *PVLDB*, pages 256–267, 1993.

[29] S. K. Jensen, T. B. Pedersen, and C. Thomsen. Time series management systems: A survey. *IEEE Transactions on Knowledge and Data Engineering*, 29(11):2581–2600, Nov. 2017.

[30] S. K. Jensen, T. B. Pedersen, and C. Thomsen. ModelarDB: Modular model-based time series management with Spark and Cassandra. *PVLDB*, 11(11):1688–1701, 2018.

[31] U. Jugel, Z. Jerzak, G. Hackenbroich, and V. Markl. M4: A visualization-oriented time series data aggregation. *PVLDB*, 7(10):797–808, 2014.

[32] H. Kondylakis, N. Dayan, K. Zoumpatianos, and T. Palpanas. Coconut: A scalable bottom-up approach for building data series indexes. *PVLDB*, 11(6):677–690, 2018.

[33] I. Lazaridis and S. Mehrotra. Capturing sensor-generated time series with quality guarantees. In *Proceedings of the 19th International Conference on Data Engineering*, pages 429–440, Mar. 2003.

[34] T. W. Liao. Clustering of time series data — a survey. *Pattern Recognition*, 38(11):1857–1874, 2005.

[35] J. Meehan, C. Aslantas, S. Zdonik, N. Tatbul, and J. Du. Data ingestion for the connected world. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research*, Jan. 2017.

[36] A. Merchant. Keynote address II: Optimal flash partitioning for storage workloads in google's colossus file system. Broomfield, CO, Oct. 2014. USENIX Association. The 2nd workshop on interactions of NVM/Flash with operating systems and workloads.

[37] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan. Gorilla: A fast, scalable, in-memory time series database. *PVLDB*, 8(12):1816–1827, 2015.

[38] T. Rakthanmanon, B. Campana, A. Mueen, G. Batista, B. Westover, Q. Zhu, J. Zakaria, and E. Keogh. Searching and mining trillions of time series subsequences under dynamic time warping. In *Proceedings of the 18th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 262–270. ACM, Aug. 2012.

[39] F. Reinartz, J. Volz, and B. Rabenstein. Prometheus – monitoring system & time series database. `http://prometheus.io/`.

[40] B. Samwel, J. Cieslewicz, B. Handy, J. Govig, P. Venetis, C. Yang, K. Peters, J. Shute, D. Tenedorio, H. Apte, and et al. F1 query: Declarative querying at scale. *PVLDB*, 11(12):1835–1848, 2018.

[41] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010.

[42] I. Solis and K. Obraczka. In-network aggregation trade-offs for data collection in wireless sensor networks. *Int. J. Sen. Netw.*, 1(3/4):200–212, Jan. 2006.

[43] N. Tatbul and S. Zdonik. Window-aware load shedding for aggregation queries over data streams. *PVLDB*, pages 799–810, 2006.

[44] S. J. Taylor. *Modelling Financial Time Series*. World Scientific, second edition, 2007.

[45] The Linux man-pages project. cgroups — Linux control groups. `http://man7.org/linux/man-pages/man7/cgroups.7.html`.

[46] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes. Large-scale cluster management at Google with Borg. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 18:1–18:17. ACM, 2015.

[47] J. Wilkinson. Practical alerting from time-series data. In B. Beyer, C. Jones, J. Petoff, and N. R. Murphy, editors, *Site Reliability Engineering*, pages 107–123. O'Reilly Media, 2016.

[48] T. W. Wlodarczyk. Overview of time series storage and processing in a cloud environment. In *Proceedings of the 4th IEEE International Conference on Cloud Computing Technology and Science*, pages 625–628, Dec. 2012.