Term Paper Report

Course Name: *Advances in Operating Systems Design*

Course Code: *CS60038*

Submitted By:

Bratin Mondal
Roll No.: 21CS10016

Datta Ksheeraj
Roll No.: 21CS30037

Department of Computer Science and Engineering

Indian Institute of Technology Kharagpur

Date of Submission: November 26, 2024

Abstract

This term paper explores two advanced technologies in the field of data management and storage: Facebook's Tectonic Filesystem and Google's Monarch. The Tectonic Filesystem is engineered for efficiency at exascale, designed to manage vast amounts of data with exceptional performance and reliability. In contrast, Monarch serves as a globally distributed, highly available time-series database, supporting many of Google's critical applications by enabling low-latency queries and efficient storage of telemetry and monitoring data at an unparalleled scale.

For both technologies, we first examine the motivation behind their design, followed by an analysis of their technical and practical aspects. Finally, we provide our evaluation and insights based on a critical assessment of the respective papers.

Contents

Chapter 1

Facebook's Tectonic Filesystem: Efficiency from Exascale

1.1 Introduction

1.1.1 Overview: What is Tectonic?

Tectonic[9] is Meta's (formerly Facebook) exabyte-scale distributed filesystem, developed to meet the extensive data storage demands of Meta's services. Unlike specialized single-tenant filesystems, Tectonic functions as a general-purpose, multi-tenant solution, capable of serving various types of tenants with diverse data requirements.

A core objective of Tectonic's architecture is to ensure high scalability and efficient resource utilization. Its multi-tenant design enables Meta to consolidate services previously managed in specialized environments, simplifying operations and reducing maintenance costs. Within Tectonic, each tenant can implement specific optimizations, achieving performance levels comparable to custom filesystems while benefiting from shared infrastructure.

Notably, Tectonic is intended solely for Meta's internal services and is not designed for external use. This focus allows Meta to tailor the filesystem to its unique requirements and optimize it specifically for its workloads, without the risks posed by potentially misbehaving external tenants, aside from unintended issues.

1.1.2 Meta's Existing Storage Systems

Meta's storage infrastructure mainly serves two types of tenants: Blob Storage and Data Warehouse.

Blob Storage

Blob Storage is used for large, often unstructured binary data like images, videos, and documents, ranging in size from small files to gigabytes. Blobs require low-latency read and write operations for user-facing applications. Based on access frequency, blobs are categorized into:

- **Hot**: Frequently accessed blobs (e.g., recently uploaded photos).
- **Warm**: Less frequently accessed blobs (e.g., older photos).

Hot Blob Storage

Hot blobs were stored in Haystack[4], a distributed system designed to handle large volumes of data efficiently at scale. Haystack employs a flat, key-value model that minimizes metadata storage by storing blob data contiguously, reducing disk I/O and enhancing retrieval performance. With a replication factor of 3, each blob is replicated across three nodes to ensure fault tolerance and high availability. The core design principle is to eliminate unnecessary metadata, as it is not critical for blob storage, thus reducing disk seeks.

Issues with Haystack

One of the major limitations of Haystack was its high IOPS (Input/Output Operations Per Second) requirements. Originally, Meta's Engineering team designed the system with a replication factor of 3.6, which included 3 for replication and an additional 1.2x for RAID6 redundancy. However, in practice, the devices struggled to meet the IOPS demands, necessitating the addition of more devices to keep up with the load. This issue arose due to advances in disk density, while IOPS capabilities remained relatively unchanged. As a result, the effective replication factor grew to 5.3x, with a significant portion of the storage capacity being underutilized.

Warm Blob Storage

Meta used f4[13], a highly efficient distributed filesystem, to store warm blobs. f4 employs Reed-Solomon erasure coding[10] for fault tolerance, which enables data redundancy without the significant storage overhead associated with traditional replication. This approach ensures both durability and scalability, making f4 well-suited for large-scale warm blob storage.

Issues with f4

Although f4 proved effective for warm blob storage, the large volume of warm blob data required a significant number of devices. However, the issue arose in that these devices also provided high IOPS capacity, which was unnecessary for the relatively low I/O demands of warm blobs. As a result, a substantial portion of the IOPS capacity remained underutilized, leading to inefficiencies in resource allocation.

Data Warehouse

A Data Warehouse stores structured data, such as user activity logs, social graph snapshots, map-reduce outputs, and other analytics data. It is typically accessed via batch processing jobs with read-heavy workloads, infrequent writes, and less emphasis on latency. Parallel processing models often lead to multiple files in the same directory being read together.

Meta's Data Warehouse storage relied on the Apache Hadoop Distributed File System (HDFS)[11], which is designed for large file storage across multiple machines. HDFS uses a single NameNode for metadata and multiple DataNodes for file storage, optimized for large files and streaming reads, making it ideal for batch processing workloads.

Issues with HDFS

Capacity of HDFS was constrained by the NameNode's metadata storage, which could not scale to meet Meta's increasing data demands. To address this, Meta's Engineering team partitioned data across multiple HDFS clusters, each with its own NameNode. This solution introduced operational complexity, resembling a 2D-bin packing problem, with one dimension representing data size and the other throughput requirements. The partitioning strategy proved inefficient, requiring manual load balancing across clusters and making system management and scalability challenging. In short, scaling HDFS became an unsustainable solution for Meta's expanding data needs.

1.1.3 Design Goals for Tectonic

The limitations of existing storage systems highlighted several key design challenges for Tectonic:

- **Scalability to Exabytes**: Tectonic had to efficiently store and serve massive amounts of metadata to meet Meta's exabyte-scale storage needs.
- **Performance Isolation**: Tectonic needed to ensure performance isolation between tenants, preventing resource contention while also allowing tenants to utilize surplus resources from others.
- **Tenant-Specific Optimizations**: Tectonic had to provide flexibility for tenants to implement custom optimizations typically found in specialized filesystems.

1.1.4 Existing Solutions

Several existing storage solutions have been explored to address similar challenges. Federated HDFS[14] and Windows Azure Storage (WAS)[5] focus on merging smaller storage clusters into larger ones, utilizing multiple independent namespaces while sharing data nodes. However, this still leads to a bin-packing problem at the namespace level, where it is difficult to determine where to place specific data. Solutions like Ceph[15] and Flat Datacenter Storage (FDS)[8] hash data objects to determine their locations, increasing the range of hash functions to scale. However, this approach requires updating the hash function during each data relocation, which can be highly costly in large-scale systems.

1.2 Design of Tectonic

1.2.1 Tectonic Architecture

A Tectonic cluster is composed of Chunk Stores, Metadata Stores, and Background Services. A Client Library, tailored for various Tectonic use cases, interfaces with the Metadata and Chunk Stores to store and retrieve data. The architecture of Tectonic is illustrated in Figure 1.1, with additional details provided in the subsequent sections. Tectonic clusters are resilient to host, rack, and power failures at the data center level. For geo-replication, tenants can deploy multiple Tectonic clusters across different data centers.

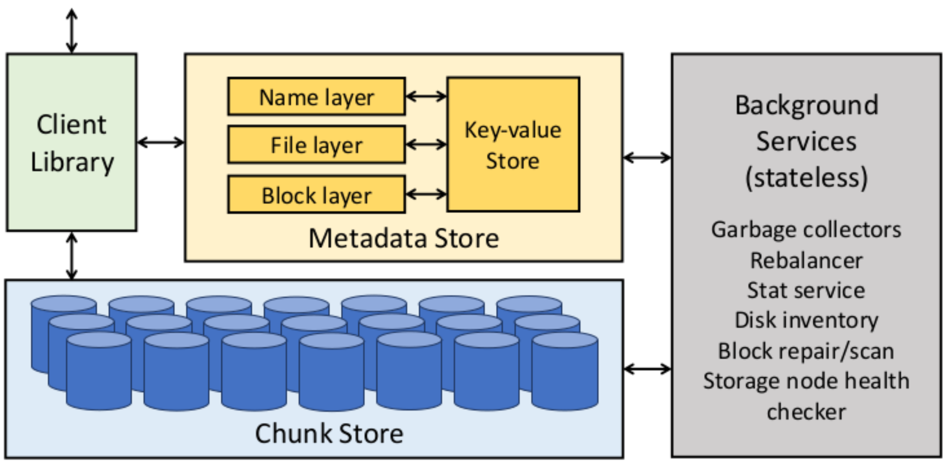

Figure 1.1: Tectonic architecture, with arrows indicating network calls. Filesystem metadata is stored in a key-value store. All components are stateless, except for the Chunk and Metadata Stores.

Chunk Store

In Tectonic, a file is composed of multiple blocks, each containing several chunks, which are the fundamental data storage unit. The Chunk Store is a flat, distributed object store that manages these chunks and is designed to scale linearly with the number of storage nodes, supporting exabytes of data. This abstraction is managed by the Client Library and the Metadata Store, which handle higher-level abstractions like blocks and files.

Chunks are stored as files on storage nodes, each running a local instance of XFS. These nodes are equipped with 36 hard drives for chunk storage and a 1 TB SSD for XFS metadata and caching frequently accessed chunks. The SSD cache is managed using a flash endurance-aware caching library to optimize performance. Tectonic ensures block durability through Reed-Solomon (RS) encoding or replication. Chunks within blocks are distributed across fault domains for fault tolerance, and background services repair damaged or lost chunks to maintain data durability.

Metadata Store

Tectonic addresses the challenge of managing large volumes of metadata by organizing it hierarchically into multiple layers. The Metadata Store is a distributed key-value store, divided into three distinct layers:

- **Name Layer**: Maps directories to the sub-directories and files they contain.
- **File Layer**: Maps files to their associated blocks.
- **Block Layer**: Maps blocks to their corresponding chunks and includes a reverse

mapping from chunks to blocks, which aids in garbage collection.

Unlike traditional directory mappings where a key maps to a list of values, Tectonic stores keys in an expanded format to enable faster updates. For example, if directory `A` contains sub-directory `B` and file `C`, the keys stored will be `(A, B)` and `(A, C)`, rather than a single key `A` mapping to both `B` and `C`. This approach accelerates updates and eliminates the need for locks during the update process. However, listing the contents of a directory requires a prefix scan over the keys.

Another design choice is the use of hash partitioning for the different layers, instead of range partitioning. This method provides better load balancing and helps prevent hotspots in the system, particularly for Data Warehouse parallel processing workloads, as discussed in Section 1.1.2.

At a lower level, the Metadata Store is powered by ZippyDB [3], with each node using RocksDB [12] for storing key-value pairs. Fault tolerance is ensured through replication, utilizing the Paxos consensus protocol [7].

While this design enables scalability and mitigates hotspots, it introduces some challenges. The layered architecture results in increased latency for metadata operations. To address this, Tectonic clients can seal directories, which allows for caching of metadata within the client library. Furthermore, Tectonic does not support atomic cross-directory operations, nor does it provide a direct API for listing the entire contents of a directory. Consequently, clients are required to implement a wrapper around the list API to retrieve the contents of a directory.

Client Library

The Client Library provides a filesystem abstraction using the metadata, operating at the chunk granularity for read and write operations. It enforces **Single Writer Semantics** by issuing a token when a client opens a file for appending. This token must accompany all subsequent metadata updates, ensuring that only the last process to open the file can perform writes.

For multiple-writer semantics, tenants can implement custom serialization protocols on top of Tectonic. Since the filesystem is intended for internal clients, such protocols are manageable and can be handled by the organization's developers using Tectonic for their specific use cases.

Background Services

Background services ensure consistency, durability, and efficient data management in Tectonic. These services maintain metadata consistency between layers, repair lost data, rebalance storage across nodes, and handle rack drains. They also publish statistics on filesystem usage.

The background services operate on one shard at a time, similar to the Metadata Store. Key services include a garbage collector that cleans up metadata inconsistencies (e.g., caused by failed Client Library operations or lazy object deletion), and a rebalancer and repair service that work together to relocate or delete chunks. The rebalancer identifies chunks needing relocation due to events like hardware failures or increased storage capacity, while the repair service handles the actual data movement, operating on a per-Block layer shard and per-disk basis to scale horizontally. The block layer and the rebalancer service work together to ensure that the total number of copysets[6] is maintained, and that chunks are distributed evenly across nodes.

A copyset refers to a group of replicas of a data chunk stored across different nodes. The system ensures that each chunk of data has multiple copies distributed across multiple nodes to protect against node failures. The copyset concept ensures that even in the event of hardware or network failures, data availability and durability are maintained by limiting the total number of distinct copysets in the system. Each chunk belongs to a specific copyset, and the number of replicas (copysets) is dynamically managed based on system requirements such as redundancy levels, storage capacity, and load balancing.

1.2.2 Multitenancy

Tectonic categorizes resources into two types: **non-ephemeral** and **ephemeral**. Non-ephemeral resources change slowly over time and typically do not have dynamic adjustment semantics. An example of this is the storage capacity allocated to a tenant, which is pre-configured and remains static.

In contrast, ephemeral resources change rapidly and require dynamic monitoring and automated adjustments. Examples include storage IOPS capacity and metadata query capacity.

One key challenge is determining the appropriate level at which to manage resources. Providing isolation at the application level is complex due to the difficulty of managing resources across multiple applications. On the other hand, isolation at the tenant level may be too generalized. To address this, Tectonic introduces the concept of `TrafficGroup` and `TrafficClass` for managing resources. A `TrafficGroup` consists of applications with similar storage requirements within a tenant, and each `TrafficGroup` is assigned a `TrafficClass`. Tectonic defines three types of `TrafficClass`: `Gold` (Latency-sensitive), `Silver` (Normal), and `Bronze` (Background services). Tectonic manages resource access both at the global level and per node.

Global Resource Management

The goal of global resource management is twofold: to limit resource usage and to efficiently share surplus resources across tenants. To achieve this, Tectonic employs a modified version of the leaky bucket algorithm, utilizing high-performance, near-real-time distributed counters.

The resource management process works as follows:

- A resource counter is incremented for the corresponding `TrafficGroup` whenever a resource is used.
- If spare capacity is available, it is allocated for use.
- If a tenant exceeds its allocated capacity, the system checks for surplus capacity within other `TrafficGroups` of the same tenant.
- If no surplus capacity is found, the system then checks for spare capacity in other tenants.

When sharing resources between `TrafficGroups`, the request is treated as the `TrafficClass` of the lower-priority group. For example, if a `Gold TrafficGroup` requests resources from a `Silver TrafficGroup`, the request is treated as `Silver`. This prevents excessive increases in high-priority traffic, ensuring that the system remains stable and that many `Gold` requests can still meet their requirements.

Per-Node Resource Management

The aim of per-node resource management is to prevent hotspots and ensure that the latency requirements for high-priority traffic, such as `Gold` class traffic, are met.

To achieve this, a Weighted Round-Robin (WRR) scheduling approach is employed. In this scheme, a request is skipped if its quota has already been used or if granting it would exceed the quota.

Several optimizations are applied to improve resource utilization:

- Lower-priority request classes can yield their turn to higher-priority requests if there will still be enough time to service the lower-priority requests afterward.
- The number of non-`Gold` traffic requests in the queue for a disk is limited if there are pending `Gold` requests.
- Disks may rearrange requests to prioritize higher-priority traffic. If a `Gold` request is pending for a certain threshold amount of time, non-`Gold` requests may be deferred to prioritize the `Gold` request.

It is important to note that resource sharing at the node level is unnecessary, as it has already been handled globally.

Access Control

Tectonic employs a token-based access control mechanism to ensure that only authorized clients can access the system. In this system, each layer generates a token that must be included in subsequent requests to the next layer. The token is piggybacked on the request-reply. This mechanism effectively controls access by verifying the presence of a valid token in each request.

1.2.3 Tenant-Specific Optimizations

As mentioned in Section 1.2.1, the client library communicates directly with the chunk stores at the chunk granularity. This approach enables the optimization of read and write operations for specific tenants.

Data Warehouse Write Optimizations

Data Warehouse storage has read after complete write semantics that is data is only read after the write operation is complete. This allows for optimizations such as:

Asynchronous Writes

To optimize write operations, applications buffer writes until enough data is accumulated to form a block of the desired size. RS-Encoding is then done at the client itself, reducing the amount of data that needs to be transmitted to different nodes. This approach also reduces the number of network calls and disk seeks, enhancing write performance.

Importantly, there is no inconsistency introduced because metadata is only updated after the complete write operation has been successfully finished. This approach guarantees that data consistency is maintained throughout the write process.

Hedged Quorum Writes

Hedged quorum writes aim to improve write latency by sending reservation requests to more nodes than the minimum required for a successful operation. Rather than waiting for the write to complete on all nodes, the system only waits until a majority of blocks have been successfully written, at which point the client is notified.

This method effectively reduces the 99th percentile latency, enhancing the system's responsiveness and providing faster acknowledgment to clients.

Blob Storage Optimizations

Consistent Partial Block Appends

To optimize append operations in blob storage, the system ensures that appends are consistent by waiting until the quorum size of appends is reached. This ensures data integrity and reduces unnecessary writes. Additionally, only the block creator is allowed to perform append operations, maintaining consistency and preventing data corruption.

Once an append operation is completed, the metadata is updated with the post-append block size and its corresponding checksum, ensuring consistency and reliability in the storage system.

Reencoding Blocks for Storage Efficiency

Reencoding blocks for storage efficiency is made possible by the Client Library-driven design. Unlike traditional file systems, which typically require preconfiguration for file storage, this design allows more flexibility in optimizing storage. Specifically, blocks are reencoded from replicated formats to Reed-Solomon (RS) after a certain amount of time has passed. This process reduces the storage overhead associated with replication, enhancing storage efficiency and reducing costs.

1.3 Tectonic in Production

Meta's Tectonic Filesystem supports diverse workloads, including blob storage and data warehousing at exabyte scale. However, certain tenants like Key-Value stores, Deployment Management Systems, and legacy systems don't use Tectonic due to specialized storage needs. Some services access Tectonic indirectly through major tenants, such as the data warehouse, to avoid the cost of developing individual client libraries.

1.3.1 Exabyte-Scale Multitenant Clusters

Tectonic is designed for efficient operation at an exabyte scale, supporting diverse workloads under a unified storage system. For example, a typical Tectonic cluster manages approximately 1250 PB of data, representing about 70% of its capacity, and handles around 10.7 billion files and 15 billion blocks. This scale demonstrates Tectonic's ability to support high-volume, varied workloads effectively.

1.3.2 Efficiency from Storage Consolidation

By consolidating blob storage and data warehouse services, Tectonic enhances resource utilization and manages workload spikes more efficiently. For instance, surplus IOPS from blob storage can be redirected to support bandwidth-intensive spikes from data warehouse workloads. This consolidation reduces the need for separate, overprovisioned systems, optimizing disk usage and lowering overall storage costs.

1.3.3 Metadata Management and Load Balancing

Tectonic manages metadata load spikes efficiently by using load-balancing techniques within its metadata store to maintain high performance. Around 1% of Name layer shards may hit peak queries per second (QPS) during load surges, but Tectonic handles these spikes with retry mechanisms to sustain responsiveness. Additionally, Tectonic co-designs with the data warehouse to mitigate metadata hotspots from frequent, simultaneous file access. For example, the `list-files` API provides both file IDs and names, allowing compute engines to distribute file IDs directly to worker nodes, reducing directory query loads on the metadata store.

1.3.4 Design Trade-offs

Tectonic's design focuses on simplicity and efficiency but involves certain trade-offs.

For **reconstruction load management**, Tectonic employs contiguous Reed-Solomon (RS) encoding, allowing most reads to be single-disk IO. However, reconstruction reads triggered by failures require higher IO, leading to "reconstruction storms." Tectonic mitigates this by capping reconstructed reads at 10%, balancing performance with resource usage according to the cluster's workload.

Direct access to storage nodes improves efficiency by avoiding extra network hops, although it adds complexity since Client Library bugs can impact application binaries. Latency-sensitive remote requests are routed through a stateless proxy, optimizing cross-datacenter data access.

Metadata latency and partitioning introduce additional trade-offs. Metadata is stored in a sharded key-value store, which increases latency compared to in-memory storage, prompting optimizations like parallelized file renames. The hash-partitioned design limits recursive directory listing and aggregate space usage queries, which are handled through periodic space aggregation.

These trade-offs enable Tectonic to support scalable, exabyte-level storage while delivering robust performance for diverse workloads.

In summary, Tectonic effectively addresses Meta's large-scale storage requirements through a balance of operational simplicity and resource efficiency, allowing it to replace multiple specialized storage systems and support exabyte-scale, multitenant environments.

1.4 Evaluation

As Tectonic is designed for Meta's internal use, it does not implement many of the semantics required by clients. Instead, it leaves these responsibilities to developers, who are expected to implement them at a higher level while using Tectonic. This design choice allows Tectonic to be more efficient and flexible. However, there are some design choices that are questionable and not explicitly clarified in the paper.

- **Token-based single writer:** The approach of using a token to ensure only one writer can write at a time could lead to deadlock situations if writers alternatively attempt to write to the same file. The paper does not explain how this situation is handled, though it is likely addressed at a higher level by the developers. An alternative could have been to use a lease mechanism, where the writer holds the lease for a fixed time, with forced relinquishment if the lease is not renewed. This would introduce some delay in the event of a writer crash but would be more robust.
- **Partial Block Quorum Appends:** The paper describes blob storage tenants waiting for acknowledgment from a quorum of storage nodes, claiming that this provides low latency. However, the handling of writes that are not acknowledged

by nodes outside the quorum is not discussed. It is unclear whether this information is stored in the metadata, allowing clients to read only from the nodes within the quorum, or if the client can read some stale data in the interest of low latency. The relaxed consistency model would likely improve performance, but if consistency is a concern, the system should have a mechanism to handle partial writes.

One option would be to maintain a list of stale chunks of nodes and prevent clients from reading from them. While this would add complexity to the system, it would offer better consistency. It would also allow later writes to be processed even if earlier writes are incomplete at some nodes. This approach would enable clients to send requests to all nodes and wait for a quorum response. However, maintaining an individual list of stale chunks for each node would introduce additional complexity and overhead.

Another approach would be to serialize writes at each node, maintaining only a list of stale nodes rather than stale chunks. While this avoids the complexity of tracking stale chunks, it would introduce latency since further writes could only proceed on non-stale nodes. This would reduce the effectiveness of partial block quorum appends, as writes would be blocked if any node is stale. This approach would add additional latency to the system.

The exact mechanism used by Tectonic is not described in the paper, so it is unclear how the system handles partial writes and stale data. The paper should have provided more details on this aspect of the system.

Chapter 2

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

2.1 Introduction

Google operates one of the largest and most complex infrastructures in the world, with millions of servers across data centers worldwide. Monitoring and managing performance metrics across this massive infrastructure requires a database capable of handling petabytes of data and millions of queries per second.

Google uses various types of databases, applications each suited for specific tasks. There should be monitoring system over all these systems. Generally a TSDB (Time-Series Database monitoring system is used which captures events along with time stamps)

2.1.1 Monarch as TSDB

A time-series database (TSDB) is a specialized database optimized to handle, store, and analyze time-series data, which consists of data points indexed by time. This type of data typically represents how a system, process, or measurement evolves over time, with each entry or observation paired with a timestamp

At Google, the time-series database (TSDB) landscape has evolved significantly over the years, primarily driven by internal projects and open-source contributions to handle the massive scale of data generated by Google's infrastructure

The initial TSDB developed and used was Borgmon which was replaced with Monarch for reasons discussed later in the report

So currently Monarch is the globally-distributed in-memory time series database monitoring system in Google

2.2 The Problem

2.2.1 Arrival of Borgmon

Initial Problem

Google's infrastructure comprised numerous clusters, each hosting thousands of servers running various applications. The complexity and scale of these clusters made it challenging to monitor system health effectively. Existing monitoring solutions lacked the ability to provide real-time insights into the status of individual servers, services, and applications. This often resulted in:

- Need to get a time-based collection of data.
- Difficulty in identifying failing or underperforming nodes.
- Slow response times to incidents due to limited visibility.
- Overwhelming volumes of alerts, many of which were not actionable.

Borgmon Solution[1]

Borgmon was designed as a time-series database to provide comprehensive cluster monitoring by implementing:

- **Real-time Monitoring:** Borgmon enabled real-time tracking of system health, resource usage, and performance metrics across all nodes in a cluster.
- **Centralized Data Collection:** It centralized data collection from all servers, allowing for a unified view of the health of the entire infrastructure.
- **Intelligent Alerting:** The system incorporated intelligent alerting mechanisms that reduced noise by prioritizing alerts based on severity and context, thus allowing engineers to focus on critical issues.

2.2.2 Arrival of Monarch[2]

Google shifted from Borgmon to Monarch to address the growing complexity and scale of its monitoring needs, especially as its infrastructure became more distributed and global. Below are the key reasons behind this shift:

- **Scalability and Performance**

  - Borgmon was originally designed to monitor Borg, Google's cluster manager, primarily handling a centralized monitoring model.
  - As Google's infrastructure expanded, including global data centers and cloud-based services, Borgmon struggled to scale effectively also making it difficult for engineers to run Borgmon reliably.

- **Semantic ambiguity**
  - Each team created their own Borgmon instance creating their own configuration and variables creating semantic ambiguity
- **No support for Complex Distribution types**
  - Some monitoring actually requires complex histograms from which other statistical information is drawn out. This is not supported by Borgmon i.e. We cant store whole of histogram in the records we store

2.3 Motivation

2.3.1 Core Solution Ideas

Keeping in mind the various problems or limitations with Borgmon mentioned in Chapter**??**, the solutions proposed for each are trivial. This slowly led to the foundation of Monarch, the now widely used replacement of Borgmon.

The solutions for each of the problems were as mentioned below:

- **Scalability and Performance:** The system instead of having a decentralized architecture, now should have a unified one. This now adds on complexity to development of such architecture hence is a good trade off
- **Semantic Ambiguity** Simple solution exists for this as we just maintain a uniform rule book all over as we already have a central monarch which we call global monarch
- **Complex data types** We have to completely change the record or row which we store as monitoring data. The change in format which we describe in future allows complex distributions such as histogram also to take from the monitored system
- **Other Additional Features** Monarch as we shall see have implemented load balancing, data compression which further enables to use the system in a more reliable way. These were also there in Borgmon but its difficult to put up these in a centralised system again

2.4 Implementation

2.4.1 Components

Monarch's architecture includes components categorized by their primary functions: those responsible for storing state, those dedicated to data ingestion, and those focused on query execution.

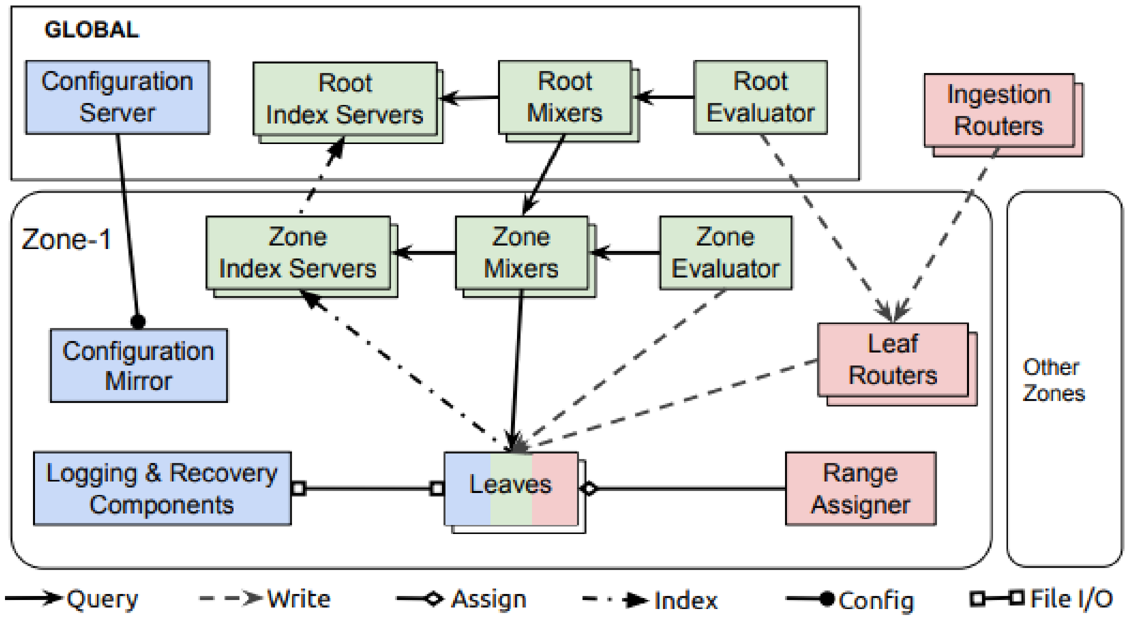

Figure 2.1: Overview of Monarch [2]

State-Holding Components

- **Leaves** serve as the primary storage for monitoring data, keeping it in an in-memory time series database.
- **Recovery logs** store monitoring data on disk, mirroring the data held in leaves. Over time, this data is moved into a long-term time series repository.
- **Global configuration server and zonal mirrors** manage configuration data, which is stored in Spanner databases [2].

Data Ingestion Components

- **Ingestion routers** direct data to leaf routers within the correct Monarch zone, relying on information embedded in time series keys for accurate routing.

- **Leaf routers** receive data bound for a specific zone and forward it to the leaves for storage.
- **Range assigners** handle the distribution of data across leaves, balancing the load within each zone.

Query Execution Components

- **Mixers** divide queries into smaller sub-queries routed to and processed by leaves, then combine the results. Queries can be initiated at the root level (handled by root mixers) or at the zone level (handled by zone mixers). Root-level queries require both root and zone mixers.
- **Index servers** provide indexing for each zone and leaf, guiding the distributed execution of queries.
- **Evaluators** periodically execute standing queries (see Section 5.2) by dispatching them to mixers, then write the results back to leaves.

2.4.2 Data Storage

Monarch organizes monitoring data into tables that store time series, which are sequences of values recorded over time. Each table includes two main types of columns:

- **Key columns:** These define unique identifiers for each time series, such as the specific machine or service being tracked.
- **Value column:** This stores the historical data points (e.g., memory usage or latency) recorded over time for each time series.

Key columns, also referred to as fields, have two sources: targets and metrics.

Targets (Monitored Entities)

- A target is the system or component that generates data, like a server, application, or process.
- Monarch links each time series to its specific source target, identifying the entity from which the data originates.
- Each target follows a schema (template) with specific fields, such as user, job, cluster, or task number for tasks in a computing cluster.
- The location field within each target helps Monarch store data near its origin (for instance, data from a specific cluster is stored in the closest zone).

- Time series data from the same target are grouped, ensuring related data is stored together and is easier to query.
- Target ranges organize targets in alphabetical order to allow efficient data distribution and load balancing, which simplifies group queries.

Metrics (What We're Measuring)

- A metric is a specific measurement related to a target, such as the number of requests a server processes or its memory usage.
- Each metric has its own schema with particular fields, such as the type of service or command for an RPC request.
- Metrics can be various data types, including numbers, strings, or distributions (groups of values that allow for calculations of averages, percentiles, etc.).

2.4.3 How Data Storage Happens

The data collection path in Monarch can be broken down into four main steps:

- **Client to Ingestion Router**
  - A client application sends time series data to a nearby ingestion router. Monarch's ingestion routers are spread across clusters, allowing data to be gathered close to its origin, thus reducing latency.
  - Clients frequently utilize Monarch's instrumentation library, which regulates the data write frequency to comply with Monarch's retention policies.
- **Ingestion Router to Leaf Router**
  - Each ingestion router extracts the location field of the target to determine the appropriate zone for the data, organizing it by geographic or logical regions.
  - After identifying the target zone, the ingestion router forwards the data to a leaf router within that zone. Zone mappings are dynamically updated.
- **Leaf Router to Leaf Node**
  - Within the assigned zone, the leaf router directs data to specific leaf nodes. These leaf nodes manage target ranges, with data sharded lexicographically according to target strings.
  - Each leaf router keeps an updated range map that indicates which leaf node is responsible for each target range. This map is periodically refreshed by updates from leaf nodes, ensuring uninterrupted data collection even if the

range assigner experiences temporary issues.

- **Data Storage on Leaf Nodes**
  - Upon reaching the assigned leaf node, data is stored in an in-memory time series store and recovery logs. This in-memory store is optimized for efficient handling of timestamp sequences, reducing memory usage.
  - Data encoding methods, such as **delta encoding** and **run-length encoding**, are applied to efficiently manage various time series types, including complex structures like tuples.

2.5 Organization of Data and Load Balancing

2.5.1 Data Organization and Load Balance

Intra-zone load balancing in Monarch is a strategy used to ensure efficient data storage and distribution across leaf nodes within a zone, minimizing load and optimizing performance. Here's an in-depth look at how this process works:

- **Schema and Lexicographic Sharding**
  - **Data Organization:** Data is organized in a table schema consisting of a target schema and a metric schema.
  - **Sharding by Target:** Only the key columns associated with the target schema are used for sharding data lexicographically. This approach groups all time series for a single target together, which minimizes the ingestion fanout, allowing a single message to carry data for multiple metrics for the same target. As a result:
    - Each target's data is only sent to a few (up to three) leaf replicas.
    - This setup scales the zone horizontally by adding more leaf nodes and simplifies query processing by limiting it to a smaller subset of leaf nodes.
  - **Intra-Target Joins:** Common joins between metrics for the same target can be processed within the leaf node, reducing query complexity and making them faster.
- **Replication Flexibility**
  - **Replication Policy:** Monarch allows users to select the number of replicas per target (ranging from 1 to 3), enabling a trade-off between availability and

storage cost.

  - **Granularity Control:** Users can choose to retain different levels of data detail for each replica, such as retaining only a fine-grained view on one replica and a coarser view on others.
  - **Individual Assignment of Replicas:** Each target range replica is assigned independently to avoid overloading a single leaf node, ensuring no leaf node holds multiple replicas of the same range.
  - **Distribution Across Failure Domains:** Leaves are distributed across clusters or failure domains. The range assigner ensures replicas of the same range are stored in different domains to enhance fault tolerance.

- **Range Assigner and Load Balancing**
  - **Load Monitoring and Adjustment:** The range assigner actively balances the load by moving, splitting, or merging ranges as needed based on CPU load and memory usage across leaf nodes.
- **Ensuring Data Availability**
  - **Simultaneous Data Collection:** During the transfer process, both the source and destination leaves temporarily collect and log data for range R to avoid any data loss and ensure continuous availability.(**Interesting optimisation**)
  - **Direct Updates from Leaves:** Leaves keep leaf routers informed about range assignments, rather than relying on the range assigner for updates(**Interesting optimisation**). This:
    * Ensures data integrity, as leaves are the main storage units.
    * Allows the system to continue working smoothly if the range assigner has a temporary failure.

2.6 Querying

2.6.1 Query Execution

In Monarch, there are two types of queries:

Ad hoc Queries These are one-time queries from users outside the system.

Standing Queries These are periodic queries whose results are stored back in

Monarch. Standing queries are used to generate alerts or pre-process data for faster and more cost-effective access later. They can be processed at either the regional zone level or the global root level, based on the type of data and query requirements. Most standing queries are handled at the zone level, making them more efficient and resilient to network issues.

Query Tree and Execution Levels

Queries are processed in a three-level hierarchy:

- **Root Mixer:** Receives the query and distributes it to zone mixers.
- **Zone Mixers:** Send the query to relevant leaf nodes based on an index.
- **Leaf Nodes:** Process the data closest to the source.

Each query is executed only at the necessary levels, which optimizes processing. The root node checks security, access control, and can rewrite queries for efficiency. Lower levels (leaves and mixers) stream data upwards, where higher levels combine results and manage the flow of data with rate control.

Replica Resolution

Data is often replicated, and replicas may vary in quality (e.g., completeness and time coverage). For accurate results, zone mixers resolve which replica is best based on quality criteria and assign target ranges to specific leaves for processing.

User Isolation

Monarch is a shared system, so resources are divided among users. Each user's queries are limited in memory and CPU through cgroups, ensuring fair use.

Query Pushdown

Monarch minimizes data transfer by processing as much of the query as possible at the lowest level. This is called "query pushdown" and improves speed by:

- Reducing the amount of data transferred to higher levels.
- Allowing more concurrent processing.

For example, if a query only involves data from one zone, it can be fully processed there without needing root-level involvement. This pushdown approach makes 95% of standing queries complete within the zone, which also prevents cross-region data traffic and reduces latency.

Specific Operations at Lower Levels

Some queries, like `group by` or `join`, are pushed to the leaf level when possible, where they process data within the smallest target ranges. For instance, if data for a specific target (like a task) remains within one leaf, that leaf can complete the operation, which reduces work for higher-level nodes.

2.6.2 Result Aggregation

- Leaf Nodes process data closest to the source. They handle their assigned "target range" and perform basic filtering and aggregation to reduce data before sending it up.
- Zone Mixers gather results from multiple leaf nodes in their region. They combine and further aggregate data, then send only summarized information to the root mixer.
- Root Mixer collects data from zone mixers. It completes any remaining aggregation, checks security, and applies final optimizations.

2.7 Evaluation

While Monarch prioritizes high availability and partition tolerance, the trade-off with data consistency could lead to stale reads. This seemed more bold decision to me, instead we should be using better consistency model which does different things based on situation.

For this I thought of implementing a Reinforcement Learning Model which slowly learns the data it handles and performs accordingly.

Also user was given choice to choose the number of replicas and level of granularity. But when scalability is considered and lot of users operate at same time they might over estimate the usage which unnecessarily creates extra load if all created maximum number of replicas.

For this the control given to users must be looked into again, there must be control mechanisms asking users about the data they are going to handle and based on that the system should have some percentage of controlling. Even complete control to the system itself is a waste.

Bibliography

[1] Borgmontsdb. `https://sre.google/sre-book/practical-alerting/`.

[2] Colin Adams, Luis Alonso, Ben Atkin, John P. Banning, Sumeer Bhola, Rick Buskens, Ming Chen, Xi Chen, Yoo Chung, Qin Jia, Nick Sakharov, George T. Talbot, Adam Jacob Tart, and Nick Taylor, editors. *Monarch: Google's Planet-Scale In-Memory Time Series Database*, 2020.

[3] M. Annamalai. Zippydb - a distributed key-value store. `https://www.youtube.com/embed/ZRP7z0HnClc`, 2015.

[4] Doug Beaver, Sanjeev Kumar, Harry C. Li, Jason Sobel, and Peter Vajgel. Finding a needle in haystack: Facebook's photo storage. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, Vancouver, BC, October 2010. USENIX Association.

[5] Brad Calder, Ju Wang, Aaron Ogus, Niranjan Nilakantan, Arild Skjolsvold, Sam McKelvie, Yikang Xu, Shashwat Srivastav, Jiesheng Wu, Huseyin Simitci, Jaidev Haridas, Chakravarthy Uddaraju, Hemal Khatri, Andrew Edwards, Vaman Bedekar, Shane Mainali, Rafay Abbasi, Arpit Agarwal, Mian Fahim ul Haq, Muhammad Ikram ul Haq, Deepali Bhardwaj, Sowmya Dayanand, Anitha Adusumilli, Marvin McNett, Sriram Sankaran, Kavitha Manivannan, and Leonidas Rigas. Windows azure storage: A highly available cloud storage service with strong consistency. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles (SOSP '11)*, pages 143–157, 2011.

[6] Asaf Cidon, Stephen Rumble, Ryan Stutsman, Sachin Katti, John Ousterhout, and Mendel Rosenblum. Copysets: Reducing the frequency of data loss in cloud storage.

In *2013 USENIX Annual Technical Conference (USENIX ATC 13)*, pages 37–48, San Jose, CA, June 2013. USENIX Association.

[7] Leslie Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, 1998.

[8] Ed Nightingale, Jeremy Elson, Jinliang Fan, Owen Hofmann, Jon Howell, and Yutaka Suzue. Flat datacenter storage. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, October 2012.

[9] Satadru Pan, Theano Stavrinos, Yunqiao Zhang, Atul Sikaria, Pavel Zakharov, Abhinav Sharma, Shiva Shankar P, Mike Shuey, Richard Wareing, Monika Gangapuram, Guanglei Cao, Christian Preseau, Pratap Singh, Kestutis Patiejunas, JR Tipton, Ethan Katz-Bassett, and Wyatt Lloyd. Facebook's tectonic filesystem: Efficiency from exascale. In *19th USENIX Conference on File and Storage Technologies (FAST 21)*, pages 217–231. USENIX Association, February 2021.

[10] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[11] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, pages 1–10, 2010.

[12] Facebook Open Source. Rocksdb. `https://rocksdb.org/`, 2020.

[13] Muralidhar Subramanian, Wyatt Lloyd, Sabyasachi Roy, Cory Hill, Ernest Lin, Weiwen Liu, Satadru Pan, Shiva Shankar, Sivakumar Viswanathan, Linpeng Tang, and Sanjeev Kumar. f4: Facebook's warm blob storage system. In *USENIX Symposium on Operating Systems Design and Implementation*, 2014.

[14] The Apache Software Foundation. Hdfs federation, 2019.

[15] Sage A. Weil, Scott A. Brandt, Ethan L. Miller, Darrell D. E. Long, and Carlos Maltzahn. Ceph: A scalable, high-performance distributed file system. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 06)*, Seattle, WA, November 2006. USENIX Association.