Goku: A Schemaless Time Series Database for Large Scale Monitoring at Pinterest

Monil Mukesh Sanghavi
Pinterest Inc.
msanghavi@pinterest.com

Ming-May Hu
Pinterest Inc.
mingmayhu@pinterest.com

Zhenxiao Luo
Pinterest Inc.
zluo@pinterest.com

Xiao Li
Pinterest Inc.
xiaoli@pinterest.com

Kapil Bajaj
Pinterest Inc.
kapil@pinterest.com

ABSTRACT

Engineers rely heavily on observability tools to monitor their business and system metrics and set up alerting on it. A reliable and efficient monitoring system is very important for development velocity. In this paper, we introduce Goku, a time series database (TSDB) we built from the ground up at Pinterest. Over the years, we have studied user patterns and common requests to constantly evolve Goku to store and serve the use cases at Pinterest with high efficiency and reduced costs. At its core, Goku uses tiered storage to store new and frequently queried metrics data in memory while leveraging solid state drive (SSD) and hard disks (HDD) for older data. Goku aggregates metrics data at write time while also rolling up datapoints with lower time granularity to low latency to certain use cases. Goku also supports modifying configurations on metrics data like time to live (TTL), rollup granularity, backfilling capability, etc. Using multiple replicas and AWS S3 as backup, Goku is highly available and fault tolerant.

PVLDB Reference Format:
Monil Mukesh Sanghavi, Ming-May Hu, Zhenxiao Luo, Xiao Li, and Kapil Bajaj. Goku: A Schemaless Time Series Database for Large Scale Monitoring at Pinterest. PVLDB, 18(2): 503 - 515, 2024.
doi:10.14778/3705829.3705862

1 INTRODUCTION

Pinterest, an industry leader in visual discovery, faces extensive system monitoring requirements due to its global user base and diverse services, such as Pins, Boards, and Recommendations. Various teams are tasked with running these services, the back-end infrastructure to support these services and ensure their seamless performance. They need to monitor a growing collection of heterogeneous entities, including servers, virtual machines, and containers, spread across multiple availability zones. Metrics from these entities must be collected, stored in time series, and queried to support key use cases like identifying and notifying when services are under performing, displaying graphs that show the state and health of the services, diagnosing problems and exploring performance and resource usage etc.

Proceedings of the VLDB Endowment, Vol. 18, No. 2 ISSN 2150-8097.
doi:10.14778/3705829.3705862

Initially, Pinterest used OpenTSDB [7] and HBase [1] for its time series data management. OpenTSDB offers an open-source, scalable time series database built on top of HBase, a distributed and scalable NoSQL database based on SSD/HDD storage. During its time of operation, the usage of OpenTSDB and HBase scaled considerably due to rapid growth in monitoring traffic. However, this expansion revealed several critical limitations, explained in subsection 7.1, with some of them being:

- **High Operation Costs:** The infrastructure was cost intensive.
- **Performance Issues:** Slow query response times were experienced due to serving from secondary storage and single machine aggregation, which led to to poor monitoring experiences and delayed alerts.
- **Significant Maintenance Effort:** Frequent manual intervention was required for maintaining the HBase clusters costing substantial engineering hours.

These challenges highlighted the need for a more efficient and scalable solution, leading to the development of Goku. In this paper, we present the architecture of Goku, which fulfills the following requirements. Initial requirements at time of development (2018/2019):

(1) OpenTSDB feature parity to seamlessly integrate into the existing observability stack. It should allow for easy addition of tags and continue to maintain a schemaless architecture.
(2) In-memory storage required for storing the last 24 hours of data to ensure rapid query responses and timely alerts.
(3) Secondary storage must be supported with data retention for up to 1 year, replacing OpenTSDB and HBase. This includes storing raw metrics data for 24 days and rolled-up data for the remaining period.
(4) The system should support a high ingestion rate of millions of datapoints per second and accommodate storing 2-3 billion time series per day in the in-memory database, with an estimated growth of 100% per year.
(5) Low query latency: Achieve p99 latency of 5 seconds for queries on in-memory time series data and 10 seconds for queries on data stored in secondary storage.
(6) Architecture should support high availability with multiple replicas each in different availability zones.
(7) Cost efficiency with scalability: Design for cost efficiency while maintaining scalability to handle growing data volumes. Over this paper, we will highlight the features or decisions which led to achieving lower costs.

(8) Fault tolerance with no data loss: Incorporate fault tolerance mechanisms, with routing capabilities to redirect to healthy clusters if a host fails. No data loss is expected once data is ingested in in-memory storage.

Additional requirements over time:

(1) Flexible metric configurations: Support multiple metric configurations, such as TTL and data granularity, with seamless transition of metrics between configurations.
(2) Support for expensive queries: Enable support for complex queries through pre-aggregation and pagination techniques.

This paper presents several novel contributions that enhance the functionality and efficiency of Goku:

- **Ingestion Using Apache Kafka [21] With Batching:** Goku's data ingestion mechanism is tightly integrated with Apache Kafka along with asynchronous logging. This approach ensures high write throughput, fault tolerance, and active health monitoring at shard level, which in turn aids in intelligent routing.
- **Cost Efficient Indexing:** Through efficient string handling techniques, Goku achieves cost-effective storage of indexes, reducing overall storage costs and enhancing query performance.
- **Namespace:** Goku supports multi-tenancy through the concept of namespaces. This feature allows multiple metric configurations such as TTL, rollup granularity, raw data retention, sharding strategy, backup locations, and ingestion topics. It also facilitates seamless movement of metrics between configurations, offering flexibility and operational efficiency.
- **RocksDB [16] Based Secondary Storage:** The secondary storage layer leverages RocksDB for serving of metrics data older than a day.

In section 2 and section 3, we start by discussing the data and query models of Goku respectively. This is followed by the system architecture in section 4. Here, we dig deeper into how Goku stores the data and achieves fault tolerance with high availability. Following this, in section 5, we shed light on the features we have implemented in Goku that help users at Pinterest immensely. In section 6, we share our learnings and cover some important changes we made in the architecture to reduce cost and improve availability of Goku. In subsection 7.1, we discuss in-production evaluation of Goku against OpenTSDB and in subsection 7.2, we share the results of evaluating Goku against other TSDBs. In section 8 , we theoretically compare Goku with other TSDBs and in section 9, we share insights about our production data. In section 10, we summarize a few future projects.

2 DATA MODEL

A time series in Goku is a key-value pair, where the key comprises a metric name and multiple tag value pairs. Tags provide additional context about the time series. For example, in Table 1, the time series tracking CPU usage of a host named "abc" in cluster "kv", availability zone "east-1a" and running "ubun-1" os would have the metric name "cpu" and tags as "host", "os", "az" and "cluster". Goku is schemaless as the tags can be added anytime with no schema definitions needed beforehand. Additionally, the tags are not necessary for queries to execute as explained in section 3.

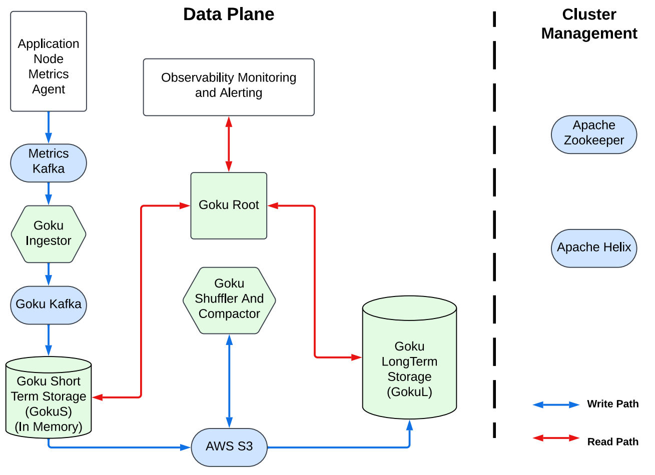

Data Plane
Cluster Management
Application Node Metrics Agent
Observability Monitoring and Alerting
Metrics Kafka
Goku Root
Goku Ingestor
Goku Shuffler And Compactor
Goku Kafka
Goku Short Term Storage (GokuS) (In Memory)
Goku LongTerm Storage (GokuL)
AWS S3
Apache Zookeeper
Apache Helix
Write Path
Read Path

Figure 1: High level architecture overview of Goku

Table 1: Five unique time series based on metric names and tag=value combinations. Id for reference only

Id	Metric	Tag 1	Tag 2	Tag 3	Tag n
1	cpu	host=abc	cluster=kv	az=east-1a	os=ubun-1
2	cpu	host=pqr	cluster=ml	az=east-1a	os=ubun-1
3	cpu	host=xyz	cluster=ml	az=east-1a	os=ubun-1
4	mem	host=abc	cluster=kv	az=east-1a	os=ubun-1
5	mem	host=xyz	cluster=ml	az=east-1a	os=ubun-1

Table 2: Example datapoints for time series id 1 in table 1 consisting of timestamp in Unix time and value in double precision

	datapoint 1	datapoint 2	datapoint 3	datapoint 4
Time	1704153600	1704153660	1704153720	1704153780
Value	3.0	4.2	5.2	4.0

In Goku, the value of a time series, referred to as datapoints, is a list of timestamp-value pairs as shown in Table 2. These timestamps are in Unix time, and the values follow 8 byte double precision format. To optimize retrieval and cost, datapoints are divided into separate buckets rather than stored as a continuous list. This is explained in more detail in subsection 4.4.

Multiple hosts can emit time series data for the same metric name, such as CPU, memory, and disk usage, with each host-specific detail included in the tags. Each time series is uniquely identified by its metric name and tag value pairs. The cardinality [6] of a metric is the total number of unique time series for that metric. For example, in Table 1, the cardinality of "cpu" is 3, and "mem" is 2. High cardinality can negatively impact query performance.

3 QUERY MODEL

Goku uses Apache Thrift [24] for query RPC and responses. For seamless migration from OpenTSDB, Goku supports all features that OpenTSDB querying [9] offers. To retrieve a time series, the user needs to specify the following fields:

Metric Name And Tags: A combination of the metric name and/or tag value pairs.

Time Range: Start and end times of the query. For example, as shown in Table 1 and Table 2, a query for the metric "cpu" for "host=abc" from Unix time 1704153600 to 1704153720 will return the datapoints [3.0, 4.2, 5.2].

The user can also specify additional fields to discover aggregated or downsampled views of their metrics. They can also specify filters in their tag values to narrow down candidate time series.

Aggregation: Aggregation combines multiple time series into a summarized form. Supported aggregations include sum, max, min, p99, p50 (median), mean, and count.

Downsampling: Downsampling reduces data granularity by aggregating datapoints over larger intervals. It needs an interval and an aggregating function.

Rate Option: The rate option calculates the rate of change of a metric over time. If enabled, the query returns differences in successive datapoints instead of raw values, which helps identify trends and anomalies.

Filters: Users can apply filters [10] to tag values such as: wildcard, regular expressions [22], explicit (tag1=value1), and operator (tag1=value1 and tag2=value2), or operator (tag1=value1 or value2), and exclusion operator (tag1 = !value1).

4 SYSTEM ARCHITECTURE

4.1 System Overview

Figure 1 shows the overall architecture of Goku. Goku is not a single cluster, but a collection of independent components including:

Goku Short Term (GokuS): a set of storage nodes which hold the last 24 hours of data in-memory for fast retrieval.

Goku Long Term (GokuL): a set of storage nodes which use persistent storage (SSD or HDD) for data older than 24 hours.

Goku Ingestor: consumes data from the applications and prepares it for GokuS.

Goku Compactor And Shuffler: prepares data for GokuL as per GokuL storage format.

Goku Root: a set of compute nodes that serve as a query endpoint for observability clients. It routes the queries to GokuS and/or GokuL storage nodes.

For high availability, we maintain 3 replicas of GokuS and GokuL in different availability zones. We maintain hosts in different availability zones for root, ingestor, compactor and shuffler.

Each of these nodes operates within its own cluster, ensuring physical separation of functionalities. This separation is crucial for several reasons:

- **Avoiding Noisy Neighbor Issues:** GokuS is a critical component as it serves the most recent time series data required for alerting. Issues in other components, like the shuffler/compactor or GokuL, which manage long-term metrics and have less stringent SLAs, should not affect the performance of GokuS.
- **Optimized Hardware Selection:** Physical isolation allows us to study the operational patterns of each node type and select the most suitable hardware for their specific needs: GokuL requires storage-optimized machines due to its heavy reliance on secondary storage. GokuS benefits from memory-optimized machines to efficiently handle in-memory data. Goku Root is best served by compute-optimized machines as it primarily functions as a routing layer and performs top-level aggregation as and when needed.
- **Deployment And Restarts:** Isolation facilitates smoother deployments and handling of cluster restarts. For example, deploying a non-trivial change in the Compactor or Goku Root, both of which are stateless services, should not trigger the recovery routine of GokuS. This is especially important because restarting GokuS involves recreating the in-memory footprint of the metrics data, which is an expensive and time-consuming operation.

By maintaining separate clusters for each of these node systems, Goku can optimize performance, reduce operational costs, and enhance the overall stability and efficiency of the platform.

4.2 Sharding Scheme

Sharding horizontally partitions data across multiple storage nodes to address performance and scalability challenges when handling large volumes of time series data. Goku employs a two-layer sharding strategy:

- **Shard-Group Assignment:** First, the metric name of a time series is hashed to determine the shard-group.
- **Shard Assignment:** Next, the full time series name (metric name + tag key-value pairs) is hashed to determine the specific shard within the shard-group.

For instance in Table 1, multiple time series with the same metric name (example: metric = "cpu") will be assigned to the same shard-group but may reside in different shards within that group. The advantages of two-layer sharding strategy:

- **Mitigating Noisy Neighbor Issues:** The sharding strategy helps isolate the impact of spammy metrics which create abnormally high time series and/or datapoints enough to spike the Goku resource usage and cause undefined behavior. If a spammy metric is introduced in the write path, it will be confined to within its shard-group. The other shard groups won't be affected and hence observability for only a select subset of metrics will be impacted.
- **Reduced Cross-AZ Failovers:** The sharding scheme reduces the amount of cross-AZ failovers, which are typically discouraged due to their high latency and infrastructure costs. Without the two-layer sharding, a single shard could receive queries for all metrics, increasing the likelihood of cross-AZ failovers if a host fails. With Goku's strategy, queries are more effectively confined within appropriate shard-groups and AZs, reducing the need for cross-AZ traffic and its associated latency and cost.

4.3 Cluster and Shard Management

Goku uses Apache Helix [17] for cluster and shard management and monitoring as shown in Figure 1. In Apache Helix, there are three roles:

- **Participant:** The nodes (Goku nodes) that actually host the distributed resources.
- **Spectator:** The service that simply observes the current state of each participant and takes necessary actions which, in our case,

is to create a cluster shard map. The cluster shard map is a map of the storage nodes to the shards they own.

- **Controller:** The node (running as a standalone cluster) that observes and controls the participant nodes. It is responsible for coordinating all transitions in the cluster and ensuring that state constraints are satisfied while maintaining cluster stability.

Apache Zookeeper [18] records the view of the cluster, which includes the health of the participant nodes and the state view.

Helix continuously monitors the health of the clusters it manages which are GokuS, GokuL and Goku shuffler and compactor. It polls for the health of the nodes at regular intervals and if there is no response, it moves the shards to the healthy nodes and updates the shard map accordingly.

4.4 Data Storage

Goku uses Gorilla compression technique [26] for compressing both its short-term in-memory data and long-term secondary storage data. Gorilla compression uses the delta of delta encoding scheme for timestamps and xor compression for values, which is strongly suited for time series data. Some examples in the industry that use Gorilla compression are InfluxDB [19], Prometheus [4], Uber's M3DB [3], and Apache IOTDB [37]. We had to make crucial decisions regarding the amount of data to pack into a single Gorilla-compressed stream. Given that we were designing for tiered storage from the start, determining an effective chunking or bucketing strategy was essential for our system's efficiency. After analyzing user query data, the observability team at Pinterest provided valuable insights. They found that users typically query shorter periods for recent data, such as the last 1 hour, last 6 hours, or last 1 day. Conversely, they tend to query longer periods for older data, such as the last 12 weeks, last 6 months, or last 1 year. Notably, 97% of the queries were for data in the last 24 hours. Based on these insights, we decided to adjust the bucket size of Gorilla-compressed metrics data as it ages, to achieve better compression. For older data, we increase the bucket size to maximize compression benefits. For recent data, we use smaller buckets. This approach benefits us by allowing fetching and decoding operations to be parallelized, enhancing performance and responsiveness for recent queries. The bucket sizes are stated in Table 3.

4.5 Tiering Strategy

We observed that most queries for metrics data are for the last 24 hours and decided to use a tiered storage format. We define tiers as partitions of the whole dataset, which allow configuring bucket sizes, rollup strategy, etc. Rollup (explained in subsection 5.2) is a downsampled time series with defined aggregations to reduce the number of datapoints stored. The aggregations we support for rolled up data are sum, max, min, average and count. It is beneficial as it makes queries faster as compute costs to fetch data and aggregate them is reduced due to fewer datapoints. Rollup is done only on the older data (> 24 hours) and at write time specifically in the Goku compactor. To summarize, we store the most recent 24 hours of data in-memory (GokuS) which is tier 0 for us, the last recent 80 days of data that is tier 1 to 4 in nodes with SSD storage, and 80 days to 384 days of data that is tier 5 in HDD based storage nodes. Table 3 explains the tiering strategy. Note that we store raw datapoints (original granularity) only for the last 24 days. Tiering time series data is a well known strategy for saving costs. In 2020, we migrated tier 5 data from GokuL SSD to GokuL HDD because of low QPS observed by monitoring the user query pattern thereby cutting costs.

Table 3: Tiering Strategy in Goku.

Tier	TTL	Bucket size	Raw	Rollup	Cluster
0	24 hours	2 hours	y	N/A	GokuS
1	30 hours	6 hours	y	15 mins	GokuL SSD
2	5 days	1 day	y	15 mins	GokuL SSD
3	24 days	4 days	y	15 mins	GokuL SSD
4	80 days	16 days	n	15 mins	GokuL SSD
5	384 days	64 days	n	60 mins	GokuL HDD

4.6 Write Path

Figure 1 shows the write path in Goku. Every application node in Pinterest has a sidecar running called metrics agent, which pushes the system metrics and application metrics to predefined Kafka topics (metrics Kafka). From here, the Goku ingestor consumes the metrics data and prepares it for GokuS. The Goku ingestor produces the datapoints to another Kafka topic (Goku Kafka). The GokuS storage nodes consume the data from Goku Kafka and back up the data in AWS S3 [12]. From S3, the Goku shuffler and compactor prepare the metrics data for consumption by GokuL storage nodes. We have intentionally separated functionalities into different clusters for independent operation. An approach could be to have the ingestor push datapoints directly into Goku's in-memory storage (GokuS) based on the shard map. However, this method introduces additional complexities:

- **Replica Management:** The ingestor would need to push writes to all replicas and maintain the state of the latest consumed writes across all replicas. It would also need local storage to buffer writes that are in transition or have failed, complicating the ingestion process. Ingestor would need a separate mechanism to delete locally saved data that is past the backfilling period.
- **State Management:** The ingestor would have to handle datapoint routing during restarts or during shard movements, adding further complexity.
- **Functionality Overlap:** Kafka inherently provides functionalities such as consumer offsets and TTL management of the topic, which simplifies ingestion. Relying on Kafka as a persistent data buffer between the ingestor and Goku's storage minimizes the need to duplicate these capabilities within the ingestor itself, leading to a more streamlined and robust system.

Another approach could have been to have GokuS Kafka consume directly from Metrics Kafka topic. However, this approach would require GokuS nodes to route the writes to right shards as the sharding scheme is not visible to the metrics agent. Even if it were, GokuS would consume one message at a time which would contain just 1 datapoint, write it and then move on to the next message. Batching multiple datapoints in a single message is highly advantageous as explained in subsubsection 4.6.1. By using Kafka as an intermediary persistent buffer, Goku efficiently manages data

Table 4: Batching ingestion experiment with 145,440,000 datapoints shows throughput improvements of almost 60x by batching multiple datapoints in single Kafka message

datapoint Batch size	1	10000
Load time (seconds)	3,846	70
Write throughput (DPs/second)	37,816	2,852,652

ingestion while leveraging Kafka's robust features for handling consumer offsets, data persistence, and fault tolerance.

4.6.1 Goku Ingestor. The Goku ingestor is responsible for efficiently processing and routing incoming datapoints. Here, we explain our first novel contribution which is utilizing Kafka and datapoint batching for achieving high write throughput. The ingestion process as shown in Figure 2 is detailed below:

- **Data Consumption:** The ingestor via reader thread pool consumes datapoints from the metrics kafka topic (adhering to the OpenTSDB telnet put format [8]) and stores them in a shared in memory queue buffer.
- **Sharding Assignment:** An internal thread pool-based worker picks datapoints from the shared queue, processes each to determine the target shard ID as per the sharding strategy. Each datapoint is then appended to a queue within an in-memory map, keyed by the target shard ID/ Kafka topic partition ID. The worker also ensures the integrity of each datapoint by checking the length of time series names, timestamps, and values.
- **Batching Mechanism:** A scheduled writer thread runs every few milliseconds, empties the queue of a partition assigned to it, divides it into smaller lists of a pre-configured batch size, and writes each batch of datapoints as a single Kafka message as shown in Figure 3, producing it to the appropriate partition ID. This batching strategy significantly enhances write throughput.

Advantages Of Goku's Batching Strategy: Other TSDBs like QuestDB [30] and TimescaleDB [35] typically batch writes for the same measurement table, but they do not support batching datapoints from multiple tables targeting the same host. Goku's shard-based batching addresses this gap, routing and batching datapoints efficiently, which is crucial for maintaining high write throughput from Goku ingestor. The high write throughput provided by GokuS is explained in subsubsection 4.6.2.

Experiment And Optimization: We conducted an experiment to test the ingestor throughput with different batch sizes. One test run was conducted with batch size 1, sending each datapoint as a single Kafka message, relying solely on Kafka's native batching [14] (configured with linger.ms and producer.buffer.batch.size). The second run had ingestor's default configured batch size 10,000. The 145,440,000 datapoints used in both runs were generated from TSBS [36] with the seed=123 and scale=1000. As seen in Table 4, we observed a fall in throughput with batch size=1 test run. This is because of the increased number of non-payload bytes sent, due to the larger volume of individual Kafka messages, elevating the compression cost in the Kafka producer batching. Given the high volume of datapoints being ingested per minute, batching at the application level in ingestor proves more efficient. This local batching, as shown in Figure 3, reduces the number of messages Kafka has to handle, with Kafka's batching serving as an additional optimization to minimize network payload.

Fault Tolerance: Goku ingestor is stateless and registers itself as part of a Kafka consumer pool when it starts up. If Kafka detects that a consumer is not healthy or not consuming, then it will perform rebalancing and move the partition to a new consumer. For fault tolerance, Goku ingestor also commits the Kafka offset to the metrics Kafka brokers for the consumer once the data has been produced to Goku Kafka. This way the messages can be redelivered to the new node which resumes data consumption after a partition rebalance completes.

4.6.2 Goku Short Term Storage (GokuS). A GokuS node can host multiple shards. A GokuS node hosting a particular shard id, consumes metric datapoints from the corresponding Kafka partition, identified by the same shard id, that the ingestor produced to. The GokuS nodes store and serve the last 24 hours of data and support backfilling capability of up to 2 hours. Backfilling refers to the process of filling in the missing or the historical datapoints in a time series dataset.

Automatic Indexing: Every time series in a shard is identified by its full name, consisting of components like metric name and tag set. As shown in Figure 4, every shard maintains a forward index which is a map with the full name as the key and the value is an index in a vector of time series objects. The empty positions in the vector are maintained in a free list. When encountering a new time series, GokuS pick an available index from a free list and adds it to the forward index. To facilitate query candidate selection, GokuS employs an inverted index (postings list). In this index, each component in the full metric name is mapped to a list of time series vector indices that contain that term. This list is stored using a Roaring Bitmap [23], enabling efficient filtering via union and intersection operations. For example, the 2 time series provisioned at index 0 and 2 are present in the forward index, and the inverted index consists of metric name components of their names. Together, these indexes help speed up the queries by identifying the candidate time series quickly. We have made the indexes along with storing the full metric name in the time series memory efficient by employing deduplication techniques. This is a novel contribution of Goku and is explained in detail in section 6.

Active Data Bucket: As shown in Figure 4, each shard owns multiple time series. Each time series stores the most recent 4 hours of data in 2 active data buckets (buckets explained in subsection 4.4) with granularity of 2 hours each. The active data buckets support backfilling and hence datapoints are stored out of order. The last data arriving datapoint trumps the earlier ones with the same timestamp if any. Each active bucket contains a time series stream which is basically a string storing Gorilla encoded datapoints. Data from an active bucket becomes immutable after 4 hours. This process is called finalization.

Logging: Every datapoint written to an active bucket time stream is asynchronously logged/appended to a log file on a disk. A new log file is created whenever a current timestamp is a multiple of 20 minutes and is named by the same time stamp. Each datapoint, along with its associated Kafka message offset, is placed into a Multi-Producer, Multi-Consumer (MPMC) queue. From this queue,

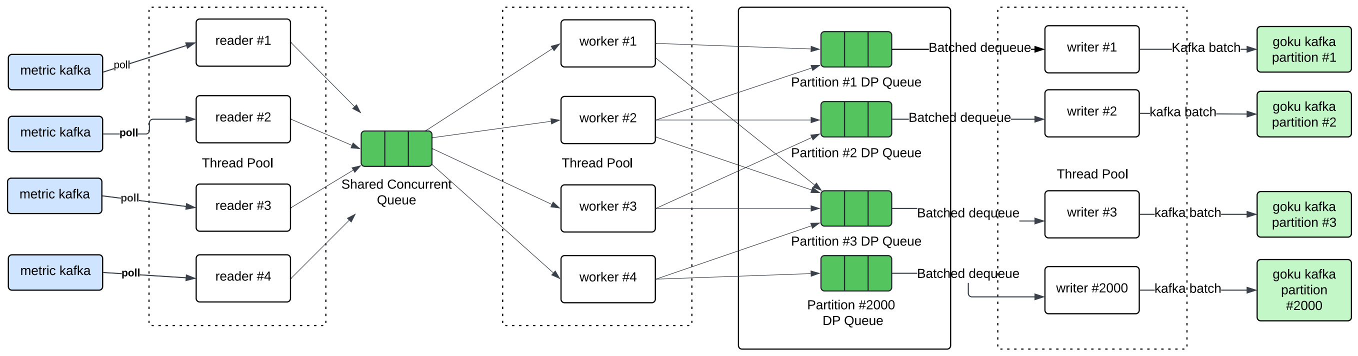


Figure 2: Goku Ingestor architecture showing flow of datapoints from metrics Kafka to Goku Kafka

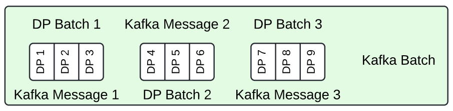
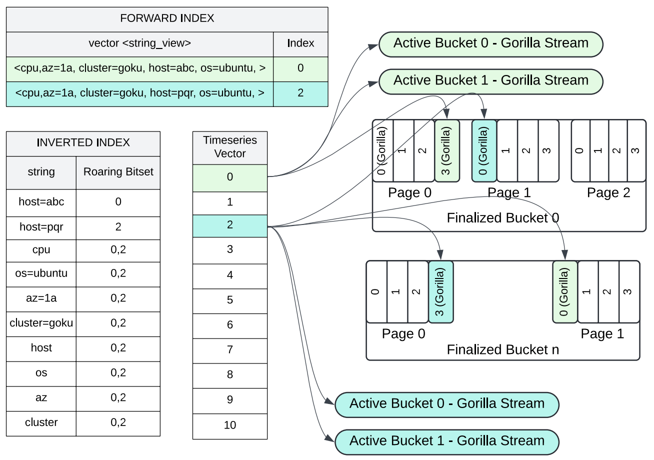


Figure 3: A single Kafka batch consisting of multiple Kafka messages each containing multiple datapoints



FORWARD INDEX	
vector <string_view>	Index
<cpu,az=1a, cluster=goku, host=abc, os=ubuntu, >	0
<cpu,az=1a, cluster=goku, host=pqr, os=ubuntu, >	2

INVERTED INDEX	
string	Roaring Bitset
host=abc	0
host=pqr	2
cpu	0,2
os=ubuntu	0,2
az=1a	0,2
cluster=goku	0,2
host	0,2
os	0,2
az	0,2
cluster	0,2

Figure 4: GokuS shard architecture containing forward index, inverted index, time series vector, active buckets and finalized buckets

a thread pool worker retrieves the datapoint and stores it in a buffer. When the buffer reaches a pre-configured size, the datapoints and associated Kafka offset are batched and appended to the current log file on disk in a specific format. Every 10 minutes, a thread uploads the modified log files to S3. The thread also uploads any local log files which are missing in S3 which helps provide fault tolerance. Because the Kafka offset is logged along with the datapoints in the log file, the recovery routine will always seek the consumer to the latest Kafka offset recovered from the log file. If there is a hardware failure on a host between when a log file is modified and the modification is uploaded to S3, the host which gets new ownership of the shard will simply seek from the Kafka offset which was recorded in the log file downloaded from S3.

Finalized Data Bucket: Every 2 hours, a thread creates a finalized data bucket from the metrics data collected between 4 to 2 hours prior. This data is also backed up in S3. After finalization, the data in the finalized bucket becomes immutable, and the file is named by the Unix timestamp from 4 hours ago. Thus, a finalized file named with a specific Unix time contains datapoints from that time to +2 hours. Only the most recent datapoint for a given timestamp is retained, discarding any previous ones. Finalized data buckets store metrics data for all time series together in a list of contiguous same-sized pages. As shown in Figure 4, every time series has a page index and page offset specifying its data location within the finalized bucket. After finalizing a bucket, the finalization thread uploads the file to S3, deletes the corresponding local log files, and then deletes these log files from S3. The thread also verifies that the configured number of finalized buckets is present locally and on S3. If not, it triggers a finalization routine for the missing bucket, ensuring fault tolerance in case of process restarts or crashes during the finalization routine.

Recovery: In the event of a recovery, GokuS employs a comprehensive routine to restore its in-memory state. If the required data is not available locally, the shard recovery routine downloads finalized bucket files and log files from S3. The data from finalized bucket files are read into in-memory buffers. Each time series is then updated with the relevant page index and page offset within these buffers. Following the loading of finalized data, the log files are replayed to ensure all recent updates are applied. Finally, the recovery routine starts consuming from the Kafka partition by seeking to the latest offset it gets from the log files, ensuring continuity in data ingestion and processing. By using this method, GokuS effectively reconstructs its state. Both the log files and finalized data bucket files are stored in local storage along with backing up to S3 for the sole purpose that recovery may be faster in cases of process restarts as files can be read from local storage and no download from S3 is needed. However, the recovery routine will always verify that if the file is present locally, then file size is same or larger than file size on S3. In cases when the expected file is not present locally or the local file size is smaller than the same on S3, it will download the file from S3 before replaying.

4.6.3 Goku Shuffler and Compactor. Based on the shards it owns, the Goku Shuffler reads finalized bucket files stored in S3 by GokuS and reshards the data according to the GokuL sharding strategy. It

then stores the resharded data back in S3. After shuffling, it adds a marker in S3 to indicate that a particular shard's bucket file has been successfully shuffled. Each time the shuffling routine runs, the shuffler checks for these markers. If any markers are missing, it schedules the shuffle utility for the unshuffled buckets. This mechanism ensures fault tolerance in case a shuffler host crashes and shards need to be managed by another shuffler.
Similarly, based on shard ownership, the Goku Compactor prepares data in the GokuL storage format, using RocksDB SST (static sorted table) [5] files. For each bucket, the compactor reads multiple smaller buckets from the previous tier, merges the data, creates indices, and generates the SST files using RocksDB's SST file writer [31] utility. Like the shuffler, it creates a marker in S3 to indicate the completion of compaction for a particular shard and tier. The compaction routine checks for these markers and schedules compactions if they are missing. For tier 0 to tier 1 compaction, the routine also ensures shuffling markers are present before scheduling. For example, based on Table 3, a tier 1 bucket of 6 hour granularity is prepared from 3 tier 0 buckets of 2 hour granularity or a tier 5 bucket for 64 days is created by merging data from 4 tier 4 buckets of 16 days each. These SST files are uploaded to S3. The keys we insert per bucket into RocksDB are:

Index Key [string] -> [id]: This dictionary key, which maps a string like metric name or tag key or tag value string to a unique numerical id assigned to that string in this bucket. For example, assuming the time series in Table 1, a possible dictionary would be { cpu : 1, mem : 2, host : 3, abc : 4, pqr : 5, xyz : 6, cluster : 7, kv : 8, ml : 9, az : 10, east-1a : 11, os : 12, ubun-1 : 13}

Index Key [id] -> [string]: This reverse dictionary key stores the reverse of the dictionary map.

Posting List Key [metric name id][tag key id][tag value id] -> [encoded time series ids]: This inverted index key gives a list of time series ids within this bucket, which contains the metric name and the tag key-value. During compaction, while creating a new bucket, every time series in the new bucket is assigned a unique numerical id to help create the data key.

Special Posting List Key [metric name id][tag key id] -> [encoded time series ids][tag value ids]: We add another posting list key with tag value id as -1 to fetch all the time series candidates in case of queries with wildcard or regexp filters.

Data Key [rollup aggregator][metric name id][time series id] -> [num tags][tag key ids][tag value ids][number of datapoints][Gorilla encoded data]: This data key maps a time series id to its data, i.e. all tag value pairs and the Gorilla encoded datapoints for the bucket duration.
Another key we insert along with the above is a marker key to indicate that the bucket has been ingested completely. For all the keys added to the SST files of a particular bucket, we prepend the keys with **[key type][tier][bucket]** to indicate the type of the key, i.e. dictionary, reverse dictionary, inverted index, or data key and the tier and bucket information.

4.6.4 *Goku Long Term Storage (GokuL).*
GokuL leverages RocksDB (an LSM tree based persistent key value store) to store long term metrics data. GokuL ingests the data prepared by the compactor from AWS S3. The GokuL process has a thread running that regularly polls if the necessary bucket has been ingested and if not,

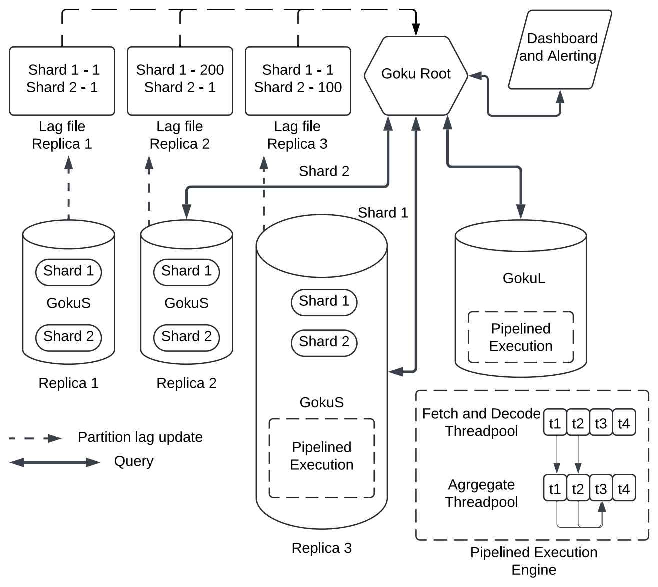


Figure 5: Goku query internals showing partition delay based health monitoring and pipelined threadpool execution

it uses the RocksDB bulk ingestion API for SST ingestion of the bucket. We designed the persistent storage format including the data format and indexing leveraging RocksDB. With local persistent storage that RocksDB provides, we can have fine control on the location of data. This is beneficial as we can implement query pushdown and have storage nodes perform local aggregation of data. Additionally, we store the datapoints of each time series in Gorilla compressed format which highly reduces the storage size. Storing Gorilla compressed metrics data in RocksDB key-value store is novel as compared to other TSDBs.

4.7 Query Path

4.7.1 *Goku Root.*
Goku Root is the query endpoint for observability alerting and web monitoring clients. It monitors the cluster shard map and routes queries to storage clusters. As shown in Figure 5, a query from the Observability client goes to Goku root. Root breaks the query into two queries if the time range touches both GokuS and GokuL. For example, a query for last 20 days is broken into GokuS query for last 1 day and GokuL query for current time - 1 day to current time - 19 days. Further, it fanouts the query to the storage nodes who own the shards in the shard group. For example, root sends the GokuS query to shard 1 and shard 2.

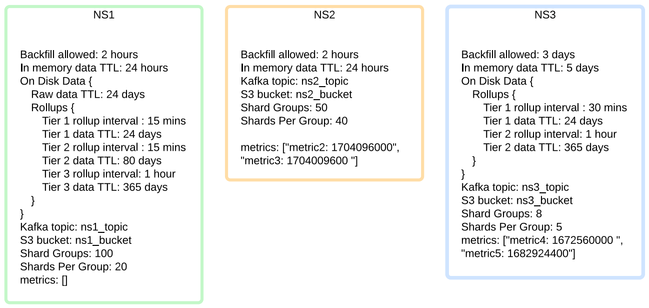
The storage nodes (both GokuS and GokuL) get a list of time series that satisfy the criteria of the filters provided in the query as described in section 3. They then fetch the data of the time series, process the data, perform aggregation if specified, and return the results to the Root node. Root merges these intermediate results from all storage nodes, performs a second round of aggregation if needed, and returns the final result to the client. Root also performs a failover of the query to replica storage nodes if the query fails from a particular storage node. This speculative failover mechanism is used to provide fault tolerance in the query path. Root also rate-limits the queries for system protection.

4.7.2 GokuS Health Monitoring. For both GokuS and GokuL, the spectator process, as described in subsection 4.3, monitors the cluster and node level state and updates the shard map. The shard map is monitored by Goku Root to know which nodes hold which shard before routing queries as shown in Figure 5. However, as GokuS ingests in real time, it's very important to know if the shard is ready for serving queries i.e. if the GokuS shard ingestion is up to speed with the corresponding offset in the Goku Kafka partition. Another reason to continuously monitor the health of the GokuS cluster is because it serves almost all of the alerting data at Pinterest and thus demands high availability. For this, we have modified the spectator process to also query the GokuS storage nodes along with the node health, the latest Kafka message timestamp they have consumed per shard, and update the lag in a file monitored by the Goku Root nodes. This provides information to the Goku Root if a shard is lagging behind in data consumption from Kafka. Based on this health monitoring, root decides whether to route a query to a shard in the particular GokuS node or not. For example, as shown in Figure 5, root will route the GokuS query to shard 1 in replica 3 because the partition lag of shard 1 is high in replica 2. This novel health monitoring of GokuS shards is tied to the Kafka based ingestion logic. It provides fault tolerance in the query path by making Root know which GokuS shard is not up to date for querying.

4.7.3 GokuS. As stated before in subsubsection 4.6.2, for fetching the target time series based on the filter provided in the query, the GokuS nodes maintain an inverted index per shard. This index maps the tag value pair to a list of time series ids in which the tag value pair is present. The list is maintained as a roaring bitset [23]. The idea is to get the bitsets associated with the tag value pairs specified in the query and merge the bitsets (intersect, union etc) based on the filter, which should list of time series ids to fetch data from. This works for AND (tag1=value1 and tag2=value2) include filter where user specifies to consider time series with two different tags, OR (tag1=value1 || value2) include filter where user specifies to use any time series which has either of the tag value pairs, and NOT exclude (tag1=!value1) filter where user specifies to not consider time series which has specific tag value pair. Additionally, Goku supports wildcard and regular expression pattern matching query filter operations as well. A query for the most recent 4 hours of data is fetched from the finalized buckets while data beyond that is fetched from the active buckets.

4.7.4 GokuL. Based on the query start time and end time, GokuL determines the buckets that need to be queried. It creates the dictionary keys, as described in subsubsection 4.6.3, to get the ids of the tag value pairs specified in the filters of the query. To get the list of time series ids, it uses the inverted index key. The lists fetched are then shortened based on the filters provided, and the data is fetched based on the final list of time series. After processing and aggregation, the data is returned to the root.

4.7.5 Pipelined And Thread Pool Based Query Execution. As shown in Figure 5, GokuS and GokuL use thread-pool based pipelined execution. There are separate thread pools for fetching (from in-memory or on-disk), decoding data and aggregation. There are


NS1
```
Backfill allowed: 2 hours
In memory data TTL: 24 hours
On Disk Data {
   Raw data TTL: 24 days
   Rollups {
      Tier 1 rollup interval : 15 mins
      Tier 1 data TTL: 24 days
      Tier 2 rollup interval : 15 mins
      Tier 2 data TTL: 80 days
      Tier 3 rollup interval: 1 hour
      Tier 3 data TTL: 365 days
   }
}
Kafka topic: ns1_topic
S3 bucket: ns1_bucket
Shard Groups: 100
Shards Per Group: 20
metrics: []
```
NS2
```
Backfill allowed: 2 hours
In memory data TTL: 24 hours
Kafka topic: ns2_topic
S3 bucket: ns2_bucket
Shard Groups: 50
Shards Per Group: 40

metrics: ["metric2: 1704096000",
"metric3: 1704009600 "]
```
NS3
```
Backfill allowed: 3 days
In memory data TTL: 5 days
On Disk Data {
   Rollups {
      Tier 1 rollup interval : 30 mins
      Tier 1 data TTL: 24 days
      Tier 2 rollup interval: 1 hour
      Tier 2 data TTL: 365 days
   }
}
Kafka topic: ns3_topic
S3 bucket: ns3_bucket
Shard Groups: 8
Shards Per Group: 5
metrics: ["metric4: 1672560000 ",
"metric5: 1682924400"]
```

Figure 6: Example configurations of 3 namespaces named NS1, NS2 and NS3.

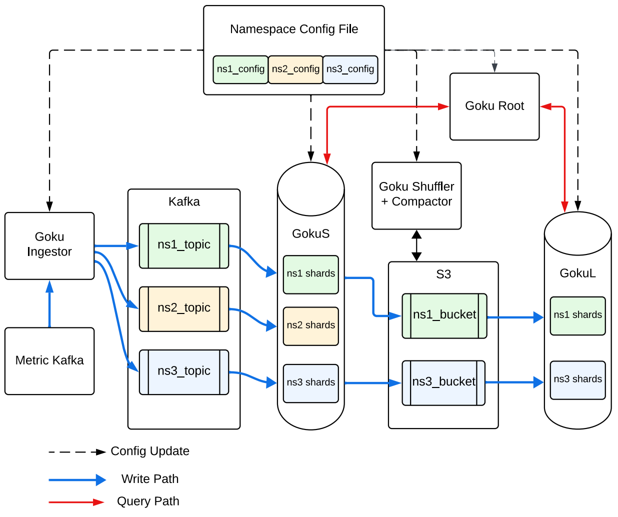


Figure 7: Goku Architecture with namespace

configurable limits on the batch size (number of time series) a thread works on. Pipelined execution allows parallelizing data fetch and compute operations thereby utilizing the system resources in an efficient way.

5 USEFUL FEATURE ADDITIONS

5.1 Namespace

Initially, Goku had a fixed set of properties for the metrics stored which is in-memory storage for most recent 1 day metrics data, disk (mix of SSD and HDD) storage for 1 - 365 days of data, raw data of 24 days, rolled up time series of 15 mins, and 1 hour granularity as data gets older, etc. These were static properties defined during the cluster setup. Adding a metric family with different properties required setting up a new cluster(s) and pipeline. As time passed, we had requests from the client team and some users to support more metric families with different configurations. A generic solution to support these ad-hoc requests was to add namespace support in Goku. A namespace is a logical collection of a unique set of metric configurations and properties like rollup support, backfilling capability, TTL, sharding strategy etc. A metric belongs to only 1 namespace and fulfills all the configured properties of the namespace. The namespace configurations are stored in a config file watched by all the hosts in the Goku ecosystem as shown in Figure 7. Any moment the contents of this file change, the Goku process running on the hosts is notified and it parses the new changes.

As shown in Figure 6, a metric belonging to namespace NS1 will have its most recent 1 day's datapoints stored in memory while older datapoints will be stored and served from disk. The metric will also have 15 minute rolled up data available for the last 80 days and 1 hour rolled up data afterwards. However, a metric belonging to namespace NS2 will not have any data stored on disk. Note how a metric belonging to namespace NS3 will have capability to ingest data as old as 3 days (backfill allowed) whereas metrics in NS1 and NS2 cannot ingest data older than 2 hours. The information about which namespace holds what metrics and the time the metric was added (Unix time rounded to next day 00:00:00 UTC) to the namespace is stored in the same namespace configuration as a list of metric prefixes under each namespace (see key:"metrics" in each configuration). For example, in Figure 6, metrics with prefixes metric2 and metric3 added at 1704096000 and 1704009600 respectively are stored in namespace NS2. All other metrics are stored in the default namespace NS1.

The Goku ingestor compares the metric prefix to find the target namespaces of a datapoint. Then it compares the datapoint timestamp with the time associated with the prefix in the config file to write the datapoint to the corresponding namespace that is the Goku Kafka topic of the namespace. The Kafka topics as well as S3 buckets for each namespace are different. Each namespace has its own set of shards in GokuS and GokuL. In the query path, using the prefix and the associated timestamp, Goku root breaks the query into independent smaller queries each of which is fulfilled by a different namespace. For example, based on Figure 6, datapoints for metric2 will be fetched from the NS1 namespace before time 1704096000 and NS2 namespace afterwards. This seamless migration of metrics between namespaces provided by Goku is novel, and we have not seen support for this from DBs which support namespace-like features like Monarch [2] and Timon [13].

In Production: We have 3 namespaces in production with 2 successful metric migrations into new namespaces as the time the feature was introduced. While 1 namespace was added with the ambition of supporting lower TTL and only in-memory data, the other added more backfilling capability for an ads based metric. In the future, we have 2 more metric migrations planned. One to move the high cardinality metrics to a new namespace with a different sharding strategy and the other to increase the TTL of long term data to 3 years for a select set of metrics.

5.2 Rollup

We support storing rolled up data in GokuL. Rollup is a write time data aggregation process that summarizes and stores time series data at higher levels of granularity. Rollup data generated during compaction benefits Goku by reducing the storage costs of abundant raw data in higher tiers as explained in subsection 4.5. It also reduces the query latency by reducing the amount of data fetched and decoded and lowers the CPU aggregation cost.

5.3 Pre-aggregation and Pagination

Pre-aggregation is when the time series of a metric are aggregated beforehand while preserving some tags and grouping the rest. This helps tackle expensive high cardinality queries that might require a lot of aggregations. The aggregation is done at the write path in GokuS. Currently, Goku has onboarded more than 70 metrics enabled for pre-aggregation. Each of these metrics have cardinality of more than 1 million with some of them having cardinality of more than 50 million.

Pagination refers to the practice of breaking up query results into smaller, manageable subsets or pages. It is commonly used when querying large volumes of time series data to avoid overwhelming the storage nodes by consuming excessive resources during data retrieval or aggregation. Goku implements pagination by returning query results bucket by bucket. We have onboarded 10 metrics (cardinality > 100K), all returning large quantities of time series datapoints due to the usage of wildcard filters in their queries.

6 LESSONS LEARNED/OPTIMIZATIONS

6.1 Highly Redundant Information In Time Series Names

For cost efficiency initiatives, we started tracking memory usage in GokuS specifically to understand the consumers of memory and check if any optimization was possible to reduce the memory usage and thus scale down the cluster capacity. We observed that:

- For a GokuS cluster replica storing 16 billion time series, around 8 TB was consumed by full metric name strings. The other top consumers of memory resources were finalized data buckets (8 TB), active data buckets (2 TB), roaring bitmap (0.5 TB), reverse index keys (0.2 TB).

The full metric name of a time series consists of the metric name and tag value pairs (metric name components) as a single string. Initially, we stored the full metric name of the time series as a key in the forward index as stated in Figure 4. The metric name components are stored as keys in the inverted index. We further observed that

- The cumulative size of keys in the forward index was almost 40x the cumulative size of keys in the inverted index.

We inferred that there was a lot of redundancy if we store full metric name strings in the forward index as metric name components are likely duplicated across time series. For example, as shown in Table 1, the tag-value pair "os=ubun-1" appears in all the 5 time series. To check what could be the best replacement of the substring in the full metric name, we tracked the size of the metric name components and observed the following:

- On an average, every time series had 8 to 16 metric name components and out of them, 6 to10 were auto generated by the observability client. The auto-generated ones signify service or project related details, availability zone information, cluster information etc.
- Almost 70-90% of the size of full metric name strings consisted of metric name component substrings sized more than 16 bytes. Further, all metric name component substrings had a minimum size of 8 bytes.

Table 5: In production comparison of OpenTSDB vs Goku

	OpenTSDB	Goku
Storage	HBase	In memory + RocksDB
p99 latency (sec)	20	GokuS - 0.03, GokuL - 1
2019 costs	5.18M	2.95M
Upkeep Load	30-60 alerts/wk	< 5 alerts/wk
Data size	531 TiB	400 TiB
Replicas	3	3
Data TTL	90 days	13 months

We decided to replace the full metric name with a vector of elements each of which contain a pointer to the single copy of the string of the metric component in the inverted index. At present, as shown in Figure 4, we use std::string_view [15] which takes 16 bytes in a 64 bit architecture to represent the metric name component. After the change was made, we observed almost 75% reduction in the memory consumption of forward index keys (around 2TB) for a similar number of time series. This change also increases the scalability of Goku as it can accommodate more time series with the same capacity (assuming sparse time series). A similar behavior was observed in the Goku Compactor cluster as well which loads multiple buckets of lower tiered metrics data in memory and creates a single bucket for the next higher tier as described in subsubsection 4.6.3. Higher tier compactions like tier 4 to 5 would require billions of time series in memory and would cause out of memory scenarios. By creating a set of metric name components and storing vectors of string views to represent metric names, we observed a sharp decrease in memory usage of almost 60% even during high tier compactions. We were able to remove the cost attributed to on-demand replacement of compactor hosts (vertical scaling) to alleviate the out of memory scenarios.

6.2 Migration from using AWS EFS to AWS S3

GokuS initially wrote logs and finalized data to AWS EFS [11], which provides network-attached storage via a POSIX-compliant file system interface, simplifying feature development. However, EFS was costly and often hit throughput limits during cluster recovery while using the bursting throughput mode. We found that default local instance storage was sufficient for storing data files. Transitioning to S3, which is more cost-effective, and local instance storage reduced costs and improved recovery times, as data was often available locally.

7 EVALUATION

7.1 In-production comparison with OpenTSDB + HBase

In 2018, we launched GokuS for storing one day of metrics data, and in 2019-2020, we introduced GokuL to replace OpenTSDB + HBase for long-term data. Comparing the in-production metrics of both setups, transitioning to Goku proved to be a clear win for Pinterest in terms of cost and service efficiency, as described in the Table 5. Some limitations of OpenTSDB that we observed were:

Table 6: Benchmark Environment Setup

Environment setup	CPU	RAM (GB)	SSD (GB)	AWS EC2 Instance Type
Other TSDBs	64	256	3800	m6id.16xlarge
Goku Ingestor	16	128	950	r6id.4xlarge
Goku Kafka Broker	8	16	475	c6id.2xlarge
GokuS	16	32	950	c6id.4xlarge
Goku Compactor	8	32	475	m6id.2xlarge
GokuL	8	32	475	m6id.2xlarge
Goku Root	8	16	475	c6id.2xlarge

High Operational Cost: Inefficient data compression techniques resulted in high costs, with a datapoint in OpenTSDB consuming almost 12-20 bytes. In contrast, Goku, using Gorilla compression, achieves more than 4x data storage at almost 0.5x the cost, as shown in Table 5.

Performance Issues: High query latencies of several seconds and frequent 60-second timeouts were observed. This was because OpenTSDB, being a stateless service, translates queries into HBase scans, often reading unnecessary data, which slows down reads. Although HBase indexes on the primary key (metric name), defining additional indices for a schemaless dataset on a NoSQL database like HBase is challenging and often requires user intervention. After reading all data from HBase, OpenTSDB performs aggregation on a single machine. This process can be slow due to extended data fetch times and limited parallelization within a single machine. Additionally, OpenTSDB machines often ran out of memory when handling large data sizes. Also, OpenTSDB created two tables in HBase: "tsdb" for storing data and "tsdb-uid" for storing metric and tag key value IDs. Whenever OpenTSDB performed read or write operations, it accessed the "tsdb-uid" table very frequently, making it a hotspot.

Scalability Issues: We started observing scale related issues in OpenTSDB + HBase mainly coming as frequent HBase crashes during high write volumes, CPU-intensive compactions, and expensive query scans. These crashes often forced HBase to read from remote HDFS data nodes instead of local replicas, resulting in sub-optimal query performance.

Goku addresses all the above issues with features like query push-down and two tier sharding for parallelized query execution and aggregation, efficient indexing to filter list of time series candidates before query execution.

7.2 Evaluation with other TSDBs

7.2.1 Test Environment: Using the widely known TSBS [36] suite, we evaluated Goku against TimescaleDB [34], InfluxDB [20] and QuestDB [29]. As stated in subsubsection 4.6.1, Goku ingests data in the OpenTSDB telnet put format [8]. Hence, for benchmarking, we added a client to TSBS that generates and sends data in this format to an HTTP endpoint set up in the Goku Ingestor. As the client batches multiple datapoints in a single request body, the Goku Ingestor splits the request into multiple time series datapoints for

Table 7: Write throughput comparison in datapoints/second

Scale	InfluxDB	TimescaleDB	QuestDB	Goku
1,000	272,505	746,998	3,434,307	2,852,652
100,000	119,782	515,829	2,304,790	2,345,161

further processing. For the read path, we generate queries in the OpenTSDB HTTP query format [25], which are then routed to the Goku Root.

As Goku uses specialized nodes, we configured a 6-node cluster for Goku and a single-node setup for the other TSDBs, ensuring that the total resources for Goku matched those of the single-node TSDBs. The setup details are described in Table 6. We used docker images from the TSDBs' websites and ran them as containers on the host with no special configurations, with the TSBS client running on the same host for the TSDBs, and on the ingestor for Goku. We conducted two tests: one at a 1,000 host scale, creating 101,000 time series with 145.44 million datapoints at 60 second intervals, and another at a 100,000 host scale, creating 10.1 million time series with 727.2 million datapoints at 20-minute intervals to test the high cardinality case.

7.2.2 Write Benchmarking. For all TSDBs, a single worker writes data without flow control to ensure fairness and a uniform client write rate. In Goku, the ingestor caches data in an in-memory buffer before producing it to Kafka. To determine the true write rate, we record timestamp of the first Kafka message created and appended to the Kafka broker, and the time when the last message is written to the async log on the GokuS side. The correct write throughput is calculated by dividing the total number of datapoints ingested by the time difference between these two timestamps. In our experiments, we observed that the first Kafka message timestamp aligns with the client start time for writing data to Goku Ingestor.

Results: As seen in Table 7, Goku demonstrates higher write throughput than InfluxDB and TimescaleDB. InfluxDB ensures fault tolerance by fsyncing to the WAL file, guaranteeing data is written to disk. TimescaleDB, based on PostgreSQL [28], prioritizes fault tolerance through atomic commits and disk writes, leading to reliable transactions. Goku, on the other hand, achieves fault tolerance by asynchronously logging the data along with the Kafka offset, thus maintaining high speed alongside reliability. When scaling up data generation to 100,000 hosts, increasing cardinality by 100 times, QuestDB maintains a high write throughput similar to Goku due to its in-memory mapped region of a column file. TimescaleDB and InfluxDB, with their robust fault tolerance mechanisms, experience a performance shift, but continue to offer reliable data handling. Goku, with its asynchronous logging mechanism, consistently achieves robust performance and reliability.

7.2.3 Read Benchmarking. As the observability client supports HTTP endpoints for the OpenTSDB query format, we used it to communicate with the Goku cluster during benchmarking. We generated 1,000 queries for two scales (host=1,000 and host=100,000) using the same seed as the write tests. The queries were for three use cases:

Table 8: Read Latency (ms) for single-groupby-1-1-1

Scale	InfluxDB	TimescaleDB	QuestDB	GokuS	GokuL
1,000	1.3	11.44	0.98	0.43	0.66
100,000	1.12	27.75	0.71	0.39	0.71

Table 9: Read Latency (ms) for single-groupby-1-1-12

Scale	InfluxDB	TimescaleDB	QuestDB	GokuS	GokuL
1,000	5.29	44.44	2.68	0.51	1.23
100,000	2.27	160.5	0.84	0.38	1.24

Table 10: Read Latency (ms) for double-groupby-1

Scale	InfluxDB	TimescaleDB	QuestDB	GokuS	GokuL
1,000	180.2	92.5	463.83	12.24	82.7
100,000	13553.22	4620.46	2454.89	1081.79	2239.31

single-groupby-1-1-1: 5-minute downsampled data with max aggregation for a single time series over 1 hour.

single-groupby-1-1-12: Same as above, but over 12 hours.

double-groupby-1: 1-hour downsampled data with average aggregation for all time series within the metric over 1 day.
After ingestion, we executed the queries on GokuS, then waited a day for data compaction and ingestion into GokuL. We reran the queries to obtain latency numbers from GokuL.

Results: As shown in Table 8, Table 9 and Table 10, GokuS, being an in-memory engine, consistently provided lower query latencies (3x - 10x) in all 3 query types. GokuL also delivered better or comparable latencies to other secondary storage solutions. In single time series tests ("single-groupby-1-1-1" and "single-groupby-1-1-12"), GokuL uses the index key and the postings list key in RocksDB for quick data retrieval and downsampling. For multiple time series queries ("double-groupby-1"), especially at higher scales, GokuL excels by using a special postings list key, as explained in subsubsection 4.6.3, to retrieve all candidate time series in a single RocksDB call, followed by parallel downsampling with a pipelined thread pool execution engine.

8 RELATED WORK

Gorilla + HBase: Goku's in-memory storage engine, GokuS, shares some similarities with Gorilla storage engine [26], such as using the Gorilla compression scheme and storing the last 24 hours of time series data. Goku offers some distinct advantages and additional features. Goku ensures no data loss by recording Kafka offsets with the asynchronous logs, which are backed up to S3 and synced to disk, whereas Gorilla may tolerate some data loss during node restarts. Goku also seamlessly handles multiple metric family configurations via namespaces. For efficient long-term storage, Goku uses RocksDB, which our production data has shown to be more cost-effective and to provide lower read latencies compared to OpenTSDB + HBase. In contrast, the Gorilla paper states plans to implement SSD-based storage between Gorilla and HBase, which

Goku already manages efficiently. Additionally, Goku prevents partial data returns during shard rebalancing with smart partition lag-based routing, thereby enhancing query reliability.

InfluxDB: InfluxDB [20], provided by InfluxData, is a robust time series database. It has a rich feature set and multiple ways to read and write data. Although InfluxDB is marketed as schemaless, Goku provides a clearer and more robust solution for handling changes in metrics, such as the addition of tags. Goku also excels in sustaining high write throughput required at Pinterest's scale, as confirmed by our benchmark results. Goku's support for metric family configurations and namespaces is more comprehensive than InfluxDB, which primarily allows setting TTL on a bucket.

TimescaleDB: TimescaleDB [34] is another feature-rich TSDB provided as a PostgreSQL extension. While its transactional nature can impact write throughput, Goku excels in this area, as confirmed by benchmarks. Goku's schemaless design simplifies adding tags to tracked metrics, which can be complex in TimescaleDB, especially with compressed data. Additionally, Goku automatically creates forward and reverse indices for quick access, whereas TimescaleDB relies on user-defined indexing.

TDengine: TDengine [33], based on PostgreSQL, offers a unique data model with supertables, allowing users to create multiple tables from a template. However, configuring TDengine requires significant user input to determine which metrics to include in a supertable, ensure synchronized emissions from data collection points, and maintain uniqueness for subtables. In contrast, Goku's schemaless nature and automatic configuration make handling metrics straightforward. Goku also supports pre-aggregated data for faster queries, whereas TDengine supports rolled-up data.

QuestDB: QuestDB [29] is a relatively new TSDB that integrates seamlessly with SQL. Unlike QuestDB, Goku inherently supports features like pre-aggregation and automatic indexing. QuestDB requires third-party integration for pre-aggregation.

Monarch: Monarch [2], Google's in-house TSDB, shares similarities with Goku, including in-memory short-term storage and a regionalized architecture for reliability. Monarch offers a comprehensive configuration plane similar to Goku's namespace but also supports access controls, metric schemas, and standing queries and alerts. However, Goku provides seamless metric migration between configurations, offering superior flexibility. Monarch's long-term storage strategy remains unclear, while Goku efficiently uses RocksDB for cost-effective, low-latency storage.

Timon: Timon [13] handles out-of-order events efficiently using blind writes and lazy merges. While Timon can append late events and eventually merge them, Goku handles out-of-order events similarly and ensures fast write rates. Goku's ingestion process leverages Kafka to buffer writes, which is advantageous when downstream nodes are not healthy. Timon relies on direct writes to storage nodes, which may be less resilient. Additionally, Goku's time-based finalization provides structured data management, compared to Timon's space-based finalization.

9 IN PRODUCTION METRICS

As shown in Table 11, between 2018 and 2022, the number of time series stored in Goku increased from 2.5 billion to 19 billion, more than fivefold. Correspondingly, daily ingested datapoints increased fourfold, indicating the addition of more sparse time series with fewer or burst datapoints. In 2022, as part of a company-wide cost-saving initiative, the observability team decided to eliminate metrics that were never queried. To achieve this, Goku began providing daily reports of the top 10,000 metrics per shard with the highest cardinality and most datapoints. This information, combined with query logs, helped identify and block unused metrics at the metrics agent sidecar, resulting in a drop in time series and datapoints by 2023. For metrics which were to be removed but which were already in the Goku pipeline, we added capability of blocking metrics in Goku Ingestor, during finalization in GokuS and during compaction. This prevented the metrics from being forwarded to the next component. The QPS (queries per second) however increased to 15K, a fourfold rise over six years. By early 2024, Goku was ingesting 4.5 trillion datapoints daily and serving 15K queries per second with a p99 latency of 100 milliseconds. GokuL stores one year's worth of metrics data, totaling over 300 trillion datapoints and 11.5 trillion time series, and serves queries with a p99 latency of 5-10 seconds. Queries for rolled-up time series have a sub-second p99 latency.

Table 11: Write and Read metrics of Goku over the years. The number of datapoints, time series as well as the QPS are averaged over a year. We do not have records of data in 2019.

Metrics / day	2018	2020	2021	2022	2023	2024
datapoints (trillion)	1	1.81	3.35	4	3	4.4
Time series (billion)	2.5	6	10	19	9	14
QPS (1000s)	5	-	-	13	14.5	15

10 FUTURE WORK

SQL Support: Currently, Goku is queried through a thrift interface. However, engineers at Pinterest want to view their metrics data along with other analytical data on Apache Superset [32] and Querybook [27]. These visualization tools connect to SQL based databases. Thus, we want to provide a SQL layer on top of Goku.

Snapshot GokuS Active Data To Disk For Faster Recovery: As stated in subsubsection 4.6.2, GokuS replays the logs for reconstructing the active data. This can be time consuming. Persisting regular snapshots of the active data to local storage and S3 will help reduce amount of logs to be replayed during the recovery routine thus reducing bootstrapping time.

Read From AWS Object Store: We want to store the infrequently accessed metrics data in AWS S3 in a queryable storage format. This might prove more cost efficient than the secondary storage based GokuL clusters due to low storage and access costs.

ACKNOWLEDGMENTS

We are grateful for the contributions of former Goku team members including Jinghan Xu, Rui Zhang, Jian Wang, Hao Jiang, Miao Wang, Tianying Chang and Jian Guo. Additionally, we thank Bo Liu, Dave Burgess, Chunyan Wang, David Chaiken, Ambud Sharma, Zhanyong Wan, the Observability Team at Pinterest, including Brian Overstreet, Wei Zhu, Naoman Abbas, Peter Kim, Colin Probasco, and others, for their unwavering support and valuable insights that have played a crucial role in shaping Goku's success.

REFERENCES

[1] [n.d.]. *Apache HBase Book*. Retrieved March 15, 2024 from https://hbase.apache.org/book.html#arch.overview

[2] 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, and Nick Taylor. 2020. Monarch: Google's Planet-Scale In-Memory Time Series Database. *PVLDB, 13(12): 3181-3194* (2020). https://www.vldb.org/pvldb/vol13/p3181-adams.pdf

[3] M3 Authors. [n.d.]. *Storage Engine*. Retrieved March 14, 2024 from https://m3db.io/docs/architecture/m3db/engine/#m3tsz

[4] Prometheus Authors. [n.d.]. *Storage*. THe Linux Foundation. Retrieved March 14, 2024 from https://prometheus.io/docs/prometheus/1.8/storage/#chunk-encoding

[5] RocksDB Authors. 2022. *A Tutorial of RocksDB SST formats*. Retrieved March 14, 2024 from https://github.com/facebook/rocksdb/wiki/A-Tutorial-of-RocksDB-SST-formats

[6] The OpenTSDB Authors. 2010. *OpenTSDB Cardinality Explanation*. Retrieved March 15, 2024 from http://opentsdb.net/docs/build/html/user_guide/writing/index.html#time-series-cardinality

[7] The OpenTSDB Authors. 2010. *OpenTSDB Overview*. Retrieved March 15, 2024 from http://opentsdb.net/overview.html

[8] The OpenTSDB Authors. 2010. *OpenTSDB Put Telnet*. Retrieved March 15, 2024 from http://opentsdb.net/docs/build/html/api_telnet/put.html

[9] The OpenTSDB Authors. 2010. *OpenTSDB Query*. Retrieved March 15, 2024 from http://opentsdb.net/docs/build/html/user_guide/query/index.html

[10] The OpenTSDB Authors. 2010. *OpenTSDB Query Filters*. Retrieved March 15, 2024 from http://opentsdb.net/docs/build/html/user_guide/query/filters.html

[11] AWS. [n.d.]. *AWS EFS*. AWS. Retrieved March 14, 2024 from https://aws.amazon.com/efs/

[12] AWS. [n.d.]. *What is Amazon S3?* AWS. Retrieved March 14, 2024 from https://docs.aws.amazon.com/AmazonS3/latest/userguide/Welcome.html

[13] Wei Cao, Yusong Gao, Feifei Li, Sheng Wang, Bingchen Lin, Ke Xu, Xiaojie Feng, Yucong Wang, Zhenjun Liu, and Gejin Zhang. 2020. Timon: A Timestamped Event Database for Efficient Telemetry Data Processing and Analytics. *SIGMOD* (2020). https://dl.acm.org/doi/10.1145/3318464.3386136

[14] Confluent. [n.d.]. *Kafka Producer Batching*. Confluent. Retrieved March 14, 2024 from https://docs.confluent.io/kafka/design/producer-design.html#batching

[15] cppreference. [n.d.]. *std::basic_string_view*. cppreference. Retrieved March 14, 2024 from https://en.cppreference.com/w/cpp/string/basic_string_view

[16] Siying Dong, Andrew Kryczka, YanQin Jin, and Michael Stumm. 2021. RocksDB: Evolution of Development Priorities in a Key-value Store Serving Large-scale Applications. *ACM Trans. Storage* 17, 4 (Oct. 2021), 32. https://doi.org/10.1145/3483840

[17] Apache Software Foundation. 2023. *Architecture*. Retrieved March 15, 2024 from https://helix.apache.org/Architecture.html

[18] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. [n.d.]. ZooKeeper: Wait-free coordination for Internet-scale systems. ([n. d.]). https://www.usenix.org/legacy/events/atc10/tech/full_papers/Hunt.pdf

[19] InfluxData. [n.d.]. *In-memory indexing and the Time-Structured Merge Tree (TSM)*. InfluxData. Retrieved March 14, 2024 from https://docs.influxdata.com/influxdb/v1/concepts/storage_engine/#floats

[20] InfluxData. [n.d.]. *InfluxDB*. Retrieved March 14, 2024 from https://github.com/influxdata/influxdb

[21] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: a Distributed Messaging System for Log Processing. *NetDB* (June 2011). https://www.microsoft.com/en-us/research/wp-content/uploads/2017/09/Kafka.pdf

[22] Microsoft Learn. [n.d.]. *Regular Expression Quick Language - Reference*. Retrieved March 15, 2024 from https://learn.microsoft.com/en-us/dotnet/standard/base-types/regular-expression-language-quick-reference

[23] D. Lemire, G. Ssi-Yan-Kai, and O. Kaser. 2018. Consistently faster and smaller compressed bitmaps with Roaring. *arXiv* 4 (March 2018). https://arxiv.org/pdf/1603.06549.pdf

[24] Aditya Agarwal Mark Slee and Marc Kwiatkowski. 2007. Thrift: Scalable Cross-Language Services Implementation. (April 2007). https://thrift.apache.org/static/files/thrift-20070401.pdf

[25] OpenTSDB. [n.d.]. *OpenTSDB http query format*. OpenTSDB. Retrieved March 14, 2024 from http://opentsdb.net/docs/build/html/api_http/query/index.html#query-api-endpoints

[26] Tuomas Pelkonen, Scott Franklin, Justin Teller, Paul Cavallaro, Qi Huang, Justin Meza, and Kaushik Veeraraghavan. 2015. Gorilla: A Fast, Scalable, In-Memory Time Series Database. *PVLDB* 8, 12 (2015), 1816–1827. https://www.vldb.org/pvldb/vol8/p1816-teller.pdf

[27] pinterest. [n.d.]. *Querybook*. Retrieved March 14, 2024 from https://www.querybook.org/

[28] Postgres. [n.d.]. *Postgres documentation*. Postgres. Retrieved March 14, 2024 from https://www.postgresql.org/docs/

[29] questdb. [n.d.]. *QuestDB*. questdb. Retrieved March 14, 2024 from https://questdb.io/docs/

[30] questdb. [n.d.]. *QuestDB write batching*. questdb. Retrieved March 14, 2024 from https://questdb.io/docs/reference/sql/insert/

[31] RocksDB. 2024. *Creating and Ingesting SST files*. Retrieved March 14, 2024 from https://github.com/facebook/rocksdb/wiki/Creating-and-Ingesting-SST-files

[32] superset. [n.d.]. *Superset*. Retrieved March 14, 2024 from https://superset.apache.org/

[33] taosdata. [n.d.]. *TDEngine*. taosdata. Retrieved March 14, 2024 from https://docs.tdengine.com/

[34] Timescale. [n.d.]. *TimescaleDB*. Timescale. Retrieved March 14, 2024 from https://docs.timescale.com/#TimescaleDB

[35] Timescale. [n.d.]. *TimescaleDB insert*. timescale. Retrieved March 14, 2024 from https://docs.timescale.com/use-timescale/latest/write-data/insert/#insert-multiple-rows

[36] Timescale. [n.d.]. *TSBS Benchmark*. Retrieved March 14, 2024 from https://github.com/timescale/tsbs

[37] Chen Wang, Xiangdong Huang, Jialin Qiao, Lei Rui, Jinrui Zhang, Rong Kang, Julian Feinauer, Kevin A. McGrail, Peng Wang, Diaohan Luo, Jun Yuan, Jianmin Wang, and Jiaguang Sun. 2020. Apache IoTDB: Time-series Database for Internet of Things. *PVLDB* 13, 12 (Aug. 2020), 2901–2904. https://doi.org/10.14778/3415478.3415504