Meta's Next-generation Realtime Monitoring and Analytics Platform

Stavros Harizopoulos, Taylor Hopper, Morton Mo, Shyam Sundar Chandrasekaran,
Tongguang Chen, Yan Cui, Nandini Ganesh, Gary Helmling, Hieu Pham, Sebastian Wong
Meta Platforms, Inc.
Menlo Park, CA
kraken-paper@fb.com

ABSTRACT

Unlike traditional database systems where data and system availability are tied together, there is a wide class of systems targeting realtime monitoring and analytics over structured logs where these properties can be decoupled. In these systems, responsiveness and freshness of data are often more important than perfectly complete answers. One such system is Meta's Scuba [2].

Historically, Scuba has favored system availability along with speed and freshness of results over data completeness and durability. While these choices allowed Scuba to grow from terabyte scale to petabyte scale and continue onboarding a variety of use cases, they also came at an operational cost of dealing with incomplete data and managing data loss.

In this paper, we present the next generation of Scuba's architecture, codenamed *Kraken*, which decouples storage management from the query serving system and introduces a single, durable source of truth. This enables tangible improvements to system fault tolerance and query performance while still respecting tolerable bounds of client observed data freshness. We also describe the journey of how we deployed Kraken into full production as we gradually turned off the older system with no user-visible down time.

PVLDB Reference Format:
Stavros Harizopoulos, Taylor Hopper, Morton Mo, Shyam Sundar Chandrasekaran, Tongguang Chen, Yan Cui, Nandini Ganesh, Gary Helmling, Hieu Pham, Sebastian Wong. Meta's Next-generation Realtime Monitoring and Analytics Platform. PVLDB, 15(12): 3522 - 3534, 2022.
doi:10.14778/3554821.3554841

1 INTRODUCTION

Scuba [2] is Meta's structured log analytics platform with tens of thousands of datasets, used by thousands of users daily. Engineers and data scientists use Scuba to debug large distributed systems, visualize system performance, and derive operational insights, among many other use cases.

Realtime monitoring and analytics systems occupy a unique point in system design space which in turn has enabled Scuba to make unique system tradeoffs. For example, the original architecture sacrificed data availability by ignoring stragglers at query time in order to achieve better query performance, and relaxed data durability guarantees to simplify the design and speed up ingestion.

The original Scuba system was deployed at Meta[1] over 12 years ago. While the type of scenarios for which Scuba is a good fit has remained relatively unchanged, the number of users and use cases on Scuba have grown significantly since then. This growth in workloads and deployment size, and the consequent increase in the operational load has motivated a re-examination of some of the original tenets of the Scuba design as the system has evolved to handle this increase in scale.

In this paper, we first describe a major rearchitecture (codenamed Kraken) of Scuba, made to address said increase in system scale. We (a) describe a new data ingestion pipeline that eliminates view divergence between geographically isolated deployments of Scuba (covered in Section 3.1), (b) explain how we used a combination of a globally consistent backup and control messages to separate the control plane functionalities from compute nodes in order to minimize the resource contention impact of data management operations on user queries and achieve a single source of truth, and (c) summarize how the query path of the system was adapted in order to accommodate these changes while still supporting fast query response times of less than 100ms at P50.

We then cover some of the aspects of how the migration onto this new architecture was accomplished in place with no user-visible time nor planned data-loss such as (a) the incremental migration and performance validation steps that were undertaken as we moved from a model where data was ingested in a single data center location to multiple data center locations, (b) how data ingested by the new and previous architecture were merged in a user-transparent fashion, and (c) how we verified we were resilient to the the new fault domains introduced by the shift from a single region to geographically distributed architecture.

We finally demonstrate Kraken's ability to recover after 10% of nodes in a deployment are taken offline during during fault injection testing as well as highlight the reductions in latency and network utilization on the query path with the Kraken architecture.

The paper is structured as follows. In Sections 2 - 3, we provide an overview of the problem space and the tradeoffs made in the original system. In Section 4, we provide a deep dive into the Kraken architecture. Sections 5 and 6 cover productionization and experimentation. We describe related work in Section 7 and conclude in Section 8.

Proceedings of the VLDB Endowment, Vol. 15, No. 12 ISSN 2150-8097.
doi:10.14778/3554821.3554841

[1] Previously known as Facebook.

2 TENETS OF DATA PROCESSING AT SCALE

Over the last two decades, an ever increasing number of companies started accumulating vast amounts of data, from text files to logs, to various docs, to web pages, all the way to event data from web apps, mobile apps, and online games (user clicks, sign ups, etc.). To cope with scale, new generations of systems targeting analytical use cases appeared, ranging from the commodity hardware based MPP Data Warehouses of the mid-2000s (Greenplum [31], Vertica [19], ParAccel [6], Aster [10]), to the scale out SQL engines of mid-2010s (Impala [17], Presto [28], F1 Query [27]), and from modern OLAP systems (Pinot [15], Druid [32], Clickhouse [7], Napa [3], Procella [5]) to modern cloud Data Warehouses (Snowflake [8], BigQuery [12], Redshift [13])—we cover some of these systems in Section 7.

The continuing shift of CPU/memory/storage/network trends led to the re-emergence of columnar storage [1], used by most of the above-mentioned systems. Open storage formats like ORC and Parquet gained in popularity and contributed to the rise of Data Lake and Lakehouse [33] architectures. A new ecosystem grew rapidly with the openness of these architectures, and new use cases started flourishing, ranging from Data Science and Interactive Data Exploration to Machine Learning, and from Advanced Analytics to Monitoring, Realtime Dashboards and Operational Intelligence. These use cases represent a wide spectrum of analytical use cases and we cluster them in the following three classes of data processing platforms:

(1) Realtime Monitoring and Analytics
(2) Modern OLAP
(3) Data Warehousing

The focus of this paper is on the first category, but we also provide in this section additional context behind the motivation and design space of all three categories to further highlight the tradeoffs when building systems for speed and scale. Note that other classes of systems such as transactional (OLTP) databases, key value stores, specialized systems for time series (Gorilla [26]), and log/text search (Elasticsearch [9]) are left out from this discussion.

2.1 Design space

When it comes to modern analytical data processing systems, whether for Monitoring, OLAP, or Data Warehousing, different designs typically pick a set of tradeoffs along six broad axes:

[Query Performance] Dashboards and interactive data exploration ideally need sub-second query latency time to maintain responsiveness, whereas certain apps and systems may require even lower response times (tens of milliseconds). Complex queries (containing many joins, UDFs) on large datasets can take minutes to hours but it is still important to speed up these queries as they often dictate the downstream landing time. While query complexity and dataset size can adversely impact performance, there are ways to trade off cost and freshness by creating secondary structures (indexes, MVs) or using expensive storage (RAM, SSD).

[Freshness] For certain applications like monitoring, realtime analytics (e.g., understanding the virality/engagement of content immediately after a post), or providing realtime feedback to ML models, having data available for querying seconds after it gets generated is crucial. On the other end of the spectrum, certain batch jobs typically execute overnight, so waiting for an hour or more before starting the job is typical. Freshness often conflicts with all other axes as working against a deadline means that there are limited opportunities for optimizations such as compaction and materialization or techniques such as mirroring/replication.

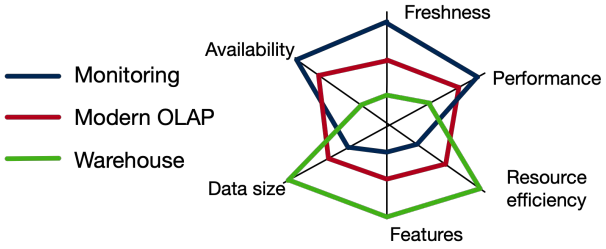

Figure 1: Design space for data processing platforms

[Availability] Different applications can tolerate various levels of unavailability and this is usually reflected on their SLOs/SLAs. A batch workload can tolerate a failover duration of a few hours if a disaster event (entire datacenter down) is expected to be rare, whereas a monitoring system not only needs to continue operating normally when one or even two datacenters are down, but it also needs to provide best-effort availability in the presence of wide outages even if that means incomplete data. The number of replicas and the type of replication (synchronous vs. asynchronous) can affect targets set for hardware footprint and freshness.

[Dataset size] At the one end of the spectrum, storing exabytes of data in a resource efficient manner means that data needs to reside on HDDs with an erasure-coding scheme for durability and one online copy for dealing with outages (in addition to an offline copy for disaster recovery). Adding more replicas or moving data to faster storage can improve availability and performance but can quickly become detrimental to resource efficiency.

[Resource efficiency] As mentioned above, there is a high correlation between hardware selection and dataset size, availability, and performance. Other than the type and size of storage (RAM vs. SSD vs. HDD), there is a tradeoff to make in the type of nodes (scaling up vs. scaling out) as well as the provisioned network bandwidth and IOPS.

[Features] The final category is a wide bucket that includes all user facing features and capabilities of a platform, such as types of queries supported, ability to execute custom code, support for complex workflows including backfills, support for nested and/or custom data types, integrations with other systems such as workload managers and schedulers, privacy checkers, and tools like forecasting and alerting.

2.2 Warehousing vs. OLAP vs. Monitoring

Fig. 1 shows the design space along the above-mentioned six axes for the three classes of data processing platforms we outlined earlier.

For Data Warehousing, the non-negotiable tenets are resource efficiency, ability to scale to the largest of data sizes, and extended support for a wide variety of use cases (complex pipelines, custom code applications, batch queries) whereas some sacrifices can be made in Availability, Freshness, and Query Performance. For availability, typically a primary region followed by secondary region

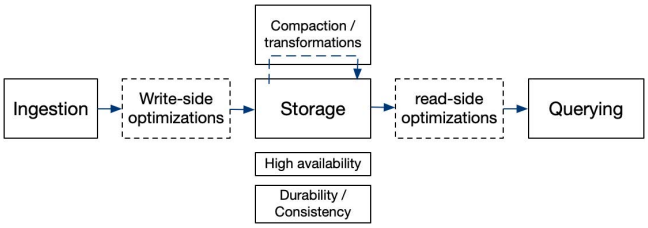

Figure 2: High level components for analytical systems

that can be switched within tens of minutes is acceptable, whereas query performance will depend on the type of queries and can range from seconds to hours. To achieve good performance, techniques such as partitioning and compaction can add to the total time to land data and so freshness is typically in the 15 min to an hour range, with certain applications (near realtime) approaching a few minutes. Pushing the envelope in the latter axes (for example, through more expensive storage) would mean that the first three axes would have to give up something.

Modern OLAP systems such as Napa [3], Druid [32], Pinot [15], and Clickhouse [7] follow a more balanced approach by targeting modest data sizes, medium resource efficiency, and a selected list of features, while improving performance, freshness, and availability. Dataset sizes for these applications are in the TB range, and total size of a cluster is typically in the TB or low PB range. To achieve high performance, these systems typically resort to the use of indexes or materialized views, or restrict the types of queries they can support. To achieve good freshness, they integrate directly with the ingestion system and try to perform as much functionality asynchronously as possible, such as compaction or replication. Among these systems, Napa is taking a further step of allowing clients to configure their own tradeoffs among performance, freshness, and resource cost.

Realtime monitoring and analytics systems sit directly opposite to the tradeoffs made by Data Warehousing systems. Availability, freshness, and performance are of utmost importance as any lapse in any of these axes can negatively impact the utility of the system. Quickly debugging with the latest data during a SEV[2] or allowing a creator to quickly assess the virality of their post are all scenarios where speed of reaction is key. In these use cases it is often acceptable to restrict the dataset sizes and the number of features while taking a hit in the resource efficiency of the system, as long as the first three categories are met at their max point.

Fig. 2 shows the high level components that are common across the three classes of systems mentioned above. Data typically flows from an ingestion service feeding off from a message bus, like Kafka [18], which acts as a durable, highly available buffer architected in a way that can help the system deal with transient failures and transient load spikes. From there, certain systems may opt for write-side optimizations such as updating Materialized Views or Indexes, before making data available in a read-optimized format. The primary store is responsible for providing durable, consistent, and highly available storage to the querying service. Most systems will perform further optimizations (suchs as more rounds of compaction) or downstream transformations and write back to the store. The querying service may have different degrees of coupling with storage and may also apply different types of read-side optimizations.

Next, we describe the tradeoffs that we have chosen over the years for Meta's realtime monitoring and analytics platform, Scuba, focusing on the tradeoff of data vs. system availability, and motivate the need to redesign the system.

[2]SEV: site event, noting severe disruption or unavailability of a service.

3 REVISITING SCUBA TRADEOFFS

As a realtime monitoring and analytics system, Scuba occupies a unique point in tradeoff space. In this section, we first outline Scuba's design and focus on the tradeoffs that it made and differentiated it from other systems (3.1) and then motivate the need for rearchitecting the system (3.2).

3.1 Scuba tradeoffs

There are five main types of tradeoffs that we applied to Scuba to scale over the years, from TBs of data to low PBs, while maintaining freshness, performance, and availability targets:

[Massive fanout tree architecture] Scuba followed an aggregation tree topology [2, 24] that over the years settled for just three levels: a lower level of up to thousands of leaf nodes, a middle level of tens of aggregators, and a top level of a root node. Unlike MPP engines and Warehouses where peer-to-peer communication quickly runs into quadratic network overhead, an aggregation tree can scale to tens of thousands of nodes. The tradeoff taken in this case is that there is no support for joins, as shuffling would have required sizeable communication overhead, and hence no support for global sorting. The benefit is that aggregation-projection-selection queries can run fast as the computation is spread among thousands of nodes.

[Memory is used mostly for caching] Scuba uses most of the memory on the leaf nodes for two types of caching: block-fragment caching (column fragments within a block that were recently accessed) and intermediate result caching. The latter form is particularly effective as many queries apply a time-shifting window and many per-block aggregations or selections can be reused. The tradeoff is that query result size and scratch space are limited (for example, Scuba restricts the number of GROUP BY groups to 400K).

[Single zone map on time] Modern warehouses and OLAP systems employ a wealth of techniques to prune data and save on computation: from zone maps [25] to bloom filters and indexes, and from sort keys to distribution keys (for co-locating joins). All these mechanisms come with computational overhead, typically during ingestion. Scuba instead opts for just a single zone map on the time column: for most monitoring applications, this turns out to be highly effective.

[Flat and flexible schema] Although Scuba ingests JSON data that can be arbitrarily nested, it flattens it at the top level to avoid ingestion overhead. This makes parsing and columnar block assembly fast and shifts the processing overhead to query time, for those queries that need to dig deeper into nested fields (but this happens after several blocks of data have already been pruned by the query filters). Scuba also foregoes the use of a centralized catalog which

would induce overhead every time there is a schema change. Instead, all columns are ingested as they appear in the input stream, and the schema is built and validated on the fly, at query time.

[Best effort data availability and durability] The final tradeoff of Scuba is that it opted for best effort data durability and availability. Each Scuba deployment is a geographically isolated deployment that independently consumes data from the message bus and only stores a single copy of the data. While this loose form of replication has no synchronization overhead at all, different deployments can easily go out of sync as nodes fail, causing different gaps in data on different deployments and possibly data loss. Since data availability is imperfect by design, Scuba also deals with straggler nodes (leaf nodes that are slow) by ignoring those nodes and returning incomplete data (the end users receive an indication of how complete the input to their query was–Scuba will typically process at least 98-99% of input data). This effectively bounds the impact of any tail latency issues to the leaf query processing timeout. While these choices make query and ingestion particularly fast, incomplete query results become an inevitability in the system.

3.2 Motivation for redesign

The original Scuba system that was described in [2] was an in-memory row store that would suffer data loss even when a node restarted. As more use cases were onboarded and grew, support for flash-backed columnar storage allowed for fast restarts [11] and better performance. To improve system and data availability, new (independent) deployments were added in geographically varied datacenter locations. Compared to Fig. 2, Scuba sidesteps write-side optimizations and support for durability and consistency. It additionally performs compaction directly on the storage nodes (and no other transformation) and as a form of read-side optimization on the query serving path it utilizes its massive-fanout tree architecture to parallelize aggregations across thousands of nodes. All Scuba deployments, including ingestion, are completely independent; queries are sent to all deployments and the response with the best data availability is selected.

As the total size of a Scuba deployment crossed into the PB range, serving tens of thousands of datasets, there were significant opportunities for improvement that motivated rethinking the architecture of the system. We group these opportunities into two categories, user-facing and reduction in operational overhead.

3.2.1 User facing opportunities. The number one priority of Scuba is to empower its users to better analyze data. The quality of the insights is only as good as the quality of the data from which they were derived. Specifically, we found that Scuba users could be better served by:

Improved consistency. With every node failure, there would be gaps in a dataset and these gaps would be different on each deployment. On top of that, having independent ingestion to each deployment (which follows *at-most-once* semantics) means that the deployments also receive slightly different versions of the data. Since Scuba returns each time the query with the best data availability, users are exposed to different results that can vary significantly (depending on the filters used) even when submitting the same query twice.

Improved durability. One partial mitigation to the consistency problem is to pin queries to a specific deployment and smooth out transient failures by submitting the same query multiple times. While this is not a practical choice for most of our users (and it also defeats the purpose of having an always fresh and speedy data store), it does not solve the problem as permanent node failures resulting into permanent data loss. For clients who need to access the entire dataset we would need to provide better durability.

Increased retention. The coupling of compute, storage, storage management, and ingestion means that the only way to provide more retention to users is to increase the size of the entire deployment (Scuba allows sampling with a ratio that can be configured differently for older data but this is orthogonal to the user requests for more overall space). The opportunity here is to find a way to utilize resource-efficient storage (HDDs) in a decoupled way that does not affect the performance of queries that do not need the extra retention.

3.2.2 Reduction in operational overhead. As usage and deployment size of Scuba continue to grow, managing the system becomes more challenging for teams operating the system. The operational burden of the system could be reduced by:

Simplified host draining. Without any replication or backup mechanism, any planned downtime for hosts (e.g., to upgrade the OS or perform hardware replacement) meant that the operators need to drain the hosts (move the data to a different set of hosts). This was a fragile and error-prone process that would significantly benefit from a robust automation.

Improved node stability. With coupled read and write paths, an ingestion load spike or an increased compaction activity could easily overwhelm those nodes that were dealing with heavy read traffic, leading to an imbalanced system and need for manual intervention from the operator on call. Separating the concerns could lead to a more balanced and stable system.

Independent scalability of storage and compute. While scaling independently storage and compute has been the cornerstone of modern cloud Warehouses like Snowflake [8], for a multitenant, coupled system like Scuba we needed to examine one by one the requests from our largest users, whether they were requests for more computational resources or more space, and come up with the best strategy for efficient resource increase and allocation.

To pursue the above opportunities and ensure the stability and growth of Meta's realtime monitoring and analytics platform for years to come, we took on the challenge of re-architecting Scuba and replacing it with a new system without any user-facing down time. Next, we describe the design and implementation of the new architecture, *Kraken*, which brings significant changes to the previous Scuba architecture.

4 KRAKEN ARCHITECTURE

We structure the description of Kraken around the lifecycle of data: from client submission to user queries. In Section 4.1, we first discuss the components used to build the new architecture. In Section 4.2, we describe a new ingestion pipeline that improves consistency and efficiency for data processing. Section 4.3 - 4.6 expand on how data is partitioned, stored, and managed. Finally,

Section 4.7 - 4.8 describe how the data is ultimately used to serve users' queries.

4.1 Building blocks

4.1.1 Scribe. Scribe [16] is a distributed message queue that aggregates data written to Kraken into per-dataset streams called *categories*. Each category is backed by multiple LogDevice [21] logs. Scribe provides a streaming API to its readers (called *tailers*) to consume from all logs at once. Scribe does not guarantee a strict output order for messages, so multiple readers of the same category may receive different views of the same underlying messages.

4.1.2 Turbine. Turbine [23] is a streaming application management service that provides the runtime environment, scaling, and checkpoint storage and distribution, for Kraken tailers. Based on the ingestion rate of a category, there may be dozens or hundreds of tailers consuming from it simultaneously. Turbine uses Shard Manager [20] to ensure tailers of the same category are geographically distributed. It also uses Zeus, a distributed metadata store with a ZooKeeper [14] API, to provide at-most-once semantics for tailers.

4.1.3 LogDevice. LogDevic [21] is a component of the ingestion pipeline. Each LogDevice cluster hosts multiple logs identified by log ID. Each message in the cluster is uniquely identifiable by its log ID and log sequence number (LSN)[3], where the LSN is guaranteed to be monotonically increasing.

Kraken's LogDevice cluster is provisioned across five physically isolated clusters spread across multiple datacenter locations. Each message in the cluster is replicated synchronously across at least three randomly selected physical clusters before it is acknowledged. Appends to individual LogDevice logs are linearizable.

Every log in the cluster has a fixed capacity, limited by the physical disk space on the machine, Kraken uses a space-based retention mechanism to keep the size of the log in check. A message is *trimmed* when it is backed up to a globally replicated BLOB storage service (Section 4.1.4). When a log gets full, for example, as a result of a system failure in the backup process, LogDevice will reject incoming messages. This signals to the writers (in this case, the Kraken tailers) to retry, and applies back pressure to the ingestion pipeline.

4.1.4 BLOB Storage. Kraken uses a BLOB storage service similar to Amazon S3. Kraken uses this to store a backup of all the data on leaf nodes. Data in the BLOB storage is synchronously replicated with read-after-write consistency.

The metadata of the BLOBs, including their logical paths, is stored in a fast key-value store (ZippyDB [22]) with a read-aside cache. This enables efficient ranged enumeration of directories. This additionally allows the metadata to be retrieved separately from the BLOBs.

Kraken uses the BLOB storage to store data, for three purposes:

(1) **Backup.** The BLOB storage serves as durable storage for all data. In case of server failure, the shards served by the server are moved to other servers, which will download the backup and resume serving the shards.
(2) **Staging area.** The BLOB storage allows Kraken to separate the management of data from the data plane. Auxiliary services such as the Compaction Service write results into the BLOB storage, and notify leaf nodes of the existence of new blocks with control messages written into the same LogDevice log (Section 4.4).
(3) **Checkpoint storage.** For any given shard, the set of all data is the union of those in the backup, and those in its corresponding LogDevice log. Kraken stores the combination of log ID and LSN, known as the *checkpoint*, of a shard as metadata (Section 4.3). The server serving the shard can therefore reconstruct all of the shard's state by first downloading the backup, and re-reading from LogDevice for all new data written after the checkpoint in the shard's metadata.

4.1.5 Shard Manager. Kraken uses Shard Manager [20] to manage shard assignment to servers, route requests to servers, load balance shards, and handle fail-over for several services in Kraken. Kraken servers implement a simple interface with *addShard* and *dropShard* functions; Shard Manager communicates shard assignment through a series of calls to these interface functions to Kraken servers.

addShard. This signals that a shard is to be added to the server. Depending on the server's role in the system, it performs different actions in preparation to serve the shard.

dropShard. This signals that a shard is to be dropped from the server. This is the opposite of **addShard**.

Load balancing. The servers keep track of and publishes performance metrics, such as CPU utilization, for each shard. These metrics are consumed by Shard Manager to make decisions on load balancing. Shard Manager triggers shard movement (via a series of calls to **addShard** and **dropShard**) when it detects load imbalance beyond a configurable threshold.

Fail-over. When a server loses connection to the Shard Manager service, it immediately stops serving the shard to prevent duplicated actions from being taken in the case of a network partition. When Shard Manager detects that a server is down it adds the shard to another server after a configurable amount of time.

Routing. Shard Manager keeps an updated *shard map* for the servers. The shard map can be used to route requests to different shards. For example, an aggregator uses the shard map to fan out a read request to leaf nodes (Section 4.7.1).

4.2 Ingestion

Fig. 3 shows the overall ingestion architecture of the system. Client machines across Meta's infrastructure submit *samples* of interesting events to Scribe [16].

Kraken's tailers run on the Turbine platform (Section 4.1.2), read incoming samples from Scribe and transform batches of them into a columnar storage format (Section 4.3) called *RowBlocks*. To keep ingestion realtime new RowBlocks are created with minimal batching on the order of seconds.

A RowBlock is first associated with a 64-bit random unique ID generated by concatenating a 32-bit random integer with a 32-bit

[3]LogDevice may choose to combine multiple user messages into one LogDevice message and assign an offset to each of the user messages. As a result, the user message is identified by the three-tuple of log ID, LSN, and offset.

timestamp. Once created, the RowBlock and its globally unique ID becomes an immutable record of the data.

RowBlocks are then randomly assigned to a *partition* of a dataset. A dataset partition is deterministically mapped into a number of *shards*. The mapping procedure is discussed in more detail in Section 4.3.1. Once partitioned and mapped, the RowBlock becomes an immutable record of data and is the unit of operation of the rest of the ingestion pipeline.

The Kraken tailers write RowBlocks to a LogDevice [21] cluster with the same number of logs as number of shards; that is, one log per shard. The LogDevice cluster serves as the throughput optimized staging area for the RowBlocks, as well as the transport for data management control messages. The logs are read by two services:

(1) Kraken backend leaf nodes. The leaf nodes store data and make it immediately available for user queries. Section 4.7 describes the query processing in more details.
(2) Backup and compaction service. This is an auxiliary service that compacts smaller RowBlocks into larger ones, and backs them up in durable storage (Section 4.5).

4.3 Data Layout

4.3.1 Sharding. Kraken uses two different sharding mechanisms on the data plane and control plane. On the data plane, a dataset is *partitioned* according to a configurable number of partitions. On the control plane, each dataset partition is mapped onto a *shard*.

A dataset starts with 32 partitions when it is first created, and may scale up to 8,192 partitions depending on the amount of data in the dataset. The number of partitions for a dataset is stored and distributed in realtime by a central configuration repository [29].

The mapping of a dataset partition to a shard is deterministic, by the following equation:

$$ShardId = \Big(\textbf{hash}\,(FormalizedPartitionName) + PartitionId^2\Big) \textbf{ rem } NumShards$$

where *FormalizedPartitionName* is simply a string concatenation of the name of the dataset and its partition ID, and *NumShards* is the total number of shards, which is chosen to be a prime number. This mapping scheme minimizes the probability of multiple partitions of the same dataset being mapped to the same shard.

Given that most data in Kraken age out over relatively short time frames (days to weeks), Kraken does not shuffle data around shards. Instead, if the number of partitions for a dataset needs to be increased, the new number is simply written to the configuration for that dataset. Over time, incoming data will, by random chance, be assigned to the new partitions (and therefore mapped to new shards), while the older data in existing partitions (and shards) age out over time. Ultimately the dataset reaches an equilibrium among its partitions.

4.3.2 BLOB Storage. Listing 1 shows the directory structure in the BLOB storage service (Section 4.1.4), where backups of all RowBlocks are stored. `scuba_backup/tree` is the root of the logical container.

Listing 1: Backup directory structure

```
scuba_backup/tree/shards
+-- ...
+-- 20349 (metadata: shard checkpoint)
|   +-- ad_metrics--8190
|   |   +-- 18446727280387347253-4294967295
|   |   +-- 18446727280387347391-4294967295
|   |   +-- 18446727280387347571-4294967295
|   |   +-- 18446727280387347783-4294967295
|   |   +-- 18446727280387347976-4294967295
|   |   +-- 18446727280387348999-4294967295
|   |   +-- 18446727280387350120-4294967295
|   |   +-- 18446727280387350907-4294967295
|   |   +-- 18446727280387352058-4294967295
|   |   +-- 18446727280387353029-4294967295
|   |   +-- (other RowBlocks)
|   +-- (other dataset partitions)
|   +-- zippydb_replica_state_dbg--1115
+-- (other shards)
+-- 99991
```

The top level directories correspond to the shards. The LogDevice checkpoints (Section 4.1.3) are persisted as metadata on the top-level shard directories.

In each shard directory are the dataset partitions, each its own directory, laid out lexicographically according to their formalized partition name.

In each dataset partition directory are individual RowBlock files. The RowBlocks are named using the following format:

$$\texttt{rowblock\_name} = \overline{\texttt{lsn}} + \overline{\texttt{offset}}$$

where $\overline{x}$ is the bitwise inversion of x.

RowBlocks are named this way such that they are naturally sorted in reverse chronological order. With efficient metadata-based ranged enumeration (Section 4.1.4), this allows fast retrieval of recent RowBlocks (or their metadata), as they are used more often by both Kraken and auxiliary systems (Sections 4.5, 4.6).

4.4 Control Plane

An important component of any modern OLAP system is to not only be able to serve queries using stored data, but also to manage them appropriately.

Historically, Scuba leaf nodes have also been responsible for performing management duties on the data they store and serve (Section 4.6). This led to two issues:

(1) **Inconsistency**. Since different deployments have different views on the data (Section 4.2), they end up performing slightly different operations on the data. This further leads to auditing difficulties when required.
(2) **Resource contention**. The data management operations take up a significant portion of the system resources, often in a bursty fashion during which regular user query workloads may be negatively affected.

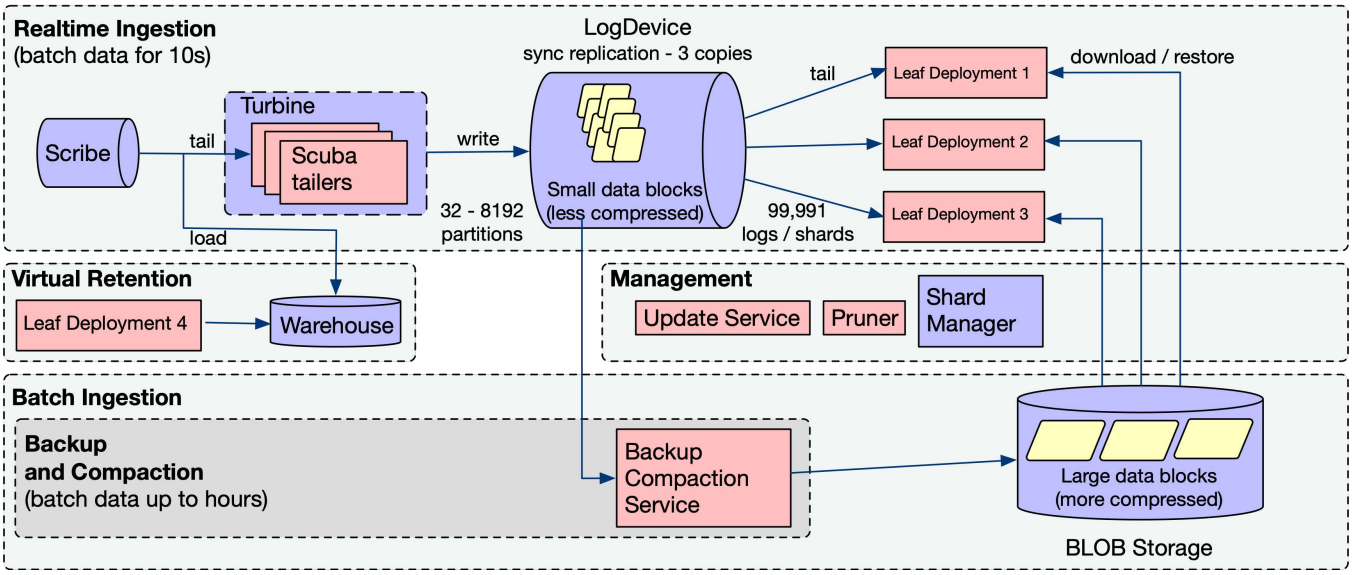

Figure 3: Kraken architecture

The new architecture separates the management and storage of data into independent services (Sections 4.5, 4.6 and 4.7). It multiplexes data management control messages, with data messages (RowBlocks), on the same LogDevice logs.

Messages in LogDevice logs may be one of several types:

(1) **Data**. This type of message carries fresh RowBlocks (with unique IDs; Section 4.2) in their payloads.
(2) **Heartbeat**. This type of message has no payload. They are only used by nodes to detect possible disconnection from the network.
(3) **Compaction**. This type of message is sent by the Backup Compaction service (Section 4.5) to signal multiple smaller RowBlocks have been compacted into a larger block. These messages carry the handle to the larger block in BLOB storage, as well as the list of IDs of the RowBlocks being replaced by the new RowBlock.
(4) **Update**. This type of message is sent by the Update service (Section 4.6) to signal that a RowBlock has to be replaced or dropped. These messages carry the handle to the new RowBlock in BLOB storage, as well as the ID of the old RowBlock being replaced or dropped.

All messages (except heartbeats) are associated with RowBlock unique IDs, so that any data management operations are *idempotent* when consumed.

4.5 Remote Backup and Compaction

An important optimization in both Scuba and Kraken is the compaction of RowBlocks [11]. A compaction operation merges several smaller RowBlocks into one larger RowBlock. Compaction of RowBlocks provides two main benefits:

(1) **Storage efficiency**. Since RowBlocks use dictionary compression [11], the more samples contained in a RowBlock, the better the compression.
(2) **Disk IO**. Since each RowBlock contains more data, fewer RowBlocks need to be loaded in order to process the same query. This reduces the number of disk I/O operations.

Further, the decoupling of compute nodes from backup and compaction operations ensures that data management operations are not in the critical path for serving queries (Section 4.4). Leaf nodes can always serve fresh data without waiting for compaction to occur. This guarantees that queries will return the most recent data.

In Kraken, the Backup and Compaction Service (BCS) is responsible for both backup and compaction of all the RowBlocks. The remote backup on BLOB storage takes place as shown in Fig. 4. After the tailers write partitioned, mapped RowBlocks into its corresponding LogDevice log (Section 4.2), the service consumes them from the logs.

Shard Manager manages the shard assignment to servers that are part of this Service. A BCS node starts up by reading from the LogDevice logs corresponding to the shards it was assigned. In Fig. 4, `Leaf Node 1` reads RowBlocks (R_1, R_2, R_3, R_4) from log 1 and uses them to serve user queries. In parallel, `BCS Node 1` reads the same RowBlocks from log 1, batches RowBlocks from the same dataset partition, compacts them into larger RowBlocks, and uploads them to BLOB storage.

`BCS Node 1` then appends notification N_1 to the same log 1, which is later received by the `Leaf Node 1` (Section 4.4). This notification will provide the handle of the compacted RowBlock on BLOB storage, as well as all *constituent* RowBlocks (R_1, R_2, R_3, R_4) from which the compacted RowBlock was merged. The leaf will download this RowBlock from BLOB storage and replace the constituent RowBlocks with the merged RowBlock R_m.

Since multiple dataset partitions may be mapped to the same shard, BCS creates a queue for each dataset partition it encounters in the log. In Fig. 4 is mapped to Shard 1, so it parses LogDevice **data** messages (Section 4.4) from the corresponding log to extract

the RowBlocks and enqueues them in the corresponding dataset partition queue.

BCS periodically flushes the queues, compacting all RowBlocks in the queue using dictionary compression, and uploads the compacted RowBlock to BLOB storage. The flushing configuration is determined by multiple factors such as the total size of the RowBlocks, the sample count, and the time window to buffer the RowBlocks.

Finally, BCS writes the LogDevice checkpoint under the shard directory as metadata (Section 4.3.2). The checkpoints are used by both BCS itself, as well as leaf nodes, to rebuild their state when they restart. Since all RowBlock operations are marked by unique RowBlock IDs, both backup and compaction operations are idempotent. If a node in BCS fails between writing the compacted RowBlock and persistence of the checkpoint, it can simply redo the same compaction on the same batch of RowBlocks, since appends to individual LogDevice logs are linearizable.

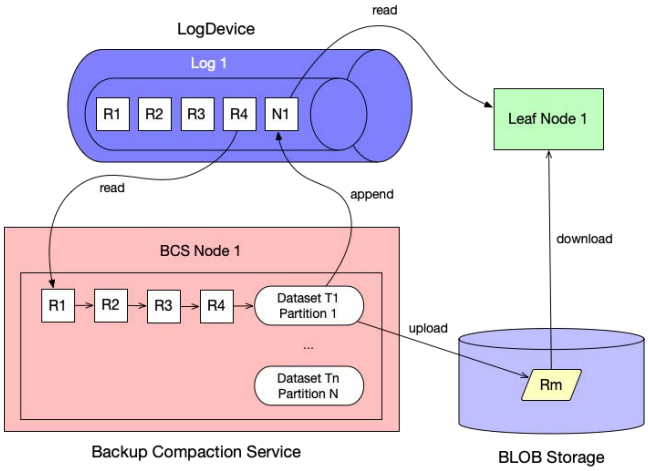

Figure 4: Backup Compaction Service architecture

4.6 Data Management

In general, data ingested in to Kraken is immutable. Kraken does not provide functionality for point deletes or updates. Each Kraken dataset is configured with a user-specified retention, and data outside of this retention is automatically deleted. Additionally, Kraken supports time-range based deletion for both the entire dataset, or for a set of columns. Due to Kraken's columnar storage format, these operations are less computationally expensive to support compared to point operations.

The service responsible for supporting these operations is the Update Service. The Update Service has two primary responsibilities. First, it will perform the operation against the source of truth data stored in Manifold. Second, it will notify Kraken leaves that an update has occurred and that they must re-synchronize their data. This Update Service is also used by a Pruner service that is responsible for ensuring that datasets adhere to their retention policy. The Update Service supports three operations over Kraken datasets.

Delete Time Range. Given a time range, all data in the dataset within that time range is deleted.

Subsample Time Range. Given a time range, all data in the dataset within that time range is subsampled. This is an optional feature for Kraken datasets that allows users to configure a subsampling policy as a part of retention for older data enabling users to observe trends over longer periods of time while storing less data.

Delete Columns. Given a time range and a set of columns, all data in the specified columns within that time range is deleted.

Given the directory structure in Section 4.3.2, these operations can be distributed in parallel across each dataset partition. Once the operation is complete on a given partition, a message is written to the LogDevice log for that specific partition, indicating the RowBlocks that need to be either re-synchronized or deleted from the leaf nodes containing that partition. The Update Service re-uses the same LogDevice log (Section 4.4) as the ingestion pipeline, so this log can be viewed by leaf nodes as an ordered change log for all modifications to the dataset partition.

4.7 Queryable Storage Nodes

The leaf nodes are the queryable data storage nodes in Kraken. Fig. 3 shows how data is ingested into 3 geo-replicated Kraken leaf deployments. Each global shard is assigned to exactly one leaf node per region so that each regional cluster has exactly one full copy of all Kraken data. These shards are assigned by Shard Manager [20] based on resource requirements of the shards and resource availability of the hosts. A leaf node only subscribes to the LogDevice [21] logs for the shards it owns. Data ingested into the leaf nodes becomes immediately available for queries.

4.7.1 Query Path. A user query first reaches a root node, which is a specialized compute node for query distribution and result set aggregation. A root node maintains up-to-date copies of datasets partitioning configuration, and shard assignments. Upon receiving the query, it first retrieves the dataset name, then it will create $N_{Partitions}$ sub-requests, one per dataset partition. By performing the deterministic mapping from table partitions to shards (described in Section 4.3.1) and looking up shard assignments, the root node will find the $N_{Partitions}$ destination leaf nodes containing all data in the requested dataset. The root node waits for partial result sets to be returned from the leaf nodes, aggregating them into the final result set, and returns it to the user.

While the query architecture is largely retained from legacy Scuba, the stronger consistency guarantees afforded by Kraken enabled much finer-grained fan-out control in the aggregation tree. This allowed us to decrease fan-out factor for smaller datasets to reduce the effect of stragglers, and increase the factor for larger datasets to increase parallelism.

4.7.2 Shard data bootstrap. During shard assignment, as part of load balancing or fail-over (described in Section 4.1.5), a leaf node will have to bootstrap a local copy of the data corresponding to the assigned shard. To do this, leaf nodes download a snapshot of data from BLOB storage and subsequently resume tailing from the appropriate LogDevice log at the appropriate LSN.

4.8 Nessie: interface with the Warehouse

Each table in Scuba usually comes with a relatively short retention to store the most recent data. However, there are use cases where

years of data is required to keep for investigation or analysis. To overcome the retention limits and leverage the power of Meta's Data Warehouse (e.g., batch processing framework, rich SQL semantics), we built and deployed alongside Kraken a feature called *virtual retention* (codenamed *Nessie*, see also Fig. 3). Nessie was designed and implemented to extend Kraken by enabling interoperability between Kraken and the Warehouse. Specifically, the data stream from a dataset's category in Scribe is converted into the Warehouse's data format and dumped into each table regularly. After data lands into the Warehouse, users are able to access the dataset using Scuba's UI. Virtual retention enables monitoring and analysis on out of retention Kraken data in a transparent way. Specifically, when a query arrives at the Kraken proxy, it is split into two queries automatically. The first query goes to Kraken deployments and accesses the data within the dataset's original retention, whereas the second query is sent to Nessie, a standalone cluster that modified the Kraken leaf code to read directly from Warehouse storage. After the proxy gets the results from both queries, it merges these two results into one and the Scuba UI presents the chart to the user designating the parts that were retrieved from the extended retention.

5 PRODUCTIONIZATION

Since Scuba is a production system used by thousands daily and heavily relied on by various automation, it is simply not an option to shut down Scuba while we roll out the new components. In this section, we describe how we prototyped, deployed, and rolled out the new Kraken architecture, with zero user-facing downtime.

The deployment of the backend was relatively straightforward: we simply deployed the new binary in a different datacenter accessible only by the developers. Similarly, new auxiliary services, such as the Backup Service (Section 4.5), are deployed as standalone jobs.

Moving the data onto the new deployment, however, required careful planning and orchestration. The following sections elaborate on how we migrated the ingestion pipeline and the data. They also expand on how we validated the new architecture as it is being deployed, so it would meet the users' expectations when the legacy system was decommissioned.

5.1 Global Tailer Migration

Prior versions of Scuba had enhanced data availability through the addition of new Scuba deployments performing independent ingestion and data storage. As previously noted (Section 3.2), this led to divergent views of data between deployments with long term user-facing inconsistencies and operational overhead. To address this, a core goal for the Kraken migration was the move to a single-writer model, with a single set of global tailer jobs distributed across multiple geographically separated data center regions. A single set of tailer jobs processes ingested data and writes RowBlocks to the staging area and backup, with leaf hosts hosting local replicas of these RowBlocks across Scuba deployments. Supporting this migration, while continuing to serve existing Scuba users, required focusing on a few key concerns, so that Scuba could continue meeting targets for reliability without any interruption in service.

5.2 Performance

Kraken shifted the Scuba ingestion stack from a region-local processing model to a fundamentally distributed processing model, in which cross-datacenter network latency can impact overall throughput. Global tailers perform cross-DC operations in two cases for writes:

(1) Appending RowBlocks to LogDevice logs for downstream consumption (Section 4.4). This incurs cross-region replication of the RowBlock data in LogDevice.
(2) Updating checkpoint storage in ZippyDB, since writes are replicated to three regions.

As an early test of the impact of cross-region latency on Scuba tailers, one of the existing Scuba deployments was migrated to ZippyDB storage with replication distributed across three data center regions. This provided on-going validation that tailer processing was not bound by checkpoint latency under a full production workload and fluctuations. RowBlock writing to LogDevice was added later, with tuning to maximize batching of the writes for throughput.

5.3 Data Migration

Another obstacle of a seamless migration to the new architecture was the physical movement of data. While many Scuba datasets limit data retention times to the orders of days or weeks, some datasets require retention lasting months or years. Fully migrating all usage to Kraken required supporting both retention categories. For datasets with shorter retention, we could simply write **fresh** data to both Kraken and legacy Scuba, and then move over the read path once the Kraken data met the configured retention time. For datasets whose retention exceeds the timeline for productionization, it was necessary to copy any **historical** data directly from storage from legacy Scuba.

5.3.1 Fresh data. To avoid data duplication and minimize data loss, this meant that both Kraken and existing Scuba tailers must see the same view of the ingested data, which, due to limits on ordering guarantees for Scribe readers, meant that a single set of tailers must populate both. One of the existing Scuba deployments was selected and moved to a new global tailer job, writing RowBlocks to both LogDevice for Kraken and the existing Scuba leaf nodes.

5.3.2 Historical data. Since fresh data becomes indistinguishable from historical data once it's persisted on Scuba, naively copying from legacy Scuba would result in duplicates. We introduced a versioning mechanism in legacy Scuba to mark any RowBlocks that existed prior to the copying. The version number of a RowBlock is stored as part of the header of the RowBlock.

Prior to copying a dataset, the tailers will be instrumented to write any new RowBlocks with a new version. This means any un-versioned RowBlock, or RowBlock with lower versions already persisted by legacy Scuba, will be distinct from the ones with the newer version. During copying, any RowBlock with the new version is ignored. This ensures that historical data is not duplicated in the migration of historical data.

5.4 Fault Tolerance

At internet scale, various types of disasters (natural, human, software, etc.) that take entire datacenters offline may occur. As Meta's main debugging tool, Scuba faces a unique challenge: often times, other teams that maintain the very infrastructure on which Scuba depends, rely on Scuba to troubleshoot their systems.

With a single source of truth, the Kraken architecture allows for more strategic redundancy. While the backend is still fully deployed redundantly in 3 datacenter regions, other components are more geographically distributed. For example:

(1) Scuba tailers (Section 4.2) are distributed across five datacenters. Under the previous Scuba architecture, a datacenter outage would render the entire Scuba deployment in the impacted datacenter offline. In Kraken, the impacted jobs in the datacenter will be moved automatically to other datacenters by Shard Manager (Section 4.1.5).
(2) Tailer state is persisted in a ZippyDB deployment replicated across five datacenters with strong consistency.
(3) Scuba's LogDevice cluster is geographically distributed across five datacenters (Section 4.1.3).

Throughout the productionization of the Kraken architecture, we routinely employed two types of testing to validate the fault tolerance of the system: (1) drain tests, and (2) Chaos Monkey-style [4] fault injection testing.

Drain tests artificially introduce geographical shard placement restrictions. With the help from Shard Manager (Section 4.1.5), during a drain test, we force shards to evacuate from a datacenter. This ensures that the system has enough **capacity** in other datacenters to accept the shards. At the same time, we monitor the system's throughput to make sure that it does not degrade beyond users' expectations.

Fault injection testing introduces faults at tactical places in the system. Several types of faults were used to ensure that Scuba is resilient under non-critical failures, and fails gracefully when critical dependencies fail. Faults include network outages (similar to natural disasters), and application level failures (similar to software bugs in systems Scuba depends on).

6 EXPERIMENTS

In this section we present comparisons of Kraken with the original Scuba architecture. Additionally, we present quantitative measurements of new capabilities offered by Kraken.

We selected a total of 204 datasets from different size classes, and analyzed the system's performance on these datasets using 100% of production traffic from employees over a period of two consecutive weeks. The query traffic comprised a mix of automated and human-driven workloads.

The typical datasets in each size class and their characteristics are listed in Table 1. 200 of the datasets were selected at random from size class T_5, and one dataset was selected from each of size classes T_1 to T_4.

6.1 Query

We first examined the query latency of Kraken compared to the legacy architecture. Stronger consistency guarantees made possible by the single source of truth in Kraken enabled finer-grained control on fan-out factors in the aggregation tree. This allowed us to minimize the effect of stragglers for datasets smaller than T_3, and increase parallelism for datasets in larger size classes. Both factors contributed to P50 latency improvements ranging from 19.6% to 71.3% when comparing Kraken's query performance against that of the legacy Scuba. Fig. 5 shows the P50 query latency of the Kraken architecture compared to legacy Scuba for datasets of in each size class.

Table 1: Datasets used in the experimental study

Dataset	Size (GB)	# Shards	# Records	# Columns
T_1	150,000	8,192	1.27 T	3,500+
T_2	50,000	8,192	869 B	578
T_3	1,000	1,024	46 B	88
T_4	500	128	17 B	12
T_5	50	32	1 B	10

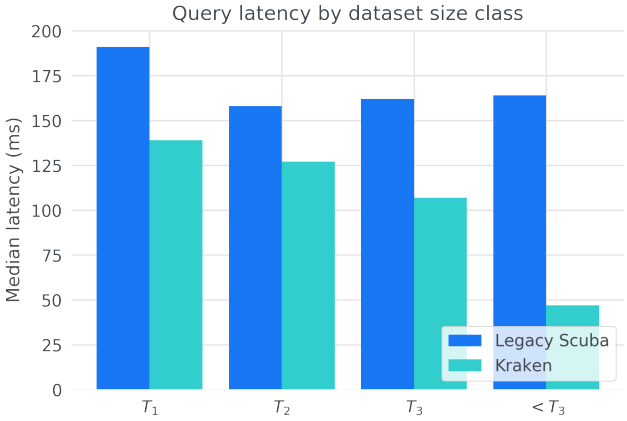

Figure 5: P50 query latency for datasets in each size class

Another important consideration for aggregation-tree based data processing systems (Section 4.7) is the network bandwidth among the nodes. Datacenter operators need to balance utilization of the network bandwidth among competing services. The new architecture achieves a 50% reduction in network utilization, due to the reduction in the fan-out factors for datasets smaller than T_3. Fig. 6 compares the bytes used to communicate among nodes in the aggregation tree for a typical two-week period for both architectures.

6.2 Ingestion

A crucial service level indicator for Scuba's ingestion pipeline is the freshness of data served to users, as most troubleshooting and debugging use cases require realtime access to fresh data. With a more complex ingestion pipeline, we built a facility to trace the end-to-end ingestion latency for datasets.[4]

[4]End-to-end ingestion latency is defined as the difference between the time Scribe (Section 4.1.1) persists the event and the time Scuba leaf nodes persist the event.

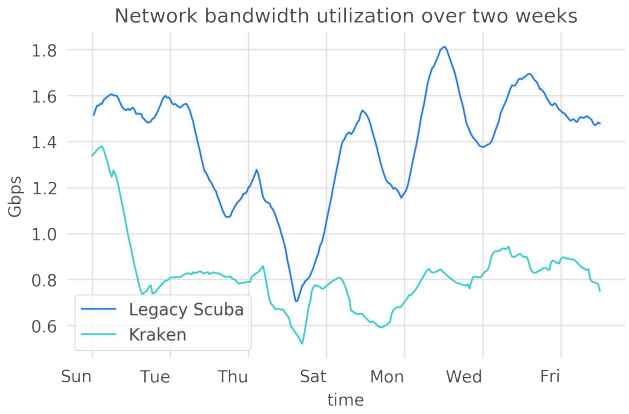

Figure 6: Network utilization by aggregation tree communication comparison

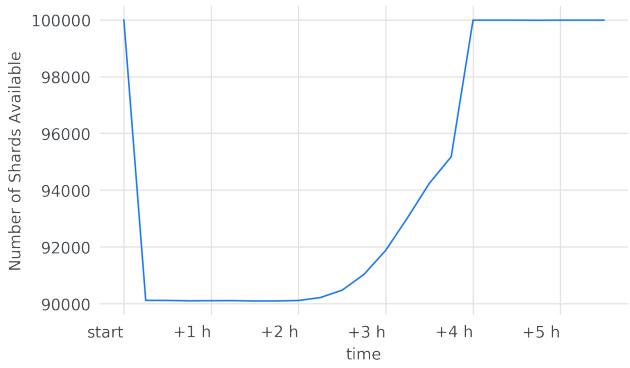

Figure 7: Recovery progress from fault injection testing

Overall, the average ingestion latency for the system is 8.22 s, whereas the P50 latency is 7.8 s. Table 2 shows the average and different percentiles of ingestion latency for datasets in the system.

Table 2: Ingestion latency for datasets in each size class

Dataset	Average	P50	P99
T_1	17.5 s	17.9 s	24.9 s
T_2	13.8 s	13.7 s	20.7 s
T_3	11.4 s	11.4 s	18.2 s
$< T_3$	17.3 s	16.8 s	20.7 s
All datasets in system	8.22 s	7.8 s	16.1 s

6.3 Recovery

An important feature of the Kraken architecture is the ability to restore missing data from source of truth. (Section 4.5) The time to recover missing data is therefore a key metric we measure by running routine fault injection tests (Section 5.4). One such test simulates complete network outage on 10% of a deployment, then measures the time taken for other servers in the same deployment to restore backups from the source of truth, and make them available for reading.

Fig. 7 shows the recovery process for one of the fault injection tests that took 10% of the leaf nodes in a deployment offline. The y-axis shows the number of shards available for reads. At the beginning of the test, approximately 10% of the shards were lost. The system gradually recovered from the failure by moving the shards to other servers.

Over 6 runs of the same test in different deployments across a period of 6 months, the average recovery time from losing 10% of servers is approximately 180 minutes.

7 RELATED WORK

Both Rockset [30] and Napa [3] offload compaction to perform optimizations at write time via creation of materialized views, which improves read-time performance at the cost of additional compute and IO. Napa takes this a step further and enables clients to trade off these write-side optimizations based on their requirements for cost, freshness and query performance. Unlike Napa, Kraken does not expose these concepts as client modifiable configuration parameters and instead elects to enable users to explicitly constrain their workloads by allowing them to configure an optional sampling policy at ingest time as discussed in [2].

Druid [32], Pinot [15], and Clickhouse [7] all support various forms of streaming and batch ingestion with ingestion speeds approaching real-time to nodes which share the dual responsibility of data storage and compute. Unlike these systems, data ingested into Kraken can be simultaneously ingested into multiple systems like Meta's Data Warehouse via a singular ingestion API if desired. This is done to take advantage of the different characteristics of the various systems and query results from the various systems can be stitched together and presented in a single UI.

Both Procella [5] and Kraken append data into logs for subsequent processing (i.e compaction) at ingest time. To improve data freshness, Procella opts for a scheme where data is dual written to best effort durable in-memory buffers and durable remote storage. Data in the in-memory buffers is made immediately queryable while data in remote storage is queryable after the background process of compaction is complete. Kraken on the other hand persists all data to durable storage after a small batching window before allowing data to be queryable.

Dremel's [24] original tree aggregation architecture is the closest system to Scuba's/Kraken's leaf architecture for serving queries. Unlike realtime monitoring systems, Dremel does not have strict targets for freshness and query performance and instead focused on interactive performance of large datasets with full support for nested data. BigQuery [12], which was originally based on Dremel, later expanded the engine to support full joins and other features commonly found in cloud Data Warehouses (UDFs, MVs, etc.).

Modern SQL engines (Impala [17], Presto [28], F1 Query [27]) paved the way for decoupling compute from storage and allowed scaling data processing to exabytes of data. Supporting the full spec of SQL means that these engines need to employ a sophisticated optimizer and also need to be able to redistribute data on the fly to support any type of join and aggregation which limits the scalability

of the clusters. Systems of this nature are present in Meta's Data Warehouse and data in Kraken is made queryable with these engines via the Nessie interface.

8 CONCLUSION

In this paper we described the design, implementation, and deployment of Meta's next generation realtime monitoring and analytics platform, Kraken, which succeeded Scuba [2] after more than a decade in production. We discussed the unique considerations of the design space for modern analytical data processing systems at scale and motivated the need for revisiting the original tradeoffs behind Scuba's design. The new Kraken architecture handled increased scale by improving on system characteristics such as eliminating best effort data durability in the prior architecture. We separated the storage management subsystem from the leaf nodes that also serve queries and introduced a new deterministic partitioning scheme which made it possible for the system to have a single source of truth, that is simultaneously not on the critical path of query serving. This enabled the system to be available and real-time as well as mitigated undesirable characteristics of the previous architecture such as view divergence between geographically isolated deployments.

The Kraken rearchitecture was accomplished in place with no user-visible down time via careful planning and orchestration to migrate system components such as the ingestion pipeline without planned data loss. With Kraken in production, we have observed immediate and tangible improvements to system fault tolerance and query performance while still respecting tolerable bounds of client observed data freshness. Over the long term, we anticipate that the Kraken architecture will enable further improvements to be made to the system such as enabling knobs for tuning data availability at query time as well as enabling more flexible and efficient topologies.

ACKNOWLEDGMENTS

Rearchitecting and fully deploying to production Meta's next generation platform for realtime monitoring and analytics has been a long journey and it would not have been possible without the help of many people including current and past team members, as well as our partner teams. We are also grateful to Gautam Shanbhag, Karen Pieper, and Munir Bandukwala for their continued support.

REFERENCES

[1] Daniel Abadi, Peter Boncz, Stavros Harizopoulos, Stratos Idreos, and Samuel Madden. 2013. The Design and Implementation of Modern Column-Oriented Database Systems. *Foundations and Trends® in Databases* 5, 3 (2013), 197–280. https://doi.org/10.1561/1900000024

[2] Lior Abraham, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, David Reiss, Subbu Subramanian, Janet L. Wiener, and Okay Zed. 2013. Scuba: Diving into Data at Facebook. *Proc. VLDB Endow.* 6, 11 (aug 2013), 1057–1067. https://doi.org/10.14778/2536222.2536231

[3] Ankur Agiwal, Kevin Lai, Gokul Nath Babu Manoharan, Indrajit Roy, Jagan Sankaranarayanan, Hao Zhang, Tao Zou, Min Chen, Jim Chen, Ming Dai, Thanh Do, Haoyu Gao, Haoyan Geng, Raman Grover, Bo Huang, Yanlai Huang, Adam Li, Jianyi Liang, Tao Lin, Li Liu, Yao Liu, Xi Mao, Maya Meng, Prashant Mishra, Jay Patel, Rajesh S R, Vijayshankar Raman, Sourashis Roy, Mayank Singh Shishodia, Tianhang Sun, Justin Tang, Junichi Tatemura, Sagar Trehan, Ramkumar Vadali, Prasanna Venkatasubramanian, Joey Zhang, Kefei Zhang, Yupu Zhang, Zeleng Zhuang, Goetz Graefe, Divyakanth Agrawal, Jeff Naughton, Sujata Sunil Kosalge, and Hakan Hacıgümüş. 2021. Napa: Powering Scalable Data Warehousing with Robust Query Performance at Google. *Proceedings of the VLDB Endowment (PVLDB)* 14 (12) (2021), 2986–2998.

[4] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. 2015. Chaos Monkey: Increasing SDN Reliability through Systematic Network Destruction. *SIGCOMM Comput. Commun. Rev.* 45, 4 (aug 2015), 371–372. https://doi.org/10.1145/2829988.2790038

[5] Biswapesh Chattopadhyay, Priyam Dutta, Weiran Liu, Ott Tinn, Andrew McCormick, Aniket Mokashi, Paul Harvey, Hector Gonzalez, David Lomax, Sagar Mittal, Roee Aharon Ebenstein, Nikita Mikhaylin, Hung ching Lee, Xiaoyan Zhao, Guanzhong Xu, Luis Antonio Perez, Farhad Shahmohammadi, Tran Bui, Neil McKay, Vera Lychagina, and Brett Elliott. 2019. Procella: Unifying serving and analytical data at YouTube. *PVLDB* 12(12) (2019), 2022–2034. https://dl.acm.org/citation.cfm?id=3360438

[6] Yijou Chen, Richard L. Cole, William J. McKenna, Sergei Perfilov, Aman Sinha, and Eugene Szedenits. 2009. Partial Join Order Optimization in the Paraccel Analytic Database. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data* (Providence, Rhode Island, USA) *(SIGMOD '09)*. Association for Computing Machinery, New York, NY, USA, 905–908. https://doi.org/10.1145/1559845.1559945

[7] ClickHouse. 2022. ClickHouse - Fast Open-Source OLAP DBMS. https://web.archive.org/web/20220621010451/https://clickhouse.com/.

[8] Benoit Dageville, Thierry Cruanes, Marcin Zukowski, Vadim Antonov, Artin Avanes, Jon Bock, Jonathan Claybaugh, Daniel Engovatov, Martin Hentschel, Jiansheng Huang, Allison W. Lee, Ashish Motivala, Abdul Q. Munir, Steven Pelley, Peter Povinec, Greg Rahn, Spyridon Triantafyllis, and Philipp Unterbrunner. 2016. The Snowflake Elastic Data Warehouse. In *Proceedings of the 2016 International Conference on Management of Data* (San Francisco, California, USA) *(SIGMOD '16)*. Association for Computing Machinery, New York, NY, USA, 215–226. https://doi.org/10.1145/2882903.2903741

[9] Elasticsearch. 2022. Elasticsearch: The Official Distributed Search & Analytics Engine | Elastic. https://web.archive.org/web/20220602153647/https://www.elastic.co/elasticsearch/.

[10] Eric Friedman, Peter Pawlowski, and John Cieslewicz. 2009. SQL/MapReduce: A Practical Approach to Self-Describing, Polymorphic, and Parallelizable User-Defined Functions. *Proc. VLDB Endow.* 2, 2 (aug 2009), 1402–1413. https://doi.org/10.14778/1687553.1687567

[11] Aakash Goel, Bhuwan Chopra, Ciprian Gerea, Dhruv Mátáni, Josh Metzler, Fahim Ul Haq, and Janet Wiener. 2014. Fast Database Restarts at Facebook. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 541–549. https://doi.org/10.1145/2588555.2595642

[12] Google. 2022. BigQuery: Cloud Data Warehouse | Google Cloud. https://web.archive.org/web/20220602225903/https://cloud.google.com/bigquery/.

[13] Anurag Gupta, Deepak Agarwal, Derek Tan, Jakub Kulesza, Rahul Pathak, Stefano Stefani, and Vidhya Srinivasan. 2015. Amazon Redshift and the Case for Simpler Data Warehouses. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data* (Melbourne, Victoria, Australia) *(SIGMOD '15)*. Association for Computing Machinery, New York, NY, USA, 1917–1923. https://doi.org/10.1145/2723372.2742795

[14] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. 2010. ZooKeeper: Wait-Free Coordination for Internet-Scale Systems. In *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference* (Boston, MA) *(USENIXATC'10)*. USENIX Association, USA, 11.

[15] Jean-François Im, Kishore Gopalakrishna, Subbu Subramaniam, Mayank Shrivastava, Adwait Tumbde, Xiaotian Jiang, Jennifer Dai, Seunghyun Lee, Neha Pawar, Jialiang Li, and Ravi Aringunram. 2018. Pinot: Realtime OLAP for 530 Million Users. In *Proceedings of the 2018 International Conference on Management of Data* (Houston, TX, USA) *(SIGMOD '18)*. Association for Computing Machinery, New York, NY, USA, 583–594. https://doi.org/10.1145/3183713.3190661

[16] Manolis Karpathiotakis, Dino Wernli, and Milos Stojanovic. 2020. Scribe: Transporting petabytes per hour via a distributed, buffered queueing system. https://engineering.fb.com/2019/10/07/data-infrastructure/scribe/

[17] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Casey Ching, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Ippokratis Pandis, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-Milne, and Michael Yoder. 2015. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *Seventh Biennial Conference on Innovative Data Systems Research, CIDR 2015, Asilomar, CA, USA, January 4-7, 2015, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2015/Papers/CIDR15_Paper28.pdf

[18] Jay Kreps, Neha Narkhede, and Jun Rao. 2011. Kafka: a Distributed Messaging System for Log Processing. In *Proceedings of NetDB'11 the 6th Workshop on Networking Meets Databases*. Association for Computing Machinery, New York, NY, USA. https://books.google.com/books?id=QRecswEACAAJ

[19] Andrew Lamb, Matt Fuller, Ramakrishna Varadarajan, Nga Tran, Ben Vandiver, Lyric Doshi, and Chuck Bear. 2012. The Vertica Analytic Database: C-Store 7 Years Later. *Proc. VLDB Endow.* 5, 12 (aug 2012), 1790–1801. https://doi.org/10.14778/2367502.2367518

[20] Sangmin Lee, Zhenhua Guo, Omer Sunercan, Jun Ying, Thawan Kooburat, Suryadeep Biswal, Jun Chen, Kun Huang, Yatpang Cheung, Yiding Zhou, Kaushik Veeraraghavan, Biren Damani, Pol Mauri Ruiz, Vikas Mehta, and Chunqiang Tang. 2021. Shard Manager: A Generic Shard Management Framework for Geo-Distributed Applications. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 553–569. https://doi.org/10.1145/3477132.3483546

[21] Mark Marchukov. 2018. LogDevice: A distributed data store for logs. https://engineering.fb.com/2017/08/31/core-data/logdevice-a-distributed-data-store-for-logs/

[22] Sarang Masti. 2021. How we built a general purpose key value store for Facebook with zippydb. https://engineering.fb.com/2021/08/06/core-data/zippydb/

[23] Yuan Mei, Luwei Cheng, Vanish Talwar, Michael Y. Levin, Gabriela Jacques-Silva, Nikhil Simha, Anirban Banerjee, Brian Smith, Tim Williamson, Serhat Yilmaz, Weitao Chen, and Guoqiang Jerry Chen. 2020. Turbine: Facebook's Service Management Platform for Stream Processing. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. 1591–1602. https://doi.org/10.1109/ICDE48307.2020.00141

[24] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, and Theo Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. In *Proc. of the 36th Int'l Conf on Very Large Data Bases*. 330–339. http://www.vldb2010.org/accept.htm

[25] Guido Moerkotte. 1998. Small Materialized Aggregates: A Light Weight Index Structure for Data Warehousing. In *VLDB*.

[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. *Proc. VLDB Endow.* 8, 12 (aug 2015), 1816–1827. https://doi.org/10.14778/2824032.2824078

[27] Bart Samwel, John Cieslewicz, Ben Handy, Jason Govig, Petros Venetis, Chanjun Yang, Keith Peters, Jeff Shute, Daniel Tenedorio, Himani Apte, Felix Weigel, David Wilhite, Jiacheng Yang, Jun Xu, Jiexing Li, Zhan Yuan, Craig Chasseur, Qiang Zeng, Ian Rae, Anurag Biyani, Andrew Harn, Yang Xia, Andrey Gubichev, Amr El-Helw, Orri Erling, Zhepeng Yan, Mohan Yang, Yiqun Wei, Thanh Do, Colin Zheng, Goetz Graefe, Somayeh Sardashti, Ahmed M. Aly, Divy Agrawal, Ashish Gupta, and Shiv Venkataraman. 2018. F1 Query: Declarative Querying at Scale. *Proc. VLDB Endow.* 11, 12 (aug 2018), 1835–1848. https://doi.org/10.14778/3229863.3229871

[28] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *2019 IEEE 35th International Conference on Data Engineering (ICDE)*. 1802–1813. https://doi.org/10.1109/ICDE.2019.00196

[29] Chunqiang Tang, Thawan Kooburat, Pradeep Venkatachalam, Akshay Chander, Zhe Wen, Aravind Narayanan, Patrick Dowell, and Robert Karl. 2015. Holistic Configuration Management at Facebook. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 328–343. https://doi.org/10.1145/2815400.2815401

[30] Venkat Venkataramani and Dhruba Borthakur. 2022. Rockset: Real-time analytics at Cloud Scale. https://web.archive.org/web/20220602005913/https://rockset.com/.

[31] Florian M. Waas. 2008. Beyond Conventional Data Warehousing - Massively Parallel Data Processing with Greenplum Database - (Invited Talk). In *Business Intelligence for the Real-Time Enterprise - Second International Workshop, BIRTE 2008, Auckland, New Zealand, August 24, 2008, Revised Selected Papers (Lecture Notes in Business Information Processing)*, Malú Castellanos, Umeshwar Dayal, and Timos Sellis (Eds.), Vol. 27. Springer, 89–96. https://doi.org/10.1007/978-3-642-03422-0_7

[32] Fangjin Yang, Eric Tschetter, Xavier Léauté, Nelson Ray, Gian Merlino, and Deep Ganguli. 2014. Druid: A Real-Time Analytical Data Store. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data* (Snowbird, Utah, USA) *(SIGMOD '14)*. Association for Computing Machinery, New York, NY, USA, 157–168. https://doi.org/10.1145/2588555.2595631

[33] Matei Zaharia, Ali Ghodsi 0002, Reynold Xin, and Michael Armbrust. 2021. Lakehouse: A New Generation of Open Platforms that Unify Data Warehousing and Advanced Analytics. In *11th Conference on Innovative Data Systems Research, CIDR 2021, Virtual Event, January 11-15, 2021, Online Proceedings*. www.cidrdb.org. http://cidrdb.org/cidr2021/papers/cidr2021_paper17.pdf