new relic

Architecting for the Future:

Unveiling the Future of Data Ingestion, Routing, and Storage Architecture for Observability.

AUTHOR:

Kranthi Erusu, Group Vice President, Engineering, New Relic

Technical White Paper / February 2025

Introduction

The complexity of modern enterprise systems has made observability essential for effective operations. As organizations adopt cloud-native, hybrid, and distributed infrastructures, the ability to process and analyze telemetry data in real time is critical for maintaining system reliability, performance, and user satisfaction. These evolving demands present significant challenges, including the need to ingest, store, and query diverse data types at scale, while optimizing cost and ensuring operational efficiency.

To address these challenges, the New Relic database (NRDB) was developed as a cloud-native, multi-tenant database optimized for observability workloads. NRDB combines high-performance querying, fault isolation, and cost-efficient scalability to meet the requirements of dynamic business environments. Its architecture enables seamless analysis of diverse data types, including immutable telemetry data, mutable business data, and external sources, all within a unified framework. These capabilities allow NRDB to deliver the reliability and performance needed for real-time analytics, supporting the growing requirements of modern data ecosystems.

This white paper traces the evolution of NRDB, emphasizing how its design has evolved to meet the demands of real-time analytics at scale. Through advancements in multi-tenant architecture and efficient querying, NRDB exemplifies how modern databases are engineered to address the challenges of today's dynamic data environments.

New Relic Database Architecture

Evolution from 2014 to Scalability

In 2014, the NRDB architecture embarked on a path inspired by the Dremel paper, aiming to create a massively distributed database system for analytics over traditional architectures like simple file storage. At that time, no analytical engines could produce query results in sub-seconds. The state of the ecosystem then consisted of distributed query engines that generated results in minutes when submitted to them as batch jobs. Observability (o11y) use cases, however, required more "real-time" query results.

To meet these requirements, the NRDB architecture was designed with three core goals: ease of use, blazing-fast performance, and a unified platform for diverse data types.

Ease of use
The NRDB architecture was designed with user-friendliness in mind. It eliminated the need for predefined schemas, index requirements, and tables, simplifying the data analytics process and allowing users to focus on deriving insights rather than managing database structures.

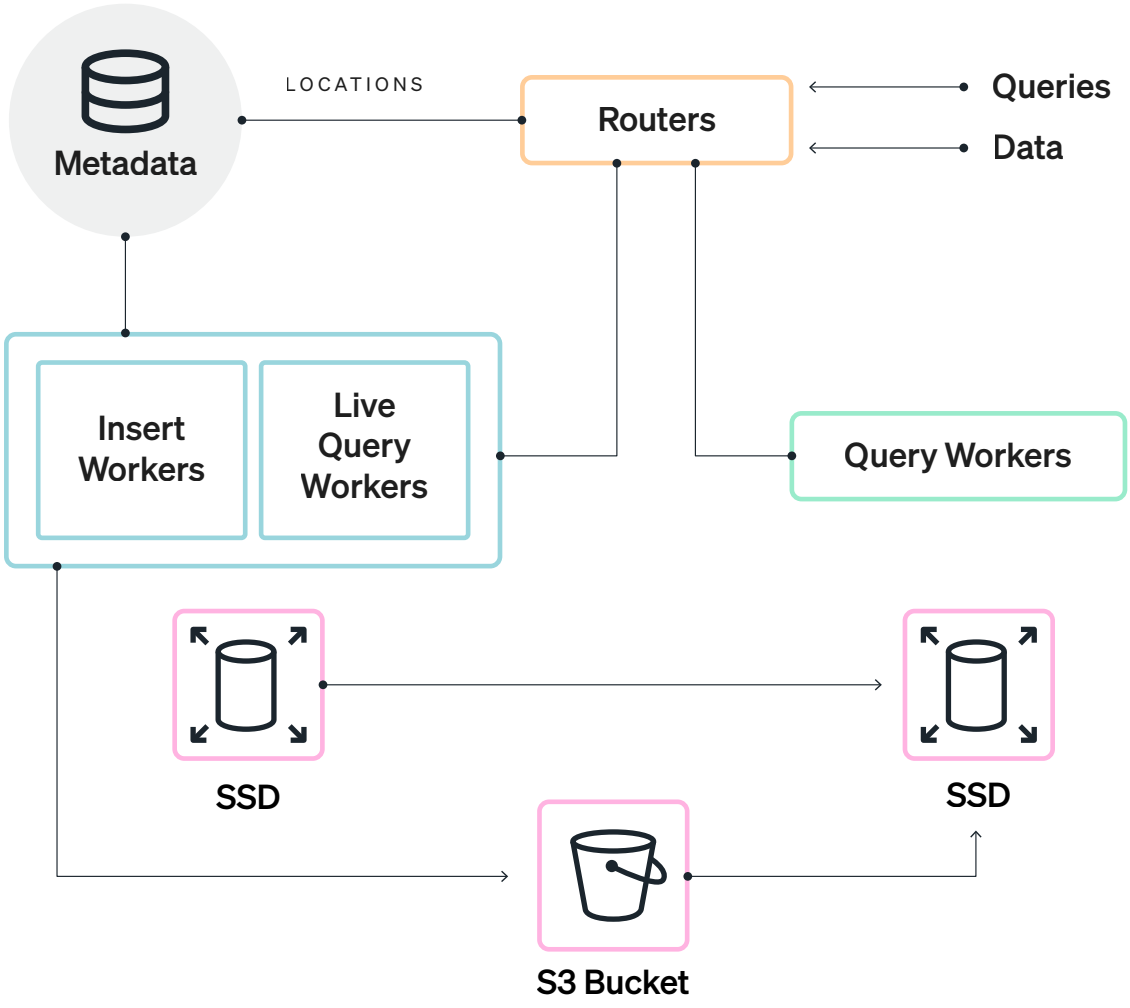
Speed
Speed was a critical factor, with the architecture aiming for sub-second "time to glass" and a median query performance of 50 ms. The goal was to enable users to analyze data through an expressive SQL-inspired language that allowed for quick and efficient data retrieval without adding a barrier to learn a new query language.

All-in-one solution
NRDB aimed to be a one-stop solution for querying various data types, including metrics, events, logs, and traces, all from a single location. This holistic approach enabled comprehensive analysis and monitoring across different data streams.

Initial Architecture

The initial setup involved Java virtual machines (JVMs) on a large array of bare metal servers, leveraging solid-state drives (SSDs) crammed into the chassis, with a 60-70% of the cost attributed to storage. The storage costs were ballooned by the practice of storing multiple copies in Amazon Simple Storage Service (S3), which was used for disaster recovery rather than operational quality of service.

The high-level architecture was quite simple, where we had a fleet of workers called insert workers to persist the data into S3 and another fleet of workers called query workers that would serve queries from storage. To ensure consistency between insert and query workers, a metadata layer was implemented, enabling customers to query data as it was being ingested.



Kafka for ingest processing, partitioning, and routing

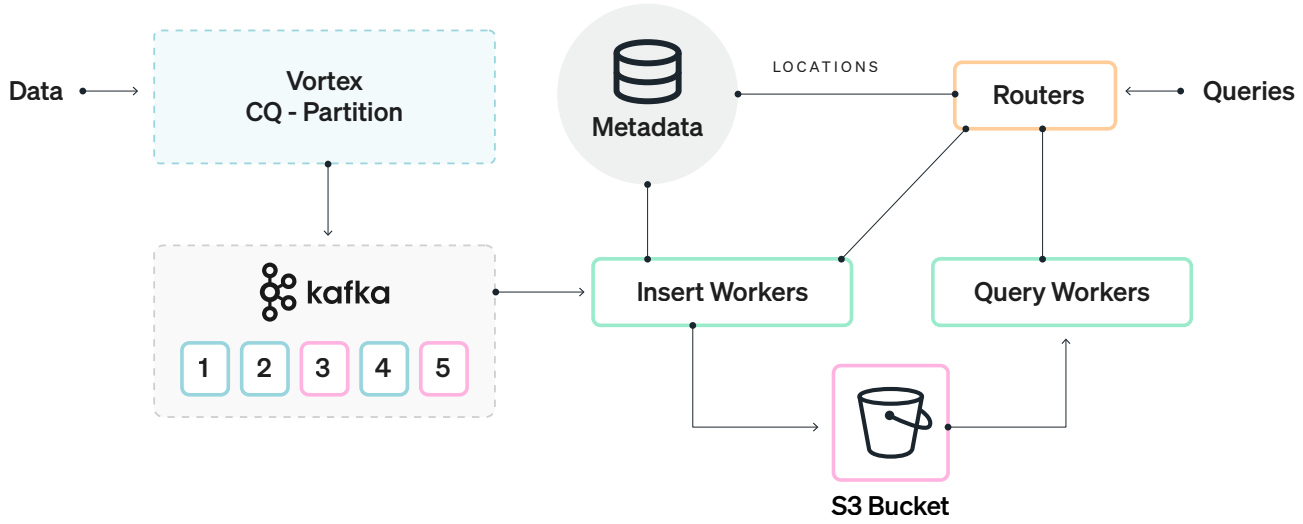
As the platform scaled to accommodate more data and customers, ingest and query workloads began to interfere with each other, and the architecture had to invest in reducing workload interference.

A key strategy was the decoupling of ingest and query operations, utilizing a message queuing and publish-subscribe construct to allow for asynchronous processing without load shedding. Kafka is a key architectural component we took a bet on, from its very early beta versions, to route data and process it along the way to synthesize additional information from telemetry (like entities) before it's sent to the insert worker for persistence.

The system partitioned the data to optimize for two main constraints:

1. Colocate large rungs of similar data within the boundary of a tenant that is likely queried together to gain efficiencies from using columnar storage.
2. Reduce the failure domain for a given customer. A failure in a single broker or an insert worker or any component along the path of the ingest cannot impact all customers.

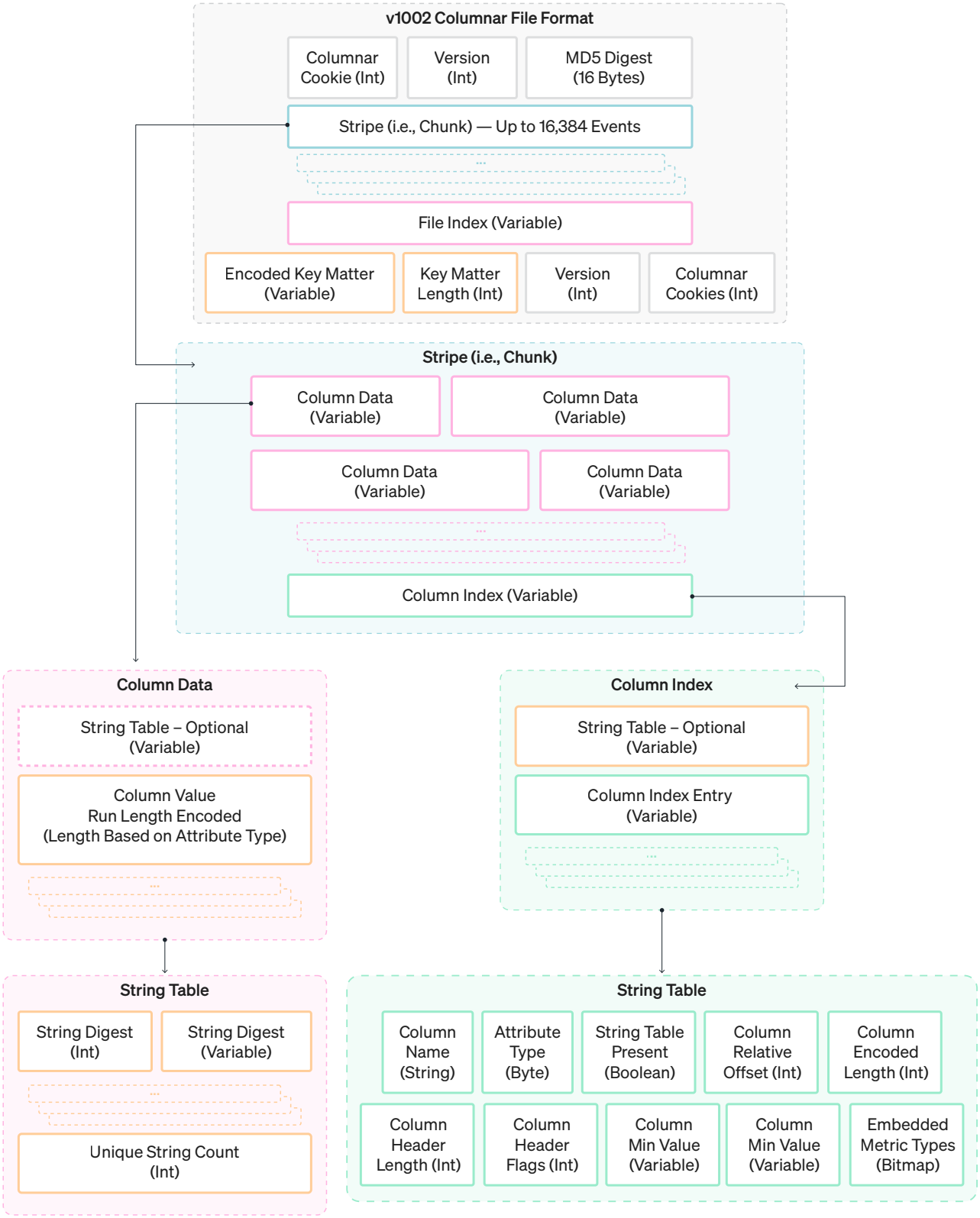
In the diagram below, you see the ingest path receiving data of all types from all customers by a component called Vortex, which applies authentication checks and partitions the data into high-level constructs called New Relic accounts. Then the data is processed and consumed through Kafka before it gets persisted in object stores.

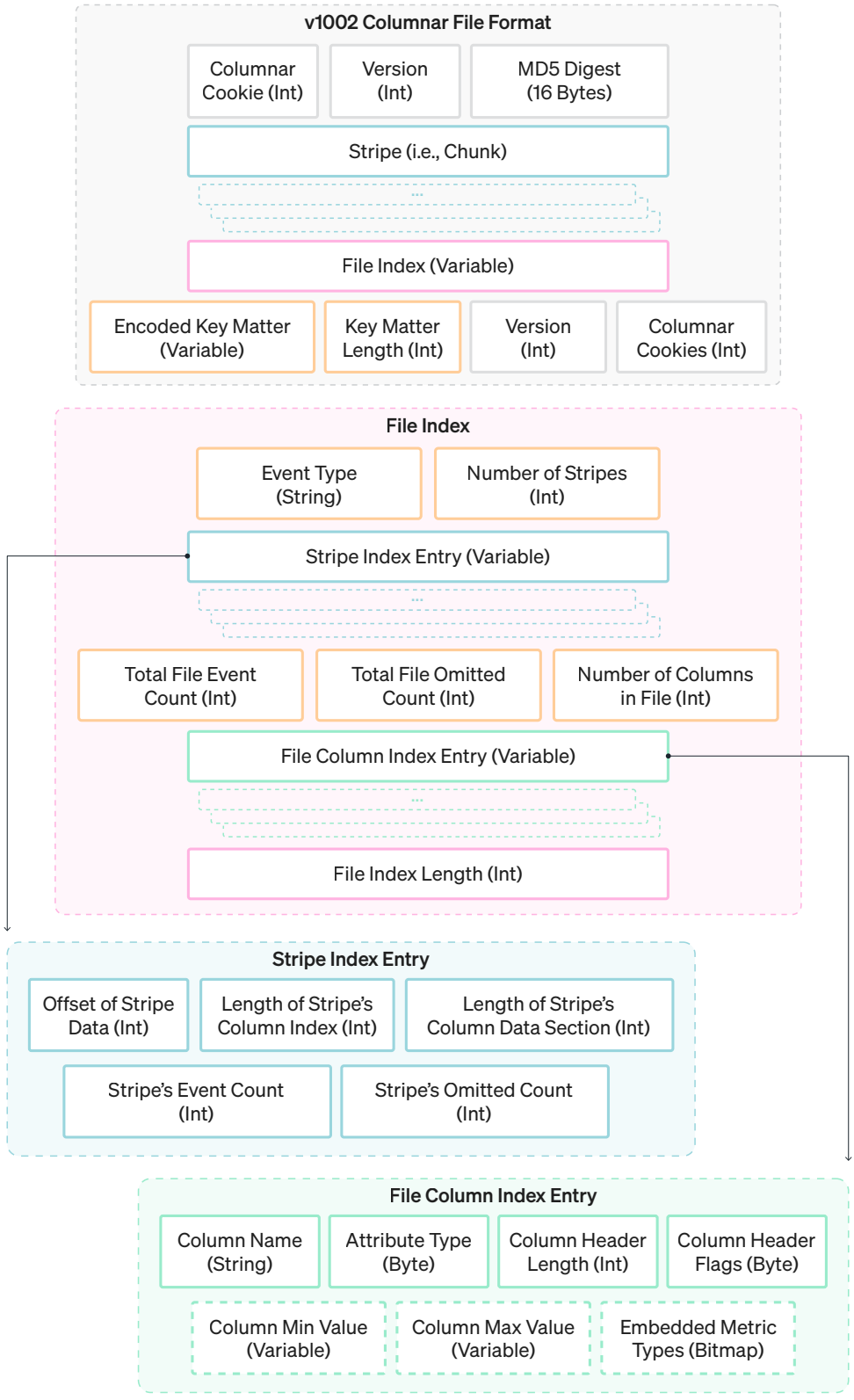


Columnar storage and archive file format

Once data reaches the insert worker, it needs to be stored in a format that is efficient for fast analytical queries. Very early on, we adopted a custom columnar format, known as the "archive file format," as the columnar approach enables efficient data storage and retrieval. Since the data is organized by columns rather than rows, it optimizes analytical queries by minimizing the amount of data read from storage. This approach can lead to performance improvements of up to 200 times for certain analytical workloads.

The diagram below shows the details of our custom columnar format, including how the data is organized and what additional metadata is created to efficiently query the data within the file.




v1002 Columnar File Format
Columnar Cookie (Int)
Version (Int)
MD5 Digest (16 Bytes)
Stripe (i.e., Chunk)
File Index (Variable)
Encoded Key Matter (Variable)
Key Matter Length (Int)
Version (Int)
Columnar Cookies (Int)
File Index
Event Type (String)
Number of Stripes (Int)
Stripe Index Entry (Variable)
Total File Event Count (Int)
Total File Omitted Count (Int)
Number of Columns in File (Int)
File Column Index Entry (Variable)
File Index Length (Int)
Stripe Index Entry
Offset of Stripe Data (Int)
Length of Stripe's Column Index (Int)
Length of Stripe's Column Data Section (Int)
Stripe's Event Count (Int)
Stripe's Omitted Count (Int)
File Column Index Entry
Column Name (String)
Attribute Type (Byte)
Column Header Length (Int)
Column Header Flags (Byte)
Column Min Value (Variable)
Column Max Value (Variable)
Embedded Metric Types (Bitmap)

To meet the growing demands of data analytics and storage, we created a specialized file format called archive file format. This format is tailored for efficiency, flexibility, and performance in analytical workloads. Below are its key features and advantages.

Key features

- **Optimized for analytical workloads:** The archive file format is specifically engineered to handle complex analytical tasks. It supports high-performance queries and large-scale data processing, making it ideal for environments that require rapid insights from vast datasets.
- **Efficient compression and encoding:** Advanced compression techniques like LZ4, Zstd are employed to significantly reduce file sizes without sacrificing data quality. This efficiency not only saves storage space but also improves data transfer speeds.
- **Diverse encoding techniques:** To enhance data efficiency, the archive file format utilizes a variety of encoding methods:
  - **Dictionary encoding:** Reduces redundancy by replacing repeated values with shorter codes.
  - **Delta encoding:** Stores differences between sequential values, which is particularly effective for time series data.
  - **Run-length encoding:** Compresses sequences of repeated values, optimizing storage for datasets with many duplicates.
- **Schema-less support:** A standout feature of the archive file format is its schema-less design. Unlike rigid schema-driven systems or less-structured schema-on-read approaches, the schema-less implementation enables the efficient storage and querying of diverse data types without requiring a predefined structure.

  The schema-less design also allows for the rapid onboarding of new telemetry sources and the seamless handling of evolving data formats, both of which are common in dynamic enterprise environments. By minimizing the operational burden of schema management while maintaining high performance, this design allows businesses to respond faster to changing data requirements without sacrificing efficiency or scalability.
- **Efficient data retrieval:** The archive file format is designed to facilitate rapid data access. By leveraging the columnar storage approach, it organizes data intelligently to minimize the time required for queries and enhance overall performance. This targeted data retrieval strategy ensures that the format maintains the significant performance improvements discussed earlier, making it highly effective for analytical workloads.
- **Predicate pushdown:** Predicate pushdown is a crucial feature for optimizing query performance. It enables the query engine to filter out unnecessary data at the storage level. By pushing down filters and limits, only relevant data is read from storage, significantly reducing input/output (I/O) operations and speeding up query execution. While this approach is now common across many relational database management systems (RDBMS), its implementation in the archive file format was designed to meet the specific challenges of observability workloads, including handling high data ingestion rates and real-time analytical queries on massive datasets. This ensures efficiency and performance in scenarios that demand low-latency responses.
- **Metadata storage:** In addition to its robust features, the archive file format includes comprehensive metadata storage. This metadata provides essential information about each file's structure and characteristics, including:
  - **Column types:** Details on the data types of each column, enabling efficient processing and type validation.
  - **Compression settings:** Information about how each column is compressed, which aids in decompression during data retrieval. For instance, high-redundancy columns may use dictionary encoding, while time-series data might leverage delta encoding for maximum compression efficiency.
  - **Summary statistics:** Basic statistics for each column (such as min, max, average) that help the query engine quickly assess which files contain relevant data.

This rich metadata enables the query engine to optimize access by quickly identifying pertinent information within each file. As a result, query responses are faster and the analysis of dataset is more efficient.

Handling the live edge of the data

Observability use cases often bias quite heavily to query the most recent data, what we call as “live edge of the data.” This requires the data to be available for querying even as it’s being stored in a columnar format.

Insert workers read the data from partitioned Kafka streams and convert it into archive files in batches. Before an archive file is flushed to our cloud object store, the data is written in row format in a block store with much faster I/O.

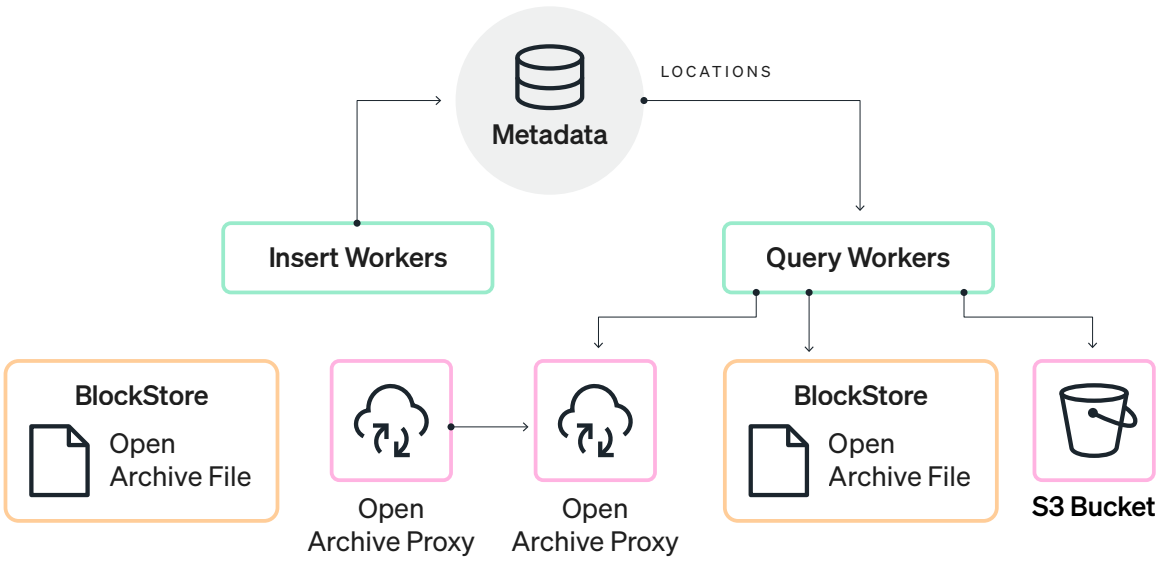
The challenge with live edge data lies in the fact that it’s still in the process of being persisted. Typically, the system waits to fill an archive file size of 26 MB before flushing it, as smaller files would lead to excessive fragmentation and degraded file read efficiency. During this time, the data resides with a single insert worker. However, the hardware of a single insert worker cannot scale to handle the disproportionate query load that live edge data often attracts.

This poses an interesting challenge in horizontally scaling the query load across multiple workers when the queries are accessing the live edge. To address this, an efficient data transport method is required. One that can transfer data from the insert worker disks to the network, using a memory-efficient technique like `sendfile`. Additionally, it should support HTTP range queries.

Query workers check the disk for the file before processing the query for that particular file. If the file isn’t found on disk, the query worker will download the finished archive file from S3 (or the configured backup store) and then proceed to process the query.

For open archive files, the file only exists on a single disk, attached to the insert worker. If the file isn’t present on disk, a full download request will be sent to the owning insert worker to retrieve the full bytes written for the archive file, similar to the behavior for closed files. If the archive file is already on disk, additional bytes might have been appended to it, so an HTTP range query can retrieve any remaining bytes, which are then appended to the file. After the bytes have been appended, the query can proceed.

This is an important trade-off, sacrificing query response times for horizontal scalability. This has increased the median latency from 50 ms to 60 ms, highlighting the importance of fast live edge queries in observability workloads and the need for horizontal scaling in that layer.

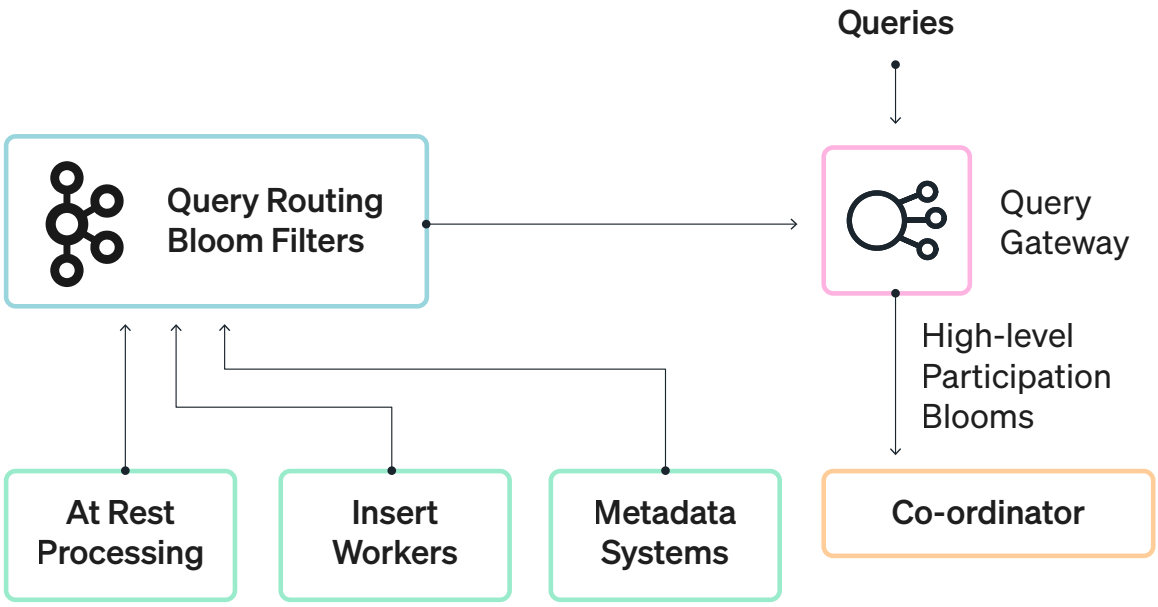


Query processing

The core principle of the query processing architecture for real-time analytics is to minimize the amount of data that needs to be examined to fulfill a query. Traditional relational database models often rely on index scans, which require searching through large portions of an index. In contrast, this architecture achieves efficiency through strategic query routing.

Query routing aggregates data from insert workers regarding the partitions and event types being ingested. It also gathers information about any data movement conducted by background processors or any metadata changes made for operational purposes. This collected information is then converted into queryable Bloom filters, which serve as an efficient means of representing the presence or absence of specific data elements.

These Bloom filters are instrumental for the query gateway, as they enable precise routing of queries to the appropriate partitions. This architectural choice enhances performance by reducing unnecessary data checks and optimizes resource utilization across the system.

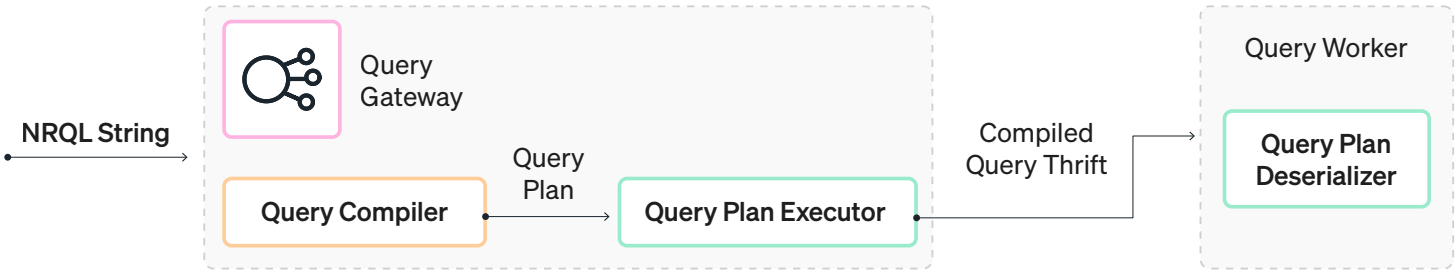


Query gateway

The process of executing a query begins with parsing and compiling a query execution plan. The language used is New Relic Query Language (NRQL), which is inspired by SQL but has been enhanced with additional function such as TIMESERIES and HISTOGRAM to provide more analytical capabilities.

The parsing and compilation of queries involve several critical steps. Initially, abstract syntax trees (ASTs) are constructed, followed by the application of various compile-time transformations and optimizations. During this phase, a few unresolved nodes may remain, which will be addressed at query execution time. Overall, this method produces a fine-grained logical execution plan characterized by minimal unresolved nodes, thereby ensuring efficient execution.

One of the key architectural advantages of this approach is the ability to generate a highly optimized execution plan that can adapt to different query requirements. Encoding the plan into Thrift facilitates seamless communication with query workers, enabling them to execute tasks efficiently across distributed systems.

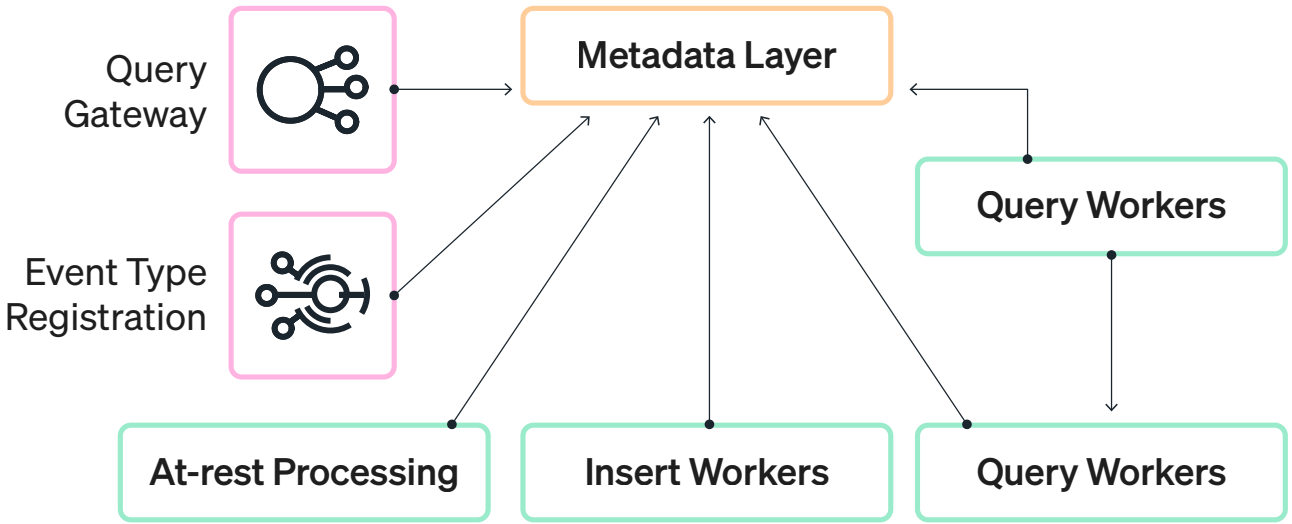


Metadata layer

The metadata layer is a critical component of modern data architectures. It serves as a unifying force, ensuring seamless communication and coordination among diverse data components. By providing a centralized repository of consistent and reliable information about the data's location, state, and characteristics, this layer fosters a cohesive data ecosystem.

Architecturally, the metadata layer plays several pivotal roles. It acts as a strategic layer of abstraction, decoupling the compute and storage layers and promoting flexibility and scalability. This abstraction enables data to be accessed and processed independently of its physical location, significantly enhancing data management and governance. Furthermore, the metadata layer offers invaluable insights into data assets. It provides precise location information for archive files, indexes, and essential statistics, empowering data analysts and engineers to efficiently locate and utilize data. For customers, it offers transparency into the data's provenance, quality, and schema, and ensures that data is easily accessible for analytics and decision-making.

The metadata layer also facilitates interoperability with external data sources and query engines. By establishing standardized data definitions and formats, it enables seamless integration and data exchange across diverse systems and platforms. This not only empowers organizations to fully leverage their data but also positions the system to support future agentic data integrations with third-party platforms. As this capability matures, customers will be able to connect their NRDB telemetry with external systems more seamlessly, unlocking additional value through expanded data ecosystems.



Cellular Architecture

As NRDB scaled from ingesting a few thousand events per minute to 10 billion events per minute and serving queries on top of a few petabytes of data to exabytes of data per day, it encountered new scaling limits and failure domains within its hosted open-source components like Kafka, ZooKeeper, Redis, and MySQL.

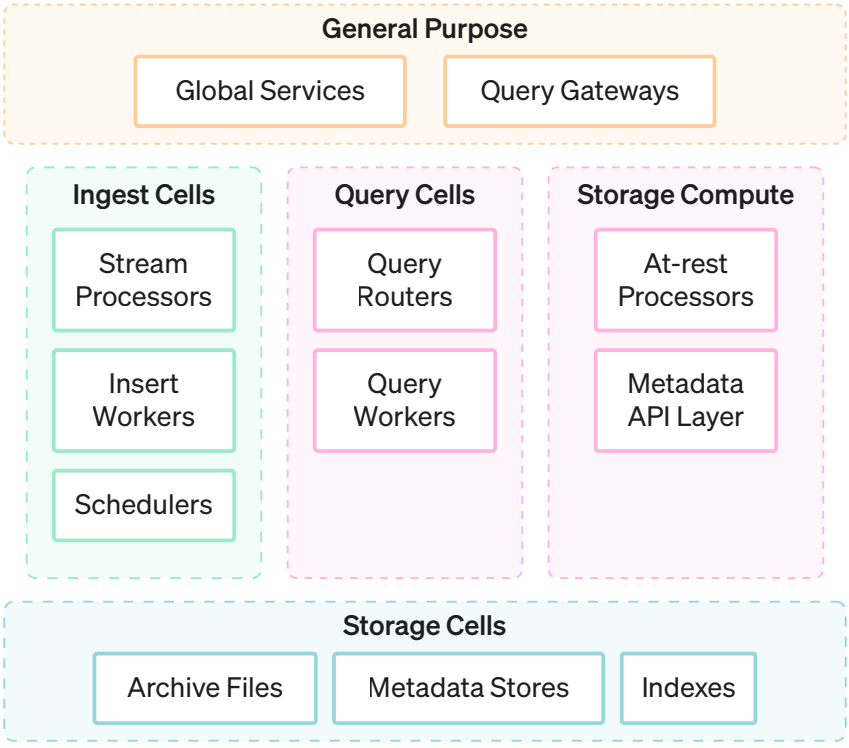
We were consistently operating these systems at the edge of their scalability. This led to operational issues that were not only difficult to debug and recover from, but also prevented us from elastically scaling to meet business needs.

We started the cellularization to:

→ **Improve resilience and fault isolation:** Enhance system resilience by creating isolated units (cells) that contain failures within specific boundaries. This means that if one cell experiences an issue—such as a deployment failure or data corruption—it doesn't affect the other cells. This isolation significantly reduces the "blast radius" of failures, allowing the overall system to maintain availability and performance even when individual components fail.

→ **Bounded load and predictable performance:** Cellular architecture allows for the division of workloads into discrete units (cells), each with defined resource limits. This design ensures that each cell can handle a specific load without affecting others, leading to predictable performance characteristics. As demand increases, new cells can be added to accommodate additional load, ensuring that the overall system remains performant and responsive under varying conditions. This bounded load approach helps maintain consistent performance metrics across the system, making it easier to forecast capacity needs and operational behavior as usage scales up.

With those goals, here's how the various components are organized into cells.



In a high-level systems diagram, you would see five such cell types to illustrate the concept. Each of these distinct cell types is designed to optimize specific functionalities:

1. Ingest cells:

→ **Real-time data ingestion:** These cells are responsible for rapidly ingesting data and processing it in real time. The goal is to maintain a "time to glass" of less than 1 second, ensuring that the time from data ingestion to visualization remains below this threshold.
→ **Stream processing:** They perform stream processing operations, such as synthesizing and updating entities and relationships from telemetry data.
→ **Shared state:** By sharing state, these cells can efficiently coordinate and optimize data processing.

2. Query cells:

→ **Low-latency query processing:** These cells are optimized for efficient query execution, aiming for low query latency.
→ **Query orchestration:** They orchestrate the query processing pipeline, ensuring optimal performance.

3. Storage cells:

→ **Durable data storage:** These cells provide reliable and durable storage for various data types, including raw data, metadata, and indexes.
→ **Performance optimization:** They offer different storage classes with specific performance characteristics (for example, I/O speed, replication) to meet diverse data storage needs.

4. Storage compute cells:

→ **Post-ingest processing:** These cells handle various post-ingestion tasks, such as data compaction, batch processing, retention enforcement, and index curation.
→ **Data transformation and enrichment:** These cells can also be used for data transformation and enrichment.

5. General purpose cells:

→ **Control plane:** These cells act as the control plane for the system, handling requests from customers for querying data and performing administrative tasks.
→ **Gateway services:** They provide gateway services to expose the system's capabilities to external users.

Data and state transfer

Data and state transfer between cells is orchestrated through enclaves. Enclaves are specialized components designed to manage communication and data exchange between different cell types. They implement messaging constructs, such as Kafka to facilitate asynchronous communication and data sharing mechanisms, like ACID-compliant data storage, to ensure data consistency and reliability. By leveraging enclaves, the system ensures reliable and efficient data transfer between cells, maintaining the integrity and performance of the overall architecture.

Limits and protection

In any database system, especially one designed for multi-tenancy like NRDB, implementing limits and protection mechanisms is crucial to ensure that no single tenant can monopolize resources or degrade performance for others.

- → **Resource quotas:** NRDB enforces quotas on resource usage per tenant to prevent overconsumption of CPU, memory, or I/O bandwidth. This ensures fair resource allocation across tenants.
- → **Rate limiting:** NRDB employs industry-standard rate limiting strategies to protect against sudden spikes in traffic that could overwhelm the system. These strategies control the number of requests a tenant can make within a specified timeframe. While these rate limiting measures align with industry standards, NRDB offers flexibility by allowing customers to tune these limits according to their specific needs. This customization capability ensures that customers can optimize performance and resource allocation based on their unique usage patterns.
- → **Data isolation:** Each tenant's data is logically separated to prevent unauthorized access or interference. This isolation is enforced at both the application and storage levels.

Auto scaling within and across cells

NRDB's architecture supports auto scaling capabilities both within individual cells (units of deployment) and across multiple cells in a distributed environment.

- → **Dynamic resource allocation:** The system continuously monitors resource usage and automatically allocates additional resources when demand increases. This ensures optimal performance during peak loads without manual intervention.
- → **Horizontal scaling:** As demand grows, NRDB can scale out by adding more cells to distribute the load evenly across the infrastructure. This horizontal scaling capability allows for seamless expansion as data volumes increase.
- → **Cell lifecycle management:** Each cell operates independently but can communicate with others for load balancing purposes. This design minimizes downtime during scaling operations while maintaining high availability.

Managing stateful data while keeping cell lifecycle lightweight

Dealing with stateful data in a lightweight manner is a significant challenge in distributed systems like NRDB. The architecture addresses this by employing several strategies:

- → **Stateless processing:** Where possible, NRDB uses stateless processing for queries to simplify scaling and reduce resource overhead. Stateless components do not retain information between requests, allowing them to be easily replicated across multiple nodes.
- → **Lightweight state management:** For operations that require stateful processing, such as session management or transaction tracking, NRDB employs lightweight state management techniques such as caching that minimize resource consumption while ensuring necessary state information is available when needed.
- → **Efficient data serialization:** Data transmitted between layers is serialized efficiently to minimize latency and bandwidth usage. This process converts complex data structures into compact, optimized formats, making them suitable for fast network transmission. By reducing the size of transmitted data and streamlining communication between system components, serialization ensures that even stateful interactions maintain low latency and high performance.

Integrating Data in Workloads

Querying mutable and external data alongside telemetry data presents unique challenges and opportunities in modern database systems. Telemetry data is predominantly immutable, which allows database systems to optimize for cache efficiency. However, contemporary telemetry applications necessitate the capability to query extensive volumes of telemetry data in conjunction with smaller sets of mutable data, such as personnel information, sales metrics, or other business intelligence data. To address this need, NRDB has evolved its query layer to facilitate such complex queries, enabling users to efficiently integrate and analyze both mutable and immutable datasets. This advancement enhances the ability to derive actionable insights from diverse sources of information, thereby supporting informed decision-making processes in various business contexts.

There are two broad areas that we dealt with:

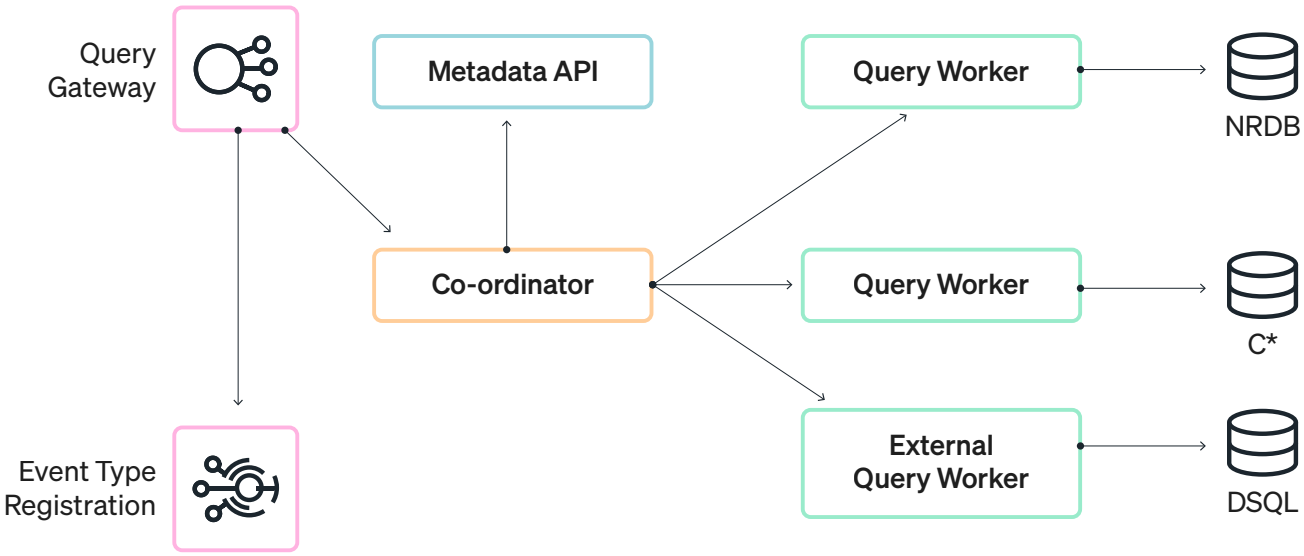
→ Querying mutable data, which involves invalidating the caches and incurring the cost of retrieving the freshest copy of the data.
→ Querying a separate database engine to get that data.

Both of these cases required the query planner to be aware of the type of data being queried.

Every piece of data in the platform is organized into a type of event, referred to as “eventTypes.” From a query planner perspective, there are three categories:

→ An eventType that is internal to NRDB and is immutable.
→ An eventType that is internal to NRDB and is mutable.
→ An eventType that is external to NRDB.

The high-level architecture is as follows:



This architecture is designed as a connector-based framework, where each connector operates a dedicated fleet of workers (compute resources) responsible for executing specific segments of queries with optimal efficiency. Each connector is equipped with the necessary logic to enhance performance, including implementing techniques like predicate pushdown. This ultimately leads to faster query execution times.

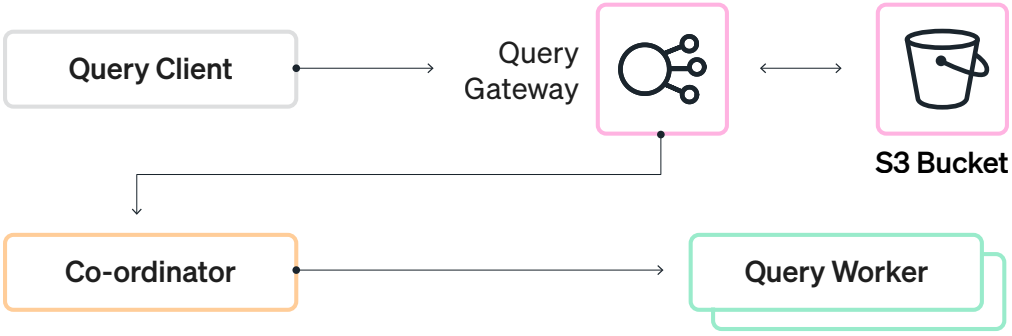
In addition to executing queries efficiently, each connector plays a crucial role in task orchestration tailored to its specific data source. This orchestration involves managing various tasks such as distributing workloads among the compute resources, handling data retrieval processes, and ensuring that queries are executed in a manner that aligns with the capabilities and constraints of the underlying data source.

Furthermore, the connectors interact seamlessly with both the coordinator and the metadata API. The coordinator oversees the overall query execution process, ensuring that all components work together harmoniously. Meanwhile, the metadata API facilitates access to essential information about data structures, schemas, and other relevant details necessary for executing queries accurately.

By supporting the querying of external data in the context of telemetry data, NRDB is leading a positive disruption in the observability space. This capability allows external metadata to provide vital context about the structure and organization of data, enabling connectors to optimize query execution without duplicating or reshaping the data. This approach aligns with industry priorities of interoperability and the ability to work with data in place, an area now actively being explored by the industry.

While this approach introduces variability in query performance, increases complexity, and does not ensure strong data consistency, it provides significant flexibility by allowing immediate querying of data. Previously, users might have spent days building entire pipelines to make such data queryable. By leveraging the connectors' ability to directly interact with external data sources, NRDB significantly reduces the time required to access critical information. Although this flexibility may result in slightly higher latencies, it represents a substantial improvement over traditional, time-consuming processes. This trade-off is especially valuable when rapid access to information is crucial such as during incident response.

The query platform also incorporates a feature for query progress checkpointing, where applicable, which significantly enhances the user experience and analytical capabilities. This functionality enables users to receive incremental results throughout the execution of their queries, rather than waiting for the entire dataset to be processed before obtaining any output. By facilitating incremental result delivery, users can begin analyzing data as it becomes available, allowing for more dynamic and responsive decision-making. This is particularly advantageous when dealing with extremely large volumes of data, as it mitigates the challenges associated with processing such datasets in their entirety. Moreover, this capability supports real-time analytical queries by allowing users to interact with data in a more fluid manner. As users receive partial results, they can adjust their queries or refine their analyses based on the insights gained from the initial outputs. This iterative approach not only improves efficiency but also enhances the overall effectiveness of data exploration and analysis in real-time contexts.



Unified Query Language

Through the platform, we use one SQL-like query language for users to interact with the data.

This spans to all data transformation activities. Users can create data (or entity) synthesis rules, data deletion (or drop rules) or obfuscation rules, all of which execute on the customer environment or the New Relic Intelligent Observability Platform.

After the data is ingested, users can configure alerts or define service levels, which the system evaluates at pre-defined cadences. Finally users can query the data directly using the query builder, through dashboards or via APIs. During all these activities, the same language is used to reduce the cognitive burden for users. While the amount of grammar supported might vary slightly, it provides an intuitive way to interact with the data.

Conclusion

The ability to process and analyze large-scale telemetry data in real time is critical for modern enterprises operating in cloud-native and distributed environments. However, building a database that balances scalability, fault tolerance, and cost efficiency is not without its challenges. Addressing issues such as efficient integration of diverse data types, predictable performance under heavy workloads, and resource optimization requires thoughtful architectural choices.

This white paper highlights how NRDB's architecture is purpose-built to tackle these challenges head-on. Its scalable and resilient design provides a robust foundation for meeting the demands of modern observability workloads, enabling organizations to achieve operational efficiency with real time analytics. NRDB's architecture is a testament to our commitment at New Relic to providing world-class solutions that help organizations achieve their business objectives with agility and confidence.

New Relic remains committed to further enhancing NRDB's capabilities through continuous innovation and by addressing evolving needs of its customers. As technology advances, NRDB will continue to deliver cutting-edge solutions, ensuring that customers can unlock the full potential of their data.

new relic.

© Copyright 2025, New Relic, Inc. All rights reserved. All trademarks, trade names, service marks and logos referenced herein belong to their respective companies. 02.2025