ROCKSET CONCEPTS, DESIGN & ARCHITECTURE

ABSTRACT

Rockset is a real-time indexing database for building data-intensive applications at scale. It enables application developers to create fast APIs using SQL search, aggregation and join queries directly on semi-structured data. This document describes the architecture of Rockset and the design choices and innovations which enable it to produce highly efficient execution of complex distributed queries across diverse data sets.

Purvi Desai
Kevin Leong

Updated August 2020

TABLE OF CONTENTS

INTRODUCTION

The world is rapidly moving towards real-time analytics in the form of applications that process different types of data from multiple sources and initiate specific actions automatically in real time. These data-driven applications are becoming the new normal across marketing, e-commerce, gaming, IoT, logistics and many other use cases. They typically combine real-time and historical data, using as much useful data as they can, to take actions within limited time windows.

Examples of data used by these applications include:

- Web-browsing history in the form of clickstream data
- Machine-generated event logs
- Sensor and GPS data from IoT endpoints
- Customer, purchase and inventory data
- Third-party data sets such as market insights from Nielsen

These data sets are often semi-structured in nature, with complicated, nested structures, and may originate from a variety of data sources, like transactional databases, streaming platforms and data lakes.

To make optimal decisions or take the best actions, data-driven applications need to run complex queries over large-scale data, but still require low latency to respond in real time. This is inherently challenging with fast-moving data without well-defined schema. Relational database management systems can provide a wide range of query functionality but would require extensive schema design and data modeling to be used for this class of application. NoSQL systems have flexible schemas but would require data denormalization to support many complex queries and are unable to support powerful joins.

A REAL-TIME INDEXING DATABASE

Rockset is a real-time indexing database designed to serve data-driven applications at scale. Examples of such applications include instant personalization, IoT automation, real-time customer 360s and many gaming apps. Rockset enables users to create fast APIs, using SQL, directly on semi-structured data and without the need for pre-defined schema.

Rockset is often used as a speed layer that indexes data from an OLTP database like MongoDB or DynamoDB, a streaming platform like Apache Kafka or Amazon Kinesis, or a data lake like Amazon S3. It acts as an external secondary index, accelerating analytic queries and enabling performance isolation for primary transactional systems. It is not a transactional database and is not designed to support OLTP workloads.

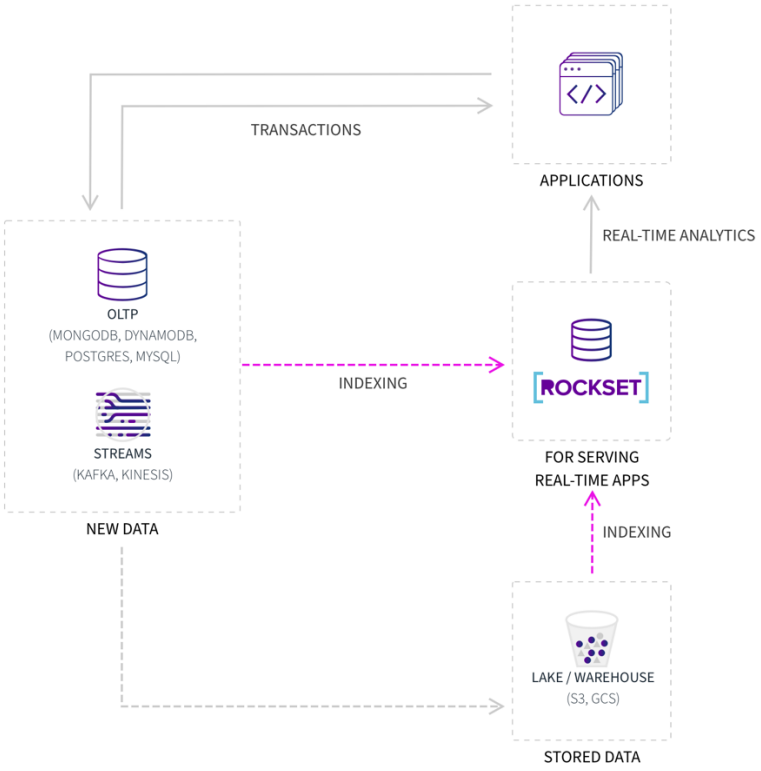

Figure 1: Rockset acts as an external secondary index to serve real-time applications

In this document, we describe the architecture of Rockset and the design choices and innovations which enable it to produce highly efficient execution of complex distributed queries across diverse data sets.

Data Model

Document Model with SQL Queries

Rockset stores data in a document data model. In traditional relational database systems, you must define a schema, which describes the tables in a database, the type of data that can be stored in each column and the relationships between tables of data. In contrast, documents describe the data in the document, as well as the actual data, allowing flexibility and variation in the structure of each document. Using a document model, Rockset can perform schemaless ingestion of data in JSON, XML, Avro, Parquet and other semi-structured formats.

Rockset differs from typical document-oriented databases in that it indexes and stores data in a way that can support relational queries using SQL. Our architecture is designed from the ground up so that schematization is not necessary at all, and the storage format is efficient even if different documents have data of different types in the same field. A schema is not inferred by sampling a subset of the data - rather, the entire data set is indexed so when new documents with new fields are added, they are immediately exposed in the form of a smart schema to users and made queryable using SQL in Rockset. This means new documents are never rejected if they unexpectedly show up with new fields or data formats.

Documents can be complex, with nested data to provide additional sub-categories of information about the object. Rockset supports arrays and objects, which can contain any value and hence be nested recursively. Rockset is designed to allow efficient ingestion, indexing and querying of deeply nested data.

In the case of nested JSON, the documents can be "flattened" in effect, so that nested arrays can be queried elegantly. Since traditional SQL specifications do not address nested documents, we have added certain custom extensions to our SQL interface to allow for easy querying of nested documents and arrays. In addition, the ability of Rockset to perform schemaless ingestion completely eliminates the need for upfront data modeling.

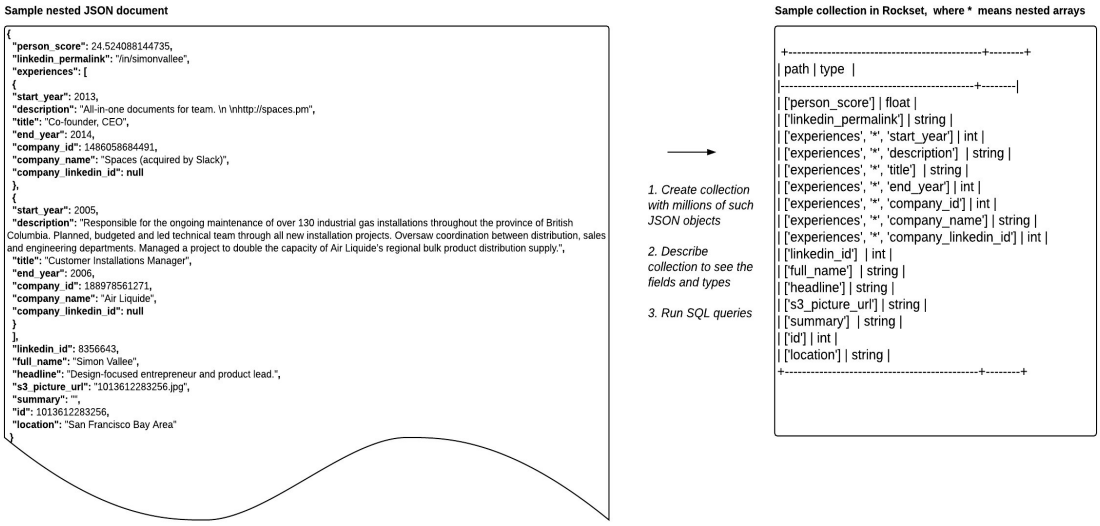

Figure 2: JSON documents in a Rockset collection

The basic data primitive in Rockset is a document. A document has a set of fields and is uniquely identified by a document id. Collections allow you to manage a set of documents and can be compared to tables in the relational world. A set of collections is a workspace. All documents within a collection and all fields within a document are mutable. Rockset provides atomic writes at the document level. You can update multiple fields within a single document atomically. Rockset does not support atomic updates to more than one document. All writes are asynchronous and the period from when data is written to the database to the time when it is visible in queries varies between 1 to 5 seconds.

Strong Dynamic Typing

Types can be static (known at compile time, or, in our case, when the query is parsed, before it starts executing) or dynamic (known only at runtime). A type system can be weak (the system tries to avoid type errors and coerces eagerly) or strong (type mismatches cause errors, coercion requires explicit cast operators). For example, MySQL has a weak, static type system. (`2 + '3foo'` is `5`; `2 + 'foo'` is `2`). PostgreSQL has a strong, static type system (it doesn't even convert between `integer` and `boolean` without a cast; `SELECT x FROM y WHERE 1` complains that `1` is not a boolean).

Rockset has *strong dynamic typing*, where the data type is associated with the value of the field in every column, rather than entire columns. This means you can execute strongly typed queries on dynamically typed data, making it easier to work with fluid datasets. This is useful not only in dynamic programming languages like Python & Ruby, but also in statically typed C++ & Java applications where any type mismatch would be a compile time error.

Cloud-Native Architecture

One of the core design principles behind Rockset is to exploit hardware elasticity in the cloud. Traditional databases built for data centers assume a fixed amount of hardware resources irrespective of the load, and then design to optimize the throughput and performance within that fixed cluster. However, with cloud economics, it costs the same to rent 100 machines for 1 hour as it does to rent 1 machine for 100 hours to do a certain amount of work. Rockset has been architected so as to optimize the price-performance ratio by aggressively exploiting the fundamentals of cloud elasticity.

Rockset's cloud-native architecture allows it to scale dynamically to make use of available cloud resources. A data request can be parallelized, and the hardware required to run it can be instantly acquired. Once the necessary tasks are scheduled and the results returned, the platform promptly sheds the hardware resources used for that request. The key components are containerized using Kubernetes for a cloud-agnostic approach.

Some of the key guiding principles behind Rockset's design include:

- **Use of shared storage rather than shared-nothing storage**
  Cloud services such as Amazon S3 provide shared storage that can be simultaneously accessed from multiple nodes using well-defined APIs. Shared storage enables Rockset to decouple compute from storage and scale each independently. This ability helps us build a cloud-native system that is orders of magnitude more efficient.

- **Disaggregated architecture**
  Rockset is designed to use only as much hardware as is truly needed for the workload it is serving. The cloud offers us the ability to utilize storage, compute and network independently of each other. Rockset's services are designed to be able to tune the consumption of each of these hardware resources independently. Additionally, a software service can be composed from a set of microservices, with each microservice limited by only one type of resource in keeping with the disaggregated architecture.

- **Resource scheduling to manage both supply and demand**
  A traditional task scheduler typically only manages demand by scheduling task requests among a fixed set of hardware resources available. Rockset's cloud-native resource scheduler, on the other hand, can manage demand along with supply. It can request new hardware resources to be provisioned to schedule new task requests based on the workload and configured policies. Also, once done serving the request, the resource scheduler sheds the newly provisioned hardware as soon as it can to optimize for price-performance.

- **Separation of durability and performance**
  Maintaining multiple replicas was the way to achieve durability in the pre-cloud systems. The downside here is, of course, the added cost for additional server capacity. However, with a cloud-native architecture, we can use the cloud object store to ensure durability without requiring additional replicas. Multiple replicas can aid query performance, but these can be brought online on demand only when there is an active query request. By using cheaper cloud object storage for durability and only spinning up compute and fast storage for replicas when needed for performance, Rockset can provide better price-performance.

- **Ability to leverage storage hierarchy**
  Rockset is designed to take advantage of the range of storage hierarchy available in the cloud. It uses hierarchical storage with the help of RocksDB-Cloud, which is a high-performance embedded storage engine optimized for SSDs. A RocksDB-Cloud instance automatically places hot data in SSDs and cold data in cloud storage. The entire database storage footprint need not be resident on costly SSD. The cloud storage contains the entire database and the local storage contains only the files that are in the working set.

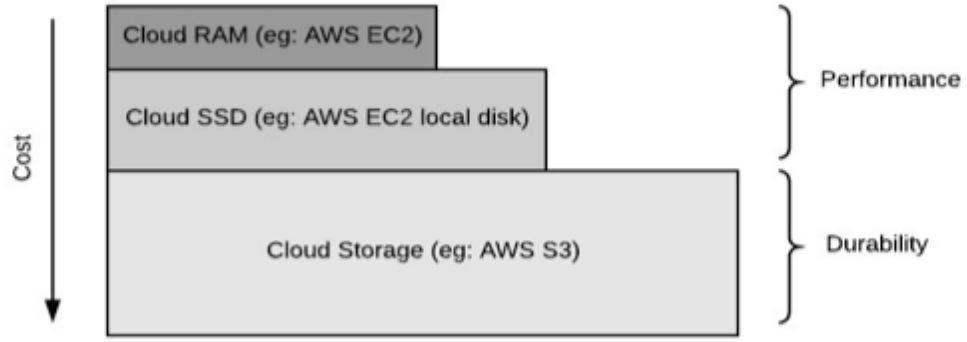

Figure 3: Hierarchical storage in the cloud

RocksDB-Cloud

RocksDB-Cloud is Rockset's embedded durable storage engine. It extends the core RocksDB engine, used in production at Facebook LinkedIn, Yahoo, Netflix, Airbnb, Pinterest and Uber, to make optimal use of hierarchical storage from SSDs to cloud storage.

RocksDB-Cloud is fully compatible with RocksDB, with the additional feature of continuous and automatic replication of database data and metadata to cloud storage (e.g. Amazon S3). In the event that the RocksDB-Cloud machine dies, another process on any other EC2 machine can reopen the same RocksDB-Cloud index.

A RocksDB-Cloud instance is cloneable. RocksDB-Cloud supports a primitive called zero-copy-clone that allows another instance of RocksDB-Cloud on another machine to clone an existing database. Both instances can run in parallel and they share some set of common database files.

Aggregator-Leaf-Tailer Architecture

Rockset employs an Aggregator-Leaf-Tailer (ALT) architecture, favored by web-scale companies, like Facebook, LinkedIn, and Google, for its efficiency and scalability. The ALT architecture has a high-performance serving layer that can serve complex queries, eliminating the need for complex data pipelines.

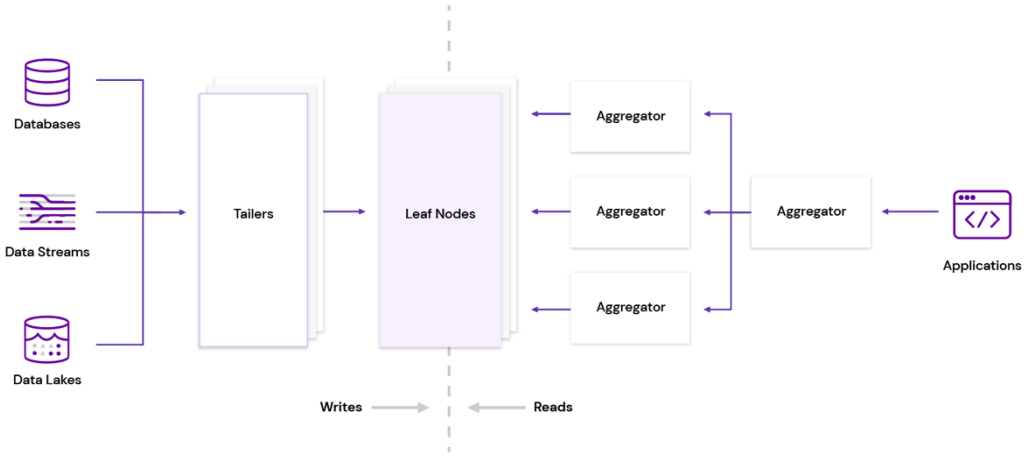

Figure 4: The Aggregator-Leaf-Tailer architecture used by Rockset

The ALT architecture comprises:

1. **Tailer**: Its job is to fetch new incoming data from a variety of data sources, be it a transactional database, like MongoDB or Amazon DynamoDB, a streaming data source, like Apache Kafka or Amazon Kinesis, or a data lake, like Amazon S3.
2. **Leaf**: It comprises a powerful indexing engine. It indexes all data that arrives from the Tailer. The indexing component builds multiple types of indexes—inverted, column-based, row-based—on the fields of a data set. The goal of indexing is to make any query on any data field fast.
3. **Aggregator**: This tier is designed to deliver low-latency aggregations of data coming from the leaves, be it columnar aggregations, joins, relevance sorting, or grouping.

The Tailer, Leaf, and Aggregator run as discrete microservices in disaggregated fashion, each of which can be independently scaled up and down as needed. The system scales Tailers when there is more data to ingest, scales Leaves when data size grows, and scales Aggregators when the number or complexity of queries increases. This independent scalability allows the system to execute queries in an efficient and cost-effective manner.

The ALT architecture gives application developers the ability to run low-latency queries on semi-structured data sets without any prior transformation. Due to the scalability of the aggregator tier, accelerated by the multiple indexes on the data, the data transformation process can occur as part of the query itself.

Separation of Storage and Compute

Traditionally, systems have been designed with tightly coupled storage and compute with the aim of keeping storage as close to compute for better performance. With an improvement in hardware over time, along with the onset of cloud-based architectures, this is no longer a binding constraint. Tight coupling of storage and compute makes scaling these resources independently a challenge, in turn leading to overprovisioning of resources. Separating compute and storage, on the other hand, provides benefits such as improved scalability, availability, and better price-performance ratios. This allows you to scale out your cluster to adapt to variable workloads in a shorter span of time.

While storage-compute separation has been embraced by recent cloud data warehouse offerings, it is a novel concept for application backends. Rockset is unique in enabling users to benefit from storage-compute separation when serving data-driven applications.

Decoupling of storage and compute requires the ability to store persistent data on remote storage. RocksDB-Cloud allows us to provide this separation between compute and storage. All of RocksDB's persistent data is stored in a collection of SST (sorted string table) files in cloud storage. A compaction process is performed to combine this set of SST files to generate new ones with overwritten or deleted keys purged. Compaction is a compute-intensive process. Traditionally, you would have compaction happen on the CPUs that are local to the server that hosts the storage as well. However, since the SST files are not modified once created, it allows us to separate the compaction compute from storage. This is achieved by using remote compaction, wherein the task of compaction is offloaded to a stateless RocksDB-Cloud server once a compaction request is received. These remote compaction servers can be auto-scaled based on the load on the system.

Independent Scaling of Ingest and Query Compute

A further characteristic of the Rockset architecture, enabled by its decoupled Aggregator and Tailer tiers, is the ability to scale ingest and query compute independently of each other. This permits Rockset to handle spikes in data ingestion or query volume flexibly without overprovisioning compute resources.

In practice, a bulk load would result in more ingest compute being dynamically spun up to ensure ingest latency is minimized, while a surge in queries due to a product launch, for instance, can be handled by allocating more query compute to ensure low query latency.

Sharding and Replication

The data in Rockset is horizontally partitioned. Each document in a Rockset collection is mapped to an abstract entity called a microshard. This mapping is performed using a microshard mapping function, which is a function of the document ID. A group of microshards comes together to form a Rockset shard. Document-based sharding makes it easier for the system to scale horizontally. Each shard maps one-to-one to a RocksDB index instance, which holds the indexed data in the form of lexicographically sorted key-value pairs.

Splitting data in large collections into multiple shards allows you to leverage shard-level parallelism, where each shard can be scanned in parallel, helping serve queries on such

collections faster without scans being a bottleneck. Rockset allows you to have multiple replicas of the same shard for availability.

Mutability of the Index

Rockset is a mutable database. It allows you to update any existing record in your database, including individual fields of an existing deeply nested document.

Several traditional columnar databases also support updates (called trickle-loading). But since millions of values from the same column are columnar-compressed and stored in a single compact object, if you want to update one record that lies in the middle of such a compact object, you would have to do a copy-on-write for that object. Objects are typically hundreds of megabytes in size, and if updating a 100-byte record in the middle of a compact object triggers a copy-on-write cycle of hundreds of megabytes, such a system cannot sustain more than a trickle of occasional updates. Most users will prefer to add a new partition to an existing table to record the updates and then at the time of query, merge these partitions together. When performing updates in Rockset, users avoid this additional level of complexity.

Rockset's underlying storage engine, RocksDB-Cloud, is a key-value store which is designed to natively support any field to be update-able.

Schemaless Ingestion

Overview of Data Ingestion Flow

Data can come in either via the write API or through the various data sources that Rockset supports.

When data comes in via the write API, it is routed to an API server. The API server is an HTTP server that processes REST operations and acts as the frontend to the Rockset cluster. It takes the data requests, performs permission checks, and writes this data sequentially to a distributed log store. This data is then tailed and indexed by the leaf nodes. Leaves are EC2 machines with local SSDs that hold the indexed data that are later used to serve queries.

Heavy writes can impact reads, and since we want to serve operational analytics queries well, reads need to be fast. The distributed log store acts as an intermediate staging area for the data before it is picked up by the leaves and indexed. It also provides data durability until the data is indexed by the leaves and persisted to S3.

Ingesters read the data from data sources and may optionally perform user-requested transformations on this data before writing it to the log store. The transformer service runs as part of the ingester. It lets you drop or map fields.

In the case of bulk ingest when a significantly large size of data is being ingested, the data is written to S3 instead and just the headers are written to the log store, so as not to make the log store a memory bottleneck.

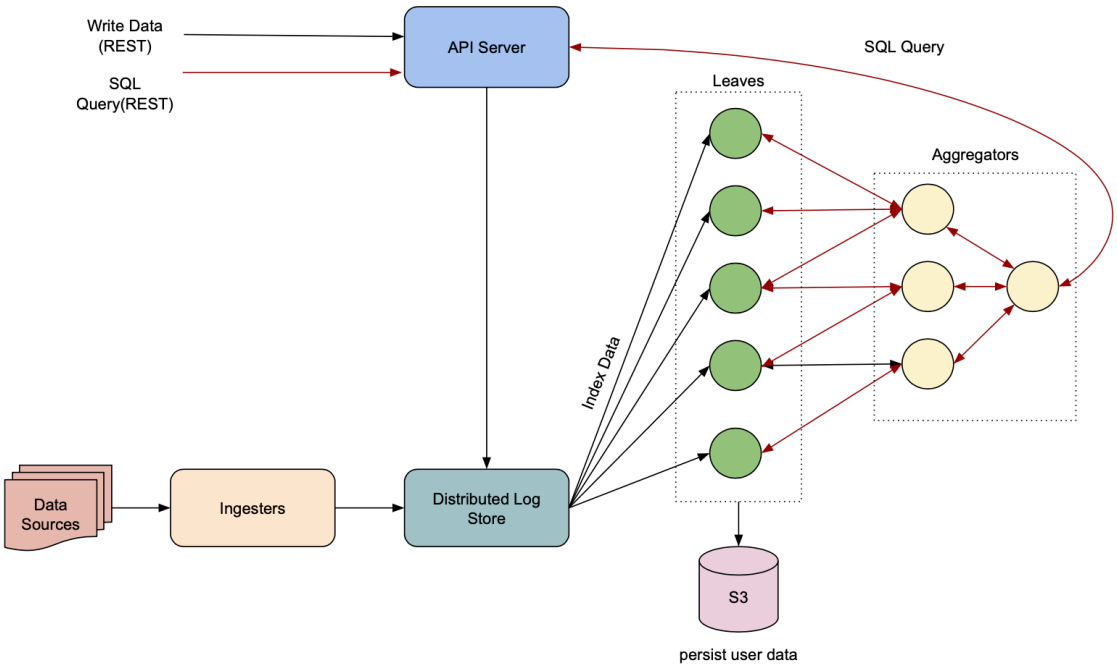

Figure 5: Data ingestion flow in Rockset

Field Mappings

Rockset supports data transformations that apply to all documents whenever they are ingested into a collection. These data transformation functions are called field mappings.

Such transformations can be useful for the following use cases:

- Drop fields
  - Do not persist certain fields, either to save space (or cost), or for privacy reasons.
- Field masking, hashing
  - Do not persist certain fields in their original form, likely for privacy reasons.
- Type coercion
  - Convert incoming string/text into types such as timestamp or geographical at the time of ingestion.
- Search tokenization
  - Analyze/tokenize incoming text fields to enable text-search queries on them.

For every ingested document, we pass it through a series of transformation steps. Each transformation takes a set of fields from the incoming document, evaluates some SQL expressions with the values of these fields as named parameters, and stores the results of these expressions as other fields in the document. These field mappings are applied to documents on all the write paths.

Built-In Connectors

Integrations allow you to securely connect external data sources such as DynamoDB, Kinesis, S3, Google Cloud Storage and more to your Rockset account. Rockset will then automatically sync your collections to remain up-to-date with respective data sources, usually within a matter of seconds.

Note that even if your desired data source is not currently supported, you can always use the Rockset API to create and update collections. However, you would have to manage your data syncing manually.

Write API

The Write API is a REST API that allows you to add documents to a collection in Rockset by sending an HTTPs POST request to Rockset's write endpoint.

You can use the Write API with Python, Javascript, or simply curl.

MongoDB

The MongoDB connector allows you to stream MongoDB's change streams to Rockset. You need to create a MongoDB integration to securely connect collections in your MongoDB account with Rockset and create a collection which continuously syncs your data from a MongoDB collection into Rockset in real-time.

Amazon DynamoDB

The Amazon DynamoDB connector allows you to use an Amazon DynamoDB table as a data source in Rockset. You first create an Amazon DynamoDB integration to securely connect tables in your AWS account with Rockset followed by creating a collection which syncs your data from an Amazon DynamoDB table into Rockset in real-time.

Apache Kafka

The Kafka connector helps you load data from Kafka into Rockset. Once users create an empty collection and configure the Rockset Kafka Connect plugin to point to their Kafka topic and Rockset collection, Kafka documents will start being written to Rockset. Only valid JSON and Avro documents can be read from Kafka and written to Rockset collections using this connector.

Amazon Kinesis

The Amazon Kinesis connector allows you to load data into Rockset using an Amazon Kinesis stream as a data source.

All you need to do is create an Amazon Kinesis integration to securely connect Kinesis streams in your AWS account with Rockset and then create a collection which syncs your data from an Amazon Kinesis stream into Rockset.

Amazon S3

The Amazon S3 connector lets you load data from an Amazon S3 bucket into a Rockset collection.

In order to use an Amazon S3 bucket as a data source in Rockset, all you need to do is create an Amazon S3 integration to securely connect buckets in your AWS account with Rockset and then create a collection which syncs your data from an Amazon S3 bucket into Rockset.

Google Cloud Storage

This connector lets you use a Google Cloud Storage bucket as a data source in Rockset. To use Google Cloud Storage as a data source, first create a Google Cloud Storage integration to securely connect buckets in your GCP account with Rockset followed by a collection which syncs your data from a Google Cloud Storage bucket into Rockset.

Bulk Load

Ingesting bulk-sourced data requires Rockset to keep up with the rate at which data floods in. In order to allow a smooth and efficient ingest experience while ingesting bulk-sourced data, Rockset provides a separate bulk load mode.

In the normal course of operations, an ingester worker serializes a source object and writes it straight to the log store. However, if the input dataset is tens of GB large, this can stress out the log store. In bulk load mode, Rockset uses S3 as a bulk-data queue instead of the log store. Only the metadata is written to log store instead. This mode also leverages larger leaf pods to be able to configure RocksDB with larger memtables. For indexing the data, the leaves tail the messages from the log store, examine at the message header, locate the Rockset document either from the message itself or from the corresponding S3 object, and index it into RocksDB-Cloud. The bulk leaves also fully compact the data and then start moving this compacted data over to regular leaves once the collection is ready to serve queries.

Converged Indexing

Rockset is built using Converged Indexing™, which is a patent-pending combination of:

- Inverted index
- Column-based index
- Row-based index

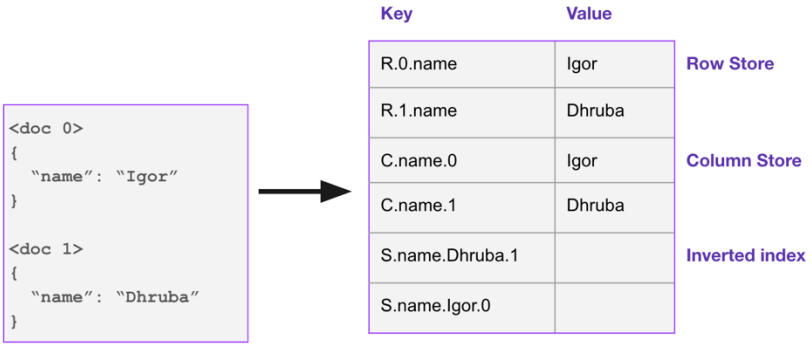

Figure 6: How a document is represented using Rockset's Converged Indexing

As a result, it is optimized for multiple access patterns, including key-value, time-series, document, search and aggregation queries. The goal of the Converged Indexing is to optimize query performance, *without* knowing in advance what the shape of the data is or what type of queries are expected. This means that point lookups and aggregate queries both need to be extremely fast. Our P99 latency for filter queries on terabytes of data is in the low milliseconds.

The following examples demonstrate how different indexes can be used to accelerate different types of queries.

Query 1

```
SELECT keyword, count(*) c
FROM search_logs
GROUP BY keyword
ORDER BY c DESC
```

There are no filters in this query; the optimizer will choose to use the column store. Because the column store keeps columns separate, this query only needs to scan values for column `keyword`, yielding a much faster performance than a traditional row store.

Query 2

```
SELECT *
FROM search_logs
WHERE keyword = 'rockset'
AND locale = 'en'
```

The optimizer will use the database statistics to determine this query needs to fetch a tiny fraction of the database. It will decide to answer the query with the search index.

Indexing Data in Real Time

The converged index is a live, real-time index that stays in sync with multiple data sources and reflects new data in less than a second. It is a covering index, meaning it does not federate back to the data source for serving any queries. This is essential for predictably delivering high performance.

Traditionally, maintaining live indexes for databases has been an expensive operation but Rockset uses a modern cloud-native approach, with hierarchical storage and a disaggregated system design to make this efficient at scale. Conceptually, our logical inverted index is separated from its physical representation as a key-value store, which is very different from other search indexing mechanisms. We also make the converged index highly space-efficient by using delta encoding between keys, and zSTD compression with dictionary encoding per file. In addition, we use Bloom filters to quickly find the keys - we use a 10-bit Bloom which gives us a 99% reduction in I/O.

Our indexes are fully mutable because each key refers to a document fragment - this means the user can update a single field in the document without triggering a reindex of the entire document. Traditional search indexes tend to suffer from reindexing storms because even if a 100-byte field in a 20K document is updated, they are forced to reindex the entire document.

TIME-SERIES DATA OPTIMIZATIONS

For time-series data, Rockset's *rolling window compaction* strategy allows you to set policies to only keep data from last "x" hours, days or weeks actively indexed and available for querying. Our time-to-live (TTL) implementation is built-in using compaction filters to automatically drop older data, which means we do not need separate I/O and additional resources to potentially run a loop that scans older data and deletes it. Traditional databases have been known to blow out the database cache and use lot of additional resources in order to support this behavior. We support sortkey behavior for event-series data based on time stamp specified or document creation time. To increase efficiency, our Converged Indexing engine stores the sortkey as part of the key-value store itself - so we do not need to retrieve *all* relevant documents as part of a query and then sort them in memory.

- At collection creation time the user can optionally map a field as event-time.
- When event-time is not explicitly specified, the document creation time will be used as event-time to determine system behavior.
- If users want to specify retention for a collection (eg: automatically purge all records older than 90 days) then we will determine how old a record is based on the event-time field.
- All queries will, by default, be sorted by event-time descending.
- All queries that have range clauses on event-time are significantly faster than similar queries on regular fields.
- All queries that include an "order by" event-time descending are significantly faster than similar queries requiring a sort on regular fields.

QUERY PROCESSING

Rockset provides a full SQL interface to query the data - including filters, aggregations and joins. When a query comes in it hits the Rockset API server and gets routed to an aggregator.

Broadly speaking, a SQL query goes through 3 main stages in Rockset:

1. Planning
2. Optimization
3. Execution

The aggregator plans the query and routes various fragments of the plan to appropriate leaves that hold the data to serve this query. The results are routed back to this aggregator which then sends the results back to the API server. We introduce additional levels of aggregators to distribute the processing across multiple aggregators for queries that may make a single aggregator a memory/computation bottleneck.

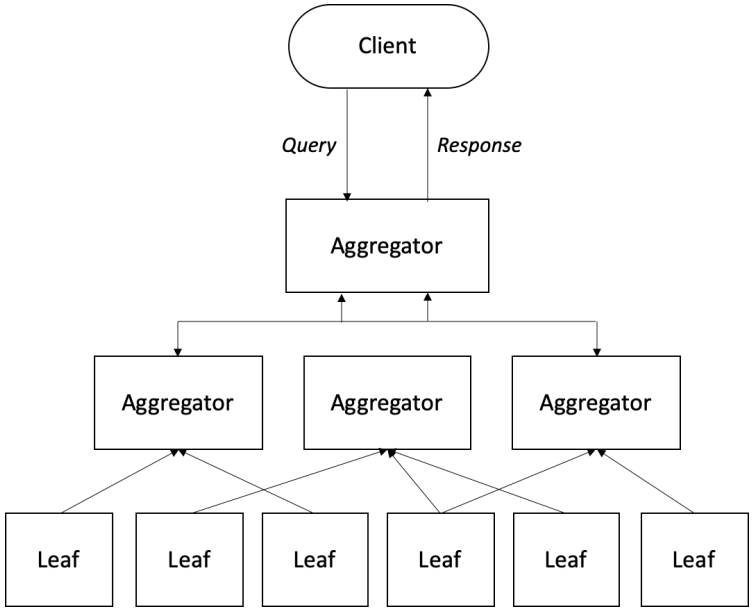

Figure 7: Query Processing

The following subsections go over the design of the query processing architecture given our unique challenges owing to working with dynamically typed data and the emphasis on performance requirements to support millisecond-latency analytics queries.

Query Planning

The first step before the planning phase is query parsing. The parser checks the SQL query string for syntactic correctness and then converts it to an abstract syntax tree (AST). This AST is the input to the query planner.

In the planning stage, a set of steps that need to be executed to complete the query is produced. This set of steps is called a **query plan**. The final query plan selected for execution is called the **execution plan**.

Query Optimization

The job of the query optimizer is to pick an execution plan with the optimal execution runtime. Rockset uses a Cost Based Optimizer (CBO) to pick an optimal execution query plan. It starts with all possible query plans in its search space and evaluates each of them by assigning a “cost” to every plan. This “cost” is mainly a function of the time required to execute that plan. The final cost of the plan is computed by breaking the query plan into simpler sub-plans and costing each of them in turn. The cost model uses information about the underlying data, such as total document count, selectivity, and distribution of values to guide the estimation of the cost of a plan.

A recursive in-memory data structure called Memo is used to efficiently store the forest of query plan alternatives generated during query planning. Plan alternatives are generated by applying a set of normalization and exploration rules to the plan nodes.

Normalization is used mainly to simplify expressions, transform equivalent expressions to a canonical form, and apply optimizations that are believed to always be beneficial in order to save

the CBO some work. We have implemented our own rule specification language (RSL) to express these normalization rules. We convert these RSL rules to C++ code snippets using our own RSL compiler.

Exploration happens as part of the query optimization stage. During this phase, the various plan alternatives are costed by costing dependent memo groups recursively, starting at a Memo's root group. It is during this phase that the most efficient join strategy, join ordering or access path would be picked.

Distributed Query Execution

The execution plan obtained as a result of exploration is simply a DAG of execution operators. For instance, an access-path operator is responsible for fetching data from the index, e.g. ColumnScan operator which fetches data by scanning the columnar index, while an aggregation operator performs aggregation computations on the data fed to it. The execution plan needs to be further prepared for distributed execution in this phase. The execution plan is first divided into fragments. Each fragment comprises a chain of execution operators.

There are primarily 2 classes of fragments:

- Leaf fragments: These are fragments of the plan that would typically be associated with retrieving data from the underlying collection. Leaf fragments are dispatched to and executed on leaf workers where shards of the collection reside.
- Aggregator fragments: These are fragments of the plan that would perform operations such as aggregations and joins on the data flowing from leaf fragments, and relay the final results to the API server. They are dispatched to and executed on aggregator workers.

The leaf fragment needs to be executed on a minimal covering of the shards that comprise the collection. Each of these leaf fragments can be executed in parallel offering shard-level parallelism.

The workers to assign these fragments are picked with the goal of keeping the load distributed among all the workers across queries. If a certain worker is unavailable to process a request, the execution engine retries scheduling the fragment on a different worker, thus ensuring that the query does not fail.

Once the fragments are scheduled to the respective workers, the execution proceeds bottom up in a non-blocking manner. No operator blocks on its input. The operator receiving data is also not forced to buffer an arbitrary amount of data from the operators feeding into it. Every operator can request a variable number of data chunks/pages from its predecessor, thus providing a way to implement non-blocking back pressure. The data exchange between operators happens in the form of data chunks, which organize data in a columnar format. This also makes it feasible for the operators to execute in a vectorized manner, where operations can be performed on a set of values instead of one value at a time, for better performance wins.

Application Development

Rockset provides client SDKs that you can use to create a collection, create integrations and load documents into it, and query collections. The following client SDKs are supported:

- Node.js
- Python
- Java
- Golang

REST API

Rockset provides a REST API that allows you to create and manage all the resources in Rockset. The endpoints are only accessible via https. The base URL of the Rockset API server is:

https://api.rs2.usw2.rockset.com

All requests must be authorized with a Rockset API key, which can be created in the Rockset console.

Some of the operations supported via the REST API are:

1. Create Collection: Create new collection in a workspace.
2. List Collection: Retrieve all collections in an org/workspace.
3. Delete Collection: Delete a collection and all its documents from Rockset.
4. Create Workspace: Create a new workspace in your org.
5. List Workspaces: List all workspaces.
6. Delete Workspace: Remove a workspace.
7. Add Documents: Add documents to a collection in Rockset
8. Query: Make a SQL query to Rockset
9. Delete Documents: Delete documents from a collection in Rockset.
10. Patch Documents: Patch documents in a collection
11. Create API Key: Create a new API key for the authenticated user.
12. List API Keys: List all API keys for the authenticated user.
13. Delete API Key: Delete an API key for the authenticated user.
14. Create Integration: Create a new integration with Rockset.
15. List Integrations: List all integrations for organization.
16. Delete Integration: Remove an integration.

Query Lambdas

Query Lambdas are named, parameterized SQL queries stored in Rockset that can be executed from a dedicated REST endpoint. Using Query Lambdas is recommended over querying directly from application code using raw SQL to build applications backed by Rockset. The advantages of doing so are described below.

Here are some of its core features:

- **Store SQL text and parameter information**
  Query Lambdas can be created and updated using the Rockset Console or by using the REST API directly. Each Query Lambda is tied to a specific query text and parameter

set. You can either set default values for query parameters (making them optional during executions of your Query Lambda), or you can make them mandatory for each execution.

- **Expose a REST endpoint to execute underlying query**
  Each Query Lambda is exposed as its own endpoint. For a Query Lambda such as:

  ```
  SELECT
    :echo as echo
  ```

  you would specify parameter echo and optionally provide a default value during the creation step. You can save this SQL-parameter pair as a Query Lambda, name it, and then execute it through a REST endpoint from anywhere given an API key.

- **Enforce version control**
  Each Query Lambda maintains a version history. Executions of a Query Lambda must also provide the version. Any update to a Query Lambda automatically creates a new version. Versioning allows you to build and test changes without affecting production queries. Each Query Lambda-version pair is immutable.

- **Organize by workspace**
  Just like Collections, each Query Lambda is associated with a particular workspace. You can set this workspace at the time of creation. Query Lambdas in one workspace are not limited to querying collections in the same workspace.

- **Organize by tag**
  If you have different sets of Query Lambda versions used to serve particular use cases, you can label each set with a corresponding tag. You can create, update, delete, and retrieve tags through the Rockset Console. Additionally, you may execute Query Lambda versions by tag, and achieve all of the aforementioned functionality through our REST API.

- **Expose usage metrics and statistics**
  Each version of Query Lambdas tracks and exposes a number of metrics. Currently supported metrics include:

  Last Executed: Time of last execution
  Last Executed By: User ID associated with last execution
  Last Execution Error: Time of last execution error
  Last Execution Error Message: Error message associated with last execution error

SECURITY

USE OF CLOUD INFRASTRUCTURE

Rockset is a data processor. All data loaded into Rockset is owned by the customer. Rockset indexes and stores all the data to allow SQL queries, but the data is always owned by the customer.

Rockset uses cloud-native best practices and exploits the underlying security policies of the public cloud it is hosted on. Architecturally, Rockset is designed to be cloud agnostic and may be run on AWS, Google Cloud, Azure or other public clouds in the future. However, currently, all of Rockset's services are run and hosted in Amazon Web Services (AWS), hence our security policies follow AWS best practices at this time. Anytime this document mentions Rockset servers, it means EC2 instances in AWS. Rockset does not operate any physical hosting facilities or physical computer hardware of its own. You can view the AWS security processes document here: https://d1.awsstatic.com/whitepapers/Security/AWS_Security_Whitepaper.pdf

DATA MASKING

For sensitive data like PII, Rockset supports data masking at the time of ingest utilizing field mappings. A field mapping allows you to specify transformations to be applied on all documents inserted into a collection. This can be used for type coercion, anonymization, tokenization, etc. When a particular field is masked using a hashing function like SHA256, only the hashed information is stored in Rockset. You can see how to do field mappings in Rockset here: https://docs.rockset.com/field-mappings/

ROLE-BASED ACCESS CONTROL

Role-based access control (RBAC) helps you manage user privileges in your account. Specify roles and privileges to ensure that you limit access to your data to the individuals that need it.

ADVANCED ENCRYPTION WITH USER-CONTROLLED KEYS

Rockset uses AWS Key Management Service to make it easy for you to create and manage keys and control the use of encryption. With advanced encryption, you can bring your own key which allows you to control the encryption and delete the key if needed.

DATA IN FLIGHT

Data in flight from customers to Rockset and from Rockset back to customers is encrypted via SSL/TLS certificates, which are created and managed by AWS Certificate Manager. An AWS application load balancer terminates SSL connections to our API endpoint. We currently use the ELBSecurityPolicy-TLS-1-1-2017-01 security policy for our HTTPS load balancers: https://docs.aws.amazon.com/elasticloadbalancing/latest/application/create-https-listener.html#describe-ssl-policies

AWS CloudFront distributions terminate SSL connections to our website and Console.

Within Rockset's Virtual Private Cloud (VPC), data is transmitted unencrypted between Rockset's internal services. Unencrypted data will never be sent outside of Rockset's VPC.

The only ways to generate traffic inside the VPC are through the Rockset API and secure VPN. Access to these is described in the following sections.

Data At Rest

Data is persisted in three places within Rockset:

1. In a log buffer service on encrypted Amazon EBS volumes. Rockset uses this log buffer as transient storage to independently scale data indexing (writes) and data serving (reads).
2. On our servers, which have local solid-state drives which are encrypted via dm-crypt.
3. In Amazon S3, where all stored objects are encrypted

In all cases, the encryption keys are managed by AWS Key Management Service (KMS). The master keys are never exposed to anyone (not even Rockset), as they never leave the KMS hardware. For evaluation accounts, the master keys used are created in Rockset's AWS account. For commercial installations, Rockset will provide a way for customers to provide their own KMS master key. All customer API keys and integration credentials are stored in a secure admin database that is encrypted at rest.

Access Controls

The only ways to access any stored data, servers, or services running inside Rockset's VPC are through the VPN server and Rockset API.

- Access to the VPN server requires a TLS auth key, a CA certificate, and the user's individual password. Access to the VPN server also requires two-factor authentication and per user certificates. There are no shared passwords for accessing the VPN server. Only employees that require development and administrative access will receive credentials to access the VPN server. Access to resources in the VPN is controlled on a per user basis. For example only required individuals have the ability to send SSH traffic. VPN access for each individual employee can be quickly revoked if necessary.
- Access to Rockset's Console is based on having a valid username/password to an Auth0 user that has been granted access to the Console. Rockset relies on Auth0 for user authentication instead of implementing our own for now: https://auth0.com/docs/overview
- Access to Rockset's API is based on API keys, which can only be created in the Console. Each API key can only access the resources granted to the user that created the key. All customer API keys and integration credentials are stored in a secure admin database that is encrypted at rest.

CONCLUSION

Rockset takes a new approach to online data, by bringing together:

- **Cloud-native architecture**
  Rockset was built to enable users to take maximum advantage of cloud efficiencies through a fully managed database service and independent scaling of individual compute and storage components.

- **Schemaless ingestion**
  Ingest semi-structured, nested data from databases, data streams and data lakes without the need for pre-defined schema. Ingest data continuously using fully managed connectors to common data sources or Rockset's Write API.

- **Converged indexing**
  Rockset automatically indexes all ingested data in multiple ways—inverted, column-based and row-based—to accelerate queries without requiring advance knowledge of query patterns.

- **Full-featured SQL**
  Compose queries and create APIs using declarative SQL, without the need to transform your data. Use the full functionality of SQL, including joins, optimized by Rockset's query engine.

Enabling developers to build real-time applications on data quickly and simply is what we, at Rockset, work towards every day. We believe you should be bottlenecked only by your creativity and not what your data infrastructure can do. Get started building with Rockset.

Version 3.0