ROCKSET CONCEPTS, DESIGN & ARCHITECTURE

ABSTRACT

Rockset is a serverless search and analytics engine that provides fast SQL on raw 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.

Shruti Bhat

October 2018
shruti@rockset.com

TABLE OF CONTENTS

Introduction

The world is rapidly moving towards data-powered automation in the form of applications that process different types of data from multiple sources and initiate specific actions automatically in real-time. These "data-powered applications" are becoming the new normal across marketing, finance, supply chain, human resources and IT departments. They typically combine streaming data sets, which are often semi-structured in nature, with regular structured data such as ERP or CRM databases.

Streaming data is typically *event data* that comes in semi-structured formats such as JSON, Parquet, XML, CSV, TSV. Example sources of such data streams include:

- Web browsing history in the form of clickstream data
- Machine generated event logs
- Sensor and GPS data from IoT end points
- Third party data sets such as market insights from Nielsen

Combining such loosely structured raw data with other structured databases is challenging because conventional database management systems would require extensive schema design and data modeling, while NoSQL systems have flexible schemas but require data denormalization and are unable to support powerful joins.

Rockset is a serverless search and analytics engine that provides fast SQL on diverse data. It is an operational data engine that effectively combines the power of search engines with columnar databases. Rockset is designed for building real-time applications and for ad-hoc interactive data science on large, diverse data sets. Examples of such data powered applications include live search and recommendation systems, e-commerce personalization engines, IoT applications, 360-degree views of customer, cybersecurity forensics and stateful microservices. Rockset is not a transactional database so it is not designed to support OLTP workloads.

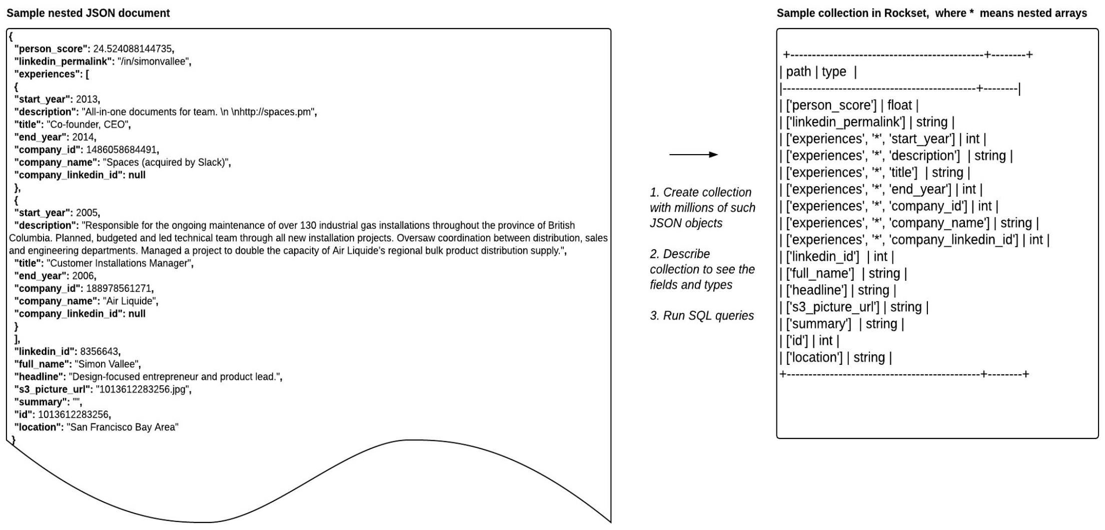
In this document, we describe the architecture of Rockset and the design choices and innovations which enable it to produce highly efficient execution of complex distributed queries across diverse data sets.

Data Model

Relational Document Model

Rockset stores data in a new type of *relational document* model. In traditional relational database systems, you must define a schema before adding records to a database. The schema is the structure supported by that particular database and 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, existing document-oriented databases contain documents, which are records that describe the data in the document, as well as the actual data. Documents can be complex with nested data to provide additional sub-categories of information about the object, and data must be denormalized when dealing with relational data in such systems.

However, Rockset does not require upfront schema definition or data denormalization since it handles semi-structured data formats such as JSON, Parquet, XML, CVS, TSV by indexing and storing them in 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 query-able in Rockset. This means new documents are never rejected if they unexpectedly show up with new fields or data formats. In the case of nested JSON, the documents are "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 schemaless aspect of Rockset completely eliminates the need for data modeling when ingesting different structured data sets from relational databases.


Sample nested JSON document
Sample collection in Rockset, where * means nested arrays
1. Create collection with millions of such JSON objects
2. Describe collection to see the fields and types
3. Run SQL queries

```
{
 "person_score": 24.524088144735,
 "linkedin_permalink": "/in/simonvallee",
 "experiences": [
 {
 "start_year": 2013,
 "description": "All-in-one documents for team. \n \nhttp://spaces.pm",
 "title": "Co-founder, CEO",
 "end_year": 2014,
 "company_id": 1486058684491,
 "company_name": "Spaces (acquired by Slack)",
 "company_linkedin_id": null
 },
 {
 "start_year": 2005,
 "description": "Responsible for the ongoing maintenance of over 130 industrial gas installations throughout the province of British
Columbia. Planned, budgeted and led technical team through all new installation projects. Oversaw coordination between distribution, sales
and engineering departments. Managed a project to double the capacity of Air Liquide's regional bulk product distribution supply.",
 "title": "Customer Installations Manager",
 "end_year": 2006,
 "company_id": 188978561271,
 "company_name": "Air Liquide",
 "company_linkedin_id": null
 }
 ],
 "linkedin_id": 8356643,
 "full_name": "Simon Vallee",
 "headline": "Design-focused entrepreneur and product lead.",
 "s3_picture_url": "1013612283256.jpg",
 "summary": "",
 "id": 1013612283256,
 "location": "San Francisco Bay Area"
}
```

```
 +--------------------------------------------+--------+
| path | type  |
|--------------------------------------------+--------|
| ['person_score'] | float |
| ['linkedin_permalink'] | string |
| ['experiences', '*', 'start_year'] | int |
| ['experiences', '*', 'description']  | string |
| ['experiences', '*', 'title']  | string |
| ['experiences', '*', 'end_year'] | int |
| ['experiences', '*', 'company_id'] | int |
| ['experiences', '*', 'company_name'] | string |
| ['experiences', '*', 'company_linkedin_id'] | int |
| ['linkedin_id']  | int |
| ['full_name']  | string |
| ['headline'] | string |
| ['s3_picture_url'] | string |
| ['summary']  | string |
| ['id'] | int |
| ['location'] | string |
+--------------------------------------------+--------+
```

Figure 1: 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 a commit marker can be used in case the user wants to block queries until all the previous data has been indexed.

STRONG DYNAMIC TYPING

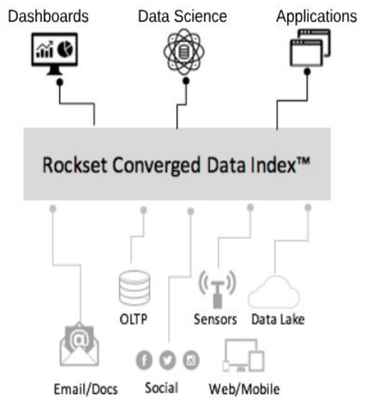
Types can be static (known at compile time, or, in our case, when the query is parsed, before it starts executing) or dynamic (known only at runtime). A type system can be weak (the system tries to avoid type errors and coerces eagerly) or strong (type mismatches cause errors, coercion requires explicit 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.

CONVERGED INDEXING™

Rockset is built using converged indexing which is a patent-pending combination of:

- Inverted index
- Columnar index
- Document index



As a result, it is optimized for key-value, time-series, document, search, aggregation and graph type 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 low milliseconds.

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 re-index of the entire document. Traditional search indexes tend to suffer from re-indexing storms because even if a 100-byte field in a 20K document is updated, they are forced to re-index 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 for potentially running 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.

Distributed Query Processing

Rockset provides a full SQL interface to query the data - including filters, aggregations and joins. Rockset's query aggregators are written in C++ to ensure low tail latencies. They act as the gateway to our distributed system and are responsible for querying the leaves, aggregating the results and returning the results to the client. In essence, Rockset scales with the latency requirements of queries. If a query needs lots of processing, the query can be parallelized and executed by spinning up several aggregators (which primarily serve the CPU and memory requirements) and bringing up many more leaf nodes (which primarily serve the storage requirements), so that results can be delivered within the desired latencies. A control loop optimizes the number of aggregators required and determines how leaf nodes need to be split and re-sharded based on size and load.

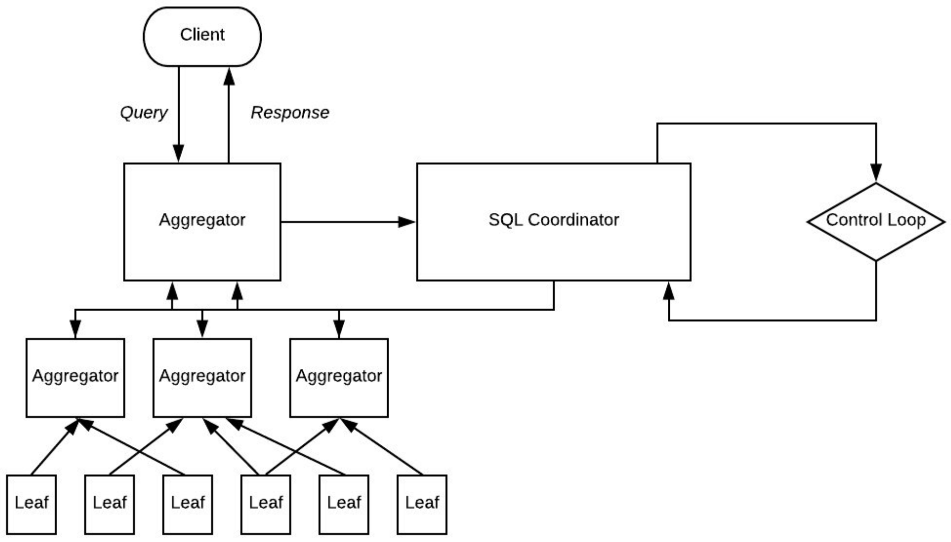


Figure 3: Distributed Query Processing

Our SQL coordinator parses the query, analyzes the requirements, plans the execution and schedules the tasks for the query execution. It also constantly monitors the allocated resources as a query executes so that in case a node dies, it can immediately re-execute that part of the plan somewhere else, ensuring that the query does not fail. The aggregators and leaf nodes required for the query execution intelligently build a self-aware mesh as they come up, rather than being polled top-down. In general, a bottom-up approach results in lower latencies because the nodes do not wait to be polled for resource availability and they can also preemptively send results for certain types of queries (example: order by). Traditional single node systems have already adopted this type of bottom-up optimization, but it is complicated to implement in a distributed system so most distributed databases still rely on the top-down approach. Rockset is one of the few distributed databases to successfully implement a bottom-up approach.

The Rockset query optimizer uses a hybrid approach in terms of rule based and cost-based optimizations. Our rule-based optimization process includes identifying the most general execution plan, identifying patterns and simplifying them. Our cost-based optimization involves identifying patterns in the query, making an execution plan based on where the data lives, how much data needs to be scanned, the amount of buffer to be used and the number of results. It then chooses the access patterns, the intermediate operators, the JOIN strategies and finally estimates CPU. Since Rockset is 100% cloud-native and delivered as SaaS, our optimizer design gives us the opportunity to use machine learning (ML models) to train it for each individual customer based on their specific query patterns going forward, rather than using generic optimization rules.

Disaggregated, 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. Hence, Rockset has been architected very differently to optimize price/performance of the database by aggressively exploiting the fundamentals of cloud elasticity.

Rockset uses a discrete microservices architecture, with all key components being containerized using Kubernetes for a cloud-agnostic approach. We use RocksDB-Cloud as an embedded storage engine, along with a custom resource scheduler and custom C++ query processing engine.

Each block in the architecture block diagram below is designed to be scalable on-demand using cloud resources. The data set is broken into shards and each shard can have multiple replicas. Data is re-sharded and additional replicas may be created as needed for load balancing like typical distributed systems. However, a unique aspect of Rockset is that some replicas are intelligently placed in local SSDs and some in durable cloud storage. Since cloud storage supports very high bandwidth but does not provide low latency, we parallelize for fast data retrieval using multiple nodes and ensure that queries are always served out of SSDs. The control loop monitors CPU load, I/O load and disk storage across all the nodes and makes auto-scaling decisions based on a combination of all three. Rockset's disaggregated architecture allows for every component of the stack to be auto-scaled up and down in the most efficient way possible to meet a user's SLA requirements. Our resource scheduler runs every 5 seconds to provision *and shed* resources.

The tailer-leaf-aggregator architecture is among the most efficient and scalable of modern data systems - it is the same architecture used by Facebook & Google for newsfeed, search, ads, spam, graph indexing.

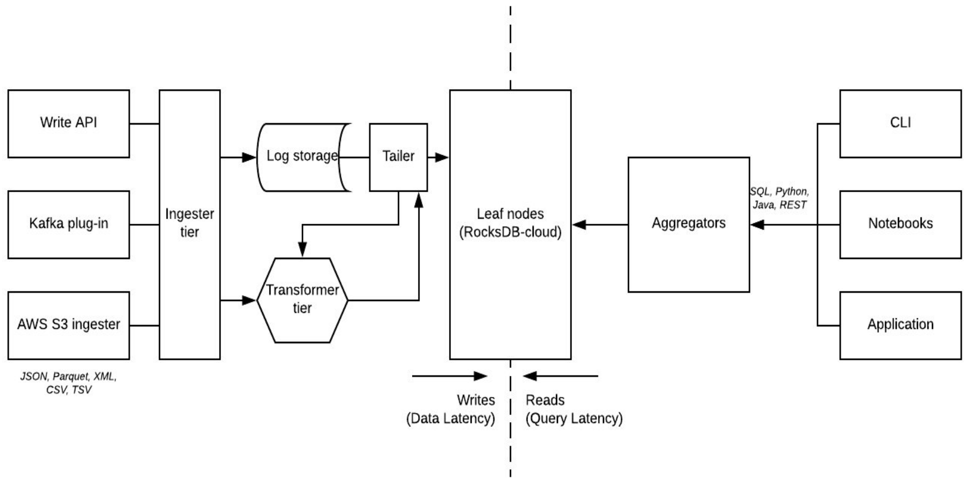


Figure 4: Disaggregated architecture

The ingest side is auto-scaled on demand (within the limits set by the user) to handle bursty ingest loads without impacting the data latency - ensuring that new writes are made available for querying within the desired SLA requirements. There are multiple ways to schemalessly ingest new data into Rockset:

- Write API - a generic write API that can continuously ingest data from other existing systems

- Source Integrations – native integrations with data streams such as Amazon Kinesis & Kafka, data lakes such as Amazon S3 and data bases such as Amazon RDS and DynamoDB

The query side is auto-scaled separately on demand (within the limits set by the user) to handle more complex or more concurrent queries without impacting the read latency. There are multiple ways to query data from Rockset:

- CLI prompt -with SQL support
- Python SDK -with a Query Builder (similar to SQLAlchemy)
- Java SDK

Rockset separates durability from performance. It uses hierarchical storage, with the help of RocksDB-Cloud which is a high performance embedded storage engine optimized for SSDs. RocksDB is used in production at Facebook, LinkedIn, Yahoo, Netflix, Airbnb, Pinterest and Uber. RocksDB-Cloud extends the core RocksDB engine in order to provide three main advantages for Rockset:

- A RocksDB-Cloud instance is durable. Continuous and automatic replication of database data and metadata to Cloud Storage (e.g. AWS S3). In the event that the RocksDB-Cloud machine dies, another process on any other EC2 machine can reopen the same RocksDB-Cloud database
- 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.
- A RocksDB-Cloud instance automatically places hot data in SSD 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.

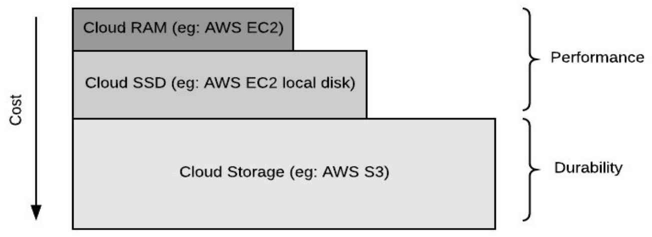

Cloud RAM (eg: AWS EC2)
Cloud SSD (eg: AWS EC2 local disk)
Cloud Storage (eg: AWS S3)
Cost
Performance
Durability

Figure 6: Hierarchical storage

SECURITY

USE OF CLOUD INFRASTRUCTURE

Rockset SaaS uses cloud native best practices and exploits the underlying security policies of the public cloud it is hosted on. Architecturally, Rockset SaaS 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 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. 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

The only ways to generate traffic inside Rockset are through the VPN server, Rockset Console, and Rockset API. 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 AWS 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 AWS 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 provides a way for customers to provide their own KMS master key.

ACCESS CONTROLS

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

- Access to the VPN server requires a TLS auth key, a CA certificate, and the user’s individual password. 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. 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 Rockset console. Each API key can only access the resources granted to the user that created

the key. Users can create and manage their own API keys, and only the admin can create and revoke any user's API keys.

Rockset supports Role Based Access Controls (RBAC) to ensure that individual users have the right level of permissions and access to view and manipulate different collections, as granted by the admin in the Rockset console.

Comparing Options For Real Time Applications On Complex Data

	Rockset	ElasticSearch	MongoDB	Aurora
Schemaless	✓	✓	✓	X
SQL	✓	X	X	✓
Distributed Query Processing	✓	✓	✓	X
Continuous Ingest	✓	X	X	X
Auto-performance Management	✓	X	X	X
Serverless	✓	X	X	✓
Transactional	X	X	✓	✓

Conclusion And Next Steps

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

- Power of SQL
- Flexibility of Schemaless
- Ease of Serverless

Enabling enterprises to put all their data to work is a complex challenge that inspires us everyday. With Rockset, our vision is that instead of spending 80% of their time preparing and organizing data, developers and data scientists can start working with their complex data sets within a few hours, - and experience the real-time performance made possible with modern cloud-native architecture and design.

Version 2.0