Evaluating Data Latency for Real-Time Databases

Measuring the time from when data is produced to when it is queryable

Abstract

A real-time database is one that can sustain a high write rate for new incoming data while allowing applications to query this fresh data. RockBench is a benchmark designed to measure the data latency of a real-time database. This paper describes RockBench in detail and the results from running the benchmark on Rockset, a real-time indexing database.

Kshitij Wadhwa
Hieu Pham

September 2020

Executive Summary 3
Testing Methodology 3
The Data Generator 4
Data Generation 5
Seeding the Database 5
The Data Latency Evaluator 6
Running the Benchmark on Rockset 6
Rockset Benchmark Results 7
Conclusion 10
Appendix 12
About Rockset 14

Executive Summary

A real-time database is one that can sustain a high write rate of new incoming data, while at the same time allow applications to make decisions based on fresh data. There is usually a time lag between when the data is produced and when it is available to query. This is called the data latency, or end-to-end latency, of the database. The data latency is different from a query latency, which is what is typically used to evaluate the performance of databases.

Data latency is one of the distinguishing factors that differentiates one real-time database from another. It is an important measure for developers of low-latency applications, like real-time personalization, IoT automation and security analytics, where speed is critical.

We designed a benchmark called RockBench that measures the data latency of a real-time database. RockBench is designed to continuously stream documents in batches of fixed size to a database and also calculate and report the data latency by querying the database at fixed intervals.

In this document, we describe RockBench in detail, including the data set that it generates and the algorithm it uses to calculate the data latency of a database. We ran the benchmark on Rockset, a real-time indexing database, to demonstrate how the benchmark can be used to test the ingestion of more than 1 billion events in a 24-hour period and measure the data latency of the database, which was in the 1-2 second range for most configurations tested.

Testing Methodology

The benchmark has two parts: a data generator and a data latency evaluator. The data generator writes events into the database at a specified speed. The data latency evaluator measures the data latency of every event and publishes aggregated results.

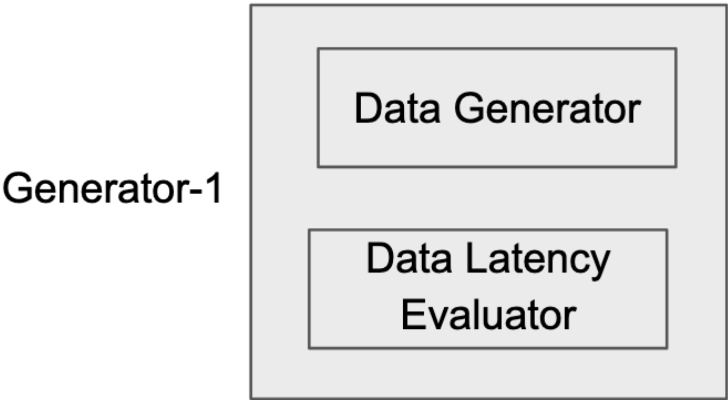

FIGURE 1: THE BENCHMARK COMPRISES A DATA GENERATOR AND A DATA LATENCY EVALUATOR

We use multiple instances of the benchmark when testing, where each instance has a unique *generator_identifier*.

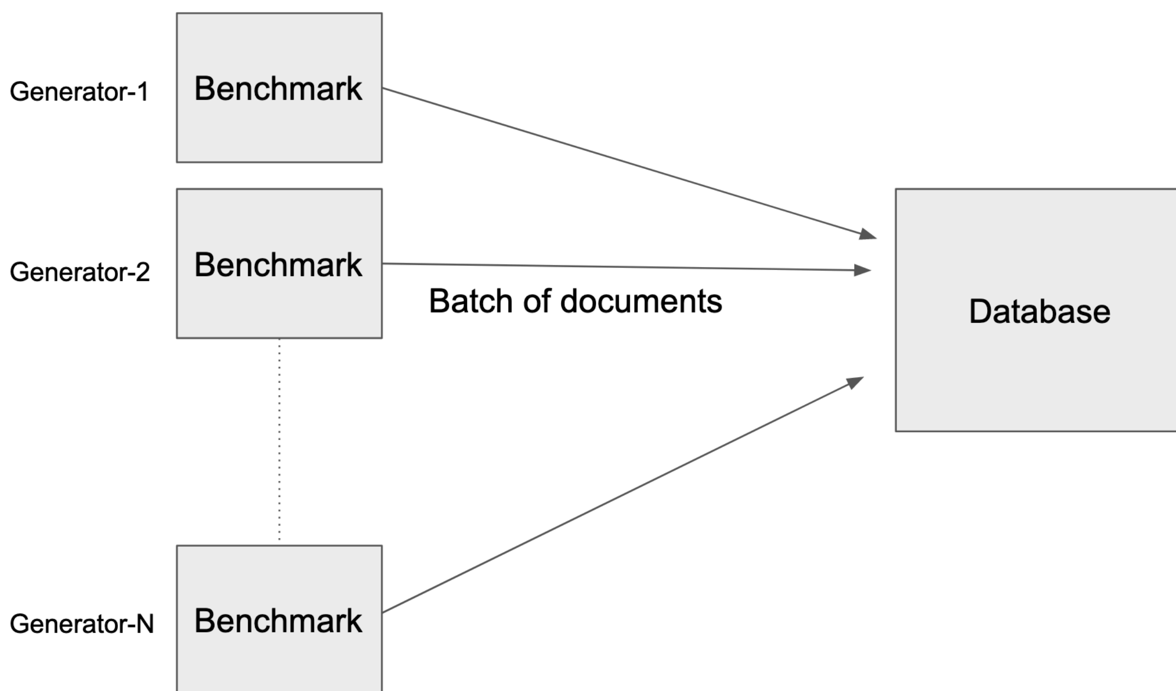

FIGURE 2: MULTIPLE INSTANCES OF THE BENCHMARK CONNECT TO THE DATABASE UNDER TEST

The Data Generator

The data generator writes documents to the database at a specified rate. It is written in Go. It uses goroutines to generate a batch of documents of fixed size at random, deposits them into the database and continues with this iteration. Multiple instances of the data generator can be run in parallel against the same database. Typically, the data generator is run in the same region where the database is located. This is done to eliminate any network latencies between the application and the database. You can find the code for the data generator at http://github.com/rockset/rockbench.

Every database has a write API and this generator uses that API to insert new documents into the database. The write API can write multiple documents in a single write call to the database. In real-life use cases, most applications batch a few events at a time before inserting those events into the database. This benchmark refers to this using the parameter *batch_size*.

Some databases use a specialized bulk-load mechanism to load stationary data , but this benchmark is not meant to measure that functionality. This benchmark is intentionally designed

to measure the data latency of streaming data, when new data is arriving at a fast rate and needs to be immediately queried and turned into real-time decisions.

Data Generation

The data generator generates documents, each document of size 1KB. Every document represents an event.

A single document has 60 fields with nested objects and arrays. Each document is uniquely identified by an *_id* field, and the data generator generates a uuid and stores it in the *_id* of every document. This means that the data generator adds new documents to the database and does not overwrite an existing document.

There is a special field called *_event_time* that is filled by the data generator when the document is created and before it is written to the database. There is another special field called *generator_identifier* in every document. The *_event_time* and *generator_identifier* are needed to calculate the end-to-end data latency that we describe in a later section. All other fields in the document are generated at random using a standard JSON generator at https://github.com/bxcodec/faker.

- The *_event_time* field is filled by the data generator by using the system-time of the machine it is running on.
- The *generator_identifier* is a 64-bit number that is picked at random by a data-generator instance, each document produced by that instance of the data generator has that field inside it.

Refer to the Appendix for a sample document generated by the data generator.

Generated documents are then batched before being sent to the database. In many databases, bigger batches tend to reduce the impact of various overheads, such as network.

Seeding the Database

The data generator initially generates 1 billion documents into the database. Since each document is 1KB in size, this results in a total data set size of 1TB. No measurements were taken while seeding the database.

This step is needed so that all succeeding benchmark measurements are done on a reasonably sized database.The reason is that, in many databases, streaming data into an empty database is much less CPU intensive compared to streaming data into a large database. Therefore, in order for this benchmark to match most production settings, we prepared a 1TB database before starting the benchmark.

The Data Latency Evaluator

The data latency evaluator generates a metric that shows the data latency of records that are arriving into the database. It is written in Go and uses goroutines to query the database at fixed intervals.

The data latency evaluator runs alongside the data generator and uses the *generator_identifier* described before to measure the latency on documents originating from that generator. The reason we use *generator_identifier* is to avoid the effect of clock skew on our measurements Since *_event_time* on a document is based on the clock time of the generator, we only want to use the data latency evaluator that is on the same machine to perform our latency measurements.

Every 30 seconds, the following query is made to retrieve the most recent document that was written to the database.

```
SELECT UNIX_MICROS(_event_time) FROM <collection-name>
   WHERE generator_identifier = '<identifier>'
   ORDER BY _event_time DESC
   LIMIT 1
```

The *_event_time* of that document is then subtracted from the current time of the machine to arrive at the data latency of the document. This measurement also includes round-trip latency—the time required to run the query and get results from the database back to the client. This metric is published to a Prometheus server and the p50 and p95 latencies are calculated across all evaluators.

Running the Benchmark on Rockset

To show how RockBench works in practice, we ran the benchmark on Rockset, using two different Rockset Virtual Instances. Each Virtual Instance was configured with fixed compute and storage. Rockset's compute autoscaling and storage autoscaling were switched off. In this experiment, we tested Rockset's 2XLarge and 4XLarge Virtual Instances. The table below shows the allocated compute and memory for both instance types.

Instance	Allocated Compute	Allocated Memory
2XLarge	64 vCPU	512 GiB
4XLarge	128 vCPU	1024 GiB

FIGURE 3: ALLOCATED COMPUTE AND MEMORY FOR THE ROCKSET VIRTUAL INSTANCES TESTED

The data generator used Rockset's write API to write new documents to the database. We ran the benchmark using write rates ranging from 4K to 24K events/sec and with batch sizes of 50 and 500 documents per write request.

We first seeded the database as described above and this seeding process took a few hours. No measurements were taken during this seeding phase. After the seeding phase was complete, we started the benchmark run to write new documents into Rockset and waited for the data latency values to stabilize before recording them. Both p50 and p95 latencies were recorded.

The data generators and data latency evaluators were run in the same AWS region as the Rockset database. We set ingest rate limiting for 2XLarge and 4XLarge to 12MB/sec and 24MB/sec respectively, which are the default values for these Virtual Instances.

Rockset Benchmark Results

Figure 4 shows the results of running the RockBench benchmark on Rockset at a batch size of 50. We record p50 and p95 data latency in milliseconds for each ingest rate. Each event is 1KB, so a 12 MB/sec ingest rate corresponds to 12K events/sec. Figures 5 and 6 graph the p50 and p95 data latencies measured at each ingest rate tested.

Batch size 50

	Data latency p50/p95 (in msec)					
Rockset config	**4K events/sec (4MB/sec)**	**8K events/sec (8MB/sec)**	**10K events/sec (10MB/sec)**	**12K events/sec (12MB/sec)**	**16K events/sec (16MB/sec)**	**20K events/sec (20MB/sec)**
Rockset 2XLarge	394/798	740/1249	1490/2403	6407/10489		
Rockset 4XLarge	340/765	630/1014	746/1053	914/1269	1352/1842	2100/3000

FIGURE 4: ROCKSET DATA LATENCY MEASURED AT A BATCH SIZE OF 50

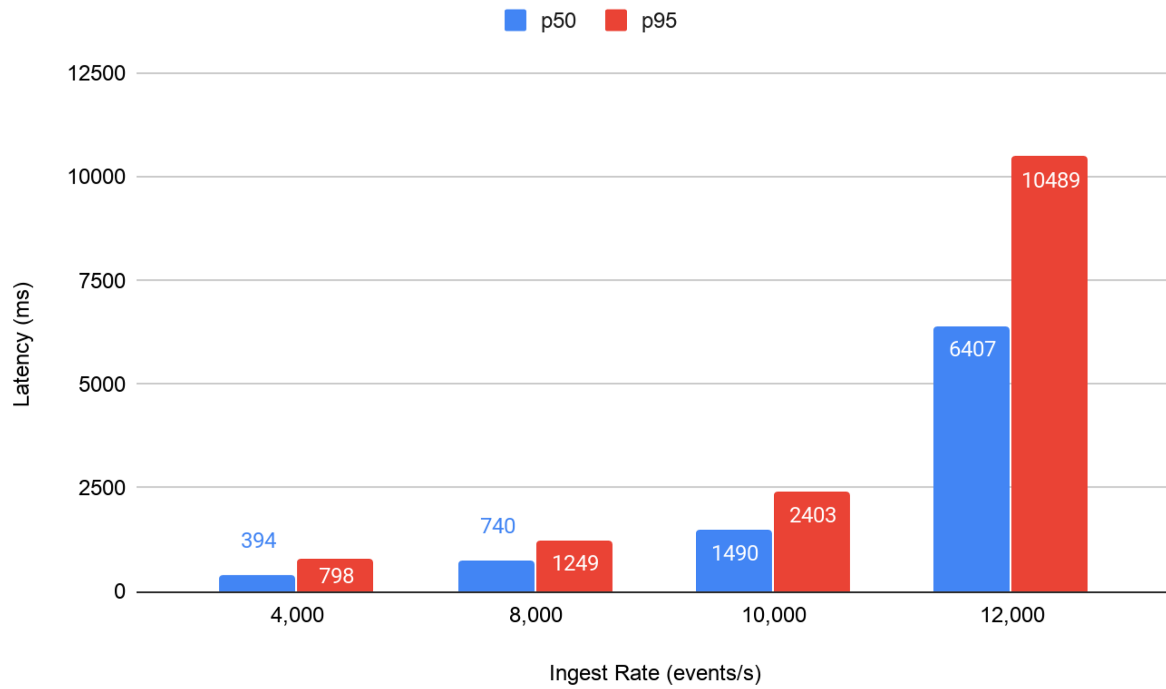

FIGURE 5: ROCKSET DATA LATENCY USING A 2XLARGE VIRTUAL INSTANCE AT A BATCH SIZE OF 50

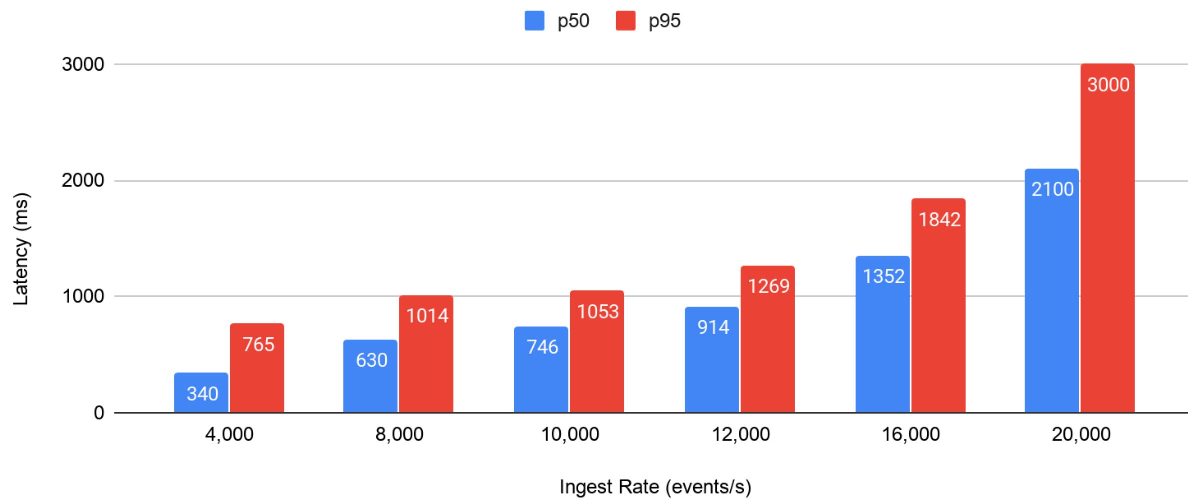

FIGURE 6: ROCKSET DATA LATENCY USING A 4XLARGE VIRTUAL INSTANCE AT A BATCH SIZE OF 50

Figure 7 shows the results of running the RockBench benchmark on Rockset at a batch size of 500. Figures 8 and 9 graph the p50 and p95 data latencies measured at each ingest rate tested.

Batch size 500

	Data latency p50/p95 (in msec)						
Rockset config	**4K events/sec (4MB/sec)**	**8K events/sec (8MB/sec)**	**10K events/sec (10MB/sec)**	**12K events/sec (12MB/sec)**	**16K events/sec (16MB/sec)**	**20K events/sec (20MB/sec)**	**24K events/sec (24MB/sec)**
Rockset 2XLarge	594/913	893/1282	1126/1589	2590/4898			
Rockset 4XLarge	638/1027	783/1109	1023/1412	1136/1593	1513/1880	1700/1900	4539/7567

FIGURE 7: ROCKSET DATA LATENCY MEASURED AT A BATCH SIZE OF 500

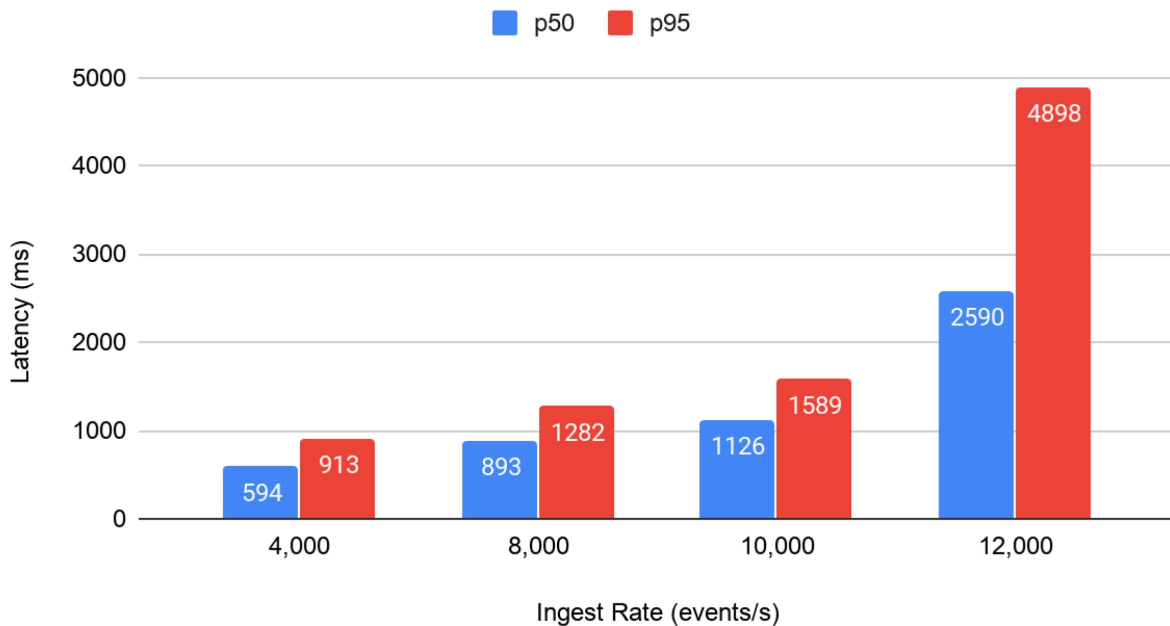

FIGURE 8: ROCKSET DATA LATENCY USING A 2XLARGE VIRTUAL INSTANCE AT A BATCH SIZE OF 500

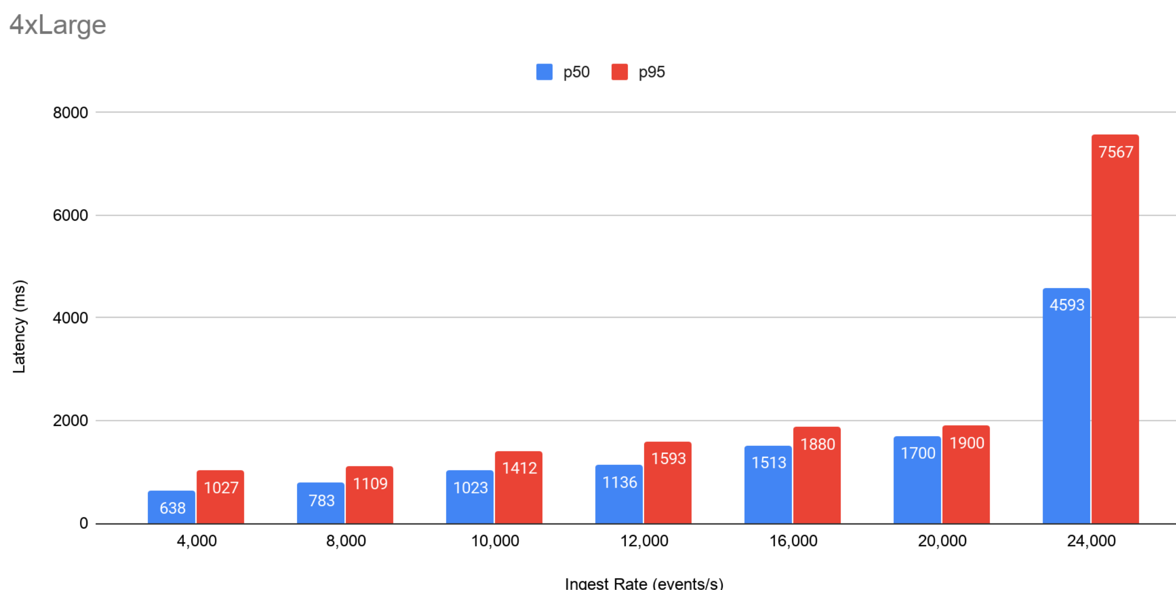

FIGURE 9: ROCKSET DATA LATENCY USING A 4XLARGE VIRTUAL INSTANCE AT A BATCH SIZE OF 500

We observe that as data is generated and ingested at higher rates, the data latency—the time from data being generated to it being available to query—increases. Application developers should use the resultant latency metrics to gauge how Rockset can meet their application requirements.

For example, if an application needs to ingest 1 billion events per day (approx. 12 MB/s based on events of size 1KB) at a batch size of 50, a Rockset 4XLarge Virtual Instance will provide p50 and p95 data latency of 0.914 sec and 1.269 sec respectively. This means that a developer can expect to consistently query data, generated at a rate of 1 billion events per day, within about a second of when the data was produced.

Using a Rockset Virtual Instance with more compute and memory resources decreases data latency. If the application is such that it requires a lower data latency, one can select a Rockset Virtual Instance with greater resources.

Rockset uses a specialized bulk-load mechanism to ingest and index stationary data at terabytes/hour, but this benchmark does not exercise that functionality.

Conclusion

When using real-time, streaming data sources, developers will need to design their applications to support specific write rates and data latencies--the time from when the data is produced to when it can be queried by the application. RockBench provides developers a simple and reliable method of measuring the data latency of the underlying database under different write rates.

In our testing, we showed how one could measure the data latencies for Rockset under various assumptions for write rate, batch size, and instance size to determine the configuration that

would best meet application requirements. The RockBench code is pluggable and extensible and the benchmark can be extended to run on other real-time databases as well. By testing different types and configurations of databases, developers can make informed decisions on which real-time database setup fits the requirements of their application.

Appendix

This is a sample document generated by the data generator:


```
{
        "_id": "1",
        "guid": "7cb1189d-fe4f-445f-bc7e-10840ec2ce61",
        "isActive": false,
        "balance": 1852,
        "picture": "img.png",
        "age": 33,
        "name": "Tyler Wallis",
        "company": "Celmax",
        "email": "tyler.wallis@celmax.gov",
        "phone": "+1 (271) 916 8502",
        "address": {
            "street": "709 Sherman Road",
            "city": "Norfolk",
            "zipcode": "8033",
            "coordinates": {
                "lat": 59.95,
                "long": 51.26
            }
        },
        "about": "Metus i elit ac pulvinar turpis.",
        "registered": "1923",
        "tags": [
            "Bibendum.",
            "Justo.",
            "Dictum.",
            "Facilisis.",
            "Facilisis.",
            "Dui.",
            "Orci.",
            "Ex.",
            "At.",
            "Ipsum."
        ],
        "friends": {
            "friend1": {
                "name": "Craig Hancock",
                "age": 28
            },
            "friend2": {
                "name": "Grace Sprowl",
                "age": 35
            },
            "friend3": {
                "name": "Aiden Leath",
                "age": 29
            },
            "friend4": {
                "name": "Ivan Muldoon",
                "age": 37
            },
            "friend5": {
                "name": "Selma Tuck",
                "age": 38
            },
            "friend6": {
                "name": "Rico Flinn",
                "age": 28
            },
```

```
        "friend7": {
            "name": "Rickey Porras",
            "age": 34
        },
        "friend8": {
            "name": "Max Milling",
            "age": 29
        },
        "friend9": {
            "name": "Eleanore Verde",
            "age": 36
        },
        "friend10": {
            "name": "Raymond Brandt",
            "age": 37
        }
    },
     "greeting": "Dictum blandit nam at et ipsum proin eget metus.
                  Orci sem id non non ligula ante purus, euismod quisque
                  curabitur tincidunt auctor. Ipsum lacinia pulvinar i
                  nisi ex velit, elit posuere in curabitur curabitur
                  non sit cras risus nunc, sit ultricies lectus pellentesque.
                  Suscipit u ac i. Pharetra dolor pretium dia tristique maximus.
                  Eget nibh pulvinar euismod pellentesque mattis enim, vehicula
                  at ipsum laoreet ut. Pharetra id ante ac amet vel ex."

}
```

About Rockset

Rockset is a real-time indexing database service for serving low latency, high concurrency analytical queries at scale. It builds a Converged Index™ on structured and semi-structured data from OLTP databases, streams and lakes in real time and exposes a RESTful SQL interface.

Find out more at rockset.com

Connect with us at support@rockset.com