Instagram Engineering Follow
Mar 5 · 6 min read

Open-sourcing a 10x reduction in Apache Cassandra tail latency

At Instagram, we have one of the world's largest deployments of the Apache Cassandra database. We began using Cassandra in 2012 to replace Redis and support product use cases like fraud detection, Feed, and the Direct inbox. At first we ran Cassandra clusters in an AWS environment, but migrated them over to Facebook's infrastructure when the rest of Instagram moved. We've had a really good experience with the reliability and availability of Cassandra, but saw room for improvement in read latency.

Last year Instagram's Cassandra team started working on a project to reduce Cassandra's read latency significantly, which we call Rocksandra. In this post, I will describe the motivation for this project, the challenges we overcame, and performance metrics in both internal and public cloud environments.

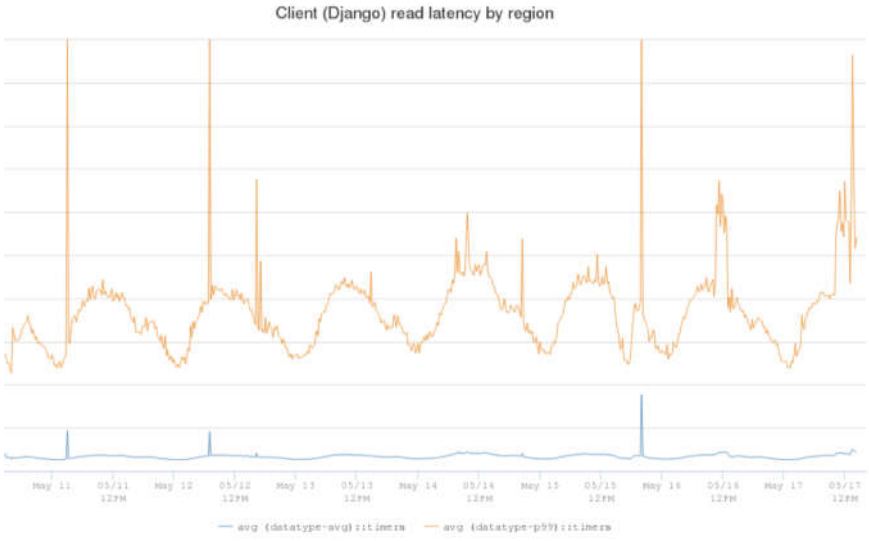
Motivation

At Instagram, we use Apache Cassandra heavily as a general key value storage service. The majority of Instagram's Cassandra requests are online, so in order to provide a reliable and responsive user experience for hundreds of millions of Instagram users, we have very tight SLA on the metrics.

Instagram maintains a 5–9s reliability SLA, which means at any given time, the request failure rate should be less than 0.001%. For performance, we actively monitor the throughput and latency of different Cassandra clusters, especially the P99 read latency.

Here's a graph that shows the client-side latency of one production Cassandra cluster. The blue line is the average read latency (5ms) and the orange line is the P99 read latency (in the range of 25ms to 60ms

and changing a lot based on client traffic).



After investigation, we found the JVM garbage collector (GC) contributed a lot to the latency spikes. We defined a metric called GC stall percentage to measure the percentage of time a Cassandra server was doing stop-the-world GC (Young Gen GC) and could not serve client requests. Here's another graph that shows the GC stall percentage on our production Cassandra servers. It was 1.25% during the lowest traffic time windows, and could be as high as 2.5% during peak hours.

The graph shows that a Cassandra server instance could spend 2.5% of runtime on garbage collections instead of serving client requests. The GC overhead obviously had a big impact on our P99 latency, so if we could lower the GC stall percentage, we would be able to reduce our P99 latency significantly.

Solution

Apache Cassandra is a distributed database with it's own LSM tree-based storage engine written in Java. We found that the components in the storage engine, like memtable, compaction, read/write path, etc., created a lot of objects in the Java heap and generated a lot of overhead to JVM. To reduce the GC impact from the storage engine, we considered different approaches and ultimately decided to develop a C++ storage engine to replace existing ones.

We did not want to build a new storage engine from scratch, so we decided to build the new storage engine on top of RocksDB.

RocksDB is an open source, high-performance embedded database for key-value data. It's written in C++, and provides official API language bindings for C++, C, and Java. RocksDB is optimized for performance, especially on fast storage like SSD. It's widely used in the industry as the storage engine for MySQL, mongoDB, and other popular databases.

Challenges

We overcame three main challenges when implementing the new storage engine on RocksDB.

The first challenge was that Cassandra does not have a pluggable storage engine architecture yet, which means the existing storage engine is coupled together with other components in the database. To find a balance between massive refactoring and quick iterations, we defined a new storage engine API, including the most common read/write and streaming interfaces. This way we could implement the new storage engine behind the API and inject it into the related code paths inside Cassandra.

Secondly, Cassandra supports rich data types and table schema, while RocksDB provides purely key-value interfaces. We carefully defined the encoding/decoding algorithms to support Cassandra's data model within RocksDB's data structure and supported same-query semantics as original Cassandra.

The third challenge was about streaming. Streaming is an important component for a distributed database like Cassandra. Whenever we join or remove a node from a Cassandra cluster, Cassandra needs to stream data among different nodes to balance the load across the cluster. The existing streaming implementation was based on the details in the current storage engine. Accordingly, we had to decouple them from each other, make an abstraction layer, and re-implement the streaming using RocksDB APIs. For high streaming throughput, we now stream data into temp sst files first, and then use the RocksDB ingest file API to bulk load them into the RocksDB instance at once.

Performance metrics

After about a year of development and testing, we have finished a first version of the implementation and successfully rolled it into several production Cassandra clusters in Instagram. In one of our production clusters, the P99 read latency dropped from 60ms to 20ms. We also observed that the GC stalls on that cluster dropped from 2.5% to 0.3%, which was a 10X reduction!

We also wanted to verify whether Rocksandra would perform well in a public cloud environment. We setup a Cassandra cluster in an AWS environment using three i3.8 xlarge EC2 instances, each with 32 cores CPU, 244GB memory, and raid0 with 4 nvme flash disks.

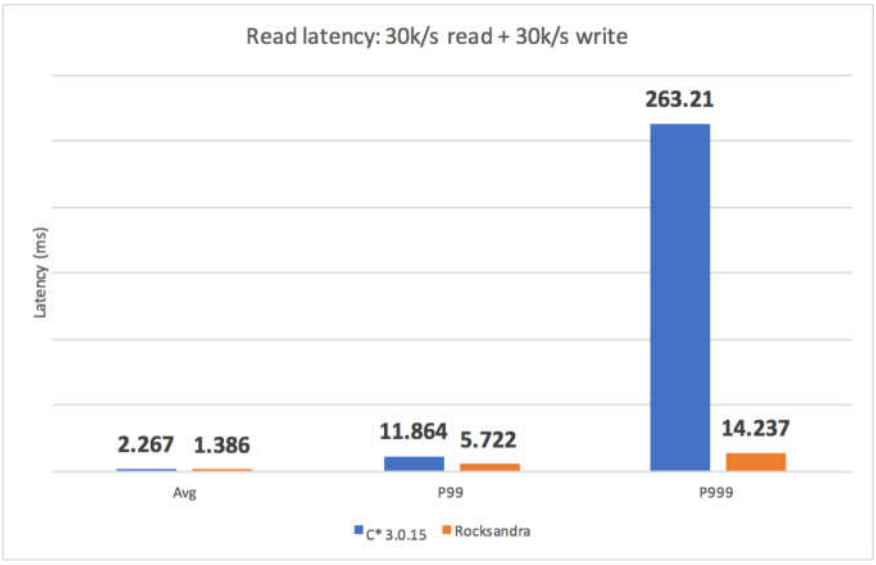
We used NDBench for the benchmark, and the default table schema in the framework:

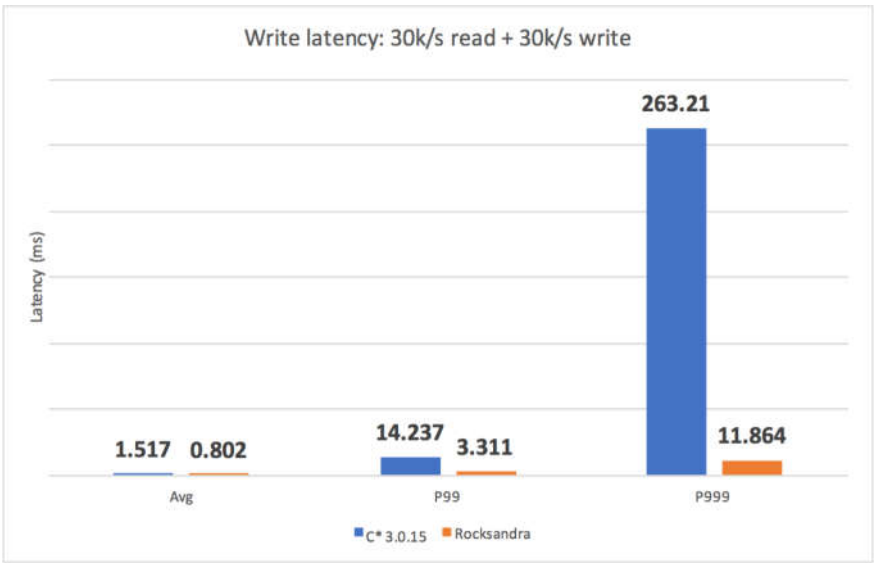
```
 TABLE emp (
  emp_uname text PRIMARY KEY,
emp_dept text,
emp_first text,
```

```
emp_last text
 )
```

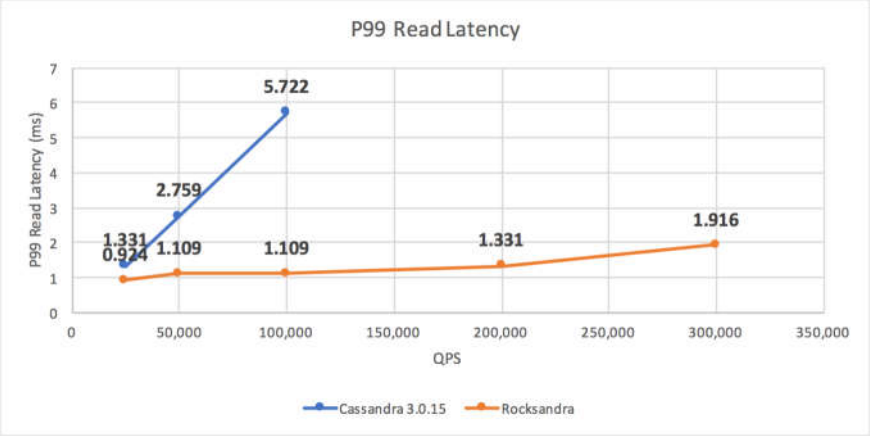
We pre-loaded 250M 6KB rows into the database (each server stores about 500GB data on disk). We configured 128 readers and 128 writers in NDBench.

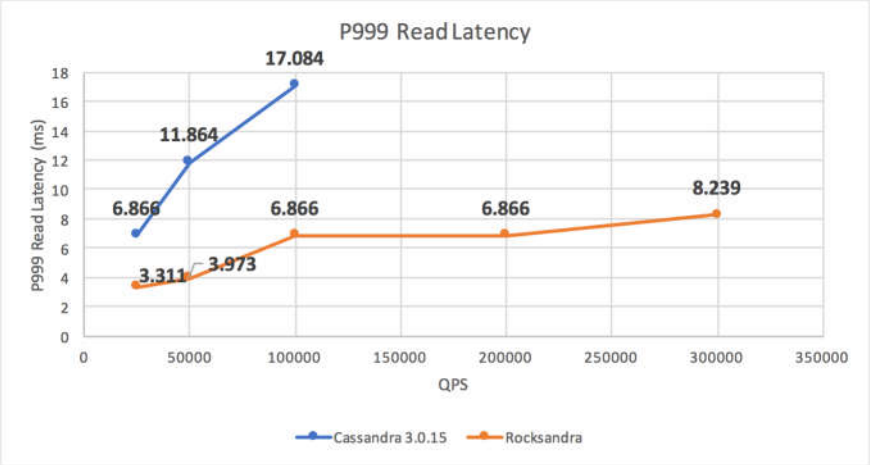
We tested different workloads and measured the avg/P99/P999 read/write latencies. As you can see, Rocksandra provided much lower and consistent tail read/write latency.





We also tested a read-only workload and observed that, at similar P99 read latency (2ms), Rocksandra could provide 10X higher read throughput (300K/s for Rocksandra vs. 30K/s for C* 3.0).





Future work

We have open sourced our Rocksandra code base and benchmark framework, which you can download from Github to try out in your own environment! Please let us know how it performs.

As our next step, we are actively working on the development of more C* features support, like secondary indexes, repair, etc. We are also working on a C* pluggable storage engine architecture to contribute our work back to the Apache Cassandra community.

If you are in the Bay Area and are interested in learning more about our Cassandra developments, join us at our next meetup event here.

Dikang Gu is an infrastructure engineer at Instagram.