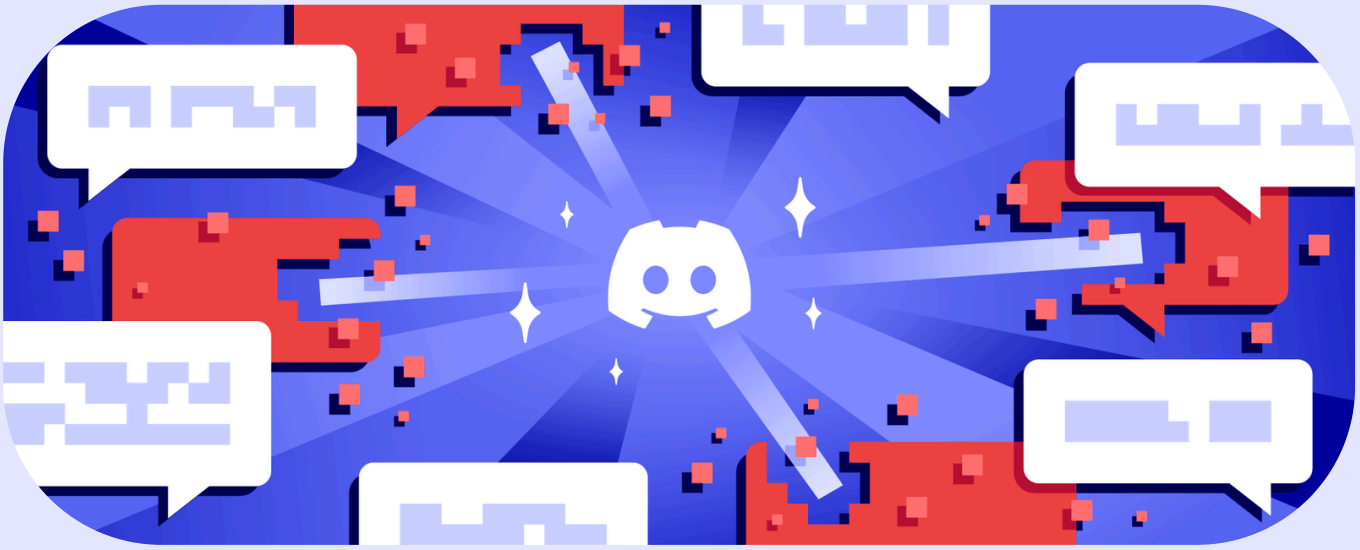

Engineering & Developers

How Discord Stores Trillions of Messages

Bo Ingram

In 2017, we wrote a blog post on how we store billions of messages. We shared our journey of how we started out using MongoDB but migrated our data to Cassandra because we were looking for a database that was scalable, fault-tolerant, and relatively low maintenance. We knew we'd be growing, and we did!

We wanted a database that grew alongside us, but hopefully, its maintenance needs wouldn't grow alongside our storage needs. Unfortunately, we found that to not be the case — our Cassandra cluster exhibited serious performance issues that required increasing amounts of effort to just maintain, not improve.

Almost six years later, we've changed a lot, and how we store messages has changed as well.

Our Cassandra Troubles

We stored our messages in a database called cassandra-messages. As its name suggests, it ran Cassandra, and it stored messages. In 2017, we ran 12 Cassandra nodes, storing billions of messages.

At the beginning of 2022, it had 177 nodes with trillions of messages. To our chagrin, it was a high-toil system — our on-call team was frequently paged for issues with the database, latency was unpredictable, and we were having to cut down on maintenance operations that became too expensive to run.

What was causing these issues? First, let's take a look at a message.

```
CREATE TABLE messages (
    channel_id bigint,
    bucket int,
    message_id bigint,
    author_id bigint,
    content text,
    PRIMARY KEY ((channel_id, bucket), message_id)
) WITH CLUSTERING ORDER BY (message_id DESC);
```

gistfile1.sql hosted with ❤ by GitHub view raw

The CQL statement above is a minimal version of our message schema. Every ID we use is a Snowflake, making it chronologically sortable. We partition our messages by the channel they're sent in, along with a bucket, which is a static time window. This partitioning means that, in Cassandra, all messages for a given channel and bucket will be stored together and replicated across three nodes (or whatever you've set the replication factor).

Within this partitioning lies a potential performance pitfall: a server with just a small group of friends tends to send orders of magnitude fewer messages than a server with hundreds of thousands of people.

In Cassandra, reads are more expensive than writes. Writes are appended to a commit log and written to an in memory structure called a memtable that is eventually flushed to disk. Reads, however, need to query the memtable and potentially multiple SSTables (on-disk files), a more expensive operation. Lots of concurrent reads as users interact with servers can hotspot a partition, which we refer to imaginatively as a "hot partition". The size of our dataset when combined with these access patterns led to struggles for our cluster.

When we encountered a hot partition, it frequently affected latency across our entire database cluster. One channel and bucket pair received a large amount of traffic, and latency in the node would increase as the node tried harder and harder to serve traffic and fell further and further behind.

Other queries to this node were affected as the node couldn't keep up. Since we perform reads and writes with quorum consistency level, all queries to the nodes that serve the hot partition suffer latency increases, resulting in broader end-user impact.

Cluster maintenance tasks also frequently caused trouble. We were prone to falling behind on compactions, where Cassandra would compact SSTables on disk for more performant reads. Not only were our reads then more expensive, but we'd also see cascading latency as a node tried to compact.

We frequently performed an operation we called the "gossip dance", where we'd take a node out of rotation to let it compact without taking traffic, bring it back in to pick up hints from Cassandra's hinted handoff, and then repeat until the compaction backlog was empty. We also spent a large amount of time tuning the JVM's garbage collector and heap settings, because GC pauses would cause significant latency spikes.

Changing Our Architecture

Our messages cluster wasn't our only Cassandra database. We had several other clusters, and each exhibited similar (though perhaps not as severe) faults.

In our previous iteration of this post, we mentioned being intrigued by ScyllaDB, a Cassandra-compatible database written in C++. Its promise of better performance, faster repairs, stronger workload isolation via its shard-per-core architecture, and a garbage collection-free life sounded quite appealing.

Although ScyllaDB is most definitely not void of issues, it is void of a garbage collector, since it's written in C++ rather than Java. Historically, our team has had many issues with the garbage collector on Cassandra, from GC pauses affecting latency, all the way to super long consecutive GC pauses that got so bad that an operator would have to manually reboot and babysit the node in question back to health. These issues were a huge source of on-call toil, and the root of many stability issues within our messages cluster.

After experimenting with ScyllaDB and observing improvements in testing, we made the decision to migrate all of our databases. While this decision could be a

blog post in itself, the short version is that by 2020, we had migrated every database but one to ScyllaDB.

The last one? Our friend, cassandra-messages.

Why hadn't we migrated it yet? To start with, it's a big cluster. With trillions of messages and nearly 200 nodes, any migration was going to be an involved effort. Additionally, we wanted to make sure our new database could be the best it could be as we worked to tune its performance. We also wanted to gain more experience with ScyllaDB in production, using it in anger and learning its pitfalls.

We also worked to improve ScyllaDB performance for our use cases. In our testing, we discovered that the performance of reverse queries was insufficient for our needs. We execute a reverse query when we attempt a database scan in the opposite order of a table's sorting, such as when we scan messages in ascending order. The ScyllaDB team prioritized improvements and implemented performant reverse queries, removing the last database blocker in our migration plan.

We were suspicious that slapping a new database on our system wasn't going to make everything magically better. Hot partitions can still be a thing in ScyllaDB, and so we also wanted to invest in improving our systems upstream of the database to help shield and facilitate better database performance.

Data Services Serving Data

With Cassandra, we struggled with hot partitions. High traffic to a given partition resulted in unbounded concurrency, leading to cascading latency in which subsequent queries would continue to grow in latency. If we could control the amount of concurrent traffic to hot partitions, we could protect the database from being overwhelmed.

To accomplish this task, we wrote what we refer to as data services — intermediary services that sit between our API monolith and our database clusters. When writing our data services, we chose a language we've been using more and more at Discord: Rust! We'd used it for a few projects previously, and it lived up to the hype for us. It gave us fast C/C++ speeds without having to sacrifice safety.

Rust touts fearless concurrency as one of its main benefits — the language should make it easy to write safe, concurrent code. Its libraries also were a great match for what we were intending to accomplish. The Tokio ecosystem is a tremendous foundation for building a system on asynchronous I/O, and the language has driver support for both Cassandra and ScyllaDB.

Additionally, we found it a joy to code in with the help the compiler gives you, the clarity of the error messages, the language constructs, and its emphasis on safety. We became quite fond of how once it compiled, it generally works. Most importantly, however, it lets us say we rewrote it in Rust (meme cred is very important).

Our data services sit between the API and our ScyllaDB clusters. They contain roughly one gRPC endpoint per database query and intentionally contain no business logic. The big feature our data services provide is request coalescing. If multiple users are requesting the same row at the same time, we'll only query the database once. The first user that makes a request causes a worker task to spin up in the service. Subsequent requests will check for the existence of that task and subscribe to it. That worker task will query the database and return the row to all subscribers.

This is the power of Rust in action: it made it easy to write safe concurrent code.

Let's imagine a big announcement on a large server that notifies @everyone: users are going to open the app and read the message, sending tons of traffic to the database. Previously, this might lead to a hot partition, and on-call would potentially need to be paged to help the system recover. With our data services, we're able to significantly reduce traffic spikes against the database.

The second part of the magic here is upstream of our data services. We implemented consistent hash-based routing to our data services to enable more effective coalescing. For each request to our data service, we provide a routing key. For messages, this is a channel ID, so all requests for the same channel go to the same instance of the service. This routing further helps reduce the load on our database.

These improvements help a lot, but they don't solve all of our problems. We're still seeing hot partitions and increased latency on our Cassandra cluster, just not quite as frequently. It buys us some time so that we can prepare our new optimal ScyllaDB cluster and execute the migration.

A Very Big Migration

Our requirements for our migration are quite straightforward: we need to migrate trillions of messages with no downtime, and we need to do it quickly because while the Cassandra situation has somewhat improved, we're frequently firefighting.

Step one is easy: we provision a new ScyllaDB cluster using our super-disk storage topology. By using Local SSDs for speed and leveraging RAID to mirror our data to a persistent disk, we get the speed of attached local disks with the durability of a persistent disk. With our cluster stood up, we can begin migrating data into it.

Our first draft of our migration plan was designed to get value quickly. We'd start using our shiny new ScyllaDB cluster for newer data using a cutover time, and then migrate historical data behind it. It adds more complexity, but what every large project needs is additional complexity, right?

We begin dual-writing new data to Cassandra and ScyllaDB and concurrently begin to provision ScyllaDB's Spark migrator. It requires a lot of tuning, and once we get it set up, we have an estimated time to completion: three months.

That timeframe doesn't make us feel warm and fuzzy inside, and we'd prefer to get value faster. We sit down as a team and brainstorm ways we can speed things up, until we remember that we've written a fast and performant database library that we could potentially extend. We elect to engage in some meme-driven engineering and rewrite the data migrator in Rust.

In an afternoon, we extended our data service library to perform large-scale data migrations. It reads token ranges from a database, checkpoints them locally via SQLite, and then firehoses them into ScyllaDB. We hook up our new and improved migrator and get a new estimate: nine days! If we can migrate data this quickly, then we can forget our complicated time-based approach and instead flip the switch for everything at once.

We turn it on and leave it running, migrating messages at speeds of up to 3.2 million per second. Several days later, we gather to watch it hit 100%, and we realize that it's stuck at 99.9999% complete (no, really). Our migrator is timing out reading the last few token ranges of data because they contain gigantic ranges of tombstones that were never compacted away in Cassandra. We compact that token range, and seconds later, the migration is complete!

We performed automated data validation by sending a small percentage of reads to both databases and comparing results, and everything looked great. The cluster held up well with full production traffic, whereas Cassandra was suffering increasingly frequent latency issues. We gathered together at our team onsite, flipped the switch to make ScyllaDB the primary database, and ate celebratory

cake!

Several Months Later...

We switched our messages database over in May 2022, but how's it held up since then?

It's been a quiet, well-behaved database (it's okay to say this because I'm not on-call this week). We're not having weekend-long firefights, nor are we juggling nodes in the cluster to attempt to preserve uptime. It's a much more efficient database — we're going from running 177 Cassandra nodes to just 72 ScyllaDB nodes. Each ScyllaDB node has 9 TB of disk space, up from the average of 4 TB per Cassandra node.

Our tail latencies have also improved drastically. For example, fetching historical messages had a p99 of between 40-125ms on Cassandra, with ScyllaDB having a nice and chill 15ms p99 latency, and message insert performance going from 5-70ms p99 on Cassandra, to a steady 5ms p99 on ScyllaDB. Thanks to the aforementioned performance improvements, we've unlocked new product use cases now that we have confidence in our messages database.

At the end of 2022, people all over the world tuned in to watch the World Cup. One thing we discovered very quickly was that goals scored showed up in our monitoring graphs. This was very cool because not only is it neat to see real-world events show up in your systems, but this gave our team an excuse to watch soccer during meetings. We weren't "watching soccer during meetings", we were "proactively monitoring our systems' performance."

We can actually tell the story of the World Cup Final via our message send graph. The match was tremendous. Lionel Messi was trying to check off the last accomplishment in his career and cement his claim to being the greatest of all time and lead Argentina to the championship, but in his way stood the massively talented Kylian Mbappe and France.

Each of the nine spikes in this graph represents an event in the match.

1. Messi hits a penalty, and Argentina goes up 1-0.

2. Argentina scores again and goes up 2-0.
3. It's halftime. There's a sustained fifteen-minute plateau as users chat a the match.
4. The big spike here is because Mbappe scores for France and scores again 90 seconds later to tie it up!
5. It's the end of regulation, and this huge match is going to extra time.
6. Not much happens in the first half of extra time, but we reach halftime and users are chatting.
7. Messi scores again, and Argentina takes the lead!
8. Mbappe strikes back to tie it up!
9. It's the end of extra time, we're heading to penalty kicks!
10. Excitement and stress grow throughout the shootout until France misses and Argentina doesn't! Argentina wins!

Coalesced messages per second

People all over the world are stressed watching this incredible match, but meanwhile, Discord and the messages database aren't breaking a sweat. We're way up on message sends and handling it perfectly. With our Rust-based data services and ScyllaDB, we're able to shoulder this traffic and provide a platform for our users to communicate.

We've built a system that can handle trillions of messages, and if this work is something that excites you, check out our careers page. We're hiring!

Bo Ingram
Senior Software Engineer @ Discord

RELATED ARTICLES

Product & Features

Gift Ideas for the Dedicat...

Product & Features

Your Discord Checkpoi...

Save and Display Your...