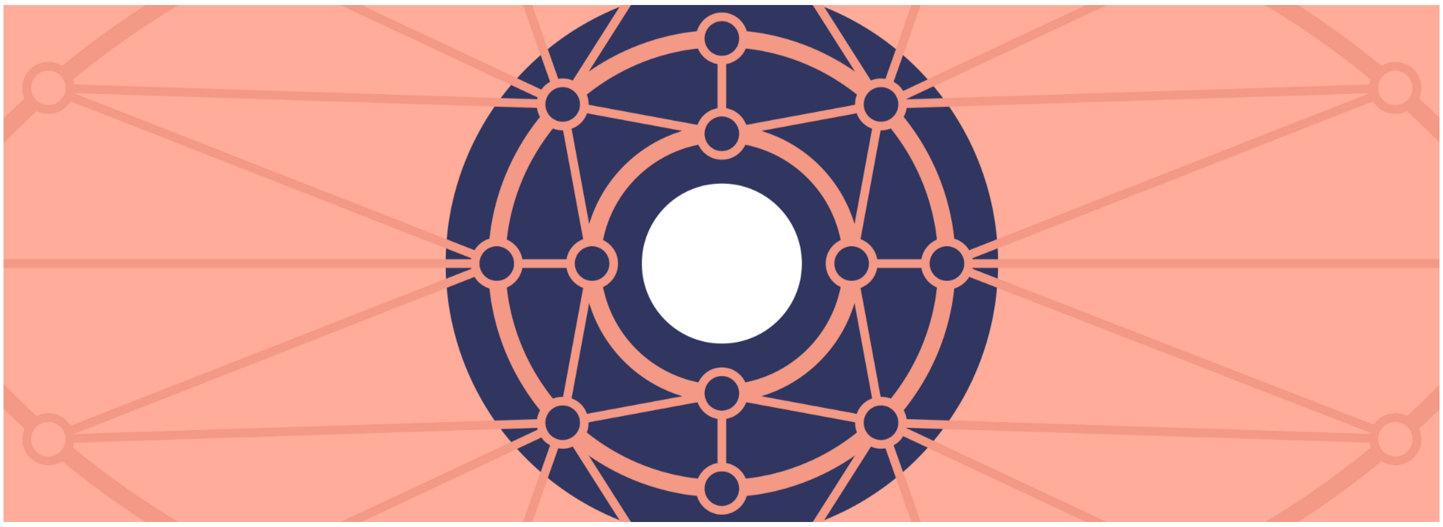
Stanislav Vishnevskiy Follow
CTO @ discordapp.com
Jan 14, 2017 · 11 min read

How Discord Stores Billions of Messages



Discord continues to grow faster than we expected and so does our user-generated content. With more users comes more chat messages. In July, we announced 40 million messages a day, in December we announced 100 million, and as of this blog post we are well past 120 million. We decided early on to store all chat history forever so users can come back at any time and have their data available on any device. This is a lot of data that is ever increasing in velocity, size, and must remain available. *How do we do it? Cassandra!*

What we were doing

The original version of Discord was built in just under two months in early 2015. Arguably, one of the best databases for iterating quickly is MongoDB. Everything on Discord was stored in a single MongoDB replica set and this was intentional, but we also planned everything for easy migration to a new database (we knew we were not going to use MongoDB sharding because it is complicated to use and not known for stability). This is actually part of our company culture: build quickly to prove out a product feature, but always with a path to a more robust solution.

The messages were stored in a MongoDB collection with a single compound index on `channel_id` and `created_at`. Around November 2015, we reached 100 million stored messages and at this time we started to see the expected issues appearing: the data and the index could no longer fit in RAM and latencies started to become unpredictable. It was time to migrate to a database more suited to the task.

Choosing the Right Database

Before choosing a new database, we had to understand our read/write patterns and why we were having problems with our current solution.

- It quickly became clear that our reads were extremely random and our read/write ratio was about 50/50.
- Voice chat heavy Discord servers send almost no messages. This means they send a message or two every few days. In a year, this kind of server is unlikely to reach 1,000 messages. The problem is that even though this is a small amount of messages it makes it harder to serve this data to the users. Just returning 50 messages to a user can result in many random seeks on disk causing disk cache evictions.
- Private text chat heavy Discord servers send a decent number of messages, easily reaching between 100 thousand to 1 million messages a year. The data they are requesting is usually very recent only. The problem is since these servers usually have under 100 members the rate at which this data is requested is low and unlikely to be in disk cache.
- Large public Discord servers send a lot of messages. They have thousands of members sending thousands of messages a day and easily rack up millions of messages a year. They almost always are requesting messages sent in the last hour and they are requesting them often. Because of that the data is usually in the disk cache.
- We knew that in the coming year we would add even more ways for users to issue random reads: the ability to view your mentions for the last 30 days then jump to that point in history, viewing plus jumping to pinned messages, and full-text search. *All of this spells more random reads!!*

Next we defined our requirements:

- **Linear scalability—**We do not want to reconsider the solution later or manually re-shard the data.
- **Automatic failover—**We love sleeping at night and build Discord to self heal as much as possible.
- **Low maintenance—**It should just work once we set it up. We should only have to add more nodes as data grows.
- **Proven to work—**We love trying out new technology, but not too new.
- **Predictable performance—**We have alerts go off when our API's response time 95th percentile goes above 80ms. We also do not want to have to cache messages in Redis or Memcached.
- **Not a blob store—**Writing thousands of messages per second would not work great if we had to constantly deserialize blobs and append to them.
- **Open source—**We believe in controlling our own destiny and don't want to depend on a third party company.

Cassandra was the only database that fulfilled all of our requirements. We can just add nodes to scale it and it can tolerate a loss of nodes without any impact on the application. Large companies such as Netflix and Apple have thousands of Cassandra nodes. Related data is stored contiguously on disk providing minimum seeks and easy distribution around the cluster. It's backed by DataStax, but still open source and community driven.

Having made the choice, we needed to prove that it would actually work.

Data Modeling

The best way to describe Cassandra to a newcomer is that it is a KKV store. The two Ks comprise the primary key. The first K is the partition key and is used to determine which node the data lives on and where it is found on disk. The partition contains multiple rows within it and a row within a partition is identified by the second K, which is the clustering key. The clustering key acts as both a primary key within the partition and how the rows are sorted. You can think of a partition as an ordered dictionary. These properties combined allow for very powerful data modeling.

Remember that messages were indexed in MongoDB using `channel_id` and `created_at`? `channel_id` became the partition key since all queries operate on a channel, but `created_at` didn't make a great clustering key because two messages can have the same creation time. Luckily every ID on Discord is actually a Snowflake (chronologically sortable), so we were able to use them instead. The primary key became `(channel_id, message_id)`, where the `message_id` is a Snowflake. This meant that when loading a channel we could tell Cassandra exactly where to range scan for messages.

Here is a simplified schema for our messages table (this omits about 10 columns).

```
CREATE TABLE messages (
  channel_id bigint,
  message_id bigint,
  author_id bigint,
  content text,
  PRIMARY KEY (channel_id, message_id)
```

While Cassandra has schemas not unlike a relational database, they are cheap to alter and do not impose any temporary performance impact. We get the best of a blob store and a relational store.

When we started importing existing messages into Cassandra we immediately began to see warnings in the logs telling us that partitions were found over 100MB in size. *What gives?! Cassandra advertises that it can support 2GB partitions!* Apparently, just because it can be done, it doesn't mean it should. Large partitions put a lot of GC pressure on Cassandra during compaction, cluster expansion, and more. Having a large partition also means the data in it cannot be distributed around the cluster. It became clear we had to somehow bound the size of partitions because a single Discord channel can exist for years and perpetually grow in size.

We decided to bucket our messages by time. We looked at the largest channels on Discord and determined if we stored about 10 days of messages within a bucket that we could comfortably stay under 100MB. Buckets had to be derivable from the `message_id` or a timestamp.

```
DISCORD_EPOCH = 1420070400000
BUCKET_SIZE = 1000 * 60 * 60 * 24 * 10


def make_bucket(snowflake):
    if snowflake is None:
        timestamp = int(time.time() * 1000) - DISCORD_
    else:
        # When a Snowflake is created it contains the
        # seconds since the DISCORD_EPOCH.
        timestamp = snowflake_id >> 22
```

Cassandra partition keys can be compounded, so our new primary key became `((channel_id, bucket), message_id)`.

```
CREATE TABLE messages (
   channel_id bigint,
   bucket int,
   message_id bigint,
   author_id bigint,
   content text,
```

To query for recent messages in the channel we generate a bucket range from current time to `channel_id` (it is also a Snowflake and has to be older than the first message). We then sequentially query partitions until enough messages are collected. The downside of this method is that rarely active Discords will have to query multiple buckets to collect enough messages over time. In practice this has proved to be fine because for active Discords enough messages are usually found in the first partition and they are the majority.

Importing messages into Cassandra went without a hitch and we were ready to try in production.

Dark Launch

Introducing a new system into production is always scary so it's a good idea to try to test it without impacting users. We setup our code to double read/write to MongoDB and Cassandra.

Immediately after launching we started getting errors in our bug tracker telling us that `author_id` was null. *How can it be null? It is a required field!*

Eventual Consistency

Cassandra is an AP database which means it trades strong consistency for availability which is something we wanted. It is an anti-pattern to read-before-write (reads are more expensive) in Cassandra and therefore everything that Cassandra does is essentially an upsert even if you provide only certain columns. You can also write to any node and it will resolve conflicts automatically using "last write wins" semantics on a per column basis. *So how did this bite us?*

Example of edit/delete race condition

In the scenario that a user edits a message at the same time as another user deletes the same message, we ended up with a row that was missing all the data except the primary key and the text since all Cassandra writes are upserts. There were two possible solutions for handling this problem:

1. Write the whole message back when editing the message. This had the possibility of resurrecting messages that were deleted and adding more chances for conflict for concurrent writes to other columns.

2. Figuring out that the message is corrupt and deleting it from database.

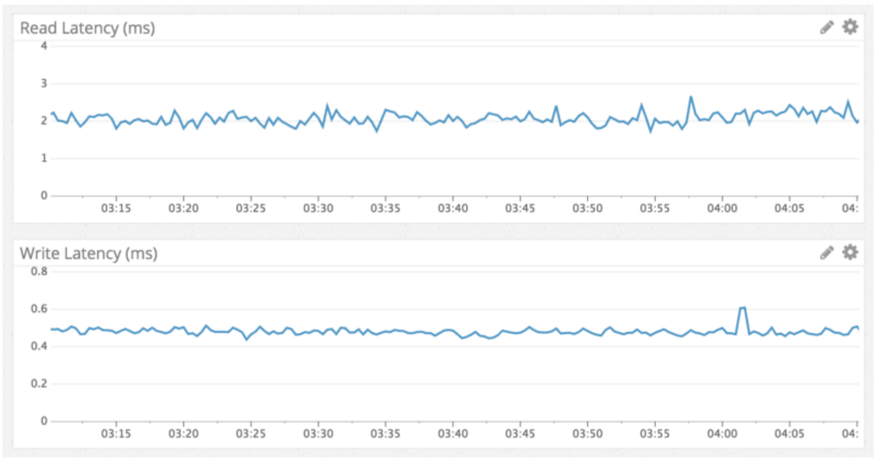
We went with the second option, which we did by choosing a column that was required (in this case `author_id`) and deleting the message if it was null.

While solving this problem, we noticed we were being very inefficient with our writes. Since Cassandra is eventually consistent it cannot just delete data immediately. It has to replicate deletes to other nodes and do it even if other nodes are temporarily unavailable. Cassandra does this by treating deletes as a form of write called a “tombstone.” On read, it just skips over tombstones it comes across. Tombstones live for a configurable amount of time (10 days by default) and are permanently deleted during compaction when that time expires.

Deleting a column and writing null to a column are the exact same thing. They both generate a tombstone. Since all writes in Cassandra are upserts, that means you are generating a tombstone even when writing null for the first time. In practice, our entire message schema contains 16 columns, but the average message only has 4 values set. We were writing 12 tombstones into Cassandra most of the time for no reason. The solution to this was simple: only write non-null values to Cassandra.

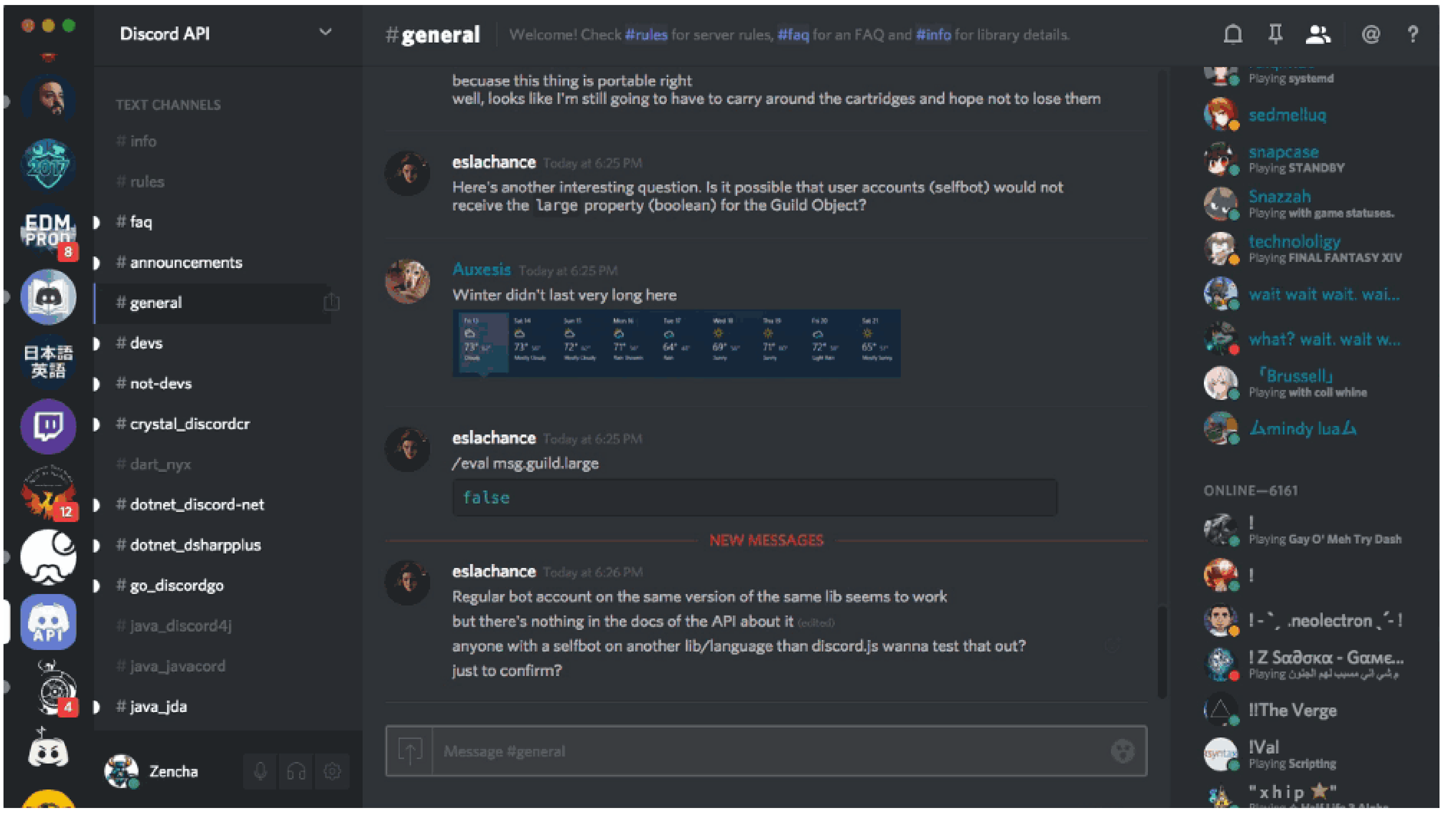
Performance

Cassandra is known to have faster writes than reads and we observed exactly that. Writes were sub-millisecond and reads were under 5 milliseconds. We observed this regardless of what data was being accessed, and performance stayed consistent during a week of testing. *Nothing was surprising, we got exactly what we expected.*



Read/Write Latency via Datadog

In line with fast, consistent read performance, here's an example of a jump to a message from over a year ago in a channel with millions of messages:



#general Welcome! Check #rules for server rules, #faq for an FAQ and #info for library details.

becuase this thing is portable right
well, looks like I'm still going to have to carry around the cartridges and hope not to lose them

eslachance Today at 6:25 PM
Here's another interesting question. Is it possible that user accounts (selfbot) would not receive the `large` property (boolean) for the Guild Object?

Auxesis Today at 6:25 PM
Winter didn't last very long here

eslachance Today at 6:25 PM
/eval msg.guild.large

```
false
```

NEW MESSAGES

eslachance Today at 6:26 PM
Regular bot account on the same version of the same lib seems to work
but there's nothing in the docs of the API about it (edited)
anyone with a selfbot on another lib/language than discord.js wanna test that out?
just to confirm?

Message #general

Jumping back one full year of chat

The Big Surprise

Everything went smoothly, so we rolled it out as our primary database and phased out MongoDB within a week . It continued to work flawlessly…for about 6 months until that one day where Cassandra became unresponsive.

We noticed Cassandra was running 10 second *"stop-the-world"* GC constantly but we had no idea why. We started digging and found a Discord channel that was taking 20 seconds to load. The Puzzles & Dragons Subreddit public Discord server was the culprit. Since it was public we joined it to take a look. To our surprise, the channel had only 1 message in it. It was at that moment that it became obvious they deleted millions of messages using our API, leaving only 1 message in the channel.

If you have been paying attention you might remember how Cassandra handles deletes using tombstones (mentioned in **Eventual Consistency**). When a user loaded this channel, even though there was only 1 message, Cassandra had to effectively scan millions of message tombstones (generating garbage faster than the JVM could collect it).

We solved this by doing the following:

- We lowered the lifespan of tombstones from 10 days down to 2 days because we run Cassandra repairs (an anti-entropy process) every night on our message cluster.
- We changed our query code to track empty buckets and avoid them in the future for a channel. This meant that if a user caused this query again then at worst Cassandra would be scanning only in the most recent bucket.

The Future

We are currently running a 12 node cluster with a replica factor of 3 and will just continue to add new Cassandra nodes as needed. We believe this will continue to work for a long time but as Discord continues to grow there is a distant future where we are storing billions of messages per day. Netflix and Apple run clusters of hundreds of nodes so we know we can punt thinking too much about this for a while. However we like to have some ideas in our pocket for the future.

Near term

- Upgrade our message cluster from Cassandra 2 to Cassandra 3. Cassandra 3 has a new storage format that can reduce storage size by more than 50%.

- Newer versions of Cassandra are better at handling more data on a single node. We currently store nearly 1TB of compressed data on each node. We believe we can safely reduce the number of nodes in the cluster by bumping this to 2TB.

Long term

- Explore using Scylla, a Cassandra compatible database written in C++. During normal operations our Cassandra nodes are actually not using too much CPU, however at non peak hours when we run repairs (an anti-entropy process) they become fairly CPU bound and the duration increases with the amount of data written since the last repair. Scylla advertises significantly lower repair times.
- Build a system to archive unused channels to flat files on Google Cloud Storage and load them back on-demand. We want to avoid doing this one and don't think we will have to do it.

Conclusion

It has now been just over a year since we made the switch and, despite *"the big surprise,"* it has been smooth sailing. We went from over 100 million total messages to more than 120 million messages a day, with performance and stability staying consistent.

Due to the success of this project we have since moved the rest of our live production data to Cassandra and that has also been a success.

In a follow-up to this post we will explore how we make billions of messages searchable.

We don't have dedicated DevOps engineers yet (only 4 backend engineers), so having a system we don't have to worry about has been great. *We are hiring, so come join us if this type of stuff tickles your fancy.*