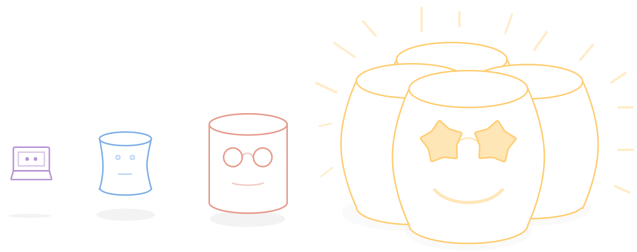


Growing up with MySQL

How we scaled our primary datastore by over 20x in 3 weeks

wpengine | December 7, 2016

Building an email system requires efficiently managing a huge amount of information essentially from day one. Our sync system has more data for a single user than most startup's database of *everything*. As of November 2016, nearly 500k accounts have been synced by the Nylas Cloud system, and our datastore grows by gigabytes each day.

This post is about how we scaled our primary datastore by over 20x after launching Nylas N1, our flagship desktop mail app. We achieved this by deploying a recently-developed tool called ProxySQL and horizontally sharding our primary keyspace. We also moved off Amazon RDS to self-managed MySQL on EC2. Combined with other techniques, this has allowed us to achieve near-zero downtime even while performing large database operations, such as schema migration, replica promotion, and automatic failover.

Once upon a time…

It was the summer of 2015. Sun was shining through the San Francisco fog. Taylor Swift and Kendrick Lamar were blasting on the radio. And the entire Nylas Cloud sync system was supported by just two sync machines, and one MySQL database managed by Amazon RDS. It was a simple design that we found easy to reason about and manage without a dedicated DBA. The Nylas ops team slept (mostly) soundly, and focused engineering work on improving our sync logic and telemetry.

We'd managed to get a surprising amount of performance from this configuration. Our team knew the dangers and distractions of premature optimization, and figured **the "easiest" way to shard was to avoid it for as long as possible**. So we maxed-out the disks, upgraded our instances, and compressed data to buy more time.

Then came the users

But this all changed on Monday October 5th, 2015. That morning we released Nylas N1, an open source email app with a beautiful UI and modern plugin-based architecture. Under the hood, N1 is powered by our cloud sync engine, which is also open source. We released the code on GitHub with a pre-built Mac binary that used our hosted sync engine. We didn't expect what came next…

Like all products worthy of argument and opinions, N1 was posted on Hacker News. That meant within minutes, we had literally tens of thousands of excited new users. And with those new users came lots and lots of data!

In less than an hour, our MySQL database performance plummeted and API latency went through the roof. We experienced a significant outage, and decided to stop the signup flood via a backup invite system. We didn't yet know the full extent of our sync system issues, but one thing was crystal clear: we needed to scale. A lot.

Fighting fires

In the days following our launch of N1, we discovered dozens of individual and unique bottlenecks in our application code. We came face-to-face with the upper limits of Redis and MySQL's insertion volume. And we even learned some interesting things about AWS network topologies.

But our largest bottleneck was our database. We worked day and night to fix things like leaking MySQL connections, subtly bad queries that wrecked performance, and ultra-long-running errant transactions. But we knew the long-term solution would require horizontally scaling our data layer to support the tens of thousands of new users. It was time to shard.

Why MySQL?

In 2013 when Nylas was founded, we picked MySQL not because of its features, but because it's widely used at huge scale. Technologies like Cassandra, MongoDB, and Postgres were shiny and evolving rapidly, but MySQL was the one datastore we were confident could handle petabyte-scale. We'd seen it at companies like Facebook, Dropbox, Pinterest, and Uber. Decisions like this are actually extremely important to get right early on. Long-term, we knew using MySQL meant we could leverage the knowledge of many expert database engineers as we scaled. (Hint: and we did!)

Out the the box, MySQL is a bit raw, and our initial needs were simple. Because of this, we decided to use Amazon RDS, a hosted MySQL appliance. While RDS doesn't give you root access on the MySQL host, it does automate a lot of the pieces of running MySQL robustly, like backups and point-release upgrades. Those backups saved our skin in the first few months, and let us move super fast.

Time to scale

We'd already compressed long text columns for a 40% storage savings and upgraded our disk to the 6 TB maximum. But InnoDB has a size limitation of 2 TB per table and there was no way our tens of thousands of new users would fit on this instance.

(Aside: We briefly considered Amazon Aurora, which is a MySQL-compatible datastore with an autoscaling storage layer that can grow up to 64TB. But even with Aurora, we'd have had to partition our data across multiple tables to get around the InnoDB limit, and eventually across multiple databases once beyond 64 TB. Plus, long-term we didn't want to be this dependent on Amazon technology.)

We scaled by horizontally partitioning our data using a very simple scheme, and to create additional MySQL database clusters running Oracle's stock MySQL on regular vanilla EC2 instances. We aimed for simplicity in this design, and therefore were able to implement these changes within just 3 weeks.

Moving off RDS

When scaling, we decided to move off RDS. This decision was based primarily on control and frustrating past experiences with the design quirks for RDS. It seems that RDS is not designed to be a solution that allows for zero downtime database operations.

RDS offers very limited support for different replication setups. It has a "multi-AZ" synchronous replication option, but enabling this creates a significant performance hit. We know this because we tried it once and this caused a 3 hour outage. Our only option was to call Amazon support and wait on hold while RDS engineers investigated.

We also looked at RDS's ability to create asynchronous read replicas for a given master, but found that a replica can't be promoted to a master without several minutes of downtime. This wasn't acceptable to us. Plus, RDS doesn't support chained replication (i.e. master → slave → slave), which we planned to use for splitting shards across database instances. And on and on.

By contrast, we had found EC2 to be incredibly reliable. We knew individual instances can still fail, but we could design a database architecture to be resilient to that failure case. We decided it was time to take off the RDS training wheels and run MySQL ourselves.

Running MySQL on EC2

Moving to MySQL on EC2 wasn't entirely straightforward. We had to write our own scripts for managing many of the tasks that RDS formerly automated for us—like provisioning new servers,

creating and promoting slaves, failing over when a master stops working properly, and creating and restoring backups.

We wrote these from scratch, but these days there are some awesome open source tools to help get you started.

One of our MySQL shard clusters typically consists of a master node and a slave node. The slave is kept up to date via the MySQL replication log, and is used for backups and non-critical reads (i.e. fine if it returns stale data).

During maintenance, such as migrating nodes to a different instance type, a cluster might have more than one slave. The extras not in production yet are marked with a “debug” EC2 tag, which excludes them from monitoring, backups and other services for active nodes.

Failover for when things go wrong

Any high-availability database cluster must handle the situations where a master node ceases to work properly. This can happen when a database process crashes or hangs, the host becomes unreachable, an underlying hardware failure, or something else. The goal is to have a recovery, either manual or automatic, where a slave node is “promoted” to replace the master node, and begins receiving both reads and primary writes.

To achieve this, we first built a script that engineers could use to manually promote a slave node. Once we were confident that this process worked every time without potential for data loss, we integrated the script with the standard `mysqlfailover` tool, with some additional tooling for detecting which nodes are masters or slaves based on our EC2 tags. If a master node fails, mysqlfailover detects this through a health check mechanism and promotes a slave to be the new master. Our configuration of mysqlfailover also updates the EC2 tags to reflect the new promoted configuration.

This failover is automatic, but it still results in a notification to our engineering team. Later on, an engineer on our team will manually recover the failed node into a new slave for the cluster. If this cannot be done, such as in the case of corrupted data, the old master is decommissioned and a new slave is created from the most recent backup.

This has resulted in a large MySQL fleet that is very resilient to failure, and also a design that is easy to manage and reason about. It’s also given us the ability to use manually triggered failovers for routine operational tasks like seamlessly rolling out tricky migrations by running them on a slave instead of the master and then failing over to the slave.

How we sharded the application

When moving from a single database to multiple databases on multiple machines, you need to design a new mechanism for mapping object IDs to their respective machine. This is also important if your application references database rows by primary key in a secondary store, like Redis.

We solved this via a simple design: a table's primary key value is a 64-bit integer where the upper bits contain the shard ID, and the lower bits are a shard-local autoincrementing counter.

```
|<--16bits-->|<--------48bits------->| +------------+----------------------+ | shard
id | autoincrement | +------------+----------------------+
```

This makes it incredibly easy to compute the shard ID from a primary key. It's just a bitshift! A query for any object can be routed to the correct shard by primary key alone, which keeps things simple from the application's perspective.

For example, our authentication layer is able to convert hashed access tokens into account IDs for API access. We were able to move the service to a sharded deployment with essentially no changes to this critical component.

Because the Nylas sync engine is open source, you can also go read the code to see the exact details of how we implement sharded session management using SQLAlchemy.

Creating new shards

Each logical shard is a uniquely numbered MySQL schema, named e.g. mailsync_22. All shards have exactly the same table structure and multiple shards generally live on one database instance. (This allows database instances to be split in the case of overload—imagine cell mitosis, but for databases.)

When creating a new shard, it's easy to compute the high-order bits via

```
N = (shard_id<<48) + 1
```

The auto increment value is set like this:

```
ALTER TABLE mailsync_. AUTO_INCREMENT=N
```

Rows inserted into the table will then have primary keys N, N+1, etc. Prior to sharding, our database

tables had autoincrementing keys starting at 1. So they just became “shard 0” and required no other migration.

If you use this approach, beware of MySQL bug #199, which has been open for 13 years and counting! Your application code must safeguard against incorrect autoincrement values. Here’s how we do it at Nylas.

Minimizing downtime during failover

After sharding our application, we noticed another problematic issue. Our mysqlfailover configuration used an Amazon VPC secondary IP address as a virtual IP to handle the routing of application traffic to the master node. But detecting a master node failure and switching traffic using this virtual IP took about 10 minutes. All requests during this time would fail, resulting in a serious outage.

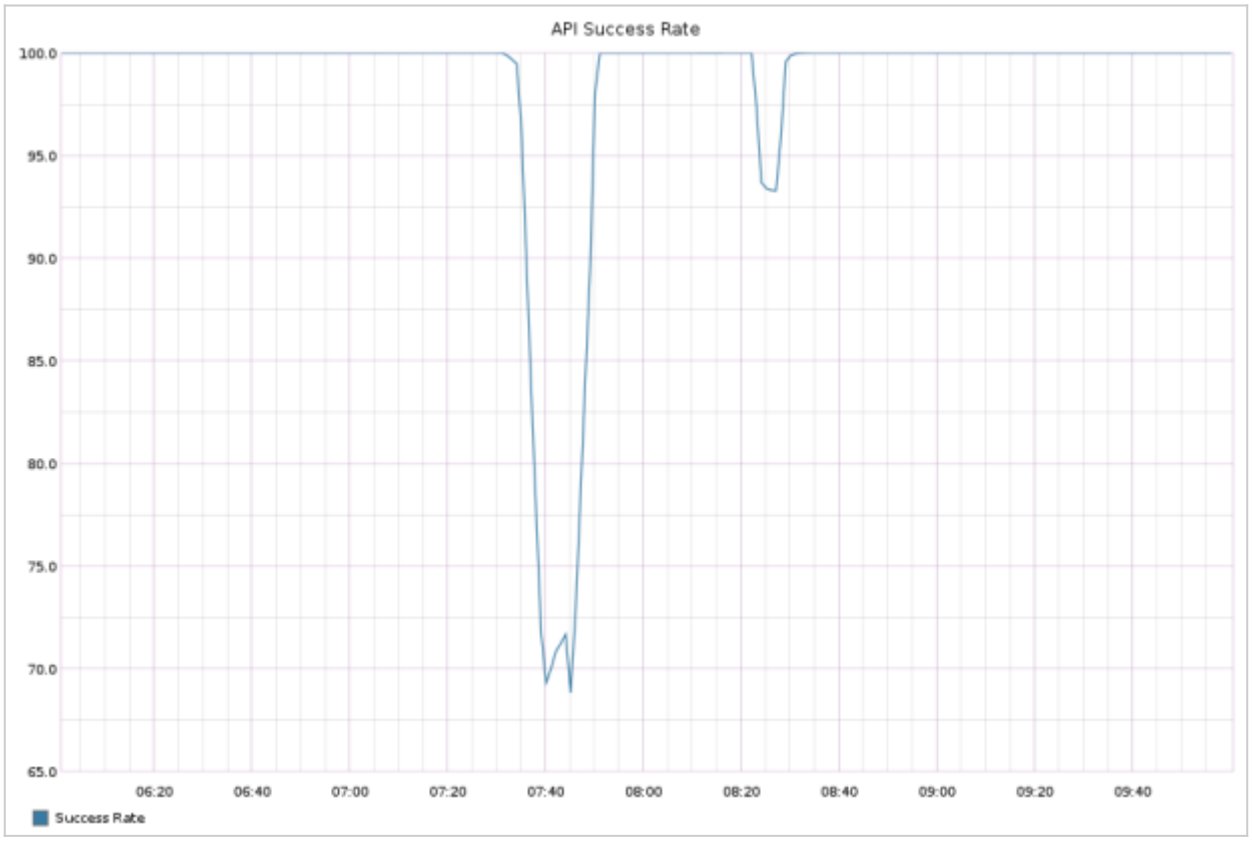
What we needed was seamless failover. Something similar to how HAProxy can hold requests as application servers reboot, but instead for MySQL.

Some folks have actually implemented a working solution using HAProxy accompanied by a highly available key-value store (like ZooKeeper) to track cluster membership. Others have used a MySQL clustering tool like Orchestrator or Galera.

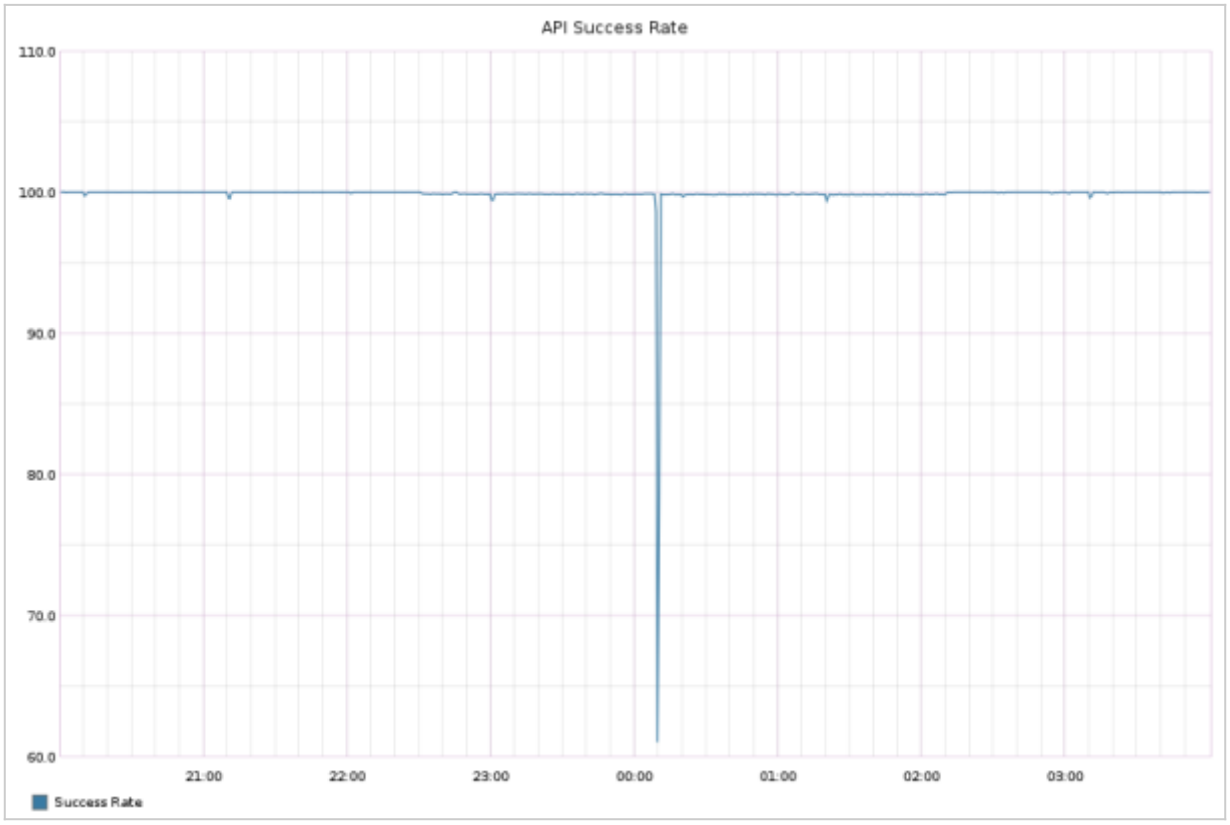
But like the other components in our system, we wanted to use the simplest possible tool that would get the job done and be easy to reason about. Ideally without new services to maintain.

ProxySQL: a proxy for MySQL

We achieved this using ProxySQL, a new open source high performance proxy created by René Cannaò. You can think of ProxySQL as essentially HAProxy but for SQL queries. Using ProxySQL, our failover downtime went from 10 minutes to 10 seconds.



Failover without ProxySQL (10 minute outage)



Failover with ProxySQL (10 second outage)

We've also found many other significant benefits to using ProxySQL:

Dynamic routing

Our configuration of ProxySQL will detect read-only queries and route them to slave replicas, which helps optimize performance across a cluster.

Connection multiplexing

If you're hitting the MySQL connection limit, you can use ProxySQL's connection multiplexing to dramatically decreases the number of outbound connections to the actual database machine. Using a proxy instead of direct database connections will also simplify your application's connection pooling.

Rogue queries

As a developer, it's pretty easy to introduce regressions or errant queries that create long-running transactions which may hog resources. ProxySQL can be configured to auto-terminate queries or transactions above a threshold, preventing accidental performance degradation.

Query statistics

ProxySQL allows you to compare queries across application nodes, which is useful to understand how load is distributed across shards and on a per-host basis. Your application is only as fast as its slowest query, and ProxySQL can help you hunt that down.

In production, we run ProxySQL locally on every service that connects to sync databases—including internal dashboard tools and scripts. Running the proxy locally instead of centrally provides built-in redundancy, and, since we drive a lot of network traffic to our databases, prevents the proxy from becoming a network bottleneck. The tool has enabled us to dynamically move shards across database instances with as little as 12 seconds of outage per cluster.

ProxySQL has a number of powerful features that we haven't started taking advantage of, such as traffic mirroring and query caching, which we hope to write about in future posts. We're also working on suggested best practices for managing its configuration with Ansible (our tool of choice). Since ProxySQL keeps track of how to map shard names to hosts and other rules for routing queries in its configuration, it's important to have an easy mechanism for keeping its configuration up-to-date across nodes. ProxySQL is young and not yet well-documented, but is commercially supported by its authors.

Migrating existing data

In March 2016, we migrated shard 0 (RDS) to our "in-house" MySQL instances on EC2 via replication.

Migrating this legacy shard to EC2 allowed us to upgrade primary keys on our largest table (by rows) from 32 to 64 bits—an operation we could not have done using RDS. The table was on track to exhaust its primary keyspace – over 4 billion rows – within a couple months.

Future work

The Nylas cloud sync infrastructure continues to scale and grow quickly, and with this work complete on our database layer, we've shifted focus to scaling our application layer. Current projects include splitting the sync logic into microservices, and introducing new processing topologies with a central data backbone like Kafka. We are also investigating a potential shift to denormalized schemas for our API storage layer to make our application development more flexible.

Thanks for reading! Have questions or comments? Let us know on Twitter!

Scaling and sharding the Nylas Cloud data layer in fall 2015 was an collaborative team project, including substantial work by Kavya Joshi, Eben Freeman, and René Cannaò.

Extra thanks to Jeff Rothschild, Peter Zaitsev, Vadim Tkachenko, & Joe Gross for advice during this period, and also to Marty Weiner of Pinterest for their scaling MySQL blog post.