shopify.engineering

Shard Balancing: Moving Shops Confidently with Zero-Downtime at Terabyte-scale

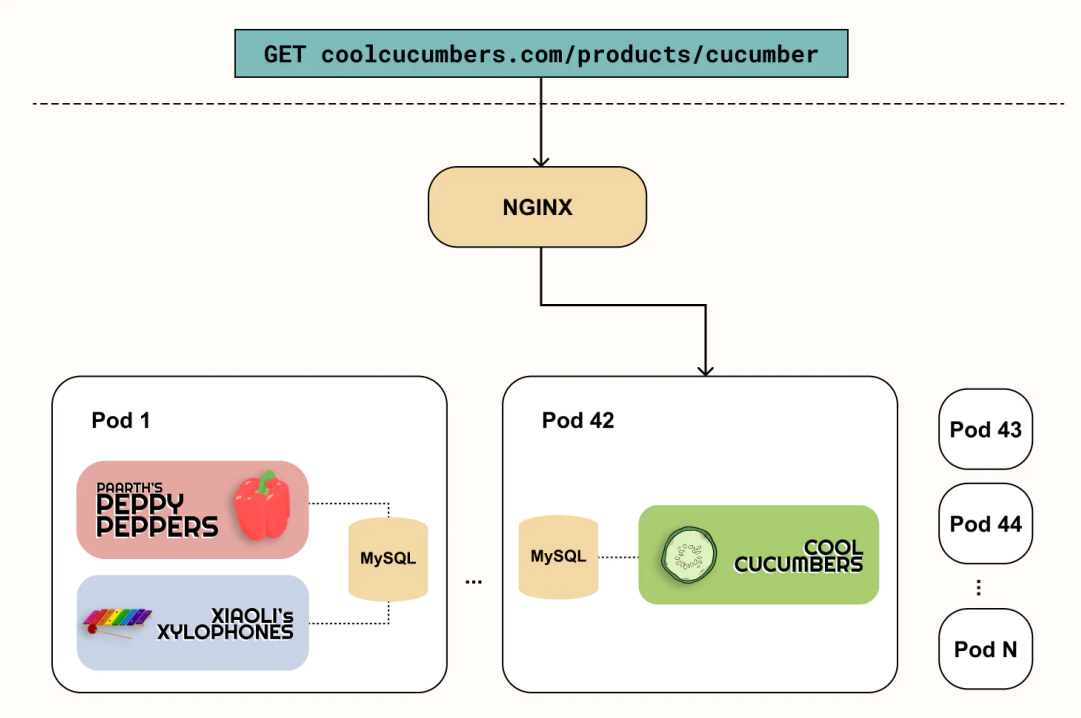
16-20 minutes

Shopify's infrastructure supports millions of merchants during their entrepreneurship journey. A key component of the current infrastructure is the underlying fleet of MySQL database shards that together persist every shop's critical data. As traffic patterns change and new merchants onboard onto the platform, it's possible that resource intensive shops end up living in the same shards. Certain database shards become unbalanced in their database utilization, shop traffic, and load. It's important to ensure the shards remain well-balanced to mitigate risk of database failure, improve productivity of the wider infrastructure, and ultimately guarantee buyers can always access their favorite shops. This post explains how we're able to balance our MySQL shards by migrating shops across shards—entirely online and with virtually zero consumer-facing downtime.

A Brief Overview of Shopify's Current Architecture

To fully understand shard balancing, it helps to briefly review Shopify's architecture. Shopify's application runtime is currently podded: the infrastructure is composed of many pods (not to be confused with Kubernetes Pods). A pod is an isolated instance of Shopify consisting of an individual MySQL database shard, along with other datastores like Redis and Memcached. Every pod

houses a unique subset of shops on the platform. Web requests for shops are processed by a load balancer that consults a routing table and forwards the request to the correct pod based on the shop.

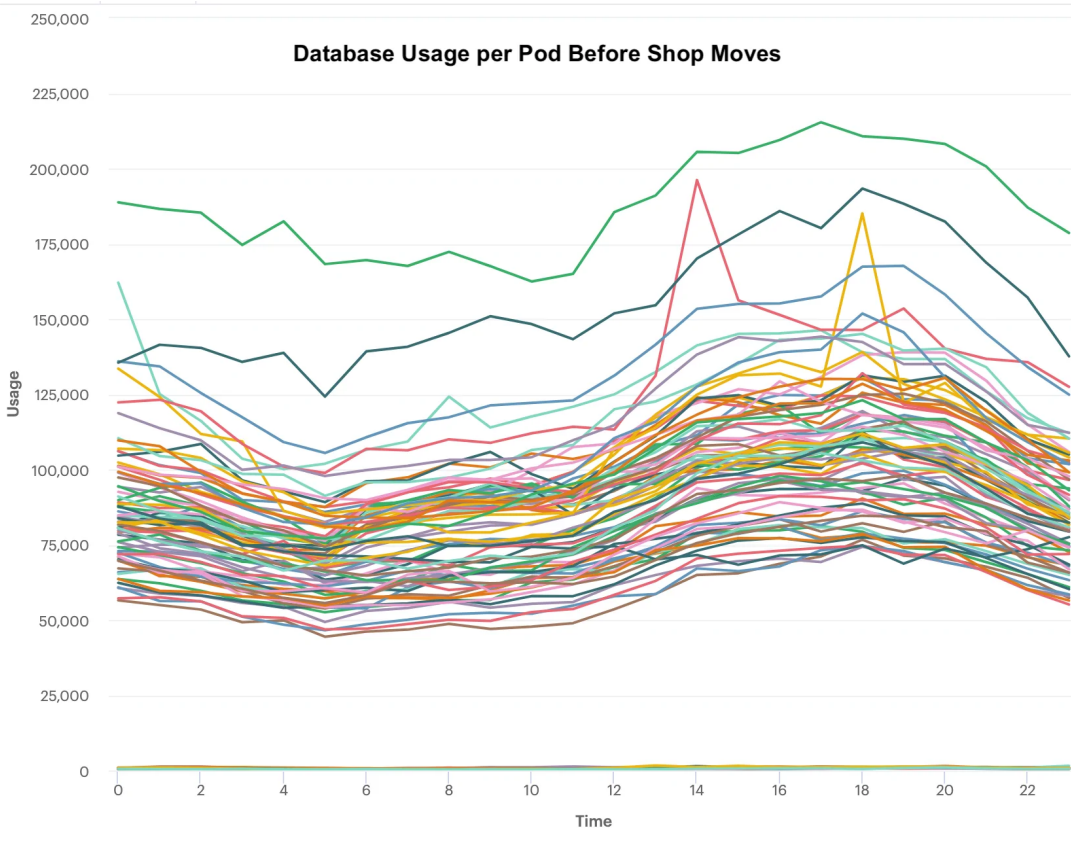


Shopify's application runtime is split into multiple pods. Each pod contains its own MySQL shard. Here, a sample request to `coolcucumbers.com` is processed by a load balancer and forwarded to Pod 42. Pod 42 contains all the data required to serve traffic for Cool Cucumbers.

The podded application runtime is supported by a sharded database topology: each pod consists of its own shard. Shopify's data model lends itself nicely to this topology as a `shop` is the identifying entity for most data models. We can attach a `shop_id` to all shop-owned tables and have it serve as a sharding key. Moving a shop from one shard to another involves selecting all records from all tables that have the desired `shop_id` and copying them to another MySQL shard. For this post, it's helpful to think of each pod as a MySQL shard.

A Shard Rebalancing Strategy

When new merchants sign up and onboard on the platform, they're assigned to an arbitrary shard. As time progresses, those merchants grow to various sizes. It may be the case that some resource intensive shops end up living in the same shards, resulting in higher database usage for some shards and lower database usage for others. These inconsistencies in database usage weaken the infrastructure for two clear reasons. First, the high traffic shards are at a larger risk of failure due to possible over-utilization. Second, the shards with low database usage are not being used productively.



The above graph highlights the variation in database usage per shard that developed over time as merchants onboarded and grew in certain shards. Each line represents the database usage for a unique shard in a given day. The least used shard and most used shard vary by almost four times and the deviation across all shards is too large.

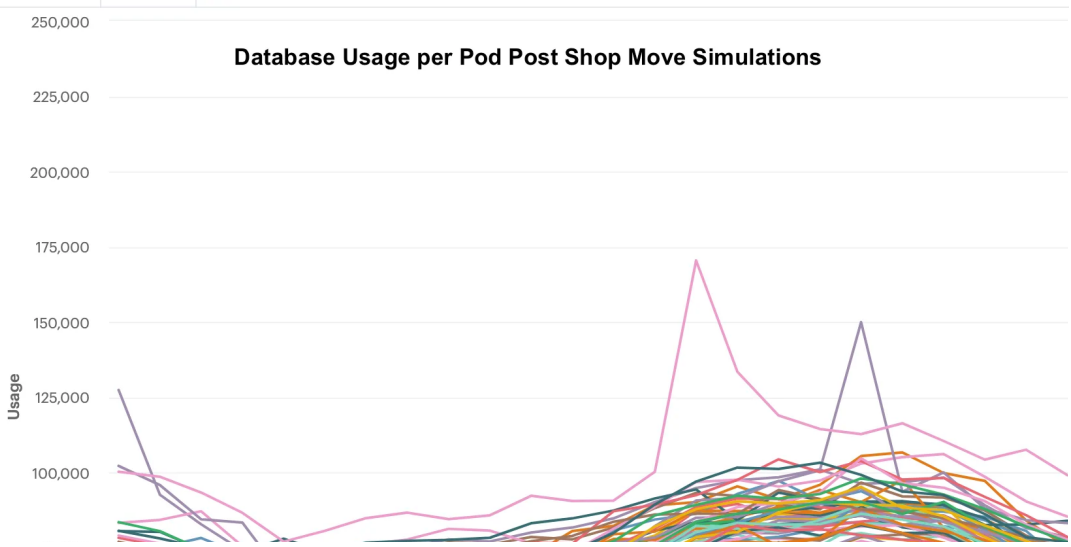
To smooth the load across the shards, we need to rebalance them. We define a balanced infrastructure as one where all pods are healthy and their shards are utilized productively. To achieve this balanced infrastructure, we require a strategy to enable the continued redistribution of shops across shards.

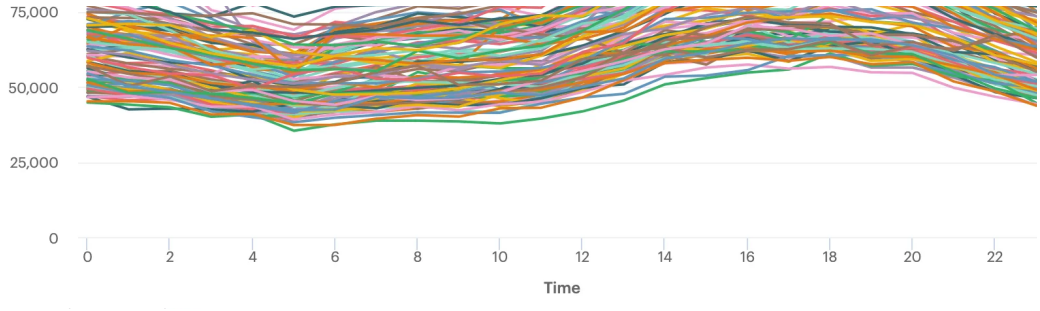
When devising this strategy for shard balancing it became clear that there are two problems to solve:

1. Which shops should live in which shards?
2. How are shops moved from one shard to another with as little downtime as possible?

Which Shops Should Live In Which Shards?

Distribution of shops on a shard based on the number of shops isn't a great strategy because the size of the data in each shop varies. One strategy used previously was to analyze historical database utilization and traffic data for shards and create a classification based on their usage patterns (ie. `high_traffic`, `low_traffic`, etc.). Applicable shops were migrated between these shard cohorts using a scheme like moving every Nth shop from `high_traffic` shards to `low_traffic` shards. The proposed moves were simulated and their forecasted effects were used to validate the hypothesis.





The graph above showcases database usage across shards after simulating moving a list of shops between shards. The result of these moves would be that the deviation amongst database usage is significantly lower, and the number of under- and over-utilized shards is smaller. The range between the least used shard and most used shard is now closer to two times. This graph suggests the infrastructure would be better tuned to serve traffic.

While this strategy was effective, it isn't the only one. Placement strategies can be arbitrarily complex and prioritize different metrics (for example, shop sizes, GMV, time to move, flash sales, etc). Usually multiple hypotheses are presented and tested against recent data. Once an ideal distribution of shops is determined, a list of shop moves is generated that achieves the desired state of our system.

How Are Shops Moved?

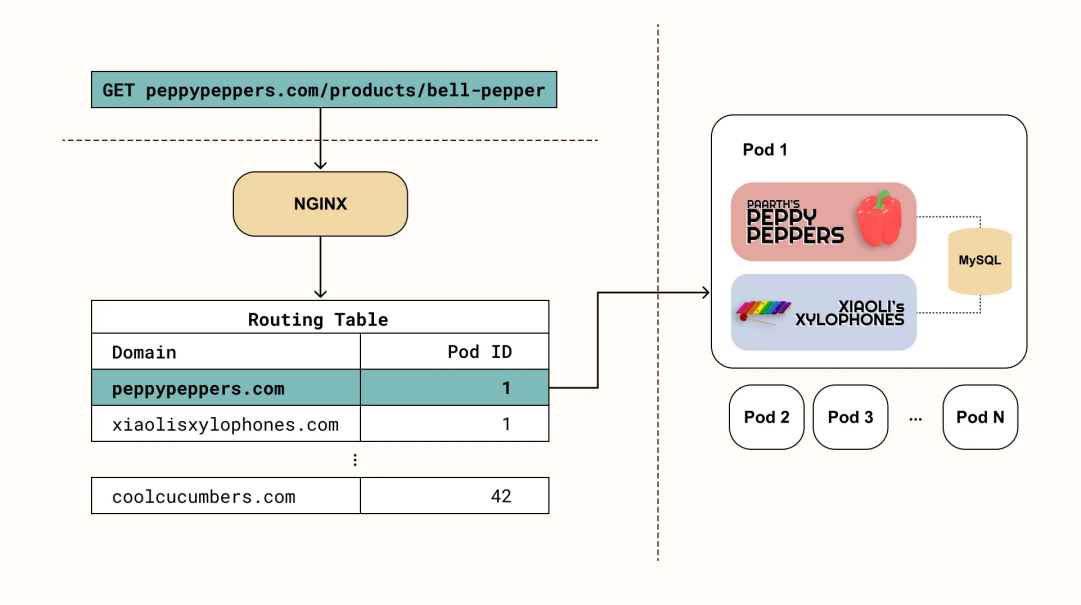
With an understanding of which shops need to live in which shards, the process of moving shops can begin. As we soon layout, moving a shop from its source shard to the desired target shard can be an involved process. It's of particular interest due to a few critical constraints that are imposed on the strategy we use.

- **Availability:** A shop move must be entirely online. This ensures the merchant and the wider platform don't incur visible downtime. As data is moved from the source database shard to the target database shard, the merchant's storefront must be available for interaction.

- **Data Integrity:** There's no data loss or corruption during the move. The process must ensure that all data existing at the start of the move is copied to the target shard. Also, it must ensure that all writes against the source database since the beginning of the move are copied.
- **Throughput**: The process to move data from shard to shard must be timely and allow for reasonable throughput. Shops sizes range, so moving many of these shops at once shouldn't put undue strain on the infrastructure.

  To help describe shop moves, we define a fictional scenario: *Paarth's Peppy Peppers* and *Xiaoli's Xylophones* are two high-traffic merchants on our platform. Both shops currently live on Pod 1. Our Data Science & Engineering team concludes that housing both of these merchants on the same pod isn't optimal. Database utilization is extremely high and burst traffic from both of these merchants is synchronized. It seems *Paarth* and *Xiaoli* have flash sales on the same day! The team suggests moving *Paarth's Peppy Peppers* to Pod 2. We'll explore the end-to-end process of how *Paarth's Peppy Peppers* makes its way to Pod 2.

  Prior to the move, a web request from an end-user would look something like this:



A GET request to *Peppy Peppers'* store is processed at an Ingress (that is an NGINX deployment). Custom routing modules parse the request and consult a routing table to identify which pod the request should be forwarded to. *Peppy Peppers'* lives in Pod 1, so the request is forwarded there.

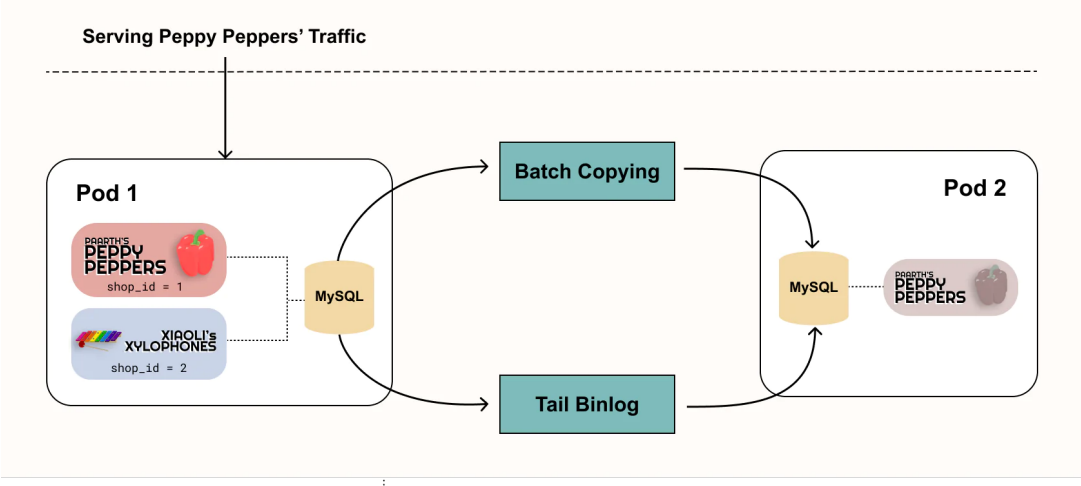
A shop move can be broken down into three distinct high-level phases:

1. Batch copying and tailing MySQL's binary log (binlog)
2. Entering cutover
3. Update control plane, serve traffic, and clean stale data

Phase One: Batch Copying and Tailing Binlog

To perform the data migration, we use a library called Ghostferry: an open-source tool, built in-house, designed to copy data from one MySQL instance to another. Its need arose when Shopify migrated to the cloud.

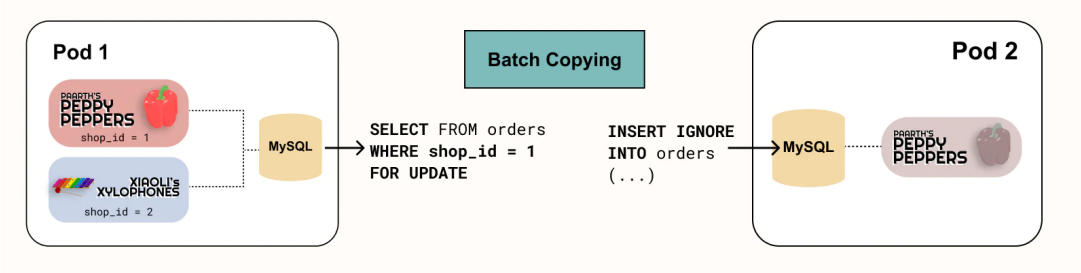
Ghostferry uses two main components to copy over data: batch copying and tailing the binlog.



Ghostferry migrates data from Pod 1 to Pod 2 by, both, batch copying between the two MySQL instances and tailing the binlog.

Ghostferry performs batch copying by iterating over the set of

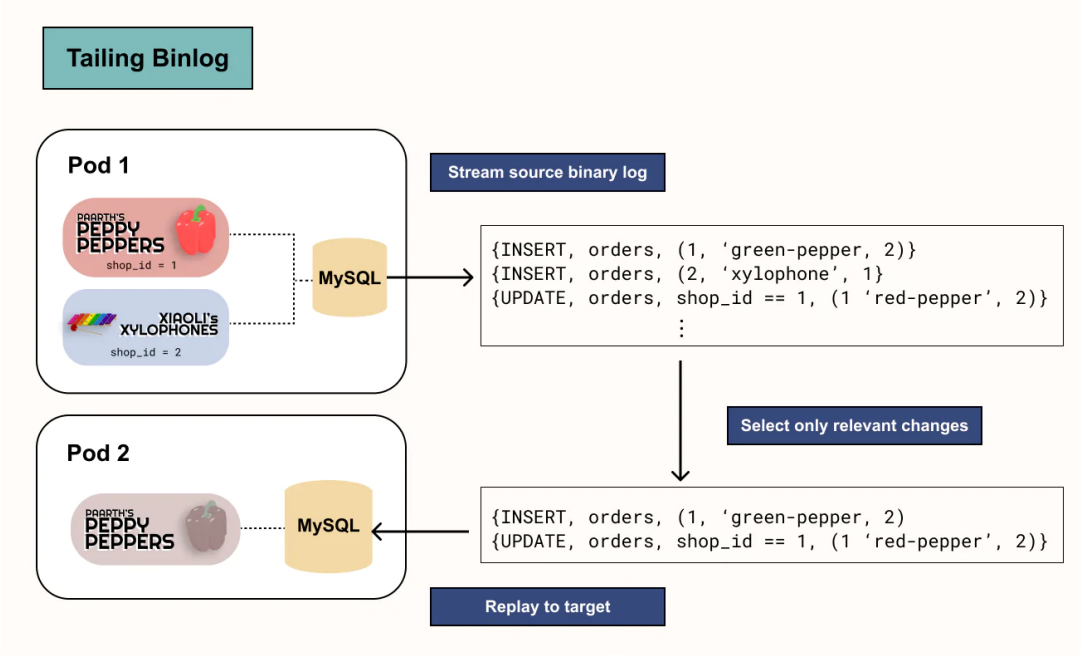
tables on the source, selecting relevant rows based on the shop's ID, and writing these rows to the target; each batch of writes is performed in its own MySQL transaction. When Ghostferry writes batches of rows to the target, it's important to ensure these same rows aren't being changed on the source, as this can lead to data corruption. To do so, Ghostferry uses MySQL's `SELECT...FOR UPDATE` statement. The `SELECT...FOR UPDATE` statement is a means of implementing locking reads: selected rows from the source are write-locked for the duration of the transaction. Ghostferry uses this statement to maintain data correctness and guarantee atomicity of the entire read-then-write transaction. Ghostferry operates safely knowing it can commit these rows on the target, while the data on the source remains unchanged, preventing data corruption that may stem from possible race conditions.



Ghostferry performs batch copying between Pod 1 and Pod 2. Peppy Peppers' orders can be migrated by running `SELECT FROM orders WHERE shop_id = 1 FOR UPDATE` on Pod 1's database. These records are then inserted into Pod 2's database.

Simultaneously, Ghostferry makes use of MySQL's binlog to track changes that occur on the source and replay those changes on the target. MySQL offers the binlog as a sink for events that describe the changes occurring to a database, positioning the binlog as the source of truth. When configured with row-based replication, the binlog contains a list of all the individual operations performed on rows in the database. Ghostferry streams these changes from the source's binlog, filters only the changes that are relevant to the

shop, and applies these changes to the target.

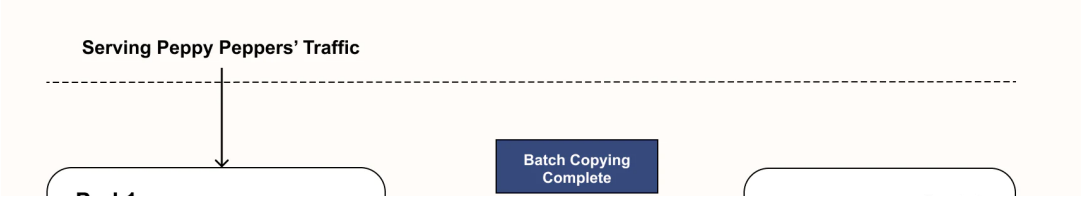


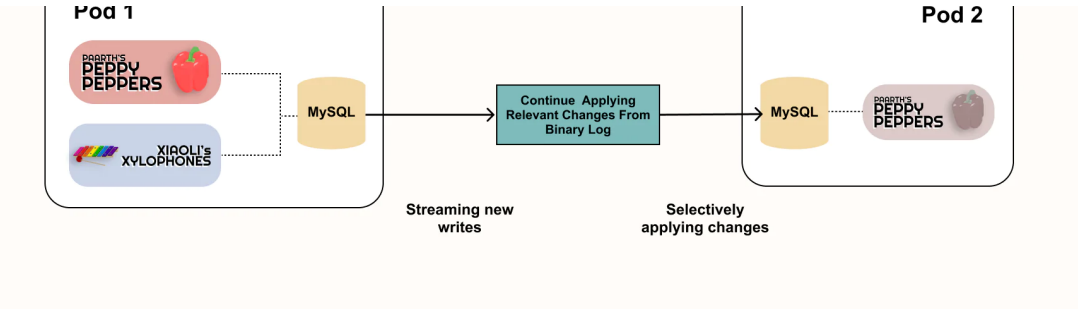
Ghostferry tails Pod 1's binlog and selects only relevant changes. For instance, it removes all events that are related to Xiaoli's Xylophones (shop_id = 2). It applies or replays relevant changes to the target database in Pod 2.

To improve throughput, Ghostferry can operate concurrently, copying data from multiple tables at a time in different threads. The copying process occurs in the background and doesn't interfere with the operation of the merchant's store. Thus, *Peppy Peppers*' storefront is active during this time and traffic is still served from Pod 1.

Phase Two: Entering Cutover

Once batch copying completes, all of the data that existed for *Peppy Peppers* at the start of the move lives in Pod 2's MySQL shard. Now, Ghostferry continues to copy the new writes and ensure they're replicated to Pod 2.

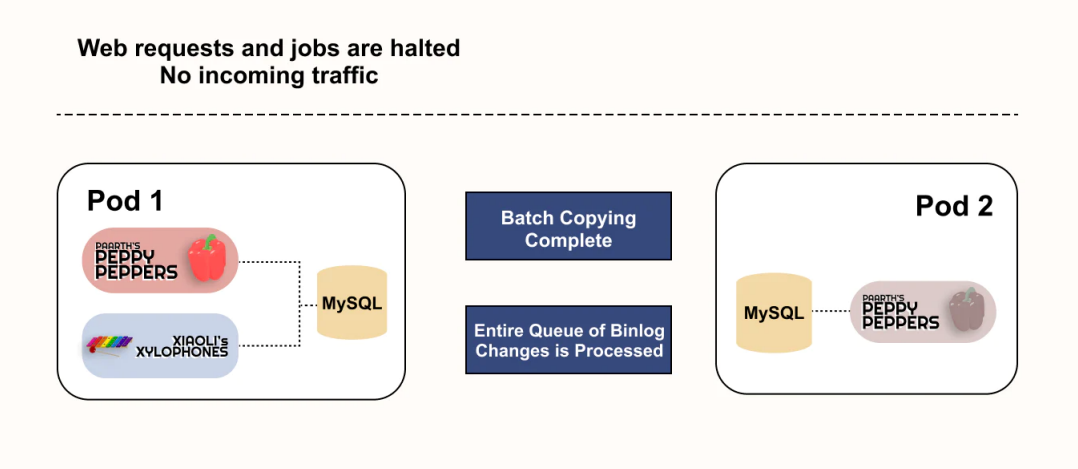




Batch copying between Pod 1 and Pod 2 is complete. The only active data transfers involved are those caused by new writes to Pod 1, and thus new entries on the binlog. These are replayed to the target on Pod 2.

Ghostferry enters the cutover when the queue of binlog events pending replay is small; the queue is considered small when the difference between newly *generated* binlog events and those events being replayed is effectively real-time—on the order of seconds.

Once the cutover is entered, writes to the source database must stop to ensure no new binlog events are dispatched. At this time, Ghostferry records the source's final binlog coordinate as the stopping coordinate. Ghostferry processes the remaining queue of events until it reaches the stopping coordinate, at which point copying is considered complete.



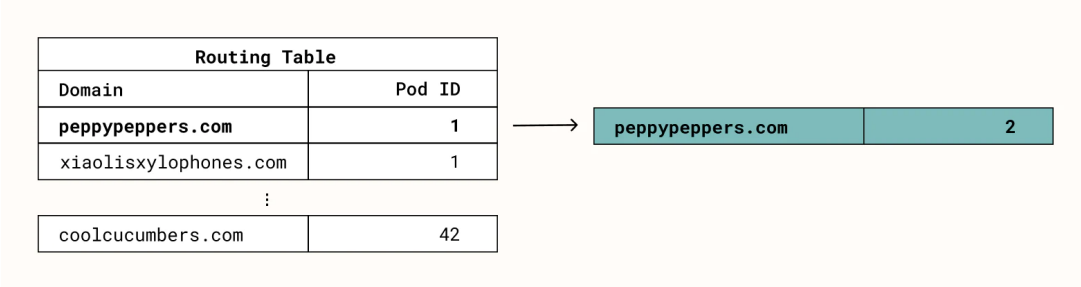
Ghostferry has completed batch copying and tailing the binlog of Pod 1 as all the shop's data has made its way to Pod 2. Web requests and jobs for Peppy Peppers' are halted so that no new

binlog events are generated. Ghostferry has entered the cutover.

One of the key constraints of the entire rebalancing strategy is avoiding data loss. To enter cutover safely, we *enforce* that no units of work (that is web requests and jobs) that can mutate the shop's data are running. To do this, we employ an application level multi-reader-single-writer (MRSW) lock. These Redis-backed locks are used to guarantee the shop mover's ability to command exclusivity over a shop. Before a shop move can begin, any unit of work scoped to that particular shop is required to hold a reader (or shared) portion of the MRSW lock. Any number of requests can hold this portion of the lock, as long as the writer (or exclusive) lock isn't held. Jobs that cannot be scoped to the particular shop are required to hold a similar global lock. To enter the cutover stage, the shop mover waits for the reader locks to be released and acquires the writer lock. This asserts that no writes for the shop are executed on that pod. If the shop mover cannot acquire the writer lock in time, it fails the move.

Phase Three: Update Control Plane, Serve Traffic, and Prune Stale Data

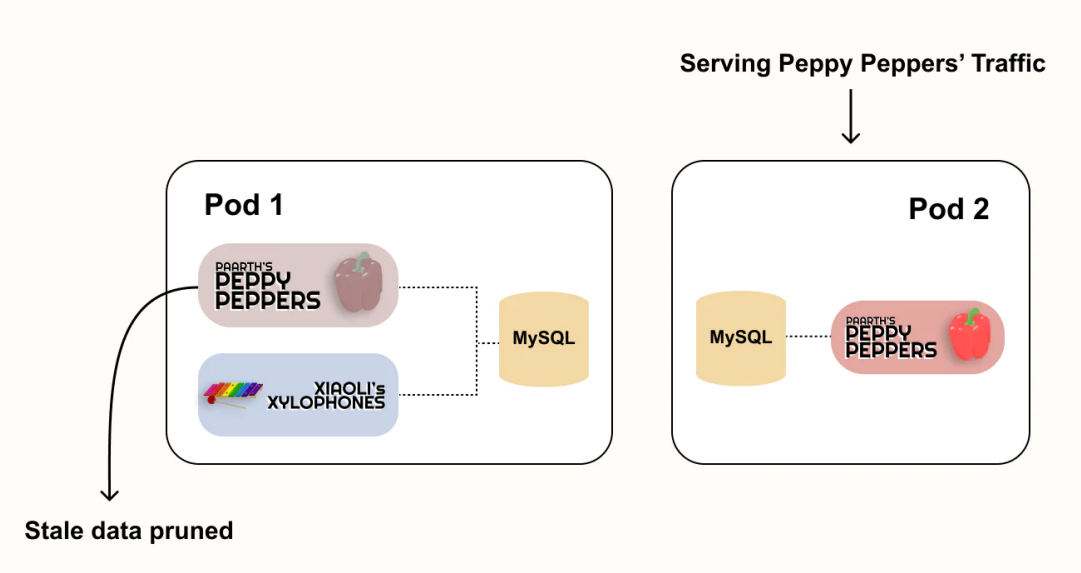
With confidence that there's no data loss, the shop mover updates the control plane. The routing table is updated to associate the shop with its *new* pod. This configuration lives in a separate database that isn't sharded.



The routing table containing the mapping between domain and Pod ID is updated. Peppy Peppers' Pod ID is updated to 2.

As soon as the routing table is updated, the shop mover releases the exclusive lock and allows units of work to proceed and ultimately write again—this time on the new pod. The only opportunity for downtime is during the cutover. Availability is a critical constraint, so cutover is designed to be a short process to minimize downtime.

So, the shop is now served from its new pod. The old pod still contains the shop's data, though. Our system begins to perform a verification to ensure the move proceeded as expected and that no new writes are surfaced to the old pod. This includes ensuring that no queries are routed to the original (source) shard for a period of time after a shop has been successfully moved. Once the move is deemed successful, stale data on the old pod is pruned.



Now, traffic for Peppy Peppers' is served from Pod 2. Pod 1 still contains Peppy Peppers' now-stale data. Pod 1 is cleaned at a later time by pruning all of its stale data.

Peppy Peppers is now served from Pod 2. Database utilization is equalized across the platform, and two high-traffic merchants are isolated at the pod level.

Risky Business: Verification and Correctness

Real-time, online data migration is a risky process. We've touched on some of the runtime and database requirements. We've also explored Ghostferry's key phases: batch copying, binlog tailing, performing cutover and updating the control plane. Still, we've yet to explore more complex features like Ghostferry's concurrency support or interrupting and resuming migrations. As more features are added, it becomes increasingly important to ensure the integrity of the code remains. To increase confidence, Shopify places a strong emphasis on correctness, safety, and verification.

The Ghostferry library and wider system boast a suite of verifiers that run before, during, and after the data migration. Verification involves ensuring no data corruption, completeness of data transfer, and authenticity. Further, Ghostferry's central algorithm has been modelled and represented in a formal specification used to argue about its correctness. It's written in TLA+.

Moving a shop from one shard to another requires engineering solutions around large, interconnected systems. The flexibility to move shops from shard to shard allows Shopify to provide a stable, well-balanced infrastructure for our merchants. With merchants creating their livelihood on the platform, it's more important than ever that Shopify remains a sturdy backbone. High-confidence shard rebalancing is simply one of the ways we can do this.

Paarth Madan is a Dev Degree intern, currently working on the Rails Infrastructure team. After joining Shopify in 2018, he spent 2 years working on the Shop app in both backend (Ruby on Rails) and mobile (React Native) capacities. He spent the last 8 months with the Database Engineering team where he developed a passion for databases, cloud infrastructure, and multi-tenancy.

Additional Information

- A Pods Architecture to Allow Shopify to Scale

- Ghostferry

Xiaoli Liang is a development manager on the Database Engineering team. Her focus is to champion a scalable, resilient and efficient database platform at Shopify with autonomous tooling for data placement, organization and development. If building systems from the ground up to solve real-world problems interests you, Xiaoli's team is hiring a Lead/Staff Production Engineer. This role is fully remote. Learn how Shopify is Digital by Default.