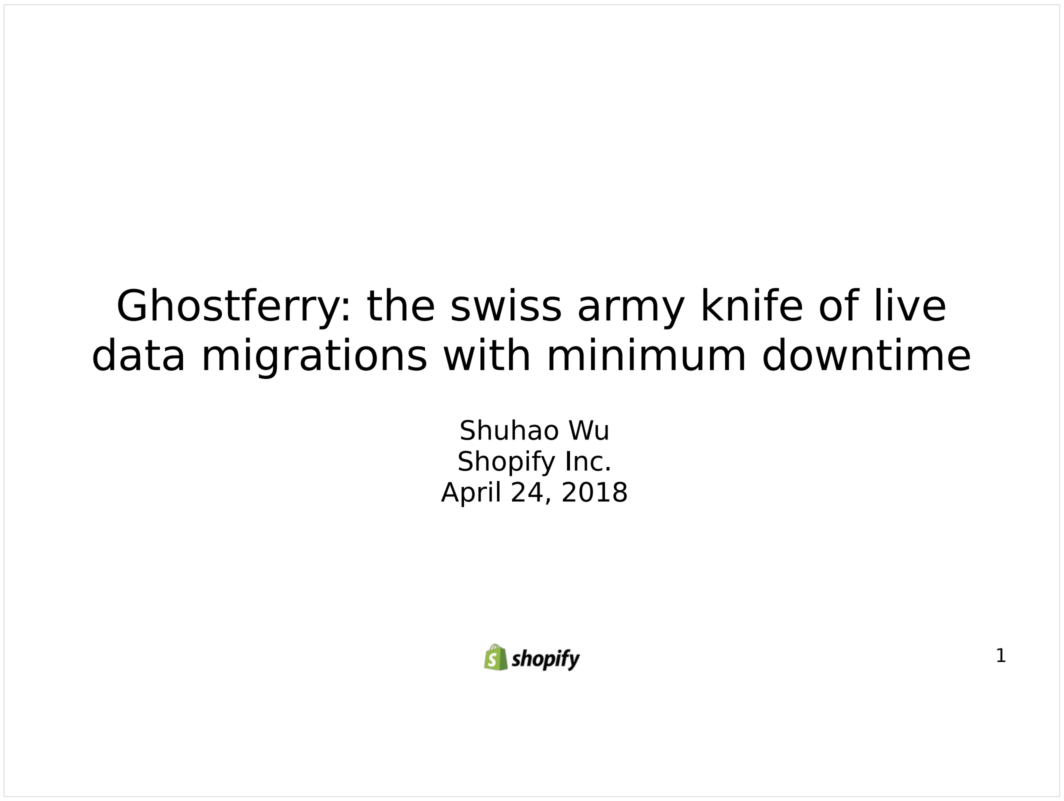


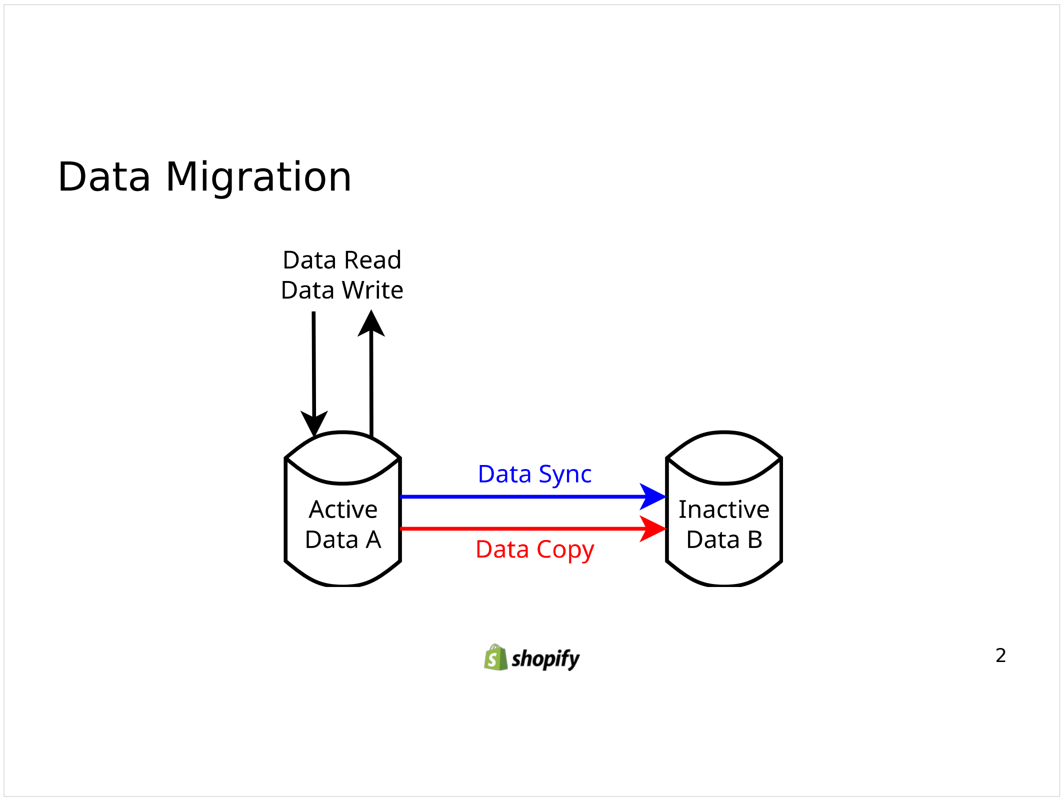
Ghostferry: the swiss army knife of live data migrations with minimum downtime

Shuhao Wu
Shopify Inc.
April 24, 2018

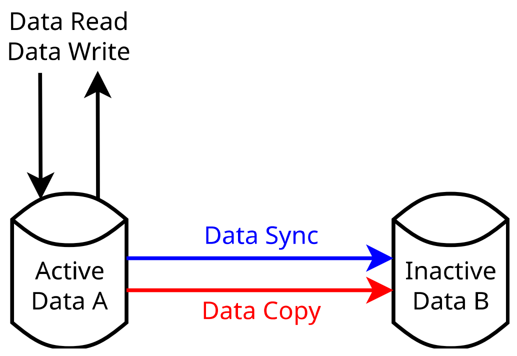
shopify

Hello! My name is Shuhao and I'm a developer on the datastores team at Shopify and we are responsible for maintaining the majority of Shopify's MySQL deployments.

Today, I'll be talking about Ghostferry, a general purpose MySQL data migration tool that allows us to migrate data between different MySQL installations of **arbitrary scale** with the **push of a button**.

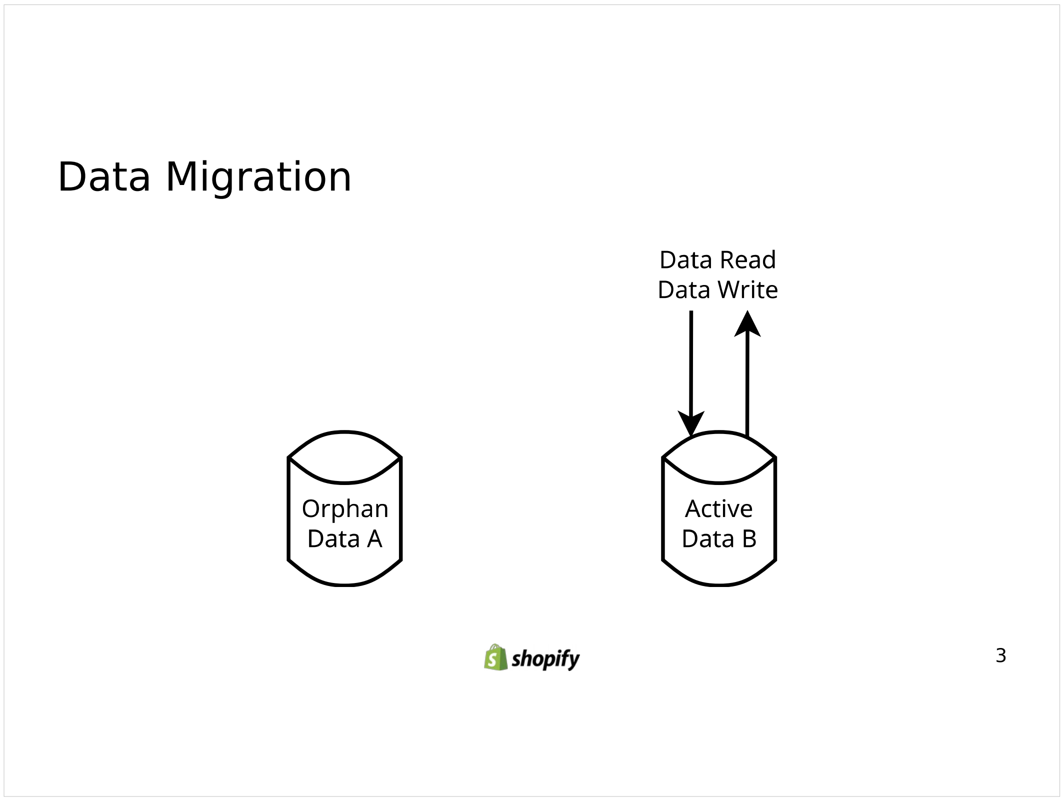


Data Migration



Before we dive into MySQL, let’s talk about data migrations in general so we are all on the same page. Specifically, we’re interested in **online data migration**, where the data can be modified during the copy process.

Datasets generally have an **associated location**, from where we access this data set. We call **changing the dataset’s location a “data migration”**. In the general case, this involves 3 basic steps: (1) we must somehow **copy** the data from the source to the target and this process **could take a long time**; (2) while the data is being copied, we have to ensure that the changes to the source data set is properly **synchronized** to the target; [GO TO NEXT SLIDE]

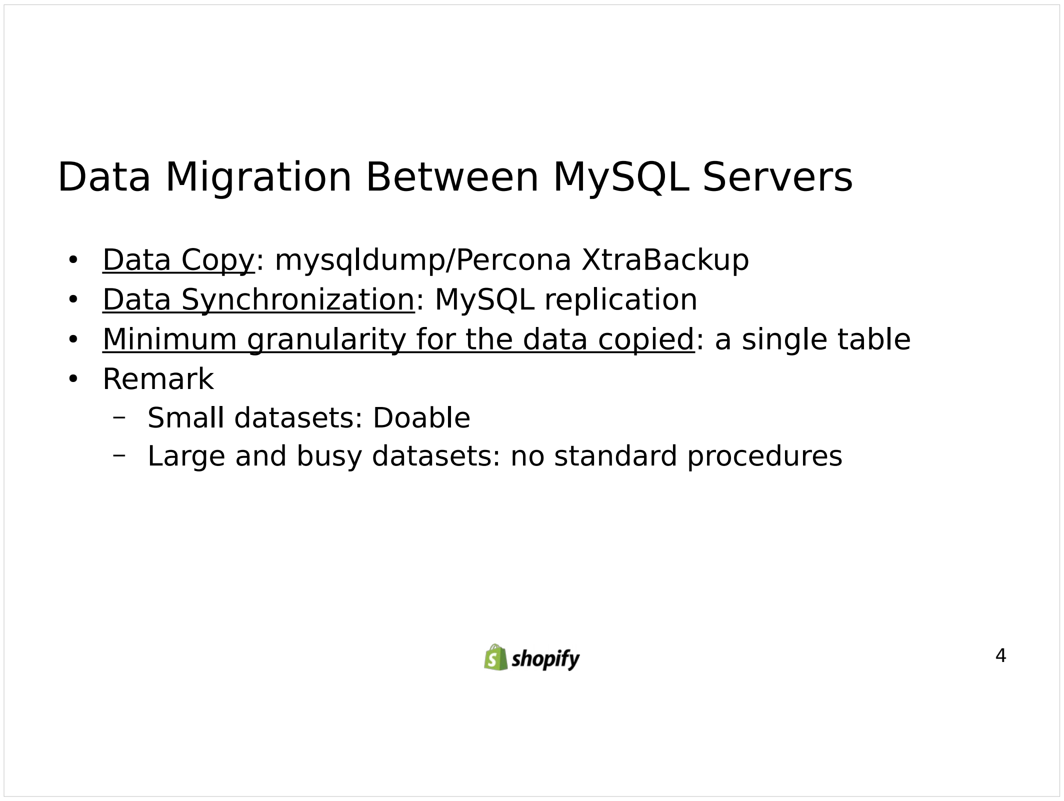


Data Migration

shopify

(3) when the data copy is complete, we need to ensure that everything that access the dataset now recognize the **target** as being the **“source of truth”** and we call this last step the “cutover” operation.

This process is fairly complex and the majority of the complexity arises from **the requirement of synchronizing** the process. We can simplify this process if we simply lock the source dataset somehow by making it **read only**. However, this **introduces downtime** for the source dataset that may be unacceptable.



Data Migration Between MySQL Servers

- Data Copy: mysqldump/Percona XtraBackup
- Data Synchronization: MySQL replication
- Minimum granularity for the data copied: a single table
- Remark
  - Small datasets: Doable
  - Large and busy datasets: no standard procedures

4

With these in mind, we can now look at moving data between MySQL servers. Specifically, ones we completely **control**, such as the ones we **deployed ourselves**.

We can use mysqldump or percona backup to copy the data and mysql replication to synchronize the data. This is relatively standard procedure. One might run into mysql performance issues with mysqldump when the dataset is **large and busy**, although you can use percona xtrabackup to **get around this**.

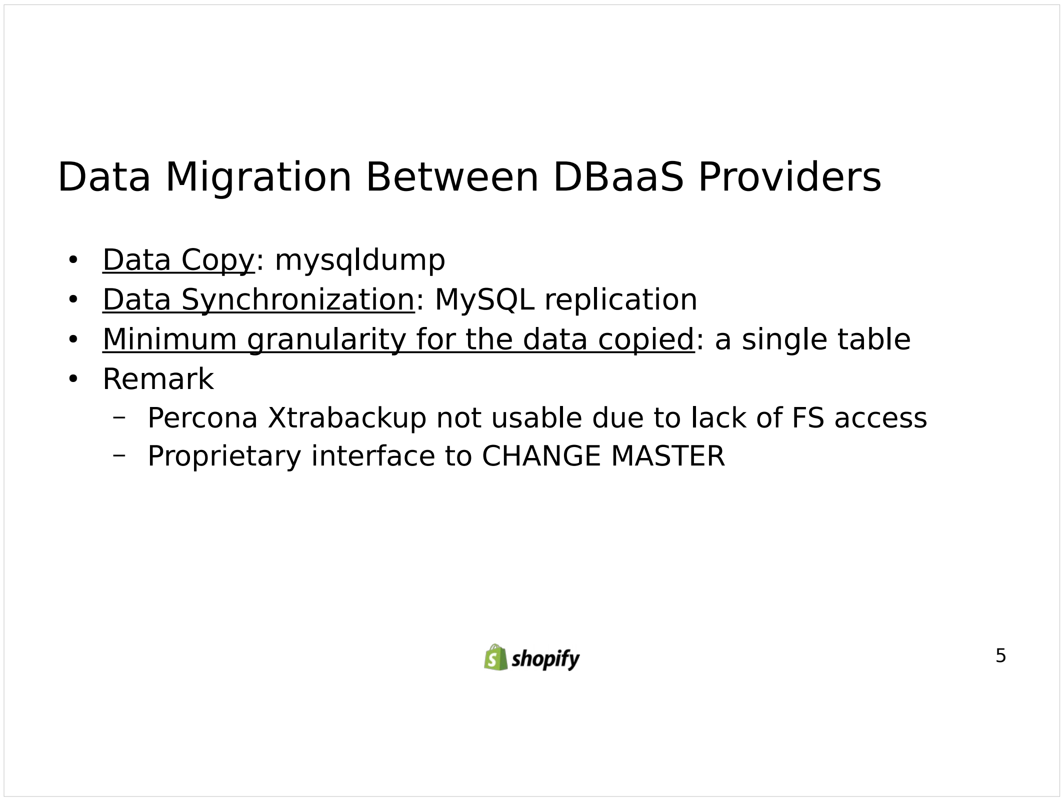
For the record, we're only dealing with scenarios here where **turning off the source database for extended periods of time is unacceptable**. Otherwise you could just set the source to read only and **skip this talk all together**.

For the cutover step, this is usually application specific. For example, you could **change the hostname** of your database to point to a different IP once the data migration is completed.

An interesting thing to look at for data migration in MySQL that we didn't address in our general case is how we can copy a subset. **While mysqldump is able to filter the data subject to a WHERE condition, mysql replication canno**t. Thus, for an online migration, the minimum amount of data we can move is a single table.

Another remark to make here is that the flow is not super easy to operate. Users have to know all the **correct commands and flags** to specify, with **many manual steps**. This can be easily doable for smaller datasets. There are no standard procedures for these larger and busier installations as there may be **special requirements**.

We discussed data migration between multiple MySQL servers that we control. **What if we don't control these instances?** This is a common scenario if you use hosted MySQL services like the ones provided by Amazon RDS, Google CloudSQL, or other database as a service providers.



Data Migration Between DBaaS Providers

- Data Copy: mysqldump
- Data Synchronization: MySQL replication
- Minimum granularity for the data copied: a single table
- Remark
  - Percona Xtrabackup not usable due to lack of FS access
  - Proprietary interface to CHANGE MASTER

shopify

5

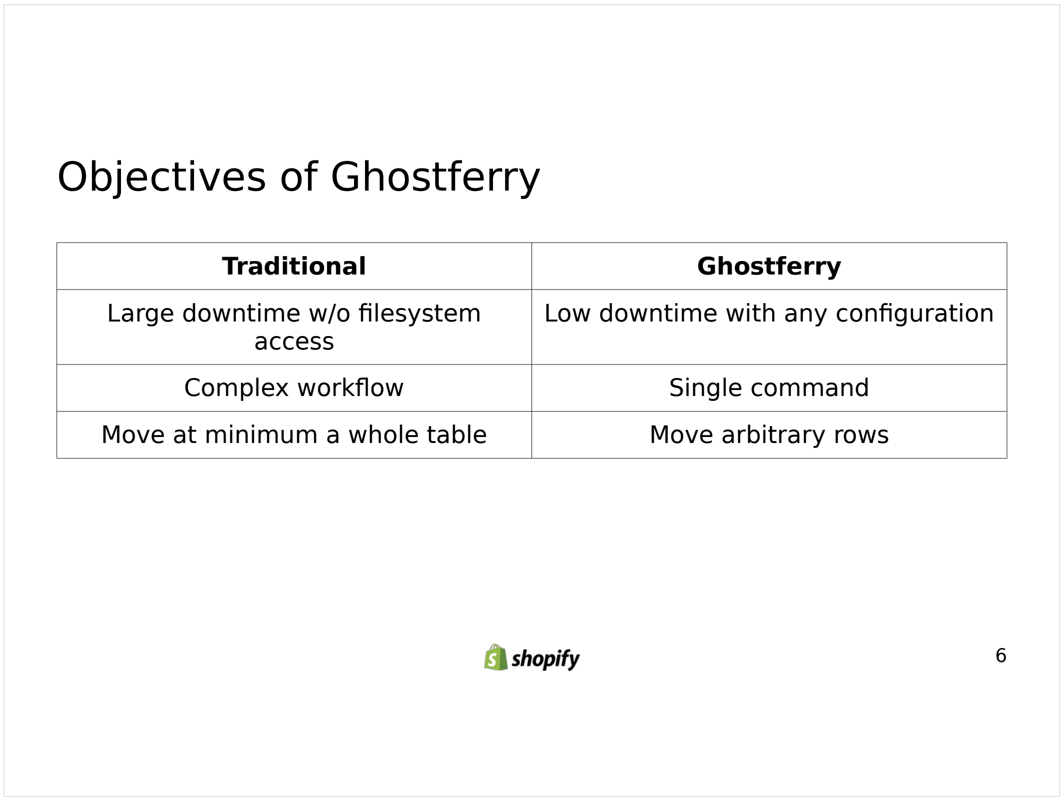
In these cases, we probably don't have access to the file system on which these databases are hosted. This means we cannot use Percona Xtrabackup and **must use mysqldump**. As mentioned, for larger datasets, we could have unacceptable performance penalties for using mysqldump.

If we are lucky enough to have access to replication, we could setup replication to synchronize our data during the copy process. However, providers don't necessarily expose all of the interfaces from MySQL such as **CHANGE MASTER directly**. CloudSQL, for example, exposes **only GTID** based replication via their tooling. This could cause issues as **another provider** might not support GTID based replication and thereby creating a situation where it is **impossible to setup replication** between the two providers directly. Such a situation means we have to introduce downtime on the source to prevent writes in order to **guarantee data consistency** during the copy process, which is not necessarily acceptable.

If a provider doesn't expose replication, the problem grows a lot more difficult and we do not address this in Ghostferry and therefore consider it to be **out of scope**. However, we don't want to claim this is impossible, either.

Once again, the **minimum granularity** of the data being copied is a single table due to limitations with how MySQL replication works.

To summarize, the procedures here are relatively standard for smaller datasets, especially if you can tolerate downtime. For larger and busier ones, it becomes **more difficult** as some tools may not be usable. Once again, the workflow here, for any scale, involves a large amount of **manual commands** and may require a lot of ad-hoc solutions depending on your setup.



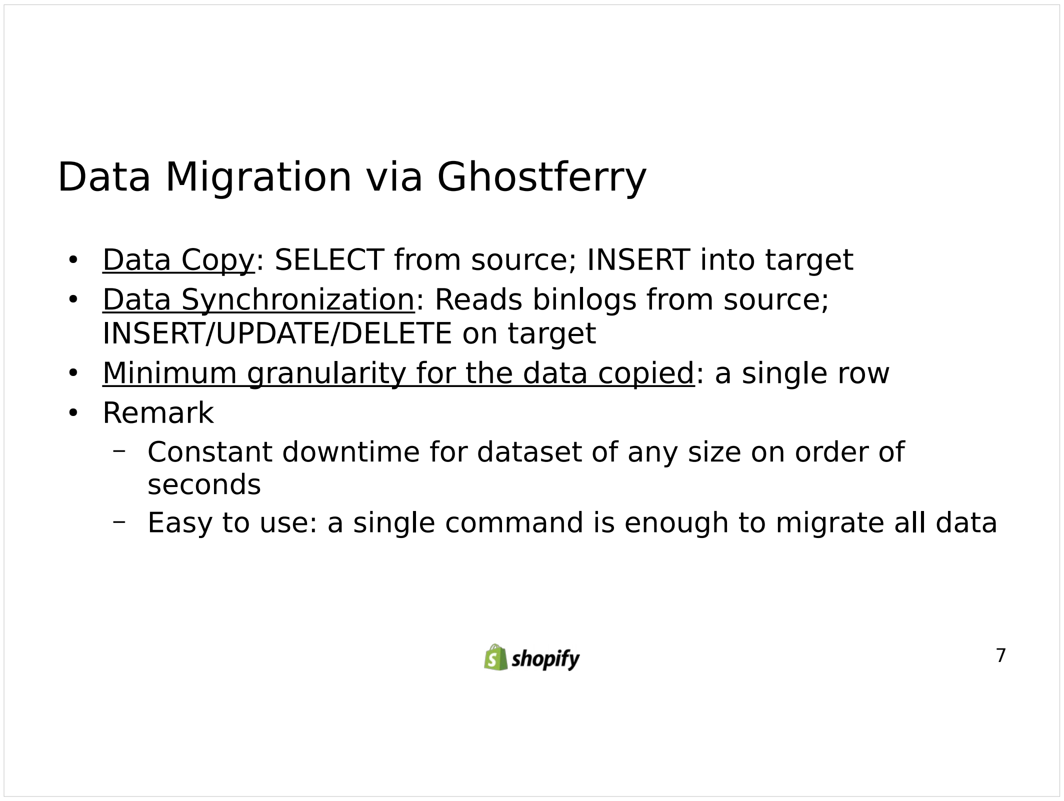
Objectives of Ghostferry

Traditional	Ghostferry
Large downtime w/o filesystem access	Low downtime with any configuration
Complex workflow	Single command
Move at minimum a whole table	Move arbitrary rows

Shopify is currently moving some databases to cloud based setups and we’ve managed to run into all of the issues we’ve discussed. Specifically, we cannot access the **filesystems** of the cloud databases and we had trouble with setting up **replication** via proprietary interfaces. This means to move, we subject the source databases to **large downtime**, which is **unacceptable** to us.

Since we are moving a large number of these databases, we wanted a **single command solution** that handled all of the details such that it can be performed by a **non DBA with a minor amount of training**.

Lastly, we also we wanted to move datasets smaller than a single table subject to a WHERE constraint. This is the nail in the coffin with **traditional tooling** as it is **not possible**. Thus we are stuck with creating a completely new solution.



Data Migration via Ghostferry

- Data Copy: SELECT from source; INSERT into target
- Data Synchronization: Reads binlogs from source; INSERT/UPDATE/DELETE on target
- Minimum granularity for the data copied: a single row
- Remark
  - Constant downtime for dataset of any size on order of seconds
  - Easy to use: a single command is enough to migrate all data

shopify

So let’s take a look at an **overview** of what Ghostferry does under the hood.

Instead of using mysqldump or percona xtrabackup to copy the data, Ghostferry simple SELECT the data from the source and INSERT them into the target. This should work universally as all MySQL instances **must expose the SELECT and INSERT** interfaces for the system to even be useful. To ensure we do not impact performance on the source database, Ghostferry can **throttle** the rate at which it SELECTs.

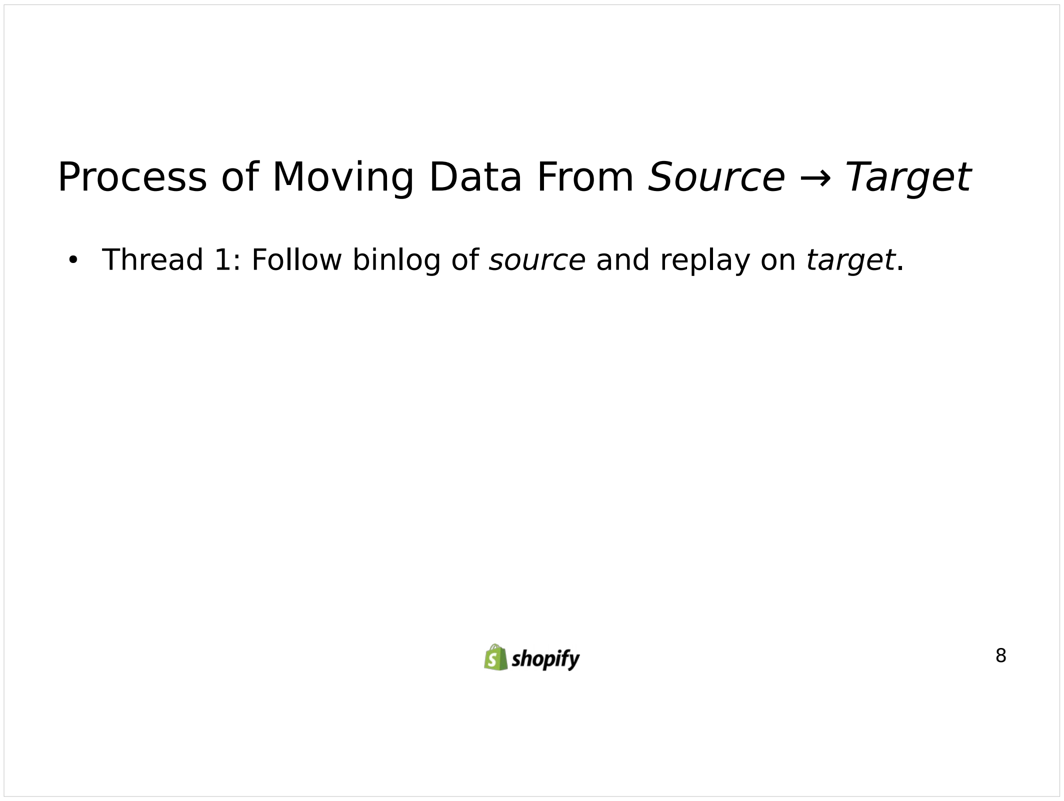
Instead of using MySQL replication to synchronize the data during the copy process, Ghostferry essentially acts as a **replication proxy**: it connects to the source as a replication slave and replays the binlogs with INSERT, UPDATE, and DELETE statements on the target.

In our design of Ghostferry, we left open the possibility for arbitrary filter on both the copy and synchronization process. This allow us to migrate arbitrary rows from the source to the target via **user specified constraints**.

The downtime incurred in this entire process happens during the cutover, where we switch the source of truth of the data to the target database. This is no different from the **best case scenario of traditional tooling**.

Since all of these processes are **integrated** into a single command, the user only needs to know how to operate Ghostferry instead of performing a bunch of commands such as mysqldump and CHANGE MASTER. This allows people to run Ghostferry to migrate their data with only a **moderate amount of training**.

Now that we have a general idea on how Ghostferry works, let’s look at **the process, step by step.**

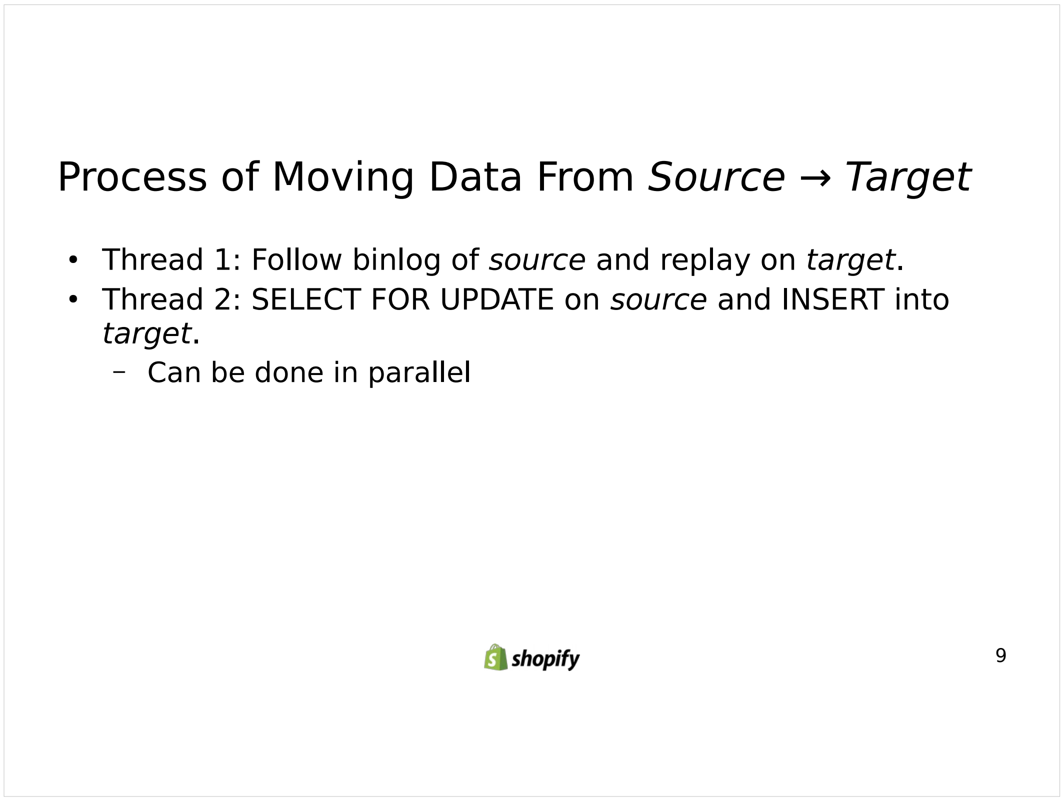


Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.

shopify

When Ghostferry starts, it immediately begins to read the source database's binlogs, which is a changelog that MySQL provides. It applies these log entries to the target in the **background**, **until the end of the run**.



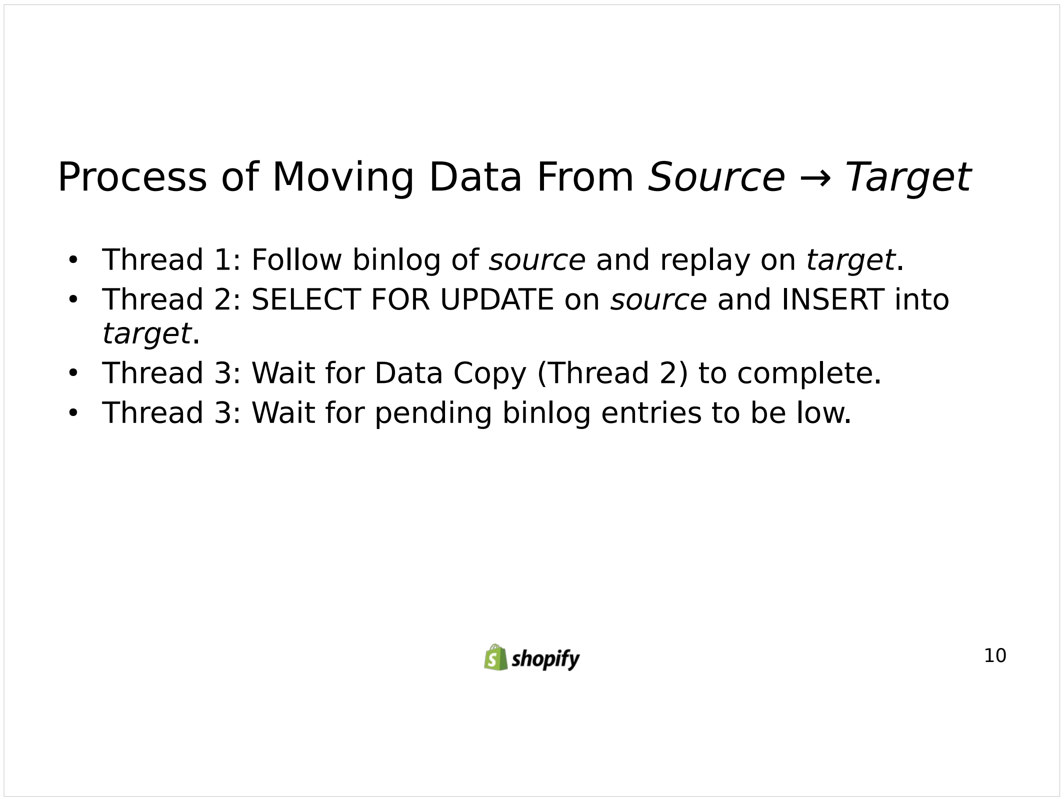
Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
  - Can be done in parallel

shopify

As the binlogs are applied in the background, Ghostferry starts a **separate thread** that copies data from the source to the target via SELECTs and INSERTs. One thing we do here is use a SELECT FOR UPDATE to lock the source rows while we insert it into the target. This is to guarantee consistency of the data being moved as otherwise there is a **race condition**.

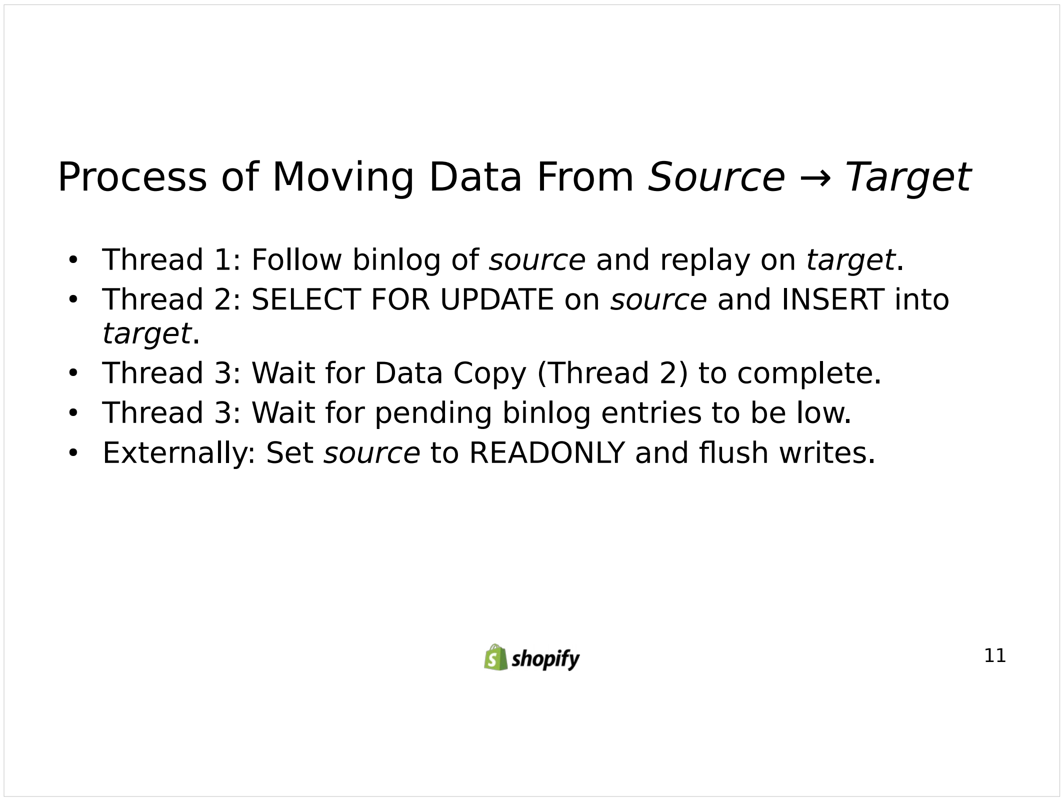
Note that this copy process can be done in **parallel** to speed up the run.



Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
- Thread 3: Wait for Data Copy (Thread 2) to complete.
- Thread 3: Wait for pending binlog entries to be low.

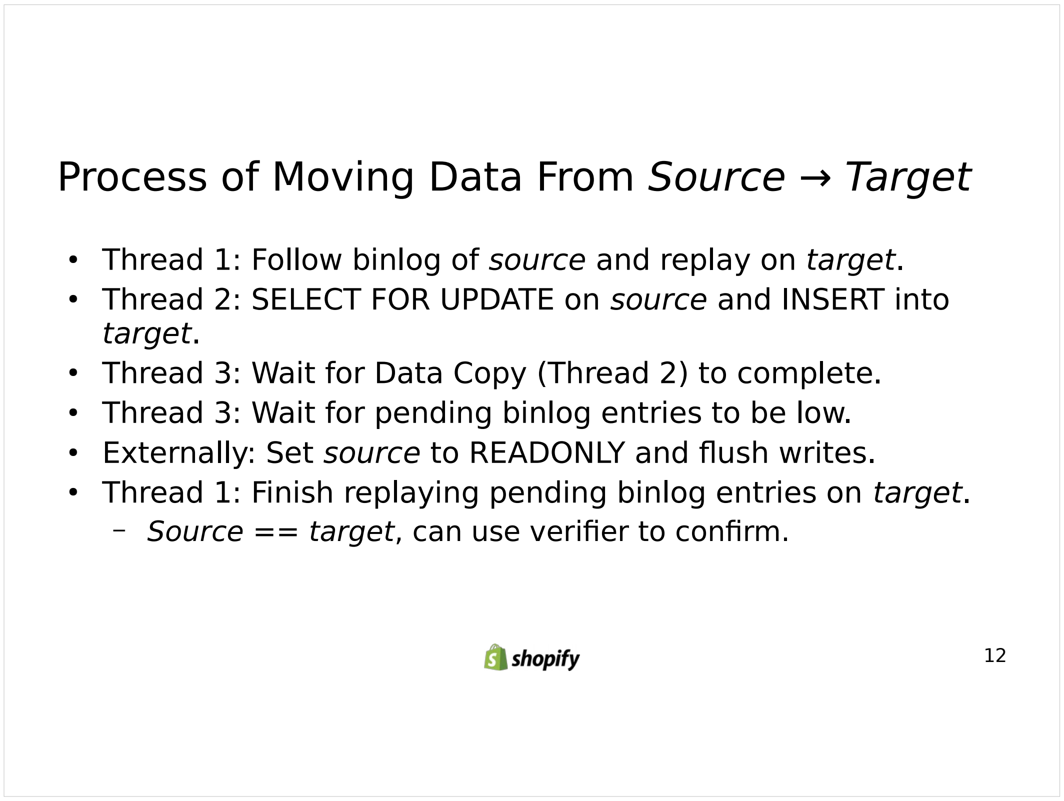
We then wait for the data copy process to complete. We also wait for the **amount of pending** binlog entries, or **Ghostferry's replication lag**, to be low **for a later step**.



Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
- Thread 3: Wait for Data Copy (Thread 2) to complete.
- Thread 3: Wait for pending binlog entries to be low.
- Externally: Set *source* to READONLY and flush writes.

In order to guarantee that the source is the same as the target at the end of the copy, we must then prevent any further writes to the source and flush all of the pending writes s**o they show up in the binlogs**. This is done **outside of Ghostferry** as there are many different ways to accomplish this. As an example, you can set the source to be READONLY via a database variable or you can put your application into some sort of read only mode to prevent writes.



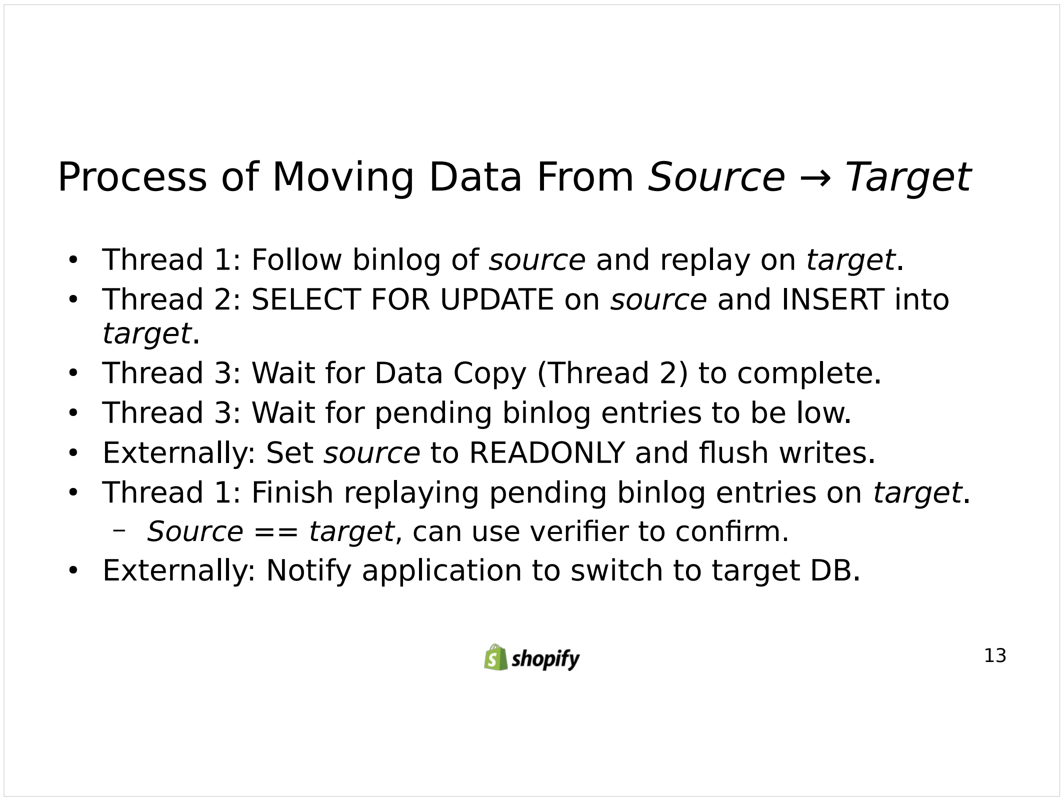
Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
- Thread 3: Wait for Data Copy (Thread 2) to complete.
- Thread 3: Wait for pending binlog entries to be low.
- Externally: Set *source* to READONLY and flush writes.
- Thread 1: Finish replaying pending binlog entries on *target*.
  - *Source* == *target*, can use verifier to confirm.

shopify

Once the source is read only, we apply the **last few remaining** entries in the binlogs to the target. This number should be small as we have already waited for the number of pending binlog entries to be low before this step. Thus, this operation should be very fast.

At this point, the source and the target are **identical**. If you don’t feel safe, you can use one of the two verifiers to verify the consistencies of your data. This does **add downtime** to the migration, however.

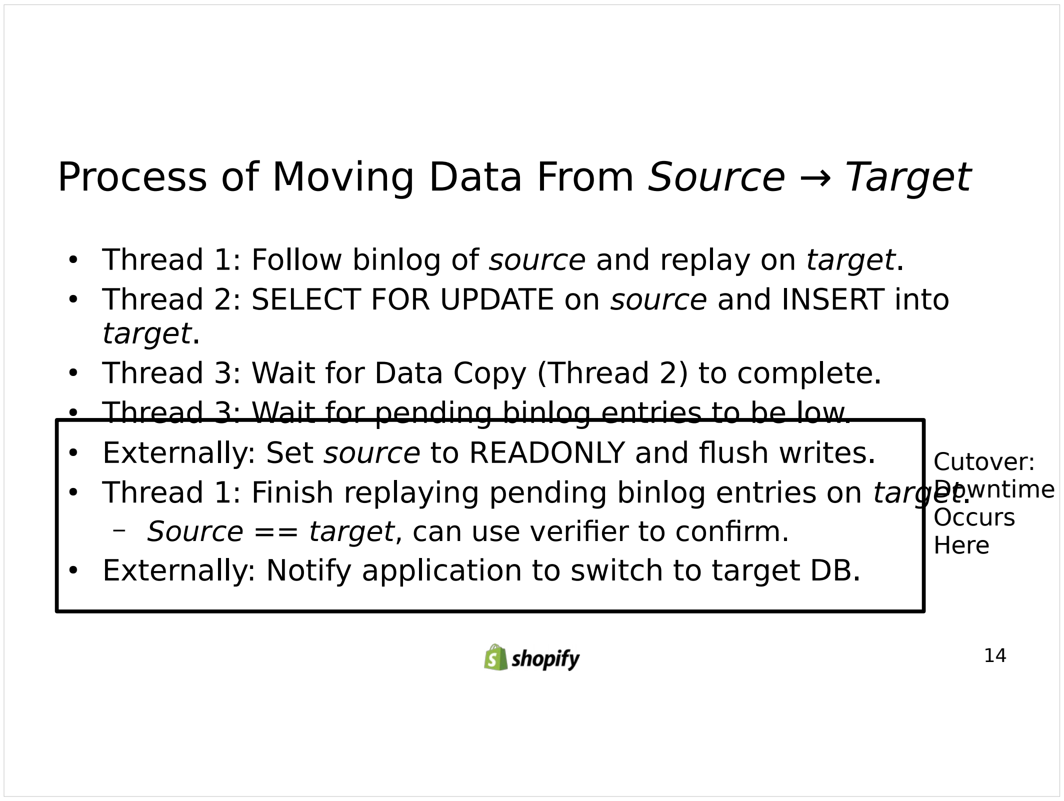


Process of Moving Data From *Source* → *Target*

- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
- Thread 3: Wait for Data Copy (Thread 2) to complete.
- Thread 3: Wait for pending binlog entries to be low.
- Externally: Set *source* to READONLY and flush writes.
- Thread 1: Finish replaying pending binlog entries on *target*.
  - *Source* == *target*, can use verifier to confirm.
- Externally: Notify application to switch to target DB.

shopify 13

Once the **optional** verification is done, you should switch the application to use the target database. As an example, you might update the hostname of the database to point to the target instead of the source.



Process of Moving Data From *Source* → *Target*

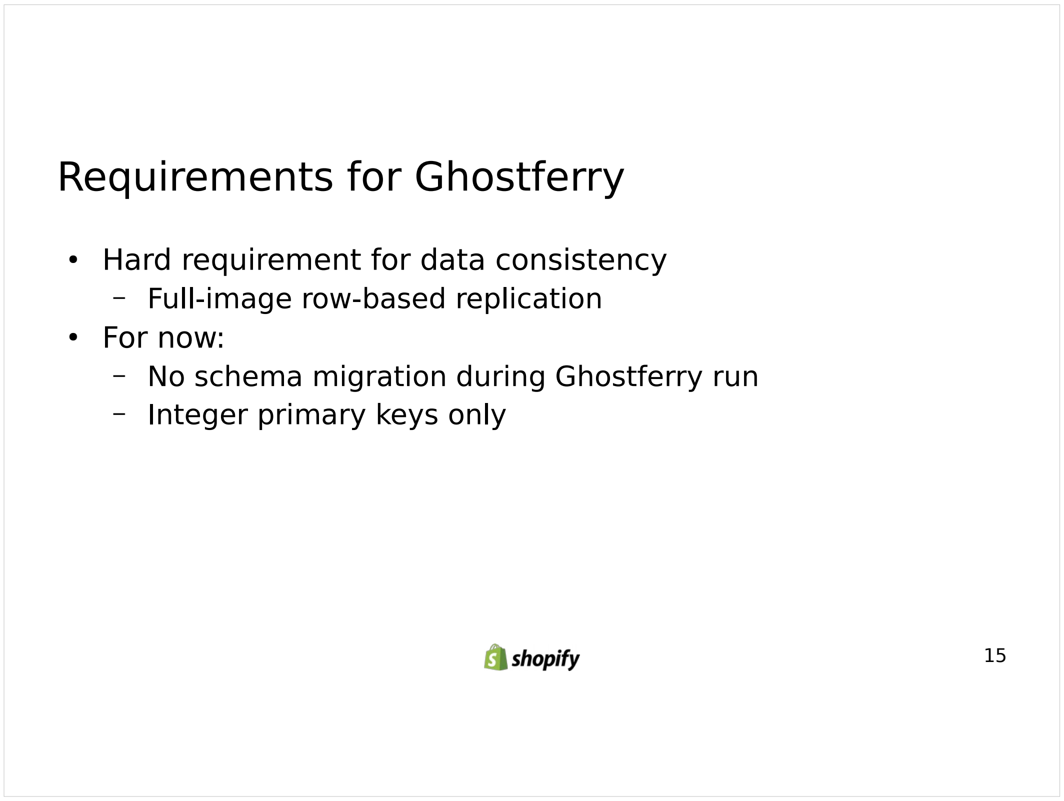
- Thread 1: Follow binlog of *source* and replay on *target*.
- Thread 2: SELECT FOR UPDATE on *source* and INSERT into *target*.
- Thread 3: Wait for Data Copy (Thread 2) to complete.
- Thread 3: Wait for pending binlog entries to be low.
- Externally: Set *source* to READONLY and flush writes.
- Thread 1: Finish replaying pending binlog entries on *target*.
  - *Source* == *target*, can use verifier to confirm.
- Externally: Notify application to switch to target DB.

Cutover: Downtime Occurs Here

14

The downtime associated with Ghostferry is incurred by these steps, known as the cutover steps. In most cases, this should be on the **order of seconds**. If you add in the verification, it will add more downtime. We have two built-in verification system that can handle databases of different sizes and even for larger databases, it can be **a matter of minutes**.

We could also delay the cutover until the least busy time for the database. Ghostferry **will wait** until you’re ready. It will continuously apply the binlog changes as they come in and thus making sure your **data is synchronized**.



Requirements for Ghostferry

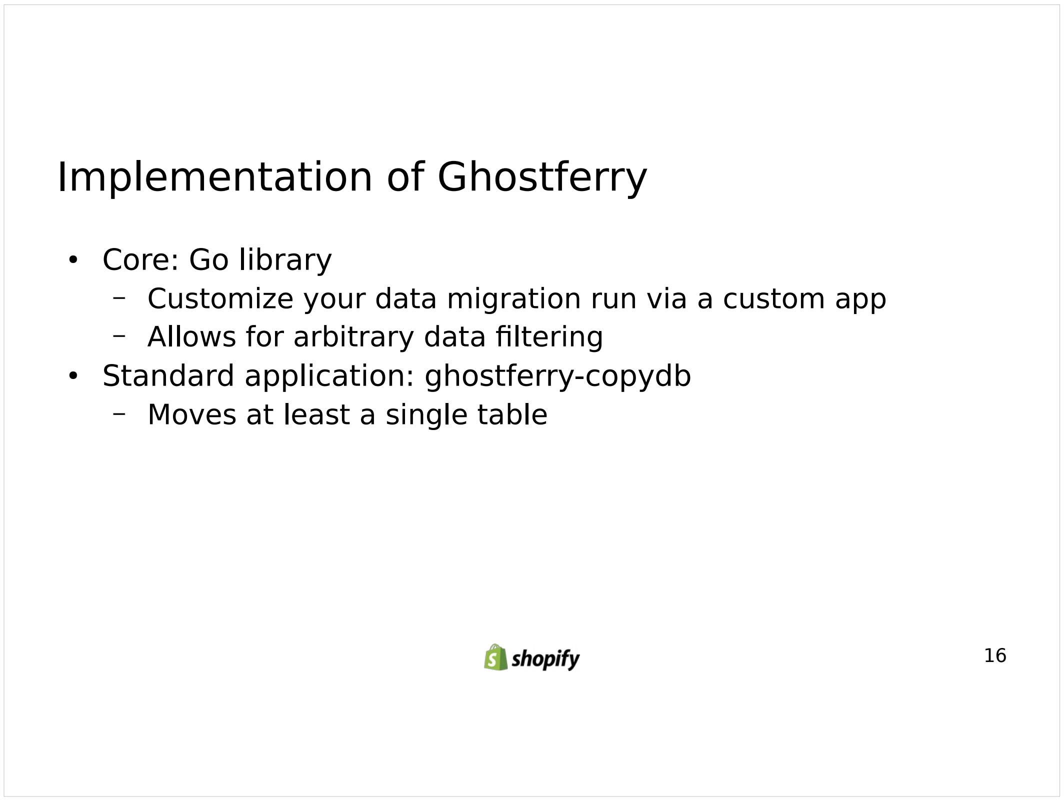
- Hard requirement for data consistency
  - Full-image row-based replication
- For now:
  - No schema migration during Ghostferry run
  - Integer primary keys only

15

So what's the catch? Naturally, this algorithm has some limitations. The most basic one is that replication must be available. Specifically, we require full image row-based replication. For some background, MySQL has two types of replication setup: statement-based and row-based. Statement-based replication is pretty much the full statements as executed on the master to be replayed on the slave. Row-based replication basically records and transmits which rows have changed and **how they have changed** after a statement on the master was executed. Full-image means that the row's **before and after states** are fully transmitted, where as a partial image could contain just the **delta**.

The full image row based replication requirement is non-negotiable for data consistency as we used formal method to **prove** that without it, races resulting in data corruption can occur.

There are also some scenarios that Ghostferry cannot deal with at this point. For now, running schema migrations while running Ghostferry is not safe. It's also not possible to run Ghostferry without an integer primary key. We're aiming to address these limitations in a **future release**.



Implementation of Ghostferry

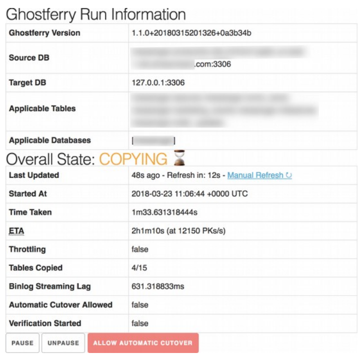
- Core: Go library
  - Customize your data migration run via a custom app
  - Allows for arbitrary data filtering
- Standard application: ghostferry-copydb
  - Moves at least a single table

16

Since moving data generally requires the **cooperation** of other external services, we implemented Ghostferry as a Go library for maximum **flexibility**. Using the APIs exposed by this library, you can fully customize your data migration runs such as filtering your data via some custom **constraints**, calling **external services** for cutover, and even **changing the data** on the fly.

However, if Ghostferry is available only as a library, it would be difficult to use off the shelf for most people. We thus implemented a **standard application** creatively named ghostferry-copydb that copies at least a single table and should be suitable for most people. This application comes with a web UI that I will show in the next slide to monitor and interact with the migration. The caveat is that there’s no arbitrary filtering for this application and the minimum filtering you can do is based on a single table, like the **traditional mysql tooling**.

Implementation of Ghostferry

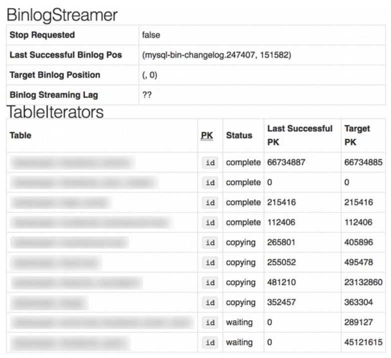


Ghostferry Run Information

Ghostferry Version	1.1.0+20180315201326+0a3b34b
Source DB	[redacted].com:3306
Target DB	127.0.0.1:3306
Applicable Tables	[redacted]
Applicable Databases	[[redacted]]

Overall State: COPYING ⌛

Last Updated	48s ago - Refresh in: 12s - Manual Refresh ↻
Started At	2018-03-23 11:06:44 +0000 UTC
Time Taken	1m33.631318444s
ETA	2h1m10s (at 12150 PKs/s)
Throttling	false
Tables Copied	4/15
Binlog Streaming Lag	631.318833ms
Automatic Cutover Allowed	false
Verification Started	false



BinlogStreamer

Stop Requested	false
Last Successful Binlog Pos	(mysql-bin-changelog.247407, 151582)
Target Binlog Position	(, 0)
Binlog Streaming Lag	??

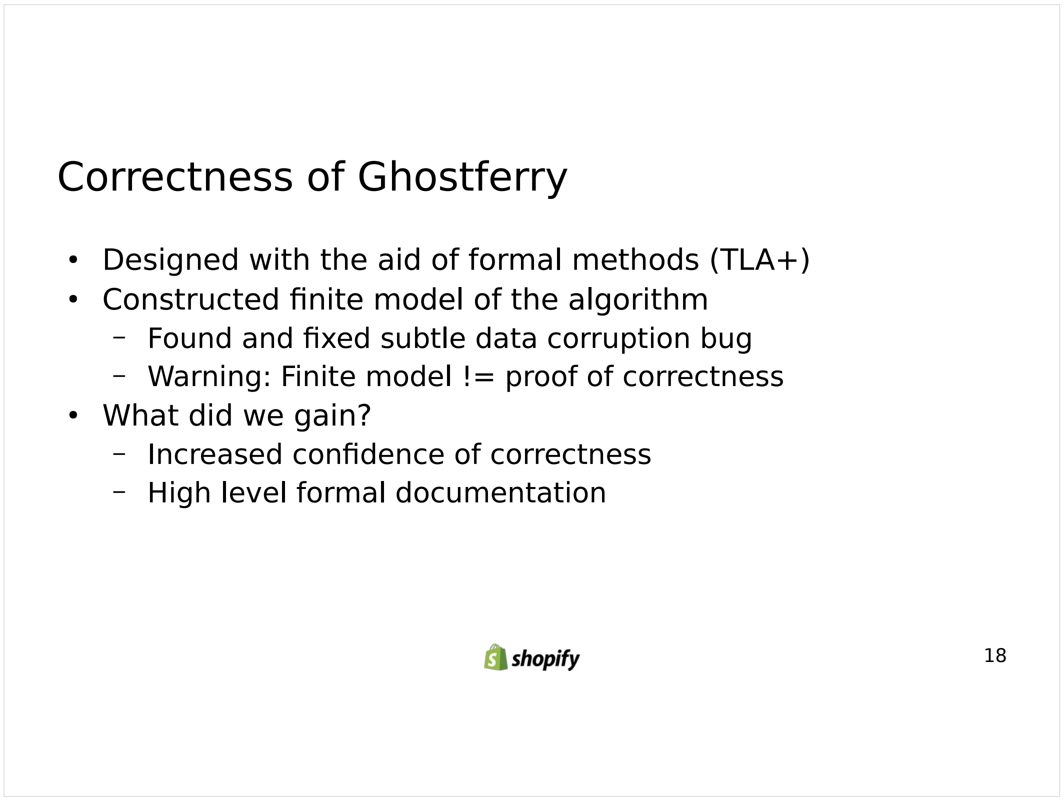
TableIterators

Table	PK	Status	Last Successful PK	Target PK
[redacted]	id	complete	66734887	66734885
[redacted]	id	complete	0	0
[redacted]	id	complete	215416	215416
[redacted]	id	complete	112406	112406
[redacted]	id	copying	265801	405896
[redacted]	id	copying	255052	495478
[redacted]	id	copying	481210	23132860
[redacted]	id	copying	352457	363304
[redacted]	id	waiting	0	289127
[redacted]	id	waiting	0	45121615

shopify

Here are the screenshots that I promised. It shows you the basic information about the migration like the current progress. It gives you some simple control over throttling and cutover.

For these screenshots in particular, you can see that we are copying with 4x parallelism. This allows us to significantly speed up the migration. This parallelism is configurable via a flag.



Correctness of Ghostferry

- Designed with the aid of formal methods (TLA+)
- Constructed finite model of the algorithm
  - Found and fixed subtle data corruption bug
  - Warning: Finite model != proof of correctness
- What did we gain?
  - Increased confidence of correctness
  - High level formal documentation

shopify

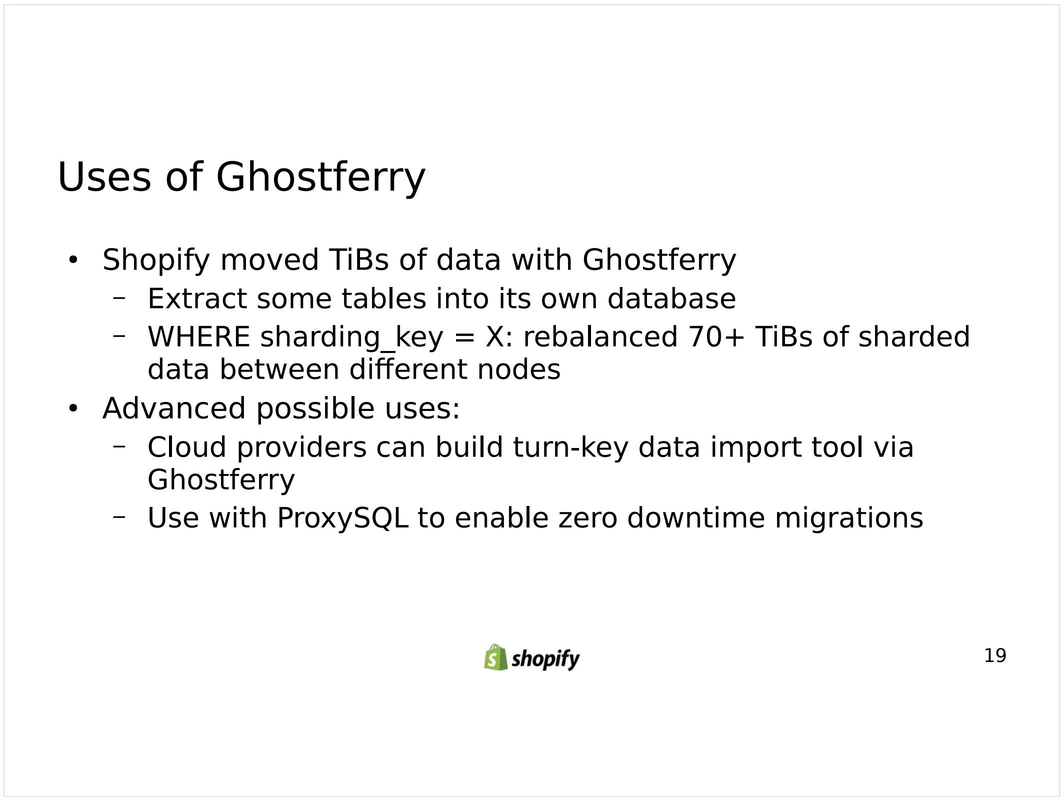
18

Safety is paramount in Ghostferry because data corruption will likely be silent. We used formal methods to **guide** us through the design process. Specifically, we used TLA+, a specification language that allowed us to **model the software** in theory. TLA+ is used by Amazon to validate parts of AWS as well as Microsoft for the design of memory units on the Xbox.

We were able to construct a finite model for the Ghostferry algorithm and verified its correctness within that **finite scope**. Via this model, we found and fixed an order of executing **bug** that can cause rare data corruptions. We also used this model to **verify the requirement** for full-image row based replication for data synchronization and the requirement of SELECT FOR UPDATE during data copy.

I must warn you that even tho we used these methods, there are **no guarantees** that Ghostferry will do the same thing as modeled. The model makes **assumptions** which may not hold true. An extreme example would be that your CPU might have a bug that causes the wrong data to be copied. It would be infeasible to verify that deep. Another word of caution is that we only used a finite model for verification. In the real world there is an **infinite variation** of data so it is possible that the finite model missed something. We **don’t believe we missed something** but some sort of inductive proof of correctness is required but this is not done at this point.

Despite these warnings, TLA+ was still very useful during the development process. Since we were able to find bugs and fix them with the model **before writing a line of code**, we have a high level of confidence that the algorithm is correct as long as the **assumptions hold tru**e. We now also have a formal, high level specification on how Ghostferry is supposed to work. Since the code is similar to the spec, it is easy to see the **intentions** of each component after reading the spec. This could be very useful for new contributors.



Uses of Ghostferry

- Shopify moved TiBs of data with Ghostferry
  - Extract some tables into its own database
  - WHERE sharding_key = X: rebalanced 70+ TiBs of sharded data between different nodes
- Advanced possible uses:
  - Cloud providers can build turn-key data import tool via Ghostferry
  - Use with ProxySQL to enable zero downtime migrations

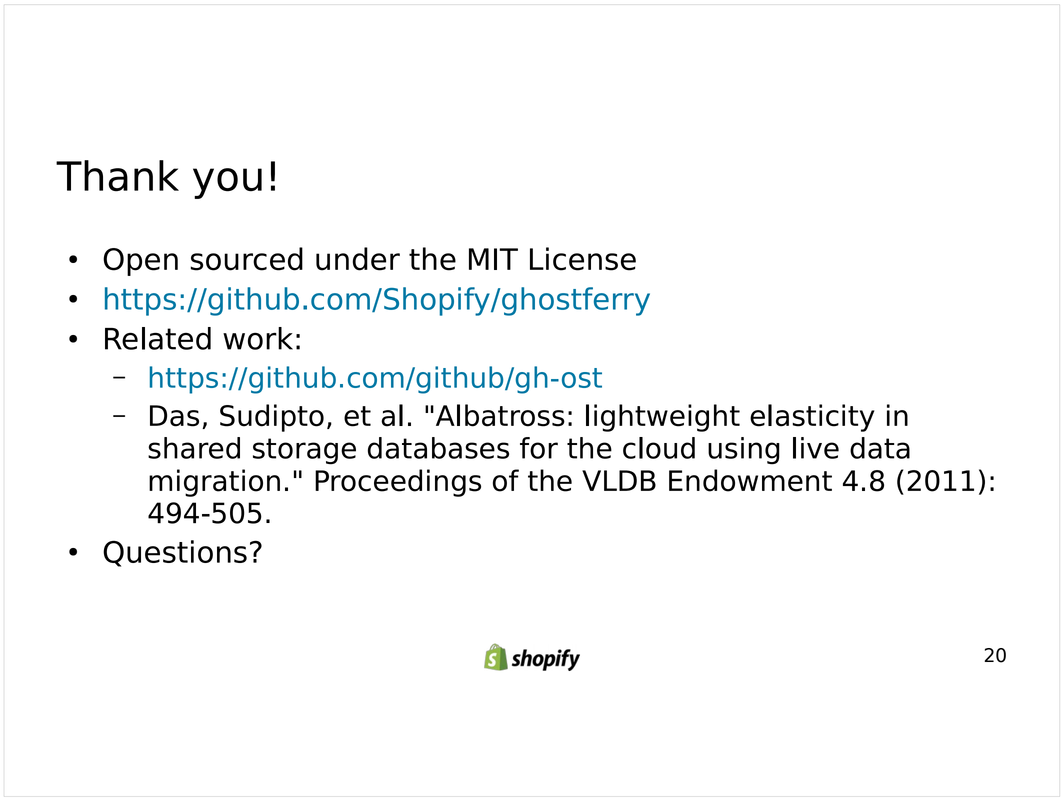
shopify

Shopify has already deployed Ghostferry into production. We have moved terabytes of data with Ghostferry, including more **advanced cases** like extracting some tables into its own database server.

Using Ghostferry's filtering capabilities, we are able to move data with a particular sharding key from one database shard to another using a **custom WHERE clause**. With this method, we have moved over 70 terabytes of sharded data between **different shards**. The application built ontop of Ghostferry that enables this type of move is being **open sourced** with Ghostferry as well. This topic can be a whole talk on its own since it has a lot of moving parts outside of the Ghostferry component. Since most of this is done by **others at Shopify**, I'll defer that talk to them.

Looking beyond what we have done and into more advanced use cases: cloud providers like Google and Amazon could use this technology to build some sort of **push button data import** tool so they can allow their customers to move data into their systems easier. This naturally also lines up well with their **economic interests**.

It's also conceivable to **get rid of all of the downtime** required in the cutover operation by using Ghostferry along with something like ProxySQL. If we go a little further on that train of though, we might even be able to figure out a way to use Ghostferry to **automatically shard** your data based on a sharding key without downtime.



Thank you!

- Open sourced under the MIT License
- https://github.com/Shopify/ghostferry
- Related work:
  - https://github.com/github/gh-ost
  - Das, Sudipto, et al. "Albatross: lightweight elasticity in shared storage databases for the cloud using live data migration." Proceedings of the VLDB Endowment 4.8 (2011): 494-505.
- Questions?

shopify

So the Ghostferry code is currently open sourced under the MIT license on Github.

The tool was originally inspired by Github's gh-ost. The name Ghostferry is to show that root. It has also gained the additional meaning of ferrying the data in the background, without anyone seeing it.

If you want to look at more academic work on this subject, there's a 2011 paper that does something similar referenced here as well.

All of this is would not be possible without the efforts of the production engineering team at Shopify, who spent a lot of work making this system as robust as possible.

That's all I have to say about Ghostferry. Any questions?