



POSTED ON JULY 22, 2021 TO CORE INFRA, DATA INFRASTRUCTURE

# Migrating Facebook to MySQL 8.0



By Herman Lee, Pradeep Nayak



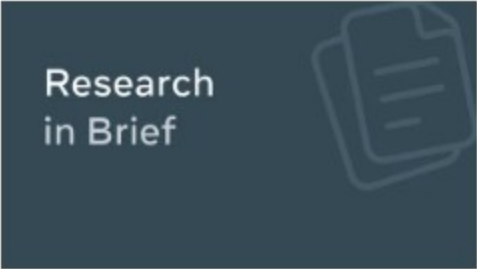
**MySQL**, an open source database developed by Oracle, powers some of Facebook’s most important workloads. We actively develop new features in MySQL to support our evolving requirements. These features change many different areas of MySQL, including client connectors, storage engine, optimizer, and replication. Each new major version of MySQL requires significant time and effort to migrate our workloads. The challenges include:

- Porting our custom features to the new version
- Ensuring replication is compatible between the major versions
- Minimizing changes needed for existing application queries
- Fixing performance regressions that prevent the server from supporting our workloads

Our last major version upgrade, to MySQL 5.6, took more than a year to roll out. When version 5.7 was released, we were still in the midst of developing our LSM-Tree storage engine, **MyRocks**, on version 5.6. Since upgrading to 5.7 while simultaneously building a new storage engine would have significantly slowed the progress on MyRocks, we opted to stay with 5.6 until MyRocks was complete. MySQL 8.0 was announced as we were finishing the rollout of MyRocks to our user database (UDB) service tier.

That version included compelling features like writeset-based parallel replication and a transactional data dictionary that provided atomic DDL support. For us, moving to 8.0 would also bring in the 5.7 features we had missed, including Document Store. Version 5.6 was approaching end of life, and we wanted to stay active within the MySQL community, especially with our work on the MyRocks storage engine. Enhancements in 8.0, like instant DDL, could speed up MyRocks schema changes, but we needed to be on the 8.0 codebase to use it. Given the benefits of the code update, we decided to migrate to 8.0. We’re sharing how we tackled our 8.0 migration project — and some of the surprises we discovered in the process. When we initially scoped out the project, it was clear that moving to 8.0 would be even more difficult than migrating to 5.6 or MyRocks.

## Related Posts



Jun 21, 2021  
**Consolidating Facebook storage infrastructure with Tectonic file system**



Feb 22, 2021  
**FOQS: Scaling a distributed priority queue**



Aug 31, 2016  
**MyRocks: A space- and write-optimized MySQL database**

## Related Positions

**Documentation Engineer / Technical Writer**  
BELLEVUE, US

**Documentation Engineer / Technical Writer**  
BURLINGAME, US

**Documentation Engineer / Technical Writer**  
NEW YORK, US

**Software Engineer, Android**  
LOS ANGELES, US

**Software Engineer, Android**  
REMOTE, US

See All Jobs



- At the time, our customized 5.6 branch had over 1,700 code patches to port to 8.0. As we were porting those changes, new Facebook MySQL features and fixes were added to the 5.6 codebase that moved the goalpost further away.
- We have many MySQL servers running in production, serving a large number of disparate applications. We also have extensive software infrastructure for managing MySQL instances. These applications perform operations like gathering statistics and managing server backups.
- Upgrading from 5.6 to 8.0 skipped over 5.7 entirely. APIs that were active in 5.6 would have been deprecated in 5.7 and possibly removed in 8.0, requiring us to update any application using the now-removed APIs.
- A number of Facebook features were not forward-compatible with similar ones in 8.0 and required a deprecation and migration path forward.
- MyRocks enhancements were needed to run in 8.0, including native partitioning and crash recovery.

## Code patches

We first set up the 8.0 branch for building and testing in our development environments. We then began the long journey to port the patches from our 5.6 branch. There were more than 1,700 patches when we started, but we were able to organize them into a few major categories. Most of our custom code had good comments and descriptions so we could easily determine whether it was still needed by the applications or if it could be dropped. Features that were enabled by special keywords or unique variable names also made it easy to determine relevance because we could search through our application codebases to find their use cases. A few patches were very obscure and required detective work — digging through old design documents, posts, and/or code review comments — to understand their history.

We sorted each patch into one of four buckets:

1. Drop: Features that were no longer used, or had equivalent functionality in 8.0, did not need to be ported.
2. Build/Client: Non-server features that supported our build environment and modified MySQL tools like mysqlbinlog, or added functionality like the async client API, were ported.
3. Non-MyRocks Server: Features in the mysqld server that were not related to our MyRocks storage engine were ported.
4. MyRocks Server: Features that supported the MyRocks storage engine were ported.

We tracked the status and relevant historical information of each patch using spreadsheets, and recorded our reasoning when dropping a patch. Multiple patches that updated the same feature were grouped together for porting. Patches ported and committed to the 8.0 branch were annotated with the 5.6 commit information. Discrepancies on porting status would inevitably arise due to the large number of patches we needed to sift through and these notes helped us resolve them.

Each of the client and server categories naturally became a software release milestone. With all client-related changes ported, we were able to update our client tooling and connector code to 8.0. Once all of the non-MyRocks server features were ported, we were able to deploy 8.0 mysqld for InnoDB servers. Finishing up the MyRocks server features enabled us to update MyRocks installations.

Some of the most complex features required significant changes for 8.0, and a few areas had major compatibility problems. For example, upstream 8.0 binlog event formats were incompatible with some of our custom 5.6 modifications. Error codes used by Facebook 5.6 features conflicted with those assigned to new features by upstream 8.0. We ultimately needed to patch our 5.6 server to be forward-compatible with 8.0.

It took a couple of years to complete porting all of these features. By the time we got to the end, we had evaluated more than 2,300 patches and ported 1,500 of those to 8.0.

## The migration path

We group together multiple mysqld instances into a single MySQL replica set. Each instance in a replica set contains the same data but is geographically distributed to a different data center to provide data availability and failover support. Each replica set has one primary instance. The remaining instances are all secondaries. The primary handles all write traffic and replicates the data asynchronously to all secondaries.

We started with replica sets consisting of 5.6 primary/5.6 secondaries and the end goal was replica sets with 8.0 primary/8.0 secondaries. We followed a plan similar to the [UDB MyRocks migration plan](#).

1. For each replica set, create and add 8.0 secondaries via a logical copy using mysqldump. These secondaries do not serve any application read traffic.
2. Enable read traffic on the 8.0 secondaries.
3. Allow the 8.0 instance to be promoted to primary.
4. Disable the 5.6 instances for read traffic.
5. Remove all the 5.6 instances.

Each replica set could transition through each of the steps above independently and stay on a step as long as needed. We separated replica sets into much smaller groups, which we shepherded through each transition. If we found problems, we could rollback to the previous step. In some cases, replica sets were able to reach the last step before others started.

To automate the transition of a large number of replica sets, we needed to build new software infrastructure. We could group replica sets together and move them through each stage by simply changing a line in a configuration file. Any replica set that encountered problems could then be individually rolled back.

### Row-based replication

As part of the 8.0 migration effort, we decided to standardize on using row-based replication (RBR). Some 8.0 features required RBR, and it simplified our MyRocks porting efforts. While most of our MySQL replica sets were already using RBR, those still running statement-based replication (SBR) could not be easily converted. These replica sets usually had tables without any high cardinality keys. Switching completely to RBR had been a goal, but the long tail of work needed to add primary keys was often prioritized lower than other projects.

Hence, we made RBR a requirement for 8.0. After evaluating and adding primary keys to every table, we switched over the last SBR replica set this year. Using RBR also gave us an alternative solution for resolving an application issue that we encountered when we moved some replica sets to 8.0 primaries, which will be discussed later.

## Automation validation

Most of the 8.0 migration process involved testing and verifying the mysqld server with our automation infrastructure and application queries.

As our MySQL fleet grew, so did the automation infrastructure we use to manage the servers. In order to ensure all of our MySQL automation was compatible with the 8.0 version, we invested in building a test environment, which leveraged test replica sets with virtual machines to verify the behaviors. We wrote integration tests to canary each piece of automation to run on both the 5.6 version and the 8.0 version and verified their correctness. We found several bugs and behavior differences as we went through this exercise.

As each piece of MySQL infrastructure was validated against our 8.0 server, we found and fixed (or worked around) a number of interesting issues:



1. Software that parsed text output from error log, mysqldump output, or server show commands easily broke. Slight changes in the server output often revealed bugs in a tool's parsing logic.
2. The 8.0's default `utf8mb4` collation settings resulted in collation mismatches between our 5.6 and 8.0 instances. 8.0 tables may use the new `utf8mb4_0900` collations even for create statements generated by 5.6's `show create table` because the 5.6 schemas using `utf8mb4_general_ci` do not explicitly specify collation. These table differences often caused problems with replication and schema verification tools.
3. The error codes for certain replication failures changed and we had to fix our automation to handle them correctly.
4. The 8.0 version's data dictionary obsoleted table `.frm` files, but some of our automation used them to detect table schema modifications.
5. We had to update our automation to support the dynamic privs introduced in 8.0.

### Application validation

We wanted the transition for applications to be as transparent as possible, but some application queries hit performance regressions or would fail on 8.0.

For the MyRocks migration, we built a MySQL shadow testing framework that captured production traffic and replayed them to test instances. For each application workload, we constructed test instances on 8.0 and replayed shadow traffic queries to them. We captured and logged the errors returning from the 8.0 server and found some interesting problems. Unfortunately, not all of these problems were found during testing. For example, the transaction deadlock was discovered by applications during the migration. We were able to roll back these applications to 5.6 temporarily while we researched different solutions.

- New reserved keywords were introduced in 8.0 and a few, such as groups and rank, conflicted with popular table column names and aliases used in application queries. These queries did not escape the names via backquotes, leading to parsing errors. Applications using software libraries that automatically escaped the column names in queries did not hit these issues, but not all applications used them. Fixing the problem was simple, but it took time to track down application owners and codebases generating these queries.
- A few REGEXP incompatibilities were also found between 5.6 and 8.0.
- A few applications hit [repeatable-read transaction deadlocks](#) involving `insert ... on duplicate key` queries on InnoDB. 5.6 had a bug which was corrected in 8.0, but the fix increased the likelihood of transaction deadlocks. After analyzing our queries, we were able to resolve them by lowering the isolation level. This option was available to us since we had made the switch to row-based replication.
- Our custom 5.6 Document Store and JSON functions were not compatible with 8.0's. Applications using Document Store needed to convert the document type to text for the migration. For the JSON functions, we added 5.6-compatible versions to the 8.0 server so that applications could migrate to the 8.0 API at a later time.

Our query and performance testing of the 8.0 server uncovered a few problems that needed to be addressed almost immediately.

- We found new mutex contention hotspots around the ACL cache. When a large number of connections were opened simultaneously, they could all block on checking ACLs.
- Similar contention was found with binlog index access when many binlog files are present and high binlog write rates rotate files frequently.
- Several queries involving temp tables were broken. The queries would return unexpected errors or take so long to run that they would time out.

Memory usage compared with 5.6 had increased, especially for our MyRocks instances, because InnoDB in 8.0 must be loaded. The default `performance_schema` settings enabled all instruments and consumed significant memory. We limited the memory usage by only enabling a small number of instruments and making code changes to disable tables that could not be manually turned off. However, not all the increased memory was being allocated by `performance_schema`. We needed to examine and modify various InnoDB internal data structures to reduce the memory



footprint further. This effort brought 8.0’s memory usage down to acceptable levels.

## What’s next

The 8.0 migration has taken a few years so far. We have converted many of our InnoDB replica sets to running entirely on 8.0. Most of the remaining ones are at various stages along the migration path. Now that most of our custom features have been ported to 8.0, updating to Oracle’s minor releases has been comparatively easier and we plan to keep pace with the latest versions.

Skipping a major version like 5.7 introduced problems, which our migration needed to solve.

First, we could not upgrade servers in place and needed to use logical dump and restore to build a new server. However, for very large mysqld instances, this can take many days on a live production server and this fragile process will likely be interrupted before it can complete. For these large instances, we had to modify our backup and restore systems to handle the rebuild.

Second, it is much harder to detect API changes because 5.7 could have provided deprecation warnings to our application clients to fix potential issues. Instead, we needed to run additional shadow tests to find failures before we could migrate the production workloads. Using mysql client software that automatically escaped schema object names helps reduce the number of compatibility issues.

Supporting two major versions within a replica set is hard. Once a replica set promotes its primary to be an 8.0 instance, it is best to disable and remove the 5.6 ones as soon as possible. Application users tend to discover new features that are supported only by 8.0, like utf8mb4\_0900 collations, and using these can break the replication stream between 8.0 and 5.6 instances.

Despite all the hurdles in our migration path, we have already seen the benefits of running 8.0. Some applications have opted for early conversion to 8.0 to utilize features like Document Store and improved datetime support. We have been considering how to support storage engine features like Instant DDL on MyRocks. Overall, the new version greatly expands on what we can do with MySQL @ Facebook.

### Share this:

 Facebook

 Threads


 X


 LinkedIn

 Hacker News

 Email

TAGS: [MYSQL](#)

 Like

 Share

1.6K people like this. Be the first of your friends.



◀ Prev

Fully Sharded Data Parallel:  
faster AI training with fewer  
GPUs

Next ▶

A linear programming approach  
for optimizing features in ML  
models

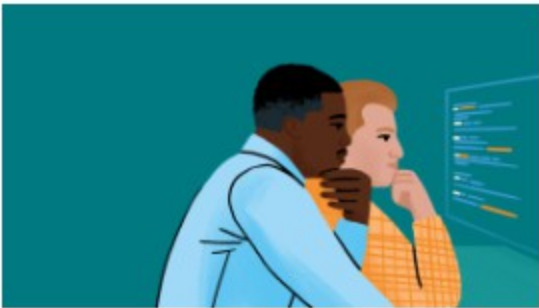


## Read More in Data Infrastructure

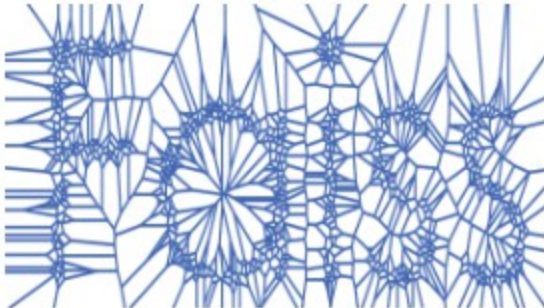
[View All ▶](#)



JUL 22, 2025



MAY 20, 2025



MAY 8, 2025



How Meta keeps its AI hardware reliable

EPISODE 73

Mobile GraphQL at Meta in 2025

Meta Tech Podcast

MAR 31, 2025

Mobile GraphQL at Meta in 2025

Meta's Full-stack HHVM optimizations for GenAI

NOV 19, 2024

Sequence learning: A paradigm shift for personalized ads recommendations

Accelerating GPU indexes in Faiss with NVIDIA cuVS

OCT 15, 2024

OCP Summit 2024: The open future of networking hardware for AI

Available Positions

- Documentation Engineer / Technical Writer  
BELLEVUE, US
- Documentation Engineer / Technical Writer  
BURLINGAME, US
- Documentation Engineer / Technical Writer  
NEW YORK, US
- Software Engineer, Android  
LOS ANGELES, US
- Software Engineer, Android  
REMOTE, US

See All Jobs

Technology at Meta

- Engineering at Meta - X  
Follow
- AI at Meta  
Read
- Meta Quest Blog  
Read
- Meta for Developers  
Read
- Meta Bug Bounty  
Learn more
- RSS  
Subscribe

Open Source

Meta believes in building community through open source technology. Explore our latest projects in Artificial Intelligence, Data Infrastructure, Development Tools, Front End, Languages, Platforms, Security, Virtual Reality, and more.

ANDROID

iOS

WEB

BACKEND

HARDWARE

Learn More

Meta

Engineering at Meta is a technical news resource for engineers interested in how we solve large-scale technical challenges at Meta.

- Home
- Company Info
- Careers