Hyperscaling Aurora MySQL-Compatible to handle sudden traffic growth

AWS Prescriptive Guidance

AWS Prescriptive Guidance: Hyperscaling Aurora MySQL-Compatible to handle sudden traffic growth

Table of Contents

Hyperscaling Aurora MySQL-Compatible to handle sudden traffic growth

Oliver Francis, Shyam Sunder Rakhecha, and Vikram singh Rai, Amazon Web Services (AWS)

October 2022

Your business on the internet might face unprecedented hypergrowth, which your existing application infrastructure is not capable of handling. This can happen in response to various factors, such as an advertisement about your product that results in more than expected interest or a sudden shift in customer buying patterns from in-person to online.

To address these unexpected loads, you have to hyperscale your infrastructure. Scaling the application tier entails adding more servers or pods. But the most difficult piece in achieving hyperscaling is the database. You can scale your database instance to the largest available instance size, but that might not solve your problem.

If you run your database workload on Amazon Aurora MySQL-Compatible Edition, this guide provides recommendations you can implement to hyperscale your database to meet sudden hypergrowth. Not all of these recommendations are long-term best practices. If you expect hypergrowth and have time to plan to address it, see the blog posts that are listed in the *Resources* section. Those posts can help you implement long-term best practices. On the other hand, if you face unexpected hypergrowth, this guide will help you manage your load and achieve reasonable stability while you plan and implement long-term solutions for your business.

Note that the recommendations outlined in this guide will require implementation help from your development team.

Targeted business outcomes

The approaches covered in this guide will help you do the following:

- Stabilize your business in the event of unexpected hypergrowth. Achieve enough stability to implement long-term best practices for hypergrowth.

- Prevent financial loss. Sudden interruptions on a hyperscaled environment can lead to a drop in business transactions that are performed on your application by your customers. This can lead to substantial financial losses in some cases. A stabilized hyperscaled environment is key to preventing long outages that result in a loss of business.

Managing connections

As the demand for your application grows, the front-end traffic increases. In a typical scenario, you set up automatic scaling at the application tier to handle such a burst of incoming traffic. As a result, the application tier starts to auto scale, and more application servers (instances) are added to meet the increase in traffic. Because all application servers have preconfigured database connection pool settings, the number of incoming connections to the database grows in proportion to the newly deployed instances.

For example, 20 application servers configured with 200 database connections each would open a total of 4,000 database connections. If the application pool scales up to 200 instances (for example, during peak hours), the total connection count will reach 40,000. Under a typical workload, most of these connections are likely idle. But the spike in the connections might limit your Amazon Aurora MySQL-Compatible Edition database's ability to scale. This is because even idle connections consume memory and other server resources, such as file descriptors. Aurora MySQL-Compatible typically uses less memory than MySQL Community Edition to maintain same number of connections. However, memory usage for idle connections is still not zero.

Configuration variables

You can control the number of incoming connections allowed to your database with two major server configuration variables: `max_connections` and `max_connect_errors`.

Configuration variable max_connections

The configuration variable `max_connections` limits the number of database connections for each MySQL instance. The best practice is to set it slightly higher than the maximum number of connections you expect to open on each database instance.

If you also enabled `performance_schema`, be extra careful with the `max_connections` setting. The Performance Schema memory structures are sized automatically based on server configuration variables, including `max_connections`. The higher you set the variable, the more memory Performance Schema uses. In extreme cases, this can lead to out-of-memory issues on smaller instance types. Note that enabling Performance Insights will automatically enable Performance Schema.

Configuration variable max_connect_errors

The configuration variable `max_connect_errors` determines how many successive interrupted connection requests are permitted from a given client host. If the client host exceeds the specified number of successive failed connection attempts, the server blocks it. Further connection attempts from that client yield an error.

```
Host 'host_name' is blocked because of many connection errors. Unblock with
'mysqladmin flush-hosts'
```

If you experience "host is blocked" errors, avoid increasing the value of the `max_connect_errors` variable. Instead, investigate the server's diagnostic counters in the `aborted_connects` status variable and the `host_cache` table. Use the collected information to identify and fix clients that run into connection issues. Also, note that this parameter has no effect if `skip_name_resolve` is set to 1 (default).

See the MySQL Reference Manual for details on the following:

- Max_connect_errors variable
- "Host is blocked" error
- Aborted_connects status variable
- Host_cache table

Implement connection pooling

A scaling event might add more application servers, which in-turn might cause the DB server to exceed the fully loaded active connections number. The addition of a connection pool or proxy layer between the application servers and the database acts like a funnel, reducing the total number of connections on the database. A proxy's main purpose is reuse of database connections by means of multiplexing.

On one side, the proxy connects to the database with a controlled number of connections. On the other side, the proxy accepts application connections. It also provides additional features, such as query caching, connection buffering, query rewriting and routing, and load balancing. The connection pool layer needs to be configured to keep the maximum number of connections to the database below the fully loaded number. Amazon RDS Proxy is a fully managed proxy that you can implement for this purpose. Amazon RDS Proxy requires no code changes for most applications, and you don't need to manage any extra infrastructure to implement the solution.

You can also explore the following third-party proxies that can be used with Aurora MySQL-Compatible:

- ProxySQL
- MariaDB MaxScale
- ScaleARC
- Heimdall Data

Avoid connection storms

Consider how your connection pool behaves in the event of an overloaded database or a replica falling too far behind the primary node. When configuring your proxy server or connection pools, ensure that you do not reset the entire connection pool based on slow database responses (caused by underlying hardware or storage issues or DB resource constraints).

Suddenly starting hundreds of connections generates a *connection storm* because a large number of requests for new connections to the database are all initiated at the same time. The storm is resource intensive. Creating a new database connection in MySQL is an expensive operation because the backend exchanges several network packets for the initial handshake, spawns a new process, allocates memory, handles authentication, and so on. If a large number of requests are received in a short period of time, the database can appear to be unresponsive.

MySQL has a mechanism to protect against such a spike in connection requests. The `back_log` variable can be set to the number of requests that can be stacked during a short time before MySQL momentarily stops answering new requests. The value is enforced by a connection handling thread, which itself might get overwhelmed by a connection storm. For more information, see the MySQL Reference Manual.

If your connection is configured to reset when the database is slow, you will be initiating the cycle again and again. Similarly, if you anticipate a sudden increase in database traffic at certain times during the day (for example, when the stock market opens), prewarm your connection pool so that you are not trying to open many connections at the same time that a high traffic load is starting.

Tuning your query workload

A well-tuned workload will take you a long way in achieving a stable solution for hypergrowth. If your workload is not well tuned, no matter the power of the Amazon Aurora cluster that you use, you will encounter bottlenecks that will degrade performance and impact the user experience of your application. The best practice is to have a process in place from the start that helps you identify and tune problematic queries in your system.

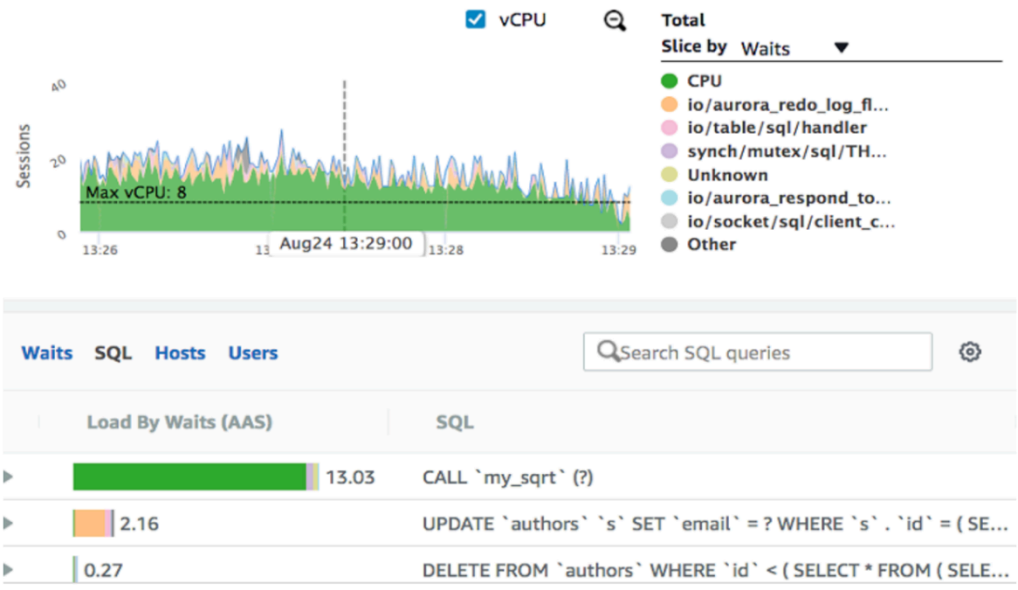
Tracking and tuning problematic queries in your workload

When encountering hypergrowth, having a well-tuned work load is half the battle won. To understand the nature of real-time workloads and performance issues your Aurora cluster is facing, ensure that your team gathers problematic queries from both your writer and your reader instances. These problematic queries need to be tuned for your workload to run at an optimal state. Amazon Aurora MySQL-Compatible Edition provides you with two ways to accomplish this:

- Performance Insights
- Slow query logs

Using Performance Insights

Performance Insights tracks the load on the Aurora writer or reader instance based on the average active sessions (AAS). The AAS value is calculated by using sampling and the number of active sessions that are waiting for a CPU to pick up their query workload and process it. Performance Insights provides a graphical interface where you can check the SQL statements that are causing the highest load by waits for active sessions.



In the previous screenshot, the call to the stored procedure `my_sqrt` is causing an average of 13.03 sessions to wait for their loads to be processed. The logical next step is to tune this procedure. You should identify SQL statements in your readers and writers that are causing load on their respective instance and tune them to improve the performance until the AAS values drop and stay below the Max vCPU dotted line in Performance Insights. If you have hit a ceiling with your tuning efforts and still see the AAS over the Max vCPU line, you can opt for a larger instance class to handle your workload. Do not opt for a larger instance without first trying to tune your query workload, because growing traffic will start exposing the fault lines created by bad queries in your workload.

Using slow query logs and publishing them to CloudWatch

The slow query log is a native MySQL feature and is complementary to Performance Insights. The best practice is to use both these methods to stay ahead of problematic queries that can cause havoc on your instances. The slow query log logs any query that slower than the dynamic variable `long_query_time`. This variable can be set up without any restart to your cluster instances.

To provide flexibility and isolation in the tuning exercise, use separate parameter groups for your writer and reader instances. This is especially important if you use read-write split. Set up a comfortable limit for `long_query_time` in your cluster instances based on your need. As you tune your load, you can aim for aggressive values in the `long_query_time` variable, because you

can set the threshold at the millisecond level. With high concurrency and a well-tuned workload, almost all your queries should run in milliseconds.

When you set Amazon Aurora MySQL-Compatible Edition to log slow queries to a file, Aurora MySQL-Compatible will write the slow query logs to the Aurora MySQL-Compatible file system and retain them for 24 hours. To retain the slow query logs for a longer period for analysis, publish them to Amazon CloudWatch. You can also build a CloudWatch Dashboard to monitor your slow queries. For more information, see the blog post Creating an Amazon CloudWatch dashboard to monitor Amazon RDS and Amazon Aurora MySQL. In addition to analyzing your slow queries on Amazon CloudWatch, you can profile slow query logs by using pt-query-digest, a tool in Percona Toolkit.

You can also choose to automate this process of downloading and profiling queries for higher efficiency in your team. Your team should check for queries that run frequently and for longer intervals, and prioritize tuning them. Aim for a state where very few queries are logged in your slow query log, and you can reduce the `long_query_time` to get more aggressive as you understand and tune your workload.

Guidance for query tuning

After you identify problematic queries in your workload, each query should be tuned. Use the following guidelines on tuning to help your workload to run more efficiently.

Minimize the number of rows scanned

Basic as it seems, this is a great piece of advice to use when you tune queries. Use the EXPLAIN statement, and review the rows column to see how many rows the optimizer scans at each join. Try to reduce the number of rows scanned by creating an optimal index, and then re-explain your query to confirm your work. For more information, see the MySQL documentation.

If you use partitioned tables, always query them with the WHERE clause that enables partition pruning so that the optimizer doesn't have to scan each partition. If your WHERE clause contains a constant for the partitioned column, the optimizer knows which partition to look for, and this makes your query more efficient.

Another facet to this advice is the design of your database. The fewer tables in your query, the faster your query will be. If you can denormalize your database design, you can get the optimizer to scan fewer rows, resulting in faster query performance.

Minimize temporary table usage and temporary tables on disk

The Aurora MySQL-Compatible optimizer creates temporary tables both on RAM and on disk if it cannot get the desired results of your query directly from indexes. Consequently, a large part of tuning is to have the right indexes that serve your workload. However, there might be queries in your workload that cannot rely only on indexes, so some operations might be performed in a temporary file. This is fine as long as you keep these to a minimum, and you ensure that very few tables are created on disk. MySQL creates disk tables when the size of the temporary table is too large to be housed in memory. The logic that MySQL uses to check the size of the internal temporary table is the smaller of the two variable values tmp_table_size and max-heap-table-size. You can tune these variables to an optimal value based on your workload so that in cases where you can't prevent temporary tables, you push them to disk only on rare occasions.

Avoid file sorts

If your workload has a lot of `ORDER BY` queries, the best way to solve them is to use the right indexes on your tables. Ensure your multicolumn indexes are designed well to avoid sorting in files. Sorting can't happen on a column if the preceding columns are not scanned with constants (`in`, `>`, `<`, `!=`, and `BETWEEN` will not allow sorting on the next column to the right). The optimal way to sort in MySQL is to place a multicolumn index that positions columns that contain constant values supplied in the query to the left of the sorting column in a contiguous structure. If in the last resort, your query is unable to return results without a file sort, move the sorting to the application.

Avoid running aggregation queries at high concurrency

Your workload might have a small number of aggregation queries to cater to some functionality within your application. This use case calls for a lot of caution. The InnoDB engine is geared for proper online transaction processing (OLTP) loads, but even a few group-by queries on high concurrency can be very heavy on the CPU and can rapidly degrade the performance of your cluster. To solve use cases where you require aggregated result sets, pre-aggregate the data in ready to read tables so that you can avoid group by queries altogether.

Test your queries for concurrency

When tuning individual queries, remember that these queries run concurrently on several vCPUs in Aurora MySQL-Compatible. Your query might run in a few milliseconds in your test environment on single runs. But this is not the whole picture. Be sure to test your query with the expected level

of concurrency on your production cluster and benchmark its performance. Release the query to production only when it meets your concurrency goals. Ensure that you use the optimizer `hint sql_no_cache` in your test scripts so that you avoid fetching results from the cache. You can use tools such as mysqlslap to perform the test at concurrency and benchmark the results.

Tuning write workloads

Implementing load-balancing and freeing up the writer instance will help your write workload to perform better during high spikes. To achieve better write performance at high concurrency, follow these additional steps.

Move referential integrity into the application layer

Although referential integrity checks are important, with hyperscaling, they could be detrimental to your load. For each write, extra scans must be performed before the write itself is run, and this results in poor performance. If your application requires strict integrity checks, build them into the application layer to prevent them from throttling your writes.

Avoid using heavy primary keys

Keep your primary keys light. The InnoDB storage engine appends the primary key to every other index that you create in your table. When your primary key is large, it affects the size of index. Storage and retrieval of data pages will slow down if the primary key is quite large. A common example is the usage of universally unique identifiers as primary keys. This is not a good approach if you target performance on an hyperscaled environment.

Use partition exchange for loading data into partitioned tables

If you're writing large sets of data into partitioned tables, the combination of LOAD DATA FROM S3 and partition exchange can improve performance, because the main table is not being accessed for the inserts. Partition exchange involves a data definition language (DDL), and it places a metadata lock on your table. Ensure that this is done when there are minimal or no queries running on the table so that the partition exchange DDL can obtain the metadata lock without waits. The exchange itself takes just milliseconds to complete.

Remove unused indexes

InnoDB optimizes its query plans based on the growth of your data, and it's a good idea to check for unused indexes in your Database and remove them. Unused indexes consume IO when the

application writes data into a table. Check the list of unused indexes, and verify that they are not indexes that are used in rare situations, such as quarterly reports. Also, note that some indexes are used to enforce uniqueness or ordering and must also be considered.

Ensure that old row versions are efficiently purged

In the InnoDB implementation of multiversion concurrency control (MVCC), when a record is modified, the current (old) version of the data being modified is first recorded as an *undo record* in an undo log. A growing history list length (HLL) value indicates that the InnoDB garbage collection threads (purge threads) are not keeping up with the write workload, or that purging is blocked by a long-running query or transaction. When garbage collection is blocked or delayed, the database can develop a substantial purge lag that can negatively affect query performance. You can use the following recommendations for optimizing the purge process.

- Keep transactions small.
- For read queries, use the READ COMMITTED isolation level.
- Increase the number of purge threads (innodb_purge_threads and innodb_purge_batch_size). Note that tuning these parameters requires a reboot.
- Monitor the HLL on a regular basis, and resolve any workload issues that prevent garbage collection from progressing.

Ensure logging doesn't cause additional contention

The general query log records client connections and disconnections as well asin addition to all statements received by the server in the order they were received. When activated, logging is synchronous, which can lead to a substantial performance penalty on a busy system. Unless required, we recommend deactivating the general log.

The slow query log records statements that took longer than the long_query_time number of seconds to run, with the default setting of 10 seconds. When the setting is set to 0, all statements are synchronously logged, which can lead to a performance penalty on busy databases.

Shed load

Improving response times and increasing available resources for critical workflows might require getting rid of extraneous load. Many of the solutions covered in this section are trade-off decisions.

They have consequences to the application, and they must be carefully considered. Consider these using these solutions in the following situations:

- You are already on the largest size instances, especially for the primary, writable database instance.
- As a last resort to provide enough headroom in the short term to implement other changes.

Immediate changes include the following:

- Move noncritical read traffic away from the primary DB instance.
- Archive or purge old data.
- Remove referential integrity.
- Disable the binlog (if in use).
- Defer non-critical extract, transform, load (ETL) processes.
- Suspend or degrade nonessential application features.

Before undertaking these actions, evaluate them in the context of long-term business goals and risks.

Separate read and write workloads

A common technique when running applications powered by MySQL is to offload read operations from the writer (primary) database instance and onto one or more read-only database replicas. By offloading reads, you can reduce overall load on the primary database instance and make room for writes. Be sure to target only reads that are not dependent on immediate *read-after-write* consistency for replicas. A more efficient approach is to move all read traffic to the replica and plan for retrying the read-after-write in the event of replication delay. There are independent read workloads that can be offloaded, such as reporting services. Other reads will require changes at the application level, where the context of why the read was issued is well known.

An alternative approach is to implement a database proxy solution as an intermediary between the application and the database, which can provide the function of read-write splitting and query routing, without application awareness. ProxySQL, MariaDB MaxScale, ScaleARC and Heimdall Data are some of the MySQL compatible proxy solutions available. These products offer multiple additional features, such as caching or connection multiplexing. When you are experiencing a sudden surge in traffic and implementing a new technology in your application stack, we

recommend starting with the basic functionality for splitting read/write queries and testing the behavior and performance before using additional features that might have unintended side effects.

Archive or purge older data

Another technique to improve database performance is to offload historical data to another table, database, or Amazon Simple Storage Service (Amazon S3). Many databases retain all the data inline for the whole history of the application. Under normal circumstances in a typical user facing application, this provides users the ability to see all their historical orders. When demand spikes suddenly, many users active on the application probably are new or are focused on placing a new order. If the historical data resides online, in a single table containing billions of rows, this compounds . The table likely also has large indexes. Both table data and indexes are stored in a tree structure. Having more entries in a table requires more levels on the tree, which requires more I/O operations to access the rows. This increases access time to find individual records. More importantly, it causes large unneeded portions of the index to be resident in the database page cache (InnoDB buffer pool).

Archiving older data to a separate table, separate database, or Amazon S3 can reduce access times for end-users, free precious cache, and improve overall application performance. Archiving older data, before the event (for example, retaining only 30 or 90 days) limits the size of the tables and indexes on critical tables. This change requires the application to be modified to query the older data from a secondary location only for the small subset of users who explicitly ask to see historical data.

Defer noncritical ETL processes

Extractions from the main database system for ETL processes can present a stability risk for highly transactional and concurrent workloads during hyperscale conditions. ETL processes are known for the following characteristics:

- Long-running transactions
- Broad locks
- CPU, memory and I/O consumption
- Contention with transactional workloads serving critical end-user path.

ETL processes that are variable or unpredictable (for example, queries such as `INSERT INTO ... SELECT FROM ...;`) adds to the overall variability in load and contention, increasing the stability risk.

Where possible, defer or reduce the frequency at which ETL processes run, especially if they are not providing critical functions. For critical ETL processes, modify them so that they operate in bounded units of work, such as a processing batches of fixed numbers of rows (for example, using `LIMIT` clauses), using separate transactions, or pull smaller amounts of data over an extended period of time to reduce peak resource demands of the DB instance.

Turn off less-critical application features

Gracefully degrading the user experience or removing noncore features when handling a surge of requests conserves database resources for critical functions. This might require a slight change in the customer experience, but a different flow is better than the site being down. Perhaps the personalization on the front page of the site that always does a database call can be temporarily disabled. Or you can stop presenting customized offers to returning customers while selecting products. Temporarily turning off features can enable the page to be cached or delivered without requiring database access.

Other examples include a frontend that has a polling mechanism checking for data changes that require a database call. Reducing the polling frequency will immediately result in fewer database calls. User interfaces that require pagination or provide users the option to retrieve sorted result sets further away from the top result require progressively more expensive database calls. Limit the number of pages in a result set to protect the database from the most expensive database calls. Use feature flags in the application layer to enable the operations team to turn off or degrade features as the application or database load increases.

Parameter tuning

Apart from the connection-related parameters, there are a handful of parameters that you can tune to improve the performance your Amazon Aurora MySQL-Compatible Edition cluster when faced with hypergrowth. When changing any database parameter, it's important to use the following best practices:

- Change one parameter at time so that you can measure the impact of the change.
- Some parameters might require a warmup period to manifest their effect after they are changed. Consider this when you observe and measure the performance of your Aurora MySQL-Compatible cluster.
- Avoid incorrect parameter values, which could prevent the database instance from starting-up.

These parameters include:

- `sync_binlog`
- `table_open_cache`
- `innodb_sync_array_size`
- `innodb_flush_log_at_trx_commit`
- `autocommit`
- `tmp_table_size`
- `max_heap_table_size`

For guidelines on configuring these parameters, see Aurora MySQL configuration parameters, Recommendations for MySQL features in Aurora MySQL, and Temporary table behavior in Aurora MySQL in the AWS documentation.

Resources

- Journey to Adopt Cloud-Native Architecture Series: #1 – Preparing your Applications for Hypergrowth (blog post)
- Journey to Adopt Cloud-Native Architecture Series: #2 – Maximizing System Throughput (blog post)
- Using Amazon RDS Proxy
- Caching strategies
- Turning Performance Insights on and off
- The InnoDB Storage Engine
- Aurora Replicas
- MariaDB Connector/J
- MariaDB MaxScale
- ProxySQL
- Heimdall Data

Document history

The following table describes significant changes to this guide. If you want to be notified about future updates, you can subscribe to an RSS feed.

Change	Description	Date
Initial publication	—	October 5, 2022