Amazon Aurora MySQL Database Administrator's Handbook

Connection Management

First Published January 2018

Updated October 20, 2021

Notices

Customers are responsible for making their own independent assessment of the information in this document. This document: (a) is for informational purposes only, (b) represents current AWS product offerings and practices, which are subject to change without notice, and (c) does not create any commitments or assurances from AWS and its affiliates, suppliers or licensors. AWS products or services are provided “as is” without warranties, representations, or conditions of any kind, whether express or implied. The responsibilities and liabilities of AWS to its customers are controlled by AWS agreements, and this document is not part of, nor does it modify, any agreement between AWS and its customers.

Contents

Abstract

This paper outlines the best practices for managing database connections, setting server connection parameters, and configuring client programs, drivers, and connectors. It's a recommended read for Amazon Aurora MySQL Database Administrators (DBAs) and application developers.

Introduction

Amazon Aurora MySQL (Aurora MySQL) is a managed relational database engine, wire-compatible with MySQL 5.6 and 5.7. Most of the drivers, connectors, and tools that you currently use with MySQL can be used with Aurora MySQL with little or no change.

Aurora MySQL database (DB) clusters provide advanced features such as:

- One primary instance that supports read/write operations and up to 15 Aurora Replicas that support read-only operations. Each of the Replicas can be automatically promoted to the primary role if the current primary instance fails.
- A cluster endpoint that automatically follows the primary instance in case of failover.
- A reader endpoint that includes all Aurora Replicas and is automatically updated when Aurora Replicas are added or removed.
- Ability to create custom DNS endpoints containing a user-configured group of database instances within a single cluster.
- Internal server connection pooling and thread multiplexing for improved scalability.
- Near-instantaneous database restarts and crash recovery.
- Access to near-real-time cluster metadata that enables application developers to build smart drivers, connecting directly to individual instances based on their read/write or read-only role.

Client-side components (applications, drivers, connectors, and proxies) that use sub-optimal configuration might not be able to react to recovery actions and DB cluster topology changes, or the reaction might be delayed. This can contribute to unexpected downtime and performance issues. To prevent that and make the most of Aurora MySQL features, AWS encourages Database Administrators (DBAs) and application developers to implement the best practices outlined in this whitepaper.

DNS endpoints

An Aurora DB cluster consists of one or more instances and a cluster volume that manages the data for those instances. There are two types of instances:

- **Primary instance** – Supports read and write statements. Currently, there can be one primary instance per DB cluster.
- **Aurora Replica** – Supports read-only statements. A DB cluster can have up to 15 Aurora Replicas. The Aurora Replicas can be used for read scaling, and are automatically used as failover targets in case of a primary instance failure.

Amazon Aurora supports the following types of Domain Name System (DNS) endpoints:

- **Cluster endpoint** – Connects you to the primary instance and automatically follows the primary instance in case of failover, that is, when the current primary instance is demoted and one of the Aurora Replicas is promoted in its place.
- **Reader endpoint** – Includes all Aurora Replicas in the DB cluster under a single DNS CNAME. You can use the reader endpoint to implement DNS round robin load balancing for read-only connections.
- **Instance endpoint** – Each instance in the DB cluster has its own individual endpoint. You can use this endpoint to connect directly to a specific instance.
- **Custom endpoints** – User-defined DNS endpoints containing a selected group of instances from a given cluster.

For more information, refer to the Overview of Amazon Aurora page.

Connection handling in Aurora MySQL and MySQL

MySQL Community Edition manages connections in a one-thread-per-connection fashion. This means that each individual user connection receives a dedicated operating system thread in the mysqld process. Issues with this type of connection handling include:

- Relatively high memory use when there is a large number of user connections, even if the connections are completely idle
- Higher internal server contention and context switching overhead when working with thousands of user connections

Aurora MySQL supports a thread pool approach that addresses these issues. You can characterize the thread pool approach as follows:

- It uses thread multiplexing, where a number of worker threads can switch between user sessions (connections). A worker thread is not fixed or dedicated to a single user session. Whenever a connection is not active (for example, is idle, waiting for user input, waiting for I/O, and so on), the worker thread can switch to another connection and do useful work.

  You can think of worker threads as CPU cores in a multi-core system. Even though you only have a few cores, you can easily run hundreds of programs simultaneously because they're not all active at the same time. This highly efficient approach means that Aurora MySQL can handle thousands of concurrent clients with just a handful of worker threads.

- The thread pool automatically scales itself. The Aurora MySQL database process continuously monitors its thread pool state and launches new workers or destroys existing ones as needed. This is transparent to the user and doesn't need any manual configuration.

Server thread pooling reduces the server-side cost of maintaining connections.

However, it doesn't eliminate the cost of setting up these connections in the first place. Opening and closing connections isn't as simple as sending a single TCP packet. For busy workloads with short-lived connections (for example, key-value or online transaction processing (OLTP)), consider using an application-side connection pool.

The following is a network packet trace for a MySQL connection handshake taking place between a client and a MySQL-compatible server located in the same Availability Zone:

```
04:23:29.547316 IP client.32918 > server.mysql: tcp 0
04:23:29.547478 IP server.mysql > client.32918: tcp 0
04:23:29.547496 IP client.32918 > server.mysql: tcp 0
04:23:29.547823 IP server.mysql > client.32918: tcp 78
04:23:29.547839 IP client.32918 > server.mysql: tcp 0
04:23:29.547865 IP client.32918 > server.mysql: tcp 191
04:23:29.547993 IP server.mysql > client.32918: tcp 0
04:23:29.548047 IP server.mysql > client.32918: tcp 11
04:23:29.548091 IP client.32918 > server.mysql: tcp 37
04:23:29.548361 IP server.mysql > client.32918: tcp 99
04:23:29.587272 IP client.32918 > server.mysql: tcp 0
```

This is a packet trace for closing the connection:

```
04:23:37.117523 IP client.32918 > server.mysql: tcp 13
04:23:37.117818 IP server.mysql > client.32918: tcp 56
04:23:37.117842 IP client.32918 > server.mysql: tcp 0
```

As you can see, even the simple act of opening and closing a single connection involves an exchange of several network packets. The connection overhead becomes more pronounced when you consider SQL statements issued by drivers as part of connection setup (for example, `SET variable_name = value` commands used to set session-level configuration). Server-side thread pooling doesn't eliminate this type of overhead.

Common misconceptions

The following are common misconceptions for database connection management.

- **If the server uses connection pooling, you don't need a pool on the application side.** As explained previously, this isn't true for workloads where connections are opened and torn down very frequently, and clients run relatively few statements per connection.

  You might not need a connection pool if your connections are long lived. This means that connection activity time is much longer than the time required to open and close the connection. You can run a packet trace with `tcpdump` and see how many packets you need to open or close connections versus how many packets you need to run your queries within those connections. Even if the connections are long lived, you can still benefit from using a connection pool to protect the database against connection surges, that is, large bursts of new connection attempts.

- **Idle connections don't use memory.** This isn't true because the operating system and the database process both allocate an in-memory descriptor for each user connection. What is typically true is that Aurora MySQL uses less memory than MySQL Community Edition to maintain the same number of connections. However, memory usage for idle connections is still not zero, even with Aurora MySQL.

  The general best practice is to avoid opening significantly more connections than you need.

- **Downtime depends entirely on database stability and database features.** This isn't true because the application design and configuration play an important role in determining how fast user traffic can recover following a database event. For more details, refer to the Best practices section of this whitepaper.

Best practices

The following are best practices for managing database connections and configuring connection drivers and pools.

Using smart drivers

The cluster and reader endpoints abstract the role changes (primary instance promotion and demotion) and topology changes (addition and removal of instances) occurring in the DB cluster. However, DNS updates are not instantaneous. In addition, they can sometimes contribute to a slightly longer delay between the time a database event occurs and the time it's noticed and handled by the application.

Aurora MySQL exposes near-real-time metadata about DB instances in the `INFORMATION_SCHEMA.REPLICA_HOST_STATUS` table.

Here is an example of a query against the metadata table:

```
mysql> select server_id, if(session_id = 'MASTER_SESSION_ID',
'writer', 'reader') as role, replica_lag_in_milliseconds from
information_schema.replica_host_status;

+------------------+--------+-----------------------------+
| server_id        | role   | replica_lag_in_milliseconds |
+------------------+--------+-----------------------------+
| aurora-node-usw2a | writer |                           0 |
| aurora-node-usw2b | reader |         19.253999710083008 |
+------------------+--------+-----------------------------+
2 rows in set (0.00 sec)
```

Notice that the table contains cluster-wide metadata. You can query the table on any instance in the DB cluster.

For the purpose of this whitepaper, a *smart driver* is a database driver or connector with the ability to read DB cluster topology from the metadata table. It can route new connections to individual instance endpoints without relying on high-level cluster

endpoints. A smart driver is also typically capable of load balancing read-only connections across the available Aurora Replicas in a round robin fashion.

The MariaDB Connector/J is an example of a third-party Java Database Connectivity

(JDBC) smart driver with native support for Aurora MySQL DB clusters. Application developers can draw inspiration from the MariaDB driver to build drivers and connectors for languages other than Java.

Refer to the MariaDB Connector/J page for details.

The AWS JDBC Driver for MySQL (preview) is a client driver designed for the high availability of Aurora MySQL. The AWS JDBC Driver for MySQL is drop-in compatible with the MySQL Connector/J driver.

The AWS JDBC Driver for MySQL takes full advantage of the failover capabilities of Aurora MySQL. The AWS JDBC Driver for MySQL fully maintains a cache of the DB cluster topology and each DB instance's role, either primary DB instance or Aurora Replica. It uses this topology to bypass the delays caused by DNS resolution so that a connection to the new primary DB instance is established as fast as possible.

Refer to the AWS JDBC Driver for MySQL GitHub repository for details.

If you're using a smart driver, the recommendations listed in the following sections still apply. A smart driver can automate and abstract certain layers of database connectivity. However, it doesn't automatically configure itself with optimal settings, or automatically make the application resilient to failures. For example, when using a smart driver, you still need to ensure that the connection validation and recycling functions are configured correctly, there's no excessive DNS caching in the underlying system and network layers, transactions are managed correctly, and so on.

It's a good idea to evaluate the use of smart drivers in your setup. Note that if a third-party driver contains Aurora MySQL–specific functionality, it doesn't mean that it has been officially tested, validated, or certified by AWS. Also note that due to the advanced built-in features and higher overall complexity, smart drivers are likely to receive updates and bug fixes more frequently than traditional (bare bones) drivers. You should regularly review the driver's release notes and use the latest available version whenever possible.

DNS caching

Unless you use a smart database driver, you depend on DNS record updates and DNS propagation for failovers, instance scaling, and load balancing across Aurora Replicas. Currently, Aurora DNS zones use a short Time-To-Live (TTL) of five seconds. Ensure that your network and client configurations don't further increase the DNS cache TTL.

Remember that DNS caching can occur anywhere from your network layer, through the operating system, to the application container. For example, Java virtual machines (JVMs) are notorious for caching DNS indefinitely unless configured otherwise. Here are some examples of issues that can occur if you don't follow DNS caching best practices:

- After a new primary instance is promoted during a failover, applications continue to send write traffic to the old instance. Data-modifying statements will fail because that instance is no longer the primary instance.
- After a DB instance is scaled up or down, applications are unable to connect to it. Due to DNS caching, applications continue to use the old IP address of that instance, which is no longer valid.
- Aurora Replicas can experience unequal utilization, for example, one DB instance receiving significantly more traffic than the others.

Connection management and pooling

Always close database connections explicitly instead of relying on the development framework or language destructors to do it. There are situations, especially in container based or code-as-a-service scenarios, when the underlying code container isn't immediately destroyed after the code completes. In such cases, you might experience database connection leaks where connections are left open and continue to hold resources (for example, memory and locks).

If you can't rely on client applications (or interactive clients) to close idle connections, use the server's `wait_timeout` and `interactive_timeout` parameters to configure idle connection timeout. The default timeout value is fairly high at 28,800 seconds (8 hours). You should tune it down to a value that's acceptable in your environment. Refer to the MySQL Reference Manual for details.

Consider using connection pooling to protect the database against connection surges. Also, consider connection pooling if the application opens large numbers of connections (for example, thousands or more per second) and the connections are short lived, that is, the time required for connection setup and teardown is significant compared to the

total connection lifetime. If your development framework or language doesn't support connection pooling, you can use a connection proxy instead.

Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon Relational Database Service (Amazon RDS) that makes applications more scalable, more resilient to database failures, and more secure. ProxySQL, MaxScale, and ScaleArc are examples of third-party proxies compatible with the MySQL protocol.

Refer to the Connection scaling section of this document for more notes on connection pools versus proxies.

By using Amazon RDS Proxy, you can allow your applications to pool and share database connections to improve their ability to scale. Amazon RDS Proxy makes applications more resilient to database failures by automatically connecting to a standby DB instance while preserving application connections.

AWS recommends the following for configuring connection pools and proxies:

- Check and validate connection health when the connection is borrowed from the pool. The validation query can be as simple as **SELECT 1**. However, in Amazon Aurora you can also use connection checks that return a different value depending on whether the instance is a primary instance (read/write) or an Aurora Replica (read-only). For example, you can use the `@@innodb_read_only` variable to determine the instance role. If the variable value is `TRUE`, you're on an Aurora Replica.
- Check and validate connections periodically even when they're not borrowed. It helps detect and clean up broken or unhealthy connections before an application thread attempts to use them.
- Don't let connections remain in the pool indefinitely. Recycle connections by closing and reopening them periodically (for example, every 15 minutes), which frees the resources associated with these connections. It also helps prevent dangerous situations such as runaway queries or zombie connections that clients have abandoned. This recommendation applies to all connections, not just idle ones.

Connection scaling

The most common technique for scaling web service capacity is to add or remove application servers (instances) in response to changes in user traffic. Each application server can use a database connection pool.

This approach causes the total number of database connections to grow proportionally with the number of application instances. For example, 20 application servers configured with 200 database connections each would require 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 web application workload, most of these connections are likely idle. In extreme cases, this can limit database scalability: idle connections do take server resources, and you're opening significantly more of them than you need. Also, the total number of connections is not easy to control because it's not something you configure directly, but rather depends on the number of application servers.

You have two options in this situation:

- Tune the connection pools on application instances. Reduce the number of connections in the pool to the acceptable minimum. This can be a stop-gap solution, but it might not be a long-term solution as your application server fleet continues to grow.
- Introduce a connection proxy between the database and the application. On one side, the proxy connects to the database with a fixed number of connections. On the other side, the proxy accepts application connections and can provide additional features such as query caching, connection buffering, query rewriting/routing, and load balancing.

  Connection proxies

- Amazon RDS Proxy is a fully managed, highly available database proxy for Amazon RDS that makes applications more scalable, more resilient to database failures, and more secure. Amazon RDS Proxy reduces the memory and CPU overhead for connection management on the database.
- Using Amazon RDS Proxy, you can handle unpredictable surges in database traffic that otherwise might cause issues due to oversubscribing connections or creating new connections at a fast rate. To protect the database against oversubscription, you can control the number of database connections that are created.

- Each RDS proxy performs connection pooling for the writer instance of its associated Amazon RDS or Aurora database. Connection pooling is an optimization that reduces the overhead associated with opening and closing connections and with keeping many connections open simultaneously. This overhead includes memory needed to handle each new connection. It also involves CPU overhead to close each connection and open a new one, such as Transport Layer Security/Secure Sockets Layer (TLS/SSL) handshaking, authentication, negotiating capabilities, and so on. Connection pooling simplifies your application logic. You don't need to write application code to minimize the number of simultaneous open connections. Connection pooling also cuts down on the amount of time a user must wait to establish a connection to the database.
- To perform load balancing for read-intensive workloads, you can create a read-only endpoint for RDS proxy. That endpoint passes connections to the reader endpoint of the cluster. That way, your proxy connections can take advantage of Aurora read scalability.
- ProxySQL, MaxScale, and ScaleArc are examples of third-party proxies compatible with the MySQL protocol. For even greater scalability and availability, you can use multiple proxy instances behind a single DNS endpoint.

Transaction management and autocommit

With autocommit enabled, each SQL statement runs within its own transaction. When the statement ends, the transaction ends as well. Between statements, the client connection is not *in transaction*. If you need a transaction to remain open for more than one statement, you explicitly begin the transaction, run the statements, and then commit or roll back the transaction.

With autocommit disabled, the connection is always *in transaction*. You can commit or roll back the current transaction, at which point the server immediately opens a new one.

Refer to the MySQL Reference Manual for details.

Running with autocommit disabled is not recommended because it encourages long-running transactions where they're not needed. Open transactions block a server's internal garbage collection mechanisms, which are essential to maintaining optimal performance. In extreme cases, garbage collection backlog leads to excessive storage consumption, elevated CPU utilization, and query slowness.

Recommendations:

- Always run with autocommit mode enabled. Set the `autocommit` parameter to **1** on the database side (which is the default) and on the application side (which might not be the default).
- Always double-check the autocommit settings on the application side. For example, Python drivers such as MySQLdb and PyMySQL disable autocommit by default.
- Manage transactions explicitly by using `BEGIN/START TRANSACTION` and `COMMIT/ROLLBACK` statements. You should start transactions when you need them and commit as soon as the transactional work is done.

Note that these recommendations are not specific to Aurora MySQL. They apply to MySQL and other databases that use the InnoDB storage engine.

Long transactions and garbage collection backlog are easy to monitor:

- You can obtain the metadata of currently running transactions from the `INFORMATION_SCHEMA.INNODB_TRX` table. The `TRX_STARTED` column contains the transaction start time, and you can use it to calculate transaction age. A transaction is worth investigating if it has been running for several minutes or more. Refer to the MySQL Reference Manual for details about the table.
- You can read the size of the garbage collection backlog from the InnoDB's `trx_rseg_history_len` counter in the `INFORMATION_SCHEMA.INNODB_METRICS` table. Refer to the MySQL Reference Manual for details about the table. The larger the counter value is, the more severe the impact might be in terms of query performance, CPU usage, and storage consumption. Values in the range of tens of thousands indicate that the garbage collection is somewhat delayed. Values in the range of millions or tens of millions might be dangerous and should be investigated.

> **Note** – In Amazon Aurora, all DB instances use the same storage volume, which means that the garbage collection is cluster-wide and not specific to each instance. Consequently, a runaway transaction on one instance can impact all instances. Therefore, you should monitor long transactions on all DB instances.

Connection handshakes

A lot of work can happen behind the scenes when an application connector or a graphical user interface (GUI) tool opens a new database session. Drivers and client tools commonly run series of statements to set up session configuration (for example, `SET SESSION variable = value`). This increases the cost of creating new connections and delays when your application can start issuing queries.

The cost of connection handshakes becomes even more important if your applications are very sensitive to latency. OLTP or key-value workloads that expect single-digit millisecond latency can be visibly impacted if each connection is expensive to open. For example, if the driver runs six statements to set up a connection and each statement takes just one millisecond to run, your application will be delayed by six milliseconds before it issues its first query.

Recommendations:

- Use the Aurora MySQL Advanced Audit, the General Query Log, or network-level packet traces (for example, with `tcpdump`) to obtain a record of statements run during a connection handshake. Whether or not you're experiencing connection or latency issues, you should be familiar with the internal operations of your database driver.
- For each handshake statement, you should be able to explain its purpose and describe its impact on queries you'll subsequently run on that connection.
- Each handshake statement requires at least one network roundtrip and will contribute to higher overall session latency. If the number of handshake statements appears to be significant relative to the number of statements doing actual work, determine if you can disable any of the handshake statements. Consider using connection pooling to reduce the number of connection handshakes.

Load balancing with the reader endpoint

Because the reader endpoint contains all Aurora Replicas, it can provide DNS-based, round robin load balancing for new connections. Every time you resolve the reader endpoint, you'll get an instance IP that you can connect to, chosen in round robin fashion.

DNS load balancing works at the connection level (not the individual query level). You must keep resolving the endpoint without caching DNS to get a different instance IP on each resolution. If you only resolve the endpoint once and then keep the connection in

your pool, every query on that connection goes to the same instance. If you cache DNS, you receive the same instance IP each time you resolve the endpoint.

You can use Amazon RDS Proxy to create additional read-only endpoints for an Aurora cluster. These endpoints perform the same kind of load-balancing as the Aurora reader endpoint. Applications can reconnect more quickly to the proxy endpoints than the Aurora reader endpoint if reader instances become unavailable.

If you don't follow best practices, these are examples of issues that can occur:

- Unequal use of Aurora Replicas, for example, one of the Aurora Replicas is receiving most or all of the traffic while the other Aurora Replicas sit idle.
- After you add or scale an Aurora Replica, it doesn't receive traffic or it begins to receive traffic after an unexpectedly long delay.
- After you remove an Aurora Replica, applications continue to send traffic to that instance.

For more information, refer to the DNS endpoints and DNS caching sections of this document.

Designing for fault tolerance and quick recovery

In large-scale database operations, you're statistically more likely to experience issues such as connection interruptions or hardware failures. You must also take operational actions more frequently, such as scaling, adding, or removing DB instances and performing software upgrades.

The only scalable way of addressing this challenge is to assume that issues and changes will occur and design your applications accordingly.

Examples:

- If Aurora MySQL detects that the primary instance has failed, it can promote a new primary instance and fail over to it, which typically happens within 30 seconds. Your application should be designed to recognize the change quickly and without manual intervention.
- If you create additional Aurora Replicas in an Aurora DB cluster, your application should automatically recognize the new Aurora Replicas and send traffic to them.
- If you remove instances from a DB cluster, your application should not try to connect to them.

Test your applications extensively and prepare a list of assumptions about how the application should react to database events. Then, experimentally validate the assumptions.

If you don't follow best practices, database events (for example, failovers, scaling, and software upgrades) might result in longer than expected downtime. For example, you might notice that a failover took 30 seconds (per the DB cluster's event notifications) but the application remained down for much longer.

Server configuration

There are two major server configuration variables worth mentioning in the context of this whitepaper: `max_connections` and `max_connect_errors`.

Configuration variable max_connections

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

If you also enabled `performance_schema`, be extra careful with the 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 for T2 and T3 instance families

Using Performance Schema on T2 and T3 Aurora DB instances with less than 8 GB of memory isn't recommended. To reduce the risk of out-of-memory issues on T2 and T3 instances:

- Don't enable Performance Schema.
- If you must use Performance Schema, leave `max_connections` at the default value.
- Disable Performance Schema if you plan to increase `max_connections` to a value significantly greater than the default value.

Refer to the MySQL Reference Manual for details about the `max_connections` variable.

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 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'
```

A common (but incorrect) practice is to set the parameter to a very high value to avoid client connectivity issues. This practice isn't recommended because it:

- Allows application owners to tolerate connection problems rather than identify and resolve the underlying cause. Connection issues can impact your application health, so they should be resolved rather than ignored.
- Can hide real threats, for example, someone actively trying to break into the server.

If you experience "host is blocked" errors, increasing the value of the `max_connect_errors` variable isn't the correct response. Instead, investigate the server's diagnostic counters in the `aborted_connects` status variable and the `host_cache` table. Then use the 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).

Refer to the MySQL Reference Manual for details on the following:

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

Conclusion

Understanding and implementing connection management best practices is critical to achieve scalability, reduce downtime, and ensure smooth integration between the application and database layers. You can apply most of the recommendations provided in this whitepaper with little to no engineering effort.

The guidance provided in this whitepaper should help you introduce improvements in your current and future application deployments using Aurora MySQL DB clusters.

Contributors

Contributors to this document include:

- Szymon Komendera, Database Engineer, Amazon Aurora
- Samuel Selvan, Database Specialist Solutions Architect, Amazon Web Services

Further reading

For additional information, refer to:

- Aurora on Amazon RDS User Guide
- Communication Errors and Aborted Connections in MySQL Reference Manual

Document revisions

Date	Description
October 20, 2021	Minor content updates to follow new style guide and hyperlinks.
July 2021	Minor content updates to the following topics: Smart Drivers, Connection Management and Pooling, and Connection Scaling.
March 2019	Minor content updates to the following topics: Introduction, DNS Endpoints, and Server Configuration.
January 2018	First publication.