MariaDB Platform is an enterprise open source database for transactional, analytical or hybrid transactional/analytical processing at scale. By preserving historical data and optimizing for real-time analytics while continuing to process transactions, MariaDB Platform provides businesses with the means to create competitive advantages and monetize data – everything from providing data-driven customers with actionable insight to empowering them with self-service analytics.

MariaDB Server
MariaDB Server is the foundation of the MariaDB Platform. It is the only open source database with the same enterprise features found in proprietary databases, including Oracle Database compatibility (e.g., PL/SQL), temporal tables, sharding, point-in-time rollback and transparent data encryption.

MariaDB ColumnStore
MariaDB ColumnStore extends MariaDB Server with distributed, columnar storage and massively parallel processing for ad hoc, interactive analytics on hundreds of billions of rows via standard SQL – with no need to create and maintain indexes, and with 10% of the disk space using high compression.

MariaDB MaxScale
MariaDB MaxScale provides MariaDB Platform with a set of services for modern applications, including transparent query routing and change-data-capture for hybrid transactional/analytical workloads, high availability (e.g., automatic failover) and advanced security (e.g., data masking).
TABLE OF CONTENTS

1 INTRODUCTION
2 CONCEPTS
2 TERMINOLOGY
3 HIGH AVAILABILITY
3 MASTER/SLAVE REPLICATION
6 MULTI-MASTER CLUSTERING
7 TRANSPARENT RECOVERY
9 READ SCALABILITY
9 MASTER/SLAVE REPLICATION
10 CLUSTERING
12 ADVANCED TOPOLOGIES
12 MULTIPLE DATA CENTERS
13 READ SCALABILITY
14 DEDICATED BACKUP
15 INTERNALS
15 MASTER/SLAVE REPLICATION
17 MULTI-MASTER CLUSTERING
19 PERFORMANCE OPTIMIZATIONS
19 REPPLICATION
20 CLUSTERING
21 CONCLUSION
INTRODUCTION

Today, companies are undergoing a digital transformation: offline operations are becoming online operations, enterprise applications are becoming customer-facing applications, and engagement is happening anywhere and everywhere via web, mobile and Internet of Things (IoT) applications – and when it comes to customer experience, availability is not a preference, it is a requirement.

It is all about the data. It has to be available 24 hours a day, 7 days a week, 365 days a year. However, because infrastructure can and will fail, the database platform must maintain, or automatically restore, availability in the event of a failure – server, database or network.

MariaDB Platform uses local storage and replication (with or without clustering) to provide high availability via multiple database servers. There is no single point of failure (SPOF). In fact, when MariaDB Platform is configured for high availability, downtime due to an unplanned infrastructure failure is all but removed.

However, when it comes to high availability, the trade-offs between performance, durability and consistency have to be considered. There are times when durability and consistency is more important than performance. There are times when performance is more important. The right trade-offs depends on the business needs, the use case and the technical requirements.

This guide explains how replication and clustering work in MariaDB Platform, how system variables and parameters can be set to improve performance, durability and/or consistency, important considerations when choosing between asynchronous, semi-synchronous and synchronous replication, and what the failover process looks like for different replication and clustering topologies.

This guide will detail:

- How master/slave replication and multi-master clustering work
- The trade-offs between asynchronous, semi-synchronous and synchronous replication
- The impact of high availability on performance, durability and consistency
- How topology detection with automatic failover improves availability
Concepts

**Mean time between failures (MTBF)** is a measure of reliability. It is the average elapsed time between failures (e.g., the time between database crashes). The longer, the better.

**Mean time to recovery (MTTR)** is a measure of maintainability. It is the average elapsed time between failure and recovery (e.g., the duration of a database failover). The shorter, the better.

Calculating availability
If the MTBF is 12 months (525,601 minutes) and the MTTR is 5 minutes, the database is available 99.999% of the time.

\[
\text{Availability} = \frac{\text{MTBF}}{\text{MTBF} + \text{MTTR}}
\]

\[
\frac{525601}{525601 + 5} = 99.999\% \text{ availability}
\]

Terminology

**Switchover** is when the active database becomes a standby database, and a standby database becomes the active database, often in the case of planned maintenance.

**Failover** is when a standby database becomes the active database because the active database has failed or become unresponsive/unreachable.

**Failback** is when a failed active database is recovered and becomes the active database with the temporary active database reverting back to a standby database.

**Shared-nothing architecture** is when there are no shared sources (e.g., CPU, disk and/or memory) between database servers – there is no single point of failure.

**Share-everything architecture** is when disk and memory is shared between database servers (e.g., Oracle RAC).

**Split-brain syndrome** is when database servers in a single cluster form separate clusters, and each cluster is unaware of the other and continues to function as if it was the only cluster.
HIGH AVAILABILITY

MariaDB Platform provides high availability with multi-master clustering (synchronous) or master/slave replication (asynchronous or semi-synchronous) with automatic failover.

Master/slave Replication

The master assigns transactions a global transaction ID (GTID) and writes them to its binary log. The slaves request the next transaction from the master by sending their current GTID, writes it their relay log and executes it.
Automatic failover

The database proxy, MariaDB MaxScale, will perform an automatic failover if the master fails by promoting the most up-to-date slave (i.e., the one with the highest GTID) to master and reconfiguring the remaining slaves to replicate from it. In addition, if automatic rejoin is enabled, the failed master will be reconfigured as a slave if it is recovered.

Asynchronous replication

With asynchronous replication, transactions are replicated after being committed. The master does not wait for any of the slaves to acknowledge the transaction before committing it. It does not affect write performance. However, if the master fails and automatic failover is enabled, there will be data loss if one or more transactions have not been replicated to a slave.

In the example below, the database proxy would promote Slave 1 to master because it is the slave with the highest GTID. However, the most recent transaction (GTID = 3) had not been replicated before the master failed. There would be data loss.
Asynchronous replication is recommended for read-intensive workloads or mixed/write-intensive workloads where the highest write performance is required.

Examples:
- **Read-intensive workloads**: product catalogs, reporting
- **Mixed workloads**: shopping carts, customer reviews
- **Write-intensive workloads**: clickstream data, sensor data

**Semi-synchronous replication**

With semi-synchronous replication, a transaction is not committed until it has been replicated to a slave. It affects write performance, but the effect is minimized by waiting for transactions to replicate to one slave rather than every slave. However, if the master fails and automatic failover is enabled, there will be no data loss because every transaction has been replicated to a slave.

In the example below, the database proxy would promote Slave 2 to master because it is the slave with the highest GTID. With semi-synchronous replication, there would be no data loss because at least one of the slaves will have every transaction written to its relay log.

![Diagram 5: the role of GTID in automatic failover with semi-synchronous replication](image)

**Note**

The master will wait for up to 10 seconds (default) for a transaction to be replicated to a slave before it reverts to asynchronous replication. If it does, and one of the slaves catches up, the master will restore semi-synchronous replication.

If all of the slaves are slow, the timeout can be reduced to maintain write performance (but with less durability), or increased to maintain durability (but with less write performance).
### System variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>rpl_semi_sync_master_enabled</td>
<td>0 (OFF)</td>
<td>1 (ON)</td>
</tr>
<tr>
<td>rpl_semi_sync_master_timeout</td>
<td>0 to n (ms, max: 18446744073709551615)</td>
<td>10000</td>
</tr>
</tbody>
</table>

Semi-synchronous replication is recommended for mixed/write-intensive workloads where high write performance and strong durability are required.

Examples:
- **Mixed workloads: inventory**

## Multi-master Clustering

MariaDB Platform supports multi-master clustering via MariaDB Cluster (i.e., Galera Cluster). The originating node assigns a transaction a GTID, and during the commit phase, sends all of the rows modified by it (i.e., writes) to every node within the cluster, including itself. If the writes are accepted by every node within the cluster, the originating node applies the writes and commits the transaction. The other nodes will apply the writes and commit the transaction asynchronously.

![Diagram of the multi-master clustering process (synchronous replication)](image)

**Automatic failover**

If there is a node failure, the cluster will automatically remove it and the database proxy, MariaDB MaxScale, will stop routing queries to it. If the database proxy was routing reads and writes to the failed node, and because every node can accept reads and writes, the database proxy will select a different node and begin routing reads and writes to it.
Synchronous replication

With synchronous replication, a transaction is not committed until its changes (i.e., modified rows) have been replicated to every node within the cluster. The write performance is limited by the slowest node within the cluster. However, if the node a write was routed to fails, there will be no data loss because the changes for every transaction will have been replicated to every node within the cluster.

In the example below, the database proxy would be routing reads and writes to Node 2 because it has the lowest priority value of the remaining nodes. There would be no data loss because with synchronous replication, every node has the changes of every transaction.

Note

The database proxy can be configured so automatic failover is deterministic (e.g., based on priority value) by setting the use_priority parameter to “true” in the Galera Cluster monitor configuration and the priority parameter in the database server configurations.

Synchronous replication is recommended for mixed/write-intensive workloads where the strongest durability is required.

Examples:

- **Mixed workloads**: customer profiles
- **Write-intensive workloads**: payments

Transparent Recovery

The automatic failover process ensures the database is always available, but it is not transparent to applications because they have to create new connections and retry queries/transactions if the master fails. However, the database proxy, MariaDB MaxScale, now includes advanced options to prevent applications from being interrupted by a failed database server with or without automatic failover.
Connection migration
When a database server fails or otherwise becomes unreachable/unavailable, client connections to it are closed by the database proxy, and applications must create new ones. However, the database proxy can now be configured to migrate connections to the master promoted by automatic failover, so applications no longer have to create a new connection when a database server fails.

Session migration
The database proxy caches session commands, allowing it to migrate sessions by recreating them on a different database server. For example, if a session is created on a slave, and the slave fails, the database proxy will replay the session commands on a different slave so the application is not interrupted. When combined with connection migration, connections and sessions are automatically migrated when a database server fails – and it is transparent to applications.

Delayed retry
If an application sends a write query to the database after the master has failed, and before automatic failover has completed, the database proxy will return an error. However, the database proxy can now be configured to wait and automatically retry the query. In effect, it waits for automatic failover to complete and then retries the query on behalf of the application, so it no longer has to retry queries after a database server has failed.

Note
The database proxy can be configured to route read queries to slaves when there is no master, so if the master has failed and automatic failover has not completed, read queries can still be executed.

Transaction replay
Further, if an automatic failover occurs after a transaction has started but before it has finished (i.e., an in-flight transaction), the database proxy can be configured to replay the transaction from the beginning on the new master. For example, if there were five inserts in a transaction and the master failed after the first two, the database proxy would wait for the automatic failover process to complete and then replay the first two inserts to the new master, allowing the transaction to continue.

Parameters (MariaDB MaxScale)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>master_reconnection</td>
<td>True (enabled)</td>
<td>False (disabled)</td>
</tr>
<tr>
<td></td>
<td>fail_instantly (close the connection)</td>
<td></td>
</tr>
<tr>
<td>master_failure_mode</td>
<td>fail_on_write (close the connection if a write query is received)</td>
<td>fail_instantly</td>
</tr>
<tr>
<td></td>
<td>error_on_write (return an error if a write query is received)</td>
<td></td>
</tr>
<tr>
<td>delayed_retry</td>
<td>True (enabled)</td>
<td>False (disabled)</td>
</tr>
<tr>
<td>delayed_retry_timeout</td>
<td>N (How long to wait before return an error)</td>
<td>10</td>
</tr>
<tr>
<td>transaction_replay</td>
<td>True (enabled)</td>
<td>False (disabled)</td>
</tr>
</tbody>
</table>
READ SCALABILITY

Master/slave Replication

If master/slave replication is used for both high availability and read scalability, the database proxy, MariaDB MaxScale, can route writes to the master and load balance reads across the slaves using read-write splitting.

However, consistency requirements should be considered. With master/slave replication (asynchronous or semi-synchronous), querying slaves can return stale reads if there are transactions pending replication – and after transactions have been replicated and persisted to the relay log on one or more slaves, they still have to be executed.

If consistent reads are required, the consistent critical reads (CCR) filter in the database proxy should be enabled. If there is a write, the CCR filter will be triggered, causing the database proxy to route all subsequent reads to the master for 60 seconds (default). If the replication lag is greater than 60 seconds (i.e., it’s taking longer than 60 seconds to replicate writes to slaves), the time can be increased.

Tip
The CCR filter can be configured so it is triggered by specific writes rather than all writes.
### Parameters

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>time</td>
<td>The length of time to route reads to the master (seconds)</td>
<td>60</td>
</tr>
<tr>
<td>count</td>
<td>The number of reads to route to the master</td>
<td>-</td>
</tr>
<tr>
<td>match</td>
<td>A regular expression to determine if a write should trigger routing</td>
<td>-</td>
</tr>
<tr>
<td>ignore</td>
<td>A regular expression to determine if a write should not trigger routing</td>
<td>-</td>
</tr>
</tbody>
</table>

Alternatively, causal reads can be enabled in the database proxy. When causal reads are enabled, the database proxy will modify the query, forcing the database to wait until it has caught up to the client before executing the query. It takes advantage of Global Transaction IDs (GTIDs). For example, if the GTID of the client’s most recent write was 2000, the database would wait until its GTID reached or exceeded 2000 before executing the query.

**Note**
The context for causal reads is clients, not applications.

### Parameters (MariaDB MaxScale)

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>causal_reads</td>
<td>True (enabled)</td>
<td>False (disabled)</td>
</tr>
<tr>
<td>causal_reads_timeout</td>
<td>N (The number of seconds to wait before routing the master)</td>
<td>10</td>
</tr>
</tbody>
</table>

### Clustering

If multi-master clustering is used for both high availability and read scalability, the database proxy can assign the role of master to a single node (and route writes to it) while assigning the role of slave to the remaining nodes (and load balancing reads across them). In multi-master clusters, routing all writes to the same node prevents deadlocks and write conflicts.
It is possible for nodes with the slave role to return stale reads because while write sets (i.e., transactions) are replicated synchronously, they are applied asynchronously. There is a small delay between receiving a write set and applying it.

However, stale reads can be prevented by setting the `wsrep_sync_wait` system variable to 1, forcing nodes to delay a read until all write sets have been applied. The impact on read latency is minimal, and ensures any data written to a node with the master role can immediately be read from any node with the slave role.

**Note**
The `wsrep_sync_wait` system variable can be set to specify the types of queries to delay (e.g., READ) until all write sets have been applied.

### System variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>wsrep_sync_wait</td>
<td>0 (DISABLED)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>1 (READ)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 (UPDATE and DELETE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 (READ, UPDATE and DELETE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 (INSERT and REPLACE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>5 (READ, INSERT and REPLACE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>6 (UPDATE, DELETE, INSERT and REPLACE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>7 (READ, UPDATE, DELETE, INSERT and REPLACE)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>8 (SHOW), 9-15 (1-7 + SHOW)</td>
<td></td>
</tr>
</tbody>
</table>
ADVANCED TOPOLOGIES

Multiple Data Centers

**Master/slave replication**

In the example below, circular replication (e.g., bidirectional replication) can be used to synchronize data between an active data center (DC1) and a passive data center (DC2). The master in DC1 is configured as a slave to the master in DC2. The master in DC2 is configured as a slave to the master in DC1. If DC1 fails, applications can connect to the database proxy, MariaDB MaxScale, in DC2.

![Diagram 10: bidirectional, asynchronous replication between data centers](image-url)
**Clustering**

In the example below, a three-node cluster is deployed across two data centers with two nodes in the active data center (DC1) and one node in the passive data center (DC2). The configuration for the database proxy in DC1, Proxy 1, assigns a priority value of 1 to Node 1, 2 to Node 2 and 3 to Node 3. It assigns the role of master to Node 1 because Node 1 has the lowest priority value. Proxy 1 uses a basic router to route all reads and writes to any node assigned the master role, Node 1 by default. If Node 1 fails, Proxy 1 will assign the role of master to Node 2 - in the same data center, DC1.

If DC1 fails, applications can connect to the database proxy in DC2, Proxy 2. The configuration for Proxy 2 assigns a priority value of 1 to Node 3, 2 to Node 2 and 3 to Node 1. It assigns the role of master to Node 3 because it has the lowest priority value. Proxy 2 uses a basic router to route all reads and writes to any node assigned the master role, Node 3 by default.

**Note**

A cluster can be deployed across multiple data centers if a) there is enough network bandwidth between them to minimize latency and b) the write sets are small (i.e., transactions to not change a lot of rows).

---

**Read Scalability**

**Master/slave replication**

In the example below, a second database proxy is configured and deployed as a binlog server to relay transactions from the master to many slaves for read scaling. The binlog server reduces the replication overhead on the master – instead of many slaves replicating from the master, a single binlog server replicates from it.

The master is configured for semi-synchronous replication, for high availability and durability, with Slave M1 and Slave M2, and for asynchronous replication, for read scalability, with the binlog server. The database proxy is configured with two routers, one for each cluster, with each router having a different port. The first will route all writes to the master in Cluster 1. The second will route all reads to the slaves in Cluster 2 (Slave R2 to Slave R100).
**Note**
The application has to use different connections for port-based read-write splitting.

---

**Dedicated Backup**

**Master/slave replication**

In the example below, one of the slaves is dedicated to backups – the database proxy does not route reads to it. However, DBAs run MariaDB Backup on it to create backups. By dedicating one of the slaves to backups and backups only, the workload on the master and remaining slaves will not be interrupted when a backup is created.

The master is configured for semi-synchronous replication with Slave 2 and Slave 3 for high availability and durability, and asynchronous replication with Slave 1 for backups.
**INTERNALS**

**Master/slave Replication**

**Components**

**Binary log (binlog)**

The master logs changes made by DML and DDL statements, and their execution time, as events in its binlog. The binlog is comprised of log files and an index file. While the binlog events are stored in a binary format, `mysqlbinlog` can display them as text.

**Group commits**

By default, the master calls `fsync()` to flush binlog writes to disk during the commit phase of a transaction. However, when there are parallel transactions, it can use a group commit to flush the binlog writes of multiple transactions with a single `fsync()` call. To enable group commits, set the `sync_binlog` system variable to 1.

**System variables**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>sync_binlog</td>
<td>0 (defer to OS)</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>n (fsync every n transactions)</td>
<td></td>
</tr>
</tbody>
</table>

**Format**

The binlog supports three logging formats: statement-based, row-based and mixed (default). In the statement-based format, the statements are stored. In the row-based format, the rows changed by a statement are stored. In the mixed format, the statements are stored by default, but if a statement is not safe for replication (e.g., it is not deterministic), the rows changed by it are stored instead – making it safe to use non-deterministic functions with replication.

**Tip**

The format can be changed to row-based to improve replication performance if a lot of statements result in changes to a small number of rows and/or take a long time to execute.
Encryption and compression
The binlog events can be encrypted to protect sensitive data and/or compressed via zlib to reduce disk and network IO. When binlog compression is enabled, individual binlog events are compressed before they are written to the binlog - and will remain compressed when replicated. The slave IO thread will decompress the binlog events before they are written to its relay log.

Note
If binlog compression is enabled, not all binlog events will be compressed. It depends on the length of the event (statement or row) and the minimum length configured for compression.

Global transaction IDs (GTIDs)
The master groups, orders and replicates binlog events using GTIDs. GTIDs are comprised of three numbers separated by a dash: domain ID, a 32-bit unsigned integer; server ID, a 32-bit unsigned integer; and sequence, a 64-bit unsigned integer. The resulting identifier ensures transactions are unique across multiple database instances, enabling multi-source replication and circular replication (i.e., bi-directional master/slave replication).

GTIDs enable slaves to continue reading binlog events when the master changes. For example, if a slave is promoted to master, the remaining slaves can be reconfigured to continue reading binlog events from it – and from where they left off, their current GTID.
In addition, GTIDs enable slaves to maintain a durable and consistent replication state. The `mysql.gtid_slave_pos` system table is where slaves store their current GTID. If this table is stored in a transactional storage engine (e.g., InnoDB), a single transaction will be used to update their current GTID and execute the binlog events associated with it.

### Logical view of binlog

<table>
<thead>
<tr>
<th>Commit ID</th>
<th>GTID</th>
<th>Server ID</th>
<th>Event type</th>
<th>Position</th>
<th>End position</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>0-1-200</td>
<td>1</td>
<td>Query</td>
<td>0</td>
<td>150</td>
</tr>
<tr>
<td>100</td>
<td>0-1-201</td>
<td>1</td>
<td>Query</td>
<td>151</td>
<td>500</td>
</tr>
<tr>
<td>100</td>
<td>0-1-202</td>
<td>1</td>
<td>Query</td>
<td>501</td>
<td>600</td>
</tr>
<tr>
<td>101</td>
<td>0-2-203</td>
<td>1</td>
<td>Query</td>
<td>601</td>
<td>800</td>
</tr>
<tr>
<td>101</td>
<td>0-2-203</td>
<td>1</td>
<td>Query</td>
<td>801</td>
<td>1000</td>
</tr>
</tbody>
</table>

### Process

1. The slave IO thread requests binlog events, includes its current GTID
2. The master returns binlog events for the next GTID(s)
3. The slave IO thread writes the binlog events to its relay log
4. The slave SQL thread reads the binlog events from its relay log
5. The slave SQL thread executes the binlog events and updates its current GTID

### Multi-master Clustering

#### Components

**MariaDB Cluster**

MariaDB Cluster, based on Galera Cluster, uses group communication, global transaction ordering, write sets and certification for synchronous replication.

**Group communication**

Clustering is based on group communication. It enables nodes to automatically join the cluster and for the cluster to automatically remove failed nodes. In the context of replication, group communication ensures total ordering of messages sent from multiple nodes.

**Write sets**

A write set contains all rows modified by a transaction (and their primary keys), and is created during the commit phase. It is replicated to every node, including the originating node, via group communication.
Global transaction ordering

When a write set is replicated, it is assigned a GTID, ensuring write sets (and thus transactions) are executed in the same order on every node. It is comprised of a UUID and a sequence number (64-bit signed integer) separated by a colon.

Certification

The write set will be certified on every node using its GTID and the primary keys of rows changed by the transaction. If the write set passes the certification test, it is applied and the transaction is committed. If it does not, the write set is discarded and the transaction is rolled back.

Process

1. Synchronous
   a. Originating node: create a write set
   b. Originating node: assign a global transaction ID to the write set and replicate it
   c. Originating node: apply the write set and commit the transaction

2. Asynchronous
   a. Other nodes: certify the write set
   b. Other nodes: apply the write set and commit the transaction

While write sets are replicated synchronously, they are certified and applied asynchronously. However, certification is deterministic – it succeeds on every node or it fails on every node. Thus, a transaction is committed on every node or it is rolled back on every node.
PERFORMANCE OPTIMIZATIONS

Replication

Parallel replication

The IO and SQL threads on slaves are responsible for replicating binlog events. The IO thread on slaves requests binlog events from the master and writes them to the relay log. The SQL thread on slaves reads binlog events from the relay log and executes them, one at a time. However, when parallel replication is enabled, the slave will use a pool of worker threads to execute multiple binlog events at the same time.

By default, slaves execute binlog events in order. There are two modes: conservative (default) and optimistic. In the conservative mode, parallel replication is limited to a group commit. If multiple transactions have the same commit id, they were executed in parallel on the master and will be executed in parallel on the slaves. However, the transactions will be committed in order. If two transactions do not have the same commit id, the second transaction will not be executed until the first transaction is in the commit phase – ensuring the transactions are committed in the same order the master committed them.

In the optimistic mode, multiple transactions will be executed in parallel regardless of the commit id. However, exceptions include DDL statements, non-transactional DML statements and transactions where a row lock wait was executed on the master. If a conflict is detected between two transactions (e.g., two transactions try to update the same row), the first transaction will be committed and the second transaction will be rolled back and retried.

Note

The slave-parallel-threads system variable must be set to a value greater than 0 to enable parallel replication. In addition, the master can be configured to wait for n microseconds or n transactions, before a group commit. If there are a lot of concurrent transactions on the master, and the number of threads on slaves is high enough, increasing the wait can reduce binlog disk IO on the master and enable slaves to commit transactions faster.

Tip

If there are few conflicts, the mode can be changed to optimistic for better performance.
System variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>slave-parallel-mode</td>
<td>optimistic</td>
<td>conservative</td>
</tr>
<tr>
<td>slave-parallel-threads</td>
<td>0 - n (max: 16383)</td>
<td>0</td>
</tr>
<tr>
<td>binlog_commit_wait_count</td>
<td>0 - n (max: 18446744073709551615)</td>
<td>0</td>
</tr>
<tr>
<td>binlog_commit_wait_usec</td>
<td>0 - n (max: 18446744073709551615)</td>
<td>100000</td>
</tr>
</tbody>
</table>

Read throttling

The master can throttle binlog replication by limiting replication bandwidth, reducing the load on the master when multiple slaves are added or when multiple slaves try to replicate a lot of binlog events at the same time.

System variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>read_binlog_speed_limit</td>
<td>0 (unlimited) - n (kb, max: 18446744073709551615)</td>
<td>0</td>
</tr>
</tbody>
</table>

Clustering

Asynchronous InnoDB logging

InnoDB logs are flushed to disk when a transaction is committed (default). However, with clustering, transactions are made durable via synchronous replication. The originating node can fail before the InnoDB log has been flushed to disk and the transaction will still be committed on the other nodes. By writing and flushing InnoDB logs to disk asynchronously, write performance can be improved.

System variables

<table>
<thead>
<tr>
<th>Variable</th>
<th>Values</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>innodb_flush_log_at_trx_commit</td>
<td>0 (write and flush once a second)</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>1 (write and flush during commit)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 (write during commit, flush once a second)</td>
<td></td>
</tr>
</tbody>
</table>
CONCLUSION

MariaDB Platform supports multiple high availability strategies to match the performance, durability and consistency guarantees of individual use cases – and includes the world’s most advanced database proxy to simplify and improve high availability, further reducing both the business impact and the administrative overhead of infrastructure downtime, planned or unplanned.

Whether it is configured with asynchronous master/slave replication for simplicity and performance, synchronous multi-master clustering for availability and consistency, or something in between, MariaDB Platform has both the flexibility and the capability to meet complex, enterprise requirements on a case-by-case basis.

MariaDB combines engineering leadership and community innovation to create open source database solutions for modern applications – meeting today’s needs and supporting tomorrow’s possibilities without sacrificing SQL, performance, reliability or security.