MariaDB

High availability with MariaDB TX

WHITE PAPER

The definitive guide

Table of Contents

Introduction

High availability

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 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, the database platform must be able to tolerate infrastructure failure. It must maintain availability or enable availability to be restored in a matter of minutes, if not seconds.

MariaDB TX 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 TX is optimized for high availability, downtime due to an unplanned infrastructure failure can be all but removed.

However, when it comes to high availability, there are trade-offs between performance, durability and consistency. In some cases, durability and/or consistency is more important than performance. In others, performance is more important. It depends on the business goals, the use case and the requirements.

This guide explains how replication and clustering works in MariaDB TX, how system variables and parameters can be set to improve performance, durability and/or consistency, the most 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 dynamic routing and automatic topology detection improve 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 (525,600 minutes) and the MTTR is 5 minutes, the database is available 99.999% of the time.

Availability = MTBF / (MTBF - MTTR)
525600 / (525600 + 5) = 99.999% availability

Terminology

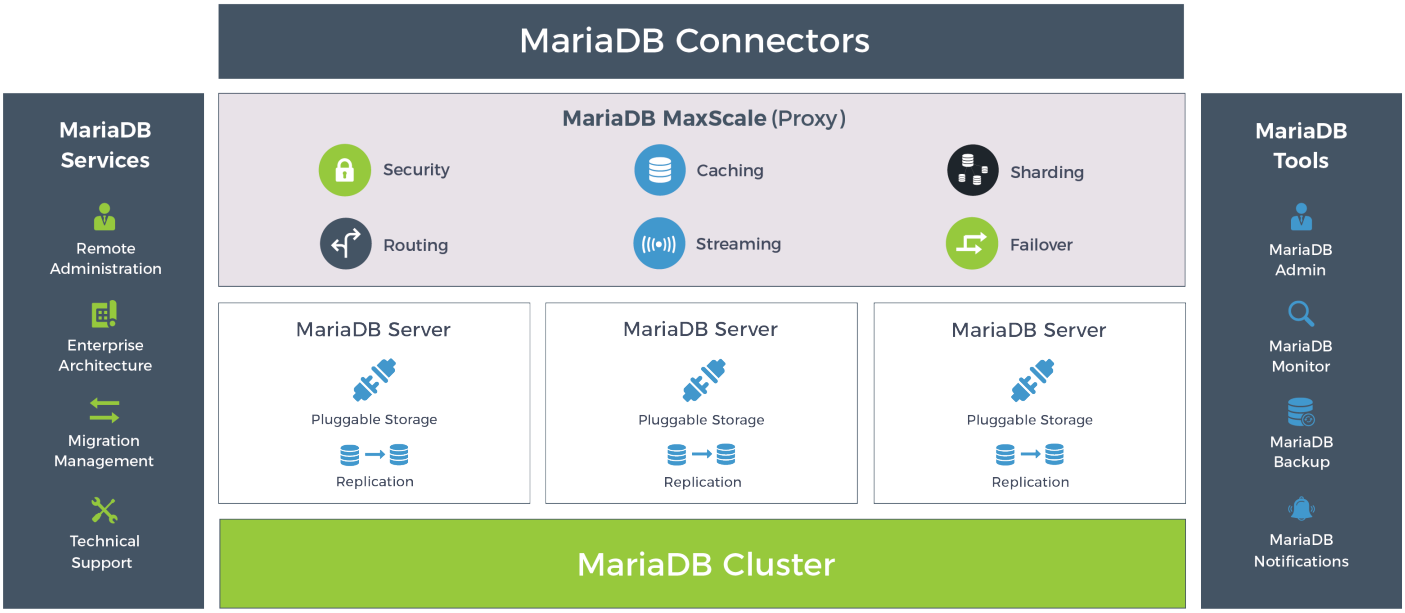
Switchover is when the primary database becomes a standby database, and a standby database becomes the primary database, often in the case of planned maintenance.

Failover is when a standby database becomes the primary database because the primary database has failed and cannot be recovered in time.

Failback is when a failed primary database is recovered and becomes the primary database again while the temporary primary database reverts to standby database again.

MariaDB TX – Born of the community. Raised in the enterprise.

MariaDB TX, with a history of proven enterprise reliability and community – led innovation, is a complete database solution for any and every enterprise. MariaDB TX, when deployed, is comprised of MariaDB connectors (e.g., JDBC/ODBC), MariaDB MaxScale (a database proxy and firewall), MariaDB Server, MariaDB Cluster (multi-master replication), MariaDB tools and access to MariaDB services – and is available via an enterprise open source subscription.



Enterprise reliability

High availability

Ensure uptime for mission-critical applications with built-in replication and automatic failover.

Disaster recovery

Recover from unexpected failure with backup and restore or binary log based rollback.

Security

Secure access to data with authentication, query filtering, data masking and encryption.

Scalability

Scale reads and writes with replication, clustering and sharding.

Performance

Meet user expectations with multi-core processors and a multithreaded architecture.

Community innovation

Open development

Participate in a 100% open and transparent development process – tests and all.

Extensible architecture

Extend everything from storage to routing to implement features as needed.

Community contribution

Benefit from an active community with contributions from industry leaders.

Streaming integration

Stream database changes as Avro or JSON objects to external systems, including Kafka.

Flexible data model

Support applications with semi–structured data by using JSON and/or dynamic columns.

Replication

Components

Binary log (binlog)

MariaDB Server logs the changes made by DML and DDL statements, and their execution time, as events in the binlog. The data is stored in a binary format, but *mysqlbinlog* can be used to display the events as text. The binlog is comprised of log files and an index file.

Group commit

MariaDB Server will, by default, call fsync() to flush binlog writes to disk during the commit phase of a transaction. However, when there are parallel transactions, it will use group commits to flush the binlog writes of multiple transactions with a single fsync() call.

System variables

Variable	Values	Default
innodb_flush_log_at_trx_commit	0 (nothing) 1 (fsync binlog) 2 (write to binlog)	1
sync_binlog	0 (defer to OS) n (fsync every n transactions)	0

Format

The binlog supports three logging formats: statements, rows and mixed (default). MariaDB Server logs statements in the statement-based format and changes to individual rows in the row-based format. In the mixed format, MariaDB Server logs statements, but if a statement is not safe for replication (e.g., it is not deterministic), it logs the changes to individual rows.

TIP The format can be changed to rows to improve replication performance if many statements result in small changes to individual rows and/or take a long time to execute.

System variables

Variable	Values	Default
binlog_format	STATEMENT ROW MIXED	MIXED

Encryption and compression

MariaDB Server can encrypt binlogs to protect sensitive data and/or compress binlog events using zlib to reduce disk and network IO. When binlog compression is enabled, the master compresses individual binlog events before writing them. The binlog events will remain compressed while being replicated to slaves. The slave IO thread will decompress the binlog events before writing them to the relay log.

NOTE MariaDB Server will not compress all binlog events. It depends on the length of the event (DDL/DML statement or row) and the minimum length configured for compression.

System variables

Variable	Values	Default
encrypt-binlog	0 (OFF) 1 (ON)	0 (OFF)
log_bin_compress	0 (OFF) 1 (ON)	0 (OFF)
log_bin_compress_min_len	10 - 1024	256

Global transaction IDs (GTIDs)

MariaDB Server 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.

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

Commit ID	GTID	Server ID	Event type	Position	End position
100	0-1-200	1	Query	0	150
100	0-1-201	1	Query	151	500
100	0-1-202	1	Query	501	600
101	0-2-203	1	Query	601	800
101	0-2-203	1	Query	801	1000

Replication 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
    a. The binlog events are executed
    b. The current GTID is updated

Parallel replication

The IO and SQL threads on slaves are responsible for replicating binlog events. The IO thread requests binlog events from the master and writes them to slave's relay log. The SQL thread reads binlog events from the slave's relay log and executes them on the slave, one at a time. However, when parallel replication is enabled, a pool of worker threads is used 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 conservative mode, parallel replication is limited to a group commit. If multiple transactions were executed at the same time on the master, they can execute at the same time on slaves. However, transactions will be committed in order.

If two transactions do not have the same commit id, worker threads will not execute the second transaction until the first one is in the commit phase – ensuring transactions are performed in the same order as they were on the master.

In optimistic mode, multiple transactions will be performed at the same time regardless of the commit id. However, exceptions include DDL statements, non-transactional DML statements and transactions where a row transpose to 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 commit and the second transaction will be rolled back and retried. If there are few conflicts, optimistic mode is much faster than conservative mode.

System variables

Variable	Values	Default
slave-parallel-mode	optimistic | conservative | aggressive | minimal | none	conservative
slave-parallel-threads	0 - n (max: 16383)	0

System variables

Variable	Values	Default
binlog_commit_wait_count	0 - n (max: 18446744073709551615)	0
binlog_commit_wait_usec	0 - n (max: 18446744073709551615)	100000

Semi-synchronous replication

In addition to asynchronous replication (default), MariaDB Server supports semi-synchronous replication. When replication is asynchronous, the master replicates the binlog events for a transaction after the commit phase. However, when replication is semi-synchronous, the master replicates the binlog events for a transaction during its commit phase. The master will wait for at least one of the slaves to write the binlog events for a transaction to their relay log and flush them to disk (i.e., call fsync).

NOTE If a replication request times out when semi-synchronous replication is enabled, the master will switch to asynchronous replication until one or more slaves catch up.

System variables

Variable	Values	Default
rpl_semi_sync_master_enabled	0 (OFF) | 1 (ON)	0
rpl_semi_sync_master_timeout	0 to n (ms, max: 18446744073709551615)	100000

Performance, durability and consistency

If performance is the most important requirement, asynchronous replication provides the highest write throughput and lowest write latency. However, it creates a window of opportunity for potential data loss if a failover occurs. If the master fails, and one or more transactions have been committed but their binlog events have not been replicated to the slaves, there will be data loss.

This is an acceptable trade-off for many use cases. In particular, use cases where writes are infrequent (e.g., updating the description in a product catalog) or and/or not critical (e.g., a single sensor reading among hundreds if not thousands per second).

If the master writes transactions to the binlog during the commit phase, there is no potential for data loss if the database crashes. The OS will flush the binlog writes in the page cache to disk. However, if the server crashes, there *is* potential for data loss.

If the master flushes the binlog writes of a transaction to disk during its commit phase, there is no potential for data loss if the server crashes. However, if a database or server crash results in a failover, there *is* potential for data loss.

System variables

	Database crash	Server crash	Database failover
Write to the binary log (M)	No data loss	Potential data loss	Potential data loss
Flush binary log to disk (M)	No data loss	No data loss	Potential data loss

If durability is the most important requirement, semi-synchronous replication ensures there is no potential for data loss if a failover occurs. The slave with semi-synchronous replication enabled and the highest current GTID can be promoted to master with no data loss.

System variables

	Database crash	Server crash	Database failover
Flush the relay log to disk (S)	No data loss	No data loss	No data loss

Performance optimization: read throttling

MariaDB Server can throttle binlog replication to slaves by limiting replication bandwidth, reducing the load on the master when multiple slaves are added or when multiple slaves try to replicate many binlog events at the same time.

Variable	Values	Default
read_binlog_speed_limit	0 (unlimited) - n (kb, max: 18446744073709551615)	0

Slave read consistency

If applications are going to read from slaves for higher read throughput, and read consistency is important (i.e., avoiding stale reads), semi-synchronous replication is eventually consistent. While semi-synchronous does not prevent stale reads on slaves, it reduces the window of opportunity – by the time a transaction is committed on the master, at least one of the slaves will have written the binlog events for it to their relay log and flushed it to disk.

This is an acceptable trade-off for many use cases where read throughput is more important than read consistency. In particular, use cases where reads are frequent but writes are not, writes are not critical or writes can be redone by the user/customer – shopping carts, for example.

Replication topologies

MariaDB Server supports multiple replication topologies, with different replication topologies supporting different levels of performance and durability – and different failover processes.

Master/slave

The simplest replication topology, and the most common, is master/slave. However, while slaves can be promoted to master to maintain availability, the master/slave topology enables on-demand read scaling, online disaster recovery and isolated BI/reporting.

A master/slave topology can be configured with asynchronous replication for the highest performance, semi-synchronous replication for the highest durability or mixed replication for high durability and high performance.

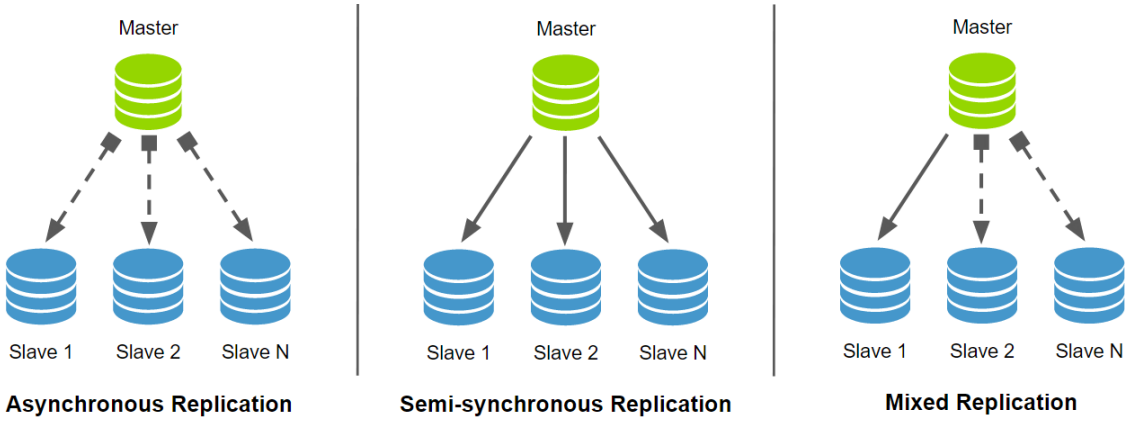


Diagram 1: master/slave topologies

If semi-synchronous replication is enabled on multiple slaves (creating multiple hot standbys), multiple failovers can be performed without the potential for data loss. However, with mixed replication, semi-synchronous replication is enabled on a single slave to eliminate the potential for data loss if the master fails (creating a single hot standby). The remaining slaves default to asynchronous replication, reducing the performance cost of semi-synchronous replication.

Failover

Diagram 2: master/slave failover process

When failover occurs, automatic or manual, a single slave is promoted to master while the remaining slaves are reconfigured to request binlog events from the new master. The failover process should promote a slave configured with semi-synchronous replication, if there is one. If there is, semi-synchronous replication should be enabled on one of the remaining slaves.

Tiered master/slave (with intermediate master/relay)

When the master replicates to many slaves, increased network and IO contention can create a performance cost. It can be avoided by configuring an intermediate master to act as a relay between the master and the slaves. The intermediate master has both a relay log (to write binlog events from the master) and a binlog (to log database changes for slaves).

A tiered master/slave topology can be configured with asynchronous replication for the highest performance, semi-synchronous replication for the highest durability or mixed replication for high durability and high performance.

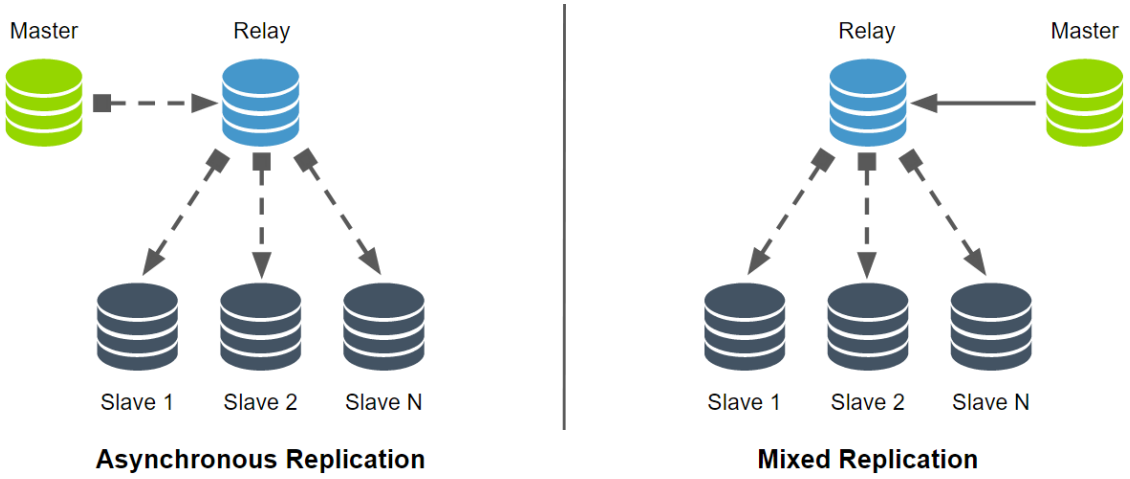


Diagram 3: tiered master/slave topologies

With mixed replication, semi-synchronous replication is enabled on the intermediate master to eliminate the potential for data loss if the master fails and to create a hot standby. The slaves default to asynchronous replication to avoid the performance cost of semi-synchronous replication.

Failover

When failover occurs, whether it be automatic or manual, the intermediate master is promoted to master and either a) the slaves are not updated, converting to master/slave topology or b) one of the slaves is promoted to intermediate master while the remaining slaves are reconfigured to request binlog events from the new intermediate master. If semi-synchronous replication is enabled on the intermediate master, there is no potential for data loss.

Option one: convert to a master/slave topology

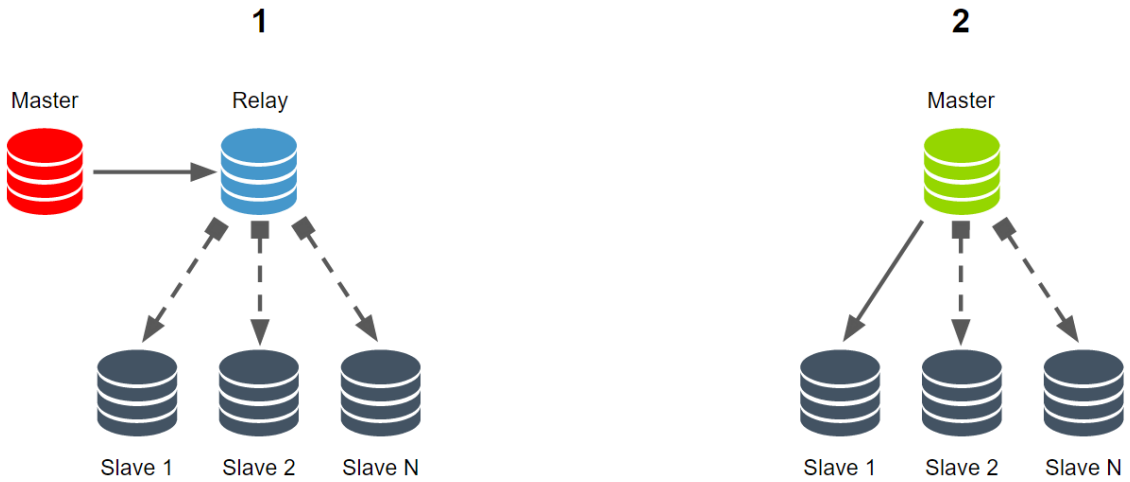


Diagram 4: tiered master/slave failover process with relay promotion

Option two: promote a slave to intermediate master

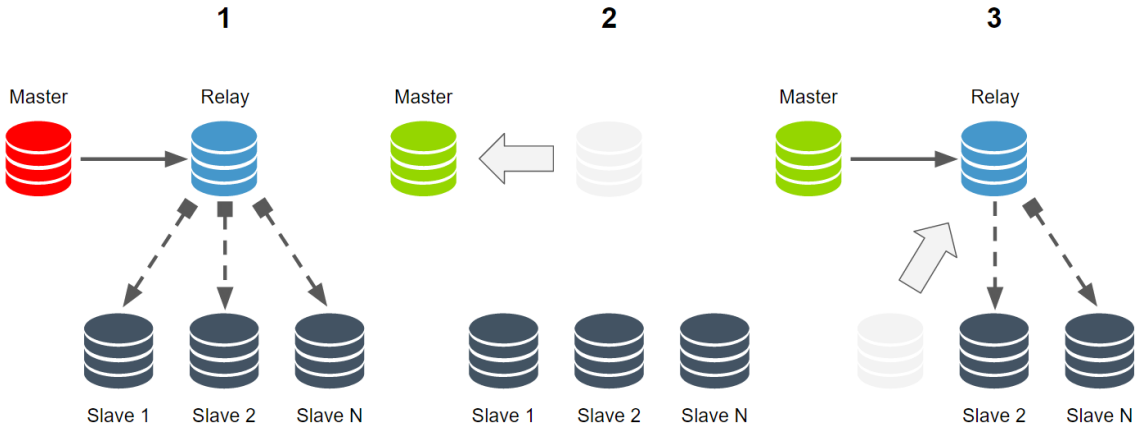


Diagram 5: tiered master/slave failover process with relay and slave promotion

Bi-directional master/slave (circular)

In the same way an intermediate master can act as both a master and a slave, every node in bi-directional replication topology is both a master and a slave – a slave to every other node. However, while configuring every node with unique auto-increment offsets prevents insert conflicts, there is no conflict detection for other writes. Thus, writes should be executed on a single node.

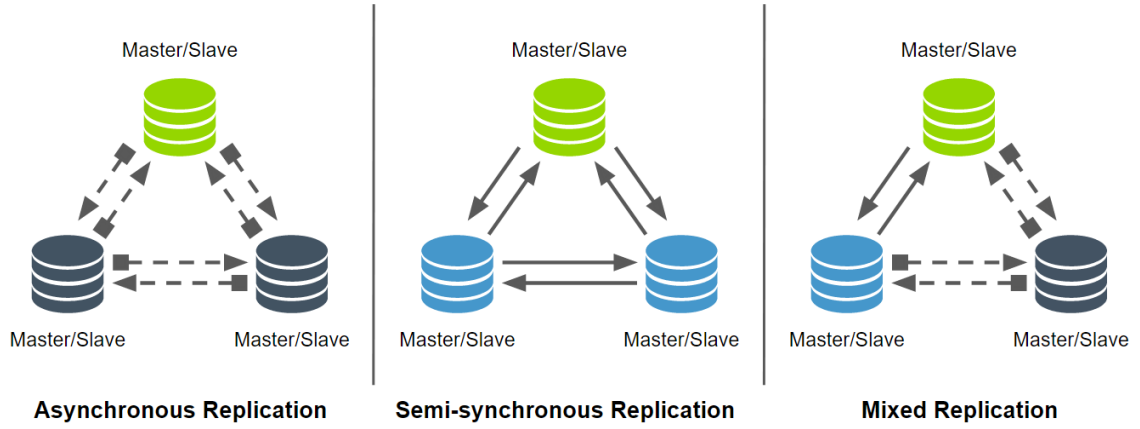


Diagram 6: bi-directional master/slave topologies

A bi-directional master/slave topology can be configured with asynchronous replication for the highest performance, semi-synchronous replication for the highest durability or mixed replication for high durability and high performance.

With mixed replication, semi-synchronous replication is enabled on one node to eliminate the potential for data loss if the master fails and to create a hot standby. The remaining nodes default to asynchronous replication to avoid the performance cost of semi-synchronous replication.

Failover

If applications write to the active master and it fails, they can write to one of the passive masters. There are no database configuration changes required. However, if semi-synchronous replication was enabled between the previous active master and the new active master, and durability remains a requirement, semi-synchronous replication should be enabled on one of the remaining passive masters.

Diagram 7: bi-directional master/slave failover process

Clustering

Components

MariaDB Cluster

MariaDB Cluster, based on Galera Cluster, enables multi-master clustering using synchronous replication based on group communication, global transaction ordering, write sets and certification. In a cluster, every node is a master. It can execute reads and writes, and it can replicate database changes (i.e., transactions) to every other node. There are no slaves.

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 from multiple masters and is used to send write sets to every node, including the originating node.

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 assigned a global transaction ID (GTID) and replicated to every node, including the originating node, via group communication.

Global transaction ordering

When a write 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 modified 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.

Replication 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. All nodes: certify the write set
   d. Originating node: apply the write set and commit the transaction
2. Asynchronous
   a. 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 even though certification is asynchronous.

Performance, durability and consistency

Performance

While clustering provides high availability, read and write scaling enable it to improve performance as well. However, write scaling is limited because write sets are replicated synchronously, and to every node. However, write sets can be applied on multiple nodes in parallel, and because write sets are row-based, SQL statements do not have to be parsed and replicated on every node.

Durability

Transactions are durable because write sets are synchronously replicated to every node during the commit phase.

Consistency

The following isolation levels are supported for reads: read uncommitted, read committed and repeatable read. It provides stronger read consistency than asynchronous or semi-synchronous replication because write sets are replicated synchronously – when a transaction is committed, every node will have the write set. In addition, there is little to no slave lag.

The window of opportunity for stale reads can be removed by configuring nodes to wait until a write set is applied before continuing to execute queries and/or transactions. However, doing so will increase the latency of delayed queries and/or transactions. The types of queries and/or transactions to delay until the write set is applied can be specified.

System variables

Variable	Values	Default
wsrep_sync_wait	0 (DISABLED) 1 (READ) 2 (UPDATE and DELETE) 3 (READ, UPDATE and DELETE) 4 (INSERT and REPLACE)	0

Failover

There is no need for manual failover because every node is a master and if a master fails, the cluster will remove it. Therefore, applications can read and write from a different master at any time, and without delay.

Example topologies

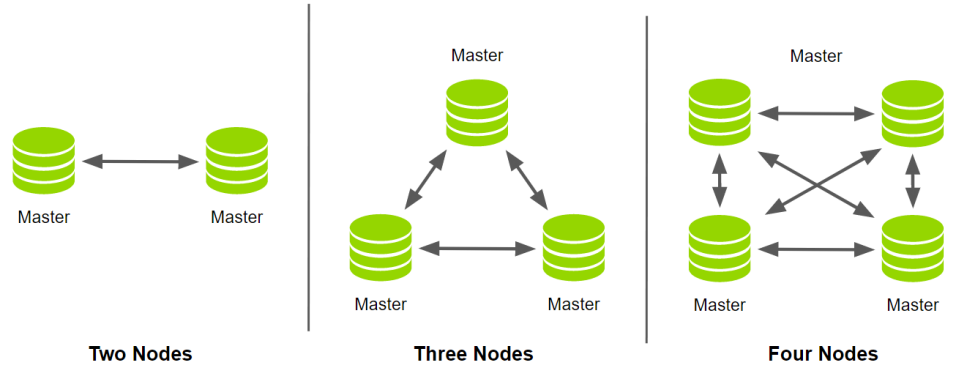


Diagram 8: multi-master topologies

Routing

MariaDB MaxScale, an advanced database proxy, improves the availability, scalability and security of MariaDB Server. In particular, MariaDB MaxScale includes a number of features to improve and/or maintain high availability during normal operations, planned or unplanned maintenance and unexpected database or infrastructure failures: intelligent and transparent routing, automatic topology detection and dynamic server configuration.

Read-write splitting

If applications will read from slaves for increased read throughput, read-write splitting can be used to route writes to the master while load balancing reads to one or more slaves.

NOTE If reads are routed to one or more slaves with semi-synchronous replication enabled, reads will be eventually consistent.

Strong consistency

If strong consistency is required, the *consistent critical reads* filter can be added. It is trigger by a write, and will temporarily a) route the next *n* reads to the master or b) route reads to the master for the next *n* seconds. In addition, regular expressions can be used to specify which reads must be rerouted and which reads can be ignored (i.e., routed to slaves).

NOTE While a *consistent critical reads* filter can be used for strong consistency, it temporarily disables read scaling – and thus temporarily lowers read throughput. The value of the time parameter should be based on slave lag so reads are rerouted to the master long enough for the write to be replicated to and applied on the slaves, but not longer.

Parameters

Parameter	Values	Default
time	0 - n (seconds)	60
count	n	DISABLED
match	regex	N/A
ignore	regex	N/A

Example one: master/slave replication with read-write splitting

MariaDB MaxScale is configured to route writes to the master and to a) route reads to Slave 1 when eventual

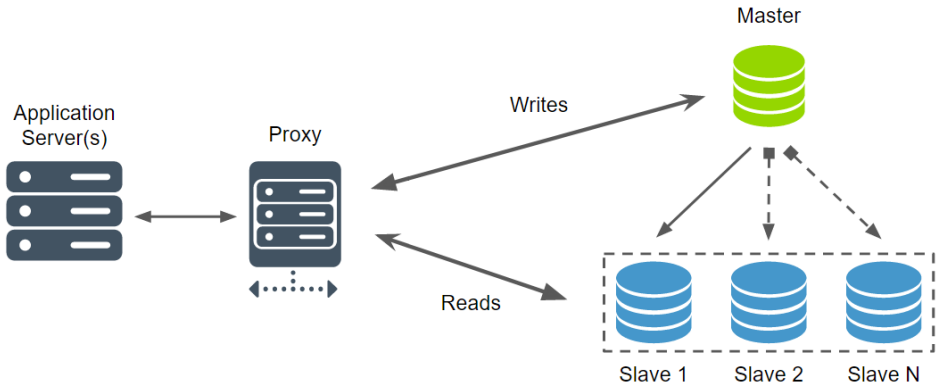


Diagram 9: master/slave topology with read-write splitting

consistency is required or b) load balance reads across all slaves when higher read throughput is required.

Example two: multi-master clustering with read-write splitting

MariaDB MaxScale will designate a single node to be the "active" master with the remaining nodes becoming "passive" masters. It will route writes to the "active" master and load balance reads across "passive" masters.

When different write sets are replicated from different nodes, the data can become inconsistent if exclusive locks are not used (i.e., SELECT... FOR UPDATE). However, if write sets are replicated from a single node and other nodes wait for write sets to be applied before continuing to execute queries, transactions will be serializable – providing strong read consistency.

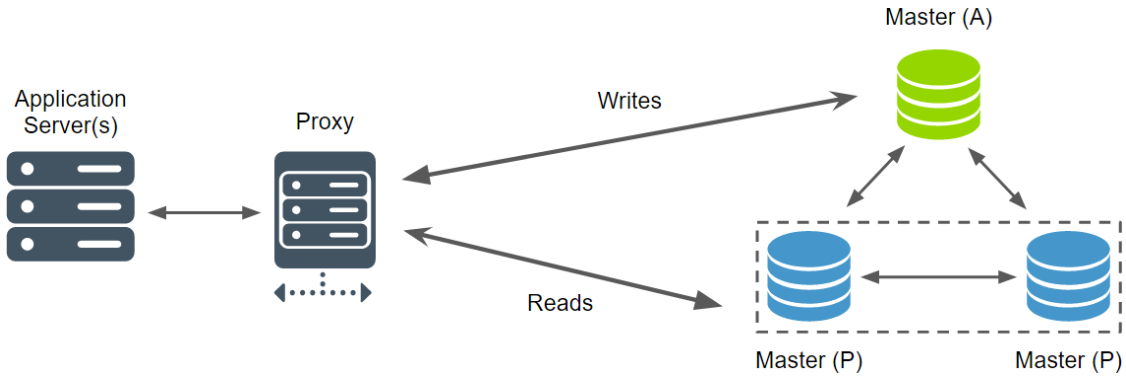


Diagram 10: multi-master topology with read-write splitting

Transparent topology changes

When a failover occurs, manual or automatic, the topology changes – for example, a slave is promoted to master or an intermediate master is promoted to slave. MariaDB MaxScale hides the topology (and thus changes to the topology) from applications, removing the need to modify application server or network configuration when the topology changes (e.g., a failover occurs).

Example: failover in a master/slave topology

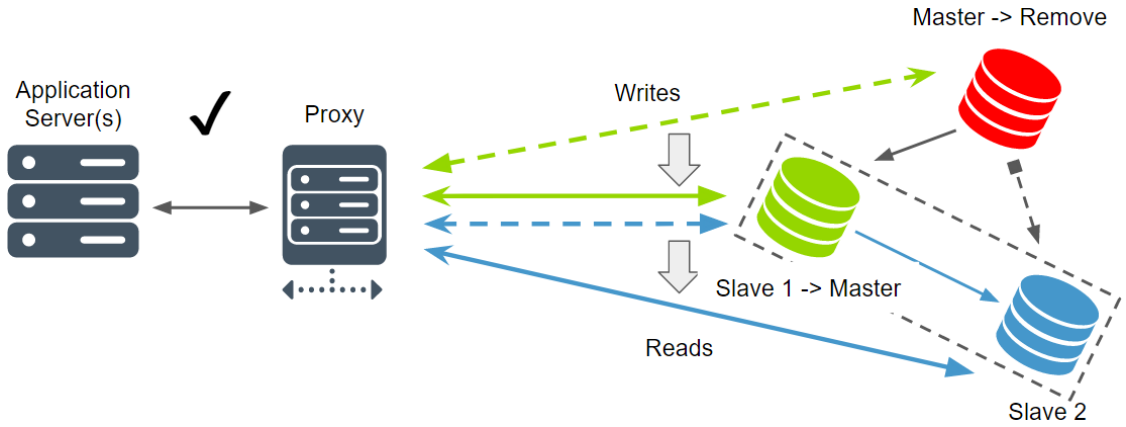


Diagram 11: multi-master failover process with read-write splitting

1. The master fails
2. The failover process changes the topology
   a. The master is removed
   b. Slave 1 is promoted to master
   c. Slave 2 is updated to replicate binlog events from the new master
3. MariaDB MaxScale detects the topology changes
   a. Writes are routed to the new master
   b. Reads are routed to Slave 2

The application servers do not know a failover has occurred, or the topology changed.

Automatic failover

There are three options for performing an automatic failover with MariaDB MaxScale: promote the slave to master when the master fails in two-node master/slave topology, execute a command line script to perform a failover and change routes when the master fails or detect topology changes and change routes when third-party tools like MHA or Replication Manager automatically perform a failover when the master fails.

Live upgrades

The *tee* filter can be used to route reads and writes to multiple databases, enabling administrators to route live traffic to multiple databases. The *tee* filter can be used, for example, to deploy a staging environment with a new version of MariaDB Server and use live traffic for functional and performance testing before putting it into production.

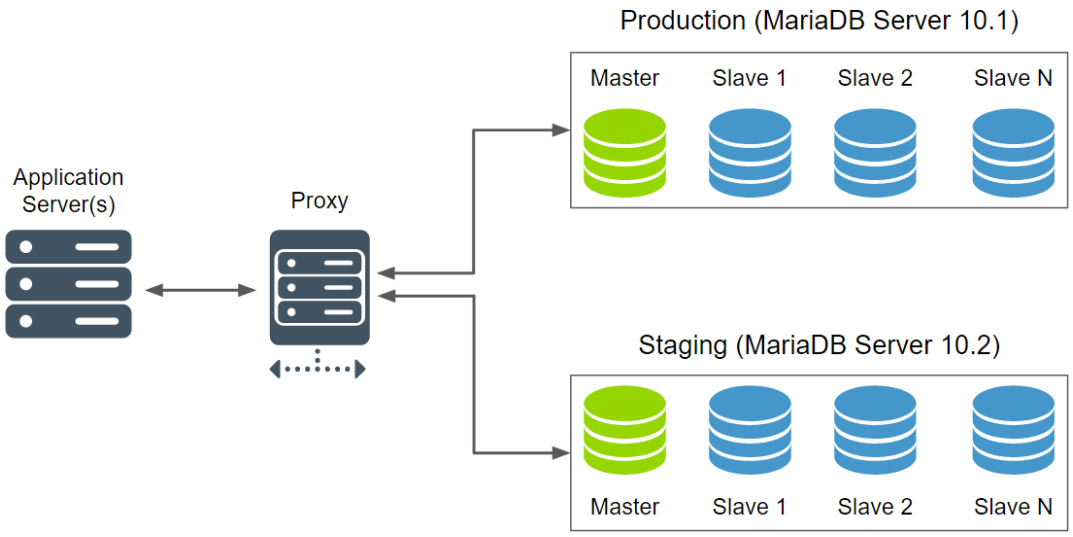


Diagram 12: routing live traffic to both staging and production environments

In fact, databases can be upgraded without updating application code. For example, if the query syntax changes in a new version of MariaDB Server, MariaDB MaxScale can translate the queries for the application(s) – and it is all transparent to the application(s).

Conclusion

MariaDB TX 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 TX 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.

Americas: sales-AMER@mariadb.com
Europe, Middle East, Africa: sales-EMEA@mariadb.com
Asia Pacific: sales-APAC@mariadb.com