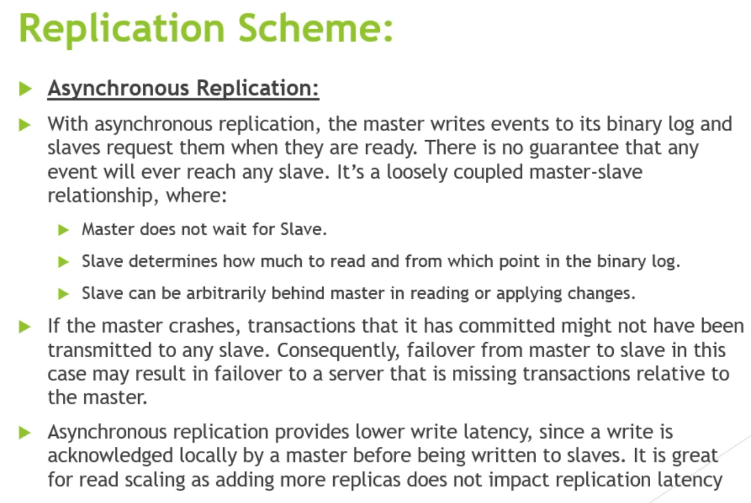
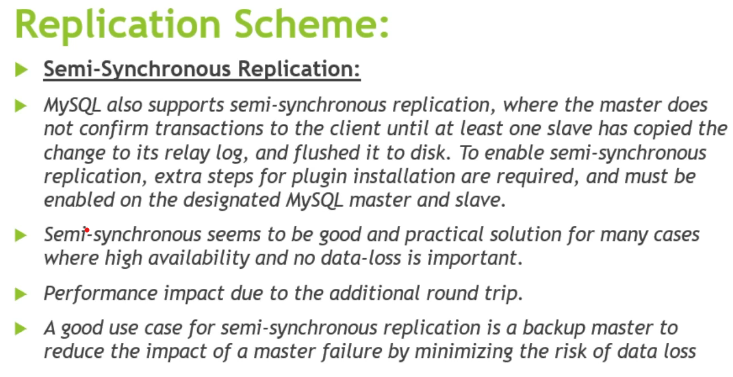
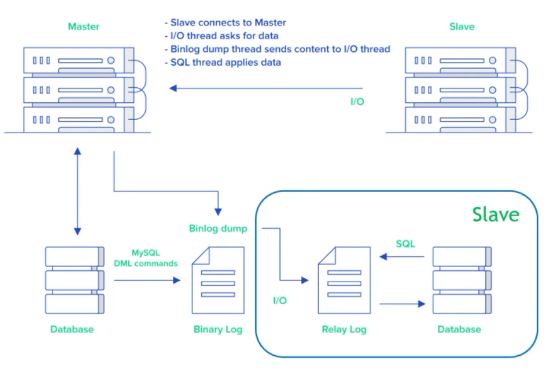
Mysql Arch for Replication

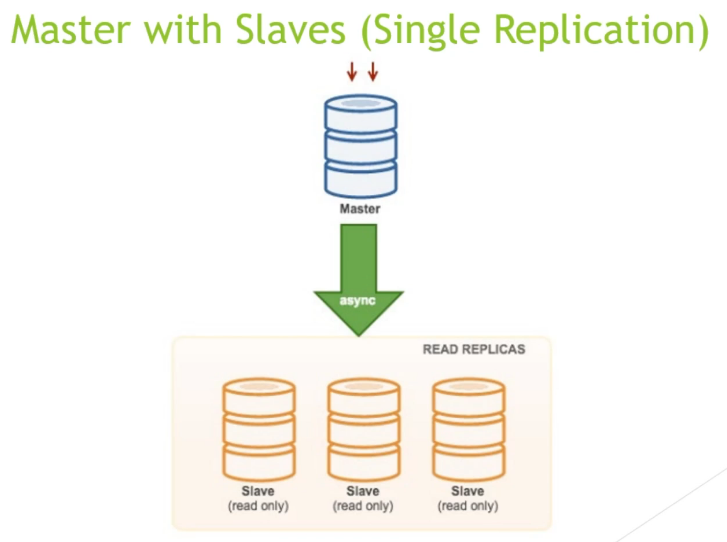


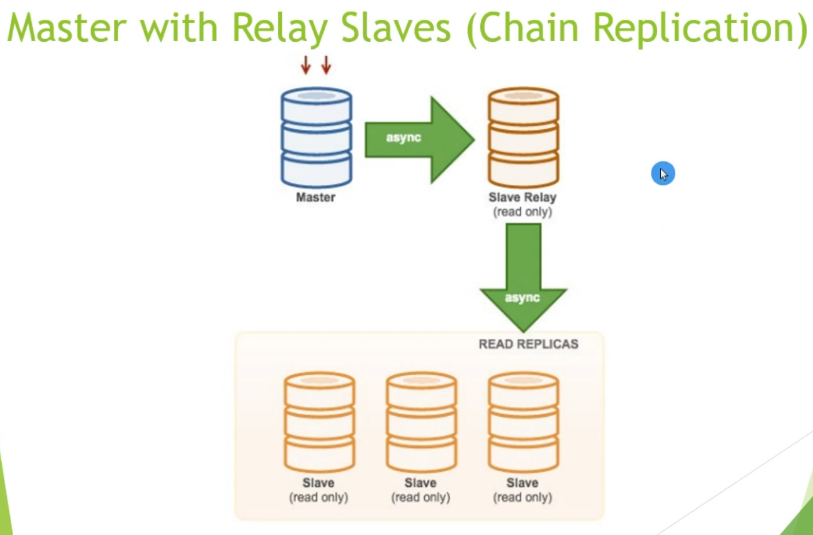


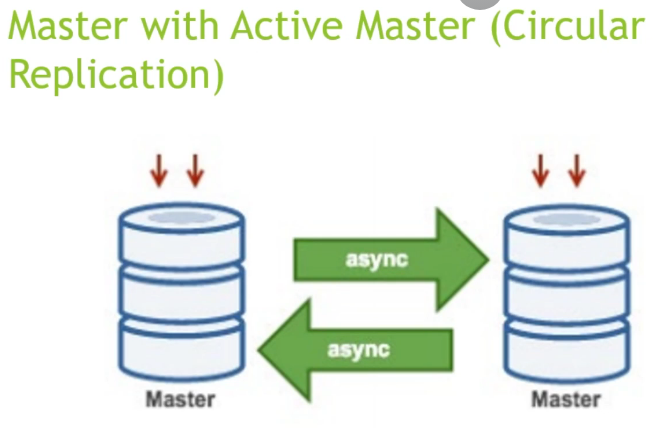


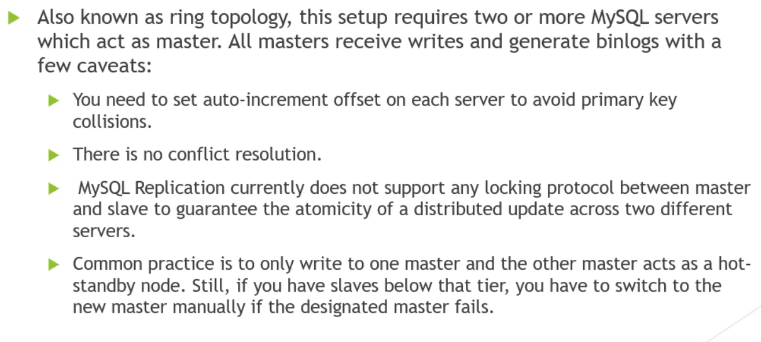
Master with Single Slave Arch for Async

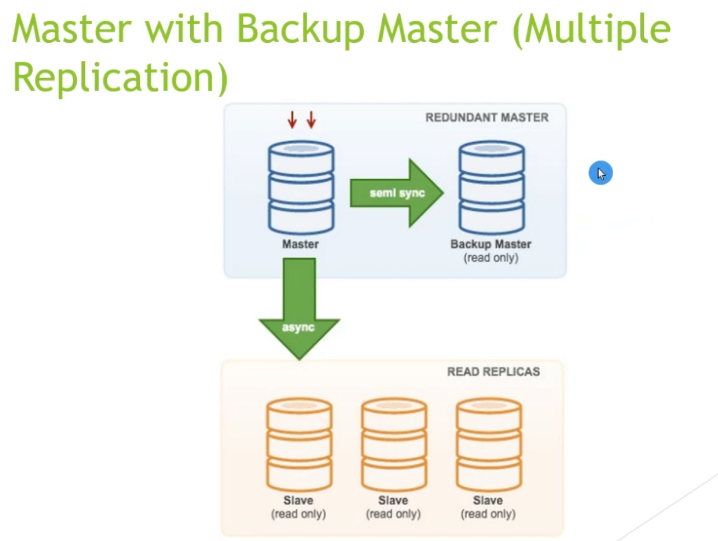


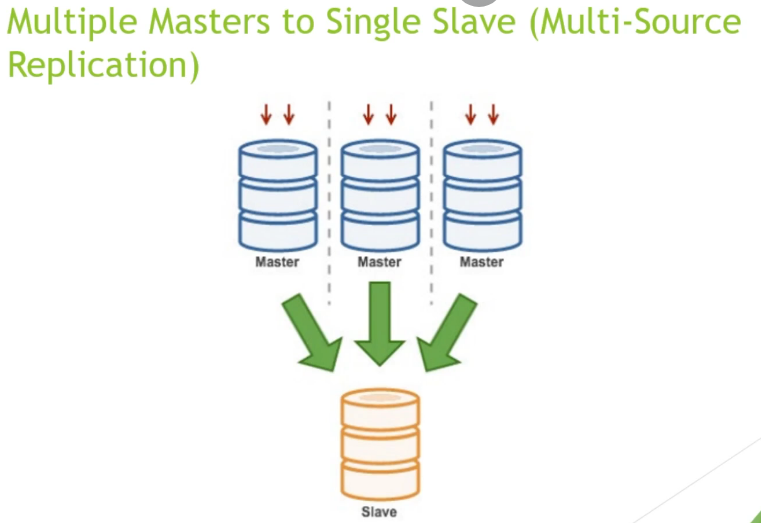




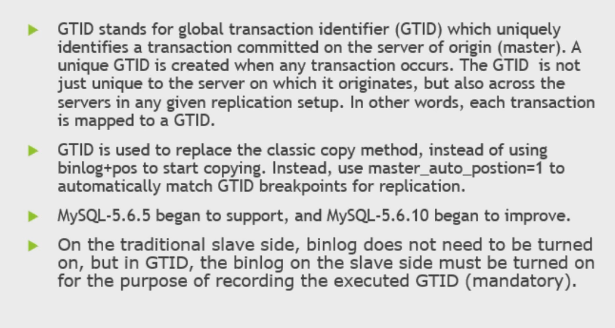


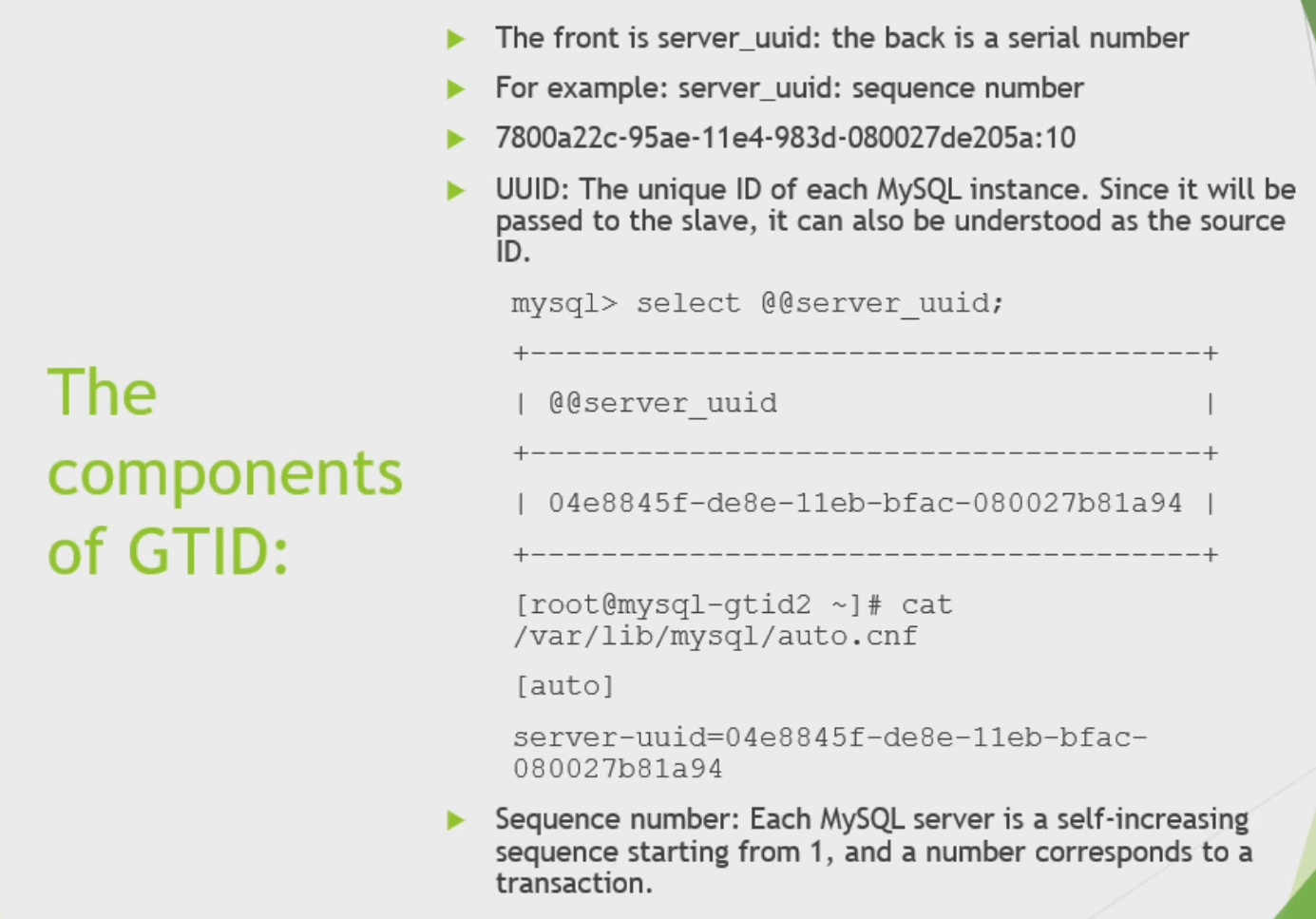




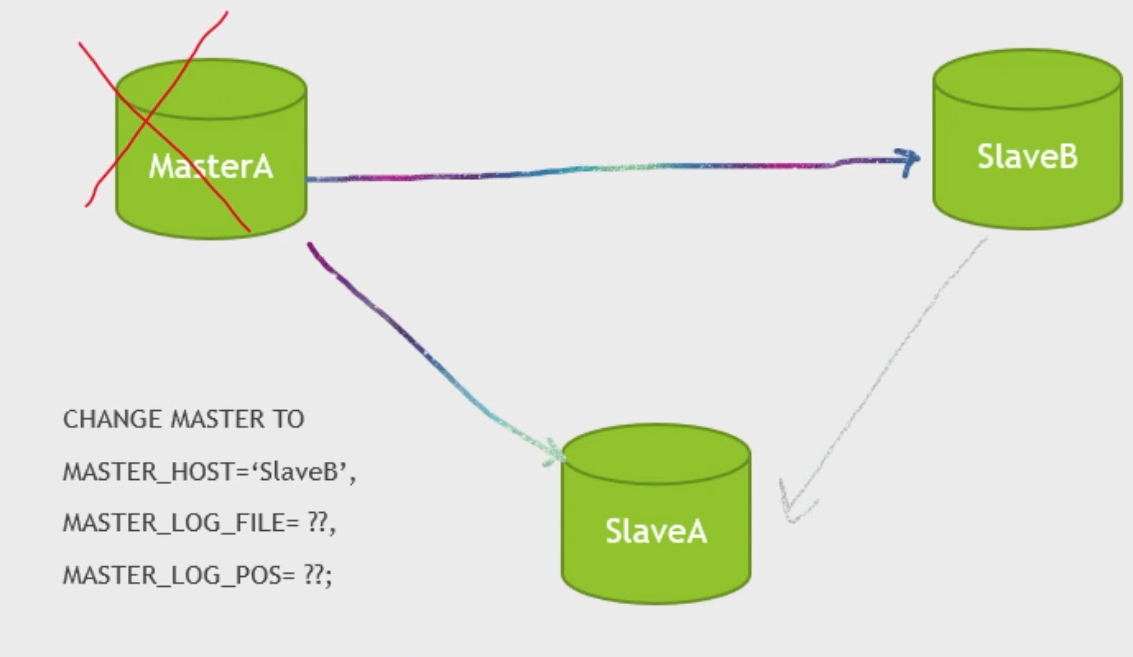


GTID Based Replication

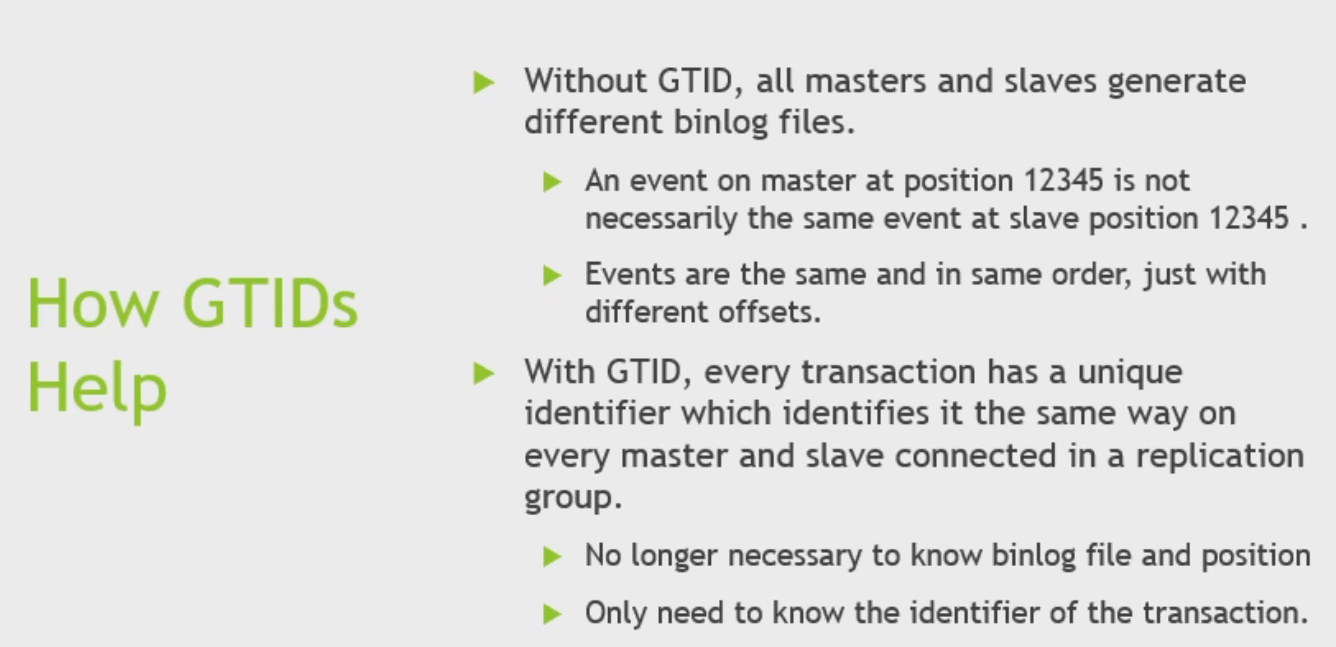


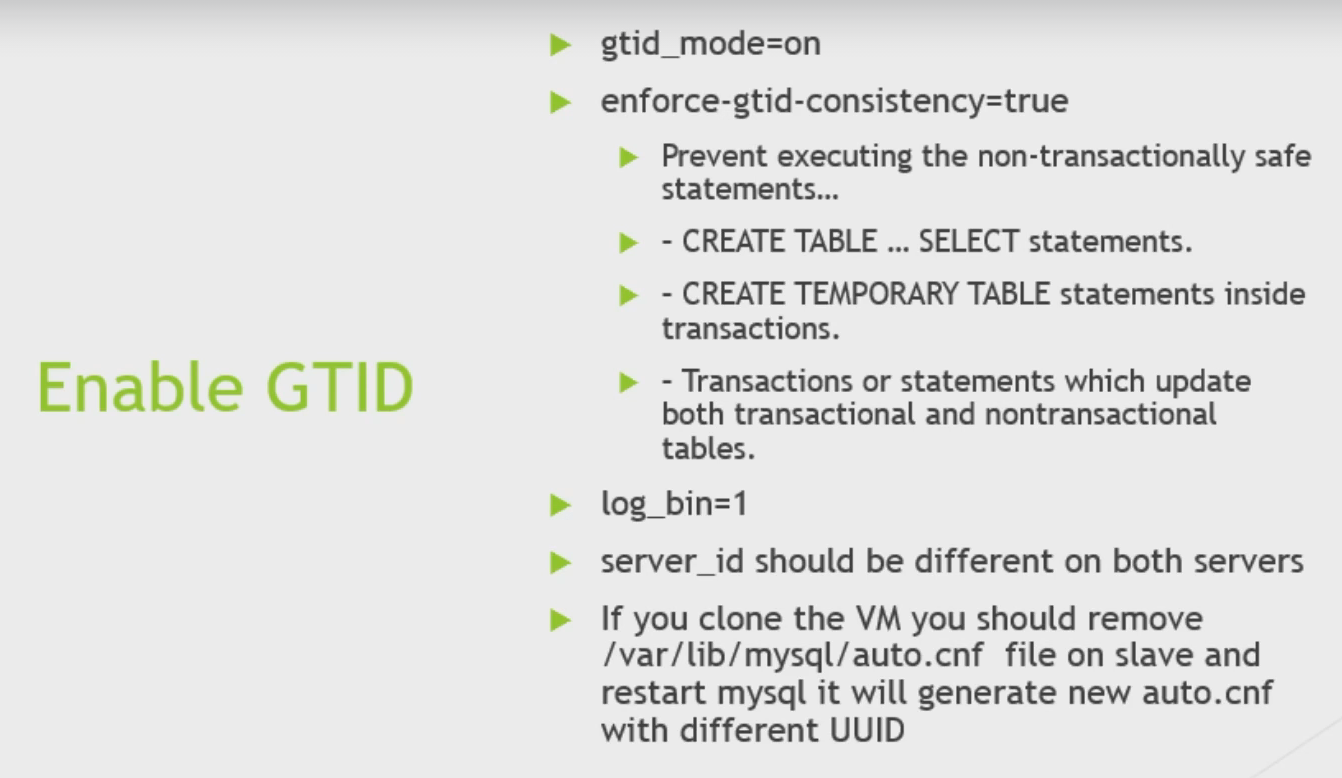


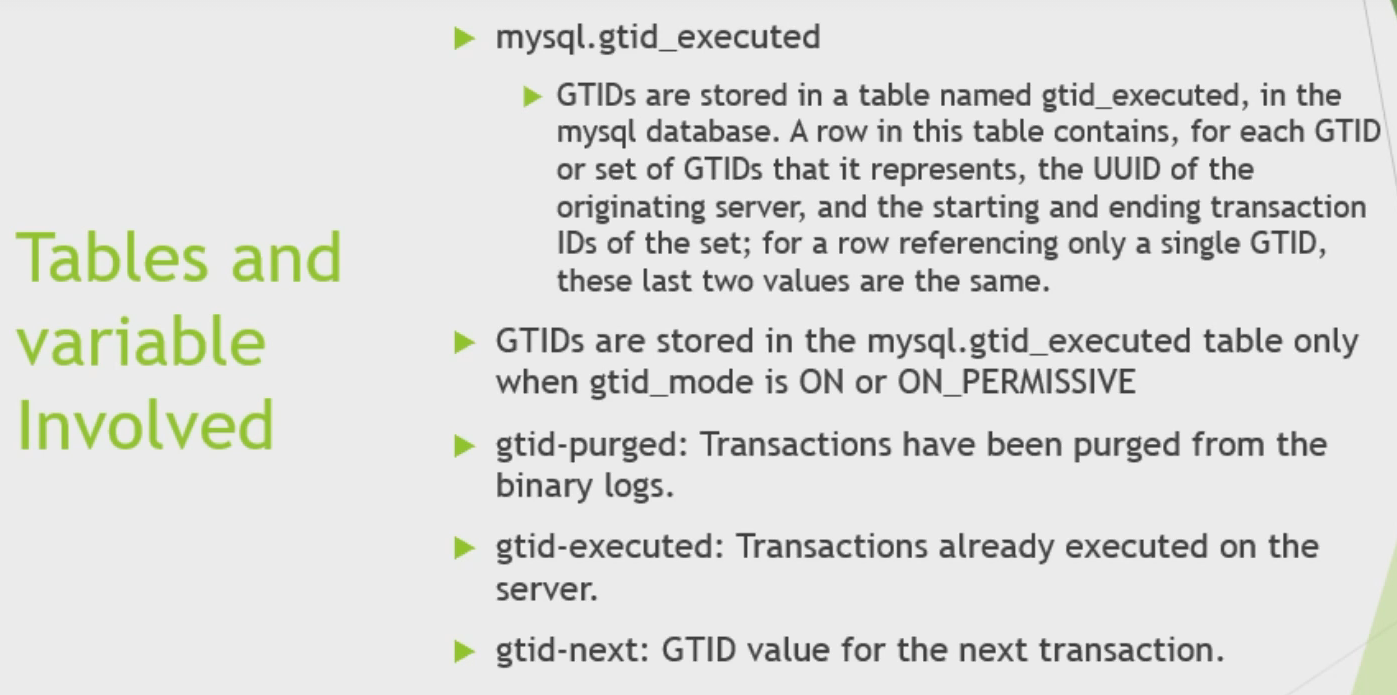


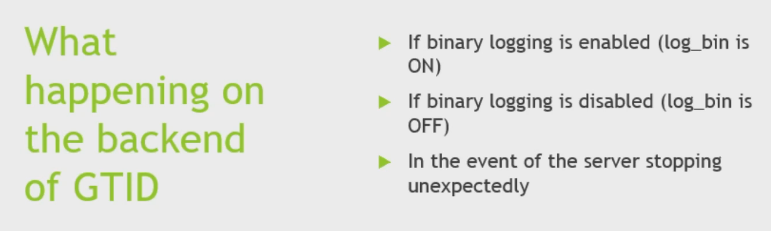




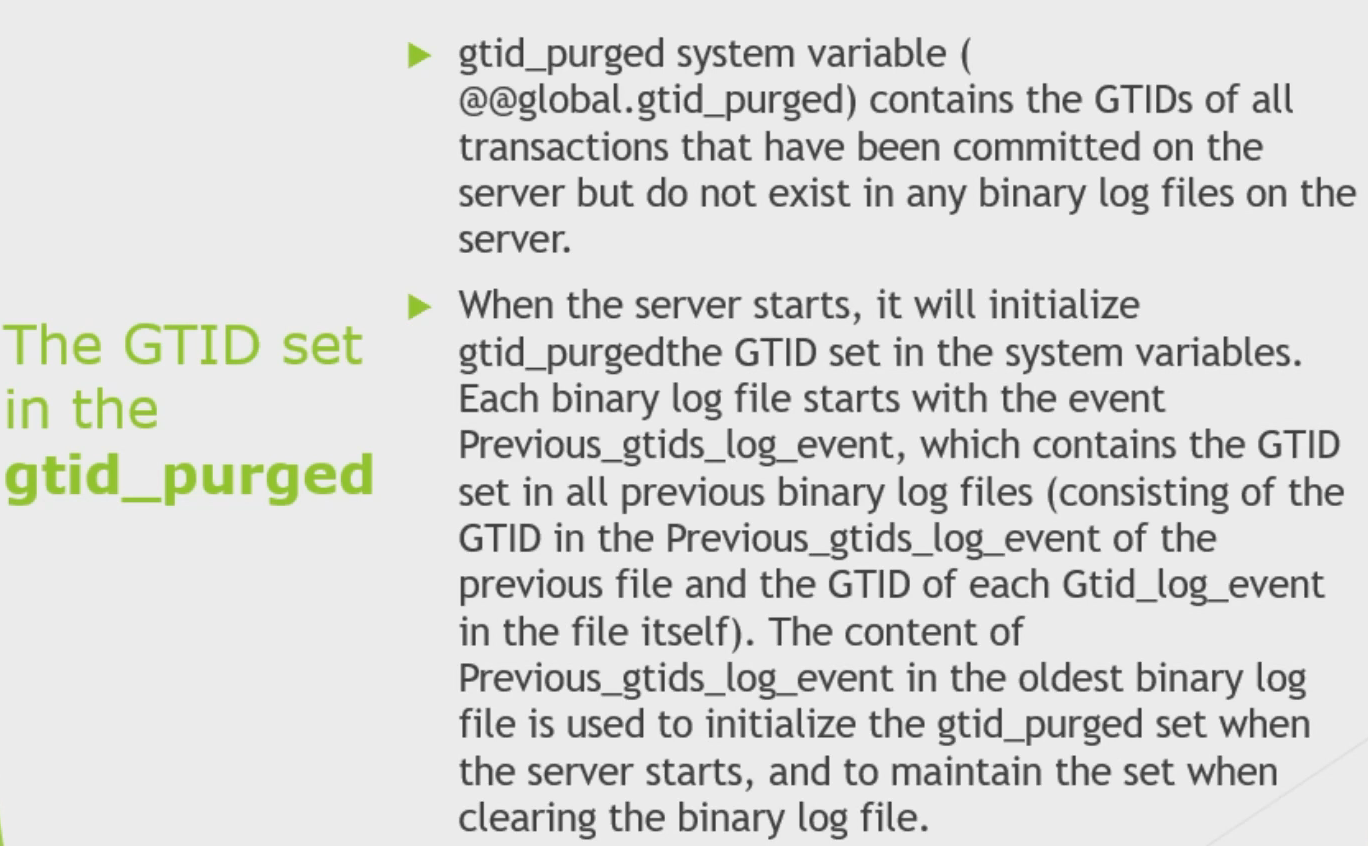




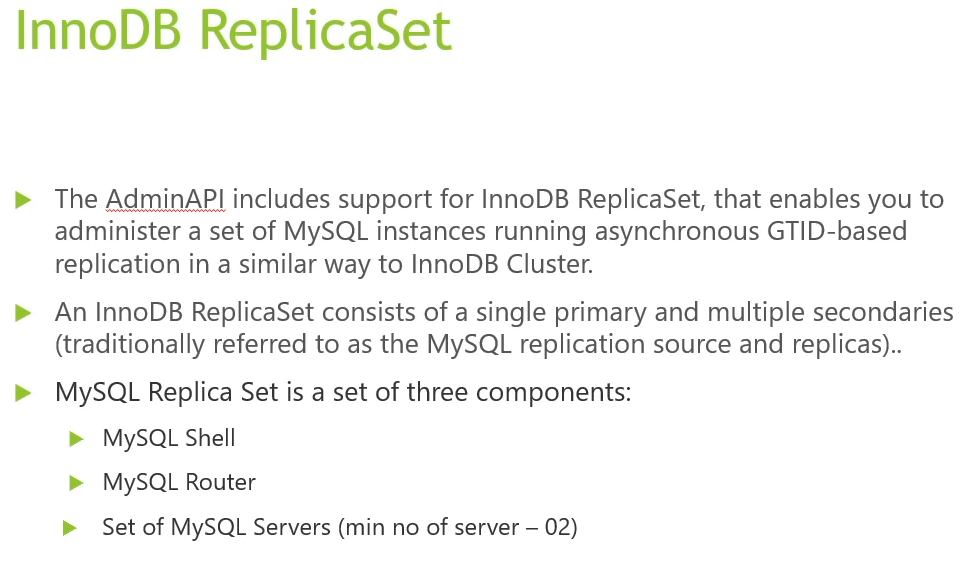








**InnoDB ReplicaSet**



Being based on MySQL Replication, an InnoDB ReplicaSet has a single primary, which replicates to one or more secondary instances. An InnoDB ReplicaSet does not provide all of the features which InnoDB Cluster provides, such as automatic failover, or multi-primary mode. But, it does support features such as configuring, adding, and removing instances in a similar way. You can manually switch over or fail over to a secondary instance, for example in the event of a failure. You can even adopt an existing Replication deployment and then administer it as an InnoDB ReplicaSet.

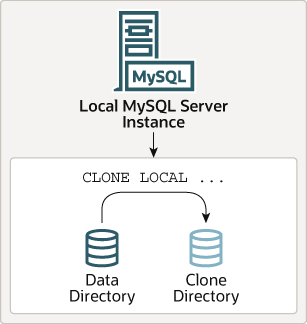
You work with InnoDB ReplicaSet using the [AdminAPI](https://dev.mysql.com/doc/mysql-shell/8.0/en/admin-api-overview.html), provided as part of MySQL Shell. AdminAPI is available in JavaScript and Python, and is well suited to scripting and automation of deployments of MySQL to achieve high-availability and scalability. By using MySQL Shell's AdminAPI, you can avoid the need to configure many instances manually. Instead, AdminAPI provides an effective modern interface to sets of MySQL instances and enables you to provision, administer, and monitor your deployment from one central tool.

InnoDB ReplicaSet supports [MySQL Clone](https://dev.mysql.com/doc/refman/8.0/en/clone-plugin.html), which enables you to provision instances simply. You can simply [add an instance](https://dev.mysql.com/doc/mysql-shell/8.0/en/add-instance-replicaset.html) to the replica set and it is automatically provisioned.

---- Clone Plugin ---

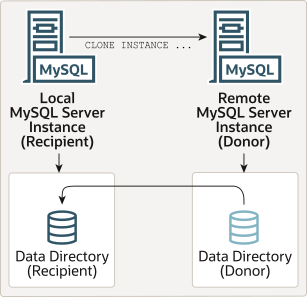
The clone plugin, permits cloning data locally or from a remote MySQL server instance. Cloned data is a physical snapshot of data stored in InnoDB that includes schemas, tables, tablespaces, and data dictionary metadata.

**Figure 5.1 Local Cloning Operation**



A local cloning operation clones data from the MySQL server instance where the cloning operation is initiated to a directory on the same server or node where MySQL server instance runs.

**Figure 5.2 Remote Cloning Operation**

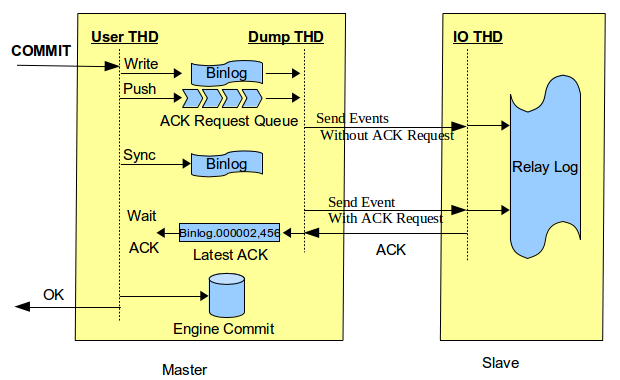


A remote cloning operation involves a local MySQL server instance (the “recipient”) where the cloning operation is initiated, and a remote MySQL server instance (the “donor”) where the source data is located. When a remote cloning operation is initiated on the recipient, cloned data is transferred over the network from the donor to the recipient

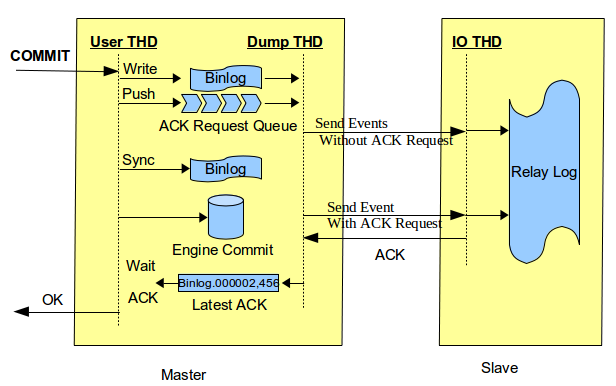
---------

Similarly, InnoDB ReplicaSet is tightly integrated with [MySQL Router](https://dev.mysql.com/doc/mysql-router/8.0/en/) **.** MySQL Router can automatically configure itself based on an InnoDB ReplicaSet, in a process called [bootstrapping](https://dev.mysql.com/doc/mysql-shell/8.0/en/admin-api-bootstrapping-router.html), which removes the need for you to configure routing manually. MySQL Router then transparently connects client applications to the InnoDB ReplicaSet, providing routing and load-balancing for client connections.

**Semi Synchronus Replication**



AFTER\_SYNC

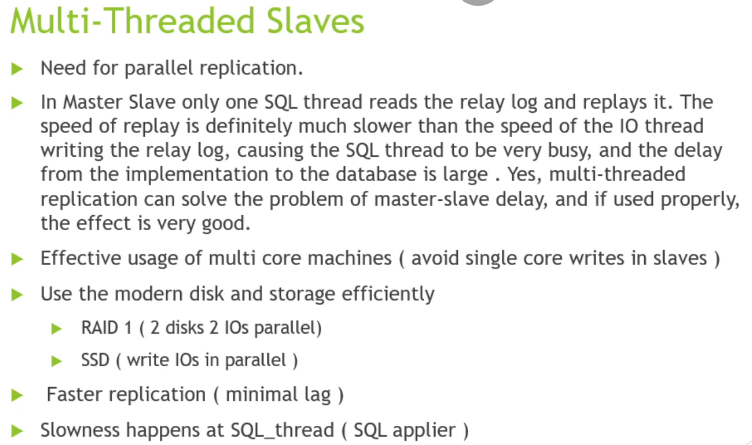


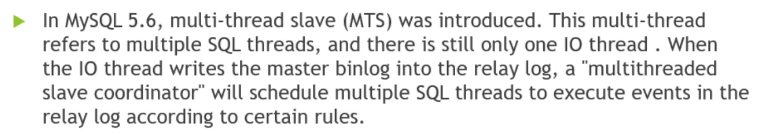
AFTER\_COMMIT

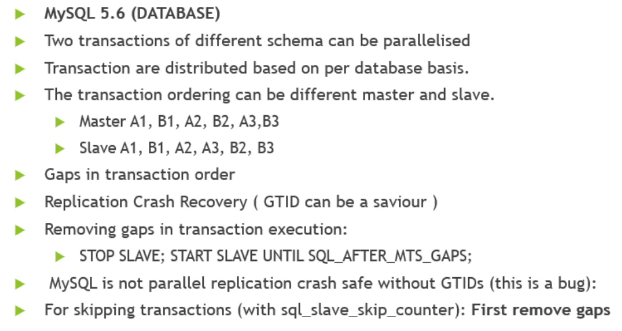
Semisynchronous replication falls between asynchronous and fully synchronous replication. The source waits until at least one replica has received and logged the events (the required number of replicas is configurable), and then commits the transaction. The source does not wait for all replicas to acknowledge receipt, and it requires only an acknowledgement from the replicas, not that the events have been fully executed and committed on the replica side. Semisynchronous replication therefore guarantees that if the source crashes, all the transactions that it has committed have been transmitted to at least one replica.

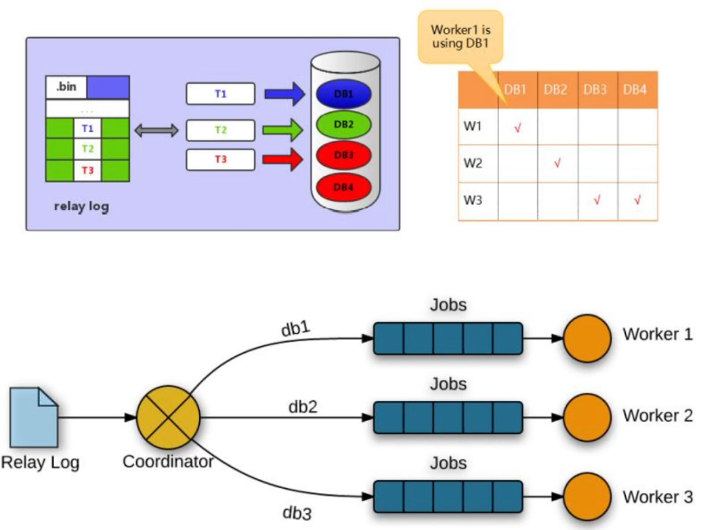
Until a semisynchronous source receives acknowledgment from the required number of replicas, the transaction is on hold and not committed.

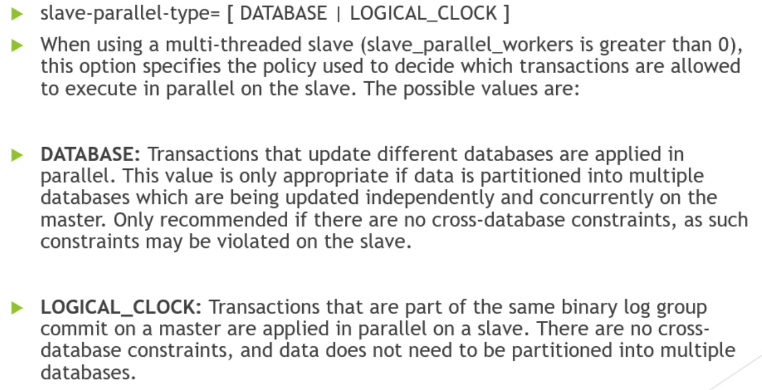
**Multi Threaded Slaves**

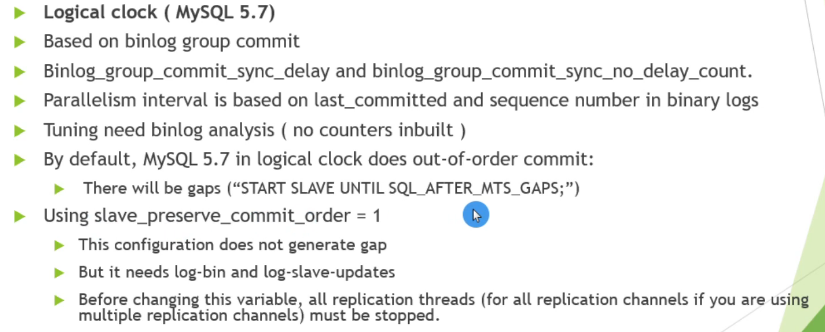














**Group Replication**

Groups can operate in a single-primary mode with automatic primary election, where only one server accepts updates at a time. Alternatively, groups can be deployed in multi-primary mode, where all servers can accept updates, even if they are issued concurrently.

There is a built-in group membership service that keeps the view of the group consistent and available for all servers at any given point in time.

Group Replication guarantees that the database service is continuously available. However, it is important to understand that if one of the group members becomes unavailable, the clients connected to that group member must be redirected, or failed over, to a different server in the group, using a connector, load balancer, router, or some form of middleware.

MySQL Group Replication provides distributed state machine replication with strong coordination between servers. Servers coordinate themselves automatically when they are part of the same group.

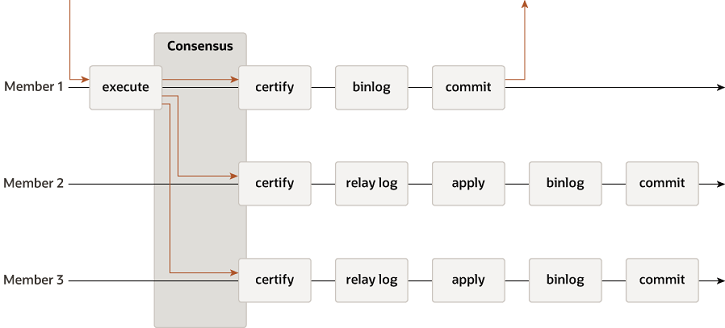
For a transaction to commit, the majority of the group have to agree on the order of a given transaction in the global sequence of transactions. Deciding to commit or abort a transaction is done by each server individually, but all servers make the same decision. If there is a network partition, resulting in a split where members are unable to reach agreement, then the system does not progress until this issue is resolved. Hence there is also a built-in, automatic, split-brain protection mechanism.

All of this is powered by the provided Group Communication System (GCS) protocols. These provide a failure detection mechanism, a group membership service, and safe and completely ordered message delivery

When a read-write transaction is ready to commit at the originating server, the server atomically broadcasts the write values (the rows that were changed) and the corresponding write set (the unique identifiers of the rows that were updated). Because the transaction is sent through an atomic broadcast, either all servers in the group receive the transaction or none do. If they receive it, then they all receive it in the same order with respect to other transactions that were sent before. All servers therefore receive the same set of transactions in the same order, and a global total order is established for the transactions.

However, there may be conflicts between transactions that execute concurrently on different servers. Such conflicts are detected by inspecting and comparing the write sets of two different and concurrent transactions, in a process called *certification*. During certification, conflict detection is carried out at row level: if two concurrent transactions, that executed on different servers, update the same row, then there is a conflict. The conflict resolution procedure states that the transaction that was ordered first commits on all servers, and the transaction ordered second aborts, and is therefore rolled back on the originating server and dropped by the other servers in the group.  This is in fact a distributed first commit wins rule. Note that if two transactions are bound to conflict more often than not, then it is a good practice to start them on the same server, where they have a chance to synchronize on the local lock manager instead of being rolled back as a result of certification.

Group Replication is an eventual consistency system, meaning that as soon as the incoming traffic slows down or stops, all group members have the same data content. While traffic is flowing, transactions can be externalized in a slightly different order, or externalized on some members before the others.



Group Replication operates either in single-primary mode or in multi-primary mode. The group's mode is a group-wide configuration setting, specified by the [group\_replication\_single\_primary\_mode](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_single_primary_mode) system variable, which must be the same on all members. ON means single-primary mode, which is the default mode, and OFF means multi-primary mode.

you can use the [group\_replication\_switch\_to\_single\_primary\_mode()](https://dev.mysql.com/doc/refman/8.0/en/group-replication-functions-for-mode.html#function_group-replication-switch-to-single-primary-mode) and [group\_replication\_switch\_to\_multi\_primary\_mode()](https://dev.mysql.com/doc/refman/8.0/en/group-replication-functions-for-mode.html#function_group-replication-switch-to-multi-primary-mode) functions to move a group from one mode to another while Group Replication is still running. These functions manage the process of changing the group's mode and ensure the safety and consistency of your data.

**Single-Primary Mode**

In single-primary mode ([group\_replication\_single\_primary\_mode=ON](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_single_primary_mode)) the group has a single primary server that is set to read-write mode. All the other members in the group are set to read-only mode (with [super\_read\_only=ON](https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html#sysvar_super_read_only)). The primary is typically the first server to bootstrap the group. All other servers that join the group learn about the primary server and are automatically set to read-only mode.

The member that is designated as the primary server can change in the following ways:

* If the existing primary leaves the group, whether voluntarily or unexpectedly, a new primary is elected automatically.
* You can appoint a specific member as the new primary using the [group\_replication\_set\_as\_primary()](https://dev.mysql.com/doc/refman/8.0/en/group-replication-functions-for-new-primary.html#function_group-replication-set-as-primary) function.
* If you use the [group\_replication\_switch\_to\_single\_primary\_mode()](https://dev.mysql.com/doc/refman/8.0/en/group-replication-functions-for-mode.html#function_group-replication-switch-to-single-primary-mode) function to change a group that was running in multi-primary mode to run in single-primary mode, a new primary is elected automatically, or you can appoint the new primary by specifying it with the function.

When a new primary is elected or appointed, it might have a backlog of changes that had been applied on the old primary but have not yet been applied on this server. In this situation, until the new primary catches up with the old primary, read-write transactions might result in conflicts and be rolled back, and read-only transactions might result in stale reads.  Group Replication's flow control mechanism, which minimizes the difference between fast and slow members, reduces the chances of this happening if it is activated and properly tuned. you can also use the [group\_replication\_consistency](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_consistency) system variable to configure the group's level of transaction consistency to prevent this issue. The setting BEFORE\_ON\_PRIMARY\_FAILOVER (or any higher consistency level) holds new transactions on a newly elected primary until the backlog has been applied.

Primary Election Algorithm

The automatic primary member election process involves each member looking at the new view of the group, ordering the potential new primary members, and choosing the member that qualifies as the most suitable. Each member makes its own decision locally, following the primary election algorithm in its MySQL Server release. Because all members must reach the same decision, members adapt their primary election algorithm if other group members are running lower MySQL Server versions, so that they have the same behavior as the member with the lowest MySQL Server version in the group.

The factors considered by members when electing a primary

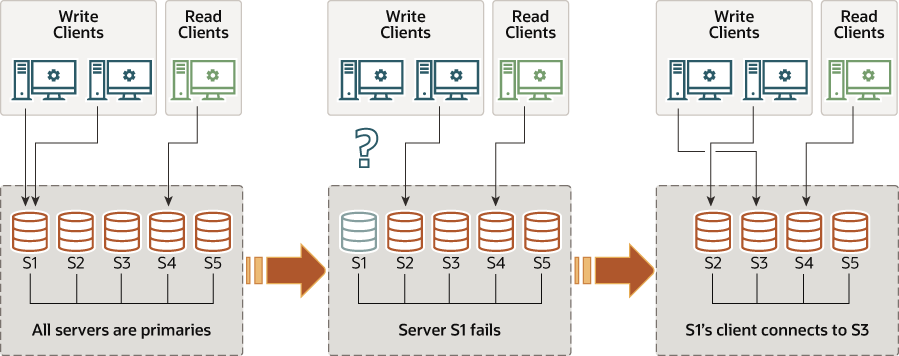
1. The first factor considered is which member or members are running the lowest MySQL Server version.

2. If more than one member is running the lowest MySQL Server version, the second factor considered is the member weight of each of those members, as specified by the [group\_replication\_member\_weight](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_member_weight) system variable on the member.

3. If more than one member is running the lowest MySQL Server version, and more than one of those members has the highest member weight (or member weighting is being ignored), the third factor considered is the lexicographical order of the generated server UUIDs of each member, as specified by the [server\_uuid](https://dev.mysql.com/doc/refman/8.0/en/replication-options.html#sysvar_server_uuid) system variable.

**Multi-Primary Mode**

In multi-primary mode ([group\_replication\_single\_primary\_mode=OFF](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_single_primary_mode)) no member has a special role. Any member that is compatible with the other group members is set to read-write mode when joining the group, and can process write transactions, even if they are issued concurrently.



Transaction Checks

When a group is deployed in multi-primary mode, transactions are checked to ensure they are compatible with the mode. The following strict consistency checks are made when Group Replication is deployed in multi-primary mode:

* If a transaction is executed under the SERIALIZABLE isolation level, then its commit fails when synchronizing itself with the group.
* If a transaction executes against a table that has foreign keys with cascading constraints, then its commit fails when synchronizing itself with the group.

The checks are controlled by the [group\_replication\_enforce\_update\_everywhere\_checks](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_enforce_update_everywhere_checks) system variable.

Data Definition Statements

In a Group Replication topology in multi-primary mode, care needs to be taken when executing data definition statements, also commonly known as data definition language (DDL).

MySQL 8.0 introduces support for atomic Data Definition Language (DDL) statements, where the complete DDL statement is either committed or rolled back as a single atomic transaction. However, DDL statements, atomic or otherwise, implicitly end any transaction that is active in the current session, as if you had done a [COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html) before executing the statement. This means that DDL statements cannot be performed within another transaction, within transaction control statements such as [START TRANSACTION ... COMMIT](https://dev.mysql.com/doc/refman/8.0/en/commit.html), or combined with other statements within the same transaction.

 If the group is deployed in single-primary mode this issue does not occur, because all changes are performed through the same server, the primary.

Version Compatibility

For optimal compatibility and performance, all members of a group should run the same version of MySQL Server and therefore of Group Replication. In multi-primary mode, this is more significant because all members would normally join the group in read-write mode. If a group includes members running more than one MySQL Server version, there is a potential for some members to be incompatible with others, because they support functions others do not, or lack functions others have. To guard against this, when a new member joins (including a former member that has been upgraded and restarted), the member carries out compatibility checks against the rest of the group.

 If a joining member is running a higher MySQL Server version than the lowest version that the existing group members are running, it joins the group but remains in read-only mode.

If a member leaves the group, the members running the version that is now the lowest are automatically set to read-write mode. When you change a group that was running in single-primary mode to run in multi-primary mode, using the [group\_replication\_switch\_to\_multi\_primary\_mode()](https://dev.mysql.com/doc/refman/8.0/en/group-replication-functions-for-mode.html#function_group-replication-switch-to-multi-primary-mode) function, Group Replication automatically sets members to the correct mode.

**Group Replication Services**

Group Membership

In MySQL Group Replication, a set of servers forms a replication group. A group has a name, which takes the form of a UUID. The group is dynamic and servers can leave (either voluntarily or involuntarily) and join it at any time. The group adjusts itself whenever servers join or leave.

Group Replication has a group membership service that defines which servers are online and participating in the group. The list of online servers is referred to as a *view*. Every server in the group has a consistent view of which servers are the members participating actively in the group at a given moment in time.

In the case where a member leaves the group voluntarily, it first initiates a dynamic group reconfiguration, during which all members have to agree on a new view without the leaving server. However, if a member leaves the group involuntarily, for example because it has stopped unexpectedly or the network connection is down, it cannot initiate the reconfiguration. In this situation, Group Replication's failure detection mechanism recognizes after a short period of time that the member has left, and a reconfiguration of the group without the failed member is proposed. As with a member that leaves voluntarily, the reconfiguration requires agreement from the majority of servers in the group. However, if the group is not able to reach agreement, for example because it partitioned in such a way that there is no majority of servers online, the system is not able to dynamically change the configuration, and blocks to prevent a split-brain situation. This situation requires intervention from an administrator.

It is possible for a member to go offline for a short time, then attempt to rejoin the group again before the failure detection mechanism has detected its failure, and before the group has been reconfigured to remove the member. In this situation, the rejoining member forgets its previous state, but if other members send it messages that are intended for its pre-crash state, this can cause issues including possible data inconsistency. If a member in this situation participates in XCom's consensus protocol, it could potentially cause XCom to deliver different values for the same consensus round, by making a different decision before and after failure.

To counter this possibility, Group Replication checks for the situation where a new incarnation of the same server is trying to join the group while its old incarnation (with the same address and port number) is still listed as a member. The new incarnation is blocked from joining the group until the old incarnation can be removed by a reconfiguration. Note that if a waiting period has been added by the [group\_replication\_member\_expel\_timeout](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_member_expel_timeout) system variable to allow additional time for members to reconnect with the group before they are expelled, a member under suspicion can become active in the group again as its current incarnation if it reconnects to the group before the suspicion times out. When a member exceeds the expel timeout and is expelled from the group, or when Group Replication is stopped on the server by a [STOP GROUP\_REPLICATION](https://dev.mysql.com/doc/refman/8.0/en/stop-group-replication.html) statement or a server failure, it must rejoin as a new incarnation.

Failure Detection

Group Replication’s failure detection mechanism is a distributed service which is able to identify that a server in the group is not communicating with the others, and is therefore suspected of being out of service. If the group’s consensus is that the suspicion is probably true, the group takes a coordinated decision to expel the member. Expelling a member that is not communicating is necessary because the group needs a majority of its members to agree on a transaction or view change. If a member is not participating in these decisions, the group must remove it to increase the chance that the group contains a majority of correctly working members, and can therefore continue to process transactions.

In a replication group, each member has a point-to-point communication channel to each other member, creating a fully connected graph. These connections are managed by the group communication engine (XCom, a Paxos variant) and use TCP/IP sockets. One channel is used to send messages to the member and the other channel is used to receive messages from the member. If a member does not receive messages from another member for 5 seconds, it suspects that the member has failed, and lists the status of that member as UNREACHABLE in its own Performance Schema table [replication\_group\_members](https://dev.mysql.com/doc/refman/8.0/en/performance-schema-replication-group-members-table.html).

If a suspicion lasts for more than 10 seconds, the suspecting member tries to propagate its view that the suspect member is faulty to the other members of the group. A suspecting member only does this if it is a notifier, as calculated from its internal XCom node number.  A suspicion only has consequences if a member is a notifier, and its suspicion lasts long enough to be propagated to the other members of the group, and the other members agree on it. In that case, the suspect member is marked for expulsion from the group in a coordinated decision, and is expelled after the waiting period set by the [group\_replication\_member\_expel\_timeout](https://dev.mysql.com/doc/refman/8.0/en/group-replication-system-variables.html#sysvar_group_replication_member_expel_timeout) system variable expires and the expelling mechanism detects and implements the expulsion.

Where the network is unstable and members frequently lose and regain connection to each other in different combinations, it is theoretically possible for a group to end up marking all its members for expulsion, after which the group would cease to exist and have to be set up again.   To counter this possibility, from MySQL 8.0.20, Group Replication's Group Communication System (GCS) tracks the group members that have been marked for expulsion, and treats them as if they were in the group of suspected members when deciding if there is a majority. This ensures at least one member remains in the group and the group can continue to exist. When an expelled member has actually been removed from the group, GCS removes its record of having marked the member for expulsion, so that the member can rejoin the group if it is able to.

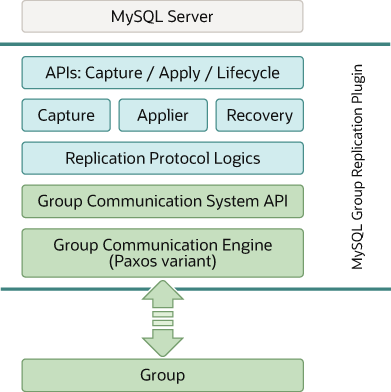
Fault-tolerance

MySQL Group Replication builds on an implementation of the Paxos distributed algorithm to provide distributed coordination between servers. As such, it requires a majority of servers to be active to reach quorum and thus make a decision. This has direct impact on the number of failures the system can tolerate without compromising itself and its overall functionality. The number of servers (n) needed to tolerate f failures is then n = 2 x f + 1.

Observability

The entire state of the system (including the view, conflict statistics and service states) can be queried through Performance Schema tables. The distributed nature of the replication protocol and the fact that server instances agree and thus synchronize on transactions and metadata makes it simpler to inspect the state of the group.

Group Replication Plugin Architecture

****

The MySQL Group Replication plugin includes a set of APIs for capture, apply, and lifecycle, which control how the plugin interacts with MySQL Server. There are interfaces to make information flow from the server to the plugin and vice versa. These interfaces isolate the MySQL Server core from the Group Replication plugin, and are mostly hooks placed in the transaction execution pipeline.

 In one direction, from server to the plugin, there are notifications for events such as the server starting, the server recovering, the server being ready to accept connections, and the server being about to commit a transaction. In the other direction, the plugin instructs the server to perform actions such as committing or aborting ongoing transactions, or queuing transactions in the relay log.

The next layer of the Group Replication plugin architecture is a set of components that react when a notification is routed to them. The capture component is responsible for keeping track of context related to transactions that are executing. The applier component is responsible for executing remote transactions on the database. The recovery component manages distributed recovery, and is responsible for getting a server that is joining the group up to date by selecting the donor, managing the catch up procedure and reacting to donor failures.

Continuing down the stack, the replication protocol module contains the specific logic of the replication protocol. It handles conflict detection, and receives and propagates transactions to the group.

The final two layers of the Group Replication plugin architecture are the Group Communication System (GCS) API, and an implementation of a Paxos-based group communication engine (XCom). The GCS API is a high level API that abstracts the properties required to build a replicated state machine  It therefore decouples the implementation of the messaging layer from the remaining upper layers of the plugin. The group communication engine handles communications with the members of the replication group.

----------------------------------------------------------------------------------------------------------------------------

Mysql InnoDb Cluster

