300 milliseconds, and the maximum possible value is 4294967295 milliseconds (approximately 49.7 days).

- `slave_compressed_protocol`

Command-Line Format	`--slave-compressed-protocol[={OFF|ON}]`
Deprecated	8.0.18
System Variable	`slave_compressed_protocol`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

`slave_compressed_protocol` is deprecated, and from MySQL 8.0.26, the alias `replica_compressed_protocol` should be used instead. In releases before MySQL 8.0.26, use `slave_compressed_protocol`.

`slave_compressed_protocol` controls whether to use compression of the source/replica connection protocol if both source and replica support it. If this variable is disabled (the default), connections are uncompressed. Changes to this variable take effect on subsequent connection attempts; this includes after issuing a `START REPLICA` statement, as well as reconnections made by a running replication I/O (receiver) thread.

Binary log transaction compression (available as of MySQL 8.0.20), which is activated by the `binlog_transaction_compression` system variable, can also be used to save bandwidth. If you use binary log transaction compression in combination with protocol compression, protocol compression has less opportunity to act on the data, but can still compress headers and those events and transaction payloads that are uncompressed. For more information on binary log transaction compression, see Section 5.4.4.5, “Binary Log Transaction Compression”.

As of MySQL 8.0.18, if `slave_compressed_protocol` is enabled, it takes precedence over any `SOURCE_COMPRESSION_ALGORITHMS` | `MASTER_COMPRESSION_ALGORITHMS` option specified for the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement. In this case, connections to the source use `zlib` compression if both the source and replica support that algorithm. If `slave_compressed_protocol` is disabled, the value of `SOURCE_COMPRESSION_ALGORITHMS` | `MASTER_COMPRESSION_ALGORITHMS` applies. For more information, see Section 4.2.8, “Connection Compression Control”.

As of MySQL 8.0.18, this system variable is deprecated. You should expect it to be removed in a future version of MySQL. See Configuring Legacy Connection Compression.

- `slave_exec_mode`

Command-Line Format	`--slave-exec-mode=mode`
System Variable	`slave_exec_mode`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`IDEMPOTENT` (NDB) `STRICT` (Other)

Valid Values	`STRICT`
	
	`IDEMPOTENT`

From MySQL 8.0.26, `slave_exec_mode` is deprecated and the alias `replica_exec_mode` should be used instead. In releases before MySQL 8.0.26, use `slave_exec_mode`.

`slave_exec_mode` controls how a replication thread resolves conflicts and errors during replication. `IDEMPOTENT` mode causes suppression of duplicate-key and no-key-found errors; `STRICT` means no such suppression takes place.

`IDEMPOTENT` mode is intended for use in multi-source replication, circular replication, and some other special replication scenarios for NDB Cluster Replication. (See Section 23.7.10, “NDB Cluster Replication: Bidirectional and Circular Replication”, and Section 23.7.12, “NDB Cluster Replication Conflict Resolution”, for more information.) NDB Cluster ignores any value explicitly set for `slave_exec_mode`, and always treats it as `IDEMPOTENT`.

In MySQL Server 8.0, `STRICT` mode is the default value.

Setting this variable takes immediate effect for all replication channels, including running channels.

For storage engines other than `NDB`, *`IDEMPOTENT` mode should be used only when you are absolutely sure that duplicate-key errors and key-not-found errors can safely be ignored*. It is meant to be used in fail-over scenarios for NDB Cluster where multi-source replication or circular replication is employed, and is not recommended for use in other cases.

- `slave_load_tmpdir`

Command-Line Format	`--slave-load-tmpdir=dir_name`
Deprecated	8.0.26
System Variable	`slave_load_tmpdir`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Directory name
Default Value	`Value of --tmpdir`

From MySQL 8.0.26, `slave_load_tmpdir` is deprecated and the alias `replica_load_tmpdir` should be used instead. In releases before MySQL 8.0.26, use `slave_load_tmpdir`.

`slave_load_tmpdir` specifies the name of the directory where the replica creates temporary files. Setting this variable takes effect for all replication channels immediately, including running channels. The variable value is by default equal to the value of the `tmpdir` system variable, or the default that applies when that system variable is not specified.

When the replication SQL thread replicates a `LOAD DATA` statement, it extracts the file to be loaded from the relay log into temporary files, and then loads these into the table. If the file loaded on the source is huge, the temporary files on the replica are huge, too. Therefore, it might be advisable to use this option to tell the replica to put temporary files in a directory located in some file system that has a lot of available space. In that case, the relay logs are huge as well, so you might also want to set the `relay_log` system variable to place the relay logs in that file system.

The directory specified by this option should be located in a disk-based file system (not a memory-based file system) so that the temporary files used to replicate `LOAD DATA` statements can survive machine restarts. The directory also should not be one that is cleared by the operating system during the system startup process. However, replication can now continue after a restart if the temporary files have been removed.

- `slave_max_allowed_packet`

Command-Line Format	`--slave-max-allowed-packet=#`
Deprecated	8.0.26
System Variable	`slave_max_allowed_packet`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1073741824`
Minimum Value	`1024`
Maximum Value	`1073741824`
Unit	bytes
Block Size	`1024`

From MySQL 8.0.26, `slave_max_allowed_packet` is deprecated and the alias `replica_max_allowed_packet` should be used instead. In releases before MySQL 8.0.26, use `slave_max_allowed_packet`.

`slave_max_allowed_packet` sets the maximum packet size in bytes that the replication SQL (applier) and I/O (receiver) threads can handle. Setting this variable takes effect for all replication channels immediately, including running channels. It is possible for a source to write binary log events longer than its `max_allowed_packet` setting once the event header is added. The setting for `slave_max_allowed_packet` must be larger than the `max_allowed_packet` setting on the source, so that large updates using row-based replication do not cause replication to fail.

This global variable always has a value that is a positive integer multiple of 1024; if you set it to some value that is not, the value is rounded down to the next highest multiple of 1024 for it is stored or used; setting `slave_max_allowed_packet` to 0 causes 1024 to be used. (A truncation warning is issued in all such cases.) The default and maximum value is 1073741824 (1 GB); the minimum is 1024.

- `slave_net_timeout`

Command-Line Format	`--slave-net-timeout=#`
Deprecated	8.0.26
System Variable	`slave_net_timeout`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`60`
Minimum Value	`1`
Maximum Value	`31536000`
Unit	seconds

From MySQL 8.0.26, `slave_net_timeout` is deprecated and the alias `replica_net_timeout` should be used instead. In releases before MySQL 8.0.26, use `slave_net_timeout`.

`slave_net_timeout` specifies the number of seconds to wait for more data or a heartbeat signal from the source before the replica considers the connection broken, aborts the read, and tries to

reconnect. Setting this variable has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` commands.

The default value is 60 seconds (one minute). The first retry occurs immediately after the timeout. The interval between retries is controlled by the `SOURCE_CONNECT_RETRY` | `MASTER_CONNECT_RETRY` option for the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement, and the number of reconnection attempts is limited by the `SOURCE_RETRY_COUNT` | `MASTER_RETRY_COUNT` option.

The heartbeat interval, which stops the connection timeout occurring in the absence of data if the connection is still good, is controlled by the `SOURCE_HEARTBEAT_PERIOD` | `MASTER_HEARTBEAT_PERIOD` option for the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement. The heartbeat interval defaults to half the value of `slave_net_timeout`, and it is recorded in the replica's connection metadata repository and shown in the `replication_connection_configuration` Performance Schema table. Note that a change to the value or default setting of `slave_net_timeout` does not automatically change the heartbeat interval, whether that has been set explicitly or is using a previously calculated default. If the connection timeout is changed, you must also issue `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` to adjust the heartbeat interval to an appropriate value so that it occurs before the connection timeout.

- `slave_parallel_type`

Command-Line Format	`--slave-parallel-type=value`
Deprecated	8.0.26
System Variable	`slave_parallel_type`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value (≥ 8.0.27)	`LOGICAL_CLOCK`
Default Value (≤ 8.0.26)	`DATABASE`

Valid Values	`DATABASE`
	
	`LOGICAL_CLOCK`

From MySQL 8.0.26, `slave_parallel_type` is deprecated and the alias `replica_parallel_type` should be used instead. In releases before MySQL 8.0.26, use `slave_parallel_type`.

For multithreaded replicas (replicas on which `replica_parallel_workers` or `slave_parallel_workers` is set to a value greater than 0), `slave_parallel_type` specifies the policy used to decide which transactions are allowed to execute in parallel on the replica. The variable has no effect on replicas for which multithreading is not enabled. The possible values are:

- `LOGICAL_CLOCK`: Transactions that are part of the same binary log group commit on a source are applied in parallel on a replica. The dependencies between transactions are tracked based on their timestamps to provide additional parallelization where possible. When this value is set, the `binlog_transaction_dependency_tracking` system variable can be used on the source to specify that write sets are used for parallelization in place of timestamps, if a write set is available for the transaction and gives improved results compared to timestamps.
- `DATABASE`: Transactions that update different databases are applied in parallel. This value is only appropriate if data is partitioned into multiple databases which are being updated independently and concurrently on the source. There must be no cross-database constraints, as such constraints may be violated on the replica.

When `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` is set, you can only use `LOGICAL_CLOCK`. Before MySQL 8.0.27, `DATABASE` is the default. From MySQL 8.0.27, multithreading is enabled by default for replica servers (`replica_parallel_workers=4` by default), so `LOGICAL_CLOCK` is the default, and the setting `replica_preserve_commit_order=ON` is also the default.

When your replication topology uses multiple levels of replicas, `LOGICAL_CLOCK` may achieve less parallelization for each level the replica is away from the source. You can reduce this effect by using `binlog_transaction_dependency_tracking` on the source to specify that write sets are used instead of timestamps for parallelization where possible.

When binary log transaction compression is enabled using the `binlog_transaction_compression` system variable, if `replica_parallel_type` or `slave_parallel_type` is set to `DATABASE`, all the databases affected by the transaction are mapped before the transaction is scheduled. The use of binary log transaction compression with the `DATABASE` policy can reduce parallelism compared to uncompressed transactions, which are mapped and scheduled for each event.

- `slave_parallel_workers`

Command-Line Format	`--slave-parallel-workers=#`
Deprecated	8.0.26
System Variable	`slave_parallel_workers`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.27)	`4`
Default Value (≤ 8.0.26)	`0`
Minimum Value	`0`
Maximum Value	`1024`

From MySQL 8.0.26, `slave_parallel_workers` is deprecated and the alias `replica_parallel_workers` should be used instead. In releases before MySQL 8.0.26, use `slave_parallel_workers`.

`slave_parallel_workers` enables multithreading on the replica and sets the number of applier threads for executing replication transactions in parallel. When the value is a number greater than 0, the replica is a multithreaded replica with the specified number of applier threads, plus a coordinator thread to manage them. If you are using multiple replication channels, each channel has this number of threads.

Before MySQL 8.0.27, the default for this system variable is 0, so replicas are not multithreaded by default. From MySQL 8.0.27, the default is 4, so replicas are multithreaded by default.

Retrying of transactions is supported when multithreading is enabled on a replica. When `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` is set, transactions on a replica are externalized on the replica in the same order as they appear in the replica's relay log. The way in which transactions are distributed among applier threads is configured by `replica_parallel_type` (from MySQL 8.0.26) or `slave_parallel_type`

(before MySQL 8.0.26). From MySQL 8.0.27, these system variables also have appropriate defaults for multithreading.

To disable parallel execution, set `replica_parallel_workers` to 0, which gives the replica a single applier thread and no coordinator thread. With this setting, the `replica_parallel_type` or `slave_parallel_type` and `replica_preserve_commit_order` or `slave_preserve_commit_order` system variables have no effect and are ignored. From MySQL 8.0.27, if parallel execution is disabled when the `CHANGE REPLICATION SOURCE TO` option `GTID_ONLY` is enabled on a replica, the replica actually uses one parallel worker to take advantage of the method for retrying transactions without accessing the file positions. With one parallel worker, the `replica_preserve_commit_order` (`slave_preserve_commit_order`) system variable also has no effect.

Setting `replica_parallel_workers` has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` statements.

Multithreaded replicas are not currently supported by NDB Cluster. See Section 23.7.3, "Known Issues in NDB Cluster Replication", for more information about how `NDB` handles settings for this variable.

- `slave_pending_jobs_size_max`

Command-Line Format	`--slave-pending-jobs-size-max=#`
Deprecated	8.0.26
System Variable	`slave_pending_jobs_size_max`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.12)	`128M`
Default Value (8.0.11)	`16M`
Minimum Value	`1024`
Maximum Value	`16EiB`
Unit	bytes
Block Size	`1024`

From MySQL 8.0.26, `slave_pending_jobs_size_max` is deprecated and the alias `replica_pending_jobs_size_max` should be used instead. In releases before MySQL 8.0.26, use `slave_pending_jobs_size_max`.

For multithreaded replicas, this variable sets the maximum amount of memory (in bytes) available to applier queues holding events not yet applied. Setting this variable has no effect on replicas for which multithreading is not enabled. Setting this variable has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` commands.

The minimum possible value for this variable is 1024 bytes; the default is 128MB. The maximum possible value is 18446744073709551615 (16 exbibytes). Values that are not exact multiples of 1024 bytes are rounded down to the next lower multiple of 1024 bytes prior to being stored.

The value of this variable is a soft limit and can be set to match the normal workload. If an unusually large event exceeds this size, the transaction is held until all the worker threads have empty queues, and then processed. All subsequent transactions are held until the large transaction has been completed.

- `slave_preserve_commit_order`

Command-Line Format	`--slave-preserve-commit-order[={OFF|ON}]`
Deprecated	8.0.26
System Variable	`slave_preserve_commit_order`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value (≥ 8.0.27)	`ON`
Default Value (≤ 8.0.26)	`OFF`

From MySQL 8.0.26, `slave_preserve_commit_order` is deprecated and the alias `replica_preserve_commit_order` should be used instead. In releases before MySQL 8.0.26, use `slave_preserve_commit_order`.

For multithreaded replicas (replicas on which `replica_parallel_workers` or `slave_parallel_workers` is set to a value greater than 0), setting `slave_preserve_commit_order=1` ensures that transactions are executed and committed on the replica in the same order as they appear in the replica's relay log. This prevents gaps in the sequence of transactions that have been executed from the replica's relay log, and preserves the same transaction history on the replica as on the source (with the limitations listed below). This variable has no effect on replicas for which multithreading is not enabled.

Before MySQL 8.0.27, the default for this system variable is `OFF`, meaning that transactions may be committed out of order. From MySQL 8.0.27, multithreading is enabled by default for replica servers (`replica_parallel_workers=4` by default), so `slave_preserve_commit_order=ON` is the default, and the setting `slave_parallel_type=LOGICAL_CLOCK` is also the default. Also from MySQL 8.0.27, the setting for `slave_preserve_commit_order` is ignored if `slave_parallel_workers` is set to 1, because in that situation the order of transactions is preserved anyway.

Up to and including MySQL 8.0.18, setting `slave_preserve_commit_order=ON` requires that binary logging (`log_bin`) and replica update logging (`log_slave_updates`) are enabled on the replica, which are the default settings from MySQL 8.0. From MySQL 8.0.19, binary logging and replica update logging are not required on the replica to set `slave_preserve_commit_order=ON`, and can be disabled if wanted. In all releases, setting `slave_preserve_commit_order=ON` requires that `slave_parallel_type` is set to `LOGICAL_CLOCK`, which is *not* the default setting before MySQL 8.0.27. Before changing the value of `slave_preserve_commit_order` and `slave_parallel_type`, the replication SQL thread (for all replication channels if you are using multiple replication channels) must be stopped.

When `slave_preserve_commit_order=OFF` is set, which is the default, the transactions that a multithreaded replica applies in parallel may commit out of order. Therefore, checking for the most recently executed transaction does not guarantee that all previous transactions from the source have been executed on the replica. There is a chance of gaps in the sequence of transactions that have been executed from the replica's relay log. This has implications for logging and recovery when using a multithreaded replica. See Section 17.5.1.34, “Replication and Transaction Inconsistencies” for more information.

When `slave_preserve_commit_order=ON` is set, the executing worker thread waits until all previous transactions are committed before committing. While a given thread is waiting for other worker threads to commit their transactions, it reports its status as `Waiting for preceding transaction to commit`. With this mode, a multithreaded replica never enters a state that the

source was not in. This supports the use of replication for read scale-out. See Section 17.4.5, “Using Replication for Scale-Out”.

Note

- `slave_preserve_commit_order=ON` does not prevent source binary log position lag, where `Exec_master_log_pos` is behind the position up to which transactions have been executed. See Section 17.5.1.34, “Replication and Transaction Inconsistencies”.
- `slave_preserve_commit_order=ON` does not preserve the commit order and transaction history if the replica uses filters on its binary log, such as `--binlog-do-db`.
- `slave_preserve_commit_order=ON` does not preserve the order of non-transactional DML updates. These might commit before transactions that precede them in the relay log, which might result in gaps in the sequence of transactions that have been executed from the replica's relay log.
- In releases before MySQL 8.0.19, `slave_preserve_commit_order=ON` does not preserve the order of statements with an `IF EXISTS` clause when the object concerned does not exist. These might commit before transactions that precede them in the relay log, which might result in gaps in the sequence of transactions that have been executed from the replica's relay log.
- A limitation to preserving the commit order on the replica can occur if statement-based replication is in use, and both transactional and non-transactional storage engines participate in a non-XA transaction that is rolled back on the source. Normally, non-XA transactions that are rolled back on the source are not replicated to the replica, but in this particular situation, the transaction might be replicated to the replica. If this does happen, a multithreaded replica without binary logging does not handle the transaction rollback, so the commit order on the replica diverges from the relay log order of the transactions in that case.

- `slave_rows_search_algorithms`

Command-Line Format	`--slave-rows-search-algorithms=value`
Deprecated	8.0.18
System Variable	`slave_rows_search_algorithms`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Set
Default Value	`INDEX_SCAN,HASH_SCAN`

Valid Values	`TABLE_SCAN,INDEX_SCAN`
	`INDEX_SCAN,HASH_SCAN`
	`TABLE_SCAN,HASH_SCAN`

	`TABLE_SCAN,INDEX_SCAN,HASH_SCAN` (equivalent to INDEX_SCAN,HASH_SCAN)

When preparing batches of rows for row-based logging and replication, this system variable controls how the rows are searched for matches, in particular whether hash scans are used. The use of this system variable is now deprecated. The default setting `INDEX_SCAN,HASH_SCAN` is optimal for performance and works correctly in all scenarios. See Section 17.5.1.27, "Replication and Row Searches".

- `slave_skip_errors`

Command-Line Format	`--slave-skip-errors=name`
Deprecated	8.0.26
System Variable	`slave_skip_errors`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	String
Default Value	`OFF`

Valid Values	`OFF`
	`[list of error codes]`
	`all`
	`ddl_exist_errors`

From MySQL 8.0.26, `slave_skip_errors` is deprecated and the alias `replica_skip_errors` should be used instead. In releases before MySQL 8.0.26, use `slave_skip_errors`.

Normally, replication stops when an error occurs on the replica, which gives you the opportunity to resolve the inconsistency in the data manually. This variable causes the replication SQL thread to continue replication when a statement returns any of the errors listed in the variable value.

- `replica_skip_errors`

Command-Line Format	`--replica-skip-errors=name`
Introduced	8.0.26
System Variable	`replica_skip_errors`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	String
Default Value	`OFF`

Valid Values	`OFF`
	`[list of error codes]`
	`all`

	`ddl_exist_errors`

From MySQL 8.0.26, use `replica_skip_errors` in place of `slave_skip_errors`, which is deprecated from that release. In releases before MySQL 8.0.26, use `slave_skip_errors`.

Normally, replication stops when an error occurs on the replica, which gives you the opportunity to resolve the inconsistency in the data manually. This variable causes the replication SQL thread to continue replication when a statement returns any of the errors listed in the variable value.

- `slave_sql_verify_checksum`

Command-Line Format	`--slave-sql-verify-checksum[={OFF|ON}]`
Deprecated	8.0.26
System Variable	`slave_sql_verify_checksum`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

From MySQL 8.0.26, `slave_sql_verify_checksum` is deprecated and the alias `replica_sql_verify_checksum` should be used instead. In releases before MySQL 8.0.26, use `slave_sql_verify_checksum`.

`slave_sql_verify_checksum` causes the replication SQL thread to verify data using the checksums read from the relay log. In the event of a mismatch, the replica stops with an error. Setting this variable takes effect for all replication channels immediately, including running channels.

Note

The replication I/O (receiver) thread always reads checksums if possible when accepting events from over the network.

- `slave_transaction_retries`

Command-Line Format	`--slave-transaction-retries=#`
Deprecated	8.0.26
System Variable	`slave_transaction_retries`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`10`
Minimum Value	`0`
Maximum Value (64-bit platforms)	`18446744073709551615`

Maximum Value (32-bit platforms)	4294967295

From MySQL 8.0.26, slave_transaction_retries is deprecated and the alias replica_transaction_retries should be used instead. In releases before MySQL 8.0.26, use slave_transaction_retries.

slave_transaction_retries sets the maximum number of times for replication SQL threads on a single-threaded or multithreaded replica to automatically retry failed transactions before stopping. Setting this variable takes effect for all replication channels immediately, including running channels. The default value is 10. Setting the variable to 0 disables automatic retrying of transactions.

If a replication SQL thread fails to execute a transaction because of an InnoDB deadlock or because the transaction's execution time exceeded InnoDB's innodb_lock_wait_timeout or NDB's TransactionDeadlockDetectionTimeout or TransactionInactiveTimeout, it automatically retries slave_transaction_retries times before stopping with an error. Transactions with a non-temporary error are not retried.

The Performance Schema table replication_applier_status shows the number of retries that took place on each replication channel, in the COUNT_TRANSACTIONS_RETRIES column. The Performance Schema table replication_applier_status_by_worker shows detailed information on transaction retries by individual applier threads on a single-threaded or multithreaded replica, and identifies the errors that caused the last transaction and the transaction currently in progress to be reattempted.

- slave_type_conversions

Command-Line Format	--slave-type-conversions=set
Deprecated	8.0.26
System Variable	slave_type_conversions
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Set
Default Value	
Valid Values	ALL_LOSSY ALL_NON_LOSSY ALL_SIGNED ALL_UNSIGNED

From MySQL 8.0.26, slave_type_conversions is deprecated and the alias replica_type_conversions should be used instead. In releases before MySQL 8.0.26, use slave_type_conversions.

slave_type_conversions controls the type conversion mode in effect on the replica when using row-based replication. Its value is a comma-delimited set of zero or more elements from the list: ALL_LOSSY, ALL_NON_LOSSY, ALL_SIGNED, ALL_UNSIGNED. Set this variable to an empty string to disallow type conversions between the source and the replica. Setting this variable takes effect for all replication channels immediately, including running channels.

For additional information on type conversion modes applicable to attribute promotion and demotion in row-based replication, see Row-based replication: attribute promotion and demotion.

- `sql_replica_skip_counter`

Introduced	8.0.26
System Variable	`sql_replica_skip_counter`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`4294967295`

From MySQL 8.0.26, use `sql_replica_skip_counter` in place of `sql_slave_skip_counter`, which is deprecated from that release. In releases before MySQL 8.0.26, use `sql_slave_skip_counter`.

`sql_replica_skip_counter` specifies the number of events from the source that a replica should skip. Setting the option has no immediate effect. The variable applies to the next `START REPLICA` statement; the next `START REPLICA` statement also changes the value back to 0. When this variable is set to a nonzero value and there are multiple replication channels configured, the `START REPLICA` statement can only be used with the `FOR CHANNEL` *`channel`* clause.

This option is incompatible with GTID-based replication, and must not be set to a nonzero value when `gtid_mode=ON` is set. If you need to skip transactions when employing GTIDs, use `gtid_executed` from the source instead. If you have enabled GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement, `sql_replica_skip_counter` is available. See Section 17.1.7.3, “Skipping Transactions”.

Important

If skipping the number of events specified by setting this variable would cause the replica to begin in the middle of an event group, the replica continues to skip until it finds the beginning of the next event group and begins from that point. For more information, see Section 17.1.7.3, “Skipping Transactions”.

- `sql_slave_skip_counter`

Deprecated	8.0.26
System Variable	`sql_slave_skip_counter`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`

Maximum Value	4294967295

From MySQL 8.0.26, sql_slave_skip_counter is deprecated and the alias sql_replica_skip_counter should be used instead. In releases before MySQL 8.0.26, use sql_slave_skip_counter.

sql_slave_skip_counter specifies the number of events from the source that a replica should skip. Setting the option has no immediate effect. The variable applies to the next START REPLICA statement; the next START REPLICA statement also changes the value back to 0. When this variable is set to a nonzero value and there are multiple replication channels configured, the START REPLICA statement can only be used with the FOR CHANNEL *channel* clause.

This option is incompatible with GTID-based replication, and must not be set to a nonzero value when gtid_mode=ON is set. If you need to skip transactions when employing GTIDs, use gtid_executed from the source instead. If you have enabled GTID assignment on a replication channel using the ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS option of the CHANGE REPLICATION SOURCE TO statement, sql_slave_skip_counter is available. See Section 17.1.7.3, “Skipping Transactions”.

Important

If skipping the number of events specified by setting this variable would cause the replica to begin in the middle of an event group, the replica continues to skip until it finds the beginning of the next event group and begins from that point. For more information, see Section 17.1.7.3, “Skipping Transactions”.

- sync_master_info

Command-Line Format	--sync-master-info=#
Deprecated	8.0.26
System Variable	sync_master_info
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000
Minimum Value	0
Maximum Value	4294967295

From MySQL 8.0.26, sync_master_info is deprecated and the alias sync_source_info should be used instead. In releases before MySQL 8.0.26, use sync_master_info.

sync_master_info specifies the number of events after which the replica updates the connection metadata repository. When the connection metadata repository is stored as an InnoDB table, which is the default from MySQL 8.0, it is updated after this number of events. If the connection metadata repository is stored as a file, which is deprecated from MySQL 8.0, the replica synchronizes its master.info file to disk (using fdatasync()) after this number of events. The default value is 10000, and a zero value means that the repository is never updated. Setting this variable takes effect for all replication channels immediately, including running channels.

- sync_relay_log

Command-Line Format	--sync-relay-log=#
System Variable	sync_relay_log
Scope	Global

Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000
Minimum Value	0
Maximum Value	4294967295

If the value of this variable is greater than 0, the MySQL server synchronizes its relay log to disk (using fdatasync()) after every sync_relay_log events are written to the relay log. Setting this variable takes effect for all replication channels immediately, including running channels.

Setting sync_relay_log to 0 causes no synchronization to be done to disk; in this case, the server relies on the operating system to flush the relay log's contents from time to time as for any other file.

A value of 1 is the safest choice because in the event of an unexpected halt you lose at most one event from the relay log. However, it is also the slowest choice (unless the disk has a battery-backed cache, which makes synchronization very fast). For information on the combination of settings on a replica that is most resilient to unexpected halts, see Section 17.4.2, “Handling an Unexpected Halt of a Replica”.

- sync_relay_log_info

Command-Line Format	--sync-relay-log-info=#
System Variable	sync_relay_log_info
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000
Minimum Value	0
Maximum Value	4294967295

The number of transactions after which the replica updates the applier metadata repository. When the applier metadata repository is stored as an InnoDB table, which is the default from MySQL 8.0, it is updated after every transaction and this system variable is ignored. If the applier metadata repository is stored as a file, which is deprecated from MySQL 8.0, the replica synchronizes its relay-log.info file to disk (using fdatasync()) after this number of transactions. The default value for sync_relay_log_info is 10000, and a zero value means that the file contents are only flushed by the operating system. Setting this variable takes effect for all replication channels immediately, including running channels.

- sync_source_info

Command-Line Format	--sync-source-info=#
Introduced	8.0.26
System Variable	sync_source_info
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000

Minimum Value	0
Maximum Value	4294967295

From MySQL 8.0.26, use sync_source_info in place of sync_master_info, which is deprecated from that release. In releases before MySQL 8.0.26, use sync_source_info.

sync_source_info specifies the number of events after which the replica updates the connection metadata repository. When the connection metadata repository is stored as an InnoDB table, which is the default from MySQL 8.0, it is updated after this number of events. If the connection metadata repository is stored as a file, which is deprecated from MySQL 8.0, the replica synchronizes its master.info file to disk (using fdatasync()) after this number of events. The default value is 10000, and a zero value means that the repository is never updated. Setting this variable takes effect for all replication channels immediately, including running channels.

- terminology_use_previous

Command-Line Format	--terminology-use-previous=#
Introduced	8.0.26
System Variable	terminology_use_previous
Scope	Global, Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	NONE
Valid Values	NONE BEFORE_8_0_26

In MySQL 8.0.26, incompatible changes were made to instrumentation names containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the terminology_use_previous system variable to BEFORE_8_0_26 to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

Set the terminology_use_previous system variable with session scope to support individual functions, or global scope to be a default for all new sessions. When global scope is used, the slow query log contains the old versions of the names.

The affected instrumentation names are given in the following list. The terminology_use_previous system variable only affects these items. It does not affect the new aliases for system variables, status variables, and command-line options that were also introduced in MySQL 8.0.26, and these can still be used when it is set.

- Instrumented locks (mutexes), visible in the mutex_instances and events_waits_* Performance Schema tables with the prefix wait/synch/mutex/
- Read/write locks, visible in the rwlock_instances and events_waits_* Performance Schema tables with the prefix wait/synch/rwlock/
- Instrumented condition variables, visible in the cond_instances and events_waits_* Performance Schema tables with the prefix wait/synch/cond/

- Instrumented memory allocations, visible in the `memory_summary_*` Performance Schema tables with the prefix `memory/sql/`
- Thread names, visible in the `threads` Performance Schema table with the prefix `thread/sql/`
- Thread stages, visible in the `events_stages_*` Performance Schema tables with the prefix `stage/sql/`, and without the prefix in the `threads` and `processlist` Performance Schema tables, the output from the `SHOW PROCESSLIST` statement, the Information Schema `processlist` table, and the slow query log
- Thread commands, visible in the `events_statements_history*` and `events_statements_summary_*_by_event_name` Performance Schema tables with the prefix `statement/com/`, and without the prefix in the `threads` and `processlist` Performance Schema tables, the output from the `SHOW PROCESSLIST` statement, the Information Schema `processlist` table, and the output from the `SHOW REPLICA STATUS` statement

17.1.6.4 Binary Logging Options and Variables

- Startup Options Used with Binary Logging
- System Variables Used with Binary Logging

You can use the `mysqld` options and system variables that are described in this section to affect the operation of the binary log as well as to control which statements are written to the binary log. For additional information about the binary log, see Section 5.4.4, “The Binary Log”. For additional information about using MySQL server options and system variables, see Section 5.1.7, “Server Command Options”, and Section 5.1.8, “Server System Variables”.

Startup Options Used with Binary Logging

The following list describes startup options for enabling and configuring the binary log. System variables used with binary logging are discussed later in this section.

- `--binlog-row-event-max-size=N`

Command-Line Format	`--binlog-row-event-max-size=#`
System Variable (≥ 8.0.14)	`binlog_row_event_max_size`
Scope (≥ 8.0.14)	Global
Dynamic (≥ 8.0.14)	No
`SET_VAR` Hint Applies (≥ 8.0.14)	No
Type	Integer
Default Value	`8192`
Minimum Value	`256`
Maximum Value (64-bit platforms)	`18446744073709551615`
Maximum Value (32-bit platforms)	`4294967295`
Unit	bytes

When row-based binary logging is used, this setting is a soft limit on the maximum size of a row-based binary log event, in bytes. Where possible, rows stored in the binary log are grouped into events with a size not exceeding the value of this setting. If an event cannot be split, the maximum size can be exceeded. The value must be (or else gets rounded down to) a multiple of 256. The default is 8192 bytes.

- `--log-bin[=base_name]`

Command-Line Format	`--log-bin=file_name`

Type	File name

Specifies the base name to use for binary log files. With binary logging enabled, the server logs all statements that change data to the binary log, which is used for backup and replication. The binary log is a sequence of files with a base name and numeric extension. The `--log-bin` option value is the base name for the log sequence. The server creates binary log files in sequence by adding a numeric suffix to the base name.

If you do not supply the `--log-bin` option, MySQL uses `binlog` as the default base name for the binary log files. For compatibility with earlier releases, if you supply the `--log-bin` option with no string or with an empty string, the base name defaults to *`host_name`*`-bin`, using the name of the host machine.

The default location for binary log files is the data directory. You can use the `--log-bin` option to specify an alternative location, by adding a leading absolute path name to the base name to specify a different directory. When the server reads an entry from the binary log index file, which tracks the binary log files that have been used, it checks whether the entry contains a relative path. If it does, the relative part of the path is replaced with the absolute path set using the `--log-bin` option. An absolute path recorded in the binary log index file remains unchanged; in such a case, the index file must be edited manually to enable a new path or paths to be used. The binary log file base name and any specified path are available as the `log_bin_basename` system variable.

In earlier MySQL versions, binary logging was disabled by default, and was enabled if you specified the `--log-bin` option. From MySQL 8.0, binary logging is enabled by default, whether or not you specify the `--log-bin` option. The exception is if you use `mysqld` to initialize the data directory manually by invoking it with the `--initialize` or `--initialize-insecure` option, when binary logging is disabled by default. It is possible to enable binary logging in this case by specifying the `--log-bin` option. When binary logging is enabled, the `log_bin` system variable, which shows the status of binary logging on the server, is set to ON.

To disable binary logging, you can specify the `--skip-log-bin` or `--disable-log-bin` option at startup. If either of these options is specified and `--log-bin` is also specified, the option specified later takes precedence. When binary logging is disabled, the `log_bin` system variable is set to OFF.

When GTIDs are in use on the server, if you disable binary logging when restarting the server after an abnormal shutdown, some GTIDs are likely to be lost, causing replication to fail. In a normal shutdown, the set of GTIDs from the current binary log file is saved in the `mysql.gtid_executed` table. Following an abnormal shutdown where this did not happen, during recovery the GTIDs are added to the table from the binary log file, provided that binary logging is still enabled. If binary logging is disabled for the server restart, the server cannot access the binary log file to recover the GTIDs, so replication cannot be started. Binary logging can be disabled safely after a normal shutdown.

The `--log-slave-updates` and `--slave-preserve-commit-order` options require binary logging. If you disable binary logging, either omit these options, or specify `--log-slave-updates=OFF` and `--skip-slave-preserve-commit-order`. MySQL disables these options by default when `--skip-log-bin` or `--disable-log-bin` is specified. If you specify `--log-slave-updates` or `--slave-preserve-commit-order` together with `--skip-log-bin` or `--disable-log-bin`, a warning or error message is issued.

In MySQL 5.7, a server ID had to be specified when binary logging was enabled, or the server would not start. In MySQL 8.0, the `server_id` system variable is set to 1 by default. The server can now be started with this default server ID when binary logging is enabled, but an informational message is issued if you do not specify a server ID explicitly by setting the `server_id` system variable. For servers that are used in a replication topology, you must specify a unique nonzero server ID for each server.

For information on the format and management of the binary log, see Section 5.4.4, “The Binary Log”.

- `--log-bin-index[=file_name]`

Command-Line Format	`--log-bin-index=file_name`
System Variable	`log_bin_index`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	File name

The name for the binary log index file, which contains the names of the binary log files. By default, it has the same location and base name as the value specified for the binary log files using the `--log-bin` option, plus the extension `.index`. If you do not specify `--log-bin`, the default binary log index file name is `binlog.index`. If you specify `--log-bin` option with no string or an empty string, the default binary log index file name is *`host_name`*`-bin.index`, using the name of the host machine.

For information on the format and management of the binary log, see Section 5.4.4, "The Binary Log".

Statement selection options. The options in the following list affect which statements are written to the binary log, and thus sent by a replication source server to its replicas. There are also options for replicas that control which statements received from the source should be executed or ignored. For details, see Section 17.1.6.3, "Replica Server Options and Variables".

- `--binlog-do-db=`*`db_name`*

Command-Line Format	`--binlog-do-db=name`
Type	String

This option affects binary logging in a manner similar to the way that `--replicate-do-db` affects replication.

The effects of this option depend on whether the statement-based or row-based logging format is in use, in the same way that the effects of `--replicate-do-db` depend on whether statement-based or row-based replication is in use. You should keep in mind that the format used to log a given statement may not necessarily be the same as that indicated by the value of `binlog_format`. For example, DDL statements such as `CREATE TABLE` and `ALTER TABLE` are always logged as statements, without regard to the logging format in effect, so the following statement-based rules for `--binlog-do-db` always apply in determining whether or not the statement is logged.

Statement-based logging. Only those statements are written to the binary log where the default database (that is, the one selected by `USE`) is *`db_name`*. To specify more than one database, use this option multiple times, once for each database; however, doing so does *not* cause cross-database statements such as `UPDATE some_db.some_table SET foo='bar'` to be logged while a different database (or no database) is selected.

Warning

To specify multiple databases you *must* use multiple instances of this option. Because database names can contain commas, the list is treated as the name of a single database if you supply a comma-separated list.

An example of what does not work as you might expect when using statement-based logging: If the server is started with `--binlog-do-db=sales` and you issue the following statements, the `UPDATE` statement is *not* logged:

```
USE prices;
UPDATE sales.january SET amount=amount+1000;
```

The main reason for this “just check the default database” behavior is that it is difficult from the statement alone to know whether it should be replicated (for example, if you are using multiple-table `DELETE` statements or multiple-table `UPDATE` statements that act across multiple databases). It is also faster to check only the default database rather than all databases if there is no need.

Another case which may not be self-evident occurs when a given database is replicated even though it was not specified when setting the option. If the server is started with `--binlog-do-db=sales`, the following `UPDATE` statement is logged even though `prices` was not included when setting `--binlog-do-db`:

```
USE sales;
UPDATE prices.discounts SET percentage = percentage + 10;
```

Because `sales` is the default database when the `UPDATE` statement is issued, the `UPDATE` is logged.

Row-based logging. Logging is restricted to database *`db_name`*. Only changes to tables belonging to *`db_name`* are logged; the default database has no effect on this. Suppose that the server is started with `--binlog-do-db=sales` and row-based logging is in effect, and then the following statements are executed:

```
USE prices;
UPDATE sales.february SET amount=amount+100;
```

The changes to the `february` table in the `sales` database are logged in accordance with the `UPDATE` statement; this occurs whether or not the `USE` statement was issued. However, when using the row-based logging format and `--binlog-do-db=sales`, changes made by the following `UPDATE` are not logged:

```
USE prices;
UPDATE prices.march SET amount=amount-25;
```

Even if the `USE prices` statement were changed to `USE sales`, the `UPDATE` statement's effects would still not be written to the binary log.

Another important difference in `--binlog-do-db` handling for statement-based logging as opposed to the row-based logging occurs with regard to statements that refer to multiple databases. Suppose that the server is started with `--binlog-do-db=db1`, and the following statements are executed:

```
USE db1;
UPDATE db1.table1, db2.table2 SET db1.table1.col1 = 10, db2.table2.col2 = 20;
```

If you are using statement-based logging, the updates to both tables are written to the binary log. However, when using the row-based format, only the changes to `table1` are logged; `table2` is in a different database, so it is not changed by the `UPDATE`. Now suppose that, instead of the `USE db1` statement, a `USE db4` statement had been used:

```
USE db4;
UPDATE db1.table1, db2.table2 SET db1.table1.col1 = 10, db2.table2.col2 = 20;
```

In this case, the `UPDATE` statement is not written to the binary log when using statement-based logging. However, when using row-based logging, the change to `table1` is logged, but not that to `table2`—in other words, only changes to tables in the database named by `--binlog-do-db` are logged, and the choice of default database has no effect on this behavior.

- `--binlog-ignore-db=`*`db_name`*

Command-Line Format	`--binlog-ignore-db=name`

Type	String

This option affects binary logging in a manner similar to the way that `--replicate-ignore-db` affects replication.

The effects of this option depend on whether the statement-based or row-based logging format is in use, in the same way that the effects of `--replicate-ignore-db` depend on whether statement-based or row-based replication is in use. You should keep in mind that the format used to log a given statement may not necessarily be the same as that indicated by the value of `binlog_format`. For example, DDL statements such as `CREATE TABLE` and `ALTER TABLE` are always logged as statements, without regard to the logging format in effect, so the following statement-based rules for `--binlog-ignore-db` always apply in determining whether or not the statement is logged.

Statement-based logging. Tells the server to not log any statement where the default database (that is, the one selected by `USE`) is *`db_name`*.

When there is no default database, no `--binlog-ignore-db` options are applied, and such statements are always logged. (Bug #11829838, Bug #60188)

Row-based format. Tells the server not to log updates to any tables in the database *`db_name`*. The current database has no effect.

When using statement-based logging, the following example does not work as you might expect. Suppose that the server is started with `--binlog-ignore-db=sales` and you issue the following statements:

```
USE prices;
UPDATE sales.january SET amount=amount+1000;
```

The `UPDATE` statement *is* logged in such a case because `--binlog-ignore-db` applies only to the default database (determined by the `USE` statement). Because the `sales` database was specified explicitly in the statement, the statement has not been filtered. However, when using row-based logging, the `UPDATE` statement's effects are *not* written to the binary log, which means that no changes to the `sales.january` table are logged; in this instance, `--binlog-ignore-db=sales` causes *all* changes made to tables in the source's copy of the `sales` database to be ignored for purposes of binary logging.

To specify more than one database to ignore, use this option multiple times, once for each database. Because database names can contain commas, the list is treated as the name of a single database if you supply a comma-separated list.

You should not use this option if you are using cross-database updates and you do not want these updates to be logged.

Checksum options. MySQL supports reading and writing of binary log checksums. These are enabled using the two options listed here:

- `--binlog-checksum={NONE|CRC32}`

Command-Line Format	`--binlog-checksum=type`
Type	String
Default Value	`CRC32`

Valid Values	`NONE`
	
	`CRC32`

Enabling this option causes the source to write checksums for events written to the binary log. Set to `NONE` to disable, or the name of the algorithm to be used for generating checksums; currently, only

CRC32 checksums are supported, and CRC32 is the default. You cannot change the setting for this option within a transaction.

To control reading of checksums by the replica (from the relay log), use the `--slave-sql-verify-checksum` option.

Testing and debugging options. The following binary log options are used in replication testing and debugging. They are not intended for use in normal operations.

- `--max-binlog-dump-events=N`

Command-Line Format	`--max-binlog-dump-events=#`
Type	Integer
Default Value	`0`

This option is used internally by the MySQL test suite for replication testing and debugging.

- `--sporadic-binlog-dump-fail`

Command-Line Format	`--sporadic-binlog-dump-fail[={OFF|ON}]`
Type	Boolean
Default Value	`OFF`

This option is used internally by the MySQL test suite for replication testing and debugging.

System Variables Used with Binary Logging

The following list describes system variables for controlling binary logging. They can be set at server startup and some of them can be changed at runtime using `SET`. Server options used to control binary logging are listed earlier in this section.

- `binlog_cache_size`

Command-Line Format	`--binlog-cache-size=#`
System Variable	`binlog_cache_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`32768`
Minimum Value	`4096`
Maximum Value (64-bit platforms)	`18446744073709547520`
Maximum Value (32-bit platforms)	`4294963200`
Unit	bytes
Block Size	`4096`

The size of the memory buffer to hold changes to the binary log during a transaction.

The block size is 4096. A value that is not an exact multiple of the block size is rounded down to the next lower multiple of the block size by MySQL Server before storing the value for the system variable. The parser allows values up to the maximum unsigned integer value for the platform (4294967295 or $2^{32}-1$ for a 32-bit system, 18446744073709551615 or $2^{64}-1$ for a 64-bit system) but the actual maximum is a block size lower.

When binary logging is enabled on the server (with the `log_bin` system variable set to ON), a binary log cache is allocated for each client if the server supports any transactional storage engines. If the data for the transaction exceeds the space in the memory buffer, the excess data is stored in a temporary file. When binary log encryption is active on the server, the memory buffer is not encrypted, but (from MySQL 8.0.17) any temporary file used to hold the binary log cache is encrypted. After each transaction is committed, the binary log cache is reset by clearing the memory buffer and truncating the temporary file if used.

If you often use large transactions, you can increase this cache size to get better performance by reducing or eliminating the need to write to temporary files. The `Binlog_cache_use` and `Binlog_cache_disk_use` status variables can be useful for tuning the size of this variable. See Section 5.4.4, “The Binary Log”.

`binlog_cache_size` sets the size for the transaction cache only; the size of the statement cache is governed by the `binlog_stmt_cache_size` system variable.

- `binlog_checksum`

Command-Line Format	`--binlog-checksum=name`
System Variable	`binlog_checksum`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String
Default Value	`CRC32`

Valid Values	`NONE`
	
	`CRC32`

When enabled, this variable causes the source to write a checksum for each event in the binary log. `binlog_checksum` supports the values `NONE` (which disables checksums) and `CRC32`. The default is `CRC32`. When `binlog_checksum` is disabled (value `NONE`), the server verifies that it is writing only complete events to the binary log by writing and checking the event length (rather than a checksum) for each event.

Setting this variable on the source to a value unrecognized by the replica causes the replica to set its own `binlog_checksum` value to `NONE`, and to stop replication with an error. If backward compatibility with older replicas is a concern, you may want to set the value explicitly to `NONE`.

Up to and including MySQL 8.0.20, Group Replication cannot make use of checksums and does not support their presence in the binary log, so you must set `binlog_checksum=NONE` when configuring a server instance to become a group member. From MySQL 8.0.21, Group Replication supports checksums, so group members may use the default setting.

Changing the value of `binlog_checksum` causes the binary log to be rotated, because checksums must be written for an entire binary log file, and never for only part of one. You cannot change the value of `binlog_checksum` within a transaction.

When binary log transaction compression is enabled using the `binlog_transaction_compression` system variable, checksums are not written for individual events in a compressed transaction payload. Instead a checksum is written for the GTID event, and a checksum for the compressed `Transaction_payload_event`.

- `binlog_direct_non_transactional_updates`

Command-Line Format	`--binlog-direct-non-transactional-updates[={OFF|ON}]`
System Variable	`binlog_direct_non_transactional_updates`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Due to concurrency issues, a replica can become inconsistent when a transaction contains updates to both transactional and nontransactional tables. MySQL tries to preserve causality among these statements by writing nontransactional statements to the transaction cache, which is flushed upon commit. However, problems arise when modifications done to nontransactional tables on behalf of a transaction become immediately visible to other connections because these changes may not be written immediately into the binary log.

The `binlog_direct_non_transactional_updates` variable offers one possible workaround to this issue. By default, this variable is disabled. Enabling `binlog_direct_non_transactional_updates` causes updates to nontransactional tables to be written directly to the binary log, rather than to the transaction cache.

As of MySQL 8.0.14, setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

`binlog_direct_non_transactional_updates` works only for statements that are replicated using the statement-based binary logging format; that is, it works only when the value of `binlog_format` is `STATEMENT`, or when `binlog_format` is `MIXED` and a given statement is being replicated using the statement-based format. This variable has no effect when the binary log format is `ROW`, or when `binlog_format` is set to `MIXED` and a given statement is replicated using the row-based format.

Important

Before enabling this variable, you must make certain that there are no dependencies between transactional and nontransactional tables; an example of such a dependency would be the statement `INSERT INTO myisam_table SELECT * FROM innodb_table`. Otherwise, such statements are likely to cause the replica to diverge from the source.

This variable has no effect when the binary log format is `ROW` or `MIXED`.

- `binlog_encryption`

Command-Line Format	`--binlog-encryption[={OFF|ON}]`
Introduced	8.0.14
System Variable	`binlog_encryption`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean

Default Value	OFF

Enables encryption for binary log files and relay log files on this server. OFF is the default. ON sets encryption on for binary log files and relay log files. Binary logging does not need to be enabled on the server to enable encryption, so you can encrypt the relay log files on a replica that has no binary log. To use encryption, a keyring plugin must be installed and configured to supply MySQL Server's keyring service. For instructions to do this, see Section 6.4.4, “The MySQL Keyring”. Any supported keyring plugin can be used to store binary log encryption keys.

When you first start the server with binary log encryption enabled, a new binary log encryption key is generated before the binary log and relay logs are initialized. This key is used to encrypt a file password for each binary log file (if the server has binary logging enabled) and relay log file (if the server has replication channels), and further keys generated from the file passwords are used to encrypt the data in the files. Relay log files are encrypted for all channels, including Group Replication applier channels and new channels that are created after encryption is activated. The binary log index file and relay log index file are never encrypted.

If you activate encryption while the server is running, a new binary log encryption key is generated at that time. The exception is if encryption was active previously on the server and was then disabled, in which case the binary log encryption key that was in use before is used again. The binary log file and relay log files are rotated immediately, and file passwords for the new files and all subsequent binary log files and relay log files are encrypted using this binary log encryption key. Existing binary log files and relay log files still present on the server are not automatically encrypted, but you can purge them if they are no longer needed.

If you deactivate encryption by changing the binlog_encryption system variable to OFF, the binary log file and relay log files are rotated immediately and all subsequent logging is unencrypted. Previously encrypted files are not automatically decrypted, but the server is still able to read them. The BINLOG_ENCRYPTION_ADMIN privilege (or the deprecated SUPER privilege) is required to activate or deactivate encryption while the server is running. Group Replication applier channels are not included in the relay log rotation request, so unencrypted logging for these channels does not start until their logs are rotated in normal use.

For more information on binary log file and relay log file encryption, see Section 17.3.2, “Encrypting Binary Log Files and Relay Log Files”.

- binlog_error_action

Command-Line Format	--binlog-error-action[=value]
System Variable	binlog_error_action
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	ABORT_SERVER
Valid Values	IGNORE_ERROR

	ABORT_SERVER

Controls what happens when the server encounters an error such as not being able to write to, flush or synchronize the binary log, which can cause the source's binary log to become inconsistent and replicas to lose synchronization.

This variable defaults to ABORT_SERVER, which makes the server halt logging and shut down whenever it encounters such an error with the binary log. On restart, recovery proceeds as in the case of an unexpected server halt (see Section 17.4.2, “Handling an Unexpected Halt of a Replica”).

When binlog_error_action is set to IGNORE_ERROR, if the server encounters such an error it continues the ongoing transaction, logs the error then halts logging, and continues performing updates. To resume binary logging log_bin must be enabled again, which requires a server restart. This setting provides backward compatibility with older versions of MySQL.

- binlog_expire_logs_seconds

Command-Line Format	--binlog-expire-logs-seconds=#
System Variable	binlog_expire_logs_seconds
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	2592000
Minimum Value	0
Maximum Value	4294967295
Unit	seconds

Sets the binary log expiration period in seconds. After their expiration period ends, binary log files can be automatically removed. Possible removals happen at startup and when the binary log is flushed. Log flushing occurs as indicated in Section 5.4, “MySQL Server Logs”.

The default binary log expiration period is 2592000 seconds, which equals 30 days (30*24*60*60 seconds). The default applies if neither binlog_expire_logs_seconds nor the deprecated system variable expire_logs_days has a value set at startup. If a non-zero value for one of the variables binlog_expire_logs_seconds or expire_logs_days is set at startup, this value is used as the binary log expiration period. If a non-zero value for both of those variables is set at startup, the value for binlog_expire_logs_seconds is used as the binary log expiration period, and the value for expire_logs_days is ignored with a warning message.

At runtime, you cannot set binlog_expire_logs_seconds or expire_logs_days to a non-zero value if the other is currently set to a non-zero value. Because the default value for binlog_expire_logs_seconds is non-zero, you must explicitly set binlog_expire_logs_seconds to zero before you can set or change the value of expire_logs_days.

Beginning with MySQL 8.0.29, automatic purging of the binary log can be disabled by setting the binlog_expire_logs_auto_purge system variable to OFF. This takes precedence over any setting for binlog_expire_logs_seconds.

In MySQL 8.0.28 and earlier, to disable automatic purging of the binary log, specify a value of 0 explicitly for binlog_expire_logs_seconds, and do not specify a value for expire_logs_days. For compatibility with earlier releases, automatic purging is also disabled if you specify a value of 0 explicitly for expire_logs_days and do not specify a value for

`binlog_expire_logs_seconds`. In that case, the default for `binlog_expire_logs_seconds` is not applied.

To remove binary log files manually, use the `PURGE BINARY LOGS` statement. See Section 13.4.1.1, "PURGE BINARY LOGS Statement".

- `binlog_expire_logs_auto_purge`

Command-Line Format	`--binlog-expire-logs-auto-purge={ON|OFF}`
Introduced	8.0.29
System Variable	`binlog_expire_logs_auto_purge`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

Enables or disables automatic purging of binary log files. Setting this variable to `ON` (the default) enables automatic purging; setting it to `OFF` disables automatic purging. The interval to wait before purging is controlled by `binlog_expire_logs_seconds` and `expire_logs_days`.

Note

Even if `binlog_expire_logs_auto_purge` is `ON`, setting both `binlog_expire_logs_seconds` and `expire_logs_days` to `0` stops automatic purging from taking place.

This variable has no effect on `PURGE BINARY LOGS`.

- `binlog_format`

Command-Line Format	`--binlog-format=format`
System Variable	`binlog_format`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`ROW`

Valid Values	`MIXED`
	`STATEMENT`
	`ROW`

This system variable sets the binary logging format, and can be any one of `STATEMENT`, `ROW`, or `MIXED`. See Section 17.2.1, "Replication Formats". The setting takes effect when binary logging

is enabled on the server, which is the case when the `log_bin` system variable is set to `ON`. From MySQL 8.0, binary logging is enabled by default.

`binlog_format` can be set at startup or at runtime, except that under some conditions, changing this variable at runtime is not possible or causes replication to fail, as described later.

The default is `ROW`. *Exception*: In NDB Cluster, the default is `MIXED`; statement-based replication is not supported for NDB Cluster.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

The rules governing when changes to this variable take effect and how long the effect lasts are the same as for other MySQL server system variables. For more information, see Section 13.7.6.1, “SET Syntax for Variable Assignment”.

When `MIXED` is specified, statement-based replication is used, except for cases where only row-based replication is guaranteed to lead to proper results. For example, this happens when statements contain loadable functions or the `UUID()` function.

For details of how stored programs (stored procedures and functions, triggers, and events) are handled when each binary logging format is set, see Section 25.7, “Stored Program Binary Logging”.

There are exceptions when you cannot switch the replication format at runtime:

- The replication format cannot be changed from within a stored function or a trigger.
- If a session has open temporary tables, the replication format cannot be changed for the session (`SET @@SESSION.binlog_format`).
- If any replication channel has open temporary tables, the replication format cannot be changed globally (`SET @@GLOBAL.binlog_format` or `SET @@PERSIST.binlog_format`).
- If any replication channel applier thread is currently running, the replication format cannot be changed globally (`SET @@GLOBAL.binlog_format` or `SET @@PERSIST.binlog_format`).

Trying to switch the replication format in any of these cases (or attempting to set the current replication format) results in an error. You can, however, use `PERSIST_ONLY` (`SET @@PERSIST_ONLY.binlog_format`) to change the replication format at any time, because this action does not modify the runtime global system variable value, and takes effect only after a server restart.

Switching the replication format at runtime is not recommended when any temporary tables exist, because temporary tables are logged only when using statement-based replication, whereas with row-based replication and mixed replication, they are not logged.

Changing the logging format on a replication source server does not cause a replica to change its logging format to match. Switching the replication format while replication is ongoing can cause issues if a replica has binary logging enabled, and the change results in the replica using `STATEMENT` format logging while the source is using `ROW` or `MIXED` format logging. A replica is not able to convert binary log entries received in `ROW` logging format to `STATEMENT` format for

use in its own binary log, so this situation can cause replication to fail. For more information, see Section 5.4.4.2, “Setting The Binary Log Format”.

The binary log format affects the behavior of the following server options:

- `--replicate-do-db`
- `--replicate-ignore-db`
- `--binlog-do-db`
- `--binlog-ignore-db`

These effects are discussed in detail in the descriptions of the individual options.

- `binlog_group_commit_sync_delay`

Command-Line Format	`--binlog-group-commit-sync-delay=#`
System Variable	`binlog_group_commit_sync_delay`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`1000000`
Unit	microseconds

Controls how many microseconds the binary log commit waits before synchronizing the binary log file to disk. By default `binlog_group_commit_sync_delay` is set to 0, meaning that there is no delay. Setting `binlog_group_commit_sync_delay` to a microsecond delay enables more transactions to be synchronized together to disk at once, reducing the overall time to commit a group of transactions because the larger groups require fewer time units per group.

When `sync_binlog=0` or `sync_binlog=1` is set, the delay specified by `binlog_group_commit_sync_delay` is applied for every binary log commit group before synchronization (or in the case of `sync_binlog=0`, before proceeding). When `sync_binlog` is set to a value *n* greater than 1, the delay is applied after every *n* binary log commit groups.

Setting `binlog_group_commit_sync_delay` can increase the number of parallel committing transactions on any server that has (or might have after a failover) a replica, and therefore can increase parallel execution on the replicas. To benefit from this effect, the replica servers must have `replica_parallel_type=LOGICAL_CLOCK` (from MySQL 8.0.26) or `slave_parallel_type=LOGICAL_CLOCK` set, and the effect is more significant when `binlog_transaction_dependency_tracking=COMMIT_ORDER` is also set. It is important to take into account both the source's throughput and the replicas' throughput when you are tuning the setting for `binlog_group_commit_sync_delay`.

Setting `binlog_group_commit_sync_delay` can also reduce the number of `fsync()` calls to the binary log on any server (source or replica) that has a binary log.

Note that setting `binlog_group_commit_sync_delay` increases the latency of transactions on the server, which might affect client applications. Also, on highly concurrent workloads, it is possible for the delay to increase contention and therefore reduce throughput. Typically, the benefits of setting a delay outweigh the drawbacks, but tuning should always be carried out to determine the optimal setting.

- `binlog_group_commit_sync_no_delay_count`

Command-Line Format	`--binlog-group-commit-sync-no-delay-count=#`
System Variable	`binlog_group_commit_sync_no_delay_count`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`100000`

The maximum number of transactions to wait for before aborting the current delay as specified by `binlog_group_commit_sync_delay`. If `binlog_group_commit_sync_delay` is set to 0, then this option has no effect.

- `binlog_max_flush_queue_time`

Command-Line Format	`--binlog-max-flush-queue-time=#`
Deprecated	Yes
System Variable	`binlog_max_flush_queue_time`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`100000`
Unit	microseconds

`binlog_max_flush_queue_time` is deprecated, and is marked for eventual removal in a future MySQL release. Formerly, this system variable controlled the time in microseconds to continue reading transactions from the flush queue before proceeding with group commit. It no longer has any effect.

- `binlog_order_commits`

Command-Line Format	`--binlog-order-commits[={OFF|ON}]`
System Variable	`binlog_order_commits`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

When this variable is enabled on a replication source server (which is the default), transaction commit instructions issued to storage engines are serialized on a single thread, so that transactions are always committed in the same order as they are written to the binary log. Disabling this variable

permits transaction commit instructions to be issued using multiple threads. Used in combination with binary log group commit, this prevents the commit rate of a single transaction being a bottleneck to throughput, and might therefore produce a performance improvement.

Transactions are written to the binary log at the point when all the storage engines involved have confirmed that the transaction is prepared to commit. The binary log group commit logic then commits a group of transactions after their binary log write has taken place. When `binlog_order_commits` is disabled, because multiple threads are used for this process, transactions in a commit group might be committed in a different order from their order in the binary log. (Transactions from a single client always commit in chronological order.) In many cases this does not matter, as operations carried out in separate transactions should produce consistent results, and if that is not the case, a single transaction ought to be used instead.

If you want to ensure that the transaction history on the source and on a multithreaded replica remains identical, set `slave_preserve_commit_order=1` on the replica.

- `binlog_rotate_encryption_master_key_at_startup`

Command-Line Format	`--binlog-rotate-encryption-master-key-at-startup[={OFF|ON}]`
Introduced	8.0.14
System Variable	`binlog_rotate_encryption_master_key_at_startup`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Specifies whether or not the binary log master key is rotated at server startup. The binary log master key is the binary log encryption key that is used to encrypt file passwords for the binary log files and relay log files on the server. When a server is started for the first time with binary log encryption enabled (`binlog_encryption=ON`), a new binary log encryption key is generated and used as the binary log master key. If the `binlog_rotate_encryption_master_key_at_startup` system variable is also set to `ON`, whenever the server is restarted, a further binary log encryption key is generated and used as the binary log master key for all subsequent binary log files and relay log files. If the `binlog_rotate_encryption_master_key_at_startup` system variable is set to `OFF`, which is the default, the existing binary log master key is used again after the server restarts. For more information on binary log encryption keys and the binary log master key, see Section 17.3.2, “Encrypting Binary Log Files and Relay Log Files”.

- `binlog_row_event_max_size`

Command-Line Format	`--binlog-row-event-max-size=#`
System Variable (≥ 8.0.14)	`binlog_row_event_max_size`
Scope (≥ 8.0.14)	Global
Dynamic (≥ 8.0.14)	No
`SET_VAR` Hint Applies (≥ 8.0.14)	No
Type	Integer
Default Value	`8192`
Minimum Value	`256`
Maximum Value (64-bit platforms)	`18446744073709551615`
Maximum Value (32-bit platforms)	`4294967295`

Unit	bytes

When row-based binary logging is used, this setting is a soft limit on the maximum size of a row-based binary log event, in bytes. Where possible, rows stored in the binary log are grouped into events with a size not exceeding the value of this setting. If an event cannot be split, the maximum size can be exceeded. The default is 8192 bytes.

The block size is 256. A value that is not an exact multiple of the block size is rounded down to the next lower multiple of the block size by MySQL Server before storing the value for the system variable. The parser allows values up to the maximum unsigned integer value for the platform (4294967295 or $2^{32}-1$ for a 32-bit system, 18446744073709551615 or $2^{64}-1$ for a 64-bit system) but the actual maximum is a block size lower.

This global system variable is read-only and can be set only at server startup. Its value can therefore only be modified by using the `PERSIST_ONLY` keyword or the `@@persist_only` qualifier with the `SET` statement.

- `binlog_row_image`

Command-Line Format	`--binlog-row-image=image_type`
System Variable	`binlog_row_image`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`full`
Valid Values	`full` (Log all columns) `minimal` (Log only changed columns, and columns needed to identify rows) `noblob` (Log all columns, except for unneeded BLOB and TEXT columns)

For MySQL row-based replication, this variable determines how row images are written to the binary log.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

In MySQL row-based replication, each row change event contains two images, a “before” image whose columns are matched against when searching for the row to be updated, and an “after” image containing the changes. Normally, MySQL logs full rows (that is, all columns) for both the before and after images. However, it is not strictly necessary to include every column in both images, and we can often save disk, memory, and network usage by logging only those columns which are actually required.

Note

When deleting a row, only the before image is logged, since there are no changed values to propagate following the deletion. When inserting a row, only the after image is logged, since there is no existing row to be matched.

Only when updating a row are both the before and after images required, and both written to the binary log.

For the before image, it is necessary only that the minimum set of columns required to uniquely identify rows is logged. If the table containing the row has a primary key, then only the primary key column or columns are written to the binary log. Otherwise, if the table has a unique key all of whose columns are `NOT NULL`, then only the columns in the unique key need be logged. (If the table has neither a primary key nor a unique key without any `NULL` columns, then all columns must be used in the before image, and logged.) In the after image, it is necessary to log only the columns which have actually changed.

You can cause the server to log full or minimal rows using the `binlog_row_image` system variable. This variable actually takes one of three possible values, as shown in the following list:

- `full`: Log all columns in both the before image and the after image.
- `minimal`: Log only those columns in the before image that are required to identify the row to be changed; log only those columns in the after image where a value was specified by the SQL statement, or generated by auto-increment.
- `noblob`: Log all columns (same as `full`), except for `BLOB` and `TEXT` columns that are not required to identify rows, or that have not changed.

Note

This variable is not supported by NDB Cluster; setting it has no effect on the logging of `NDB` tables.

The default value is `full`.

When using `minimal` or `noblob`, deletes and updates are guaranteed to work correctly for a given table if and only if the following conditions are true for both the source and destination tables:

- All columns must be present and in the same order; each column must use the same data type as its counterpart in the other table.
- The tables must have identical primary key definitions.

(In other words, the tables must be identical with the possible exception of indexes that are not part of the tables' primary keys.)

If these conditions are not met, it is possible that the primary key column values in the destination table may prove insufficient to provide a unique match for a delete or update. In this event, no warning or error is issued; the source and replica silently diverge, thus breaking consistency.

Setting this variable has no effect when the binary logging format is `STATEMENT`. When `binlog_format` is `MIXED`, the setting for `binlog_row_image` is applied to changes that are logged using row-based format, but this setting has no effect on changes logged as statements.

Setting `binlog_row_image` on either the global or session level does not cause an implicit commit; this means that this variable can be changed while a transaction is in progress without affecting the transaction.

- `binlog_row_metadata`

Command-Line Format	`--binlog-row-metadata=metadata_type`
System Variable	`binlog_row_metadata`
Scope	Global
Dynamic	Yes

SET_VAR Hint Applies	No
Type	Enumeration
Default Value	MINIMAL
Valid Values	FULL (All metadata is included) MINIMAL (Limit included metadata)

Configures the amount of table metadata added to the binary log when using row-based logging. When set to MINIMAL, the default, only metadata related to SIGNED flags, column character set and geometry types are logged. When set to FULL complete metadata for tables is logged, such as column name, ENUM or SET string values, PRIMARY KEY information, and so on.

The extended metadata serves the following purposes:

- Replicas use the metadata to transfer data when its table structure is different from the source's.
- External software can use the metadata to decode row events and store the data into external databases, such as a data warehouse.

- binlog_row_value_options

Command-Line Format	--binlog-row-value-options=#
System Variable	binlog_row_value_options
Scope	Global, Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Set
Default Value	
Valid Values	PARTIAL_JSON

When set to PARTIAL_JSON, this enables use of a space-efficient binary log format for updates that modify only a small portion of a JSON document, which causes row-based replication to write only the modified parts of the JSON document to the after-image for the update in the binary log, rather than writing the full document (see Partial Updates of JSON Values). This works for an UPDATE statement which modifies a JSON column using any sequence of JSON_SET(), JSON_REPLACE(), and JSON_REMOVE(). If the server is unable to generate a partial update, the full document is used instead.

The default value is an empty string, which disables use of the format. To unset binlog_row_value_options and revert to writing the full JSON document, set its value to the empty string.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

binlog_row_value_options=PARTIAL_JSON takes effect only when binary logging is enabled and binlog_format is set to ROW or MIXED. Statement-based replication *always* logs only the modified parts of the JSON document, regardless of any value set for binlog_row_value_options. To maximize the amount of space saved, use binlog_row_image=NOBLOB or binlog_row_image=MINIMAL together with this option.

`binlog_row_image=FULL` saves less space than either of these, since the full JSON document is stored in the before-image, and the partial update is stored only in the after-image.

`mysqlbinlog` output includes partial JSON updates in the form of events encoded as base-64 strings using `BINLOG` statements. If the `--verbose` option is specified, `mysqlbinlog` displays the partial JSON updates as readable JSON using pseudo-SQL statements.

MySQL Replication generates an error if a modification cannot be applied to the JSON document on the replica. This includes a failure to find the path. Be aware that, even with this and other safety checks, if a JSON document on a replica has diverged from that on the source and a partial update is applied, it remains theoretically possible to produce a valid but unexpected JSON document on the replica.

- `binlog_rows_query_log_events`

Command-Line Format	`--binlog-rows-query-log-events[={OFF|ON}]`
System Variable	`binlog_rows_query_log_events`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

This system variable affects row-based logging only. When enabled, it causes the server to write informational log events such as row query log events into its binary log. This information can be used for debugging and related purposes, such as obtaining the original query issued on the source when it cannot be reconstructed from the row updates.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

These informational events are normally ignored by MySQL programs reading the binary log and so cause no issues when replicating or restoring from backup. To view them, increase the verbosity level by using mysqlbinlog's `--verbose` option twice, either as `-vv` or `--verbose --verbose`.

- `binlog_stmt_cache_size`

Command-Line Format	`--binlog-stmt-cache-size=#`
System Variable	`binlog_stmt_cache_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`32768`
Minimum Value	`4096`
Maximum Value (64-bit platforms)	`18446744073709547520`
Maximum Value (32-bit platforms)	`4294963200`
Unit	bytes

Block Size	4096

The size of the memory buffer for the binary log to hold nontransactional statements issued during a transaction.

The block size is 4096. A value that is not an exact multiple of the block size is rounded down to the next lower multiple of the block size by MySQL Server before storing the value for the system variable. The parser allows values up to the maximum unsigned integer value for the platform (4294967295 or $2^{32}-1$ for a 32-bit system, 18446744073709551615 or $2^{64}-1$ for a 64-bit system) but the actual maximum is a block size lower.

When binary logging is enabled on the server (with the `log_bin` system variable set to ON), separate binary log transaction and statement caches are allocated for each client if the server supports any transactional storage engines. If the data for the nontransactional statements used in the transaction exceeds the space in the memory buffer, the excess data is stored in a temporary file. When binary log encryption is active on the server, the memory buffer is not encrypted, but (from MySQL 8.0.17) any temporary file used to hold the binary log cache is encrypted. After each transaction is committed, the binary log statement cache is reset by clearing the memory buffer and truncating the temporary file if used.

If you often use large nontransactional statements during transactions, you can increase this cache size to get better performance by reducing or eliminating the need to write to temporary files. The `Binlog_stmt_cache_use` and `Binlog_stmt_cache_disk_use` status variables can be useful for tuning the size of this variable. See Section 5.4.4, “The Binary Log”.

The `binlog_cache_size` system variable sets the size for the transaction cache.

- `binlog_transaction_compression`

Command-Line Format	`--binlog-transaction-compression[={OFF|ON}]`
Introduced	8.0.20
System Variable	`binlog_transaction_compression`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Enables compression for transactions that are written to binary log files on this server. `OFF` is the default. Use the `binlog_transaction_compression_level_zstd` system variable to set the level for the zstd algorithm that is used for compression.

When binary log transaction compression is enabled, transaction payloads are compressed and then written to the binary log file as a single event (`Transaction_payload_event`). Compressed transaction payloads remain in a compressed state while they are sent in the replication stream to replicas, other Group Replication group members, or clients such as `mysqlbinlog`, and are written to the relay log still in their compressed state. Binary log transaction compression therefore saves storage space both on the originator of the transaction and on the recipient (and for their backups), and saves network bandwidth when the transactions are sent between server instances.

For `binlog_transaction_compression=ON` to have a direct effect, binary logging must be enabled on the server. When a MySQL server instance has no binary log, if it is at a release from MySQL 8.0.20, it can receive, handle, and display compressed transaction payloads regardless of its value for `binlog_transaction_compression`. Compressed transaction payloads received by

such server instances are written in their compressed state to the relay log, so they benefit indirectly from compression carried out by other servers in the replication topology.

This system variable cannot be changed within the context of a transaction. Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

For more information on binary log transaction compression, including details of what events are and are not compressed, and changes in behavior when transaction compression is in use, see Section 5.4.4.5, “Binary Log Transaction Compression”.

Prior to NDB 8.0.31: Setting this variable when the server is running has no effect on logging of transactions on `NDB` tables. Binary log transaction compression can be enabled for `NDB` tables by starting MySQL with `--binlog-transaction-compression=ON` on the command line or in an option file but cannot be enabled or disabled while the server is running.

In NDB 8.0.31 and later: You can use the `ndb_log_transaction_compression` system variable to enable this feature for `NDB`. In addition, setting `--binlog-transaction-compression=ON` on the command line or in a `my.cnf` file causes `ndb_log_transaction_compression` to be enabled on server startup. See the description of the variable for further information.

- `binlog_transaction_compression_level_zstd`

Command-Line Format	`--binlog-transaction-compression-level-zstd=#`
Introduced	8.0.20
System Variable	`binlog_transaction_compression_level_zstd`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`3`
Minimum Value	`1`
Maximum Value	`22`

Sets the compression level for binary log transaction compression on this server, which is enabled by the `binlog_transaction_compression` system variable. The value is an integer that determines the compression effort, from 1 (the lowest effort) to 22 (the highest effort). If you do not specify this system variable, the compression level is set to 3.

As the compression level increases, the data compression ratio increases, which reduces the storage space and network bandwidth required for the transaction payload. However, the effort required for data compression also increases, taking time and CPU and memory resources on the originating server. Increases in the compression effort do not have a linear relationship to increases in the data compression ratio.

This system variable cannot be changed within the context of a transaction. Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

This variable has no effect on logging of transactions on `NDB` tables; in NDB Cluster 8.0.31 and later, you can use `ndb_log_transaction_compression_level_zstd` instead.

- `binlog_transaction_dependency_tracking`

Command-Line Format	`--binlog-transaction-dependency-tracking=value`
System Variable	`binlog_transaction_dependency_tracking`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`COMMIT_ORDER`

Valid Values	`COMMIT_ORDER`
	
	`WRITESET`
	
	`WRITESET_SESSION`

For a replication source server that has multithreaded replicas (replicas on which `replica_parallel_workers` or `slave_parallel_workers` is is greater than 0), `binlog_transaction_dependency_tracking` specifies how the source `mysqld` generates the dependency information that it writes in the binary log to help replicas determine which transactions can be executed in parallel.

The dependency information written by the replication source is represented using logical timestamps. (Thus, setting this variable requires that `replica_parallel_type` or `slave_parallel_type` already be set to `LOGICAL_CLOCK`.) There are two logical timestamps, listed here, for each transaction:

- `sequence_number`: This is 1 for the first transaction in a given binary log, 2 for the second transaction, and so on. The numbering restarts with 1 in each binary log file.
- `last_committed`: This refers to the `sequence_number` of the most recently committed transaction found to conflict with the current transaction. This value is always less than `sequence_number`.

`binlog_transaction_dependency_tracking` controls the choice of scheme used to compute these logical timestamps. Available choices are listed here:

- `COMMIT_ORDER`: Two transactions are considered to be independent if the commit-time window of the first transaction overlaps with the commit-time window of the second transaction. This the default.

  The commit-time window begins immediately following the execution of the last statement of the transaction, and ends immediately after the storage engine commit ends. Since transactions hold all row locks between these two points in time, we know that they cannot update the same rows.

- `WRITESET`: Logical timestamps are computed based on `COMMIT_ORDER` in combination with a second scheme based on write sets for the transaction. Each row in the transaction adds a set of one or more hashes to the transaction's write set, one of each unique key in the row. (If there are no unique, nonnullable keys, a hash of the row is used.) This includes both deleted and inserted rows; for updated rows, both the old and the new row are also included.

  Two transactions are considered conflicting if their write sets overlap—that is, if there is some number (hash) that occurs in the write sets of both transactions. In addition, due to the way the write sets are computed, there are periodic serialization points, such that the write set computation process regards every transaction after a serialization point as conflicting with every transaction before the serialization point. Serialization points affect only dependencies computed by the `WRITESET` algorithm; transactions on opposite sides of the

serialization point may have overlapping commit-time windows, and so can be parallelized on replica in spite of this. Serialization points occur for DDL statements, for transactions updating a table having a foreign key, and for transactions where the session value of `transaction_write_set_extraction` is not the same as the global value. A serialization point is also imposed if the transactions committed since the previous serialization point have generated a total of at least `binlog_transaction_dependency_history_size` unique hashes.

For multithreaded replicas to work with NDB Cluster replication (supported in NDB 8.0.33 and later), this variable must be set to `WRITESET` on the source. See Section 23.7.11, "NDB Cluster Replication Using the Multithreaded Applier", for more information.

- `WRITESET_SESSION`: Two transactions are considered dependent if either of the following statements is true:
  - The transactions are dependent according to `WRITESET`.
  - The transactions were committed in the same user session.

In `WRITESET` or `WRITESET_SESSION` mode, the source uses `COMMIT_ORDER` to generate dependency information for transactions that have empty or partial write sets, transactions that update tables without primary or unique keys, and transactions that update parent tables in a foreign key relationship.

To set `binlog_transaction_dependency_tracking` to `WRITESET` or `WRITESET_SESSION`, `transaction_write_set_extraction` must be set to a value other than `OFF`; the default value (`XXHASH64`) is sufficient for this. `transaction_write_set_extraction` cannot be changed whenever the value of `binlog_transaction_dependency_tracking` is `WRITESET` or `WRITESET_SESSION`. Any change in the value does not take effect for replicated transactions until after the replica has been stopped and restarted with `STOP REPLICA` and `START REPLICA`.

The number of row hashes to be kept and checked for the latest transaction to have changed a given row is determined by the value of `binlog_transaction_dependency_history_size`.

Group Replication carries out its own parallelization after certification when applying transactions from the relay log, independently of any value set for `binlog_transaction_dependency_tracking`, but this variable does affect how transactions are written to the binary logs on Group Replication members. The dependency information in those logs is used to assist the process of state transfer from a donor's binary log for distributed recovery, which takes place whenever a member joins or rejoins the group. For that process, setting `binlog_transaction_dependency_tracking` to `WRITESET` can improve performance for a group member, depending on the group's workload.

- `binlog_transaction_dependency_history_size`

Command-Line Format	`--binlog-transaction-dependency-history-size=#`
System Variable	`binlog_transaction_dependency_history_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`25000`
Minimum Value	`1`

Maximum Value	1000000

Sets an upper limit on the number of row hashes which are kept in memory and used for looking up the transaction that last modified a given row. Once this number of hashes has been reached, the history is purged.

- expire_logs_days

Command-Line Format	--expire-logs-days=#
Deprecated	Yes
System Variable	expire_logs_days
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	0
Minimum Value	0
Maximum Value	99
Unit	days

Specifies the number of days before automatic removal of binary log files. expire_logs_days is deprecated, and you should expect it to be removed in a future release. Instead, use binlog_expire_logs_seconds, which sets the binary log expiration period in seconds. If you do not set a value for either system variable, the default expiration period is 30 days. Possible removals happen at startup and when the binary log is flushed. Log flushing occurs as indicated in Section 5.4, “MySQL Server Logs”.

Any non-zero value that you specify at startup for expire_logs_days is ignored if binlog_expire_logs_seconds is also specified, and the value of binlog_expire_logs_seconds is used instead as the binary log expiration period. A warning message is issued in this situation. A non-zero startup value for expire_logs_days is only applied as the binary log expiration period if binlog_expire_logs_seconds is not specified or is specified as 0.

At runtime, you cannot set binlog_expire_logs_seconds or expire_logs_days to a non-zero value if the other is currently set to a non-zero value. Because the default value for binlog_expire_logs_seconds is non-zero, you must explicitly set binlog_expire_logs_seconds to zero before you can set or change the value of expire_logs_days.

To disable automatic purging of the binary log, specify a value of 0 explicitly for binlog_expire_logs_seconds, and do not specify a value for expire_logs_days. For compatibility with earlier releases, automatic purging is also disabled if you specify a value of 0 explicitly for expire_logs_days and do not specify a value for binlog_expire_logs_seconds. In that case, the default for binlog_expire_logs_seconds is not applied.

To remove binary log files manually, use the PURGE BINARY LOGS statement. See Section 13.4.1.1, “PURGE BINARY LOGS Statement”.

- log_bin

System Variable	log_bin
Scope	Global
Dynamic	No

SET_VAR Hint Applies	No
Type	Boolean

Shows the status of binary logging on the server, either enabled (ON) or disabled (OFF). With binary logging enabled, the server logs all statements that change data to the binary log, which is used for backup and replication. ON means that the binary log is available, OFF means that it is not in use. The --log-bin option can be used to specify a base name and location for the binary log.

In earlier MySQL versions, binary logging was disabled by default, and was enabled if you specified the --log-bin option. From MySQL 8.0, binary logging is enabled by default, with the log_bin system variable set to ON, whether or not you specify the --log-bin option. The exception is if you use mysqld to initialize the data directory manually by invoking it with the --initialize or --initialize-insecure option, when binary logging is disabled by default. It is possible to enable binary logging in this case by specifying the --log-bin option.

If the --skip-log-bin or --disable-log-bin option is specified at startup, binary logging is disabled, with the log_bin system variable set to OFF. If either of these options is specified and --log-bin is also specified, the option specified later takes precedence.

For information on the format and management of the binary log, see Section 5.4.4, "The Binary Log".

- log_bin_basename

System Variable	log_bin_basename
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	File name

Holds the base name and path for the binary log files, which can be set with the --log-bin server option. The maximum variable length is 256. In MySQL 8.0, if the --log-bin option is not supplied, the default base name is binlog. For compatibility with MySQL 5.7, if the --log-bin option is supplied with no string or with an empty string, the default base name is *host_name*-bin, using the name of the host machine. The default location is the data directory.

- log_bin_index

Command-Line Format	--log-bin-index=file_name
System Variable	log_bin_index
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	File name

Holds the base name and path for the binary log index file, which can be set with the --log-bin-index server option. The maximum variable length is 256.

- log_bin_trust_function_creators

Command-Line Format	--log-bin-trust-function-creators[={OFF|ON}]
System Variable	log_bin_trust_function_creators
Scope	Global

Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

This variable applies when binary logging is enabled. It controls whether stored function creators can be trusted not to create stored functions that may cause unsafe events to be written to the binary log. If set to 0 (the default), users are not permitted to create or alter stored functions unless they have the SUPER privilege in addition to the CREATE ROUTINE or ALTER ROUTINE privilege. A setting of 0 also enforces the restriction that a function must be declared with the DETERMINISTIC characteristic, or with the READS SQL DATA or NO SQL characteristic. If the variable is set to 1, MySQL does not enforce these restrictions on stored function creation. This variable also applies to trigger creation. See Section 25.7, “Stored Program Binary Logging”.

- log_bin_use_v1_row_events

Command-Line Format	--log-bin-use-v1-row-events[={OFF|ON}]
Deprecated	8.0.18
System Variable	log_bin_use_v1_row_events
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

This read-only system variable is deprecated. Setting the system variable to ON at server startup enabled row-based replication with replicas running MySQL Server 5.5 and earlier by writing the binary log using Version 1 binary log row events, instead of Version 2 binary log row events which are the default as of MySQL 5.6.

- log_replica_updates

Command-Line Format	--log-replica-updates[={OFF|ON}]
Introduced	8.0.26
System Variable	log_replica_updates
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

From MySQL 8.0.26, use log_replica_updates in place of log_slave_updates, which is deprecated from that release. In releases before MySQL 8.0.26, use log_slave_updates.

log_replica_updates specifies whether updates received by a replica server from a replication source server should be logged to the replica's own binary log.

Enabling this variable causes the replica to write the updates that are received from a source and performed by the replication SQL thread to the replica's own binary log. Binary logging, which is controlled by the --log-bin option and is enabled by default, must also be enabled on the replica for updates to be logged. See Section 17.1.6, “Replication and Binary Logging Options and

Variables”. `log_replica_updates` is enabled by default, unless you specify `--skip-log-bin` to disable binary logging, in which case MySQL also disables replica update logging by default. If you need to disable replica update logging when binary logging is enabled, specify `--log-replica-updates=OFF` at replica server startup.

Enabling `log_replica_updates` enables replication servers to be chained. For example, you might want to set up replication servers using this arrangement:

```
A -> B -> C
```

Here, `A` serves as the source for the replica `B`, and `B` serves as the source for the replica `C`. For this to work, `B` must be both a source *and* a replica. With binary logging enabled and `log_replica_updates` enabled, which are the default settings, updates received from `A` are logged by `B` to its binary log, and can therefore be passed on to `C`.

- `log_slave_updates`

Command-Line Format	`--log-slave-updates[={OFF|ON}]`
Deprecated	8.0.26
System Variable	`log_slave_updates`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

From MySQL 8.0.26, `log_slave_updates` is deprecated and the alias `log_replica_updates` should be used instead. In releases before MySQL 8.0.26, use `log_slave_updates`.

`log_slave_updates` specifies whether updates received by a replica server from a replication source server should be logged to the replica's own binary log.

Enabling this variable causes the replica to write the updates that are received from a source and performed by the replication SQL thread to the replica's own binary log. Binary logging, which is controlled by the `--log-bin` option and is enabled by default, must also be enabled on the replica for updates to be logged. See Section 17.1.6, “Replication and Binary Logging Options and Variables”. `log_slave_updates` is enabled by default, unless you specify `--skip-log-bin` to disable binary logging, in which case MySQL also disables replica update logging by default. If you need to disable replica update logging when binary logging is enabled, specify `--log-slave-updates=OFF` at replica server startup.

Enabling `log_slave_updates` enables replication servers to be chained. For example, you might want to set up replication servers using this arrangement:

```
A -> B -> C
```

Here, `A` serves as the source for the replica `B`, and `B` serves as the source for the replica `C`. For this to work, `B` must be both a source *and* a replica. With binary logging enabled and `log_slave_updates` enabled, which are the default settings, updates received from `A` are logged by `B` to its binary log, and can therefore be passed on to `C`.

- `log_statements_unsafe_for_binlog`

Command-Line Format	`--log-statements-unsafe-for-binlog[={OFF|ON}]`
System Variable	`log_statements_unsafe_for_binlog`

Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

If error 1592 is encountered, controls whether the generated warnings are added to the error log or not.

- master_verify_checksum

Command-Line Format	--master-verify-checksum[={OFF|ON}]
Deprecated	8.0.26
System Variable	master_verify_checksum
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

From MySQL 8.0.26, master_verify_checksum is deprecated and the alias source_verify_checksum should be used instead. In releases before MySQL 8.0.26, use master_verify_checksum.

Enabling master_verify_checksum causes the source to verify events read from the binary log by examining checksums, and to stop with an error in the event of a mismatch. master_verify_checksum is disabled by default; in this case, the source uses the event length from the binary log to verify events, so that only complete events are read from the binary log.

- max_binlog_cache_size

Command-Line Format	--max-binlog-cache-size=#
System Variable	max_binlog_cache_size
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	18446744073709547520
Minimum Value	4096
Maximum Value	18446744073709547520
Unit	bytes
Block Size	4096

If a transaction requires more than this many bytes of memory, the server generates a Multi-statement transaction required more than 'max_binlog_cache_size' bytes of storage error. The minimum value is 4096. The maximum possible value is 16EiB (exbibytes). The

maximum recommended value is 4GB; this is due to the fact that MySQL currently cannot work with binary log positions greater than 4GB.

The block size is 4096. A value that is not an exact multiple of the block size is rounded down to the next lower multiple of the block size by MySQL Server before storing the value for the system variable. The parser allows values up to the maximum unsigned integer value for the platform (4294967295 or $2^{32}-1$ for a 32-bit system, 18446744073709551615 or $2^{64}-1$ for a 64-bit system) but the actual maximum is a block size lower.

`max_binlog_cache_size` sets the size for the transaction cache only; the upper limit for the statement cache is governed by the `max_binlog_stmt_cache_size` system variable.

The visibility to sessions of `max_binlog_cache_size` matches that of the `binlog_cache_size` system variable; in other words, changing its value affects only new sessions that are started after the value is changed.

- `max_binlog_size`

Command-Line Format	`--max-binlog-size=#`
System Variable	`max_binlog_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1073741824`
Minimum Value	`4096`
Maximum Value	`1073741824`
Unit	bytes
Block Size	`4096`

If a write to the binary log causes the current log file size to exceed the value of this variable, the server rotates the binary logs (closes the current file and opens the next one). The minimum value is 4096 bytes. The maximum and default value is 1GB. Encrypted binary log files have an additional 512-byte header, which is included in `max_binlog_size`.

A transaction is written in one chunk to the binary log, so it is never split between several binary logs. Therefore, if you have big transactions, you might see binary log files larger than `max_binlog_size`.

If `max_relay_log_size` is 0, the value of `max_binlog_size` applies to relay logs as well.

With GTIDs in use on the server, when `max_binlog_size` is reached, if the system table `mysql.gtid_executed` cannot be accessed to write the GTIDs from the current binary log file, the binary log cannot be rotated. In this situation, the server responds according to its `binlog_error_action` setting. If `IGNORE_ERROR` is set, an error is logged on the server and binary logging is halted, or if `ABORT_SERVER` is set, the server shuts down.

- `max_binlog_stmt_cache_size`

Command-Line Format	`--max-binlog-stmt-cache-size=#`
System Variable	`max_binlog_stmt_cache_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No

Type	Integer
Default Value	18446744073709547520
Minimum Value	4096
Maximum Value	18446744073709547520
Unit	bytes
Block Size	4096

If nontransactional statements within a transaction require more than this many bytes of memory, the server generates an error. The minimum value is 4096. The maximum and default values are 4GB on 32-bit platforms and 16EB (exabytes) on 64-bit platforms.

The block size is 4096. A value that is not an exact multiple of the block size is rounded down to the next lower multiple of the block size by MySQL Server before storing the value for the system variable. The parser allows values up to the maximum unsigned integer value for the platform (4294967295 or $2^{32}-1$ for a 32-bit system, 18446744073709551615 or $2^{64}-1$ for a 64-bit system) but the actual maximum is a block size lower.

`max_binlog_stmt_cache_size` sets the size for the statement cache only; the upper limit for the transaction cache is governed exclusively by the `max_binlog_cache_size` system variable.

- `original_commit_timestamp`

System Variable	`original_commit_timestamp`
Scope	Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Numeric

For internal use by replication. When re-executing a transaction on a replica, this is set to the time when the transaction was committed on the original source, measured in microseconds since the epoch. This allows the original commit timestamp to be propagated throughout a replication topology.

Setting the session value of this system variable is a restricted operation. The session user must have either the `REPLICATION_APPLIER` privilege (see Section 17.3.3, “Replication Privilege Checks”), or privileges sufficient to set restricted session variables (see Section 5.1.9.1, “System Variable Privileges”). However, note that the variable is not intended for users to set; it is set automatically by the replication infrastructure.

- `source_verify_checksum`

Command-Line Format	`--source-verify-checksum[={OFF|ON}]`
Introduced	8.0.26
System Variable	`source_verify_checksum`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean

Default Value	OFF

From MySQL 8.0.26, use source_verify_checksum in place of master_verify_checksum, which is deprecated from that release. In releases before MySQL 8.0.26, use master_verify_checksum.

Enabling source_verify_checksum causes the source to verify events read from the binary log by examining checksums, and to stop with an error in the event of a mismatch. source_verify_checksum is disabled by default; in this case, the source uses the event length from the binary log to verify events, so that only complete events are read from the binary log.

- sql_log_bin

System Variable	sql_log_bin
Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

This variable controls whether logging to the binary log is enabled for the current session (assuming that the binary log itself is enabled). The default value is ON. To disable or enable binary logging for the current session, set the session sql_log_bin variable to OFF or ON.

Set this variable to OFF for a session to temporarily disable binary logging while making changes to the source you do not want replicated to the replica.

Setting the session value of this system variable is a restricted operation. The session user must have privileges sufficient to set restricted session variables. See Section 5.1.9.1, “System Variable Privileges”.

It is not possible to set the session value of sql_log_bin within a transaction or subquery.

Setting this variable to OFF prevents GTIDs from being assigned to transactions in the binary log. If you are using GTIDs for replication, this means that even when binary logging is later enabled again, the GTIDs written into the log from this point do not account for any transactions that occurred in the meantime, so in effect those transactions are lost.

- sync_binlog

Command-Line Format	--sync-binlog=#
System Variable	sync_binlog
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	1
Minimum Value	0
Maximum Value	4294967295

Controls how often the MySQL server synchronizes the binary log to disk.

- sync_binlog=0: Disables synchronization of the binary log to disk by the MySQL server. Instead, the MySQL server relies on the operating system to flush the binary log to disk from time

to time as it does for any other file. This setting provides the best performance, but in the event of a power failure or operating system crash, it is possible that the server has committed transactions that have not been synchronized to the binary log.

- `sync_binlog=1`: Enables synchronization of the binary log to disk before transactions are committed. This is the safest setting but can have a negative impact on performance due to the increased number of disk writes. In the event of a power failure or operating system crash, transactions that are missing from the binary log are only in a prepared state. This permits the automatic recovery routine to roll back the transactions, which guarantees that no transaction is lost from the binary log.
- `sync_binlog=N`, where `N` is a value other than 0 or 1: The binary log is synchronized to disk after `N` binary log commit groups have been collected. In the event of a power failure or operating system crash, it is possible that the server has committed transactions that have not been flushed to the binary log. This setting can have a negative impact on performance due to the increased number of disk writes. A higher value improves performance, but with an increased risk of data loss.

For the greatest possible durability and consistency in a replication setup that uses `InnoDB` with transactions, use these settings:

- `sync_binlog=1`.
- `innodb_flush_log_at_trx_commit=1`.

Caution

Many operating systems and some disk hardware fool the flush-to-disk operation. They may tell `mysqld` that the flush has taken place, even though it has not. In this case, the durability of transactions is not guaranteed even with the recommended settings, and in the worst case, a power outage can corrupt `InnoDB` data. Using a battery-backed disk cache in the SCSI disk controller or in the disk itself speeds up file flushes, and makes the operation safer. You can also try to disable the caching of disk writes in hardware caches.

- `transaction_write_set_extraction`

Command-Line Format	`--transaction-write-set-extraction[=value]`
Deprecated	8.0.26
System Variable	`transaction_write_set_extraction`
Scope	Global, Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`XXHASH64`

Valid Values	`OFF`
	
	`MURMUR32`

	XXHASH64

This system variable specifies the algorithm used to hash the writes extracted during a transaction. The default is XXHASH64. OFF means that write sets are not collected.

transaction_write_set_extraction is deprecated as of MySQL 8.0.26; expect it to be removed in a future MySQL release.

The XXHASH64 setting is required for Group Replication, where the process of extracting the writes from a transaction is used for conflict detection and certification on all group members (see Section 18.3.1, "Group Replication Requirements"). For a replication source server that has multithreaded replicas (replicas on which replica_parallel_workers or slave_parallel_workers is set to a value greater than 0), where binlog_transaction_dependency_tracking is set to WRITESET or WRITESET_SESSION, transaction_write_set_extraction must not be OFF. While the current value of binlog_transaction_dependency_tracking is WRITESET or WRITESET_SESSION, you cannot change the value of transaction_write_set_extraction.

As of MySQL 8.0.14, setting the session value of this system variable is a restricted operation; the session user must have privileges sufficient to set restricted session variables (see Section 5.1.9.1, "System Variable Privileges"). binlog_format must be set to ROW to change the value of transaction_write_set_extraction. If you change the value, the new value does not take effect on replicated transactions until after the replica has been stopped and restarted with STOP REPLICA and START REPLICA.

17.1.6.5 Global Transaction ID System Variables

The MySQL Server system variables described in this section are used to monitor and control Global Transaction Identifiers (GTIDs). For additional information, see Section 17.1.3, "Replication with Global Transaction Identifiers".

- binlog_gtid_simple_recovery

Command-Line Format	--binlog-gtid-simple-recovery[={OFF|ON}]
System Variable	binlog_gtid_simple_recovery
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

This variable controls how binary log files are iterated during the search for GTIDs when MySQL starts or restarts.

When binlog_gtid_simple_recovery=TRUE, which is the default in MySQL 8.0, the values of gtid_executed and gtid_purged are computed at startup based on the values of Previous_gtids_log_event in the most recent and oldest binary log files. For a description of the computation, see The gtid_purged System Variable. This setting accesses only two binary log files during server restart. If all binary logs on the server were generated using MySQL 5.7.8 or later, binlog_gtid_simple_recovery=TRUE can always safely be used.

If any binary logs from MySQL 5.7.7 or older are present on the server (for example, following an upgrade of an older server to MySQL 8.0), with binlog_gtid_simple_recovery=TRUE, gtid_executed and gtid_purged might be initialized incorrectly in the following two situations:

- The newest binary log was generated by MySQL 5.7.5 or earlier, and `gtid_mode` was `ON` for some binary logs but `OFF` for the newest binary log.
- A `SET @@GLOBAL.gtid_purged` statement was issued on MySQL 5.7.7 or earlier, and the binary log that was active at the time of the `SET @@GLOBAL.gtid_purged` statement has not yet been purged.

If an incorrect GTID set is computed in either situation, it remains incorrect even if the server is later restarted with `binlog_gtid_simple_recovery=FALSE`. If either of these situations apply or might apply on the server, set `binlog_gtid_simple_recovery=FALSE` before starting or restarting the server.

When `binlog_gtid_simple_recovery=FALSE` is set, the method of computing `gtid_executed` and `gtid_purged` as described in The `gtid_purged` System Variable is changed to iterate the binary log files as follows:

- Instead of using the value of `Previous_gtids_log_event` and GTID log events from the newest binary log file, the computation for `gtid_executed` iterates from the newest binary log file, and uses the value of `Previous_gtids_log_event` and any GTID log events from the first binary log file where it finds a `Previous_gtids_log_event` value. If the server's most recent binary log files do not have GTID log events, for example if `gtid_mode=ON` was used but the server was later changed to `gtid_mode=OFF`, this process can take a long time.
- Instead of using the value of `Previous_gtids_log_event` from the oldest binary log file, the computation for `gtid_purged` iterates from the oldest binary log file, and uses the value of `Previous_gtids_log_event` from the first binary log file where it finds either a nonempty `Previous_gtids_log_event` value, or at least one GTID log event (indicating that the use of GTIDs starts at that point). If the server's older binary log files do not have GTID log events, for example if `gtid_mode=ON` was only set recently on the server, this process can take a long time.

- `enforce_gtid_consistency`

Command-Line Format	`--enforce-gtid-consistency[=value]`
System Variable	`enforce_gtid_consistency`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`OFF`

Valid Values	`OFF`
	`ON`
	`WARN`

Depending on the value of this variable, the server enforces GTID consistency by allowing execution of only statements that can be safely logged using a GTID. You *must* set this variable to `ON` before enabling GTID based replication.

The values that `enforce_gtid_consistency` can be configured to are:

- `OFF`: all transactions are allowed to violate GTID consistency.
- `ON`: no transaction is allowed to violate GTID consistency.

- `WARN`: all transactions are allowed to violate GTID consistency, but a warning is generated in this case.

`--enforce-gtid-consistency` only takes effect if binary logging takes place for a statement. If binary logging is disabled on the server, or if statements are not written to the binary log because they are removed by a filter, GTID consistency is not checked or enforced for the statements that are not logged.

Only statements that can be logged using GTID safe statements can be logged when `enforce_gtid_consistency` is set to `ON`, so the operations listed here cannot be used with this option:

- `CREATE TEMPORARY TABLE` or `DROP TEMPORARY TABLE` statements inside transactions.
- Transactions or statements that update both transactional and nontransactional tables. There is an exception that nontransactional DML is allowed in the same transaction or in the same statement as transactional DML, if all *nontransactional* tables are temporary.
- `CREATE TABLE ... SELECT` statements, prior to MySQL 8.0.21. From MySQL 8.0.21, `CREATE TABLE ... SELECT` statements are allowed for storage engines that support atomic DDL.

For more information, see Section 17.1.3.7, “Restrictions on Replication with GTIDs”.

Prior to MySQL 5.7 and in early releases in that release series, the boolean `enforce_gtid_consistency` defaulted to `OFF`. To maintain compatibility with these earlier releases, the enumeration defaults to `OFF`, and setting `--enforce-gtid-consistency` without a value is interpreted as setting the value to `ON`. The variable also has multiple textual aliases for the values: `0=OFF=FALSE`, `1=ON=TRUE`,`2=WARN`. This differs from other enumeration types but maintains compatibility with the boolean type used in previous releases. These changes impact on what is returned by the variable. Using `SELECT @@ENFORCE_GTID_CONSISTENCY`, `SHOW VARIABLES LIKE 'ENFORCE_GTID_CONSISTENCY'`, and `SELECT * FROM INFORMATION_SCHEMA.VARIABLES WHERE 'VARIABLE_NAME' = 'ENFORCE_GTID_CONSISTENCY'`, all return the textual form, not the numeric form. This is an incompatible change, since `@@ENFORCE_GTID_CONSISTENCY` returns the numeric form for booleans but returns the textual form for `SHOW` and the Information Schema.

- `gtid_executed`

System Variable	`gtid_executed`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	String
Unit	set of GTIDs

When used with global scope, this variable contains a representation of the set of all transactions executed on the server and GTIDs that have been set by a `SET gtid_purged` statement. This is the same as the value of the `Executed_Gtid_Set` column in the output of `SHOW MASTER STATUS` and `SHOW REPLICA STATUS`. The value of this variable is a GTID set, see GTID Sets for more information.

When the server starts, `@@GLOBAL.gtid_executed` is initialized. See `binlog_gtid_simple_recovery` for more information on how binary logs are iterated to populate

`gtid_executed`. GTIDs are then added to the set as transactions are executed, or if any `SET gtid_purged` statement is executed.

The set of transactions that can be found in the binary logs at any given time is equal to `GTID_SUBTRACT(@@GLOBAL.gtid_executed, @@GLOBAL.gtid_purged)`; that is, to all transactions in the binary log that have not yet been purged.

Issuing `RESET MASTER` causes the global value (but not the session value) of this variable to be reset to an empty string. GTIDs are not otherwise removed from this set other than when the set is cleared due to `RESET MASTER`.

In some older releases, this variable could also be used with session scope, where it contained a representation of the set of transactions that are written to the cache in the current session. The session scope is now deprecated.

- `gtid_executed_compression_period`

Command-Line Format	`--gtid-executed-compression-period=#`
System Variable	`gtid_executed_compression_period`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value (≥ 8.0.23)	`0`
Default Value (≤ 8.0.22)	`1000`
Minimum Value	`0`
Maximum Value	`4294967295`

Compress the `mysql.gtid_executed` table each time this many transactions have been processed. When binary logging is enabled on the server, this compression method is not used, and instead the `mysql.gtid_executed` table is compressed on each binary log rotation. When binary logging is disabled on the server, the compression thread sleeps until the specified number of transactions have been executed, then wakes up to perform compression of the `mysql.gtid_executed` table. Setting the value of this system variable to 0 means that the thread never wakes up, so this explicit compression method is not used. Instead, compression occurs implicitly as required.

From MySQL 8.0.17, `InnoDB` transactions are written to the `mysql.gtid_executed` table by a separate process to non-`InnoDB` transactions. If the server has a mix of `InnoDB` transactions and non-`InnoDB` transactions, the compression controlled by this system variable interferes with the work of this process and can slow it significantly. For this reason, from that release it is recommended that you set `gtid_executed_compression_period` to 0.

From MySQL 8.0.23, `InnoDB` and non-`InnoDB` transactions are written to the `mysql.gtid_executed` table by the same process, and the `gtid_executed_compression_period` default value is 0.

See mysql.gtid_executed Table Compression for more information.

- `gtid_mode`

Command-Line Format	`--gtid-mode=MODE`
System Variable	`gtid_mode`
Scope	Global
Dynamic	Yes

SET_VAR Hint Applies	No
Type	Enumeration
Default Value	OFF
Valid Values	OFF OFF_PERMISSIVE ON_PERMISSIVE ON

Controls whether GTID based logging is enabled and what type of transactions the logs can contain. You must have privileges sufficient to set global system variables. See Section 5.1.9.1, "System Variable Privileges". enforce_gtid_consistency must be set to ON before you can set gtid_mode=ON. Before modifying this variable, see Section 17.1.4, "Changing GTID Mode on Online Servers".

Logged transactions can be either anonymous or use GTIDs. Anonymous transactions rely on binary log file and position to identify specific transactions. GTID transactions have a unique identifier that is used to refer to transactions. The different modes are:

- OFF: Both new and replicated transactions must be anonymous.
- OFF_PERMISSIVE: New transactions are anonymous. Replicated transactions can be either anonymous or GTID transactions.
- ON_PERMISSIVE: New transactions are GTID transactions. Replicated transactions can be either anonymous or GTID transactions.
- ON: Both new and replicated transactions must be GTID transactions.

Changes from one value to another can only be one step at a time. For example, if gtid_mode is currently set to OFF_PERMISSIVE, it is possible to change to OFF or ON_PERMISSIVE but not to ON.

The values of gtid_purged and gtid_executed are persistent regardless of the value of gtid_mode. Therefore even after changing the value of gtid_mode, these variables contain the correct values.

- gtid_next

System Variable	gtid_next
Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	AUTOMATIC
Valid Values	AUTOMATIC ANONYMOUS UUID:NUMBER

This variable is used to specify whether and how the next GTID is obtained.

Setting the session value of this system variable is a restricted operation. The session user must have either the REPLICATION_APPLIER privilege (see Section 17.3.3, "Replication Privilege

Checks"), or privileges sufficient to set restricted session variables (see Section 5.1.9.1, "System Variable Privileges").

`gtid_next` can take any of the following values:

- `AUTOMATIC`: Use the next automatically-generated global transaction ID.
- `ANONYMOUS`: Transactions do not have global identifiers, and are identified by file and position only.
- A global transaction ID in *`UUID`*:*`NUMBER`* format.

Exactly which of the above options are valid depends on the setting of `gtid_mode`, see Section 17.1.4.1, "Replication Mode Concepts" for more information. Setting this variable has no effect if `gtid_mode` is `OFF`.

After this variable has been set to *`UUID`*:*`NUMBER`*, and a transaction has been committed or rolled back, an explicit `SET GTID_NEXT` statement must again be issued before any other statement.

`DROP TABLE` or `DROP TEMPORARY TABLE` fails with an explicit error when used on a combination of nontemporary tables with temporary tables, or of temporary tables using transactional storage engines with temporary tables using nontransactional storage engines.

- `gtid_owned`

System Variable	`gtid_owned`
Scope	Global, Session
Dynamic	No
`SET_VAR` Hint Applies	No
Type	String
Unit	set of GTIDs

This read-only variable is primarily for internal use. Its contents depend on its scope.

- When used with global scope, `gtid_owned` holds a list of all the GTIDs that are currently in use on the server, with the IDs of the threads that own them. This variable is mainly useful for a multi-threaded replica to check whether a transaction is already being applied on another thread. An applier thread takes ownership of a transaction's GTID all the time it is processing the transaction, so `@@global.gtid_owned` shows the GTID and owner for the duration of processing. When a transaction has been committed (or rolled back), the applier thread releases ownership of the GTID.
- When used with session scope, `gtid_owned` holds a single GTID that is currently in use by and owned by this session. This variable is mainly useful for testing and debugging the use of GTIDs when the client has explicitly assigned a GTID for the transaction by setting `gtid_next`. In this case, `@@session.gtid_owned` displays the GTID all the time the client is processing the transaction, until the transaction has been committed (or rolled back). When the client has finished processing the transaction, the variable is cleared. If `gtid_next=AUTOMATIC` is used for the session, `gtid_owned` is populated only briefly during the execution of the commit statement for the transaction, so it cannot be observed from the session concerned, although it is listed if `@@global.gtid_owned` is read at the right point. If you have a requirement to track the GTIDs that are handled by a client in a session, you can enable the session state tracker controlled by the `session_track_gtids` system variable.

- `gtid_purged`

System Variable	`gtid_purged`

Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	String
Unit	set of GTIDs

The global value of the gtid_purged system variable (@@GLOBAL.gtid_purged) is a GTID set consisting of the GTIDs of all the transactions that have been committed on the server, but do not exist in any binary log file on the server. gtid_purged is a subset of gtid_executed. The following categories of GTIDs are in gtid_purged:

- GTIDs of replicated transactions that were committed with binary logging disabled on the replica.
- GTIDs of transactions that were written to a binary log file that has now been purged.
- GTIDs that were added explicitly to the set by the statement SET @@GLOBAL.gtid_purged.

When the server starts, the global value of gtid_purged is initialized to a set of GTIDs. For information on how this GTID set is computed, see The gtid_purged System Variable. If binary logs from MySQL 5.7.7 or older are present on the server, you might need to set binlog_gtid_simple_recovery=FALSE in the server's configuration file to produce the correct computation. See the description for binlog_gtid_simple_recovery for details of the situations in which this setting is needed.

Issuing RESET MASTER causes the value of gtid_purged to be reset to an empty string.

You can set the value of gtid_purged in order to record on the server that the transactions in a certain GTID set have been applied, although they do not exist in any binary log on the server. An example use case for this action is when you are restoring a backup of one or more databases on a server, but you do not have the relevant binary logs containing the transactions on the server.

Important

GTIDs are only available on a server instance up to the number of non-negative values for a signed 64-bit integer (2 to the power of 63, minus 1). If you set the value of gtid_purged to a number that approaches this limit, subsequent commits can cause the server to run out of GTIDs and take the action specified by binlog_error_action. From MySQL 8.0.23, a warning message is issued when the server instance is approaching the limit.

From MySQL 8.0, there are two ways to set the value of gtid_purged. You can either replace the value of gtid_purged with your specified GTID set, or you can append your specified GTID set to the GTID set that is already held by gtid_purged. If the server has no existing GTIDs, for example an empty server that you are provisioning with a backup of an existing database, both methods have the same result. If you are restoring a backup that overlaps the transactions that are already on the server, for example replacing a corrupted table with a partial dump from the source made using mysqldump (which includes the GTIDs of all the transactions on the server, even though the dump is partial), use the first method of replacing the value of gtid_purged. If you are restoring a backup that is disjoint from the transactions that are already on the server, for example provisioning a multi-source replica using dumps from two different servers, use the second method of adding to the value of gtid_purged.

- To replace the value of gtid_purged with your specified GTID set, use the following statement:

```
SET @@GLOBAL.gtid_purged = 'gtid_set'
```

gtid_set must be a superset of the current value of gtid_purged, and must not intersect with gtid_subtract(gtid_executed,gtid_purged). In other words, the new GTID set

must include any GTIDs that were already in `gtid_purged`, and **must not** include any GTIDs in `gtid_executed` that have not yet been purged. `gtid_set` also cannot include any GTIDs that are in `@@global.gtid_owned`, that is, the GTIDs for transactions that are currently being processed on the server.

The result is that the global value of `gtid_purged` is set equal to `gtid_set`, and the value of `gtid_executed` becomes the union of `gtid_set` and the previous value of `gtid_executed`.

- To append your specified GTID set to `gtid_purged`, use the following statement with a plus sign (+) before the GTID set:

```
SET @@GLOBAL.gtid_purged = '+gtid_set'
```

`gtid_set` **must not** intersect with the current value of `gtid_executed`. In other words, the new GTID set must not include any GTIDs in `gtid_executed`, including transactions that are already also in `gtid_purged`. `gtid_set` also cannot include any GTIDs that are in `@@global.gtid_owned`, that is, the GTIDs for transactions that are currently being processed on the server.

The result is that `gtid_set` is added to both `gtid_executed` and `gtid_purged`.

Note

If any binary logs from MySQL 5.7.7 or older are present on the server (for example, following an upgrade of an older server to MySQL 8.0), after issuing a `SET @@GLOBAL.gtid_purged` statement, you might need to set `binlog_gtid_simple_recovery=FALSE` in the server's configuration file before restarting the server, otherwise `gtid_purged` can be computed incorrectly. See the description for `binlog_gtid_simple_recovery` for details of the situations in which this setting is needed.

17.1.7 Common Replication Administration Tasks

Once replication has been started it executes without requiring much regular administration. This section describes how to check the status of replication, how to pause a replica, and how to skip a failed transaction on a replica.

Tip

To deploy multiple instances of MySQL, you can use InnoDB Cluster which enables you to easily administer a group of MySQL server instances in MySQL Shell. InnoDB Cluster wraps MySQL Group Replication in a programmatic environment that enables you easily deploy a cluster of MySQL instances to achieve high availability. In addition, InnoDB Cluster interfaces seamlessly with MySQL Router, which enables your applications to connect to the cluster without writing your own failover process. For similar use cases that do not require high availability, however, you can use InnoDB ReplicaSet. Installation instructions for MySQL Shell can be found here.

17.1.7.1 Checking Replication Status

The most common task when managing a replication process is to ensure that replication is taking place and that there have been no errors between the replica and the source.

The `SHOW REPLICA STATUS` statement, which you must execute on each replica, provides information about the configuration and status of the connection between the replica server and the source server. From MySQL 8.0.22, `SHOW SLAVE STATUS` is deprecated, and `SHOW REPLICA STATUS` is available to use instead. The Performance Schema has replication tables that provide this information in a more accessible form. See Section 27.12.11, “Performance Schema Replication Tables”.

The replication heartbeat information shown in the Performance Schema replication tables lets you check that the replication connection is active even if the source has not sent events to the replica recently. The source sends a heartbeat signal to a replica if there are no updates to, and no unsent events in, the binary log for a longer period than the heartbeat interval. The `MASTER_HEARTBEAT_PERIOD` setting on the source (set by the `CHANGE MASTER TO` statement) specifies the frequency of the heartbeat, which defaults to half of the connection timeout interval for the replica (specified by the system variable `replica_net_timeout` or `slave_net_timeout`). The `replication_connection_status` Performance Schema table shows when the most recent heartbeat signal was received by a replica, and how many heartbeat signals it has received.

If you are using the `SHOW REPLICA STATUS` statement to check on the status of an individual replica, the statement provides the following information:

```
mysql> SHOW REPLICA STATUS\G
*************************** 1. row ***************************
             Replica_IO_State: Waiting for source to send event
                  Source_Host: source1
                  Source_User: root
                  Source_Port: 3306
                Connect_Retry: 60
              Source_Log_File: mysql-bin.000004
          Read_Source_Log_Pos: 931
               Relay_Log_File: replica1-relay-bin.000056
                Relay_Log_Pos: 950
        Relay_Source_Log_File: mysql-bin.000004
           Replica_IO_Running: Yes
          Replica_SQL_Running: Yes
              Replicate_Do_DB:
          Replicate_Ignore_DB:
           Replicate_Do_Table:
       Replicate_Ignore_Table:
      Replicate_Wild_Do_Table:
  Replicate_Wild_Ignore_Table:
                   Last_Errno: 0
                   Last_Error:
                 Skip_Counter: 0
          Exec_Source_Log_Pos: 931
              Relay_Log_Space: 1365
              Until_Condition: None
               Until_Log_File:
                Until_Log_Pos: 0
           Source_SSL_Allowed: No
           Source_SSL_CA_File:
           Source_SSL_CA_Path:
              Source_SSL_Cert:
            Source_SSL_Cipher:
               Source_SSL_Key:
        Seconds_Behind_Source: 0
Source_SSL_Verify_Server_Cert: No
                Last_IO_Errno: 0
                Last_IO_Error:
               Last_SQL_Errno: 0
               Last_SQL_Error:
  Replicate_Ignore_Server_Ids: 0
```

The key fields from the status report to examine are:

- `Replica_IO_State`: The current status of the replica. See Section 8.14.5, “Replication I/O (Receiver) Thread States”, and Section 8.14.6, “Replication SQL Thread States”, for more information.
- `Replica_IO_Running`: Whether the I/O (receiver) thread for reading the source's binary log is running. Normally, you want this to be `Yes` unless you have not yet started replication or have explicitly stopped it with `STOP REPLICA`.
- `Replica_SQL_Running`: Whether the SQL thread for executing events in the relay log is running. As with the I/O thread, this should normally be `Yes`.

- `Last_IO_Error`, `Last_SQL_Error`: The last errors registered by the I/O (receiver) and SQL (applier) threads when processing the relay log. Ideally these should be blank, indicating no errors.
- `Seconds_Behind_Source`: The number of seconds that the replication SQL (applier) thread is behind processing the source binary log. A high number (or an increasing one) can indicate that the replica is unable to handle events from the source in a timely fashion.

  A value of 0 for `Seconds_Behind_Source` can usually be interpreted as meaning that the replica has caught up with the source, but there are some cases where this is not strictly true. For example, this can occur if the network connection between source and replica is broken but the replication I/O (receiver) thread has not yet noticed this; that is, the time period set by `replica_net_timeout` or `slave_net_timeout` has not yet elapsed.

  It is also possible that transient values for `Seconds_Behind_Source` may not reflect the situation accurately. When the replication SQL (applier) thread has caught up on I/O, `Seconds_Behind_Source` displays 0; but when the replication I/O (receiver) thread is still queuing up a new event, `Seconds_Behind_Source` may show a large value until the replication applier thread finishes executing the new event. This is especially likely when the events have old timestamps; in such cases, if you execute `SHOW REPLICA STATUS` several times in a relatively short period, you may see this value change back and forth repeatedly between 0 and a relatively large value.

Several pairs of fields provide information about the progress of the replica in reading events from the source binary log and processing them in the relay log:

- (`Master_Log_file`, `Read_Master_Log_Pos`): Coordinates in the source binary log indicating how far the replication I/O (receiver) thread has read events from that log.
- (`Relay_Master_Log_File`, `Exec_Master_Log_Pos`): Coordinates in the source binary log indicating how far the replication SQL (applier) thread has executed events received from that log.
- (`Relay_Log_File`, `Relay_Log_Pos`): Coordinates in the replica relay log indicating how far the replication SQL (applier) thread has executed the relay log. These correspond to the preceding coordinates, but are expressed in replica relay log coordinates rather than source binary log coordinates.

On the source, you can check the status of connected replicas using `SHOW PROCESSLIST` to examine the list of running processes. Replica connections have `Binlog Dump` in the `Command` field:

```
mysql> SHOW PROCESSLIST \G;
*************************** 4. row ***************************
     Id: 10
   User: root
   Host: replica1:58371
     db: NULL
Command: Binlog Dump
   Time: 777
  State: Has sent all binlog to slave; waiting for binlog to be updated
   Info: NULL
```

Because it is the replica that drives the replication process, very little information is available in this report.

For replicas that were started with the `--report-host` option and are connected to the source, the `SHOW REPLICAS` (or before MySQL 8.0.22, `SHOW SLAVE HOSTS`) statement on the source shows basic information about the replicas. The output includes the ID of the replica server, the value of the `--report-host` option, the connecting port, and source ID:

```
mysql> SHOW REPLICAS;
+-----------+----------+------+------------------+-----------+
| Server_id | Host     | Port | Rpl_recovery_rank | Source_id |
+-----------+----------+------+------------------+-----------+
|        10 | replica1 | 3306 |                0 |         1 |
+-----------+----------+------+------------------+-----------+
```

```
1 row in set (0.00 sec)
```

17.1.7.2 Pausing Replication on the Replica

You can stop and start replication on the replica using the `STOP REPLICA` and `START REPLICA` statements. From MySQL 8.0.22, `STOP SLAVE` and `START SLAVE` are deprecated, and `STOP REPLICA` and `START REPLICA` are available to use instead.

To stop processing of the binary log from the source, use `STOP REPLICA`:

```
mysql> STOP SLAVE;
Or from MySQL 8.0.22:
mysql> STOP REPLICA;
```

When replication is stopped, the replication I/O (receiver) thread stops reading events from the source binary log and writing them to the relay log, and the SQL thread stops reading events from the relay log and executing them. You can pause the I/O (receiver) or SQL (applier) thread individually by specifying the thread type:

```
mysql> STOP SLAVE IO_THREAD;
mysql> STOP SLAVE SQL_THREAD;
Or from MySQL 8.0.22:
mysql> STOP REPLICA IO_THREAD;
mysql> STOP REPLICA SQL_THREAD;
```

To start execution again, use the `START REPLICA` statement:

```
mysql> START SLAVE;
Or from MySQL 8.0.22:
mysql> START REPLICA;
```

To start a particular thread, specify the thread type:

```
mysql> START SLAVE IO_THREAD;
mysql> START SLAVE SQL_THREAD;
Or from MySQL 8.0.22:
mysql> START REPLICA IO_THREAD;
mysql> START REPLICA SQL_THREAD;
```

For a replica that performs updates only by processing events from the source, stopping only the SQL thread can be useful if you want to perform a backup or other task. The I/O (receiver) thread continues to read events from the source but they are not executed. This makes it easier for the replica to catch up when you restart the SQL (applier) thread.

Stopping only the receiver thread enables the events in the relay log to be executed by the applier thread up to the point where the relay log ends. This can be useful when you want to pause execution to catch up with events already received from the source, when you want to perform administration on the replica but also ensure that it has processed all updates to a specific point. This method can also be used to pause event receipt on the replica while you conduct administration on the source. Stopping the receiver thread but permitting the applier thread to run helps ensure that there is not a massive backlog of events to be executed when replication is started again.

17.1.7.3 Skipping Transactions

If replication stops due to an issue with an event in a replicated transaction, you can resume replication by skipping the failed transaction on the replica. Before skipping a transaction, ensure that the replication I/O (receiver) thread is stopped as well as the SQL (applier) thread.

First you need to identify the replicated event that caused the error. Details of the error and the last successfully applied transaction are recorded in the Performance Schema table `replication_applier_status_by_worker`. You can use `mysqlbinlog` to retrieve and display the events that were logged around the time of the error. For instructions to do this, see Section 7.5, “Point-in-Time (Incremental) Recovery”. Alternatively, you can issue `SHOW RELAYLOG EVENTS` on the replica or `SHOW BINLOG EVENTS` on the source.

Before skipping the transaction and restarting the replica, check these points:

- Is the transaction that stopped replication from an unknown or untrusted source? If so, investigate the cause in case there are any security considerations that indicate the replica should not be restarted.
- Does the transaction that stopped replication need to be applied on the replica? If so, either make the appropriate corrections and reapply the transaction, or manually reconcile the data on the replica.
- Did the transaction that stopped replication need to be applied on the source? If not, undo the transaction manually on the server where it originally took place.

To skip the transaction, choose one of the following methods as appropriate:

- When GTIDs are in use (`gtid_mode` is `ON`), see Skipping Transactions With GTIDs .
- When GTIDs are not in use or are being phased in (`gtid_mode` is `OFF`, `OFF_PERMISSIVE`, or `ON_PERMISSIVE`), see Skipping Transactions Without GTIDs.
- If you have enabled GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement, see Skipping Transactions Without GTIDs. Using `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on a replication channel is not the same as introducing GTID-based replication for the channel, and you cannot use the transaction skipping method for GTID-based replication with those channels.

To restart replication after skipping the transaction, issue `START REPLICA`, with the `FOR CHANNEL` clause if the replica is a multi-source replica.

Skipping Transactions With GTIDs

When GTIDs are in use (`gtid_mode` is `ON`), the GTID for a committed transaction is persisted on the replica even if the content of the transaction is filtered out. This feature prevents a replica from retrieving previously filtered transactions when it reconnects to the source using GTID auto-positioning. It can also be used to skip a transaction on the replica, by committing an empty transaction in place of the failing transaction.

This method of skipping transactions is not suitable when you have enabled GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement.

If the failing transaction generated an error in a worker thread, you can obtain its GTID directly from the `APPLYING_TRANSACTION` field in the Performance Schema table `replication_applier_status_by_worker`. To see what the transaction is, issue `SHOW RELAYLOG EVENTS` on the replica or `SHOW BINLOG EVENTS` on the source, and search the output for a transaction preceded by that GTID.

When you have assessed the failing transaction for any other appropriate actions as described previously (such as security considerations), to skip it, commit an empty transaction on the replica that has the same GTID as the failing transaction. For example:

```
SET GTID_NEXT='aaa-bbb-ccc-ddd:N';
BEGIN;
COMMIT;
SET GTID_NEXT='AUTOMATIC';
```

The presence of this empty transaction on the replica means that when you issue a `START REPLICA` statement to restart replication, the replica uses the auto-skip function to ignore the failing transaction, because it sees a transaction with that GTID has already been applied. If the replica is a multi-source replica, you do not need to specify the channel name when you commit the empty transaction, but you do need to specify the channel name when you issue `START REPLICA`.

Note that if binary logging is in use on this replica, the empty transaction enters the replication stream if the replica becomes a source or primary in the future. If you need to avoid this possibility, consider flushing and purging the replica's binary logs, as in this example:

```
FLUSH LOGS;
PURGE BINARY LOGS TO 'binlog.000146';
```

The GTID of the empty transaction is persisted, but the transaction itself is removed by purging the binary log files.

Skipping Transactions Without GTIDs

To skip failing transactions when GTIDs are not in use or are being phased in (`gtid_mode` is `OFF`, `OFF_PERMISSIVE`, or `ON_PERMISSIVE`), you can skip a specified number of events by issuing a `SET GLOBAL sql_replica_skip_counter` statement (from MySQL 8.0.26) or a `SET GLOBAL sql_slave_skip_counter` statement. Alternatively, you can skip past an event or events by issuing a `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement to move the source binary log position forward.

These methods are also suitable when you have enabled GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement.

When you use these methods, it is important to understand that you are not necessarily skipping a complete transaction, as is always the case with the GTID-based method described previously. These non-GTID-based methods are not aware of transactions as such, but instead operate on events. The binary log is organized as a sequence of groups known as event groups, and each event group consists of a sequence of events.

- For transactional tables, an event group corresponds to a transaction.
- For nontransactional tables, an event group corresponds to a single SQL statement.

A single transaction can contain changes to both transactional and nontransactional tables.

When you use a `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter` statement to skip events and the resulting position is in the middle of an event group, the replica continues to skip events until it reaches the end of the group. Execution then starts with the next event group. The `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement does not have this function, so you must be careful to identify the correct location to restart replication at the beginning of an event group. However, using `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` means you do not have to count the events that need to be skipped, as you do with `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter`, and instead you can just specify the location to restart.

Skipping Transactions With `SET GLOBAL sql_slave_skip_counter`

When you have assessed the failing transaction for any other appropriate actions as described previously (such as security considerations), count the number of events that you need to skip. One event normally corresponds to one SQL statement in the binary log, but note that statements that use `AUTO_INCREMENT` or `LAST_INSERT_ID()` count as two events in the binary log. When binary log transaction compression is in use, a compressed transaction payload (`Transaction_payload_event`) is counted as a single counter value, so all the events inside it are skipped as a unit.

If you want to skip the complete transaction, you can count the events to the end of the transaction, or you can just skip the relevant event group. Remember that with `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter`, the replica continues to skip to the end of an event group. Make sure you do not skip too far forward and go into the next event group or transaction so that it is not also skipped.

Issue the `SET` statement as follows, where *`N`* is the number of events from the source to skip:

```
SET GLOBAL sql_slave_skip_counter = N

Or from MySQL 8.0.26:
SET GLOBAL sql_replica_skip_counter = N
```

This statement cannot be issued if `gtid_mode=ON` is set, or if the replication I/O (receiver) and SQL (applier) threads are running.

The `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter` statement has no immediate effect. When you issue the `START REPLICA` statement for the next time following this `SET` statement, the new value for the system variable `sql_replica_skip_counter` or `sql_slave_skip_counter` is applied, and the events are skipped. That `START REPLICA` statement also automatically sets the value of the system variable back to 0. If the replica is a multi-source replica, when you issue that `START REPLICA` statement, the `FOR CHANNEL` clause is required. Make sure that you name the correct channel, otherwise events are skipped on the wrong channel.

Skipping Transactions With `CHANGE MASTER TO`

When you have assessed the failing transaction for any other appropriate actions as described previously (such as security considerations), identify the coordinates (file and position) in the source's binary log that represent a suitable position to restart replication. This can be the start of the event group following the event that caused the issue, or the start of the next transaction. The replication I/O (receiver) thread begins reading from the source at these coordinates the next time the thread starts, skipping the failing event. Make sure that you have identified the position accurately, because this statement does not take event groups into account.

Issue the `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement as follows, where *`source_log_name`* is the binary log file that contains the restart position, and *`source_log_pos`* is the number representing the restart position as stated in the binary log file:

```
CHANGE MASTER TO MASTER_LOG_FILE='source_log_name', MASTER_LOG_POS=source_log_pos;

Or from MySQL 8.0.24:
CHANGE REPLICATION SOURCE TO SOURCE_LOG_FILE='source_log_name', SOURCE_LOG_POS=source_log_pos;
```

If the replica is a multi-source replica, you must use the `FOR CHANNEL` clause to name the appropriate channel on the `CHANGE REPLICATION SOURCE TO` or `CHANGE MASTER TO` statement.

This statement cannot be issued if `SOURCE_AUTO_POSITION=1` or `MASTER_AUTO_POSITION=1` is set, or if the replication I/O (receiver) and SQL (applier) threads are running. If you need to use this method of skipping a transaction when `SOURCE_AUTO_POSITION=1` or `MASTER_AUTO_POSITION=1` is normally set, you can change the setting to `SOURCE_AUTO_POSITION=0` or `MASTER_AUTO_POSITION=0` while issuing the statement, then change it back again afterwards. For example:

```
CHANGE MASTER TO MASTER_AUTO_POSITION=0, MASTER_LOG_FILE='binlog.000145', MASTER_LOG_POS=235;
CHANGE MASTER TO MASTER_AUTO_POSITION=1;

Or from MySQL 8.0.24:

CHANGE REPLICATION SOURCE TO SOURCE_AUTO_POSITION=0, SOURCE_LOG_FILE='binlog.000145', SOURCE_LOG_POS=23
CHANGE REPLICATION SOURCE TO SOURCE_AUTO_POSITION=1;
```

17.2 Replication Implementation

Replication is based on the source server keeping track of all changes to its databases (updates, deletes, and so on) in its binary log. The binary log serves as a written record of all events that modify database structure or content (data) from the moment the server was started. Typically, `SELECT` statements are not recorded because they modify neither database structure nor content.

Each replica that connects to the source requests a copy of the binary log. That is, it pulls the data from the source, rather than the source pushing the data to the replica. The replica also executes the events from the binary log that it receives. This has the effect of repeating the original changes just as they were made on the source. Tables are created or their structure modified, and data is inserted, deleted, and updated according to the changes that were originally made on the source.

Because each replica is independent, the replaying of the changes from the source's binary log occurs independently on each replica that is connected to the source. In addition, because each replica

receives a copy of the binary log only by requesting it from the source, the replica is able to read and update the copy of the database at its own pace and can start and stop the replication process at will without affecting the ability to update to the latest database status on either the source or replica side.

For more information on the specifics of the replication implementation, see Section 17.2.3, “Replication Threads”.

Source servers and replicas report their status in respect of the replication process regularly so that you can monitor them. See Section 8.14, “Examining Server Thread (Process) Information”, for descriptions of all replicated-related states.

The source's binary log is written to a local relay log on the replica before it is processed. The replica also records information about the current position with the source's binary log and the local relay log. See Section 17.2.4, “Relay Log and Replication Metadata Repositories”.

Database changes are filtered on the replica according to a set of rules that are applied according to the various configuration options and variables that control event evaluation. For details on how these rules are applied, see Section 17.2.5, “How Servers Evaluate Replication Filtering Rules”.

17.2.1 Replication Formats

Replication works because events written to the binary log are read from the source and then processed on the replica. The events are recorded within the binary log in different formats according to the type of event. The different replication formats used correspond to the binary logging format used when the events were recorded in the source's binary log. The correlation between binary logging formats and the terms used during replication are:

- When using statement-based binary logging, the source writes SQL statements to the binary log. Replication of the source to the replica works by executing the SQL statements on the replica. This is called *statement-based replication* (which can be abbreviated as *SBR*), which corresponds to the MySQL statement-based binary logging format.
- When using row-based logging, the source writes *events* to the binary log that indicate how individual table rows are changed. Replication of the source to the replica works by copying the events representing the changes to the table rows to the replica. This is called *row-based replication* (which can be abbreviated as *RBR*).

  Row-based logging is the default method.
- You can also configure MySQL to use a mix of both statement-based and row-based logging, depending on which is most appropriate for the change to be logged. This is called *mixed-format logging*. When using mixed-format logging, a statement-based log is used by default. Depending on certain statements, and also the storage engine being used, the log is automatically switched to row-based in particular cases. Replication using the mixed format is referred to as *mixed-based replication* or *mixed-format replication*. For more information, see Section 5.4.4.3, “Mixed Binary Logging Format”.

NDB Cluster. The default binary logging format in MySQL NDB Cluster 8.0 is `MIXED`. You should note that NDB Cluster Replication always uses row-based replication, and that the `NDB` storage engine is incompatible with statement-based replication. See Section 23.7.2, “General Requirements for NDB Cluster Replication”, for more information.

When using `MIXED` format, the binary logging format is determined in part by the storage engine being used and the statement being executed. For more information on mixed-format logging and the rules governing the support of different logging formats, see Section 5.4.4.3, “Mixed Binary Logging Format”.

The logging format in a running MySQL server is controlled by setting the `binlog_format` server system variable. This variable can be set with session or global scope. The rules governing when and how the new setting takes effect are the same as for other MySQL server system variables. Setting the variable for the current session lasts only until the end of that session, and the change is not visible to other sessions. Setting the variable globally takes effect for clients that connect after the change,

but not for any current client sessions, including the session where the variable setting was changed. To make the global system variable setting permanent so that it applies across server restarts, you must set it in an option file. For more information, see Section 13.7.6.1, “SET Syntax for Variable Assignment”.

There are conditions under which you cannot change the binary logging format at runtime or doing so causes replication to fail. See Section 5.4.4.2, “Setting The Binary Log Format”.

Changing the global `binlog_format` value requires privileges sufficient to set global system variables. Changing the session `binlog_format` value requires privileges sufficient to set restricted session system variables. See Section 5.1.9.1, “System Variable Privileges”.

The statement-based and row-based replication formats have different issues and limitations. For a comparison of their relative advantages and disadvantages, see Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

With statement-based replication, you may encounter issues with replicating stored routines or triggers. You can avoid these issues by using row-based replication instead. For more information, see Section 25.7, “Stored Program Binary Logging”.

17.2.1.1 Advantages and Disadvantages of Statement-Based and Row-Based Replication

Each binary logging format has advantages and disadvantages. For most users, the mixed replication format should provide the best combination of data integrity and performance. If, however, you want to take advantage of the features specific to the statement-based or row-based replication format when performing certain tasks, you can use the information in this section, which provides a summary of their relative advantages and disadvantages, to determine which is best for your needs.

- Advantages of statement-based replication
- Disadvantages of statement-based replication
- Advantages of row-based replication
- Disadvantages of row-based replication

Advantages of statement-based replication

- Proven technology.
- Less data written to log files. When updates or deletes affect many rows, this results in *much* less storage space required for log files. This also means that taking and restoring from backups can be accomplished more quickly.
- Log files contain all statements that made any changes, so they can be used to audit the database.

Disadvantages of statement-based replication

- **Statements that are unsafe for SBR.**
  Not all statements which modify data (such as `INSERT DELETE`, `UPDATE`, and `REPLACE` statements) can be replicated using statement-based replication. Any nondeterministic behavior is difficult to replicate when using statement-based replication. Examples of such Data Modification Language (DML) statements include the following:
  - A statement that depends on a loadable function or stored program that is nondeterministic, since the value returned by such a function or stored program or depends on factors other than the parameters supplied to it. (Row-based replication, however, simply replicates the value returned by the function or stored program, so its effect on table rows and data is the same on both the source and replica.) See Section 17.5.1.16, “Replication of Invoked Features”, for more information.
  - `DELETE` and `UPDATE` statements that use a `LIMIT` clause without an `ORDER BY` are nondeterministic. See Section 17.5.1.18, “Replication and LIMIT”.

- Locking read statements (`SELECT ... FOR UPDATE` and `SELECT ... FOR SHARE`) that use `NOWAIT` or `SKIP LOCKED` options. See Locking Read Concurrency with NOWAIT and SKIP LOCKED.
- Deterministic loadable functions must be applied on the replicas.
- Statements using any of the following functions cannot be replicated properly using statement-based replication:
  - `LOAD_FILE()`
  - `UUID()`, `UUID_SHORT()`
  - `USER()`
  - `FOUND_ROWS()`
  - `SYSDATE()` (unless both the source and the replica are started with the `--sysdate-is-now` option)
  - `GET_LOCK()`
  - `IS_FREE_LOCK()`
  - `IS_USED_LOCK()`
  - `MASTER_POS_WAIT()`
  - `RAND()`
  - `RELEASE_LOCK()`
  - `SOURCE_POS_WAIT()`
  - `SLEEP()`
  - `VERSION()`

  However, all other functions are replicated correctly using statement-based replication, including `NOW()` and so forth.

  For more information, see Section 17.5.1.14, “Replication and System Functions”.

Statements that cannot be replicated correctly using statement-based replication are logged with a warning like the one shown here:

```
[Warning] Statement is not safe to log in statement format.
```

A similar warning is also issued to the client in such cases. The client can display it using `SHOW WARNINGS`.

- `INSERT ... SELECT` requires a greater number of row-level locks than with row-based replication.
- `UPDATE` statements that require a table scan (because no index is used in the `WHERE` clause) must lock a greater number of rows than with row-based replication.
- For `InnoDB`: An `INSERT` statement that uses `AUTO_INCREMENT` blocks other nonconflicting `INSERT` statements.
- For complex statements, the statement must be evaluated and executed on the replica before the rows are updated or inserted. With row-based replication, the replica only has to modify the affected rows, not execute the full statement.

- If there is an error in evaluation on the replica, particularly when executing complex statements, statement-based replication may slowly increase the margin of error across the affected rows over time. See Section 17.5.1.29, “Replica Errors During Replication”.
- Stored functions execute with the same `NOW()` value as the calling statement. However, this is not true of stored procedures.
- Deterministic loadable functions must be applied on the replicas.
- Table definitions must be (nearly) identical on source and replica. See Section 17.5.1.9, “Replication with Differing Table Definitions on Source and Replica”, for more information.
- As of MySQL 8.0.22, DML operations that read data from MySQL grant tables (through a join list or subquery) but do not modify them are performed as non-locking reads on the MySQL grant tables and are therefore not safe for statement-based replication. For more information, see Grant Table Concurrency.

Advantages of row-based replication

- All changes can be replicated. This is the safest form of replication.

Note

Statements that update the information in the `mysql` system schema, such as `GRANT`, `REVOKE` and the manipulation of triggers, stored routines (including stored procedures), and views, are all replicated to replicas using statement-based replication.

For statements such as `CREATE TABLE ... SELECT`, a `CREATE` statement is generated from the table definition and replicated using statement-based format, while the row insertions are replicated using row-based format.

- Fewer row locks are required on the source, which thus achieves higher concurrency, for the following types of statements:
  - `INSERT ... SELECT`
  - `INSERT` statements with `AUTO_INCREMENT`
  - `UPDATE` or `DELETE` statements with `WHERE` clauses that do not use keys or do not change most of the examined rows.
- Fewer row locks are required on the replica for any `INSERT`, `UPDATE`, or `DELETE` statement.

Disadvantages of row-based replication

- RBR can generate more data that must be logged. To replicate a DML statement (such as an `UPDATE` or `DELETE` statement), statement-based replication writes only the statement to the binary log. By contrast, row-based replication writes each changed row to the binary log. If the statement changes many rows, row-based replication may write significantly more data to the binary log; this is true even for statements that are rolled back. This also means that making and restoring a backup can require more time. In addition, the binary log is locked for a longer time to write the data, which may cause concurrency problems. Use `binlog_row_image=minimal` to reduce the disadvantage considerably.
- Deterministic loadable functions that generate large `BLOB` values take longer to replicate with row-based replication than with statement-based replication. This is because the `BLOB` column value is logged, rather than the statement generating the data.
- You cannot see on the replica what statements were received from the source and executed. However, you can see what data was changed using `mysqlbinlog` with the options `--base64-output=DECODE-ROWS` and `--verbose`.

Alternatively, use the `binlog_rows_query_log_events` variable, which if enabled adds a `Rows_query` event with the statement to `mysqlbinlog` output when the `-vv` option is used.

- For tables using the `MyISAM` storage engine, a stronger lock is required on the replica for `INSERT` statements when applying them as row-based events to the binary log than when applying them as statements. This means that concurrent inserts on `MyISAM` tables are not supported when using row-based replication.

17.2.1.2 Usage of Row-Based Logging and Replication

MySQL uses statement-based logging (SBL), row-based logging (RBL) or mixed-format logging. The type of binary log used impacts the size and efficiency of logging. Therefore the choice between row-based replication (RBR) or statement-based replication (SBR) depends on your application and environment. This section describes known issues when using a row-based format log, and describes some best practices using it in replication.

For additional information, see Section 17.2.1, “Replication Formats”, and Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

For information about issues specific to NDB Cluster Replication (which depends on row-based replication), see Section 23.7.3, “Known Issues in NDB Cluster Replication”.

- **Row-based logging of temporary tables.** As noted in Section 17.5.1.31, “Replication and Temporary Tables”, temporary tables are not replicated when using row-based format or (from MySQL 8.0.4) mixed format. For more information, see Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

  Temporary tables are not replicated when using row-based or mixed format because there is no need. In addition, because temporary tables can be read only from the thread which created them, there is seldom if ever any benefit obtained from replicating them, even when using statement-based format.

  You can switch from statement-based to row-based binary logging format at runtime even when temporary tables have been created. However, in MySQL 8.0, you cannot switch from row-based or mixed format for binary logging to statement-based format at runtime, due to any `CREATE TEMPORARY TABLE` statements having been omitted from the binary log in the previous mode.

  The MySQL server tracks the logging mode that was in effect when each temporary table was created. When a given client session ends, the server logs a `DROP TEMPORARY TABLE IF EXISTS` statement for each temporary table that still exists and was created when statement-based binary logging was in use. If row-based or mixed format binary logging was in use when the table was created, the `DROP TEMPORARY TABLE IF EXISTS` statement is not logged. In releases before MySQL 8.0.4 and 5.7.25, the `DROP TEMPORARY TABLE IF EXISTS` statement was logged regardless of the logging mode that was in effect.

  Nontransactional DML statements involving temporary tables are allowed when using `binlog_format=ROW`, as long as any nontransactional tables affected by the statements are temporary tables (Bug #14272672).

- **RBL and synchronization of nontransactional tables.** When many rows are affected, the set of changes is split into several events; when the statement commits, all of these events are written to the binary log. When executing on the replica, a table lock is taken on all tables involved, and then the rows are applied in batch mode. Depending on the engine used for the replica's copy of the table, this may or may not be effective.

- **Latency and binary log size.** RBL writes changes for each row to the binary log and so its size can increase quite rapidly. This can significantly increase the time required to make changes on the replica that match those on the source. You should be aware of the potential for this delay in your applications.

- **Reading the binary log.** `mysqlbinlog` displays row-based events in the binary log using the `BINLOG` statement (see Section 13.7.8.1, “BINLOG Statement”). This statement displays an event as a base 64-encoded string, the meaning of which is not evident. When invoked with the `--base64-output=DECODE-ROWS` and `--verbose` options, `mysqlbinlog` formats the contents of the binary log to be human readable. When binary log events were written in row-based format and you want to read or recover from a replication or database failure you can use this command to read contents of the binary log. For more information, see Section 4.6.9.2, “mysqlbinlog Row Event Display”.

- **Binary log execution errors and replica execution mode.** Using `slave_exec_mode=IDEMPOTENT` is generally only useful with MySQL NDB Cluster replication, for which `IDEMPOTENT` is the default value. (See Section 23.7.10, “NDB Cluster Replication: Bidirectional and Circular Replication”). When the system variable `replica_exec_mode` or `slave_exec_mode` is `IDEMPOTENT`, a failure to apply changes from RBL because the original row cannot be found does not trigger an error or cause replication to fail. This means that it is possible that updates are not applied on the replica, so that the source and replica are no longer synchronized. Latency issues and use of nontransactional tables with RBR when `replica_exec_mode` or `slave_exec_mode` is `IDEMPOTENT` can cause the source and replica to diverge even further. For more information about `replica_exec_mode` and `slave_exec_mode`, see Section 5.1.8, “Server System Variables”.

  For other scenarios, setting `replica_exec_mode` or `slave_exec_mode` to `STRICT` is normally sufficient; this is the default value for storage engines other than `NDB`.

- **Filtering based on server ID not supported.** You can filter based on server ID by using the `IGNORE_SERVER_IDS` option for the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). This option works with statement-based and row-based logging formats, but is deprecated for use when `GTID_MODE=ON` is set. Another method to filter out changes on some replicas is to use a `WHERE` clause that includes the relation `@@server_id <> id_value` clause with `UPDATE` and `DELETE` statements. For example, `WHERE @@server_id <> 1`. However, this does not work correctly with row-based logging. To use the `server_id` system variable for statement filtering, use statement-based logging.

- **RBL, nontransactional tables, and stopped replicas.** When using row-based logging, if the replica server is stopped while a replica thread is updating a nontransactional table, the replica database can reach an inconsistent state. For this reason, it is recommended that you use a transactional storage engine such as `InnoDB` for all tables replicated using the row-based format. Use of `STOP REPLICA` or `STOP REPLICA SQL_THREAD` (or before MySQL 8.0.22, SLAVE instead of REPLICA) prior to shutting down the replica MySQL server helps prevent issues from occurring, and is always recommended regardless of the logging format or storage engine you use.

17.2.1.3 Determination of Safe and Unsafe Statements in Binary Logging

The “safeness” of a statement in MySQL replication refers to whether the statement and its effects can be replicated correctly using statement-based format. If this is true of the statement, we refer to the statement as *safe*; otherwise, we refer to it as *unsafe*.

In general, a statement is safe if it deterministic, and unsafe if it is not. However, certain nondeterministic functions are *not* considered unsafe (see Nondeterministic functions not considered unsafe, later in this section). In addition, statements using results from floating-point math functions—which are hardware-dependent—are always considered unsafe (see Section 17.5.1.12, “Replication and Floating-Point Values”).

Handling of safe and unsafe statements. A statement is treated differently depending on whether the statement is considered safe, and with respect to the binary logging format (that is, the current value of `binlog_format`).

- When using row-based logging, no distinction is made in the treatment of safe and unsafe statements.

- When using mixed-format logging, statements flagged as unsafe are logged using the row-based format; statements regarded as safe are logged using the statement-based format.

- When using statement-based logging, statements flagged as being unsafe generate a warning to this effect. Safe statements are logged normally.

Each statement flagged as unsafe generates a warning. If a large number of such statements were executed on the source, this could lead to excessively large error log files. To prevent this, MySQL has a warning suppression mechanism. Whenever the 50 most recent `ER_BINLOG_UNSAFE_STATEMENT` warnings have been generated more than 50 times in any 50-second period, warning suppression is enabled. When activated, this causes such warnings not to be written to the error log; instead, for each 50 warnings of this type, a note `The last warning was repeated N times in last S seconds` is written to the error log. This continues as long as the 50 most recent such warnings were issued in 50 seconds or less; once the rate has decreased below this threshold, the warnings are once again logged normally. Warning suppression has no effect on how the safety of statements for statement-based logging is determined, nor on how warnings are sent to the client. MySQL clients still receive one warning for each such statement.

For more information, see Section 17.2.1, “Replication Formats”.

Statements considered unsafe.
Statements with the following characteristics are considered unsafe:

- **Statements containing system functions that may return a different value on the replica.** These functions include `FOUND_ROWS()`, `GET_LOCK()`, `IS_FREE_LOCK()`, `IS_USED_LOCK()`, `LOAD_FILE()`, `MASTER_POS_WAIT()`, `RAND()`, `RELEASE_LOCK()`, `ROW_COUNT()`, `SESSION_USER()`, `SLEEP()`, `SOURCE_POS_WAIT()`, `SYSDATE()`, `SYSTEM_USER()`, `USER()`, `UUID()`, and `UUID_SHORT()`.

  Nondeterministic functions not considered unsafe. Although these functions are not deterministic, they are treated as safe for purposes of logging and replication: `CONNECTION_ID()`, `CURDATE()`, `CURRENT_DATE()`, `CURRENT_TIME()`, `CURRENT_TIMESTAMP()`, `CURTIME()`,, `LAST_INSERT_ID()`, `LOCALTIME()`, `LOCALTIMESTAMP()`, `NOW()`, `UNIX_TIMESTAMP()`, `UTC_DATE()`, `UTC_TIME()`, and `UTC_TIMESTAMP()`.

  For more information, see Section 17.5.1.14, “Replication and System Functions”.

- **References to system variables.** Most system variables are not replicated correctly using the statement-based format. See Section 17.5.1.39, “Replication and Variables”. For exceptions, see Section 5.4.4.3, “Mixed Binary Logging Format”.

- **Loadable Functions.** Since we have no control over what a loadable function does, we must assume that it is executing unsafe statements.

- **Fulltext plugin.** This plugin may behave differently on different MySQL servers; therefore, statements depending on it could have different results. For this reason, all statements relying on the fulltext plugin are treated as unsafe in MySQL.

- **Trigger or stored program updates a table having an AUTO_INCREMENT column.** This is unsafe because the order in which the rows are updated may differ on the source and the replica.

  In addition, an `INSERT` into a table that has a composite primary key containing an `AUTO_INCREMENT` column that is not the first column of this composite key is unsafe.

  For more information, see Section 17.5.1.1, “Replication and AUTO_INCREMENT”.

- **INSERT ... ON DUPLICATE KEY UPDATE statements on tables with multiple primary or unique keys.** When executed against a table that contains more than one primary or unique key, this statement is considered unsafe, being sensitive to the order in which the storage engine checks the keys, which is not deterministic, and on which the choice of rows updated by the MySQL Server depends.

An `INSERT ... ON DUPLICATE KEY UPDATE` statement against a table having more than one unique or primary key is marked as unsafe for statement-based replication. (Bug #11765650, Bug #58637)

- **Updates using LIMIT.** The order in which rows are retrieved is not specified, and is therefore considered unsafe. See Section 17.5.1.18, “Replication and LIMIT”.
- **Accesses or references log tables.** The contents of the system log table may differ between source and replica.
- **Nontransactional operations after transactional operations.** Within a transaction, allowing any nontransactional reads or writes to execute after any transactional reads or writes is considered unsafe.

  For more information, see Section 17.5.1.35, “Replication and Transactions”.
- **Accesses or references self-logging tables.** All reads and writes to self-logging tables are considered unsafe. Within a transaction, any statement following a read or write to self-logging tables is also considered unsafe.
- **LOAD DATA statements.** `LOAD DATA` is treated as unsafe and when `binlog_format=MIXED` the statement is logged in row-based format. When `binlog_format=STATEMENT` `LOAD DATA` does not generate a warning, unlike other unsafe statements.
- **XA transactions.** If two XA transactions committed in parallel on the source are being prepared on the replica in the inverse order, locking dependencies can occur with statement-based replication that cannot be safely resolved, and it is possible for replication to fail with deadlock on the replica. When `binlog_format=STATEMENT` is set, DML statements inside XA transactions are flagged as being unsafe and generate a warning. When `binlog_format=MIXED` or `binlog_format=ROW` is set, DML statements inside XA transactions are logged using row-based replication, and the potential issue is not present.
- **`DEFAULT` clause that refers to a nondeterministic function.** If an expression default value refers to a nondeterministic function, any statement that causes the expression to be evaluated is unsafe for statement-based replication. This includes statements such as `INSERT`, `UPDATE`, and `ALTER TABLE`. Unlike most other unsafe statements, this category of statement cannot be replicated safely in row-based format. When `binlog_format` is set to `STATEMENT`, the statement is logged and executed but a warning message is written to the error log. When `binlog_format` is set to `MIXED` or `ROW`, the statement is not executed and an error message is written to the error log. For more information on the handling of explicit defaults, see Explicit Default Handling as of MySQL 8.0.13.

For additional information, see Section 17.5.1, “Replication Features and Issues”.

17.2.2 Replication Channels

In MySQL multi-source replication, a replica opens multiple replication channels, one for each source server. The replication channels represent the path of transactions flowing from a source to the replica. Each replication channel has its own receiver (I/O) thread, one or more applier (SQL) threads, and relay log. When transactions from a source are received by a channel's receiver thread, they are added to the channel's relay log file and passed through to the channel's applier threads. This enables each channel to function independently.

This section describes how channels can be used in a replication topology, and the impact they have on single-source replication. For instructions to configure sources and replicas for multi-source replication, to start, stop and reset multi-source replicas, and to monitor multi-source replication, see Section 17.1.5, “MySQL Multi-Source Replication”.

The maximum number of channels that can be created on one replica server in a multi-source replication topology is 256. Each replication channel must have a unique (nonempty) name, as

explained in Section 17.2.2.4, “Replication Channel Naming Conventions”. The error codes and messages that are issued when multi-source replication is enabled specify the channel that generated the error.

Note

Each channel on a multi-source replica must replicate from a different source. You cannot set up multiple replication channels from a single replica to a single source. This is because the server IDs of replicas must be unique in a replication topology. The source distinguishes replicas only by their server IDs, not by the names of the replication channels, so it cannot recognize different replication channels from the same replica.

A multi-source replica can also be set up as a multi-threaded replica, by setting the system variable `replica_parallel_workers` (from MySQL 8.0.26) or `slave_parallel_workers` (before MySQL 8.0.26) to a value greater than 0. When you do this on a multi-source replica, each channel on the replica has the specified number of applier threads, plus a coordinator thread to manage them. You cannot configure the number of applier threads for individual channels.

From MySQL 8.0, multi-source replicas can be configured with replication filters on specific replication channels. Channel specific replication filters can be used when the same database or table is present on multiple sources, and you only need the replica to replicate it from one source. For GTID-based replication, if the same transaction might arrive from multiple sources (such as in a diamond topology), you must ensure the filtering setup is the same on all channels. For more information, see Section 17.2.5.4, “Replication Channel Based Filters”.

To provide compatibility with previous versions, the MySQL server automatically creates on startup a default channel whose name is the empty string (`""`). This channel is always present; it cannot be created or destroyed by the user. If no other channels (having nonempty names) have been created, replication statements act on the default channel only, so that all replication statements from older replicas function as expected (see Section 17.2.2.2, “Compatibility with Previous Replication Statements”. Statements applying to replication channels as described in this section can be used only when there is at least one named channel.

17.2.2.1 Commands for Operations on a Single Channel

To enable MySQL replication operations to act on individual replication channels, use the `FOR CHANNEL` *`channel`* clause with the following replication statements:

- `CHANGE REPLICATION SOURCE TO`
- `CHANGE MASTER TO`
- `START REPLICA` (or before MySQL 8.0.22, `START SLAVE`)
- `STOP REPLICA` (or before MySQL 8.0.22, `STOP SLAVE`)
- `SHOW RELAYLOG EVENTS`
- `FLUSH RELAY LOGS`
- `SHOW REPLICA STATUS` (or before MySQL 8.0.22, `SHOW SLAVE STATUS`)
- `RESET REPLICA` (or before MySQL 8.0.22, `RESET SLAVE`)

The following functions have a `channel` parameter:

- `MASTER_POS_WAIT()`
- `SOURCE_POS_WAIT()`

The following statements are disallowed for the `group_replication_recovery` channel:

- `START REPLICA`
- `STOP REPLICA`

The following statements are disallowed for the `group_replication_applier` channel:

- `START REPLICA`
- `STOP REPLICA`
- `SHOW REPLICA STATUS`

`FLUSH RELAY LOGS` is now permitted for the `group_replication_applier` channel, but if the request is received while a transaction is being applied, the request is performed after the transaction ends. The requester must wait while the transaction is completed and the rotation takes place. This behavior prevents transactions from being split, which is not permitted for Group Replication.

17.2.2.2 Compatibility with Previous Replication Statements

When a replica has multiple channels and a `FOR CHANNEL` *`channel`* option is not specified, a valid statement generally acts on all available channels, with some specific exceptions.

For example, the following statements behave as expected for all except certain Group Replication channels:

- `START REPLICA` starts replication threads for all channels, except the `group_replication_recovery` and `group_replication_applier` channels.
- `STOP REPLICA` stops replication threads for all channels, except the `group_replication_recovery` and `group_replication_applier` channels.
- `SHOW REPLICA STATUS` reports the status for all channels, except the `group_replication_applier` channel.
- `RESET REPLICA` resets all channels.

Warning

Use `RESET REPLICA` with caution as this statement deletes all existing channels, purges their relay log files, and recreates only the default channel.

Some replication statements cannot operate on all channels. In this case, error 1964 `Multiple channels exist on the replica. Please provide channel name as an argument.` is generated. The following statements and functions generate this error when used in a multi-source replication topology and a `FOR CHANNEL` *`channel`* option is not used to specify which channel to act on:

- `SHOW RELAYLOG EVENTS`
- `CHANGE REPLICATION SOURCE TO`
- `CHANGE MASTER TO`
- `MASTER_POS_WAIT()`
- `SOURCE_POS_WAIT()`

Note that a default channel always exists in a single source replication topology, where statements and functions behave as in previous versions of MySQL.

17.2.2.3 Startup Options and Replication Channels

This section describes startup options which are impacted by the addition of replication channels.

The `master_info_repository` and `relay_log_info_repository` system variables must *not* be set to `FILE` when you use replication channels. In MySQL 8.0, the `FILE` setting is deprecated, and `TABLE` is the default, so the system variables can be omitted. From MySQL 8.0.23, they must be omitted because their use is deprecated from that release. If these system variables are set to `FILE`, attempting to add more sources to a replica fails with `ER_SLAVE_NEW_CHANNEL_WRONG_REPOSITORY`.

The following startup options now affect *all* channels in a replication topology.

- `--log-replica-updates` or `--log-slave-updates`

  All transactions received by the replica (even from multiple sources) are written in the binary log.

- `--relay-log-purge`

  When set, each channel purges its own relay log automatically.

- `--replica-transaction-retries` or `--slave-transaction-retries`

  The specified number of transaction retries can take place on all applier threads of all channels.

- `--skip-replica-start` or `--skip-slave-start` (or `skip_replica_start` or `skip_slave_start` system variable set)

  No replication threads start on any channels.

- `--replica-skip-errors` or `--slave-skip-errors`

  Execution continues and errors are skipped for all channels.

The values set for the following startup options apply on each channel; since these are `mysqld` startup options, they are applied on every channel.

- `--max-relay-log-size=`*`size`*

  Maximum size of the individual relay log file for each channel; after reaching this limit, the file is rotated.

- `--relay-log-space-limit=`*`size`*

  Upper limit for the total size of all relay logs combined, for each individual channel. For *`N`* channels, the combined size of these logs is limited to `relay_log_space_limit * `*`N`*.

- `--replica-parallel-workers=`*`value`* or `--slave-parallel-workers=`*`value`*

  Number of replication applier threads per channel.

- `replica_checkpoint_group` or `slave_checkpoint_group`

  Waiting time by an receiver thread for each source.

- `--relay-log-index=filename`

  Base name for each channel's relay log index file. See Section 17.2.2.4, “Replication Channel Naming Conventions”.

- `--relay-log=filename`

  Denotes the base name of each channel's relay log file. See Section 17.2.2.4, “Replication Channel Naming Conventions”.

- `--replica-net-timeout=N` or `--slave-net-timeout=N`

  This value is set per channel, so that each channel waits for *`N`* seconds to check for a broken connection.

- `--replica-skip-counter=N` or `--slave-skip-counter=N`

  This value is set per channel, so that each channel skips *`N`* events from its source.

17.2.2.4 Replication Channel Naming Conventions

This section describes how naming conventions are impacted by replication channels.

Each replication channel has a unique name which is a string with a maximum length of 64 characters and is case-insensitive. Because channel names are used in the replica's applier metadata repository table, the character set used for these is always UTF-8. Although you are generally free to use any name for channels, the following names are reserved:

- `group_replication_applier`
- `group_replication_recovery`

The name you choose for a replication channel also influences the file names used by a multi-source replica. The relay log files and index files for each channel are named *`relay_log_basename-channel`*`.xxxxxx`, where *`relay_log_basename`* is a base name specified using the `relay_log` system variable, and *`channel`* is the name of the channel logged to this file. If you do not specify the `relay_log` system variable, a default file name is used that also includes the name of the channel.

17.2.3 Replication Threads

MySQL replication capabilities are implemented using three main threads, one on the source server and two on the replica:

- **Binary log dump thread.** The source creates a thread to send the binary log contents to a replica when the replica connects. This thread can be identified in the output of `SHOW PROCESSLIST` on the source as the `Binlog Dump` thread.

  The binary log dump thread acquires a lock on the source's binary log for reading each event that is to be sent to the replica. As soon as the event has been read, the lock is released, even before the event is sent to the replica.

- **Replication I/O receiver thread.** When a `START REPLICA` statement is issued on a replica server, the replica creates an I/O (receiver) thread, which connects to the source and asks it to send the updates recorded in its binary logs.

  The replication receiver thread reads the updates that the source's `Binlog Dump` thread sends (see previous item) and copies them to local files that comprise the replica's relay log.

  The state of this thread is shown as `Slave_IO_running` in the output of `SHOW REPLICA STATUS`.

- **Replication SQL applier thread.** The replica creates an SQL (applier) thread to read the relay log that is written by the replication receiver thread and execute the transactions contained in it.

There are three main threads for each connection between a source and a replica. A source that has multiple replicas creates one binary log dump thread for each replica currently connected; each replica has its own replication receiver and applier threads.

A replica uses two threads to separate reading updates from the source and executing them into independent tasks. Thus, the task of reading transactions is not slowed down if the process of applying

them is slow. For example, if the replica server has not been running for a while, its receiver thread can quickly fetch all the binary log contents from the source when the replica starts, even if the applier thread lags far behind. If the replica stops before the SQL thread has executed all the fetched statements, the receiver thread has at least fetched everything so that a safe copy of the transactions is stored locally in the replica's relay logs, ready for execution the next time that the replica starts.

You can enable further parallelization for tasks on a replica by setting the system variable `replica_parallel_workers` (MySQL 8.0.26 or later) or `slave_parallel_workers` (prior to MySQL 8.0.26) to a value greater than 0 (the default). When this system variable is set, the replica creates the specified number of worker threads to apply transactions, plus a coordinator thread to manage them. If you are using multiple replication channels, each channel has this number of threads. A replica with `replica_parallel_workers` or `slave_parallel_workers` set to a value greater than 0 is called a multithreaded replica. With this setup, transactions that fail can be retried.

Note

Multithreaded replicas are supported by NDB Cluster beginning with NDB 8.0.33. (Previously, `NDB` silently ignored any setting for `replica_parallel_workers`.) See Section 23.7.11, “NDB Cluster Replication Using the Multithreaded Applier”, for more information.

17.2.3.1 Monitoring Replication Main Threads

The `SHOW PROCESSLIST` statement provides information that tells you what is happening on the source and on the replica regarding replication. For information on source states, see Section 8.14.4, “Replication Source Thread States”. For replica states, see Section 8.14.5, “Replication I/O (Receiver) Thread States”, and Section 8.14.6, “Replication SQL Thread States”.

The following example illustrates how the three main replication threads, the binary log dump thread, replication I/O (receiver) thread, and replication SQL (applier) thread, show up in the output from `SHOW PROCESSLIST`.

On the source server, the output from `SHOW PROCESSLIST` looks like this:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 2
   User: root
   Host: localhost:32931
     db: NULL
Command: Binlog Dump
   Time: 94
  State: Has sent all binlog to slave; waiting for binlog to
         be updated
   Info: NULL
```

Here, thread 2 is a `Binlog Dump` thread that services a connected replica. The `State` information indicates that all outstanding updates have been sent to the replica and that the source is waiting for more updates to occur. If you see no `Binlog Dump` threads on a source server, this means that replication is not running; that is, no replicas are currently connected.

On a replica server, the output from `SHOW PROCESSLIST` looks like this:

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
     Id: 10
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 11
  State: Waiting for master to send event
   Info: NULL
```

```
*************************** 2. row ***************************
     Id: 11
   User: system user
   Host:
     db: NULL
Command: Connect
   Time: 11
  State: Has read all relay log; waiting for the slave I/O
         thread to update it
   Info: NULL
```

The `State` information indicates that thread 10 is the replication I/O (receiver) thread that is communicating with the source server, and thread 11 is the replication SQL (applier) thread that is processing the updates stored in the relay logs. At the time that `SHOW PROCESSLIST` was run, both threads were idle, waiting for further updates.

The value in the `Time` column can show how late the replica is compared to the source. See Section A.14, “MySQL 8.0 FAQ: Replication”. If sufficient time elapses on the source side without activity on the `Binlog Dump` thread, the source determines that the replica is no longer connected. As for any other client connection, the timeouts for this depend on the values of `net_write_timeout` and `net_retry_count`; for more information about these, see Section 5.1.8, “Server System Variables”.

The `SHOW REPLICA STATUS` statement provides additional information about replication processing on a replica server. See Section 17.1.7.1, “Checking Replication Status”.

17.2.3.2 Monitoring Replication Applier Worker Threads

On a multithreaded replica, the Performance Schema tables `replication_applier_status_by_coordinator` and `replication_applier_status_by_worker` show status information for the replica's coordinator thread and applier worker threads respectively. For a replica with multiple channels, the threads for each channel are identified.

A multithreaded replica's coordinator thread also prints statistics to the replica's error log on a regular basis if the verbosity setting is set to display informational messages. The statistics are printed depending on the volume of events that the coordinator thread has assigned to applier worker threads, with a maximum frequency of once every 120 seconds. The message lists the following statistics for the relevant replication channel, or the default replication channel (which is not named):

Seconds elapsed	The difference in seconds between the current time and the last time this information was printed to the error log.
Events assigned	The total number of events that the coordinator thread has queued to all applier worker threads since the coordinator thread was started.
Worker queues filled over overrun level	The current number of events that are queued to any of the applier worker threads in excess of the overrun level, which is set at 90% of the maximum queue length of 16384 events. If this value is zero, no applier worker threads are operating at the upper limit of their capacity.
Waited due to worker queue full	The number of times that the coordinator thread had to wait to schedule an event because an applier worker thread's queue was full. If this value is zero, no applier worker threads exhausted their capacity.
Waited due to the total size	The number of times that the coordinator thread had to wait to schedule an event because the `replica_pending_jobs_size_max` or

	`slave_pending_jobs_size_max` limit had been reached. This system variable sets the maximum amount of memory (in bytes) available to applier worker thread queues holding events not yet applied. If an unusually large event exceeds this size, the transaction is held until all the applier worker threads have empty queues, and then processed. All subsequent transactions are held until the large transaction has been completed.
Waited at clock conflicts	The number of nanoseconds that the coordinator thread had to wait to schedule an event because a transaction that the event depended on had not yet been committed. If `replica_parallel_type` or `slave_parallel_type` is set to `DATABASE` (rather than `LOGICAL_CLOCK`), this value is always zero.
Waited (count) when workers occupied	The number of times that the coordinator thread slept for a short period, which it might do in two situations. The first situation is where the coordinator thread assigns an event and finds the applier worker thread's queue is filled beyond the underrun level of 10% of the maximum queue length, in which case it sleeps for a maximum of 1 millisecond. The second situation is where `replica_parallel_type` or `slave_parallel_type` is set to `LOGICAL_CLOCK` and the coordinator thread needs to assign the first event of a transaction to an applier worker thread's queue, it only does this to a worker with an empty queue, so if no queues are empty, the coordinator thread sleeps until one becomes empty.
Waited when workers occupied	The number of nanoseconds that the coordinator thread slept while waiting for an empty applier worker thread queue (that is, in the second situation described above, where `replica_parallel_type` or `slave_parallel_type` is set to `LOGICAL_CLOCK` and the first event of a transaction needs to be assigned).

17.2.4 Relay Log and Replication Metadata Repositories

A replica server creates several repositories of information to use for the replication process:

- The replica's *relay log*, which is written by the replication I/O (receiver) thread, contains the transactions read from the replication source server's binary log. The transactions in the relay log are applied on the replica by the replication SQL (applier) thread. For information about the relay log, see Section 17.2.4.1, “The Relay Log”.
- The replica's *connection metadata repository* contains information that the replication receiver thread needs to connect to the replication source server and retrieve transactions from the source's binary log. The connection metadata repository is written to the `mysql.slave_master_info` table.
- The replica's *applier metadata repository* contains information that the replication applier thread needs to read and apply transactions from the replica's relay log. The applier metadata repository is written to the `mysql.slave_relay_log_info` table.

The replica's connection metadata repository and applier metadata repository are collectively known as the replication metadata repositories. For information about these, see Section 17.2.4.2, “Replication Metadata Repositories”.

Making replication resilient to unexpected halts. The `mysql.slave_master_info` and `mysql.slave_relay_log_info` tables are created using the transactional storage engine `InnoDB`. Updates to the replica's applier metadata repository table are committed together with the transactions, meaning that the replica's progress information recorded in that repository is always consistent with what has been applied to the database, even in the event of an unexpected server halt. For

information on the combination of settings on the replica that is most resilient to unexpected halts, see Section 17.4.2, “Handling an Unexpected Halt of a Replica”.

17.2.4.1 The Relay Log

The relay log, like the binary log, consists of a set of numbered files containing events that describe database changes, and an index file that contains the names of all used relay log files. The default location for relay log files is the data directory.

The term “relay log file” generally denotes an individual numbered file containing database events. The term “relay log” collectively denotes the set of numbered relay log files plus the index file.

Relay log files have the same format as binary log files and can be read using `mysqlbinlog` (see Section 4.6.9, “mysqlbinlog — Utility for Processing Binary Log Files”). If binary log transaction compression (available as of MySQL 8.0.20) is in use, transaction payloads written to the relay log are compressed in the same way as for the binary log. For more information on binary log transaction compression, see Section 5.4.4.5, “Binary Log Transaction Compression”.

For the default replication channel, relay log file names have the default form *`host_name`*`-relay-bin.`*`nnnnnn`*, where *`host_name`* is the name of the replica server host and *`nnnnnn`* is a sequence number. Successive relay log files are created using successive sequence numbers, beginning with `000001`. For non-default replication channels, the default base name is *`host_name`*`-relay-bin-`*`channel`*, where *`channel`* is the name of the replication channel recorded in the relay log.

The replica uses an index file to track the relay log files currently in use. The default relay log index file name is *`host_name`*`-relay-bin.index` for the default channel, and *`host_name`*`-relay-bin-`*`channel`*`.index` for non-default replication channels.

The default relay log file and relay log index file names and locations can be overridden with, respectively, the `relay_log` and `relay_log_index` system variables (see Section 17.1.6, “Replication and Binary Logging Options and Variables”).

If a replica uses the default host-based relay log file names, changing a replica's host name after replication has been set up can cause replication to fail with the errors `Failed to open the relay log` and `Could not find target log during relay log initialization`. This is a known issue (see Bug #2122). If you anticipate that a replica's host name might change in the future (for example, if networking is set up on the replica such that its host name can be modified using DHCP), you can avoid this issue entirely by using the `relay_log` and `relay_log_index` system variables to specify relay log file names explicitly when you initially set up the replica. This causes the names to be independent of server host name changes.

If you encounter the issue after replication has already begun, one way to work around it is to stop the replica server, prepend the contents of the old relay log index file to the new one, and then restart the replica. On a Unix system, this can be done as shown here:

```
$> cat new_relay_log_name.index >> old_relay_log_name.index
$> mv old_relay_log_name.index new_relay_log_name.index
```

A replica server creates a new relay log file under the following conditions:

- Each time the replication I/O (receiver) thread starts.
- When the logs are flushed (for example, with `FLUSH LOGS` or `mysqladmin flush-logs`).
- When the size of the current relay log file becomes too large, which is determined as follows:
  - If the value of `max_relay_log_size` is greater than 0, that is the maximum relay log file size.
  - If the value of `max_relay_log_size` is 0, `max_binlog_size` determines the maximum relay log file size.

The replication SQL (applier) thread automatically deletes each relay log file after it has executed all events in the file and no longer needs it. There is no explicit mechanism for deleting relay logs because the replication SQL thread takes care of doing so. However, `FLUSH LOGS` rotates relay logs, which influences when the replication SQL thread deletes them.

17.2.4.2 Replication Metadata Repositories

A replica server creates two replication metadata repositories, the connection metadata repository and the applier metadata repository. The replication metadata repositories survive a replica server's shutdown. If binary log file position based replication is in use, when the replica restarts, it reads the two repositories to determine how far it previously proceeded in reading the binary log from the source and in processing its own relay log. If GTID-based replication is in use, the replica does not use the replication metadata repositories for that purpose, but does need them for the other metadata that they contain.

- The replica's *connection metadata repository* contains information that the replication I/O (receiver) thread needs to connect to the replication source server and retrieve transactions from the source's binary log. The metadata in this repository includes the connection configuration, the replication user account details, the SSL settings for the connection, and the file name and position where the replication receiver thread is currently reading from the source's binary log.
- The replica's *applier metadata repository* contains information that the replication SQL (applier) thread needs to read and apply transactions from the replica's relay log. The metadata in this repository includes the file name and position up to which the replication applier thread has executed the transactions in the relay log, and the equivalent position in the source's binary log. It also includes metadata for the process of applying transactions, such as the number of worker threads and the `PRIVILEGE_CHECKS_USER` account for the channel.

The connection metadata repository is written to the `slave_master_info` table in the `mysql` system schema, and the applier metadata repository is written to the `slave_relay_log_info` table in the `mysql` system schema. A warning message is issued if `mysqld` is unable to initialize the tables for the replication metadata repositories, but the replica is allowed to continue starting. This situation is most likely to occur when upgrading from a version of MySQL that does not support the use of tables for the repositories to one in which they are supported.

Important

1. Do not attempt to update or insert rows in the `mysql.slave_master_info` or `mysql.slave_relay_log_info` tables manually. Doing so can cause undefined behavior, and is not supported. Execution of any statement requiring a write lock on either or both of the `slave_master_info` and `slave_relay_log_info` tables is disallowed while replication is ongoing (although statements that perform only reads are permitted at any time).
2. Access privileges for the connection metadata repository table `mysql.slave_master_info` should be restricted to the database administrator, because it contains the replication user account name and password for connecting to the source. Use a restricted access mode to protect database backups that include this table. From MySQL 8.0.21, you can clear the replication user account credentials from the connection metadata repository, and instead always provide them using the `START REPLICA` statement or `START GROUP_REPLICATION` statement that starts the replication channel. This approach means that the replication channel always needs operator intervention to restart, but the account name and password are not recorded in the replication metadata repositories.

`RESET REPLICA` clears the data in the replication metadata repositories, with the exception of the replication connection parameters (depending on the MySQL Server release). For details, see the description for `RESET REPLICA`.

From MySQL 8.0.27, you can set the `GTID_ONLY` option on the `CHANGE REPLICATION SOURCE TO` statement to stop a replication channel from persisting file names and file positions in the replication metadata repositories. This avoids writes and reads to the tables in situations where GTID-based replication does not actually require them. With the `GTID_ONLY` setting, the connection metadata repository and the applier metadata repository are not updated when the replica queues and applies events in a transaction, or when the replication threads are stopped and started. File positions are tracked in memory, and can be viewed using a `SHOW REPLICA STATUS` statement if they are needed. The replication metadata repositories are only synchronized in the following situations:

- When a `CHANGE REPLICATION SOURCE TO` statement is issued.
- When a `RESET REPLICA` statement is issued. `RESET REPLICA ALL` deletes rather than updates the repositories, so they are synchronized implicitly.
- When a replication channel is initialized.
- If the replication metadata repositories are moved from files to tables.

Before MySQL 8.0, to create the replication metadata repositories as tables, it was necessary to specify `master_info_repository=TABLE` and `relay_log_info_repository=TABLE` at server startup. Otherwise, the repositories were created as files in the data directory named `master.info` and `relay-log.info`, or with alternative names and locations specified by the `--master-info-file` option and `relay_log_info_file` system variable. From MySQL 8.0, creating the replication metadata repositories as tables is the default, and the use of all these system variables is deprecated.

The `mysql.slave_master_info` and `mysql.slave_relay_log_info` tables are created using the `InnoDB` transactional storage engine. Updates to the applier metadata repository table are committed together with the transactions, meaning that the replica's progress information recorded in that repository is always consistent with what has been applied to the database, even in the event of an unexpected server halt. For information on the combination of settings on a replica that is most resilient to unexpected halts, see Section 17.4.2, “Handling an Unexpected Halt of a Replica”.

When you back up the replica's data or transfer a snapshot of its data to create a new replica, ensure that you include the `mysql.slave_master_info` and `mysql.slave_relay_log_info` tables containing the replication metadata repositories. For cloning operations, note that when the replication metadata repositories are created as tables, they are copied to the recipient during a cloning operation, but when they are created as files, they are not copied. When binary log file position based replication is in use, the replication metadata repositories are needed to resume replication after restarting the restored, copied, or cloned replica. If you do not have the relay log files, but still have the applier metadata repository, you can check it to determine how far the replication SQL thread has executed in the source's binary log. Then you can use a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) with the `SOURCE_LOG_FILE` | `MASTER_LOG_FILE` and `SOURCE_LOG_POS` | `MASTER_LOG_POS` options to tell the replica to re-read the binary logs from the source from that point (provided that the required binary logs still exist on the source).

One additional repository, the applier worker metadata repository, is created primarily for internal use, and holds status information about worker threads on a multithreaded replica. The applier worker metadata repository includes the names and positions for the relay log file and the source's binary log file for each worker thread. If the applier metadata repository is created as a table, which is the default, the applier worker metadata repository is written to the `mysql.slave_worker_info` table. If the applier metadata repository is written to a file, the applier worker metadata repository is written to the `worker-relay-log.info` file. For external use, status information for worker threads is presented in the Performance Schema `replication_applier_status_by_worker` table.

The replication metadata repositories originally contained information similar to that shown in the output of the `SHOW REPLICA STATUS` statement, which is discussed in Section 13.4.2, “SQL Statements for Controlling Replica Servers”. Further information has since been added to the replication metadata repositories which is not displayed by the `SHOW REPLICA STATUS` statement.

For the connection metadata repository, the following table shows the correspondence between the columns in the `mysql.slave_master_info` table, the columns displayed by `SHOW REPLICA STATUS`, and the lines in the deprecated `master.info` file.

`slave_master_info` Table Column	`SHOW REPLICA STATUS` Column	`master.info` File Line	Description
`Number_of_lines`	[None]	1	Number of columns in the table (or lines in the file)
`Master_log_name`	`Source_Log_File`	2	The name of the binary log currently being read from the source
`Master_log_pos`	`Read_Source_Log_Pos`	3	The current position within the binary log that has been read from the source
`Host`	`Source_Host`	4	The host name of the replication source server
`User_name`	`Source_User`	5	The replication user account name used to connect to the source
`User_password`	Password (not shown by `SHOW REPLICA STATUS`)	6	The replication user account password used to connect to the source
`Port`	`Source_Port`	7	The network port used to connect to the replication source server
`Connect_retry`	`Connect_Retry`	8	The period (in seconds) that the replica waits before trying to reconnect to the source
`Enabled_ssl`	`Source_SSL_Allowed`	9	Whether the replica supports SSL connections
`Ssl_ca`	`Source_SSL_CA_File`	10	The file used for the Certificate Authority (CA) certificate
`Ssl_capath`	`Source_SSL_CA_Path`	11	The path to the Certificate Authority (CA) certificate
`Ssl_cert`	`Source_SSL_Cert`	12	The name of the SSL certificate file
`Ssl_cipher`	`Source_SSL_Cipher`	13	The list of possible ciphers used in the handshake for the SSL connection
`Ssl_key`	`Source_SSL_Key`	14	The name of the SSL key file
`Ssl_verify_server_cert`	`Source_SSL_Verify_Server_Cert`	15	Whether to verify the server certificate

`slave_master_info` Table Column	`SHOW REPLICA STATUS` Column	`master.info` File Line	Description
`Heartbeat`	[None]	16	Interval between replication heartbeats, in seconds
`Bind`	`Source_Bind`	17	Which of the replica's network interfaces should be used for connecting to the source
`Ignored_server_ids`	`Replicate_Ignore_Server_Ids`	18	The list of server IDs to be ignored. Note that for `Ignored_server_ids` the list of server IDs is preceded by the total number of server IDs to ignore.
`Uuid`	`Source_UUID`	19	The source's unique ID
`Retry_count`	`Source_Retry_Count`	20	Maximum number of reconnection attempts permitted
`Ssl_crl`	[None]	21	Path to an SSL certificate revocation-list file
`Ssl_crlpath`	[None]	22	Path to a directory containing SSL certificate revocation-list files
`Enabled_auto_position`	`Auto_position`	23	Whether GTID auto-positioning is in use or not
`Channel_name`	`Channel_name`	24	The name of the replication channel
`Tls_version`	`Source_TLS_Version`	25	TLS version on the source
`Public_key_path`	`Source_public_key_path`	26	Name of the RSA public key file
`Get_public_key`	`Get_source_public_key`	27	Whether to request RSA public key from source
`Network_namespace`	`Network_namespace`	28	Network namespace
`Master_compression_algorithm`	[None]	29	Permitted compression algorithms for the connection to the source
`Master_zstd_compression_level`	[None]	30	`zstd` compression level
`Tls_ciphersuites`	[None]	31	Permitted ciphersuites for TLSv1.3
`Source_connection_auto_failover`	[None]	32	Whether the asynchronous connection failover mechanism is activated
`Gtid_only`	[None]	33	Whether the channel uses only GTIDs

`slave_master_info` Table Column	`SHOW REPLICA STATUS` Column	`master.info` File Line	Description
			and does not persist positions

For the applier metadata repository, the following table shows the correspondence between the columns in the `mysql.slave_relay_log_info` table, the columns displayed by `SHOW REPLICA STATUS`, and the lines in the deprecated `relay-log.info` file.

`slave_relay_log_info` Table Column	`SHOW REPLICA STATUS` Column	Line in `relay-log.info` File	Description
`Number_of_lines`	[None]	1	Number of columns in the table or lines in the file
`Relay_log_name`	`Relay_Log_File`	2	The name of the current relay log file
`Relay_log_pos`	`Relay_Log_Pos`	3	The current position within the relay log file; events up to this position have been executed on the replica database
`Master_log_name`	`Relay_Source_Log_File`	4	The name of the source's binary log file from which the events in the relay log file were read
`Master_log_pos`	`Exec_Source_Log_Pos`	5	The equivalent position within the source's binary log file of the events that have been executed on the replica
`Sql_delay`	`SQL_Delay`	6	The number of seconds that the replica must lag the source
`Number_of_workers`	[None]	7	The number of worker threads for applying replication transactions in parallel
`Id`	[None]	8	ID used for internal purposes; currently this is always 1
`Channel_name`	`Channel_name`	9	The name of the replication channel
`Privilege_checks_username`	[None]	10	The user name for the `PRIVILEGE_CHECKS_USER` account for the channel
`Privilege_checks_hostname`	[None]	11	The host name for the `PRIVILEGE_CHECKS_USER` account for the channel
`Require_row_format`	[None]	12	Whether the channel accepts only row-based events

`slave_relay_log_info` Table Column	`SHOW REPLICA STATUS` Column	Line in `relay-log.info` File	Description
`Require_table_primary_key_check`	[None]	13	The channel's policy on whether tables must have primary keys for `CREATE TABLE` and `ALTER TABLE` operations
`Assign_gtids_to_anonymous_transactions_type`	[None]	14	If the channel assigns a GTID to replicated transactions that do not already have one, using the replica's local UUID, this value is `LOCAL`; if the channel does so using instead a UUID which has been set manually, the value is `UUID`. If the channel does not assign a GTID in such cases, the value is `OFF`.
`Assign_gtids_to_anonymous_transactions_value`	[None]	15	The UUID used in the GTIDs assigned to anonymous transactions

17.2.5 How Servers Evaluate Replication Filtering Rules

If a replication source server does not write a statement to its binary log, the statement is not replicated. If the server does log the statement, the statement is sent to all replicas and each replica determines whether to execute it or ignore it.

On the source, you can control which databases to log changes for by using the `--binlog-do-db` and `--binlog-ignore-db` options to control binary logging. For a description of the rules that servers use in evaluating these options, see Section 17.2.5.1, “Evaluation of Database-Level Replication and Binary Logging Options”. You should not use these options to control which databases and tables are replicated. Instead, use filtering on the replica to control the events that are executed on the replica.

On the replica side, decisions about whether to execute or ignore statements received from the source are made according to the `--replicate-*` options that the replica was started with. (See Section 17.1.6, “Replication and Binary Logging Options and Variables”.) The filters governed by these options can also be set dynamically using the `CHANGE REPLICATION FILTER` statement. The rules governing such filters are the same whether they are created on startup using `--replicate-*` options or while the replica server is running by `CHANGE REPLICATION FILTER`. Note that replication filters cannot be used on Group Replication-specific channels on a MySQL server instance that is configured for Group Replication, because filtering transactions on some servers would make the group unable to reach agreement on a consistent state.

In the simplest case, when there are no `--replicate-*` options, the replica executes all statements that it receives from the source. Otherwise, the result depends on the particular options given.

Database-level options (`--replicate-do-db`, `--replicate-ignore-db`) are checked first; see Section 17.2.5.1, “Evaluation of Database-Level Replication and Binary Logging Options”, for a description of this process. If no database-level options are used, option checking proceeds to any table-level options that may be in use (see Section 17.2.5.2, “Evaluation of Table-Level Replication Options”, for a discussion of these). If one or more database-level options are used but none are matched, the statement is not replicated.

For statements affecting databases only (that is, `CREATE DATABASE`, `DROP DATABASE`, and `ALTER DATABASE`), database-level options always take precedence over any `--replicate-wild-do-table` options. In other words, for such statements, `--replicate-wild-do-table` options are checked if and only if there are no database-level options that apply.

To make it easier to determine what effect a given set of options has, it is recommended that you avoid mixing `do-*` and `ignore-*` options, or options containing wildcards with options which do not.

If any `--replicate-rewrite-db` options were specified, they are applied before the `--replicate-*` filtering rules are tested.

Note

All replication filtering options follow the same rules for case sensitivity that apply to names of databases and tables elsewhere in the MySQL server, including the effects of the `lower_case_table_names` system variable.

Beginning with MySQL 8.0.31, filtering rules are applied before performing any privilege checks; if a transaction is filtered out, no privilege check is performed for that transaction, and thus no error can be raised by it. See Section 17.5.1.29, “Replica Errors During Replication”, for more information.

17.2.5.1 Evaluation of Database-Level Replication and Binary Logging Options

When evaluating replication options, the replica begins by checking to see whether there are any `--replicate-do-db` or `--replicate-ignore-db` options that apply. When using `--binlog-do-db` or `--binlog-ignore-db`, the process is similar, but the options are checked on the source.

The database that is checked for a match depends on the binary log format of the statement that is being handled. If the statement has been logged using the row format, the database where data is to be changed is the database that is checked. If the statement has been logged using the statement format, the default database (specified with a `USE` statement) is the database that is checked.

Note

Only DML statements can be logged using the row format. DDL statements are always logged as statements, even when `binlog_format=ROW`. All DDL statements are therefore always filtered according to the rules for statement-based replication. This means that you must select the default database explicitly with a `USE` statement in order for a DDL statement to be applied.

For replication, the steps involved are listed here:

1. Which logging format is used?
   - **STATEMENT.** Test the default database.
   - **ROW.** Test the database affected by the changes.
2. Are there any `--replicate-do-db` options?
   - **Yes.** Does the database match any of them?
     - **Yes.** Continue to Step 4.
     - **No.** Ignore the update and exit.
   - **No.** Continue to step 3.
3. Are there any `--replicate-ignore-db` options?
   - **Yes.** Does the database match any of them?

- **Yes.** Ignore the update and exit.
- **No.** Continue to step 4.

- **No.** Continue to step 4.

4. Proceed to checking the table-level replication options, if there are any. For a description of how these options are checked, see Section 17.2.5.2, “Evaluation of Table-Level Replication Options”.

Important

A statement that is still permitted at this stage is not yet actually executed. The statement is not executed until all table-level options (if any) have also been checked, and the outcome of that process permits execution of the statement.

For binary logging, the steps involved are listed here:

1. Are there any `--binlog-do-db` or `--binlog-ignore-db` options?
   - **Yes.** Continue to step 2.
   - **No.** Log the statement and exit.
2. Is there a default database (has any database been selected by `USE`)?
   - **Yes.** Continue to step 3.
   - **No.** Ignore the statement and exit.
3. There is a default database. Are there any `--binlog-do-db` options?
   - **Yes.** Do any of them match the database?
     - **Yes.** Log the statement and exit.
     - **No.** Ignore the statement and exit.
   - **No.** Continue to step 4.
4. Do any of the `--binlog-ignore-db` options match the database?
   - **Yes.** Ignore the statement and exit.
   - **No.** Log the statement and exit.

Important

For statement-based logging, an exception is made in the rules just given for the `CREATE DATABASE`, `ALTER DATABASE`, and `DROP DATABASE` statements. In those cases, the database being *created, altered, or dropped* replaces the default database when determining whether to log or ignore updates.

`--binlog-do-db` can sometimes mean “ignore other databases”. For example, when using statement-based logging, a server running with only `--binlog-do-db=sales` does not write to the binary log statements for which the default database differs from `sales`. When using row-based logging with the same option, the server logs only those updates that change data in `sales`.

17.2.5.2 Evaluation of Table-Level Replication Options

The replica checks for and evaluates table options only if either of the following two conditions is true:

- No matching database options were found.
- One or more database options were found, and were evaluated to arrive at an “execute” condition according to the rules described in the previous section (see Section 17.2.5.1, “Evaluation of Database-Level Replication and Binary Logging Options”).

First, as a preliminary condition, the replica checks whether statement-based replication is enabled. If so, and the statement occurs within a stored function, the replica executes the statement and exits. If row-based replication is enabled, the replica does not know whether a statement occurred within a stored function on the source, so this condition does not apply.

Note

For statement-based replication, replication events represent statements (all changes making up a given event are associated with a single SQL statement); for row-based replication, each event represents a change in a single table row (thus a single statement such as `UPDATE mytable SET mycol = 1` may yield many row-based events). When viewed in terms of events, the process of checking table options is the same for both row-based and statement-based replication.

Having reached this point, if there are no table options, the replica simply executes all events. If there are any `--replicate-do-table` or `--replicate-wild-do-table` options, the event must match one of these if it is to be executed; otherwise, it is ignored. If there are any `--replicate-ignore-table` or `--replicate-wild-ignore-table` options, all events are executed except those that match any of these options.

Important

Table-level replication filters are only applied to tables that are explicitly mentioned and operated on in the query. They do not apply to tables that are implicitly updated by the query. For example, a `GRANT` statement, which updates the `mysql.user` system table but does not mention that table, is not affected by a filter that specifies `mysql.%` as the wildcard pattern.

The following steps describe this evaluation in more detail. The starting point is the end of the evaluation of the database-level options, as described in Section 17.2.5.1, “Evaluation of Database-Level Replication and Binary Logging Options”.

1. Are there any table replication options?
   - **Yes.** Continue to step 2.
   - **No.** Execute the update and exit.
2. Which logging format is used?
   - **STATEMENT.** Carry out the remaining steps for each statement that performs an update.
   - **ROW.** Carry out the remaining steps for each update of a table row.
3. Are there any `--replicate-do-table` options?
   - **Yes.** Does the table match any of them?
     - **Yes.** Execute the update and exit.
     - **No.** Continue to step 4.
   - **No.** Continue to step 4.
4. Are there any `--replicate-ignore-table` options?

- **Yes.** Does the table match any of them?
  - **Yes.** Ignore the update and exit.
  - **No.** Continue to step 5.
- **No.** Continue to step 5.

5. Are there any `--replicate-wild-do-table` options?
   - **Yes.** Does the table match any of them?
     - **Yes.** Execute the update and exit.
     - **No.** Continue to step 6.
   - **No.** Continue to step 6.
6. Are there any `--replicate-wild-ignore-table` options?
   - **Yes.** Does the table match any of them?
     - **Yes.** Ignore the update and exit.
     - **No.** Continue to step 7.
   - **No.** Continue to step 7.
7. Is there another table to be tested?
   - **Yes.** Go back to step 3.
   - **No.** Continue to step 8.
8. Are there any `--replicate-do-table` or `--replicate-wild-do-table` options?
   - **Yes.** Ignore the update and exit.
   - **No.** Execute the update and exit.

Note

Statement-based replication stops if a single SQL statement operates on both a table that is included by a `--replicate-do-table` or `--replicate-wild-do-table` option, and another table that is ignored by a `--replicate-ignore-table` or `--replicate-wild-ignore-table` option. The replica must either execute or ignore the complete statement (which forms a replication event), and it cannot logically do this. This also applies to row-based replication for DDL statements, because DDL statements are always logged as statements, without regard to the logging format in effect. The only type of statement that can update both an included and an ignored table and still be replicated successfully is a DML statement that has been logged with `binlog_format=ROW`.

17.2.5.3 Interactions Between Replication Filtering Options

If you use a combination of database-level and table-level replication filtering options, the replica first accepts or ignores events using the database options, then it evaluates all events permitted by those options according to the table options. This can sometimes lead to results that seem counterintuitive. It is also important to note that the results vary depending on whether the operation is logged using statement-based or row-based binary logging format. If you want to be sure that your replication filters

always operate in the same way independently of the binary logging format, which is particularly important if you are using mixed binary logging format, follow the guidance in this topic.

The effect of the replication filtering options differs between binary logging formats because of the way the database name is identified. With statement-based format, DML statements are handled based on the current database, as specified by the `USE` statement. With row-based format, DML statements are handled based on the database where the modified table exists. DDL statements are always filtered based on the current database, as specified by the `USE` statement, regardless of the binary logging format.

An operation that involves multiple tables can also be affected differently by replication filtering options depending on the binary logging format. Operations to watch out for include transactions involving multi-table `UPDATE` statements, triggers, cascading foreign keys, stored functions that update multiple tables, and DML statements that invoke stored functions that update one or more tables. If these operations update both filtered-in and filtered-out tables, the results can vary with the binary logging format.

If you need to guarantee that your replication filters operate consistently regardless of the binary logging format, particularly if you are using mixed binary logging format (`binlog_format=MIXED`), use only table-level replication filtering options, and do not use database-level replication filtering options. Also, do not use multi-table DML statements that update both filtered-in and filtered-out tables.

If you need to use a combination of database-level and table-level replication filters, and want these to operate as consistently as possible, choose one of the following strategies:

1. If you use row-based binary logging format (`binlog_format=ROW`), for DDL statements, rely on the `USE` statement to set the database and do not specify the database name. You can consider changing to row-based binary logging format for improved consistency with replication filtering. See Section 5.4.4.2, “Setting The Binary Log Format” for the conditions that apply to changing the binary logging format.

2. If you use statement-based or mixed binary logging format (`binlog_format=STATEMENT` or `MIXED`), for both DML and DDL statements, rely on the `USE` statement and do not use the database name. Also, do not use multi-table DML statements that update both filtered-in and filtered-out tables.

Example 17.7 A `--replicate-ignore-db` option and a `--replicate-do-table` option

On the replication source server, the following statements are issued:

```
USE db1;
CREATE TABLE t2 LIKE t1;
INSERT INTO db2.t3 VALUES (1);
```

The replica has the following replication filtering options set:

```
replicate-ignore-db = db1
replicate-do-table = db2.t3
```

The DDL statement `CREATE TABLE` creates the table in `db1`, as specified by the preceding `USE` statement. The replica filters out this statement according to its `--replicate-ignore-db = db1` option, because `db1` is the current database. This result is the same whatever the binary logging format is on the replication source server. However, the result of the DML `INSERT` statement is different depending on the binary logging format:

- If row-based binary logging format is in use on the source (`binlog_format=ROW`), the replica evaluates the `INSERT` operation using the database where the table exists, which is named as `db2`. The database-level option `--replicate-ignore-db = db1`, which is evaluated first, therefore does not apply. The table-level option `--replicate-do-table = db2.t3` does apply, so the replica applies the change to table `t3`.

- If statement-based binary logging format is in use on the source (`binlog_format=STATEMENT`), the replica evaluates the `INSERT` operation using the default database, which was set by the `USE` statement to `db1` and has not been changed. According to its database-level `--replicate-ignore-db = db1` option, it therefore ignores the operation and does not apply the change to table `t3`. The table-level option `--replicate-do-table = db2.t3` is not checked, because the statement already matched a database-level option and was ignored.

If the `--replicate-ignore-db = db1` option on the replica is necessary, and the use of statement-based (or mixed) binary logging format on the source is also necessary, the results can be made consistent by omitting the database name from the `INSERT` statement and relying on a `USE` statement instead, as follows:

```
USE db1;
CREATE TABLE t2 LIKE t1;
USE db2;
INSERT INTO t3 VALUES (1);
```

In this case, the replica always evaluates the `INSERT` statement based on the database `db2`. Whether the operation is logged in statement-based or row-based binary format, the results remain the same.

17.2.5.4 Replication Channel Based Filters

This section explains how to work with replication filters when multiple replication channels exist, for example in a multi-source replication topology. Before MySQL 8.0, all replication filters were global, so filters were applied to all replication channels. From MySQL 8.0, replication filters can be global or channel specific, enabling you to configure multi-source replicas with replication filters on specific replication channels. Channel specific replication filters are particularly useful in a multi-source replication topology when the same database or table is present on multiple sources, and the replica is only required to replicate it from one source.

For instructions to set up replication channels, see Section 17.1.5, “MySQL Multi-Source Replication”, and for more information on how they work, see Section 17.2.2, “Replication Channels”.

Important

Each channel on a multi-source replica must replicate from a different source. You cannot set up multiple replication channels from a single replica to a single source, even if you use replication filters to select different data to replicate on each channel. This is because the server IDs of replicas must be unique in a replication topology. The source distinguishes replicas only by their server IDs, not by the names of the replication channels, so it cannot recognize different replication channels from the same replica.

Important

On a MySQL server instance that is configured for Group Replication, channel specific replication filters can be used on replication channels that are not directly involved with Group Replication, such as where a group member also acts as a replica to a source that is outside the group. They cannot be used on the `group_replication_applier` or `group_replication_recovery` channels. Filtering on these channels would make the group unable to reach agreement on a consistent state.

Important

For a multi-source replica in a diamond topology (where the replica replicates from two or more sources, which in turn replicate from a common source), when GTID-based replication is in use, ensure that any replication filters or other channel configuration are identical on all channels on the multi-source replica. With GTID-based replication, filters are applied only to the transaction

> data, and GTIDs are not filtered out. This happens so that a replica's GTID set stays consistent with the source's, meaning GTID auto-positioning can be used without re-acquiring filtered out transactions each time. In the case where the downstream replica is multi-source and receives the same transaction from multiple sources in a diamond topology, the downstream replica now has multiple versions of the transaction, and the result depends on which channel applies the transaction first. The second channel to attempt it skips the transaction using GTID auto-skip, because the transaction's GTID was added to the `gtid_executed` set by the first channel. With identical filtering on the channels, there is no problem because all versions of the transaction contain the same data, so the results are the same. However, with different filtering on the channels, the database can become inconsistent and replication can hang.

Overview of Replication Filters and Channels

When multiple replication channels exist, for example in a multi-source replication topology, replication filters are applied as follows:

- Any global replication filter specified is added to the global replication filters of the filter type (`do_db`, `do_ignore_table`, and so on).
- Any channel specific replication filter adds the filter to the specified channel's replication filters for the specified filter type.
- Each replication channel copies global replication filters to its channel specific replication filters if no channel specific replication filter of this type is configured.
- Each channel uses its channel specific replication filters to filter the replication stream.

The syntax to create channel specific replication filters extends the existing SQL statements and command options. When a replication channel is not specified the global replication filter is configured to ensure backwards compatibility. The `CHANGE REPLICATION FILTER` statement supports the `FOR CHANNEL` clause to configure channel specific filters online. The `--replicate-*` command options to configure filters can specify a replication channel using the form `--replicate-`*`filter_type`*`=`*`channel_name`*`:`*`filter_details`*. Suppose channels `channel_1` and `channel_2` exist before the server starts; in this case, starting the replica with the command line options `--replicate-do-db=db1` `--replicate-do-db=channel_1:db2` `--replicate-do-db=db3` `--replicate-ignore-db=db4` `--replicate-ignore-db=channel_2:db5` `--replicate-wild-do-table=channel_1:db6.t1%` would result in:

- *Global replication filters*: `do_db=db1,db3`; `ignore_db=db4`
- *Channel specific filters on channel_1*: `do_db=db2`; `ignore_db=db4`; `wild-do-table=db6.t1%`
- *Channel specific filters on channel_2*: `do_db=db1,db3`; `ignore_db=db5`

These same rules could be applied at startup when included in the replica's `my.cnf` file, like this:

```
replicate-do-db=db1
replicate-do-db=channel_1:db2
replicate-ignore-db=db4
replicate-ignore-db=channel_2:db5
replicate-wild-do-table=db6.channel_1.t1%
```

To monitor the replication filters in such a setup use the `replication_applier_global_filters` and `replication_applier_filters` tables.

Configuring Channel Specific Replication Filters at Startup

The replication filter related command options can take an optional *`channel`* followed by a colon, followed by the filter specification. The first colon is interpreted as a separator, subsequent colons are

interpreted as literal colons. The following command options support channel specific replication filters using this format:

- `--replicate-do-db=channel:database_id`
- `--replicate-ignore-db=channel:database_id`
- `--replicate-do-table=channel:table_id`
- `--replicate-ignore-table=channel:table_id`
- `--replicate-rewrite-db=channel:db1-db2`
- `--replicate-wild-do-table=channel:table pattern`
- `--replicate-wild-ignore-table=channel:table pattern`

All of the options just listed can be used in the replica's `my.cnf` file, as with most other MySQL server startup options, by omitting the two leading dashes. See Overview of Replication Filters and Channels, for a brief example, as well as Section 4.2.2.2, “Using Option Files”.

If you use a colon but do not specify a *`channel`* for the filter option, for example `--replicate-do-db=:database_id`, the option configures the replication filter for the default replication channel. The default replication channel is the replication channel which always exists once replication has been started, and differs from multi-source replication channels which you create manually. When neither the colon nor a *`channel`* is specified the option configures the global replication filters, for example `--replicate-do-db=database_id` configures the global `--replicate-do-db` filter.

If you configure multiple `rewrite-db=from_name->to_name` options with the same *`from_name`* database, all filters are added together (put into the `rewrite_do` list) and the first one takes effect.

The *`pattern`* used for the `--replicate-wild-*-table` options can include any characters allowed in identifiers as well as the wildcards `%` and `_`. These work the same way as when used with the `LIKE` operator; for example, `tbl%` matches any table name beginning with `tbl`, and `tbl_` matches any table name matching `tbl` plus one additional character.

Changing Channel Specific Replication Filters Online

In addition to the `--replicate-*` options, replication filters can be configured using the `CHANGE REPLICATION FILTER` statement. This removes the need to restart the server, but the replication SQL thread must be stopped while making the change. To make this statement apply the filter to a specific channel, use the `FOR CHANNEL channel` clause. For example:

```
CHANGE REPLICATION FILTER REPLICATE_DO_DB=(db1) FOR CHANNEL channel_1;
```

When a `FOR CHANNEL` clause is provided, the statement acts on the specified channel's replication filters. If multiple types of filters (`do_db`, `do_ignore_table`, `wild_do_table`, and so on) are specified, only the specified filter types are replaced by the statement. In a replication topology with multiple channels, for example on a multi-source replica, when no `FOR CHANNEL` clause is provided, the statement acts on the global replication filters and all channels' replication filters, using a similar logic as the `FOR CHANNEL` case. For more information see Section 13.4.2.2, “CHANGE REPLICATION FILTER Statement”.

Removing Channel Specific Replication Filters

When channel specific replication filters have been configured, you can remove the filter by issuing an empty filter type statement. For example to remove all `REPLICATE_REWRITE_DB` filters from a replication channel named `channel_1` issue:

```
CHANGE REPLICATION FILTER REPLICATE_REWRITE_DB=() FOR CHANNEL channel_1;
```

Any REPLICATE_REWRITE_DB filters previously configured, using either command options or CHANGE REPLICATION FILTER, are removed.

The RESET REPLICA ALL statement removes channel specific replication filters that were set on channels deleted by the statement. When the deleted channel or channels are recreated, any global replication filters specified for the replica are copied to them, and no channel specific replication filters are applied.

17.3 Replication Security

To protect against unauthorized access to data that is stored on and transferred between replication source servers and replicas, set up all the servers involved using the security measures that you would choose for any MySQL instance in your installation, as described in Chapter 6, *Security*. In addition, for servers in a replication topology, consider implementing the following security measures:

- Set up sources and replicas to use encrypted connections to transfer the binary log, which protects this data in motion. Encryption for these connections must be activated using a CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO statement, in addition to setting up the servers to support encrypted network connections. See Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”.
- Encrypt the binary log files and relay log files on sources and replicas, which protects this data at rest, and also any data in use in the binary log cache. Binary log encryption is activated using the binlog_encryption system variable. See Section 17.3.2, “Encrypting Binary Log Files and Relay Log Files”.
- Apply privilege checks to replication appliers, which help to secure replication channels against the unauthorized or accidental use of privileged or unwanted operations. Privilege checks are implemented by setting up a PRIVILEGE_CHECKS_USER account, which MySQL uses to verify that you have authorized each specific transaction for that channel. See Section 17.3.3, “Replication Privilege Checks”.

For Group Replication, binary log encryption and privilege checks can be used as a security measure on replication group members. You should also consider encrypting the connections between group members, comprising group communication connections and distributed recovery connections, and applying IP address allowlisting to exclude untrusted hosts. For information on these security measures specific to Group Replication, see Section 18.6, “Group Replication Security”.

17.3.1 Setting Up Replication to Use Encrypted Connections

To use an encrypted connection for the transfer of the binary log required during replication, both the source and the replica servers must support encrypted network connections. If either server does not support encrypted connections (because it has not been compiled or configured for them), replication through an encrypted connection is not possible.

Setting up encrypted connections for replication is similar to doing so for client/server connections. You must obtain (or create) a suitable security certificate that you can use on the source, and a similar certificate (from the same certificate authority) on each replica. You must also obtain suitable key files.

For more information on setting up a server and client for encrypted connections, see Section 6.3.1, “Configuring MySQL to Use Encrypted Connections”.

To enable encrypted connections on the source, you must create or obtain suitable certificate and key files, and then add the following configuration parameters to the [mysqld] section of the source my.cnf file, changing the file names as necessary:

```
[mysqld]
ssl_ca=cacert.pem
ssl_cert=server-cert.pem
```

```
ssl_key=server-key.pem
```

The paths to the files may be relative or absolute; we recommend that you always use complete paths for this purpose.

The configuration parameters are as follows:

- `ssl_ca`: The path name of the Certificate Authority (CA) certificate file. (`ssl_capath` is similar but specifies the path name of a directory of CA certificate files.)
- `ssl_cert`: The path name of the server public key certificate file. This certificate can be sent to the client and authenticated against the CA certificate that it has.
- `ssl_key`: The path name of the server private key file.

To enable encrypted connections on the replica, use the `CHANGE REPLICATION SOURCE TO` statement (MySQL 8.0.23 and later) or `CHANGE MASTER TO` statement (prior to MySQL 8.0.23).

- To name the replica's certificate and SSL private key files using `CHANGE REPLICATION SOURCE TO` (`CHANGE MASTER TO`), add the appropriate `SOURCE_SSL_`*`xxx`* (`MASTER_SSL_`*`xxx`*) options, like this:

  ```
      -> SOURCE_SSL_CA = 'ca_file_name',
      -> SOURCE_SSL_CAPATH = 'ca_directory_name',
      -> SOURCE_SSL_CERT = 'cert_file_name',
      -> SOURCE_SSL_KEY = 'key_file_name',
  ```

  These options correspond to the `--ssl-`*`xxx`* options with the same names, as described in Command Options for Encrypted Connections. For these options to take effect, `SOURCE_SSL=1` must also be set. For a replication connection, specifying a value for either of `SOURCE_SSL_CA` or `SOURCE_SSL_CAPATH` corresponds to setting `--ssl-mode=VERIFY_CA`. The connection attempt succeeds only if a valid matching Certificate Authority (CA) certificate is found using the specified information.

- To activate host name identity verification, add the `SOURCE_SSL_VERIFY_SERVER_CERT` option, like this:

  ```
      -> SOURCE_SSL_VERIFY_SERVER_CERT=1,
  ```

  This option corresponds to the `--ssl-verify-server-cert` option, which is deprecated in MySQL 5.7 and removed in MySQL 8.0. For a replication connection, specifying `MASTER_SSL_VERIFY_SERVER_CERT=1` corresponds to setting `--ssl-mode=VERIFY_IDENTITY`, as described in Command Options for Encrypted Connections. For this option to take effect, `SOURCE_SSL=1` must also be set. Host name identity verification does not work with self-signed certificates.

- To activate certificate revocation list (CRL) checks, add the `SOURCE_SSL_CRL` or `SOURCE_SSL_CRLPATH` option, as shown here:

  ```
      -> SOURCE_SSL_CRL = 'crl_file_name',
      -> SOURCE_SSL_CRLPATH = 'crl_directory_name',
  ```

  These options correspond to the `--ssl-`*`xxx`* options with the same names, as described in Command Options for Encrypted Connections. If they are not specified, no CRL checking takes place.

- To specify lists of ciphers, ciphersuites, and encryption protocols permitted by the replica for the replication connection, use the `SOURCE_SSL_CIPHER`, `SOURCE_TLS_VERSION`, and `SOURCE_TLS_CIPHERSUITES` options, like this:

  ```
      -> SOURCE_SSL_CIPHER = 'cipher_list',
      -> SOURCE_TLS_VERSION = 'protocol_list',
      -> SOURCE_TLS_CIPHERSUITES = 'ciphersuite_list',
  ```

- The SOURCE_SSL_CIPHER option specifies a colon-separated list of one or more ciphers permitted by the replica for the replication connection.
- The SOURCE_TLS_VERSION option specifies a comma-separated list of the TLS encryption protocols permitted by the replica for the replication connection, in a format like that for the tls_version server system variable. The connection procedure negotiates the use of the highest TLS version that both the source and the replica permit. To be able to connect, the replica must have at least one TLS version in common with the source.
- The SOURCE_TLS_CIPHERSUITES option (available beginning with MySQL 8.0.19) specifies a colon-separated list of one or more ciphersuites that are permitted by the replica for the replication connection if TLSv1.3 is used for the connection. If this option is set to NULL when TLSv1.3 is used (which is the default if you do not set the option), the ciphersuites that are enabled by default are allowed. If you set the option to an empty string, no cipher suites are allowed, and TLSv1.3 is therefore not used.

The protocols, ciphers, and ciphersuites that you can specify in these lists depend on the SSL library used to compile MySQL. For information about the formats, the permitted values, and the defaults if you do not specify the options, see Section 6.3.2, "Encrypted Connection TLS Protocols and Ciphers".

Note

In MySQL 8.0.16 through 8.0.18, MySQL supports TLSv1.3, but the SOURCE_TLS_CIPHERSUITES option is not available. In these releases, if TLSv1.3 is used for connections between a source and replica, the source must permit the use of at least one TLSv1.3 ciphersuite that is enabled by default. From MySQL 8.0.19, you can use the option to specify any selection of ciphersuites, including only non-default ciphersuites if you want.

- After the source information has been updated, start the replication process on the replica, like this:

```
mysql> START SLAVE;
```

Beginning with MySQL 8.0.22, START REPLICA is preferred, as shown here:

```
mysql> START REPLICA;
```

You can use the SHOW REPLICA STATUS (prior to MySQL 8.0.22, SHOW SLAVE STATUS) statement to confirm that an encrypted connection was established successfully.

- Requiring encrypted connections on the replica does not ensure that the source requires encrypted connections from replicas. If you want to ensure that the source only accepts replicas that connect using encrypted connections, create a replication user account on the source using the REQUIRE SSL option, then grant that user the REPLICATION SLAVE privilege. For example:

```
mysql> CREATE USER 'repl'@'%.example.com' IDENTIFIED BY 'password'
    -> REQUIRE SSL;
mysql> GRANT REPLICATION SLAVE ON *.*
    -> TO 'repl'@'%.example.com';
```

If you have an existing replication user account on the source, you can add REQUIRE SSL to it with this statement:

```
mysql> ALTER USER 'repl'@'%.example.com' REQUIRE SSL;
```

17.3.2 Encrypting Binary Log Files and Relay Log Files

From MySQL 8.0.14, binary log files and relay log files can be encrypted, helping to protect these files and the potentially sensitive data contained in them from being misused by outside attackers, and also from unauthorized viewing by users of the operating system where they are stored. The encryption

algorithm used for the files, the AES (Advanced Encryption Standard) cipher algorithm, is built in to MySQL Server and cannot be configured.

You enable this encryption on a MySQL server by setting the `binlog_encryption` system variable to `ON`. `OFF` is the default. The system variable sets encryption on for binary log files and relay log files. Binary logging does not need to be enabled on the server to enable encryption, so you can encrypt the relay log files on a replica that has no binary log. To use encryption, a keyring component or plugin must be installed and configured to supply MySQL Server's keyring service. For instructions to do this, see Section 6.4.4, “The MySQL Keyring”. Any supported keyring component or plugin can be used to store binary log encryption keys.

When you first start the server with encryption enabled, a new binary log encryption key is generated before the binary log and relay logs are initialized. This key is used to encrypt a file password for each binary log file (if the server has binary logging enabled) and relay log file (if the server has replication channels), and further keys generated from the file passwords are used to encrypt the data in the files. The binary log encryption key that is currently in use on the server is called the binary log master key. The two tier encryption key architecture means that the binary log master key can be rotated (replaced by a new master key) as required, and only the file password for each file needs to be re-encrypted with the new master key, not the whole file. Relay log files are encrypted for all channels, including new channels that are created after encryption is activated. The binary log index file and relay log index file are never encrypted.

If you activate encryption while the server is running, a new binary log encryption key is generated at that time. The exception is if encryption was active previously on the server and was then disabled, in which case the binary log encryption key that was in use before is used again. The binary log file and relay log files are rotated immediately, and file passwords for the new files and all subsequent binary log files and relay log files are encrypted using this binary log encryption key. Existing binary log files and relay log files still present on the server are not encrypted, but you can purge them if they are no longer needed.

If you deactivate encryption by changing the `binlog_encryption` system variable to `OFF`, the binary log file and relay log files are rotated immediately and all subsequent logging is unencrypted. Previously encrypted files are not automatically decrypted, but the server is still able to read them. The `BINLOG_ENCRYPTION_ADMIN` privilege is required to activate or deactivate encryption while the server is running.

Encrypted and unencrypted binary log files can be distinguished using the magic number at the start of the file header for encrypted log files (`0xFD62696E`), which differs from that used for unencrypted log files (`0xFE62696E`). The `SHOW BINARY LOGS` statement shows whether each binary log file is encrypted or unencrypted.

When binary log files have been encrypted, `mysqlbinlog` cannot read them directly, but can read them from the server using the `--read-from-remote-server` option. From MySQL 8.0.14, `mysqlbinlog` returns a suitable error if you attempt to read an encrypted binary log file directly, but older versions of `mysqlbinlog` do not recognize the file as a binary log file at all. If you back up encrypted binary log files using `mysqlbinlog`, note that the copies of the files that are generated using `mysqlbinlog` are stored in an unencrypted format.

Binary log encryption can be combined with binary log transaction compression (available as of MySQL 8.0.20). For more information on binary log transaction compression, see Section 5.4.4.5, “Binary Log Transaction Compression”.

17.3.2.1 Scope of Binary Log Encryption

When binary log encryption is active for a MySQL server instance, the encryption coverage is as follows:

- Data at rest that is written to the binary log files and relay log files is encrypted from the point in time where encryption is started, using the two tier encryption architecture described above. Existing

binary log files and relay log files that were present on the server when you started encryption are not encrypted. You can purge these files when they are no longer needed.

- Data in motion in the replication event stream, which is sent to MySQL clients including `mysqlbinlog`, is decrypted for transmission, and should therefore be protected in transit by the use of connection encryption (see Section 6.3, “Using Encrypted Connections” and Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”).
- Data in use that is held in the binary log transaction and statement caches during a transaction is in unencrypted format in the memory buffer that stores the cache. The data is written to a temporary file on disk if it exceeds the space available in the memory buffer. From MySQL 8.0.17, when binary log encryption is active on the server, temporary files used to hold the binary log cache are encrypted using AES-CTR (AES Counter mode) for stream encryption. Because the temporary files are volatile and tied to a single process, they are encrypted using single-tier encryption, using a randomly generated file password and initialization vector that exist only in memory and are never stored on disk or in the keyring. After each transaction is committed, the binary log cache is reset: the memory buffer is cleared, any temporary file used to hold the binary log cache is truncated, and a new file password and initialization vector are randomly generated for use with the next transaction. This reset also takes place when the server is restarted after a normal shutdown or an unexpected halt.

Note

If you use `LOAD DATA` when `binlog_format=STATEMENT` is set, which is not recommended as the statement is considered unsafe for statement-based replication, a temporary file containing the data is created on the replica where the changes are applied. These temporary files are not encrypted when binary log encryption is active on the server. Use row-based or mixed binary logging format instead, which do not create the temporary files.

17.3.2.2 Binary Log Encryption Keys

The binary log encryption keys used to encrypt the file passwords for the log files are 256-bit keys that are generated specifically for each MySQL server instance using MySQL Server's keyring service (see Section 6.4.4, “The MySQL Keyring”). The keyring service handles the creation, retrieval, and deletion of the binary log encryption keys. A server instance only creates and removes keys generated for itself, but it can read keys generated for other instances if they are stored in the keyring, as in the case of a server instance that has been cloned by file copying.

Important

The binary log encryption keys for a MySQL server instance must be included in your backup and recovery procedures, because if the keys required to decrypt the file passwords for current and retained binary log files or relay log files are lost, it might not be possible to start the server.

The format of binary log encryption keys in the keyring is as follows:

```
MySQLReplicationKey_{UUID}_{SEQ_NO}
```

For example:

```
MySQLReplicationKey_00508583-b5ce-11e8-a6a5-0010e0734796_1
```

`{UUID}` is the true UUID generated by the MySQL server (the value of the `server_uuid` system variable). `{SEQ_NO}` is the sequence number for the binary log encryption key, which is incremented by 1 for each new key that is generated on the server.

The binary log encryption key that is currently in use on the server is called the binary log master key. The sequence number for the current binary log master key is stored in the keyring. The binary log master key is used to encrypt each new log file's file password, which is a randomly generated 32-byte file password specific to the log file that is used to encrypt the file data. The file password is encrypted using AES-CBC (AES Cipher Block Chaining mode) with the 256-bit binary log encryption key and

a random initialization vector (IV), and is stored in the log file's file header. The file data is encrypted using AES-CTR (AES Counter mode) with a 256-bit key generated from the file password and a nonce also generated from the file password. It is technically possible to decrypt an encrypted file offline, if the binary log encryption key used to encrypt the file password is known, by using tools available in the OpenSSL cryptography toolkit.

If you use file copying to clone a MySQL server instance that has encryption active so its binary log files and relay log files are encrypted, ensure that the keyring is also copied, so that the clone server can read the binary log encryption keys from the source server. When encryption is activated on the clone server (either at startup or subsequently), the clone server recognizes that the binary log encryption keys used with the copied files include the generated UUID of the source server. It automatically generates a new binary log encryption key using its own generated UUID, and uses this to encrypt the file passwords for subsequent binary log files and relay log files. The copied files continue to be read using the source server's keys.

17.3.2.3 Binary Log Master Key Rotation

When binary log encryption is enabled, you can rotate the binary log master key at any time while the server is running by issuing `ALTER INSTANCE ROTATE BINLOG MASTER KEY`. When the binary log master key is rotated manually using this statement, the passwords for the new and subsequent files are encrypted using the new binary log master key, and also the file passwords for existing encrypted binary log files and relay log files are re-encrypted using the new binary log master key, so the encryption is renewed completely. You can rotate the binary log master key on a regular basis to comply with your organization's security policy, and also if you suspect that the current or any of the previous binary log master keys might have been compromised.

When you rotate the binary log master key manually, MySQL Server takes the following actions in sequence:

1. A new binary log encryption key is generated with the next available sequence number, stored on the keyring, and used as the new binary log master key.
2. The binary log and relay log files are rotated on all channels.
3. The new binary log master key is used to encrypt the file passwords for the new binary log and relay log files, and subsequent files until the key is changed again.
4. The file passwords for existing encrypted binary log files and relay log files on the server are re-encrypted in turn using the new binary log master key, starting with the most recent files. Any unencrypted files are skipped.
5. Binary log encryption keys that are no longer in use for any files after the re-encryption process are removed from the keyring.

The `BINLOG_ENCRYPTION_ADMIN` privilege is required to issue `ALTER INSTANCE ROTATE BINLOG MASTER KEY`, and the statement cannot be used if the `binlog_encryption` system variable is set to `OFF`.

As the final step of the binary log master key rotation process, all binary log encryption keys that no longer apply to any retained binary log files or relay log files are cleaned up from the keyring. If a retained binary log file or relay log file cannot be initialized for re-encryption, the relevant binary log encryption keys are not deleted in case the files can be recovered in the future. For example, this might be the case if a file listed in a binary log index file is currently unreadable, or if a channel fails to initialize. If the server UUID changes, for example because a backup created using MySQL Enterprise Backup is used to set up a new replica, issuing `ALTER INSTANCE ROTATE BINLOG MASTER KEY` on the new server does not delete any earlier binary log encryption keys that include the original server UUID.

If any of the first four steps of the binary log master key rotation process cannot be completed correctly, an error message is issued explaining the situation and the consequences for the encryption status of the binary log files and relay log files. Files that were previously encrypted are always left in an

encrypted state, but their file passwords might still be encrypted using an old binary log master key. If you see these errors, first retry the process by issuing `ALTER INSTANCE ROTATE BINLOG MASTER KEY` again. Then investigate the status of individual files to see what is blocking the process, especially if you suspect that the current or any of the previous binary log master keys might have been compromised.

If the final step of the binary log master key rotation process cannot be completed correctly, a warning message is issued explaining the situation. The warning message identifies whether the process could not clean up the auxiliary keys in the keyring for rotating the binary log master key, or could not clean up unused binary log encryption keys. You can choose to ignore the message as the keys are auxiliary keys or no longer in use, or you can issue `ALTER INSTANCE ROTATE BINLOG MASTER KEY` again to retry the process.

If the server stops and is restarted with binary log encryption still set to `ON` during the binary log master key rotation process, new binary log files and relay log files after the restart are encrypted using the new binary log master key. However, the re-encryption of existing files is not continued, so files that did not get re-encrypted before the server stopped are left encrypted using the previous binary log master key. To complete re-encryption and clean up unused binary log encryption keys, issue `ALTER INSTANCE ROTATE BINLOG MASTER KEY` again after the restart.

`ALTER INSTANCE ROTATE BINLOG MASTER KEY` actions are not written to the binary log and are not executed on replicas. Binary log master key rotation can therefore be carried out in replication environments including a mix of MySQL versions. To schedule regular rotation of the binary log master key on all applicable source and replica servers, you can enable the MySQL Event Scheduler on each server and issue the `ALTER INSTANCE ROTATE BINLOG MASTER KEY` statement using a `CREATE EVENT` statement. If you rotate the binary log master key because you suspect that the current or any of the previous binary log master keys might have been compromised, issue the statement on every applicable source and replica server. Issuing the statement on individual servers ensures that you can verify immediate compliance, even in the case of replicas that are lagging, belong to multiple replication topologies, or are not currently active in the replication topology but have binary log and relay log files.

The `binlog_rotate_encryption_master_key_at_startup` system variable controls whether the binary log master key is automatically rotated when the server is restarted. If this system variable is set to `ON`, a new binary log encryption key is generated and used as the new binary log master key whenever the server is restarted. If it is set to `OFF`, which is the default, the existing binary log master key is used again after the restart. When the binary log master key is rotated at startup, the file passwords for the new binary log and relay log files are encrypted using the new key. The file passwords for the existing encrypted binary log files and relay log files are not re-encrypted, so they remain encrypted using the old key, which remains available on the keyring.

17.3.3 Replication Privilege Checks

By default, MySQL replication (including Group Replication) does not carry out privilege checks when transactions that were already accepted by another server are applied on a replica or group member. From MySQL 8.0.18, you can create a user account with the appropriate privileges to apply the transactions that are normally replicated on a channel, and specify this as the `PRIVILEGE_CHECKS_USER` account for the replication applier, using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). MySQL then checks each transaction against the user account's privileges to verify that you have authorized the operation for that channel. The account can also be safely used by an administrator to apply or reapply transactions from `mysqlbinlog` output, for example to recover from a replication error on the channel.

The use of a `PRIVILEGE_CHECKS_USER` account helps secure a replication channel against the unauthorized or accidental use of privileged or unwanted operations. The `PRIVILEGE_CHECKS_USER` account provides an additional layer of security in situations such as these:

- You are replicating between a server instance on your organization's network, and a server instance on another network, such as an instance supplied by a cloud service provider.

- You want to have multiple on-premise or off-site deployments administered as separate units, without giving one administrator account privileges on all the deployments.
- You want to have an administrator account that enables an administrator to perform only operations that are directly relevant to the replication channel and the databases it replicates, rather than having wide privileges on the server instance.

You can increase the security of a replication channel where privilege checks are applied by adding one or both of these options to the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement when you specify the `PRIVILEGE_CHECKS_USER` account for the channel:

- The `REQUIRE_ROW_FORMAT` option (available from MySQL 8.0.19) makes the replication channel accept only row-based replication events. When `REQUIRE_ROW_FORMAT` is set, you must use row-based binary logging (`binlog_format=ROW`) on the source server. In MySQL 8.0.18, `REQUIRE_ROW_FORMAT` is not available, but the use of row-based binary logging for secured replication channels is still strongly recommended. With statement-based binary logging, some administrator-level privileges might be required for the `PRIVILEGE_CHECKS_USER` account to execute transactions successfully.
- The `REQUIRE_TABLE_PRIMARY_KEY_CHECK` option (available from MySQL 8.0.20) makes the replication channel use its own policy for primary key checks. Setting `ON` means that primary keys are always required, and setting `OFF` means that primary keys are never required. The default setting, `STREAM`, sets the session value of the `sql_require_primary_key` system variable using the value that is replicated from the source for each transaction. When `PRIVILEGE_CHECKS_USER` is set, setting `REQUIRE_TABLE_PRIMARY_KEY_CHECK` to either `ON` or `OFF` means that the user account does not need session administration level privileges to set restricted session variables, which are required to change the value of `sql_require_primary_key`. It also normalizes the behavior across replication channels for different sources.

You grant the `REPLICATION_APPLIER` privilege to enable a user account to appear as the `PRIVILEGE_CHECKS_USER` for a replication applier thread, and to execute the internal-use `BINLOG` statements used by mysqlbinlog. The user name and host name for the `PRIVILEGE_CHECKS_USER` account must follow the syntax described in Section 6.2.4, “Specifying Account Names”, and the user must not be an anonymous user (with a blank user name) or the `CURRENT_USER`. To create a new account, use `CREATE USER`. To grant this account the `REPLICATION_APPLIER` privilege, use the `GRANT` statement. For example, to create a user account `priv_repl`, which can be used manually by an administrator from any host in the `example.com` domain, and requires an encrypted connection, issue the following statements:

```
mysql> SET sql_log_bin = 0;
mysql> CREATE USER 'priv_repl'@'%.example.com' IDENTIFIED BY 'password' REQUIRE SSL;
mysql> GRANT REPLICATION_APPLIER ON *.* TO 'priv_repl'@'%.example.com';
mysql> SET sql_log_bin = 1;
```

The `SET sql_log_bin` statements are used so that the account management statements are not added to the binary log and sent to the replication channels (see Section 13.4.1.3, “SET sql_log_bin Statement”).

Important

The `caching_sha2_password` authentication plugin is the default for new users created from MySQL 8.0 (for details, see Section 6.4.1.2, “Caching SHA-2 Pluggable Authentication”). To connect to a server using a user account that authenticates with this plugin, you must either set up an encrypted connection as described in Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”, or enable the unencrypted connection to support password exchange using an RSA key pair.

After setting up the user account, use the `GRANT` statement to grant additional privileges to enable the user account to make the database changes that you expect the applier thread to carry out, such as updating specific tables held on the server. These same privileges enable an administrator to use the

account if they need to execute any of those transactions manually on the replication channel. If an unexpected operation is attempted for which you did not grant the appropriate privileges, the operation is disallowed and the replication applier thread stops with an error. Section 17.3.3.1, “Privileges For The Replication PRIVILEGE_CHECKS_USER Account” explains what additional privileges the account needs. For example, to grant the `priv_repl` user account the `INSERT` privilege to add rows to the `cust` table in `db1`, issue the following statement:

```
mysql> GRANT INSERT ON db1.cust TO 'priv_repl'@'%.example.com';
```

You assign the `PRIVILEGE_CHECKS_USER` account for a replication channel using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). If replication is running, issue `STOP REPLICA` (or before MySQL 8.0.22, `STOP SLAVE`) before the `CHANGE MASTER TO` statement, and `START REPLICA` after it. The use of row-based binary logging is strongly recommended when `PRIVILEGE_CHECKS_USER` is set, and from MySQL 8.0.19 you can use the statement to set `REQUIRE_ROW_FORMAT` to enforce this.

When you restart the replication channel, checks on dynamic privileges are applied from that point on. However, static global privileges are not active in the applier's context until you reload the grant tables, because these privileges are not changed for a connected client. To activate static privileges, perform a flush-privileges operation. This can be done by issuing a `FLUSH PRIVILEGES` statement or by executing a `mysqladmin flush-privileges` or `mysqladmin reload` command.

For example, to start privilege checks on the channel `channel_1` on a running replica in MySQL 8.0.23 and later, issue the following statements:

```
mysql> STOP REPLICA FOR CHANNEL 'channel_1';
mysql> CHANGE REPLICATION SOURCE TO
     >    PRIVILEGE_CHECKS_USER = 'priv_repl'@'%.example.com',
     >    REQUIRE_ROW_FORMAT = 1 FOR CHANNEL 'channel_1';
mysql> FLUSH PRIVILEGES;
mysql> START REPLICA FOR CHANNEL 'channel_1';
```

Prior to MySQL 8.0.23, you can use the statements shown here:

```
mysql> STOP SLAVE FOR CHANNEL 'channel_1';
mysql> CHANGE MASTER TO
     >    PRIVILEGE_CHECKS_USER = 'priv_repl'@'%.example.com',
     >    REQUIRE_ROW_FORMAT = 1 FOR CHANNEL 'channel_1';
mysql> FLUSH PRIVILEGES;
mysql> START SLAVE FOR CHANNEL 'channel_1';
```

If you do not specify a channel and no other channels exist, the statement is applied to the default channel. The user name and host name for the `PRIVILEGE_CHECKS_USER` account for a channel are shown in the Performance Schema `replication_applier_configuration` table, where they are properly escaped so they can be copied directly into SQL statements to execute individual transactions.

In MySQL 8.0.31 and later, if you are using the `Rewriter` plugin, you should grant the `PRIVILEGE_CHECKS_USER` user account the `SKIP_QUERY_REWRITE` privilege. This prevents statements issued by this user from being rewritten. See Section 5.6.4, “The Rewriter Query Rewrite Plugin”, for more information.

When `REQUIRE_ROW_FORMAT` is set for a replication channel, the replication applier does not create or drop temporary tables, and so does not set the `pseudo_thread_id` session system variable. It does not execute `LOAD DATA INFILE` instructions, and so does not attempt file operations to access or delete the temporary files associated with data loads (logged as a `Format_description_log_event`). It does not execute `INTVAR`, `RAND`, and `USER_VAR` events, which are used to reproduce the client's connection state for statement-based replication. (An exception is `USER_VAR` events that are associated with DDL queries, which are executed.) It does not execute any statements that are logged within DML transactions. If the replication applier detects any of these types of event while attempting to queue or apply a transaction, the event is not applied, and replication stops with an error.

You can set REQUIRE_ROW_FORMAT for a replication channel whether or not you set a PRIVILEGE_CHECKS_USER account. The restrictions implemented when you set this option increase the security of the replication channel even without privilege checks. You can also specify the --require-row-format option when you use mysqlbinlog, to enforce row-based replication events in mysqlbinlog output.

Security Context. By default, when a replication applier thread is started with a user account specified as the PRIVILEGE_CHECKS_USER, the security context is created using default roles, or with all roles if activate_all_roles_on_login is set to ON.

You can use roles to supply a general privilege set to accounts that are used as PRIVILEGE_CHECKS_USER accounts, as in the following example. Here, instead of granting the INSERT privilege for the db1.cust table directly to a user account as in the earlier example, this privilege is granted to the role priv_repl_role along with the REPLICATION_APPLIER privilege. The role is then used to grant the privilege set to two user accounts, both of which can now be used as PRIVILEGE_CHECKS_USER accounts:

```
mysql> SET sql_log_bin = 0;
mysql> CREATE USER 'priv_repa'@'%.example.com'
                  IDENTIFIED BY 'password'
                  REQUIRE SSL;
mysql> CREATE USER 'priv_repb'@'%.example.com'
                  IDENTIFIED BY 'password'
                  REQUIRE SSL;
mysql> CREATE ROLE 'priv_repl_role';
mysql> GRANT REPLICATION_APPLIER TO 'priv_repl_role';
mysql> GRANT INSERT ON db1.cust TO 'priv_repl_role';
mysql> GRANT 'priv_repl_role' TO
                  'priv_repa'@'%.example.com',
                  'priv_repb'@'%.example.com';
mysql> SET DEFAULT ROLE 'priv_repl_role' TO
                  'priv_repa'@'%.example.com',
                  'priv_repb'@'%.example.com';
mysql> SET sql_log_bin = 1;
```

Be aware that when the replication applier thread creates the security context, it checks the privileges for the PRIVILEGE_CHECKS_USER account, but does not carry out password validation, and does not carry out checks relating to account management, such as checking whether the account is locked. The security context that is created remains unchanged for the lifetime of the replication applier thread.

Limitation. In MySQL 8.0.18 only, if the replica mysqld is restarted immediately after issuing a RESET REPLICA statement (due to an unexpected server exit or deliberate restart), the PRIVILEGE_CHECKS_USER account setting, which is held in the mysql.slave_relay_log_info table, is lost and must be respecified. When you use privilege checks in that release, always verify that they are in place after a restart, and respecify them if required. From MySQL 8.0.19, the PRIVILEGE_CHECKS_USER account setting is preserved in this situation, so it is retrieved from the table and reapplied to the channel.

17.3.3.1 Privileges For The Replication PRIVILEGE_CHECKS_USER Account

The user account that is specified using the CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO statement as the PRIVILEGE_CHECKS_USER account for a replication channel must have the REPLICATION_APPLIER privilege, otherwise the replication applier thread does not start. As explained in Section 17.3.3, “Replication Privilege Checks”, the account requires further privileges that are sufficient to apply all the expected transactions expected on the replication channel. These privileges are checked only when relevant transactions are executed.

The use of row-based binary logging (binlog_format=ROW) is strongly recommended for replication channels that are secured using a PRIVILEGE_CHECKS_USER account. With statement-based binary logging, some administrator-level privileges might be required for the PRIVILEGE_CHECKS_USER account to execute transactions successfully. From MySQL 8.0.19, the REQUIRE_ROW_FORMAT setting can be applied to secured channels, which restricts the channel from executing events that would require these privileges.

The REPLICATION_APPLIER privilege explicitly or implicitly allows the PRIVILEGE_CHECKS_USER account to carry out the following operations that a replication thread needs to perform:

- Setting the value of the system variables gtid_next, original_commit_timestamp, original_server_version, immediate_server_version, and pseudo_replica_mode or pseudo_slave_mode, to apply appropriate metadata and behaviors when executing transactions.
- Executing internal-use BINLOG statements to apply mysqlbinlog output, provided that the account also has permission for the tables and operations in those statements.
- Updating the system tables mysql.gtid_executed, mysql.slave_relay_log_info, mysql.slave_worker_info, and mysql.slave_master_info, to update replication metadata. (If events access these tables explicitly for other purposes, you must grant the appropriate privileges on the tables.)
- Applying a binary log Table_map_log_event, which provides table metadata but does not make any database changes.

If the REQUIRE_TABLE_PRIMARY_KEY_CHECK option of the CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO statement is set to the default of STREAM, the PRIVILEGE_CHECKS_USER account needs privileges sufficient to set restricted session variables, so that it can change the value of the sql_require_primary_key system variable for the duration of a session to match the setting replicated from the source. The SESSION_VARIABLES_ADMIN privilege gives the account this capability. This privilege also allows the account to apply mysqlbinlog output that was created using the --disable-log-bin option. If you set REQUIRE_TABLE_PRIMARY_KEY_CHECK to either ON or OFF, the replica always uses that value for the sql_require_primary_key system variable in replication operations, and so does not need these session administration level privileges.

If table encryption is in use, the table_encryption_privilege_check system variable is set to ON, and the encryption setting for the tablespace involved in any event differs from the applying server's default encryption setting (specified by the default_table_encryption system variable), the PRIVILEGE_CHECKS_USER account needs the TABLE_ENCRYPTION_ADMIN privilege in order to override the default encryption setting. It is strongly recommended that you do not grant this privilege. Instead, ensure that the default encryption setting on a replica matches the encryption status of the tablespaces that it replicates, and that replication group members have the same default encryption setting, so that the privilege is not needed.

In order to execute specific replicated transactions from the relay log, or transactions from mysqlbinlog output as required, the PRIVILEGE_CHECKS_USER account must have the following privileges:

- For a row insertion logged in row format (which are logged as a Write_rows_log_event), the INSERT privilege on the relevant table.
- For a row update logged in row format (which are logged as an Update_rows_log_event), the UPDATE privilege on the relevant table.
- For a row deletion logged in row format (which are logged as a Delete_rows_log_event), the DELETE privilege on the relevant table.

If statement-based binary logging is in use (which is not recommended with a PRIVILEGE_CHECKS_USER account), for a transaction control statement such as BEGIN or COMMIT or DML logged in statement format (which are logged as a Query_log_event), the PRIVILEGE_CHECKS_USER account needs privileges to execute the statement contained in the event.

If LOAD DATA operations need to be carried out on the replication channel, use row-based binary logging (binlog_format=ROW). With this logging format, the FILE privilege is not needed to execute the event, so do not give the PRIVILEGE_CHECKS_USER account this privilege. The use of row-based binary logging is strongly recommended with replication channels that are secured using a PRIVILEGE_CHECKS_USER account. If REQUIRE_ROW_FORMAT is set for the channel, row-based

binary logging is required. The `Format_description_log_event`, which deletes any temporary files created by `LOAD DATA` events, is processed without privilege checks. For more information, see Section 17.5.1.19, “Replication and LOAD DATA”.

If the `init_replica` or `init_slave` system variable is set to specify one or more SQL statements to be executed when the replication SQL thread starts, the `PRIVILEGE_CHECKS_USER` account must have the privileges needed to execute these statements.

It is recommended that you never give any ACL privileges to the `PRIVILEGE_CHECKS_USER` account, including `CREATE USER`, `CREATE ROLE`, `DROP ROLE`, and `GRANT OPTION`, and do not permit the account to update the `mysql.user` table. With these privileges, the account could be used to create or modify user accounts on the server. To avoid ACL statements issued on the source server being replicated to the secured channel for execution (where they fail in the absence of these privileges), you can issue `SET sql_log_bin = 0` before all ACL statements and `SET sql_log_bin = 1` after them, to omit the statements from the source's binary log. Alternatively, you can set a dedicated current database before executing all ACL statements, and use a replication filter (`--binlog-ignore-db`) to filter out this database on the replica.

17.3.3.2 Privilege Checks For Group Replication Channels

From MySQL 8.0.19, as well as securing asynchronous and semi-synchronous replication, you may choose to use a `PRIVILEGE_CHECKS_USER` account to secure the two replication applier threads used by Group Replication. The `group_replication_applier` thread on each group member is used for applying the group's transactions, and the `group_replication_recovery` thread on each group member is used for state transfer from the binary log as part of distributed recovery when the member joins or rejoins the group.

To secure one of these threads, stop Group Replication, then issue the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) with the `PRIVILEGE_CHECKS_USER` option, specifying `group_replication_applier` or `group_replication_recovery` as the channel name. For example:

```
mysql> STOP GROUP_REPLICATION;
mysql> CHANGE MASTER TO PRIVILEGE_CHECKS_USER = 'gr_repl'@'%.example.com'
          FOR CHANNEL 'group_replication_recovery';
mysql> FLUSH PRIVILEGES;
mysql> START GROUP_REPLICATION;

Or from MySQL 8.0.23:
mysql> STOP GROUP_REPLICATION;
mysql> CHANGE REPLICATION SOURCE TO PRIVILEGE_CHECKS_USER = 'gr_repl'@'%.example.com'
          FOR CHANNEL 'group_replication_recovery';
mysql> FLUSH PRIVILEGES;
mysql> START GROUP_REPLICATION;
```

For Group Replication channels, the `REQUIRE_ROW_FORMAT` setting is automatically enabled when the channel is created, and cannot be disabled, so you do not need to specify this.

Important

In MySQL 8.0.19, ensure that you do not issue the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement with the `PRIVILEGE_CHECKS_USER` option while Group Replication is running. This action causes the relay log files for the channel to be purged, which might cause the loss of transactions that have been received and queued in the relay log, but not yet applied.

Group Replication requires every table that is to be replicated by the group to have a defined primary key, or primary key equivalent where the equivalent is a non-null unique key. Rather than using the checks carried out by the `sql_require_primary_key` system variable, Group Replication has its own built-in set of checks for primary keys or primary key equivalents. You may set the `REQUIRE_TABLE_PRIMARY_KEY_CHECK` option of the `CHANGE REPLICATION SOURCE TO` |

`CHANGE MASTER TO` statement to `ON` for a Group Replication channel. However, be aware that you might find some transactions that are permitted under Group Replication's built-in checks are not permitted under the checks carried out when you set `sql_require_primary_key = ON` or `REQUIRE_TABLE_PRIMARY_KEY_CHECK = ON`. For this reason, new and upgraded Group Replication channels from MySQL 8.0.20 (when the option was introduced) have `REQUIRE_TABLE_PRIMARY_KEY_CHECK` set to the default of `STREAM`, rather than to `ON`.

If a remote cloning operation is used for distributed recovery in Group Replication (see Section 18.5.4.2, "Cloning for Distributed Recovery"), from MySQL 8.0.19, the `PRIVILEGE_CHECKS_USER` account and related settings from the donor are cloned to the joining member. If the joining member is set to start Group Replication on boot, it automatically uses the account for privilege checks on the appropriate replication channels.

In MySQL 8.0.18, due to a number of limitations, it is recommended that you do not use a `PRIVILEGE_CHECKS_USER` account with Group Replication channels.

17.3.3.3 Recovering From Failed Replication Privilege Checks

If a privilege check against the `PRIVILEGE_CHECKS_USER` account fails, the transaction is not executed and replication stops for the channel. Details of the error and the last applied transaction are recorded in the Performance Schema `replication_applier_status_by_worker` table. Follow this procedure to recover from the error:

1. Identify the replicated event that caused the error and verify whether or not the event is expected and from a trusted source. You can use `mysqlbinlog` to retrieve and display the events that were logged around the time of the error. For instructions to do this, see Section 7.5, "Point-in-Time (Incremental) Recovery".

2. If the replicated event is not expected or is not from a known and trusted source, investigate the cause. If you can identify why the event took place and there are no security considerations, proceed to fix the error as described below.

3. If the `PRIVILEGE_CHECKS_USER` account should have been permitted to execute the transaction, but has been misconfigured, grant the missing privileges to the account, use a `FLUSH PRIVILEGES` statement or execute a `mysqladmin flush-privileges` or `mysqladmin reload` command to reload the grant tables, then restart replication for the channel.

4. If the transaction needs to be executed and you have verified that it is trusted, but the `PRIVILEGE_CHECKS_USER` account should not have this privilege normally, you can grant the required privilege to the `PRIVILEGE_CHECKS_USER` account temporarily. After the replicated event has been applied, remove the privilege from the account, and take any necessary steps to ensure the event does not recur if it is avoidable.

5. If the transaction is an administrative action that should only have taken place on the source and not on the replica, or should only have taken place on a single replication group member, skip the transaction on the server or servers where it stopped replication, then issue `START REPLICA` to restart replication on the channel. To avoid the situation in future, you could issue such administrative statements with `SET sql_log_bin = 0` before them and `SET sql_log_bin = 1` after them, so that they are not logged on the source.

6. If the transaction is a DDL or DML statement that should not have taken place on either the source or the replica, skip the transaction on the server or servers where it stopped replication, undo the transaction manually on the server where it originally took place, then issue `START REPLICA` to restart replication.

To skip a transaction, if GTIDs are in use, commit an empty transaction that has the GTID of the failing transaction, for example:

```
SET GTID_NEXT='aaa-bbb-ccc-ddd:N';
BEGIN;
```

```
COMMIT;
SET GTID_NEXT='AUTOMATIC';
```

If GTIDs are not in use, issue a `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter` statement to skip the event. For instructions to use this alternative method and more details about skipping transactions, see Section 17.1.7.3, “Skipping Transactions”.

17.4 Replication Solutions

Replication can be used in many different environments for a range of purposes. This section provides general notes and advice on using replication for specific solution types.

For information on using replication in a backup environment, including notes on the setup, backup procedure, and files to back up, see Section 17.4.1, “Using Replication for Backups”.

For advice and tips on using different storage engines on the source and replica, see Section 17.4.4, “Using Replication with Different Source and Replica Storage Engines”.

Using replication as a scale-out solution requires some changes in the logic and operation of applications that use the solution. See Section 17.4.5, “Using Replication for Scale-Out”.

For performance or data distribution reasons, you may want to replicate different databases to different replicas. See Section 17.4.6, “Replicating Different Databases to Different Replicas”

As the number of replicas increases, the load on the source can increase and lead to reduced performance (because of the need to replicate the binary log to each replica). For tips on improving your replication performance, including using a single secondary server as the source, see Section 17.4.7, “Improving Replication Performance”.

For guidance on switching sources, or converting replicas into sources as part of an emergency failover solution, see Section 17.4.8, “Switching Sources During Failover”.

For information on security measures specific to servers in a replication topology, see Section 17.3, “Replication Security”.

17.4.1 Using Replication for Backups

To use replication as a backup solution, replicate data from the source to a replica, and then back up the replica. The replica can be paused and shut down without affecting the running operation of the source, so you can produce an effective snapshot of “live” data that would otherwise require the source to be shut down.

How you back up a database depends on its size and whether you are backing up only the data, or the data and the replica state so that you can rebuild the replica in the event of failure. There are therefore two choices:

- If you are using replication as a solution to enable you to back up the data on the source, and the size of your database is not too large, the `mysqldump` tool may be suitable. See Section 17.4.1.1, “Backing Up a Replica Using mysqldump”.
- For larger databases, where `mysqldump` would be impractical or inefficient, you can back up the raw data files instead. Using the raw data files option also means that you can back up the binary and relay logs that make it possible to re-create the replica in the event of a replica failure. For more information, see Section 17.4.1.2, “Backing Up Raw Data from a Replica”.

Another backup strategy, which can be used for either source or replica servers, is to put the server in a read-only state. The backup is performed against the read-only server, which then is changed back to its usual read/write operational status. See Section 17.4.1.3, “Backing Up a Source or Replica by Making It Read Only”.

17.4.1.1 Backing Up a Replica Using mysqldump

Using `mysqldump` to create a copy of a database enables you to capture all of the data in the database in a format that enables the information to be imported into another instance of MySQL Server (see Section 4.5.4, “mysqldump — A Database Backup Program”). Because the format of the information is SQL statements, the file can easily be distributed and applied to running servers in the event that you need access to the data in an emergency. However, if the size of your data set is very large, `mysqldump` may be impractical.

Tip

Consider using the MySQL Shell dump utilities, which provide parallel dumping with multiple threads, file compression, and progress information display, as well as cloud features such as Oracle Cloud Infrastructure Object Storage streaming, and MySQL Database Service compatibility checks and modifications. Dumps can be easily imported into a MySQL Server instance or a MySQL Database Service DB System using the MySQL Shell load dump utilities. Installation instructions for MySQL Shell can be found here.

When using `mysqldump`, you should stop replication on the replica before starting the dump process to ensure that the dump contains a consistent set of data:

1. Stop the replica from processing requests. You can stop replication completely on the replica using `mysqladmin`:

   ```
   $> mysqladmin stop-slave
   ```

   Alternatively, you can stop only the replication SQL thread to pause event execution:

   ```
   $> mysql -e 'STOP SLAVE SQL_THREAD;'
   Or from MySQL 8.0.22:
   $> mysql -e 'STOP REPLICA SQL_THREAD;'
   ```

   This enables the replica to continue to receive data change events from the source's binary log and store them in the relay logs using the replication receiver thread, but prevents the replica from executing these events and changing its data. Within busy replication environments, permitting the replication receiver thread to run during backup may speed up the catch-up process when you restart the replication applier thread.

2. Run `mysqldump` to dump your databases. You may either dump all databases or select databases to be dumped. For example, to dump all databases:

   ```
   $> mysqldump --all-databases > fulldb.dump
   ```

3. Once the dump has completed, start replication again:

   ```
   $> mysqladmin start-slave
   ```

In the preceding example, you may want to add login credentials (user name, password) to the commands, and bundle the process up into a script that you can run automatically each day.

If you use this approach, make sure you monitor the replication process to ensure that the time taken to run the backup does not affect the replica's ability to keep up with events from the source. See Section 17.1.7.1, “Checking Replication Status”. If the replica is unable to keep up, you may want to add another replica and distribute the backup process. For an example of how to configure this scenario, see Section 17.4.6, “Replicating Different Databases to Different Replicas”.

17.4.1.2 Backing Up Raw Data from a Replica

To guarantee the integrity of the files that are copied, backing up the raw data files on your MySQL replica should take place while your replica server is shut down. If the MySQL server is still running,

background tasks may still be updating the database files, particularly those involving storage engines with background processes such as `InnoDB`. With `InnoDB`, these problems should be resolved during crash recovery, but since the replica server can be shut down during the backup process without affecting the execution of the source it makes sense to take advantage of this capability.

To shut down the server and back up the files:

1. Shut down the replica MySQL server:

```
$> mysqladmin shutdown
```

2. Copy the data files. You can use any suitable copying or archive utility, including `cp`, `tar` or `WinZip`. For example, assuming that the data directory is located under the current directory, you can archive the entire directory as follows:

```
$> tar cf /tmp/dbbackup.tar ./data
```

3. Start the MySQL server again. Under Unix:

```
$> mysqld_safe &
```

   Under Windows:

```
C:\> "C:\Program Files\MySQL\MySQL Server 8.0\bin\mysqld"
```

Normally you should back up the entire data directory for the replica MySQL server. If you want to be able to restore the data and operate as a replica (for example, in the event of failure of the replica), in addition to the data, you need to have the replica's connection metadata repository and applier metadata repository, and the relay log files. These items are needed to resume replication after you restore the replica's data. Assuming tables have been used for the replica's connection metadata repository and applier metadata repository (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”), which is the default in MySQL 8.0, these tables are backed up along with the data directory. If files have been used for the repositories, which is deprecated, you must back these up separately. The relay log files must be backed up separately if they have been placed in a different location to the data directory.

If you lose the relay logs but still have the `relay-log.info` file, you can check it to determine how far the replication SQL thread has executed in the source's binary logs. Then you can use `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) with the `SOURCE_LOG_FILE` | `MASTER_LOG_FILE` and `SOURCE_LOG_POS` | `MASTER_LOG_POS` options to tell the replica to re-read the binary logs from that point. This requires that the binary logs still exist on the source server.

If your replica is replicating `LOAD DATA` statements, you should also back up any `SQL_LOAD-*` files that exist in the directory that the replica uses for this purpose. The replica needs these files to resume replication of any interrupted `LOAD DATA` operations. The location of this directory is the value of the system variable `replica_load_tmpdir` (from MySQL 8.0.26) or `slave_load_tmpdir` (before MySQL 8.0.26). If the server was not started with that variable set, the directory location is the value of the `tmpdir` system variable.

17.4.1.3 Backing Up a Source or Replica by Making It Read Only

It is possible to back up either source or replica servers in a replication setup by acquiring a global read lock and manipulating the `read_only` system variable to change the read-only state of the server to be backed up:

1. Make the server read-only, so that it processes only retrievals and blocks updates.
2. Perform the backup.
3. Change the server back to its normal read/write state.

Note

The instructions in this section place the server to be backed up in a state that is safe for backup methods that get the data from the server, such as `mysqldump` (see Section 4.5.4, “mysqldump — A Database Backup Program”). You should not attempt to use these instructions to make a binary backup by copying files directly because the server may still have modified data cached in memory and not flushed to disk.

The following instructions describe how to do this for a source and for a replica. For both scenarios discussed here, suppose that you have the following replication setup:

- A source server S1
- A replica server R1 that has S1 as its source
- A client C1 connected to S1
- A client C2 connected to R1

In either scenario, the statements to acquire the global read lock and manipulate the `read_only` variable are performed on the server to be backed up and do not propagate to any replicas of that server.

Scenario 1: Backup with a Read-Only Source

Put the source S1 in a read-only state by executing these statements on it:

```
mysql> FLUSH TABLES WITH READ LOCK;
mysql> SET GLOBAL read_only = ON;
```

While S1 is in a read-only state, the following properties are true:

- Requests for updates sent by C1 to S1 block because the server is in read-only mode.
- Requests for query results sent by C1 to S1 succeed.
- Making a backup on S1 is safe.
- Making a backup on R1 is not safe. This server is still running, and might be processing the binary log or update requests coming from client C2.

While S1 is read only, perform the backup. For example, you can use `mysqldump`.

After the backup operation on S1 completes, restore S1 to its normal operational state by executing these statements:

```
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

Although performing the backup on S1 is safe (as far as the backup is concerned), it is not optimal for performance because clients of S1 are blocked from executing updates.

This strategy applies to backing up a source in a replication setup, but can also be used for a single server in a nonreplication setting.

Scenario 2: Backup with a Read-Only Replica

Put the replica R1 in a read-only state by executing these statements on it:

```
mysql> FLUSH TABLES WITH READ LOCK;
```

```
mysql> SET GLOBAL read_only = ON;
```

While R1 is in a read-only state, the following properties are true:

- The source S1 continues to operate, so making a backup on the source is not safe.
- The replica R1 is stopped, so making a backup on the replica R1 is safe.

These properties provide the basis for a popular backup scenario: Having one replica busy performing a backup for a while is not a problem because it does not affect the entire network, and the system is still running during the backup. In particular, clients can still perform updates on the source server, which remains unaffected by backup activity on the replica.

While R1 is read only, perform the backup. For example, you can use `mysqldump`.

After the backup operation on R1 completes, restore R1 to its normal operational state by executing these statements:

```
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

After the replica is restored to normal operation, it again synchronizes to the source by catching up with any outstanding updates from the source's binary log.

17.4.2 Handling an Unexpected Halt of a Replica

In order for replication to be resilient to unexpected halts of the server (sometimes described as crash-safe) it must be possible for the replica to recover its state before halting. This section describes the impact of an unexpected halt of a replica during replication, and how to configure a replica for the best chance of recovery to continue replication.

After an unexpected halt of a replica, upon restart the replication SQL thread must recover information about which transactions have been executed already. The information required for recovery is stored in the replica's applier metadata repository. From MySQL 8.0, this repository is created by default as an `InnoDB` table named `mysql.slave_relay_log_info`. By using this transactional storage engine the information is always recoverable upon restart. Updates to the applier metadata repository are committed together with the transactions, meaning that the replica's progress information recorded in that repository is always consistent with what has been applied to the database, even in the event of an unexpected server halt. For more information on the applier metadata repository, see Section 17.2.4, “Relay Log and Replication Metadata Repositories”.

DML transactions and also atomic DDL update the replication positions in the replica's applier metadata repository in the `mysql.slave_relay_log_info` table together with applying the changes to the database, as an atomic operation. In all other cases, including DDL statements that are not fully atomic, and exempted storage engines that do not support atomic DDL, the `mysql.slave_relay_log_info` table might be missing updates associated with replicated data if the server halts unexpectedly. Restoring updates in this case is a manual process. For details on atomic DDL support in MySQL 8.0, and the resulting behavior for the replication of certain statements, see Section 13.1.1, “Atomic Data Definition Statement Support”.

The recovery process by which a replica recovers from an unexpected halt varies depending on the configuration of the replica. The details of the recovery process are influenced by the chosen method of replication, whether the replica is single-threaded or multithreaded, and the setting of relevant system variables. The overall aim of the recovery process is to identify what transactions had already been applied on the replica's database before the unexpected halt occurred, and retrieve and apply the transactions that the replica missed following the unexpected halt.

- For GTID-based replication, the recovery process needs the GTIDs of the transactions that were already received or committed by the replica. The missing transactions can be retrieved from the

source using GTID auto-positioning, which automatically compares the source's transactions to the replica's transactions and identifies the missing transactions.

- For file position based replication, the recovery process needs an accurate replication SQL thread (applier) position showing the last transaction that was applied on the replica. Based on that position, the replication I/O thread (receiver) retrieves from the source's binary log all of the transactions that should be applied on the replica from that point on.

Using GTID-based replication makes it easiest to configure replication to be resilient to unexpected halts. GTID auto-positioning means the replica can reliably identify and retrieve missing transactions, even if there are gaps in the sequence of applied transactions.

The following information provides combinations of settings that are appropriate for different types of replica to guarantee recovery as far as this is under the control of replication.

Important

Some factors outside the control of replication can have an impact on the replication recovery process and the overall state of replication after the recovery process. In particular, the settings that influence the recovery process for individual storage engines might result in transactions being lost in the event of an unexpected halt of a replica, and therefore unavailable to the replication recovery process. The `innodb_flush_log_at_trx_commit=1` setting mentioned in the list below is a key setting for a replication setup that uses `InnoDB` with transactions. However, other settings specific to `InnoDB` or to other storage engines, especially those relating to flushing or synchronization, can also have an impact. Always check for and apply recommendations made by your chosen storage engines about crash-safe settings.

The following combination of settings on a replica is the most resilient to unexpected halts:

- When GTID-based replication is in use (`gtid_mode=ON`), set `SOURCE_AUTO_POSITION=1` | `MASTER_AUTO_POSITION=1`, which activates GTID auto-positioning for the connection to the source to automatically identify and retrieve missing transactions. This option is set using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). If the replica has multiple replication channels, you need to set this option for each channel individually. For details of how GTID auto-positioning works, see Section 17.1.3.3, “GTID Auto-Positioning”. When file position based replication is in use, `SOURCE_AUTO_POSITION=1` | `MASTER_AUTO_POSITION=1` is not used, and instead the binary log position or relay log position is used to control where replication starts.

- From MySQL 8.0.27, when GTID-based replication is in use (`gtid_mode=ON`), set `GTID_ONLY=1`, which makes the replica use only GTIDs in the recovery process, and stop persisting binary log and relay log file names and file positions in the replication metadata repositories. This option is set using a `CHANGE REPLICATION SOURCE TO` statement. If the replica has multiple replication channels, you need to set this option for each channel individually. With `GTID_ONLY=1`, during recovery, the file position information is ignored and GTID auto-skip is used to skip transactions that have already been supplied, rather than identifying the correct file position. This strategy is more efficient provided that you purge relay logs using the default setting for `relay_log_purge`, which means only one relay log file needs to be inspected.

- Set `sync_relay_log=1`, which instructs the replication receiver thread to synchronize the relay log to disk after each received transaction is written to it. This means the replica's record of the current position read from the source's binary log (in the applier metadata repository) is never ahead of the record of transactions saved in the relay log. Note that although this setting is the safest, it is also the slowest due to the number of disk writes involved. With `sync_relay_log > 1`, or `sync_relay_log=0` (where synchronization is handled by the operating system), in the event of an unexpected halt of a replica there might be committed transactions that have not been synchronized to disk. Such transactions can cause the recovery process to fail if the recovering replica, based on the information it has in the relay log as last synchronized to disk, tries to retrieve and apply the

transactions again instead of skipping them. Setting `sync_relay_log=1` is particularly important for a multi-threaded replica, where the recovery process fails if gaps in the sequence of transactions cannot be filled using the information in the relay log. For a single-threaded replica, the recovery process only needs to use the relay log if the relevant information is not available in the applier metadata repository.

- Set `innodb_flush_log_at_trx_commit=1`, which synchronizes the `InnoDB` logs to disk before each transaction is committed. This setting, which is the default, ensures that `InnoDB` tables and the `InnoDB` logs are saved on disk so that there is no longer a requirement for the information in the relay log regarding the transaction. Combined with the setting `sync_relay_log=1`, this setting further ensures that the content of the `InnoDB` tables and the `InnoDB` logs is consistent with the content of the relay log at all times, so that purging the relay log files cannot cause unfillable gaps in the replica's history of transactions in the event of an unexpected halt.

- Set `relay_log_info_repository = TABLE`, which stores the replication SQL thread position in the `InnoDB` table `mysql.slave_relay_log_info`, and updates it together with the transaction commit to ensure a record that is always accurate. This setting is the default from MySQL 8.0, and the `FILE` setting is deprecated. From MySQL 8.0.23, the use of the system variable itself is deprecated, so omit it and allow it to default. If the `FILE` setting is used, which was the default in earlier releases, the information is stored in a file in the data directory that is updated after the transaction has been applied. This creates a risk of losing synchrony with the source depending at which stage of processing a transaction the replica halts at, or even corruption of the file itself. With the setting `relay_log_info_repository = FILE`, recovery is not guaranteed.

- Set `relay_log_recovery = ON`, which enables automatic relay log recovery immediately following server startup. This global variable defaults to `OFF` and is read-only at runtime, but you can set it to `ON` with the `--relay-log-recovery` option at replica startup following an unexpected halt of a replica. Note that this setting ignores the existing relay log files, in case they are corrupted or inconsistent. The relay log recovery process starts a new relay log file and fetches transactions from the source beginning at the replication SQL thread position recorded in the applier metadata repository. The previous relay log files are removed over time by the replica's normal purge mechanism.

For a multithreaded replica, setting `relay_log_recovery = ON` automatically handles any inconsistencies and gaps in the sequence of transactions that have been executed from the relay log. These gaps can occur when file position based replication is in use. (For more details, see Section 17.5.1.34, “Replication and Transaction Inconsistencies”.) The relay log recovery process deals with gaps using the same method as the `START REPLICA UNTIL SQL_AFTER_MTS_GAPS` (or before MySQL 8.0.22, START SLAVE instead of START REPLICA) statement would. When the replica reaches a consistent gap-free state, the relay log recovery process goes on to fetch further transactions from the source beginning at the replication SQL thread position. When GTID-based replication is in use, from MySQL 8.0.18 a multithreaded replica checks first whether `MASTER_AUTO_POSITION` is set to `ON`, and if it is, omits the step of calculating the transactions that should be skipped or not skipped, so that the old relay logs are not required for the recovery process.

17.4.3 Monitoring Row-based Replication

The current progress of the replication applier (SQL) thread when using row-based replication is monitored through Performance Schema instrument stages, enabling you to track the processing of operations and check the amount of work completed and work estimated. When these Performance Schema instrument stages are enabled the `events_stages_current` table shows stages for applier threads and their progress. For background information, see Section 27.12.5, “Performance Schema Stage Event Tables”.

To track progress of all three row-based replication event types (write, update, delete):

- Enable the three Performance Schema stages by issuing:

```
mysql> UPDATE performance_schema.setup_instruments SET ENABLED = 'YES'
```

```
    -> WHERE NAME LIKE 'stage/sql/Applying batch of row changes%';
```

- Wait for some events to be processed by the replication applier thread and then check progress by looking into the `events_stages_current` table. For example to get progress for `update` events issue:

```
mysql> SELECT WORK_COMPLETED, WORK_ESTIMATED FROM performance_schema.events_stages_current
    -> WHERE EVENT_NAME LIKE 'stage/sql/Applying batch of row changes (update)'
```

- If `binlog_rows_query_log_events` is enabled, information about queries is stored in the binary log and is exposed in the `processlist_info` field. To see the original query that triggered this event:

```
mysql> SELECT db, processlist_state, processlist_info FROM performance_schema.threads
    -> WHERE processlist_state LIKE 'stage/sql/Applying batch of row changes%' AND thread_id = N;
```

17.4.4 Using Replication with Different Source and Replica Storage Engines

It does not matter for the replication process whether the original table on the source and the replicated table on the replica use different storage engine types. In fact, the `default_storage_engine` system variable is not replicated.

This provides a number of benefits in the replication process in that you can take advantage of different engine types for different replication scenarios. For example, in a typical scale-out scenario (see Section 17.4.5, “Using Replication for Scale-Out”), you want to use `InnoDB` tables on the source to take advantage of the transactional functionality, but use `MyISAM` on the replicas where transaction support is not required because the data is only read. When using replication in a data-logging environment you may want to use the `Archive` storage engine on the replica.

Configuring different engines on the source and replica depends on how you set up the initial replication process:

- If you used `mysqldump` to create the database snapshot on your source, you could edit the dump file text to change the engine type used on each table.

  Another alternative for `mysqldump` is to disable engine types that you do not want to use on the replica before using the dump to build the data on the replica. For example, you can add the `--skip-federated` option on your replica to disable the `FEDERATED` engine. If a specific engine does not exist for a table to be created, MySQL uses the default engine type, usually `InnoDB`. (This requires that the `NO_ENGINE_SUBSTITUTION` SQL mode is not enabled.) If you want to disable additional engines in this way, you may want to consider building a special binary to be used on the replica that supports only the engines you want.

- If you use raw data files (a binary backup) to set up the replica, it is not possible to change the initial table format. Instead, use `ALTER TABLE` to change the table types after the replica has been started.

- For new source/replica replication setups where there are currently no tables on the source, avoid specifying the engine type when creating new tables.

If you are already running a replication solution and want to convert your existing tables to another engine type, follow these steps:

1. Stop the replica from running replication updates:

   ```
   mysql> STOP SLAVE;
   Or from MySQL 8.0.22:
   mysql> STOP REPLICA;
   ```

   This makes it possible to change engine types without interruption.

2. Execute an `ALTER TABLE ... ENGINE='engine_type'` for each table to be changed.

3. Start the replication process again:

```
mysql> START SLAVE;
```

Or, beginning with MySQL 8.0.22:

```
mysql> START REPLICA;
```

Although the `default_storage_engine` variable is not replicated, be aware that `CREATE TABLE` and `ALTER TABLE` statements that include the engine specification are replicated to the replica correctly. If, in the case of a `CSV` table, you execute this statement:

```
mysql> ALTER TABLE csvtable ENGINE='MyISAM';
```

This statement is replicated; the table's engine type on the replica is converted to `InnoDB`, even if you have previously changed the table type on the replica to an engine other than `CSV`. If you want to retain engine differences on the source and replica, you should be careful to use the `default_storage_engine` variable on the source when creating a new table. For example, instead of:

```
mysql> CREATE TABLE tablea (columna int) Engine=MyISAM;
```

Use this format:

```
mysql> SET default_storage_engine=MyISAM;
mysql> CREATE TABLE tablea (columna int);
```

When replicated, the `default_storage_engine` variable is ignored, and the `CREATE TABLE` statement executes on the replica using the replica's default engine.

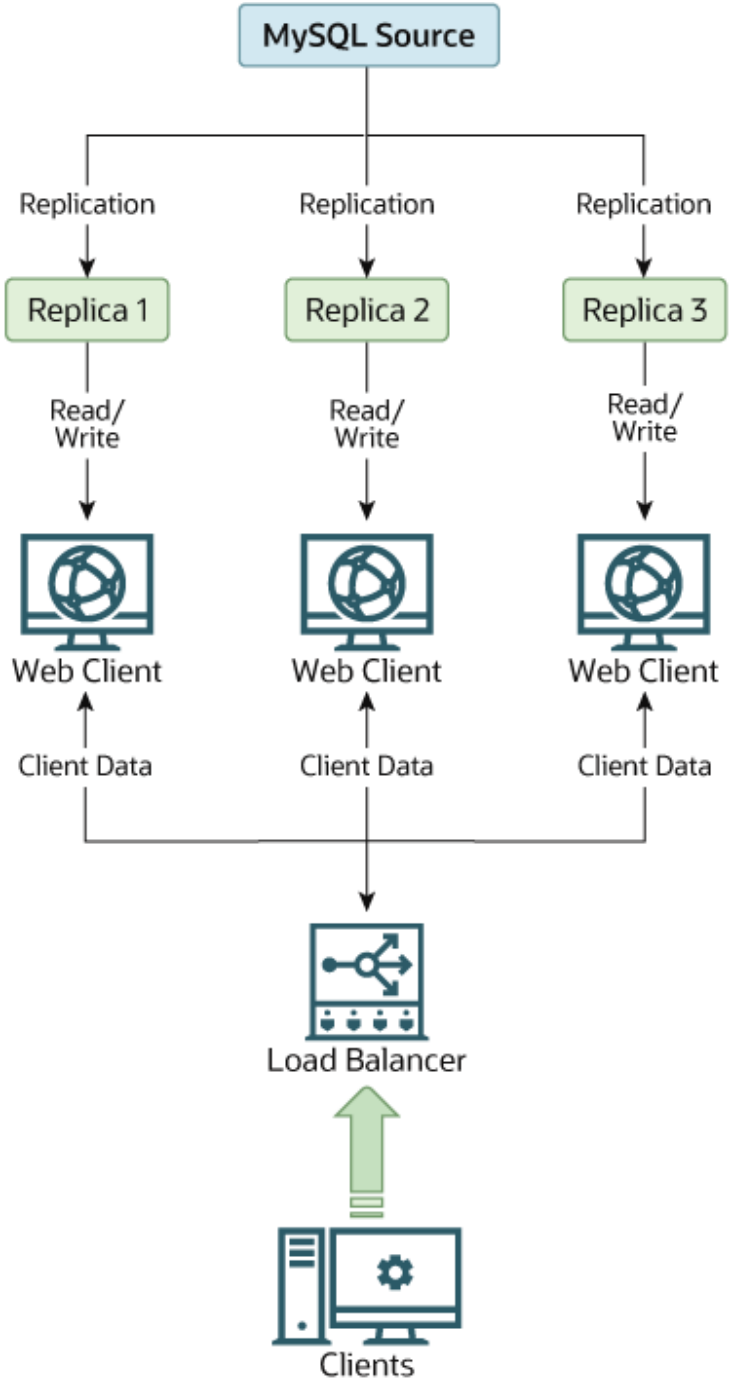
17.4.5 Using Replication for Scale-Out

You can use replication as a scale-out solution; that is, where you want to split up the load of database queries across multiple database servers, within some reasonable limitations.

Because replication works from the distribution of one source to one or more replicas, using replication for scale-out works best in an environment where you have a high number of reads and low number of writes/updates. Most websites fit into this category, where users are browsing the website, reading articles, posts, or viewing products. Updates only occur during session management, or when making a purchase or adding a comment/message to a forum.

Replication in this situation enables you to distribute the reads over the replicas, while still enabling your web servers to communicate with the source when a write is required. You can see a sample replication layout for this scenario in Figure 17.1, “Using Replication to Improve Performance During Scale-Out”.

Figure 17.1 Using Replication to Improve Performance During Scale-Out



If the part of your code that is responsible for database access has been properly abstracted/modularized, converting it to run with a replicated setup should be very smooth and easy. Change the implementation of your database access to send all writes to the source, and to send reads to either the source or a replica. If your code does not have this level of abstraction, setting up a replicated system gives you the opportunity and motivation to clean it up. Start by creating a wrapper library or module that implements the following functions:

- `safe_writer_connect()`
- `safe_reader_connect()`
- `safe_reader_statement()`

- `safe_writer_statement()`

`safe_` in each function name means that the function takes care of handling all error conditions. You can use different names for the functions. The important thing is to have a unified interface for connecting for reads, connecting for writes, doing a read, and doing a write.

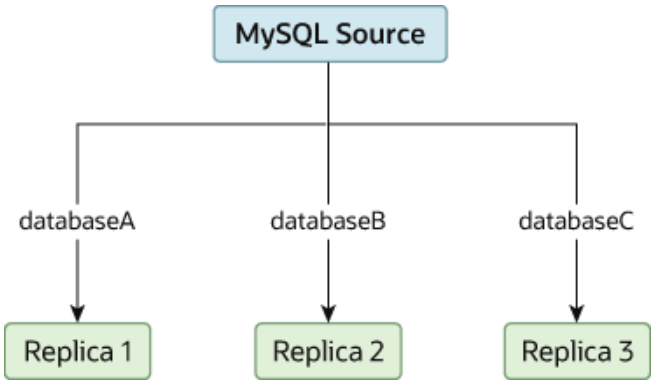
Then convert your client code to use the wrapper library. This may be a painful and scary process at first, but it pays off in the long run. All applications that use the approach just described are able to take advantage of a source/replica configuration, even one involving multiple replicas. The code is much easier to maintain, and adding troubleshooting options is trivial. You need modify only one or two functions (for example, to log how long each statement took, or which statement among those issued gave you an error).

If you have written a lot of code, you may want to automate the conversion task by writing a conversion script. Ideally, your code uses consistent programming style conventions. If not, then you are probably better off rewriting it anyway, or at least going through and manually regularizing it to use a consistent style.

17.4.6 Replicating Different Databases to Different Replicas

There may be situations where you have a single source server and want to replicate different databases to different replicas. For example, you may want to distribute different sales data to different departments to help spread the load during data analysis. A sample of this layout is shown in Figure 17.2, “Replicating Databases to Separate Replicas”.

Figure 17.2 Replicating Databases to Separate Replicas



You can achieve this separation by configuring the source and replicas as normal, and then limiting the binary log statements that each replica processes by using the `--replicate-wild-do-table` configuration option on each replica.

Important

You should *not* use `--replicate-do-db` for this purpose when using statement-based replication, since statement-based replication causes this option's effects to vary according to the database that is currently selected. This applies to mixed-format replication as well, since this enables some updates to be replicated using the statement-based format.

However, it should be safe to use `--replicate-do-db` for this purpose if you are using row-based replication only, since in this case the currently selected database has no effect on the option's operation.

For example, to support the separation as shown in Figure 17.2, “Replicating Databases to Separate Replicas”, you should configure each replica as follows, before executing `START REPLICA`:

- Replica 1 should use `--replicate-wild-do-table=databaseA.%`.

- Replica 2 should use `--replicate-wild-do-table=databaseB.%`.
- Replica 3 should use `--replicate-wild-do-table=databaseC.%`.

Each replica in this configuration receives the entire binary log from the source, but executes only those events from the binary log that apply to the databases and tables included by the `--replicate-wild-do-table` option in effect on that replica.

If you have data that must be synchronized to the replicas before replication starts, you have a number of choices:

- Synchronize all the data to each replica, and delete the databases, tables, or both that you do not want to keep.
- Use `mysqldump` to create a separate dump file for each database and load the appropriate dump file on each replica.
- Use a raw data file dump and include only the specific files and databases that you need for each replica.

Note

This does not work with `InnoDB` databases unless you use `innodb_file_per_table`.

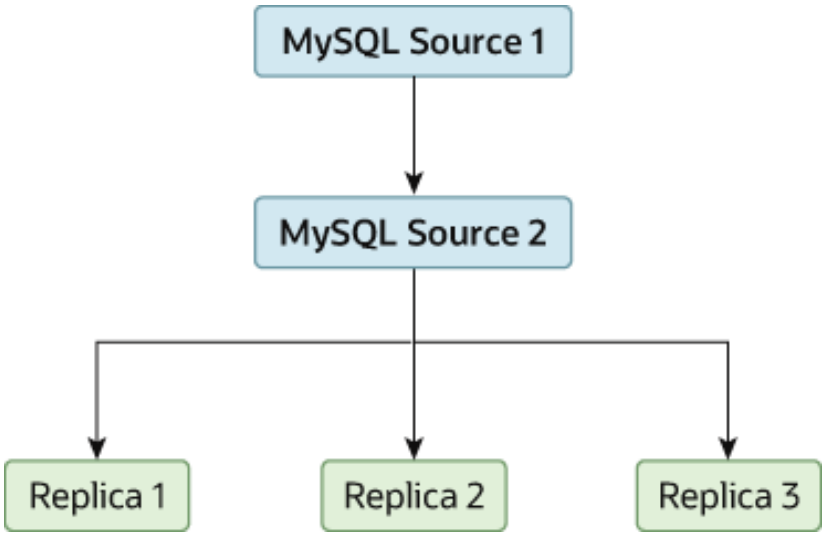
17.4.7 Improving Replication Performance

As the number of replicas connecting to a source increases, the load, although minimal, also increases, as each replica uses a client connection to the source. Also, as each replica must receive a full copy of the source's binary log, the network load on the source may also increase and create a bottleneck.

If you are using a large number of replicas connected to one source, and that source is also busy processing requests (for example, as part of a scale-out solution), then you may want to improve the performance of the replication process.

One way to improve the performance of the replication process is to create a deeper replication structure that enables the source to replicate to only one replica, and for the remaining replicas to connect to this primary replica for their individual replication requirements. A sample of this structure is shown in Figure 17.3, “Using an Additional Replication Source to Improve Performance”.

Figure 17.3 Using an Additional Replication Source to Improve Performance



For this to work, you must configure the MySQL instances as follows:

- Source 1 is the primary source where all changes and updates are written to the database. Binary logging is enabled on both source servers, which is the default.
- Source 2 is the replica to the server Source 1 that provides the replication functionality to the remainder of the replicas in the replication structure. Source 2 is the only machine permitted to connect to Source 1. Source 2 has the `--log-slave-updates` option enabled (which is the default). With this option, replication instructions from Source 1 are also written to Source 2's binary log so that they can then be replicated to the true replicas.
- Replica 1, Replica 2, and Replica 3 act as replicas to Source 2, and replicate the information from Source 2, which actually consists of the upgrades logged on Source 1.

The above solution reduces the client load and the network interface load on the primary source, which should improve the overall performance of the primary source when used as a direct database solution.

If your replicas are having trouble keeping up with the replication process on the source, there are a number of options available:

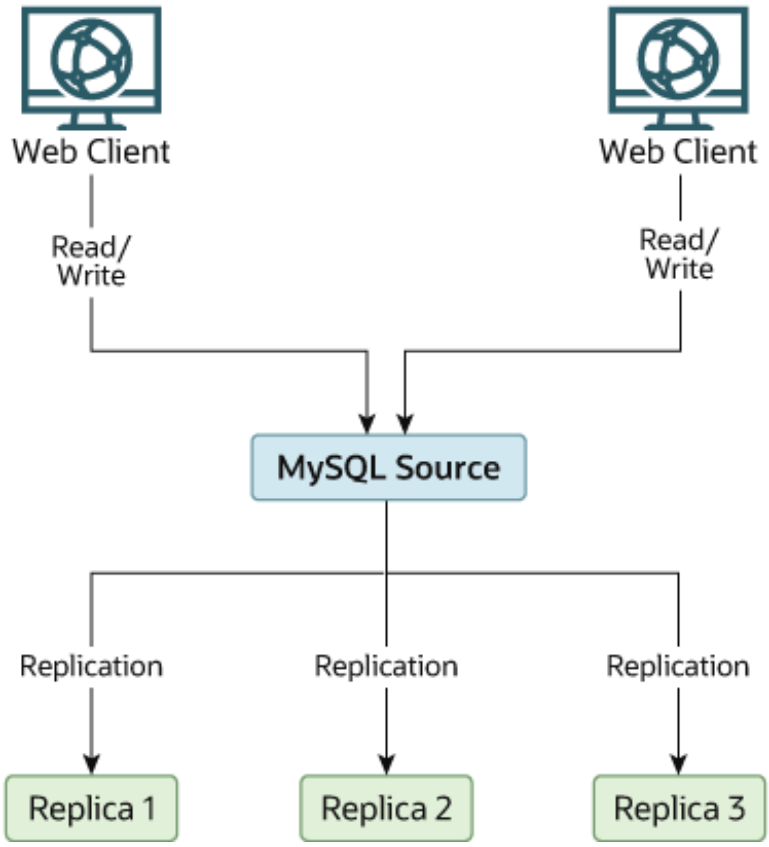
- If possible, put the relay logs and the data files on different physical drives. To do this, set the `relay_log` system variable to specify the location of the relay log.
- If heavy disk I/O activity for reads of the binary log file and relay log files is an issue, consider increasing the value of the `rpl_read_size` system variable. This system variable controls the minimum amount of data read from the log files, and increasing it might reduce file reads and I/O stalls when the file data is not currently cached by the operating system. Note that a buffer the size of this value is allocated for each thread that reads from the binary log and relay log files, including dump threads on sources and coordinator threads on replicas. Setting a large value might therefore have an impact on memory consumption for servers.
- If the replicas are significantly slower than the source, you may want to divide up the responsibility for replicating different databases to different replicas. See Section 17.4.6, “Replicating Different Databases to Different Replicas”.
- If your source makes use of transactions and you are not concerned about transaction support on your replicas, use `MyISAM` or another nontransactional engine on the replicas. See Section 17.4.4, “Using Replication with Different Source and Replica Storage Engines”.
- If your replicas are not acting as sources, and you have a potential solution in place to ensure that you can bring up a source in the event of failure, then you can disable the system variable `log_replica_updates` (from MySQL 8.0.26) or `log_slave_updates` (before MySQL 8.0.26) on the replicas. This prevents “dumb” replicas from also logging events they have executed into their own binary log.

17.4.8 Switching Sources During Failover

You can tell a replica to change to a new source using the `CHANGE REPLICATION SOURCE TO` statement (prior to MySQL 8.0.23: `CHANGE MASTER TO`. The replica does not check whether the databases on the source are compatible with those on the replica; it simply begins reading and executing events from the specified coordinates in the new source's binary log. In a failover situation, all the servers in the group are typically executing the same events from the same binary log file, so changing the source of the events should not affect the structure or integrity of the database, provided that you exercise care in making the change.

Replicas should be run with binary logging enabled (the `--log-bin` option), which is the default. If you are not using GTIDs for replication, then the replicas should also be run with `--log-slave-updates=OFF` (logging replica updates is the default). In this way, the replica is ready to become a source without restarting the replica `mysqld`. Assume that you have the structure shown in Figure 17.4, “Redundancy Using Replication, Initial Structure”.

Figure 17.4 Redundancy Using Replication, Initial Structure



In this diagram, the `Source` holds the source database, the `Replica*` hosts are replicas, and the `Web Client` machines are issuing database reads and writes. Web clients that issue only reads (and would normally be connected to the replicas) are not shown, as they do not need to switch to a new server in the event of failure. For a more detailed example of a read/write scale-out replication structure, see Section 17.4.5, “Using Replication for Scale-Out”.

Each MySQL replica (`Replica 1`, `Replica 2`, and `Replica 3`) is a replica running with binary logging enabled, and with `--log-slave-updates=OFF`. Because updates received by a replica from the source are not written to the binary log when `--log-slave-updates=OFF` is specified, the binary log on each replica is initially empty. If for some reason `Source` becomes unavailable, you can pick one of the replicas to become the new source. For example, if you pick `Replica 1`, all `Web Clients` should be redirected to `Replica 1`, which writes the updates to its binary log. `Replica 2` and `Replica 3` should then replicate from `Replica 1`.

The reason for running the replica with `--log-slave-updates=OFF` is to prevent replicas from receiving updates twice in case you cause one of the replicas to become the new source. If `Replica 1` has `--log-slave-updates` enabled, which is the default, it writes any updates that it receives from `Source` in its own binary log. This means that, when `Replica 2` changes from `Source` to `Replica 1` as its source, it may receive updates from `Replica 1` that it has already received from `Source`.

Make sure that all replicas have processed any statements in their relay log. On each replica, issue `STOP REPLICA IO_THREAD`, then check the output of `SHOW PROCESSLIST` until you see `Has read all relay log`. When this is true for all replicas, they can be reconfigured to the new setup. On the replica `Replica 1` being promoted to become the source, issue `STOP REPLICA` and `RESET MASTER`.

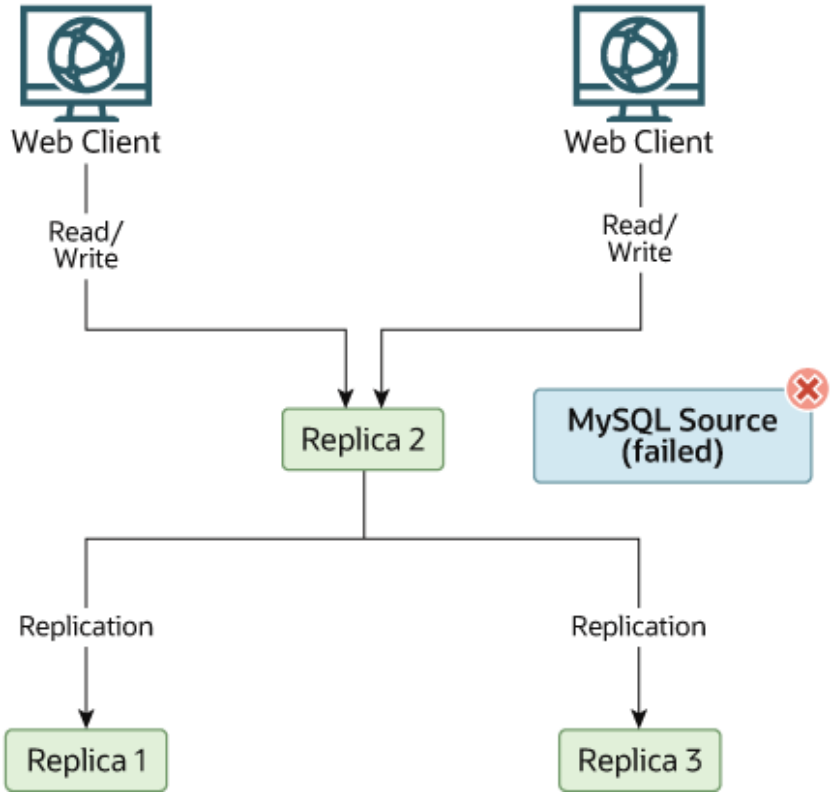
On the other replicas `Replica 2` and `Replica 3`, use `STOP REPLICA` and `CHANGE REPLICATION SOURCE TO SOURCE_HOST='Replica1'` or `CHANGE MASTER TO MASTER_HOST='Replica1'` (where `'Replica1'` represents the real host name of `Replica 1`). To use `CHANGE REPLICATION`

`SOURCE TO`, add all information about how to connect to `Replica 1` from `Replica 2` or `Replica 3` (*`user`*, *`password`*, *`port`*). When issuing the statement in this scenario, there is no need to specify the name of the `Replica 1` binary log file or log position to read from, since the first binary log file and position 4 are the defaults. Finally, execute `START REPLICA` on `Replica 2` and `Replica 3`.

Once the new replication setup is in place, you need to tell each `Web Client` to direct its statements to `Replica 1`. From that point on, all updates sent by `Web Client` to `Replica 1` are written to the binary log of `Replica 1`, which then contains every update sent to `Replica 1` since `Source` became unavailable.

The resulting server structure is shown in Figure 17.5, “Redundancy Using Replication, After Source Failure”.

Figure 17.5 Redundancy Using Replication, After Source Failure



When `Source` becomes available again, you should make it a replica of `Replica 1`. To do this, issue on `Source` the same `CHANGE REPLICATION SOURCE TO` (or `CHANGE MASTER TO`) statement as that issued on `Replica 2` and `Replica 3` previously. `Source` then becomes a replica of `Replica 1` and picks up the `Web Client` writes that it missed while it was offline.

To make `Source` a source again, use the preceding procedure as if `Replica 1` were unavailable and `Source` were to be the new source. During this procedure, do not forget to run `RESET MASTER` on `Source` before making `Replica 1`, `Replica 2`, and `Replica 3` replicas of `Source`. If you fail to do this, the replicas may pick up stale writes from the `Web Client` applications dating from before the point at which `Source` became unavailable.

You should be aware that there is no synchronization between replicas, even when they share the same source, and thus some replicas might be considerably ahead of others. This means that in some cases the procedure outlined in the previous example might not work as expected. In practice, however, relay logs on all replicas should be relatively close together.

One way to keep applications informed about the location of the source is to have a dynamic DNS entry for the source host. With `BIND`, you can use `nsupdate` to update the DNS dynamically.

17.4.9 Switching Sources and Replicas with Asynchronous Connection Failover

Beginning with MySQL 8.0.22, you can use the asynchronous connection failover mechanism to automatically establish an asynchronous (source to replica) replication connection to a new source after the existing connection from a replica to its source fails. The asynchronous connection failover mechanism can be used to keep a replica synchronized with multiple MySQL servers or groups of servers that share data. The list of potential source servers is stored on the replica, and in the event of a connection failure, a new source is selected from the list based on a weighted priority that you set.

From MySQL 8.0.23, the asynchronous connection failover mechanism also supports Group Replication topologies, by automatically monitoring changes to group membership and distinguishing between primary and secondary servers. When you add a group member to the source list and define it as part of a managed group, the asynchronous connection failover mechanism updates the source list to keep it in line with membership changes, adding and removing group members automatically as they join or leave. Only online group members that are in the majority are used for connections and obtaining status. The last remaining member of a managed group is not removed automatically even if it leaves the group, so that the configuration of the managed group is kept. However, you can delete a managed group manually if it is no longer needed.

From MySQL 8.0.27, the asynchronous connection failover mechanism also enables a replica that is part of a managed replication group to automatically reconnect to the sender if the current receiver (the primary of the group) fails. This feature works with Group Replication, on a group configured in single-primary mode, where the group's primary is a replica that has a replication channel using the mechanism. The feature is designed for a group of senders and a group of receivers to keep synchronized with each other even when some members are temporarily unavailable. It also synchronizes a group of receivers with one or more senders that are not part of a managed group. A replica that is not part of a replication group cannot use this feature.

The requirements for using the asynchronous connection failover mechanism are as follows:

- GTIDs must be in use on the source and the replica (`gtid_mode=ON`), and the `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement must be enabled on the replica, so that GTID auto-positioning is used for the connection to the source.
- The same replication user account and password must exist on all the source servers in the source list for the channel. This account is used for the connection to each of the sources. You can set up different accounts for different channels.
- The replication user account must be given `SELECT` permissions on the Performance Schema tables, for example, by issuing `GRANT SELECT ON performance_schema.* TO 'repl_user';`
- The replication user account and password cannot be specified on the statement used to start replication, because they need to be available on the automatic restart for the connection to the alternative source. They must be set for the channel using the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement on the replica, and recorded in the replication metadata repositories.
- If the channel where the asynchronous connection failover mechanism is in use is on the primary of a Group Replication single-primary mode group, from MySQL 8.0.27, asynchronous connection failover between replicas is also active by default. In this situation, the replication channel and the replication user account and password for the channel must be set up on all the secondary servers in the replication group, and on any new joining members. If the new servers are provisioned using MySQL's clone functionality, this all happens automatically.

Important

If you do not want asynchronous connection failover to take place between replicas in this situation, disable it by disabling the member action `mysql_start_failover_channels_if_primary` for the group, using the `group_replication_disable_member_action` function. When the feature is disabled, you do not need to configure the replication channel on the secondary group members, but if the primary goes offline or into an error state, replication stops for the channel.

From MySQL Shell 8.0.27 and MySQL 8.0.27, MySQL InnoDB ClusterSet is available to provide disaster tolerance for InnoDB Cluster deployments by linking a primary InnoDB Cluster with one or more replicas of itself in alternate locations, such as different datacenters. Consider using this solution instead to simplify the setup of a new multi-group deployment for replication, failover, and disaster recovery. You can adopt an existing Group Replication deployment as an InnoDB Cluster.

InnoDB ClusterSet and InnoDB Cluster are designed to abstract and simplify the procedures for setting up, managing, monitoring, recovering, and repairing replication groups. InnoDB ClusterSet automatically manages replication from a primary cluster to replica clusters using a dedicated ClusterSet replication channel. You can use administrator commands to trigger a controlled switchover or emergency failover between groups if the primary cluster is not functioning normally. Servers and groups can easily be added to or removed from the InnoDB ClusterSet deployment after the initial setup when demand changes. For more information, see MySQL InnoDB ClusterSet.

17.4.9.1 Asynchronous Connection Failover for Sources

To activate asynchronous connection failover for a replication channel set `SOURCE_CONNECTION_AUTO_FAILOVER=1` on the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) for the channel. GTID auto-positioning must be in use for the channel (`SOURCE_AUTO_POSITION = 1` | `MASTER_AUTO_POSITION = 1`).

Important

When the existing connection to a source fails, the replica first retries the same connection the number of times specified by the `SOURCE_RETRY_COUNT` | `MASTER_RETRY_COUNT` option of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement. The interval between attempts is set by the `SOURCE_CONNECT_RETRY` | `MASTER_CONNECT_RETRY` option. When these attempts are exhausted, the asynchronous connection failover mechanism takes over. Note that the defaults for these options, which were designed for a connection to a single source, make the replica retry the same connection for 60 days. To ensure that the asynchronous connection failover mechanism can be activated promptly, set `SOURCE_RETRY_COUNT` | `MASTER_RETRY_COUNT` and `SOURCE_CONNECT_RETRY` | `MASTER_CONNECT_RETRY` to minimal numbers that just allow a few retry attempts with the same source, in case the connection failure is caused by a transient network outage. Suitable values are `SOURCE_RETRY_COUNT=3` | `MASTER_RETRY_COUNT=3` and `SOURCE_CONNECT_RETRY=10` | `MASTER_CONNECT_RETRY=10`, which make the replica retry the connection 3 times with 10-second intervals between.

You also need to set the source list for the replication channel, to specify the sources that are available for failover. You set and manage source lists using the `asynchronous_connection_failover_add_source` and `asynchronous_connection_failover_delete_source` functions to add and remove single replication source servers. To add and remove managed groups of servers, use the `asynchronous_connection_failover_add_managed` and `asynchronous_connection_failover_delete_managed` functions instead.

The functions name the relevant replication channel and specify the host name, port number, network namespace, and weighted priority (1-100, with 100 being the highest priority) of a MySQL instance to add to or delete from the channel's source list. For a managed group, you also specify the type of managed service (currently only Group Replication is available), and the identifier of the managed group (for Group Replication, this is the value of the `group_replication_group_name` system variable). When you add a managed group, you only need to add one group member, and the replica automatically adds the rest from the current group membership. When you delete a managed group, you delete the entire group together.

In MySQL 8.0.22, the asynchronous connection failover mechanism is activated following the failure of the replica's connection to the source, and it issues a `START REPLICA` statement to attempt to connect to a new source. In this release, the connection fails over if the replication receiver thread stops due to the source stopping or due to a network failure. The connection does not fail over in any other situations, such as when the replication threads are stopped by a `STOP REPLICA` statement.

From MySQL 8.0.23, the asynchronous connection failover mechanism also fails over the connection if another available server on the source list has a higher priority (weight) setting. This feature ensures that the replica stays connected to the most suitable source server at all times, and it applies to both managed groups and single (non-managed) servers. For a managed group, a source's weight is assigned depending on whether it is a primary or a secondary server. So assuming that you set up the managed group to give a higher weight to a primary and a lower weight to a secondary, when the primary changes, the higher weight is assigned to the new primary, so the replica changes over the connection to it. The asynchronous connection failover mechanism additionally changes connection if the currently connected managed source server leaves the managed group, or is no longer in the majority in the managed group.

When failing over a connection, the source with the highest priority (weight) setting among the alternative sources listed in the source list for the channel is chosen for the first connection attempt. The replica checks first that it can connect to the source server, or in the case of a managed group, that the source server has `ONLINE` status in the group (not `RECOVERING` or unavailable). If the highest weighted source is not available, the replica tries with all the listed sources in descending order of weight, then starts again from the highest weighted source. If multiple sources have the same weight, the replica orders them randomly. If the replica needs to start working through the list again, it includes and retries the source to which the original connection failure occurred.

The source lists are stored in the `mysql.replication_asynchronous_connection_failover` and `mysql.replication_asynchronous_connection_failover_managed` tables, and can be viewed in the Performance Schema tables `replication_asynchronous_connection_failover` and `replication_asynchronous_connection_failover_managed`. The replica uses a monitor thread to track the membership of managed groups and update the source list (`thread/sql/replica_monitor`). The setting for the `SOURCE_CONNECTION_AUTO_FAILOVER` option of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement, and the source list, are transferred to a clone of the replica during a remote cloning operation.

17.4.9.2 Asynchronous Connection Failover for Replicas

From MySQL 8.0.27, asynchronous connection failover for replicas is automatically activated for a replication channel on a Group Replication primary when you set `SOURCE_CONNECTION_AUTO_FAILOVER=1` on the `CHANGE REPLICATION SOURCE TO` statement for the channel. The feature is designed for a group of senders and a group of receivers to keep synchronized with each other even when some members are temporarily unavailable. When the feature is active and correctly configured, if the primary that is replicating goes offline or into an error state, the new primary starts replication on the same channel when it is elected. The new primary uses the source list for the channel to select the source with the highest priority (weight) setting, which might not be the same as the original source.

To configure this feature, the replication channel and the replication user account and password for the channel must be set up on all the member servers in the replication group, and on any new

joining members. Ensure that the SOURCE_RETRY_COUNT and SOURCE_CONNECT_RETRY settings are set to minimal numbers that just allow a few retry attempts, for example 3 and 10. You can set up the replication channel using the CHANGE REPLICATION SOURCE TO statement, or if the new servers are provisioned using MySQL's clone functionality, this all happens automatically. The SOURCE_CONNECTION_AUTO_FAILOVER setting for the channel is broadcast to group members from the primary when they join. If you later disable SOURCE_CONNECTION_AUTO_FAILOVER for the channel on the primary, this is also broadcast to the secondary servers, and they change the status of the channel to match.

Asynchronous connection failover for replicas is activated and deactivated using the Group Replication member action mysql_start_failover_channels_if_primary, which is enabled by default. You can disable it for the whole group by disabling that member action on the primary, using the group_replication_disable_member_action function, as in this example:

```
mysql> SELECT group_replication_disable_member_action("mysql_start_failover_channels_if_primary", "AFTE
```

The function can only be changed on a primary, and must be enabled or disabled for the whole group, so you cannot have some members providing failover and others not. When the mysql_start_failover_channels_if_primary member action is disabled, the channel does not need to be configured on secondary members, but if the primary goes offline or into an error state, replication stops for the channel. Note that if there is more than one channel with SOURCE_CONNECTION_AUTO_FAILOVER=1 , the member action covers all the channels, so they cannot be individually enabled and disabled by that method. Set SOURCE_CONNECTION_AUTO_FAILOVER=0 on the primary to disable an individual channel.

The source list for a channel with SOURCE_CONNECTION_AUTO_FAILOVER=1 is broadcast to all group members when they join, and also when it changes. This is the case whether the sources are a managed group for which the membership is updated automatically, or whether they are added or changed manually using the asynchronous_connection_failover_add_source(), asynchronous_connection_failover_delete_source(), asynchronous_connection_failover_add_managed() or asynchronous_connection_failover_delete_managed() functions. All group members receive the current source list as recorded in the mysql.replication_asynchronous_connection_failover and mysql.replication_asynchronous_connection_failover_managed tables. Because the sources do not have to be in a managed group, you can set up the function to synchronize a group of receivers with one or more alternative standalone senders, or even a single sender. However, a standalone replica that is not part of a replication group cannot use this feature.

17.4.10 Semisynchronous Replication

In addition to the built-in asynchronous replication, MySQL 8.0 supports an interface to semisynchronous replication that is implemented by plugins. This section discusses what semisynchronous replication is and how it works. The following sections cover the administrative interface to semisynchronous replication and how to install, configure, and monitor it.

MySQL replication by default is asynchronous. The source writes events to its binary log and replicas request them when they are ready. The source does not know whether or when a replica has retrieved and processed the transactions, and there is no guarantee that any event ever reaches any replica. With asynchronous replication, if the source crashes, transactions that it has committed might not have been transmitted to any replica. Failover from source to replica in this case might result in failover to a server that is missing transactions relative to the source.

With fully synchronous replication, when a source commits a transaction, all replicas have also committed the transaction before the source returns to the session that performed the transaction. Fully synchronous replication means failover from the source to any replica is possible at any time. The drawback of fully synchronous replication is that there might be a lot of delay to complete a transaction.

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.

Compared to asynchronous replication, semisynchronous replication provides improved data integrity, because when a commit returns successfully, it is known that the data exists in at least two places. Until a semisynchronous source receives acknowledgment from the required number of replicas, the transaction is on hold and not committed.

Compared to fully synchronous replication, semisynchronous replication is faster, because it can be configured to balance your requirements for data integrity (the number of replicas acknowledging receipt of the transaction) with the speed of commits, which are slower due to the need to wait for replicas.

Important

With semisynchronous replication, if the source crashes and a failover to a replica is carried out, the failed source should not be reused as the replication source, and should be discarded. It could have transactions that were not acknowledged by any replica, which were therefore not committed before the failover.

If your goal is to implement a fault-tolerant replication topology where all the servers receive the same transactions in the same order, and a server that crashes can rejoin the group and be brought up to date automatically, you can use Group Replication to achieve this. For information, see Chapter 18, *Group Replication*.

The performance impact of semisynchronous replication compared to asynchronous replication is the tradeoff for increased data integrity. The amount of slowdown is at least the TCP/IP roundtrip time to send the commit to the replica and wait for the acknowledgment of receipt by the replica. This means that semisynchronous replication works best for close servers communicating over fast networks, and worst for distant servers communicating over slow networks. Semisynchronous replication also places a rate limit on busy sessions by constraining the speed at which binary log events can be sent from source to replica. When one user is too busy, this slows it down, which can be useful in some deployment situations.

Semisynchronous replication between a source and its replicas operates as follows:

- A replica indicates whether it is semisynchronous-capable when it connects to the source.
- If semisynchronous replication is enabled on the source side and there is at least one semisynchronous replica, a thread that performs a transaction commit on the source blocks and waits until at least one semisynchronous replica acknowledges that it has received all events for the transaction, or until a timeout occurs.
- The replica acknowledges receipt of a transaction's events only after the events have been written to its relay log and flushed to disk.
- If a timeout occurs without any replica having acknowledged the transaction, the source reverts to asynchronous replication. When at least one semisynchronous replica catches up, the source returns to semisynchronous replication.
- Semisynchronous replication must be enabled on both the source and replica sides. If semisynchronous replication is disabled on the source, or enabled on the source but on no replicas, the source uses asynchronous replication.

While the source is blocking (waiting for acknowledgment from a replica), it does not return to the session that performed the transaction. When the block ends, the source returns to the session,

which then can proceed to execute other statements. At this point, the transaction has committed on the source side, and receipt of its events has been acknowledged by at least one replica. The number of replica acknowledgments the source must receive per transaction before returning to the session is configurable, and defaults to one acknowledgement (see Section 17.4.10.2, “Configuring Semisynchronous Replication”).

Blocking also occurs after rollbacks that are written to the binary log, which occurs when a transaction that modifies nontransactional tables is rolled back. The rolled-back transaction is logged even though it has no effect for transactional tables because the modifications to the nontransactional tables cannot be rolled back and must be sent to replicas.

For statements that do not occur in transactional context (that is, when no transaction has been started with `START TRANSACTION` or `SET autocommit = 0`), autocommit is enabled and each statement commits implicitly. With semisynchronous replication, the source blocks for each such statement, just as it does for explicit transaction commits.

By default, the source waits for replica acknowledgment of the transaction receipt after syncing the binary log to disk, but before committing the transaction to the storage engine. As an alternative, you can configure the source so that the source waits for replica acknowledgment after committing the transaction to the storage engine, using the `rpl_semi_sync_source_wait_point` or `rpl_semi_sync_master_wait_point` system variable. This setting affects the replication characteristics and the data that clients can see on the source. For more information, see Section 17.4.10.2, “Configuring Semisynchronous Replication”.

From MySQL 8.0.23, you can improve the performance of semisynchronous replication by enabling the system variables `replication_sender_observe_commit_only`, which limits callbacks, and `replication_optimize_for_static_plugin_config`, which adds shared locks and avoids unnecessary lock acquisitions. These settings help as the number of replicas increases, because contention for locks can slow down performance. Semisynchronous replication source servers can also get performance benefits from enabling these system variables, because they use the same locking mechanisms as the replicas.

17.4.10.1 Installing Semisynchronous Replication

Semisynchronous replication is implemented using plugins, which must be installed on the source and on the replicas to make semisynchronous replication available on the instances. There are different plugins for a source and for a replica. After a plugin has been installed, you control it by means of the system variables associated with it. These system variables are available only when the associated plugin has been installed.

This section describes how to install the semisynchronous replication plugins. For general information about installing plugins, see Section 5.6.1, “Installing and Uninstalling Plugins”.

To use semisynchronous replication, the following requirements must be satisfied:

- The capability of installing plugins requires a MySQL server that supports dynamic loading. To verify this, check that the value of the `have_dynamic_loading` system variable is `YES`. Binary distributions should support dynamic loading.
- Replication must already be working, see Section 17.1, “Configuring Replication”.
- There must not be multiple replication channels configured. Semisynchronous replication is only compatible with the default replication channel. See Section 17.2.2, “Replication Channels”.

From MySQL 8.0.26, new versions of the plugins that implement semisynchronous replication, one for the source server and one for the replica, are supplied. The new plugins replace the terms “master” and “slave” with “source” and “replica” in system variables and status variables, and you can install these versions instead of the old ones. You cannot have both the new and the old version of the relevant plugin installed on an instance. If you use the new version of the plugins, the new system variables and status variables are available but the old ones are not. If you use the old version of the plugins, the old system variables and status variables are available but the new ones are not.

The file name suffix for the plugin library files differs per platform (for example, `.so` for Unix and Unix-like systems, and `.dll` for Windows). The plugin and library file names are as follows:

- Source server, old terminology: `rpl_semi_sync_master` plugin (`semisync_master.so` or `semisync_master.dll` library)
- Source server, new terminology (from MySQL 8.0.26): `rpl_semi_sync_source` plugin (`semisync_source.so` or `semisync_source.dll` library)
- Replica, old terminology: `rpl_semi_sync_slave` plugin (`semisync_slave.so` or `semisync_slave.dll` library)
- Replica, new terminology (from MySQL 8.0.26): `rpl_semi_sync_replica` plugin (`semisync_replica.so` or `semisync_replica.dll` library)

To be usable by a source or replica server, the appropriate plugin library file must be located in the MySQL plugin directory (the directory named by the `plugin_dir` system variable). If necessary, configure the plugin directory location by setting the value of `plugin_dir` at server startup. The source plugin library file must be present in the plugin directory of the source server. The replica plugin library file must be present in the plugin directory of each replica server.

To set up semisynchronous replication, use the following instructions. The `INSTALL PLUGIN`, `SET GLOBAL`, `STOP REPLICA`, and `START REPLICA` statements mentioned here require the `REPLICATION_SLAVE_ADMIN` privilege (or the deprecated `SUPER` privilege).

To load the plugins, use the `INSTALL PLUGIN` statement on the source and on each replica that is to be semisynchronous, adjusting the `.so` suffix for your platform as necessary.

On the source:

```
INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';

Or from MySQL 8.0.26:
INSTALL PLUGIN rpl_semi_sync_source SONAME 'semisync_source.so';
```

On each replica:

```
INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.so';

Or from MySQL 8.0.26:
INSTALL PLUGIN rpl_semi_sync_replica SONAME 'semisync_replica.so';
```

If an attempt to install a plugin results in an error on Linux similar to that shown here, you must install `libimf`:

```
mysql> INSTALL PLUGIN rpl_semi_sync_source SONAME 'semisync_source.so';
ERROR 1126 (HY000): Can't open shared library
'/usr/local/mysql/lib/plugin/semisync_source.so'
(errno: 22 libimf.so: cannot open shared object file:
No such file or directory)
```

You can obtain `libimf` from https://dev.mysql.com/downloads/os-linux.html.

To verify plugin installation, examine the Information Schema `PLUGINS` table or use the `SHOW PLUGINS` statement (see Section 5.6.2, “Obtaining Server Plugin Information”). For example:

```
mysql> SELECT PLUGIN_NAME, PLUGIN_STATUS
       FROM INFORMATION_SCHEMA.PLUGINS
       WHERE PLUGIN_NAME LIKE '%semi%';
+----------------------+---------------+
| PLUGIN_NAME          | PLUGIN_STATUS |
+----------------------+---------------+
| rpl_semi_sync_source | ACTIVE        |
+----------------------+---------------+
```

If a plugin fails to initialize, check the server error log for diagnostic messages.

After a semisynchronous replication plugin has been installed, it is disabled by default. The plugins must be enabled both on the source side and the replica side to enable semisynchronous replication. If only one side is enabled, replication is asynchronous. To enable the plugins, set the appropriate system variable either at runtime using `SET GLOBAL`, or at server startup on the command line or in an option file. For example:

```
On the source:
SET GLOBAL rpl_semi_sync_master_enabled = 1;

Or from MySQL 8.0.26 with the rpl_semi_sync_source plugin:
SET GLOBAL rpl_semi_sync_source_enabled = 1;
```

```
On each replica:
SET GLOBAL rpl_semi_sync_slave_enabled = 1;

Or from MySQL 8.0.26 with the rpl_semi_sync_replica plugin:
SET GLOBAL rpl_semi_sync_replica_enabled = 1;
```

If you enable semisynchronous replication on a replica at runtime, you must also start the replication I/O (receiver) thread (stopping it first if it is already running) to cause the replica to connect to the source and register as a semisynchronous replica:

```
STOP SLAVE IO_THREAD;
START SLAVE IO_THREAD;

Or from MySQL 8.0.22:
STOP REPLICA IO_THREAD;
START REPLICA IO_THREAD;
```

If the replication I/O (receiver) thread is already running and you do not restart it, the replica continues to use asynchronous replication.

A setting listed in an option file takes effect each time the server starts. For example, you can set the variables in `my.cnf` files on the source and replica servers as follows:

```
 On the source:

[mysqld]
rpl_semi_sync_master_enabled=1

Or from MySQL 8.0.26 with the rpl_semi_sync_source plugin:
rpl_semi_sync_source_enabled=1
```

```
 On each replica:

[mysqld]
rpl_semi_sync_slave_enabled=1

Or from MySQL 8.0.26 with the rpl_semi_sync_source plugin:
rpl_semi_sync_replica_enabled=1
```

You can configure the behavior of the semisynchronous replication plugins using the system variables that become available when you install the plugins. For information on key system variables, see Section 17.4.10.2, “Configuring Semisynchronous Replication”.

17.4.10.2 Configuring Semisynchronous Replication

When you install the source and replica plugins for semisynchronous replication (see Section 17.4.10.1, “Installing Semisynchronous Replication”), system variables become available to control plugin behavior.

To check the current values of the status variables for semisynchronous replication, use `SHOW VARIABLES`:

```
mysql> SHOW VARIABLES LIKE 'rpl_semi_sync%';
```

From MySQL 8.0.26, new versions of the source and replica plugins are supplied, which replace the terms “master” and “slave” with “source” and “replica” in system variables and status variables.

If you install the new `rpl_semi_sync_source` and `rpl_semi_sync_replica` plugins, the new system variables and status variables are available but the old ones are not. If you install the old `rpl_semi_sync_master` and `rpl_semi_sync_slave` plugins, the old system variables and status variables are available but the new ones are not. You cannot have both the new and the old version of the relevant plugin installed on an instance.

All the `rpl_semi_sync_xxx` system variables are described at Section 17.1.6.2, “Replication Source Options and Variables” and Section 17.1.6.3, “Replica Server Options and Variables”. Some key system variables are:

`rpl_semi_sync_source_enabled` or `rpl_semi_sync_master_enabled`: Controls whether semisynchronous replication is enabled on the source server. To enable or disable the plugin, set this variable to 1 or 0, respectively. The default is 0 (off).

`rpl_semi_sync_replica_enabled` or `rpl_semi_sync_slave_enabled`: Controls whether semisynchronous replication is enabled on the replica.

`rpl_semi_sync_source_timeout` or `rpl_semi_sync_master_timeout`: A value in milliseconds that controls how long the source waits on a commit for acknowledgment from a replica before timing out and reverting to asynchronous replication. The default value is 10000 (10 seconds).

`rpl_semi_sync_source_wait_for_replica_count` or `rpl_semi_sync_master_wait_for_slave_count`: Controls the number of replica acknowledgments the source must receive per transaction before returning to the session. The default is 1, meaning that the source only waits for one replica to acknowledge receipt of the transaction's events.

The `rpl_semi_sync_source_wait_point` or `rpl_semi_sync_master_wait_point` system variable controls the point at which a semisynchronous source server waits for replica acknowledgment of transaction receipt before returning a status to the client that committed the transaction. These values are permitted:

- `AFTER_SYNC` (the default): The source writes each transaction to its binary log and the replica, and syncs the binary log to disk. The source waits for replica acknowledgment of transaction receipt after the sync. Upon receiving acknowledgment, the source commits the transaction to the storage engine and returns a result to the client, which then can proceed.
- `AFTER_COMMIT`: The source writes each transaction to its binary log and the replica, syncs the binary log, and commits the transaction to the storage engine. The source waits for replica acknowledgment of transaction receipt after the commit. Upon receiving acknowledgment, the source returns a result to the client, which then can proceed.

The replication characteristics of these settings differ as follows:

- With `AFTER_SYNC`, all clients see the committed transaction at the same time, which is after it has been acknowledged by the replica and committed to the storage engine on the source. Thus, all clients see the same data on the source.

  In the event of source failure, all transactions committed on the source have been replicated to the replica (saved to its relay log). An unexpected exit of the source and failover to the replica is lossless because the replica is up to date. As noted above, the source should not be reused after the failover.

- With `AFTER_COMMIT`, the client issuing the transaction gets a return status only after the server commits to the storage engine and receives replica acknowledgment. After the commit and before replica acknowledgment, other clients can see the committed transaction before the committing client.

  If something goes wrong such that the replica does not process the transaction, then in the event of an unexpected source exit and failover to the replica, it is possible for such clients to see a loss of data relative to what they saw on the source.

From MySQL 8.0.23, you can improve the performance of semisynchronous replication by enabling the system variables `replication_sender_observe_commit_only`, which limits callbacks, and `replication_optimize_for_static_plugin_config`, which adds shared locks and avoids unnecessary lock acquisitions. These settings help as the number of replicas increases, because contention for locks can slow down performance. Semisynchronous replication source servers can also get performance benefits from enabling these system variables, because they use the same locking mechanisms as the replicas.

17.4.10.3 Semisynchronous Replication Monitoring

The plugins for semisynchronous replication expose a number of status variables that enable you to monitor their operation. To check the current values of the status variables, use `SHOW STATUS`:

```
mysql> SHOW STATUS LIKE 'Rpl_semi_sync%';
```

From MySQL 8.0.26, new versions of the source and replica plugins are supplied, which replace the terms “master” and “slave” with “source” and “replica” in system variables and status variables. If you install the new `rpl_semi_sync_source` and `rpl_semi_sync_replica` plugins, the new system variables and status variables are available but the old ones are not. If you install the old `rpl_semi_sync_master` and `rpl_semi_sync_slave` plugins, the old system variables and status variables are available but the new ones are not. You cannot have both the new and the old version of the relevant plugin installed on an instance.

All `Rpl_semi_sync_`*`xxx`* status variables are described at Section 5.1.10, “Server Status Variables”. Some examples are:

- `Rpl_semi_sync_source_clients` or `Rpl_semi_sync_master_clients`

  The number of semisynchronous replicas that are connected to the source server.

- `Rpl_semi_sync_source_status` or `Rpl_semi_sync_master_status`

  Whether semisynchronous replication currently is operational on the source server. The value is 1 if the plugin has been enabled and a commit acknowledgment has not occurred. It is 0 if the plugin is not enabled or the source has fallen back to asynchronous replication due to commit acknowledgment timeout.

- `Rpl_semi_sync_source_no_tx` or `Rpl_semi_sync_master_no_tx`

  The number of commits that were not acknowledged successfully by a replica.

- `Rpl_semi_sync_source_yes_tx` or `Rpl_semi_sync_master_yes_tx`

  The number of commits that were acknowledged successfully by a replica.

- `Rpl_semi_sync_replica_status` or `Rpl_semi_sync_slave_status`

  Whether semisynchronous replication currently is operational on the replica. This is 1 if the plugin has been enabled and the replication I/O (receiver) thread is running, 0 otherwise.

When the source switches between asynchronous or semisynchronous replication due to commit-blocking timeout or a replica catching up, it sets the value of the `Rpl_semi_sync_source_status` or `Rpl_semi_sync_master_status` status variable appropriately. Automatic fallback from semisynchronous to asynchronous replication on the source means that it is possible for the `rpl_semi_sync_source_enabled` or `rpl_semi_sync_master_enabled` system variable to have a value of 1 on the source side even when semisynchronous replication is in fact not operational at the moment. You can monitor the `Rpl_semi_sync_source_status` or `Rpl_semi_sync_master_status` status variable to determine whether the source currently is using asynchronous or semisynchronous replication.

17.4.11 Delayed Replication

MySQL supports delayed replication such that a replica server deliberately executes transactions later than the source by at least a specified amount of time. This section describes how to configure a replication delay on a replica, and how to monitor replication delay.

In MySQL 8.0, the method of delaying replication depends on two timestamps, `immediate_commit_timestamp` and `original_commit_timestamp` (see Replication Delay Timestamps). If all servers in the replication topology are running MySQL 8.0 or above, delayed replication is measured using these timestamps. If either the immediate source or replica is not using these timestamps, the implementation of delayed replication from MySQL 5.7 is used (see Delayed Replication). This section describes delayed replication between servers which are all using these timestamps.

The default replication delay is 0 seconds. Use a `CHANGE REPLICATION SOURCE TO SOURCE_DELAY=N` statement (from MySQL 8.0.23) or a `CHANGE MASTER TO MASTER_DELAY=N` statement (before MySQL 8.0.23) to set the delay to *`N`* seconds. A transaction received from the source is not executed until at least *`N`* seconds later than its commit on the immediate source. The delay happens per transaction (not event as in previous MySQL versions) and the actual delay is imposed only on `gtid_log_event` or `anonymous_gtid_log_event`. The other events in the transaction always follow these events without any waiting time imposed on them.

Note

`START REPLICA` and `STOP REPLICA` take effect immediately and ignore any delay. `RESET REPLICA` resets the delay to 0.

The `replication_applier_configuration` Performance Schema table contains the `DESIRED_DELAY` column which shows the delay configured using the `SOURCE_DELAY` | `MASTER_DELAY` option. The `replication_applier_status` Performance Schema table contains the `REMAINING_DELAY` column which shows the number of delay seconds remaining.

Delayed replication can be used for several purposes:

- To protect against user mistakes on the source. With a delay you can roll back a delayed replica to the time just before the mistake.
- To test how the system behaves when there is a lag. For example, in an application, a lag might be caused by a heavy load on the replica. However, it can be difficult to generate this load level. Delayed replication can simulate the lag without having to simulate the load. It can also be used to debug conditions related to a lagging replica.
- To inspect what the database looked like in the past, without having to reload a backup. For example, by configuring a replica with a delay of one week, if you then need to see what the database looked like before the last few days' worth of development, the delayed replica can be inspected.

Replication Delay Timestamps

MySQL 8.0 provides a new method for measuring delay (also referred to as replication lag) in replication topologies that depends on the following timestamps associated with the GTID of each transaction (instead of each event) written to the binary log.

- `original_commit_timestamp`: the number of microseconds since epoch when the transaction was written (committed) to the binary log of the original source.
- `immediate_commit_timestamp`: the number of microseconds since epoch when the transaction was written (committed) to the binary log of the immediate source.

The output of `mysqlbinlog` displays these timestamps in two formats, microseconds from epoch and also `TIMESTAMP` format, which is based on the user defined time zone for better readability. For example:

```
#170404 10:48:05 server id 1  end_log_pos 233 CRC32 0x016ce647     GTID    last_committed=0
\ sequence_number=1    original_committed_timestamp=1491299285661130    immediate_commit_timestamp=1491
# original_commit_timestamp=1491299285661130 (2017-04-04 10:48:05.661130 WEST)
# immediate_commit_timestamp=1491299285843771 (2017-04-04 10:48:05.843771 WEST)
 /*!80001 SET @@SESSION.original_commit_timestamp=1491299285661130*//*!*/;
   SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa:1'/*!*/;
# at 233
```

As a rule, the `original_commit_timestamp` is always the same on all replicas where the transaction is applied. In source-replica replication, the `original_commit_timestamp` of a transaction in the (original) source's binary log is always the same as its `immediate_commit_timestamp`. In the replica's relay log, the `original_commit_timestamp` and `immediate_commit_timestamp` of the transaction are the same as in the source's binary log; whereas in its own binary log, the transaction's `immediate_commit_timestamp` corresponds to when the replica committed the transaction.

In a Group Replication setup, when the original source is a member of a group, the `original_commit_timestamp` is generated when the transaction is ready to be committed. In other words, when it finished executing on the original source and its write set is ready to be sent to all members of the group for certification. When the original source is a server outside the group, the `original_commit_timestamp` is preserved. The same `original_commit_timestamp` for a particular transaction is replicated to all servers in the group, and to any replica outside the group that is replicating from a member. From MySQL 8.0.26, each recipient of the transaction also stores the local commit time in its binary log using `immediate_commit_timestamp`.

View change events, which are exclusive to Group Replication, are a special case. Transactions containing these events are generated by each group member but share the same GTID (so, they are not first executed in a source and then replicated to the group, but all members of the group execute and apply the same transaction). Before MySQL 8.0.26, these transactions have their `original_commit_timestamp` set to zero, and they appear this way in viewable output. From MySQL 8.0.26, for improved observability, group members set local timestamp values for transactions associated with view change events.

Monitoring Replication Delay

One of the most common ways to monitor replication delay (lag) in previous MySQL versions was by relying on the `Seconds_Behind_Master` field in the output of `SHOW REPLICA STATUS`. However, this metric is not suitable when using replication topologies more complex than the traditional source-replica setup, such as Group Replication. The addition of `immediate_commit_timestamp` and `original_commit_timestamp` to MySQL 8 provides a much finer degree of information about replication delay. The recommended method to monitor replication delay in a topology that supports these timestamps is using the following Performance Schema tables.

- `replication_connection_status`: current status of the connection to the source, provides information on the last and current transaction the connection thread queued into the relay log.
- `replication_applier_status_by_coordinator`: current status of the coordinator thread that only displays information when using a multithreaded replica, provides information on the last transaction buffered by the coordinator thread to a worker's queue, as well as the transaction it is currently buffering.
- `replication_applier_status_by_worker`: current status of the thread(s) applying transactions received from the source, provides information about the transactions applied by the replication SQL thread, or by each worker thread when using a multithreaded replica.

Using these tables you can monitor information about the last transaction the corresponding thread processed and the transaction that thread is currently processing. This information comprises:

- a transaction's GTID
- a transaction's `original_commit_timestamp` and `immediate_commit_timestamp`, retrieved from the replica's relay log

- the time a thread started processing a transaction
- for the last processed transaction, the time the thread finished processing it

In addition to the Performance Schema tables, the output of `SHOW REPLICA STATUS` has three fields that show:

- `SQL_Delay`: A nonnegative integer indicating the replication delay configured using `CHANGE REPLICATION SOURCE TO SOURCE_DELAY=N` (from MySQL 8.0.23) or `CHANGE MASTER TO MASTER_DELAY=N` (before MySQL 8.0.23), measured in seconds.
- `SQL_Remaining_Delay`: When `Replica_SQL_Running_State` is `Waiting until MASTER_DELAY seconds after master executed event`, this field contains an integer indicating the number of seconds left of the delay. At other times, this field is `NULL`.
- `Replica_SQL_Running_State`: A string indicating the state of the SQL thread (analogous to `Replica_IO_State`). The value is identical to the `State` value of the SQL thread as displayed by `SHOW PROCESSLIST`.

When the replication SQL thread is waiting for the delay to elapse before executing an event, `SHOW PROCESSLIST` displays its `State` value as `Waiting until MASTER_DELAY seconds after master executed event`.

17.5 Replication Notes and Tips

17.5.1 Replication Features and Issues

The following sections provide information about what is supported and what is not in MySQL replication, and about specific issues and situations that may occur when replicating certain statements.

Statement-based replication depends on compatibility at the SQL level between the source and replica. In other words, successful statement-based replication requires that any SQL features used be supported by both the source and the replica servers. If you use a feature on the source server that is available only in the current version of MySQL, you cannot replicate to a replica that uses an earlier version of MySQL. Such incompatibilities can also occur within a release series as well as between versions.

If you are planning to use statement-based replication between MySQL 8.0 and a previous MySQL release series, it is a good idea to consult the edition of the *MySQL Reference Manual* corresponding to the earlier release series for information regarding the replication characteristics of that series.

With MySQL's statement-based replication, there may be issues with replicating stored routines or triggers. You can avoid these issues by using MySQL's row-based replication instead. For a detailed list of issues, see Section 25.7, “Stored Program Binary Logging”. For more information about row-based logging and row-based replication, see Section 5.4.4.1, “Binary Logging Formats”, and Section 17.2.1, “Replication Formats”.

For additional information specific to replication and `InnoDB`, see Section 15.19, “InnoDB and MySQL Replication”. For information relating to replication with NDB Cluster, see Section 23.7, “NDB Cluster Replication”.

17.5.1.1 Replication and AUTO_INCREMENT

Statement-based replication of `AUTO_INCREMENT`, `LAST_INSERT_ID()`, and `TIMESTAMP` values is carried out subject to the following exceptions:

- A statement invoking a trigger or function that causes an update to an `AUTO_INCREMENT` column is not replicated correctly using statement-based replication. These statements are marked as unsafe. (Bug #45677)

- An `INSERT` into a table that has a composite primary key that includes an `AUTO_INCREMENT` column that is not the first column of this composite key is not safe for statement-based logging or replication. These statements are marked as unsafe. (Bug #11754117, Bug #45670)

  This issue does not affect tables using the `InnoDB` storage engine, since an `InnoDB` table with an AUTO_INCREMENT column requires at least one key where the auto-increment column is the only or leftmost column.

- Adding an `AUTO_INCREMENT` column to a table with `ALTER TABLE` might not produce the same ordering of the rows on the replica and the source. This occurs because the order in which the rows are numbered depends on the specific storage engine used for the table and the order in which the rows were inserted. If it is important to have the same order on the source and replica, the rows must be ordered before assigning an `AUTO_INCREMENT` number. Assuming that you want to add an `AUTO_INCREMENT` column to a table `t1` that has columns `col1` and `col2`, the following statements produce a new table `t2` identical to `t1` but with an `AUTO_INCREMENT` column:

  ```
  CREATE TABLE t2 LIKE t1;
  ALTER TABLE t2 ADD id INT AUTO_INCREMENT PRIMARY KEY;
  INSERT INTO t2 SELECT * FROM t1 ORDER BY col1, col2;
  ```

  Important

  To guarantee the same ordering on both source and replica, the `ORDER BY` clause must name *all* columns of `t1`.

  The instructions just given are subject to the limitations of `CREATE TABLE ... LIKE`: Foreign key definitions are ignored, as are the `DATA DIRECTORY` and `INDEX DIRECTORY` table options. If a table definition includes any of those characteristics, create `t2` using a `CREATE TABLE` statement that is identical to the one used to create `t1`, but with the addition of the `AUTO_INCREMENT` column.

  Regardless of the method used to create and populate the copy having the `AUTO_INCREMENT` column, the final step is to drop the original table and then rename the copy:

  ```
  DROP t1;
  ALTER TABLE t2 RENAME t1;
  ```

  See also Section B.3.6.1, “Problems with ALTER TABLE”.

17.5.1.2 Replication and BLACKHOLE Tables

The `BLACKHOLE` storage engine accepts data but discards it and does not store it. When performing binary logging, all inserts to such tables are always logged, regardless of the logging format in use. Updates and deletes are handled differently depending on whether statement based or row based logging is in use. With the statement based logging format, all statements affecting `BLACKHOLE` tables are logged, but their effects ignored. When using row-based logging, updates and deletes to such tables are simply skipped—they are not written to the binary log. A warning is logged whenever this occurs.

For this reason we recommend when you replicate to tables using the `BLACKHOLE` storage engine that you have the `binlog_format` server variable set to `STATEMENT`, and not to either `ROW` or `MIXED`.

17.5.1.3 Replication and Character Sets

The following applies to replication between MySQL servers that use different character sets:

- If the source has databases with a character set different from the global `character_set_server` value, you should design your `CREATE TABLE` statements so that they do not implicitly rely on the database default character set. A good workaround is to state the character set and collation explicitly in `CREATE TABLE` statements.

17.5.1.4 Replication and CHECKSUM TABLE

`CHECKSUM TABLE` returns a checksum that is calculated row by row, using a method that depends on the table row storage format. The storage format is not guaranteed to remain the same between MySQL versions, so the checksum value might change following an upgrade.

17.5.1.5 Replication of CREATE SERVER, ALTER SERVER, and DROP SERVER

The statements `CREATE SERVER`, `ALTER SERVER`, and `DROP SERVER` are not written to the binary log, regardless of the binary logging format that is in use.

17.5.1.6 Replication of CREATE ... IF NOT EXISTS Statements

MySQL applies these rules when various `CREATE ... IF NOT EXISTS` statements are replicated:

- Every `CREATE DATABASE IF NOT EXISTS` statement is replicated, whether or not the database already exists on the source.
- Similarly, every `CREATE TABLE IF NOT EXISTS` statement without a `SELECT` is replicated, whether or not the table already exists on the source. This includes `CREATE TABLE IF NOT EXISTS ... LIKE`. Replication of `CREATE TABLE IF NOT EXISTS ... SELECT` follows somewhat different rules; see Section 17.5.1.7, “Replication of CREATE TABLE ... SELECT Statements”, for more information.
- `CREATE EVENT IF NOT EXISTS` is always replicated, whether or not the event named in the statement already exists on the source.
- `CREATE USER` is written to the binary log only if successful. If the statement includes `IF NOT EXISTS`, it is considered successful, and is logged as long as at least one user named in the statement is created; in such cases, the statement is logged as written; this includes references to existing users that were not created. See CREATE USER Binary Logging, for more information.
- (*MySQL 8.0.29 and later*:) `CREATE PROCEDURE IF NOT EXISTS`, `CREATE FUNCTION IF NOT EXISTS`, or `CREATE TRIGGER IF NOT EXISTS`, if successful, is written in its entirety to the binary log (including the `IF NOT EXISTS` clause), whether or not the statement raised a warning because the object (procedure, function, or trigger) already existed.

17.5.1.7 Replication of CREATE TABLE ... SELECT Statements

MySQL applies these rules when `CREATE TABLE ... SELECT` statements are replicated:

- `CREATE TABLE ... SELECT` always performs an implicit commit (Section 13.3.3, “Statements That Cause an Implicit Commit”).
- If the destination table does not exist, logging occurs as follows. It does not matter whether `IF NOT EXISTS` is present.
  - `STATEMENT` or `MIXED` format: The statement is logged as written.
  - `ROW` format: The statement is logged as a `CREATE TABLE` statement followed by a series of insert-row events.

    Prior to MySQL 8.0.21, the statement is logged as two transactions. As of MySQL 8.0.21, on storage engines that support atomic DDL, it is logged as one transaction. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.
- If the `CREATE TABLE ... SELECT` statement fails, nothing is logged. This includes the case that the destination table exists and `IF NOT EXISTS` is not given.
- If the destination table exists and `IF NOT EXISTS` is given, MySQL 8.0 ignores the statement completely; nothing is inserted or logged.

MySQL 8.0 does not allow a `CREATE TABLE ... SELECT` statement to make any changes in tables other than the table that is created by the statement.

17.5.1.8 Replication of CURRENT_USER()

The following statements support use of the `CURRENT_USER()` function to take the place of the name of, and possibly the host for, an affected user or a definer:

- `DROP USER`
- `RENAME USER`
- `GRANT`
- `REVOKE`
- `CREATE FUNCTION`
- `CREATE PROCEDURE`
- `CREATE TRIGGER`
- `CREATE EVENT`
- `CREATE VIEW`
- `ALTER EVENT`
- `ALTER VIEW`
- `SET PASSWORD`

When binary logging is enabled and `CURRENT_USER()` or `CURRENT_USER` is used as the definer in any of these statements, MySQL Server ensures that the statement is applied to the same user on both the source and the replica when the statement is replicated. In some cases, such as statements that change passwords, the function reference is expanded before it is written to the binary log, so that the statement includes the user name. For all other cases, the name of the current user on the source is replicated to the replica as metadata, and the replica applies the statement to the current user named in the metadata, rather than to the current user on the replica.

17.5.1.9 Replication with Differing Table Definitions on Source and Replica

Source and target tables for replication do not have to be identical. A table on the source can have more or fewer columns than the replica's copy of the table. In addition, corresponding table columns on the source and the replica can use different data types, subject to certain conditions.

Note

Replication between tables which are partitioned differently from one another is not supported. See Section 17.5.1.24, “Replication and Partitioning”.

In all cases where the source and target tables do not have identical definitions, the database and table names must be the same on both the source and the replica. Additional conditions are discussed, with examples, in the following two sections.

Replication with More Columns on Source or Replica

You can replicate a table from the source to the replica such that the source and replica copies of the table have differing numbers of columns, subject to the following conditions:

- Columns common to both versions of the table must be defined in the same order on the source and the replica. (This is true even if both tables have the same number of columns.)
- Columns common to both versions of the table must be defined before any additional columns.

This means that executing an ALTER TABLE statement on the replica where a new column is inserted into the table within the range of columns common to both tables causes replication to fail, as shown in the following example:

Suppose that a table t, existing on the source and the replica, is defined by the following CREATE TABLE statement:

```
CREATE TABLE t (
    c1 INT,
    c2 INT,
    c3 INT
);
```

Suppose that the ALTER TABLE statement shown here is executed on the replica:

```
ALTER TABLE t ADD COLUMN cnew1 INT AFTER c3;
```

The previous ALTER TABLE is permitted on the replica because the columns c1, c2, and c3 that are common to both versions of table t remain grouped together in both versions of the table, before any columns that differ.

However, the following ALTER TABLE statement cannot be executed on the replica without causing replication to break:

```
ALTER TABLE t ADD COLUMN cnew2 INT AFTER c2;
```

Replication fails after execution on the replica of the ALTER TABLE statement just shown, because the new column cnew2 comes between columns common to both versions of t.

- Each “extra” column in the version of the table having more columns must have a default value.

  A column's default value is determined by a number of factors, including its type, whether it is defined with a DEFAULT option, whether it is declared as NULL, and the server SQL mode in effect at the time of its creation; for more information, see Section 11.6, “Data Type Default Values”).

In addition, when the replica's copy of the table has more columns than the source's copy, each column common to the tables must use the same data type in both tables.

Examples. The following examples illustrate some valid and invalid table definitions:

More columns on the source. The following table definitions are valid and replicate correctly:

```
source> CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
replica>  CREATE TABLE t1 (c1 INT, c2 INT);
```

The following table definitions would raise an error because the definitions of the columns common to both versions of the table are in a different order on the replica than they are on the source:

```
source> CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
replica>  CREATE TABLE t1 (c2 INT, c1 INT);
```

The following table definitions would also raise an error because the definition of the extra column on the source appears before the definitions of the columns common to both versions of the table:

```
source> CREATE TABLE t1 (c3 INT, c1 INT, c2 INT);
replica>  CREATE TABLE t1 (c1 INT, c2 INT);
```

More columns on the replica. The following table definitions are valid and replicate correctly:

```
source> CREATE TABLE t1 (c1 INT, c2 INT);
replica>  CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
```

The following definitions raise an error because the columns common to both versions of the table are not defined in the same order on both the source and the replica:

```
source> CREATE TABLE t1 (c1 INT, c2 INT);
replica>  CREATE TABLE t1 (c2 INT, c1 INT, c3 INT);
```

The following table definitions also raise an error because the definition for the extra column in the replica's version of the table appears before the definitions for the columns which are common to both versions of the table:

```
source> CREATE TABLE t1 (c1 INT, c2 INT);
replica>  CREATE TABLE t1 (c3 INT, c1 INT, c2 INT);
```

The following table definitions fail because the replica's version of the table has additional columns compared to the source's version, and the two versions of the table use different data types for the common column `c2`:

```
source> CREATE TABLE t1 (c1 INT, c2 BIGINT);
replica>  CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
```

Replication of Columns Having Different Data Types

Corresponding columns on the source's and the replica's copies of the same table ideally should have the same data type. However, this is not always strictly enforced, as long as certain conditions are met.

It is usually possible to replicate from a column of a given data type to another column of the same type and same size or width, where applicable, or larger. For example, you can replicate from a `CHAR(10)` column to another `CHAR(10)`, or from a `CHAR(10)` column to a `CHAR(25)` column without any problems. In certain cases, it also possible to replicate from a column having one data type (on the source) to a column having a different data type (on the replica); when the data type of the source's version of the column is promoted to a type that is the same size or larger on the replica, this is known as *attribute promotion*.

Attribute promotion can be used with both statement-based and row-based replication, and is not dependent on the storage engine used by either the source or the replica. However, the choice of logging format does have an effect on the type conversions that are permitted; the particulars are discussed later in this section.

Important

Whether you use statement-based or row-based replication, the replica's copy of the table cannot contain more columns than the source's copy if you wish to employ attribute promotion.

Statement-based replication. When using statement-based replication, a simple rule of thumb to follow is, “If the statement run on the source would also execute successfully on the replica, it should also replicate successfully”. In other words, if the statement uses a value that is compatible with the type of a given column on the replica, the statement can be replicated. For example, you can insert any value that fits in a `TINYINT` column into a `BIGINT` column as well; it follows that, even if you change the type of a `TINYINT` column in the replica's copy of a table to `BIGINT`, any insert into that column on the source that succeeds should also succeed on the replica, since it is impossible to have a legal `TINYINT` value that is large enough to exceed a `BIGINT` column.

Row-based replication: attribute promotion and demotion. Row-based replication supports attribute promotion and demotion between smaller data types and larger types. It is also possible to specify whether or not to permit lossy (truncated) or non-lossy conversions of demoted column values, as explained later in this section.

Lossy and non-lossy conversions. In the event that the target type cannot represent the value being inserted, a decision must be made on how to handle the conversion. If we permit the conversion but truncate (or otherwise modify) the source value to achieve a “fit” in the target column, we make what is known as a *lossy conversion*. A conversion which does not require truncation or similar modifications to fit the source column value in the target column is a *non-lossy* conversion.

Type conversion modes. The global value of the system variable `replica_type_conversions` (from MySQL 8.0.26) or `slave_type_conversions` (before MySQL 8.0.26) controls the type conversion mode used on the replica. This variable takes a set of values from the following list, which describes the effects of each mode on the replica's type-conversion behavior:

ALL_LOSSY	In this mode, type conversions that would mean loss of information are permitted. This does not imply that non-lossy conversions are permitted, merely that only cases requiring either lossy conversions or no conversion at all are permitted; for example, enabling *only* this mode permits an `INT` column to be converted to `TINYINT` (a lossy conversion), but not a `TINYINT` column to an `INT` column (non-lossy). Attempting the latter conversion in this case would cause replication to stop with an error on the replica.
ALL_NON_LOSSY	This mode permits conversions that do not require truncation or other special handling of the source value; that is, it permits conversions where the target type has a wider range than the source type. Setting this mode has no bearing on whether lossy conversions are permitted; this is controlled with the `ALL_LOSSY` mode. If only `ALL_NON_LOSSY` is set, but not `ALL_LOSSY`, then attempting a conversion that would result in the loss of data (such as `INT` to `TINYINT`, or `CHAR(25)` to `VARCHAR(20)`) causes the replica to stop with an error.
ALL_LOSSY,ALL_NON_LOSSY	When this mode is set, all supported type conversions are permitted, whether or not they are lossy conversions.
ALL_SIGNED	Treat promoted integer types as signed values (the default behavior).
ALL_UNSIGNED	Treat promoted integer types as unsigned values.
ALL_SIGNED,ALL_UNSIGNED	Treat promoted integer types as signed if possible, otherwise as unsigned.
[*empty*]	When `replica_type_conversions` or `slave_type_conversions` is not set, no attribute promotion or demotion is permitted; this means that all columns in the source and target tables must be of the same types. This mode is the default.

When an integer type is promoted, its signedness is not preserved. By default, the replica treats all such values as signed. You can control this behavior using `ALL_SIGNED`, `ALL_UNSIGNED`, or both. `ALL_SIGNED` tells the replica to treat all promoted integer types as signed; `ALL_UNSIGNED` instructs it to treat these as unsigned. Specifying both causes the replica to treat the value as signed if possible, otherwise to treat it as unsigned; the order in which they are listed is not significant. Neither `ALL_SIGNED` nor `ALL_UNSIGNED` has any effect if at least one of `ALL_LOSSY` or `ALL_NONLOSSY` is not also used.

Changing the type conversion mode requires restarting the replica with the new `replica_type_conversions` or `slave_type_conversions` setting.

Supported conversions. Supported conversions between different but similar data types are shown in the following list:

- Between any of the integer types `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT`, and `BIGINT`.

This includes conversions between the signed and unsigned versions of these types.

Lossy conversions are made by truncating the source value to the maximum (or minimum) permitted by the target column. For ensuring non-lossy conversions when going from unsigned to signed types, the target column must be large enough to accommodate the range of values in the source column. For example, you can demote `TINYINT UNSIGNED` non-lossily to `SMALLINT`, but not to `TINYINT`.

- Between any of the decimal types `DECIMAL`, `FLOAT`, `DOUBLE`, and `NUMERIC`.

  `FLOAT` to `DOUBLE` is a non-lossy conversion; `DOUBLE` to `FLOAT` can only be handled lossily. A conversion from `DECIMAL(M,D)` to `DECIMAL(M',D')` where `D' >= D` and `(M'-D') >= (M-D)` is non-lossy; for any case where `M' < M`, `D' < D`, or both, only a lossy conversion can be made.

  For any of the decimal types, if a value to be stored cannot be fit in the target type, the value is rounded down according to the rounding rules defined for the server elsewhere in the documentation. See Section 12.25.4, “Rounding Behavior”, for information about how this is done for decimal types.

- Between any of the string types `CHAR`, `VARCHAR`, and `TEXT`, including conversions between different widths.

  Conversion of a `CHAR`, `VARCHAR`, or `TEXT` to a `CHAR`, `VARCHAR`, or `TEXT` column the same size or larger is never lossy. Lossy conversion is handled by inserting only the first `N` characters of the string on the replica, where `N` is the width of the target column.

  Important

  Replication between columns using different character sets is not supported.

- Between any of the binary data types `BINARY`, `VARBINARY`, and `BLOB`, including conversions between different widths.

  Conversion of a `BINARY`, `VARBINARY`, or `BLOB` to a `BINARY`, `VARBINARY`, or `BLOB` column the same size or larger is never lossy. Lossy conversion is handled by inserting only the first `N` bytes of the string on the replica, where `N` is the width of the target column.

- Between any 2 `BIT` columns of any 2 sizes.

  When inserting a value from a `BIT(M)` column into a `BIT(M')` column, where `M' > M`, the most significant bits of the `BIT(M')` columns are cleared (set to zero) and the `M` bits of the `BIT(M)` value are set as the least significant bits of the `BIT(M')` column.

  When inserting a value from a source `BIT(M)` column into a target `BIT(M')` column, where `M' < M`, the maximum possible value for the `BIT(M')` column is assigned; in other words, an “all-set” value is assigned to the target column.

Conversions between types not in the previous list are not permitted.

17.5.1.10 Replication and DIRECTORY Table Options

If a `DATA DIRECTORY` or `INDEX DIRECTORY` table option is used in a `CREATE TABLE` statement on the source server, the table option is also used on the replica. This can cause problems if no corresponding directory exists in the replica host file system or if it exists but is not accessible to the replica MySQL server. This can be overridden by using the `NO_DIR_IN_CREATE` server SQL mode on the replica, which causes the replica to ignore the `DATA DIRECTORY` and `INDEX DIRECTORY` table options when replicating `CREATE TABLE` statements. The result is that `MyISAM` data and index files are created in the table's database directory.

For more information, see Section 5.1.11, “Server SQL Modes”.

17.5.1.11 Replication of DROP ... IF EXISTS Statements

The `DROP DATABASE IF EXISTS`, `DROP TABLE IF EXISTS`, and `DROP VIEW IF EXISTS` statements are always replicated, even if the database, table, or view to be dropped does not exist on the source. This is to ensure that the object to be dropped no longer exists on either the source or the replica, once the replica has caught up with the source.

`DROP ... IF EXISTS` statements for stored programs (stored procedures and functions, triggers, and events) are also replicated, even if the stored program to be dropped does not exist on the source.

17.5.1.12 Replication and Floating-Point Values

With statement-based replication, values are converted from decimal to binary. Because conversions between decimal and binary representations of them may be approximate, comparisons involving floating-point values are inexact. This is true for operations that use floating-point values explicitly, or that use values that are converted to floating-point implicitly. Comparisons of floating-point values might yield different results on source and replica servers due to differences in computer architecture, the compiler used to build MySQL, and so forth. See Section 12.3, “Type Conversion in Expression Evaluation”, and Section B.3.4.8, “Problems with Floating-Point Values”.

17.5.1.13 Replication and FLUSH

Some forms of the `FLUSH` statement are not logged because they could cause problems if replicated to a replica: `FLUSH LOGS` and `FLUSH TABLES WITH READ LOCK`. For a syntax example, see Section 13.7.8.3, “FLUSH Statement”. The `FLUSH TABLES`, `ANALYZE TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` statements are written to the binary log and thus replicated to replicas. This is not normally a problem because these statements do not modify table data.

However, this behavior can cause difficulties under certain circumstances. If you replicate the privilege tables in the `mysql` database and update those tables directly without using `GRANT`, you must issue a `FLUSH PRIVILEGES` on the replicas to put the new privileges into effect. In addition, if you use `FLUSH TABLES` when renaming a `MyISAM` table that is part of a `MERGE` table, you must issue `FLUSH TABLES` manually on the replicas. These statements are written to the binary log unless you specify `NO_WRITE_TO_BINLOG` or its alias `LOCAL`.

17.5.1.14 Replication and System Functions

Certain functions do not replicate well under some conditions:

- The `USER()`, `CURRENT_USER()` (or `CURRENT_USER`), `UUID()`, `VERSION()`, and `LOAD_FILE()` functions are replicated without change and thus do not work reliably on the replica unless row-based replication is enabled. (See Section 17.2.1, “Replication Formats”.)

  `USER()` and `CURRENT_USER()` are automatically replicated using row-based replication when using `MIXED` mode, and generate a warning in `STATEMENT` mode. (See also Section 17.5.1.8, “Replication of CURRENT_USER()”.) This is also true for `VERSION()` and `RAND()`.

- For `NOW()`, the binary log includes the timestamp. This means that the value *as returned by the call to this function on the source* is replicated to the replica. To avoid unexpected results when replicating between MySQL servers in different time zones, set the time zone on both source and replica. For more information, see Section 17.5.1.33, “Replication and Time Zones”.

  To explain the potential problems when replicating between servers which are in different time zones, suppose that the source is located in New York, the replica is located in Stockholm, and both servers are using local time. Suppose further that, on the source, you create a table `mytable`, perform an `INSERT` statement on this table, and then select from the table, as shown here:

```
mysql> CREATE TABLE mytable (mycol TEXT);
Query OK, 0 rows affected (0.06 sec)
```

```
mysql> INSERT INTO mytable VALUES ( NOW() );
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM mytable;
+---------------------+
| mycol               |
+---------------------+
| 2009-09-01 12:00:00 |
+---------------------+
1 row in set (0.00 sec)
```

Local time in Stockholm is 6 hours later than in New York; so, if you issue `SELECT NOW()` on the replica at that exact same instant, the value `2009-09-01 18:00:00` is returned. For this reason, if you select from the replica's copy of `mytable` after the `CREATE TABLE` and `INSERT` statements just shown have been replicated, you might expect `mycol` to contain the value `2009-09-01 18:00:00`. However, this is not the case; when you select from the replica's copy of `mytable`, you obtain exactly the same result as on the source:

```
mysql> SELECT * FROM mytable;
+---------------------+
| mycol               |
+---------------------+
| 2009-09-01 12:00:00 |
+---------------------+
1 row in set (0.00 sec)
```

Unlike `NOW()`, the `SYSDATE()` function is not replication-safe because it is not affected by `SET TIMESTAMP` statements in the binary log and is nondeterministic if statement-based logging is used. This is not a problem if row-based logging is used.

An alternative is to use the `--sysdate-is-now` option to cause `SYSDATE()` to be an alias for `NOW()`. This must be done on the source and the replica to work correctly. In such cases, a warning is still issued by this function, but can safely be ignored as long as `--sysdate-is-now` is used on both the source and the replica.

`SYSDATE()` is automatically replicated using row-based replication when using `MIXED` mode, and generates a warning in `STATEMENT` mode.

See also Section 17.5.1.33, “Replication and Time Zones”.

- *The following restriction applies to statement-based replication only, not to row-based replication.* The `GET_LOCK()`, `RELEASE_LOCK()`, `IS_FREE_LOCK()`, and `IS_USED_LOCK()` functions that handle user-level locks are replicated without the replica knowing the concurrency context on the source. Therefore, these functions should not be used to insert into a source table because the content on the replica would differ. For example, do not issue a statement such as `INSERT INTO mytable VALUES(GET_LOCK(...))`.

  These functions are automatically replicated using row-based replication when using `MIXED` mode, and generate a warning in `STATEMENT` mode.

As a workaround for the preceding limitations when statement-based replication is in effect, you can use the strategy of saving the problematic function result in a user variable and referring to the variable in a later statement. For example, the following single-row `INSERT` is problematic due to the reference to the `UUID()` function:

```
INSERT INTO t VALUES(UUID());
```

To work around the problem, do this instead:

```
SET @my_uuid = UUID();
INSERT INTO t VALUES(@my_uuid);
```

That sequence of statements replicates because the value of `@my_uuid` is stored in the binary log as a user-variable event prior to the `INSERT` statement and is available for use in the `INSERT`.

The same idea applies to multiple-row inserts, but is more cumbersome to use. For a two-row insert, you can do this:

```
SET @my_uuid1 = UUID(); @my_uuid2 = UUID();
INSERT INTO t VALUES(@my_uuid1),(@my_uuid2);
```

However, if the number of rows is large or unknown, the workaround is difficult or impracticable. For example, you cannot convert the following statement to one in which a given individual user variable is associated with each row:

```
INSERT INTO t2 SELECT UUID(), * FROM t1;
```

Within a stored function, `RAND()` replicates correctly as long as it is invoked only once during the execution of the function. (You can consider the function execution timestamp and random number seed as implicit inputs that are identical on the source and replica.)

The `FOUND_ROWS()` and `ROW_COUNT()` functions are not replicated reliably using statement-based replication. A workaround is to store the result of the function call in a user variable, and then use that in the `INSERT` statement. For example, if you wish to store the result in a table named `mytable`, you might normally do so like this:

```
SELECT SQL_CALC_FOUND_ROWS FROM mytable LIMIT 1;
INSERT INTO mytable VALUES( FOUND_ROWS() );
```

However, if you are replicating `mytable`, you should use `SELECT ... INTO`, and then store the variable in the table, like this:

```
SELECT SQL_CALC_FOUND_ROWS INTO @found_rows FROM mytable LIMIT 1;
INSERT INTO mytable VALUES(@found_rows);
```

In this way, the user variable is replicated as part of the context, and applied on the replica correctly.

These functions are automatically replicated using row-based replication when using `MIXED` mode, and generate a warning in `STATEMENT` mode. (Bug #12092, Bug #30244)

17.5.1.15 Replication and Fractional Seconds Support

MySQL 8.0 permits fractional seconds for `TIME`, `DATETIME`, and `TIMESTAMP` values, with up to microseconds (6 digits) precision. See Section 11.2.6, “Fractional Seconds in Time Values”.

17.5.1.16 Replication of Invoked Features

Replication of invoked features such as loadable functions and stored programs (stored procedures and functions, triggers, and events) provides the following characteristics:

- The effects of the feature are always replicated.
- The following statements are replicated using statement-based replication:
  - `CREATE EVENT`
  - `ALTER EVENT`
  - `DROP EVENT`
  - `CREATE PROCEDURE`
  - `DROP PROCEDURE`
  - `CREATE FUNCTION`
  - `DROP FUNCTION`
  - `CREATE TRIGGER`

- `DROP TRIGGER`

However, the *effects* of features created, modified, or dropped using these statements are replicated using row-based replication.

Note

Attempting to replicate invoked features using statement-based replication produces the warning `Statement is not safe to log in statement format`. For example, trying to replicate a loadable function with statement-based replication generates this warning because it currently cannot be determined by the MySQL server whether the function is deterministic. If you are absolutely certain that the invoked feature's effects are deterministic, you can safely disregard such warnings.

- In the case of `CREATE EVENT` and `ALTER EVENT`:
  - The status of the event is set to `SLAVESIDE_DISABLED` on the replica regardless of the state specified (this does not apply to `DROP EVENT`).
  - The source on which the event was created is identified on the replica by its server ID. The `ORIGINATOR` column in `INFORMATION_SCHEMA.EVENTS` stores this information. See Section 13.7.7.18, “SHOW EVENTS Statement”, for more information.
- The feature implementation resides on the replica in a renewable state so that if the source fails, the replica can be used as the source without loss of event processing.

To determine whether there are any scheduled events on a MySQL server that were created on a different server (that was acting as a source), query the Information Schema `EVENTS` table in a manner similar to what is shown here:

```
SELECT EVENT_SCHEMA, EVENT_NAME
    FROM INFORMATION_SCHEMA.EVENTS
    WHERE STATUS = 'SLAVESIDE_DISABLED';
```

Alternatively, you can use the `SHOW EVENTS` statement, like this:

```
SHOW EVENTS
    WHERE STATUS = 'SLAVESIDE_DISABLED';
```

When promoting a replica having such events to a source, you must enable each event using `ALTER EVENT` *`event_name`* `ENABLE`, where *`event_name`* is the name of the event.

If more than one source was involved in creating events on this replica, and you wish to identify events that were created only on a given source having the server ID *`source_id`*, modify the previous query on the `EVENTS` table to include the `ORIGINATOR` column, as shown here:

```
SELECT EVENT_SCHEMA, EVENT_NAME, ORIGINATOR
    FROM INFORMATION_SCHEMA.EVENTS
    WHERE STATUS = 'SLAVESIDE_DISABLED'
    AND   ORIGINATOR = 'source_id'
```

You can employ `ORIGINATOR` with the `SHOW EVENTS` statement in a similar fashion:

```
SHOW EVENTS
    WHERE STATUS = 'SLAVESIDE_DISABLED'
    AND   ORIGINATOR = 'source_id'
```

Before enabling events that were replicated from the source, you should disable the MySQL Event Scheduler on the replica (using a statement such as `SET GLOBAL event_scheduler = OFF;`), run any necessary `ALTER EVENT` statements, restart the server, then re-enable the Event Scheduler on the replica afterward (using a statement such as `SET GLOBAL event_scheduler = ON;`)-

If you later demote the new source back to being a replica, you must disable manually all events enabled by the `ALTER EVENT` statements. You can do this by storing in a separate table the event names from the `SELECT` statement shown previously, or using `ALTER EVENT` statements to rename the events with a common prefix such as `replicated_` to identify them.

If you rename the events, then when demoting this server back to being a replica, you can identify the events by querying the `EVENTS` table, as shown here:

```
SELECT CONCAT(EVENT_SCHEMA, '.', EVENT_NAME) AS 'Db.Event'
      FROM INFORMATION_SCHEMA.EVENTS
      WHERE INSTR(EVENT_NAME, 'replicated_') = 1;
```

17.5.1.17 Replication of JSON Documents

Before MySQL 8.0, an update to a JSON column was always written to the binary log as the complete document. In MySQL 8.0, it is possible to log partial updates to JSON documents (see Partial Updates of JSON Values), which is more efficient. The logging behavior depends on the format used, as described here:

Statement-based replication. JSON partial updates are always logged as partial updates. This cannot be disabled when using statement-based logging.

Row-based replication. JSON partial updates are not logged as such by default, but instead are logged as complete documents. To enable logging of partial updates, set `binlog_row_value_options=PARTIAL_JSON`. If a replication source has this variable set, partial updates received from that source are handled and applied by a replica regardless of the replica's own setting for the variable.

Servers running MySQL 8.0.2 or earlier do not recognize the log events used for JSON partial updates. For this reason, when replicating to such a server from a server running MySQL 8.0.3 or later, `binlog_row_value_options` must be disabled on the source by setting this variable to `''` (empty string). See the description of this variable for more information.

17.5.1.18 Replication and LIMIT

Statement-based replication of `LIMIT` clauses in `DELETE`, `UPDATE`, and `INSERT ... SELECT` statements is unsafe since the order of the rows affected is not defined. (Such statements can be replicated correctly with statement-based replication only if they also contain an `ORDER BY` clause.) When such a statement is encountered:

- When using `STATEMENT` mode, a warning that the statement is not safe for statement-based replication is now issued.

  When using `STATEMENT` mode, warnings are issued for DML statements containing `LIMIT` even when they also have an `ORDER BY` clause (and so are made deterministic). This is a known issue. (Bug #42851)

- When using `MIXED` mode, the statement is now automatically replicated using row-based mode.

17.5.1.19 Replication and LOAD DATA

`LOAD DATA` is considered unsafe for statement-based logging (see Section 17.2.1.3, “Determination of Safe and Unsafe Statements in Binary Logging”). When `binlog_format=MIXED` is set, the statement is logged in row-based format. When `binlog_format=STATEMENT` is set, note that `LOAD DATA` does not generate a warning, unlike other unsafe statements.

If you use `LOAD DATA` with `binlog_format=STATEMENT`, each replica on which the changes are to be applied creates a temporary file containing the data. The replica then uses a `LOAD DATA` statement to apply the changes. This temporary file is not encrypted, even if binary log encryption is active on the source, If encryption is required, use row-based or mixed binary logging format instead, for which replicas do not create the temporary file.

If a PRIVILEGE_CHECKS_USER account has been used to help secure the replication channel (see Section 17.3.3, “Replication Privilege Checks”), it is strongly recommended that you log LOAD DATA operations using row-based binary logging (binlog_format=ROW). If REQUIRE_ROW_FORMAT is set for the channel, row-based binary logging is required. With this logging format, the FILE privilege is not needed to execute the event, so do not give the PRIVILEGE_CHECKS_USER account this privilege. If you need to recover from a replication error involving a LOAD DATA INFILE operation logged in statement format, and the replicated event is trusted, you could grant the FILE privilege to the PRIVILEGE_CHECKS_USER account temporarily, removing it after the replicated event has been applied.

When mysqlbinlog reads log events for LOAD DATA statements logged in statement-based format, a generated local file is created in a temporary directory. These temporary files are not automatically removed by mysqlbinlog or any other MySQL program. If you do use LOAD DATA statements with statement-based binary logging, you should delete the temporary files yourself after you no longer need the statement log. For more information, see Section 4.6.9, “mysqlbinlog — Utility for Processing Binary Log Files”.

17.5.1.20 Replication and max_allowed_packet

max_allowed_packet sets an upper limit on the size of any single message between the MySQL server and clients, including replicas. If you are replicating large column values (such as might be found in TEXT or BLOB columns) and max_allowed_packet is too small on the source, the source fails with an error, and the replica shuts down the replication I/O (receiver) thread. If max_allowed_packet is too small on the replica, this also causes the replica to stop the I/O thread.

Row-based replication sends all columns and column values for updated rows from the source to the replica, including values of columns that were not actually changed by the update. This means that, when you are replicating large column values using row-based replication, you must take care to set max_allowed_packet large enough to accommodate the largest row in any table to be replicated, even if you are replicating updates only, or you are inserting only relatively small values.

On a multi-threaded replica (with replica_parallel_workers > 0 or slave_parallel_workers > 0), ensure that the system variable replica_pending_jobs_size_max or slave_pending_jobs_size_max is set to a value equal to or greater than the setting for the max_allowed_packet system variable on the source. The default setting for replica_pending_jobs_size_max or slave_pending_jobs_size_max, 128M, is twice the default setting for max_allowed_packet, which is 64M. max_allowed_packet limits the packet size that the source can send, but the addition of an event header can produce a binary log event exceeding this size. Also, in row-based replication, a single event can be significantly larger than the max_allowed_packet size, because the value of max_allowed_packet only limits each column of the table.

The replica actually accepts packets up to the limit set by its replica_max_allowed_packet or slave_max_allowed_packet setting, which default to the maximum setting of 1GB, to prevent a replication failure due to a large packet. However, the value of replica_pending_jobs_size_max or slave_pending_jobs_size_max controls the memory that is made available on the replica to hold incoming packets. The specified memory is shared among all the replica worker queues.

The value of replica_pending_jobs_size_max or slave_pending_jobs_size_max is a soft limit, and if an unusually large event (consisting of one or multiple packets) exceeds this size, the transaction is held until all the replica workers have empty queues, and then processed. All subsequent transactions are held until the large transaction has been completed. So although unusual events larger than replica_pending_jobs_size_max or slave_pending_jobs_size_max can be processed, the delay to clear the queues of all the replica workers and the wait to queue subsequent transactions can cause lag on the replica and decreased concurrency of the replica workers. replica_pending_jobs_size_max or slave_pending_jobs_size_max should therefore be set high enough to accommodate most expected event sizes.

17.5.1.21 Replication and MEMORY Tables

When a replication source server shuts down and restarts, its `MEMORY` tables become empty. To replicate this effect to replicas, the first time that the source uses a given `MEMORY` table after startup, it logs an event that notifies replicas that the table must be emptied by writing a `DELETE` or (from MySQL 8.0.22) `TRUNCATE TABLE` statement for that table to the binary log. This generated event is identifiable by a comment in the binary log, and if GTIDs are in use on the server, it has a GTID assigned. The statement is always logged in statement format, even if the binary logging format is set to `ROW`, and it is written even if `read_only` or `super_read_only` mode is set on the server. Note that the replica still has outdated data in a `MEMORY` table during the interval between the source's restart and its first use of the table. To avoid this interval when a direct query to the replica could return stale data, you can set the `init_file` system variable to name a file containing statements that populate the `MEMORY` table on the source at startup.

When a replica server shuts down and restarts, its `MEMORY` tables become empty. This causes the replica to be out of synchrony with the source and may lead to other failures or cause the replica to stop:

- Row-format updates and deletes received from the source may fail with `Can't find record in 'memory_table'`.
- Statements such as `INSERT INTO ... SELECT FROM memory_table` may insert a different set of rows on the source and replica.

The replica also writes a `DELETE` or (from MySQL 8.0.22) `TRUNCATE TABLE` statement to its own binary log, which is passed on to any downstream replicas, causing them to empty their own `MEMORY` tables.

The safe way to restart a replica that is replicating `MEMORY` tables is to first drop or delete all rows from the `MEMORY` tables on the source and wait until those changes have replicated to the replica. Then it is safe to restart the replica.

An alternative restart method may apply in some cases. When `binlog_format=ROW`, you can prevent the replica from stopping if you set `replica_exec_mode=IDEMPOTENT` (from MySQL 8.0.26) or `slave_exec_mode=IDEMPOTENT` (before MySQL 8.0.26) before you start the replica again. This allows the replica to continue to replicate, but its `MEMORY` tables still differ from those on the source. This is acceptable if the application logic is such that the contents of `MEMORY` tables can be safely lost (for example, if the `MEMORY` tables are used for caching). `replica_exec_mode=IDEMPOTENT` or `slave_exec_mode=IDEMPOTENT` applies globally to all tables, so it may hide other replication errors in non-`MEMORY` tables.

(The method just described is not applicable in NDB Cluster, where `replica_exec_mode` or `slave_exec_mode` is always `IDEMPOTENT`, and cannot be changed.)

The size of `MEMORY` tables is limited by the value of the `max_heap_table_size` system variable, which is not replicated (see Section 17.5.1.39, “Replication and Variables”). A change in `max_heap_table_size` takes effect for `MEMORY` tables that are created or updated using `ALTER TABLE ... ENGINE = MEMORY` or `TRUNCATE TABLE` following the change, or for all `MEMORY` tables following a server restart. If you increase the value of this variable on the source without doing so on the replica, it becomes possible for a table on the source to grow larger than its counterpart on the replica, leading to inserts that succeed on the source but fail on the replica with `Table is full` errors. This is a known issue (Bug #48666). In such cases, you must set the global value of `max_heap_table_size` on the replica as well as on the source, then restart replication. It is also recommended that you restart both the source and replica MySQL servers, to ensure that the new value takes complete (global) effect on each of them.

See Section 16.3, “The MEMORY Storage Engine”, for more information about `MEMORY` tables.

17.5.1.22 Replication of the mysql System Schema

Data modification statements made to tables in the `mysql` schema are replicated according to the value of `binlog_format`; if this value is `MIXED`, these statements are replicated using row-based

format. However, statements that would normally update this information indirectly—such `GRANT`, `REVOKE`, and statements manipulating triggers, stored routines, and views—are replicated to replicas using statement-based replication.

17.5.1.23 Replication and the Query Optimizer

It is possible for the data on the source and replica to become different if a statement is written in such a way that the data modification is nondeterministic; that is, left up the query optimizer. (In general, this is not a good practice, even outside of replication.) Examples of nondeterministic statements include `DELETE` or `UPDATE` statements that use `LIMIT` with no `ORDER BY` clause; see Section 17.5.1.18, "Replication and LIMIT", for a detailed discussion of these.

17.5.1.24 Replication and Partitioning

Replication is supported between partitioned tables as long as they use the same partitioning scheme and otherwise have the same structure, except where an exception is specifically allowed (see Section 17.5.1.9, "Replication with Differing Table Definitions on Source and Replica").

Replication between tables that have different partitioning is generally not supported. This because statements (such as `ALTER TABLE ... DROP PARTITION`) that act directly on partitions in such cases might produce different results on the source and the replica. In the case where a table is partitioned on the source but not on the replica, any statements that operate on partitions on the source's copy of the replica fail on the replica. When the replica's copy of the table is partitioned but the source's copy is not, statements that act directly on partitions cannot be run on the source without causing errors there. To avoid stopping replication or creating inconsistencies between the source and replica, always ensure that a table on the source and the corresponding replicated table on the replica are partitioned in the same way.

17.5.1.25 Replication and REPAIR TABLE

When used on a corrupted or otherwise damaged table, it is possible for the `REPAIR TABLE` statement to delete rows that cannot be recovered. However, any such modifications of table data performed by this statement are not replicated, which can cause source and replica to lose synchronization. For this reason, in the event that a table on the source becomes damaged and you use `REPAIR TABLE` to repair it, you should first stop replication (if it is still running) before using `REPAIR TABLE`, then afterward compare the source's and replica's copies of the table and be prepared to correct any discrepancies manually, before restarting replication.

17.5.1.26 Replication and Reserved Words

You can encounter problems when you attempt to replicate from an older source to a newer replica and you make use of identifiers on the source that are reserved words in the newer MySQL version running on the replica. For example, a table column named `rank` on a MySQL 5.7 source that is replicating to a MySQL 8.0 replica could cause a problem because `RANK` is a reserved word beginning in MySQL 8.0.

Replication can fail in such cases with Error 1064 `You have an error in your SQL syntax...`, *even if a database or table named using the reserved word or a table having a column named using the reserved word is excluded from replication*. This is due to the fact that each SQL event must be parsed by the replica prior to execution, so that the replica knows which database object or objects would be affected. Only after the event is parsed can the replica apply any filtering rules defined by `--replicate-do-db`, `--replicate-do-table`, `--replicate-ignore-db`, and `--replicate-ignore-table`.

To work around the problem of database, table, or column names on the source which would be regarded as reserved words by the replica, do one of the following:

- Use one or more `ALTER TABLE` statements on the source to change the names of any database objects where these names would be considered reserved words on the replica, and change any SQL statements that use the old names to use the new names instead.

- In any SQL statements using these database object names, write the names as quoted identifiers using backtick characters (`).

For listings of reserved words by MySQL version, see Keywords and Reserved Words in MySQL 8.0, in the *MySQL Server Version Reference*. For identifier quoting rules, see Section 9.2, “Schema Object Names”.

17.5.1.27 Replication and Row Searches

When a replica using row-based replication format applies an `UPDATE` or `DELETE` operation, it must search the relevant table for the matching rows. The algorithm used to carry out this process uses one of the table's indexes to carry out the search as the first choice, and a hash table if there are no suitable indexes.

The algorithm first assesses the available indexes in the table definition to see if there is any suitable index to use, and if there are multiple possibilities, which index is the best fit for the operation. The algorithm ignores the following types of index:

- Fulltext indexes.
- Hidden indexes.
- Generated indexes.
- Multi-valued indexes.
- Any index where the before-image of the row event does not contain all the columns of the index.

If there are no suitable indexes after ruling out these index types, the algorithm does not use an index for the search. If there are suitable indexes, one index is selected from the candidates, in the following priority order:

1. A primary key.
2. A unique index where every column in the index has a NOT NULL attribute. If more than one such index is available, the algorithm chooses the leftmost of these indexes.
3. Any other index. If more than one such index is available, the algorithm chooses the leftmost of these indexes.

If the algorithm is able to select a primary key or a unique index where every column in the index has a `NOT NULL` attribute, it uses this index to iterate over the rows in the `UPDATE` or `DELETE` operation. For each row in the row event, the algorithm looks up the row in the index to locate the table record to update. If no matching record is found, it returns the error `ER_KEY_NOT_FOUND` and stops the replication applier thread.

If the algorithm was not able to find a suitable index, or was only able to find an index that was non-unique or contained nulls, a hash table is used to assist in identifying the table records. The algorithm creates a hash table containing the rows in the `UPDATE` or `DELETE` operation, with the key as the full before-image of the row. The algorithm then iterates over all the records in the target table, using the selected index if it found one, or else performing a full table scan. For each record in the target table, it determines whether that row exists in the hash table. If the row is found in the hash table, the record in the target table is updated, and the row is deleted from the hash table. When all the records in the target table have been checked, the algorithm verifies whether the hash table is now empty. If there are any unmatched rows remaining in the hash table, the algorithm returns the error `ER_KEY_NOT_FOUND` and stops the replication applier thread.

The `slave_rows_search_algorithms` system variable was previously used to control how rows are searched for matches. The use of this system variable is now deprecated, because the default setting, which uses an index scan followed by a hash scan as described above, is optimal for performance and works correctly in all scenarios.

17.5.1.28 Replication and Source or Replica Shutdowns

It is safe to shut down a replication source server and restart it later. When a replica loses its connection to the source, the replica tries to reconnect immediately and retries periodically if that fails. The default is to retry every 60 seconds. This may be changed with the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). A replica also is able to deal with network connectivity outages. However, the replica notices the network outage only after receiving no data from the source for `replica_net_timeout` or `slave_net_timeout` seconds. If your outages are short, you may want to decrease the value of `replica_net_timeout` or `slave_net_timeout`. See Section 17.4.2, “Handling an Unexpected Halt of a Replica”.

An unclean shutdown (for example, a crash) on the source side can result in the source's binary log having a final position less than the most recent position read by the replica, due to the source's binary log file not being flushed. This can cause the replica not to be able to replicate when the source comes back up. Setting `sync_binlog=1` in the source server's `my.cnf` file helps to minimize this problem because it causes the source to flush its binary log more frequently. For the greatest possible durability and consistency in a replication setup using `InnoDB` with transactions, you should also set `innodb_flush_log_at_trx_commit=1`. With this setting, the contents of the `InnoDB` redo log buffer are written out to the log file at each transaction commit and the log file is flushed to disk. Note that the durability of transactions is still not guaranteed with this setting, because operating systems or disk hardware may tell `mysqld` that the flush-to-disk operation has taken place, even though it has not.

Shutting down a replica cleanly is safe because it keeps track of where it left off. However, be careful that the replica does not have temporary tables open; see Section 17.5.1.31, “Replication and Temporary Tables”. Unclean shutdowns might produce problems, especially if the disk cache was not flushed to disk before the problem occurred:

- For transactions, the replica commits and then updates `relay-log.info`. If an unexpected exit occurs between these two operations, relay log processing proceeds further than the information file indicates and the replica re-executes the events from the last transaction in the relay log after it has been restarted.
- A similar problem can occur if the replica updates `relay-log.info` but the server host crashes before the write has been flushed to disk. To minimize the chance of this occurring, set `sync_relay_log_info=1` in the replica `my.cnf` file. Setting `sync_relay_log_info` to 0 causes no writes to be forced to disk and the server relies on the operating system to flush the file from time to time.

The fault tolerance of your system for these types of problems is greatly increased if you have a good uninterruptible power supply.

17.5.1.29 Replica Errors During Replication

If a statement produces the same error (identical error code) on both the source and the replica, the error is logged, but replication continues.

If a statement produces different errors on the source and the replica, the replication SQL thread terminates, and the replica writes a message to its error log and waits for the database administrator to decide what to do about the error. This includes the case that a statement produces an error on the source or the replica, but not both. To address the issue, connect to the replica manually and determine the cause of the problem. `SHOW REPLICA STATUS` (or before MySQL 8.0.22, `SHOW SLAVE STATUS`) is useful for this. Then fix the problem and run `START REPLICA` (or before MySQL 8.0.22, `START SLAVE`). For example, you might need to create a nonexistent table before you can start the replica again.

Note

If a temporary error is recorded in the replica's error log, you do not necessarily have to take any action suggested in the quoted error message. Temporary

> errors should be handled by the client retrying the transaction. For example, if the replication SQL thread records a temporary error relating to a deadlock, you do not need to restart the transaction manually on the replica, unless the replication SQL thread subsequently terminates with a nontemporary error message.

If this error code validation behavior is not desirable, some or all errors can be masked out (ignored) with the `--slave-skip-errors` option.

For nontransactional storage engines such as `MyISAM`, it is possible to have a statement that only partially updates a table and returns an error code. This can happen, for example, on a multiple-row insert that has one row violating a key constraint, or if a long update statement is killed after updating some of the rows. If that happens on the source, the replica expects execution of the statement to result in the same error code. If it does not, the replication SQL thread stops as described previously.

If you are replicating between tables that use different storage engines on the source and replica, keep in mind that the same statement might produce a different error when run against one version of the table, but not the other, or might cause an error for one version of the table, but not the other. For example, since `MyISAM` ignores foreign key constraints, an `INSERT` or `UPDATE` statement accessing an `InnoDB` table on the source might cause a foreign key violation but the same statement performed on a `MyISAM` version of the same table on the replica would produce no such error, causing replication to stop.

Beginning with MySQL 8.0.31, replication filter rules are applied first, prior to making any privilege or row format checks, making it possible to filter out any transactions that fail validation; no checks are performed and thus no errors are raised for transactions which have been filtered out. This means that the replica can accept only that part of the database to which a given user has been granted access (as long as any updates to this part of the database use the row-based replication format). This may be helpful when performing an upgrade or when migrating to a system or application that uses administration tables to which the inbound replication user does not have access. See also Section 17.2.5, “How Servers Evaluate Replication Filtering Rules”.

17.5.1.30 Replication and Server SQL Mode

Using different server SQL mode settings on the source and the replica may cause the same `INSERT` statements to be handled differently on the source and the replica, leading the source and replica to diverge. For best results, you should always use the same server SQL mode on the source and on the replica. This advice applies whether you are using statement-based or row-based replication.

If you are replicating partitioned tables, using different SQL modes on the source and the replica is likely to cause issues. At a minimum, this is likely to cause the distribution of data among partitions to be different in the source's and replica's copies of a given table. It may also cause inserts into partitioned tables that succeed on the source to fail on the replica.

For more information, see Section 5.1.11, “Server SQL Modes”.

17.5.1.31 Replication and Temporary Tables

In MySQL 8.0, when `binlog_format` is set to `ROW` or `MIXED`, statements that exclusively use temporary tables are not logged on the source, and therefore the temporary tables are not replicated. Statements that involve a mix of temporary and nontemporary tables are logged on the source only for the operations on nontemporary tables, and the operations on temporary tables are not logged. This means that there are never any temporary tables on the replica to be lost in the event of an unplanned shutdown by the replica. For more information about row-based replication and temporary tables, see Row-based logging of temporary tables.

When `binlog_format` is set to `STATEMENT`, operations on temporary tables are logged on the source and replicated on the replica, provided that the statements involving temporary tables can be logged safely using statement-based format. In this situation, loss of replicated temporary tables on

the replica can be an issue. In statement-based replication mode, `CREATE TEMPORARY TABLE` and `DROP TEMPORARY TABLE` statements cannot be used inside a transaction, procedure, function, or trigger when GTIDs are in use on the server (that is, when the `enforce_gtid_consistency` system variable is set to `ON`). They can be used outside these contexts when GTIDs are in use, provided that `autocommit=1` is set.

Because of the differences in behavior between row-based or mixed replication mode and statement-based replication mode regarding temporary tables, you cannot switch the replication format at runtime, if the change applies to a context (global or session) that contains any open temporary tables. For more details, see the description of the `binlog_format` option.

Safe replica shutdown when using temporary tables. In statement-based replication mode, temporary tables are replicated except in the case where you stop the replica server (not just the replication threads) and you have replicated temporary tables that are open for use in updates that have not yet been executed on the replica. If you stop the replica server, the temporary tables needed by those updates are no longer available when the replica is restarted. To avoid this problem, do not shut down the replica while it has temporary tables open. Instead, use the following procedure:

1. Issue a `STOP REPLICA SQL_THREAD` statement.
2. Use `SHOW STATUS` to check the value of the `Replica_open_temp_tables` or `Slave_open_temp_tables` status variable.
3. If the value is not 0, restart the replication SQL thread with `START REPLICA SQL_THREAD` and repeat the procedure later.
4. When the value is 0, issue a `mysqladmin shutdown` command to stop the replica.

Temporary tables and replication options. By default, with statement-based replication, all temporary tables are replicated; this happens whether or not there are any matching `--replicate-do-db`, `--replicate-do-table`, or `--replicate-wild-do-table` options in effect. However, the `--replicate-ignore-table` and `--replicate-wild-ignore-table` options are honored for temporary tables. The exception is that to enable correct removal of temporary tables at the end of a session, a replica always replicates a `DROP TEMPORARY TABLE IF EXISTS` statement, regardless of any exclusion rules that would normally apply for the specified table.

A recommended practice when using statement-based replication is to designate a prefix for exclusive use in naming temporary tables that you do not want replicated, then employ a `--replicate-wild-ignore-table` option to match that prefix. For example, you might give all such tables names beginning with `norep` (such as `norepmytable`, `norepyourtable`, and so on), then use `--replicate-wild-ignore-table=norep%` to prevent them from being replicated.

17.5.1.32 Replication Retries and Timeouts

The global value of the system variable `replica_transaction_retries` (from MySQL 8.0.26) or `slave_transaction_retries` (before MySQL 8.0.26) sets the maximum number of times for applier threads on a single-threaded or multithreaded replica to automatically retry failed transactions before stopping. Transactions are automatically retried when the SQL thread fails to execute them because of an `InnoDB` deadlock, or when the transaction's execution time exceeds the `InnoDB` `innodb_lock_wait_timeout` value. If a transaction has a non-temporary error that prevents it from succeeding, it is not retried.

The default setting for `replica_transaction_retries` or `slave_transaction_retries` is 10, meaning that a failing transaction with an apparently temporary error is retried 10 times before the applier thread stops. Setting the variable to 0 disables automatic retrying of transactions. On a multithreaded replica, the specified number of transaction retries can take place on all applier threads of all channels. The Performance Schema table `replication_applier_status` shows the total number of transaction retries that took place on each replication channel, in the `COUNT_TRANSACTIONS_RETRIES` column.

The process of retrying transactions can cause lag on a replica or on a Group Replication group member, which can be configured as a single-threaded or multithreaded replica. The Performance Schema table `replication_applier_status_by_worker` shows detailed information on transaction retries by the applier threads on a single-threaded or multithreaded replica. This data includes timestamps showing how long it took the applier thread to apply the last transaction from start to finish (and when the transaction currently in progress was started), and how long this was after the commit on the original source and the immediate source. The data also shows the number of retries for the last transaction and the transaction currently in progress, and enables you to identify the transient errors that caused the transactions to be retried. You can use this information to see whether transaction retries are the cause of replication lag, and investigate the root cause of the failures that led to the retries.

17.5.1.33 Replication and Time Zones

By default, source and replica servers assume that they are in the same time zone. If you are replicating between servers in different time zones, the time zone must be set on both source and replica. Otherwise, statements depending on the local time on the source are not replicated properly, such as statements that use the `NOW()` or `FROM_UNIXTIME()` functions.

Verify that your combination of settings for the system time zone (`system_time_zone`), server current time zone (the global value of `time_zone`), and per-session time zones (the session value of `time_zone`) on the source and replica is producing the correct results. In particular, if the `time_zone` system variable is set to the value `SYSTEM`, indicating that the server time zone is the same as the system time zone, this can cause the source and replica to apply different time zones. For example, a source could write the following statement in the binary log:

```
SET @@session.time_zone='SYSTEM';
```

If this source and its replica have a different setting for their system time zones, this statement can produce unexpected results on the replica, even if the replica's global `time_zone` value has been set to match the source's. For an explanation of MySQL Server's time zone settings, and how to change them, see Section 5.1.15, “MySQL Server Time Zone Support”.

See also Section 17.5.1.14, “Replication and System Functions”.

17.5.1.34 Replication and Transaction Inconsistencies

Inconsistencies in the sequence of transactions that have been executed from the relay log can occur depending on your replication configuration. This section explains how to avoid inconsistencies and solve any problems they cause.

The following types of inconsistencies can exist:

- *Half-applied transactions*. A transaction which updates non-transactional tables has applied some but not all of its changes.
- *Gaps*. A gap in the externalized transaction set appears when, given an ordered sequence of transactions, a transaction that is later in the sequence is applied before some other transaction that is prior in the sequence. Gaps can only appear when using a multithreaded replica.

  To avoid gaps occurring on a multithreaded replica, set `replica_preserve_commit_order=ON` (from MySQL 8.0.26) or `slave_preserve_commit_order=ON` (before MySQL 8.0.26). From MySQL 8.0.27, this setting is the default, because all replicas are multithreaded by default from that release.

  Up to and including MySQL 8.0.18, preserving the commit order requires that binary logging (`log_bin`) and replica update logging (`log_replica_updates` or `log_slave_updates`) are also enabled, which are the default settings from MySQL 8.0. From MySQL 8.0.19, binary logging and replica update logging are not required on the replica to set `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON`, and can be disabled if wanted.

In all releases, setting `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` requires that `replica_parallel_type` (from MySQL 8.0.26) or `slave_parallel_type` (before MySQL 8.0.26) is set to `LOGICAL_CLOCK`. From MySQL 8.0.27 (but not for earlier releases), this is the default setting.

In some specific situations, as listed in the description for `replica_preserve_commit_order` and `slave_preserve_commit_order`, setting `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` cannot preserve commit order on the replica, so in these cases gaps might still appear in the sequence of transactions that have been executed from the replica's relay log.

Setting `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` does not prevent source binary log position lag.

- *Source binary log position lag*. Even in the absence of gaps, it is possible that transactions after `Exec_master_log_pos` have been applied. That is, all transactions up to point `N` have been applied, and no transactions after `N` have been applied, but `Exec_master_log_pos` has a value smaller than `N`. In this situation, `Exec_master_log_pos` is a “low-water mark” of the transactions applied, and lags behind the position of the most recently applied transaction. This can only happen on multithreaded replicas. Enabling `replica_preserve_commit_order` or `slave_preserve_commit_order` does not prevent source binary log position lag.

The following scenarios are relevant to the existence of half-applied transactions, gaps, and source binary log position lag:

1. While replication threads are running, there may be gaps and half-applied transactions.
2. `mysqld` shuts down. Both clean and unclean shutdown abort ongoing transactions and may leave gaps and half-applied transactions.
3. `KILL` of replication threads (the SQL thread when using a single-threaded replica, the coordinator thread when using a multithreaded replica). This aborts ongoing transactions and may leave gaps and half-applied transactions.
4. Error in applier threads. This may leave gaps. If the error is in a mixed transaction, that transaction is half-applied. When using a multithreaded replica, workers which have not received an error complete their queues, so it may take time to stop all threads.
5. `STOP REPLICA` when using a multithreaded replica. After issuing `STOP REPLICA`, the replica waits for any gaps to be filled and then updates `Exec_master_log_pos`. This ensures it never leaves gaps or source binary log position lag, unless any of the cases above applies, in other words, before `STOP REPLICA` completes, either an error happens, or another thread issues `KILL`, or the server restarts. In these cases, `STOP REPLICA` returns successfully.
6. If the last transaction in the relay log is only half-received and the multithreaded replica's coordinator thread has started to schedule the transaction to a worker, then `STOP REPLICA` waits up to 60 seconds for the transaction to be received. After this timeout, the coordinator gives up and aborts the transaction. If the transaction is mixed, it may be left half-completed.
7. `STOP REPLICA` when the ongoing transaction updates transactional tables only, in which case it is rolled back and `STOP REPLICA` stops immediately. If the ongoing transaction is mixed, `STOP REPLICA` waits up to 60 seconds for the transaction to complete. After this timeout, it aborts the transaction, so it may be left half-completed.

The global setting for the system variable `rpl_stop_replica_timeout` (from MySQL 8.0.26) or `rpl_stop_slave_timeout` (before MySQL 8.0.26) is unrelated to the process of stopping the replication threads. It only makes the client that issues `STOP REPLICA` return to the client, but the replication threads continue to try to stop.

If a replication channel has gaps, it has the following consequences:

1. The replica database is in a state that may never have existed on the source.

2. The field `Exec_master_log_pos` in `SHOW REPLICA STATUS` is only a “low-water mark”. In other words, transactions appearing before the position are guaranteed to have committed, but transactions after the position may have committed or not.

3. `CHANGE REPLICATION SOURCE TO` and `CHANGE MASTER TO` statements for that channel fail with an error, unless the applier threads are running and the statement only sets receiver options.

4. If `mysqld` is started with `--relay-log-recovery`, no recovery is done for that channel, and a warning is printed.

5. If `mysqldump` is used with `--dump-replica` or `--dump-slave`, it does not record the existence of gaps; thus it prints `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` with `RELAY_LOG_POS` set to the “low-water mark” position in `Exec_master_log_pos`.

   After applying the dump on another server, and starting the replication threads, transactions appearing after the position are replicated again. Note that this is harmless if GTIDs are enabled (however, in that case it is not recommended to use `--dump-replica` or `--dump-slave`).

If a replication channel has source binary log position lag but no gaps, cases 2 to 5 above apply, but case 1 does not.

The source binary log position information is persisted in binary format in the internal table `mysql.slave_worker_info`. `START REPLICA [SQL_THREAD]` always consults this information so that it applies only the correct transactions. This remains true even if `replica_parallel_workers` or `slave_parallel_workers` has been changed to 0 before `START REPLICA`, and even if `START REPLICA` is used with `UNTIL` clauses. `START REPLICA UNTIL SQL_AFTER_MTS_GAPS` only applies as many transactions as needed in order to fill in the gaps. If `START REPLICA` is used with `UNTIL` clauses that tell it to stop before it has consumed all the gaps, then it leaves remaining gaps.

Warning

`RESET REPLICA` removes the relay logs and resets the replication position. Thus issuing `RESET REPLICA` on a multithreaded replica with gaps means the replica loses any information about the gaps, without correcting the gaps. In this situation, if binary log position based replication is in use, the recovery process fails.

When GTID-based replication is in use (`GTID_MODE=ON`) and `SOURCE_AUTO_POSITION` is set for the replication channel using the `CHANGE REPLICATION SOURCE TO` statement, the old relay logs are not required for the recovery process. Instead, the replica can use GTID auto-positioning to calculate what transactions it is missing compared to the source. From MySQL 8.0.26, the process used for binary log position based replication to resolve gaps on a multithreaded replica is skipped entirely when GTID-based replication is in use. When the process is skipped, a `START REPLICA UNTIL SQL_AFTER_MTS_GAPS` statement behaves differently, and does not attempt to check for gaps in the sequence of transactions. You can also issue `CHANGE REPLICATION SOURCE TO` statements, which are not permitted on a non-GTID replica where there are gaps.

17.5.1.35 Replication and Transactions

Mixing transactional and nontransactional statements within the same transaction. In general, you should avoid transactions that update both transactional and nontransactional tables in a replication environment. You should also avoid using any statement that accesses both transactional (or temporary) and nontransactional tables and writes to any of them.

The server uses these rules for binary logging:

- If the initial statements in a transaction are nontransactional, they are written to the binary log immediately. The remaining statements in the transaction are cached and not written to the binary

log until the transaction is committed. (If the transaction is rolled back, the cached statements are written to the binary log only if they make nontransactional changes that cannot be rolled back. Otherwise, they are discarded.)

- For statement-based logging, logging of nontransactional statements is affected by the `binlog_direct_non_transactional_updates` system variable. When this variable is `OFF` (the default), logging is as just described. When this variable is `ON`, logging occurs immediately for nontransactional statements occurring anywhere in the transaction (not just initial nontransactional statements). Other statements are kept in the transaction cache and logged when the transaction commits. `binlog_direct_non_transactional_updates` has no effect for row-format or mixed-format binary logging.

Transactional, nontransactional, and mixed statements.
To apply those rules, the server considers a statement nontransactional if it changes only nontransactional tables, and transactional if it changes only transactional tables. A statement that references both nontransactional and transactional tables and updates *any* of the tables involved is considered a “mixed” statement. Mixed statements, like transactional statements, are cached and logged when the transaction commits.

A mixed statement that updates a transactional table is considered unsafe if the statement also performs either of the following actions:

- Updates or reads a temporary table
- Reads a nontransactional table and the transaction isolation level is less than REPEATABLE_READ

A mixed statement following the update of a transactional table within a transaction is considered unsafe if it performs either of the following actions:

- Updates any table and reads from any temporary table
- Updates a nontransactional table and `binlog_direct_non_transactional_updates` is OFF

For more information, see Section 17.2.1.3, “Determination of Safe and Unsafe Statements in Binary Logging”.

Note

A mixed statement is unrelated to mixed binary logging format.

In situations where transactions mix updates to transactional and nontransactional tables, the order of statements in the binary log is correct, and all needed statements are written to the binary log even in case of a `ROLLBACK`. However, when a second connection updates the nontransactional table before the first connection transaction is complete, statements can be logged out of order because the second connection update is written immediately after it is performed, regardless of the state of the transaction being performed by the first connection.

Using different storage engines on source and replica. It is possible to replicate transactional tables on the source using nontransactional tables on the replica. For example, you can replicate an `InnoDB` source table as a `MyISAM` replica table. However, if you do this, there are problems if the replica is stopped in the middle of a `BEGIN` ... `COMMIT` block because the replica restarts at the beginning of the `BEGIN` block.

It is also safe to replicate transactions from `MyISAM` tables on the source to transactional tables, such as tables that use the `InnoDB` storage engine, on the replica. In such cases, an `AUTOCOMMIT=1` statement issued on the source is replicated, thus enforcing `AUTOCOMMIT` mode on the replica.

When the storage engine type of the replica is nontransactional, transactions on the source that mix updates of transactional and nontransactional tables should be avoided because they can cause inconsistency of the data between the source transactional table and the replica nontransactional

table. That is, such transactions can lead to source storage engine-specific behavior with the possible effect of replication going out of synchrony. MySQL does not issue a warning about this, so extra care should be taken when replicating transactional tables from the source to nontransactional tables on the replicas.

Changing the binary logging format within transactions. The `binlog_format` and `binlog_checksum` system variables are read-only as long as a transaction is in progress.

Every transaction (including `autocommit` transactions) is recorded in the binary log as though it starts with a `BEGIN` statement, and ends with either a `COMMIT` or a `ROLLBACK` statement. This is even true for statements affecting tables that use a nontransactional storage engine (such as `MyISAM`).

Note

For restrictions that apply specifically to XA transactions, see Section 13.3.8.3, “Restrictions on XA Transactions”.

17.5.1.36 Replication and Triggers

With statement-based replication, triggers executed on the source also execute on the replica. With row-based replication, triggers executed on the source do not execute on the replica. Instead, the row changes on the source resulting from trigger execution are replicated and applied on the replica.

This behavior is by design. If under row-based replication the replica applied the triggers as well as the row changes caused by them, the changes would in effect be applied twice on the replica, leading to different data on the source and the replica.

If you want triggers to execute on both the source and the replica, perhaps because you have different triggers on the source and replica, you must use statement-based replication. However, to enable replica-side triggers, it is not necessary to use statement-based replication exclusively. It is sufficient to switch to statement-based replication only for those statements where you want this effect, and to use row-based replication the rest of the time.

A statement invoking a trigger (or function) that causes an update to an `AUTO_INCREMENT` column is not replicated correctly using statement-based replication. MySQL 8.0 marks such statements as unsafe. (Bug #45677)

A trigger can have triggers for different combinations of trigger event (`INSERT`, `UPDATE`, `DELETE`) and action time (`BEFORE`, `AFTER`), and multiple triggers are permitted.

For brevity, “multiple triggers” here is shorthand for “multiple triggers that have the same trigger event and action time.”

Upgrades. Multiple triggers are not supported in versions earlier than MySQL 5.7. If you upgrade servers in a replication topology that use a version earlier than MySQL 5.7, upgrade the replicas first and then upgrade the source. If an upgraded replication source server still has old replicas using MySQL versions that do not support multiple triggers, an error occurs on those replicas if a trigger is created on the source for a table that already has a trigger with the same trigger event and action time.

Downgrades. If you downgrade a server that supports multiple triggers to an older version that does not, the downgrade has these effects:

- For each table that has triggers, all trigger definitions are in the `.TRG` file for the table. However, if there are multiple triggers with the same trigger event and action time, the server executes only one of them when the trigger event occurs. For information about `.TRG` files, see the Table Trigger Storage section of the MySQL Server Doxygen documentation, available at https://dev.mysql.com/doc/index-other.html.
- If triggers for the table are added or dropped subsequent to the downgrade, the server rewrites the table's `.TRG` file. The rewritten file retains only one trigger per combination of trigger event and action time; the others are lost.

To avoid these problems, modify your triggers before downgrading. For each table that has multiple triggers per combination of trigger event and action time, convert each such set of triggers to a single trigger as follows:

1. For each trigger, create a stored routine that contains all the code in the trigger. Values accessed using `NEW` and `OLD` can be passed to the routine using parameters. If the trigger needs a single result value from the code, you can put the code in a stored function and have the function return the value. If the trigger needs multiple result values from the code, you can put the code in a stored procedure and return the values using `OUT` parameters.
2. Drop all triggers for the table.
3. Create one new trigger for the table that invokes the stored routines just created. The effect for this trigger is thus the same as the multiple triggers it replaces.

17.5.1.37 Replication and TRUNCATE TABLE

`TRUNCATE TABLE` is normally regarded as a DML statement, and so would be expected to be logged and replicated using row-based format when the binary logging mode is `ROW` or `MIXED`. However this caused issues when logging or replicating, in `STATEMENT` or `MIXED` mode, tables that used transactional storage engines such as `InnoDB` when the transaction isolation level was `READ COMMITTED` or `READ UNCOMMITTED`, which precludes statement-based logging.

`TRUNCATE TABLE` is treated for purposes of logging and replication as DDL rather than DML so that it can be logged and replicated as a statement. However, the effects of the statement as applicable to `InnoDB` and other transactional tables on replicas still follow the rules described in Section 13.1.37, “TRUNCATE TABLE Statement” governing such tables. (Bug #36763)

17.5.1.38 Replication and User Name Length

The maximum length for user names in MySQL 8.0 is 32 characters. Replication of user names longer than 16 characters fails when the replica runs a version of MySQL previous to 5.7, because those versions support only shorter user names. This occurs only when replicating from a newer source to an older replica, which is not a recommended configuration.

17.5.1.39 Replication and Variables

System variables are not replicated correctly when using `STATEMENT` mode, except for the following variables when they are used with session scope:

- `auto_increment_increment`
- `auto_increment_offset`
- `character_set_client`
- `character_set_connection`
- `character_set_database`
- `character_set_server`
- `collation_connection`
- `collation_database`
- `collation_server`
- `foreign_key_checks`
- `identity`
- `last_insert_id`

- `lc_time_names`
- `pseudo_thread_id`
- `sql_auto_is_null`
- `time_zone`
- `timestamp`
- `unique_checks`

When `MIXED` mode is used, the variables in the preceding list, when used with session scope, cause a switch from statement-based to row-based logging. See Section 5.4.4.3, “Mixed Binary Logging Format”.

`sql_mode` is also replicated except for the `NO_DIR_IN_CREATE` mode; the replica always preserves its own value for `NO_DIR_IN_CREATE`, regardless of changes to it on the source. This is true for all replication formats.

However, when `mysqlbinlog` parses a `SET @@sql_mode = mode` statement, the full *`mode`* value, including `NO_DIR_IN_CREATE`, is passed to the receiving server. For this reason, replication of such a statement may not be safe when `STATEMENT` mode is in use.

The `default_storage_engine` system variable is not replicated, regardless of the logging mode; this is intended to facilitate replication between different storage engines.

The `read_only` system variable is not replicated. In addition, the enabling this variable has different effects with regard to temporary tables, table locking, and the `SET PASSWORD` statement in different MySQL versions.

The `max_heap_table_size` system variable is not replicated. Increasing the value of this variable on the source without doing so on the replica can lead eventually to `Table is full` errors on the replica when trying to execute `INSERT` statements on a `MEMORY` table on the source that is thus permitted to grow larger than its counterpart on the replica. For more information, see Section 17.5.1.21, “Replication and MEMORY Tables”.

In statement-based replication, session variables are not replicated properly when used in statements that update tables. For example, the following sequence of statements does not insert the same data on the source and the replica:

```
SET max_join_size=1000;
INSERT INTO mytable VALUES(@@max_join_size);
```

This does not apply to the common sequence:

```
SET time_zone=...;
INSERT INTO mytable VALUES(CONVERT_TZ(..., ..., @@time_zone));
```

Replication of session variables is not a problem when row-based replication is being used, in which case, session variables are always replicated safely. See Section 17.2.1, “Replication Formats”.

The following session variables are written to the binary log and honored by the replica when parsing the binary log, regardless of the logging format:

- `sql_mode`
- `foreign_key_checks`
- `unique_checks`
- `character_set_client`
- `collation_connection`

- `collation_database`
- `collation_server`
- `sql_auto_is_null`

Important

Even though session variables relating to character sets and collations are written to the binary log, replication between different character sets is not supported.

To help reduce possible confusion, we recommend that you always use the same setting for the `lower_case_table_names` system variable on both source and replica, especially when you are running MySQL on platforms with case-sensitive file systems. The `lower_case_table_names` setting can only be configured when initializing the server.

17.5.1.40 Replication and Views

Views are always replicated to replicas. Views are filtered by their own name, not by the tables they refer to. This means that a view can be replicated to the replica even if the view contains a table that would normally be filtered out by `replication-ignore-table` rules. Care should therefore be taken to ensure that views do not replicate table data that would normally be filtered for security reasons.

Replication from a table to a same-named view is supported using statement-based logging, but not when using row-based logging. Trying to do so when row-based logging is in effect causes an error.

17.5.2 Replication Compatibility Between MySQL Versions

MySQL supports replication from one release series to the next higher release series. For example, you can replicate from a source running MySQL 5.6 to a replica running MySQL 5.7, from a source running MySQL 5.7 to a replica running MySQL 8.0, and so on. However, you might encounter difficulties when replicating from an older source to a newer replica if the source uses statements or relies on behavior no longer supported in the version of MySQL used on the replica. For example, foreign key names longer than 64 characters are no longer supported from MySQL 8.0.

The use of more than two MySQL Server versions is not supported in replication setups involving multiple sources, regardless of the number of source or replica MySQL servers. This restriction applies not only to release series, but to version numbers within the same release series as well. For example, if you are using a chained or circular replication setup, you cannot use MySQL 8.0.22, MySQL 8.0.24, and MySQL 8.0.28 concurrently, although you could use any two of these releases together.

Important

It is strongly recommended to use the most recent release available within a given MySQL release series because replication (and other) capabilities are continually being improved. It is also recommended to upgrade sources and replicas that use early releases of a release series of MySQL to GA (production) releases when the latter become available for that release series.

From MySQL 8.0.14, the server version is recorded in the binary log for each transaction for the server that originally committed the transaction (`original_server_version`), and for the server that is the immediate source of the current server in the replication topology (`immediate_server_version`).

Replication from newer sources to older replicas might be possible, but is generally not supported. This is due to a number of factors:

- **Binary log format changes.** The binary log format can change between major releases. While we attempt to maintain backward compatibility, this is not always possible. A source might also have

optional features enabled that are not understood by older replicas, such as binary log transaction compression, where the resulting compressed transaction payloads cannot be read by a replica at a release before MySQL 8.0.20.

This also has significant implications for upgrading replication servers; see Section 17.5.3, “Upgrading a Replication Topology”, for more information.

- For more information about row-based replication, see Section 17.2.1, “Replication Formats”.
- **SQL incompatibilities.** You cannot replicate from a newer source to an older replica using statement-based replication if the statements to be replicated use SQL features available on the source but not on the replica.

  However, if both the source and the replica support row-based replication, and there are no data definition statements to be replicated that depend on SQL features found on the source but not on the replica, you can use row-based replication to replicate the effects of data modification statements even if the DDL run on the source is not supported on the replica.

In MySQL 8.0.26, incompatible changes were made to replication instrumentation names, including the names of thread stages, containing the terms “master”, which is changed to “source”, “slave”, which is changed to “replica”, and “mts” (for “multithreaded slave”), which is changed to “mta” (for “multithreaded applier”). Monitoring tools that work with these instrumentation names might be impacted. If the incompatible changes have an impact for you, set the `terminology_use_previous` system variable to `BEFORE_8_0_26` to make MySQL Server use the old versions of the names for the objects specified in the previous list. This enables monitoring tools that rely on the old names to continue working until they can be updated to use the new names.

For more information on potential replication issues, see Section 17.5.1, “Replication Features and Issues”.

17.5.3 Upgrading a Replication Topology

When you upgrade servers that participate in a replication topology, you need to take into account each server's role in the topology and look out for issues specific to replication. For general information and instructions for upgrading a MySQL Server instance, see Section 2.10, “Upgrading MySQL”.

As explained in Section 17.5.2, “Replication Compatibility Between MySQL Versions”, MySQL supports replication from a source running one release series to a replica running the next higher release series, but does not support replication from a source running a later release to a replica running an earlier release. A replica at an earlier release might not have the required capability to process transactions that can be handled by the source at a later release. You must therefore upgrade all of the replicas in a replication topology to the target MySQL Server release, before you upgrade the source server to the target release. In this way you will never be in the situation where a replica still at the earlier release is attempting to handle transactions from a source at the later release.

In a replication topology where there are multiple sources (multi-source replication), the use of more than two MySQL Server versions is not supported, regardless of the number of source or replica MySQL servers. This restriction applies not only to release series, but to version numbers within the same release series as well. For example, you cannot use MySQL 8.0.22, MySQL 8.0.24, and MySQL 8.0.28 concurrently in such a setup, although you could use any two of these releases together.

If you need to downgrade the servers in a replication topology, the source must be downgraded before the replicas are downgraded. On the replicas, you must ensure that the binary log and relay log have been fully processed, and remove them before proceeding with the downgrade.

Behavior Changes Between Releases

Although this upgrade sequence is correct, it is possible to still encounter replication difficulties when replicating from a source at an earlier release that has not yet been upgraded, to a replica at a later release that has been upgraded. This can happen if the source uses statements or relies on behavior

that is no longer supported in the later release installed on the replica. You can use MySQL Shell's upgrade checker utility `util.checkForServerUpgrade()` to check MySQL 5.7 server instances or MySQL 8.0 server instances for upgrade to a GA MySQL 8.0 release. The utility identifies anything that needs to be fixed for that server instance so that it does not cause an issue after the upgrade, including features and behaviors that are no longer available in the later release. See Upgrade Checker Utility for information on the upgrade checker utility.

If you are upgrading an existing replication setup from a version of MySQL that does not support global transaction identifiers (GTIDs) to a version that does, only enable GTIDs on the source and the replicas when you have made sure that the setup meets all the requirements for GTID-based replication. See Section 17.1.3.4, “Setting Up Replication Using GTIDs” for information about converting binary log file position based replication setups to use GTID-based replication.

Changes affecting operations in strict SQL mode (`STRICT_TRANS_TABLES` or `STRICT_ALL_TABLES`) may result in replication failure on an upgraded replica. If you use statement-based logging (`binlog_format=STATEMENT`), if a replica is upgraded before the source, the source executes statements which succeed there but which may fail on the replica and so cause replication to stop. To deal with this, stop all new statements on the source and wait until the replicas catch up, then upgrade the replicas. Alternatively, if you cannot stop new statements, temporarily change to row-based logging on the source (`binlog_format=ROW`) and wait until all replicas have processed all binary logs produced up to the point of this change, then upgrade the replicas.

The default character set has changed from `latin1` to `utf8mb4` in MySQL 8.0. In a replicated setting, when upgrading from MySQL 5.7 to 8.0, it is advisable to change the default character set back to the character set used in MySQL 5.7 before upgrading. After the upgrade is completed, the default character set can be changed to `utf8mb4`. Assuming that the previous defaults were used, one way to preserve them is to start the server with these lines in the `my.cnf` file:

```
[mysqld]
character_set_server=latin1
collation_server=latin1_swedish_ci
```

Standard Upgrade Procedure

To upgrade a replication topology, follow the instructions in Section 2.10, “Upgrading MySQL” for each individual MySQL Server instance, using this overall procedure:

1. Upgrade the replicas first. On each replica instance:
   - Carry out the preliminary checks and steps described in Section 2.10.5, “Preparing Your Installation for Upgrade”.
   - Shut down MySQL Server.
   - Upgrade the MySQL Server binaries or packages.
   - Restart MySQL Server.
   - If you have upgraded to a release earlier than MySQL 8.0.16, invoke `mysql_upgrade` manually to upgrade the system tables and schemas. When the server is running with global transaction identifiers (GTIDs) enabled (`gtid_mode=ON`), do not enable binary logging by `mysql_upgrade` (so do not use the `--write-binlog` option). Then shut down and restart the server.
   - If you have upgraded to MySQL 8.0.16 or later, do not invoke `mysql_upgrade`. From that release, MySQL Server performs the entire MySQL upgrade procedure, disabling binary logging during the upgrade.
   - Restart replication using a `START REPLICA` or `START SLAVE` statement.
2. When all the replicas have been upgraded, follow the same steps to upgrade and restart the source server, with the exception of the `START REPLICA` or `START SLAVE` statement. If you made a

temporary change to row-based logging or to the default character set, you can revert the change now.

Upgrade Procedure With Table Repair Or Rebuild

Some upgrades may require that you drop and re-create database objects when you move from one MySQL series to the next. For example, collation changes might require that table indexes be rebuilt. Such operations, if necessary, are detailed at Section 2.10.4, “Changes in MySQL 8.0”. It is safest to perform these operations separately on the replicas and the source, and to disable replication of these operations from the source to the replica. To achieve this, use the following procedure:

1. Stop all the replicas and upgrade the binaries or packages. Restart them with the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable, so that they do not connect to the source. Perform any table repair or rebuilding operations needed to re-create database objects, such as use of `REPAIR TABLE` or `ALTER TABLE`, or dumping and reloading tables or triggers.
2. Disable the binary log on the source. To do this without restarting the source, execute a `SET sql_log_bin = OFF` statement. Alternatively, stop the source and restart it with the `--skip-log-bin` option. If you restart the source, you might also want to disallow client connections. For example, if all clients connect using TCP/IP, enable the `skip_networking` system variable when you restart the source.
3. With the binary log disabled, perform any table repair or rebuilding operations needed to re-create database objects. The binary log must be disabled during this step to prevent these operations from being logged and sent to the replicas later.
4. Re-enable the binary log on the source. If you set `sql_log_bin` to `OFF` earlier, execute a `SET sql_log_bin = ON` statement. If you restarted the source to disable the binary log, restart it without `--skip-log-bin`, and without enabling the `skip_networking` system variable so that clients and replicas can connect.
5. Restart the replicas, this time without the `--skip-slave-start` option or `skip_slave_start` system variable.

17.5.4 Troubleshooting Replication

If you have followed the instructions but your replication setup is not working, the first thing to do is *check the error log for messages*. Many users have lost time by not doing this soon enough after encountering problems.

If you cannot tell from the error log what the problem was, try the following techniques:

- Verify that the source has binary logging enabled by issuing a `SHOW MASTER STATUS` statement. Binary logging is enabled by default. If binary logging is enabled, `Position` is nonzero. If binary logging is not enabled, verify that you are not running the source with any settings that disable binary logging, such as the `--skip-log-bin` option.
- Verify that the `server_id` system variable was set at startup on both the source and replica and that the ID value is unique on each server.
- Verify that the replica is running. Use `SHOW REPLICA STATUS` to check whether the `Replica_IO_Running` and `Replica_SQL_Running` values are both `Yes`. If not, verify the options that were used when starting the replica server. For example, the `--skip-slave-start` command line option, or from MySQL 8.0.24, the `skip_slave_start` system variable, prevents the replication threads from starting until you issue a `START REPLICA` statement.
- If the replica is running, check whether it established a connection to the source. Use `SHOW PROCESSLIST`, find the I/O (receiver) and SQL (applier) threads and check their `State` column to see what they display. See Section 17.2.3, “Replication Threads”. If the receiver thread state says `Connecting to master`, check the following:

- Verify the privileges for the replication user on the source.
- Check that the host name of the source is correct and that you are using the correct port to connect to the source. The port used for replication is the same as used for client network communication (the default is `3306`). For the host name, ensure that the name resolves to the correct IP address.
- Check the configuration file to see whether the `skip_networking` system variable has been enabled on the source or replica to disable networking. If so, comment the setting or remove it.
- If the source has a firewall or IP filtering configuration, ensure that the network port being used for MySQL is not being filtered.
- Check that you can reach the source by using `ping` or `traceroute`/`tracert` to reach the host.

- If the replica was running previously but has stopped, the reason usually is that some statement that succeeded on the source failed on the replica. This should never happen if you have taken a proper snapshot of the source, and never modified the data on the replica outside of the replication threads. If the replica stops unexpectedly, it is a bug or you have encountered one of the known replication limitations described in Section 17.5.1, “Replication Features and Issues”. If it is a bug, see Section 17.5.5, “How to Report Replication Bugs or Problems”, for instructions on how to report it.
- If a statement that succeeded on the source refuses to run on the replica, try the following procedure if it is not feasible to do a full database resynchronization by deleting the replica's databases and copying a new snapshot from the source:

  1. Determine whether the affected table on the replica is different from the source table. Try to understand how this happened. Then make the replica's table identical to the source's and run `START REPLICA`.
  2. If the preceding step does not work or does not apply, try to understand whether it would be safe to make the update manually (if needed) and then ignore the next statement from the source.
  3. If you decide that the replica can skip the next statement from the source, issue the following statements:

     ```
     mysql> SET GLOBAL sql_slave_skip_counter = N;
     mysql> START SLAVE;

     Or from MySQL 8.0.26:
     mysql> SET GLOBAL sql_replica_skip_counter = N;
     mysql> START REPLICA;
     ```

     The value of *`N`* should be 1 if the next statement from the source does not use `AUTO_INCREMENT` or `LAST_INSERT_ID()`. Otherwise, the value should be 2. The reason for using a value of 2 for statements that use `AUTO_INCREMENT` or `LAST_INSERT_ID()` is that they take two events in the binary log of the source.

     See also SET GLOBAL sql_slave_skip_counter Syntax.

  4. If you are sure that the replica started out perfectly synchronized with the source, and that no one has updated the tables involved outside of the replication threads, then presumably the discrepancy is the result of a bug. If you are running the most recent version of MySQL, please report the problem. If you are running an older version, try upgrading to the latest production release to determine whether the problem persists.

17.5.5 How to Report Replication Bugs or Problems

When you have determined that there is no user error involved, and replication still either does not work at all or is unstable, it is time to send us a bug report. We need to obtain as much information as

possible from you to be able to track down the bug. Please spend some time and effort in preparing a good bug report.

If you have a repeatable test case that demonstrates the bug, please enter it into our bugs database using the instructions given in Section 1.5, “How to Report Bugs or Problems”. If you have a “phantom” problem (one that you cannot duplicate at will), use the following procedure:

1. Verify that no user error is involved. For example, if you update the replica outside of the replication threads, the data goes out of synchrony, and you can have unique key violations on updates. In this case, the replication thread stops and waits for you to clean up the tables manually to bring them into synchrony. *This is not a replication problem. It is a problem of outside interference causing replication to fail.*

2. Ensure that the replica is running with binary logging enabled (the `log_bin` system variable), and with the `--log-slave-updates` option enabled, which causes the replica to log the updates that it receives from the source into its own binary logs. These settings are the defaults.

3. Save all evidence before resetting the replication state. If we have no information or only sketchy information, it becomes difficult or impossible for us to track down the problem. The evidence you should collect is:

   - All binary log files from the source
   - All binary log files from the replica
   - The output of `SHOW MASTER STATUS` from the source at the time you discovered the problem
   - The output of `SHOW REPLICA STATUS` from the replica at the time you discovered the problem
   - Error logs from the source and the replica

4. Use `mysqlbinlog` to examine the binary logs. The following should be helpful to find the problem statement. *`log_file`* and *`log_pos`* are the `Master_Log_File` and `Read_Master_Log_Pos` values from `SHOW REPLICA STATUS`.

   ```
   $> mysqlbinlog --start-position=log_pos log_file | head
   ```

After you have collected the evidence for the problem, try to isolate it as a separate test case first. Then enter the problem with as much information as possible into our bugs database using the instructions at Section 1.5, “How to Report Bugs or Problems”.

Chapter 18 Group Replication

Table of Contents

This chapter explains MySQL Group Replication and how to install, configure and monitor groups. MySQL Group Replication enables you to create elastic, highly-available, fault-tolerant replication topologies.

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. Servers can leave and join the group and the view is updated accordingly. Sometimes servers can leave the group unexpectedly, in which case the failure detection mechanism detects this and notifies the group that the view has changed. This is all automatic.

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. Group Replication does not have an inbuilt method to do this. For example, see MySQL Router 8.0.

Group Replication is provided as a plugin to MySQL Server. You can follow the instructions in this chapter to configure the plugin on each of the server instances that you want in the group, start up the group, and monitor and administer the group. An alternative way to deploy a group of MySQL server instances is by using InnoDB Cluster.

Tip

To deploy multiple instances of MySQL, you can use InnoDB Cluster which enables you to easily administer a group of MySQL server instances in MySQL Shell. InnoDB Cluster wraps MySQL Group Replication in a programmatic environment that enables you easily deploy a cluster of MySQL instances to achieve high availability. In addition, InnoDB Cluster interfaces seamlessly with MySQL Router, which enables your applications to connect to the cluster without writing your own failover process. For similar use cases that do not require high availability, however, you can use InnoDB ReplicaSet. Installation instructions for MySQL Shell can be found here.

The chapter is structured as follows:

- Section 18.1, “Group Replication Background” provides an introduction to groups and how Group Replication works.
- Section 18.2, “Getting Started” explains how to configure multiple MySQL Server instances to create a group.
- Section 18.3, “Requirements and Limitations” explains architecture and setup requirements and limitations for Group Replication.
- Section 18.4, “Monitoring Group Replication” explains how to monitor a group.
- Section 18.5, “Group Replication Operations” explains how to work with a group.
- Section 18.6, “Group Replication Security” explains how to secure a group.
- Section 18.7, “Group Replication Performance and Troubleshooting” explains how to fine tune performance for a group.
- Section 18.8, “Upgrading Group Replication” explains how to upgrade a group.
- Section 18.9, “Group Replication System Variables” is a reference for the system variables specific to Group Replication.
- Section 18.10, “Frequently Asked Questions” provides answers to some technical questions about deploying and operating Group Replication.

18.1 Group Replication Background

This section provides background information on MySQL Group Replication.

The most common way to create a fault-tolerant system is to resort to making components redundant, in other words the component can be removed and the system should continue to operate as expected. This creates a set of challenges that raise complexity of such systems to a whole different level. Specifically, replicated databases have to deal with the fact that they require maintenance and administration of several servers instead of just one. Moreover, as servers are cooperating together to create the group several other classic distributed systems problems have to be dealt with, such as network partitioning or split brain scenarios.

Therefore, the ultimate challenge is to fuse the logic of the database and data replication with the logic of having several servers coordinated in a consistent and simple way. In other words, to have multiple servers agreeing on the state of the system and the data on each and every change that the system goes through. This can be summarized as having servers reaching agreement on each database state transition, so that they all progress as one single database or alternatively that they eventually converge to the same state. Meaning that they need to operate as a (distributed) state machine.

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. The group can operate in a single-primary mode with automatic primary election, where only one server accepts updates at a time. Alternatively, for more advanced users the group can be deployed in multi-primary mode, where all servers can accept updates, even if they are issued concurrently. This power comes at the expense of applications having to work around the limitations imposed by such deployments.

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. Servers can leave and join the group and the view is updated accordingly. Sometimes servers can leave the group unexpectedly, in which case the failure detection mechanism detects this and notifies the group that the view has changed. This is all automatic.

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. All these properties are key to creating a system which ensures that data is consistently replicated across the group of servers. At the very core of this technology lies an implementation of the Paxos algorithm. It acts as the group communication engine.

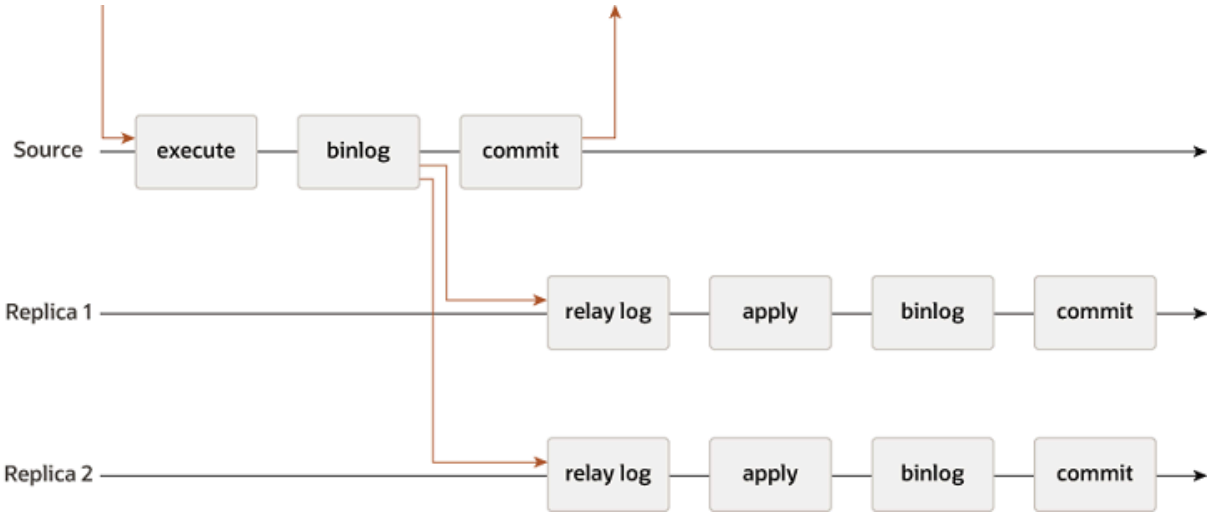
18.1.1 Replication Technologies

Before getting into the details of MySQL Group Replication, this section introduces some background concepts and an overview of how things work. This provides some context to help understand what is required for Group Replication and what the differences are between classic asynchronous MySQL Replication and Group Replication.

18.1.1.1 Source to Replica Replication

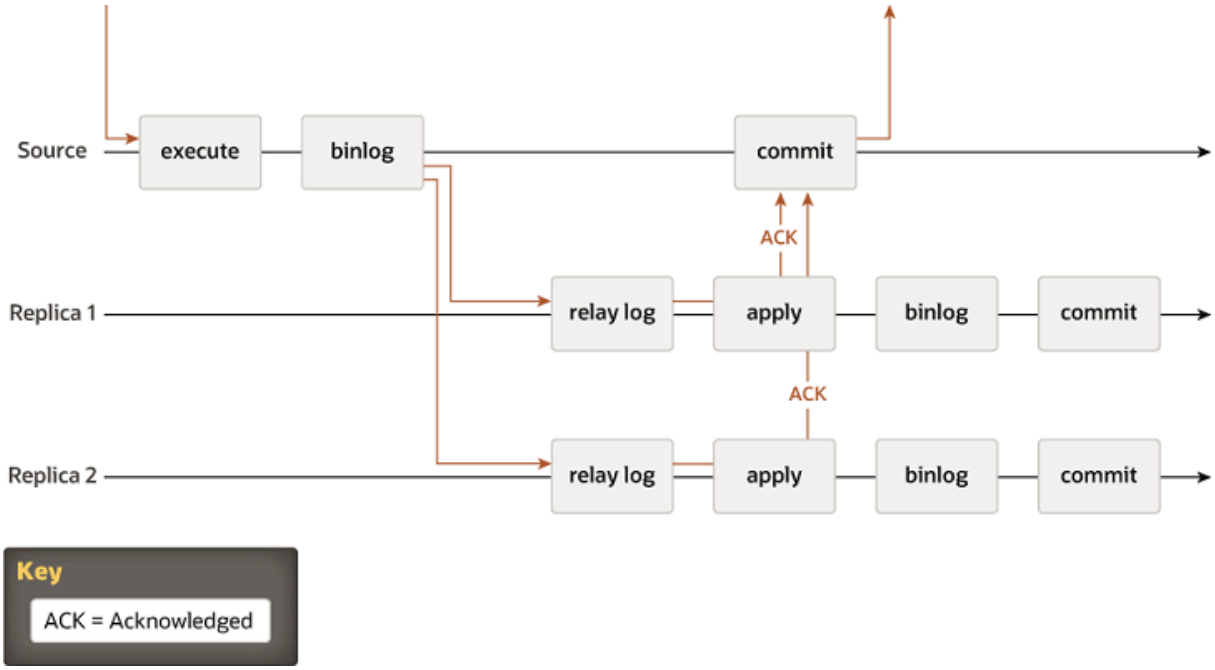
Traditional MySQL Replication provides a simple source to replica approach to replication. The source is the primary, and there are one or more replicas, which are secondaries. The source applies transactions, commits them and then they are later (thus asynchronously) sent to the replicas to be either re-executed (in statement-based replication) or applied (in row-based replication). It is a shared-nothing system, where all servers have a full copy of the data by default.

Figure 18.1 MySQL Asynchronous Replication



There is also semisynchronous replication, which adds one synchronization step to the protocol. This means that the primary waits, at apply time, for the secondary to acknowledge that it has *received* the transaction. Only then does the primary resume the commit operation.

Figure 18.2 MySQL Semisynchronous Replication



In the two pictures there is a diagram of the classic asynchronous MySQL Replication protocol (and its semisynchronous variant as well). The arrows between the different instances represent messages exchanged between servers or messages exchanged between servers and the client application.

18.1.1.2 Group Replication

Group Replication is a technique that can be used to implement fault-tolerant systems. The replication group is a set of servers that each have their own entire copy of the data (a shared-nothing replication scheme), and interact with each other through message passing. The communication layer provides a set of guarantees such as atomic message and total order message delivery. These are very powerful properties that translate into very useful abstractions that one can resort to build more advanced database replication solutions.

MySQL Group Replication builds on top of such properties and abstractions and implements a multi-source update everywhere replication protocol. A replication group is formed by multiple servers and each server in the group may execute transactions independently at any time. However, all read-write transactions commit only after they have been approved by the group. In other words, for any read-write transaction the group needs to decide whether it commits or not, so the commit operation is not a unilateral decision from the originating server. Read-only transactions need no coordination within the group and commit immediately.

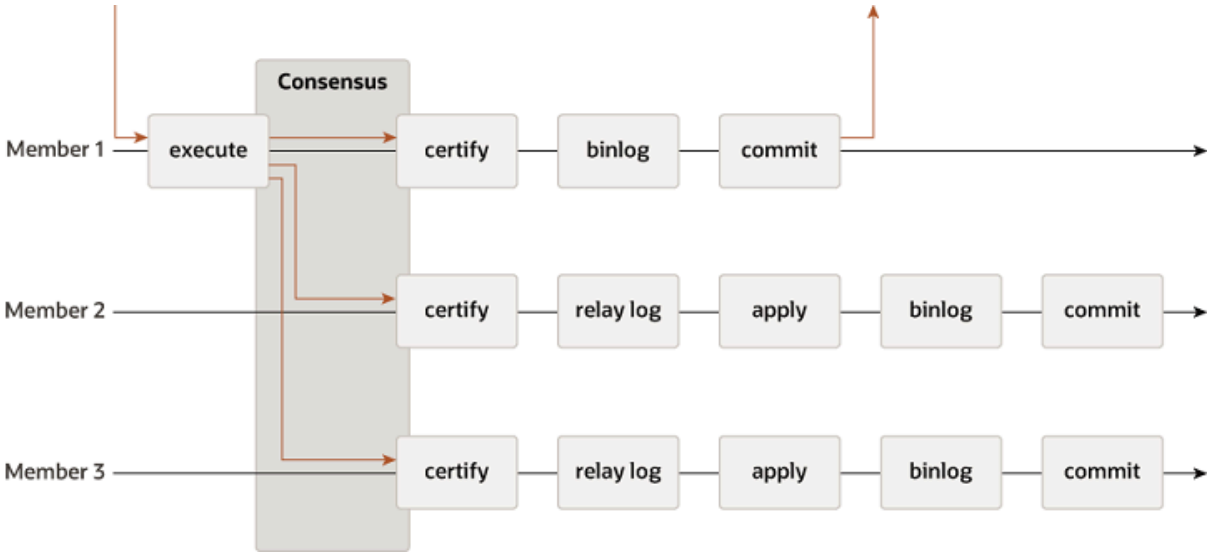
When a read-write transaction is ready to commit at the originating server, the server atomically broadcasts the write values (the rows that were changed) and the corresponding write set (the unique identifiers of the rows that were updated). Because the transaction is sent through an atomic broadcast, 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. For example, if t1 and t2 execute concurrently at different sites, both changing the same row, and t2 is ordered before t1, then t2 wins the conflict and t1 is rolled back. 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.

For applying and externalizing the certified transactions, Group Replication permits servers to deviate from the agreed order of the transactions if this does not break consistency and validity. 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. For example, in multi-primary mode, a local transaction might be externalized immediately following certification, although a remote transaction that is earlier in the global order has not yet been applied. This is permitted when the certification process has established that there is no conflict between the transactions. In single-primary mode, on the primary server, there is a small chance that concurrent, non-conflicting local transactions might be committed and externalized in a different order from the global order agreed by Group Replication. On the secondaries, which do not accept writes from clients, transactions are always committed and externalized in the agreed order.

The following figure depicts the MySQL Group Replication protocol and by comparing it to MySQL Replication (or even MySQL semisynchronous replication) you can see some differences. Some underlying consensus and Paxos related messages are missing from this picture for the sake of clarity.

Figure 18.3 MySQL Group Replication Protocol



18.1.2 Group Replication Use Cases

Group Replication enables you to create fault-tolerant systems with redundancy by replicating the system state to a set of servers. Even if some of the servers subsequently fail, as long it is not all or a majority, the system is still available. Depending on the number of servers which fail the group might have degraded performance or scalability, but it is still available. Server failures are isolated and independent. They are tracked by a group membership service which relies on a distributed failure detector that is able to signal when any servers leave the group, either voluntarily or due to an unexpected halt. There is a distributed recovery procedure to ensure that when servers join the group they are brought up to date automatically. There is no need for server failover, and the multi-source update everywhere nature ensures that even updates are not blocked in the event of a single server failure. To summarize, MySQL Group Replication guarantees that the database service is continuously available.

It is important to understand that although the database service is available, in the event of an unexpected server exit, those clients connected to it must be redirected, or failed over, to a different server. This is not something Group Replication attempts to resolve. A connector, load balancer, router, or some form of middleware are more suitable to deal with this issue. For example see MySQL Router 8.0.

To summarize, MySQL Group Replication provides a highly available, highly elastic, dependable MySQL service.

Tip

To deploy multiple instances of MySQL, you can use InnoDB Cluster which enables you to easily administer a group of MySQL server instances in MySQL Shell. InnoDB Cluster wraps MySQL Group Replication in a programmatic environment that enables you easily deploy a cluster of MySQL instances to achieve high availability. In addition, InnoDB Cluster interfaces seamlessly with MySQL Router, which enables your applications to connect to the cluster without writing your own failover process. For similar use cases that do not require high availability, however, you can use InnoDB ReplicaSet. Installation instructions for MySQL Shell can be found here.

Example Use Cases

The following examples are typical use cases for Group Replication.

- *Elastic Replication* - Environments that require a very fluid replication infrastructure, where the number of servers has to grow or shrink dynamically and with as few side-effects as possible. For instance, database services for the cloud.
- *Highly Available Shards* - Sharding is a popular approach to achieve write scale-out. Use MySQL Group Replication to implement highly available shards, where each shard maps to a replication group.
- *Alternative to asynchronous Source-Replica replication* - In certain situations, using a single source server makes it a single point of contention. Writing to an entire group may prove more scalable under certain circumstances.
- *Autonomic Systems* - Additionally, you can deploy MySQL Group Replication purely for the automation that is built into the replication protocol (described already in this and previous chapters).

18.1.3 Multi-Primary and Single-Primary Modes

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` 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. It is not possible to have members of the group deployed in different modes, for example one member configured in multi-primary mode while another member is in single-primary mode.

You cannot change the value of `group_replication_single_primary_mode` manually while Group Replication is running. From MySQL 8.0.13, you can use the `group_replication_switch_to_single_primary_mode()` and `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. In earlier releases, to change the group's mode you must stop Group Replication and change the value of `group_replication_single_primary_mode` on all members. Then carry out a full reboot of the group (a bootstrap by a server with `group_replication_bootstrap_group=ON`) to implement the change to the new operating configuration. You do not need to restart the servers.

Regardless of the deployed mode, Group Replication does not handle client-side failover. That must be handled by a middleware framework such as MySQL Router 8.0, a proxy, a connector, or the application itself.

18.1.3.1 Single-Primary Mode

In single-primary mode (`group_replication_single_primary_mode=ON`) 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`). 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.

In single-primary mode, Group Replication enforces that only a single server writes to the group, so compared to multi-primary mode, consistency checking can be less strict and DDL statements do not need to be handled with any extra care. The option `group_replication_enforce_update_everywhere_checks` enables or disables strict consistency checks for a group. When deploying in single-primary mode, or changing the group to single-primary mode, this system variable must be set to `OFF`.

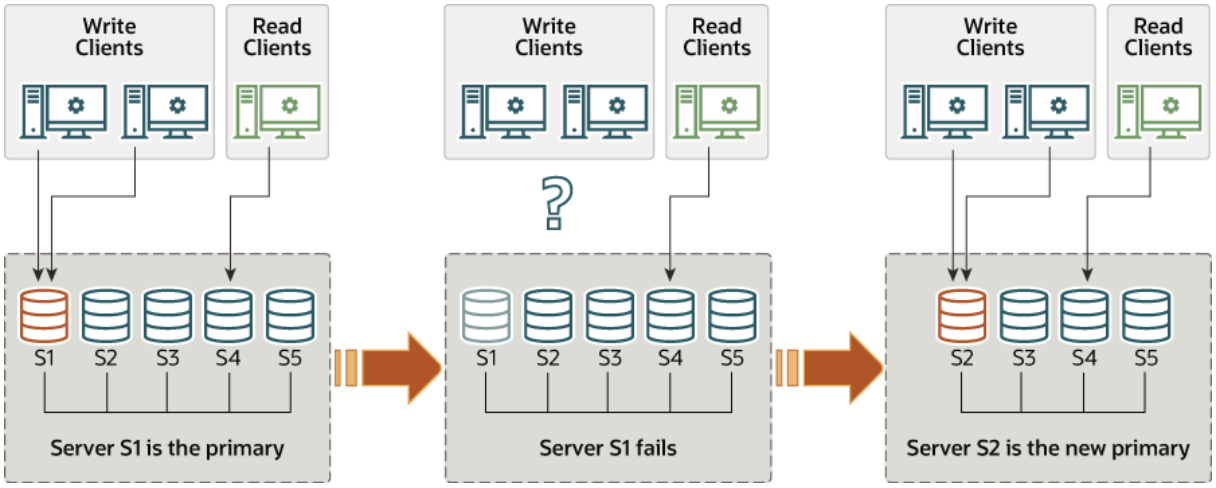
The member that is designated as the primary server can change in the following ways:

- 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()` function.

- If you use the `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.

The functions can only be used when all group members are running MySQL 8.0.13 or higher. When a new primary server is elected automatically or appointed manually, it is automatically set to read-write, and the other group members remain as secondaries, and as such, read-only. Figure 18.4, “New Primary Election” shows this process.

Figure 18.4 New Primary Election



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. For more information on flow control, see Section 18.7.2, “Flow Control”. From MySQL 8.0.14, you can also use the `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. For more information on transaction consistency, see Section 18.5.3, “Transaction Consistency Guarantees”. If flow control and transaction consistency guarantees are not used for a group, it is a good practice to wait for the new primary to apply its replication-related relay log before re-routing client applications to it.

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, in order, are as follows:

1. The first factor considered is which member or members are running the lowest MySQL Server version. If all group members are running MySQL 8.0.17 or higher, members are first ordered by the patch version of their release. If any members are running MySQL Server 5.7 or MySQL 8.0.16 or lower, members are first ordered by the major version of their release, and the patch version is ignored.

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` system variable on the member. If any member of the group is running MySQL Server 5.7, where this system variable was not available, this factor is ignored.

The `group_replication_member_weight` system variable specifies a number in the range 0-100. All members default to a weight of 50, so set a weight below this to lower their ordering, and a weight above it to increase their ordering. You can use this weighting function to prioritize the use of better hardware or to ensure failover to a specific member during scheduled maintenance of the primary.

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` system variable. The member with the lowest server UUID is chosen as the primary. This factor acts as a guaranteed and predictable tie-breaker so that all group members reach the same decision if it cannot be determined by any important factors.

Finding the Primary

To find out which server is currently the primary when deployed in single-primary mode, use the `MEMBER_ROLE` column in the `performance_schema.replication_group_members` table. For example:

```
mysql> SELECT MEMBER_HOST, MEMBER_ROLE FROM performance_schema.replication_group_members;
+-------------------------+-------------+
| MEMBER_HOST             | MEMBER_ROLE |
+-------------------------+-------------+
| remote1.example.com     | PRIMARY     |
| remote2.example.com     | SECONDARY   |
| remote3.example.com     | SECONDARY   |
+-------------------------+-------------+
```

Warning

The `group_replication_primary_member` status variable has been deprecated and is scheduled to be removed in a future version.

Alternatively use the `group_replication_primary_member` status variable.

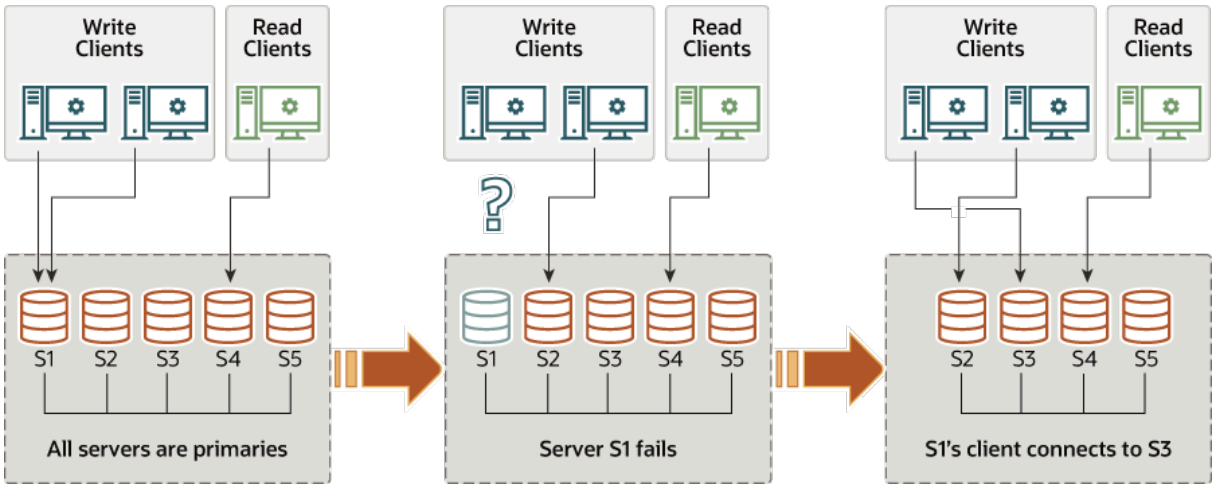
```
mysql> SHOW STATUS LIKE 'group_replication_primary_member'
```

18.1.3.2 Multi-Primary Mode

In multi-primary mode (`group_replication_single_primary_mode=OFF`) 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.

If a member stops accepting write transactions, for example, in the event of an unexpected server exit, clients connected to it can be redirected, or failed over, to any other member that is in read-write mode. Group Replication does not handle client-side failover itself, so you need to arrange this using a middleware framework such as MySQL Router 8.0, a proxy, a connector, or the application itself. Figure 18.5, “Client Failover” shows how clients can reconnect to an alternative group member if a member leaves the group.

Figure 18.5 Client Failover



Group Replication is an eventual consistency system. This means 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 on some members before the others, especially if some members have less write throughput than others, creating the possibility of stale reads. In multi-primary mode, slower members can also build up an excessive backlog of transactions to certify and apply, leading to a greater risk of conflicts and certification failure. To limit these issues, you can activate and tune Group Replication's flow control mechanism to minimize the difference between fast and slow members. For more information on flow control, see Section 18.7.2, “Flow Control”.

From MySQL 8.0.14, if you want to have a transaction consistency guarantee for every transaction in the group, you can do this using the `group_replication_consistency` system variable. You can choose a setting that suits the workload of your group and your priorities for data reads and writes, taking into account the performance impact of the synchronization required to increase consistency. You can also set the system variable for individual sessions to protect particularly concurrency-sensitive transactions. For more information on transaction consistency, see Section 18.5.3, “Transaction Consistency Guarantees”.

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` system variable. In multi-primary mode, the system variable should normally be set to `ON`, but the checks can optionally be deactivated by setting the system variable to `OFF`. When deploying in single-primary mode, the system variable must be set to `OFF`.

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` before executing the statement. This means that DDL statements

cannot be performed within another transaction, within transaction control statements such as `START TRANSACTION ... COMMIT`, or combined with other statements within the same transaction.

Group Replication is based on an optimistic replication paradigm, where statements are optimistically executed and rolled back later if necessary. Each server executes without securing group agreement first. Therefore, more care needs to be taken when replicating DDL statements in multi-primary mode. If you make schema changes (using DDL) and changes to the data that an object contains (using DML) for the same object, the changes need to be handled through the same server while the schema operation has not yet completed and replicated everywhere. Failure to do so can result in data inconsistency when operations are interrupted or only partially completed. 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.

For details on atomic DDL support in MySQL 8.0, and the resulting changes in behavior for the replication of certain statements, see Section 13.1.1, “Atomic Data Definition Statement Support”.

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.

One result of these compatibility checks is particularly important in multi-primary mode. 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. (In a group that is running in single-primary mode, newly added members default to being read-only in any case.) Members running MySQL 8.0.17 or higher take into account the patch version of the release when checking their compatibility. Members running MySQL 8.0.16 or lower, or MySQL 5.7, only take into account the major version.

In a group running in multi-primary mode with members that use different MySQL Server versions, Group Replication automatically manages the read-write and read-only status of members running MySQL 8.0.17 or higher. 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()` function, Group Replication automatically sets members to the correct mode. Members are automatically placed in read-only mode if they are running a higher MySQL server version than the lowest version present in the group, and members running the lowest version are placed in read-write mode.

For full information on version compatibility in a group and how this influences the behavior of a group during an upgrade process, see Section 18.8.1, “Combining Different Member Versions in a Group” .

18.1.4 Group Replication Services

This section introduces some of the services that Group Replication builds on.

18.1.4.1 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.

If a server joins the group, it automatically brings itself up to date by fetching the missing state from an existing server. If a server leaves the group, for instance it was taken down for maintenance, the remaining servers notice that it has left and reconfigure the group automatically.

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.

Group members must agree not only on transaction commits, but also on which is the current view. If existing members agree that a new server should become part of the group, the group is reconfigured to integrate that server in it, which triggers a view change. If a server leaves the group, either voluntarily or not, the group dynamically rearranges its configuration and a view change is triggered.

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, from MySQL 5.7.22 and in MySQL 8.0, 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` 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` statement or a server failure, it must rejoin as a new incarnation.

18.1.4.2 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`. Usually, two members will suspect each other of having failed

because they are each not communicating with the other. It is possible, though less likely, that member A suspects member B of having failed but member B does not suspect member A of having failed - perhaps due to a routing or firewall issue. A member can also create a suspicion of itself. A member that is isolated from the rest of the group suspects that all the others have failed.

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. If a member is actually isolated from the rest of the group, it might attempt to propagate its view, but that will have no consequences as it cannot secure a quorum of the other members to agree on it. 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` 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.

For information on the Group Replication system variables that you can configure to specify the responses of working group members to failure situations, and the actions taken by group members that are suspected of having failed, see Section 18.7.7, “Responses to Failure Detection and Network Partitioning”.

18.1.4.3 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`.

In practice this means that to tolerate one failure the group must have three servers in it. As such if one server fails, there are still two servers to form a majority (two out of three) and allow the system to continue to make decisions automatically and progress. However, if a second server fails *involuntarily*, then the group (with one server left) blocks, because there is no majority to reach a decision.

The following is a small table illustrating the formula above.

Group Size	Majority	Instant Failures Tolerated
1	1	0
2	2	0
3	2	1
4	3	1
5	3	2
6	4	2
7	4	3

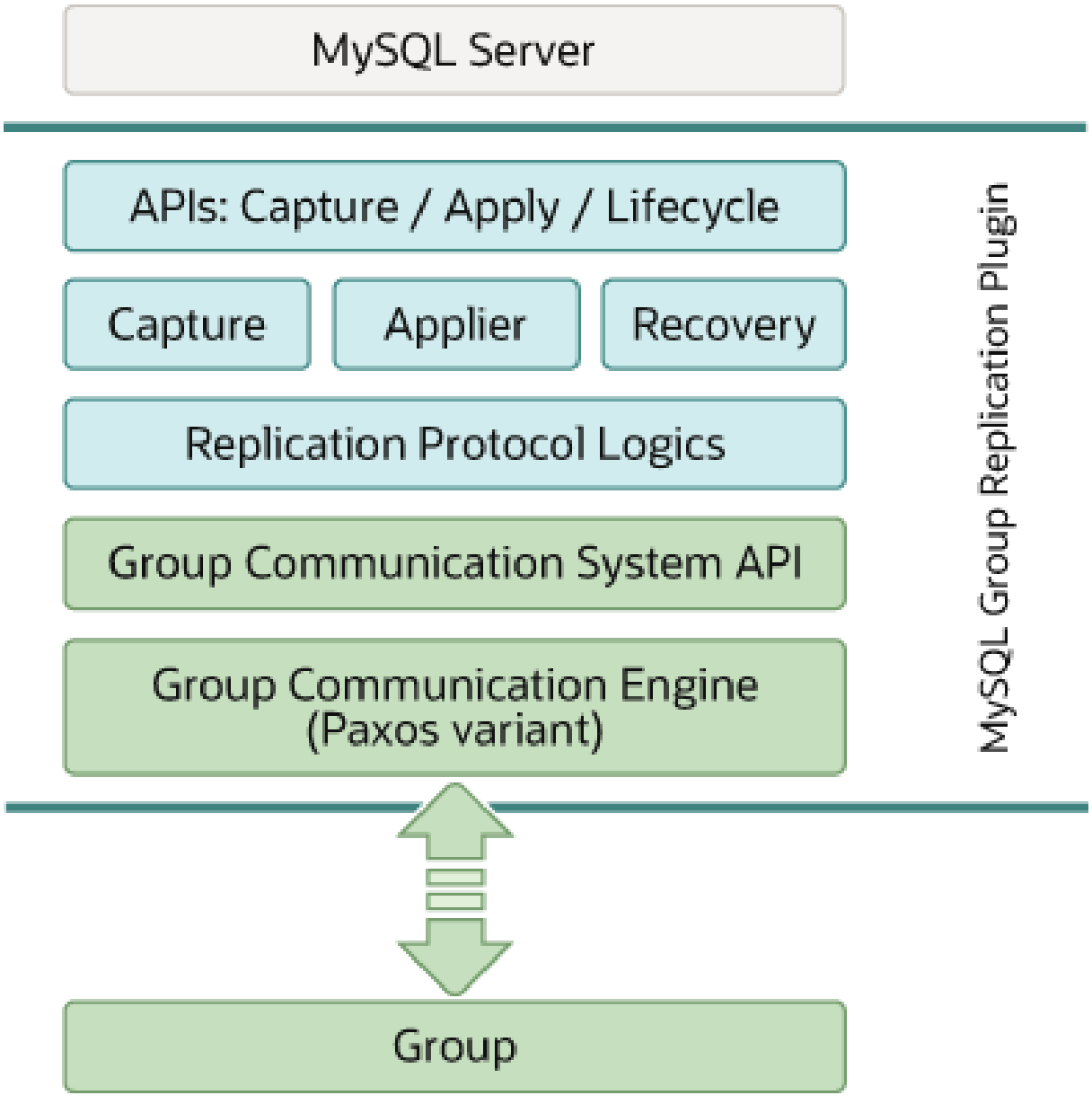
18.1.4.4 Observability

There is a lot of automation built into the Group Replication plugin. Nonetheless, you might sometimes need to understand what is happening behind the scenes. This is where the instrumentation of Group Replication and Performance Schema becomes important. 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. For example, you can connect to a single server in the group and obtain both local and global information by issuing select statements on the Group Replication related Performance Schema tables. For more information, see Section 18.4, “Monitoring Group Replication”.

18.1.5 Group Replication Plugin Architecture

MySQL Group Replication is a MySQL plugin and it builds on the existing MySQL replication infrastructure, taking advantage of features such as the binary log, row-based logging, and global transaction identifiers. It integrates with current MySQL frameworks, such as the performance schema or plugin and service infrastructures. The following figure presents a block diagram depicting the overall architecture of MySQL Group Replication.

Figure 18.6 Group Replication Plugin Block Diagram



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 (see Section 18.1, “Group Replication Background”). 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.

18.2 Getting Started

MySQL Group Replication is provided as a plugin for the MySQL server; each server in a group requires configuration and installation of the plugin. This section provides a detailed tutorial with the steps required to create a replication group with at least three members.

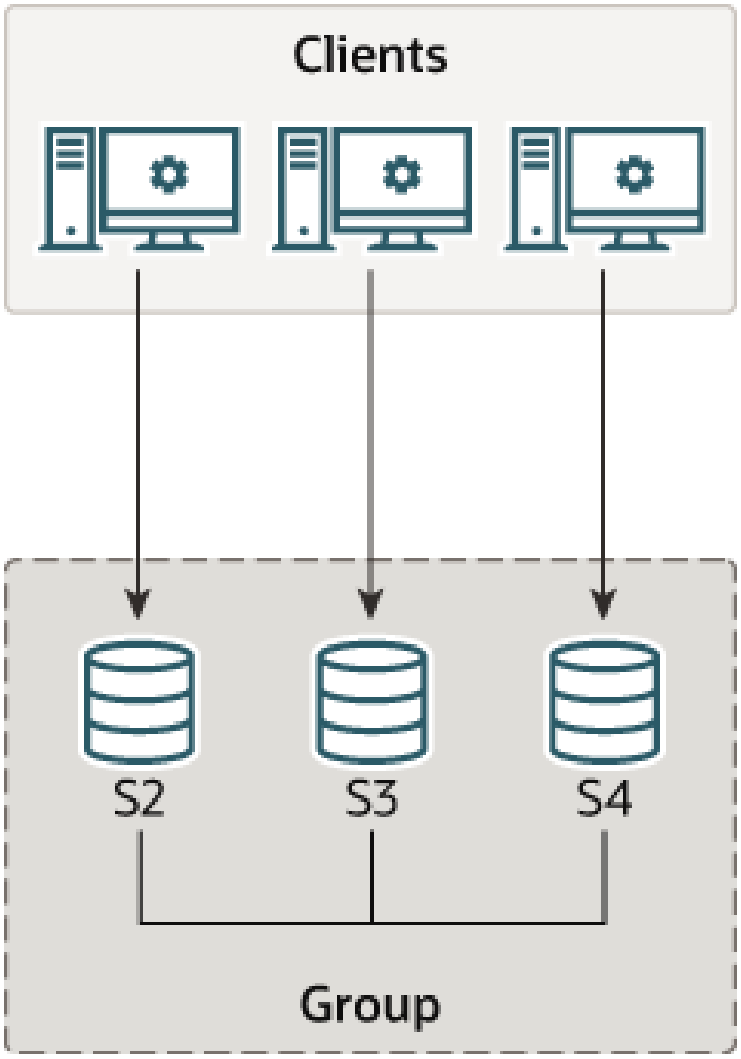
Tip

To deploy multiple instances of MySQL, you can use InnoDB Cluster which enables you to easily administer a group of MySQL server instances in MySQL Shell. InnoDB Cluster wraps MySQL Group Replication in a programmatic environment that enables you easily deploy a cluster of MySQL instances to achieve high availability. In addition, InnoDB Cluster interfaces seamlessly with MySQL Router, which enables your applications to connect to the cluster without writing your own failover process. For similar use cases that do not require high availability, however, you can use InnoDB ReplicaSet. Installation instructions for MySQL Shell can be found here.

18.2.1 Deploying Group Replication in Single-Primary Mode

Each of the MySQL server instances in a group can run on an independent physical host machine, which is the recommended way to deploy Group Replication. This section explains how to create a replication group with three MySQL Server instances, each running on a different host machine. See Section 18.2.2, “Deploying Group Replication Locally” for information about deploying multiple MySQL server instances running Group Replication on the same host machine, for example for testing purposes.

Figure 18.7 Group Architecture



This tutorial explains how to get and deploy MySQL Server with the Group Replication plugin, how to configure each server instance before creating a group, and how to use Performance Schema monitoring to verify that everything is working correctly.

18.2.1.1 Deploying Instances for Group Replication

The first step is to deploy at least three instances of MySQL Server, this procedure demonstrates using multiple hosts for the instances, named s1, s2, and s3. It is assumed that MySQL Server is installed on each host (see Chapter 2, *Installing and Upgrading MySQL*). The Group Replication plugin is provided with MySQL Server 8.0; no additional software is required, although the plugin must be installed in the running MySQL server. See Section 18.2.1.1, “Deploying Instances for Group Replication”; for additional information, see Section 5.6, “MySQL Server Plugins”.

In this example, three instances are used for the group, which is the minimum number of instances to create a group. Adding more instances increases the fault tolerance of the group. For example if the group consists of three members, in event of failure of one instance the group can continue. But in the event of another failure the group can no longer continue processing write transactions. By adding more instances, the number of servers which can fail while the group continues to process transactions also increases. The maximum number of instances which can be used in a group is nine. For more information see Section 18.1.4.2, “Failure Detection”.

18.2.1.2 Configuring an Instance for Group Replication

This section explains the configuration settings required for MySQL Server instances that you want to use for Group Replication. For background information, see Section 18.3, “Requirements and Limitations”.

- Storage Engines
- Replication Framework
- Group Replication Settings

Storage Engines

For Group Replication, data must be stored in the InnoDB transactional storage engine (for details of why, see Section 18.3.1, “Group Replication Requirements”). The use of other storage engines, including the temporary `MEMORY` storage engine, might cause errors in Group Replication. Set the `disabled_storage_engines` system variable as follows to prevent their use:

```
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"
```

Note that with the `MyISAM` storage engine disabled, when you are upgrading a MySQL instance to a release where `mysql_upgrade` is still used (before MySQL 8.0.16), `mysql_upgrade` might fail with an error. To handle this, you can re-enable that storage engine while you run `mysql_upgrade`, then disable it again when you restart the server. For more information, see Section 4.4.5, “mysql_upgrade — Check and Upgrade MySQL Tables”.

Replication Framework

The following settings configure replication according to the MySQL Group Replication requirements.

```
server_id=1
gtid_mode=ON
enforce_gtid_consistency=ON
```

These settings configure the server to use the unique identifier number 1, to enable Section 17.1.3, “Replication with Global Transaction Identifiers”, and to allow execution of only statements that can be safely logged using a GTID.

Up to and including MySQL 8.0.20, the following setting is also required:

```
binlog_checksum=NONE
```

This setting disables checksums for events written to the binary log, which default to being enabled. From MySQL 8.0.21, Group Replication supports the presence of checksums in the binary log and can use them to verify the integrity of events on some channels, so you can use the default setting. For more details, see Section 18.3.2, “Group Replication Limitations”.

If you are using a version of MySQL earlier than 8.0.3, where the defaults were improved for replication, you also need to add these lines to the member's option file. If you have any of these system variables in the option file in later versions, ensure that they are set as shown. For more details see Section 18.3.1, “Group Replication Requirements”.

```
log_bin=binlog
log_slave_updates=ON
binlog_format=ROW
master_info_repository=TABLE
relay_log_info_repository=TABLE
transaction_write_set_extraction=XXHASH64
```

Group Replication Settings

At this point the option file ensures that the server is configured and is instructed to instantiate the replication infrastructure under a given configuration. The following section configures the Group Replication settings for the server.

```
plugin_load_add='group_replication.so'
```

```
group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
group_replication_start_on_boot=off
group_replication_local_address= "s1:33061"
group_replication_group_seeds= "s1:33061,s2:33061,s3:33061"
group_replication_bootstrap_group=off
```

- `plugin-load-add` adds the Group Replication plugin to the list of plugins which the server loads at startup. This is preferable in a production deployment to installing the plugin manually.
- Configuring `group_replication_group_name` tells the plugin that the group that it is joining, or creating, is named "aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa".

  The value of `group_replication_group_name` must be a valid UUID. You can use `SELECT UUID()` to generate one. This UUID forms part of the GTIDs that are used when transactions received by group members from clients, and view change events that are generated internally by the group members, are written to the binary log.
- Configuring the `group_replication_start_on_boot` variable to `off` instructs the plugin to not start operations automatically when the server starts. This is important when setting up Group Replication as it ensures you can configure the server before manually starting the plugin. Once the member is configured you can set `group_replication_start_on_boot` to `on` so that Group Replication starts automatically upon server boot.
- Configuring `group_replication_local_address` sets the network address and port which the member uses for internal communication with other members in the group. Group Replication uses this address for internal member-to-member connections involving remote instances of the group communication engine (XCom, a Paxos variant).

  Important

  The group replication local address must be different to the host name and port used for SQL client connections, which are defined by MySQL Server's `hostname` and `port` system variables. It must not be used for client applications. It must be only be used for internal communication between the members of the group while running Group Replication.

  The network address configured by `group_replication_local_address` must be resolvable by all group members. For example, if each server instance is on a different machine with a fixed network address, you could use the IP address of the machine, such as 10.0.0.1. If you use a host name, you must use a fully qualified name, and ensure it is resolvable through DNS, correctly configured `/etc/hosts` files, or other name resolution processes. From MySQL 8.0.14, IPv6 addresses (or host names that resolve to them) can be used as well as IPv4 addresses. A group can contain a mix of members using IPv6 and members using IPv4. For more information on Group Replication support for IPv6 networks and on mixed IPv4 and IPv6 groups, see Section 18.5.5, “Support For IPv6 And For Mixed IPv6 And IPv4 Groups”.

  The recommended port for `group_replication_local_address` is 33061. `group_replication_local_address` is used by Group Replication as the unique identifier for a group member within the replication group. You can use the same port for all members of a replication group as long as the host names or IP addresses are all different, as demonstrated in this tutorial. Alternatively you can use the same host name or IP address for all members as long as the ports are all different, for example as shown in Section 18.2.2, “Deploying Group Replication Locally”.

  The connection that an existing member offers to a joining member for Group Replication's distributed recovery process is not the network address configured by `group_replication_local_address`. Up to MySQL 8.0.20, group members offer their standard SQL client connection to joining members for distributed recovery, as specified by MySQL Server's `hostname` and `port` system variables. From MySQL 8.0.21, group members may advertise an alternative list of distributed recovery endpoints as dedicated client connections for joining members. For more details, see Section 18.5.4.1, “Connections for Distributed Recovery”.

Important

Distributed recovery can fail if a joining member cannot correctly identify the other members using the host name as defined by MySQL Server's `hostname` system variable. It is recommended that operating systems running MySQL have a properly configured unique host name, either using DNS or local settings. The host name that the server is using for SQL client connections can be verified in the `Member_host` column of the Performance Schema table `replication_group_members`. If multiple group members externalize a default host name set by the operating system, there is a chance of the joining member not resolving it to the correct member address and not being able to connect for distributed recovery. In this situation you can use MySQL Server's `report_host` system variable to configure a unique host name to be externalized by each of the servers.

- Configuring `group_replication_group_seeds` sets the hostname and port of the group members which are used by the new member to establish its connection to the group. These members are called the seed members. Once the connection is established, the group membership information is listed in the Performance Schema table `replication_group_members`. Usually the `group_replication_group_seeds` list contains the `hostname:port` of each of the group member's `group_replication_local_address`, but this is not obligatory and a subset of the group members can be chosen as seeds.

  Important

  The `hostname:port` listed in `group_replication_group_seeds` is the seed member's internal network address, configured by `group_replication_local_address` and not the `hostname:port` used for SQL client connections, which is shown for example in the Performance Schema table `replication_group_members`.

  The server that starts the group does not make use of this option, since it is the initial server and as such, it is in charge of bootstrapping the group. In other words, any existing data which is on the server bootstrapping the group is what is used as the data for the next joining member. The second server joining asks the one and only member in the group to join, any missing data on the second server is replicated from the donor data on the bootstrapping member, and then the group expands. The third server joining can ask any of these two to join, data is synchronized to the new member, and then the group expands again. Subsequent servers repeat this procedure when joining.

  Warning

  When joining multiple servers at the same time, make sure that they point to seed members that are already in the group. Do not use members that are also joining the group as seeds, because they might not yet be in the group when contacted.

  It is good practice to start the bootstrap member first, and let it create the group. Then make it the seed member for the rest of the members that are joining. This ensures that there is a group formed when joining the rest of the members.

  Creating a group and joining multiple members at the same time is not supported. It might work, but chances are that the operations race and then the act of joining the group ends up in an error or a time out.

  A joining member must communicate with a seed member using the same protocol (IPv4 or IPv6) that the seed member advertises in the `group_replication_group_seeds` option. For the purpose of IP address permissions for Group Replication, the allowlist on the seed member must include an IP address for the joining member for the protocol offered by the seed member, or a

host name that resolves to an address for that protocol. This address or host name must be set up and permitted in addition to the joining member's `group_replication_local_address` if the protocol for that address does not match the seed member's advertised protocol. If a joining member does not have a permitted address for the appropriate protocol, its connection attempt is refused. For more information, see Section 18.6.4, “Group Replication IP Address Permissions”.

- Configuring `group_replication_bootstrap_group` instructs the plugin whether to bootstrap the group or not. In this case, even though s1 is the first member of the group we set this variable to off in the option file. Instead we configure `group_replication_bootstrap_group` when the instance is running, to ensure that only one member actually bootstraps the group.

Important

The `group_replication_bootstrap_group` variable must only be enabled on one server instance belonging to a group at any time, usually the first time you bootstrap the group (or in case the entire group is brought down and back up again). If you bootstrap the group multiple times, for example when multiple server instances have this option set, then they could create an artificial split brain scenario, in which two distinct groups with the same name exist. Always set `group_replication_bootstrap_group=off` after the first server instance comes online.

The system variables described in this tutorial are the required configuration settings to start a new member, but further system variables are also available to configure group members. These are listed in Section 18.9, “Group Replication System Variables”.

Important

A number of system variables, some specific to Group Replication and others not, are group-wide configuration settings that must have the same value on all group members. If the group members have a value set for one of these system variables, and a joining member has a different value set for it, the joining member cannot join the group and an error message is returned. If the group members have a value set for this system variable, and the joining member does not support the system variable, it cannot join the group. These system variables are all identified in Section 18.9, “Group Replication System Variables”.

18.2.1.3 User Credentials For Distributed Recovery

Group Replication uses a distributed recovery process to synchronize group members when joining them to the group. Distributed recovery involves transferring transactions from a donor's binary log to a joining member using a replication channel named `group_replication_recovery`. You must therefore set up a replication user with the correct permissions so that Group Replication can establish direct member-to-member replication channels. If group members have been set up to support the use of a remote cloning operation as part of distributed recovery, which is available from MySQL 8.0.17, this replication user is also used as the clone user on the donor, and requires the correct permissions for this role too. For a complete description of distributed recovery, see Section 18.5.4, “Distributed Recovery”.

The same replication user must be used for distributed recovery on every group member. The process of creating the replication user for distributed recovery can be captured in the binary log, and then you can rely on distributed recovery to replicate the statements used to create the user. Alternatively, you can disable binary logging before creating the replication user, and then create the user manually on each member, for example if you want to avoid the changes being propagated to other server instances. If you do this, ensure you re-enable binary logging once you have configured the user.

Important

If distributed recovery connections for your group use SSL, the replication user must be created on each server *before* the joining member connects to the donor. For instructions to set up SSL for distributed recovery connections and create a replication user that requires SSL, see Section 18.6.3, “Securing Distributed Recovery Connections”

Important

By default, users created in MySQL 8 use Section 6.4.1.2, “Caching SHA-2 Pluggable Authentication”. If the replication user for distributed recovery uses the caching SHA-2 authentication plugin, and you are *not* using SSL for distributed recovery connections, RSA key-pairs are used for password exchange. You can either copy the public key of the replication user to the joining member, or configure the donors to provide the public key when requested. For instructions to do this, see Section 18.6.3.1, “Secure User Credentials for Distributed Recovery”.

To create the replication user for distributed recovery, follow these steps:

1. Start the MySQL server instance, then connect a client to it.
2. If you want to disable binary logging in order to create the replication user separately on each instance, do so by issuing the following statement:

```
mysql> SET SQL_LOG_BIN=0;
```

3. Create a MySQL user with the following privileges:

   - `REPLICATION SLAVE`, which is required for making a distributed recovery connection to a donor to retrieve data.
   - `CONNECTION_ADMIN`, which ensures that Group Replication connections are not terminated if one of the servers involved is placed in offline mode.
   - `BACKUP_ADMIN`, if the servers in the replication group are set up to support cloning (see Section 18.5.4.2, “Cloning for Distributed Recovery”). This privilege is required for a member to act as the donor in a cloning operation for distributed recovery.
   - `GROUP_REPLICATION_STREAM`, if the MySQL communication stack is in use for the replication group (see Section 18.6.1, “Communication Stack for Connection Security Management”). This privilege is required for the user account to be able to establish and maintain connections for Group Replication using the MySQL communication stack.

   In this example the user *`rpl_user`* with the password *`password`* is shown. When configuring your servers use a suitable user name and password:

```
mysql> CREATE USER rpl_user@'%' IDENTIFIED BY 'password';
mysql> GRANT REPLICATION SLAVE ON *.* TO rpl_user@'%';
mysql> GRANT CONNECTION_ADMIN ON *.* TO rpl_user@'%';
mysql> GRANT BACKUP_ADMIN ON *.* TO rpl_user@'%';
mysql> GRANT GROUP_REPLICATION_STREAM ON *.* TO rpl_user@'%';
mysql> FLUSH PRIVILEGES;
```

4. If you disabled binary logging, enable it again as soon as you have created the user, by issuing the following statement:

```
mysql> SET SQL_LOG_BIN=1;
```

5. When you have created the replication user, you must supply the user credentials to the server for use with distributed recovery. You can do this by setting the user credentials as the credentials for the `group_replication_recovery` channel, using a `CHANGE REPLICATION SOURCE`

TO statement (from MySQL 8.0.23) or CHANGE MASTER TO statement (before MySQL 8.0.23). Alternatively, from MySQL 8.0.21, you can specify the user credentials for distributed recovery on the START GROUP_REPLICATION statement.

- User credentials set using CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO are stored in plain text in the replication metadata repositories on the server. They are applied whenever Group Replication is started, including automatic starts if the group_replication_start_on_boot system variable is set to ON.
- User credentials specified on START GROUP_REPLICATION are saved in memory only, and are removed by a STOP GROUP_REPLICATION statement or server shutdown. You must issue a START GROUP_REPLICATION statement to provide the credentials again, so you cannot start Group Replication automatically with these credentials. This method of specifying the user credentials helps to secure the Group Replication servers against unauthorized access.

For more information on the security implications of each method of providing the user credentials, see Providing Replication User Credentials Securely. If you choose to provide the user credentials using a CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO statement, issue the following statement on the server instance now, replacing *rpl_user* and *password* with the values used when creating the user:

```
mysql> CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='password' \\
        FOR CHANNEL 'group_replication_recovery';

Or from MySQL 8.0.23:
mysql> CHANGE REPLICATION SOURCE TO SOURCE_USER='rpl_user', SOURCE_PASSWORD='password' \\
        FOR CHANNEL 'group_replication_recovery';
```

18.2.1.4 Launching Group Replication

It is first necessary to ensure that the Group Replication plugin is installed on server s1. If you used plugin_load_add='group_replication.so' in the option file then the Group Replication plugin is already installed, and you can proceed to the next step. Otherwise, you must install the plugin manually; to do this, connect to the server using the mysql client, and issue the SQL statement shown here:

```
mysql> INSTALL PLUGIN group_replication SONAME 'group_replication.so';
```

Important

The mysql.session user must exist before you can load Group Replication. mysql.session was added in MySQL version 8.0.2. If your data dictionary was initialized using an earlier version you must perform the MySQL upgrade procedure (see Section 2.10, “Upgrading MySQL”). If the upgrade is not run, Group Replication fails to start with the error message There was an error when trying to access the server with user: mysql.session@localhost. Make sure the user is present in the server and that mysql_upgrade was ran after a server update.

To check that the plugin was installed successfully, issue SHOW PLUGINS; and check the output. It should show something like this:

```
mysql> SHOW PLUGINS;
+----------------------------+----------+--------------------+----------------------+-------------+
| Name                       | Status   | Type               | Library              | License     |
+----------------------------+----------+--------------------+----------------------+-------------+
| binlog                     | ACTIVE   | STORAGE ENGINE     | NULL                 | PROPRIETARY |

(...)

| group_replication          | ACTIVE   | GROUP REPLICATION  | group_replication.so | PROPRIETARY |
```

```
+---------------------------+----------+-------------------+---------------------+-------------+
```

18.2.1.5 Bootstrapping the Group

The process of starting a group for the first time is called bootstrapping. You use the `group_replication_bootstrap_group` system variable to bootstrap a group. The bootstrap should only be done by a single server, the one that starts the group and only once. This is why the value of the `group_replication_bootstrap_group` option was not stored in the instance's option file. If it is saved in the option file, upon restart the server automatically bootstraps a second group with the same name. This would result in two distinct groups with the same name. The same reasoning applies to stopping and restarting the plugin with this option set to `ON`. Therefore to safely bootstrap the group, connect to s1 and issue the following statements:

```
mysql> SET GLOBAL group_replication_bootstrap_group=ON;
mysql> START GROUP_REPLICATION;
mysql> SET GLOBAL group_replication_bootstrap_group=OFF;
```

Or if you are providing user credentials for distributed recovery on the `START GROUP_REPLICATION` statement (which you can from MySQL 8.0.21), issue the following statements:

```
mysql> SET GLOBAL group_replication_bootstrap_group=ON;
mysql> START GROUP_REPLICATION USER='rpl_user', PASSWORD='password';
mysql> SET GLOBAL group_replication_bootstrap_group=OFF;
```

Once the `START GROUP_REPLICATION` statement returns, the group has been started. You can check that the group is now created and that there is one member in it:

```
mysql> SELECT * FROM performance_schema.replication_group_members;
+---------------------------+--------------------------------------+-------------+-------------+-------
| CHANNEL_NAME              | MEMBER_ID                            | MEMBER_HOST | MEMBER_PORT | MEMBER
+---------------------------+--------------------------------------+-------------+-------------+-------
| group_replication_applier | ce9be252-2b71-11e6-b8f4-00212844f856 |   s1        |       3306  | ONLINE
+---------------------------+--------------------------------------+-------------+-------------+-------
1 row in set (0.0108 sec)
```

The information in this table confirms that there is a member in the group with the unique identifier `ce9be252-2b71-11e6-b8f4-00212844f856`, that it is `ONLINE` and is at `s1` listening for client connections on port `3306`.

For the purpose of demonstrating that the server is indeed in a group and that it is able to handle load, create a table and add some content to it.

```
mysql> CREATE DATABASE test;
mysql> USE test;
mysql> CREATE TABLE t1 (c1 INT PRIMARY KEY, c2 TEXT NOT NULL);
mysql> INSERT INTO t1 VALUES (1, 'Luis');
```

Check the content of table `t1` and the binary log.

```
mysql> SELECT * FROM t1;
+----+------+
| c1 | c2   |
+----+------+
|  1 | Luis |
+----+------+

mysql> SHOW BINLOG EVENTS;
+---------------+-----+----------------+-----------+-------------+-------------------------------------
| Log_name      | Pos | Event_type     | Server_id | End_log_pos | Info
+---------------+-----+----------------+-----------+-------------+-------------------------------------
| binlog.000001 |   4 | Format_desc    |         1 |         123 | Server ver: 8.0.32-log, Binlog ver:
| binlog.000001 | 123 | Previous_gtids |         1 |         150 |
| binlog.000001 | 150 | Gtid           |         1 |         211 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-a
| binlog.000001 | 211 | Query          |         1 |         270 | BEGIN
| binlog.000001 | 270 | View_change    |         1 |         369 | view_id=14724817264259180:1
| binlog.000001 | 369 | Query          |         1 |         434 | COMMIT
```

```
| binlog.000001 | 434 | Gtid           |         1 |         495 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa-
| binlog.000001 | 495 | Query          |         1 |         585 | CREATE DATABASE test
| binlog.000001 | 585 | Gtid           |         1 |         646 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa-
| binlog.000001 | 646 | Query          |         1 |         770 | use `test`; CREATE TABLE t1 (c1 INT PRIN
| binlog.000001 | 770 | Gtid           |         1 |         831 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa-
| binlog.000001 | 831 | Query          |         1 |         899 | BEGIN
| binlog.000001 | 899 | Table_map      |         1 |         942 | table_id: 108 (test.t1)
| binlog.000001 | 942 | Write_rows     |         1 |         984 | table_id: 108 flags: STMT_END_F
| binlog.000001 | 984 | Xid            |         1 |        1011 | COMMIT /* xid=38 */
+---------------+-----+----------------+-----------+-------------+-------------------------------------
```

As seen above, the database and the table objects were created and their corresponding DDL statements were written to the binary log. Also, the data was inserted into the table and written to the binary log, so it can be used for distributed recovery by state transfer from a donor's binary log.

18.2.1.6 Adding Instances to the Group

At this point, the group has one member in it, server s1, which has some data in it. It is now time to expand the group by adding the other two servers configured previously.

Adding a Second Instance

In order to add a second instance, server s2, first create the configuration file for it. The configuration is similar to the one used for server s1, except for things such as the `server_id`. These different lines are highlighted in the listing below.

```
[mysqld]

#
# Disable other storage engines
#
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"

#
# Replication configuration parameters
#
server_id=2
gtid_mode=ON
enforce_gtid_consistency=ON
binlog_checksum=NONE           # Not needed from 8.0.21

#
# Group Replication configuration
#
plugin_load_add='group_replication.so'
group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
group_replication_start_on_boot=off
group_replication_local_address= "s2:33061"
group_replication_group_seeds= "s1:33061,s2:33061,s3:33061"
group_replication_bootstrap_group= off
```

Similar to the procedure for server s1, with the option file in place you launch the server. Then configure the distributed recovery credentials as follows. The commands are the same as used when setting up server s1 as the user is shared within the group. This member needs to have the same replication user configured in Section 18.2.1.3, “User Credentials For Distributed Recovery”. If you are relying on distributed recovery to configure the user on all members, when s2 connects to the seed s1 the replication user is replicated or cloned to s1. If you did not have binary logging enabled when you configured the user credentials on s1, and a remote cloning operation is not used for state transfer, you must create the replication user on s2. In this case, connect to s2 and issue:

```
SET SQL_LOG_BIN=0;
CREATE USER rpl_user@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO rpl_user@'%';
GRANT CONNECTION_ADMIN ON *.* TO rpl_user@'%';
GRANT BACKUP_ADMIN ON *.* TO rpl_user@'%';
GRANT GROUP_REPLICATION_STREAM ON *.* TO rpl_user@'%';
FLUSH PRIVILEGES;
```

```
SET SQL_LOG_BIN=1;
```

If you are providing user credentials using a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement, issue the following statement after that:

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='password' \\
 FOR CHANNEL 'group_replication_recovery';

Or from MySQL 8.0.23:
CHANGE REPLICATION SOURCE TO SOURCE_USER='rpl_user', SOURCE_PASSWORD='password' \\
 FOR CHANNEL 'group_replication_recovery';
```

Tip

If you are using the caching SHA-2 authentication plugin, the default in MySQL 8, see Replication User With The Caching SHA-2 Authentication Plugin.

If necessary, install the Group Replication plugin, see Section 18.2.1.4, “Launching Group Replication”.

Start Group Replication and s2 starts the process of joining the group.

```
mysql> START GROUP_REPLICATION;
```

Or if you are providing user credentials for distributed recovery on the `START GROUP_REPLICATION` statement (which you can from MySQL 8.0.21):

```
mysql> START GROUP_REPLICATION USER='rpl_user', PASSWORD='password';
```

Unlike the previous steps that were the same as those executed on s1, here there is a difference in that you do *not* need to bootstrap the group because the group already exists. In other words on s2 `group_replication_bootstrap_group` is set to `OFF`, and you do not issue `SET GLOBAL group_replication_bootstrap_group=ON;` before starting Group Replication, because the group has already been created and bootstrapped by server s1. At this point server s2 only needs to be added to the already existing group.

Tip

When Group Replication starts successfully and the server joins the group it checks the `super_read_only` variable. By setting `super_read_only` to ON in the member's configuration file, you can ensure that servers which fail when starting Group Replication for any reason do not accept transactions. If the server should join the group as a read-write instance, for example as the primary in a single-primary group or as a member of a multi-primary group, when the `super_read_only` variable is set to ON then it is set to OFF upon joining the group.

Checking the `performance_schema.replication_group_members` table again shows that there are now two `ONLINE` servers in the group.

```
mysql> SELECT * FROM performance_schema.replication_group_members;
+---------------------------+--------------------------------------+-------------+-------------+-------
| CHANNEL_NAME              | MEMBER_ID                            | MEMBER_HOST | MEMBER_PORT | MEMBER
+---------------------------+--------------------------------------+-------------+-------------+-------
| group_replication_applier | 395409e1-6dfa-11e6-970b-00212844f856 |   s1        |        3306 | ONLINE
| group_replication_applier | ac39f1e6-6dfa-11e6-a69d-00212844f856 |   s2        |        3306 | ONLINE
+---------------------------+--------------------------------------+-------------+-------------+-------
```

When s2 attempted to join the group, Section 18.5.4, “Distributed Recovery” ensured that s2 applied the same transactions which s1 had applied. Once this process completed, s2 could join the group as a member, and at this point it is marked as `ONLINE`. In other words it must have already caught up with server s1 automatically. Once s2 is `ONLINE`, it then begins to process transactions with the group. Verify that s2 has indeed synchronized with server s1 as follows.

```
mysql> SHOW DATABASES LIKE 'test';
```

```
+-----------------+
| Database (test) |
+-----------------+
| test            |
+-----------------+

mysql> SELECT * FROM test.t1;
+----+------+
| c1 | c2   |
+----+------+
|  1 | Luis |
+----+------+

mysql> SHOW BINLOG EVENTS;
+---------------+------+----------------+-----------+-------------+-----------------------------------------
| Log_name      | Pos  | Event_type     | Server_id | End_log_pos | Info
+---------------+------+----------------+-----------+-------------+-----------------------------------------
| binlog.000001 |    4 | Format_desc    |         2 |         123 | Server ver: 8.0.32-log, Binlog ver: 4
| binlog.000001 |  123 | Previous_gtids |         2 |         150 |
| binlog.000001 |  150 | Gtid           |         1 |         211 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  211 | Query          |         1 |         270 | BEGIN
| binlog.000001 |  270 | View_change    |         1 |         369 | view_id=14724832985483517:1
| binlog.000001 |  369 | Query          |         1 |         434 | COMMIT
| binlog.000001 |  434 | Gtid           |         1 |         495 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  495 | Query          |         1 |         585 | CREATE DATABASE test
| binlog.000001 |  585 | Gtid           |         1 |         646 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  646 | Query          |         1 |         770 | use `test`; CREATE TABLE t1 (c1 INT PRI
| binlog.000001 |  770 | Gtid           |         1 |         831 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  831 | Query          |         1 |         890 | BEGIN
| binlog.000001 |  890 | Table_map      |         1 |         933 | table_id: 108 (test.t1)
| binlog.000001 |  933 | Write_rows     |         1 |         975 | table_id: 108 flags: STMT_END_F
| binlog.000001 |  975 | Xid            |         1 |        1002 | COMMIT /* xid=30 */
| binlog.000001 | 1002 | Gtid           |         1 |        1063 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 | 1063 | Query          |         1 |        1122 | BEGIN
| binlog.000001 | 1122 | View_change    |         1 |        1261 | view_id=14724832985483517:2
| binlog.000001 | 1261 | Query          |         1 |        1326 | COMMIT
+---------------+------+----------------+-----------+-------------+-----------------------------------------
```

As seen above, the second server has been added to the group and it has replicated the changes from server s1 automatically. In other words, the transactions applied on s1 up to the point in time that s2 joined the group have been replicated to s2.

Adding Additional Instances

Adding additional instances to the group is essentially the same sequence of steps as adding the second server, except that the configuration has to be changed as it had to be for server s2. To summarise the required commands:

1. Create the configuration file.

```
[mysqld]

#
# Disable other storage engines
#
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"

#
# Replication configuration parameters
#
server_id=3
gtid_mode=ON
enforce_gtid_consistency=ON
binlog_checksum=NONE           # Not needed from 8.0.21

#
# Group Replication configuration
#
plugin_load_add='group_replication.so'
group_replication_group_name="aaaaaaaa-aaaa-aaaa-aaaa-aaaaaaaaaaaa"
```

```
group_replication_start_on_boot=off
group_replication_local_address= "s3:33061"
group_replication_group_seeds= "s1:33061,s2:33061,s3:33061"
group_replication_bootstrap_group= off
```

2. Start the server and connect to it. Create the replication user for distributed recovery.

```
SET SQL_LOG_BIN=0;
CREATE USER rpl_user@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO rpl_user@'%';
GRANT CONNECTION_ADMIN ON *.* TO rpl_user@'%';
GRANT BACKUP_ADMIN ON *.* TO rpl_user@'%';
GRANT GROUP_REPLICATION_STREAM ON *.* TO rpl_user@'%';
FLUSH PRIVILEGES;
SET SQL_LOG_BIN=1;
```

   If you are providing user credentials using a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement, issue the following statement after that:

```
CHANGE MASTER TO MASTER_USER='rpl_user', MASTER_PASSWORD='password' \\
 FOR CHANNEL 'group_replication_recovery';

Or from MySQL 8.0.23:
CHANGE REPLICATION SOURCE TO SOURCE_USER='rpl_user', SOURCE_PASSWORD='password' \\
 FOR CHANNEL 'group_replication_recovery';
```

3. Install the Group Replication plugin if necessary.

```

INSTALL PLUGIN group_replication SONAME 'group_replication.so';
```

4. Start Group Replication.

```
mysql> START GROUP_REPLICATION;
```

   Or if you are providing user credentials for distributed recovery on the `START GROUP_REPLICATION` statement (which you can from MySQL 8.0.21):

```
mysql> START GROUP_REPLICATION USER='rpl_user', PASSWORD='password';
```

At this point server s3 is booted and running, has joined the group and caught up with the other servers in the group. Consulting the `performance_schema.replication_group_members` table again confirms this is the case.

```
mysql> SELECT * FROM performance_schema.replication_group_members;
+---------------------------+--------------------------------------+-------------+-------------+-------
| CHANNEL_NAME              | MEMBER_ID                            | MEMBER_HOST | MEMBER_PORT | MEMBER
+---------------------------+--------------------------------------+-------------+-------------+-------
| group_replication_applier | 395409e1-6dfa-11e6-970b-00212844f856 |   s1        |        3306 | ONLINE
| group_replication_applier | 7eb217ff-6df3-11e6-966c-00212844f856 |   s3        |        3306 | ONLINE
| group_replication_applier | ac39f1e6-6dfa-11e6-a69d-00212844f856 |   s2        |        3306 | ONLINE
+---------------------------+--------------------------------------+-------------+-------------+-------
```

Issuing this same query on server s2 or server s1 yields the same result. Also, you can verify that server s3 has caught up:

```
mysql> SHOW DATABASES LIKE 'test';
+-----------------+
| Database (test) |
+-----------------+
| test            |
+-----------------+

mysql> SELECT * FROM test.t1;
+----+------+
| c1 | c2   |
+----+------+
|  1 | Luis |
+----+------+
```

```
mysql> SHOW BINLOG EVENTS;
+---------------+------+----------------+-----------+-------------+-------------------------------------
| Log_name      | Pos  | Event_type     | Server_id | End_log_pos | Info
+---------------+------+----------------+-----------+-------------+-------------------------------------
| binlog.000001 |    4 | Format_desc    |         3 |         123 | Server ver: 8.0.32-log, Binlog ver: 4
| binlog.000001 |  123 | Previous_gtids |         3 |         150 |
| binlog.000001 |  150 | Gtid           |         1 |         211 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  211 | Query          |         1 |         270 | BEGIN
| binlog.000001 |  270 | View_change    |         1 |         369 | view_id=14724832985483517:1
| binlog.000001 |  369 | Query          |         1 |         434 | COMMIT
| binlog.000001 |  434 | Gtid           |         1 |         495 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  495 | Query          |         1 |         585 | CREATE DATABASE test
| binlog.000001 |  585 | Gtid           |         1 |         646 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  646 | Query          |         1 |         770 | use `test`; CREATE TABLE t1 (c1 INT PRI
| binlog.000001 |  770 | Gtid           |         1 |         831 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 |  831 | Query          |         1 |         890 | BEGIN
| binlog.000001 |  890 | Table_map      |         1 |         933 | table_id: 108 (test.t1)
| binlog.000001 |  933 | Write_rows     |         1 |         975 | table_id: 108 flags: STMT_END_F
| binlog.000001 |  975 | Xid            |         1 |        1002 | COMMIT /* xid=29 */
| binlog.000001 | 1002 | Gtid           |         1 |        1063 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 | 1063 | Query          |         1 |        1122 | BEGIN
| binlog.000001 | 1122 | View_change    |         1 |        1261 | view_id=14724832985483517:2
| binlog.000001 | 1261 | Query          |         1 |        1326 | COMMIT
| binlog.000001 | 1326 | Gtid           |         1 |        1387 | SET @@SESSION.GTID_NEXT= 'aaaaaaaa-aaaa
| binlog.000001 | 1387 | Query          |         1 |        1446 | BEGIN
| binlog.000001 | 1446 | View_change    |         1 |        1585 | view_id=14724832985483517:3
| binlog.000001 | 1585 | Query          |         1 |        1650 | COMMIT
+---------------+------+----------------+-----------+-------------+-------------------------------------
```

18.2.2 Deploying Group Replication Locally

The most common way to deploy Group Replication is using multiple server instances, to provide high availability. It is also possible to deploy Group Replication locally, for example for testing purposes. This section explains how you can deploy Group Replication locally.

Important

Group Replication is usually deployed on multiple hosts because this ensures that high-availability is provided. The instructions in this section are not suitable for production deployments because all MySQL server instances are running on the same single host. In the event of failure of this host, the whole group fails. Therefore this information should be used for testing purposes and it should not be used in a production environments.

This section explains how to create a replication group with three MySQL Server instances on one physical machine. This means that three data directories are needed, one per server instance, and that you need to configure each instance independently. This - procedure assumes that MySQL Server was downloaded and unpacked - into the directory named `mysql-8.0`. Each MySQL server instance requires a specific data directory. Create a directory named `data`, then in that directory create a subdirectory for each server instance, for example s1, s2 and s3, and initialize each one.

```
mysql-8.0/bin/mysqld --initialize-insecure --basedir=$PWD/mysql-8.0 --datadir=$PWD/data/s1
mysql-8.0/bin/mysqld --initialize-insecure --basedir=$PWD/mysql-8.0 --datadir=$PWD/data/s2
mysql-8.0/bin/mysqld --initialize-insecure --basedir=$PWD/mysql-8.0 --datadir=$PWD/data/s3
```

Inside `data/s1`, `data/s2`, `data/s3` is an initialized data directory, containing the mysql system database and related tables and much more. To learn more about the initialization procedure, see Section 2.9.1, “Initializing the Data Directory”.

Warning

Do not use `-initialize-insecure` in production environments, it is only used here to simplify the tutorial. For more information on security settings, see Section 18.6, “Group Replication Security”.

Configuration of Local Group Replication Members

When you are following Section 18.2.1.2, “Configuring an Instance for Group Replication”, you need to add configuration for the data directories added in the previous section. For example:

```
[mysqld]

# server configuration
datadir=<full_path_to_data>/data/s1
basedir=<full_path_to_bin>/mysql-8.0/

port=24801
socket=<full_path_to_sock_dir>/s1.sock
```

These settings configure MySQL server to use the data directory created earlier and which port the server should open and start listening for incoming connections.

Note

The non-default port of 24801 is used because in this tutorial the three server instances use the same hostname. In a setup with three different machines this would not be required.

Group Replication requires a network connection between the members, which means that each member must be able to resolve the network address of all of the other members. For example in this tutorial all three instances run on one machine, so to ensure that the members can contact each other you could add a line to the option file such as `report_host=127.0.0.1`.

Then each member needs to be able to connect to the other members on their `group_replication_local_address`. For example in the option file of member s1 add:

```
group_replication_local_address= "127.0.0.1:24901"
group_replication_group_seeds= "127.0.0.1:24901,127.0.0.1:24902,127.0.0.1:24903"
```

This configures s1 to use port 24901 for internal group communication with seed members. For each server instance you want to add to the group, make these changes in the option file of the member. For each member you must ensure a unique address is specified, so use a unique port per instance for `group_replication_local_address`. Usually you want all members to be able to serve as seeds for members that are joining the group and have not got the transactions processed by the group. In this case, add all of the ports to `group_replication_group_seeds` as shown above.

The remaining steps of Section 18.2.1, “Deploying Group Replication in Single-Primary Mode” apply equally to a group which you have deployed locally in this way.

18.3 Requirements and Limitations

This section lists and explains the requirements and limitations of Group Replication.

18.3.1 Group Replication Requirements

- Infrastructure
- Server Instance Configuration

Server instances that you want to use for Group Replication must satisfy the following requirements.

Infrastructure

- **InnoDB Storage Engine.** Data must be stored in the `InnoDB` transactional storage engine. Transactions are executed optimistically and then, at commit time, are checked for conflicts. If there are conflicts, in order to maintain consistency across the group, some transactions are rolled back.

This means that a transactional storage engine is required. Moreover, `InnoDB` provides some additional functionality that enables better management and handling of conflicts when operating together with Group Replication. The use of other storage engines, including the temporary `MEMORY` storage engine, might cause errors in Group Replication. Convert any tables in other storage engines to use `InnoDB` before using the instance with Group Replication. You can prevent the use of other storage engines by setting the `disabled_storage_engines` system variable on group members, for example:

```
disabled_storage_engines="MyISAM,BLACKHOLE,FEDERATED,ARCHIVE,MEMORY"
```

- **Primary Keys.** Every table that is to be replicated by the group must have a defined primary key, or primary key equivalent where the equivalent is a non-null unique key. Such keys are required as a unique identifier for every row within a table, enabling the system to determine which transactions conflict by identifying exactly which rows each transaction has modified. Group Replication has its own built-in set of checks for primary keys or primary key equivalents, and does not use the checks carried out by the `sql_require_primary_key` system variable. You may set `sql_require_primary_key=ON` for a server instance where Group Replication is running, and you may set the `REQUIRE_TABLE_PRIMARY_KEY_CHECK` option of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement to `ON` for a Group Replication channel. However, be aware that you might find some transactions that are permitted under Group Replication's built-in checks are not permitted under the checks carried out when you set `sql_require_primary_key=ON` or `REQUIRE_TABLE_PRIMARY_KEY_CHECK=ON`.

- **Network Performance.** MySQL Group Replication is designed to be deployed in a cluster environment where server instances are very close to each other. The performance and stability of a group can be impacted by both network latency and network bandwidth. Bi-directional communication must be maintained at all times between all group members. If either inbound or outbound communication is blocked for a server instance (for example, by a firewall, or by connectivity issues), the member cannot function in the group, and the group members (including the member with issues) might not be able to report the correct member status for the affected server instance.

  From MySQL 8.0.14, you can use an IPv4 or IPv6 network infrastructure, or a mix of the two, for TCP communication between remote Group Replication servers. There is also nothing preventing Group Replication from operating over a virtual private network (VPN).

  Also from MySQL 8.0.14, where Group Replication server instances are co-located and share a local group communication engine (XCom) instance, a dedicated input channel with lower overhead is used for communication where possible instead of the TCP socket. For certain Group Replication tasks that require communication between remote XCom instances, such as joining a group, the TCP network is still used, so network performance influences the group's performance.

Server Instance Configuration

The following options must be configured as shown on server instances that are members of a group.

- **Unique Server Identifier.** Use the `server_id` system variable to configure the server with a unique server ID, as required for all servers in replication topologies. The server ID must be a positive integer between 1 and (2^{32})−1, and it must be different from every other server ID in use by any other server in the replication topology.

- **Binary Log Active.** Set `--log-bin[=log_file_name]`. From MySQL 8.0, binary logging is enabled by default, and you do not need to specify this option unless you want to change the name of the binary log files. Group Replication replicates the binary log's contents, therefore the binary log needs to be on for it to operate. See Section 5.4.4, “The Binary Log”.

- **Replica Updates Logged.** Set `log_replica_updates=ON` (from MySQL 8.0.26) or `log_slave_updates=ON` (before MySQL 8.0.26). From MySQL 8.0, this setting is the default, so you do not need to specify it. Group members need to log transactions that are received from their donors at joining time and applied through the replication applier, and to log all transactions that they

receive and apply from the group. This enables Group Replication to carry out distributed recovery by state transfer from an existing group member's binary log.

- **Binary Log Row Format.** Set `binlog_format=row`. This setting is the default, so you do not need to specify it. Group Replication relies on the row-based replication format to propagate changes consistently among the servers in the group, and extract the necessary information to detect conflicts among transactions that execute concurrently in different servers in the group. From MySQL 8.0.19, the `REQUIRE_ROW_FORMAT` setting is automatically added to Group Replication's channels to enforce the use of row-based replication when the transactions are applied. See Section 17.2.1, “Replication Formats” and Section 17.3.3, “Replication Privilege Checks”.

- **Binary Log Checksums Off (to MySQL 8.0.20).** Up to and including MySQL 8.0.20, set `binlog_checksum=NONE`. In these releases, Group Replication cannot make use of checksums and does not support their presence in the binary log. From MySQL 8.0.21, Group Replication supports checksums, so group members may use the default setting `binlog_checksum=CRC32`, and you do not need to specify it.

- **Global Transaction Identifiers On.** Set `gtid_mode=ON` and `enforce_gtid_consistency=ON`. These settings are not the defaults. GTID-based replication is required for Group Replication, which uses global transaction identifiers to track the transactions that have been committed on every server instance in the group. See Section 17.1.3, “Replication with Global Transaction Identifiers”.

- **Replication Information Repositories.** Set `master_info_repository=TABLE` and `relay_log_info_repository=TABLE`. In MySQL 8.0, these settings are the default, and the `FILE` setting is deprecated. From MySQL 8.0.23, the use of these system variables is deprecated, so omit the system variables and just allow the default. The replication applier needs to have the replication metadata written to the `mysql.slave_master_info` and `mysql.slave_relay_log_info` system tables to ensure the Group Replication plugin has consistent recoverability and transactional management of the replication metadata. See Section 17.2.4.2, “Replication Metadata Repositories”.

- **Transaction Write Set Extraction.** Set `transaction_write_set_extraction=XXHASH64` so that while collecting rows to log them to the binary log, the server collects the write set as well. In MySQL 8.0, this setting is the default, and from MySQL 8.0.26, the use of the system variable is deprecated. The write set is based on the primary keys of each row and is a simplified and compact view of a tag that uniquely identifies the row that was changed. Group Replication uses this information for conflict detection and certification on all group members.

- **Default Table Encryption.** Set `default_table_encryption` to the same value on all group members. Default schema and tablespace encryption can be either enabled (`ON`) or disabled (`OFF`, the default) as long as the setting is the same on all members.

- **Lower Case Table Names.** Set `lower_case_table_names` to the same value on all group members. A setting of 1 is correct for the use of the `InnoDB` storage engine, which is required for Group Replication. Note that this setting is not the default on all platforms.

- **Binary Log Dependency Tracking.** Setting `binlog_transaction_dependency_tracking=WRITESET_SESSION` can improve performance for a group member, depending on the group's workload. Group Replication carries out its own parallelization after certification when applying transactions from the relay log, independently of the value set for `binlog_transaction_dependency_tracking`. However, the value of `binlog_transaction_dependency_tracking` does affect how transactions are written to the binary logs on Group Replication members. The dependency information in those logs is used to assist the process of state transfer from a donor's binary log for distributed recovery, which takes place whenever a member joins or rejoins the group.

- **Multithreaded Appliers.** Group Replication members can be configured as multithreaded replicas, enabling transactions to be applied in parallel. From MySQL 8.0.27, all replicas are configured as multithreaded by default. A nonzero value for the system variable `replica_parallel_workers` (from MySQL 8.0.26) or `slave_parallel_workers` (before

MySQL 8.0.26) enables the multithreaded applier on the member. The default from MySQL 8.0.27 is 4 parallel applier threads, and up to 1024 parallel applier threads can be specified. For a multithreaded replica, the following settings are also required, which are the defaults from MySQL 8.0.27:

`replica_preserve_commit_order=ON` (from MySQL 8.0.26) or `slave_preserve_commit_order=ON` (before MySQL 8.0.26)
: This setting is required to ensure that the final commit of parallel transactions is in the same order as the original transactions. Group Replication relies on consistency mechanisms built around the guarantee that all participating members receive and apply committed transactions in the same order.

`replica_parallel_type=LOGICAL_CLOCK` (from MySQL 8.0.26) or `slave_parallel_type=LOGICAL_CLOCK` (before MySQL 8.0.26)
: This setting is required with `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON`. It specifies the policy used to decide which transactions are allowed to execute in parallel on the replica.

Setting `replica_parallel_workers=0` or `slave_parallel_workers=0` disables parallel execution and gives the replica a single applier thread and no coordinator thread. With that setting, the `replica_parallel_type` or `slave_parallel_type` and `replica_preserve_commit_order` or `slave_preserve_commit_order` options have no effect and are ignored. From MySQL 8.0.27, if parallel execution is disabled when GTIDs are in use on a replica, the replica actually uses one parallel worker, to take advantage of the method for retrying transactions without accessing the file positions. However, this behavior does not change anything for the user.

- **Detached XA transactions.** MySQL 8.0.29 and later supports detached XA transactions. A detached transaction is one which, once prepared, is no longer connected to the current session. This happens automatically as part of executing `XA PREPARE`. The prepared XA transaction can be committed or rolled back by another connection, and the current session can then initiate another XA transaction or local transaction without waiting for the transaction that was just prepared to complete.

  When detached XA transaction support is enabled (`xa_detach_on_prepare = ON`) it is possible for any connection to this server to list (using `XA RECOVER`), roll back, or commit any prepared XA transaction. In addition, you cannot use temporary tables within detached XA transactions.

  You can disable support for detached XA transactions by setting `xa_detach_on_prepare` to `OFF`, but this is not recommended. In particular, if this server is being set up as an instance in MySQL group replication, you should leave this variable set to its default value (`ON`).

  See Section 13.3.8.2, “XA Transaction States”, for more information.

18.3.2 Group Replication Limitations

The following known limitations exist for Group Replication. Note that the limitations and issues described for multi-primary mode groups can also apply in single-primary mode clusters during a failover event, while the newly elected primary flushes out its applier queue from the old primary.

Tip

Group Replication is built on GTID based replication, therefore you should also be aware of Section 17.1.3.7, “Restrictions on Replication with GTIDs”.

- **`--upgrade=MINIMAL` option.** Group Replication cannot be started following a MySQL Server upgrade that uses the MINIMAL option (`--upgrade=MINIMAL`), which does not upgrade system tables on which the replication internals depend.

- **Gap Locks.** Group Replication's certification process for concurrent transactions does not take into account gap locks, as information about gap locks is not available outside of `InnoDB`. See Gap Locks for more information.

  Note

  For a group in multi-primary mode, unless you rely on `REPEATABLE READ` semantics in your applications, we recommend using the `READ COMMITTED` isolation level with Group Replication. InnoDB does not use gap locks in `READ COMMITTED`, which aligns the local conflict detection within InnoDB with the distributed conflict detection performed by Group Replication. For a group in single-primary mode, only the primary accepts writes, so the `READ COMMITTED` isolation level is not important to Group Replication.

- **Table Locks and Named Locks.** The certification process does not take into account table locks (see Section 13.3.6, "LOCK TABLES and UNLOCK TABLES Statements") or named locks (see `GET_LOCK()`).

- **Binary Log Checksums.** Up to and including MySQL 8.0.20, Group Replication cannot make use of checksums and does not support their presence in the binary log, so you must set `binlog_checksum=NONE` when configuring a server instance to become a group member. From MySQL 8.0.21, Group Replication supports checksums, so group members may use the default setting `binlog_checksum=CRC32`. The setting for `binlog_checksum` does not have to be the same for all members of a group.

  When checksums are available, Group Replication does not use them to verify incoming events on the `group_replication_applier` channel, because events are written to that relay log from multiple sources and before they are actually written to the originating server's binary log, which is when a checksum is generated. Checksums are used to verify the integrity of events on the `group_replication_recovery` channel and on any other replication channels on group members.

- **SERIALIZABLE Isolation Level.** `SERIALIZABLE` isolation level is not supported in multi-primary groups by default. Setting a transaction isolation level to `SERIALIZABLE` configures Group Replication to refuse to commit the transaction.

- **Concurrent DDL versus DML Operations.** Concurrent data definition statements and data manipulation statements executing against the same object but on different servers is not supported when using multi-primary mode. During execution of Data Definition Language (DDL) statements on an object, executing concurrent Data Manipulation Language (DML) on the same object but on a different server instance has the risk of conflicting DDL executing on different instances not being detected.

- **Foreign Keys with Cascading Constraints.** Multi-primary mode groups (members all configured with `group_replication_single_primary_mode=OFF`) do not support tables with multi-level foreign key dependencies, specifically tables that have defined `CASCADING` foreign key constraints. This is because foreign key constraints that result in cascading operations executed by a multi-primary mode group can result in undetected conflicts and lead to inconsistent data across the members of the group. Therefore we recommend setting `group_replication_enforce_update_everywhere_checks=ON` on server instances used in multi-primary mode groups to avoid undetected conflicts.

  In single-primary mode this is not a problem as it does not allow concurrent writes to multiple members of the group and thus there is no risk of undetected conflicts.

- **Multi-primary Mode Deadlock.** When a group is operating in multi-primary mode, `SELECT .. FOR UPDATE` statements can result in a deadlock. This is because the lock is not shared across the members of the group, therefore the expectation for such a statement might not be reached.

- **Replication Filters.** Global replication filters cannot be used on a MySQL server instance that is configured for Group Replication, because filtering transactions on some servers would make the

group unable to reach agreement on a consistent state. Channel specific replication filters can be used on replication channels that are not directly involved with Group Replication, such as where a group member also acts as a replica to a source that is outside the group. They cannot be used on the `group_replication_applier` or `group_replication_recovery` channels.

- **Encrypted Connections.** Support for the TLSv1.3 protocol is available in MySQL Server as of MySQL 8.0.16, provided that MySQL was compiled using OpenSSL 1.1.1 or higher. In MySQL 8.0.16 and MySQL 8.0.17, if the server supports TLSv1.3, the protocol is not supported in the group communication engine and cannot be used by Group Replication. Group Replication supports TLSv1.3 from MySQL 8.0.18, where it can be used for group communication connections and distributed recovery connections.

  In MySQL 8.0.18, TLSv1.3 can be used in Group Replication for the distributed recovery connection, but the `group_replication_recovery_tls_version` and `group_replication_recovery_tls_ciphersuites` system variables are not available. The donor servers must therefore permit the use of at least one TLSv1.3 ciphersuite that is enabled by default, as listed in Section 6.3.2, “Encrypted Connection TLS Protocols and Ciphers”. From MySQL 8.0.19, you can use the options to configure client support for any selection of ciphersuites, including only non-default ciphersuites if you want.

- **Cloning Operations.** Group Replication initiates and manages cloning operations for distributed recovery, but group members that have been set up to support cloning may also participate in cloning operations that a user initiates manually. In releases before MySQL 8.0.20, you cannot initiate a cloning operation manually if the operation involves a group member on which Group Replication is running. From MySQL 8.0.20, you can do this, provided that the cloning operation does not remove and replace the data on the recipient. The statement to initiate the cloning operation must therefore include the `DATA DIRECTORY` clause if Group Replication is running. See Cloning for Other Purposes.

Limit on Group Size

The maximum number of MySQL servers that can be members of a single replication group is 9. If further members attempt to join the group, their request is refused. This limit has been identified from testing and benchmarking as a safe boundary where the group performs reliably on a stable local area network.

Limits on Transaction Size

If an individual transaction results in message contents which are large enough that the message cannot be copied between group members over the network within a 5-second window, members can be suspected of having failed, and then expelled, just because they are busy processing the transaction. Large transactions can also cause the system to slow due to problems with memory allocation. To avoid these issues use the following mitigations:

- If unnecessary expulsions occur due to large messages, use the system variable `group_replication_member_expel_timeout` to allow additional time before a member under suspicion of having failed is expelled. You can allow up to an hour after the initial 5-second detection period before a suspect member is expelled from the group. From MySQL 8.0.21, an additional 5 seconds is allowed by default.

- Where possible, try and limit the size of your transactions before they are handled by Group Replication. For example, split up files used with `LOAD DATA` into smaller chunks.

- Use the system variable `group_replication_transaction_size_limit` to specify a maximum transaction size that the group accepts. In MySQL 8.0, this system variable defaults to a maximum transaction size of 150000000 bytes (approximately 143 MB). Transactions above this size are rolled back and are not sent to Group Replication's Group Communication System (GCS) for distribution to the group. Adjust the value of this variable depending on the maximum message size that you need the group to tolerate, bearing in mind that the time taken to process a transaction is proportional to its size.

- Use the system variable `group_replication_compression_threshold` to specify a message size above which compression is applied. This system variable defaults to 1000000 bytes (1 MB), so large messages are automatically compressed. Compression is carried out by Group Replication's Group Communication System (GCS) when it receives a message that was permitted by the `group_replication_transaction_size_limit` setting but exceeds the `group_replication_compression_threshold` setting. For more information, see Section 18.7.4, “Message Compression”.

- Use the system variable `group_replication_communication_max_message_size` to specify a message size above which messages are fragmented. This system variable defaults to 10485760 bytes (10 MiB), so large messages are automatically fragmented. GCS carries out fragmentation after compression if the compressed message still exceeds the `group_replication_communication_max_message_size` limit. In order for a replication group to use fragmentation, all group members must be at MySQL 8.0.16 or above, and the Group Replication communication protocol version in use by the group must allow fragmentation. For more information, see Section 18.7.5, “Message Fragmentation”.

The maximum transaction size, message compression, and message fragmentation can all be deactivated by specifying a zero value for the relevant system variable. If you have deactivated all these safeguards, the upper size limit for a message that can be handled by the applier thread on a member of a replication group is the value of the member's `replica_max_allowed_packet` or `slave_max_allowed_packet` system variable, which have a default and maximum value of 1073741824 bytes (1 GB). A message that exceeds this limit fails when the receiving member attempts to handle it. The upper size limit for a message that a group member can originate and attempt to transmit to the group is 4294967295 bytes (approximately 4 GB). This is a hard limit on the packet size that is accepted by the group communication engine for Group Replication (XCom, a Paxos variant), which receives messages after GCS has handled them. A message that exceeds this limit fails when the originating member attempts to broadcast it.

18.4 Monitoring Group Replication

You can use the MySQL Performance Schema to monitor Group Replication. These Performance Schema tables display information specific to Group Replication:

- `replication_group_member_stats`: See Section 18.4.3, “The replication_group_members Table”, for more information.

- `replication_group_members`: See Section 18.4.4, “The replication_group_member_stats Table”, for more information.

- `replication_group_communication_information`: See Section 27.12.11.15, “The replication_group_communication_information Table”, for more information.

These Performance Schema replication tables also show information relating to Group Replication:

- `replication_connection_status` shows information regarding Group Replication, such as transactions received from the group and queued in the applier queue (relay log).

- `replication_applier_status` shows the states of channels and threads relating to Group Replication. These can also be used to monitor what individual worker threads are doing.

Replication channels created by the Group Replication plugin are listed here:

- `group_replication_recovery`: Used for replication changes related to distributed recovery.

- `group_replication_applier`: Used for the incoming changes from the group, to apply transactions coming directly from the group.

Beginning with MySQL 8.0.21, messages relating to Group Replication lifecycle events other than errors are classified as system messages; these are always written to the replication group

member' error log. You can use this information to review the history of a given server's membership in a replication group. (Previously, such events were classified as information messages; for a MySQL server from a release prior to 8.0.21, these can be added to the error log by setting `log_error_verbosity` to `3`.)

Some lifecycle events that affect the whole group are logged on every group member, such as a new member entering `ONLINE` status in the group or a primary election. Other events are logged only on the member where they take place, such as super read only mode being enabled or disabled on the member, or the member leaving the group. A number of lifecycle events that can indicate an issue if they occur frequently are logged as warning messages, including a member becoming unreachable and then reachable again, and a member starting distributed recovery by state transfer from the binary log or by a remote cloning operation.

Note

If you are monitoring one or more secondary instances using `mysqladmin`, you should be aware that a `FLUSH STATUS` statement executed by this utility creates a GTID event on the local instance which may impact future group operations.

18.4.1 GTIDs and Group Replication

Group Replication uses GTIDs (global transaction identifiers) to track exactly which transactions have been committed on every server instance. The settings `gtid_mode=ON` and `enforce_gtid_consistency=ON` are required on all group members. Incoming transactions from clients are assigned a GTID by the group member that receives them. Any replicated transactions that are received by group members on asynchronous replication channels from source servers outside the group retain the GTIDs that they have when they arrive on the group member.

The GTIDs that are assigned to incoming transactions from clients use the group name specified by the `group_replication_group_name` system variable as the UUID part of the identifier, rather than the server UUID of the individual group member that received the transaction. All the transactions received directly by the group can therefore be identified and are grouped together in GTID sets, and it does not matter which member originally received them. Each group member has a block of consecutive GTIDs reserved for its use, and when these are consumed it reserves more. The `group_replication_gtid_assignment_block_size` system variable sets the size of the blocks, with a default of 1 million GTIDs in each block.

View change events (`View_change_log_event`), which are generated by the group itself when a new member joins, are given GTIDs when they are recorded in the binary log. By default, the GTIDs for these events also use the group name specified by the `group_replication_group_name` system variable as the UUID part of the identifier. From MySQL 8.0.26, you can set the Group Replication system variable `group_replication_view_change_uuid` to use an alternative UUID in the GTIDs for view change events, so that they are easy to distinguish from transactions received by the group from clients. This can be useful if your setup allows for failover between groups, and you need to identify and discard transactions that were specific to the backup group. The alternative UUID must be different from the server UUIDs of the members. It must also be different from any UUIDs in the GTIDs applied to anonymous transactions using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement.

From MySQL 8.0.27, the settings `GTID_ONLY=1`, `REQUIRE_ROW_FORMAT = 1`, and `SOURCE_AUTO_POSITION = 1` are applied for the Group Replication channels `group_replication_applier` and `group_replication_recovery`. The settings are made automatically on the Group Replication channels when they are created, or when a member server in a replication group is upgraded to 8.0.27 or higher. These options are normally set using a `CHANGE REPLICATION SOURCE TO` statement, but note that you cannot disable them for a Group Replication channel. With these options set, the group member does not persist file names and file positions in the replication metadata repositories for these channels. GTID auto-positioning and GTID auto-skip are used to locate the correct receiver and applier positions when necessary.