MySQL Replication
Abstract

This is the MySQL Replication extract from the MySQL 5.7 Reference Manual.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2020-05-01 (revision: 65870)
# Table of Contents

Preface and Legal Notices ........................................................................................................ vii  
1 Replication .......................................................................................................................... 1  
2 Configuring Replication ........................................................................................................ 3  
   2.1 Binary Log File Position Based Replication Configuration Overview .................... 4  
   2.2 Setting Up Binary Log File Position Based Replication .................................................. 4  
      2.2.1 Setting the Replication Master Configuration .................................................... 5  
      2.2.2 Creating a User for Replication ........................................................................... 6  
      2.2.3 Obtaining the Replication Master Binary Log Coordinates ................................ 6  
      2.2.4 Choosing a Method for Data Snapshots .............................................................. 7  
      2.2.5 Setting Up Replication Slaves ............................................................................. 9  
      2.2.6 Adding Slaves to a Replication Environment ..................................................... 12  
   2.3 Replication with Global Transaction Identifiers ......................................................... 13  
      2.3.1 GTID Format and Storage .................................................................................. 14  
      2.3.2 GTID Life Cycle .................................................................................................. 18  
      2.3.3 GTID Auto-Positioning ....................................................................................... 22  
      2.3.4 Setting Up Replication Using GTIDs ................................................................. 23  
      2.3.5 Using GTIDs for Failover and Scaleout ............................................................... 25  
      2.3.6 Restrictions on Replication with GTIDs ............................................................ 28  
      2.3.7 Stored Function Examples to Manipulate GTIDs ................................................ 29  
      2.3.8 Using GTIDs for Failover and Scaleout ............................................................... 29  
   2.4 MySQL Multi-Source Replication .................................................................................. 34  
      2.4.1 Configuring Multi-Source Replication ................................................................. 34  
      2.4.2 Provisioning a Multi-Source Replication Slave for GTID-Based Replication .... 35  
      2.4.3 Adding GTID-Based Masters to a Multi-Source Replication Slave ................. 36  
      2.4.4 Adding a Binary Log Based Master to a Multi-Source Replication Slave ........... 37  
      2.4.5 Starting Multi-Source Replication Slaves ........................................................... 37  
      2.4.6 Stopping Multi-Source Replication Slaves .......................................................... 38  
      2.4.7 Resetting Multi-Source Replication Slaves ........................................................ 38  
      2.4.8 Multi-Source Replication Monitoring ................................................................. 38  
   2.5 Changing Replication Modes on Online Servers ......................................................... 39  
      2.5.1 Replication Mode Concepts ............................................................................... 40  
      2.5.2 Enabling GTID Transactions Online ................................................................. 42  
      2.5.3 Disabling GTID Transactions Online ................................................................. 43  
      2.5.4 Verifying Replication of Anonymous Transactions ......................................... 44  
   2.6 Replication and Binary Logging Options and Variables ............................................... 45  
      2.6.1 Replication and Binary Logging Option and Variable Reference ....................... 47  
      2.6.2 Replication Master Options and Variables ......................................................... 53  
      2.6.3 Replication Slave Options and Variables ........................................................... 60  
      2.6.4 Binary Logging Options and Variables ............................................................. 91  
      2.6.5 Global Transaction ID System Variables .......................................................... 112  
   2.7 Common Replication Administration Tasks .................................................................. 119  
      2.7.1 Checking Replication Status ............................................................................. 119  
      2.7.2 Pausing Replication on the Slave ...................................................................... 122  
      2.7.3 Skipping Transactions ...................................................................................... 122  
   3 Replication Solutions ........................................................................................................ 127  
      3.1 Using Replication for Backups ................................................................................... 127  
         3.1.1 Backing Up a Slave Using mysqlslap ......................................................... 127  
         3.1.2 Backing Up Raw Data from a Slave ................................................................. 128  
         3.1.3 Backing Up a Master or Slave by Making It Read Only .................................. 129  
      3.2 Handling an Unexpected Halt of a Replication Slave ............................................... 131  
      3.3 Using Replication with Different Master and Slave Storage Engines ...................... 133  
      3.4 Using Replication for Scale-Out ............................................................................. 134  
      3.5 Replicating Different Databases to Different Slaves ............................................... 135  
      3.6 Improving Replication Performance ...................................................................... 136  
      3.7 Switching Masters During Failover ....................................................................... 137  
      3.8 Setting Up Replication to Use Encrypted Connections ......................................... 139
Preface and Legal Notices

This is the MySQL Replication extract from the MySQL 5.7 Reference Manual.

**Licensing information—MySQL 5.7.** This product may include third-party software, used under license. If you are using a Commercial release of MySQL 5.7, see the MySQL 5.7 Commercial Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of MySQL 5.7, see the MySQL 5.7 Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

**Licensing information—MySQL NDB Cluster 7.5.** This product may include third-party software, used under license. If you are using a Commercial release of NDB Cluster 7.5, see the MySQL NDB Cluster 7.5 Commercial Release License Information User Manual for licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of NDB Cluster 7.5, see the MySQL NDB Cluster 7.5 Community Release License Information User Manual for licensing information relating to third-party software that may be included in this Community release.

**Licensing information—MySQL NDB Cluster 7.6.** If you are using a Commercial release of MySQL NDB Cluster 7.6, see the MySQL NDB Cluster 7.6 Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Commercial release. If you are using a Community release of MySQL NDB Cluster 7.6, see the MySQL NDB Cluster 7.6 Community Release License Information User Manual for licensing information, including licensing information relating to third-party software that may be included in this Community release.

**Legal Notices**

Copyright © 1997, 2020, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

**U.S. GOVERNMENT END USERS:** Oracle programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, delivered to U.S. Government end users are "commercial computer software" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, use, duplication, disclosure, modification, and adaptation of the programs, including any operating system, integrated software, any programs installed on the hardware, and/or documentation, shall be subject to license terms and license restrictions applicable to the programs. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and
other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.
Chapter 1 Replication

Replication enables data from one MySQL database server (the master) to be copied to one or more MySQL database servers (the slaves). Replication is asynchronous by default; slaves do not need to be connected permanently to receive updates from the master. Depending on the configuration, you can replicate all databases, selected databases, or even selected tables within a database.

Advantages of replication in MySQL include:

- Scale-out solutions - spreading the load among multiple slaves to improve performance. In this environment, all writes and updates must take place on the master server. Reads, however, may take place on one or more slaves. This model can improve the performance of writes (since the master is dedicated to updates), while dramatically increasing read speed across an increasing number of slaves.
- Data security - because data is replicated to the slave, and the slave can pause the replication process, it is possible to run backup services on the slave without corrupting the corresponding master data.
- Analytics - live data can be created on the master, while the analysis of the information can take place on the slave without affecting the performance of the master.
- Long-distance data distribution - you can use replication to create a local copy of data for a remote site to use, without permanent access to the master.

For information on how to use replication in such scenarios, see Chapter 3, Replication Solutions.

MySQL 5.7 supports different methods of replication. The traditional method is based on replicating events from the master's binary log, and requires the log files and positions in them to be synchronized between master and slave. The newer method based on global transaction identifiers (GTIDs) is transactional and therefore does not require working with log files or positions within these files, which greatly simplifies many common replication tasks. Replication using GTIDs guarantees consistency between master and slave as long as all transactions committed on the master have also been applied on the slave. For more information about GTIDs and GTID-based replication in MySQL, see Section 2.3, "Replication with Global Transaction Identifiers". For information on using binary log file position based replication, see Chapter 2, Configuring Replication.

Replication in MySQL supports different types of synchronization. The original type of synchronization is one-way, asynchronous replication, in which one server acts as the master, while one or more other servers act as slaves. This is in contrast to the synchronous replication which is a characteristic of NDB Cluster (see MySQL NDB Cluster 7.5 and NDB Cluster 7.6). In MySQL 5.7, semisynchronous replication is supported in addition to the built-in asynchronous replication. With semisynchronous replication, a commit performed on the master blocks before returning to the session that performed the transaction until at least one slave acknowledges that it has received and logged the events for the transaction; see Section 3.9, "Semisynchronous Replication". MySQL 5.7 also supports delayed replication such that a slave server deliberately lags behind the master by at least a specified amount of time; see Section 3.10, "Delayed Replication". For scenarios where synchronous replication is required, use NDB Cluster (see MySQL NDB Cluster 7.5 and NDB Cluster 7.6).

There are a number of solutions available for setting up replication between servers, and the best method to use depends on the presence of data and the engine types you are using. For more information on the available options, see Section 2.2, "Setting Up Binary Log File Position Based Replication".

There are two core types of replication format, Statement Based Replication (SBR), which replicates entire SQL statements, and Row Based Replication (RBR), which replicates only the changed rows. You can also use a third variety, Mixed Based Replication (MBR). For more information on the different replication formats, see Section 5.1, "Replication Formats".

Replication is controlled through a number of different options and variables. For more information, see Section 2.6, "Replication and Binary Logging Options and Variables".
You can use replication to solve a number of different problems, including performance, supporting the backup of different databases, and as part of a larger solution to alleviate system failures. For information on how to address these issues, see Chapter 3, *Replication Solutions*.

For notes and tips on how different data types and statements are treated during replication, including details of replication features, version compatibility, upgrades, and potential problems and their resolution, see Chapter 4, *Replication Notes and Tips*. For answers to some questions often asked by those who are new to MySQL Replication, see MySQL 5.7 FAQ: Replication.

For detailed information on the implementation of replication, how replication works, the process and contents of the binary log, background threads and the rules used to decide how statements are recorded and replicated, see Chapter 5, *Replication Implementation*. 
Chapter 2 Configuring Replication

Table of Contents

2.1 Binary Log File Position Based Replication Configuration Overview ........................................... 4
2.2 Setting Up Binary Log File Position Based Replication ............................................................... 4
  2.2.1 Setting the Replication Master Configuration ................................................................. 5
  2.2.2 Creating a User for Replication ......................................................................................... 6
  2.2.3 Obtaining the Replication Master Binary Log Coordinates ............................................. 6
  2.2.4 Choosing a Method for Data Snapshots ........................................................................... 7
  2.2.5 Setting Up Replication Slaves ........................................................................................... 9
  2.2.6 Adding Slaves to a Replication Environment ..................................................................... 12
2.3 Replication with Global Transaction Identifiers ............................................................................ 13
  2.3.1 GTID Format and Storage ................................................................................................. 14
  2.3.2 GTID Life Cycle ............................................................................................................... 18
  2.3.3 GTID Auto-Positioning ..................................................................................................... 22
  2.3.4 Setting Up Replication Using GTIDs ............................................................................... 23
  2.3.5 Using GTIDs for Failover and Scaleout ........................................................................... 25
  2.3.6 Restrictions on Replication with GTIDs ........................................................................... 28
  2.3.7 Stored Function Examples to Manipulate GTIDs ............................................................... 29
2.4 MySQL Multi-Source Replication ................................................................................................. 34
  2.4.1 Configuring Multi-Source Replication .............................................................................. 34
  2.4.2 Provisioning a Multi-Source Replication Slave for GTID-Based Replication .................... 35
  2.4.3 Adding GTID-Based Masters to a Multi-Source Replication Slave .................................... 36
  2.4.4 Adding a Binary Log Based Master to a Multi-Source Replication Slave ......................... 37
  2.4.5 Starting Multi-Source Replication Slaves ........................................................................ 37
  2.4.6 Stopping Multi-Source Replication Slaves ....................................................................... 38
  2.4.7 Resetting Multi-Source Replication Slaves ...................................................................... 38
  2.4.8 Multi-Source Replication Monitoring ............................................................................. 38
2.5 Changing Replication Modes on Online Servers ........................................................................ 39
  2.5.1 Replication Mode Concepts .............................................................................................. 40
  2.5.2 Enabling GTID Transactions Online ................................................................................. 42
  2.5.3 Disabling GTID Transactions Online ................................................................................ 43
  2.5.4 Verifying Replication of Anonymous Transactions ........................................................... 44
2.6 Replication and Binary Logging Options and Variables ............................................................... 45
  2.6.1 Replication and Binary Logging Option and Variable Reference ......................................... 47
  2.6.2 Replication Master Options and Variables ......................................................................... 53
  2.6.3 Replication Slave Options and Variables .......................................................................... 60
  2.6.4 Binary Logging Options and Variables .............................................................................. 91
  2.6.5 Global Transaction ID System Variables .......................................................................... 112
2.7 Common Replication Administration Tasks .................................................................................. 119
  2.7.1 Checking Replication Status ............................................................................................. 119
  2.7.2 Pausing Replication on the Slave ....................................................................................... 122
  2.7.3 Skipping Transactions ........................................................................................................ 122

This section describes how to configure the different types of replication available in MySQL and includes the setup and configuration required for a replication environment, including step-by-step instructions for creating a new replication environment. The major components of this section are:

- For a guide to setting up two or more servers for replication using binary log file positions, Section 2.2, “Setting Up Binary Log File Position Based Replication”, deals with the configuration of the servers and provides methods for copying data between the master and slaves.
- For a guide to setting up two or more servers for replication using GTID transactions, Section 2.3, “Replication with Global Transaction Identifiers”, deals with the configuration of the servers.
• Events in the binary log are recorded using a number of formats. These are referred to as statement-based replication (SBR) or row-based replication (RBR). A third type, mixed-format replication (MIXED), uses SBR or RBR replication automatically to take advantage of the benefits of both SBR and RBR formats when appropriate. The different formats are discussed in Section 5.1, “Replication Formats”.

• Detailed information on the different configuration options and variables that apply to replication is provided in Section 2.6, “Replication and Binary Logging Options and Variables”.

• Once started, the replication process should require little administration or monitoring. However, for advice on common tasks that you may want to execute, see Section 2.7, “Common Replication Administration Tasks”.

2.1 Binary Log File Position Based Replication Configuration Overview

This section describes replication between MySQL servers based on the binary log file position method, where the MySQL instance operating as the master (the source of the database changes) writes updates and changes as “events” to the binary log. The information in the binary log is stored in different logging formats according to the database changes being recorded. Slaves are configured to read the binary log from the master and to execute the events in the binary log on the slave’s local database.

Each slave receives a copy of the entire contents of the binary log. It is the responsibility of the slave to decide which statements in the binary log should be executed. Unless you specify otherwise, all events in the master binary log are executed on the slave. If required, you can configure the slave to process only events that apply to particular databases or tables.

Important
You cannot configure the master to log only certain events.

Each slave keeps a record of the binary log coordinates: the file name and position within the file that it has read and processed from the master. This means that multiple slaves can be connected to the master and executing different parts of the same binary log. Because the slaves control this process, individual slaves can be connected and disconnected from the server without affecting the master’s operation. Also, because each slave records the current position within the binary log, it is possible for slaves to be disconnected, reconnect and then resume processing.

The master and each slave must be configured with a unique ID (using the server_id system variable). In addition, each slave must be configured with information about the master host name, log file name, and position within that file. These details can be controlled from within a MySQL session using the CHANGE MASTER TO statement on the slave. The details are stored within the slave’s master info repository, which can be either a file or a table (see Section 5.4, “Replication Relay and Status Logs”).

2.2 Setting Up Binary Log File Position Based Replication

This section describes how to set up a MySQL server to use binary log file position based replication. There are a number of different methods for setting up replication, and the exact method to use depends on how you are setting up replication, and whether you already have data within your master database.

There are some generic tasks that are common to all setups:

• On the master, you must enable binary logging and configure a unique server ID. This might require a server restart. See Section 2.2.1, “Setting the Replication Master Configuration”.

• On each slave that you want to connect to the master, you must configure a unique server ID. This might require a server restart. See Section 2.2.5.1, “Setting the Replication Slave Configuration”.
• Optionally, create a separate user for your slaves to use during authentication with the master when reading the binary log for replication. See Section 2.2.2, “Creating a User for Replication”.

• Before creating a data snapshot or starting the replication process, on the master you should record the current position in the binary log. You need this information when configuring the slave so that the slave knows where within the binary log to start executing events. See Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”.

• If you already have data on the master and want to use it to synchronize the slave, you need to create a data snapshot to copy the data to the slave. The storage engine you are using has an impact on how you create the snapshot. When you are using MyISAM, you must stop processing statements on the master to obtain a read-lock, then obtain its current binary log coordinates and dump its data, before permitting the master to continue executing statements. If you do not stop the execution of statements, the data dump and the master status information will not match, resulting in inconsistent or corrupted databases on the slaves. For more information on replicating a MyISAM master, see Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”. If you are using InnoDB, you do not need a read-lock and a transaction that is long enough to transfer the data snapshot is sufficient. For more information, see InnoDB and MySQL Replication.

• Configure the slave with settings for connecting to the master, such as the host name, login credentials, and binary log file name and position. See Section 2.2.5.2, “Setting the Master Configuration on the Slave”.

Note
Certain steps within the setup process require the SUPER privilege. If you do not have this privilege, it might not be possible to enable replication.

After configuring the basic options, select your scenario:

• To set up replication for a fresh installation of a master and slaves that contain no data, see Section 2.2.5.3, “Setting Up Replication between a New Master and Slaves”.

• To set up replication of a new master using the data from an existing MySQL server, see Section 2.2.5.4, “Setting Up Replication with Existing Data”.

• To add replication slaves to an existing replication environment, see Section 2.2.6, “Adding Slaves to a Replication Environment”.

Before administering MySQL replication servers, read this entire chapter and try all statements mentioned in SQL Statements for Controlling Master Servers, and SQL Statements for Controlling Slave Servers. Also familiarize yourself with the replication startup options described in Section 2.6, “Replication and Binary Logging Options and Variables”.

2.2.1 Setting the Replication Master Configuration

To configure a master to use binary log file position based replication, you must enable binary logging and establish a unique server ID. If this has not already been done, a server restart is required.

Binary logging must be enabled on the master because the binary log is the basis for replicating changes from the master to its slaves. If binary logging is not enabled on the master using the \texttt{log-bin} option, replication is not possible.

Each server within a replication group must be configured with a unique server ID. This ID is used to identify individual servers within the group, and must be a positive integer between 1 and \(2^{32}-1\). How you organize and select the numbers is your choice.

To configure the binary log and server ID options, shut down the MySQL server and edit the \texttt{my.cnf} or \texttt{my.ini} file. Within the [mysqld] section of the configuration file, add the \texttt{log-bin} and \texttt{server-id} options. If these options already exist, but are commented out, uncomment the options and alter them according to your needs. For example, to enable binary logging using a log file name prefix of \texttt{mysql-bin}, and configure a server ID of 1, use these lines:
2.2.2 Creating a User for Replication

Each slave connects to the master using a MySQL user name and password, so there must be a user account on the master that the slave can use to connect. The user name is specified by the MASTER_USER option on the CHANGE MASTER TO command when you set up a replication slave. Any account can be used for this operation, providing it has been granted the REPLICATION_SLAVE privilege. You can choose to create a different account for each slave, or connect to the master using the same account for each slave.

Although you do not have to create an account specifically for replication, you should be aware that the replication user name and password are stored in plain text in the master info repository file or table (see Section 5.4.2, “Slave Status Logs”). Therefore, you may want to create a separate account that has privileges only for the replication process, to minimize the possibility of compromise to other accounts.

To create a new account, use CREATE USER. To grant this account the privileges required for replication, use the GRANT statement. If you create an account solely for the purposes of replication, that account needs only the REPLICATION_SLAVE privilege. For example, to set up a new user, repl, that can connect for replication from any host within the example.com domain, issue these statements on the master:

```
mysql> CREATE USER 'repl'@'%.example.com' IDENTIFIED BY 'password';
```
```
mysql> GRANT REPLICATION_SLAVE ON *.* TO 'repl'@'%.example.com';
```

See Account Management Statements, for more information on statements for manipulation of user accounts.

2.2.3 Obtaining the Replication Master Binary Log Coordinates

To configure the slave to start the replication process at the correct point, you need to note the master’s current coordinates within its binary log.

```
CREATE USER repl@%.example.com IDENTIFIED BY 'password';
```
```
GRANT REPLICATION_SLAVE ON *.* TO repl@%.example.com;
```

If you are planning to shut down the master to create a data snapshot, you can optionally skip this procedure and instead store a copy of the binary log index file along with the data snapshot. In that case, remember to use the same account name and password when you create the slave on the new master.
Choosing a Method for Data Snapshots

When creating a new binary log file on restart, the master creates a new binary log file on restart. The master binary log coordinates where the slave must start the replication process are therefore the start of that new file, which is the next binary log file on the master following after the files that are listed in the copied binary log index file.

To obtain the master binary log coordinates, follow these steps:

1. Start a session on the master by connecting to it with the command-line client, and flush all tables and block write statements by executing the `FLUSH TABLES WITH READ LOCK` statement:

   
   ```
   mysql> FLUSH TABLES WITH READ LOCK;
   ```
   
   **Warning**
   Leave the client from which you issued the `FLUSH TABLES` statement running so that the read lock remains in effect. If you exit the client, the lock is released.

2. In a different session on the master, use the `SHOW MASTER STATUS` statement to determine the current binary log file name and position:

   
   ```
   mysql > SHOW MASTER STATUS;
   +------------------+----------+--------------+------------------+
   | File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
   +------------------+----------+--------------+------------------+
   | mysql-bin.000003 | 73       | test         | manual,mysql     |
   +------------------+----------+--------------+------------------+
   ```

   The `File` column shows the name of the log file and the `Position` column shows the position within the file. In this example, the binary log file is `mysql-bin.000003` and the position is 73. Record these values. You need them later when you are setting up the slave. They represent the replication coordinates at which the slave should begin processing new updates from the master.

   If the master has been running previously without binary logging enabled, the log file name and position values displayed by `SHOW MASTER STATUS` or `mysqldump --master-data` are empty. In that case, the values that you need to use later when specifying the slave’s log file and position are the empty string (`''`) and 4.

   You now have the information you need to enable the slave to start reading from the binary log in the correct place to start replication.

   The next step depends on whether you have existing data on the master. Choose one of the following options:

   • If you have existing data that needs to be synchronized with the slave before you start replication, leave the client running so that the lock remains in place. This prevents any further changes being made, so that the data copied to the slave is in synchrony with the master. Proceed to Section 2.2.4, “Choosing a Method for Data Snapshots”.

   • If you are setting up a new master and slave replication group, you can exit the first session to release the read lock. See Section 2.2.5.3, “Setting Up Replication between a New Master and Slaves” for how to proceed.

### 2.2.4 Choosing a Method for Data Snapshots

If the master database contains existing data it is necessary to copy this data to each slave. There are different ways to dump the data from the master database. The following sections describe possible options.

To select the appropriate method of dumping the database, choose between these options:

• Use the `mysqldump` tool to create a dump of all the databases you want to replicate. This is the recommended method, especially when using InnoDB.
Choosing a Method for Data Snapshots

- If your database is stored in binary portable files, you can copy the raw data files to a slave. This can be more efficient than using `mysqldump` and importing the file on each slave, because it skips the overhead of updating indexes as the `INSERT` statements are replayed. With storage engines such as InnoDB this is not recommended.

2.2.4.1 Creating a Data Snapshot Using `mysqldump`

To create a snapshot of the data in an existing master database, use the `mysqldump` tool. Once the data dump has been completed, import this data into the slave before starting the replication process.

The following example dumps all databases to a file named `dbdump.db`, and includes the `--master-data` option which automatically appends the `CHANGE MASTER TO` statement required on the slave to start the replication process:

```shell
shell> mysqldump --all-databases --master-data > dbdump.db
```

Note

If you do not use `--master-data`, then it is necessary to lock all tables in a separate session manually. See Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”.

It is possible to exclude certain databases from the dump using the `mysqldump` tool. If you want to choose which databases to include in the dump, do not use `--all-databases`. Choose one of these options:

- Exclude all the tables in the database using `--ignore-table` option.
- Name only those databases which you want dumped using the `--databases` option.

For more information, see `mysqldump — A Database Backup Program`.

To import the data, either copy the dump file to the slave, or access the file from the master when connecting remotely to the slave.

2.2.4.2 Creating a Data Snapshot Using Raw Data Files

This section describes how to create a data snapshot using the raw files which make up the database. Employing this method with a table using a storage engine that has complex caching or logging algorithms requires extra steps to produce a perfect “point in time” snapshot: the initial copy command could leave out cache information and logging updates, even if you have acquired a global read lock. How the storage engine responds to this depends on its crash recovery abilities.

If you use InnoDB tables, you can use the `mysqlbackup` command from the MySQL Enterprise Backup component to produce a consistent snapshot. This command records the log name and offset corresponding to the snapshot to be used on the slave. MySQL Enterprise Backup is a commercial product that is included as part of a MySQL Enterprise subscription. See MySQL Enterprise Backup Overview for detailed information.

This method also does not work reliably if the master and slave have different values for `ft_stopword_file`, `ft_min_word_len`, or `ft_max_word_len` and you are copying tables having full-text indexes.

Assuming the above exceptions do not apply to your database, use the cold backup technique to obtain a reliable binary snapshot of InnoDB tables: do a slow shutdown of the MySQL Server, then copy the data files manually.

To create a raw data snapshot of MyISAM tables when your MySQL data files exist on a single file system, you can use standard file copy tools such as `cp` or `copy`, a remote copy tool such as `scp` or `rsync`, an archiving tool such as `zip` or `tar`, or a file system snapshot tool such as `dump`. If you are replicating only certain databases, copy only those files that relate to those tables. For InnoDB, all tables in all databases are stored in the `system tablespace` files, unless you have the `innodb_file_per_table` option enabled.
Setting Up Replication Slaves

The following files are not required for replication:

- Files relating to the mysql database.
- The master info repository file, if used (see Section 5.4, “Replication Relay and Status Logs”).
- The master’s binary log files, with the exception of the binary log index file if you are going to use this to locate the master binary log coordinates for the slave.
- Any relay log files.

Depending on whether you are using InnoDB tables or not, choose one of the following:

If you are using InnoDB tables, and also to get the most consistent results with a raw data snapshot, shut down the master server during the process, as follows:

1. Acquire a read lock and get the master’s status. See Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”.

2. In a separate session, shut down the master server:

```
shell> mysqladmin shutdown
```

3. Make a copy of the MySQL data files. The following examples show common ways to do this. You need to choose only one of them:

```
shell> tar cf /tmp/db.tar ./data
shell> zip -r /tmp/db.zip ./data
shell> rsync --recursive ./data /tmp/dbdata
```

4. Restart the master server.

If you are not using InnoDB tables, you can get a snapshot of the system from a master without shutting down the server as described in the following steps:

1. Acquire a read lock and get the master’s status. See Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”.

2. Make a copy of the MySQL data files. The following examples show common ways to do this. You need to choose only one of them:

```
shell> tar cf /tmp/db.tar ./data
shell> zip -r /tmp/db.zip ./data
shell> rsync --recursive ./data /tmp/dbdata
```

3. In the client where you acquired the read lock, release the lock:

```
mysql> UNLOCK TABLES;
```

Once you have created the archive or copy of the database, copy the files to each slave before starting the slave replication process.

### 2.2.5 Setting Up Replication Slaves

The following sections describe how to set up slaves. Before you proceed, ensure that you have:

- Configured the MySQL master with the necessary configuration properties. See Section 2.2.1, “Setting the Replication Master Configuration”.
- Obtained the master status information, or a copy of the master’s binary log index file made during a shutdown for the data snapshot. See Section 2.2.3, “Obtaining the Replication Master Binary Log Coordinates”.
- On the master, released the read lock:

```
mysql> UNLOCK TABLES;
```
### 2.2.5.1 Setting the Replication Slave Configuration

Each replication slave must have a unique server ID. If this has not already been done, this part of slave setup requires a server restart.

If the slave server ID is not already set, or the current value conflicts with the value that you have chosen for the master server, shut down the slave server and edit the `mysqld` section of the configuration file to specify a unique server ID. For example:

```
[mysqld]
server-id=2
```

After making the changes, restart the server.

If you are setting up multiple slaves, each one must have a unique `server_id` value that differs from that of the master and from any of the other slaves.

**Note**

If you do not set `server_id` (or set it explicitly to its default value of 0), the slave refuses to connect to a master.

You do not have to enable binary logging on the slave for replication to be set up. However, if you enable binary logging on the slave, you can use the slave’s binary log for data backups and crash recovery, and also use the slave as part of a more complex replication topology. For example, where this slave then acts as a master to other slaves.

### 2.2.5.2 Setting the Master Configuration on the Slave

To set up the slave to communicate with the master for replication, configure the slave with the necessary connection information. To do this, execute the following statement on the slave, replacing the option values with the actual values relevant to your system:

```
mysql> CHANGE MASTER TO
    -> MASTER_HOST='master_host_name',
    -> MASTER_USER='replication_user_name',
    -> MASTER_PASSWORD='replication_password',
    -> MASTER_LOG_FILE='recorded_log_file_name',
    -> MASTER_LOG_POS=recorded_log_position;
```

**Note**

Replication cannot use Unix socket files. You must be able to connect to the master MySQL server using TCP/IP.

The `CHANGE MASTER TO` statement has other options as well. For example, it is possible to set up secure replication using SSL. For a full list of options, and information about the maximum permissible length for the string-valued options, see `CHANGE MASTER TO Statement`.

The next steps depend on whether you have existing data to import to the slave or not. See Section 2.2.4, “Choosing a Method for Data Snapshots” for more information. Choose one of the following:

- If you do not have a snapshot of a database to import, see Section 2.2.5.3, “Setting Up Replication between a New Master and Slaves”.
- If you have a snapshot of a database to import, see Section 2.2.5.4, “Setting Up Replication with Existing Data”.

### 2.2.5.3 Setting Up Replication between a New Master and Slaves

When there is no snapshot of a previous database to import, configure the slave to start the replication from the new master.

To set up replication between a master and a new slave:
1. Start up the MySQL slave and connect to it.

2. Execute a `CHANGE MASTER TO` statement to set the master replication server configuration. See Section 2.2.5.2, “Setting the Master Configuration on the Slave”.

Perform these slave setup steps on each slave.

This method can also be used if you are setting up new servers but have an existing dump of the databases from a different server that you want to load into your replication configuration. By loading the data into a new master, the data is automatically replicated to the slaves.

If you are setting up a new replication environment using the data from a different existing database server to create a new master, run the dump file generated from that server on the new master. The database updates are automatically propagated to the slaves:

```
shell> mysql -h master < fulldb.dump
```

### 2.2.5.4 Setting Up Replication with Existing Data

When setting up replication with existing data, transfer the snapshot from the master to the slave before starting replication. The process for importing data to the slave depends on how you created the snapshot of data on the master.

Choose one of the following:

If you used `mysqldump`:

1. Start the slave, using the `--skip-slave-start` option so that replication does not start.

2. Import the dump file:

```
shell> mysql < fulldb.dump
```

If you created a snapshot using the raw data files:

1. Extract the data files into your slave data directory. For example:

```
shell> tar xvf dbdump.tar
```

You may need to set permissions and ownership on the files so that the slave server can access and modify them.

2. Start the slave, using the `--skip-slave-start` option so that replication does not start.

3. Configure the slave with the replication coordinates from the master. This tells the slave the binary log file and position within the file where replication needs to start. Also, configure the slave with the login credentials and host name of the master. For more information on the `CHANGE MASTER TO` statement required, see Section 2.2.5.2, “Setting the Master Configuration on the Slave”.

4. Start the slave threads:

```
mysql> START SLAVE;
```

After you have performed this procedure, the slave connects to the master and replicates any updates that have occurred on the master since the snapshot was taken.

If the `server_id` system variable for the master is not correctly set, slaves cannot connect to it. Similarly, if you have not set `server_id` correctly for the slave, you get the following error in the slave's error log:

```
Warning: You should set server-id to a non-0 value if master_host is set; we will force server id to 2, but this MySQL server will not act as a slave.
```

You also find error messages in the slave's error log if it is not able to replicate for any other reason.
The slave stores information about the master you have configured in its master info repository. The master info repository can be in the form of files or a table, as determined by the value set for the `master_info_repository` system variable. When a slave runs with `master_info_repository=FILE`, two files are stored in the data directory, named `master.info` and `relay-log.info`. If `master_info_repository=TABLE` instead, this information is saved in the `master_slave_info` table in the `mysql` database. In either case, do not remove or edit the files or table. Always use the `CHANGE MASTER TO` statement to change replication parameters. The slave can use the values specified in the statement to update the status files automatically. See Section 5.4, “Replication Relay and Status Logs”, for more information.

**Note**

The contents of the master info repository override some of the server options specified on the command line or in `my.cnf`. See Section 2.6, “Replication and Binary Logging Options and Variables”, for more details.

A single snapshot of the master suffices for multiple slaves. To set up additional slaves, use the same master snapshot and follow the slave portion of the procedure just described.

## 2.2.6 Adding Slaves to a Replication Environment

You can add another slave to an existing replication configuration without stopping the master. To do this, you can set up the new slave by copying the data directory of an existing slave, and giving the new slave a different server ID (which is user-specified) and server UUID (which is generated at startup).

To duplicate an existing slave:

1. Stop the existing slave and record the slave status information, particularly the master binary log file and relay log file positions. You can view the slave status either in the Performance Schema replication tables (see Performance Schema Replication Tables), or by issuing `SHOW SLAVE STATUS` as follows:

   ```
   mysql> STOP SLAVE;
   mysql> SHOW SLAVE STATUS\G
   ```

2. Shut down the existing slave:

   ```
   shell> mysqladmin shutdown
   ```

3. Copy the data directory from the existing slave to the new slave, including the log files and relay log files. You can do this by creating an archive using `tar` or `WinZip`, or by performing a direct copy using a tool such as `cp` or `rsync`.

**Important**

- Before copying, verify that all the files relating to the existing slave actually are stored in the data directory. For example, the `InnoDB` system tablespace, undo tablespace, and redo log might be stored in an alternative location. `InnoDB` tablespace files and file-per-table tablespaces might have been created in other directories. The binary logs and relay logs for the slave might be in their own directories outside the data directory. Check through the system variables that are set for the existing slave and look for any alternative paths that have been specified. If you find any, copy these directories over as well.

- During copying, if files have been used for the master info and relay log info repositories (see Section 5.4, “Replication Relay and Status Logs”), which is the default in MySQL 5.7, ensure that you also copy these files from the existing slave to the new slave. If tables have been used for the repositories, the tables are in the data directory.
Replication with Global Transaction Identifiers

- After copying, delete the auto.cnf file from the copy of the data directory on the new slave, so that the new slave is started with a different generated server UUID. The server UUID must be unique.

A common problem that is encountered when adding new replication slaves is that the new slave fails with a series of warning and error messages like these:

```
071118 16:44:10 [Warning] Neither --relay-log nor --relay-log-index were used; so replication may break when this MySQL server acts as a slave and has his hostname changed!! Please use '--relay-log=new_slave_hostname-relay-bin' to avoid this problem.
071118 16:44:10 [ERROR] Failed to open the relay log './old_slave_hostname-relay-bin.003525'(relay_log_pos 22940879)
071118 16:44:10 [ERROR] Could not find target log during relay log initialization
071118 16:44:10 [ERROR] Failed to initialize the master info structure
```

This situation can occur if the `relay_log` system variable is not specified, as the relay log files contain the host name as part of their file names. This is also true of the relay log index file if the `relay_log_index` system variable is not used. For more information about these variables, see Section 2.6, "Replication and Binary Logging Options and Variables".

To avoid this problem, use the same value for `relay_log` on the new slave that was used on the existing slave. If this option was not set explicitly on the existing slave, use `existing_slave_hostname-relay-bin`. If this is not possible, copy the existing slave's relay log index file to the new slave and set the `relay_log_index` system variable on the new slave to match what was used on the existing slave. If this option was not set explicitly on the existing slave, use `existing_slave_hostname-relay-bin.index`. Alternatively, if you have already tried to start the new slave after following the remaining steps in this section and have encountered errors like those described previously, then perform the following steps:

a. If you have not already done so, issue `STOP SLAVE` on the new slave.
   
   If you have already started the existing slave again, issue `STOP SLAVE` on the existing slave as well.

b. Copy the contents of the existing slave’s relay log index file into the new slave’s relay log index file, making sure to overwrite any content already in the file.

c. Proceed with the remaining steps in this section.

4. When copying is complete, restart the existing slave.

5. On the new slave, edit the configuration and give the new slave a unique server ID (using the `server_id` system variable) that is not used by the master or any of the existing slaves.

6. Start the new slave server, specifying the `--skip-slave-start` option so that replication does not start yet. Use the Performance Schema replication tables or issue `SHOW SLAVE STATUS` to confirm that the new slave has the correct settings when compared with the existing slave. Also display the server ID and server UUID and verify that these are correct and unique for the new slave.

7. Start the slave threads by issuing a `START SLAVE` statement:

```
mysql> START SLAVE;
```

The new slave now uses the information in its master info repository to start the replication process.

**2.3 Replication with Global Transaction Identifiers**

This section explains transaction-based replication using *global transaction identifiers* (GTIDs). When using GTIDs, each transaction can be identified and tracked as it is committed on the originating server and applied by any slaves; this means that it is not necessary when using GTIDs to refer to log files.
or positions within those files when starting a new slave or failing over to a new master, which greatly simplifies these tasks. Because GTID-based replication is completely transaction-based, it is simple to determine whether masters and slaves are consistent; as long as all transactions committed on a master are also committed on a slave, consistency between the two is guaranteed. You can use either statement-based or row-based replication with GTIDs (see Section 5.1, “Replication Formats”); however, for best results, we recommend that you use the row-based format.

GTIDs are always preserved between master and slave. This means that you can always determine the source for any transaction applied on any slave by examining its binary log. In addition, once a transaction with a given GTID is committed on a given server, any subsequent transaction having the same GTID is ignored by that server. Thus, a transaction committed on the master can be applied no more than once on the slave, which helps to guarantee consistency.

This section discusses the following topics:

- How GTIDs are defined and created, and how they are represented in a MySQL server (see Section 2.3.1, “GTID Format and Storage”).
- The life cycle of a GTID (see Section 2.3.2, “GTID Life Cycle”).
- The auto-positioning function for synchronizing a slave and master that use GTIDs (see Section 2.3.3, “GTID Auto-Positioning”).
- A general procedure for setting up and starting GTID-based replication (see Section 2.3.4, “Setting Up Replication Using GTIDs”).
- Suggested methods for provisioning new replication servers when using GTIDs (see Section 2.3.5, “Using GTIDs for Failover and Scaleout”).
- Restrictions and limitations that you should be aware of when using GTID-based replication (see Section 2.3.6, “Restrictions on Replication with GTIDs”).
- Stored functions that you can use to work with GTIDs (see Section 2.3.7, “Stored Function Examples to Manipulate GTIDs”).

For information about MySQL Server options and variables relating to GTID-based replication, see Section 2.6.5, “Global Transaction ID System Variables”. See also Functions Used with Global Transaction Identifiers (GTIDs), which describes SQL functions supported by MySQL 5.7 for use with GTIDs.

### 2.3.1 GTID Format and Storage

A global transaction identifier (GTID) is a unique identifier created and associated with each transaction committed on the server of origin (the master). This identifier is unique not only to the server on which it originated, but is unique across all servers in a given replication topology.

GTID assignment distinguishes between client transactions, which are committed on the master, and replicated transactions, which are reproduced on a slave. When a client transaction is committed on the master, it is assigned a new GTID, provided that the transaction was written to the binary log. Client transactions are guaranteed to have monotonically increasing GTIDs without gaps between the generated numbers. If a client transaction is not written to the binary log (for example, because the transaction was filtered out, or the transaction was read-only), it is not assigned a GTID on the server of origin.

Replicated transactions retain the same GTID that was assigned to the transaction on the server of origin. The GTID is present before the replicated transaction begins to execute, and is persisted even if the replicated transaction is not written to the binary log on the slave, or is filtered out on the slave. The MySQL system table `mysql.gtid_executed` is used to preserve the assigned GTIDs of all the transactions applied on a MySQL server, except those that are stored in a currently active binary log file.
The auto-skip function for GTIDs means that a transaction committed on the master can be applied no more than once on the slave, which helps to guarantee consistency. Once a transaction with a given GTID has been committed on a given server, any attempt to execute a subsequent transaction with the same GTID is ignored by that server. No error is raised, and no statement in the transaction is executed.

If a transaction with a given GTID has started to execute on a server, but has not yet committed or rolled back, any attempt to start a concurrent transaction on the server with the same GTID will block. The server neither begins to execute the concurrent transaction nor returns control to the client. Once the first attempt at the transaction commits or rolls back, concurrent sessions that were blocking on the same GTID may proceed. If the first attempt rolled back, one concurrent session proceeds to attempt the transaction, and any other concurrent sessions that were blocking on the same GTID remain blocked. If the first attempt committed, all the concurrent sessions stop being blocked, and auto-skip all the statements of the transaction.

A GTID is represented as a pair of coordinates, separated by a colon character (:), as shown here:

\[ \text{GTID} = \text{source_id}:\text{transaction_id} \]

The source_id identifies the originating server. Normally, the master's server_uuid is used for this purpose. The transaction_id is a sequence number determined by the order in which the transaction was committed on the master. For example, the first transaction to be committed has 1 as its transaction_id, and the tenth transaction to be committed on the same originating server is assigned a transaction_id of 10. It is not possible for a transaction to have 0 as a sequence number in a GTID. For example, the twenty-third transaction to be committed originally on the server with the UUID 3E11FA47-71CA-11E1-9E33-C80AA9429562 has this GTID:

3E11FA47-71CA-11E1-9E33-C80AA9429562:23

The GTID for a transaction is shown in the output from mysqlbinlog, and it is used to identify an individual transaction in the Performance Schema replication status tables, for example, replication_applier_status_by_worker. The value stored by the @GLOBAL.gtid_next system variable is a single GTID.

**GTID Sets**

A GTID set is a set comprising one or more single GTIDs or ranges of GTIDs. GTID sets are used in a MySQL server in several ways. For example, the values stored by the gtid_executed and gtid_purged system variables are GTID sets. The START SLAVE clauses UNTIL SQL_BEFORE_GTIDS and UNTIL SQL_AFTER_GTIDS can be used to make a slave process transactions only up to the first GTID in a GTID set, or stop after the last GTID in a GTID set. The built-in functions GTID_SUBSET() and GTID_SUBTRACT() require GTID sets as input.

A range of GTIDs originating from the same server can be collapsed into a single expression, as shown here:

3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5

The above example represents the first through fifth transactions originating on the MySQL server whose server_uuid is 3E11FA47-71CA-11E1-9E33-C80AA9429562. Multiple single GTIDs or ranges of GTIDs originating from the same server can also be included in a single expression, with the GTIDs or ranges separated by colons, as in the following example:


A GTID set can include any combination of single GTIDs and ranges of GTIDs, and it can include GTIDs originating from different servers. This example shows the GTID set stored in the gtid_executed system variable (@GLOBAL.gtid_executed) of a slave that has applied transactions from more than one master:

When GTID sets are returned from server variables, UUIDs are in alphabetical order, and numeric intervals are merged and in ascending order.

The syntax for a GTID set is as follows:

```
gtid_set:     
  uuid_set [, uuid_set] ...  
  | ''

uuid_set:     
  uuid: interval[:interval]...

uuid:   
hhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhh

h:   
[0-9][A-F]

interval: 
  n[-n]
(n >= 1)
```

**mysql.gtid_executed Table**

GTIDs are stored in a table named `gtid_executed`, in the `mysql` database. A row in this table contains, for each GTID or set of GTIDs that it represents, the UUID of the originating server, and the starting and ending transaction IDs of the set; for a row referencing only a single GTID, these last two values are the same.

The `mysql.gtid_executed` table is created (if it does not already exist) when MySQL Server is installed or upgraded, using a `CREATE TABLE` statement similar to that shown here:

```
CREATE TABLE gtid_executed (  
  source_uuid CHAR(36) NOT NULL,  
  interval_start BIGINT(20) NOT NULL,  
  interval_end BIGINT(20) NOT NULL,  
  PRIMARY KEY (source_uuid, interval_start)
)
```

**Warning**

As with other MySQL system tables, do not attempt to create or modify this table yourself.

The `mysql.gtid_executed` table is provided for internal use by the MySQL server. It enables a slave to use GTIDs when binary logging is disabled on the slave, and it enables retention of the GTID state when the binary logs have been lost. Note that the `mysql.gtid_executed` table is cleared if you issue `RESET MASTER`.

GTIDs are stored in the `mysql.gtid_executed` table only when `gtid_mode` is `ON` or `ON_PERMISSIVE`. The point at which GTIDs are stored depends on whether binary logging is enabled or disabled:

- If binary logging is disabled (`log_bin` is `OFF`), or if `log_slave_updates` is disabled, the server stores the GTID belonging to each transaction together with the transaction in the table. In addition, the table is compressed periodically at a user-configurable rate; see `mysql.gtid_executed Table Compression`, for more information. This situation can only apply on a replication slave where binary logging or slave update logging is disabled. It does not apply on a replication master, because on a master, binary logging must be enabled for replication to take place.

- If binary logging is enabled (`log_bin` is `ON`), whenever the binary log is rotated or the server is shut down, the server writes GTIDs for all transactions that were written into the previous binary log into
the `mysql.gtid_executed` table. This situation applies on a replication master, or a replication slave where binary logging is enabled.

In the event of the server stopping unexpectedly, the set of GTIDs from the current binary log file is not saved in the `mysql.gtid_executed` table. These GTIDs are added to the table from the binary log file during recovery. The exception to this is if binary logging is not enabled when the server is restarted. In this situation, the server cannot access the binary log file to recover the GTIDs, so replication cannot be started.

When binary logging is enabled, the `mysql.gtid_executed` table does not hold a complete record of the GTIDs for all executed transactions. That information is provided by the global value of the `gtid_executed` system variable. Always use `@@GLOBAL.gtid_executed`, which is updated after every commit, to represent the GTID state for the MySQL server, and do not query the `mysql.gtid_executed` table.

**mysql.gtid_executed Table Compression**

Over the course of time, the `mysql.gtid_executed` table can become filled with many rows referring to individual GTIDs that originate on the same server, and whose transaction IDs make up a range, similar to what is shown here:

<table>
<thead>
<tr>
<th>source_uuid</th>
<th>interval_start</th>
<th>interval_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>37</td>
<td>37</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>38</td>
<td>38</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>39</td>
<td>39</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>40</td>
<td>40</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>41</td>
<td>41</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>42</td>
<td>42</td>
</tr>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>43</td>
<td>43</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

To save space, the MySQL server compresses the `mysql.gtid_executed` table periodically by replacing each such set of rows with a single row that spans the entire interval of transaction identifiers, like this:

<table>
<thead>
<tr>
<th>source_uuid</th>
<th>interval_start</th>
<th>interval_end</th>
</tr>
</thead>
<tbody>
<tr>
<td>3E11FA47-71CA-11E1-9E33-C80AA9429562</td>
<td>37</td>
<td>43</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can control the number of transactions that are allowed to elapse before the table is compressed, and thus the compression rate, by setting the `gtid_executed_compression_period` system variable. This variable's default value is 1000, meaning that by default, compression of the table is performed after each 1000 transactions. Setting `gtid_executed_compression_period` to 0 prevents the compression from being performed at all, and you should be prepared for a potentially large increase in the amount of disk space that may be required by the `gtid_executed` table if you do this.

**Note**

When binary logging is enabled, the value of `gtid_executed_compression_period` is *not* used and the `mysql.gtid_executed` table is compressed on each binary log rotation.

Compression of the `mysql.gtid_executed` table is performed by a dedicated foreground thread named `thread/sql/compress_gtid_table`. This thread is not listed in the output of `SHOW PROCESSLIST`, but it can be viewed as a row in the `threads` table, as shown here:

```
mysql> SELECT * FROM performance_schema.threads WHERE NAME LIKE 'thread/sql/compress_gtid_table'\G
*************************** 1. row ***************************
17
```
The thread/sql/compress_gtid_table thread normally sleeps until gtid_executed_compression_period transactions have been executed, then wakes up to perform compression of the mysql.gtid_executed table as described previously. It then sleeps until another gtid_executed_compression_period transactions have taken place, then wakes up to perform the compression again, repeating this loop indefinitely. Setting this value to 0 when binary logging is disabled means that the thread always sleeps and never wakes up.

### 2.3.2 GTID Life Cycle

The life cycle of a GTID consists of the following steps:

1. A transaction is executed and committed on the master. This client transaction is assigned a GTID composed of the master's UUID and the smallest nonzero transaction sequence number not yet used on this server. The GTID is written to the master's binary log (immediately preceding the transaction itself in the log). If a client transaction is not written to the binary log (for example, because the transaction was filtered out, or the transaction was read-only), it is not assigned a GTID.

2. If a GTID was assigned for the transaction, the GTID is persisted atomically at commit time by writing it to the binary log at the beginning of the transaction (as a `Gtid_log_event`). Whenever the binary log is rotated or the server is shut down, the server writes GTIDs for all transactions that were written into the previous binary log file into the `mysql.gtid_executed` table.

3. If a GTID was assigned for the transaction, the GTID is externalized non-atomically (very shortly after the transaction is committed) by adding it to the set of GTIDs in the `gtid_executed` system variable (`@@GLOBAL.gtid_executed`). This GTID set contains a representation of the set of all committed GTID transactions, and it is used in replication as a token that represents the server state. With binary logging enabled (as required for the master), the set of GTIDs in the `gtid_executed` system variable is a complete record of the transactions applied, but the `mysql.gtid_executed` table is not, because the most recent history is still in the current binary log file.

4. After the binary log data is transmitted to the slave and stored in the slave’s relay log (using established mechanisms for this process, see Chapter 5, Replication Implementation, for details), the slave reads the GTID and sets the value of its `gtid_next` system variable as this GTID. This tells the slave that the next transaction must be logged using this GTID. It is important to note that the slave sets `gtid_next` in a session context.

5. The slave verifies that no thread has yet taken ownership of the GTID in `gtid_next` in order to process the transaction. By reading and checking the replicated transaction’s GTID first, before processing the transaction itself, the slave guarantees not only that no previous transaction having this GTID has been applied on the slave, but also that no other session has already read this GTID but has not yet committed the associated transaction. So if multiple clients attempt to apply the same transaction concurrently, the server resolves this by letting only one of them execute. The `gtid_owned` system variable (`@@GLOBAL.gtid_owned`) for the slave shows each GTID that is
currently in use and the ID of the thread that owns it. If the GTID has already been used, no error is
raised, and the auto-skip function is used to ignore the transaction.

6. If the GTID has not been used, the slave applies the replicated transaction. Because `gtid_next`
is set to the GTID already assigned by the master, the slave does not attempt to generate a new
GTID for this transaction, but instead uses the GTID stored in `gtid_next`.

7. If binary logging is enabled on the slave, the GTID is persisted atomically at commit time by writing
it to the binary log at the beginning of the transaction (as a `Gtid_log_event`). Whenever the
binary log is rotated or the server is shut down, the server writes GTIDs for all transactions that
were written into the previous binary log file into the `mysql.gtid_executed` table.

8. If binary logging is disabled on the slave, the GTID is persisted atomically by writing it directly into
the `mysql.gtid_executed` table. MySQL appends a statement to the transaction to insert the
GTID into the table. In this situation, the `mysql.gtid_executed` table is a complete record of the
transactions applied on the slave. Note that in MySQL 5.7, the operation to insert the GTID into the
table is atomic for DML statements, but not for DDL statements, so if the server exits unexpectedly
after a transaction involving DDL statements, the GTID state might become inconsistent. From
MySQL 8.0, the operation is atomic for DDL statements as well as for DML statements.

9. Very shortly after the replicated transaction is committed on the slave, the GTID is externalized
non-atomically by adding it to the set of GTIDs in the `gtid_executed` system variable
(`@@GLOBAL.gtid_executed`) for the slave. As for the master, this GTID set contains a
representation of the set of all committed GTID transactions. If binary logging is disabled on the
slave, the `mysql.gtid_executed` table is also a complete record of the transactions applied on
the slave. If binary logging is enabled on the slave, meaning that some GTIDs are only recorded
in the binary log, the set of GTIDs in the `gtid_executed` system variable is the only complete
record.

Client transactions that are completely filtered out on the master are not assigned a GTID, therefore
they are not added to the set of transactions in the `gtid_executed` system variable, or added
to the `mysql.gtid_executed` table. However, the GTIDs of replicated transactions that are
completely filtered out on the slave are persisted. If binary logging is enabled on the slave, the
filtered-out transaction is written to the binary log as a `Gtid_log_event` followed by an empty
transaction containing only `BEGIN` and `COMMIT` statements. If binary logging is disabled, the GTID of
the filtered-out transaction is written to the `mysql.gtid_executed` table. Preserving the GTIDs for
filtered-out transactions ensures that the `mysql.gtid_executed` table and the set of GTIDs in the
`gtid_executed` system variable can be compressed. It also ensures that the filtered-out transactions
are not retrieved again if the slave reconnects to the master, as explained in Section 2.3.3, “GTID Auto-
Positioning”.

On a multithreaded replication slave (with `slave_parallel_workers > 0`), transactions
can be applied in parallel, so replicated transactions can commit out of order (unless
`slave_preserve_commit_order=1` is set). When that happens, the set of GTIDs in the
`gtid_executed` system variable will contain multiple GTID ranges with gaps between them. (On a
master or a single-threaded replication slave, there will be monotonically increasing GTIDs without
gaps between the numbers.) Gaps on multithreaded replication slaves only occur among the most
recently applied transactions, and are filled in as replication progresses. When replication threads are
stopped cleanly using the `STOP SLAVE` statement, ongoing transactions are applied so that the gaps
are filled in. In the event of a shutdown such as a server failure or the use of the `KILL` statement to
stop replication threads, the gaps might remain.

What changes are assigned a GTID?

The typical scenario is that the server generates a new GTID for a committed transaction. However,
GTIDs can also be assigned to other changes besides transactions, and in some cases a single
transaction can be assigned multiple GTIDs.

Every database change (DDL or DML) that is written to the binary log is assigned a GTID. This
includes changes that are autocommitted, and changes that are committed using `BEGIN` and `COMMIT`
or `START TRANSACTION` statements. A GTID is also assigned to the creation, alteration, or deletion of a database, and of a non-table database object such as a procedure, function, trigger, event, view, user, role, or grant.

Non-transactional updates as well as transactional updates are assigned GTIDs. In addition, for a non-transactional update, if a disk write failure occurs while attempting to write to the binary log cache and a gap is therefore created in the binary log, the resulting incident log event is assigned a GTID.

When a table is automatically dropped by a generated statement in the binary log, a GTID is assigned to the statement. Temporary tables are dropped automatically when a replication slave begins to apply events from a master that has just been started, and when statement-based replication is in use (`binlog_format=STATEMENT`) and a user session that has open temporary tables disconnects. Tables that use the `MEMORY` storage engine are deleted automatically the first time they are accessed after the server is started, because rows might have been lost during the shutdown.

When a transaction is not written to the binary log on the server of origin, the server does not assign a GTID to it. This includes transactions that are rolled back and transactions that are executed while binary logging is disabled on the server of origin, either globally (with `--skip-log-bin` specified in the server's configuration) or for the session (`SET @SESSION.sql_log_bin = 0`). This also includes no-op transactions when row-based replication is in use (`binlog_format=ROW`).

XA transactions are assigned separate GTIDs for the `XA PREPARE` phase of the transaction and the `XA COMMIT` or `XA ROLLBACK` phase of the transaction. XA transactions are persistently prepared so that users can commit them or roll them back in the case of a failure (which in a replication topology might include a failover to another server). The two parts of the transaction are therefore replicated separately, so they must have their own GTIDs, even though a non-XA transaction that is rolled back would not have a GTID.

In the following special cases, a single statement can generate multiple transactions, and therefore be assigned multiple GTIDs:

- A stored procedure is invoked that commits multiple transactions. One GTID is generated for each transaction that the procedure commits.
- A multi-table `DROP TABLE` statement drops tables of different types.
- A `CREATE TABLE ... SELECT` statement is issued when row-based replication is in use (`binlog_format=ROW`). One GTID is generated for the `CREATE TABLE` action and one GTID is generated for the row-insert actions.

The `gtid_next` System Variable

By default, for new transactions committed in user sessions, the server automatically generates and assigns a new GTID. When the transaction is applied on a replication slave, the GTID from the server of origin is preserved. You can change this behavior by setting the session value of the `gtid_next` system variable:

- When `gtid_next` is set to `AUTOMATIC`, which is the default, and a transaction is committed and written to the binary log, the server automatically generates and assigns a new GTID. If a transaction is rolled back or not written to the binary log for another reason, the server does not generate and assign a GTID.
- If you set `gtid_next` to a valid GTID (consisting of a UUID and a transaction sequence number, separated by a colon), the server assigns that GTID to your transaction. This GTID is assigned and added to `gtid_executed` even when the transaction is not written to the binary log, or when the transaction is empty.

Note that after you set `gtid_next` to a specific GTID, and the transaction has been committed or rolled back, an explicit `SET @@SESSION.gtid_next` statement must be issued before any other statement. You can use this to set the GTID value back to `AUTOMATIC` if you do not want to assign any more GTIDs explicitly.
When replication applier threads apply replicated transactions, they use this technique, setting @@SESSION.gtid_next explicitly to the GTID of the replicated transaction as assigned on the server of origin. This means the GTID from the server of origin is retained, rather than a new GTID being generated and assigned by the replication slave. It also means the GTID is added to gtid_executed on the replication slave even when binary logging or slave update logging is disabled on the slave, or when the transaction is a no-op or is filtered out on the slave.

It is possible for a client to simulate a replicated transaction by setting @@SESSION.gtid_next to a specific GTID before executing the transaction. This technique is used by mysqlbinlog to generate a dump of the binary log that the client can replay to preserve GTIDs. A simulated replicated transaction committed through a client is completely equivalent to a replicated transaction committed through a replication applier thread, and they cannot be distinguished after the fact.

The gtid_purged System Variable

The set of GTIDs in the gtid_purged system variable (@GLOBAL.gtid_purged) contains 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 slave.
• 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.

You can change 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. When you add GTIDs to gtid_purged, they are also added to gtid_executed. 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. In MySQL 5.7, you can only change the value of gtid_purged when gtid_executed (and therefore gtid_purged) is empty. For details of how to do this, see the description for gtid_purged.

The sets of GTIDs in the gtid_executed and gtid_purged system variables are initialized when the server starts. Every binary log file begins with the event Previous_gtids_log_event, which contains the set of GTIDs in all previous binary log files (composed from the GTIDs in the preceding file's Previous_gtids_log_event, and the GTIDs of every Gtid_log_event in the preceding file itself). The contents of Previous_gtids_log_event in the oldest and most recent binary log files are used to compute the gtid_executed and gtid_purged sets at server startup:

• gtid_executed is computed as the union of the GTIDs in Previous_gtids_log_event in the most recent binary log file, the GTIDs of transactions in that binary log file, and the GTIDs stored in the mysql.gtid_executed table. This GTID set contains all the GTIDs that have been used (or added explicitly to gtid_purged) on the server, whether or not they are currently in a binary log file on the server. It does not include the GTIDs for transactions that are currently being processed on the server (@GLOBAL.gtid_owned).

• gtid_purged is computed by first adding the GTIDs in Previous_gtids_log_event in the most recent binary log file and the GTIDs of transactions in that binary log file, and the GTIDs stored in the mysql.gtid_executed table. This GTID set contains all the GTIDs that have been used (or added explicitly to gtid_purged) on the server, whether or not they are currently in a binary log file on the server. It does not include the GTIDs for transactions that are currently being processed on the server (@GLOBAL.gtid_owned). Next, the GTIDs in Previous_gtids_log_event in the oldest binary log file are subtracted from gtid_in_binlog. This step gives the set of GTIDs that are currently recorded in a binary log on the server (gtids_in_binlog_not_purged). Finally, gtid_in_binlog_not_purged is subtracted from gtid_executed. The result is the set of GTIDs that have been used on the server, but are not currently recorded in a binary log file on the server, and this result is used to initialize gtid_purged.

If binary logs from MySQL 5.7.7 or older are involved in these computations, it is possible for incorrect GTID sets to be computed for gtid_executed and gtid_purged, and they remain incorrect even if
the server is later restarted. For details, see the description for the `binlog_gtid_simple_recovery` system variable, which controls how the binary logs are iterated to compute the GTID sets. If one of the situations described there applies on a server, set `binlog_gtid_simple_recovery=FALSE` in the server's configuration file before starting it. That setting makes the server iterate all the binary log files (not just the newest and oldest) to find where GTID events start to appear. This process could take a long time if the server has a large number of binary log files without GTID events.

### Resetting the GTID Execution History

If you need to reset the GTID execution history on a server, use the `RESET MASTER` statement. For example, you might need to do this after carrying out test queries to verify a replication setup on new GTID-enabled servers, or when you want to join a new server to a replication group but it contains some unwanted local transactions that are not accepted by Group Replication.

#### Warning

Use `RESET MASTER` with caution to avoid losing any wanted GTID execution history and binary log files.

Before issuing `RESET MASTER`, ensure that you have backups of the server's binary log files and binary log index file, if any, and obtain and save the GTID set held in the global value of the `gtid_executed` system variable (for example, by issuing a `SELECT @@GLOBAL.gtid_executed` statement and saving the results). If you are removing unwanted transactions from that GTID set, use `mysqlbinlog` to examine the contents of the transactions to ensure that they have no value, contain no data that must be saved or replicated, and did not result in data changes on the server.

When you issue `RESET MASTER`, the following reset operations are carried out:

- The value of the `gtid_purged` system variable is set to an empty string (`''`).
- The global value (but not the session value) of the `gtid_executed` system variable is set to an empty string.
- The `mysql.gtid_executed` table is cleared (see `mysql.gtid_executed Table`).
- If the server has binary logging enabled, the existing binary log files are deleted and the binary log index file is cleared.

Note that `RESET MASTER` is the method to reset the GTID execution history even if the server is a replication slave where binary logging is disabled. `RESET SLAVE` has no effect on the GTID execution history.

### 2.3.3 GTID Auto-Positioning

GTIDs replace the file-offset pairs previously required to determine points for starting, stopping, or resuming the flow of data between master and slave. When GTIDs are in use, all the information that the slave needs for synchronizing with the master is obtained directly from the replication data stream.

To start a slave using GTID-based replication, you do not include `MASTER_LOG_FILE` or `MASTER_LOG_POS` options in the `CHANGE MASTER TO` statement used to direct the slave to replicate from a given master. These options specify the name of the log file and the starting position within the file, but with GTIDs the slave does not need this nonlocal data. Instead, you need to enable the `MASTER_AUTO_POSITION` option. For full instructions to configure and start masters and slaves using GTID-based replication, see Section 2.3.4, "Setting Up Replication Using GTIDs".

The `MASTER_AUTO_POSITION` option is disabled by default. If multi-source replication is enabled on the slave, you need to set the option for each applicable replication channel. Disabling the `MASTER_AUTO_POSITION` option again makes the slave revert to file-based replication, in which case you must also specify one or both of the `MASTER_LOG_FILE` or `MASTER_LOG_POS` options.
When a replication slave has GTIDs enabled (GTID_MODE=ON, ON_PERMISSIVE, or OFF_PERMISSIVE) and the MASTER_AUTO_POSITION option enabled, auto-positioning is activated for connection to the master. The master must have GTID_MODE=ON set in order for the connection to succeed. In the initial handshake, the slave sends a GTID set containing the transactions that it has already received, committed, or both. This GTID set is equal to the union of the set of GTIDs in the gtid_executed system variable (@GLOBAL.gtid_executed), and the set of GTIDs recorded in the Performance Schema replication_connection_status table as received transactions (the result of the statement SELECT RECEIVED_TRANSACTION_SET FROM PERFORMANCE_SCHEMA.replication_connection_status).

The master responds by sending all transactions recorded in its binary log whose GTID is not included in the GTID set sent by the slave. This exchange ensures that the master only sends the transactions with a GTID that the slave has not already received or committed. If the slave receives transactions from more than one master, as in the case of a diamond topology, the auto-skip function ensures that the transactions are not applied twice.

If any of the transactions that should be sent by the master have been purged from the master's binary log, or added to the set of GTIDs in the gtid_purged system variable by another method, the master sends the error ER_MASTER_HAS_PURGED_REQUIRED_GTIDS to the slave, and replication does not start. The GTIDs of the missing purged transactions are identified and listed in the master's error log in the warning message ER_FOUND_MISSING_GTIDS. The slave cannot recover automatically from this error because parts of the transaction history that are needed to catch up with the master have been purged. Attempting to reconnect without the MASTER_AUTO_POSITION option enabled only results in the loss of the purged transactions on the slave. The correct approach to recover from this situation is for the slave to replicate the missing transactions listed in the ER_FOUND_MISSING_GTIDS message from another source, or for the slave to be replaced by a new slave created from a more recent backup. Consider revising the binary log expiration period on the master to ensure that the situation does not occur again.

If during the exchange of transactions it is found that the slave has received or committed transactions with the master's UUID in the GTID, but the master itself does not have a record of them, the master sends the error ER_SLAVE_HAS_MORE_GTIDS_THAN_MASTER to the slave and replication does not start. This situation can occur if a master that does not have sync_binlog=1 set experiences a power failure or operating system crash, and loses committed transactions that have not yet been synchronized to the binary log file, but have been received by the slave. The master and slave can diverge if any clients commit transactions on the master after it is restarted, which can lead to the situation where the master and slave are using the same GTID for different transactions. The correct approach to recover from this situation is to check manually whether the master and slave have diverged. If the same GTID is now in use for different transactions, you either need to perform manual conflict resolution for individual transactions as required, or remove either the master or the slave from the replication topology. If the issue is only missing transactions on the master, you can make the master into a slave instead, allow it to catch up with the other servers in the replication topology, and then make it a master again if needed.

2.3.4 Setting Up Replication Using GTIDs

This section describes a process for configuring and starting GTID-based replication in MySQL 5.7. This is a “cold start” procedure that assumes either that you are starting the replication master for the first time, or that it is possible to stop it; for information about provisioning replication slaves using GTIDs from a running master, see Section 2.3.5, “Using GTIDs for Failover and Scaleout”. For information about changing GTID mode on servers online, see Section 2.5, “Changing Replication Modes on Online Servers”.

The key steps in this startup process for the simplest possible GTID replication topology, consisting of one master and one slave, are as follows:

1. If replication is already running, synchronize both servers by making them read-only.
2. Stop both servers.
3. Restart both servers with GTIDs enabled and the correct options configured.

   The `mysqld` options necessary to start the servers as described are discussed in the example that follows later in this section.

4. Instruct the slave to use the master as the replication data source and to use auto-positioning. The SQL statements needed to accomplish this step are described in the example that follows later in this section.

5. Take a new backup. Binary logs containing transactions without GTIDs cannot be used on servers where GTIDs are enabled, so backups taken before this point cannot be used with your new configuration.

6. Start the slave, then disable read-only mode on both servers, so that they can accept updates.

In the following example, two servers are already running as master and slave, using MySQL's binary log position-based replication protocol. If you are starting with new servers, see Section 2.2.2, “Creating a User for Replication” for information about adding a specific user for replication connections and Section 2.2.1, “Setting the Replication Master Configuration” for information about setting the `server_id` variable. The following examples show how to store `mysqld` startup options in server's option file, see Using Option Files for more information. Alternatively you can use startup options when running `mysqld`.

Most of the steps that follow require the use of the MySQL `root` account or another MySQL user account that has the `SUPER` privilege. `mysqladmin shutdown` requires either the `SUPER` privilege or the `SHUTDOWN` privilege.

**Step 1: Synchronize the servers.** This step is only required when working with servers which are already replicating without using GTIDs. For new servers proceed to Step 3. Make the servers read-only by setting the `read_only` system variable to `ON` on each server by issuing the following:

```
mysql> SET @@GLOBAL.read_only = ON;
```

Wait for all ongoing transactions to commit or roll back. Then, allow the slave to catch up with the master. **It is extremely important that you make sure the slave has processed all updates before continuing.**

If you use binary logs for anything other than replication, for example to do point in time backup and restore, wait until you do not need the old binary logs containing transactions without GTIDs. Ideally, wait for the server to purge all binary logs, and wait for any existing backup to expire.

**Important**

It is important to understand that logs containing transactions without GTIDs cannot be used on servers where GTIDs are enabled. Before proceeding, you must be sure that transactions without GTIDs do not exist anywhere in the topology.

**Step 2: Stop both servers.** Stop each server using `mysqladmin` as shown here, where `username` is the user name for a MySQL user having sufficient privileges to shut down the server:

```
shell> mysqladmin -u username -p shutdown
```

Then supply this user's password at the prompt.

**Step 3: Start both servers with GTIDs enabled.** To enable GTID-based replication, each server must be started with GTID mode enabled by setting the `gtid_mode` variable to `ON`, and with the `enforce_gtid_consistency` variable enabled to ensure that only statements which are safe for GTID-based replication are logged. For example:

```
gtid_mode=ON
```
Using GTIDs for Failover and Scaleout

enforce-gtid-consistency=ON

In addition, you should start slaves with the `--skip-slave-start` option before configuring the slave settings. For more information on GTID related options and variables, see Section 2.6.5, “Global Transaction ID System Variables”.

It is not mandatory to have binary logging enabled in order to use GTIDs when using the `mysql.gtid_executed` Table. Masters must always have binary logging enabled in order to be able to replicate. However, slave servers can use GTIDs but without binary logging. If you need to disable binary logging on a slave server, you can do this by specifying the `--skip-log-bin` and `--log-slave-updates=OFF` options for the slave.

**Step 4: Configure the slave to use GTID-based auto-positioning.**  Tell the slave to use the master with GTID based transactions as the replication data source, and to use GTID-based auto-positioning rather than file-based positioning. Issue a `CHANGE MASTER TO` statement on the slave, including the `MASTER_AUTO_POSITION` option in the statement to tell the slave that the master's transactions are identified by GTIDs.

You may also need to supply appropriate values for the master's host name and port number as well as the user name and password for a replication user account which can be used by the slave to connect to the master; if these have already been set prior to Step 1 and no further changes need to be made, the corresponding options can safely be omitted from the statement shown here.

```
mysql> CHANGE MASTER TO
    > MASTER_HOST = host,
    > MASTER_PORT = port,
    > MASTER_USER = user,
    > MASTER_PASSWORD = password,
    > MASTER_AUTO_POSITION = 1;
```

Neither the `MASTER_LOG_FILE` option nor the `MASTER_LOG_POS` option may be used with `MASTER_AUTO_POSITION` set equal to 1. Attempting to do so causes the `CHANGE MASTER TO` statement to fail with an error.

**Step 5: Take a new backup.**  Existing backups that were made before you enabled GTIDs can no longer be used on these servers now that you have enabled GTIDs. Take a new backup at this point, so that you are not left without a usable backup.

For instance, you can execute `FLUSH LOGS` on the server where you are taking backups. Then either explicitly take a backup or wait for the next iteration of any periodic backup routine you may have set up.

**Step 6: Start the slave and disable read-only mode.**  Start the slave like this:

```
mysql> START SLAVE;
```

The following step is only necessary if you configured a server to be read-only in Step 1. To allow the server to begin accepting updates again, issue the following statement:

```
mysql> SET @@GLOBAL.read_only = OFF;
```

GTID-based replication should now be running, and you can begin (or resume) activity on the master as before. Section 2.3.5, “Using GTIDs for Failover and Scaleout”, discusses creation of new slaves when using GTIDs.

### 2.3.5 Using GTIDs for Failover and Scaleout

There are a number of techniques when using MySQL Replication with Global Transaction Identifiers (GTIDs) for provisioning a new slave which can then be used for scaleout, being promoted to master as necessary for failover. This section describes the following techniques:
• Simple replication
• Copying data and transactions to the slave
• Injecting empty transactions
• Excluding transactions with gtid_purged
• Restoring GTID mode slaves

Global transaction identifiers were added to MySQL Replication for the purpose of simplifying in general management of the replication data flow and of failover activities in particular. Each identifier uniquely identifies a set of binary log events that together make up a transaction. GTIDs play a key role in applying changes to the database: the server automatically skips any transaction having an identifier which the server recognizes as one that it has processed before. This behavior is critical for automatic replication positioning and correct failover.

The mapping between identifiers and sets of events comprising a given transaction is captured in the binary log. This poses some challenges when provisioning a new server with data from another existing server. To reproduce the identifier set on the new server, it is necessary to copy the identifiers from the old server to the new one, and to preserve the relationship between the identifiers and the actual events. This is neccessary for restoring a slave that is immediately available as a candidate to become a new master on failover or switchover.

Simple replication. The easiest way to reproduce all identifiers and transactions on a new server is to make the new server into the slave of a master that has the entire execution history, and enable global transaction identifiers on both servers. See Section 2.3.4, “Setting Up Replication Using GTIDs”, for more information.

Once replication is started, the new server copies the entire binary log from the master and thus obtains all information about all GTIDs.

This method is simple and effective, but requires the slave to read the binary log from the master; it can sometimes take a comparatively long time for the new slave to catch up with the master, so this method is not suitable for fast failover or restoring from backup. This section explains how to avoid fetching all of the execution history from the master by copying binary log files to the new server.

Copying data and transactions to the slave. Executing the entire transaction history can be time-consuming when the source server has processed a large number of transactions previously, and this can represent a major bottleneck when setting up a new replication slave. To eliminate this requirement, a snapshot of the data set, the binary logs and the global transaction information the source server contains can be imported to the new slave. The source server can be either the master or the slave, but you must ensure that the source has processed all required transactions before copying the data.

There are several variants of this method, the difference being in the manner in which data dumps and transactions from binary logs are transfered to the slave, as outlined here:

Data Set

1. Create a dump file using `mysqldump` on the source server. Set the `--master-data` option to include a CHANGE MASTER TO statement with binary logging information. Set the `--set-gtid-purged` option to `AUTO` (the default) or `ON`, to include information about executed transactions in the dump. Then use the `mysql` client to import the dump file on the target server.

2. Alternatively, create a data snapshot of the source server using raw data files, then copy these files to the target server, following the instructions in Section 2.2.4, “Choosing a Method for Data Snapshots”. If you use InnoDB tables, you can use
Using GTIDs for Failover and Scaleout

the `mysqlbackup` command from the MySQL Enterprise Backup component to produce a consistent snapshot. This command records the log name and offset corresponding to the snapshot to be used on the slave. MySQL Enterprise Backup is a commercial product that is included as part of a MySQL Enterprise subscription. See MySQL Enterprise Backup Overview for detailed information.

3. Alternatively, stop both the source and target servers, copy the contents of the source’s data directory to the new slave’s data directory, then restart the slave. If you use this method, the slave must be configured for GTID-based replication, in other words with `gtid_mode=ON`. For instructions and important information for this method, see Section 2.2.6, “Adding Slaves to a Replication Environment”.

Transaction History

If the source server has a complete transaction history in its binary logs (that is, the GTID set `@@GLOBAL.gtid_purged` is empty), you can use these methods.

1. Import the binary logs from the source server to the new slave using `mysqlbinlog`, with the `--read-from-remote-server` and `--read-from-remote-master` options.

2. Alternatively, copy the source server’s binary log files to the slave. You can make copies from the slave using `mysqlbinlog` with the `--read-from-remote-server` and `--raw` options. These can be read into the slave by using `mysqlbinlog > file` (without the `--raw` option) to export the binary log files to SQL files, then passing these files to the `mysql` client for processing. Ensure that all of the binary log files are processed using a single `mysql` process, rather than multiple connections. For example:

   ```shell
   shell> mysqlbinlog copied-binlog.000001 copied-binlog.000002 | mysql -u root -p
   ```

   For more information, see Using `mysqlbinlog` to Back Up Binary Log Files.

This method has the advantage that a new server is available almost immediately; only those transactions that were committed while the snapshot or dump file was being replayed still need to be obtained from the existing master. This means that the slave’s availability is not instantaneous, but only a relatively short amount of time should be required for the slave to catch up with these few remaining transactions.

Copying over binary logs to the target server in advance is usually faster than reading the entire transaction execution history from the master in real time. However, it may not always be feasible to move these files to the target when required, due to size or other considerations. The two remaining methods for provisioning a new slave discussed in this section use other means to transfer information about transactions to the new slave.

Injecting empty transactions. The master’s global `gtid_executed` variable contains the set of all transactions executed on the master. Rather than copy the binary logs when taking a snapshot to provision a new server, you can instead note the content of `gtid_executed` on the server from which the snapshot was taken. Before adding the new server to the replication chain, simply commit an empty transaction on the new server for each transaction identifier contained in the master’s `gtid_executed`, like this:

```sql
SET GTID_NEXT='aaa-bbb-ccc-ddd:N';
BEGIN;
```
Once all transaction identifiers have been reinstated in this way using empty transactions, you must flush and purge the slave's binary logs, as shown here, where $N$ is the nonzero suffix of the current binary log file name:

```sql
FLUSH LOGS;
PURGE BINARY LOGS TO 'master-bin.00000N';
```

You should do this to prevent this server from flooding the replication stream with false transactions in the event that it is later promoted to master. (The `FLUSH LOGS` statement forces the creation of a new binary log file; `PURGE BINARY LOGS` purges the empty transactions, but retains their identifiers.)

This method creates a server that is essentially a snapshot, but in time is able to become a master as its binary log history converges with that of the replication stream (that is, as it catches up with the master or masters). This outcome is similar in effect to that obtained using the remaining provisioning method, which we discuss in the next few paragraphs.

### Excluding transactions with `gtid_purged`.

The master's global `gtid_purged` variable contains the set of all transactions that have been purged from the master's binary log. As with the method discussed previously (see Injecting empty transactions), you can record the value of `gtid_executed` on the server from which the snapshot was taken (in place of copying the binary logs to the new server). Unlike the previous method, there is no need to commit empty transactions (or to issue `PURGE BINARY LOGS`); instead, you can set `gtid_purged` on the slave directly, based on the value of `gtid_executed` on the server from which the backup or snapshot was taken.

As with the method using empty transactions, this method creates a server that is functionally a snapshot, but in time is able to become a master as its binary log history converges with that of the replication master or group.

### Restoring GTID mode slaves.

When restoring a slave in a GTID based replication setup that has encountered an error, injecting an empty transaction may not solve the problem because an event does not have a GTID.

Use `mysqlbinlog` to find the next transaction, which is probably the first transaction in the next log file after the event. Copy everything up to the `COMMIT` for that transaction, being sure to include the `SET @@SESSION.GTID_NEXT`. Even if you are not using row-based replication, you can still run binary log row events in the command line client.

Stop the slave and run the transaction you copied. The `mysqlbinlog` output sets the delimiter to `/*!` */;`, so set it back:

```sql
mysql> DELIMITER ;
```

Restart replication from the correct position automatically:

```sql
mysql> SET GTID_NEXT=automatic;
mysql> RESET SLAVE;
mysql> START SLAVE;
```

## 2.3.6 Restrictions on Replication with GTIDs

Because GTID-based replication is dependent on transactions, some features otherwise available in MySQL are not supported when using it. This section provides information about restrictions on and limitations of replication with GTIDs.

### Updates involving nontransactional storage engines.

When using GTIDs, updates to tables using nontransactional storage engines such as `MyISAM` cannot be made in the same statement or transaction as updates to tables using transactional storage engines such as `InnoDB`. 
This restriction is due to the fact that updates to tables that use a nontransactional storage engine mixed with updates to tables that use a transactional storage engine within the same transaction can result in multiple GTIDs being assigned to the same transaction.

Such problems can also occur when the master and the slave use different storage engines for their respective versions of the same table, where one storage engine is transactional and the other is not. Also be aware that triggers that are defined to operate on nontransactional tables can be the cause of these problems.

In any of the cases just mentioned, the one-to-one correspondence between transactions and GTIDs is broken, with the result that GTID-based replication cannot function correctly.

CREATE TABLE ... SELECT statements.  CREATE TABLE ... SELECT statements are not allowed when using GTID-based replication. When binlog_format is set to STATEMENT, a CREATE TABLE ... SELECT statement is recorded in the binary log as one transaction with one GTID, but if ROW format is used, the statement is recorded as two transactions with two GTIDs. If a master used STATEMENT format and a slave used ROW format, the slave would be unable to handle the transaction correctly, therefore the CREATE TABLE ... SELECT statement is disallowed with GTIDs to prevent this scenario.

Temporary tables.  CREATE TEMPORARY TABLE and DROP TEMPORARY TABLE statements are not supported inside transactions, procedures, functions, and triggers when using GTIDs (that is, when the enforce_gtid_consistency system variable is set to ON). It is possible to use these statements with GTIDs enabled, but only outside of any transaction, and only with autocommit=1.

Preventing execution of unsupported statements.  To prevent execution of statements that would cause GTID-based replication to fail, all servers must be started with the --enforce-gtid-consistency option when enabling GTIDs. This causes statements of any of the types discussed previously in this section to fail with an error.

Note that --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.

For information about other required startup options when enabling GTIDs, see Section 2.3.4, “Setting Up Replication Using GTIDs”.

Skipping transactions.  sql_slave_skip_counter is not supported when using GTIDs. If you need to skip transactions, use the value of the master's gtid_executed variable instead. For instructions, see Section 2.7.3, “Skipping Transactions”.

Ignoring servers.  The IGNORE_SERVER_IDS option of the CHANGE MASTER TO statement is deprecated when using GTIDs, because transactions that have already been applied are automatically ignored. Before starting GTID-based replication, check for and clear all ignored server ID lists that have previously been set on the servers involved. The SHOW SLAVE STATUS statement, which can be issued for individual channels, displays the list of ignored server IDs if there is one. If there is no list, the Replicate_Ignore_Server_Ids field is blank.

GTID mode and mysqldump.  It is possible to import a dump made using mysqldump into a MySQL server running with GTID mode enabled, provided that there are no GTIDs in the target server's binary log.

GTID mode and mysql_upgrade.  When the server is running with global transaction identifiers (GTIDs) enabled (gtid_mode=ON), do not enable binary logging by mysql_upgrade (the --write-binlog option).

2.3.7 Stored Function Examples to Manipulate GTIDs

MySQL includes some built-in (native) functions for use with GTID-based replication. These functions are as follows:
### Stored Function Examples to Manipulate GTIDs

#### GTID_SUBSET(set1, set2)
Given two sets of global transaction identifiers \( \text{set1} \) and \( \text{set2} \), returns true if all GTIDs in \( \text{set1} \) are also in \( \text{set2} \). Returns false otherwise.

#### GTID_SUBTRACT(set1, set2)
Given two sets of global transaction identifiers \( \text{set1} \) and \( \text{set2} \), returns only those GTIDs from \( \text{set1} \) that are not in \( \text{set2} \).

#### WAIT_FOR_EXECUTED_GTID_SET(gtgid_set, timeout)
Wait until the server has applied all of the transactions whose global transaction identifiers are contained in \( \text{gtid_set} \). The optional timeout stops the function from waiting after the specified number of seconds have elapsed.

#### WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS(gtgid_set, timeout[, channel])
Like \( \text{WAIT_FOR_EXECUTED_GTID_SET()} \), but for a single started replication channel. Use \( \text{WAIT_FOR_EXECUTED_GTID_SET()} \) instead to ensure all channels are covered in all states.

For details of these functions, see [Functions Used with Global Transaction Identifiers (GTIDs)](#).

You can define your own stored functions to work with GTIDs. For information on defining stored functions, see [Stored Objects](#). The following examples show some useful stored functions that can be created based on the built-in \( \text{GTID_SUBSET()} \) and \( \text{GTID_SUBTRACT()} \) functions.

Note that in these stored functions, the delimiter command has been used to change the MySQL statement delimiter to a vertical bar, as follows:

```sql
mysql> delimiter |
```

All of these functions take string representations of GTID sets as arguments, so GTID sets must always be quoted when used with them.

This function returns nonzero (true) if two GTID sets are the same set, even if they are not formatted in the same way.

```sql
CREATE FUNCTION GTID_IS_EQUAL(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT)
RETURNS INT
RETURN GTID_SUBSET(gtid_set_1, gtid_set_2) AND GTID_SUBSET(gtid_set_2, gtid_set_1) |
```

This function returns nonzero (true) if two GTID sets are disjoint.

```sql
CREATE FUNCTION GTID_IS_DISJOINT(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT)
RETURNS INT
RETURN GTID_SUBSET(gtid_set_1, GTID_SUBTRACT(gtid_set_1, gtid_set_2)) |
```

This function returns nonzero (true) if two GTID sets are disjoint, and \( \text{sum} \) is the union of the two sets.

```sql
CREATE FUNCTION GTID_IS_DISJOINT_UNION(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT, sum LONGTEXT)
RETURNS INT
RETURN GTID_IS_EQUAL(GTID_SUBTRACT(sum, gtid_set_1), gtid_set_2) AND GTID_IS_EQUAL(GTID_SUBTRACT(sum, gtid_set_2), gtid_set_1) |
```

This function returns a normalized form of the GTID set, in all uppercase, with no whitespace and no duplicates. The UUIDs are arranged in alphabetic order and intervals are arranged in numeric order.

```sql
CREATE FUNCTION GTID_NORMALIZE(g LONGTEXT)
RETURNS LONGTEXT
RETURN GTID_SUBTRACT(g, '') |
```

This function returns the union of two GTID sets.

```sql
CREATE FUNCTION GTID_UNION(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT)
RETURNS LONGTEXT
RETURN GTID_NORMALIZE(CONCAT(gtid_set_1, ',', gtid_set_2)) |
```

This function returns the intersection of two GTID sets.
CREATE FUNCTION GTID_INTERSECTION(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT)
RETURNS LONGTEXT
RETURN GTID_SUBTRACT(gtid_set_1, GTID_SUBTRACT(gtid_set_1, gtid_set_2))|

This function returns the symmetric difference between two GTID sets, that is, the GTIDs that exist in \texttt{gtid\_set\_1} but not in \texttt{gtid\_set\_2}, and also the GTIDs that exist in \texttt{gtid\_set\_2} but not in \texttt{gtid\_set\_1}.

CREATE FUNCTION GTID_SYMMETRIC_DIFFERENCE(gtid_set_1 LONGTEXT, gtid_set_2 LONGTEXT)
RETURNS LONGTEXT
RETURN GTID_SUBTRACT(CONCAT(gtid_set_1, ',', gtid_set_2), GTID_INTERSECTION(gtid_set_1, gtid_set_2))|

This function removes from a GTID set all the GTIDs from a specified origin, and returns the remaining GTIDs, if any. The UUID is the identifier used by the server where the transaction originated, which is normally the \texttt{server\_uuid} value.

CREATE FUNCTION GTID_SUBTRACT_UUID(gtid_set LONGTEXT, uuid TEXT)
RETURNS LONGTEXT
RETURN GTID_SUBTRACT(gtid_set, CONCAT(UUID, ':1-', (1 << 63) - 2))|

This function reverses the previously listed function to return only those GTIDs from the GTID set that originate from the server with the specified identifier (UUID).

CREATE FUNCTION GTID_INTERSECTION_WITH_UUID(gtid_set LONGTEXT, uuid TEXT)
RETURNS LONGTEXT
RETURN GTID_SUBTRACT(gtid_set, GTID_SUBTRACT_UUID(gtid_set, uuid))|

**Example 2.1 Verifying that a replication slave is up to date**

The built-in functions \texttt{GTID\_SUBSET} and \texttt{GTID\_SUBTRACT} can be used to check that a replication slave has applied at least every transaction that a master has applied.

To perform this check with \texttt{GTID\_SUBSET}, execute the following statement on the slave:

\begin{verbatim}
SELECT GTID_SUBSET(master_gtid_executed, slave_gtid_executed)
\end{verbatim}

If this returns 0 (false), some GTIDs in \texttt{master_gtid_executed} are not present in \texttt{slave_gtid_executed}, so the master has applied some transactions that the slave has not applied, and the slave is therefore not up to date.

To perform the check with \texttt{GTID\_SUBTRACT}, execute the following statement on the slave:

\begin{verbatim}
SELECT GTID_SUBTRACT(master_gtid_executed, slave_gtid_executed)
\end{verbatim}

This statement returns any GTIDs that are in \texttt{master_gtid_executed} but not in \texttt{slave_gtid_executed}. If any GTIDs are returned, the master has applied some transactions that the slave has not applied, and the slave is therefore not up to date.

**Example 2.2 Backup and restore scenario**

The stored functions \texttt{GTID\_IS\_EQUAL}, \texttt{GTID\_IS\_DISJOINT}, and \texttt{GTID\_IS\_DISJOINT\_UNION} could be used to verify backup and restore operations involving multiple databases and servers. In this example scenario, \texttt{server1} contains database \texttt{db1}, and \texttt{server2} contains database \texttt{db2}. The goal is to copy database \texttt{db2} to \texttt{server1}, and the result on \texttt{server1} should be the union of the two databases. The procedure used is to back up \texttt{server2} using \texttt{mysqlpump} or \texttt{mysqldump}, then restore this backup on \texttt{server1}.

Provided the backup program's option \texttt{--set-gtid-purged} was set to \texttt{ON} or the default of \texttt{AUTO}, the program's output contains a SET \texttt{@@GLOBAL.gtid_purged} statement that will add the \texttt{gtid\_executed} set from \texttt{server2} to the \texttt{gtid\_purged} set on \texttt{server1}. The \texttt{gtid\_purged} set contains the GTIDs of all the transactions that have been committed on a server but do not exist in any binary log file on the server. When database \texttt{db2} is copied to \texttt{server1}, the GTIDs of the transactions
committed on server2, which are not in the binary log files on server1, must be added to server1’s gtid_purged set to make the set complete.

The stored functions can be used to assist with the following steps in this scenario:

• **Use GTID_IS_EQUAL** to verify that the backup operation computed the correct GTID set for the SET @@GLOBAL.gtid_purged statement. On server2, extract that statement from the mysqlpump or mysqldump output, and store the GTID set into a local variable, such as $gtid_purged_set. Then execute the following statement:

  server2> SELECT GTID_IS_EQUAL($gtid_purged_set, @@GLOBAL.gtid_executed);

  If the result is 1, the two GTID sets are equal, and the set has been computed correctly.

• **Use GTID_IS_DISJOINT** to verify that the GTID set in the mysqlpump or mysqldump output does not overlap with the gtid_executed set on server1. If there is any overlap, with identical GTIDs present on both servers for some reason, you will see errors when copying database db2 to server1. To check, on server1, extract and store the gtid_purged set from the output into a local variable as above, then execute the following statement:

  server1> SELECT GTID_IS_DISJOINT($gtid_purged_set, @@GLOBAL.gtid_executed);

  If the result is 1, there is no overlap between the two GTID sets, so no duplicate GTIDs are present.

• **Use GTID_IS_DISJOINT_UNION** to verify that the restore operation resulted in the correct GTID state on server1. Before restoring the backup, on server1, obtain the existing gtid_executed set by executing the following statement:

  server1> SELECT @@GLOBAL.gtid_executed;

  Store the result in a local variable $original_gtid_executed. Also store the gtid_purged set in a local variable as described above. When the backup from server2 has been restored onto server1, execute the following statement to verify the GTID state:

  server1> SELECT GTID_IS_DISJOINT_UNION($original_gtid_executed, $gtid_purged_set, @@GLOBAL.gtid_executed);

  If the result is 1, the stored function has verified that the original gtid_executed set from server1 ($original_gtid_executed) and the gtid_purged set that was added from server2 ($gtid_purged_set) have no overlap, and also that the updated gtid_executed set on server1 now consists of the previous gtid_executed set from server1 plus the gtid_purged set from server2, which is the desired result. Ensure that this check is carried out before any further transactions take place on server1, otherwise the new transactions in the gtid_executed set will cause it to fail.

**Example 2.3 Selecting the most up-to-date slave for manual failover**

The stored function GTID_UNION could be used to identify the most up-to-date replication slave from a set of slaves, in order to perform a manual failover operation after a replication master has stopped unexpectedly. If some of the slaves are experiencing replication lag, this stored function can be used to compute the most up-to-date slave without waiting for all the slaves to apply their existing relay logs, and therefore to minimize the failover time. The function can return the union of the gtid_executed set on each slave with the set of transactions received by the slave, which is recorded in the Performance Schema table replication_connection_status. You can compare these results to find which slave’s record of transactions is the most up-to-date, even if not all of the transactions have been committed yet.

On each replication slave, compute the complete record of transactions by issuing the following statement:

```
SELECT GTID_UNION(RECEIVED_TRANSACTION_SET, @@GLOBAL.gtid_executed)
FROM performance_schema.replication_connection_status
```
WHERE channel_name = 'name';

You can then compare the results from each slave to see which one has the most up-to-date record of transactions, and use this slave as the new replication master.

**Example 2.4 Checking for extraneous transactions on a replication slave**

The stored function `GTID_SUBTRACT_UUID` could be used to check whether a replication slave has received transactions that did not originate from its designated master or masters. If it has, there might be an issue with your replication setup, or with a proxy, router, or load balancer. This function works by removing from a GTID set all the GTIDs from a specified originating server, and returning the remaining GTIDs, if any.

For a replication slave with a single master, issue the following statement, giving the identifier of the originating replication master, which is normally the `server_uuid` value:

```
SELECT GTID_SUBTRACT_UUID(@@GLOBAL.gtid_executed, server_uuid_of_master);
```

If the result is not empty, the transactions returned are extra transactions that did not originate from the designated master.

For a slave in a multi-master replication topology, repeat the function, for example:

```
SELECT GTID_SUBTRACT_UUID(GTID_SUBTRACT_UUID(@@GLOBAL.gtid_executed, server_uuid_of_master_1), server_uuid_of_master_2);
```

If the result is not empty, the transactions returned are extra transactions that did not originate from any of the designated masters.

**Example 2.5 Verifying that a server in a replication topology is read-only**

The stored function `GTID_INTERSECTION_WITH_UUID` could be used to verify that a server has not originated any GTIDs and is in a read-only state. The function returns only those GTIDs from the GTID set that originate from the server with the specified identifier. If any of the transactions in the server's `gtid_executed` set have the server's own identifier, the server itself originated those transactions.

You can issue the following statement on the server to check:

```
SELECT GTID_INTERSECTION_WITH_UUID(@@GLOBAL.gtid_executed, my_server_uuid);
```

**Example 2.6 Validating an additional slave in a multi-master replication setup**

The stored function `GTID_INTERSECTION_WITH_UUID` could be used to find out if a slave attached to a multi-master replication setup has applied all the transactions originating from one particular master. In this scenario, `master1` and `master2` are both masters and slaves and replicate to each other. `master2` also has its own replication slave. The replication slave will also receive and apply `master1`'s transactions if `master2` is configured with `log_slave_updates=ON`, but it will not do so if `master2` uses `log_slave_updates=OFF`. Whatever the case, we currently only want to find out if the replication slave is up to date with `master2`. In this situation, the stored function `GTID_INTERSECTION_WITH_UUID` can be used to identify the transactions that `master2` originated, discarding the transactions that `master2` has replicated from `master1`. The built-in function `GTID_SUBSET` can then be used to compare the result to the `gtid_executed` set on the slave. If the slave is up to date with `master2`, the `gtid_executed` set on the slave contains all the transactions in the intersection set (the transactions that originated from `master2`).

To carry out this check, store `master2`'s `gtid_executed` set, `master2`'s server UUID, and the slave's `gtid_executed` set, into client-side variables as follows:

```
$master2_gtid_executed :=
master2> SELECT @@GLOBAL.gtid_executed;
$master2_server_uuid :=
master2> SELECT @@GLOBAL.server_uuid;
$slave_gtid_executed :=
slave> SELECT @@GLOBAL.gtid_executed;
```
Then use `GTID_INTERSECTION_WITH_UUID` and `GTID_SUBSET` with these variables as input, as follows:

```sql
SELECT GTID_SUBSET(GTID_INTERSECTION_WITH_UUID($master2_gtid_executed, $master2_server_uuid), $slave_gtid_executed);
```

The server identifier from `master2` ($`master2_server_uuid`) is used with `GTID_INTERSECTION_WITH_UUID` to identify and return only those GTIDs from `master2`'s `gtid_executed` set that originated on `master2`, omitting those that originated on `master1`. The resulting GTID set is then compared with the set of all executed GTIDs on the slave, using `GTID_SUBSET`. If this statement returns nonzero (true), all the identified GTIDs from `master2` (the first set input) are also in the slave's `gtid_executed` set (the second set input), meaning that the slave has replicated all the transactions that originated from `master2`.

### 2.4 MySQL Multi-Source Replication

MySQL multi-source replication enables a replication slave to receive transactions from multiple immediate masters in parallel. In a multi-source replication topology, a slave creates a replication channel for each master that it should receive transactions from. For more information on how replication channels function, see Section 5.3, “Replication Channels”.

You might choose to implement multi-source replication to achieve goals like these:

- Backing up multiple servers to a single server.
- Merging table shards.
- Consolidating data from multiple servers to a single server.

Multi-source replication does not implement any conflict detection or resolution when applying transactions, and those tasks are left to the application if required.

#### Note

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

A multi-source replication slave can also be set up as a multi-threaded replication slave, by setting the `slave_parallel_workers` system variable to a value greater than 0. When you do this on a multi-source replication slave, each channel on the slave 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.

This section provides tutorials on how to configure masters and slaves for multi-source replication, how to start, stop and reset multi-source slaves, and how to monitor multi-source replication.

#### 2.4.1 Configuring Multi-Source Replication

A multi-source replication topology requires at least two masters and one slave configured. In these tutorials, we will assume you have two masters `master1` and `master2`, and a replication slave `slavehost`. The slave will replicate one database from each of the masters, `db1` from `master1` and `db2` from `master2`.

Masters in a multi-source replication topology can be configured to use either GTID-based replication, or binary log position-based replication. See Section 2.3.4, “Setting Up Replication Using GTIDs” for...
how to configure a master using GTID-based replication. See Section 2.2.1, “Setting the Replication Master Configuration” for how to configure a master using file position based replication.

Slaves in a multi-source replication topology require TABLE repositories for the master info log and relay log info log, as specified by the master_info_repository and relay_log_info_repository system variables. Multi-source replication is not compatible with FILE repositories.

To modify an existing replication slave that is using a FILE repository for the slave status logs to use TABLE repositories, you can convert the existing replication repositories dynamically by using the mysql client to issue the following statements on the slave:

```
mysql> STOP SLAVE;
mysql> SET GLOBAL master_info_repository = 'TABLE';
mysql> SET GLOBAL relay_log_info_repository = 'TABLE';
```

Create a suitable user account on all the masters that the slave can use to connect. You can use the same account on all the masters, or a different account on each. If you create an account solely for the purposes of replication, that account needs only the REPLICATION SLAVE privilege. For example, to set up a new user, ted, that can connect from the replication slave slavehost, use the mysql client to issue these statements on each of the masters:

```
mysql> CREATE USER 'ted'@'slavehost' IDENTIFIED BY 'password';
mysql> GRANT REPLICATION SLAVE ON *.* TO 'ted'@'slavehost';
```

For more details, see Section 2.2.2, “Creating a User for Replication”.

### 2.4.2 Provisioning a Multi-Source Replication Slave for GTID-Based Replication

If the masters in the multi-source replication topology have existing data, it can save time to provision the slave with the relevant data before starting replication. In a multi-source replication topology, copying the data directory cannot be used to provision the slave with data from all of the masters, and you might also want to replicate only specific databases from each master. The best strategy for provisioning such a slave is therefore to use mysqldump to create an appropriate dump file on each master, then use the mysql client to import the dump file on the slave.

If you are using GTID-based replication, you need to pay attention to the SET @@GLOBAL.gtid_purged statement that mysqldump places in the dump output. This statement transfers the GTIDs for the transactions executed on the master to the slave, and the slave requires this information. However, for any case more complex than provisioning one new, empty slave from one master, you need to check what effect the statement will have in the slave's MySQL release, and handle the statement accordingly. The following guidance summarizes suitable actions, but for more details, see the mysqldump documentation.

In MySQL 5.6 and 5.7, the SET @@GLOBAL.gtid_purged statement written by mysqldump replaces the value of gtid_purged on the slave. Also in those releases that value can only be changed when the slave's record of transactions with GTIDs (the gtid_executed set) is empty. In a multi-source replication topology, you must therefore remove the SET @@GLOBAL.gtid_purged statement from the dump output before replaying the dump files, because you will not be able to apply a second or subsequent dump file including this statement. As an alternative to removing the SET @@GLOBAL.gtid_purged statement, if you are provisioning the slave with two partial dumps from the same master, and the GTID set in the second dump is the same as the first (so no new transactions have been executed on the master in between the dumps), you can set mysqldump's --set-gtid-purged option to OFF when you output the second dump file, to omit the statement.

For MySQL 5.6 and 5.7, these limitations mean all the dump files from the masters must be applied in a single operation on a slave with an empty gtid_executed set. You can clear a slave's GTID execution history by issuing RESET MASTER on the slave, but if you have other, wanted transactions with GTIDs on the slave, choose an alternative method of provisioning from those described in Section 2.3.5, “Using GTIDs for Failover and Scaleout”.

---

**Note:** The content provided is a natural representation of the document page, focusing on the key sections and commands related to provisioning a multi-source replication slave for GTID-based replication. The example commands and guidelines are intended to illustrate the process and considerations involved in configuring and provisioning such a slave.
In the following provisioning example, we assume that the `SET @@GLOBAL.gtid_purged` statement needs to be removed from the files and handled manually. We also assume that there are no wanted transactions with GTIDs on the slave before provisioning starts.

1. To create dump files for a database named `db1` on `master1` and a database named `db2` on `master2`, run `mysqldump` for `master1` as follows:

   ```
   mysqldump -u<user> -p<password> --single-transaction --triggers --routines --set-gtid-purged=ON --databases db1 > dumpM1.sql
   ```

   Then run `mysqldump` for `master2` as follows:

   ```
   mysqldump -u<user> -p<password> --single-transaction --triggers --routines --set-gtid-purged=ON --databases db2 > dumpM2.sql
   ```

2. Record the `gtid_purged` value that `mysqldump` added to each of the dump files. For example, for dump files created on MySQL 5.6 or 5.7, you can extract the value like this:

   ```
   cat dumpM1.sql | grep GTID_PURGED | cut -f2 -d'=' | cut -f2 -d$'"'
   cat dumpM2.sql | grep GTID_PURGED | cut -f2 -d'=' | cut -f2 -d$'"'
   ```

   The result in each case should be a GTID set, for example:

   - `master2: 224DA167-0C0C-11E8-8442-00059A3C7B00:1-2695`

3. Remove the line from each dump file that contains the `SET @@GLOBAL.gtid_purged` statement. For example:

   ```
   sed '/GTID_PURGED/d' dumpM1.sql > dumpM1_nopurge.sql
   sed '/GTID_PURGED/d' dumpM2.sql > dumpM2_nopurge.sql
   ```

4. Use the `mysql` client to import each edited dump file into the slave. For example:

   ```
   mysql -u<user> -p<password> < dumpM1_nopurge.sql
   mysql -u<user> -p<password> < dumpM2_nopurge.sql
   ```

5. On the slave, issue `RESET MASTER` to clear the GTID execution history (assuming, as explained above, that all the dump files have been imported and that there are no wanted transactions with GTIDs on the slave). Then issue a `SET @@GLOBAL.gtid_purged` statement to set the `gtid_purged` value to the union of all the GTID sets from all the dump files, as you recorded in Step 2. For example:

   ```
   mysql> RESET MASTER;
   mysql> SET @@GLOBAL.gtid_purged = "2174B383-5441-11E8-B90A-C80AA9429562:1-1029, 224DA167-0C0C-11E8-8442-00059A3C7B00:1-2695"
   ```

   If there are, or might be, overlapping transactions between the GTID sets in the dump files, you can use the stored functions described in Section 2.3.7, "Stored Function Examples to Manipulate GTIDs" to check this beforehand and to calculate the union of all the GTID sets.

### 2.4.3 Adding GTID-Based Masters to a Multi-Source Replication Slave

These steps assume you have enabled GTIDs for transactions on the masters using `gtid_mode=ON`, created a replication user, ensured that the slave is using `TABLE` based replication repositories, and provisioned the slave with data from the masters if appropriate.

Use the `CHANGE MASTER TO` statement to configure a replication channel for each master on the replication slave (see Section 5.3, "Replication Channels"). The `FOR CHANNEL` clause is used to specify the channel. For GTID-based replication, GTID auto-positioning is used to synchronize with the master (see Section 2.3.3, "GTID Auto-Positioning"). The `MASTER_AUTO_POSITION` option is set to specify the use of auto-positioning.

For example, to add `master1` and `master2` as masters to the replication slave, use the `mysql` client to issue the `CHANGE MASTER TO` statement twice on the slave, like this:

```
mysql> CHANGE MASTER TO MASTER_HOST="master1", MASTER_USER="ted", \
```
Adding a Binary Log Based Master to a Multi-Source Replication Slave

These steps assume that you have enabled binary logging on the master using `--log-bin`, the slave is using `TABLE` based replication repositories, and that you have enabled a replication user and noted the current binary log position. You need to know the current `MASTER_LOG_FILE` and `MASTER_LOG_POSITION`.

Use the `CHANGE MASTER TO` statement to configure a replication channel for each master on the replication slave (see Section 5.3, “Replication Channels”). The `FOR CHANNEL` clause is used to specify the channel. For example, to add `master1` and `master2` as masters to the replication slave, use the `mysql` client to issue the `CHANGE MASTER TO` statement twice on the slave, like this:

```sql
mysql> CHANGE MASTER TO MASTER_HOST="master1", MASTER_USER="ted", MASTER_PASSWORD="password", \ MASTER_LOG_FILE='master1-bin.000006', MASTER_LOG_POS=628 FOR CHANNEL "master_1";
mysql> CHANGE MASTER TO MASTER_HOST="master2", MASTER_USER="ted", MASTER_PASSWORD="password", \ MASTER_LOG_FILE='master2-bin.000018', MASTER_LOG_POS=104 FOR CHANNEL "master_2";
```

For the full syntax of the `CHANGE MASTER TO` statement and other available options, see `CHANGE MASTER TO Statement`.

To make the replication slave replicate only database `db1` from `master1`, and only database `db2` from `master2`, use the `mysql` client to issue the `CHANGE REPLICATION FILTER` statement for each channel, like this:

```sql
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db1.%') FOR CHANNEL "master_1";
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db2.%') FOR CHANNEL "master_2";
```

For the full syntax of the `CHANGE REPLICATION FILTER` statement and other available options, see `CHANGE REPLICATION FILTER Statement`.

2.4.5 Starting Multi-Source Replication Slaves

Once you have added channels for all of the replication masters, issue a `START SLAVE` statement to start replication. When you have enabled multiple channels on a slave, you can choose to either start all channels, or select a specific channel to start. For example, to start the two channels separately, use the `mysql` client to issue the following statements:

```sql
mysql> START SLAVE FOR CHANNEL "master_1";
mysql> START SLAVE FOR CHANNEL "master_2";
```

For the full syntax of the `START SLAVE` command and other available options, see `START SLAVE Statement`.

To verify that both channels have started and are operating correctly, you can issue `SHOW SLAVE STATUS` statements on the slave, for example:
2.4.6 Stopping Multi-Source Replication Slaves

The `STOP SLAVE` statement can be used to stop a multi-source replication slave. By default, if you use the `STOP SLAVE` statement on a multi-source replication slave all channels are stopped. Optionally, use the `FOR CHANNEL channel` clause to stop only a specific channel.

- To stop all currently configured replication channels:
  ```sql
  STOP SLAVE;
  ```

- To stop only a named channel, use a `FOR CHANNEL channel` clause:
  ```sql
  STOP SLAVE FOR CHANNEL "master_1";
  ```

For the full syntax of the `STOP SLAVE` command and other available options, see `STOP SLAVE Statement`.

2.4.7 Resetting Multi-Source Replication Slaves

The `RESET SLAVE` statement can be used to reset a multi-source replication slave. By default, if you use the `RESET SLAVE` statement on a multi-source replication slave all channels are reset. Optionally, use the `FOR CHANNEL channel` clause to reset only a specific channel.

- To reset all currently configured replication channels:
  ```sql
  RESET SLAVE;
  ```

- To reset only a named channel, use a `FOR CHANNEL channel` clause:
  ```sql
  RESET SLAVE FOR CHANNEL "master_1";
  ```

For GTID-based replication, note that `RESET SLAVE` has no effect on the slave’s GTID execution history. If you want to clear this, issue `RESET MASTER` on the slave.

`RESET SLAVE` makes the slave forget its replication position, and clears the relay log, but it does not change any replication connection parameters (such as the master host) or replication filters. If you want to remove these for a channel, issue `RESET SLAVE ALL`.

For the full syntax of the `RESET SLAVE` command and other available options, see `RESET SLAVE Statement`.

2.4.8 Multi-Source Replication Monitoring

To monitor the status of replication channels the following options exist:

- Using the replication Performance Schema tables. The first column of these tables is `Channel_Name`. This enables you to write complex queries based on `Channel_Name` as a key. See `Performance Schema Replication Tables`.

- Using `SHOW SLAVE STATUS FOR CHANNEL channel`. By default, if the `FOR CHANNEL channel` clause is not used, this statement shows the slave status for all channels with one row per channel. The identifier `Channel_name` is added as a column in the result set. If a `FOR CHANNEL channel` clause is provided, the results show the status of only the named replication channel.

  Note
  
  The `SHOW VARIABLES` statement does not work with multiple replication channels. The information that was available through these variables has been
Changing Replication Modes on Online Servers

2.4.8.1 Monitoring Channels Using Performance Schema Tables

This section explains how to use the replication Performance Schema tables to monitor channels. You can choose to monitor all channels, or a subset of the existing channels.

To monitor the connection status of all channels:

mysql> SELECT * FROM replication_connection_status\G;
*************************** 1. row ***************************
CHANNEL_NAME: master_1
GROUP_NAME:
SOURCE_UUID: 046e41f8-a223-11e4-a975-0811960cc264
THREAD_ID: 24
SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 0
LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
RECEIVED_TRANSACTION_SET: 046e41f8-a223-11e4-a975-0811960cc264:4-37
LAST_ERROR_NUMBER: 0
LAST_ERROR_MESSAGE:
LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
*************************** 2. row ***************************
CHANNEL_NAME: master_2
GROUP_NAME:
SOURCE_UUID: 7475e474-a223-11e4-a978-0811960cc264
THREAD_ID: 26
SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 0
LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
RECEIVED_TRANSACTION_SET: 7475e474-a223-11e4-a978-0811960cc264:4-6
LAST_ERROR_NUMBER: 0
LAST_ERROR_MESSAGE:
LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
2 rows in set (0.00 sec)

In the above output there are two channels enabled, and as shown by the CHANNEL_NAME field they are called master_1 and master_2.

The addition of the CHANNEL_NAME field enables you to query the Performance Schema tables for a specific channel. To monitor the connection status of a named channel, use a WHERE CHANNEL_NAME=channel clause:

mysql> SELECT * FROM replication_connection_status WHERE CHANNEL_NAME='master_1'\G
*************************** 1. row ***************************
CHANNEL_NAME: master_1
GROUP_NAME:
SOURCE_UUID: 046e41f8-a223-11e4-a975-0811960cc264
THREAD_ID: 24
SERVICE_STATE: ON
COUNT_RECEIVED_HEARTBEATS: 0
LAST_HEARTBEAT_TIMESTAMP: 0000-00-00 00:00:00
RECEIVED_TRANSACTION_SET: 046e41f8-a223-11e4-a975-0811960cc264:4-37
LAST_ERROR_NUMBER: 0
LAST_ERROR_MESSAGE:
LAST_ERROR_TIMESTAMP: 0000-00-00 00:00:00
1 row in set (0.00 sec)

Similarly, the WHERE CHANNEL_NAME=channel clause can be used to monitor the other replication Performance Schema tables for a specific channel. For more information, see Performance Schema Replication Tables.

2.5 Changing Replication Modes on Online Servers
This section describes how to change the mode of replication being used without having to take the server offline.

### 2.5.1 Replication Mode Concepts

To be able to safely configure the replication mode of an online server it is important to understand some key concepts of replication. This section explains these concepts and is essential reading before attempting to modify the replication mode of an online server.

The modes of replication available in MySQL rely on different techniques for identifying transactions which are logged. The types of transactions used by replication are as follows:

- **GTID transactions** are identified by a global transaction identifier (GTID) in the form `UUID:NUMBER`. Every GTID transaction in a log is always preceded by a `Gtid_log_event`. GTID transactions can be addressed using either the GTID or using the file name and position.

- **Anonymous transactions** do not have a GTID assigned, and MySQL ensures that every anonymous transaction in a log is preceded by an `Anonymous_gtid_log_event`. In previous versions, anonymous transactions were not preceded by any particular event. Anonymous transactions can only be addressed using file name and position.

When using GTIDs you can take advantage of auto-positioning and automatic fail-over, as well as use `WAIT_FOR_EXECUTED_GTID_SET()`, `session_track_gtids`, and monitor replicated transactions using Performance Schema tables. With GTIDs enabled you cannot use `sql_slave_skip_counter`, instead use empty transactions.

Transactions in a relay log that was received from a master running a previous version of MySQL may not be preceded by any particular event at all, but after being replayed and logged in the slave’s binary log, they are preceded with an `Anonymous_gtid_log_event`.

The ability to configure the replication mode online means that the `gtid_mode` and `enforce_gtid_consistency` variables are now both dynamic and can be set from a top-level statement by an account that has privileges sufficient to set global system variables. See [System Variable Privileges](#). In previous versions, both of these variables could only be configured using the appropriate option at server start, meaning that changes to the replication mode required a server restart. In all versions `gtid_mode` could be set to `OFF` or `ON`, which corresponded to whether GTIDs were used to identify transactions or not. When `gtid_mode=ON` it is not possible to replicate anonymous transactions, and when `gtid_mode=OFF` only anonymous transactions can be replicated.

As of MySQL 5.7.6, the `gtid_mode` variable has two additional states, `OFF_PERMISSIVE` and `ON_PERMISSIVE`. When `gtid_mode=OFF_PERMISSIVE` then *new* transactions are anonymous while permitting replicated transactions to be either GTID or anonymous transactions. When `gtid_mode=ON_PERMISSIVE` then *new* transactions use GTIDs while permitting replicated transactions to be either GTID or anonymous transactions. This means it is possible to have a replication topology that has servers using both anonymous and GTID transactions. For example a master with `gtid_mode=ON` could be replicating to a slave with `gtid_mode=ON_PERMISSIVE`. The valid values for `gtid_mode` are as follows and in this order:

- **OFF**
- **OFF_PERMISSIVE**
- **ON_PERMISSIVE**
- **ON**

It is important to note that the state of `gtid_mode` can only be changed by one step at a time based on the above order. 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`. This is to ensure that the process of changing from anonymous transactions to GTID transactions online is correctly handled by the server. When you switch between `gtid_mode=ON` and `gtid_mode=OFF`, the GTID state (in other words the value of
gtid_executed) is persistent. This ensures that the GTID set that has been applied by the server is always retained, regardless of changes between types of gtid_mode.

As part of the changes introduced by MySQL 5.7.6, the fields related to GTIDs have been modified so that they display the correct information regardless of the currently selected gtid_mode. This means that fields which display GTID sets, such as gtid_executed, gtid_purged, RECEIVED_TRANSACTION_SET in the replication_connection_status Performance Schema table, and the GTID related results of SHOW SLAVE STATUS, now return the empty string when there are no GTIDs present. Fields that display a single GTID, such as CURRENT_TRANSACTION in the Performance Schema replication_applier_status_by_worker table, now display ANONYMOUS when GTID transactions are not being used.

Replication from a master using gtid_mode=ON provides the ability to use auto-positioning, configured using the CHANGE MASTER TO MASTER_AUTO_POSITION = 1; statement. The replication topology being used impacts on whether it is possible to enable auto-positioning or not, as this feature relies on GTIDs and is not compatible with anonymous transactions. An error is generated if auto-positioning is enabled and an anonymous transaction is encountered. It is strongly recommended to ensure there are no anonymous transactions remaining in the topology before enabling auto-positioning, see Section 2.5.2, “Enabling GTID Transactions Online”. The valid combinations of gtid_mode and auto-positioning on master and slave are shown in the following table, where the master’s gtid_mode is shown on the horizontal and the slave’s gtid_mode is on the vertical:

Table 2.1 Valid Combinations of Master and Slave gtid_mode

<table>
<thead>
<tr>
<th>Master/Slave</th>
<th>gtid_mode</th>
</tr>
</thead>
<tbody>
<tr>
<td>Master</td>
<td>Slave</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF_PERMISSIVE</td>
</tr>
<tr>
<td>OFF</td>
<td>ON_PERMISSIVE</td>
</tr>
<tr>
<td>OFF</td>
<td>ON</td>
</tr>
<tr>
<td>OFF_PERMISSIVE</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF_PERMISSIVE</td>
<td>OFF_PERMISSIVE</td>
</tr>
<tr>
<td>OFF_PERMISSIVE</td>
<td>ON_PERMISSIVE</td>
</tr>
<tr>
<td>OFF_PERMISSIVE</td>
<td>ON</td>
</tr>
<tr>
<td>ON_PERMISSIVE</td>
<td>OFF</td>
</tr>
<tr>
<td>ON_PERMISSIVE</td>
<td>OFF_PERMISSIVE</td>
</tr>
<tr>
<td>ON_PERMISSIVE</td>
<td>ON_PERMISSIVE</td>
</tr>
<tr>
<td>ON_PERMISSIVE</td>
<td>ON</td>
</tr>
</tbody>
</table>

In the above table, the entries are:

- **Y**: the gtid_mode of master and slave is compatible
- **N**: the gtid_mode of master and slave is not compatible
- *****: auto-positioning can be used

The currently selected gtid_mode also impacts on the gtid_next variable. The following table shows the behavior of the server for the different values of gtid_mode and gtid_next.

Table 2.2 Valid Combinations of gtid_mode and gtid_next

<table>
<thead>
<tr>
<th>gtid_mode</th>
<th>gtid_next</th>
</tr>
</thead>
<tbody>
<tr>
<td>AUTOMATIC</td>
<td>binary</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>idempotent</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>off</td>
</tr>
<tr>
<td>ANONYMOUS</td>
<td>UUID:NUMBER</td>
</tr>
<tr>
<td>ANONYMOUS</td>
<td>GTID</td>
</tr>
<tr>
<td>ANONYMOUS</td>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>ANONYMOUS</td>
<td>AUTOMATIC</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>GTID</td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td>GTID</td>
</tr>
</tbody>
</table>

In the above table, the entries are:
Enabling GTID Transactions Online

- **ANONYMOUS**: generate an anonymous transaction.
- **Error**: generate an error and fail to execute `SET GTID_NEXT`.
- **UUID:NUMBER**: generate a GTID with the specified UUID:NUMBER.
- **New GTID**: generate a GTID with an automatically generated number.

When the binary log is off and `gtid_next` is set to **AUTOMATIC**, then no GTID is generated. This is consistent with the behavior of previous versions.

### 2.5.2 Enabling GTID Transactions Online

This section describes how to enable GTID transactions, and optionally auto-positioning, on servers that are already online and using anonymous transactions. This procedure does not require taking the server offline and is suited to use in production. However, if you have the possibility to take the servers offline when enabling GTID transactions that process is easier.

Before you start, ensure that the servers meet the following pre-conditions:

- **All servers in your topology must use MySQL 5.7.6 or later. You cannot enable GTID transactions online on any single server unless all servers which are in the topology are using this version.**
- **All servers have `gtid_mode` set to the default value **OFF**.**

The following procedure can be paused at any time and later resumed where it was, or reversed by jumping to the corresponding step of Section 2.5.3, "Disabling GTID Transactions Online", the online procedure to disable GTIDs. This makes the procedure fault-tolerant because any unrelated issues that may appear in the middle of the procedure can be handled as usual, and then the procedure continued where it was left off.

**Note**

It is crucial that you complete every step before continuing to the next step.

To enable GTID transactions:

1. **On each server, execute:**
   ```sql
   SET @@GLOBAL.ENFORCE_GTID_CONSISTENCY = WARN;
   ```
   Let the server run for a while with your normal workload and monitor the logs. If this step causes any warnings in the log, adjust your application so that it only uses GTID-compatible features and does not generate any warnings.

   **Important**

   This is the first important step. You must ensure that no warnings are being generated in the error logs before going to the next step.

2. **On each server, execute:**
   ```sql
   SET @@GLOBAL.ENFORCE_GTID_CONSISTENCY = ON;
   ```

3. **On each server, execute:**
   ```sql
   SET @@GLOBAL.GTID_MODE = OFF_PERMISSIVE;
   ```
   It does not matter which server executes this statement first, but it is important that all servers complete this step before any server begins the next step.

4. **On each server, execute:**
Disabling GTID Transactions Online

SET @@GLOBAL.GTID_MODE = ON_PERMISSIVE;

It does not matter which server executes this statement first.

5. On each server, wait until the status variable ONGOING_ANONYMOUS_TRANSACTION_COUNT is zero. This can be checked using:

SHOW STATUS LIKE 'ONGOING_ANONYMOUS_TRANSACTION_COUNT';

Note
On a replication slave, it is theoretically possible that this shows zero and then nonzero again. This is not a problem, it suffices that it shows zero once.

6. Wait for all transactions generated up to step 5 to replicate to all servers. You can do this without stopping updates: the only important thing is that all anonymous transactions get replicated.

See Section 2.5.4, “Verifying Replication of Anonymous Transactions” for one method of checking that all anonymous transactions have replicated to all servers.

7. If you use binary logs for anything other than replication, for example point in time backup and restore, wait until you do not need the old binary logs having transactions without GTIDs.

For instance, after step 6 has completed, you can execute FLUSH LOGS on the server where you are taking backups. Then either explicitly take a backup or wait for the next iteration of any periodic backup routine you may have set up.

Ideally, wait for the server to purge all binary logs that existed when step 6 was completed. Also wait for any backup taken before step 6 to expire.

Important
This is the second important point. It is vital to understand that binary logs containing anonymous transactions, without GTIDs cannot be used after the next step. After this step, you must be sure that transactions without GTIDs do not exist anywhere in the topology.

8. On each server, execute:

SET @@GLOBAL.GTID_MODE = ON;

9. On each server, add gtid_mode=ON and enforce_gtid_consistency=ON to my.cnf.

You are now guaranteed that all transactions have a GTID (except transactions generated in step 5 or earlier, which have already been processed). To start using the GTID protocol so that you can later perform automatic fail-over, execute the following on each slave. Optionally, if you use multi-source replication, do this for each channel and include the FOR CHANNEL channel clause:

STOP SLAVE [FOR CHANNEL 'channel'];
CHANGE MASTER TO MASTER_AUTO_POSITION = 1 [FOR CHANNEL 'channel'];
START SLAVE [FOR CHANNEL 'channel'];

2.5.3 Disabling GTID Transactions Online

This section describes how to disable GTID transactions on servers that are already online. This procedure does not require taking the server offline and is suited to use in production. However, if you have the possibility to take the servers offline when disabling GTIDs mode that process is easier.

The process is similar to enabling GTID transactions while the server is online, but reversing the steps. The only thing that differs is the point at which you wait for logged transactions to replicate.

Before you start, ensure that the servers meet the following pre-conditions:
Verifying Replication of Anonymous Transactions

- **All** servers in your topology must use MySQL 5.7.6 or later. You cannot disable GTID transactions online on any single server unless all servers which are in the topology are using this version.

- **All** servers have `gtid_mode` set to **ON**.

1. Execute the following on each slave, and if you using multi-source replication, do it for each channel and include the `FOR CHANNEL` channel clause:

   ```sql
   STOP SLAVE [FOR CHANNEL 'channel'];
   CHANGE MASTER TO MASTER_AUTO_POSITION = 0, MASTER_LOG_FILE = file, \
   MASTER_LOG_POS = position [FOR CHANNEL 'channel'];
   START SLAVE [FOR CHANNEL 'channel'];
   ```

2. On each server, execute:

   ```sql
   SET @@GLOBAL.GTID_MODE = ON_PERMISSIVE;
   ```

3. On each server, execute:

   ```sql
   SET @@GLOBAL.GTID_MODE = OFF_PERMISSIVE;
   ```

4. On each server, wait until the variable `@@GLOBAL.GTID_OWNED` is equal to the empty string. This can be checked using:

   ```sql
   SELECT @@GLOBAL.GTID_OWNED;
   ```

   On a replication slave, it is theoretically possible that this is empty and then nonempty again. This is not a problem, it suffices that it is empty once.

5. Wait for all transactions that currently exist in any binary log to replicate to all slaves. See Section 2.5.4, “Verifying Replication of Anonymous Transactions” for one method of checking that all anonymous transactions have replicated to all servers.

6. If you use binary logs for anything else than replication, for example to do point in time backup or restore: wait until you do not need the old binary logs having GTID transactions.

   For instance, after step 5 has completed, you can execute `FLUSH LOGS` on the server where you are taking the backup. Then either explicitly take a backup or wait for the next iteration of any periodic backup routine you may have set up.

   Ideally, wait for the server to purge all binary logs that existed when step 5 was completed. Also wait for any backup taken before step 5 to expire.

   **Important**

   This is the one important point during this procedure. It is important to understand that logs containing GTID transactions cannot be used after the next step. Before proceeding you must be sure that GTID transactions do not exist anywhere in the topology.

7. On each server, execute:

   ```sql
   SET @@GLOBAL.GTID_MODE = OFF;
   ```

8. On each server, set `gtid_mode=OFF` in `my.cnf`.

   If you want to set `enforce_gtid_consistency=OFF`, you can do so now. After setting it, you should add `enforce_gtid_consistency=OFF` to your configuration file.

   If you want to downgrade to an earlier version of MySQL, you can do so now, using the normal downgrade procedure.

**2.5.4 Verifying Replication of Anonymous Transactions**
This section explains how to monitor a replication topology and verify that all anonymous transactions have been replicated. This is helpful when changing the replication mode online as you can verify that it is safe to change to GTID transactions.

There are several possible ways to wait for transactions to replicate:

The simplest method, which works regardless of your topology but relies on timing is as follows: if you are sure that the slave never lags more than N seconds, just wait for a bit more than N seconds. Or wait for a day, or whatever time period you consider safe for your deployment.

A safer method in the sense that it does not depend on timing: if you only have a master with one or more slaves, do the following:

1. On the master, execute:
   ```sql
   SHOW MASTER STATUS;
   ```
   Note down the values in the `File` and `Position` column.

2. On every slave, use the file and position information from the master to execute:
   ```sql
   SELECT MASTER_POS_WAIT(file, position);
   ```
   If you have a master and multiple levels of slaves, or in other words you have slaves of slaves, repeat step 2 on each level, starting from the master, then all the direct slaves, then all the slaves of slaves, and so on.

   If you use a circular replication topology where multiple servers may have write clients, perform step 2 for each master-slave connection, until you have completed the full circle. Repeat the whole process so that you do the full circle twice.

   For example, suppose you have three servers A, B, and C, replicating in a circle so that A -> B -> C -> A. The procedure is then:
   - Do step 1 on A and step 2 on B.
   - Do step 1 on B and step 2 on C.
   - Do step 1 on C and step 2 on A.
   - Do step 1 on A and step 2 on B.
   - Do step 1 on B and step 2 on C.
   - Do step 1 on C and step 2 on A.

2.6 Replication and Binary Logging Options and Variables

The following sections contain information about `mysqld` options and server variables that are used in replication and for controlling the binary log. Options and variables for use on replication masters and replication slaves are covered separately, as are options and variables relating to binary logging and global transaction identifiers (GTIDs). A set of quick-reference tables providing basic information about these options and variables is also included.

Of particular importance is the `server_id` system variable.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--server-id=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>server_id</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
</tbody>
</table>
Replication and Binary Logging Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

This variable specifies the server ID. In MySQL 5.7, `server_id` must be specified if binary logging is enabled, otherwise the server is not allowed to start.

`server_id` is set to 0 by default. On a replication master and each replication slave, you must specify `server_id` to establish a unique replication ID in the range from 1 to $2^{32} - 1$. “Unique”, means that each ID must be different from every other ID in use by any other replication master or slave. For additional information, see Section 2.6.2, “Replication Master Options and Variables”, and Section 2.6.3, “Replication Slave Options and Variables”.

If the server ID is set to 0, binary logging takes place, but a master with a server ID of 0 refuses any connections from slaves, and a slave with a server ID of 0 refuses to connect to a master. Note that although you can change the server ID dynamically to a nonzero value, doing so does not enable replication to start immediately. You must change the server ID and then restart the server to initialize the replication slave.

For more information, see Section 2.2.5.1, “Setting the Replication Slave Configuration”.

`server_uuid`

In MySQL 5.7, the server generates a true UUID in addition to the `server_id` value supplied by the user. This is available as the global, read-only `server_uuid` system variable.

```
Note
The presence of the `server_uuid` system variable in MySQL 5.7 does not change the requirement for setting a unique `server_id` value for each MySQL server as part of preparing and running MySQL replication, as described earlier in this section.
```

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>server_uuid</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

When starting, the MySQL server automatically obtains a UUID as follows:

1. Attempt to read and use the UUID written in the file `data_dir/auto.cnf` (where `data_dir` is the server's data directory).

2. If `data_dir/auto.cnf` is not found, generate a new UUID and save it to this file, creating the file if necessary.

The `auto.cnf` file has a format similar to that used for `my.cnf` or `my.ini` files. In MySQL 5.7, `auto.cnf` has only a single `[auto]` section containing a single `server_uuid` setting and value; the file's contents appear similar to what is shown here:

```
[auto]
server_uuid=8a94f357-aab4-11df-86ab-c80aa9429562
```
Replication and Binary Logging Option and Variable Reference

Important

The `auto.cnf` file is automatically generated; do not attempt to write or modify this file.

When using MySQL replication, masters and slaves know each other’s UUIDs. The value of a slave’s UUID can be seen in the output of `SHOW SLAVE HOSTS`. Once `START SLAVE` has been executed, the value of the master’s UUID is available on the slave in the output of `SHOW SLAVE STATUS`.

Note

Issuing a `STOP SLAVE` or `RESET SLAVE` statement does not reset the master’s UUID as used on the slave.

A server’s `server_uuid` is also used in GTIDs for transactions originating on that server. For more information, see Section 2.3, “Replication with Global Transaction Identifiers”.

When starting, the slave I/O thread generates an error and aborts if its master’s UUID is equal to its own unless the `--replicate-same-server-id` option has been set. In addition, the slave I/O thread generates a warning if either of the following is true:

- No master having the expected `server_uuid` exists.
- The master’s `server_uuid` has changed, although no `CHANGE MASTER TO` statement has ever been executed.

2.6.1 Replication and Binary Logging Option and Variable Reference

The following two sections provide basic information about the MySQL command-line options and system variables applicable to replication and the binary log.

Replication Options and Variables

The command-line options and system variables in the following list relate to replication masters and replication slaves. Section 2.6.2, “Replication Master Options and Variables”, provides more detailed information about options and variables relating to replication master servers. For more information about options and variables relating to replication slaves, see Section 2.6.3, “Replication Slave Options and Variables”.

- `abort-slave-event-count`: Option used by mysql-test for debugging and testing of replication.
- `auto_increment_increment`: AUTO_INCREMENT columns are incremented by this value.
- `auto_increment_offset`: Offset added to AUTO_INCREMENT columns.
- `binlog_gtid_simple_recovery`: Controls how binary logs are iterated during GTID recovery.
- `Com_change_master`: Count of CHANGE MASTER TO statements.
- `Com_show_master_status`: Count of SHOW MASTER STATUS statements.
- `Com_show_slave_hosts`: Count of SHOW SLAVE HOSTS statements.
- `Com_show_slave_status`: Count of SHOW SLAVE STATUS statements.
- `Com_slave_start`: Count of START SLAVE statements.
- `Com_slave_stop`: Count of STOP SLAVE statements.
- `disconnect-slave-event-count`: Option used by mysql-test for debugging and testing of replication.
- **enforce_gtid_consistency**: Prevents execution of statements that cannot be logged in a transactionally safe manner.

- **expire_logs_days**: Purge binary logs after this many days.

- **gtid_executed**: Global: All GTIDs in the binary log (global) or current transaction (session). Read-only.

- **gtid_executed_compression_period**: Compress gtid_executed table each time this many transactions have occurred. 0 means never compress this table. Applies only when binary logging is disabled.

- **gtid_mode**: Controls whether GTID based logging is enabled and what type of transactions the logs can contain.

- **gtid_next**: Specifies the GTID for the next statement to execute; see documentation for details.

- **gtid_owned**: The set of GTIDs owned by this client (session), or by all clients, together with the thread ID of the owner (global). Read-only.

- **gtid_purged**: The set of all GTIDs that have been purged from the binary log.

- **init_slave**: Statements that are executed when a slave connects to a master.

- **log_bin_trust_function_creators**: If equal to 0 (the default), then when --log-bin is used, creation of a stored function is allowed only to users having the SUPER privilege and only if the function created does not break binary logging.

- **log_builtin_as_identified_by_password**: Whether to log CREATE/ALTER USER, GRANT in backward-compatible fashion.

- **log_statements_unsafe_for_binlog**: Disables error 1592 warnings being written to the error log.

- **master-info-file**: The location and name of the file that remembers the master and where the I/O replication thread is in the master's binary logs.

- **master-retry-count**: Number of tries the slave makes to connect to the master before giving up.

- **master_info_repository**: Whether to write master status information and replication I/O thread location in the master's binary logs to a file or table.

- **max_relay_log_size**: If nonzero, relay log is rotated automatically when its size exceeds this value. If zero, size at which rotation occurs is determined by the value of max_binlog_size.

- **relay_log**: The location and base name to use for relay logs.

- **relay_log_basename**: Complete path to relay log, including file name.

- **relay_log_index**: The location and name to use for the file that keeps a list of the last relay logs.

- **relay_log_info_file**: File in which the slave records information about the relay logs.

- **relay_log_info_repository**: Whether to write the replication SQL thread's location in the relay logs to a file or a table.

- **relay_log_purge**: Determines whether relay logs are purged.

- **relay_log_recovery**: Whether automatic recovery of relay log files from master at startup is enabled; must be enabled for a crash-safe slave.

- **relay_log_space_limit**: Maximum space to use for all relay logs.
- **replicate-do-db**: Tells the slave SQL thread to restrict replication to the specified database.
- **replicate-do-table**: Tells the slave SQL thread to restrict replication to the specified table.
- **replicate-ignore-db**: Tells the slave SQL thread not to replicate to the specified database.
- **replicate-ignore-table**: Tells the slave SQL thread not to replicate to the specified table.
- **replicate-rewrite-db**: Updates to a database with a different name than the original.
- **replicate-same-server-id**: In replication, if enabled, do not skip events having our server id.
- **replicate-wild-do-table**: Tells the slave thread to restrict replication to the tables that match the specified wildcard pattern.
- **replicate-wild-ignore-table**: Tells the slave thread not to replicate to the tables that match the given wildcard pattern.
- **report_host**: Host name or IP of the slave to be reported to the master during slave registration.
- **report_password**: An arbitrary password that the slave server should report to the master. Not the same as the password for the MySQL replication user account.
- **report_port**: Port for connecting to slave reported to the master during slave registration.
- **report_user**: An arbitrary user name that a slave server should report to the master. Not the same as the name used with the MySQL replication user account.
- **Rpl_semi_sync_master_clients**: Number of semisynchronous slaves.
- **rpl_semi_sync_master_enabled**: Whether semisynchronous replication is enabled on the master.
- **Rpl_semi_sync_master_net_avg_wait_time**: The average time the master waited for a slave reply.
- **Rpl_semi_sync_master_net_wait_time**: The total time the master waited for slave replies.
- **Rpl_semi_sync_master_net_waits**: The total number of times the master waited for slave replies.
- **Rpl_semi_sync_master_no_times**: Number of times the master turned off semisynchronous replication.
- **Rpl_semi_sync_master_no_tx**: Number of commits not acknowledged successfully.
- **Rpl_semi_sync_master_status**: Whether semisynchronous replication is operational on the master.
- **Rpl_semi_sync_master_timefunc_failures**: Number of times the master failed when calling time functions.
- **rpl_semi_sync_master_timeout**: Number of milliseconds to wait for slave acknowledgment.
- **rpl_semi_sync_master_trace_level**: The semisynchronous replication debug trace level on the master.
- **Rpl_semi_sync_master_tx_avg_wait_time**: The average time the master waited for each transaction.
- **Rpl_semi_sync_master_tx_wait_time**: The total time the master waited for transactions.
- **Rpl_semi_sync_master_tx_waits**: The total number of times the master waited for transactions.
• **rpl_semi_sync_master_wait_for_slave_count**: How many slave acknowledgments the master must receive per transaction before proceeding.

• **rpl_semi_sync_master_wait_no_slave**: Whether master waits for timeout even with no slaves.

• **rpl_semi_sync_master_wait_point**: The wait point for slave transaction receipt acknowledgment.

• **Rpl_semi_sync_master_wait_pos_backtraverse**: The total number of times the master waited for an event with binary coordinates lower than events waited for previously.

• **Rpl_semi_sync_master_wait_sessions**: Number of sessions currently waiting for slave replies.

• **Rpl_semi_sync_master_yes_tx**: Number of commits acknowledged successfully.

• **rpl_semi_sync_slave_enabled**: Whether semisynchronous replication is enabled on slave.

• **Rpl_semi_sync_slave_status**: Whether semisynchronous replication is operational on slave.

• **rpl_semi_sync_slave_trace_level**: The semisynchronous replication debug trace level on the slave.

• **rpl_stop_slave_timeout**: Set the number of seconds that STOP SLAVE waits before timing out.

• **server_uuid**: The server's globally unique ID, automatically (re)generated at server start.

• **show-slave-auth-info**: Show user name and password in SHOW SLAVE HOSTS on this master.

• **skip-slave-start**: If set, slave is not autostarted.

• **slave_load_tmpdir**: The location where the slave should put its temporary files when replicating LOAD DATA statements.

• **slave_net_timeout**: Number of seconds to wait for more data from a master/slave connection before aborting the read.

• **slave_skip-errors**: Tells the slave thread to continue replication when a query returns an error from the provided list.

• **slave_checkpoint_group**: Maximum number of transactions processed by a multithreaded slave before a checkpoint operation is called to update progress status. Not supported by NDB Cluster.

• **slave_checkpoint_period**: Update progress status of multithreaded slave and flush relay log info to disk after this number of milliseconds. Not supported by NDB Cluster.

• **slave_compressed_protocol**: Use compression of master/slave protocol.

• **slave_exec_mode**: Allows for switching the slave thread between IDEMPOTENT mode (key and some other errors suppressed) and STRICT mode; STRICT mode is the default, except for NDB Cluster, where IDEMPOTENT is always used.

• **Slave_heartbeat_period**: The slave's replication heartbeat interval, in seconds.

• **Slave_last_heartbeat**: Shows when the latest heartbeat signal was received, in TIMESTAMP format.

• **slave_max_allowed_packet**: Maximum size, in bytes, of a packet that can be sent from a replication master to a slave; overrides max_allowed_packet.

• **Slave_open_temp_tables**: Number of temporary tables that the slave SQL thread currently has open.
• **slave_parallel_type**: Tells the slave to use timestamp information (LOGICAL_CLOCK) or database partitioning (DATABASE) to parallelize transactions.

• **slave_parallel_workers**: Number of applier threads for executing replication transactions in parallel. A value of 0 disables slave multithreading. Not supported by MySQL Cluster.

• **slave_pending_jobs_size_max**: Maximum size of slave worker queues holding events not yet applied.

• **slave_preserve_commit_order**: Ensures that all commits by slave workers happen in the same order as on the master to maintain consistency when using parallel applier threads.

• **Slave_received_heartbeats**: Number of heartbeats received by a replication slave since previous reset.

• **Slave_retried_transactions**: The total number of times since startup that the replication slave SQL thread has retried transactions.

• **slave_rows_search_algorithms**: Determines search algorithms used for slave update batching. Any 2 or 3 from the list INDEX_SEARCH, TABLE_SCAN, HASH_SCAN.

• **Slave_rows_last_search_algorithm_used**: Search algorithm most recently used by this slave to locate rows for row-based replication (index, table, or hash scan).

• **Slave_running**: The state of this server as a replication slave (slave I/O thread status).

• **slave_transaction_retries**: Number of times the slave SQL thread will retry a transaction in case it failed with a deadlock or elapsed lock wait timeout, before giving up and stopping.

• **slave_type_conversions**: Controls type conversion mode on replication slave. Value is a list of zero or more elements from the list: ALL_LOSSY, ALL_NON_LOSSY. Set to an empty string to disallow type conversions between master and slave.

• **sql_log_bin**: Controls binary logging for the current session.

• **sql_slave_skip_counter**: Number of events from the master that a slave server should skip. Not compatible with GTID replication.

• **sync_master_info**: Synchronize master.info to disk after every #th event.

• **sync_relay_log**: Synchronize relay log to disk after every #th event.

• **sync_relay_log_info**: Synchronize relay.info file to disk after every #th event.

• **transaction_write_set_extraction**: Defines the algorithm used to hash the writes extracted during a transaction.

For a listing of all command-line options, system and status variables used with `mysql`, see Server Option, System Variable, and Status Variable Reference.

### Binary Logging Options and Variables

The command-line options and system variables in the following list relate to the binary log, Section 2.6.4, “Binary Logging Options and Variables”, provides more detailed information about options and variables relating to binary logging. For additional general information about the binary log, see The Binary Log.

• **binlog-checksum**: Enable/disable binary log checksums.

• **binlog-do-db**: Limits binary logging to specific databases.

• **binlog-format**: Specifies the format of the binary log.
• **binlog-ignore-db**: Tells the master that updates to the given database should not be logged to the binary log.

• **binlog-row-event-max-size**: Binary log max event size.

• **Binlog_cache_disk_use**: Number of transactions that used a temporary file instead of the binary log cache.

• **binlog_cache_size**: Size of the cache to hold the SQL statements for the binary log during a transaction.

• **Binlog_cache_use**: Number of transactions that used the temporary binary log cache.

• **binlog_checksum**: Enable/disable binary log checksums.

• **binlog_direct_non_transactional_updates**: Causes updates using statement format to nontransactional engines to be written directly to binary log. See documentation before using.

• **binlog_error_action**: Controls what happens when the server cannot write to the binary log.

• **binlog_group_commit_sync_delay**: Sets the number of microseconds to wait before synchronizing transactions to disk.

• **binlog_group_commit_sync_no_delay_count**: Sets the maximum number of transactions to wait for before aborting the current delay specified by binlog_group_commit_sync_delay.

• **binlog_max_flush_queue_time**: How long to read transactions before flushing to binary log.

• **binlog_order_commits**: Whether to commit in same order as writes to binary log.

• **binlog_row_image**: Use full or minimal images when logging row changes.

• **binlog_rows_query_log_events**: When enabled, enables logging of rows query log events when using row-based logging. Disabled by default. Do not enable when producing logs for pre-5.6 slaves/readers.

• **Binlog_stmt_cache_disk_use**: Number of nontransactional statements that used a temporary file instead of the binary log statement cache.

• **binlog_stmt_cache_size**: Size of the cache to hold nontransactional statements for the binary log during a transaction.

• **Binlog_stmt_cache_use**: Number of statements that used the temporary binary log statement cache.

• **binlog_transaction_dependency_tracking**: Source of dependency information (commit timestamps or transaction write sets) from which to assess which transactions can be executed in parallel by slave's multithreaded applier.

• **binlog_transaction_dependency_history_size**: Number of row hashes kept for looking up transaction that last updated some row.

• **Com_show_binlog_events**: Count of SHOW BINLOG EVENTS statements.

• **Com_show_binlogs**: Count of SHOW BINLOGS statements.

• **log-bin**: Specifies the base name for binary log files.

• **log-bin-index**: Specifies the name for the binary log index file.

• **log_bin**: Whether the binary log is enabled.

• **log_bin_basename**: Path and base name for binary log files.
Replication Master Options and Variables

- `log_bin_use_v1_row_events`: Whether server is using version 1 binary log row events.
- `log_slave_updates`: Whether the slave should log the updates performed by its SQL thread to its own binary log.
- `master_verify_checksum`: Cause master to examine checksums when reading from the binary log.
- `max_binlog_dump_events`: Option used by mysql-test for debugging and testing of replication.
- `max_binlog_cache_size`: Can be used to restrict the total size used to cache a multi-statement transaction.
- `max_binlog_size`: Binary log will be rotated automatically when size exceeds this value.
- `max_binlog_stmt_cache_size`: Can be used to restrict the total size used to cache all nontransactional statements during a transaction.
- `slave_sql_verify_checksum`: Cause slave to examine checksums when reading from the relay log.
- `slave_sql_verify_checksum`: Cause slave to examine checksums when reading from relay log.
- `sporadic-binlog-dump-fail`: Option used by mysql-test for debugging and testing of replication.
- `sync_binlog`: Synchronously flush binary log to disk after every #th event.

For a listing of all command-line options, system and status variables used with `mysqld`, see Server Option, System Variable, and Status Variable Reference.

### 2.6.2 Replication Master Options and Variables

This section describes the server options and system variables that you can use on replication master servers. You can specify the options either on the command line or in an option file. You can specify system variable values using `SET`.

On the master and each slave, you must set the `server_id` system variable to establish a unique replication ID. For each server, you should pick a unique positive integer in the range from 1 to $2^{32} - 1$, and each ID must be different from every other ID in use by any other replication master or slave. Example: `server-id=3`.

For options used on the master for controlling binary logging, see Section 2.6.4, "Binary Logging Options and Variables".

**Startup Options for Replication Masters**

The following list describes startup options for controlling replication master servers. Replication-related system variables are discussed later in this section.

- `--show-slave-auth-info`
System Variables Used on Replication Masters

The following system variables are used to control replication masters:

- **auto_increment_increment**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--auto-increment-increment=##</td>
</tr>
<tr>
<td>System Variable</td>
<td>auto_increment_increment</td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>65535</td>
</tr>
</tbody>
</table>

`auto_increment_increment` and `auto_increment_offset` are intended for use with master-to-master replication, and can be used to control the operation of `AUTO_INCREMENT` columns. Both variables have global and session values, and each can assume an integer value between 1 and 65,535 inclusive. Setting the value of either of these two variables to 0 causes its value to be set to 1 instead. Attempting to set the value of either of these two variables to an integer greater than 65,535 or less than 0 causes its value to be set to 65,535 instead. Attempting to set the value of `auto_increment_increment` or `auto_increment_offset` to a noninteger value produces an error, and the actual value of the variable remains unchanged.

**Note**

`auto_increment_increment` is also supported for use with NDB tables.

When Group Replication is started on a server, the value of `auto_increment_increment` is changed to the value of `group_replication_auto_increment_increment`, which defaults to 7, and the value of `auto_increment_offset` is changed to the server ID. The changes are reverted when Group Replication is stopped. These changes are only made and reverted if `auto_increment_increment` and `auto_increment_offset` each have their default value of 1. If their values have already been modified from the default, Group Replication does not alter them.

`auto_increment_increment` and `auto_increment_offset` affect `AUTO_INCREMENT` column behavior as follows:

- **auto_increment_increment** controls the interval between successive column values. For example:

```
mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
|--------------------------+-------+
| auto_increment_increment | 1     |
| auto_increment_offset    | 1     |
+--------------------------+-------+
2 rows in set (0.00 sec)

mysql> CREATE TABLE autoinc1
  -> (col INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
Query OK, 0 rows affected (0.04 sec)

mysql> SET @@auto_increment_increment=10;
Query OK, 0 rows affected (0.00 sec)

mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
|--------------------------+-------+
| auto_increment_increment | 10    |
| auto_increment_offset    | 1     |
+--------------------------+-------+
2 rows in set (0.00 sec)
```
Replication Master Options and Variables

<table>
<thead>
<tr>
<th>Variable_name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>auto_increment_increment</td>
<td>10</td>
</tr>
<tr>
<td>auto_increment_offset</td>
<td>1</td>
</tr>
</tbody>
</table>

2 rows in set (0.01 sec)

```sql
mysql> INSERT INTO autoinc1 VALUES (NULL), (NULL), (NULL), (NULL);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```sql
mysql> SELECT col FROM autoinc1;
+-----+
<table>
<thead>
<tr>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>31</td>
</tr>
</tbody>
</table>
+-----+
```

4 rows in set (0.00 sec)

- `auto_increment_offset` determines the starting point for the AUTO_INCREMENT column value. Consider the following, assuming that these statements are executed during the same session as the example given in the description for `auto_increment_increment`:

```sql
mysql> SET @@auto_increment_offset=5;
Query OK, 0 rows affected (0.00 sec)
```

```sql
mysql> SHOW VARIABLES LIKE 'auto_in%';
+--------------------------+-------+
| Variable_name            | Value |
|--------------------------+-------+
| auto_increment_increment | 10    |
| auto_increment_offset    | 5     |
+--------------------------+-------+
```

mysql> CREATE TABLE autoinc2
-> (col INT NOT NULL AUTO_INCREMENT PRIMARY KEY);
```
Query OK, 0 rows affected (0.06 sec)
```

mysql> INSERT INTO autoinc2 VALUES (NULL), (NULL), (NULL), (NULL);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```sql
mysql> SELECT col FROM autoinc2;
+-----+
<table>
<thead>
<tr>
<th>col</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>35</td>
</tr>
</tbody>
</table>
+-----+
```

4 rows in set (0.02 sec)

When the value of `auto_increment_offset` is greater than that of `auto_increment_increment`, the value of `auto_increment_offset` is ignored.

If either of these variables is changed, and then new rows inserted into a table containing an AUTO_INCREMENT column, the results may seem counterintuitive because the series of AUTO_INCREMENT values is calculated without regard to any values already present in the column,
and the next value inserted is the least value in the series that is greater than the maximum existing value in the `AUTO_INCREMENT` column. The series is calculated like this:

\[ \text{auto_increment_offset} + N \times \text{auto_increment_increment} \]

where \( N \) is a positive integer value in the series \([1, 2, 3, ...]\). For example:

```sql
mysql> SHOW VARIABLES LIKE 'auto_inc%';
+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| auto_increment_increment | 10    |
| auto_increment_offset    | 5     |
+--------------------------+-------+
2 rows in set (0.00 sec)
```

```sql
mysql> SELECT col FROM autoinc1;
+-----+
| col |
+-----+
| 1   |
| 11  |
| 21  |
| 31  |
+-----+
4 rows in set (0.00 sec)
```

```sql
mysql> INSERT INTO autoinc1 VALUES (NULL), (NULL), (NULL), (NULL);
Query OK, 4 rows affected (0.00 sec)
Records: 4  Duplicates: 0  Warnings: 0
```

```sql
mysql> SELECT col FROM autoinc1;
+-----+
| col |
+-----+
| 1   |
| 11  |
| 21  |
| 31  |
| 35  |
| 45  |
| 55  |
| 65  |
+-----+
8 rows in set (0.00 sec)
```

The values shown for `auto_increment_increment` and `auto_increment_offset` generate the series \( 5 + N \times 10 \), that is, \([5, 15, 25, 35, 45, ...]\). The highest value present in the `col` column prior to the `INSERT` is 31, and the next available value in the `AUTO_INCREMENT` series is 35, so the inserted values for `col` begin at that point and the results are as shown for the `SELECT` query.

It is not possible to restrict the effects of these two variables to a single table; these variables control the behavior of all `AUTO_INCREMENT` columns in all tables on the MySQL server. If the global value of either variable is set, its effects persist until the global value is changed or overridden by setting the session value, or until `mysqld` is restarted. If the local value is set, the new value affects `AUTO_INCREMENT` columns for all tables into which new rows are inserted by the current user for the duration of the session, unless the values are changed during that session.

The default value of `auto_increment_increment` is 1. See Section 4.1.1, “Replication and `AUTO_INCREMENT`”.

- `auto_increment_offset`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--auto-increment-offset=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>auto_increment_offset</code></td>
</tr>
</tbody>
</table>
Replication Master Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>65535</td>
</tr>
</tbody>
</table>

This variable has a default value of 1. If it is left with its default value, and Group Replication is started on the server, it is changed to the server ID. For more information, see the description for `auto_increment_increment`.

**Note**

`auto_increment_offset` is also supported for use with NDB tables.

- **rpl_semi_sync_master_enabled**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--rpl-semi-sync-master-enabled[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td>rpl_semi_sync_master_enabled</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Controls whether semisynchronous replication is enabled on the master. To enable or disable the plugin, set this variable to **ON** or **OFF** (or 1 or 0), respectively. The default is **OFF**.

This variable is available only if the master-side semisynchronous replication plugin is installed.

- **rpl_semi_sync_master_timeout**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--rpl-semi-sync-master-timeout=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td>rpl_semi_sync_master_timeout</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>10000</td>
</tr>
</tbody>
</table>

A value in milliseconds that controls how long the master waits on a commit for acknowledgment from a slave before timing out and reverting to asynchronous replication. The default value is 10000 (10 seconds).

This variable is available only if the master-side semisynchronous replication plugin is installed.

- **rpl_semi_sync_master_trace_level**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--rpl-semi-sync-master-trace-level=#</code></td>
</tr>
</tbody>
</table>
Replication Master Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>rpl_semi_sync_master_trace_level</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>32</td>
</tr>
</tbody>
</table>

The semisynchronous replication debug trace level on the master. Four levels are defined:

- \(1\) = general level (for example, time function failures)
- \(16\) = detail level (more verbose information)
- \(32\) = net wait level (more information about network waits)
- \(64\) = function level (information about function entry and exit)

This variable is available only if the master-side semisynchronous replication plugin is installed.

- **rpl_semi_sync_master_wait_for_slave_count**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--rpl-semi-sync-master-wait-for-slave-count=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>rpl_semi_sync_master_wait_for_slave_count</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>65535</td>
</tr>
</tbody>
</table>

The number of slave acknowledgments the master must receive per transaction before proceeding. By default `rpl_semi_sync_master_wait_for_slave_count` is 1, meaning that semisynchronous replication proceeds after receiving a single slave acknowledgment. Performance is best for small values of this variable.

For example, if `rpl_semi_sync_master_wait_for_slave_count` is 2, then 2 slaves must acknowledge receipt of the transaction before the timeout period configured by `rpl_semi_sync_master_timeout` for semisynchronous replication to proceed. If less slaves acknowledge receipt of the transaction during the timeout period, the master reverts to normal replication.

**Note**

This behavior also depends on `rpl_semi_sync_master_wait_no_slave`

This variable is available only if the master-side semisynchronous replication plugin is installed.

- **rpl_semi_sync_master_wait_no_slave**
### Replication Master Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td>ON</td>
</tr>
</tbody>
</table>

Controls whether the master waits for the timeout period configured by `rpl_semi_sync_master_timeout` to expire, even if the slave count drops to less than the number of slaves configured by `rpl_semi_sync_master_wait_for_slave_count` during the timeout period.

When the value of `rpl_semi_sync_master_wait_no_slave` is **ON** (the default), it is permissible for the slave count to drop to less than `rpl_semi_sync_master_wait_for_slave_count` during the timeout period. As long as enough slaves acknowledge the transaction before the timeout period expires, semisynchronous replication continues.

When the value of `rpl_semi_sync_master_wait_no_slave` is **OFF**, if the slave count drops to less than the number configured in `rpl_semi_sync_master_wait_for_slave_count` at any time during the timeout period configured by `rpl_semi_sync_master_timeout`, the master reverts to normal replication.

This variable is available only if the master-side semisynchronous replication plugin is installed.

- `rpl_semi_sync_master_wait_point`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td><code>--rpl-semi-sync-master-wait-point=value</code></td>
</tr>
<tr>
<td><strong>System Variable</strong></td>
<td><code>rpl_semi_sync_master_wait_point</code></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Enumeration</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td><strong>AFTER_SYNC</strong></td>
</tr>
<tr>
<td><strong>Valid Values</strong></td>
<td><strong>AFTER_SYNC</strong></td>
</tr>
<tr>
<td></td>
<td><strong>AFTER_COMMIT</strong></td>
</tr>
</tbody>
</table>

This variable controls the point at which a semisynchronous replication master waits for slave acknowledgment of transaction receipt before returning a status to the client that committed the transaction. These values are permitted:

- **AFTER_SYNC** (the default): The master writes each transaction to its binary log and the slave, and syncs the binary log to disk. The master waits for slave acknowledgment of transaction receipt after the sync. Upon receiving acknowledgment, the master commits the transaction to the storage engine and returns a result to the client, which then can proceed.

- **AFTER_COMMIT**: The master writes each transaction to its binary log and the slave, syncs the binary log, and commits the transaction to the storage engine. The master waits for slave
acknowledgment of transaction receipt after the commit. Upon receiving acknowledgment, the master 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: After it has been acknowledged by the slave and committed to the storage engine on the master. Thus, all clients see the same data on the master.

  In the event of master failure, all transactions committed on the master have been replicated to the slave (saved to its relay log). A crash of the master and failover to the slave is lossless because the slave is up to date. Note, however, that the master cannot be restarted in this scenario and must be discarded, because its binary log might contain uncommitted transactions that would cause a conflict with the slave when externalized after binary log recovery.

- **With `AFTER_COMMIT`,** the client issuing the transaction gets a return status only after the server commits to the storage engine and receives slave acknowledgment. After the commit and before slave acknowledgment, other clients can see the committed transaction before the committing client.

  If something goes wrong such that the slave does not process the transaction, then in the event of a master crash and failover to the slave, it is possible that such clients will see a loss of data relative to what they saw on the master.

This variable is available only if the master-side semisynchronous replication plugin is installed.

`rpl_semi_sync_master_wait_point` was added in MySQL 5.7.2. For older versions, semisynchronous master behavior is equivalent to a setting of `AFTER_COMMIT`.

This change introduces a version compatibility constraint because it increments the semisynchronous interface version: Servers for MySQL 5.7.2 and up do not work with semisynchronous replication plugins from older versions, nor do servers from older versions work with semisynchronous replication plugins for MySQL 5.7.2 and up.

### 2.6.3 Replication Slave Options and Variables

This section explains the server options and system variables that apply to slave replication servers and contains the following:

- **Startup Options for Replication Slaves**
- **Options for Logging Slave Status to Tables**
- **System Variables Used on Replication Slaves**

Specify the options either on the command line or in an option file. Many of the options can be set while the server is running by using the `CHANGE MASTER TO` statement. Specify system variable values using `SET`.

**Server ID.** On the master and each slave, you must set the `server_id` system variable to establish a unique replication ID in the range from 1 to \(2^{32} - 1\). “Unique” means that each ID must be different from every other ID in use by any other replication master or slave. Example `my.cnf` file:

```ini
[mysql]
server-id=3
```

### Startup Options for Replication Slaves

This section explains startup options for controlling replication slave servers. Many of these options can be set while the server is running by using the `CHANGE MASTER TO` statement. Others, such as the `--replicate-*` options, can be set only when the slave server starts. Replication-related system variables are discussed later in this section.
Replication Slave Options and Variables

• **--log-warnings [=level]**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--log-warnings [=#]</td>
</tr>
<tr>
<td>Deprecated</td>
<td>Yes</td>
</tr>
<tr>
<td>System Variable</td>
<td>log_warnings</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>2</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

**Note**
The `log_error_verbosity` system variable is preferred over, and should be used instead of, the `--log-warnings` option or `log_warnings` system variable. For more information, see the descriptions of `log_error_verbosity` and `log_warnings`. The `--log-warnings` command-line option and `log_warnings` system variable are deprecated and will be removed in a future MySQL release.

Causes the server to record more messages to the error log about what it is doing. With respect to replication, the server generates warnings that it succeeded in reconnecting after a network or connection failure, and provides information about how each slave thread started. This variable is set to 2 by default. To disable it, set it to 0. The server logs messages about statements that are unsafe for statement-based logging if the value is greater than 0. Aborted connections and access-denied errors for new connection attempts are logged if the value is greater than 1. See Communication Errors and Aborted Connections.

**Note**
The effects of this option are not limited to replication. It affects diagnostic messages across a spectrum of server activities.

• **--master-info-file=file_name**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--master-info-file=file_name</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
<tr>
<td>Default Value</td>
<td>master.info</td>
</tr>
</tbody>
</table>

The name to use for the file in which the slave records information about the master. The default name is `master.info` in the data directory. For information about the format of this file, see Section 5.4.2, “Slave Status Logs”.

• **--master-retry-count=count**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--master-retry-count=#</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deprecated</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>86400</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>18446744073709551615 (64-bit platforms)</td>
</tr>
<tr>
<td></td>
<td>4294967295 (32-bit platforms)</td>
</tr>
</tbody>
</table>

The number of times that the slave tries to reconnect to the master before giving up. The default value is 86400 times. A value of 0 means “infinite”, and the slave attempts to connect forever. Reconnection attempts are triggered when the slave reaches its connection timeout (specified by the `slave_net_timeout` system variable) without receiving data or a heartbeat signal from the master. Reconnection is attempted at intervals set by the `MASTER_CONNECT_RETRY` option of the `CHANGE MASTER TO` statement (which defaults to every 60 seconds).

This option is deprecated and will be removed in a future MySQL release. Use the `MASTER_RETRY_COUNT` option of the `CHANGE MASTER TO` statement instead.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--max-relay-log-size=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>max_relay_log_size</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1073741824</td>
</tr>
</tbody>
</table>

The size at which the server rotates relay log files automatically. If this value is nonzero, the relay log is rotated automatically when its size exceeds this value. If this value is zero (the default), the size at which relay log rotation occurs is determined by the value of `max_binlog_size`. For more information, see Section 5.4.1, “The Slave Relay Log”.

- `--relay-log-purge={0|1}`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--relay-log-purge=[OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>relay_log_purge</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

Disable or enable automatic purging of relay logs as soon as they are no longer needed. The default value is 1 (enabled). This is a global variable that can be changed dynamically with `SET GLOBAL relay_log_purge = N`. Disabling purging of relay logs when enabling the `--relay-log-recovery` option puts data consistency at risk.
- **--relay-log-space-limit=size**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--relay-log-space-limit=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>relay_log_space_limit</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

This option places an upper limit on the total size in bytes of all relay logs on the slave. A value of 0 means “no limit”. This is useful for a slave server host that has limited disk space. When the limit is reached, the I/O thread stops reading binary log events from the master server until the SQL thread has caught up and deleted some unused relay logs. Note that this limit is not absolute: There are cases where the SQL thread needs more events before it can delete relay logs. In that case, the I/O thread exceeds the limit until it becomes possible for the SQL thread to delete some relay logs because not doing so would cause a deadlock. You should not set `--relay-log-space-limit` to less than twice the value of `--max-relay-log-size` (or `--max-binlog-size` if `--max-relay-log-size` is 0). In that case, there is a chance that the I/O thread waits for free space because `--relay-log-space-limit` is exceeded, but the SQL thread has no relay log to purge and is unable to satisfy the I/O thread. This forces the I/O thread to ignore `--relay-log-space-limit` temporarily.

- **--replicate-do-db=db_name**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--replicate-do-db=name</code></td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Creates a replication filter using the name of a database. Such filters can also be created using `CHANGE REPLICATION FILTER REPLICATE_DO_DB`. The precise effect of this filtering depends on whether statement-based or row-based replication is in use, and are described in the next several paragraphs.

**Note**

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.

**Statement-based replication.** Tell the slave SQL thread to restrict replication to statements 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...
replicate cross-database statements such as `UPDATE some_db.some_table SET foo='bar'` 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, if you supply a comma separated list then the list will be treated as the name of a single database.

An example of what does not work as you might expect when using statement-based replication: If the slave is started with `--replicate-do-db=sales` and you issue the following statements on the master, the `UPDATE` statement is not replicated:

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

The main reason for this “check just 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.

**Row-based replication.** Tells the slave SQL thread to restrict replication to database `db_name`. Only tables belonging to `db_name` are changed; the current database has no effect on this. Suppose that the slave is started with `--replicate-do-db=sales` and row-based replication is in effect, and then the following statements are run on the master:

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

The `february` table in the `sales` database on the slave is changed in accordance with the `UPDATE` statement; this occurs whether or not the `USE` statement was issued. However, issuing the following statements on the master has no effect on the slave when using row-based replication and `--replicate-do-db=sales`:

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

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

Another important difference in how `--replicate-do-db` is handled in statement-based replication as opposed to row-based replication occurs with regard to statements that refer to multiple databases. Suppose that the slave is started with `--replicate-do-db=db1`, and the following statements are executed on the master:

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

If you are using statement-based replication, then both tables are updated on the slave. However, when using row-based replication, only `table1` is affected on the slave; since `table2` is in a different database, `table2` on the slave is not changed by the `UPDATE`. Now suppose that, instead of the `USE db1` statement, a `USE db4` statement had been used:

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

In this case, the `UPDATE` statement would have no effect on the slave when using statement-based replication. However, if you are using row-based replication, the `UPDATE` would change `table1` on
the slave, but not \texttt{table2}—in other words, only tables in the database named by \texttt{--replicate-do-db} are changed, and the choice of default database has no effect on this behavior.

If you need cross-database updates to work, use \texttt{--replicate-wild-do-table=db_name.%} instead. See Section 5.5, “How Servers Evaluate Replication Filtering Rules”.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
Property & Value \\
\hline
Command-Line Format & \texttt{--replicate-ignore-db=name} \\
Type & String \\
\hline
\end{tabular}
\end{table}

\begin{itemize}
\item \texttt{--replicate-ignore-db=db_name}
\end{itemize}

Note
\begin{itemize}
\item This option affects replication in the same manner that \texttt{--binlog-do-db} affects binary logging, and the effects of the replication format on how \texttt{--replicate-do-db} affects replication behavior are the same as those of the logging format on the behavior of \texttt{--binlog-do-db}.
\item This option has no effect on \texttt{BEGIN}, \texttt{COMMIT}, or \texttt{ROLLBACK} statements.
\end{itemize}

\begin{itemize}
\item \texttt{--replicate-ignore-db}\texttt{=db_name}
\end{itemize}

\begin{itemize}
\item \texttt{--replicate-ignore-db}\texttt{=db_name}
\end{itemize}

Notes on filtering:

\begin{itemize}
\item 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.
\end{itemize}

\textbf{Statement-based replication.} Tells the slave SQL thread not to replicate any statement where the default database (that is, the one selected by \texttt{USE}) is \texttt{db_name}.

\textbf{Row-based replication.} Tells the slave SQL thread not to update any tables in the database \texttt{db_name}. The default database has no effect.

When using statement-based replication, the following example does not work as you might expect. Suppose that the slave is started with \texttt{--replicate-ignore-db=sales} and you issue the following statements on the master:

\begin{verbatim}
USE prices;
UPDATE sales.january SET amount=amount+1000;
\end{verbatim}

The \texttt{UPDATE} statement \textit{is} replicated in such a case because \texttt{--replicate-ignore-db} applies only to the default database (determined by the \texttt{USE} statement). Because the \texttt{sales} database was specified explicitly in the statement, the statement has not been filtered. However, when using row-based replication, the \texttt{UPDATE} statement's effects are \textit{not} propagated to the slave, and the slave's copy of the \texttt{sales.january} table is unchanged; in this instance, \texttt{--replicate-ignore-}
db=sales causes all changes made to tables in the master's copy of the sales database to be ignored by the slave.

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

You should not use this option if you are using cross-database updates and you do not want these updates to be replicated. See Section 5.5, “How Servers Evaluate Replication Filtering Rules”.

If you need cross-database updates to work, use --replicate-wild-ignore-table=db_name.

Note
This option affects replication in the same manner that --binlog-ignore-db affects binary logging, and the effects of the replication format on how --replicate-ignore-db affects replication behavior are the same as those of the logging format on the behavior of --binlog-ignore-db.

This option has no effect on BEGIN, COMMIT, or ROLLBACK statements.

• --replicate-do-table=db_name.tbl_name

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--replicate-do-table=name</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Creates a replication filter by telling the slave SQL thread to restrict replication to a given table. To specify more than one table, use this option multiple times, once for each table. This works for both cross-database updates and default database updates, in contrast to --replicate-do-db. See Section 5.5, “How Servers Evaluate Replication Filtering Rules”. You can also create such a filter by issuing a CHANGE REPLICATION FILTER REPLICATE_DO_TABLE statement.

Note
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.

This option affects only statements that apply to tables. It does not affect statements that apply only to other database objects, such as stored routines. To filter statements operating on stored routines, use one or more of the --replicate-*-db options.

• --replicate-ignore-table=db_name.tbl_name

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--replicate-ignore-table=name</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Creates a replication filter by telling the slave SQL thread not to replicate any statement that updates the specified table, even if any other tables might be updated by the same statement. To specify more than one table to ignore, use this option multiple times, once for each table. This works for cross-database updates, in contrast to --replicate-ignore-db. See Section 5.5, “How
Replication Slave Options and Variables

Servers Evaluate Replication Filtering Rules. You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE` statement.

Note

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.

This option affects only statements that apply to tables. It does not affect statements that apply only to other database objects, such as stored routines. To filter statements operating on stored routines, use one or more of the `--replicate-*db` options.

- `--replicate-rewrite-db=from_name->to_name`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--replicate-rewrite-db=old_name-&gt;new_name</code></td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Tells the slave to create a replication filter that translates the default database (that is, the one selected by `USE`) to `to_name` if it was `from_name` on the master. Only statements involving tables are affected (not statements such as `CREATE DATABASE`, `DROP DATABASE`, and `ALTER DATABASE`), and only if `from_name` is the default database on the master. To specify multiple rewrites, use this option multiple times. The server uses the first one with a `from_name` value that matches. The database name translation is done before the `--replicate-*` rules are tested. You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_REWRITE_DB` statement.

Note

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.

Statements in which table names are qualified with database names when using this option do not work with table-level replication filtering options such as `--replicate-do-table`. Suppose we have a database named `a` on the master, one named `b` on the slave, each containing a table `t`, and have started the master with `--replicate-rewrite-db='a->b'`. At a later point in time, we execute `DELETE FROM a.t`. In this case, no relevant filtering rule works, for the reasons shown here:

1. `--replicate-do-table=a.t` does not work because the slave has table `t` in database `b`.
2. `--replicate-do-table=b.t` does not match the original statement and so is ignored.
3. `--replicate-do-table=*t` is handled identically to `--replicate-do-table=a.t`, and thus does not work, either.

Similarly, the `--replication-rewrite-db` option does not work with cross-database updates.

If you use this option on the command line and the `>` character is special to your command interpreter, quote the option value. For example:

```
shell> mysqld --replicate-rewrite-db="olddb->newdb"
```
Replication Slave Options and Variables

• **--replicate-same-server-id**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--replicate-same-server-id</code>[={OFF</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

To be used on slave servers. Usually you should use the default setting of 0, to prevent infinite loops caused by circular replication. If set to 1, the slave does not skip events having its own server ID. Normally, this is useful only in rare configurations. Cannot be set to 1 if `log_slave_updates` is enabled. By default, the slave I/O thread does not write binary log events to the relay log if they have the slave’s server ID (this optimization helps save disk usage). If you want to use `--replicate-same-server-id`, be sure to start the slave with this option before you make the slave read its own events that you want the slave SQL thread to execute.

• **--replicate-wild-do-table=db_name.tbl_name**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--replicate-wild-do-table</code>[=name]</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Creates a replication filter by telling the slave thread to restrict replication to statements where any of the updated tables match the specified database and table name patterns. Patterns can contain the % and _ wildcard characters, which have the same meaning as for the LIKE pattern-matching operator. To specify more than one table, use this option multiple times, once for each table. This works for cross-database updates. See Section 5.5, “How Servers Evaluate Replication Filtering Rules”. You can also create such a filter by issuing a CHANGE REPLICATION FILTER REPlicate_WILD_DO_TABLE statement.

**Note**

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.

This option applies to tables, views, and triggers. It does not apply to stored procedures and functions, or events. To filter statements operating on the latter objects, use one or more of the **--replicate-*-db** options.

As an example, `--replicate-wild-do-table=foo%.bar%` replicates only updates that use a table where the database name starts with foo and the table name starts with bar.

If the table name pattern is %, it matches any table name and the option also applies to database-level statements (CREATE DATABASE, DROP DATABASE, and ALTER DATABASE). For example, if you use `--replicate-wild-do-table=foo\%.`, database-level statements are replicated if the database name matches the pattern foo%.

To include literal wildcard characters in the database or table name patterns, escape them with a backslash. For example, to replicate all tables of a database that is named my\_own\%db, but not replicate tables from the mylownAABCdb database, you should escape the _ and % characters like this: `--replicate-wild-do-table=\my\_own\%db`. If you use the option on the command line, you might need to double the backslashes or quote the option value, depending on your command interpreter. For example, with the bash shell, you would need to type `--replicate-wild-do-table=\\my\_own\\%db`. 
Replication Slave Options and Variables

- **--replicate-wild-ignore-table=db_name.tbl_name**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--replicate-wild-ignore-table=name</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

Creates a replication filter which keeps the slave thread from replicating a statement in which any table matches the given wildcard pattern. To specify more than one table to ignore, use this option multiple times, once for each table. This works for cross-database updates. See Section 5.5, "How Servers Evaluate Replication Filtering Rules". You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE` statement.

**Note**

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.

As an example, `--replicate-wild-ignore-table=foo%.bar%` does not replicate updates that use a table where the database name starts with `foo` and the table name starts with `bar`.

For information about how matching works, see the description of the `--replicate-wild-do-table` option. The rules for including literal wildcard characters in the option value are the same as for `--replicate-wild-ignore-table` as well.

- **--skip-slave-start**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--skip-slave-start[{OFF</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Tells the slave server not to start the slave threads when the server starts. To start the threads later, use a `START SLAVE` statement.

- **--slave-skip-errors=[err_code1, err_code2, ... | all | ddl_exist_errors]**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-skip-errors=name</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_skip_errors</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
<tr>
<td>Valid Values</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td>[list of error codes]</td>
</tr>
<tr>
<td></td>
<td>all</td>
</tr>
</tbody>
</table>

Note: 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.

As an example, `--replicate-wild-ignore-table=foo%.bar%` does not replicate updates that use a table where the database name starts with `foo` and the table name starts with `bar`.

For information about how matching works, see the description of the `--replicate-wild-do-table` option. The rules for including literal wildcard characters in the option value are the same as for `--replicate-wild-ignore-table` as well.
**Replication Slave Options and Variables**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>--slave-skip-errors=1062,1053</td>
<td></td>
</tr>
<tr>
<td>--slave-skip-errors=all</td>
<td></td>
</tr>
<tr>
<td>--slave-skip-errors=ddl_exist_errors</td>
<td></td>
</tr>
<tr>
<td>--slave-sql-verify-checksum=0</td>
<td></td>
</tr>
</tbody>
</table>

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

Do not use this option unless you fully understand why you are getting errors. If there are no bugs in your replication setup and client programs, and no bugs in MySQL itself, an error that stops replication should never occur. Indiscriminate use of this option results in slaves becoming hopelessly out of synchrony with the master, with you having no idea why this has occurred.

For error codes, you should use the numbers provided by the error message in your slave error log and in the output of `SHOW SLAVE STATUS`. Errors, Error Codes, and Common Problems, lists server error codes.

The shorthand value `ddl_exist_errors` is equivalent to the error code list `1007,1008,1050,1051,1054,1060,1061,1068,1094,1146`.

You can also (but should not) use the very nonrecommended value of `all` to cause the slave to ignore all error messages and keeps going regardless of what happens. Needless to say, if you use `all`, there are no guarantees regarding the integrity of your data. Please do not complain (or file bug reports) in this case if the slave's data is not anywhere close to what it is on the master. You have been warned.

Examples:

```
--slave-skip-errors=1062,1053
--slave-skip-errors=all
--slave-skip-errors=ddl_exist_errors
```

- `--slave-sql-verify-checksum={0|1}`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--slave-sql-verify-checksum={OFF</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

When this option is enabled, the slave examines checksums read from the relay log, in the event of a mismatch, the slave stops with an error.

The following options are used internally by the MySQL test suite for replication testing and debugging. They are not intended for use in a production setting.

- `--abort-slave-event-count`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--abort-slave-event-count=#</code></td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
</tbody>
</table>

When this option is set to some positive integer `value` other than 0 (the default) it affects replication behavior as follows: After the slave SQL thread has started, `value` log events are permitted to be executed; after that, the slave SQL thread does not receive any more events, just as if the network connection from the master were cut. The slave thread continues to run, and the output from `SHOW`
Replication Slave Options and Variables

**SLAVE STATUS** displays *Yes* in both the *Slave_IO_Running* and the *Slave_SQL_Running* columns, but no further events are read from the relay log.

- **--disconnect-slave-event-count**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--disconnect-slave-event-count=#</code></td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
</tbody>
</table>

**Options for Logging Slave Status to Tables**

MySQL 5.7 supports logging of replication slave status information to tables rather than files. Writing of the master info log and the relay log info log can be configured separately using these two system variables:

- **master_info_repository**
- **relay_log_info_repository**

For information about these variables, see Section 2.6.3, “Replication Slave Options and Variables”.

These variables can be used to make a replication slave resilient to unexpected halts. See Section 3.2, “Handling an Unexpected Halt of a Replication Slave”, for more information.

The info log tables and their contents are considered local to a given MySQL Server. They are not replicated, and changes to them are not written to the binary log.

For more information, see Section 5.4, “Replication Relay and Status Logs”.

**System Variables Used on Replication Slaves**

The following list describes system variables for controlling replication slave servers. They can be set at server startup and some of them can be changed at runtime using `SET`. Server options used with replication slaves are listed earlier in this section.

- **init_slave**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--init-slave=name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>init_slave</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

This variable is similar to `init_connect`, but is a string to be executed by a slave server each time the SQL thread starts. The format of the string is the same as for the `init_connect` variable. The setting of this variable takes effect for subsequent `START SLAVE` statements.

**Note**

The SQL thread sends an acknowledgment to the client before it executes `init_slave`. Therefore, it is not guaranteed that `init_slave` has been executed when `START SLAVE` returns. See `START SLAVE Statement`, for more information.

- **log_slow_slave_statements**
### Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td>`--log-slow-slave-statements=[{OFF</td>
</tr>
<tr>
<td><strong>System Variable</strong></td>
<td><code>log_slow_slave_statements</code></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Boolean</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td>OFF</td>
</tr>
</tbody>
</table>

When the slow query log is enabled, this variable enables logging for queries that have taken more than `long_query_time` seconds to execute on the slave. Note that if row-based replication is in use (`binlog_format=ROW`), `log_slow_slave_statements` has no effect. Queries are only added to the slave's slow query log when they are logged in statement format in the binary log, that is, when `binlog_format=STATEMENT` is set, or when `binlog_format=MIXED` is set and the statement is logged in statement format. Slow queries that are logged in row format when `binlog_format=MIXED` is set, or that are logged when `binlog_format=ROW` is set, are not added to the slave's slow query log, even if `log_slow_slave_statements` is enabled.

Setting `log_slow_slave_statements` has no immediate effect. The state of the variable applies on all subsequent `START SLAVE` statements. Also note that the global setting for `long_query_time` applies for the lifetime of the SQL thread. If you change that setting, you must stop and restart the slave's SQL thread to implement the change there (for example, by issuing `STOP SLAVE` and `START SLAVE` statements with the `SQL_THREAD` option).

#### master_info_repository

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td>`--master-info-repository={FILE</td>
</tr>
<tr>
<td><strong>System Variable</strong></td>
<td><code>master_info_repository</code></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>String</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td>FILE</td>
</tr>
<tr>
<td><strong>Valid Values</strong></td>
<td>FILE, TABLE</td>
</tr>
</tbody>
</table>

The setting of this variable determines whether the slave server logs master status and connection information to a `FILE (master.info)`, or to a `TABLE (mysql.slave_master_info)`. You can change the value of this variable only when no replication threads are executing.

The setting of this variable also has a direct influence on the effect had by the setting of the `sync_master_info` system variable; see that variable description for further information.

For `FILE` logging, you can change the name of the file using the `--master-info-file` server option.

This variable must be set to `TABLE` before configuring multiple replication channels. If you are using multiple replication channels, you cannot set the value back to `FILE`.

#### max_relay_log_size

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td><code>--max-relay-log-size=#</code></td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>max_relay_log_size</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1073741824</td>
</tr>
</tbody>
</table>

If a write by a replication slave to its relay log causes the current log file size to exceed the value of this variable, the slave rotates the relay logs (closes the current file and opens the next one). If `max_relay_log_size` is 0, the server uses `max_binlog_size` for both the binary log and the relay log. If `max_relay_log_size` is greater than 0, it constrains the size of the relay log, which enables you to have different sizes for the two logs. You must set `max_relay_log_size` to between 4096 bytes and 1GB (inclusive), or to 0. The default value is 0. See Section 5.2, “Replication Implementation Details”.

- `relay_log`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log=file_name</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
</tbody>
</table>

The base name for relay log files. For the default replication channel, the default base name for relay logs is `host_name-relay-bin`. For non-default replication channels, the default base name for relay logs is `host_name-relay-bin-channel`, where `channel` is the name of the replication channel recorded in this relay log.

The server writes the file in the data directory unless the base name is given with a leading absolute path name to specify a different directory. The server creates relay log files in sequence by adding a numeric suffix to the base name.

Due to the manner in which MySQL parses server options, if you specify this variable at server startup, you must supply a value; the default base name is used only if the option is not actually specified. If you specify the `relay_log` system variable at server startup without specifying a value, unexpected behavior is likely to result; this behavior depends on the other options used, the order in which they are specified, and whether they are specified on the command line or in an option file. For more information about how MySQL handles server options, see Specifying Program Options.

If you specify this variable, the value specified is also used as the base name for the relay log index file. You can override this behavior by specifying a different relay log index file base name using the `relay_log_index` system variable.

When the server reads an entry from the index file, 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
**relay_log** system variable. An absolute path remains unchanged; in such a case, the index must be edited manually to enable the new path or paths to be used.

You may find the **relay_log** system variable useful in performing the following tasks:

- Creating relay logs whose names are independent of host names.
- If you need to put the relay logs in some area other than the data directory because your relay logs tend to be very large and you do not want to decrease **max_relay_log_size**.
- To increase speed by using load-balancing between disks.

You can obtain the relay log file name (and path) from the **relay_log_basename** system variable.

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><strong>relay_log_basename</strong></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>datadir + '/' + hostname + '-relay-bin'</code></td>
</tr>
</tbody>
</table>

Holds the name and complete path to the relay log file. This variable is set by the server and is read only.

- **relay_log_index**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--relay-log-index=file_name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><strong>relay_log_index</strong></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>*host_name*'-relay-bin.index</code></td>
</tr>
</tbody>
</table>

The name for the relay log index file. For the default replication channel, the default name is `host_name-relay-bin.index`. For non-default replication channels, the default name is `host_name-relay-bin-channel.index`, where `channel` is the name of the replication channel recorded in this relay log index.

The server writes the file in the data directory unless the name is given with a leading absolute path name to specify a different directory. name.

Due to the manner in which MySQL parses server options, if you specify this variable at server startup, you must supply a value; the default base name is used only if the option is not actually specified. If you specify the **relay_log_index** system variable at server startup without specifying a value, unexpected behavior is likely to result; this behavior depends on the other options used, the order in which they are specified, and whether they are specified on the command line or in an option file. For more information about how MySQL handles server options, see [Specifying Program Options](#).
Replication Slave Options and Variables

- **relay_log_info_file**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log-info-file=file_name</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log_info_file</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
<tr>
<td>Default Value</td>
<td>relay-log.info</td>
</tr>
</tbody>
</table>

The name of the file in which the slave records information about the relay logs, when `relay_log_info_repository=FILE`. If `relay_log_info_repository=TABLE`, it is the file name that would be used in case the repository was changed to `FILE`. The default name is `relay-log.info` in the data directory.

- **relay_log_info_repository**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log-info-repository=value</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log_info_repository</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>Default Value</td>
<td>FILE</td>
</tr>
<tr>
<td>Valid Values</td>
<td>FILE, TABLE</td>
</tr>
</tbody>
</table>

This variable determines whether the slave's position in the relay logs is written to a `FILE` (`relay-log.info`) or to a `TABLE` (`mysql.slave_relay_log_info`). You can change the value of this variable only when no replication threads are executing.

The setting of this variable also has a direct influence on the effect had by the setting of the `sync_relay_log_info` system variable; see that variable description for further information.

This variable must be set to `TABLE` before configuring multiple replication channels. If you are using multiple replication channels then you cannot set the value back to `FILE`.

- **relay_log_purge**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log-purge=[OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log_purge</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

Disables or enables automatic purging of relay log files as soon as they are not needed any more. The default value is 1 (ON).

- **relay_log_recovery**
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log-recovery=[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log_recovery</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

If enabled, this variable enables automatic relay log recovery immediately following server startup. The recovery process creates a new relay log file, initializes the SQL thread position to this new relay log, and initializes the I/O thread to the SQL thread position. Reading of the relay log from the master then continues. This global variable is read-only at runtime. Its value can set with the `--relay-log-recovery` option at slave startup, which should be used following an unexpected halt of a replication slave to ensure that no possibly corrupted relay logs are processed. This option can be enabled to make a slave resilient to unexpected halts. See Section 3.2, “Handling an Unexpected Halt of a Replication Slave” for more information.

This variable also interacts with the `relay_log_purge` variable, which controls purging of logs when they are no longer needed. Enabling `relay_log_recovery` when `relay_log_purge` is disabled risks reading the relay log from files that were not purged, leading to data inconsistency.

When using a multithreaded slave (in other words `slave_parallel_workers` is greater than 0), inconsistencies such as gaps can occur in the sequence of transactions that have been executed from the relay log. Enabling `relay_log_recovery` when there are inconsistencies causes an error and the option has no effect. The solution in this situation is to issue `START SLAVE UNTIL SQL_AFTER_MTS_GAPS`, which brings the server to a more consistent state, then issue `RESET SLAVE` to remove the relay logs. See Section 4.1.32, “Replication and Transaction Inconsistencies” for more information.

**Note**

This variable does not affect the following Group Replication channels:

- `group_replication_applier`
- `group_replication_recovery`

Any other channels running on a group are affected, such as a channel which is replicating from an outside master or another group.

- `relay_log_space_limit`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--relay-log-space-limit=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>relay_log_space_limit</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>18446744073709551615 (64-bit platforms)</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

The maximum amount of space to use for all relay logs.

- **report_host**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--report-host=host_name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td>report_host</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

The host name or IP address of the slave to be reported to the master during slave registration. This value appears in the output of `SHOW SLAVE HOSTS` on the master server. Leave the value unset if you do not want the slave to register itself with the master.

**Note**

It is not sufficient for the master to simply read the IP address of the slave from the TCP/IP socket after the slave connects. Due to NAT and other routing issues, that IP may not be valid for connecting to the slave from the master or other hosts.

- **report_password**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--report-password=name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td>report_password</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

The account password of the slave to be reported to the master during slave registration. This value appears in the output of `SHOW SLAVE HOSTS` on the master server if the master was started with `--show-slave-auth-info`.

Although the name of this variable might imply otherwise, `report_password` is not connected to the MySQL user privilege system and so is not necessarily (or even likely to be) the same as the password for the MySQL replication user account.

- **report_port**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--report-port=port_num</code></td>
</tr>
<tr>
<td>System Variable</td>
<td>report_port</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
</tbody>
</table>

Default Value: `[slave_port]`
### Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>65535</td>
</tr>
</tbody>
</table>

The TCP/IP port number for connecting to the slave, to be reported to the master during slave registration. Set this only if the slave is listening on a nondefault port or if you have a special tunnel from the master or other clients to the slave. If you are not sure, do not use this option.

The default value for this option is the port number actually used by the slave. This is also the default value displayed by `SHOW SLAVE HOSTS`.

- **report_user**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--report-user=name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>report_user</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

The account user name of the slave to be reported to the master during slave registration. This value appears in the output of `SHOW SLAVE HOSTS` on the master server if the master was started with `--show-slave-auth-info`.

Although the name of this variable might imply otherwise, `report_user` is not connected to the MySQL user privilege system and so is not necessarily (or even likely to be) the same as the name of the MySQL replication user account.

- **rpl_semi_sync_slave_enabled**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--rpl-semi-sync-slave-enabled[={OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>rpl_semi_sync_slave_enabled</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Controls whether semisynchronous replication is enabled on the slave. To enable or disable the plugin, set this variable to `ON` or `OFF` (or 1 or 0), respectively. The default is `OFF`.

This variable is available only if the slave-side semisynchronous replication plugin is installed.

- **rpl_semi_sync_slave_trace_level**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--rpl-semi-sync-slave-trace-level=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>rpl_semi_sync_slave_trace_level</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>32</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

The semisynchronous replication debug trace level on the slave. See `rpl_semi_sync_master_trace_level` for the permissible values.

This variable is available only if the slave-side semisynchronous replication plugin is installed.

• `rpl_stop_slave_timeout`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--rpl-stop-slave-timeout=seconds</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>rpl_stop_slave_timeout</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>31536000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>2</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>31536000</td>
</tr>
</tbody>
</table>

You can control the length of time (in seconds) that `STOP SLAVE` waits before timing out by setting this variable. This can be used to avoid deadlocks between `STOP SLAVE` and other slave SQL statements using different client connections to the slave.

The maximum and default value of `rpl_stop_slave_timeout` is 31536000 seconds (1 year). The minimum is 2 seconds. Changes to this variable take effect for subsequent `STOP SLAVE` statements.

This variable affects only the client that issues a `STOP SLAVE` statement. When the timeout is reached, the issuing client returns an error message stating that the command execution is incomplete. The client then stops waiting for the slave threads to stop, but the slave threads continue to try to stop, and the `STOP SLAVE` instruction remains in effect. Once the slave threads are no longer busy, the `STOP SLAVE` statement is executed and the slave stops.

• `slave_checkpoint_group`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--slave-checkpoint-group=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_checkpoint_group</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>512</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>32</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>524280</td>
</tr>
<tr>
<td>Block Size</td>
<td>8</td>
</tr>
</tbody>
</table>

Sets the maximum number of transactions that can be processed by a multithreaded slave before a checkpoint operation is called to update its status as shown by `SHOW SLAVE STATUS`. Setting this
variable has no effect on slaves for which multithreading is not enabled. Setting this variable has no immediate effect. The state of the variable applies on all subsequent `START SLAVE` commands.

**Note**

Multithreaded slaves are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See [Known Issues in NDB Cluster Replication](#), for more information.

This variable works in combination with the `slave_checkpoint_period` system variable in such a way that, when either limit is exceeded, the checkpoint is executed and the counters tracking both the number of transactions and the time elapsed since the last checkpoint are reset.

The minimum allowed value for this variable is 32, unless the server was built using `-DWITH_DEBUG`, in which case the minimum value is 1. The effective value is always a multiple of 8; you can set it to a value that is not such a multiple, but the server rounds it down to the next lower multiple of 8 before storing the value. *(Exception: No such rounding is performed by the debug server.)* Regardless of how the server was built, the default value is 512, and the maximum allowed value is 524280.

- **slave_checkpoint_period**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--slave-checkpoint-period=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slaveCheckpoint_period</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>300</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>4G</td>
</tr>
</tbody>
</table>

Sets the maximum time (in milliseconds) that is allowed to pass before a checkpoint operation is called to update the status of a multithreaded slave as shown by `SHOW SLAVE STATUS`. Setting this variable has no effect on slaves for which multithreading is not enabled. Setting this variable takes effect for all replication channels immediately, including running channels.

**Note**

Multithreaded slaves are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See [Known Issues in NDB Cluster Replication](#), for more information.

This variable works in combination with the `slave_checkpoint_group` system variable in such a way that, when either limit is exceeded, the checkpoint is executed and the counters tracking both the number of transactions and the time elapsed since the last checkpoint are reset.

The minimum allowed value for this variable is 1, unless the server was built using `-DWITH_DEBUG`, in which case the minimum value is 0. Regardless of how the server was built, the default value is 300, and the maximum possible value is 4294967296 (4GB).

- **slave_compressed_protocol**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--slave-compressed-protocol=[OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_compressed_protocol</code></td>
</tr>
</tbody>
</table>
### Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Whether to use compression of the master/slave protocol if both master and slave 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 SLAVE` statement, as well as reconnections made by a running I/O thread (for example, after setting the `MASTER_RETRY_COUNT` option for the `CHANGE MASTER TO` statement). See also [Connection Compression Control](#).

- **slave_exec_mode**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--slave-exec-mode=mode</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_exec_mode</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
</tbody>
</table>
| Default Value           | IDEMPOTENT (NDB)  
                         | STRICT (Other)  |
| Valid Values            | IDEMPOTENT  
                         | STRICT        |

Controls how a slave 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-master replication, circular replication, and some other special replication scenarios for NDB Cluster Replication. (See NDB Cluster Replication: Multi-Master and Circular Replication, and 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 5.7, STRICT mode is the default value.

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-master replication or circular replication is employed, and is not recommended for use in other cases.

- **slave_load_tmpdir**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--slave-load-tmpdir=dir_name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_load_tmpdir</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Directory name</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>Value of --tmpdir</td>
</tr>
</tbody>
</table>

The name of the directory where the slave 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 slave 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 master is huge, the temporary files on the slave are huge, too. Therefore, it might be advisable to use this option to tell the slave 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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-max-allowed-packet=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_max_allowed_packet</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1073741824</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1073741824</td>
</tr>
</tbody>
</table>

This variable sets the maximum packet size for the slave SQL and I/O threads, so that large updates using row-based replication do not cause replication to fail because an update exceeded `max_allowed_packet`. Setting this variable takes effect for all replication channels immediately, including running channels.

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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-net-timeout=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_net_timeout</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>60</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
</tbody>
</table>

The number of seconds to wait for more data or a heartbeat signal from the master before the slave 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 SLAVE` commands.

The first retry occurs immediately after the timeout. The interval between retries is controlled by the `MASTER_CONNECT_RETRY` option for the `CHANGE MASTER TO` statement, and the number of reconnection attempts is limited by the `MASTER_RETRY_COUNT` option for the `CHANGE MASTER TO` statement.

The heartbeat interval, which stops the connection timeout occurring in the absence of data if the connection is still good, is controlled by the `MASTER_HEARTBEAT_PERIOD` option for the `CHANGE MASTER TO` statement. The heartbeat interval defaults to half the value of `slave_net_timeout`, and it is recorded in the master info log 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 MASTER TO` to adjust the heartbeat interval to an appropriate value so that it occurs before the connection timeout.

### `slave_parallel_type`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Form</td>
<td><code>--slave-parallel-type=value</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_parallel_type</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td>DATABASE</td>
</tr>
<tr>
<td>Valid Values</td>
<td>DATABASE, <code>LOGICAL_CLOCK</code></td>
</tr>
</tbody>
</table>

When using a multithreaded slave (`slave_parallel_workers` is greater than 0), this variable specifies the policy used to decide which transactions are allowed to execute in parallel on the slave. The variable has no effect on slaves 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 master are applied in parallel on a slave. 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 master 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 master. There must be no cross-database constraints, as such constraints may be violated on the slave.

When `slave_preserve_commit_order=1` is set, you can only use `LOGICAL_CLOCK`.

If your replication topology uses multiple levels of slaves, `LOGICAL_CLOCK` may achieve less parallelization for each level the slave is away from the master. You can reduce this effect by using
Replication Slave Options and Variables

**binlog_transaction_dependency_tracking** on the master to specify that write sets are used instead of timestamps for parallelization where possible.

- **slave_parallel_workers**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-parallel-workers=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_parallel_workers</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1024</td>
</tr>
</tbody>
</table>

Sets the number of slave applier threads for executing replication transactions in parallel. Setting this variable to a number greater than 0 creates a multithreaded slave with this number of applier threads. When set to 0 (the default) parallel execution is disabled and the slave uses a single applier thread. Setting `slave_parallel_workers` has no immediate effect. The state of the variable applies on all subsequent `START SLAVE` statements.

**Note**
Multithreaded slaves are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See Known Issues in NDB Cluster Replication, for more information.

A multithreaded slave provides parallel execution by using a coordinator thread and the number of applier threads configured by this variable. The way which transactions are distributed among applier threads is configured by `slave_parallel_type`. The transactions that the slave applies in parallel may commit out of order, unless `slave_preserve_commit_order=1`. Therefore, checking for the most recently executed transaction does not guarantee that all previous transactions from the master have been executed on the slave. This has implications for logging and recovery when using a multithreaded slave. For example, on a multithreaded slave the `START SLAVE UNTIL` statement only supports using `SQL_AFTER_MTS_GAPS`.

In MySQL 5.7, retrying of transactions is supported when multithreading is enabled on a slave. In previous versions, `slave_transaction_retries` was treated as equal to 0 when using multithreaded slaves.

- **slave_pending_jobs_size_max**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-pending-jobs-size-max=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_pending_jobs_size_max</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>16M</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1024</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>16EiB</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Block Size</td>
<td>1024</td>
</tr>
</tbody>
</table>

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

The minimum possible value for this variable is 1024; the default is 16MB. The maximum possible value is 18446744073709551615 (16 exabytes). Values that are not exact multiples of 1024 are rounded down to the next-highest multiple of 1024 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 slave workers have empty queues, and then processed. All subsequent transactions are held until the large transaction has been completed.

- `slave_preserve_commit_order`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--slave-presence-commit-order=[OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>slave_preserve_commit_order</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

For multithreaded slaves, the setting 1 for this variable ensures that transactions are externalized on the slave in the same order as they appear in the slave's relay log, and prevents gaps in the sequence of transactions that have been executed from the relay log. This variable has no effect on slaves for which multithreading is not enabled. Note that `slave_preserve_commit_order=1` does not preserve the order of non-transactional DML updates, so these might commit before transactions that precede them in the relay log, which might result in gaps.

`slave_preserve_commit_order=1` requires that `--log-bin` and `--log-slave-updates` are enabled on the slave, and `slave_parallel_type` is set to `LOGICAL_CLOCK`. Before changing this variable, all replication threads (for all replication channels if you are using multiple replication channels) must be stopped.

With `slave_preserve_commit_order` enabled, the executing thread waits until all previous transactions are committed before committing. While the slave thread is waiting for other workers to commit their transactions it reports its status as `Waiting for preceding transaction to commit`. (Prior to MySQL 5.7.8, this was shown as `Waiting for its turn to commit`.) Enabling this mode on a multithreaded slave ensures that it never enters a state that the master was not in. This supports the use of replication for read scale-out. See Section 3.4, "Using Replication for Scale-Out".

If `slave_preserve_commit_order=0` is set, the transactions that the slave 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 master have been executed on the slave. There is a chance of gaps in the sequence of transactions that have been executed from the slave's relay log. This has implications for logging and recovery when using a multithreaded slave. Note that the setting `slave_preserve_commit_order=1` prevents gaps, but does not prevent master log position lag (where `Exec_master_log_pos` is behind the position up to which transactions have been executed). See Section 4.1.32, "Replication and Transaction Inconsistencies" for more information.
• **slave_rows_search_algorithms**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-rows-search-algorithms=value</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_rows_search_algorithms</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Set</td>
</tr>
<tr>
<td>Default Value</td>
<td>TABLE_SCAN, INDEX_SCAN</td>
</tr>
<tr>
<td>Valid Values</td>
<td>INDEX_SCAN, HASH_SCAN</td>
</tr>
<tr>
<td></td>
<td>TABLE_SCAN, HASH_SCAN</td>
</tr>
<tr>
<td></td>
<td>TABLE_SCAN, INDEX_SCAN, HASH_SCAN (equivalent to INDEX_SCAN,HASH_SCAN)</td>
</tr>
</tbody>
</table>

When preparing batches of rows for row-based logging and replication, this variable controls how the rows are searched for matches, in particular whether hash scans are used. Setting this variable takes effect for all replication channels immediately, including running channels.

Specify a comma-separated list of the following combinations of 2 values from the list `INDEX_SCAN, TABLE_SCAN, HASH_SCAN`. The value is expected as a string, so if set at runtime rather than at server startup, the value must be quoted. In addition, the value must not contain any spaces. The recommended combinations (lists) and their effects are shown in the following table:

<table>
<thead>
<tr>
<th>Index used / option value</th>
<th>INDEX_SCAN, HASH_SCAN, INDEX_SCAN, TABLE_SCAN</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Primary key or unique key</strong></td>
<td>Index scan</td>
</tr>
<tr>
<td><strong>(Other) Key</strong></td>
<td>Hash scan over index</td>
</tr>
<tr>
<td><strong>No index</strong></td>
<td>Hash scan</td>
</tr>
</tbody>
</table>

• The default value is `INDEX_SCAN, TABLE_SCAN`, which means that all searches that can use indexes do use them, and searches without any indexes use table scans.

• To use hashing for any searches that do not use a primary or unique key, set `INDEX_SCAN, HASH_SCAN`. Specifying `INDEX_SCAN, HASH_SCAN` has the same effect as specifying `INDEX_SCAN, TABLE_SCAN, HASH_SCAN`, which is allowed.

• Do not use the combination `TABLE_SCAN, HASH_SCAN`. This setting forces hashing for all searches. It has no advantage over `INDEX_SCAN, HASH_SCAN`, and it can lead to “record not found” errors or duplicate key errors in the case of a single event containing multiple updates to the same row, or updates that are order-dependent.

The order in which the algorithms are specified in the list makes no difference to the order in which they are displayed by a `SELECT` or `SHOW VARIABLES` statement.

It is possible to specify a single value, but this is not optimal, because setting a single value limits searches to using only that algorithm. In particular, setting `INDEX_SCAN` alone is not recommended, as in that case searches are unable to find rows at all if no index is present.
• **slave_skip_errors**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-skip-errors=name</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_skip_errors</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
<tr>
<td>Valid Values</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td>[list of error codes]</td>
</tr>
<tr>
<td></td>
<td>all</td>
</tr>
<tr>
<td></td>
<td>ddl_exist_errors</td>
</tr>
</tbody>
</table>

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

• **slave_sql_verify_checksum**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-sql-verify-checksum={OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_sql_verify_checksum</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

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

**Note**

The slave I/O thread always reads checksums if possible when accepting events from over the network.

• **slave_transaction_retries**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-transaction-retries=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_transaction_retries</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>10</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Value (64-bit</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>platforms)</td>
<td></td>
</tr>
<tr>
<td>Maximum Value (32-bit</td>
<td>4294967295</td>
</tr>
<tr>
<td>platforms)</td>
<td></td>
</tr>
</tbody>
</table>

If a replication slave 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 default value for slave_transaction_retries is 10. Setting the variable to 0 disables automatic retrying of transactions. Setting the variable takes effect for all replication channels immediately, including running channels.

As of MySQL 5.7.5, retrying of transactions is supported when multithreading is enabled on a slave. In previous versions, slave_transaction_retries was treated as equal to 0 when using multithreaded slaves.

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.

• slave_type_conversions

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--slave-type-conversions=set</td>
</tr>
<tr>
<td>System Variable</td>
<td>slave_type_conversions</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Set</td>
</tr>
<tr>
<td>Default Value</td>
<td></td>
</tr>
<tr>
<td>Valid Values</td>
<td>ALL_LOSSY</td>
</tr>
<tr>
<td></td>
<td>ALL_NON_LOSSY</td>
</tr>
<tr>
<td></td>
<td>ALL_SIGNED</td>
</tr>
<tr>
<td></td>
<td>ALL_UNSIGNED</td>
</tr>
</tbody>
</table>

Controls the type conversion mode in effect on the slave when using row-based replication. In MySQL 5.7.2 and higher, 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 master and the slave. Setting this variable takes effect for all replication channels immediately, including running channels.

ALL_SIGNED and ALL_UNSIGNED were added in MySQL 5.7.2 (Bug#15831300). 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_slave_skip_counter

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>sql_slave_skip_counter</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
</tbody>
</table>
Replication Slave Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
</tbody>
</table>

The number of events from the master that a slave server should skip. Setting the option has no immediate effect. The variable applies to the next `START SLAVE` statement; the next `START SLAVE` 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 SLAVE` 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`. If you need to skip transactions when employing GTIDs, use `gtid_executed` from the master instead. See Section 2.7.3, “Skipping Transactions”.

### Important

If skipping the number of events specified by setting this variable would cause the slave to begin in the middle of an event group, the slave continues to skip until it finds the beginning of the next event group and begins from that point. For more information, see Section 2.7.3, “Skipping Transactions”.

- `sync_master_info`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--sync-master-info=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>sync_master_info</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>10000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

The effects of this variable on a replication slave depend on whether the slave's `master_info_repository` is set to `FILE` or `TABLE`, as explained in the following paragraphs.

**master_info_repository = FILE.** If the value of `sync_master_info` is greater than 0, the slave synchronizes its `master.info` file to disk (using `fdatasync()`) after every `sync_master_info` events. If it is 0, the MySQL server performs no synchronization of the `master.info` file to disk; instead, the server relies on the operating system to flush its contents periodically as with any other file.

**master_info_repository = TABLE.** If the value of `sync_master_info` is greater than 0, the slave updates its master info repository table after every `sync_master_info` events. If it is 0, the table is never updated.

The default value for `sync_master_info` is 10000. Setting this variable takes effect for all replication channels immediately, including running channels.
### Replication Slave Options and Variables

#### sync_relay_log

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--sync-relay-log=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>sync_relay_log</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>10000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

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 a crash 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).

#### sync_relay_log_info

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--sync-relay-log-info=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>sync_relay_log_info</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>10000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

The default value for `sync_relay_log_info` is 10000. Setting this variable takes effect for all replication channels immediately, including running channels.

The effects of this variable on the replication slave depend on the server's `relay_log_info_repository` setting (FILE or TABLE). If the setting is TABLE, the effects of the variable also depend on whether the storage engine used by the relay log info table is transactional (such as InnoDB) or not transactional (MyISAM). The effects of these factors on the behavior of the server for `sync_relay_log_info` values of zero and greater than zero are as follows:

- `sync_relay_log_info = 0` • If `relay_log_info_repository` is set to FILE, the MySQL server performs no synchronization of the relay-log.info...
file to disk; instead, the server relies on the operating system to flush its contents periodically as with any other file.

- If `relay_log_info_repository` is set to `TABLE`, and the storage engine for that table is transactional, the table is updated after each transaction. (The `sync_relay_log_info` setting is effectively ignored in this case.)

- If `relay_log_info_repository` is set to `TABLE`, and the storage engine for that table is not transactional, the table is never updated.

- If `relay_log_info_repository` is set to `FILE`, the slave synchronizes its `relay-log.info` file to disk (using `fdatasync()` after every N transactions.

- If `relay_log_info_repository` is set to `TABLE`, and the storage engine for that table is transactional, the table is updated after each transaction. (The `sync_relay_log_info` setting is effectively ignored in this case.)

- If `relay_log_info_repository` is set to `TABLE`, and the storage engine for that table is not transactional, the table is updated after every N events.

### 2.6.4 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 The Binary Log. For additional information about using MySQL server options and system variables, see Server Command Options, and 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`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-row-event-max-size=#</code></td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>8192</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>256</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

Specify the maximum size of a row-based binary log event, in bytes. Rows are grouped into events smaller than this size if possible. The value should be a multiple of 256. The default is 8192. See Section 5.1, “Replication Formats”.

91
Binary Logging Options and Variables

- **--log-bin[=base_name]**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--log-bin=file_name</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
</tbody>
</table>

Enables binary logging. 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. For information on the format and management of the binary log, see The Binary Log.

If you supply a value for the **--log-bin** option, the value is used as the base name for the log sequence. The server creates binary log files in sequence by adding a numeric suffix to the base name. In MySQL 5.7, the base name defaults to `host_name-bin`, using the name of the host machine. It is recommended that you specify a base name, so that you can continue to use the same binary log file names regardless of changes to the default name.

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. (In older versions of MySQL, manual intervention was required whenever relocating the binary log or relay log files.) (Bug #11745230, Bug #12133)

Setting this option causes the log_bin system variable to be set to ON (or 1), and not to the base name. The binary log file base name and any specified path are available as the log_bin_basename system variable.

If you specify the **--log-bin** option without also specifying the server_id system variable, the server is not allowed to start. (Bug #11763963, Bug #56739)

When GTIDs are in use on the server, if binary logging is not enabled 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.

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--log-bin-index=file_name</td>
</tr>
<tr>
<td>System Variable</td>
<td>log_bin_index</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
</tbody>
</table>

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 omit the file name and do not specify one with **--log-**
**Binary Logging Options and Variables**

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 [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 master server to its slaves. There are also options for slave servers that control which statements received from the master should be executed or ignored. For details, see [Section 2.6.3, “Replication Slave Options and Variables”](#).

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-do-db=name</code></td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

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 will be 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:

```sql
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`:

```sql
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:

```sql
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:

```sql
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:

```sql
USE db1;
UPDATE db1.table1 SET col1 = 10, db2.table2 SET 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:

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td><code>--binlog-ignore-db=name</code></td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>String</td>
</tr>
</tbody>
</table>

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...
Binary Logging Options and Variables

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`.

Prior to MySQL 5.7.2, this option caused any statements containing fully qualified table names not to
be logged if there was no default database specified (that is, when `SELECT DATABASE()` returned
`NULL`). In MySQL 5.7.2 and higher, 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:

```sql
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 master'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 will be 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}`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-checksum=type</code></td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
<tr>
<td>Default Value</td>
<td>CRC32</td>
</tr>
<tr>
<td>Valid Values</td>
<td>NONE</td>
</tr>
<tr>
<td></td>
<td>CRC32</td>
</tr>
</tbody>
</table>

Enabling this option causes the master 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 slave (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.
Binary Logging Options and Variables

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--max-binlog-dump-events=</code>#</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
</tbody>
</table>

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

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--sporadic-binlog-dump-fail[={OFF</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

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`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-cache-size=</code>#</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_cache_size</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>32768</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>4096</td>
</tr>
<tr>
<td>Maximum Value (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Maximum Value (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

The size of the cache to hold changes to the binary log during a transaction. A binary log cache is allocated for each client if the server supports any transactional storage engines and if the server has the binary log enabled (`--log-bin` option). If you often use large transactions, you can increase this cache size to get better performance. The `Binlog_cache_use` and `Binlog_cache_disk_use` status variables can be useful for tuning the size of this variable. See 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`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-checksum=name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_checksum</code></td>
</tr>
</tbody>
</table>
**Property** | **Value**
---|---
Scope | Global
Dynamic | Yes
Type | String
Default Value | CRC32
Valid Values | NONE, CRC32

When enabled, this variable causes the master to write a checksum for each event in the binary log. `binlog_checksum` supports the values NONE (disabled) and CRC32. The default is CRC32. You cannot change the value of `binlog_checksum` within a transaction.

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.

Changing the value of this variable causes the binary log to be rotated; checksums are always written to an entire binary log file, and never to only part of one.

Setting this variable on the master to a value unrecognized by the slave causes the slave to set its own `binlog_checksum` value to NONE, and to stop replication with an error. (Bug #13553750, Bug #61096) If backward compatibility with older slaves is a concern, you may want to set the value explicitly to NONE.

- `binlog_direct_non_transactional_updates`

**Property** | **Value**
---|---
Command-Line Format | `--binlog-direct-non-transactional-updates[={OFF|ON}]`
System Variable | `binlog_direct_non_transactional_updates`
Scope | Global, Session
Dynamic | Yes
Type | Boolean
Default Value | OFF

Due to concurrency issues, a slave 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.

`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
Binary Logging Options and Variables

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 slave to diverge from the master.

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

- **binlog_error_action**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-error-action[=value]</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_error_action</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>ABORT_SERVER</code></td>
</tr>
<tr>
<td>Valid Values</td>
<td><code>IGNORE_ERROR</code></td>
</tr>
</tbody>
</table>

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 master's binary log to become inconsistent and replication slaves to lose synchronization.

In MySQL 5.7.7 and higher, 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 3.2, “Handling an Unexpected Halt of a Replication Slave”).

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.

In previous releases this variable was named `binlogging_impossible_mode`.

- **binlog_format**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-format=format</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_format</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>ROW</code></td>
</tr>
<tr>
<td>Valid Values</td>
<td><code>ROW</code></td>
</tr>
</tbody>
</table>

| Valid Values      | `STATEMENT`                    |
This variable sets the binary logging format, and can be any one of `STATEMENT`, `ROW`, or `MIXED`. See Section 5.1, “Replication Formats”.

`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.

Prior to MySQL 5.7.7, the default format was `STATEMENT`. In MySQL 5.7.7 and higher, 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 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 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 user-defined functions (UDF) 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 Stored Program Binary Logging.

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

- From within a stored function or a trigger.
- If the session is currently in row-based replication mode and has open temporary tables.
- From within a transaction.

Trying to switch the format in those cases results in an error.

Changing the logging format on a replication master does not cause a replication slave to change its logging format to match. Switching the replication format while replication is ongoing can cause issues if a replication slave has binary logging enabled, and the change results in the slave using `STATEMENT` format logging while the master is using `ROW` or `MIXED` format logging. A replication slave 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 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.
### Binary Logging Options and Variables

- **binlog_group_commit_sync_delay**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-group-commit-sync-delay=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_group_commit_sync_delay</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1000000</td>
</tr>
</tbody>
</table>

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 replication slave, and therefore can increase parallel execution on the replication slaves. To benefit from this effect, the slave servers must have `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 master’s throughput and the slaves’ 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 (master or slave) 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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-group-commit-sync-no-delay-count=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_group_commit_sync_no_delay_count</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
</tbody>
</table>

100
### Binary Logging Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Value</td>
<td>1000000</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-max-flush-queue-time=#</code></td>
</tr>
<tr>
<td>Deprecated</td>
<td>Yes</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_max_flush_queue_time</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>100000</td>
</tr>
</tbody>
</table>

Formerly, this controlled the time in microseconds to continue reading transactions from the flush queue before proceeding with group commit. In MySQL 5.7, this variable no longer has any effect.

`binlog_max_flush_queue_time` is deprecated as of MySQL 5.7.9, and is marked for eventual removal in a future MySQL release.

- **binlog_order_commits**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--binlog-order-commits=[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_order_commits</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

When this variable is enabled on a replication master (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 master and on a multithreaded replication slave remains identical, set `slave_preserve_commit_order=1` on the replication slave.

- **`binlog_row_image`**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-row-image=image_type</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_row_image</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>full</code></td>
</tr>
<tr>
<td>Valid Values</td>
<td><code>full</code> (Log all columns)</td>
</tr>
<tr>
<td></td>
<td><code>minimal</code> (Log only changed columns, and columns needed to identify rows)</td>
</tr>
<tr>
<td></td>
<td><code>noblob</code> (Log all columns, except for unneeded BLOB and TEXT columns)</td>
</tr>
</tbody>
</table>

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

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 master and slave 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_rows_query_log_events**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--binlog-rows-query-log-events[={OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><strong>binlog_rows_query_log_events</strong></td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

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 master when it cannot be reconstructed from the row updates.

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 `mysqldbinlog`'s `--verbose` option twice, either as `--vv` or `--verbose --verbose`.

• **binlog_stmt_cache_size**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--binlog-stmt-cache-size=#</code></td>
</tr>
</tbody>
</table>
### Binary Logging Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>System Variable</strong></td>
<td><code>binlog_stmt_cache_size</code></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Integer</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td>32768</td>
</tr>
<tr>
<td><strong>Minimum Value</strong></td>
<td>4096</td>
</tr>
<tr>
<td><strong>Maximum Value</strong> (64-bit platforms)</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td><strong>Maximum Value</strong> (32-bit platforms)</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

This variable determines the size of the cache for the binary log to hold nontransactional statements issued during a transaction. Separate binary log transaction and statement caches are allocated for each client if the server supports any transactional storage engines and if the server has the binary log enabled (`--log-bin` option). If you often use large nontransactional statements during transactions, you can increase this cache size to get better performance. The `Binlog_stmt_cache_use` and `Binlog_stmt_cache_disk_use` status variables can be useful for tuning the size of this variable. See [The Binary Log](#).

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

- `binlog_transaction_dependency_tracking`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Command-Line Format</strong></td>
<td><code>--binlog-transaction-dependency-tracking=value</code></td>
</tr>
<tr>
<td><strong>Introduced</strong></td>
<td>5.7.22</td>
</tr>
<tr>
<td><strong>System Variable</strong></td>
<td><code>binlog_transaction_dependency_tracking</code></td>
</tr>
<tr>
<td><strong>Scope</strong></td>
<td>Global</td>
</tr>
<tr>
<td><strong>Dynamic</strong></td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Type</strong></td>
<td>Enumeration</td>
</tr>
<tr>
<td><strong>Default Value</strong></td>
<td><code>COMMIT_ORDER</code></td>
</tr>
<tr>
<td><strong>Valid Values</strong></td>
<td><code>COMMIT_ORDER</code>, <code>WRITESET</code>, <code>WRITESET_SESSION</code></td>
</tr>
</tbody>
</table>

The source of dependency information that the master uses to determine which transactions can be executed in parallel by the slave’s multithreaded applier. This variable can take one of the three values described in the following list:

- **COMMIT_ORDER**: Dependency information is generated from the master's commit timestamps. This is the default. This mode is also used for any transactions without write sets, even if this variable's is `WRITESET` or `WRITESET_SESSION`; this is also the case for transactions updating tables without primary keys and transactions updating tables having foreign key constraints.

- **WRITESET**: Dependency information is generated from the master's write set, and any transactions which write different tuples can be parallelized.
• **WRITESET_SESSION**: Dependency information is generated from the master's write set, but no two updates from the same session can be reordered.

**WRITESET** and **WRITESET_SESSION** modes do not deliver any transaction dependencies that are newer than those that would have been returned in **COMMIT_ORDER** mode.

The value of this variable cannot be set to anything other than **COMMIT_ORDER** if **transaction_write_set_extraction** is **OFF**. You should also note that the value of **transaction_write_set_extraction** cannot be changed if the current value of **binlog_transaction_dependency_tracking** is **WRITESET** or **WRITESET_SESSION**.

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**.

• **binlog_transaction_dependency_history_size**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--binlog-transaction-dependency-history-size=#</td>
</tr>
<tr>
<td>Introduced</td>
<td>5.7.22</td>
</tr>
<tr>
<td>System Variable</td>
<td>binlog_transaction_dependency_history_size</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>25000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>1</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>1000000</td>
</tr>
</tbody>
</table>

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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--expire-logs-days=#</td>
</tr>
<tr>
<td>System Variable</td>
<td>expire_logs_days</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>0</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>99</td>
</tr>
</tbody>
</table>

The number of days for automatic binary log file removal. The default is 0, which means “no automatic removal.” Possible removals happen at startup and when the binary log is flushed. Log flushing occurs as indicated in **MySQL Server Logs**.

To remove binary log files manually, use the **PURGE BINARY LOGS** statement. See **PURGE BINARY LOGS Statement**.
Binary Logging Options and Variables

- **log_bin**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>log_bin</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
</tbody>
</table>

Whether the binary log is enabled. If the `--log-bin` option is used, then the value of this variable is **ON**; otherwise it is **OFF**. This variable reports only on the status of binary logging (enabled or disabled); it does not actually report the value to which `--log-bin` is set.

See [The Binary Log](#).

- **log_bin_basename**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>log_bin_basename</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
</tbody>
</table>

Holds the base name and path for the binary log files, which can be set with the `--log-bin` server option. In MySQL 5.7, the default base name is the name of the host machine with the suffix `-bin`. The default location is the data directory.

- **log_bin_index**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--log-bin-index=file_name</code></td>
</tr>
<tr>
<td>System Variable</td>
<td>log_bin_index</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>File name</td>
</tr>
</tbody>
</table>

Holds the base name and path for the binary log index file, which can be set with the `--log-bin-index` server option.

- **log_bin_trust_function_creators**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--log-bin-trust-function-creators[={OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td>log_bin_trust_function_creators</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

This variable applies when binary logging is enabled. It controls whether stored function creators can be trusted not to create stored functions that will 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 Stored Program Binary Logging.

- **log_bin_use_v1_row_events**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--log-bin-use-v1-row-events[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>log_bin_use_v1_row_events</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Whether Version 2 binary logging is in use. If this variable is 0 (disabled, the default), Version 2 binary log events are in use. If this variable is 1 (enabled), the server writes the binary log using Version 1 logging events (the only version of binary log events used in previous releases), and thus produces a binary log that can be read by older slaves.

MySQL 5.7 uses Version 2 binary log row events by default. However, Version 2 events cannot be read by MySQL Server releases prior to MySQL 5.6.6. Enabling `log_bin_use_v1_row_events` causes `mysqld` to write the binary log using Version 1 logging events.

This variable is read-only at runtime. To switch between Version 1 and Version 2 binary event binary logging, it is necessary to set `log_bin_use_v1_row_events` at server startup.

Other than when performing upgrades of NDB Cluster Replication, `log_bin_use_v1_row_events` is chiefly of interest when setting up replication conflict detection and resolution using `NDDB $EPOCH_TRANS()` as the conflict detection function, which requires Version 2 binary log row events. Thus, this variable and `--ndb-log-transaction-id` are not compatible.

**Note**

MySQL NDB Cluster 7.5 uses Version 2 binary log events by default. You should keep this mind when planning upgrades or downgrades, and for setups using NDB Cluster Replication.

For more information, see NDB Cluster Replication Conflict Resolution.

- **log_builtin_as_identified_by_password**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--log-builtin-as-identified-by-password[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>log_builtin_as_identified_by_password</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

This variable affects binary logging of user-management statements. When enabled, the variable has the following effects:

- Binary logging for `CREATE USER` statements involving built-in authentication plugins rewrites the statements to include an `IDENTIFIED BY PASSWORD` clause.
• `SET PASSWORD` statements are logged as `SET PASSWORD` statements, rather than being rewritten to `ALTER USER` statements.

• `SET PASSWORD` statements are changed to log the hash of the password instead of the supplied cleartext (unencrypted) password.

Enabling this variable ensures better compatibility for cross-version replication with 5.6 and pre-5.7.6 slaves, and for applications that expect this syntax in the binary log.

• `log_slave_updates`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--log-slave-updates[{OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>log_slave_updates</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Whether updates received by a slave server from a master server should be logged to the slave's own binary log.

Normally, a slave does not log to its own binary log any updates that are received from a master server. Enabling this variable causes the slave to write the updates performed by its SQL thread to its own binary log. For this option to have any effect, the slave must also be started with the `--log-bin` option to enable binary logging. See Section 2.6, “Replication and Binary Logging Options and Variables”.

`log_slave_updates` is enabled when you want to chain replication servers. For example, you might want to set up replication servers using this arrangement:

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

Here, `A` serves as the master for the slave `B`, and `B` serves as the master for the slave `C`. For this to work, `B` must be both a master and a slave. You must start both `A` and `B` with `--log-bin` to enable binary logging, and `B` with `log_slave_updates` enabled so that updates received from `A` are logged by `B` to its binary log.

• `log_statements Unsafe for binlog`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--log-statements-unsafe-for-binlog[{OFF</td>
</tr>
<tr>
<td>Introduced</td>
<td>5.7.11</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>log_statements_unsafe_for_binlog</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--master-verify-checksum=[OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>master_verify_checksum</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
</tbody>
</table>

Enabling this variable causes the master 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 master 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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--max-binlog-cache-size=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>max_binlog_cache_size</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>18446744073709551615</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>4096</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>18446744073709551615</td>
</tr>
</tbody>
</table>

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 16EB (exabytes). The maximum recommended value is 4GB; this is due to the fact that MySQL currently cannot work with binary log positions greater than 4GB.

`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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--max-binlog-size=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>max_binlog_size</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1073741824</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>4096</td>
</tr>
</tbody>
</table>
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.

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.

- **max_binlog_stmt_cache_size**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Value</td>
<td>1073741824</td>
</tr>
</tbody>
</table>

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.

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.

- **sql_log_bin**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>sql_log_bin</td>
</tr>
<tr>
<td>Scope</td>
<td>Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
</tbody>
</table>
Binary Logging Options and Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

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 master you do not want replicated to the slave.

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 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.

The global `sql_log_bin` variable is read only and cannot be modified. The global scope is deprecated and will be removed in a future MySQL release.

- `sync_binlog`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>--sync-binlog=#</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>sync_binlog</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

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`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--transaction-write-set-extraction[=value]</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>transaction_write_set_extraction</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
<tr>
<td>Valid Values (≥ 5.7.14)</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td>MURMUR32</td>
</tr>
<tr>
<td></td>
<td>XXHASH64</td>
</tr>
<tr>
<td>Valid Values (≤ 5.7.13)</td>
<td>OFF</td>
</tr>
<tr>
<td></td>
<td>MURMUR32</td>
</tr>
</tbody>
</table>

Defines the algorithm used to generate a hash identifying the writes associated with a transaction. If you are using Group Replication, the hash value is used for distributed conflict detection and handling. On 64-bit systems running Group Replication, we recommend setting this to `XXHASH64` in order to avoid unnecessary hash collisions which result in certification failures and the roll back of user transactions. See Group Replication Requirements.

**Note**

The value of this variable cannot be changed when `binlog_transaction_dependency_tracking` is set to either of `WRITESET` or `WRITESET_SESSION`.

### 2.6.5 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 2.3, “Replication with Global Transaction Identifiers”.
Global Transaction ID System Variables

- `binlog_gtid_simple_recovery`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td>`--binlog-gtid-simple-recovery[={OFF</td>
</tr>
<tr>
<td>System Variable</td>
<td><code>binlog_gtid_simple_recovery</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

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, 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 and you are using MySQL 5.7.8 or later, `binlog_gtid_simple_recovery=TRUE` can always safely be used.

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 will remain incorrect even if the server is later restarted with `binlog_gtid_simple_recovery=FALSE`. If either of these situations apply on the server, set `binlog_gtid_simple_recovery=FALSE` before starting or restarting the server. To check for the second situation, if you are using MySQL 5.7.7 or earlier, after issuing a `SET @@GLOBAL.gtid_purged` statement note down the current binary log file name, which can be checked using `SHOW MASTER STATUS`. If the server is restarted before this file has been purged, then you should set `binlog_gtid_simple_recovery=FALSE`.

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.

In MySQL version 5.7.5, this variable was added as `simplified_binlog_gtid_recovery` and in MySQL version 5.7.6 it was renamed to `binlog_gtid_simple_recovery`.

- **enforce_gtid_consistency**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--enforce-gtid-consistency [=value]</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>enforce_gtid_consistency</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
<tr>
<td>Valid Values</td>
<td>OFF, ON, WARN</td>
</tr>
</tbody>
</table>

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. **WARN** was added in MySQL 5.7.6.

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 TABLE ... SELECT** statements
- **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.

`--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.

For more information, see Section 2.3.6, “Restrictions on Replication with GTIDs”.

Prior to MySQL 5.7.6, the boolean `enforce_gtid_consistency` defaulted to **OFF**. To maintain compatibility with previous releases, in MySQL 5.7.6 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 versions. 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`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>gtid_executed</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

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 SLAVE 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`.

Prior to MySQL 5.7.7, 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 was deprecated in MySQL 5.7.7.

•  `gtid_executed_compression_period`

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--gtid-executed-compression-period=#</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>gtid_executed_compression_period</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>1000</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>0</td>
</tr>
</tbody>
</table>
Global Transaction ID System Variables

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Maximum Value</td>
<td>4294967295</td>
</tr>
</tbody>
</table>

Compress the `mysql.gtid_executed` table each time this many transactions have been processed. A setting of 0 means that this table is not compressed. Since no compression of the table occurs when using the binary log, setting the value of the variable has no effect unless binary logging is disabled.

See [mysql.gtid_executed Table Compression](#), for more information.

This variable was added in MySQL version 5.7.5 as `executed_gtids_compression_period` and renamed in MySQL version 5.7.6 to `gtid_executed_compression_period`.

- **gtid_mode**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Command-Line Format</td>
<td><code>--gtid-mode=MODE</code></td>
</tr>
<tr>
<td>System Variable</td>
<td><code>gtid_mode</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td>OFF</td>
</tr>
<tr>
<td>Valid Values</td>
<td>OFF&lt;br&gt;OFF_PERMISSIVE&lt;br&gt;ON_PERMISSIVE&lt;br&gt;ON</td>
</tr>
</tbody>
</table>

Controls whether GTID based logging is enabled and what type of transactions the logs can contain. Prior to MySQL 5.7.6, this variable was read-only and was set using `--gtid-mode` at server startup only. Prior to MySQL 5.7.5, starting the server with `--gtid-mode=ON` required that the server also be started with the `--log-bin` and `--log-slave-updates` options. As of MySQL 5.7.5, this is no longer a requirement. See [mysql.gtid_executed Table](#).

MySQL 5.7.6 enables this variable to be set dynamically. You must have privileges sufficient to set global system variables. See [System Variable Privileges](#). `enforce_gtid_consistency` must be true before you can set `gtid_mode=ON`. Before modifying this variable, see Section 2.5, “Changing Replication Modes on Online Servers”.

Transactions logged in MySQL 5.7.6 and higher 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 OFF_PERMISSIVE and ON_PERMISSIVE modes added in MySQL 5.7.6 permit a mix of these transaction types in the topology. The different modes are now:

- **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.
Global Transaction ID System Variables

- **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`.

In MySQL 5.7.6 and higher, 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. In MySQL 5.7.5 and earlier, the values of `gtid_purged` and `gtid_executed` are not persistent while `gtid_mode=OFF`. Therefore, after changing `gtid_mode` to `OFF`, once all binary logs containing GTIDs are purged, the values of these variables are lost.

- **gtid_next**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>gtid_next</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td><code>AUTOMATIC</code></td>
</tr>
<tr>
<td>Valid Values</td>
<td><code>AUTOMATIC</code></td>
</tr>
<tr>
<td></td>
<td><code>ANONYMOUS</code></td>
</tr>
<tr>
<td></td>
<td><code>UUID:NUMBER</code></td>
</tr>
</tbody>
</table>

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 privileges sufficient to set restricted session variables. See [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 2.5.1, “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.

In MySQL 5.7.5 and higher, `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. Prior to MySQL 5.7.5, when GTIDs were enabled but `gtid_next` was not `AUTOMATIC`, `DROP TABLE` did not work correctly when used with either of these combinations of tables. (Bug #17620053)

In MySQL 5.7.1, you cannot execute any of the statements `CHANGE MASTER TO`, `START SLAVE`, `STOP SLAVE`, `REPAIR TABLE`, `OPTIMIZE TABLE`, `ANALYZE TABLE`, `CHECK TABLE`, `CREATE SERVER`, `ALTER SERVER`, `DROP SERVER`, `CACHE INDEX`, `LOAD INDEX INTO CACHE`, `FLUSH`, or
RESET when `gtid_next` is set to any value other than `AUTOMATIC`; in such cases, the statement fails with an error. Such statements are not disallowed in MySQL 5.7.2 and later. (Bug #16062608, Bug #16715809, Bug #69045) (Bug #16062608)

- **gtid_owned**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>gtid_owned</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global, Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

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 replication slave 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 only populated briefly during the execution of the commit statement for the transaction, so it cannot be observed from the session concerned, although it will be 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**

<table>
<thead>
<tr>
<th>Property</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>gtid_purged</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>String</td>
</tr>
</tbody>
</table>

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 slave.
- 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 or restarts, 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.

In MySQL 5.7, it is possible to update the value of `gtid_purged` only when `gtid_executed` is the empty string, and therefore `gtid_purged` is the empty string. This is the case either when replication has not been started previously, or when replication did not previously use GTIDs. Prior to MySQL 5.7.6, `gtid_purged` was also settable only when `gtid_mode=ON`. In MySQL 5.7.6 and higher, `gtid_purged` is settable regardless of the value of `gtid_mode`.

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

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

**Note**

If you are using MySQL 5.7.7 or earlier, 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. If all binary logs on the server were generated using MySQL 5.7.8 or later and you are using MySQL 5.7.8 or later, `binlog_gtid_simple_recovery=TRUE` (which is the default setting from MySQL 5.7.7) can always safely be used.

---

### 2.7 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 and how to pause a slave.

#### 2.7.1 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 slave and the master.

The `SHOW SLAVE STATUS` statement, which you must execute on each slave, provides information about the configuration and status of the connection between the slave server and the master server. From MySQL 5.7, the Performance Schema has replication tables that provide this information in a more accessible form. See Performance Schema Replication Tables.

The `SHOW STATUS` statement also provided some information relating specifically to replication slaves. As of MySQL version 5.7.5, the following status variables previously monitored using `SHOW STATUS` were deprecated and moved to the Performance Schema replication tables:

- `Slave_retried_transactions`
- `Slave_last_heartbeat`
- `Slave_received_heartbeats`
- `Slave_heartbeat_period`
- `Slave_running`
The replication heartbeat information shown in the Performance Schema replication tables lets you check that the replication connection is active even if the master has not sent events to the slave recently. The master sends a heartbeat signal to a slave 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 master (set by the `CHANGE MASTER TO` statement) specifies the frequency of the heartbeat, which defaults to half of the connection timeout interval for the slave (`slave_net_timeout`). The `replication_connection_status` Performance Schema table shows when the most recent heartbeat signal was received by a replication slave, and how many heartbeat signals it has received.

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

```
mysql> SHOW SLAVE STATUS\G
*************************** 1. row ***************************
Slave_IO_State: Waiting for master to send event
Master_Host: master1
Master_User: root
Master_Port: 3306
Connect_Retry: 60
Master_Log_File: mysql-bin.000004
Read_Master_Log_Pos: 931
Relay_Log_File: slave1-relay-bin.000056
Relay_Log_Pos: 950
Relay_Master_Log_File: mysql-bin.000004
Slave_IO_Running: Yes
Slave_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_Front: 0
Exec_Master_Log_Pos: 931
Relay_Log_Space: 1365
Until_Condition: None
Until_Log_File:
Until_Log_Pos: 0
Master_SSL_Allowed: No
Master_SSL_CA_File:
Master_SSL_CA_Path:
Master_SSL_Cert:
Master_SSL_Cipher:
Master_SSL_Key:
Seconds_Behind_Master: 0
Master_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:

- **`Slave_IO_State`**: The current status of the slave. See `Replication Slave I/O Thread States`, and `Replication Slave SQL Thread States`, for more information.

- **`Slave_IO_Running`**: Whether the I/O thread for reading the master'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 SLAVE`.

- **`Slave_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**.
Checking Replication Status

- **Last_IO_Error, Last_SQL_Error**: The last errors registered by the I/O and SQL threads when processing the relay log. Ideally these should be blank, indicating no errors.

- **Seconds_Behind_Master**: The number of seconds that the slave SQL thread is behind processing the master binary log. A high number (or an increasing one) can indicate that the slave is unable to handle events from the master in a timely fashion.

A value of 0 for Seconds_Behind_Master can usually be interpreted as meaning that the slave has caught up with the master, but there are some cases where this is not strictly true. For example, this can occur if the network connection between master and slave is broken but the slave I/O thread has not yet noticed this—that is, slave_net_timeout has not yet elapsed.

It is also possible that transient values for Seconds_Behind_Master may not reflect the situation accurately. When the slave SQL thread has caught up on I/O, Seconds_Behind_Master displays 0; but when the slave I/O thread is still queuing up a new event, Seconds_Behind_Master may show a large value until the SQL thread finishes executing the new event. This is especially likely when the events have old timestamps; in such cases, if you execute SHOW SLAVE 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 slave in reading events from the master binary log and processing them in the relay log:

- **(Master_Log_file, Read_Master_Log_Pos)**: Coordinates in the master binary log indicating how far the slave I/O thread has read events from that log.

- **(Relay_Master_Log_File, Exec_Master_Log_Pos)**: Coordinates in the master binary log indicating how far the slave SQL thread has executed events received from that log.

- **(Relay_Log_File, Relay_Log_Pos)**: Coordinates in the slave relay log indicating how far the slave SQL thread has executed the relay log. These correspond to the preceding coordinates, but are expressed in slave relay log coordinates rather than master binary log coordinates.

On the master, you can check the status of connected slaves using SHOW PROCESSLIST to examine the list of running processes. Slave connections have Binlog Dump in the Command field:

```
mysql> SHOW PROCESSLIST \G;
*************************** 4. row ***************************
    Id: 10
User: root
Host: slave1:58371
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 slave that drives the replication process, very little information is available in this report.

For slaves that were started with the --report-host option and are connected to the master, the SHOW SLAVE HOSTS statement on the master shows basic information about the slaves. The output includes the ID of the slave server, the value of the --report-host option, the connecting port, and master ID:

```
mysql> SHOW SLAVE HOSTS;
+-----------+--------+------+-------------------+-----------+
| Server_id | Host   | Port | Rpl_recovery_rank | Master_id |
|-----------+--------+------+-------------------+-----------+
|        10 | slave1 | 3306 |                 0 |         1 |
+-----------+--------+------+-------------------+-----------+
1 row in set (0.00 sec)
```
2.7.2 Pausing Replication on the Slave

You can stop and start replication on the slave using the `STOP SLAVE` and `START SLAVE` statements.

To stop processing of the binary log from the master, use `STOP SLAVE`:

```
mysql> STOP SLAVE;
```

When replication is stopped, the slave I/O thread stops reading events from the master 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 or SQL thread individually by specifying the thread type:

```
mysql> STOP SLAVE IO_THREAD;
mysql> STOP SLAVE SQL_THREAD;
```

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

```
mysql> START SLAVE;
```

To start a particular thread, specify the thread type:

```
mysql> START SLAVE IO_THREAD;
mysql> START SLAVE SQL_THREAD;
```

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

Stopping only the I/O thread enables the events in the relay log to be executed by the SQL 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 master, when you want to perform administration on the slave 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 slave while you conduct administration on the master. Stopping the I/O thread but permitting the SQL thread to run helps ensure that there is not a massive backlog of events to be executed when replication is started again.

2.7.3 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 slave. Before skipping a transaction, ensure that the replication slave's I/O thread is stopped as well as the SQL 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 Point-in-Time (Incremental) Recovery Using the Binary Log. Alternatively, you can issue `SHOW RELAYLOG EVENTS` on the slave or `SHOW BINLOG EVENTS` on the master.

Before skipping the transaction and restarting the slave, 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 slave should not be restarted.
- Does the transaction that stopped replication need to be applied on the slave? If so, either make the appropriate corrections and reapply the transaction, or manually reconcile the data on the slave.
- Did the transaction that stopped replication need to be applied on the master? 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 Section 2.7.3.1, “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 Section 2.7.3.2, “Skipping Transactions Without GTIDs”.

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

2.7.3.1 Skipping Transactions With GTIDs

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

If the failing transaction generated an error in a worker thread, you can obtain its GTID directly from the `LAST_SEEN_TRANSACTION` field in the Performance Schema table `replication_applier_status_by_worker`. To see what the transaction is, issue `SHOW RELAYLOG EVENTS` on the slave or `SHOW BINLOG EVENTS` on the master, 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 slave that has the same GTID as the failing transaction. For example:

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

The presence of this empty transaction on the slave means that when you issue a `START SLAVE` statement to restart replication, the slave uses the auto-skip function to ignore the failing transaction, because it sees a transaction with that GTID has already been applied. If the slave is a multi-source replication slave, 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 SLAVE`.

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

```sql
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.

2.7.3.2 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_slave_skip_counter` statement. Alternatively, you can skip past an event or events by issuing a `CHANGE MASTER TO` statement to move the master log position forward.

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_slave_skip_counter` statement to skip events and the resulting position is in the middle of an event group, the slave continues to skip events until it reaches the end of the group. Execution then starts with the next event group. The `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 MASTER TO` means you do not have to count the events that need to be skipped, as you do with a `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_slave_skip_counter`, the slave 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, as this will then also be skipped.

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

```
SET GLOBAL sql_slave_skip_counter = N
```

This statement cannot be issued if `gtid_mode=ON` is set, or if the slave threads are running.

The `SET GLOBAL sql_slave_skip_counter` statement has no immediate effect. When you issue the `START SLAVE` statement for the next time following this `SET` statement, the new value for the system variable `sql_slave_skip_counter` is applied, and the events are skipped. That `START SLAVE` statement also automatically sets the value of the system variable back to 0. If the slave is a multi-source replication slave, when you issue that `START SLAVE` statement, the `FOR CHANNEL` clause is required. Make sure that you name the correct channel, otherwise events will be 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 master’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 slave I/O thread will begin reading from the master 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 MASTER TO` statement as follows, where `master_log_name` is the binary log file that contains the restart position, and `master_log_pos` is the number representing the restart position as stated in the binary log file:

```
CHANGE MASTER TO MASTER_LOG_FILE='master_log_name', MASTER_LOG_POS=master_log_pos;
```

If the slave is a multi-source replication slave, you must use the `FOR CHANNEL` clause to name the appropriate channel on the `CHANGE MASTER TO` statement.
This statement cannot be issued if `MASTER_AUTO_POSITION=1` is set, or if the slave threads are running. If you need to use this method of skipping a transaction when `MASTER_AUTO_POSITION=1` is normally set, you can change the setting to `MASTER_AUTO_POSITION=1` 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;
```
Chapter 3 Replication Solutions

Table of Contents

3.1 Using Replication for Backups .......................................................... 127
  3.1.1 Backing Up a Slave Using mysqldump ........................................... 128
  3.1.2 Backing Up Raw Data from a Slave ............................................. 128
  3.1.3 Backing Up a Master or Slave by Making It Read Only ..................... 129
3.2 Handling an Unexpected Halt of a Replication Slave .............................. 131
3.3 Using Replication with Different Master and Slave Storage Engines ............ 133
3.4 Using Replication for Scale-Out ........................................................ 134
3.5 Replicating Different Databases to Different Slaves .............................. 135
3.6 Improving Replication Performance .................................................... 136
3.7 Switching Masters During Failover ...................................................... 137
3.8 Setting Up Replication to Use Encrypted Connections ......................... 139
3.9 Semisynchronous Replication ............................................................. 141
  3.9.1 Semisynchronous Replication Administrative Interface .................... 143
  3.9.2 Semisynchronous Replication Installation and Configuration .............. 144
  3.9.3 Semisynchronous Replication Monitoring ..................................... 146
3.10 Delayed Replication ........................................................................ 146

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 3.1, “Using Replication for Backups”.

For advice and tips on using different storage engines on the master and slaves, see Section 3.3, “Using Replication with Different Master and Slave 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 3.4, “Using Replication for Scale-Out”.

For performance or data distribution reasons, you may want to replicate different databases to different replication slaves. See Section 3.5, “Replicating Different Databases to Different Slaves”

As the number of replication slaves increases, the load on the master can increase and lead to reduced performance (because of the need to replicate the binary log to each slave). For tips on improving your replication performance, including using a single secondary server as a replication master, see Section 3.6, “Improving Replication Performance”.

For guidance on switching masters, or converting slaves into masters as part of an emergency failover solution, see Section 3.7, “Switching Masters During Failover”.

To secure your replication communication, you can encrypt the communication channel. For step-by-step instructions, see Section 3.8, “Setting Up Replication to Use Encrypted Connections”.

3.1 Using Replication for Backups

To use replication as a backup solution, replicate data from the master to a slave, and then back up the data slave. The slave can be paused and shut down without affecting the running operation of the master, so you can produce an effective snapshot of “live” data that would otherwise require the master 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 replication slave state so that you can rebuild the slave in the event of failure. There are therefore two choices:
Back up a Slave Using `mysqldump`

- If you are using replication as a solution to enable you to back up the data on the master, and the size of your database is not too large, the `mysqldump` tool may be suitable. See Section 3.1.1, “Back up a Slave 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 will enable you to recreate the slave in the event of a slave failure. For more information, see Section 3.1.2, “Back up Raw Data from a Slave”.

Another backup strategy, which can be used for either master or slave 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 3.1.3, “Back up a Master or Slave by Making It Read Only”.

### 3.1.1 Backing Up a Slave 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 `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.

When using `mysqldump`, you should stop replication on the slave before starting the dump process to ensure that the dump contains a consistent set of data:

1. Stop the slave from processing requests. You can stop replication completely on the slave using `mysqladmin`:

   ```shell
   mysqladmin stop-slave
   ```

   Alternatively, you can stop only the slave SQL thread to pause event execution:

   ```shell
   mysql -e 'STOP SLAVE SQL_THREAD;'
   ```

   This enables the slave to continue to receive data change events from the master’s binary log and store them in the relay logs using the I/O thread, but prevents the slave from executing these events and changing its data. Within busy replication environments, permitting the I/O thread to run during backup may speed up the catch-up process when you restart the slave SQL 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:

   ```shell
   mysqldump --all-databases > fulldb.dump
   ```

3. Once the dump has completed, start slave operations again:

   ```shell
   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 slave replication process to ensure that the time taken to run the backup does not affect the slave's ability to keep up with events from the master. See Section 2.7.1, “Checking Replication Status”. If the slave is unable to keep up, you may want to add another slave and distribute the backup process. For an example of how to configure this scenario, see Section 3.5, “Replicating Different Databases to Different Slaves”.

### 3.1.2 Backing Up Raw Data from a Slave

To guarantee the integrity of the files that are copied, backing up the raw data files on your MySQL replication slave should take place while your slave server is shut down. If the MySQL server is still
Backing Up a Master or Slave by Making It Read Only

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 slave server can be shut down during the backup process without affecting the execution of the master it makes sense to take advantage of this capability.

To shut down the server and back up the files:

1. Shut down the slave MySQL server:

   `shell> 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:

   `shell> tar cf /tmp/dbbackup.tar ./data`

3. Start the MySQL server again. Under Unix:

   `shell> mysqld_safe &`

   Under Windows:

   `C:\> "C:\Program Files\MySQL\MySQL Server 5.7\bin\mysqld"`

Normally you should back up the entire data directory for the slave MySQL server. If you want to be able to restore the data and operate as a slave (for example, in the event of failure of the slave), then in addition to the slave's data, you should also back up the slave status files, the master info and relay log info repositories, and the relay log files. These files are needed to resume replication after you restore the slave's data.

If you lose the relay logs but still have the `relay-log.info` file, you can check it to determine how far the SQL thread has executed in the master binary logs. Then you can use `CHANGE MASTER TO` with the `MASTER_LOG_FILE` and `MASTER_LOG_POS` options to tell the slave to re-read the binary logs from that point. This requires that the binary logs still exist on the master server.

If your slave is replicating `LOAD DATA` statements, you should also back up any `SQL_LOAD-*` files that exist in the directory that the slave uses for this purpose. The slave needs these files to resume replication of any interrupted `LOAD DATA` operations. The location of this directory is the value of the `slave_load_tmpdir` system variable. If the server was not started with that variable set, the directory location is the value of the `tmpdir` system variable.

### 3.1.3 Backing Up a Master or Slave by Making It Read Only

It is possible to back up either master or slave 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 `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 master server and for a slave server. For both scenarios discussed here, suppose that you have the following replication setup:

- A master server M1
- A slave server S1 that has M1 as its master
- A client C1 connected to M1
- A client C2 connected to S1

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 slaves of that server.

**Scenario 1: Backup with a Read-Only Master**

Put the master M1 in a read-only state by executing these statements on it:

```sql
mysql> FLUSH TABLES WITH READ LOCK;
mysql> SET GLOBAL read_only = ON;
```

While M1 is in a read-only state, the following properties are true:

- Requests for updates sent by C1 to M1 will block because the server is in read-only mode.
- Requests for query results sent by C1 to M1 will succeed.
- Making a backup on M1 is safe.
- Making a backup on S1 is not safe. This server is still running, and might be processing the binary log or update requests coming from client C2

While M1 is read only, perform the backup. For example, you can use `mysqldump`.

After the backup operation on M1 completes, restore M1 to its normal operational state by executing these statements:

```sql
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

Although performing the backup on M1 is safe (as far as the backup is concerned), it is not optimal for performance because clients of M1 are blocked from executing updates.

This strategy applies to backing up a master server in a replication setup, but can also be used for a single server in a nonreplication setting.

**Scenario 2: Backup with a Read-Only Slave**

Put the slave S1 in a read-only state by executing these statements on it:

```sql
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:

- The master M1 will continue to operate, so making a backup on the master is not safe.
- The slave S1 is stopped, so making a backup on the slave S1 is safe.

These properties provide the basis for a popular backup scenario: Having one slave 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 master server, which remains unaffected by backup activity on the slave.

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:

```sql
mysql> SET GLOBAL read_only = OFF;
mysql> UNLOCK TABLES;
```

After the slave is restored to normal operation, it again synchronizes to the master by catching up with any outstanding updates from the binary log of the master.

### 3.2 Handling an Unexpected Halt of a Replication Slave

In order for replication to be resilient to unexpected halts of the server (sometimes described as crash-safe) it must be possible for the slave to recover its state before halting. This section describes the impact of an unexpected halt of a slave during replication and how to configure a slave for the best chance of recovery to continue replication.

After an unexpected halt of a replication slave, upon restart the slave’s SQL thread must recover which transactions have been executed already. The information required for recovery is stored in the slave’s relay log info log. In older MySQL Server versions, this log could only be created as a file in the data directory that was updated after the transaction had been applied. This held the risk of losing synchrony with the master depending at which stage of processing a transaction the slave halted at, or even corruption of the file itself. In MySQL 5.7 you can instead use an InnoDB table to store the relay log info log. By using this transactional storage engine the information is always recoverable upon restart. As a table, updates to the relay log info log are committed together with the transactions, meaning that the slave’s progress information recorded in that log is always consistent with what has been applied to the database, even in the event of an unexpected server halt.

To configure MySQL 5.7 to store the relay log info log as an InnoDB table, set the system variable `relay_log_info_repository` to `TABLE`. The server then stores information required for the recovery of the slave’s SQL thread in the `mysql.slave_relay_log_info` table. For further information on the slave logs, see Section 5.4, “Replication Relay and Status Logs”.

Exactly how a replication slave recovers from an unexpected halt is influenced by the chosen method of replication, whether the slave is single-threaded or multithreaded, the setting of variables such as `relay_log_recovery`, and whether features such as `MASTER_AUTO_POSITION` are being used.

The following table shows the impact of these different factors on how a single-threaded slave recovers from an unexpected halt.

<table>
<thead>
<tr>
<th>MASTER_AUTO_POSITION</th>
<th>relay_log_recovery</th>
<th>relay_log_info_repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
</tr>
</tbody>
</table>

As the table shows, when using a single-threaded slave the following configurations are most resilient to unexpected halts:

- When using GTIDs and `MASTER_AUTO_POSITION`, set `relay_log_recovery=1`. With this configuration the setting of `relay_log_info_repository` and other variables does not impact on recovery. Note that to guarantee recovery, `sync_binlog=1` (which is the default) must also
be set on the slave, so that the slave's binary log is synchronized to disk at each write. Otherwise, committed transactions might not be present in the slave's binary log.

- When using file position based replication, set `relay_log_recovery=1` and `relay_log_info_repository=TABLE`.

  **Note**  
  During recovery the relay log is lost.

The following table shows the impact of these different factors on how a multithreaded slave recovers from an unexpected halt.

<table>
<thead>
<tr>
<th>GTID</th>
<th>sync_relay_log</th>
<th>MASTER_AUTO_POSITION</th>
<th>relay_log_recovery</th>
<th>relay_log_info_repository</th>
<th>Crash type</th>
<th>Recovery guaranteed</th>
<th>Relay log impact</th>
</tr>
</thead>
<tbody>
<tr>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
<td>Any</td>
<td>Yes</td>
<td>Lost</td>
</tr>
<tr>
<td>OFF</td>
<td>&gt;1</td>
<td>OFF</td>
<td>OFF</td>
<td>TABLE</td>
<td>Server</td>
<td>Yes</td>
<td>Lost</td>
</tr>
<tr>
<td>OFF</td>
<td>&gt;1</td>
<td>OFF</td>
<td>OFF</td>
<td>Any</td>
<td>OS</td>
<td>No</td>
<td>Lost</td>
</tr>
<tr>
<td>OFF</td>
<td>1</td>
<td>OFF</td>
<td>0</td>
<td>TABLE</td>
<td>OS</td>
<td>No</td>
<td>Remains</td>
</tr>
<tr>
<td>ON</td>
<td>Any</td>
<td>ON</td>
<td>Any</td>
<td>Any</td>
<td>Any</td>
<td>Yes</td>
<td>Lost</td>
</tr>
</tbody>
</table>

As the table shows, when using a multithreaded slave the following configurations are most resilient to unexpected halts:

- When using GTIDs and `MASTER_AUTO_POSITION=ON`, set `relay_log_recovery=1`. With this configuration the setting of `relay_log_info_repository` and other variables does not impact on recovery. From MySQL 5.7.28 a multithreaded slave automatically skips relay log recovery when `MASTER_AUTO_POSITION` is set to `ON`, so the setting for `relay_log_recovery` makes no difference.

- When using file position based replication, set `relay_log_recovery=1`, `sync_relay_log=1`, and `relay_log_info_repository=TABLE`.

  **Note**  
  During recovery the relay log is lost.

It is important to note the impact of `sync_relay_log=1`, which requires a write of to the relay log per transaction. Although this setting is the most resilient to an unexpected halt, with at most one unwritten transaction being lost, it also has the potential to greatly increase the load on storage. Without `sync_relay_log=1`, the effect of an unexpected halt depends on how the relay log is handled by the operating system. Also note that when `relay_log_recovery=0`, the next time the slave is started after an unexpected halt the relay log is processed as part of recovery. After this process completes, the relay log is deleted.

An unexpected halt of a multithreaded replication slave using the recommended file position based replication configuration above may result in a relay log with transaction inconsistencies (gaps in the sequence of transactions) caused by the unexpected halt. See Section 4.1.32, “Replication and Transaction Inconsistencies”. In MySQL 5.7.13 and later, if the relay log recovery process encounters such transaction inconsistencies they are filled and the recovery process continues automatically. In MySQL versions prior to MySQL 5.7.13, this process was not automatic and required starting the server with `relay_log_recovery=0`, starting the slave with `START SLAVE UNTIL`
Using Replication with Different Master and Slave Storage Engines

SQL_AFTER_MTS_GAPS to fix any transaction inconsistencies, and then restarting the slave with `relay_log_recovery=1`.

When you are using multi-source replication and `relay_log_recovery=1`, after restarting due to an unexpected halt all replication channels go through the relay log recovery process. Any inconsistencies found in the relay log due to an unexpected halt of a multithreaded slave are filled.

3.3 Using Replication with Different Master and Slave Storage Engines

It does not matter for the replication process whether the source table on the master and the replicated table on the slave use different engine types. In fact, the `default_storage_engine` and `storage_engine` system variables are 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 3.4, “Using Replication for Scale-Out”), you want to use InnoDB tables on the master to take advantage of the transactional functionality, but use MyISAM on the slaves 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 slave.

Configuring different engines on the master and slave depends on how you set up the initial replication process:

- If you used `mysqldump` to create the database snapshot on your master, 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 slave before using the dump to build the data on the slave. For example, you can add the `--skip-federated` option on your slave to disable the FEDERATED engine. If a specific engine does not exist for a table to be created, MySQL will use the default engine type, usually MyISAM. (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 slave that only supports the engines you want.

- If you are using raw data files (a binary backup) to set up the slave, you will be unable to change the initial table format. Instead, use `ALTER TABLE` to change the table types after the slave has been started.

- For new master/slave replication setups where there are currently no tables on the master, 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 slave from running replication updates:

   ```sql
   mysql> STOP SLAVE;
   ```

   This will enable you to change engine types without interruptions.

2. Execute an `ALTER TABLE ... ENGINE='engine_type'` for each table to be changed.

3. Start the slave replication process again:

   ```sql
   mysql> START SLAVE;
   ```

   Although the `default_storage_engine` variable is not replicated, be aware that `CREATE TABLE` and `ALTER TABLE` statements that include the engine specification will be correctly replicated to the slave. For example, if you have a CSV table and you execute:

   ```sql
   mysql> ALTER TABLE csvtable Engine='MyISAM';
   ```
The above statement will be replicated to the slave and the engine type on the slave will be converted to MyISAM, even if you have previously changed the table type on the slave to an engine other than CSV. If you want to retain engine differences on the master and slave, you should be careful to use the default_storage_engine variable on the master when creating a new table. For example, instead of:

```sql
mysql> CREATE TABLE tablea (columna int) Engine=MyISAM;
```

Use this format:

```sql
mysql> SET default_storage_engine=MyISAM;
mysql> CREATE TABLE tablea (columna int);
```

When replicated, the default_storage_engine variable will be ignored, and the CREATE TABLE statement will execute on the slave using the slave's default engine.

### 3.4 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 master to one or more slaves, 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 replication slaves, while still enabling your web servers to communicate with the replication master when a write is required. You can see a sample replication layout for this scenario in Figure 3.1, "Using Replication to Improve Performance During Scale-Out".

**Figure 3.1 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 master, and to send reads to either the master or a slave. 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()`
Replicating Different Databases to Different Slaves

- `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 master/slave configuration, even one involving multiple slaves. 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 using the `replace` utility that comes with standard MySQL distributions, or write your own 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.

3.5 Replicating Different Databases to Different Slaves

There may be situations where you have a single master and want to replicate different databases to different slaves. 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 3.2, “Using Replication to Replicate Databases to Separate Replication Slaves”.

![Figure 3.2 Using Replication to Replicate Databases to Separate Replication Slaves](image)

You can achieve this separation by configuring the master and slaves as normal, and then limiting the binary log statements that each slave processes by using the `--replicate-wild-do-table` configuration option on each slave.

**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 3.2, “Using Replication to Replicate Databases to Separate Replication Slaves”, you should configure each replication slave as follows, before executing `START SLAVE`:

- Replication slave 1 should use `--replicate-wild-do-table=databaseA.%`.
- Replication slave 2 should use `--replicate-wild-do-table=databaseB.%`.

135
Improving Replication Performance

- Replication slave 3 should use `--replicate-wild-do-table=databaseC.%`.

Each slave in this configuration receives the entire binary log from the master, 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 slave.

If you have data that must be synchronized to the slaves before replication starts, you have a number of choices:

- Synchronize all the data to each slave, 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 slave.
- Use a raw data file dump and include only the specific files and databases that you need for each slave.

Note

This does not work with InnoDB databases unless you use `innodb_file_per_table`.

3.6 Improving Replication Performance

As the number of slaves connecting to a master increases, the load, although minimal, also increases, as each slave uses a client connection to the master. Also, as each slave must receive a full copy of the master binary log, the network load on the master may also increase and create a bottleneck.

If you are using a large number of slaves connected to one master, and that master 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 master to replicate to only one slave, and for the remaining slaves to connect to this primary slave for their individual replication requirements. A sample of this structure is shown in Figure 3.3, “Using an Additional Replication Host to Improve Performance”.

Figure 3.3 Using an Additional Replication Host to Improve Performance

For this to work, you must configure the MySQL instances as follows:

- Master 1 is the primary master where all changes and updates are written to the database. Binary logging should be enabled on this machine.
- Master 2 is the slave to the Master 1 that provides the replication functionality to the remainder of the slaves in the replication structure. Master 2 is the only machine permitted to connect to Master 1. Master 2 also has binary logging enabled, and the `log_slave_updates` system variable enabled so that replication instructions from Master 1 are also written to Master 2's binary log so that they can then be replicated to the true slaves.
- Slave 1, Slave 2, and Slave 3 act as slaves to Master 2, and replicate the information from Master 2, which actually consists of the upgrades logged on Master 1.
Switching Masters During Failover

The above solution reduces the client load and the network interface load on the primary master, which should improve the overall performance of the primary master when used as a direct database solution.

If your slaves are having trouble keeping up with the replication process on the master, 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 the slaves are significantly slower than the master, you may want to divide up the responsibility for replicating different databases to different slaves. See Section 3.5, “Replicating Different Databases to Different Slaves”.

- If your master makes use of transactions and you are not concerned about transaction support on your slaves, use `MyISAM` or another nontransactional engine on the slaves. See Section 3.3, “Using Replication with Different Master and Slave Storage Engines”.

- If your slaves are not acting as masters, and you have a potential solution in place to ensure that you can bring up a master in the event of failure, then you can disable the `log_slave_updates` system variable on the slaves. This prevents “dumb” slaves from also logging events they have executed into their own binary log.

3.7 Switching Masters During Failover

You can tell a slave to change to a new master using the `CHANGE MASTER TO` statement. The slave does not check whether the databases on the master are compatible with those on the slave; it simply begins reading and executing events from the specified coordinates in the new master'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.

Slaves should be run with the `--log-bin` option, and if not using GTIDs then they should also be run without enabling the `log_slave_updates` system variable. In this way, the slave is ready to become a master without restarting the slave `mysqld`. Assume that you have the structure shown in Figure 3.4, “Redundancy Using Replication, Initial Structure”.

Figure 3.4 Redundancy Using Replication, Initial Structure

In this diagram, the MySQL Master holds the master database, the MySQL Slave hosts are replication slaves, and the Web Client machines are issuing database reads and writes. Web clients that issue only reads (and would normally be connected to the slaves) 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 3.4, “Using Replication for Scale-Out”.

137
Each MySQL Slave (Slave 1, Slave 2, and Slave 3) is a slave running with --log-bin and without enabling the log_slave_updates system variable. Because updates received by a slave from the master are not logged in the binary log unless log_slave_updates is enabled, the binary log on each slave is empty initially. If for some reason MySQL Master becomes unavailable, you can pick one of the slaves to become the new master. For example, if you pick Slave 1, all Web Clients should be redirected to Slave 1, which writes the updates to its binary log. Slave 2 and Slave 3 should then replicate from Slave 1.

The reason for running the slave without log_slave_updates enabled is to prevent slaves from receiving updates twice in case you cause one of the slaves to become the new master. If Slave 1 has log_slave_updates enabled, it writes any updates that it receives from Master in its own binary log. This means that, when Slave 2 changes from Master to Slave 1 as its master, it may receive updates from Slave 1 that it has already received from Master.

Make sure that all slaves have processed any statements in their relay log. On each slave, issue STOP SLAVE IO_THREAD, then check the output of SHOW PROCESSLIST until you see Has read all relay log. When this is true for all slaves, they can be reconfigured to the new setup. On the slave Slave 1 being promoted to become the master, issue STOP SLAVE and RESET MASTER.

On the other slaves Slave 2 and Slave 3, use STOP SLAVE and CHANGE MASTER TO MASTER_HOST='Slave1' (where 'Slave1' represents the real host name of Slave 1). To use CHANGE MASTER TO, add all information about how to connect to Slave 1 from Slave 2 or Slave 3 (user, password, port). When issuing the CHANGE MASTER TO statement in this, there is no need to specify the name of the Slave 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 SLAVE on Slave 2 and Slave 3.

Once the new replication setup is in place, you need to tell each Web Client to direct its statements to Slave 1. From that point on, all updates statements sent by Web Client to Slave 1 are written to the binary log of Slave 1, which then contains every update statement sent to Slave 1 since Master died.

The resulting server structure is shown in Figure 3.5, “Redundancy Using Replication, After Master Failure”.

Figure 3.5 Redundancy Using Replication, After Master Failure

When Master becomes available again, you should make it a slave of Slave 1. To do this, issue on Master the same CHANGE MASTER TO statement as that issued on Slave 2 and Slave 3.
previously. Master then becomes a slave of Slave 1 and picks up the Web Client writes that it missed while it was offline.

To make Master a master again, use the preceding procedure as if Slave 1 was unavailable and Master was to be the new master. During this procedure, do not forget to run RESET MASTER on Master before making Slave 1, Slave 2, and Slave 3 slaves of Master. If you fail to do this, the slaves may pick up stale writes from the Web Client applications dating from before the point at which Master became unavailable.

You should be aware that there is no synchronization between slaves, even when they share the same master, and thus some slaves 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 slaves should be relatively close together.

One way to keep applications informed about the location of the master is to have a dynamic DNS entry for the master. With bind you can use nsupdate to update the DNS dynamically.

### 3.8 Setting Up Replication to Use Encrypted Connections

To use an encrypted connection for the transfer of the binary log required during replication, both the master and the slave 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 master, and a similar certificate (from the same certificate authority) on each slave. You must also obtain suitable key files.

For more information on setting up a server and client for encrypted connections, see Configuring MySQL to Use Encrypted Connections.

To enable encrypted connections on the master, you must create or obtain suitable certificate and key files, and then add the following configuration parameters to the master’s configuration within the [mysqld] section of the master’s 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 slave, use the CHANGE MASTER TO statement. You can either name the slave certificate and SSL private key files required for the encrypted connection in the [client] section of the slave’s my.cnf file, or you can explicitly specify that information using the CHANGE MASTER TO statement. For more information on the CHANGE MASTER TO statement, see CHANGE MASTER TO Statement.

- To name the slave certificate and key files using an option file, add the following lines to the [client] section of the slave’s my.cnf file, changing the file names as necessary:

```
[client]
ssl-ca=cacert.pem
```
ssl-cert=client-cert.pem
ssl-key=client-key.pem

- Restart the slave server, using the `--skip-slave-start` option to prevent the slave from connecting to the master. Use `CHANGE MASTER TO` to specify the master configuration, and add the `MASTER_SSL` option to connect using encryption:

```
mysql> CHANGE MASTER TO
   -> MASTER_HOST='master_hostname',
   -> MASTER_USER='repl',
   -> MASTER_PASSWORD='password',
   -> MASTER_SSL=1;
```

Setting `MASTER_SSL=1` for a replication connection and then setting no further `MASTER_SSL_xxx` options corresponds to setting `--ssl-mode=REQUIRED` for the client, as described in Command Options for Encrypted Connections. With `MASTER_SSL=1`, the connection attempt only succeeds if an encrypted connection can be established. A replication connection does not fall back to an unencrypted connection, so there is no setting corresponding to the `--ssl-mode=PREFERRED` setting for replication. If `MASTER_SSL=0` is set, this corresponds to `--ssl-mode=DISABLED`.

- To name the slave certificate and SSL private key files using the `CHANGE MASTER TO` statement, if you did not do this in the slave's `my.cnf` file, add the appropriate `MASTER_SSL_xxx` options:

```
   -> MASTER_SSL_CA = 'ca_file_name',
   -> MASTER_SSL_CAPATH = 'ca_directory_name',
   -> MASTER_SSL_CERT = 'cert_file_name',
   -> MASTER_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, `MASTER_SSL=1` must also be set. For a replication connection, specifying a value for either of `MASTER_SSL_CA` or `MASTER_SSL_CAPATH`, or specifying these options in the slave's `my.cnf` file, corresponds to setting `--ssl-mode=VERIFY_CA`. The connection attempt only succeeds if a valid matching Certificate Authority (CA) certificate is found using the specified information.

- To activate host name identity verification, add the `MASTER_SSL_VERIFY_SERVER_CERT` option:

```
   -> MASTER_SSL_VERIFY_SERVER_CERT=1,
```

This option corresponds to the `--ssl-verify-server-cert` option, which is deprecated as of MySQL 5.7.11 and is 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, `MASTER_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 `MASTER_SSL_CRL` or `MASTER_SSL_CRLPATH` option:

```
   -> MASTER_SSL_CRL = 'crl_file_name',
   -> MASTER_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 and encryption protocols permitted by the slave for the replication connection, add the `MASTER_SSL_CIPHER` and `MASTER_TLS_VERSION` options:

```
   -> MASTER_SSL_CIPHER = 'cipher_list',
   -> MASTER_TLS_VERSION = 'protocol_list',
```

The `MASTER_SSL_CIPHER` option specifies the list of ciphers permitted by the slave for the replication connection, with one or more cipher names separated by colons. The
The **MASTER_TLS_VERSION** option specifies the encryption protocols permitted by the slave for the replication connection. The format is like that for the **tls_version** system variable, with one or more comma-separated protocol versions. The protocols and ciphers that you can use in these lists depend on the SSL library used to compile MySQL. For information about the formats and permitted values, see Encrypted Connection TLS Protocols and Ciphers.

- After the master information has been updated, start the slave replication process:
  ```sql
  mysql> START SLAVE;
  ```

You can use the **SHOW SLAVE STATUS** statement to confirm that an encrypted connection was established successfully.

- Requiring encrypted connections on the slave does not ensure that the master requires encrypted connections from slaves. If you want to ensure that the master only accepts replication slaves that connect using encrypted connections, create a replication user account on the master using the **REQUIRE SSL** option, then grant that user the **REPLICATION SLAVE** privilege. For example:
  ```sql
  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 master, you can add **REQUIRE SSL** to it with this statement:
  ```sql
  mysql> ALTER USER 'repl'@'%.example.com' REQUIRE SSL;
  ```

### 3.9 Semisynchronous Replication

In addition to the built-in asynchronous replication, MySQL 5.7 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 master writes events to its binary log but does not know whether or when a slave has retrieved and processed them. With asynchronous replication, if the master crashes, transactions that it has committed might not have been transmitted to any slave. Consequently, failover from master to slave in this case may result in failover to a server that is missing transactions relative to the master.

Semisynchronous replication can be used as an alternative to asynchronous replication:

- A slave indicates whether it is semisynchronous-capable when it connects to the master.
- If semisynchronous replication is enabled on the master side and there is at least one semisynchronous slave, a thread that performs a transaction commit on the master blocks and waits until at least one semisynchronous slave acknowledges that it has received all events for the transaction, or until a timeout occurs.
- The slave 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 slave having acknowledged the transaction, the master reverts to asynchronous replication. When at least one semisynchronous slave catches up, the master returns to semisynchronous replication.
- Semisynchronous replication must be enabled on both the master and slave sides. If semisynchronous replication is disabled on the master, or enabled on the master but on no slaves, the master uses asynchronous replication.

While the master is blocking (waiting for acknowledgment from a slave), it does not return to the session that performed the transaction. When the block ends, the master returns to the session, which
then can proceed to execute other statements. At this point, the transaction has committed on the master side, and receipt of its events has been acknowledged by at least one slave.

The number of slave acknowledgments the master must receive per transaction before proceeding is configurable using the `rpl_semi_sync_master_wait_for_slave_count` system variable. The default value is 1.

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 slaves.

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 master blocks for each such statement, just as it does for explicit transaction commits.

To understand what the “semi” in “semisynchronous replication” means, compare it with asynchronous and fully synchronous replication:

• With asynchronous replication, the master writes events to its binary log and slaves request them when they are ready. There is no guarantee that any event will ever reach any slave.

• With fully synchronous replication, when a master commits a transaction, all slaves also will have committed the transaction before the master returns to the session that performed the transaction. The drawback of this is that there might be a lot of delay to complete a transaction.

• Semisynchronous replication falls between asynchronous and fully synchronous replication. The master waits only until at least one slave has received and logged the events. It does not wait for all slaves to acknowledge receipt, and it requires only receipt, not that the events have been fully executed and committed on the slave side.

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 master receives acknowledgment from the number of slaves configured by `rpl_semi_sync_master_wait_for_slave_count`, the transaction is on hold and not committed.

Semisynchronous replication also places a rate limit on busy sessions by constraining the speed at which binary log events can be sent from master to slave. When one user is too busy, this will slow it down, which is useful in some deployment situations.

Semisynchronous replication does have some performance impact because commits are slower due to the need to wait for slaves. This 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 slave and wait for the acknowledgment of receipt by the slave. This means that semisynchronous replication works best for close servers communicating over fast networks, and worst for distant servers communicating over slow networks.

The `rpl_semi_sync_master_wait_point` system variable controls the point at which a semisynchronous replication master waits for slave acknowledgment of transaction receipt before returning a status to the client that committed the transaction. These values are permitted:

• `AFTER_SYNC` (the default): The master writes each transaction to its binary log and the slave, and syncs the binary log to disk. The master waits for slave acknowledgment of transaction receipt after the sync. Upon receiving acknowledgment, the master commits the transaction to the storage engine and returns a result to the client, which then can proceed.

• `AFTER_COMMIT`: The master writes each transaction to its binary log and the slave, syncs the binary log, and commits the transaction to the storage engine. The master waits for slave acknowledgment of transaction receipt after the commit. Upon receiving acknowledgment, the master 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: After it has been acknowledged by the slave and committed to the storage engine on the master. Thus, all clients see the same data on the master.

In the event of master failure, all transactions committed on the master have been replicated to the slave (saved to its relay log). A crash of the master and failover to the slave is lossless because the slave is up to date.

- With `AFTER_COMMIT`, the client issuing the transaction gets a return status only after the server commits to the storage engine and receives slave acknowledgment. After the commit and before slave acknowledgment, other clients can see the committed transaction before the committing client.

If something goes wrong such that the slave does not process the transaction, then in the event of a master crash and failover to the slave, it is possible that such clients will see a loss of data relative to what they saw on the master.

### 3.9.1 Semisynchronous Replication Administrative Interface

The administrative interface to semisynchronous replication has several components:

- Two plugins implement semisynchronous capability. There is one plugin for the master side and one for the slave side.

- System variables control plugin behavior. Some examples:
  - `rpl_semi_sync_master_enabled`
    Controls whether semisynchronous replication is enabled on the master. To enable or disable the plugin, set this variable to 1 or 0, respectively. The default is 0 (off).
  
  - `rpl_semi_sync_master_timeout`
    A value in milliseconds that controls how long the master waits on a commit for acknowledgment from a slave before timing out and reverting to asynchronous replication. The default value is 10000 (10 seconds).
  
  - `rpl_semi_sync_slave_enabled`
    Similar to `rpl_semi_sync_master_enabled`, but controls the slave plugin.

All `rpl_semi_sync_` system variables are described at Server System Variables.

- Status variables enable semisynchronous replication monitoring. Some examples:
  - `Rpl_semi_sync_master_clients`
    The number of semisynchronous slaves.
  
  - `Rpl_semi_sync_master_status`
    Whether semisynchronous replication currently is operational on the master. 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 master has fallen back to asynchronous replication due to commit acknowledgment timeout.
  
  - `Rpl_semi_sync_master_no_tx`
    The number of commits that were not acknowledged successfully by a slave.
  
  - `Rpl_semi_sync_master_yes_tx`
The number of commits that were acknowledged successfully by a slave.

- **Rpl_semi_sync_slave_status**

  Whether semisynchronous replication currently is operational on the slave. This is 1 if the plugin has been enabled and the slave I/O thread is running, 0 otherwise.

  All **Rpl_semi_sync_xxx** status variables are described at [Server Status Variables](#).

The system and status variables are available only if the appropriate master or slave plugin has been installed with **INSTALL PLUGIN**.

### 3.9.2 Semisynchronous Replication Installation and Configuration

Semisynchronous replication is implemented using plugins, so the plugins must be installed into the server to make them available. After a plugin has been installed, you control it by means of the system variables associated with it. These system variables are unavailable until the associated plugin has been installed.

This section describes how to install the semisynchronous replication plugins. For general information about installing plugins, see [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 [Chapter 2, Configuring Replication](#).

- There must not be multiple replication channels configured. Semisynchronous replication is only compatible with the default replication channel. See [Section 5.3, “Replication Channels”](#).

To set up semisynchronous replication, use the following instructions. The **INSTALL PLUGIN, SET GLOBAL, STOP SLAVE, and START SLAVE** statements mentioned here require the **SUPER** privilege.

MySQL distributions include semisynchronous replication plugin files for the master side and the slave side.

To be usable by a master or slave 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 plugin library file base names are **semisync_master** and **semisync_slave**. The file name suffix differs per platform (for example, `.so` for Unix and Unix-like systems, `.dll` for Windows).

The master plugin library file must be present in the plugin directory of the master server. The slave plugin library file must be present in the plugin directory of each slave server.

To load the plugins, use the **INSTALL PLUGIN** statement on the master and on each slave that is to be semisynchronous (adjust the `.so` suffix for your platform as necessary).

**On the master:**

```sql
INSTALL PLUGIN rpl_semi_sync_master SONAME 'semisync_master.so';
```

**On each slave:**

```sql
INSTALL PLUGIN rpl_semi_sync_slave SONAME 'semisync_slave.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_master SONAME 'semisync_master.so';
ERROR 1126 (HY000): Can't open shared library
'/usr/local/mysql/lib/plugin/semisync_master.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 see which plugins are installed, use the SHOW PLUGINS statement, or query the INFORMATION_SCHEMA.PLUGINS table.

To verify plugin installation, examine the INFORMATION_SCHEMA.PLUGINS table or use the SHOW PLUGINS statement (see Obtaining Server Plugin Information). For example:

```sql
mysql> SELECT PLUGIN_NAME, PLUGIN_STATUS
FROM INFORMATION_SCHEMA.PLUGINS
WHERE PLUGIN_NAME LIKE '%semi%';
```

If the 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 master side and the slave side to enable semisynchronous replication. If only one side is enabled, replication will be asynchronous.

To control whether an installed plugin is enabled, set the appropriate system variables. You can set these variables at runtime using SET GLOBAL, or at server startup on the command line or in an option file.

At runtime, these master-side system variables are available:

```sql
SET GLOBAL rpl_semi_sync_master_enabled = {0|1};
SET GLOBAL rpl_semi_sync_master_timeout = N;
```

On the slave side, this system variable is available:

```sql
SET GLOBAL rpl_semi_sync_slave_enabled = {0|1};
```

For `rpl_semi_sync_master_enabled` or `rpl_semi_sync_slave_enabled`, the value should be 1 to enable semisynchronous replication or 0 to disable it. By default, these variables are set to 0.

For `rpl_semi_sync_master_timeout`, the value \(N\) is given in milliseconds. The default value is 10000 (10 seconds).

If you enable semisynchronous replication on a slave at runtime, you must also start the slave I/O thread (stopping it first if it is already running) to cause the slave to connect to the master and register as a semisynchronous slave:

```sql
STOP SLAVE IO_THREAD;
START SLAVE IO_THREAD;
```

If the I/O thread is already running and you do not restart it, the slave continues to use asynchronous replication.

At server startup, the variables that control semisynchronous replication can be set as command-line options or in an option file. 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 master and slave sides as follows.

On the master:

```ini
[mysqld]
```

On the slave:

```ini
[mysqld]
```
3.9.3 Semisynchronous Replication Monitoring

The plugins for the semisynchronous replication capability expose several system and status variables that you can examine to determine its configuration and operational state.

The system variable reflect how semisynchronous replication is configured. To check their values, use `SHOW VARIABLES`:

```sql
mysql> SHOW VARIABLES LIKE 'rpl_semi_sync%';
```

The status variables enable you to monitor the operation of semisynchronous replication. To check their values, use `SHOW STATUS`:

```sql
mysql> SHOW STATUS LIKE 'Rpl_semi_sync%';
```

When the master switches between asynchronous or semisynchronous replication due to commit-blocking timeout or a slave catching up, it sets the value of the `Rpl_semi_sync_master_status` status variable appropriately. Automatic fallback from semisynchronous to asynchronous replication on the master means that it is possible for the `rpl_semi_sync_master_enabled` system variable to have a value of 1 on the master side even when semisynchronous replication is in fact not operational at the moment. You can monitor the `Rpl_semi_sync_master_status` status variable to determine whether the master currently is using asynchronous or semisynchronous replication.

To see how many semisynchronous slaves are connected, check `Rpl_semi_sync_master_clients`.

The number of commits that have been acknowledged successfully or unsuccessfully by slaves are indicated by the `Rpl_semi_sync_master_yes_tx` and `Rpl_semi_sync_master_no_tx` variables.

On the slave side, `Rpl_semi_sync_slave_status` indicates whether semisynchronous replication currently is operational.

3.10 Delayed Replication

MySQL 5.7 supports delayed replication such that a slave server deliberately lags behind the master by at least a specified amount of time. The default delay is 0 seconds. Use the `MASTER_DELAY` option for `CHANGE MASTER TO` to set the delay to \( N \) seconds:

```sql
CHANGE MASTER TO MASTER_DELAY = N;
```

An event received from the master is not executed until at least \( N \) seconds later than its execution on the master. The exceptions are that there is no delay for format description events or log file rotation events, which affect only the internal state of the SQL thread.

Delayed replication can be used for several purposes:

- To protect against user mistakes on the master. A DBA can roll back a delayed slave to the time just before the disaster.

- 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 slave. 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 slave.
To inspect what the database looked like long ago, without having to reload a backup. For example, if the delay is one week and the DBA needs to see what the database looked like before the last few days' worth of development, the delayed slave can be inspected.

`START SLAVE` and `STOP SLAVE` take effect immediately and ignore any delay. `RESET SLAVE` resets the delay to 0.

`SHOW SLAVE STATUS` has three fields that provide information about the delay:

- **SQL_Delay**: A nonnegative integer indicating the number of seconds that the slave must lag the master.

- **SQL_Remaining_Delay**: When `Slave_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`.

- **Slave_SQL_Running_State**: A string indicating the state of the SQL thread (analogous to `Slave_IO_State`). The value is identical to the `State` value of the SQL thread as displayed by `SHOW PROCESSLIST`.

When the slave 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`. 
Chapter 4 Replication Notes and Tips

Table of Contents

4.1 Replication Features and Issues .................................................................................. 149
  4.1.1 Replication and AUTO_INCREMENT .................................................................... 150
  4.1.2 Replication and BLACKHOLE Tables .................................................................. 151
  4.1.3 Replication and Character Sets .......................................................................... 151
  4.1.4 Replication and CHECKSUM TABLE .................................................................. 151
  4.1.5 Replication of CREATE ... IF NOT EXISTS Statements .................................... 151
  4.1.6 Replication of CREATE TABLE ... SELECT Statements ..................................... 151
  4.1.7 Replication of CREATE SERVER, ALTER SERVER, and DROP SERVER .............. 152
  4.1.8 Replication of CURRENT_USER() ...................................................................... 152
  4.1.9 Replication of DROP ... IF EXISTS Statements ................................................ 153
  4.1.10 Replication with Differing Table Definitions on Master and Slave .................... 153
  4.1.11 Replication and DIRECTORY Table Options ..................................................... 157
  4.1.12 Replication and Floating-Point Values .............................................................. 157
  4.1.13 Replication and Fractional Seconds Support ..................................................... 157
  4.1.14 Replication and FLUSH ..................................................................................... 158
  4.1.15 Replication and System Functions .................................................................... 158
  4.1.16 Replication of Invoked Features ...................................................................... 160
  4.1.17 Replication and LIMIT ..................................................................................... 162
  4.1.18 Replication and LOAD DATA .......................................................................... 162
  4.1.19 Replication and max_allowed_packet .............................................................. 162
  4.1.20 Replication and MEMORY Tables .................................................................... 162
  4.1.21 Replication of the mysql System Database ..................................................... 163
  4.1.22 Replication and the Query Optimizer .............................................................. 163
  4.1.23 Replication and Partitioning ............................................................................ 163
  4.1.24 Replication and REPAIR TABLE ..................................................................... 164
  4.1.25 Replication and Reserved Words ..................................................................... 164
  4.1.26 Replication and Master or Slave Shutdowns ...................................................... 164
  4.1.27 Slave Errors During Replication ..................................................................... 165
  4.1.28 Replication and Server SQL Mode .................................................................... 166
  4.1.29 Replication and Temporary Tables ................................................................... 166
  4.1.30 Replication Retries and Timeouts .................................................................... 166
  4.1.31 Replication and Time Zones ............................................................................ 167
  4.1.32 Replication and Transaction Inconsistencies .................................................... 167
  4.1.33 Replication and Transactions .......................................................................... 169
  4.1.34 Replication and Triggers .................................................................................. 170
  4.1.35 Replication and TRUNCATE TABLE .................................................................. 171
  4.1.36 Replication and User Name Length ................................................................... 171
  4.1.37 Replication and Variables .............................................................................. 172
  4.1.38 Replication and Views ..................................................................................... 173

4.2 Replication Compatibility Between MySQL Versions .................................................. 173
4.3 Upgrading a Replication Setup ...................................................................................... 174
4.4 Troubleshooting Replication .......................................................................................... 176
4.5 How to Report Replication Bugs or Problems ................................................................. 177

4.1 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 master and slave. In other words, successful statement-based replication requires that any SQL features used
be supported by both the master and the slave servers. If you use a feature on the master server that is available only in the current version of MySQL, you cannot replicate to a slave 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 5.7 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 Stored Program Binary Logging. For more information about row-based logging and row-based replication, see Binary Logging Formats, and Section 5.1, “Replication Formats”.

For additional information specific to replication and InnoDB, see InnoDB and MySQL Replication. For information relating to replication with NDB Cluster, see NDB Cluster Replication.

### 4.1.1 Replication and AUTO_INCREMENT

Statement-based replication of AUTO_INCREMENT, LAST_INSERT_ID(), and TIMESTAMP values is done correctly, subject to the following exceptions:

- When using statement-based replication prior to MySQL 5.7.1, AUTO_INCREMENT columns in tables on the slave must match the same columns on the master; that is, AUTO_INCREMENT columns must be replicated to AUTO_INCREMENT columns.

- 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 slave and the master. 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 master and slave, 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:

```sql
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 master and slave, 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 Problems with ALTER TABLE.

4.1.2 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 (Bug #13004581).

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.

4.1.3 Replication and Character Sets

The following applies to replication between MySQL servers that use different character sets:

- If the master 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.

4.1.4 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.

4.1.5 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 master.

- Similarly, every CREATE TABLE IF NOT EXISTS statement without a SELECT is replicated, whether or not the table already exists on the master. This includes CREATE TABLE IF NOT EXISTS ... LIKE. Replication of CREATE TABLE IF NOT EXISTS ... SELECT follows somewhat different rules; see Section 4.1.6, “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 master.

4.1.6 Replication of CREATE TABLE ... SELECT Statements

This section discusses how MySQL replicates CREATE TABLE ... SELECT statements.

MySQL 5.7 does not allow a CREATE TABLE ... SELECT statement to make any changes in tables other than the table that is created by the statement. Some older versions of MySQL permitted these statements to do so; this means that, when using statement-based replication between a MySQL 5.6 or later slave and a master running a previous version of MySQL, a CREATE TABLE ... SELECT statement causing changes in other tables on the master fails on the slave, causing replication to stop. To prevent this from happening, you should use row-based replication, rewrite the offending statement
before running it on the master, or upgrade the master to MySQL 5.7. (If you choose to upgrade the master, keep in mind that such a CREATE TABLE ... SELECT statement fails following the upgrade unless it is rewritten to remove any side effects on other tables.) This is not an issue when using row-based replication, because the statement is logged as a CREATE TABLE statement with any changes to table data logged as row-insert events, rather than as the entire CREATE TABLE ... SELECT.

These behaviors are not dependent on MySQL version:

• CREATE TABLE ... SELECT always performs an implicit commit (Statements That Cause an Implicit Commit).

• If 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.

• If the statement fails, nothing is logged. This includes the case that the destination table exists and IF NOT EXISTS is not given.

When the destination table exists and IF NOT EXISTS is given, MySQL 5.7 ignores the statement completely; nothing is inserted or logged.

**4.1.7 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.

**4.1.8 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 master and the slave 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 master is replicated to the slave as metadata, and the slave applies the statement to the current user named in the metadata, rather than to the current user on the slave.

4.1.9 Replication of DROP ... IF EXISTS Statements

The \texttt{DROP DATABASE IF EXISTS}, \texttt{DROP TABLE IF EXISTS}, and \texttt{DROP VIEW IF EXISTS} statements are always replicated, even if the database, table, or view to be dropped does not exist on the master. This is to ensure that the object to be dropped no longer exists on either the master or the slave, once the slave has caught up with the master.

\texttt{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 master.

4.1.10 Replication with Differing Table Definitions on Master and Slave

Source and target tables for replication do not have to be identical. A table on the master can have more or fewer columns than the slave's copy of the table. In addition, corresponding table columns on the master and the slave can use different data types, subject to certain conditions.

\textbf{Note}

Replication between tables which are partitioned differently from one another is not supported. See Section 4.1.23, "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 master and the slave. Additional conditions are discussed, with examples, in the following two sections.

4.1.10.1 Replication with More Columns on Master or Slave

You can replicate a table from the master to the slave such that the master and slave 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 master and the slave.

  (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 \texttt{ALTER TABLE} statement on the slave 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 \texttt{t}, existing on the master and the slave, is defined by the following \texttt{CREATE TABLE} statement:

```sql
CREATE TABLE t {
  c1 INT,
  c2 INT,
  c3 INT
};
```

Suppose that the \texttt{ALTER TABLE} statement shown here is executed on the slave:

```sql
ALTER TABLE t ADD COLUMN cnew1 INT AFTER c3;
```

The previous \texttt{ALTER TABLE} is permitted on the slave because the columns \texttt{c1}, \texttt{c2}, and \texttt{c3} that are common to both versions of table \texttt{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 slave without causing replication to break:

```sql
ALTER TABLE t ADD COLUMN cnew2 INT AFTER c2;
```

Replication fails after execution on the slave 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 Data Type Default Values).

In addition, when the slave’s copy of the table has more columns than the master’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 master.** The following table definitions are valid and replicate correctly:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
slave> 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 slave than they are on the master:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
slave> 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 master appears before the definitions of the columns common to both versions of the table:

```sql
master> CREATE TABLE t1 (c3 INT, c1 INT, c2 INT);
slave> CREATE TABLE t1 (c1 INT, c2 INT);
```

**More columns on the slave.** The following table definitions are valid and replicate correctly:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT);
slave> 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 master and the slave:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT);
slave> CREATE TABLE t1 (c2 INT, c1 INT);
```

The following table definitions also raise an error because the definition for the extra column in the slave’s version of the table appears before the definitions for the columns which are common to both versions of the table:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT);
slave> CREATE TABLE t1 (c3 INT, c1 INT, c2 INT);
```

The following table definitions fail because the slave’s version of the table has additional columns compared to the master’s version, and the two versions of the table use different data types for the common column `c2`:

```sql
master> CREATE TABLE t1 (c1 INT, c2 INT);
slave> CREATE TABLE t1 (c1 INT, c2 INT, c3 INT);
```

### 4.1.10.2 Replication of Columns Having Different Data Types
Corresponding columns on the master's and the slave'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 is also possible to replicate from a column having one data type (on the master) to a column having a different data type (on the slave); when the data type of the master's version of the column is promoted to a type that is the same size or larger on the slave, 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 master or the slave. 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 slave's copy of the table cannot contain more columns than the master'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 master would also execute successfully on the slave, 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 slave, 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 slave's copy of a table to `BIGINT`, any insert into that column on the master that succeeds should also succeed on the slave, since it is impossible to have a legal `TINYINT` value that is large enough to exceed a `BIGINT` column.

Prior to MySQL 5.7.1, when using statement-based replication, `AUTO_INCREMENT` columns were required to be the same on both the master and the slave; otherwise, updates could be applied to the wrong table on the slave. (Bug #12669186)

**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 (slave_type_conversions variable).** The setting of the `slave_type_conversions` global server variable controls the type conversion mode used on the slave. This variable takes a set of values from the following table, which shows the effects of each mode on the slave's type-conversion behavior:

<table>
<thead>
<tr>
<th>Mode</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALL_LOSSY</td>
<td>In this mode, type conversions that would mean loss of information are permitted.</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
</tbody>
</table>
Replication with Differing Table Definitions on Master and Slave

<table>
<thead>
<tr>
<th>Mode</th>
<th>Effect</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>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 slave.</td>
</tr>
<tr>
<td>ALL_NON_LOSSY</td>
<td>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 slave to stop with an error.</td>
</tr>
<tr>
<td>ALL_LOSSY, ALL_NON_LOSSY</td>
<td>When this mode is set, all supported type conversions are permitted, whether or not they are lossy conversions.</td>
</tr>
<tr>
<td>ALL_SIGNED</td>
<td>Treat promoted integer types as signed values (the default behavior).</td>
</tr>
<tr>
<td>ALL_UNSIGNED</td>
<td>Treat promoted integer types as unsigned values.</td>
</tr>
<tr>
<td>ALL_SIGNED, ALL_UNSIGNED</td>
<td>Treat promoted integer types as signed if possible, otherwise as unsigned.</td>
</tr>
<tr>
<td>[empty]</td>
<td>When 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.</td>
</tr>
</tbody>
</table>

When an integer type is promoted, its signedness is not preserved. By default, the slave treats all such values as signed. Beginning with MySQL 5.7.2, you can control this behavior using ALL_SIGNED, ALL_UNSIGNED, or both. (Bug#15831300) ALL_SIGNED tells the slave to treat all promoted integer types as signed; ALL_UNSIGNED instructs it to treat these as unsigned. Specifying both causes the slave 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 slave with the new 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 \(D' \geq D\) and \((M' - D') \geq (M - D)\) 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 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 slave, 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 slave, 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.

### 4.1.11 Replication and DIRECTORY Table Options

If a DATA DIRECTORY or INDEX DIRECTORY table option is used in a CREATE TABLE statement on the master server, the table option is also used on the slave. This can cause problems if no corresponding directory exists in the slave host file system or if it exists but is not accessible to the slave server. This can be overridden by using the NO_DIR_IN_CREATE server SQL mode on the slave, which causes the slave 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 Server SQL Modes.

### 4.1.12 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 master and slave servers due to differences in computer architecture, the compiler used to build MySQL, and so forth. See Type Conversion in Expression Evaluation, and Problems with Floating-Point Values.

### 4.1.13 Replication and Fractional Seconds Support

MySQL 5.7 permits fractional seconds for TIME, DATETIME, and TIMESTAMP values, with up to microseconds (6 digits) precision. See Fractional Seconds in Time Values.
There may be problems replicating from a master server that understands fractional seconds to an older slave (MySQL 5.6.3 and earlier) that does not:

- For `CREATE TABLE` statements containing columns that have an `fsp` ( fractional seconds precision) value greater than 0, replication will fail due to parser errors.

- Statements that use temporal data types with an `fsp` value of 0 will work for with statement-based logging but not row-based logging. In the latter case, the data types have binary formats and type codes on the master that differ from those on the slave.

- Some expression results will differ on master and slave. Examples: On the master, the `timestamp` system variable returns a value that includes a microseconds fractional part; on the slave, it returns an integer. On the master, functions that return a result that includes the current time (such as `CURTIME()`, `SYSDATE()`, or `UTC_TIMESTAMP()`) interpret an argument as an `fsp` value and the return value includes a fractional seconds part of that many digits. On the slave, these functions permit an argument but ignore it.

### 4.1.14 Replication and FLUSH

Some forms of the `FLUSH` statement are not logged because they could cause problems if replicated to a slave: `FLUSH LOGS` and `FLUSH TABLES WITH READ LOCK`. For a syntax example, see `FLUSH Statement`. The `FLUSH TABLES`, `ANALYZE TABLE`, `OPTIMIZE TABLE`, and `REPAIR TABLE` statements are written to the binary log and thus replicated to slaves. 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 slaves 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 slaves. These statements are written to the binary log unless you specify `NO_WRITE_TO_BINLOG` or its alias `LOCAL`.

### 4.1.15 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 slave unless row-based replication is enabled. (See Section 5.1, “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 4.1.8, “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 master is replicated to the slave. To avoid unexpected results when replicating between MySQL servers in different time zones, set the time zone on both master and slave. For more information, see Section 4.1.31, “Replication and Time Zones”.

To explain the potential problems when replicating between servers which are in different time zones, suppose that the master is located in New York, the slave is located in Stockholm, and both servers are using local time. Suppose further that, on the master, you create a table `mytable`, perform an `INSERT` statement on this table, and then select from the table, as shown here:

```sql
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               |
| 158                 |
```
Replication and System Functions

Local time in Stockholm is 6 hours later than in New York; so, if you issue `SELECT NOW()` on the slave at that exact same instant, the value `2009-09-01 18:00:00` is returned. For this reason, if you select from the slave'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 slave's copy of `mytable`, you obtain exactly the same result as on the master:

```
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 master and the slave 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 master and the slave.

`SYSDATE()` is automatically replicated using row-based replication when using `MIXED` mode, and generates a warning in `STATEMENT` mode.

See also `Section 4.1.31, “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 slave knowing the concurrency context on the master. Therefore, these functions should not be used to insert into a master table because the content on the slave 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 master and slave.)

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 slave 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)

Prior to MySQL 5.7.3, the value of `LAST_INSERT_ID()` was not replicated correctly if any filtering options such as `--replicate-ignore-db` and `--replicate-do-table` were enabled on the slave. (Bug #17234370, BUG# 69861)

### 4.1.16 Replication of Invoked Features

Replication of invoked features such as user-defined functions (UDFs) 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 UDF with statement-based replication generates this warning because it currently cannot be determined by the MySQL server whether the UDF 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 slave regardless of the state specified (this does not apply to `DROP EVENT`).
  - The master on which the event was created is identified on the slave by its server ID. The `ORIGINATOR` column in `INFORMATION_SCHEMA.EVENTS` and the `originator` column in `mysql.event` store this information. See the `INFORMATION_SCHEMA(EVENTS)` table, and the `SHOW EVENTS` statement, for more information.
  - The feature implementation resides on the slave in a renewable state so that if the master fails, the slave can be used as the master 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 replication master), query the `INFORMATION_SCHEMA.EVENTS` table in a manner similar to what is shown here:

```sql
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 replication slave having such events to a replication master, you must enable each event using `ALTER EVENT event_name ENABLE`, where `event_name` is the name of the event.

If more than one master was involved in creating events on this slave, and you wish to identify events that were created only on a given master having the server ID `master_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 = 'master_id';
```

You can employ `ORIGINATOR` with the `SHOW EVENTS` statement in a similar fashion:

```
SHOW EVENTS
WHERE STATUS = 'SLAVESIDE_DISABLED'
AND ORIGINATOR = 'master_id';
```

Before enabling events that were replicated from the master, you should disable the MySQL Event Scheduler on the slave (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 slave afterward (using a statement such as `SET GLOBAL event_scheduler = ON;`).

If you later demote the new master back to being a replication slave, 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 replication slave, you can identify the events by querying the `EVENTS` table, as shown here:

```sql
SELECT CONCAT(EVENT_SCHEMA, '.', EVENT_NAME) AS 'Db.Event'
FROM INFORMATION_SCHEMA.EVENTS
WHERE INSTR(EVENT_NAME, 'replicated_') = 1;
```

### 4.1.17 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.

### 4.1.18 Replication and LOAD DATA

`LOAD DATA` is considered unsafe for statement-based logging (see Section 5.1.3, “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.

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 `mysqlbinlog — Utility for Processing Binary Log Files`.

### 4.1.19 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 replication slaves. 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 master, the master fails with an error, and the slave shuts down the I/O thread. If `max_allowed_packet` is too small on the slave, this also causes the slave to stop the I/O thread.

Row-based replication currently sends all columns and column values for updated rows from the master to the slave, 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.

### 4.1.20 Replication and MEMORY Tables

When a master server shuts down and restarts, its `MEMORY` tables become empty. To replicate this effect to slaves, the first time that the master uses a given `MEMORY` table after startup, it logs an event that notifies slaves that the table must be emptied by writing a `DELETE` 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.

When a slave server shuts down and restarts, its `MEMORY` tables become empty. This causes the slave to be out of synchrony with the master and may lead to other failures or cause the slave to stop.
4.1.21 Replication of the mysql System Database

Data modification statements made to tables in the mysql database 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 as GRANT, REVOKE, and statements manipulating triggers, stored routines, and views—are replicated to slaves using statement-based replication.

4.1.22 Replication and the Query Optimizer

It is possible for the data on the master and slave 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 4.1.17, “Replication and LIMIT”, for a detailed discussion of these.

4.1.23 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 4.1.10, “Replication with Differing Table Definitions on Master and Slave”).

Replication between tables having different partitioning is generally not supported. This because statements (such as ALTER TABLE ... DROP PARTITION) acting directly on partitions in such cases may produce different results on master and slave. In the case where a table is partitioned on the master but not on the slave, any statements operating on partitions on the master’s copy of the
slave fail on the slave. When the slave's copy of the table is partitioned but the master's copy is not, statements acting on partitions cannot be run on the master without causing errors there.

Due to these dangers of causing replication to fail entirely (on account of failed statements) and of inconsistencies (when the result of a partition-level SQL statement produces different results on master and slave), we recommend that insure that the partitioning of any tables to be replicated from the master is matched by the slave's versions of these tables.

4.1.24 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 master and slave to lose synchronization. For this reason, in the event that a table on the master 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 master's and slave's copies of the table and be prepared to correct any discrepancies manually, before restarting replication.

4.1.25 Replication and Reserved Words

You can encounter problems when you attempt to replicate from an older master to a newer slave and you make use of identifiers on the master that are reserved words in the newer MySQL version running on the slave. An example of this is using a table column named virtual on a 5.6 master that is replicating to a 5.7 or higher slave because VIRTUAL is a reserved word beginning in MySQL 5.7. 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 slave prior to execution, so that the slave knows which database object or objects would be affected; only after the event is parsed can the slave 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 master which would be regarded as reserved words by the slave, do one of the following:

- Use one or more ALTER TABLE statements on the master to change the names of any database objects where these names would be considered reserved words on the slave, 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 Reserved Words, in the MySQL Server Version Reference. For identifier quoting rules, see Schema Object Names.

4.1.26 Replication and Master or Slave Shutdowns

It is safe to shut down a master server and restart it later. When a slave loses its connection to the master, the slave 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 MASTER TO statement. A slave also is able to deal with network connectivity outages. However, the slave notices the network outage only after receiving no data from the master for slave_net_timeout seconds. If your outages are short, you may want to decrease slave_net_timeout. See Section 3.2, “Handling an Unexpected Halt of a Replication Slave”.

An unclean shutdown (for example, a crash) on the master side can result in the master binary log having a final position less than the most recent position read by the slave, due to the master binary log file not being flushed. This can cause the slave not to be able to replicate when the master comes back up. Setting sync_binlog=1 in the master my.cnf file helps to minimize this problem because it causes the master 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 slave cleanly is safe because it keeps track of where it left off. However, be careful that the slave does not have temporary tables open; see Section 4.1.29, “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 slave commits and then updates `relay-log.info`. If a crash occurs between these two operations, relay log processing will have proceeded further than the information file indicates and the slave will re-execute the events from the last transaction in the relay log after it has been restarted.

- A similar problem can occur if the slave 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 slave `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.

4.1.27 Slave Errors During Replication

If a statement produces the same error (identical error code) on both the master and the slave, the error is logged, but replication continues.

If a statement produces different errors on the master and the slave, the slave SQL thread terminates, and the slave 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 master or the slave, but not both. To address the issue, connect to the slave manually and determine the cause of the problem. `SHOW SLAVE STATUS` is useful for this. Then fix the problem and run `START SLAVE`. For example, you might need to create a nonexistent table before you can start the slave again.

**Note**

If a temporary error is recorded in the slave’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 slave SQL thread records a temporary error relating to a deadlock, you do not need to restart the transaction manually on the slave, unless the slave 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 master, the slave expects execution of the statement to result in the same error code. If it does not, the slave SQL thread stops as described previously.

If you are replicating between tables that use different storage engines on the master and slave, 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 master might cause a foreign key violation but the same statement performed
on a **MyISAM** version of the same table on the slave would produce no such error, causing replication to stop.

### 4.1.28 Replication and Server SQL Mode

Using different server SQL mode settings on the master and the slave may cause the same **INSERT** statements to be handled differently on the master and the slave, leading the master and slave to diverge. For best results, you should always use the same server SQL mode on the master and on the slave. 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 master and the slave is likely to cause issues. At a minimum, this is likely to cause the distribution of data among partitions to be different in the master’s and slave’s copies of a given table. It may also cause inserts into partitioned tables that succeed on the master to fail on the slave.

For more information, see [Server SQL Modes](#). In particular, see [SQL Mode Changes in MySQL 5.7](#), which describes changes in MySQL 5.7, so that you can assess whether your applications will be affected.

### 4.1.29 Replication and Temporary Tables

The discussion in the following paragraphs does not apply when **binlog_format=ROW** because, in that case, temporary tables are not replicated; this means that there are never any temporary tables on the slave to be lost in the event of an unplanned shutdown by the slave. The remainder of this section applies only when using statement-based or mixed-format replication. Loss of replicated temporary tables on the slave can be an issue, whenever **binlog_format** is **STATEMENT** or **MIXED**, for statements involving temporary tables that can be logged safely using statement-based format.

For more information about row-based replication and temporary tables, see [Row-based logging of temporary tables](#).

**Safe slave shutdown when using temporary tables.** Temporary tables are replicated except in the case where you stop the slave server (not just the slave threads) and you have replicated temporary tables that are open for use in updates that have not yet been executed on the slave. If you stop the slave server, the temporary tables needed by those updates are no longer available when the slave is restarted. To avoid this problem, do not shut down the slave while it has temporary tables open. Instead, use the following procedure:

1. Issue a **STOP SLAVE SQL_THREAD** statement.
2. Use **SHOW STATUS** to check the value of the **Slave_open_temp_tables** variable.
3. If the value is not 0, restart the slave SQL thread with **START SLAVE SQL_THREAD** and repeat the procedure later.
4. When the value is 0, issue a **mysqladmin shutdown** command to stop the slave.

**Temporary tables and replication options.** By default, 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 replication slave 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 or mixed-format 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.

### 4.1.30 Replication Retries and Timeouts
The global system variable `slave_transaction_retries` affects replication as follows: If the slave SQL thread fails to execute a transaction because of an InnoDB deadlock or because it exceeded the InnoDB `innodb_lock_wait_timeout` value, or the NDB `TransactionDeadlockDetectionTimeout` or `TransactionInactiveTimeout` value, the slave automatically retries the transaction `slave_transaction_retries` times before stopping with an error. The default value is 10. The total retry count can be seen in the output of `SHOW STATUS`; see `Server Status Variables`.

### 4.1.31 Replication and Time Zones

By default, master and slave 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 master and slave. Otherwise, statements depending on the local time on the master 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 master and slave 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 master and slave to apply different time zones. For example, a master could write the following statement in the binary log:

```
SET @session.time_zone='SYSTEM';
```

If this master and its slave have a different setting for their system time zones, this statement can produce unexpected results on the slave, even if the slave's global `time_zone` value has been set to match the master's. For an explanation of MySQL Server's time zone settings, and how to change them, see MySQL Server Time Zone Support.

See also Section 4.1.15, “Replication and System Functions”.

### 4.1.32 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 is a transaction that has not been fully applied, even though some transaction later in the sequence has been applied. Gaps can only appear when using a multithreaded slave. To avoid gaps occurring, set `slave_preserve_commit_order=1`, which requires `slave_parallel_type=LOGICAL_CLOCK`, and that `log-bin` and `log-slave-updates` are also enabled. Note that `slave_preserve_commit_order=1` does not preserve the order of non-transactional DML updates, so these might commit before transactions that precede them in the relay log, which might result in gaps.

- **Master 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 slaves. Enabling `slave_preserve_commit_order` does not prevent master log position lag.

The following scenarios are relevant to the existence of half-applied transactions, gaps, and master log position lag:
Replication and Transaction Inconsistencies

1. While slave 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 slave, the coordinator thread when using a multithreaded slave). 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 slave, workers which have not received an error complete their queues, so it may take time to stop all threads.

5. `STOP SLAVE` when using a multithreaded slave. After issuing `STOP SLAVE`, the slave waits for any gaps to be filled and then updates `Exec_master_log_pos`. This ensures it never leaves gaps or master log position lag, unless any of the cases above applies, in other words, before `STOP SLAVE` completes, either an error happens, or another thread issues `KILL`, or the server restarts. In these cases, `STOP SLAVE` returns successfully.

6. If the last transaction in the relay log is only half-received and the multithreaded slave coordinator has started to schedule the transaction to a worker, then `STOP SLAVE` 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 SLAVE` when using a single-threaded slave. If the ongoing transaction only updates transactional tables, it is rolled back and `STOP SLAVE` stops immediately. If the ongoing transaction is mixed, `STOP SLAVE` 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 variable `rpl_stop_slave_timeout` is unrelated to the process of stopping the replication threads. It only makes the client that issues `STOP SLAVE` 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 slave database is in a state that may never have existed on the master.

2. The field `Exec_master_log_pos` in `SHOW SLAVE 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 MASTER TO` statements for that channel fail with an error, unless the applier threads are running and the `CHANGE MASTER TO` 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-slave`, it does not record the existence of gaps; thus it prints `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-slave`).

If a replication channel has master log position lag but no gaps, cases 2 to 5 above apply, but case 1 does not.

The master log position information is persisted in binary format in the internal table `mysql.slave_worker_info`. `START SLAVE` (SQL THREAD) always consults this information so that it applies only the correct transactions. This remains true even if `slave_parallel_workers` has been changed to 0 before `START SLAVE`, and even if `START SLAVE` is used with `UNTIL` clauses.
START_SLAVE UNTIL SQL_AFTER_MTS_GAPS only applies as many transactions as needed in order to fill in the gaps. If START_SLAVE is used with UNTIL clauses that tell it to stop before it has consumed all the gaps, then it leaves remaining gaps.

**Warning**

RESET_SLAVE removes the relay logs and resets the replication position. Thus issuing RESET_SLAVE on a slave with gaps means the slave loses any information about the gaps, without correcting the gaps.

### 4.1.33 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. (In some past MySQL versions, only a statement that updated both nontransactional and transactional tables was considered mixed.) 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 5.1.3, “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 master and slave.** It is possible to replicate transactional tables on the master using nontransactional tables on the slave. For example, you can replicate an InnoDB master table as a MyISAM slave table. However, if you do this, there are problems if the slave is stopped in the middle of a BEGIN ... COMMIT block because the slave restarts at the beginning of the BEGIN block.

It is also safe to replicate transactions from MyISAM tables on the master to transactional tables—such as tables that use the InnoDB storage engine—on the slave. In such cases, an AUTOCOMMIT=1 statement issued on the master is replicated, thus enforcing AUTOCOMMIT mode on the slave.

When the storage engine type of the slave is nontransactional, transactions on the master that mix updates of transactional and nontransactional tables should be avoided because they can cause inconsistency of the data between the master transactional table and the slave nontransactional table. That is, such transactions can lead to master storage engine-specific behavior with the possible effect of replication going out of synchrony. MySQL does not issue a warning about this currently, so extra care should be taken when replicating transactional tables from the master to nontransactional tables on the slaves.

**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 Restrictions on XA Transactions.

**4.1.34 Replication and Triggers**

With statement-based replication, triggers executed on the master also execute on the slave. With row-based replication, triggers executed on the master do not execute on the slave. Instead, the row changes on the master resulting from trigger execution are replicated and applied on the slave.

This behavior is by design. If under row-based replication the slave applied the triggers as well as the row changes caused by them, the changes would in effect be applied twice on the slave, leading to different data on the master and the slave.

If you want triggers to execute on both the master and the slave—perhaps because you have different triggers on the master and slave—you must use statement-based replication. However, to enable slave-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 5.7 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), but before MySQL 5.7.2 cannot have multiple triggers that have the
same trigger event and action time. MySQL 5.7.2 lifts this limitation and multiple triggers are permitted. This change has replication implications for upgrades and downgrades.

For brevity, “multiple triggers” here is shorthand for “multiple triggers that have the same trigger event and action time.”

**Upgrades.** Suppose that you upgrade an old server that does not support multiple triggers to MySQL 5.7.2 or higher. If the new server is a replication master and has old slaves that do not support multiple triggers, an error occurs on those slaves if a trigger is created on the master for a table that already has a trigger with the same trigger event and action time. To avoid this problem, upgrade the slaves first, then upgrade the master.

**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 remain 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.

### 4.1.35 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 replication slaves still follow the rules described in the `TRUNCATE TABLE Statement` governing such tables. (Bug #36763)

### 4.1.36 Replication and User Name Length

The maximum length of MySQL user names was increased from 16 characters to 32 characters in MySQL 5.7.8. Replication of user names longer than 16 characters to a slave that supports only shorter user names will fail. However, this should occur only when replicating from a newer master to an older slave, which is not a recommended configuration.
4.1.37 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 Mixed Binary Logging Format.

`sql_mode` is also replicated except for the `NO_DIR_IN_CREATE` mode; the slave always preserves its own value for `NO_DIR_IN_CREATE`, regardless of changes to it on the master. 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` and `storage_engine` system variables are 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 master without doing so on the slave can lead eventually to `Table is full` errors on the slave when trying to execute `INSERT` statements on a `MEMORY` table on the master that is thus permitted to
Replication and Views

grow larger than its counterpart on the slave. For more information, see Section 4.1.20, "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 will not insert the same data on the master and the slave:

```sql
SET max_join_size=1000;
INSERT INTO mytable VALUES(@max_join_size);
```

This does not apply to the common sequence:

```sql
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 5.1, “Replication Formats”.

The following session variables are written to the binary log and honored by the replication slave 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 master and slave, especially when you are running MySQL on platforms with case-sensitive file systems.

### 4.1.38 Replication and Views

Views are always replicated to slaves. Views are filtered by their own name, not by the tables they refer to. This means that a view can be replicated to the slave 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. (Bug #11752707, Bug #43975)

### 4.2 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 master running MySQL 5.6 to a slave running MySQL 5.7, from a master...
Upgrading a Replication Setup

running MySQL 5.7 to a slave running MySQL 8.0, and so on. However, you may encounter difficulties when replicating from an older master to a newer slave if the master uses statements or relies on behavior no longer supported in the version of MySQL used on the slave. 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 masters, regardless of the number of master or slave 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 5.7.22, MySQL 5.7.23, and MySQL 5.7.24 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 masters and slaves that use early releases of a release series of MySQL to GA (production) releases when the latter become available for that release series.

Replication from newer masters to older slaves may 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.

  This also has significant implications for upgrading replication servers; see Section 4.3, "Upgrading a Replication Setup" for more information.

- For more information about row-based replication, see Section 5.1, "Replication Formats".

- **SQL incompatibilities.** You cannot replicate from a newer master to an older slave using statement-based replication if the statements to be replicated use SQL features available on the master but not on the slave.

  However, if both the master and the slave support row-based replication, and there are no data definition statements to be replicated that depend on SQL features found on the master but not on the slave, you can use row-based replication to replicate the effects of data modification statements even if the DDL run on the master is not supported on the slave.

  For more information on potential replication issues, see Section 4.1, "Replication Features and Issues".

4.3 Upgrading a Replication Setup

When you upgrade servers that participate in a replication setup, the procedure for upgrading depends on the current server versions and the version to which you are upgrading. This section provides information about how upgrading affects replication. For general information about upgrading MySQL, see Upgrading MySQL.

When you upgrade a master to 5.7 from an earlier MySQL release series, you should first ensure that all the slaves of this master are using the same 5.7.x release. If this is not the case, you should first upgrade the slaves. To upgrade each slave, shut it down, upgrade it to the appropriate 5.7.x version, restart it, and restart replication. Relay logs created by the slave after the upgrade are in 5.7 format.

Changes affecting operations in strict SQL mode (**STRICT_TRANS_TABLES** or **STRICT_ALL_TABLES**) may result in replication failure on an updated slave. For example, as of MySQL 5.7.2, the server restricts insertion of a **DEFAULT** value of 0 for temporal data types in strict mode. If you use statement-based logging (**binlog_format=STATEMENT**), if a slave is upgraded before the master, the nonupgraded master will execute statements without error that may fail on the slave and replication will...
stop. To deal with this, stop all new statements on the master and wait until the slaves catch up. Then upgrade the slaves. Alternatively, if you cannot stop new statements, temporarily change to row-based logging on the master (binlog_format=ROW) and wait until all slaves have processed all binary logs produced up to the point of this change. Then upgrade the slaves.

After the slaves have been upgraded, shut down the master, upgrade it to the same 5.7.x release as the slaves, and restart it. If you had temporarily changed the master to row-based logging, change it back to statement-based logging. The 5.7 master is able to read the old binary logs written prior to the upgrade and to send them to the 5.7 slaves. The slaves recognize the old format and handle it properly. Binary logs created by the master subsequent to the upgrade are in 5.7 format. These too are recognized by the 5.7 slaves.

In other words, when upgrading to MySQL 5.7, the slaves must be MySQL 5.7 before you can upgrade the master to 5.7. Note that downgrading from 5.7 to older versions does not work so simply: You must ensure that any 5.7 binary log or relay log has been fully processed, so that you can remove it before proceeding with the downgrade.

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 Changes in MySQL 5.7. It is safest to perform these operations separately on the slaves and the master, and to disable replication of these operations from the master to the slave. To achieve this, use the following procedure:

1. Stop all the slaves and upgrade them. Restart them with the --skip-slave-start option so that they do not connect to the master. 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 master. To do this without restarting the master, execute a SET sql_log_bin = OFF statement. Alternatively, stop the master and restart it without the --log-bin option. If you restart the master, 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 master.

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 slaves later.

4. Re-enable the binary log on the master. If you set sql_log_bin to OFF earlier, execute a SET sql_log_bin = ON statement. If you restarted the master to disable the binary log, restart it with --log-bin, and without enabling the skip_networking system variable so that clients and slaves can connect.

5. Restart the slaves, this time without the --skip-slave-start option.

If you are upgrading an existing replication setup from a version of MySQL that does not support global transaction identifiers to a version that does, you should not enable GTIDs on either the master or the slave before making sure that the setup meets all the requirements for GTID-based replication. For example server_uuid, which was added in MySQL 5.6, must exist for GTIDs to function correctly. See Section 2.3.4, “Setting Up Replication Using GTIDs”, which contains information about converting existing replication setups to use GTID-based replication.

When the server is running with global transaction identifiers (GTIDs) enabled (gtid_mode=ON), do not enable binary logging by mysql_upgrade.

It is not recommended to load a dump file when GTIDs are enabled on the server (gtid_mode=ON), if your dump file includes system tables. mysqldump issues DML instructions for the system tables which use the non-transactional MyISAM storage engine, and this combination is not permitted when GTIDs are enabled. Also be aware that loading a dump file from a server with GTIDs enabled, into another server with GTIDs enabled, causes different transaction identifiers to be generated.
4.4 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 master has binary logging enabled by issuing a `SHOW MASTER STATUS` statement. If logging is enabled, `Position` is nonzero. If binary logging is not enabled, verify that you are running the master with the `--log-bin` option.

- Verify that the `server_id` system variable was set at startup on both the master and slave and that the ID value is unique on each server.

- Verify that the slave is running. Use `SHOW SLAVE STATUS` to check whether the `Slave_IO_Running` and `Slave_SQL_Running` values are both `Yes`. If not, verify the options that were used when starting the slave server. For example, `--skip-slave-start` prevents the slave threads from starting until you issue a `START SLAVE` statement.

- If the slave is running, check whether it established a connection to the master. Use `SHOW PROCESSLIST`, find the I/O and SQL threads and check their `State` column to see what they display. See Section 5.2, “Replication Implementation Details”. If the I/O thread state says `Connecting to master`, check the following:
  - Verify the privileges for the user being used for replication on the master.
  - Check that the host name of the master is correct and that you are using the correct port to connect to the master. 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 master or slave to disable networking. If so, comment the setting or remove it.
  - If the master 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 master by using `ping` or `traceroute/tracert` to reach the host.

- If the slave was running previously but has stopped, the reason usually is that some statement that succeeded on the master failed on the slave. This should never happen if you have taken a proper snapshot of the master, and never modified the data on the slave outside of the slave thread. If the slave stops unexpectedly, it is a bug or you have encountered one of the known replication limitations described in Section 4.1, “Replication Features and Issues”. If it is a bug, see Section 4.5, “How to Report Replication Bugs or Problems”, for instructions on how to report it.

- If a statement that succeeded on the master refuses to run on the slave, try the following procedure if it is not feasible to do a full database resynchronization by deleting the slave’s databases and copying a new snapshot from the master:
  1. Determine whether the affected table on the slave is different from the master table. Try to understand how this happened. Then make the slave’s table identical to the master’s and run `START SLAVE`.
  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 master.
  3. If you decide that the slave can skip the next statement from the master, issue the following statements:
SET GLOBAL sql_slave_skip_counter = 1;
START SLAVE;

The value of N should be 1 if the next statement from the master 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 master.

See also SET GLOBAL sql_slave_skip_counter Statement.

4. If you are sure that the slave started out perfectly synchronized with the master, and that no one has updated the tables involved outside of the slave thread, 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.

4.5 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 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 slave outside of the slave thread, the data goes out of synchrony, and you can have unique key violations on updates. In this case, the slave 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. Run the slave with the --log-slave-updates and --log-bin options. These options cause the slave to log the updates that it receives from the master into its own binary logs.

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 master
   • All binary log files from the slave
   • The output of SHOW MASTER STATUS from the master at the time you discovered the problem
   • The output of SHOW SLAVE STATUS from the slave at the time you discovered the problem
   • Error logs from the master and the slave

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 SLAVE STATUS.

   shell> 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 How to Report Bugs or Problems.
# Chapter 5 Replication Implementation

## Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1 Replication Formats</td>
<td>179</td>
</tr>
<tr>
<td>5.1.1 Advantages and Disadvantages of Statement-Based and Row-Based Replication</td>
<td>180</td>
</tr>
<tr>
<td>5.1.2 Usage of Row-Based Logging and Replication</td>
<td>183</td>
</tr>
<tr>
<td>5.1.3 Determination of Safe and Unsafe Statements in Binary Logging</td>
<td>185</td>
</tr>
<tr>
<td>5.2 Replication Implementation Details</td>
<td>187</td>
</tr>
<tr>
<td>5.3 Replication Channels</td>
<td>188</td>
</tr>
<tr>
<td>5.3.1 Commands for Operations on a Single Channel</td>
<td>189</td>
</tr>
<tr>
<td>5.3.2 Compatibility with Previous Replication Statements</td>
<td>189</td>
</tr>
<tr>
<td>5.3.3 Startup Options and Replication Channels</td>
<td>190</td>
</tr>
<tr>
<td>5.3.4 Replication Channel Naming Conventions</td>
<td>191</td>
</tr>
<tr>
<td>5.4 Replication Relay and Status Logs</td>
<td>192</td>
</tr>
<tr>
<td>5.4.1 The Slave Relay Log</td>
<td>192</td>
</tr>
<tr>
<td>5.4.2 Slave Status Logs</td>
<td>193</td>
</tr>
<tr>
<td>5.5 How Servers Evaluate Replication Filtering Rules</td>
<td>197</td>
</tr>
<tr>
<td>5.5.1 Evaluation of Database-Level Replication and Binary Logging Options</td>
<td>198</td>
</tr>
<tr>
<td>5.5.2 Evaluation of Table-Level Replication Options</td>
<td>200</td>
</tr>
<tr>
<td>5.5.3 Replication Rule Application</td>
<td>202</td>
</tr>
</tbody>
</table>

Replication is based on the master 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 slave that connects to the master requests a copy of the binary log. That is, it pulls the data from the master, rather than the master pushing the data to the slave. The slave 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 master. Tables are created or their structure modified, and data is inserted, deleted, and updated according to the changes that were originally made on the master.

Because each slave is independent, the replaying of the changes from the master's binary log occurs independently on each slave that is connected to the master. In addition, because each slave receives a copy of the binary log only by requesting it from the master, the slave 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 master or slave side.

For more information on the specifics of the replication implementation, see Section 5.2, “Replication Implementation Details”.

Masters and slaves report their status in respect of the replication process regularly so that you can monitor them. See Examining Thread Information, for descriptions of all replicated-related states.

The master binary log is written to a local relay log on the slave before it is processed. The slave also records information about the current position with the master's binary log and the local relay log. See Section 5.4, “Replication Relay and Status Logs”.

Database changes are filtered on the slave 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 5.5, “How Servers Evaluate Replication Filtering Rules”.

## 5.1 Replication Formats

Replication works because events written to the binary log are read from the master and then processed on the slave. 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 master's binary log. The correlation between binary logging formats and the terms used during replication are:

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

- 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 Mixed Binary Logging Format.

Prior to MySQL 5.7.7, statement-based format was the default. In MySQL 5.7.7 and later, row-based format is the default.

**NDB Cluster.** The default binary logging format in MySQL NDB Cluster 7.5 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 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 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 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 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 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 5.1.1, “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 Stored Program Binary Logging.

### 5.1.1 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 UDF or stored program that is nondeterministic, since the value returned by such a UDF 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 UDF or stored program, so its effect on table rows and data is the same on both the master and slave.) See **Section 4.1.16, “Replication of Invoked Features”**, for more information.
  - DELETE and UPDATE statements that use a LIMIT clause without an ORDER BY are nondeterministic. See **Section 4.1.17, “Replication and LIMIT”**.
  - Deterministic UDFs must be applied on the slaves.
  - 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 master and the slave are started with the --sysdate-is-now option)
    - GET_LOCK()
    - IS_FREE_LOCK()
    - IS_USED_LOCK()
    - MASTER_POS_WAIT()
Advantages and Disadvantages of Statement-Based and Row-Based Replication

- \texttt{RAND()}
- \texttt{RELEASE\_LOCK()}
- \texttt{SLEEP()}
- \texttt{VERSION()}

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

For more information, see Section 4.1.15, “Replication and System Functions”.

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

\texttt{[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 \texttt{SHOW WARNINGS}.

- \texttt{INSERT ... SELECT} requires a greater number of row-level locks than with row-based replication.
- \texttt{UPDATE} statements that require a table scan (because no index is used in the \texttt{WHERE} clause) must lock a greater number of rows than with row-based replication.
- For InnoDB: An \texttt{INSERT} statement that uses \texttt{AUTO\_INCREMENT} blocks other nonconflicting \texttt{INSERT} statements.
- For complex statements, the statement must be evaluated and executed on the slave before the rows are updated or inserted. With row-based replication, the slave only has to modify the affected rows, not execute the full statement.
- If there is an error in evaluation on the slave, particularly when executing complex statements, statement-based replication may slowly increase the margin of error across the affected rows over time. See Section 4.1.27, “Slave Errors During Replication”.
- Stored functions execute with the same \texttt{NOW()} value as the calling statement. However, this is not true of stored procedures.
- Deterministic UDFs must be applied on the slaves.
- Table definitions must be (nearly) identical on master and slave. See Section 4.1.10, “Replication with Differing Table Definitions on Master and Slave”, for more information.

**Advantages of row-based replication**

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

  \textbf{Note}

  Statements that update the information in the \texttt{mysql} system database—such as \texttt{GRANT}, \texttt{REVOKE} and the manipulation of triggers, stored routines (including stored procedures), and views—are all replicated to slaves using statement-based replication.

  For statements such as \texttt{CREATE TABLE ... SELECT}, a \texttt{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 master, which thus achieves higher concurrency, for the following types of statements:
Usage of Row-Based Logging and Replication

- **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 slave 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 UDFs 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 slave what statements were received from the master 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 slave 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.

**5.1.2 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 5.1, “Replication Formats”, and Section 5.1.1, “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 Known Issues in NDB Cluster Replication.

- **Row-based logging of temporary tables.** As noted in Section 4.1.29, “Replication and Temporary Tables”, temporary tables are not replicated when using row-based format. When using mixed format logging, “safe” statements involving temporary tables are logged using statement-based format. For more information, see Section 5.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”.

Temporary tables are not replicated when using row-based 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.
Usage of Row-Based Logging and Replication

You can switch from statement-based to row-based binary logging format at runtime even when temporary tables have been created. From MySQL 5.7.25, 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 \texttt{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 \texttt{DROP TEMPORARY TABLE IF EXISTS} statement is not logged. In previous releases, the \texttt{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 \texttt{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 slave, 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 slave'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 slave that match those on the master. You should be aware of the potential for this delay in your applications.

- **Reading the binary log.** \texttt{mysqlbinlog} displays row-based events in the binary log using the \texttt{BINLOG} statement (see \texttt{BINLOG Statement}). This statement displays an event as a base 64-encoded string, the meaning of which is not evident. When invoked with the \texttt{--base64-output=DECODE-ROWS} and \texttt{--verbose} options, \texttt{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 \texttt{mysqlbinlog Row Event Display}.

- **Binary log execution errors and slave\_exec\_mode.** Using \texttt{slave\_exec\_mode=}IDEMPOTENT is generally only useful with MySQL NDB Cluster replication, for which IDEMPOTENT is the default value. (See NDB Cluster Replication: Multi-Master and Circular Replication). When \texttt{slave\_exec\_mode=}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 slave, so that the master and slave are no longer synchronized. Latency issues and use of nontransactional tables with RBR when \texttt{slave\_exec\_mode=}IDEMPOTENT can cause the master and slave to diverge even further. For more information about \texttt{slave\_exec\_mode}, see \texttt{Server System Variables}.

For other scenarios, setting \texttt{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 \texttt{IGNORE\_SERVER\_IDS} option for the \texttt{CHANGE MASTER TO} statement. This option works with statement-based and row-based logging formats. Another method to filter out changes on some slaves is to use a \texttt{WHERE} clause that includes the relation @server\_id <> id\_value clause with \texttt{UPDATE} and \texttt{DELETE} statements. For example, \texttt{WHERE @server\_id <> 1}. However, this does not work correctly with row-based logging. To use the \texttt{server\_id} system variable for statement filtering, use statement-based logging.

- **Database-level replication options.** The effects of the \texttt{--replicate-do-db}, \texttt{--replicate-ignore-db}, and \texttt{--replicate-rewrite-db} options differ considerably depending on whether row-based or statement-based logging is used. Therefore, it is recommended to avoid database-level options and instead use table-level options such as \texttt{--replicate-do-table} and \texttt{--replicate-ignore-table}. For more information about these options and the impact replication format has on how they operate, see Section 2.6, “Replication and Binary Logging Options and Variables”.

184
• **RBL, nontransactional tables, and stopped slaves.** When using row-based logging, if the slave server is stopped while a slave thread is updating a nontransactional table, the slave 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 SLAVE** or **STOP SLAVE SQL_THREAD** prior to shutting down the slave MySQL server helps prevent issues from occurring, and is always recommended regardless of the logging format or storage engine you use.

### 5.1.3 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 4.1.12, “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. Formerly, if a large number of such statements were executed on the master, this could lead to excessively large error log files. To prevent this, MySQL 5.7 provides a warning suppression mechanism, which behaves as follows: 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 5.1, “Replication Formats”.

**Statements considered unsafe.**

Statements with the following characteristics are considered unsafe:

- **Statements containing system functions that may return a different value on slave.** These functions include **FOUND_ROWS()**, **GET_LOCK()**, **IS_FREE_LOCK()**, **IS_USED_LOCK()**, **LOAD_FILE()**, **MASTER_POS_WAIT()**, **PASSWORD()**, **RAND()**, **RELEASE_LOCK()**, **ROW_COUNT()**, **SESSION_USER()**, **SLEEP()**, **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 4.1.15, "Replication and System Functions".

- **References to system variables.** Most system variables are not replicated correctly using the statement-based format. See Section 4.1.37, "Replication and Variables". For exceptions, see Mixed Binary Logging Format.

- **UDFs.** Since we have no control over what a UDF 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 (Bug #11756280, Bug #48183).

- **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 master and the slave. 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 4.1.1, "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 4.1.17, "Replication and LIMIT".

- **Accesses or references log tables.** The contents of the system log table may differ between master and slave.

- **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 4.1.33, "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 master are being prepared on the slave 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 slave.

  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.

  For additional information, see Section 4.1, "Replication Features and Issues".
5.2 Replication Implementation Details

MySQL replication capabilities are implemented using three threads, one on the master server and two on the slave:

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

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

- **Slave I/O thread.** When a `START SLAVE` statement is issued on a slave server, the slave creates an I/O thread, which connects to the master and asks it to send the updates recorded in its binary logs.

  The slave I/O thread reads the updates that the master's Binlog Dump thread sends (see previous item) and copies them to local files that comprise the slave's relay log.

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

- **Slave SQL thread.** The slave creates an SQL thread to read the relay log that is written by the slave I/O thread and execute the events contained therein.

In the preceding description, there are three threads per master/slave connection. A master that has multiple slaves creates one binary log dump thread for each currently connected slave, and each slave has its own I/O and SQL threads.

A slave uses two threads to separate reading updates from the master and executing them into independent tasks. Thus, the task of reading statements is not slowed down if statement execution is slow. For example, if the slave server has not been running for a while, its I/O thread can quickly fetch all the binary log contents from the master when the slave starts, even if the SQL thread lags far behind. If the slave stops before the SQL thread has executed all the fetched statements, the I/O thread has at least fetched everything so that a safe copy of the statements is stored locally in the slave's relay logs, ready for execution the next time that the slave starts.

The `SHOW PROCESSLIST` statement provides information that tells you what is happening on the master and on the slave regarding replication. For information on master states, see Replication Master Thread States. For slave states, see Replication Slave I/O Thread States, and Replication Slave SQL Thread States.

The following example illustrates how the three threads show up in the output from `SHOW PROCESSLIST`.

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

```
mysql> SHOW PROCESSLIST\G
*************************** 1. row ***************************
  Id: 2
  User: root
  Host: localhost:32331
  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 replication thread that services a connected slave. The State information indicates that all outstanding updates have been sent to the slave and that the master is
waiting for more updates to occur. If you see no Binlog Dump threads on a master server, this means that replication is not running; that is, no slaves are currently connected.

On a slave 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 I/O thread that is communicating with the master server, and thread 11 is the SQL 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 slave is compared to the master. See MySQL 5.7 FAQ: Replication. If sufficient time elapses on the master side without activity on the Binlog Dump thread, the master determines that the slave 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 Server System Variables.

The SHOW SLAVE STATUS statement provides additional information about replication processing on a slave server. See Section 2.7.1, "Checking Replication Status".

### 5.3 Replication Channels

In MySQL multi-source replication, a slave opens multiple replication channels, one for each master. The replication channels represent the path of transactions flowing from a master to the slave. Each replication channel has its own receiver (I/O) thread, one or more applier (SQL) threads, and relay log. When transactions from a master 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 masters and slaves for multi-source replication, to start, stop and reset multi-source slaves, and to monitor multi-source replication, see Section 2.4, "MySQL Multi-Source Replication".

The maximum number of channels that can be created on one slave in a multi-source replication topology is 256. Each replication channel must have a unique (nonempty) name, as explained in Section 5.3.4, "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 replication slave must replicate from a different master. You cannot set up multiple replication channels from a single slave to a single master. This is because the server IDs of replication slaves must be unique in a replication topology. The master distinguishes slaves only by their
server IDs, not by the names of the replication channels, so it cannot recognize different replication channels from the same slave.

A multi-source replication slave can also be set up as a multi-threaded replication slave, by setting the `slave_parallel_workers` system variable to a value greater than 0. When you do this on a multi-source replication slave, each channel on the slave 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.

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 slaves function as expected (see Section 5.3.2, “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.

### 5.3.1 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 MASTER TO`
- `START SLAVE`
- `STOP SLAVE`
- `SHOW RELAYLOG EVENTS`
- `FLUSH RELAY LOGS`
- `SHOW SLAVE STATUS`
- `RESET SLAVE`

Similarly, an additional `channel` parameter is introduced for the following functions:

- `MASTER_POS_WAIT()`
- `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()`

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

- `START SLAVE`
- `STOP SLAVE`

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

- `START SLAVE`
- `STOP SLAVE`
- `SHOW SLAVE STATUS`
- `FLUSH RELAY LOGS`

### 5.3.2 Compatibility with Previous Replication Statements

When a replication slave 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 SLAVE** starts replication threads for all channels, except the `group_replication_recovery` and `group_replication_applier` channels.

- **STOP SLAVE** stops replication threads for all channels, except the `group_replication_recovery` and `group_replication_applier` channels.

- **SHOW SLAVE STATUS** reports the status for all channels, except the `group_replication_applier` channel.

- **FLUSH RELAY LOGS** flushes the relay logs for all channels, except the `group_replication_applier` channel.

- **RESET SLAVE** resets all channels.

  **Warning**

  Use **RESET SLAVE** 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 slave. 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 MASTER TO**
- **MASTER_POS_WAIT()**
- **WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()**

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

### 5.3.3 Startup Options and Replication Channels

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

The following startup settings *must* be configured correctly to use multi-source replication.

- **relay_log_info_repository**
  - This must be set to **TABLE**. If this variable is set to **FILE**, attempting to add more sources to a slave fails with error **ER_SLAVE_NEW_CHANNEL_WRONG_REPOSITORY**.

- **master_info_repository**
  - This must be set to **TABLE**. If this variable is set to **FILE**, attempting to add more sources to a slave fails with error **ER_SLAVE_NEW_CHANNEL_WRONG_REPOSITORY**.

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

- **--log-slave-updates**
  - All transactions received by the slave (even from multiple sources) are written in the binary log.

- **--relay-log-purge**
  - When set, each channel purges its own relay log automatically.
Replication Channel Naming Conventions

• --slave-transaction-retries
  Applier threads of all channels retry transactions.

• --skip-slave-start
  No replication threads start on any channels.

• --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 \( \text{relay_log_space_limit} \times N \).

• --slave-parallel-workers=value
  Number of slave parallel workers per channel.

• slave_checkpoint_group
  Waiting time by an I/O thread for each source.

• --relay-log-index=filename
  Base name for each channel’s relay log index file. See Section 5.3.4, “Replication Channel Naming Conventions”.

• --relay-log=filename
  Denotes the base name of each channel’s relay log file. See Section 5.3.4, “Replication Channel Naming Conventions”.

• --slave-net-timeout=N
  This value is set per channel, so that each channel waits for \( N \) seconds to check for a broken connection.

• --slave-skip-counter=N
  This value is set per channel, so that each channel skips \( N \) events from its master.

5.3.4 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 slave tables, 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 replication slave. 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.

### 5.4 Replication Relay and Status Logs

During replication, a slave server creates several logs that hold the binary log events relayed from the master to the slave, and to record information about the current status and location within the relay log. There are three types of logs used in the process, listed here:

- The **master info log** contains status and current configuration information for the slave’s connection to the master. This log holds information on the master host name, login credentials, and coordinates indicating how far the slave has read from the master’s binary log.

  This log can be written to the `mysql.slave_master_info` table instead of a file, by starting the slave with `master_info_repository=TABLE`.

- The **relay log** consists of the events read from the binary log of the master and written by the slave I/O thread. Events in the relay log are executed on the slave as part of the SQL thread.

- The **relay log info log** holds status information about the execution point within the slave’s relay log.

  This log can be written to the `mysql.slave_relay_log_info` table instead of a file by starting the slave with `relay_log_info_repository=TABLE`.

When tables are used for the slave status logs, a warning is given if `mysqld` is unable to initialize the replication logging tables, but the slave is allowed to continue starting. This situation is most likely to occur when upgrading from a version of MySQL that does not support slave logging tables to one in which they are supported.

**Important**

Do not attempt to update or insert rows in the `slave_master_info` or `slave_relay_log_info` table 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, while statements that perform only reads are permitted at any time.

If you set `master_info_repository` and `relay_log_info_repository` to `TABLE`, the `mysql.slave_master_info` and `mysql.slave_relay_log_info` tables are created using the transactional storage engine InnoDB. As a table, updates to the relay log info log are committed together with the transactions, meaning that the slave’s progress information recorded in that log is always consistent with what has been applied to the database, even in the event of an unexpected server halt. The `--relay-log-recovery` option must be enabled on the slave to guarantee resilience. For more details, see [Section 3.2, “Handling an Unexpected Halt of a Replication Slave”](#).

### 5.4.1 The Slave 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 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 `mysqlbinlog — Utility for Processing Binary Log Files`).
Slave Status Logs

By default, relay log file names have the form `host_name-relay-bin.nnnnnn` in the data directory, where `host_name` is the name of the slave server host and `nnnnnn` is a sequence number. Successive relay log files are created using successive sequence numbers, beginning with `000001`. The slave 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` in the data directory.

The default relay log file and relay log index file names can be overridden with, respectively, the `relay_log` and `relay_log_index` system variables (see Section 2.6, “Replication and Binary Logging Options and Variables”).

If a slave uses the default host-based relay log file names, changing a slave'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 slave's host name might change in the future (for example, if networking is set up on the slave 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 slave. This will make the names 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 slave server, prepend the contents of the old relay log index file to the new one, and then restart the slave. On a Unix system, this can be done as shown here:

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

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

- Each time the I/O 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,” 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 SQL 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 SQL thread takes care of doing so. However, `FLUSH LOGS` rotates relay logs, which influences when the SQL thread deletes them.

5.4.2 Slave Status Logs

A replication slave server creates two logs. By default, these logs are files named `master.info` and `relay-log.info` and created in the data directory. The names and locations of these files can be changed by using the `--master-info-file` option and `relay_log_info_file` system variable, respectively. Either or both of these logs can also be written to tables in the `mysql` database by starting the server with the appropriate option: use the `master_info_repository` system variable to have the master info log written to the `mysql.slave_master_info` table, and use the `relay_log_info_repository` system variable to have the relay log info log written to the `mysql.slave_relay_log_info` table. See Section 2.6, “Replication and Binary Logging Options and Variables”.

The two status logs contain information similar to that shown in the output of the `SHOW SLAVE STATUS` statement, which is discussed in SQL Statements for Controlling Slave Servers. Because the status logs are stored on disk, they survive a slave server's shutdown. The next time the slave starts up, it reads the two logs to determine how far it has proceeded in reading binary logs from the master and in processing its own relay logs.
Access to the master info log file or table should be restricted because it contains the password for connecting to the master. See Passwords and Logging.

If you set `master_info_repository` and `relay_log_info_repository` to TABLE, the `mysql.slave_master_info` and `mysql.slave_relay_log_info` tables are created using the InnoDB transactional storage engine. As a table, updates to the relay log info log are committed together with the transactions, meaning that the slave's progress information recorded in that log is always consistent with what has been applied to the database, even in the event of an unexpected server halt. The `--relay-log-recovery` option must be enabled on the slave to guarantee resilience. For more details, see Section 3.2, “Handling an Unexpected Halt of a Replication Slave”.

One additional slave status log is created primarily for internal use, and holds status information about worker threads on a multithreaded replication slave. This slave worker log includes the names and positions for the relay log file and master binary log file for each worker thread. If the relay log info log for the slave is created as a table, the slave worker log is written to the `mysql.slave_worker_info` table. If the relay log info log is written to a file, the slave worker log 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 slave I/O thread updates the master info log. The following table shows the correspondence between the lines in the `master.info` file, the columns in the `mysql.slave_master_info` table, and the columns displayed by `SHOW SLAVE STATUS`.

<table>
<thead>
<tr>
<th>master.info File Line</th>
<th>slave_master_info Table Column</th>
<th>SHOW SLAVE STATUS Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number_of_lines</td>
<td>[None]</td>
<td>Number of lines in the file, or columns in the table</td>
</tr>
<tr>
<td>2</td>
<td>Master_log_name</td>
<td>Master_Log_File</td>
<td>The name of the master binary log currently being read from the master</td>
</tr>
<tr>
<td>3</td>
<td>Master_log_pos</td>
<td>Read_Master_Log_Pos</td>
<td>The current position within the master binary log that have been read from the master</td>
</tr>
<tr>
<td>4</td>
<td>Host</td>
<td>Master_HOST</td>
<td>The host name of the master</td>
</tr>
<tr>
<td>5</td>
<td>User_name</td>
<td>Master_User</td>
<td>The user name used to connect to the master</td>
</tr>
<tr>
<td>6</td>
<td>User_password</td>
<td>Password (not shown by SHOW SLAVE STATUS)</td>
<td>The password used to connect to the master</td>
</tr>
<tr>
<td>7</td>
<td>Port</td>
<td>Master_Prot</td>
<td>The network port used to connect to the master</td>
</tr>
<tr>
<td>8</td>
<td>Connect_retry</td>
<td>Connect_Retry</td>
<td>The period (in seconds) that the slave will wait before trying to...</td>
</tr>
<tr>
<td>master.info File Line</td>
<td>slave_master_info Column</td>
<td>SHOW SLAVE STATUS Column</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------------</td>
<td>--------------------------</td>
<td>--------------------------</td>
<td>-------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>reconnect to the master</td>
</tr>
<tr>
<td>9</td>
<td>Enabled_ssl</td>
<td>Master_SSL_Allowed</td>
<td>Indicates whether the server supports SSL connections</td>
</tr>
<tr>
<td>10</td>
<td>Ssl_ca</td>
<td>Master_SSL_CA_File</td>
<td>The file used for the Certificate Authority (CA) certificate</td>
</tr>
<tr>
<td>11</td>
<td>Ssl_capath</td>
<td>Master_SSL_CA_Path</td>
<td>The path to the Certificate Authority (CA) certificates</td>
</tr>
<tr>
<td>12</td>
<td>Ssl_cert</td>
<td>Master_SSL_Cert</td>
<td>The name of the SSL certificate file</td>
</tr>
<tr>
<td>13</td>
<td>Ssl_cipher</td>
<td>Master_SSL_Cipher</td>
<td>The list of possible ciphers used in the handshake for the SSL connection</td>
</tr>
<tr>
<td>14</td>
<td>Ssl_key</td>
<td>Master_SSL_Key</td>
<td>The name of the SSL key file</td>
</tr>
<tr>
<td>15</td>
<td>Ssl_verify_server_cert</td>
<td>Master_SSL.Verify_Server_Cert</td>
<td>Whether to verify the server certificate</td>
</tr>
<tr>
<td>16</td>
<td>Heartbeat</td>
<td>[None]</td>
<td>Interval between replication heartbeats, in seconds</td>
</tr>
<tr>
<td>17</td>
<td>Bind</td>
<td>Master_Bind</td>
<td>Which of the slave's network interfaces should be used for connecting to the master</td>
</tr>
<tr>
<td>18</td>
<td>Ignored_server_ids</td>
<td>Replicate_Ignore_Server_Ids</td>
<td>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.</td>
</tr>
<tr>
<td>19</td>
<td>Uuid</td>
<td>Master_UUID</td>
<td>The master's unique ID</td>
</tr>
<tr>
<td>20</td>
<td>Retry_count</td>
<td>Master_Retry_Count</td>
<td>Maximum number of reconnection attempts permitted</td>
</tr>
</tbody>
</table>
### Slave Status Logs

<table>
<thead>
<tr>
<th>master.info File Line</th>
<th>slave_master_info Table Column</th>
<th>SHOW SLAVE STATUS Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
<td>Ssl_crl</td>
<td>[None]</td>
<td>Path to an SSL certificate revocation-list file</td>
</tr>
<tr>
<td>22</td>
<td>Ssl_crlpath</td>
<td>[None]</td>
<td>Path to a directory containing SSL certificate revocation-list files</td>
</tr>
<tr>
<td>23</td>
<td>Enabled_auto_position</td>
<td>Auto_position</td>
<td>If autopositioning is in use or not</td>
</tr>
<tr>
<td>24</td>
<td>Channel_name</td>
<td>Channel_name</td>
<td>The name of the replication channel</td>
</tr>
<tr>
<td>25</td>
<td>Tls_version</td>
<td>Master_TLS_Version</td>
<td>TLS version on master</td>
</tr>
</tbody>
</table>

The slave SQL thread updates the relay log info log. The relay-log.info file includes a line count and a replication delay value. The following table shows the correspondence between the lines in the relay-log.info file, the columns in the mysql.slave_relay_log_info table, and the columns displayed by `SHOW SLAVE STATUS`.

<table>
<thead>
<tr>
<th>Line in</th>
<th>slave_relay_log_info Table Column</th>
<th>SHOW SLAVE STATUS Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Number_of_lines</td>
<td>[None]</td>
<td>Number of lines in the file or columns in the table</td>
</tr>
<tr>
<td>2</td>
<td>Relay_log_name</td>
<td>Relay_Log_File</td>
<td>The name of the current relay log file</td>
</tr>
<tr>
<td>3</td>
<td>Relay_log_pos</td>
<td>Relay_Log_Pos</td>
<td>The current position within the relay log file; events up to this position have been executed on the slave database</td>
</tr>
<tr>
<td>4</td>
<td>Master_log_name</td>
<td>Relay_Master_Log_File</td>
<td>The name of the master binary log file from which the events in the relay log file were read</td>
</tr>
<tr>
<td>5</td>
<td>Master_log_pos</td>
<td>Exec_Master_Log_Pos</td>
<td>The equivalent position within the master's binary log file of events that have already been executed</td>
</tr>
<tr>
<td>6</td>
<td>Sql_delay</td>
<td>SQL_Delay</td>
<td>The number of seconds that the slave must lag the master</td>
</tr>
<tr>
<td>7</td>
<td>Number_of_workers</td>
<td>[None]</td>
<td>The number of slave worker threads</td>
</tr>
</tbody>
</table>
How Servers Evaluate Replication Filtering Rules

<table>
<thead>
<tr>
<th>Line in relay-log.info</th>
<th>slave_relay_log_info Table Column</th>
<th>SHOW SLAVE STATUS Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>for executing replication events (transactions) in parallel</td>
</tr>
<tr>
<td>8</td>
<td>Id</td>
<td>[None]</td>
<td>ID used for internal purposes; currently this is always 1</td>
</tr>
<tr>
<td>9</td>
<td>Channel_name</td>
<td>Channel_name</td>
<td>The name of the replication channel</td>
</tr>
</tbody>
</table>

In versions of MySQL prior to MySQL 5.6, the relay-log.info file does not include a line count or a delay value (and the slave_relay_log_info table is not available).

<table>
<thead>
<tr>
<th>Line</th>
<th>Status Column</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Relay_Log_File</td>
<td>The name of the current relay log file</td>
</tr>
<tr>
<td>2</td>
<td>Relay_Log_Pos</td>
<td>The current position within the relay log file; events up to this position have been executed on the slave database</td>
</tr>
<tr>
<td>3</td>
<td>Relay_Master_Log_File</td>
<td>The name of the master binary log file from which the events in the relay log file were read</td>
</tr>
<tr>
<td>4</td>
<td>Exec_Master_Log_Pos</td>
<td>The equivalent position within the master's binary log file of events that have already been executed</td>
</tr>
</tbody>
</table>

Note

If you downgrade a slave server to a version older than MySQL 5.6, the older server does not read the relay-log.info file correctly. To address this, modify the file in a text editor by deleting the initial line containing the number of lines.

The contents of the relay-log.info file and the states shown by the SHOW SLAVE STATUS statement might not match if the relay-log.info file has not been flushed to disk. Ideally, you should only view relay-log.info on a slave that is offline (that is, mysql is not running). For a running system, you can use SHOW SLAVE STATUS, or query the slave_master_info and slave_relay_log_info tables if you are writing the status logs to tables.

When you back up the slave's data, you should back up these two status logs, along with the relay log files. The status logs are needed to resume replication after you restore the data from the slave. If you lose the relay logs but still have the relay log info log, you can check it to determine how far the SQL thread has executed in the master binary logs. Then you can use CHANGE MASTER TO with the MASTER_LOG_FILE and MASTER_LOG_POS options to tell the slave to re-read the binary logs from that point. Of course, this requires that the binary logs still exist on the master.

5.5 How Servers Evaluate Replication Filtering Rules

If a master 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 slaves and each slave determines whether to execute it or ignore it.

On the master, 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 5.5.1, “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 slave to control the events that are executed on the slave.

On the slave side, decisions about whether to execute or ignore statements received from the master are made according to the `--replicate-*` options that the slave was started with. (See Section 2.6, “Replication and Binary Logging Options and Variables”.) The filters governed by these options can also be set dynamically using the CHANGE REPLICA TION FILTER statement. The rules governing such filters are the same whether they are created on startup using `--replicate-*` options or while the slave server is running by CHANGE REPLICA TION FILTER. Note that 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.

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

Database-level options (`--replicate-do-db, --replicate-ignore-db`) are checked first; see Section 5.5.1, “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 5.5.2, “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. This is a change in behavior from previous versions of MySQL, where the statement `CREATE DATABASE dbx` was not replicated if the slave had been started with `--replicate-do-db=dbx --replicate-wild-do-table=db%.t1`. (Bug #46110)

To make it easier to determine what effect an option set will have, it is recommended that you avoid mixing “do” and “ignore” options, or wildcard and nonwildcard options.

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

```
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>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.</td>
</tr>
</tbody>
</table>
```

5.5.1 Evaluation of Database-Level Replication and Binary Logging Options

When evaluating replication options, the slave 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 master.

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.

```
<table>
<thead>
<tr>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>Only DML statements can be logged using the row format. DDL statements are always logged as statements, even when <code>binlog_format=ROW</code>. 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 <code>USE</code> statement in order for a DDL statement to be applied.</td>
</tr>
</tbody>
</table>
```
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 5.5.2, “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?
5.5.2 Evaluation of Table-Level Replication Options

The slave 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 5.5.1, “Evaluation of Database-Level Replication and Binary Logging Options”).

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

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 5.5.1, “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.

---

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`. 
• **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 slave 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`.

### 5.5.3 Replication Rule Application

This section provides additional explanation and examples of usage for different combinations of replication filtering options.

Some typical combinations of replication filter rule types are given in the following table:

<table>
<thead>
<tr>
<th>Condition (Types of Options)</th>
<th>Outcome</th>
</tr>
</thead>
<tbody>
<tr>
<td>No <code>--replicate-*</code> options at all:</td>
<td>The slave executes all events that it receives from the master.</td>
</tr>
<tr>
<td><code>--replicate-*-db</code> options, but no table options:</td>
<td>The slave accepts or ignores events using the database options. It executes all events permitted by those options because there are no table restrictions.</td>
</tr>
<tr>
<td><code>--replicate-*-table</code> options, but no database options:</td>
<td>All events are accepted at the database-checking stage because there are no database conditions. The slave executes or ignores events based solely on the table options.</td>
</tr>
<tr>
<td>A combination of database and table options:</td>
<td>The slave 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, and that may be different depending on whether you are using statement-based or row-based replication; see the text for an example.</td>
</tr>
</tbody>
</table>

A more complex example follows, in which we examine the outcomes for both statement-based and row-based settings.

Suppose that we have two tables `mytbl1` in database `db1` and `mytbl2` in database `db2` on the master, and the slave is running with the following options (and no other replication filtering options):

```
replicate-ignore-db = db1
replicate-do-table  = db2.tbl2
```

Now we execute the following statements on the master:

```
USE db1;
INSERT INTO db2.tbl2 VALUES (1);
```

The results on the slave vary considerably depending on the binary log format, and may not match initial expectations in either case.

**Statement-based replication.** The `USE` statement causes `db1` to be the default database. Thus the `--replicate-ignore-db` option matches, and the `INSERT` statement is ignored. The table options are not checked.

**Row-based replication.** The default database has no effect on how the slave reads database options when using row-based replication. Thus, the `USE` statement makes no difference in how the `--replicate-ignore-db` option is handled: the database specified by this option does not match the database where the `INSERT` statement changes data, so the slave proceeds to check the table options. The table specified by `--replicate-do-table` matches the table to be updated, and the row is inserted.