```
833 OS file reads, 605 OS file writes, 208 OS fsyncs
0.00 reads/s, 0 avg bytes/read, 0.00 writes/s, 0.00 fsyncs/s
-------------------------------------
INSERT BUFFER AND ADAPTIVE HASH INDEX
-------------------------------------
Ibuf: size 1, free list len 0, seg size 2, 0 merges
merged operations:
 insert 0, delete mark 0, delete 0
discarded operations:
 insert 0, delete mark 0, delete 0
Hash table size 553253, node heap has 0 buffer(s)
Hash table size 553253, node heap has 1 buffer(s)
Hash table size 553253, node heap has 3 buffer(s)
Hash table size 553253, node heap has 0 buffer(s)
Hash table size 553253, node heap has 0 buffer(s)
Hash table size 553253, node heap has 0 buffer(s)
Hash table size 553253, node heap has 0 buffer(s)
Hash table size 553253, node heap has 0 buffer(s)
0.00 hash searches/s, 0.00 non-hash searches/s
---
LOG
---
Log sequence number          19643450
Log buffer assigned up to    19643450
Log buffer completed up to   19643450
Log written up to            19643450
Log flushed up to            19643450
Added dirty pages up to      19643450
Pages flushed up to          19643450
Last checkpoint at           19643450
129 log i/o's done, 0.00 log i/o's/second
----------------------
BUFFER POOL AND MEMORY
----------------------
Total large memory allocated 2198863872
Dictionary memory allocated 409606
Buffer pool size   131072
Free buffers       130095
Database pages     973
Old database pages 0
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 810, created 163, written 404
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 973, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
----------------------
INDIVIDUAL BUFFER POOL INFO
----------------------
---BUFFER POOL 0
Buffer pool size   65536
Free buffers       65043
Database pages     491
Old database pages 0
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 411, created 80, written 210
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
Buffer pool hit rate 1000 / 1000, young-making rate 0 / 1000 not 0 / 1000
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 491, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
---BUFFER POOL 1
Buffer pool size   65536
```

```
Free buffers       65052
Database pages     482
Old database pages 0
Modified db pages  0
Pending reads      0
Pending writes: LRU 0, flush list 0, single page 0
Pages made young 0, not young 0
0.00 youngs/s, 0.00 non-youngs/s
Pages read 399, created 83, written 194
0.00 reads/s, 0.00 creates/s, 0.00 writes/s
No buffer pool page gets since the last printout
Pages read ahead 0.00/s, evicted without access 0.00/s, Random read ahead 0.00/s
LRU len: 482, unzip_LRU len: 0
I/O sum[0]:cur[0], unzip sum[0]:cur[0]
--------------
ROW OPERATIONS
--------------
0 queries inside InnoDB, 0 queries in queue
0 read views open inside InnoDB
Process ID=5772, Main thread ID=140286437054208 , state=sleeping
Number of rows inserted 57, updated 354, deleted 4, read 4421
0.00 inserts/s, 0.00 updates/s, 0.00 deletes/s, 0.00 reads/s
----------------------------
END OF INNODB MONITOR OUTPUT
============================
```

Standard Monitor Output Sections

For a description of each metric reported by the Standard Monitor, refer to the Metrics chapter in the Oracle Enterprise Manager for MySQL Database User's Guide.

- `Status`

  This section shows the timestamp, the monitor name, and the number of seconds that per-second averages are based on. The number of seconds is the elapsed time between the current time and the last time `InnoDB` Monitor output was printed.

- `BACKGROUND THREAD`

  The `srv_master_thread` lines shows work done by the main background thread.

- `SEMAPHORES`

  This section reports threads waiting for a semaphore and statistics on how many times threads have needed a spin or a wait on a mutex or a rw-lock semaphore. A large number of threads waiting for semaphores may be a result of disk I/O, or contention problems inside `InnoDB`. Contention can be due to heavy parallelism of queries or problems in operating system thread scheduling. Setting the `innodb_thread_concurrency` system variable smaller than the default value might help in such situations. The `Spin rounds per wait` line shows the number of spinlock rounds per OS wait for a mutex.

  Mutex metrics are reported by `SHOW ENGINE INNODB MUTEX`.

- `LATEST FOREIGN KEY ERROR`

  This section provides information about the most recent foreign key constraint error. It is not present if no such error has occurred. The contents include the statement that failed as well as information about the constraint that failed and the referenced and referencing tables.

- `LATEST DETECTED DEADLOCK`

  This section provides information about the most recent deadlock. It is not present if no deadlock has occurred. The contents show which transactions are involved, the statement each was attempting to execute, the locks they have and need, and which transaction `InnoDB` decided to roll back to break the deadlock. The lock modes reported in this section are explained in Section 15.7.1, “InnoDB Locking”.

- `TRANSACTIONS`

  If this section reports lock waits, your applications might have lock contention. The output can also help to trace the reasons for transaction deadlocks.

- `FILE I/O`

  This section provides information about threads that `InnoDB` uses to perform various types of I/O. The first few of these are dedicated to general `InnoDB` processing. The contents also display information for pending I/O operations and statistics for I/O performance.

  The number of these threads are controlled by the `innodb_read_io_threads` and `innodb_write_io_threads` parameters. See Section 15.14, “InnoDB Startup Options and System Variables”.

- `INSERT BUFFER AND ADAPTIVE HASH INDEX`

  This section shows the status of the `InnoDB` insert buffer (also referred to as the change buffer) and the adaptive hash index.

  For related information, see Section 15.5.2, “Change Buffer”, and Section 15.5.3, “Adaptive Hash Index”.

- `LOG`

  This section displays information about the `InnoDB` log. The contents include the current log sequence number, how far the log has been flushed to disk, and the position at which `InnoDB` last took a checkpoint. (See Section 15.11.3, “InnoDB Checkpoints”.) The section also displays information about pending writes and write performance statistics.

- `BUFFER POOL AND MEMORY`

  This section gives you statistics on pages read and written. You can calculate from these numbers how many data file I/O operations your queries currently are doing.

  For buffer pool statistics descriptions, see Monitoring the Buffer Pool Using the InnoDB Standard Monitor. For additional information about the operation of the buffer pool, see Section 15.5.1, “Buffer Pool”.

- `ROW OPERATIONS`

  This section shows what the main thread is doing, including the number and performance rate for each type of row operation.

15.18 InnoDB Backup and Recovery

This section covers topics related to `InnoDB` backup and recovery.

- For information about backup techniques applicable to `InnoDB`, see Section 15.18.1, “InnoDB Backup”.
- For information about point-in-time recovery, recovery from disk failure or corruption, and how `InnoDB` performs crash recovery, see Section 15.18.2, “InnoDB Recovery”.

15.18.1 InnoDB Backup

The key to safe database management is making regular backups. Depending on your data volume, number of MySQL servers, and database workload, you can use these backup techniques, alone or in combination: hot backup with *MySQL Enterprise Backup*; cold backup by copying files while the MySQL server is shut down; logical backup with `mysqldump` for smaller data volumes or to record the

structure of schema objects. Hot and cold backups are physical backups that copy actual data files, which can be used directly by the `mysqld` server for faster restore.

Using *MySQL Enterprise Backup* is the recommended method for backing up `InnoDB` data.

Note

`InnoDB` does not support databases that are restored using third-party backup tools.

Hot Backups

The `mysqlbackup` command, part of the MySQL Enterprise Backup component, lets you back up a running MySQL instance, including `InnoDB` tables, with minimal disruption to operations while producing a consistent snapshot of the database. When `mysqlbackup` is copying `InnoDB` tables, reads and writes to `InnoDB` tables can continue. MySQL Enterprise Backup can also create compressed backup files, and back up subsets of tables and databases. In conjunction with the MySQL binary log, users can perform point-in-time recovery. MySQL Enterprise Backup is part of the MySQL Enterprise subscription. For more details, see Section 30.2, “MySQL Enterprise Backup Overview”.

Cold Backups

If you can shut down the MySQL server, you can make a physical backup that consists of all files used by `InnoDB` to manage its tables. Use the following procedure:

1. Perform a slow shutdown of the MySQL server and make sure that it stops without errors.
2. Copy all `InnoDB` data files (`ibdata` files and `.ibd` files) into a safe place.
3. Copy all `InnoDB` redo log files (`#ib_redo`*N* files in MySQL 8.0.30 and higher or `ib_logfile` files in earlier releases) to a safe place.
4. Copy your `my.cnf` configuration file or files to a safe place.

Logical Backups Using mysqldump

In addition to physical backups, it is recommended that you regularly create logical backups by dumping your tables using `mysqldump`. A binary file might be corrupted without you noticing it. Dumped tables are stored into text files that are human-readable, so spotting table corruption becomes easier. Also, because the format is simpler, the chance for serious data corruption is smaller. `mysqldump` also has a `--single-transaction` option for making a consistent snapshot without locking out other clients. See Section 7.3.1, “Establishing a Backup Policy”.

Replication works with `InnoDB` tables, so you can use MySQL replication capabilities to keep a copy of your database at database sites requiring high availability. See Section 15.19, “InnoDB and MySQL Replication”.

15.18.2 InnoDB Recovery

This section describes `InnoDB` recovery. Topics include:

- Point-in-Time Recovery
- Recovery from Data Corruption or Disk Failure
- InnoDB Crash Recovery
- Tablespace Discovery During Crash Recovery

Point-in-Time Recovery

To recover an InnoDB database to the present from the time at which the physical backup was made, you must run MySQL server with binary logging enabled, even before taking the backup. To achieve point-in-time recovery after restoring a backup, you can apply changes from the binary log that occurred after the backup was made. See Section 7.5, “Point-in-Time (Incremental) Recovery”.

Recovery from Data Corruption or Disk Failure

If your database becomes corrupted or disk failure occurs, you must perform the recovery using a backup. In the case of corruption, first find a backup that is not corrupted. After restoring the base backup, do a point-in-time recovery from the binary log files using `mysqlbinlog` and `mysql` to restore the changes that occurred after the backup was made.

In some cases of database corruption, it is enough to dump, drop, and re-create one or a few corrupt tables. You can use the `CHECK TABLE` statement to check whether a table is corrupt, although `CHECK TABLE` naturally cannot detect every possible kind of corruption.

In some cases, apparent database page corruption is actually due to the operating system corrupting its own file cache, and the data on disk may be okay. It is best to try restarting the computer first. Doing so may eliminate errors that appeared to be database page corruption. If MySQL still has trouble starting because of InnoDB consistency problems, see Section 15.21.3, “Forcing InnoDB Recovery” for steps to start the instance in recovery mode, which permits you to dump the data.

InnoDB Crash Recovery

To recover from an unexpected MySQL server exit, the only requirement is to restart the MySQL server. InnoDB automatically checks the logs and performs a roll-forward of the database to the present. InnoDB automatically rolls back uncommitted transactions that were present at the time of the crash.

InnoDB crash recovery consists of several steps:

- Tablespace discovery

  Tablespace discovery is the process that InnoDB uses to identify tablespaces that require redo log application. See Tablespace Discovery During Crash Recovery.

- Redo log application

  Redo log application is performed during initialization, before accepting any connections. If all changes are flushed from the buffer pool to the tablespaces (`ibdata*` and `*.ibd` files) at the time of the shutdown or crash, redo log application is skipped. InnoDB also skips redo log application if redo log files are missing at startup.

  - The current maximum auto-increment counter value is written to the redo log each time the value changes, which makes it crash-safe. During recovery, InnoDB scans the redo log to collect counter value changes and applies the changes to the in-memory table object.

    For more information about how InnoDB handles auto-increment values, see Section 15.6.1.6, “AUTO_INCREMENT Handling in InnoDB”, and InnoDB AUTO_INCREMENT Counter Initialization.

  - When encountering index tree corruption, InnoDB writes a corruption flag to the redo log, which makes the corruption flag crash-safe. InnoDB also writes in-memory corruption flag data to an engine-private system table on each checkpoint. During recovery, InnoDB reads corruption flags from both locations and merges results before marking in-memory table and index objects as corrupt.

  - Removing redo logs to speed up recovery is not recommended, even if some data loss is acceptable. Removing redo logs should only be considered after a clean shutdown, with `innodb_fast_shutdown` set to `0` or `1`.

- Roll back of incomplete transactions

  Incomplete transactions are any transactions that were active at the time of unexpected exit or fast shutdown. The time it takes to roll back an incomplete transaction can be three or four times the amount of time a transaction is active before it is interrupted, depending on server load.

  You cannot cancel transactions that are being rolled back. In extreme cases, when rolling back transactions is expected to take an exceptionally long time, it may be faster to start `InnoDB` with an `innodb_force_recovery` setting of `3` or greater. See Section 15.21.3, “Forcing InnoDB Recovery”.

- Change buffer merge

  Applying changes from the change buffer (part of the system tablespace) to leaf pages of secondary indexes, as the index pages are read to the buffer pool.

- Purge

  Deleting delete-marked records that are no longer visible to active transactions.

The steps that follow redo log application do not depend on the redo log (other than for logging the writes) and are performed in parallel with normal processing. Of these, only rollback of incomplete transactions is special to crash recovery. The insert buffer merge and the purge are performed during normal processing.

After redo log application, `InnoDB` attempts to accept connections as early as possible, to reduce downtime. As part of crash recovery, `InnoDB` rolls back transactions that were not committed or in `XA PREPARE` state when the server exited. The rollback is performed by a background thread, executed in parallel with transactions from new connections. Until the rollback operation is completed, new connections may encounter locking conflicts with recovered transactions.

In most situations, even if the MySQL server was killed unexpectedly in the middle of heavy activity, the recovery process happens automatically and no action is required of the DBA. If a hardware failure or severe system error corrupted `InnoDB` data, MySQL might refuse to start. In this case, see Section 15.21.3, “Forcing InnoDB Recovery”.

For information about the binary log and `InnoDB` crash recovery, see Section 5.4.4, “The Binary Log”.

Tablespace Discovery During Crash Recovery

If, during recovery, `InnoDB` encounters redo logs written since the last checkpoint, the redo logs must be applied to affected tablespaces. The process that identifies affected tablespaces during recovery is referred to as *tablespace discovery*.

Tablespace discovery relies on the `innodb_directories` setting, which defines the directories to scan at startup for tablespace files. The `innodb_directories` default setting is NULL, but the directories defined by `innodb_data_home_dir`, `innodb_undo_directory`, and `datadir` are always appended to the `innodb_directories` argument value when InnoDB builds a list of directories to scan at startup. These directories are appended regardless of whether an `innodb_directories` setting is specified explicitly. Tablespace files defined with an absolute path or that reside outside of the directories appended to the `innodb_directories` setting should be added to the `innodb_directories` setting. Recovery is terminated if any tablespace file referenced in a redo log has not been discovered previously.

15.19 InnoDB and MySQL Replication

It is possible to use replication in a way where the storage engine on the replica is not the same as the storage engine on the source. For example, you can replicate modifications to an `InnoDB` table on the source to a `MyISAM` table on the replica. For more information see, Section 17.4.4, “Using Replication with Different Source and Replica Storage Engines”.

For information about setting up a replica, see Section 17.1.2.6, “Setting Up Replicas”, and Section 17.1.2.5, “Choosing a Method for Data Snapshots”. To make a new replica without taking down the source or an existing replica, use the MySQL Enterprise Backup product.

Transactions that fail on the source do not affect replication. MySQL replication is based on the binary log where MySQL writes SQL statements that modify data. A transaction that fails (for example, because of a foreign key violation, or because it is rolled back) is not written to the binary log, so it is not sent to replicas. See Section 13.3.1, “START TRANSACTION, COMMIT, and ROLLBACK Statements”.

Replication and CASCADE. Cascading actions for `InnoDB` tables on the source are executed on the replica *only* if the tables sharing the foreign key relation use `InnoDB` on both the source and replica. This is true whether you are using statement-based or row-based replication. Suppose that you have started replication, and then create two tables on the source, where `InnoDB` is defined as the default storage engine, using the following `CREATE TABLE` statements:

```
CREATE TABLE fc1 (
    i INT PRIMARY KEY,
    j INT
);

CREATE TABLE fc2 (
    m INT PRIMARY KEY,
    n INT,
    FOREIGN KEY ni (n) REFERENCES fc1 (i)
        ON DELETE CASCADE
);
```

If the replica has `MyISAM` defined as the default storage engine, the same tables are created on the replica, but they use the `MyISAM` storage engine, and the `FOREIGN KEY` option is ignored. Now we insert some rows into the tables on the source:

```
source> INSERT INTO fc1 VALUES (1, 1), (2, 2);
Query OK, 2 rows affected (0.09 sec)
Records: 2  Duplicates: 0  Warnings: 0

source> INSERT INTO fc2 VALUES (1, 1), (2, 2), (3, 1);
Query OK, 3 rows affected (0.19 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

At this point, on both the source and the replica, table `fc1` contains 2 rows, and table `fc2` contains 3 rows, as shown here:

```
source> SELECT * FROM fc1;
+---+------+
| i | j    |
+---+------+
| 1 |    1 |
| 2 |    2 |
+---+------+
2 rows in set (0.00 sec)

source> SELECT * FROM fc2;
+---+------+
| m | n    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    1 |
+---+------+
3 rows in set (0.00 sec)

replica> SELECT * FROM fc1;
+---+------+
| i | j    |
+---+------+
| 1 |    1 |
| 2 |    2 |
```

```
+---+------+
2 rows in set (0.00 sec)

replica> SELECT * FROM fc2;
+---+------+
| m | n    |
+---+------+
| 1 |    1 |
| 2 |    2 |
| 3 |    1 |
+---+------+
3 rows in set (0.00 sec)
```

Now suppose that you perform the following `DELETE` statement on the source:

```
source> DELETE FROM fc1 WHERE i=1;
Query OK, 1 row affected (0.09 sec)
```

Due to the cascade, table `fc2` on the source now contains only 1 row:

```
source> SELECT * FROM fc2;
+---+---+
| m | n |
+---+---+
| 2 | 2 |
+---+---+
1 row in set (0.00 sec)
```

However, the cascade does not propagate on the replica because on the replica the `DELETE` for `fc1` deletes no rows from `fc2`. The replica's copy of `fc2` still contains all of the rows that were originally inserted:

```
replica> SELECT * FROM fc2;
+---+---+
| m | n |
+---+---+
| 1 | 1 |
| 3 | 1 |
| 2 | 2 |
+---+---+
3 rows in set (0.00 sec)
```

This difference is due to the fact that the cascading deletes are handled internally by the `InnoDB` storage engine, which means that none of the changes are logged.

15.20 InnoDB memcached Plugin

Note

The `InnoDB memcached` plugin is deprecated as of MySQL 8.0.22; expect support for it to be removed in a future version of MySQL.

The `InnoDB memcached` plugin (`daemon_memcached`) provides an integrated `memcached` daemon that automatically stores and retrieves data from `InnoDB` tables, turning the MySQL server into a fast "key-value store". Instead of formulating queries in SQL, you can use simple `get`, `set`, and `incr` operations that avoid the performance overhead associated with SQL parsing and constructing a query optimization plan. You can also access the same `InnoDB` tables through SQL for convenience, complex queries, bulk operations, and other strengths of traditional database software.

This "NoSQL-style" interface uses the `memcached` API to speed up database operations, letting `InnoDB` handle memory caching using its buffer pool mechanism. Data modified through `memcached` operations such as `add`, `set`, and `incr` are stored to disk, in `InnoDB` tables. The combination of `memcached` simplicity and `InnoDB` reliability and consistency provides users with the best of both worlds, as explained in Section 15.20.1, "Benefits of the InnoDB memcached Plugin". For an architectural overview, see Section 15.20.2, "InnoDB memcached Architecture".

15.20.1 Benefits of the InnoDB memcached Plugin

This section outlines advantages the `daemon_memcached` plugin. The combination of `InnoDB` tables and `memcached` offers advantages over using either by themselves.

- Direct access to the `InnoDB` storage engine avoids the parsing and planning overhead of SQL.
- Running `memcached` in the same process space as the MySQL server avoids the network overhead of passing requests back and forth.
- Data written using the `memcached` protocol is transparently written to an `InnoDB` table, without going through the MySQL SQL layer. You can control frequency of writes to achieve higher raw performance when updating non-critical data.
- Data requested through the `memcached` protocol is transparently queried from an `InnoDB` table, without going through the MySQL SQL layer.
- Subsequent requests for the same data is served from the `InnoDB` buffer pool. The buffer pool handles the in-memory caching. You can tune performance of data-intensive operations using `InnoDB` configuration options.
- Data can be unstructured or structured, depending on the type of application. You can create a new table for data, or use existing tables.
- `InnoDB` can handle composing and decomposing multiple column values into a single `memcached` item value, reducing the amount of string parsing and concatenation required in your application. For example, you can store the string value `2|4|6|8` in the `memcached` cache, and have `InnoDB` split the value based on a separator character, then store the result in four numeric columns.
- The transfer between memory and disk is handled automatically, simplifying application logic.
- Data is stored in a MySQL database to protect against crashes, outages, and corruption.
- You can access the underlying `InnoDB` table through SQL for reporting, analysis, ad hoc queries, bulk loading, multi-step transactional computations, set operations such as union and intersection, and other operations suited to the expressiveness and flexibility of SQL.
- You can ensure high availability by using the `daemon_memcached` plugin on a source server in combination with MySQL replication.
- The integration of `memcached` with MySQL provides a way to make in-memory data persistent, so you can use it for more significant kinds of data. You can use more `add`, `incr`, and similar write operations in your application without concern that data could be lost. You can stop and start the `memcached` server without losing updates made to cached data. To guard against unexpected outages, you can take advantage of `InnoDB` crash recovery, replication, and backup capabilities.
- The way `InnoDB` does fast primary key lookups is a natural fit for `memcached` single-item queries. The direct, low-level database access path used by the `daemon_memcached` plugin is much more efficient for key-value lookups than equivalent SQL queries.
- The serialization features of `memcached`, which can turn complex data structures, binary files, or even code blocks into storable strings, offer a simple way to get such objects into a database.
- Because you can access the underlying data through SQL, you can produce reports, search or update across multiple keys, and call functions such as `AVG()` and `MAX()` on `memcached` data. All of these operations are expensive or complicated using `memcached` by itself.
- You do not need to manually load data into `memcached` at startup. As particular keys are requested by an application, values are retrieved from the database automatically, and cached in memory using the `InnoDB` buffer pool.
- Because `memcached` consumes relatively little CPU, and its memory footprint is easy to control, it can run comfortably alongside a MySQL instance on the same system.

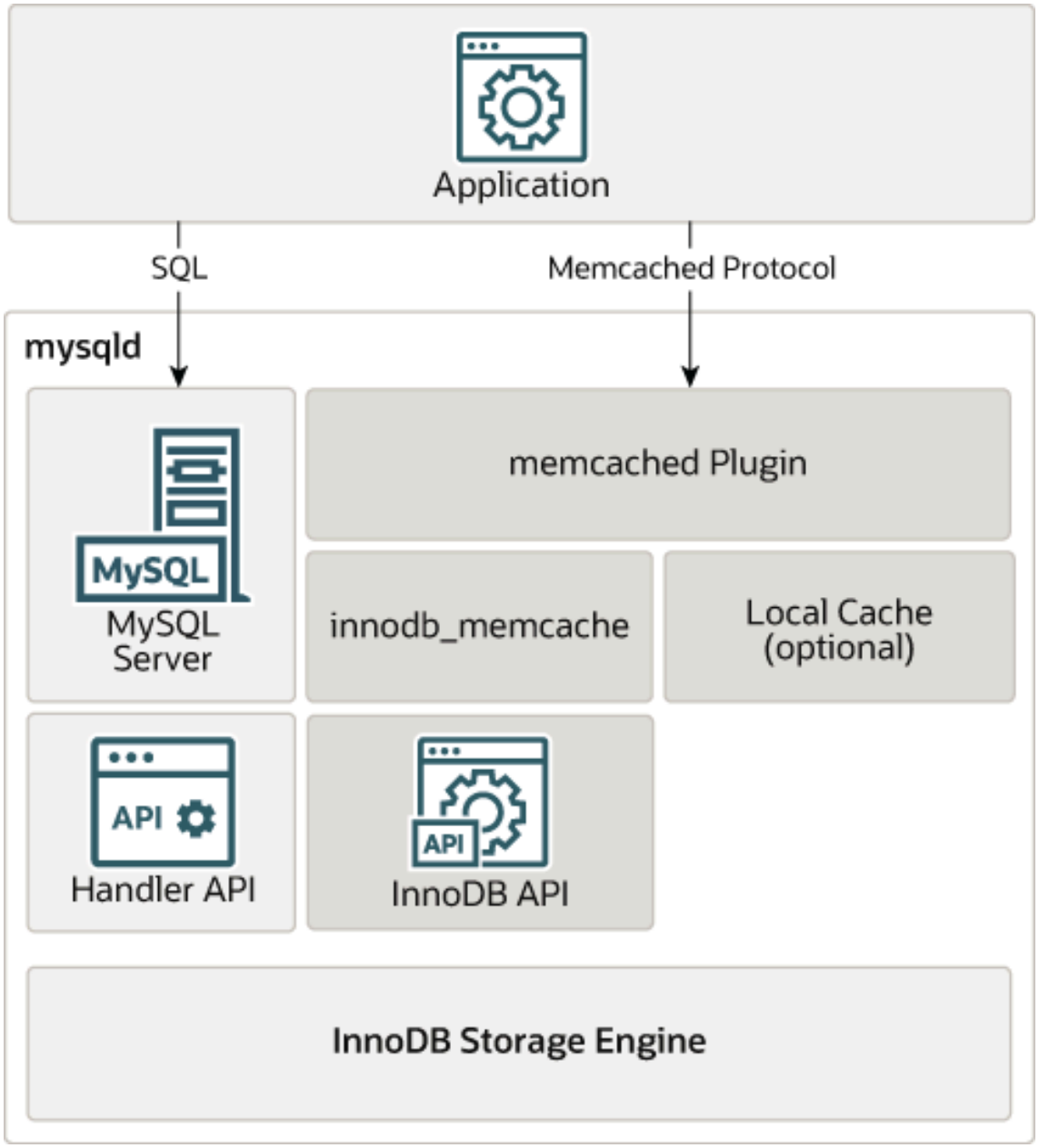
- Because data consistency is enforced by mechanisms used for regular `InnoDB` tables, you do not have to worry about stale `memcached` data or fallback logic to query the database in the case of a missing key.

15.20.2 InnoDB memcached Architecture

The `InnoDB` `memcached` plugin implements `memcached` as a MySQL plugin daemon that accesses the `InnoDB` storage engine directly, bypassing the MySQL SQL layer.

The following diagram illustrates how an application accesses data through the `daemon_memcached` plugin, compared with SQL.

Figure 15.4 MySQL Server with Integrated `memcached` Server



Features of the `daemon_memcached` plugin:

- `memcached` as a daemon plugin of `mysqld`. Both `mysqld` and `memcached` run in the same process space, with very low latency access to data.

- Direct access to `InnoDB` tables, bypassing the SQL parser, the optimizer, and even the Handler API layer.
- Standard `memcached` protocols, including the text-based protocol and the binary protocol. The `daemon_memcached` plugin passes all 55 compatibility tests of the `memcapable` command.
- Multi-column support. You can map multiple columns into the “value” part of the key-value store, with column values delimited by a user-specified separator character.
- By default, the `memcached` protocol is used to read and write data directly to `InnoDB`, letting MySQL manage in-memory caching using the `InnoDB` buffer pool. The default settings represent a combination of high reliability and the fewest surprises for database applications. For example, default settings avoid uncommitted data on the database side, or stale data returned for `memcached get` requests.
- Advanced users can configure the system as a traditional `memcached` server, with all data cached only in the `memcached` engine (memory caching), or use a combination of the “`memcached` engine” (memory caching) and the `InnoDB` `memcached` engine (`InnoDB` as back-end persistent storage).
- Control over how often data is passed back and forth between `InnoDB` and `memcached` operations through the `innodb_api_bk_commit_interval`, `daemon_memcached_r_batch_size`, and `daemon_memcached_w_batch_size` configuration options. Batch size options default to a value of 1 for maximum reliability.
- The ability to specify `memcached` options through the `daemon_memcached_option` configuration parameter. For example, you can change the port that `memcached` listens on, reduce the maximum number of simultaneous connections, change the maximum memory size for a key-value pair, or enable debugging messages for the error log.
- The `innodb_api_trx_level` configuration option controls the transaction isolation level on queries processed by `memcached`. Although `memcached` has no concept of transactions, you can use this option to control how soon `memcached` sees changes caused by SQL statements issued on the table used by the `daemon_memcached` plugin. By default, `innodb_api_trx_level` is set to `READ UNCOMMITTED`.
- The `innodb_api_enable_mdl` option can be used to lock the table at the MySQL level, so that the mapped table cannot be dropped or altered by DDL through the SQL interface. Without the lock, the table can be dropped from the MySQL layer, but kept in `InnoDB` storage until `memcached` or some other user stops using it. “MDL” stands for “metadata locking”.

15.20.3 Setting Up the InnoDB memcached Plugin

This section describes how to set up the `daemon_memcached` plugin on a MySQL server. Because the `memcached` daemon is tightly integrated with the MySQL server to avoid network traffic and minimize latency, you perform this process on each MySQL instance that uses this feature.

Note

Before setting up the `daemon_memcached` plugin, consult Section 15.20.5, “Security Considerations for the InnoDB memcached Plugin” to understand the security procedures required to prevent unauthorized access.

Prerequisites

- The `daemon_memcached` plugin is only supported on Linux, Solaris, and macOS platforms. Other operating systems are not supported.
- When building MySQL from source, you must build with `-DWITH_INNODB_MEMCACHED=ON`. This build option generates two shared libraries in the MySQL plugin directory (`plugin_dir`) that are required to run the `daemon_memcached` plugin:
  - `libmemcached.so`: the `memcached` daemon plugin to MySQL.

  - `innodb_engine.so`: an `InnoDB` API plugin to `memcached`.

- `libevent` must be installed.

  - If you did not build MySQL from source, the `libevent` library is not included in your installation. Use the installation method for your operating system to install `libevent` 1.4.12 or later. For example, depending on the operating system, you might use `apt-get`, `yum`, or `port install`. For example, on Ubuntu Linux, use:

    ```
    sudo apt-get install libevent-dev
    ```

  - If you installed MySQL from a source code release, `libevent` 1.4.12 is bundled with the package and is located at the top level of the MySQL source code directory. If you use the bundled version of `libevent`, no action is required. If you want to use a local system version of `libevent`, you must build MySQL with the `-DWITH_LIBEVENT` build option set to `system` or `yes`.

Installing and Configuring the InnoDB memcached Plugin

1. Configure the `daemon_memcached` plugin so it can interact with `InnoDB` tables by running the `innodb_memcached_config.sql` configuration script, which is located in *`MYSQL_HOME`*`/share`. This script installs the `innodb_memcache` database with three required tables (`cache_policies`, `config_options`, and `containers`). It also installs the `demo_test` sample table in the `test` database.

   ```
   mysql> source MYSQL_HOME/share/innodb_memcached_config.sql
   ```

   Running the `innodb_memcached_config.sql` script is a one-time operation. The tables remain in place if you later uninstall and re-install the `daemon_memcached` plugin.

   ```
   mysql> USE innodb_memcache;
   mysql> SHOW TABLES;
   +---------------------------+
   | Tables_in_innodb_memcache |
   +---------------------------+
   | cache_policies            |
   | config_options            |
   | containers                |
   +---------------------------+

   mysql> USE test;
   mysql> SHOW TABLES;
   +----------------+
   | Tables_in_test |
   +----------------+
   | demo_test      |
   +----------------+
   ```

   Of these tables, the `innodb_memcache.containers` table is the most important. Entries in the `containers` table provide a mapping to `InnoDB` table columns. Each `InnoDB` table used with the `daemon_memcached` plugin requires an entry in the `containers` table.

   The `innodb_memcached_config.sql` script inserts a single entry in the `containers` table that provides a mapping for the `demo_test` table. It also inserts a single row of data into the `demo_test` table. This data allows you to immediately verify the installation after the setup is completed.

   ```
   mysql> SELECT * FROM innodb_memcache.containers\G
   *************************** 1. row ***************************
                     name: aaa
               db_schema: test
                db_table: demo_test
          key_columns: c1
         value_columns: c2
                 flags: c3
            cas_column: c4
   ```

```
    expire_time_column: c5
unique_idx_name_on_key: PRIMARY

mysql> SELECT * FROM test.demo_test;
+----+-----------------+------+------+------+
| c1 | c2              | c3   | c4   | c5   |
+----+-----------------+------+------+------+
| AA | HELLO, HELLO    |    8 |    0 |    0 |
+----+-----------------+------+------+------+
```

For more information about `innodb_memcache` tables and the `demo_test` sample table, see Section 15.20.8, "InnoDB memcached Plugin Internals".

2. Activate the `daemon_memcached` plugin by running the `INSTALL PLUGIN` statement:

```
mysql> INSTALL PLUGIN daemon_memcached soname "libmemcached.so";
```

Once the plugin is installed, it is automatically activated each time the MySQL server is restarted.

Verifying the InnoDB and memcached Setup

To verify the `daemon_memcached` plugin setup, use a `telnet` session to issue `memcached` commands. By default, the `memcached` daemon listens on port 11211.

1. Retrieve data from the `test.demo_test` table. The single row of data in the `demo_test` table has a key value of `AA`.

```
telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
get AA
VALUE AA 8 12
HELLO, HELLO
END
```

2. Insert data using a `set` command.

```
set BB 10 0 16
GOODBYE, GOODBYE
STORED
```

where:

- `set` is the command to store a value
- `BB` is the key
- `10` is a flag for the operation; ignored by `memcached` but may be used by the client to indicate any type of information; specify `0` if unused
- `0` is the expiration time (TTL); specify `0` if unused
- `16` is the length of the supplied value block in bytes
- `GOODBYE, GOODBYE` is the value that is stored

3. Verify that the data inserted is stored in MySQL by connecting to the MySQL server and querying the `test.demo_test` table.

```
mysql> SELECT * FROM test.demo_test;
+----+-----------------+------+------+------+
| c1 | c2              | c3   | c4   | c5   |
+----+-----------------+------+------+------+
| AA | HELLO, HELLO    |    8 |    0 |    0 |
| BB | GOODBYE, GOODBYE |   10 |    1 |    0 |
+----+-----------------+------+------+------+
```

4. Return to the telnet session and retrieve the data that you inserted earlier using key `BB`.

```
get BB
VALUE BB 10 16
GOODBYE, GOODBYE
END
quit
```

If you shut down the MySQL server, which also shuts off the integrated `memcached` server, further attempts to access the `memcached` data fail with a connection error. Normally, the `memcached` data also disappears at this point, and you would require application logic to load the data back into memory when `memcached` is restarted. However, the `InnoDB` `memcached` plugin automates this process for you.

When you restart MySQL, `get` operations once again return the key-value pairs you stored in the earlier `memcached` session. When a key is requested and the associated value is not already in the memory cache, the value is automatically queried from the MySQL `test.demo_test` table.

Creating a New Table and Column Mapping

This example shows how to setup your own `InnoDB` table with the `daemon_memcached` plugin.

1. Create an `InnoDB` table. The table must have a key column with a unique index. The key column of the city table is `city_id`, which is defined as the primary key. The table must also include columns for `flags`, `cas`, and `expiry` values. There may be one or more value columns. The `city` table has three value columns (`name`, `state`, `country`).

Note

There is no special requirement with respect to column names as along as a valid mapping is added to the `innodb_memcache.containers` table.

```
mysql> CREATE TABLE city (
       city_id VARCHAR(32),
       name VARCHAR(1024),
       state VARCHAR(1024),
       country VARCHAR(1024),
       flags INT,
       cas BIGINT UNSIGNED,
       expiry INT,
       primary key(city_id)
       ) ENGINE=InnoDB;
```

2. Add an entry to the `innodb_memcache.containers` table so that the `daemon_memcached` plugin knows how to access the `InnoDB` table. The entry must satisfy the `innodb_memcache.containers` table definition. For a description of each field, see Section 15.20.8, “InnoDB memcached Plugin Internals”.

```
mysql> DESCRIBE innodb_memcache.containers;
+------------------------+--------------+------+-----+---------+-------+
| Field                  | Type         | Null | Key | Default | Extra |
+------------------------+--------------+------+-----+---------+-------+
| name                   | varchar(50)  | NO   | PRI | NULL    |       |
| db_schema              | varchar(250) | NO   |     | NULL    |       |
| db_table               | varchar(250) | NO   |     | NULL    |       |
| key_columns            | varchar(250) | NO   |     | NULL    |       |
| value_columns          | varchar(250) | YES  |     | NULL    |       |
| flags                  | varchar(250) | NO   |     | 0       |       |
| cas_column             | varchar(250) | YES  |     | NULL    |       |
| expire_time_column     | varchar(250) | YES  |     | NULL    |       |
| unique_idx_name_on_key | varchar(250) | NO   |     | NULL    |       |
+------------------------+--------------+------+-----+---------+-------+
```

The `innodb_memcache.containers` table entry for the city table is defined as:

```
mysql> INSERT INTO `innodb_memcache`.`containers` (
```

```
      `name`, `db_schema`, `db_table`, `key_columns`, `value_columns`,
      `flags`, `cas_column`, `expire_time_column`, `unique_idx_name_on_key`)
      VALUES ('default', 'test', 'city', 'city_id', 'name|state|country',
      'flags','cas','expiry','PRIMARY');
```

- `default` is specified for the `containers.name` column to configure the `city` table as the default `InnoDB` table to be used with the `daemon_memcached` plugin.
- Multiple `InnoDB` table columns (`name`, `state`, `country`) are mapped to `containers.value_columns` using a "|" delimiter.
- The `flags`, `cas_column`, and `expire_time_column` fields of the `innodb_memcache.containers` table are typically not significant in applications using the `daemon_memcached` plugin. However, a designated `InnoDB` table column is required for each. When inserting data, specify `0` for these columns if they are unused.

3. After updating the `innodb_memcache.containers` table, restart the `daemon_memcache` plugin to apply the changes.

```
mysql> UNINSTALL PLUGIN daemon_memcached;

mysql> INSTALL PLUGIN daemon_memcached soname "libmemcached.so";
```

4. Using telnet, insert data into the `city` table using a `memcached set` command.

```
telnet localhost 11211
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
set B 0 0 22
BANGALORE|BANGALORE|IN
STORED
```

5. Using MySQL, query the `test.city` table to verify that the data you inserted was stored.

```
mysql> SELECT * FROM test.city;
+---------+-----------+-----------+---------+-------+------+--------+
| city_id | name      | state     | country | flags | cas  | expiry |
+---------+-----------+-----------+---------+-------+------+--------+
| B       | BANGALORE | BANGALORE | IN      |     0 |    3 |      0 |
+---------+-----------+-----------+---------+-------+------+--------+
```

6. Using MySQL, insert additional data into the `test.city` table.

```
mysql> INSERT INTO city VALUES ('C','CHENNAI','TAMIL NADU','IN', 0, 0 ,0);
mysql> INSERT INTO city VALUES ('D','DELHI','DELHI','IN', 0, 0, 0);
mysql> INSERT INTO city VALUES ('H','HYDERABAD','TELANGANA','IN', 0, 0, 0);
mysql> INSERT INTO city VALUES ('M','MUMBAI','MAHARASHTRA','IN', 0, 0, 0);
```

Note

It is recommended that you specify a value of `0` for the `flags`, `cas_column`, and `expire_time_column` fields if they are unused.

7. Using telnet, issue a `memcached get` command to retrieve data you inserted using MySQL.

```
get H
VALUE H 0 22
HYDERABAD|TELANGANA|IN
END
```

Configuring the InnoDB memcached Plugin

Traditional `memcached` configuration options may be specified in a MySQL configuration file or a `mysqld` startup string, encoded in the argument of the `daemon_memcached_option` configuration parameter. `memcached` configuration options take effect when the plugin is loaded, which occurs each time the MySQL server is started.

For example, to make `memcached` listen on port 11222 instead of the default port 11211, specify `-p11222` as an argument of the `daemon_memcached_option` configuration option:

```
mysqld .... --daemon_memcached_option="-p11222"
```

Other `memcached` options can be encoded in the `daemon_memcached_option` string. For example, you can specify options to reduce the maximum number of simultaneous connections, change the maximum memory size for a key-value pair, or enable debugging messages for the error log, and so on.

There are also configuration options specific to the `daemon_memcached` plugin. These include:

- `daemon_memcached_engine_lib_name`: Specifies the shared library that implements the `InnoDB memcached` plugin. The default setting is `innodb_engine.so`.
- `daemon_memcached_engine_lib_path`: The path of the directory containing the shared library that implements the `InnoDB memcached` plugin. The default is NULL, representing the plugin directory.
- `daemon_memcached_r_batch_size`: Defines the batch commit size for read operations (`get`). It specifies the number of `memcached` read operations after which a commit occurs. `daemon_memcached_r_batch_size` is set to 1 by default so that every `get` request accesses the most recently committed data in the `InnoDB` table, whether the data was updated through `memcached` or by SQL. When the value is greater than 1, the counter for read operations is incremented with each `get` call. A `flush_all` call resets both read and write counters.
- `daemon_memcached_w_batch_size`: Defines the batch commit size for write operations (`set`, `replace`, `append`, `prepend`, `incr`, `decr`, and so on). `daemon_memcached_w_batch_size` is set to 1 by default so that no uncommitted data is lost in case of an outage, and so that SQL queries on the underlying table access the most recent data. When the value is greater than 1, the counter for write operations is incremented for each `add`, `set`, `incr`, `decr`, and `delete` call. A `flush_all` call resets both read and write counters.

By default, you do not need to modify `daemon_memcached_engine_lib_name` or `daemon_memcached_engine_lib_path`. You might configure these options if, for example, you want to use a different storage engine for `memcached` (such as the NDB `memcached` engine).

`daemon_memcached` plugin configuration parameters may be specified in the MySQL configuration file or in a `mysqld` startup string. They take effect when you load the `daemon_memcached` plugin.

When making changes to `daemon_memcached` plugin configuration, reload the plugin to apply the changes. To do so, issue the following statements:

```
mysql> UNINSTALL PLUGIN daemon_memcached;

mysql> INSTALL PLUGIN daemon_memcached soname "libmemcached.so";
```

Configuration settings, required tables, and data are preserved when the plugin is restarted.

For additional information about enabling and disabling plugins, see Section 5.6.1, “Installing and Uninstalling Plugins”.

15.20.4 InnoDB memcached Multiple get and Range Query Support

The `daemon_memcached` plugin supports multiple get operations (fetching multiple key-value pairs in a single `memcached` query) and range queries.

Multiple get Operations

The ability to fetch multiple key-value pairs in a single `memcached` query improves read performance by reducing communication traffic between the client and server. For `InnoDB`, it means fewer transactions and open-table operations.

The following example demonstrates multiple-get support. The example uses the `test.city` table described in Creating a New Table and Column Mapping.

```
mysql> USE test;
mysql> SELECT * FROM test.city;
+---------+-----------+-------------+---------+-------+------+--------+
| city_id | name      | state       | country | flags | cas  | expiry |
+---------+-----------+-------------+---------+-------+------+--------+
| B       | BANGALORE | BANGALORE   | IN      |     0 |    1 |      0 |
| C       | CHENNAI   | TAMIL NADU  | IN      |     0 |    0 |      0 |
| D       | DELHI     | DELHI       | IN      |     0 |    0 |      0 |
| H       | HYDERABAD | TELANGANA   | IN      |     0 |    0 |      0 |
| M       | MUMBAI    | MAHARASHTRA | IN      |     0 |    0 |      0 |
+---------+-----------+-------------+---------+-------+------+--------+
```

Run a `get` command to retrieve all values from the `city` table. The results are returned in a key-value pair sequence.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
get B C D H M
VALUE B 0 22
BANGALORE|BANGALORE|IN
VALUE C 0 21
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
VALUE M 0 21
MUMBAI|MAHARASHTRA|IN
END
```

When retrieving multiple values in a single `get` command, you can switch tables (using *`@@containers.name`* notation) to retrieve the value for the first key, but you cannot switch tables for subsequent keys. For example, the table switch in this example is valid:

```
get @@aaa.AA BB
VALUE @@aaa.AA 8 12
HELLO, HELLO
VALUE BB 10 16
GOODBYE, GOODBYE
END
```

Attempting to switch tables again in the same `get` command to retrieve a key value from a different table is not supported.

There is no limit the number of keys that can be retrieved by a multiple get operation, but there is a 128MB memory limit for storing the result.

Range Queries

For range queries, the `daemon_memcached` plugin supports the following comparison operators: `<`, `>`, `<=`, `>=`. An operator must be preceded by an `@` symbol. When a range query finds multiple matching key-value pairs, results are returned in a key-value pair sequence.

The following examples demonstrate range query support. The examples use the `test.city` table described in Creating a New Table and Column Mapping.

```
mysql> SELECT * FROM test.city;
+---------+-----------+-------------+---------+-------+------+--------+
| city_id | name      | state       | country | flags | cas  | expiry |
+---------+-----------+-------------+---------+-------+------+--------+
| B       | BANGALORE | BANGALORE   | IN      |     0 |    1 |      0 |
| C       | CHENNAI   | TAMIL NADU  | IN      |     0 |    0 |      0 |
| D       | DELHI     | DELHI       | IN      |     0 |    0 |      0 |
```

```
| H       | HYDERABAD | TELANGANA   | IN      |     0 |    0 |      0 |
| M       | MUMBAI    | MAHARASHTRA | IN      |     0 |    0 |      0 |
+---------+-----------+-------------+---------+-------+------+--------+
```

Open a telnet session:

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
```

To get all values greater than `B`, enter `get @>B`:

```
get @>B
VALUE C 0 21
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
VALUE M 0 21
MUMBAI|MAHARASHTRA|IN
END
```

To get all values less than `M`, enter `get @<M`:

```
get @<M
VALUE B 0 22
BANGALORE|BANGALORE|IN
VALUE C 0 21
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
END
```

To get all values less than and including `M`, enter `get @<=M`:

```
get @<=M
VALUE B 0 22
BANGALORE|BANGALORE|IN
VALUE C 0 21
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
VALUE M 0 21
MUMBAI|MAHARASHTRA|IN
```

To get values greater than `B` but less than `M`, enter `get @>B@<M`:

```
get @>B@<M
VALUE C 0 21
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
END
```

A maximum of two comparison operators can be parsed, one being either a 'less than' (`@<`) or 'less than or equal to' (`@<=`) operator, and the other being either a 'greater than' (`@>`) or 'greater than or equal to' (`@>=`) operator. Any additional operators are assumed to be part of the key. For example, if you issue a `get` command with three operators, the third operator (`@>C`) is treated as part of the key, and the `get` command searches for values smaller than `M` and greater than `B@>C`.

```
get @<M@>B@>C
VALUE C 0 21
```

```
CHENNAI|TAMIL NADU|IN
VALUE D 0 14
DELHI|DELHI|IN
VALUE H 0 22
HYDERABAD|TELANGANA|IN
```

15.20.5 Security Considerations for the InnoDB memcached Plugin

Caution

Consult this section before deploying the `daemon_memcached` plugin on a production server, or even on a test server if the MySQL instance contains sensitive data.

Because `memcached` does not use an authentication mechanism by default, and the optional SASL authentication is not as strong as traditional DBMS security measures, only keep non-sensitive data in the MySQL instance that uses the `daemon_memcached` plugin, and wall off any servers that use this configuration from potential intruders. Do not allow `memcached` access to these servers from the Internet; only allow access from within a firewalled intranet, ideally from a subnet whose membership you can restrict.

Password-Protecting memcached Using SASL

SASL support provides the capability to protect your MySQL database from unauthenticated access through `memcached` clients. This section explains how to enable SASL with the `daemon_memcached` plugin. The steps are almost identical to those performed to enabled SASL for a traditional `memcached` server.

SASL stands for “Simple Authentication and Security Layer”, a standard for adding authentication support to connection-based protocols. `memcached` added SASL support in version 1.4.3.

SASL authentication is only supported with the binary protocol.

`memcached` clients are only able to access `InnoDB` tables that are registered in the `innodb_memcache.containers` table. Even though a DBA can place access restrictions on such tables, access through `memcached` applications cannot be controlled. For this reason, SASL support is provided to control access to `InnoDB` tables associated with the `daemon_memcached` plugin.

The following section shows how to build, enable, and test an SASL-enabled `daemon_memcached` plugin.

Building and Enabling SASL with the InnoDB memcached Plugin

By default, an SASL-enabled `daemon_memcached` plugin is not included in MySQL release packages, since an SASL-enabled `daemon_memcached` plugin requires building `memcached` with SASL libraries. To enable SASL support, download the MySQL source and rebuild the `daemon_memcached` plugin after downloading the SASL libraries:

1. Install the SASL development and utility libraries. For example, on Ubuntu, use `apt-get` to obtain the libraries:

   ```
   sudo apt-get -f install libsasl2-2 sasl2-bin libsasl2-2 libsasl2-dev libsasl2-modules
   ```

2. Build the `daemon_memcached` plugin shared libraries with SASL capability by adding `ENABLE_MEMCACHED_SASL=1` to your `cmake` options. `memcached` also provides *simple cleartext password support*, which facilitates testing. To enable simple cleartext password support, specify the `ENABLE_MEMCACHED_SASL_PWDB=1` `cmake` option.

   In summary, add following three `cmake` options:

   ```
   cmake ... -DWITH_INNODB_MEMCACHED=1 -DENABLE_MEMCACHED_SASL=1 -DENABLE_MEMCACHED_SASL_PWDB=1
   ```

3. Install the `daemon_memcached` plugin, as described in Section 15.20.3, “Setting Up the InnoDB memcached Plugin”.

4. Configure a user name and password file. (This example uses `memcached` simple cleartext password support.)

   a. In a file, create a user named `testname` and define the password as `testpasswd`:

   ```
   echo "testname:testpasswd:::::::" >/home/jy/memcached-sasl-db
   ```

   b. Configure the `MEMCACHED_SASL_PWDB` environment variable to inform `memcached` of the user name and password file:

   ```
   export MEMCACHED_SASL_PWDB=/home/jy/memcached-sasl-db
   ```

   c. Inform `memcached` that a cleartext password is used:

   ```
   echo "mech_list: plain" > /home/jy/work2/msasl/clients/memcached.conf
   export SASL_CONF_PATH=/home/jy/work2/msasl/clients
   ```

5. Enable SASL by restarting the MySQL server with the `memcached -S` option encoded in the `daemon_memcached_option` configuration parameter:

   ```
   mysqld ... --daemon_memcached_option="-S"
   ```

6. To test the setup, use an SASL-enabled client such as SASL-enabled libmemcached.

   ```
   memcp --servers=localhost:11211 --binary  --username=testname
     --password=password myfile.txt

   memcat --servers=localhost:11211 --binary --username=testname
     --password=password myfile.txt
   ```

   If you specify an incorrect user name or password, the operation is rejected with a `memcache error AUTHENTICATION FAILURE` message. In this case, examine the cleartext password set in the `memcached-sasl-db` file to verify that the credentials you supplied are correct.

There are other methods to test SASL authentication with `memcached`, but the method described above is the most straightforward.

15.20.6 Writing Applications for the InnoDB memcached Plugin

Typically, writing an application for the `InnoDB memcached` plugin involves some degree of rewriting or adapting existing code that uses MySQL or the `memcached` API.

- With the `daemon_memcached` plugin, instead of many traditional `memcached` servers running on low-powered machines, you have the same number of `memcached` servers as MySQL servers, running on relatively high-powered machines with substantial disk storage and memory. You might reuse some existing code that works with the `memcached` API, but adaptation is likely required due to the different server configuration.

- The data stored through the `daemon_memcached` plugin goes into `VARCHAR`, `TEXT`, or `BLOB` columns, and must be converted to do numeric operations. You can perform the conversion on the application side, or by using the `CAST()` function in queries.

- Coming from a database background, you might be used to general-purpose SQL tables with many columns. The tables accessed by `memcached` code likely have only a few or even a single column holding data values.

- You might adapt parts of your application that perform single-row queries, inserts, updates, or deletes, to improve performance in critical sections of code. Both queries (read) and DML (write)

operations can be substantially faster when performed through the `InnoDB` `memcached` interface. The performance improvement for writes is typically greater than the performance improvement for reads, so you might focus on adapting code that performs logging or records interactive choices on a website.

The following sections explore these points in more detail.

15.20.6.1 Adapting an Existing MySQL Schema for the InnoDB memcached Plugin

Consider these aspects of `memcached` applications when adapting an existing MySQL schema or application to use the `daemon_memcached` plugin:

- `memcached` keys cannot contain spaces or newlines, because these characters are used as separators in the ASCII protocol. If you are using lookup values that contain spaces, transform or hash them into values without spaces before using them as keys in calls to `add()`, `set()`, `get()`, and so on. Although theoretically these characters are allowed in keys in programs that use the binary protocol, you should restrict the characters used in keys to ensure compatibility with a broad range of clients.
- If there is a short numeric primary key column in an `InnoDB` table, use it as the unique lookup key for `memcached` by converting the integer to a string value. If the `memcached` server is used for multiple applications, or with more than one `InnoDB` table, consider modifying the name to ensure that it is unique. For example, prepend the table name, or the database name and the table name, before the numeric value.

Note

The `daemon_memcached` plugin supports inserts and reads on mapped `InnoDB` tables that have an `INTEGER` defined as the primary key.

- You cannot use a partitioned table for data queried or stored using `memcached`.
- The `memcached` protocol passes numeric values around as strings. To store numeric values in the underlying `InnoDB` table, to implement counters that can be used in SQL functions such as `SUM()` or `AVG()`, for example:
  - Use `VARCHAR` columns with enough characters to hold all the digits of the largest expected number (and additional characters if appropriate for the negative sign, decimal point, or both).
  - In any query that performs arithmetic using column values, use the `CAST()` function to convert the values from string to integer, or to some other numeric type. For example:

```
# Alphabetic entries are returned as zero.

SELECT CAST(c2 as unsigned integer) FROM demo_test;

# Since there could be numeric values of 0, can't disqualify them.
# Test the string values to find the ones that are integers, and average only those.

SELECT AVG(cast(c2 as unsigned integer)) FROM demo_test
  WHERE c2 BETWEEN '0' and '9999999999';

# Views let you hide the complexity of queries. The results are already converted;
# no need to repeat conversion functions and WHERE clauses each time.

CREATE VIEW numbers AS SELECT c1 KEY, CAST(c2 AS UNSIGNED INTEGER) val
  FROM demo_test WHERE c2 BETWEEN '0' and '9999999999';
SELECT SUM(val) FROM numbers;
```

Note

Any alphabetic values in the result set are converted into 0 by the call to `CAST()`. When using functions such as `AVG()`, which depend on the

number of rows in the result set, include `WHERE` clauses to filter out non-numeric values.

- If the `InnoDB` column used as a key could have values longer than 250 bytes, hash the value to less than 250 bytes.
- To use an existing table with the `daemon_memcached` plugin, define an entry for it in the `innodb_memcache.containers` table. To make that table the default for all `memcached` requests, specify a value of `default` in the `name` column, then restart the MySQL server to make the change take effect. If you use multiple tables for different classes of `memcached` data, set up multiple entries in the `innodb_memcache.containers` table with `name` values of your choice, then issue a `memcached` request in the form of `get @@name` or `set @@name` within the application to specify the table to be used for subsequent `memcached` requests.

  For an example of using a table other than the predefined `test.demo_test` table, see Example 15.13, "Using Your Own Table with an InnoDB memcached Application". For the required table layout, see Section 15.20.8, "InnoDB memcached Plugin Internals".
- To use multiple `InnoDB` table column values with `memcached` key-value pairs, specify column names separated by comma, semicolon, space, or pipe characters in the `value_columns` field of the `innodb_memcache.containers` entry for the `InnoDB` table. For example, specify `col1,col2,col3` or `col1|col2|col3` in the `value_columns` field.

  Concatenate the column values into a single string using the pipe character as a separator before passing the string to `memcached` `add` or `set` calls. The string is unpacked automatically into the correct column. Each `get` call returns a single string containing the column values that is also delimited by the pipe character. You can unpack the values using the appropriate application language syntax.

Example 15.13 Using Your Own Table with an InnoDB memcached Application

This example shows how to use your own table with a sample Python application that uses `memcached` for data manipulation.

The example assumes that the `daemon_memcached` plugin is installed as described in Section 15.20.3, "Setting Up the InnoDB memcached Plugin". It also assumes that your system is configured to run a Python script that uses the `python-memcache` module.

1. Create the `multicol` table which stores country information including population, area, and driver side data (`'R'` for right and `'L'` for left).

```
mysql> USE test;

mysql> CREATE TABLE `multicol` (
        `country` varchar(128) NOT NULL DEFAULT '',
        `population` varchar(10) DEFAULT NULL,
        `area_sq_km` varchar(9) DEFAULT NULL,
        `drive_side` varchar(1) DEFAULT NULL,
        `c3` int(11) DEFAULT NULL,
        `c4` bigint(20) unsigned DEFAULT NULL,
        `c5` int(11) DEFAULT NULL,
       PRIMARY KEY (`country`)
       ) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

2. Insert a record into the `innodb_memcache.containers` table so that the `daemon_memcached` plugin can access the `multicol` table.

```
mysql> INSERT INTO innodb_memcache.containers
       (name,db_schema,db_table,key_columns,value_columns,flags,cas_column,
       expire_time_column,unique_idx_name_on_key)
       VALUES
       ('bbb','test','multicol','country','population,area_sq_km,drive_side',
       'c3','c4','c5','PRIMARY');
```

```
mysql> COMMIT;
```

- The `innodb_memcache.containers` record for the `multicol` table specifies a `name` value of `'bbb'`, which is the table identifier.

  Note

  If a single `InnoDB` table is used for all `memcached` applications, the `name` value can be set to `default` to avoid using `@@` notation to switch tables.

- The `db_schema` column is set to `test`, which is the name of the database where the `multicol` table resides.
- The `db_table` column is set to `multicol`, which is the name of the `InnoDB` table.
- `key_columns` is set to the unique `country` column. The `country` column is defined as the primary key in the `multicol` table definition.
- Rather than a single `InnoDB` table column to hold a composite data value, data is divided among three table columns (`population`, `area_sq_km`, and `drive_side`). To accommodate multiple value columns, a comma-separated list of columns is specified in the `value_columns` field. The columns defined in the `value_columns` field are the columns used when storing or retrieving values.
- Values for the `flags`, `expire_time`, and `cas_column` fields are based on values used in the `demo.test` sample table. These fields are typically not significant in applications that use the `daemon_memcached` plugin because MySQL keeps data synchronized, and there is no need to worry about data expiring or becoming stale.
- The `unique_idx_name_on_key` field is set to `PRIMARY`, which refers to the primary index defined on the unique `country` column in the `multicol` table.

3. Copy the sample Python application into a file. In this example, the sample script is copied to a file named `multicol.py`.

   The sample Python application inserts data into the `multicol` table and retrieves data for all keys, demonstrating how to access an `InnoDB` table through the `daemon_memcached` plugin.

```
import sys, os
import memcache

def connect_to_memcached():
  memc = memcache.Client(['127.0.0.1:11211'], debug=0);
  print "Connected to memcached."
  return memc

def banner(message):
  print
  print "=" * len(message)
  print message
  print "=" * len(message)

country_data = [
("Canada","34820000","9984670","R"),
("USA","314242000","9826675","R"),
("Ireland","6399152","84421","L"),
("UK","62262000","243610","L"),
("Mexico","113910608","1972550","R"),
("Denmark","5543453","43094","R"),
("Norway","5002942","385252","R"),
("UAE","8264070","83600","R"),
("India","1210193422","3287263","L"),
("China","1347350000","9640821","R"),
]

def switch_table(memc,table):
```

```
  key = "@@" + table
  print "Switching default table to '" + table + "' by issuing GET for '" + key + "'."
  result = memc.get(key)

def insert_country_data(memc):
  banner("Inserting initial data via memcached interface")
  for item in country_data:
    country = item[0]
    population = item[1]
    area = item[2]
    drive_side = item[3]

    key = country
    value = "|".join([population,area,drive_side])
    print "Key = " + key
    print "Value = " + value

    if memc.add(key,value):
      print "Added new key, value pair."
    else:
      print "Updating value for existing key."
      memc.set(key,value)

def query_country_data(memc):
  banner("Retrieving data for all keys (country names)")
  for item in country_data:
    key = item[0]
    result = memc.get(key)
    print "Here is the result retrieved from the database for key " + key + ":"
    print result
    (m_population, m_area, m_drive_side) = result.split("|")
    print "Unpacked population value: " + m_population
    print "Unpacked area value      : " + m_area
    print "Unpacked drive side value: " + m_drive_side

if __name__ == '__main__':

  memc = connect_to_memcached()
  switch_table(memc,"bbb")
  insert_country_data(memc)
  query_country_data(memc)

  sys.exit(0)
```

Sample Python application notes:

- No database authorization is required to run the application, since data manipulation is performed through the `memcached` interface. The only required information is the port number on the local system where the `memcached` daemon listens.
- To make sure the application uses the `multicol` table, the `switch_table()` function is called, which performs a dummy `get` or `set` request using `@@` notation. The `name` value in the request is `bbb`, which is the `multicol` table identifier defined in the `innodb_memcache.containers.name` field.

  A more descriptive `name` value might be used in a real-world application. This example simply illustrates that a table identifier is specified rather than the table name in `get @@...` requests.
- The utility functions used to insert and query data demonstrate how to turn a Python data structure into pipe-separated values for sending data to MySQL with `add` or `set` requests, and how to unpack the pipe-separated values returned by `get` requests. This extra processing is only required when mapping a single `memcached` value to multiple MySQL table columns.

4. Run the sample Python application.

```
$> python multicol.py
```

If successful, the sample application returns this output:

```
Connected to memcached.
Switching default table to 'bbb' by issuing GET for '@@bbb'.

============================================
Inserting initial data via memcached interface
============================================
Key = Canada
Value = 34820000|9984670|R
Added new key, value pair.
Key = USA
Value = 314242000|9826675|R
Added new key, value pair.
Key = Ireland
Value = 6399152|84421|L
Added new key, value pair.
Key = UK
Value = 62262000|243610|L
Added new key, value pair.
Key = Mexico
Value = 113910608|1972550|R
Added new key, value pair.
Key = Denmark
Value = 5543453|43094|R
Added new key, value pair.
Key = Norway
Value = 5002942|385252|R
Added new key, value pair.
Key = UAE
Value = 8264070|83600|R
Added new key, value pair.
Key = India
Value = 1210193422|3287263|L
Added new key, value pair.
Key = China
Value = 1347350000|9640821|R
Added new key, value pair.

==========================================
Retrieving data for all keys (country names)
==========================================
Here is the result retrieved from the database for key Canada:
34820000|9984670|R
Unpacked population value: 34820000
Unpacked area value      : 9984670
Unpacked drive side value: R
Here is the result retrieved from the database for key USA:
314242000|9826675|R
Unpacked population value: 314242000
Unpacked area value      : 9826675
Unpacked drive side value: R
Here is the result retrieved from the database for key Ireland:
6399152|84421|L
Unpacked population value: 6399152
Unpacked area value      : 84421
Unpacked drive side value: L
Here is the result retrieved from the database for key UK:
62262000|243610|L
Unpacked population value: 62262000
Unpacked area value      : 243610
Unpacked drive side value: L
Here is the result retrieved from the database for key Mexico:
113910608|1972550|R
Unpacked population value: 113910608
Unpacked area value      : 1972550
Unpacked drive side value: R
Here is the result retrieved from the database for key Denmark:
```

```
5543453|43094|R
Unpacked population value: 5543453
Unpacked area value      : 43094
Unpacked drive side value: R
Here is the result retrieved from the database for key Norway:
5002942|385252|R
Unpacked population value: 5002942
Unpacked area value      : 385252
Unpacked drive side value: R
Here is the result retrieved from the database for key UAE:
8264070|83600|R
Unpacked population value: 8264070
Unpacked area value      : 83600
Unpacked drive side value: R
Here is the result retrieved from the database for key India:
1210193422|3287263|L
Unpacked population value: 1210193422
Unpacked area value      : 3287263
Unpacked drive side value: L
Here is the result retrieved from the database for key China:
1347350000|9640821|R
Unpacked population value: 1347350000
Unpacked area value      : 9640821
Unpacked drive side value: R
```

5. Query the `innodb_memcache.containers` table to view the record you inserted earlier for the `multicol` table. The first record is the sample entry for the `demo_test` table that is created during the initial `daemon_memcached` plugin setup. The second record is the entry you inserted for the `multicol` table.

```
mysql> SELECT * FROM innodb_memcache.containers\G
*************************** 1. row ***************************
                  name: aaa
             db_schema: test
              db_table: demo_test
           key_columns: c1
         value_columns: c2
                 flags: c3
            cas_column: c4
    expire_time_column: c5
unique_idx_name_on_key: PRIMARY
*************************** 2. row ***************************
                  name: bbb
             db_schema: test
              db_table: multicol
           key_columns: country
         value_columns: population,area_sq_km,drive_side
                 flags: c3
            cas_column: c4
    expire_time_column: c5
unique_idx_name_on_key: PRIMARY
```

6. Query the `multicol` table to view data inserted by the sample Python application. The data is available for MySQL queries, which demonstrates how the same data can be accessed using SQL or through applications (using the appropriate MySQL Connector or API).

```
mysql> SELECT * FROM test.multicol;
+---------+------------+------------+------------+------+------+------+
| country | population | area_sq_km | drive_side | c3   | c4   | c5   |
+---------+------------+------------+------------+------+------+------+
| Canada  | 34820000   | 9984670    | R          |    0 |   11 |    0 |
| China   | 1347350000 | 9640821    | R          |    0 |   20 |    0 |
| Denmark | 5543453    | 43094      | R          |    0 |   16 |    0 |
| India   | 1210193422 | 3287263    | L          |    0 |   19 |    0 |
| Ireland | 6399152    | 84421      | L          |    0 |   13 |    0 |
| Mexico  | 113910608  | 1972550    | R          |    0 |   15 |    0 |
| Norway  | 5002942    | 385252     | R          |    0 |   17 |    0 |
| UAE     | 8264070    | 83600      | R          |    0 |   18 |    0 |
| UK      | 62262000   | 243610     | L          |    0 |   14 |    0 |
| USA     | 314242000  | 9826675    | R          |    0 |   12 |    0 |
+---------+------------+------------+------------+------+------+------+
```

Note

Always allow sufficient size to hold necessary digits, decimal points, sign characters, leading zeros, and so on when defining the length for columns that are treated as numbers. Too-long values in a string column such as a `VARCHAR` are truncated by removing some characters, which could produce nonsensical numeric values.

7. Optionally, run report-type queries on the `InnoDB` table that stores the `memcached` data.

   You can produce reports through SQL queries, performing calculations and tests across any columns, not just the `country` key column. (Because the following examples use data from only a few countries, the numbers are for illustration purposes only.) The following queries return the average population of countries where people drive on the right, and the average size of countries whose names start with “U”:

```
mysql> SELECT AVG(population) FROM multicol WHERE drive_side = 'R';
+-------------------+
| avg(population)   |
+-------------------+
| 261304724.7142857 |
+-------------------+

mysql> SELECT SUM(area_sq_km) FROM multicol WHERE country LIKE 'U%';
+-----------------+
| sum(area_sq_km) |
+-----------------+
|        10153885 |
+-----------------+
```

   Because the `population` and `area_sq_km` columns store character data rather than strongly typed numeric data, functions such as `AVG()` and `SUM()` work by converting each value to a number first. This approach *does not work* for operators such as `<` or `>`, for example, when comparing character-based values, `9 > 1000`, which is not expected from a clause such as `ORDER BY population DESC`. For the most accurate type treatment, perform queries against views that cast numeric columns to the appropriate types. This technique lets you issue simple `SELECT *` queries from database applications, while ensuring that casting, filtering, and ordering is correct. The following example shows a view that can be queried to find the top three countries in descending order of population, with the results reflecting the latest data in the `multicol` table, and with population and area figures treated as numbers:

```
mysql> CREATE VIEW populous_countries AS
       SELECT
       country,
       cast(population as unsigned integer) population,
       cast(area_sq_km as unsigned integer) area_sq_km,
       drive_side FROM multicol
       ORDER BY CAST(population as unsigned integer) DESC
       LIMIT 3;

mysql> SELECT * FROM populous_countries;
+---------+------------+------------+------------+
| country | population | area_sq_km | drive_side |
+---------+------------+------------+------------+
| China   | 1347350000 |    9640821 | R          |
| India   | 1210193422 |    3287263 | L          |
| USA     |  314242000 |    9826675 | R          |
+---------+------------+------------+------------+

mysql> DESC populous_countries;
+------------+---------------------+------+-----+---------+-------+
| Field      | Type                | Null | Key | Default | Extra |
+------------+---------------------+------+-----+---------+-------+
| country    | varchar(128)        | NO   |     |         |       |
| population | bigint(10) unsigned | YES  |     | NULL    |       |
| area_sq_km | int(9) unsigned     | YES  |     | NULL    |       |
```

```
| drive_side | varchar(1)          | YES  |     | NULL    |       |
+------------+---------------------+------+-----+---------+-------+
```

15.20.6.2 Adapting a memcached Application for the InnoDB memcached Plugin

Consider these aspects of MySQL and `InnoDB` tables when adapting existing `memcached` applications to use the `daemon_memcached` plugin:

- If there are key values longer than a few bytes, it may be more efficient to use a numeric auto-increment column as the primary key of the `InnoDB` table, and to create a unique secondary index on the column that contains the `memcached` key values. This is because `InnoDB` performs best for large-scale insertions if primary key values are added in sorted order (as they are with auto-increment values). Primary key values are included in secondary indexes, which takes up unnecessary space if the primary key is a long string value.
- If you store several different classes of information using `memcached`, consider setting up a separate `InnoDB` table for each type of data. Define additional table identifiers in the `innodb_memcache.containers` table, and use the `@@table_id.key` notation to store and retrieve items from different tables. Physically dividing different types of information allows you tune the characteristics of each table for optimum space utilization, performance, and reliability. For example, you might enable compression for a table that holds blog posts, but not for a table that holds thumbnail images. You might back up one table more frequently than another because it holds critical data. You might create additional secondary indexes on tables that are frequently used to generate reports using SQL.
- Preferably, configure a stable set of table definitions for use with the `daemon_memcached` plugin, and leave the tables in place permanently. Changes to the `innodb_memcache.containers` table take effect the next time the `innodb_memcache.containers` table is queried. Entries in the containers table are processed at startup, and are consulted whenever an unrecognized table identifier (as defined by `containers.name`) is requested using `@@` notation. Thus, new entries are visible as soon as you use the associated table identifier, but changes to existing entries require a server restart before they take effect.
- When you use the default `innodb_only` caching policy, calls to `add()`, `set()`, `incr()`, and so on can succeed but still trigger debugging messages such as `while expecting 'STORED', got unexpected response 'NOT_STORED`. Debug messages occur because new and updated values are sent directly to the `InnoDB` table without being saved in the memory cache, due to the `innodb_only` caching policy.

15.20.6.3 Tuning InnoDB memcached Plugin Performance

Because using `InnoDB` in combination with `memcached` involves writing all data to disk, whether immediately or sometime later, raw performance is expected to be somewhat slower than using `memcached` by itself. When using the `InnoDB` `memcached` plugin, focus tuning goals for `memcached` operations on achieving better performance than equivalent SQL operations.

Benchmarks suggest that queries and DML operations (inserts, updates, and deletes) that use the `memcached` interface are faster than traditional SQL. DML operations typically see a larger improvements. Therefore, consider adapting write-intensive applications to use the `memcached` interface first. Also consider prioritizing adaptation of write-intensive applications that use fast, lightweight mechanisms that lack reliability.

Adapting SQL Queries

The types of queries that are most suited to simple `GET` requests are those with a single clause or a set of `AND` conditions in the `WHERE` clause:

```
SQL:
SELECT col FROM tbl WHERE key = 'key_value';
```

```
memcached:
get key_value

SQL:
SELECT col FROM tbl WHERE col1 = val1 and col2 = val2 and col3 = val3;

memcached:
# Since you must always know these 3 values to look up the key,
# combine them into a unique string and use that as the key
# for all ADD, SET, and GET operations.
key_value = val1 + ":" + val2 + ":" + val3
get key_value

SQL:
SELECT 'key exists!' FROM tbl
  WHERE EXISTS (SELECT col1 FROM tbl WHERE KEY = 'key_value') LIMIT 1;

memcached:
# Test for existence of key by asking for its value and checking if the call succeeds,
# ignoring the value itself. For existence checking, you typically only store a very
# short value such as "1".
get key_value
```

Using System Memory

For best performance, deploy the `daemon_memcached` plugin on machines that are configured as typical database servers, where the majority of system RAM is devoted to the `InnoDB` buffer pool, through the `innodb_buffer_pool_size` configuration option. For systems with multi-gigabyte buffer pools, consider raising the value of `innodb_buffer_pool_instances` for maximum throughput when most operations involve data that is already cached in memory.

Reducing Redundant I/O

`InnoDB` has a number of settings that let you choose the balance between high reliability, in case of a crash, and the amount of I/O overhead during high write workloads. For example, consider setting the `innodb_doublewrite` to `0` and `innodb_flush_log_at_trx_commit` to `2`. Measure performance with different `innodb_flush_method` settings.

For other ways to reduce or tune I/O for table operations, see Section 8.5.8, “Optimizing InnoDB Disk I/O”.

Reducing Transactional Overhead

A default value of 1 for `daemon_memcached_r_batch_size` and `daemon_memcached_w_batch_size` is intended for maximum reliability of results and safety of stored or updated data.

Depending on the type of application, you might increase one or both of these settings to reduce the overhead of frequent commit operations. On a busy system, you might increase `daemon_memcached_r_batch_size`, knowing that changes to data made through SQL may not become visible to `memcached` immediately (that is, until *`N`* more `get` operations are processed). When processing data where every write operation must be reliably stored, leave `daemon_memcached_w_batch_size` set to `1`. Increase the setting when processing large numbers of updates intended only for statistical analysis, where losing the last *`N`* updates in an unexpected exit is an acceptable risk.

For example, imagine a system that monitors traffic crossing a busy bridge, recording data for approximately 100,000 vehicles each day. If the application counts different types of vehicles to analyze traffic patterns, changing `daemon_memcached_w_batch_size` from `1` to `100` reduces I/O overhead for commit operations by 99%. In case of an outage, a maximum of 100 records are lost, which may be an acceptable margin of error. If instead the application performed automated toll collection for each car, you would set `daemon_memcached_w_batch_size` to `1` to ensure that each toll record is immediately saved to disk.

Because of the way InnoDB organizes memcached key values on disk, if you have a large number of keys to create, it may be faster to sort the data items by key value in the application and add them in sorted order, rather than create keys in arbitrary order.

The memslap command, which is part of the regular memcached distribution but not included with the daemon_memcached plugin, can be useful for benchmarking different configurations. It can also be used to generate sample key-value pairs to use in your own benchmarks.

15.20.6.4 Controlling Transactional Behavior of the InnoDB memcached Plugin

Unlike traditional memcached, the daemon_memcached plugin allows you to control durability of data values produced through calls to add, set, incr, and so on. By default, data written through the memcached interface is stored to disk, and calls to get return the most recent value from disk. Although the default behavior does not offer the best possible raw performance, it is still fast compared to the SQL interface for InnoDB tables.

As you gain experience using the daemon_memcached plugin, you can consider relaxing durability settings for non-critical classes of data, at the risk of losing some updated values in the event of an outage, or returning data that is slightly out-of-date.

Frequency of Commits

One tradeoff between durability and raw performance is how frequently new and changed data is committed. If data is critical, is should be committed immediately so that it is safe in case of an unexpected exit or outage. If data is less critical, such as counters that are reset after an unexpected exit or logging data that you can afford to lose, you might prefer higher raw throughput that is available with less frequent commits.

When a memcached operation inserts, updates, or deletes data in the underlying InnoDB table, the change might be committed to the InnoDB table instantly (if daemon_memcached_w_batch_size=1) or some time later (if the daemon_memcached_w_batch_size value is greater than 1). In either case, the change cannot be rolled back. If you increase the value of daemon_memcached_w_batch_size to avoid high I/O overhead during busy times, commits could become infrequent when the workload decreases. As a safety measure, a background thread automatically commits changes made through the memcached API at regular intervals. The interval is controlled by the innodb_api_bk_commit_interval configuration option, which has a default setting of 5 seconds.

When a memcached operation inserts or updates data in the underlying InnoDB table, the changed data is immediately visible to other memcached requests because the new value remains in the memory cache, even if it is not yet committed on the MySQL side.

Transaction Isolation

When a memcached operation such as get or incr causes a query or DML operation on the underlying InnoDB table, you can control whether the operation sees the very latest data written to the table, only data that has been committed, or other variations of transaction isolation level. Use the innodb_api_trx_level configuration option to control this feature. The numeric values specified for this option correspond to isolation levels such as REPEATABLE READ. See the description of the innodb_api_trx_level option for information about other settings.

A strict isolation level ensures that data you retrieve is not rolled back or changed suddenly causing subsequent queries to return different values. However, strict isolation levels require greater locking overhead, which can cause waits. For a NoSQL-style application that does not use long-running transactions, you can typically use the default isolation level or switch to a less strict isolation level.

Disabling Row Locks for memcached DML Operations

The innodb_api_disable_rowlock option can be used to disable row locks when memcached requests through the daemon_memcached plugin cause DML operations. By default,

`innodb_api_disable_rowlock` is set to `OFF` which means that `memcached` requests row locks for `get` and `set` operations. When `innodb_api_disable_rowlock` is set to `ON`, `memcached` requests a table lock instead of row locks.

The `innodb_api_disable_rowlock` option is not dynamic. It must be specified at startup on the `mysqld` command line or entered in a MySQL configuration file.

Allowing or Disallowing DDL

By default, you can perform DDL operations such as `ALTER TABLE` on tables used by the `daemon_memcached` plugin. To avoid potential slowdowns when these tables are used for high-throughput applications, disable DDL operations on these tables by enabling `innodb_api_enable_mdl` at startup. This option is less appropriate when accessing the same tables through both `memcached` and SQL, because it blocks `CREATE INDEX` statements on the tables, which could be important for running reporting queries.

Storing Data on Disk, in Memory, or Both

The `innodb_memcache.cache_policies` table specifies whether to store data written through the `memcached` interface to disk (`innodb_only`, the default); in memory only, as with traditional `memcached` (`cache_only`); or both (`caching`).

With the `caching` setting, if `memcached` cannot find a key in memory, it searches for the value in an `InnoDB` table. Values returned from `get` calls under the `caching` setting could be out-of-date if the values were updated on disk in the `InnoDB` table but are not yet expired from the memory cache.

The caching policy can be set independently for `get`, `set` (including `incr` and `decr`), `delete`, and `flush` operations.

For example, you might allow `get` and `set` operations to query or update a table and the `memcached` memory cache at the same time (using the `caching` setting), while making `delete`, `flush`, or both operate only on the in-memory copy (using the `cache_only` setting). That way, deleting or flushing an item only expires the item from the cache, and the latest value is returned from the `InnoDB` table the next time the item is requested.

```
mysql> SELECT * FROM innodb_memcache.cache_policies;
+--------------+-------------+-------------+---------------+--------------+
| policy_name  | get_policy  | set_policy  | delete_policy | flush_policy |
+--------------+-------------+-------------+---------------+--------------+
| cache_policy | innodb_only | innodb_only | innodb_only   | innodb_only  |
+--------------+-------------+-------------+---------------+--------------+

mysql> UPDATE innodb_memcache.cache_policies SET set_policy = 'caching'
       WHERE policy_name = 'cache_policy';
```

`innodb_memcache.cache_policies` values are only read at startup. After changing values in this table, uninstall and reinstall the `daemon_memcached` plugin to ensure that changes take effect.

```
mysql> UNINSTALL PLUGIN daemon_memcached;

mysql> INSTALL PLUGIN daemon_memcached soname "libmemcached.so";
```

15.20.6.5 Adapting DML Statements to memcached Operations

Benchmarks suggest that the `daemon_memcached` plugin speeds up DML operations (inserts, updates, and deletes) more than it speeds up queries. Therefore, consider focussing initial development efforts on write-intensive applications that are I/O-bound, and look for opportunities to use MySQL with the `daemon_memcached` plugin for new write-intensive applications.

Single-row DML statements are the easiest types of statements to turn into `memcached` operations. `INSERT` becomes `add`, `UPDATE` becomes `set`, `incr` or `decr`, and `DELETE` becomes `delete`. These

operations are guaranteed to only affect one row when issued through the `memcached` interface, because the *`key`* is unique within the table.

In the following SQL examples, `t1` refers to the table used for `memcached` operations, based on the configuration in the `innodb_memcache.containers` table. `key` refers to the column listed under `key_columns`, and `val` refers to the column listed under `value_columns`.

```
INSERT INTO t1 (key,val) VALUES (some_key,some_value);
SELECT val FROM t1 WHERE key = some_key;
UPDATE t1 SET val = new_value WHERE key = some_key;
UPDATE t1 SET val = val + x WHERE key = some_key;
DELETE FROM t1 WHERE key = some_key;
```

The following `TRUNCATE TABLE` and `DELETE` statements, which remove all rows from the table, correspond to the `flush_all` operation, where `t1` is configured as the table for `memcached` operations, as in the previous example.

```
TRUNCATE TABLE t1;
DELETE FROM t1;
```

15.20.6.6 Performing DML and DDL Statements on the Underlying InnoDB Table

You can access the underlying `InnoDB` table (which is `test.demo_test` by default) through standard SQL interfaces. However, there are some restrictions:

- When querying a table that is also accessed through the `memcached` interface, remember that `memcached` operations can be configured to be committed periodically rather than after every write operation. This behavior is controlled by the `daemon_memcached_w_batch_size` option. If this option is set to a value greater than `1`, use `READ UNCOMMITTED` queries to find rows that were just inserted.

```
mysql> SET SESSSION TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;

mysql> SELECT * FROM demo_test;
+------+------+------+------+-----------+------+------+------+------+------+------+
| cx   | cy   | c1   | cz   | c2        | ca   | CB   | c3   | cu   | c4   | C5   |
+------+------+------+------+-----------+------+------+------+------+------+------+
| NULL | NULL | a11  | NULL | 123456789 | NULL | NULL |   10 | NULL |    3 | NULL |
+------+------+------+------+-----------+------+------+------+------+------+------+
```

- When modifying a table using SQL that is also accessed through the `memcached` interface, you can configure `memcached` operations to start a new transaction periodically rather than for every read operation. This behavior is controlled by the `daemon_memcached_r_batch_size` option. If this option is set to a value greater than `1`, changes made to the table using SQL are not immediately visible to `memcached` operations.
- The `InnoDB` table is either IS (intention shared) or IX (intention exclusive) locked for all operations in a transaction. If you increase `daemon_memcached_r_batch_size` and `daemon_memcached_w_batch_size` substantially from their default value of `1`, the table is most likely locked between each operation, preventing DDL statements on the table.

15.20.7 The InnoDB memcached Plugin and Replication

Because the `daemon_memcached` plugin supports the MySQL binary log, source server through the `memcached` interface can be replicated for backup, balancing intensive read workloads, and high availability. All `memcached` commands are supported with binary logging.

You do not need to set up the `daemon_memcached` plugin on replica servers. The primary advantage of this configuration is increased write throughput on the source. The speed of the replication mechanism is not affected.

The following sections show how to use the binary log capability when using the `daemon_memcached` plugin with MySQL replication. It is assumed that you have completed the setup described in Section 15.20.3, “Setting Up the InnoDB memcached Plugin”.

Enabling the InnoDB memcached Binary Log

1. To use the `daemon_memcached` plugin with the MySQL binary log, enable the `innodb_api_enable_binlog` configuration option on the source server. This option can only be set at server startup. You must also enable the MySQL binary log on the source server using the `--log-bin` option. You can add these options to the MySQL configuration file, or on the `mysqld` command line.

   ```
   mysqld ... --log-bin --innodb_api_enable_binlog=1
   ```

2. Configure the source and replica server, as described in Section 17.1.2, "Setting Up Binary Log File Position Based Replication".

3. Use `mysqldump` to create a source data snapshot, and sync the snapshot to the replica server.

   ```
   source $> mysqldump --all-databases --lock-all-tables > dbdump.db
   replica $> mysql < dbdump.db
   ```

4. On the source server, issue `SHOW MASTER STATUS` to obtain the source binary log coordinates.

   ```
   mysql> SHOW MASTER STATUS;
   ```

5. On the replica server, use a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) to set up a replica server using the source binary log coordinates.

   ```
   mysql> CHANGE MASTER TO
          MASTER_HOST='localhost',
          MASTER_USER='root',
          MASTER_PASSWORD='',
          MASTER_PORT = 13000,
          MASTER_LOG_FILE='0.000001,
          MASTER_LOG_POS=114;

   Or from MySQL 8.0.23:
   mysql> CHANGE REPLICATION SOURCE TO
          SOURCE_HOST='localhost',
          SOURCE_USER='root',
          SOURCE_PASSWORD='',
          SOURCE_PORT = 13000,
          SOURCE_LOG_FILE='0.000001,
          SOURCE_LOG_POS=114;
   ```

6. Start the replica.

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

   If the error log prints output similar to the following, the replica is ready for replication.

   ```
   2013-09-24T13:04:38.639684Z 49 [Note] Replication I/O thread: connected to
   source 'root@localhost:13000', replication started in log '0.000001'
   at position 114
   ```

Testing the InnoDB memcached Replication Configuration

This example demonstrates how to test the `InnoDB memcached` replication configuration using the `memcached` and telnet to insert, update, and delete data. A MySQL client is used to verify results on the source and replica servers.

The example uses the `demo_test` table, which was created by the `innodb_memcached_config.sql` configuration script during the initial setup of the `daemon_memcached` plugin. The `demo_test` table contains a single example record.

1. Use the `set` command to insert a record with a key of `test1`, a flag value of `10`, an expiration value of `0`, a cas value of 1, and a value of `t1`.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
set test1 10 0 1
t1
STORED
```

2. On the source server, check that the record was inserted into the `demo_test` table. Assuming the `demo_test` table was not previously modified, there should be two records. The example record with a key of `AA`, and the record you just inserted, with a key of `test1`. The `c1` column maps to the key, the `c2` column to the value, the `c3` column to the flag value, the `c4` column to the cas value, and the `c5` column to the expiration time. The expiration time was set to 0, since it is unused.

```
mysql> SELECT * FROM test.demo_test;
+-------+--------------+------+------+------+
| c1    | c2           | c3   | c4   | c5   |
+-------+--------------+------+------+------+
| AA    | HELLO, HELLO |    8 |    0 |    0 |
| test1 | t1           |   10 |    1 |    0 |
+-------+--------------+------+------+------+
```

3. Check to verify that the same record was replicated to the replica server.

```
mysql> SELECT * FROM test.demo_test;
+-------+--------------+------+------+------+
| c1    | c2           | c3   | c4   | c5   |
+-------+--------------+------+------+------+
| AA    | HELLO, HELLO |    8 |    0 |    0 |
| test1 | t1           |   10 |    1 |    0 |
+-------+--------------+------+------+------+
```

4. Use the `set` command to update the key to a value of `new`.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
set test1 10 0 2
new
STORED
```

The update is replicated to the replica server (notice that the `cas` value is also updated).

```
mysql> SELECT * FROM test.demo_test;
+-------+--------------+------+------+------+
| c1    | c2           | c3   | c4   | c5   |
+-------+--------------+------+------+------+
| AA    | HELLO, HELLO |    8 |    0 |    0 |
| test1 | new          |   10 |    2 |    0 |
+-------+--------------+------+------+------+
```

5. Delete the `test1` record using a `delete` command.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
delete test1
DELETED
```

When the `delete` operation is replicated to the replica, the `test1` record on the replica is also deleted.

```
mysql> SELECT * FROM test.demo_test;
+----+--------------+------+------+------+
| c1 | c2           | c3   | c4   | c5   |
+----+--------------+------+------+------+
| AA | HELLO, HELLO |    8 |    0 |    0 |
```

```
+----+--------------+------+------+------+
```

6. Remove all rows from the table using the `flush_all` command.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
flush_all
OK
```

```
mysql> SELECT * FROM test.demo_test;
Empty set (0.00 sec)
```

7. Telnet to the source server and enter two new records.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'
set test2 10 0 4
again
STORED
set test3 10 0 5
again1
STORED
```

8. Confirm that the two records were replicated to the replica server.

```
mysql> SELECT * FROM test.demo_test;
+-------+--------------+------+------+------+
| c1    | c2           | c3   | c4   | c5   |
+-------+--------------+------+------+------+
| test2 | again        |   10 |    4 |    0 |
| test3 | again1       |   10 |    5 |    0 |
+-------+--------------+------+------+------+
```

9. Remove all rows from the table using the `flush_all` command.

```
telnet 127.0.0.1 11211
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
flush_all
OK
```

10. Check to ensure that the `flush_all` operation was replicated on the replica server.

```
mysql> SELECT * FROM test.demo_test;
Empty set (0.00 sec)
```

InnoDB memcached Binary Log Notes

Binary Log Format:

- Most `memcached` operations are mapped to DML statements (analogous to insert, delete, update). Since there is no actual SQL statement being processed by the MySQL server, all `memcached` commands (except for `flush_all`) use Row-Based Replication (RBR) logging, which is independent of any server `binlog_format` setting.
- The `memcached flush_all` command is mapped to the `TRUNCATE TABLE` command in MySQL 5.7 and earlier. Since DDL commands can only use statement-based logging, the `flush_all` command is replicated by sending a `TRUNCATE TABLE` statement. In MySQL 8.0 and later, `flush_all` is mapped to `DELETE` but is still replicated by sending a `TRUNCATE TABLE` statement.

Transactions:

- The concept of transactions has not typically been part of `memcached` applications. For performance considerations, `daemon_memcached_r_batch_size` and `daemon_memcached_w_batch_size`

are used to control the batch size for read and write transactions. These settings do not affect replication. Each SQL operation on the underlying `InnoDB` table is replicated after successful completion.

- The default value of `daemon_memcached_w_batch_size` is `1`, which means that each `memcached` write operation is committed immediately. This default setting incurs a certain amount of performance overhead to avoid inconsistencies in the data that is visible on the source and replica servers. The replicated records are always available immediately on the replica server. If you set `daemon_memcached_w_batch_size` to a value greater than `1`, records inserted or updated through `memcached` are not immediately visible on the source server; to view the records on the source server before they are committed, issue `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED`.

15.20.8 InnoDB memcached Plugin Internals

InnoDB API for the InnoDB memcached Plugin

The `InnoDB` `memcached` engine accesses `InnoDB` through `InnoDB` APIs, most of which are directly adopted from embedded `InnoDB`. `InnoDB` API functions are passed to the `InnoDB` `memcached` engine as callback functions. `InnoDB` API functions access the `InnoDB` tables directly, and are mostly DML operations with the exception of `TRUNCATE TABLE`.

`memcached` commands are implemented through the `InnoDB` `memcached` API. The following table outlines how `memcached` commands are mapped to DML or DDL operations.

Table 15.27 memcached Commands and Associated DML or DDL Operations

memcached Command	DML or DDL Operations
`get`	a read/fetch command
`set`	a search followed by an `INSERT` or `UPDATE` (depending on whether or not a key exists)
`add`	a search followed by an `INSERT` or `UPDATE`
`replace`	a search followed by an `UPDATE`
`append`	a search followed by an `UPDATE` (appends data to the result before `UPDATE`)
`prepend`	a search followed by an `UPDATE` (prepends data to the result before `UPDATE`)
`incr`	a search followed by an `UPDATE`
`decr`	a search followed by an `UPDATE`
`delete`	a search followed by a `DELETE`
`flush_all`	`TRUNCATE TABLE` (DDL)

InnoDB memcached Plugin Configuration Tables

This section describes configuration tables used by the `daemon_memcached` plugin. The `cache_policies` table, `config_options` table, and `containers` table are created by the `innodb_memcached_config.sql` configuration script in the `innodb_memcache` database.

```
mysql> USE innodb_memcache;
Database changed
mysql> SHOW TABLES;
+---------------------------+
| Tables_in_innodb_memcache |
+---------------------------+
| cache_policies            |
| config_options            |
```

```
| containers                |
+---------------------------+
```

cache_policies Table

The `cache_policies` table defines a cache policy for the `InnoDB` `memcached` installation. You can specify individual policies for `get`, `set`, `delete`, and `flush` operations, within a single cache policy. The default setting for all operations is `innodb_only`.

- `innodb_only`: Use `InnoDB` as the data store.
- `cache_only`: Use the `memcached` engine as the data store.
- `caching`: Use both `InnoDB` and the `memcached` engine as data stores. In this case, if `memcached` cannot find a key in memory, it searches for the value in an `InnoDB` table.
- `disable`: Disable caching.

Table 15.28 cache_policies Columns

Column	Description
`policy_name`	Name of the cache policy. The default cache policy name is `cache_policy`.
`get_policy`	The cache policy for get operations. Valid values are `innodb_only`, `cache_only`, `caching`, or `disabled`. The default setting is `innodb_only`.
`set_policy`	The cache policy for set operations. Valid values are `innodb_only`, `cache_only`, `caching`, or `disabled`. The default setting is `innodb_only`.
`delete_policy`	The cache policy for delete operations. Valid values are `innodb_only`, `cache_only`, `caching`, or `disabled`. The default setting is `innodb_only`.
`flush_policy`	The cache policy for flush operations. Valid values are `innodb_only`, `cache_only`, `caching`, or `disabled`. The default setting is `innodb_only`.

config_options Table

The `config_options` table stores `memcached`-related settings that can be changed at runtime using SQL. Supported configuration options are `separator` and `table_map_delimiter`.

Table 15.29 config_options Columns

Column	Description
`Name`	Name of the `memcached`-related configuration option. The following configuration options are supported by the `config_options` table: • `separator`: Used to separate values of a long string into separate values when there are multiple `value_columns` defined. By default, the `separator` is a `|` character. For example, if you define `col1, col2` as value columns, and you define `|` as the separator, you can issue the following `memcached` command to insert values into `col1` and `col2`, respectively:

Column	Description

	`set keyx 10 0 19`
	`valuecolx|valuecoly`
	
	`valuecol1x` is stored in `col1` and `valuecoly` is stored in `col2`.
	
	• `table_map_delimiter`: The character separating the schema name and the table name when you use the `@@` notation in a key name to access a key in a specific table. For example, `@@t1.some_key` and `@@t2.some_key` have the same key value, but are stored in different tables.

`Value`	The value assigned to the `memcached`-related configuration option.

containers Table

The `containers` table is the most important of the three configuration tables. Each `InnoDB` table that is used to store `memcached` values must have an entry in the `containers` table. The entry provides a mapping between `InnoDB` table columns and container table columns, which is required for `memcached` to work with `InnoDB` tables.

The `containers` table contains a default entry for the `test.demo_test` table, which is created by the `innodb_memcached_config.sql` configuration script. To use the `daemon_memcached` plugin with your own `InnoDB` table, you must create an entry in the `containers` table.

Table 15.30 containers Columns

Column	Description
`name`	The name given to the container. If an `InnoDB` table is not requested by name using `@@` notation, the `daemon_memcached` plugin uses the `InnoDB` table with a `containers.name` value of `default`. If there is no such entry, the first entry in the `containers` table, ordered alphabetically by `name` (ascending), determines the default `InnoDB` table.
`db_schema`	The name of the database where the `InnoDB` table resides. This is a required value.
`db_table`	The name of the `InnoDB` table that stores `memcached` values. This is a required value.
`key_columns`	The column in the `InnoDB` table that contains lookup key values for `memcached` operations. This is a required value.
`value_columns`	The `InnoDB` table columns (one or more) that store `memcached` data. Multiple columns can be specified using the separator character specified in the `innodb_memcached.config_options` table. By default, the separator is a pipe character ("|"). To specify multiple columns, separate them with the defined separator character. For example: `col1|col2|col3`. This is a required value.
`flags`	The `InnoDB` table columns that are used as flags (a user-defined numeric value that is

Column	Description
	stored and retrieved along with the main value) for `memcached`. A flag value can be used as a column specifier for some operations (such as `incr`, `prepend`) if a `memcached` value is mapped to multiple columns, so that an operation is performed on a specified column. For example, if you have mapped a `value_columns` to three `InnoDB` table columns, and only want the increment operation performed on one columns, use the `flags` column to specify the column. If you do not use the `flags` column, set a value of `0` to indicate that it is unused.
`cas_column`	The `InnoDB` table column that stores compare-and-swap (cas) values. The `cas_column` value is related to the way `memcached` hashes requests to different servers and caches data in memory. Because the `InnoDB memcached` plugin is tightly integrated with a single `memcached` daemon, and the in-memory caching mechanism is handled by MySQL and the InnoDB buffer pool, this column is rarely needed. If you do not use this column, set a value of `0` to indicate that it is unused.
`expire_time_column`	The `InnoDB` table column that stores expiration values. The `expire_time_column` value is related to the way `memcached` hashes requests to different servers and caches data in memory. Because the `InnoDB memcached` plugin is tightly integrated with a single `memcached` daemon, and the in-memory caching mechanism is handled by MySQL and the InnoDB buffer pool, this column is rarely needed. If you do not use this column, set a value of `0` to indicate that the column is unused. The maximum expire time is defined as `INT_MAX32` or 2147483647 seconds (approximately 68 years).
`unique_idx_name_on_key`	The name of the index on the key column. It must be a unique index. It can be the primary key or a secondary index. Preferably, use the primary key of the `InnoDB` table. Using the primary key avoids a lookup that is performed when using a secondary index. You cannot make a covering index for `memcached` lookups; `InnoDB` returns an error if you try to define a composite secondary index over both the key and value columns.

containers Table Column Constraints

- You must supply a value for `db_schema`, `db_name`, `key_columns`, `value_columns` and `unique_idx_name_on_key`. Specify `0` for `flags`, `cas_column`, and `expire_time_column` if they are unused. Failing to do so could cause your setup to fail.
- `key_columns`: The maximum limit for a `memcached` key is 250 characters, which is enforced by `memcached`. The mapped key must be a non-Null `CHAR` or `VARCHAR` type.
- `value_columns`: Must be mapped to a `CHAR`, `VARCHAR`, or `BLOB` column. There is no length restriction and the value can be NULL.

- `cas_column`: The `cas` value is a 64 bit integer. It must be mapped to a `BIGINT` of at least 8 bytes. If you do not use this column, set a value of `0` to indicate that it is unused.
- `expiration_time_column`: Must mapped to an `INTEGER` of at least 4 bytes. Expiration time is defined as a 32-bit integer for Unix time (the number of seconds since January 1, 1970, as a 32-bit value), or the number of seconds starting from the current time. For the latter, the number of seconds may not exceed 60*60*24*30 (the number of seconds in 30 days). If the number sent by a client is larger, the server considers it to be a real Unix time value rather than an offset from the current time. If you do not use this column, set a value of `0` to indicate that it is unused.
- `flags`: Must be mapped to an `INTEGER` of at least 32-bits and can be NULL. If you do not use this column, set a value of `0` to indicate that it is unused.

A pre-check is performed at plugin load time to enforce column constraints. If mismatches are found, the plugin is not loaded.

Multiple Value Column Mapping

- During plugin initialization, when `InnoDB memcached` is configured with information defined in the `containers` table, each mapped column defined in `containers.value_columns` is verified against the mapped `InnoDB` table. If multiple `InnoDB` table columns are mapped, there is a check to ensure that each column exists and is the right type.
- At run-time, for `memcached` insert operations, if there are more delimited values than the number of mapped columns, only the number of mapped values are taken. For example, if there are six mapped columns, and seven delimited values are provided, only the first six delimited values are taken. The seventh delimited value is ignored.
- If there are fewer delimited values than mapped columns, unfilled columns are set to NULL. If an unfilled column cannot be set to NULL, insert operations fail.
- If a table has more columns than mapped values, the extra columns do not affect results.

The demo_test Example Table

The `innodb_memcached_config.sql` configuration script creates a `demo_test` table in the `test` database, which can be used to verify `InnoDB memcached` plugin installation immediately after setup.

The `innodb_memcached_config.sql` configuration script also creates an entry for the `demo_test` table in the `innodb_memcache.containers` table.

```
mysql> SELECT * FROM innodb_memcache.containers\G
*************************** 1. row ***************************
                name: aaa
           db_schema: test
            db_table: demo_test
         key_columns: c1
       value_columns: c2
               flags: c3
          cas_column: c4
  expire_time_column: c5
unique_idx_name_on_key: PRIMARY

mysql> SELECT * FROM test.demo_test;
+----+-----------------+------+------+------+
| c1 | c2              | c3   | c4   | c5   |
+----+-----------------+------+------+------+
| AA | HELLO, HELLO    |    8 |    0 |    0 |
+----+-----------------+------+------+------+
```

15.20.9 Troubleshooting the InnoDB memcached Plugin

This section describes issues that you may encounter when using the `InnoDB memcached` plugin.

- If you encounter the following error in the MySQL error log, the server might fail to start:

  ```
  failed to set rlimit for open files. Try running as root or requesting
  smaller maxconns value.
  ```

  The error message is from the `memcached` daemon. One solution is to raise the OS limit for the number of open files. The commands for checking and increasing the open file limit varies by operating system. This example shows commands for Linux and macOS:

  ```
  # Linux
  $> ulimit -n
  1024
  $> ulimit -n 4096
  $> ulimit -n
  4096

  # macOS
  $> ulimit -n
  256
  $> ulimit -n 4096
  $> ulimit -n
  4096
  ```

  The other solution is to reduce the number of concurrent connections permitted for the `memcached` daemon. To do so, encode the `-c memcached` option in the `daemon_memcached_option` configuration parameter in the MySQL configuration file. The `-c` option has a default value of 1024.

  ```
  [mysqld]
  ...
  loose-daemon_memcached_option='-c 64'
  ```

- To troubleshoot problems where the `memcached` daemon is unable to store or retrieve `InnoDB` table data, encode the `-vvv memcached` option in the `daemon_memcached_option` configuration parameter in the MySQL configuration file. Examine the MySQL error log for debug output related to `memcached` operations.

  ```
  [mysqld]
  ...
  loose-daemon_memcached_option='-vvv'
  ```

- If columns specified to hold `memcached` values are the wrong data type, such as a numeric type instead of a string type, attempts to store key-value pairs fail with no specific error code or message.

- If the `daemon_memcached` plugin causes MySQL server startup issues, you can temporarily disable the `daemon_memcached` plugin while troubleshooting by adding this line under the `[mysqld]` group in the MySQL configuration file:

  ```
  daemon_memcached=OFF
  ```

  For example, if you run the `INSTALL PLUGIN` statement before running the `innodb_memcached_config.sql` configuration script to set up the necessary database and tables, the server might unexpectedly exit and fail to start. The server could also fail to start if you incorrectly configure an entry in the `innodb_memcache.containers` table.

  To uninstall the `memcached` plugin for a MySQL instance, issue the following statement:

  ```
  mysql> UNINSTALL PLUGIN daemon_memcached;
  ```

- If you run more than one instance of MySQL on the same machine with the `daemon_memcached` plugin enabled in each instance, use the `daemon_memcached_option` configuration parameter to specify a unique `memcached` port for each `daemon_memcached` plugin.

- If an SQL statement cannot find the `InnoDB` table or finds no data in the table, but `memcached` API calls retrieve the expected data, you may be missing an entry for the `InnoDB` table in the `innodb_memcache.containers` table, or you may have not switched to the correct `InnoDB`

table by issuing a `get` or `set` request using `@@table_id` notation. This problem could also occur if you change an existing entry in the `innodb_memcache.containers` table without restarting the MySQL server afterward. The free-form storage mechanism is flexible enough that your requests to store or retrieve a multi-column value such as `col1|col2|col3` may still work, even if the daemon is using the `test.demo_test` table which stores values in a single column.

- When defining your own `InnoDB` table for use with the `daemon_memcached` plugin, and columns in the table are defined as `NOT NULL`, ensure that values are supplied for the `NOT NULL` columns when inserting a record for the table into the `innodb_memcache.containers` table. If the `INSERT` statement for the `innodb_memcache.containers` record contains fewer delimited values than there are mapped columns, unfilled columns are set to `NULL`. Attempting to insert a `NULL` value into a `NOT NULL` column causes the `INSERT` to fail, which may only become evident after you reinitialize the `daemon_memcached` plugin to apply changes to the `innodb_memcache.containers` table.

- If `cas_column` and `expire_time_column` fields of the `innodb_memcached.containers` table are set to `NULL`, the following error is returned when attempting to load the `memcached` plugin:

  ```
  InnoDB_Memcached: column 6 in the entry for config table 'containers' in
  database 'innodb_memcache' has an invalid NULL value.
  ```

  The `memcached` plugin rejects usage of `NULL` in the `cas_column` and `expire_time_column` columns. Set the value of these columns to `0` when the columns are unused.

- As the length of the `memcached` key and values increase, you might encounter size and length limits.

  - When the key exceeds 250 bytes, `memcached` operations return an error. This is currently a fixed limit within `memcached`.

  - `InnoDB` table limits may be encountered if values exceed 768 bytes in size, 3072 bytes in size, or half of the `innodb_page_size` value. These limits primarily apply if you intend to create an index on a value column to run report-generating queries on that column using SQL. See Section 15.22, “InnoDB Limits” for details.

  - The maximum size for the key-value combination is 1 MB.

- If you share configuration files across MySQL servers of different versions, using the latest configuration options for the `daemon_memcached` plugin could cause startup errors on older MySQL versions. To avoid compatibility problems, use the `loose` prefix with option names. For example, use `loose-daemon_memcached_option='-c 64'` instead of `daemon_memcached_option='-c 64'`.

- There is no restriction or check in place to validate character set settings. `memcached` stores and retrieves keys and values in bytes and is therefore not character set sensitive. However, you must ensure that the `memcached` client and the MySQL table use the same character set.

- `memcached` connections are blocked from accessing tables that contain an indexed virtual column. Accessing an indexed virtual column requires a callback to the server, but a `memcached` connection does not have access to the server code.

15.21 InnoDB Troubleshooting

The following general guidelines apply to troubleshooting `InnoDB` problems:

- When an operation fails or you suspect a bug, look at the MySQL server error log (see Section 5.4.2, “The Error Log”). Server Error Message Reference provides troubleshooting information for some of the common `InnoDB`-specific errors that you may encounter.

- If the failure is related to a deadlock, run with the `innodb_print_all_deadlocks` option enabled so that details about each deadlock are printed to the MySQL server error log. For information about deadlocks, see Section 15.7.5, “Deadlocks in InnoDB”.

- If the issue is related to the InnoDB data dictionary, see Section 15.21.4, “Troubleshooting InnoDB Data Dictionary Operations”.
- When troubleshooting, it is usually best to run the MySQL server from the command prompt, rather than through mysqld_safe or as a Windows service. You can then see what mysqld prints to the console, and so have a better grasp of what is going on. On Windows, start mysqld with the --console option to direct the output to the console window.
- Enable the InnoDB Monitors to obtain information about a problem (see Section 15.17, “InnoDB Monitors”). If the problem is performance-related, or your server appears to be hung, you should enable the standard Monitor to print information about the internal state of InnoDB. If the problem is with locks, enable the Lock Monitor. If the problem is with table creation, tablespaces, or data dictionary operations, refer to the InnoDB Information Schema system tables to examine contents of the InnoDB internal data dictionary.

  InnoDB temporarily enables standard InnoDB Monitor output under the following conditions:

  - A long semaphore wait
  - InnoDB cannot find free blocks in the buffer pool
  - Over 67% of the buffer pool is occupied by lock heaps or the adaptive hash index
- If you suspect that a table is corrupt, run CHECK TABLE on that table.

15.21.1 Troubleshooting InnoDB I/O Problems

The troubleshooting steps for InnoDB I/O problems depend on when the problem occurs: during startup of the MySQL server, or during normal operations when a DML or DDL statement fails due to problems at the file system level.

Initialization Problems

If something goes wrong when InnoDB attempts to initialize its tablespace or its log files, delete all files created by InnoDB: all ibdata files and all redo log files (#ib_redo*N* files in MySQL 8.0.30 and higher or ib_logfile files in earlier releases). If you created any InnoDB tables, also delete any .ibd files from the MySQL database directories. Then try initializing InnoDB again. For easiest troubleshooting, start the MySQL server from a command prompt so that you see what is happening.

Runtime Problems

If InnoDB prints an operating system error during a file operation, usually the problem has one of the following solutions:

- Make sure the InnoDB data file directory and the InnoDB log directory exist.
- Make sure mysqld has access rights to create files in those directories.
- Make sure mysqld can read the proper my.cnf or my.ini option file, so that it starts with the options that you specified.
- Make sure the disk is not full and you are not exceeding any disk quota.
- Make sure that the names you specify for subdirectories and data files do not clash.
- Doublecheck the syntax of the innodb_data_home_dir and innodb_data_file_path values. In particular, any MAX value in the innodb_data_file_path option is a hard limit, and exceeding that limit causes a fatal error.

15.21.2 Troubleshooting Recovery Failures

From MySQL 8.0.26, checkpoints and advancing the checkpoint LSN are not permitted until redo log recovery is complete and data dictionary dynamic metadata (srv_dict_metadata) is transferred

to data dictionary table (`dict_table_t`) objects. Should the redo log run out of space during recovery or after recovery (but before data dictionary dynamic metadata is transferred to data dictionary table objects) as a result of this change, an `innodb_force_recovery` restart may be required, starting with at least the `SRV_FORCE_NO_IBUF_MERGE` setting or, in case that fails, the `SRV_FORCE_NO_LOG_REDO` setting. If an `innodb_force_recovery` restart fails in this scenario, recovery from backup may be necessary. (Bug #32200595)

15.21.3 Forcing InnoDB Recovery

To investigate database page corruption, you might dump your tables from the database with `SELECT ... INTO OUTFILE`. Usually, most of the data obtained in this way is intact. Serious corruption might cause `SELECT * FROM tbl_name` statements or `InnoDB` background operations to unexpectedly exit or assert, or even cause `InnoDB` roll-forward recovery to crash. In such cases, you can use the `innodb_force_recovery` option to force the `InnoDB` storage engine to start up while preventing background operations from running, so that you can dump your tables. For example, you can add the following line to the `[mysqld]` section of your option file before restarting the server:

```
[mysqld]
innodb_force_recovery = 1
```

For information about using option files, see Section 4.2.2.2, “Using Option Files”.

Warning

Only set `innodb_force_recovery` to a value greater than 0 in an emergency situation, so that you can start `InnoDB` and dump your tables. Before doing so, ensure that you have a backup copy of your database in case you need to recreate it. Values of 4 or greater can permanently corrupt data files. Only use an `innodb_force_recovery` setting of 4 or greater on a production server instance after you have successfully tested the setting on a separate physical copy of your database. When forcing `InnoDB` recovery, you should always start with `innodb_force_recovery=1` and only increase the value incrementally, as necessary.

`innodb_force_recovery` is 0 by default (normal startup without forced recovery). The permissible nonzero values for `innodb_force_recovery` are 1 to 6. A larger value includes the functionality of lesser values. For example, a value of 3 includes all of the functionality of values 1 and 2.

If you are able to dump your tables with an `innodb_force_recovery` value of 3 or less, then you are relatively safe that only some data on corrupt individual pages is lost. A value of 4 or greater is considered dangerous because data files can be permanently corrupted. A value of 6 is considered drastic because database pages are left in an obsolete state, which in turn may introduce more corruption into B-trees and other database structures.

As a safety measure, `InnoDB` prevents `INSERT`, `UPDATE`, or `DELETE` operations when `innodb_force_recovery` is greater than 0. An `innodb_force_recovery` setting of 4 or greater places `InnoDB` in read-only mode.

- `1` (`SRV_FORCE_IGNORE_CORRUPT`)

  Lets the server run even if it detects a corrupt page. Tries to make `SELECT * FROM tbl_name` jump over corrupt index records and pages, which helps in dumping tables.

- `2` (`SRV_FORCE_NO_BACKGROUND`)

  Prevents the master thread and any purge threads from running. If an unexpected exit would occur during the purge operation, this recovery value prevents it.

- `3` (`SRV_FORCE_NO_TRX_UNDO`)

  Does not run transaction rollbacks after crash recovery.

- 4 (SRV_FORCE_NO_IBUF_MERGE)

  Prevents insert buffer merge operations. If they would cause a crash, does not do them. Does not calculate table statistics. This value can permanently corrupt data files. After using this value, be prepared to drop and recreate all secondary indexes. Sets InnoDB to read-only.

- 5 (SRV_FORCE_NO_UNDO_LOG_SCAN)

  Does not look at undo logs when starting the database: InnoDB treats even incomplete transactions as committed. This value can permanently corrupt data files. Sets InnoDB to read-only.

- 6 (SRV_FORCE_NO_LOG_REDO)

  Does not do the redo log roll-forward in connection with recovery. This value can permanently corrupt data files. Leaves database pages in an obsolete state, which in turn may introduce more corruption into B-trees and other database structures. Sets InnoDB to read-only.

You can SELECT from tables to dump them. With an innodb_force_recovery value of 3 or less you can DROP or CREATE tables. DROP TABLE is also supported with an innodb_force_recovery value greater than 3. DROP TABLE is not permitted with an innodb_force_recovery value greater than 4.

If you know that a given table is causing an unexpected exit on rollback, you can drop it. If you encounter a runaway rollback caused by a failing mass import or ALTER TABLE, you can kill the mysqld process and set innodb_force_recovery to 3 to bring the database up without the rollback, and then DROP the table that is causing the runaway rollback.

If corruption within the table data prevents you from dumping the entire table contents, a query with an ORDER BY *primary_key* DESC clause might be able to dump the portion of the table after the corrupted part.

If a high innodb_force_recovery value is required to start InnoDB, there may be corrupted data structures that could cause complex queries (queries containing WHERE, ORDER BY, or other clauses) to fail. In this case, you may only be able to run basic SELECT * FROM t queries.

15.21.4 Troubleshooting InnoDB Data Dictionary Operations

Information about table definitions is stored in the InnoDB data dictionary. If you move data files around, dictionary data can become inconsistent.

If a data dictionary corruption or consistency issue prevents you from starting InnoDB, see Section 15.21.3, “Forcing InnoDB Recovery” for information about manual recovery.

Cannot Open Datafile

With innodb_file_per_table enabled (the default), the following messages may appear at startup if a file-per-table tablespace file (.ibd file) is missing:

```
[ERROR] InnoDB: Operating system error number 2 in a file operation.
[ERROR] InnoDB: The error means the system cannot find the path specified.
[ERROR] InnoDB: Cannot open datafile for read-only: './test/t1.ibd' OS error: 71
[Warning] InnoDB: Ignoring tablespace `test/t1` because it could not be opened.
```

To address these messages, issue DROP TABLE statement to remove data about the missing table from the data dictionary.

Restoring Orphan File-Per-Table ibd Files

This procedure describes how to restore orphan file-per-table .ibd files to another MySQL instance. You might use this procedure if the system tablespace is lost or unrecoverable and you want to restore .ibd file backups on a new MySQL instance.

The procedure is not supported for general tablespace `.ibd` files.

The procedure assumes that you only have `.ibd` file backups, you are recovering to the same version of MySQL that initially created the orphan `.ibd` files, and that `.ibd` file backups are clean. See Section 15.6.1.4, “Moving or Copying InnoDB Tables” for information about creating clean backups.

Table import limitations outlined in Section 15.6.1.3, “Importing InnoDB Tables” are applicable to this procedure.

1. On the new MySQL instance, recreate the table in a database of the same name.

```
mysql> CREATE DATABASE sakila;

mysql> USE sakila;

mysql> CREATE TABLE actor (
         actor_id SMALLINT UNSIGNED NOT NULL AUTO_INCREMENT,
         first_name VARCHAR(45) NOT NULL,
         last_name VARCHAR(45) NOT NULL,
         last_update TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
         PRIMARY KEY  (actor_id),
         KEY idx_actor_last_name (last_name)
       )ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;
```

2. Discard the tablespace of the newly created table.

```
mysql> ALTER TABLE sakila.actor DISCARD TABLESPACE;
```

3. Copy the orphan `.ibd` file from your backup directory to the new database directory.

```
$> cp /backup_directory/actor.ibd path/to/mysql-5.7/data/sakila/
```

4. Ensure that the `.ibd` file has the necessary file permissions.

5. Import the orphan `.ibd` file. A warning is issued indicating that `InnoDB` is attempting to import the file without schema verification.

```
mysql> ALTER TABLE sakila.actor IMPORT TABLESPACE; SHOW WARNINGS;
Query OK, 0 rows affected, 1 warning (0.15 sec)

Warning | 1810 | InnoDB: IO Read error: (2, No such file or directory)
Error opening './sakila/actor.cfg', will attempt to import
without schema verification
```

6. Query the table to verify that the `.ibd` file was successfully restored.

```
mysql> SELECT COUNT(*) FROM sakila.actor;
+----------+
| count(*) |
+----------+
|      200 |
+----------+
```

15.21.5 InnoDB Error Handling

The following items describe how `InnoDB` performs error handling. `InnoDB` sometimes rolls back only the statement that failed, other times it rolls back the entire transaction.

- If you run out of file space in a tablespace, a MySQL `Table is full` error occurs and `InnoDB` rolls back the SQL statement.
- A transaction deadlock causes `InnoDB` to roll back the entire transaction. Retry the entire transaction when this happens.

  A lock wait timeout causes `InnoDB` to roll back the current statement (the statement that was waiting for the lock and encountered the timeout). To have the entire transaction roll back, start the

server with `--innodb-rollback-on-timeout` enabled. Retry the statement if using the default behavior, or the entire transaction if `--innodb-rollback-on-timeout` is enabled.

Both deadlocks and lock wait timeouts are normal on busy servers and it is necessary for applications to be aware that they may happen and handle them by retrying. You can make them less likely by doing as little work as possible between the first change to data during a transaction and the commit, so the locks are held for the shortest possible time and for the smallest possible number of rows. Sometimes splitting work between different transactions may be practical and helpful.

- A duplicate-key error rolls back the SQL statement, if you have not specified the `IGNORE` option in your statement.
- A `row too long error` rolls back the SQL statement.
- Other errors are mostly detected by the MySQL layer of code (above the `InnoDB` storage engine level), and they roll back the corresponding SQL statement. Locks are not released in a rollback of a single SQL statement.

During implicit rollbacks, as well as during the execution of an explicit `ROLLBACK` SQL statement, `SHOW PROCESSLIST` displays `Rolling back` in the `State` column for the relevant connection.

15.22 InnoDB Limits

This section describes limits for `InnoDB` tables, indexes, tablespaces, and other aspects of the `InnoDB` storage engine.

- A table can contain a maximum of 1017 columns. Virtual generated columns are included in this limit.
- A table can contain a maximum of 64 secondary indexes.
- The index key prefix length limit is 3072 bytes for `InnoDB` tables that use `DYNAMIC` or `COMPRESSED` row format.

  The index key prefix length limit is 767 bytes for `InnoDB` tables that use the `REDUNDANT` or `COMPACT` row format. For example, you might hit this limit with a column prefix index of more than 191 characters on a `TEXT` or `VARCHAR` column, assuming a `utf8mb4` character set and the maximum of 4 bytes for each character.

  Attempting to use an index key prefix length that exceeds the limit returns an error.

  If you reduce the `InnoDB` page size to 8KB or 4KB by specifying the `innodb_page_size` option when creating the MySQL instance, the maximum length of the index key is lowered proportionally, based on the limit of 3072 bytes for a 16KB page size. That is, the maximum index key length is 1536 bytes when the page size is 8KB, and 768 bytes when the page size is 4KB.

  The limits that apply to index key prefixes also apply to full-column index keys.

- A maximum of 16 columns is permitted for multicolumn indexes. Exceeding the limit returns an error.

```
ERROR 1070 (42000): Too many key parts specified; max 16 parts allowed
```

- The maximum row size, excluding any variable-length columns that are stored off-page, is slightly less than half of a page for 4KB, 8KB, 16KB, and 32KB page sizes. For example, the maximum row size for the default `innodb_page_size` of 16KB is about 8000 bytes. However, for an `InnoDB` page size of 64KB, the maximum row size is approximately 16000 bytes. `LONGBLOB` and `LONGTEXT` columns must be less than 4GB, and the total row size, including `BLOB` and `TEXT` columns, must be less than 4GB.

  If a row is less than half a page long, all of it is stored locally within the page. If it exceeds half a page, variable-length columns are chosen for external off-page storage until the row fits within half a page, as described in Section 15.11.2, “File Space Management”.

- Although InnoDB supports row sizes larger than 65,535 bytes internally, MySQL itself imposes a row-size limit of 65,535 for the combined size of all columns. See Section 8.4.7, “Limits on Table Column Count and Row Size”.
- On some older operating systems, files must be less than 2GB. This is not an InnoDB limitation. If you require a large system tablespace, configure it using several smaller data files rather than one large data file, or distribute table data across file-per-table and general tablespace data files.
- The combined maximum size for InnoDB log files is 512GB.
- The minimum tablespace size is slightly larger than 10MB. The maximum tablespace size depends on the InnoDB page size.

  Table 15.31 InnoDB Maximum Tablespace Size

InnoDB Page Size	Maximum Tablespace Size
4KB	16TB
8KB	32TB
16KB	64TB
32KB	128TB
64KB	256TB

  The maximum tablespace size is also the maximum size for a table.
- An InnoDB instance supports up to 2^32 (4294967296) tablespaces, with a small number of those tablespaces reserved for undo and temporary tables.
- Shared tablespaces support up to 2^32 (4294967296) tables.
- The path of a tablespace file, including the file name, cannot exceed the MAX_PATH limit on Windows. Prior to Windows 10, the MAX_PATH limit is 260 characters. As of Windows 10, version 1607, MAX_PATH limitations are removed from common Win32 file and directory functions, but you must enable the new behavior.
- For limits associated with concurrent read-write transactions, see Section 15.6.6, “Undo Logs”.

15.23 InnoDB Restrictions and Limitations

This section describes restrictions and limitations of the InnoDB storage engine.

- You cannot create a table with a column name that matches the name of an internal InnoDB column (including DB_ROW_ID, DB_TRX_ID, and DB_ROLL_PTR. This restriction applies to use of the names in any lettercase.

```
mysql> CREATE TABLE t1 (c1 INT, db_row_id INT) ENGINE=INNODB;
ERROR 1166 (42000): Incorrect column name 'db_row_id'
```

- SHOW TABLE STATUS does not provide accurate statistics for InnoDB tables except for the physical size reserved by the table. The row count is only a rough estimate used in SQL optimization.
- InnoDB does not keep an internal count of rows in a table because concurrent transactions might “see” different numbers of rows at the same time. Consequently, SELECT COUNT(*) statements only count rows visible to the current transaction.

  For information about how InnoDB processes SELECT COUNT(*) statements, refer to the COUNT() description in Section 12.20.1, “Aggregate Function Descriptions”.
- ROW_FORMAT=COMPRESSED is unsupported for page sizes greater than 16KB.
- A MySQL instance using a particular InnoDB page size (innodb_page_size) cannot use data files or log files from an instance that uses a different page size.

- For limitations associated with importing tables using the *Transportable Tablespaces* feature, see Table Import Limitations.
- For limitations associated with online DDL, see Section 15.12.8, “Online DDL Limitations”.
- For limitations associated with general tablespaces, see General Tablespace Limitations.
- For limitations associated with data-at-rest encryption, see Encryption Limitations.

Chapter 16 Alternative Storage Engines

Table of Contents

Storage engines are MySQL components that handle the SQL operations for different table types. `InnoDB` is the default and most general-purpose storage engine, and Oracle recommends using it for tables except for specialized use cases. (The `CREATE TABLE` statement in MySQL 8.0 creates `InnoDB` tables by default.)

MySQL Server uses a pluggable storage engine architecture that enables storage engines to be loaded into and unloaded from a running MySQL server.

To determine which storage engines your server supports, use the `SHOW ENGINES` statement. The value in the `Support` column indicates whether an engine can be used. A value of `YES`, `NO`, or `DEFAULT` indicates that an engine is available, not available, or available and currently set as the default storage engine.

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
```

```
  Savepoints: YES
*************************** 3. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 4. row ***************************
      Engine: BLACKHOLE
     Support: YES
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
...
```

This chapter covers use cases for special-purpose MySQL storage engines. It does not cover the default `InnoDB` storage engine or the `NDB` storage engine which are covered in Chapter 15, *The InnoDB Storage Engine* and Chapter 23, *MySQL NDB Cluster 8.0*. For advanced users, it also contains a description of the pluggable storage engine architecture (see Section 16.11, “Overview of MySQL Storage Engine Architecture”).

For information about features offered in commercial MySQL Server binaries, see *MySQL Editions*, on the MySQL website. The storage engines available might depend on which edition of MySQL you are using.

For answers to commonly asked questions about MySQL storage engines, see Section A.2, “MySQL 8.0 FAQ: Storage Engines”.

MySQL 8.0 Supported Storage Engines

- `InnoDB`: The default storage engine in MySQL 8.0. `InnoDB` is a transaction-safe (ACID compliant) storage engine for MySQL that has commit, rollback, and crash-recovery capabilities to protect user data. `InnoDB` row-level locking (without escalation to coarser granularity locks) and Oracle-style consistent nonlocking reads increase multi-user concurrency and performance. `InnoDB` stores user data in clustered indexes to reduce I/O for common queries based on primary keys. To maintain data integrity, `InnoDB` also supports `FOREIGN KEY` referential-integrity constraints. For more information about `InnoDB`, see Chapter 15, *The InnoDB Storage Engine*.

- `MyISAM`: These tables have a small footprint. Table-level locking limits the performance in read/write workloads, so it is often used in read-only or read-mostly workloads in Web and data warehousing configurations.

- `Memory`: Stores all data in RAM, for fast access in environments that require quick lookups of non-critical data. This engine was formerly known as the `HEAP` engine. Its use cases are decreasing; `InnoDB` with its buffer pool memory area provides a general-purpose and durable way to keep most or all data in memory, and `NDBCLUSTER` provides fast key-value lookups for huge distributed data sets.

- `CSV`: Its tables are really text files with comma-separated values. CSV tables let you import or dump data in CSV format, to exchange data with scripts and applications that read and write that same format. Because CSV tables are not indexed, you typically keep the data in `InnoDB` tables during normal operation, and only use CSV tables during the import or export stage.

- `Archive`: These compact, unindexed tables are intended for storing and retrieving large amounts of seldom-referenced historical, archived, or security audit information.

- `Blackhole`: The Blackhole storage engine accepts but does not store data, similar to the Unix `/dev/null` device. Queries always return an empty set. These tables can be used in replication configurations where DML statements are sent to replica servers, but the source server does not keep its own copy of the data.
- `NDB` (also known as `NDBCLUSTER`): This clustered database engine is particularly suited for applications that require the highest possible degree of uptime and availability.
- `Merge`: Enables a MySQL DBA or developer to logically group a series of identical `MyISAM` tables and reference them as one object. Good for VLDB environments such as data warehousing.
- `Federated`: Offers the ability to link separate MySQL servers to create one logical database from many physical servers. Very good for distributed or data mart environments.
- `Example`: This engine serves as an example in the MySQL source code that illustrates how to begin writing new storage engines. It is primarily of interest to developers. The storage engine is a “stub” that does nothing. You can create tables with this engine, but no data can be stored in them or retrieved from them.

You are not restricted to using the same storage engine for an entire server or schema. You can specify the storage engine for any table. For example, an application might use mostly `InnoDB` tables, with one `CSV` table for exporting data to a spreadsheet and a few `MEMORY` tables for temporary workspaces.

Choosing a Storage Engine

The various storage engines provided with MySQL are designed with different use cases in mind. The following table provides an overview of some storage engines provided with MySQL, with clarifying notes following the table.

Table 16.1 Storage Engines Feature Summary

Feature	MyISAM	Memory	InnoDB	Archive	NDB
B-tree indexes	Yes	Yes	Yes	No	No
Backup/point-in-time recovery (note 1)	Yes	Yes	Yes	Yes	Yes
Cluster database support	No	No	No	No	Yes
Clustered indexes	No	No	Yes	No	No
Compressed data	Yes (note 2)	No	Yes	Yes	No
Data caches	No	N/A	Yes	No	Yes
Encrypted data	Yes (note 3)	Yes (note 3)	Yes (note 4)	Yes (note 3)	Yes (note 3)
Foreign key support	No	No	Yes	No	Yes (note 5)
Full-text search indexes	Yes	No	Yes (note 6)	No	No
Geospatial data type support	Yes	No	Yes	Yes	Yes
Geospatial indexing support	Yes	No	Yes (note 7)	No	No

Feature	MyISAM	Memory	InnoDB	Archive	NDB
Hash indexes	No	Yes	No (note 8)	No	Yes
Index caches	Yes	N/A	Yes	No	Yes
Locking granularity	Table	Table	Row	Row	Row
MVCC	No	No	Yes	No	No
Replication support (note 1)	Yes	Limited (note 9)	Yes	Yes	Yes
Storage limits	256TB	RAM	64TB	None	384EB
T-tree indexes	No	No	No	No	Yes
Transactions	No	No	Yes	No	Yes
Update statistics for data dictionary	Yes	Yes	Yes	Yes	Yes

Notes:

1. Implemented in the server, rather than in the storage engine.

2. Compressed MyISAM tables are supported only when using the compressed row format. Tables using the compressed row format with MyISAM are read only.

3. Implemented in the server via encryption functions.

4. Implemented in the server via encryption functions; In MySQL 5.7 and later, data-at-rest encryption is supported.

5. Support for foreign keys is available in MySQL Cluster NDB 7.3 and later.

6. Support for FULLTEXT indexes is available in MySQL 5.6 and later.

7. Support for geospatial indexing is available in MySQL 5.7 and later.

8. InnoDB utilizes hash indexes internally for its Adaptive Hash Index feature.

9. See the discussion later in this section.

16.1 Setting the Storage Engine

When you create a new table, you can specify which storage engine to use by adding an `ENGINE` table option to the `CREATE TABLE` statement:

```
-- ENGINE=INNODB not needed unless you have set a different
-- default storage engine.
CREATE TABLE t1 (i INT) ENGINE = INNODB;
-- Simple table definitions can be switched from one to another.
CREATE TABLE t2 (i INT) ENGINE = CSV;
CREATE TABLE t3 (i INT) ENGINE = MEMORY;
```

When you omit the `ENGINE` option, the default storage engine is used. The default engine is `InnoDB` in MySQL 8.0. You can specify the default engine by using the `--default-storage-engine` server startup option, or by setting the `default-storage-engine` option in the `my.cnf` configuration file.

You can set the default storage engine for the current session by setting the `default_storage_engine` variable:

```
SET default_storage_engine=NDBCLUSTER;
```

The storage engine for `TEMPORARY` tables created with `CREATE TEMPORARY TABLE` can be set separately from the engine for permanent tables by setting the `default_tmp_storage_engine`, either at startup or at runtime.

To convert a table from one storage engine to another, use an `ALTER TABLE` statement that indicates the new engine:

```
ALTER TABLE t ENGINE = InnoDB;
```

See Section 13.1.20, “CREATE TABLE Statement”, and Section 13.1.9, “ALTER TABLE Statement”.

If you try to use a storage engine that is not compiled in or that is compiled in but deactivated, MySQL instead creates a table using the default storage engine. For example, in a replication setup, perhaps your source server uses `InnoDB` tables for maximum safety, but the replica servers use other storage engines for speed at the expense of durability or concurrency.

By default, a warning is generated whenever `CREATE TABLE` or `ALTER TABLE` cannot use the default storage engine. To prevent confusing, unintended behavior if the desired engine is unavailable, enable the `NO_ENGINE_SUBSTITUTION` SQL mode. If the desired engine is unavailable, this setting produces an error instead of a warning, and the table is not created or altered. See Section 5.1.11, “Server SQL Modes”.

MySQL may store a table's index and data in one or more other files, depending on the storage engine. Table and column definitions are stored in the MySQL data dictionary. Individual storage engines create any additional files required for the tables that they manage. If a table name contains special characters, the names for the table files contain encoded versions of those characters as described in Section 9.2.4, “Mapping of Identifiers to File Names”.

16.2 The MyISAM Storage Engine

`MyISAM` is based on the older (and no longer available) `ISAM` storage engine but has many useful extensions.

Table 16.2 MyISAM Storage Engine Features

Feature	Support
B-tree indexes	Yes
Backup/point-in-time recovery (Implemented in the server, rather than in the storage engine.)	Yes
Cluster database support	No
Clustered indexes	No
Compressed data	Yes (Compressed MyISAM tables are supported only when using the compressed row format. Tables using the compressed row format with MyISAM are read only.)
Data caches	No
Encrypted data	Yes (Implemented in the server via encryption functions.)
Foreign key support	No
Full-text search indexes	Yes
Geospatial data type support	Yes
Geospatial indexing support	Yes
Hash indexes	No

Feature	Support
Index caches	Yes
Locking granularity	Table
MVCC	No
Replication support (Implemented in the server, rather than in the storage engine.)	Yes
Storage limits	256TB
T-tree indexes	No
Transactions	No
Update statistics for data dictionary	Yes

Each `MyISAM` table is stored on disk in two files. The files have names that begin with the table name and have an extension to indicate the file type. The data file has an `.MYD` (`MYData`) extension. The index file has an `.MYI` (`MYIndex`) extension. The table definition is stored in the MySQL data dictionary.

To specify explicitly that you want a `MyISAM` table, indicate that with an `ENGINE` table option:

```
CREATE TABLE t (i INT) ENGINE = MYISAM;
```

In MySQL 8.0, it is normally necessary to use `ENGINE` to specify the `MyISAM` storage engine because `InnoDB` is the default engine.

You can check or repair `MyISAM` tables with the `mysqlcheck` client or `myisamchk` utility. You can also compress `MyISAM` tables with `myisampack` to take up much less space. See Section 4.5.3, “mysqlcheck — A Table Maintenance Program”, Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”, and Section 4.6.6, “myisampack — Generate Compressed, Read-Only MyISAM Tables”.

In MySQL 8.0, the `MyISAM` storage engine provides no partitioning support. *Partitioned `MyISAM` tables created in previous versions of MySQL cannot be used in MySQL 8.0*. For more information, see Section 24.6.2, “Partitioning Limitations Relating to Storage Engines”. For help with upgrading such tables so that they can be used in MySQL 8.0, see Section 2.10.4, “Changes in MySQL 8.0”.

`MyISAM` tables have the following characteristics:

- All data values are stored with the low byte first. This makes the data machine and operating system independent. The only requirements for binary portability are that the machine uses two's-complement signed integers and IEEE floating-point format. These requirements are widely used among mainstream machines. Binary compatibility might not be applicable to embedded systems, which sometimes have peculiar processors.

  There is no significant speed penalty for storing data low byte first; the bytes in a table row normally are unaligned and it takes little more processing to read an unaligned byte in order than in reverse order. Also, the code in the server that fetches column values is not time critical compared to other code.

- All numeric key values are stored with the high byte first to permit better index compression.

- Large files (up to 63-bit file length) are supported on file systems and operating systems that support large files.

- There is a limit of $(2^{32})^2$ (1.844E+19) rows in a `MyISAM` table.

- The maximum number of indexes per `MyISAM` table is 64.

  The maximum number of columns per index is 16.

- The maximum key length is 1000 bytes. This can also be changed by changing the source and recompiling. For the case of a key longer than 250 bytes, a larger key block size than the default of 1024 bytes is used.
- When rows are inserted in sorted order (as when you are using an `AUTO_INCREMENT` column), the index tree is split so that the high node only contains one key. This improves space utilization in the index tree.
- Internal handling of one `AUTO_INCREMENT` column per table is supported. `MyISAM` automatically updates this column for `INSERT` and `UPDATE` operations. This makes `AUTO_INCREMENT` columns faster (at least 10%). Values at the top of the sequence are not reused after being deleted. (When an `AUTO_INCREMENT` column is defined as the last column of a multiple-column index, reuse of values deleted from the top of a sequence does occur.) The `AUTO_INCREMENT` value can be reset with `ALTER TABLE` or `myisamchk`.
- Dynamic-sized rows are much less fragmented when mixing deletes with updates and inserts. This is done by automatically combining adjacent deleted blocks and by extending blocks if the next block is deleted.
- `MyISAM` supports concurrent inserts: If a table has no free blocks in the middle of the data file, you can `INSERT` new rows into it at the same time that other threads are reading from the table. A free block can occur as a result of deleting rows or an update of a dynamic length row with more data than its current contents. When all free blocks are used up (filled in), future inserts become concurrent again. See Section 8.11.3, “Concurrent Inserts”.
- You can put the data file and index file in different directories on different physical devices to get more speed with the `DATA DIRECTORY` and `INDEX DIRECTORY` table options to `CREATE TABLE`. See Section 13.1.20, “CREATE TABLE Statement”.
- `BLOB` and `TEXT` columns can be indexed.
- `NULL` values are permitted in indexed columns. This takes 0 to 1 bytes per key.
- Each character column can have a different character set. See Chapter 10, *Character Sets, Collations, Unicode*.
- There is a flag in the `MyISAM` index file that indicates whether the table was closed correctly. If `mysqld` is started with the `myisam_recover_options` system variable set, `MyISAM` tables are automatically checked when opened, and are repaired if the table wasn't closed properly.
- `myisamchk` marks tables as checked if you run it with the `--update-state` option. `myisamchk --fast` checks only those tables that don't have this mark.
- `myisamchk --analyze` stores statistics for portions of keys, as well as for entire keys.
- `myisampack` can pack `BLOB` and `VARCHAR` columns.

`MyISAM` also supports the following features:

- Support for a true `VARCHAR` type; a `VARCHAR` column starts with a length stored in one or two bytes.
- Tables with `VARCHAR` columns may have fixed or dynamic row length.
- The sum of the lengths of the `VARCHAR` and `CHAR` columns in a table may be up to 64KB.
- Arbitrary length `UNIQUE` constraints.

Additional Resources

- A forum dedicated to the `MyISAM` storage engine is available at https://forums.mysql.com/list.php?21.

16.2.1 MyISAM Startup Options

The following options to `mysqld` can be used to change the behavior of `MyISAM` tables. For additional information, see Section 5.1.7, “Server Command Options”.

Table 16.3 MyISAM Option and Variable Reference

Name	Cmd-Line	Option File	System Var	Status Var	Var Scope	Dynamic
bulk_insert_buffer_size	Yes	Yes	Yes		Both	Yes
concurrent_insert	Yes	Yes	Yes		Global	Yes
delay_key_write	Yes	Yes	Yes		Global	Yes
have_rtree_keys			Yes		Global	No
key_buffer_size	Yes	Yes	Yes		Global	Yes
log-isam	Yes	Yes				
myisam-block-size	Yes	Yes				
myisam_data_pointer_size	Yes	Yes	Yes		Global	Yes
myisam_max_sort_file_size	Yes	Yes	Yes		Global	Yes
myisam_mmap_size	Yes	Yes	Yes		Global	No
myisam_recover_options	Yes	Yes	Yes		Global	No
myisam_repair_threads	Yes	Yes	Yes		Both	Yes
myisam_sort_buffer_size	Yes	Yes	Yes		Both	Yes
myisam_stats_method	Yes	Yes	Yes		Both	Yes
myisam_use_mmap	Yes	Yes	Yes		Global	Yes
tmp_table_size	Yes	Yes	Yes		Both	Yes

The following system variables affect the behavior of `MyISAM` tables. For additional information, see Section 5.1.8, “Server System Variables”.

- `bulk_insert_buffer_size`

  The size of the tree cache used in bulk insert optimization.

  Note

  This is a limit *per thread*!

- `delay_key_write=ALL`

  Don't flush key buffers between writes for any `MyISAM` table.

  Note

  If you do this, you should not access `MyISAM` tables from another program (such as from another MySQL server or with `myisamchk`) when the tables are in use. Doing so risks index corruption. Using `--external-locking` does not eliminate this risk.

- `myisam_max_sort_file_size`

  The maximum size of the temporary file that MySQL is permitted to use while re-creating a `MyISAM` index (during `REPAIR TABLE`, `ALTER TABLE`, or `LOAD DATA`). If the file size would be larger than this value, the index is created using the key cache instead, which is slower. The value is given in bytes.

- `myisam_recover_options=mode`

  Set the mode for automatic recovery of crashed `MyISAM` tables.

- `myisam_sort_buffer_size`

  Set the size of the buffer used when recovering tables.

Automatic recovery is activated if you start `mysqld` with the `myisam_recover_options` system variable set. In this case, when the server opens a `MyISAM` table, it checks whether the table is marked as crashed or whether the open count variable for the table is not 0 and you are running the server with external locking disabled. If either of these conditions is true, the following happens:

- The server checks the table for errors.
- If the server finds an error, it tries to do a fast table repair (with sorting and without re-creating the data file).
- If the repair fails because of an error in the data file (for example, a duplicate-key error), the server tries again, this time re-creating the data file.
- If the repair still fails, the server tries once more with the old repair option method (write row by row without sorting). This method should be able to repair any type of error and has low disk space requirements.

If the recovery wouldn't be able to recover all rows from previously completed statements and you didn't specify `FORCE` in the value of the `myisam_recover_options` system variable, automatic repair aborts with an error message in the error log:

```
Error: Couldn't repair table: test.g00pages
```

If you specify `FORCE`, a warning like this is written instead:

```
Warning: Found 344 of 354 rows when repairing ./test/g00pages
```

If the automatic recovery value includes `BACKUP`, the recovery process creates files with names of the form *`tbl_name-datetime`*`.BAK`. You should have a `cron` script that automatically moves these files from the database directories to backup media.

16.2.2 Space Needed for Keys

`MyISAM` tables use B-tree indexes. You can roughly calculate the size for the index file as `(key_length+4)/0.67`, summed over all keys. This is for the worst case when all keys are inserted in sorted order and the table doesn't have any compressed keys.

String indexes are space compressed. If the first index part is a string, it is also prefix compressed. Space compression makes the index file smaller than the worst-case figure if a string column has a lot of trailing space or is a `VARCHAR` column that is not always used to the full length. Prefix compression is used on keys that start with a string. Prefix compression helps if there are many strings with an identical prefix.

In `MyISAM` tables, you can also prefix compress numbers by specifying the `PACK_KEYS=1` table option when you create the table. Numbers are stored with the high byte first, so this helps when you have many integer keys that have an identical prefix.

16.2.3 MyISAM Table Storage Formats

`MyISAM` supports three different storage formats. Two of them, fixed and dynamic format, are chosen automatically depending on the type of columns you are using. The third, compressed format, can be

created only with the `myisampack` utility (see Section 4.6.6, “myisampack — Generate Compressed, Read-Only MyISAM Tables”).

When you use `CREATE TABLE` or `ALTER TABLE` for a table that has no `BLOB` or `TEXT` columns, you can force the table format to `FIXED` or `DYNAMIC` with the `ROW_FORMAT` table option.

See Section 13.1.20, “CREATE TABLE Statement”, for information about `ROW_FORMAT`.

You can decompress (unpack) compressed `MyISAM` tables using `myisamchk --unpack`; see Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”, for more information.

16.2.3.1 Static (Fixed-Length) Table Characteristics

Static format is the default for `MyISAM` tables. It is used when the table contains no variable-length columns (`VARCHAR`, `VARBINARY`, `BLOB`, or `TEXT`). Each row is stored using a fixed number of bytes.

Of the three `MyISAM` storage formats, static format is the simplest and most secure (least subject to corruption). It is also the fastest of the on-disk formats due to the ease with which rows in the data file can be found on disk: To look up a row based on a row number in the index, multiply the row number by the row length to calculate the row position. Also, when scanning a table, it is very easy to read a constant number of rows with each disk read operation.

The security is evidenced if your computer crashes while the MySQL server is writing to a fixed-format `MyISAM` file. In this case, `myisamchk` can easily determine where each row starts and ends, so it can usually reclaim all rows except the partially written one. `MyISAM` table indexes can always be reconstructed based on the data rows.

Note

Fixed-length row format is available only for tables having no `BLOB` or `TEXT` columns. Creating a table having such columns with an explicit `ROW_FORMAT` clause does not raise an error or warning; the format specification is ignored.

Static-format tables have these characteristics:

- `CHAR` and `VARCHAR` columns are space-padded to the specified column width, although the column type is not altered. `BINARY` and `VARBINARY` columns are padded with `0x00` bytes to the column width.
- `NULL` columns require additional space in the row to record whether their values are `NULL`. Each `NULL` column takes one bit extra, rounded up to the nearest byte.
- Very quick.
- Easy to cache.
- Easy to reconstruct after a crash, because rows are located in fixed positions.
- Reorganization is unnecessary unless you delete a huge number of rows and want to return free disk space to the operating system. To do this, use `OPTIMIZE TABLE` or `myisamchk -r`.
- Usually require more disk space than dynamic-format tables.
- The expected row length in bytes for static-sized rows is calculated using the following expression:

```
row length = 1
             + (sum of column lengths)
             + (number of NULL columns + delete_flag + 7)/8
             + (number of variable-length columns)
```

`delete_flag` is 1 for tables with static row format. Static tables use a bit in the row record for a flag that indicates whether the row has been deleted. *`delete_flag`* is 0 for dynamic tables because the flag is stored in the dynamic row header.

16.2.3.2 Dynamic Table Characteristics

Dynamic storage format is used if a `MyISAM` table contains any variable-length columns (`VARCHAR`, `VARBINARY`, `BLOB`, or `TEXT`), or if the table was created with the `ROW_FORMAT=DYNAMIC` table option.

Dynamic format is a little more complex than static format because each row has a header that indicates how long it is. A row can become fragmented (stored in noncontiguous pieces) when it is made longer as a result of an update.

You can use `OPTIMIZE TABLE` or `myisamchk -r` to defragment a table. If you have fixed-length columns that you access or change frequently in a table that also contains some variable-length columns, it might be a good idea to move the variable-length columns to other tables just to avoid fragmentation.

Dynamic-format tables have these characteristics:

- All string columns are dynamic except those with a length less than four.
- Each row is preceded by a bitmap that indicates which columns contain the empty string (for string columns) or zero (for numeric columns). This does not include columns that contain `NULL` values. If a string column has a length of zero after trailing space removal, or a numeric column has a value of zero, it is marked in the bitmap and not saved to disk. Nonempty strings are saved as a length byte plus the string contents.
- `NULL` columns require additional space in the row to record whether their values are `NULL`. Each `NULL` column takes one bit extra, rounded up to the nearest byte.
- Much less disk space usually is required than for fixed-length tables.
- Each row uses only as much space as is required. However, if a row becomes larger, it is split into as many pieces as are required, resulting in row fragmentation. For example, if you update a row with information that extends the row length, the row becomes fragmented. In this case, you may have to run `OPTIMIZE TABLE` or `myisamchk -r` from time to time to improve performance. Use `myisamchk -ei` to obtain table statistics.
- More difficult than static-format tables to reconstruct after a crash, because rows may be fragmented into many pieces and links (fragments) may be missing.
- The expected row length for dynamic-sized rows is calculated using the following expression:

```
3
+ (number of columns + 7) / 8
+ (number of char columns)
+ (packed size of numeric columns)
+ (length of strings)
+ (number of NULL columns + 7) / 8
```

  There is a penalty of 6 bytes for each link. A dynamic row is linked whenever an update causes an enlargement of the row. Each new link is at least 20 bytes, so the next enlargement probably goes in the same link. If not, another link is created. You can find the number of links using `myisamchk -ed`. All links may be removed with `OPTIMIZE TABLE` or `myisamchk -r`.

16.2.3.3 Compressed Table Characteristics

Compressed storage format is a read-only format that is generated with the `myisampack` tool. Compressed tables can be uncompressed with `myisamchk`.

Compressed tables have the following characteristics:

- Compressed tables take very little disk space. This minimizes disk usage, which is helpful when using slow disks (such as CD-ROMs).

- Each row is compressed separately, so there is very little access overhead. The header for a row takes up one to three bytes depending on the biggest row in the table. Each column is compressed differently. There is usually a different Huffman tree for each column. Some of the compression types are:
  - Suffix space compression.
  - Prefix space compression.
  - Numbers with a value of zero are stored using one bit.
  - If values in an integer column have a small range, the column is stored using the smallest possible type. For example, a `BIGINT` column (eight bytes) can be stored as a `TINYINT` column (one byte) if all its values are in the range from `-128` to `127`.
  - If a column has only a small set of possible values, the data type is converted to `ENUM`.
  - A column may use any combination of the preceding compression types.
- Can be used for fixed-length or dynamic-length rows.

Note

While a compressed table is read only, and you cannot therefore update or add rows in the table, DDL (Data Definition Language) operations are still valid. For example, you may still use `DROP` to drop the table, and `TRUNCATE TABLE` to empty the table.

16.2.4 MyISAM Table Problems

The file format that MySQL uses to store data has been extensively tested, but there are always circumstances that may cause database tables to become corrupted. The following discussion describes how this can happen and how to handle it.

16.2.4.1 Corrupted MyISAM Tables

Even though the `MyISAM` table format is very reliable (all changes to a table made by an SQL statement are written before the statement returns), you can still get corrupted tables if any of the following events occur:

- The `mysqld` process is killed in the middle of a write.
- An unexpected computer shutdown occurs (for example, the computer is turned off).
- Hardware failures.
- You are using an external program (such as `myisamchk`) to modify a table that is being modified by the server at the same time.
- A software bug in the MySQL or `MyISAM` code.

Typical symptoms of a corrupt table are:

- You get the following error while selecting data from the table:

```
Incorrect key file for table: '...'. Try to repair it
```

- Queries don't find rows in the table or return incomplete results.

You can check the health of a `MyISAM` table using the `CHECK TABLE` statement, and repair a corrupted `MyISAM` table with `REPAIR TABLE`. When `mysqld` is not running, you can also check or repair a table with the `myisamchk` command. See Section 13.7.3.2, “CHECK TABLE Statement”,

Section 13.7.3.5, “REPAIR TABLE Statement”, and Section 4.6.4, “myisamchk — MyISAM Table-Maintenance Utility”.

If your tables become corrupted frequently, you should try to determine why this is happening. The most important thing to know is whether the table became corrupted as a result of an unexpected server exit. You can verify this easily by looking for a recent `restarted mysqld` message in the error log. If there is such a message, it is likely that table corruption is a result of the server dying. Otherwise, corruption may have occurred during normal operation. This is a bug. You should try to create a reproducible test case that demonstrates the problem. See Section B.3.3.3, “What to Do If MySQL Keeps Crashing”, and Section 5.9, “Debugging MySQL”.

16.2.4.2 Problems from Tables Not Being Closed Properly

Each `MyISAM` index file (`.MYI` file) has a counter in the header that can be used to check whether a table has been closed properly. If you get the following warning from `CHECK TABLE` or `myisamchk`, it means that this counter has gone out of sync:

```
clients are using or haven't closed the table properly
```

This warning doesn't necessarily mean that the table is corrupted, but you should at least check the table.

The counter works as follows:

- The first time a table is updated in MySQL, a counter in the header of the index files is incremented.
- The counter is not changed during further updates.
- When the last instance of a table is closed (because a `FLUSH TABLES` operation was performed or because there is no room in the table cache), the counter is decremented if the table has been updated at any point.
- When you repair the table or check the table and it is found to be okay, the counter is reset to zero.
- To avoid problems with interaction with other processes that might check the table, the counter is not decremented on close if it was zero.

In other words, the counter can become incorrect only under these conditions:

- A `MyISAM` table is copied without first issuing `LOCK TABLES` and `FLUSH TABLES`.
- MySQL has crashed between an update and the final close. (The table may still be okay because MySQL always issues writes for everything between each statement.)
- A table was modified by `myisamchk --recover` or `myisamchk --update-state` at the same time that it was in use by `mysqld`.
- Multiple `mysqld` servers are using the table and one server performed a `REPAIR TABLE` or `CHECK TABLE` on the table while it was in use by another server. In this setup, it is safe to use `CHECK TABLE`, although you might get the warning from other servers. However, `REPAIR TABLE` should be avoided because when one server replaces the data file with a new one, this is not known to the other servers.

  In general, it is a bad idea to share a data directory among multiple servers. See Section 5.8, “Running Multiple MySQL Instances on One Machine”, for additional discussion.

16.3 The MEMORY Storage Engine

The `MEMORY` storage engine (formerly known as `HEAP`) creates special-purpose tables with contents that are stored in memory. Because the data is vulnerable to crashes, hardware issues, or power outages, only use these tables as temporary work areas or read-only caches for data pulled from other tables.

Table 16.4 MEMORY Storage Engine Features

Feature	Support
B-tree indexes	Yes
Backup/point-in-time recovery (Implemented in the server, rather than in the storage engine.)	Yes
Cluster database support	No
Clustered indexes	No
Compressed data	No
Data caches	N/A
Encrypted data	Yes (Implemented in the server via encryption functions.)
Foreign key support	No
Full-text search indexes	No
Geospatial data type support	No
Geospatial indexing support	No
Hash indexes	Yes
Index caches	N/A
Locking granularity	Table
MVCC	No
Replication support (Implemented in the server, rather than in the storage engine.)	Limited (See the discussion later in this section.)
Storage limits	RAM
T-tree indexes	No
Transactions	No
Update statistics for data dictionary	Yes

When to Use MEMORY or NDB Cluster

Developers looking to deploy applications that use the `MEMORY` storage engine for important, highly available, or frequently updated data should consider whether NDB Cluster is a better choice. A typical use case for the `MEMORY` engine involves these characteristics:

- Operations involving transient, non-critical data such as session management or caching. When the MySQL server halts or restarts, the data in `MEMORY` tables is lost.
- In-memory storage for fast access and low latency. Data volume can fit entirely in memory without causing the operating system to swap out virtual memory pages.
- A read-only or read-mostly data access pattern (limited updates).

NDB Cluster offers the same features as the `MEMORY` engine with higher performance levels, and provides additional features not available with `MEMORY`:

- Row-level locking and multiple-thread operation for low contention between clients.
- Scalability even with statement mixes that include writes.
- Optional disk-backed operation for data durability.
- Shared-nothing architecture and multiple-host operation with no single point of failure, enabling 99.999% availability.
- Automatic data distribution across nodes; application developers need not craft custom sharding or partitioning solutions.
- Support for variable-length data types (including `BLOB` and `TEXT`) not supported by `MEMORY`.

Partitioning

`MEMORY` tables cannot be partitioned.

Performance Characteristics

`MEMORY` performance is constrained by contention resulting from single-thread execution and table lock overhead when processing updates. This limits scalability when load increases, particularly for statement mixes that include writes.

Despite the in-memory processing for `MEMORY` tables, they are not necessarily faster than `InnoDB` tables on a busy server, for general-purpose queries, or under a read/write workload. In particular, the table locking involved with performing updates can slow down concurrent usage of `MEMORY` tables from multiple sessions.

Depending on the kinds of queries performed on a `MEMORY` table, you might create indexes as either the default hash data structure (for looking up single values based on a unique key), or a general-purpose B-tree data structure (for all kinds of queries involving equality, inequality, or range operators such as less than or greater than). The following sections illustrate the syntax for creating both kinds of indexes. A common performance issue is using the default hash indexes in workloads where B-tree indexes are more efficient.

Characteristics of MEMORY Tables

The `MEMORY` storage engine does not create any files on disk. The table definition is stored in the MySQL data dictionary.

`MEMORY` tables have the following characteristics:

- Space for `MEMORY` tables is allocated in small blocks. Tables use 100% dynamic hashing for inserts. No overflow area or extra key space is needed. No extra space is needed for free lists. Deleted rows are put in a linked list and are reused when you insert new data into the table. `MEMORY` tables also have none of the problems commonly associated with deletes plus inserts in hashed tables.
- `MEMORY` tables use a fixed-length row-storage format. Variable-length types such as `VARCHAR` are stored using a fixed length.

- MEMORY tables cannot contain BLOB or TEXT columns.
- MEMORY includes support for AUTO_INCREMENT columns.
- Non-TEMPORARY MEMORY tables are shared among all clients, just like any other non-TEMPORARY table.

DDL Operations for MEMORY Tables

To create a MEMORY table, specify the clause ENGINE=MEMORY on the CREATE TABLE statement.

```
CREATE TABLE t (i INT) ENGINE = MEMORY;
```

As indicated by the engine name, MEMORY tables are stored in memory. They use hash indexes by default, which makes them very fast for single-value lookups, and very useful for creating temporary tables. However, when the server shuts down, all rows stored in MEMORY tables are lost. The tables themselves continue to exist because their definitions are stored in the MySQL data dictionary, but they are empty when the server restarts.

This example shows how you might create, use, and remove a MEMORY table:

```
mysql> CREATE TABLE test ENGINE=MEMORY
           SELECT ip,SUM(downloads) AS down
           FROM log_table GROUP BY ip;
mysql> SELECT COUNT(ip),AVG(down) FROM test;
mysql> DROP TABLE test;
```

The maximum size of MEMORY tables is limited by the max_heap_table_size system variable, which has a default value of 16MB. To enforce different size limits for MEMORY tables, change the value of this variable. The value in effect for CREATE TABLE, or a subsequent ALTER TABLE or TRUNCATE TABLE, is the value used for the life of the table. A server restart also sets the maximum size of existing MEMORY tables to the global max_heap_table_size value. You can set the size for individual tables as described later in this section.

Indexes

The MEMORY storage engine supports both HASH and BTREE indexes. You can specify one or the other for a given index by adding a USING clause as shown here:

```
CREATE TABLE lookup
    (id INT, INDEX USING HASH (id))
    ENGINE = MEMORY;
CREATE TABLE lookup
    (id INT, INDEX USING BTREE (id))
    ENGINE = MEMORY;
```

For general characteristics of B-tree and hash indexes, see Section 8.3.1, “How MySQL Uses Indexes”.

MEMORY tables can have up to 64 indexes per table, 16 columns per index and a maximum key length of 3072 bytes.

If a MEMORY table hash index has a high degree of key duplication (many index entries containing the same value), updates to the table that affect key values and all deletes are significantly slower. The degree of this slowdown is proportional to the degree of duplication (or, inversely proportional to the index cardinality). You can use a BTREE index to avoid this problem.

MEMORY tables can have nonunique keys. (This is an uncommon feature for implementations of hash indexes.)

Columns that are indexed can contain NULL values.

User-Created and Temporary Tables

`MEMORY` table contents are stored in memory, which is a property that `MEMORY` tables share with internal temporary tables that the server creates on the fly while processing queries. However, the two types of tables differ in that `MEMORY` tables are not subject to storage conversion, whereas internal temporary tables are:

- If an internal temporary table becomes too large, the server automatically converts it to on-disk storage, as described in Section 8.4.4, “Internal Temporary Table Use in MySQL”.
- User-created `MEMORY` tables are never converted to disk tables.

Loading Data

To populate a `MEMORY` table when the MySQL server starts, you can use the `init_file` system variable. For example, you can put statements such as `INSERT INTO ... SELECT` or `LOAD DATA` into a file to load the table from a persistent data source, and use `init_file` to name the file. See Section 5.1.8, “Server System Variables”, and Section 13.2.9, “LOAD DATA Statement”.

MEMORY Tables and Replication

When a replication source server shuts down and restarts, its `MEMORY` tables become empty. To replicate this effect to replicas, the first time that the source uses a given `MEMORY` table after startup, it logs an event that notifies replicas that the table must be emptied by writing a `DELETE` or (from MySQL 8.0.22) `TRUNCATE TABLE` statement for that table to the binary log. When a replica server shuts down and restarts, its `MEMORY` tables also become empty, and it writes a `DELETE` or (from MySQL 8.0.22) `TRUNCATE TABLE` statement to its own binary log, which is passed on to any downstream replicas.

When you use `MEMORY` tables in a replication topology, in some situations, the table on the source and the table on the replica can differ. For information on handling each of these situations to prevent stale reads or errors, see Section 17.5.1.21, “Replication and MEMORY Tables”.

Managing Memory Use

The server needs sufficient memory to maintain all `MEMORY` tables that are in use at the same time.

Memory is not reclaimed if you delete individual rows from a `MEMORY` table. Memory is reclaimed only when the entire table is deleted. Memory that was previously used for deleted rows is re-used for new rows within the same table. To free all the memory used by a `MEMORY` table when you no longer require its contents, execute `DELETE` or `TRUNCATE TABLE` to remove all rows, or remove the table altogether using `DROP TABLE`. To free up the memory used by deleted rows, use `ALTER TABLE ENGINE=MEMORY` to force a table rebuild.

The memory needed for one row in a `MEMORY` table is calculated using the following expression:

```
SUM_OVER_ALL_BTREE_KEYS(max_length_of_key + sizeof(char*) * 4)
+ SUM_OVER_ALL_HASH_KEYS(sizeof(char*) * 2)
+ ALIGN(length_of_row+1, sizeof(char*))
```

`ALIGN()` represents a round-up factor to cause the row length to be an exact multiple of the `char` pointer size. `sizeof(char*)` is 4 on 32-bit machines and 8 on 64-bit machines.

As mentioned earlier, the `max_heap_table_size` system variable sets the limit on the maximum size of `MEMORY` tables. To control the maximum size for individual tables, set the session value of this variable before creating each table. (Do not change the global `max_heap_table_size` value unless you intend the value to be used for `MEMORY` tables created by all clients.) The following example creates two `MEMORY` tables, with a maximum size of 1MB and 2MB, respectively:

```
mysql> SET max_heap_table_size = 1024*1024;
```

```
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t1 (id INT, UNIQUE(id)) ENGINE = MEMORY;
Query OK, 0 rows affected (0.01 sec)

mysql> SET max_heap_table_size = 1024*1024*2;
Query OK, 0 rows affected (0.00 sec)

mysql> CREATE TABLE t2 (id INT, UNIQUE(id)) ENGINE = MEMORY;
Query OK, 0 rows affected (0.00 sec)
```

Both tables revert to the server's global `max_heap_table_size` value if the server restarts.

You can also specify a `MAX_ROWS` table option in `CREATE TABLE` statements for `MEMORY` tables to provide a hint about the number of rows you plan to store in them. This does not enable the table to grow beyond the `max_heap_table_size` value, which still acts as a constraint on maximum table size. For maximum flexibility in being able to use `MAX_ROWS`, set `max_heap_table_size` at least as high as the value to which you want each `MEMORY` table to be able to grow.

Additional Resources

A forum dedicated to the `MEMORY` storage engine is available at https://forums.mysql.com/list.php?92.

16.4 The CSV Storage Engine

The `CSV` storage engine stores data in text files using comma-separated values format.

The `CSV` storage engine is always compiled into the MySQL server.

To examine the source for the `CSV` engine, look in the `storage/csv` directory of a MySQL source distribution.

When you create a `CSV` table, the server creates a plain text data file having a name that begins with the table name and has a `.CSV` extension. When you store data into the table, the storage engine saves it into the data file in comma-separated values format.

```
mysql> CREATE TABLE test (i INT NOT NULL, c CHAR(10) NOT NULL)
       ENGINE = CSV;
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO test VALUES(1,'record one'),(2,'record two');
Query OK, 2 rows affected (0.05 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM test;
+---+------------+
| i | c          |
+---+------------+
| 1 | record one |
| 2 | record two |
+---+------------+
2 rows in set (0.00 sec)
```

Creating a `CSV` table also creates a corresponding metafile that stores the state of the table and the number of rows that exist in the table. The name of this file is the same as the name of the table with the extension `CSM`.

If you examine the `test.CSV` file in the database directory created by executing the preceding statements, its contents should look like this:

```
"1","record one"
"2","record two"
```

This format can be read, and even written, by spreadsheet applications such as Microsoft Excel.

16.4.1 Repairing and Checking CSV Tables

The CSV storage engine supports the CHECK TABLE and REPAIR TABLE statements to verify and, if possible, repair a damaged CSV table.

When running the CHECK TABLE statement, the CSV file is checked for validity by looking for the correct field separators, escaped fields (matching or missing quotation marks), the correct number of fields compared to the table definition and the existence of a corresponding CSV metafile. The first invalid row discovered causes an error. Checking a valid table produces output like that shown here:

```
mysql> CHECK TABLE csvtest;
+--------------+-------+----------+----------+
| Table        | Op    | Msg_type | Msg_text |
+--------------+-------+----------+----------+
| test.csvtest | check | status   | OK       |
+--------------+-------+----------+----------+
```

A check on a corrupted table returns a fault such as

```
mysql> CHECK TABLE csvtest;
+--------------+-------+----------+----------+
| Table        | Op    | Msg_type | Msg_text |
+--------------+-------+----------+----------+
| test.csvtest | check | error    | Corrupt  |
+--------------+-------+----------+----------+
```

To repair a table, use REPAIR TABLE, which copies as many valid rows from the existing CSV data as possible, and then replaces the existing CSV file with the recovered rows. Any rows beyond the corrupted data are lost.

```
mysql> REPAIR TABLE csvtest;
+--------------+--------+----------+----------+
| Table        | Op     | Msg_type | Msg_text |
+--------------+--------+----------+----------+
| test.csvtest | repair | status   | OK       |
+--------------+--------+----------+----------+
```

Warning

During repair, only the rows from the CSV file up to the first damaged row are copied to the new table. All other rows from the first damaged row to the end of the table are removed, even valid rows.

16.4.2 CSV Limitations

The CSV storage engine does not support indexing.

The CSV storage engine does not support partitioning.

All tables that you create using the CSV storage engine must have the NOT NULL attribute on all columns.

16.5 The ARCHIVE Storage Engine

The ARCHIVE storage engine produces special-purpose tables that store large amounts of unindexed data in a very small footprint.

Table 16.5 ARCHIVE Storage Engine Features

Feature	Support
B-tree indexes	No

Feature	Support
Backup/point-in-time recovery (Implemented in the server, rather than in the storage engine.)	Yes
Cluster database support	No
Clustered indexes	No
Compressed data	Yes
Data caches	No
Encrypted data	Yes (Implemented in the server via encryption functions.)
Foreign key support	No
Full-text search indexes	No
Geospatial data type support	Yes
Geospatial indexing support	No
Hash indexes	No
Index caches	No
Locking granularity	Row
MVCC	No
Replication support (Implemented in the server, rather than in the storage engine.)	Yes
Storage limits	None
T-tree indexes	No
Transactions	No
Update statistics for data dictionary	Yes

The `ARCHIVE` storage engine is included in MySQL binary distributions. To enable this storage engine if you build MySQL from source, invoke `CMake` with the `-DWITH_ARCHIVE_STORAGE_ENGINE` option.

To examine the source for the `ARCHIVE` engine, look in the `storage/archive` directory of a MySQL source distribution.

You can check whether the `ARCHIVE` storage engine is available with the `SHOW ENGINES` statement.

When you create an `ARCHIVE` table, the storage engine creates files with names that begin with the table name. The data file has an extension of `.ARZ`. An `.ARN` file may appear during optimization operations.

The `ARCHIVE` engine supports `INSERT`, `REPLACE`, and `SELECT`, but not `DELETE` or `UPDATE`. It does support `ORDER BY` operations, `BLOB` columns, and spatial data types (see Section 11.4.1, “Spatial Data Types”). Geographic spatial reference systems are not supported. The `ARCHIVE` engine uses row-level locking.

The `ARCHIVE` engine supports the `AUTO_INCREMENT` column attribute. The `AUTO_INCREMENT` column can have either a unique or nonunique index. Attempting to create an index on any other column results in an error. The `ARCHIVE` engine also supports the `AUTO_INCREMENT` table option in `CREATE TABLE` statements to specify the initial sequence value for a new table or reset the sequence value for an existing table, respectively.

`ARCHIVE` does not support inserting a value into an `AUTO_INCREMENT` column less than the current maximum column value. Attempts to do so result in an `ER_DUP_KEY` error.

The `ARCHIVE` engine ignores `BLOB` columns if they are not requested and scans past them while reading.

The ARCHIVE storage engine does not support partitioning.

Storage: Rows are compressed as they are inserted. The ARCHIVE engine uses zlib lossless data compression (see http://www.zlib.net/). You can use OPTIMIZE TABLE to analyze the table and pack it into a smaller format (for a reason to use OPTIMIZE TABLE, see later in this section). The engine also supports CHECK TABLE. There are several types of insertions that are used:

- An INSERT statement just pushes rows into a compression buffer, and that buffer flushes as necessary. The insertion into the buffer is protected by a lock. A SELECT forces a flush to occur.
- A bulk insert is visible only after it completes, unless other inserts occur at the same time, in which case it can be seen partially. A SELECT never causes a flush of a bulk insert unless a normal insert occurs while it is loading.

Retrieval: On retrieval, rows are uncompressed on demand; there is no row cache. A SELECT operation performs a complete table scan: When a SELECT occurs, it finds out how many rows are currently available and reads that number of rows. SELECT is performed as a consistent read. Note that lots of SELECT statements during insertion can deteriorate the compression, unless only bulk inserts are used. To achieve better compression, you can use OPTIMIZE TABLE or REPAIR TABLE. The number of rows in ARCHIVE tables reported by SHOW TABLE STATUS is always accurate. See Section 13.7.3.4, “OPTIMIZE TABLE Statement”, Section 13.7.3.5, “REPAIR TABLE Statement”, and Section 13.7.7.38, “SHOW TABLE STATUS Statement”.

Additional Resources

- A forum dedicated to the ARCHIVE storage engine is available at https://forums.mysql.com/list.php?112.

16.6 The BLACKHOLE Storage Engine

The BLACKHOLE storage engine acts as a “black hole” that accepts data but throws it away and does not store it. Retrievals always return an empty result:

```
mysql> CREATE TABLE test(i INT, c CHAR(10)) ENGINE = BLACKHOLE;
Query OK, 0 rows affected (0.03 sec)

mysql> INSERT INTO test VALUES(1,'record one'),(2,'record two');
Query OK, 2 rows affected (0.00 sec)
Records: 2  Duplicates: 0  Warnings: 0

mysql> SELECT * FROM test;
Empty set (0.00 sec)
```

To enable the BLACKHOLE storage engine if you build MySQL from source, invoke CMake with the -DWITH_BLACKHOLE_STORAGE_ENGINE option.

To examine the source for the BLACKHOLE engine, look in the sql directory of a MySQL source distribution.

When you create a BLACKHOLE table, the server creates the table definition in the global data dictionary. There are no files associated with the table.

The BLACKHOLE storage engine supports all kinds of indexes. That is, you can include index declarations in the table definition.

The maximum key length is 3072 bytes as of MySQL 8.0.27. Prior to 8.0.27, the maximum key length is 1000 bytes.

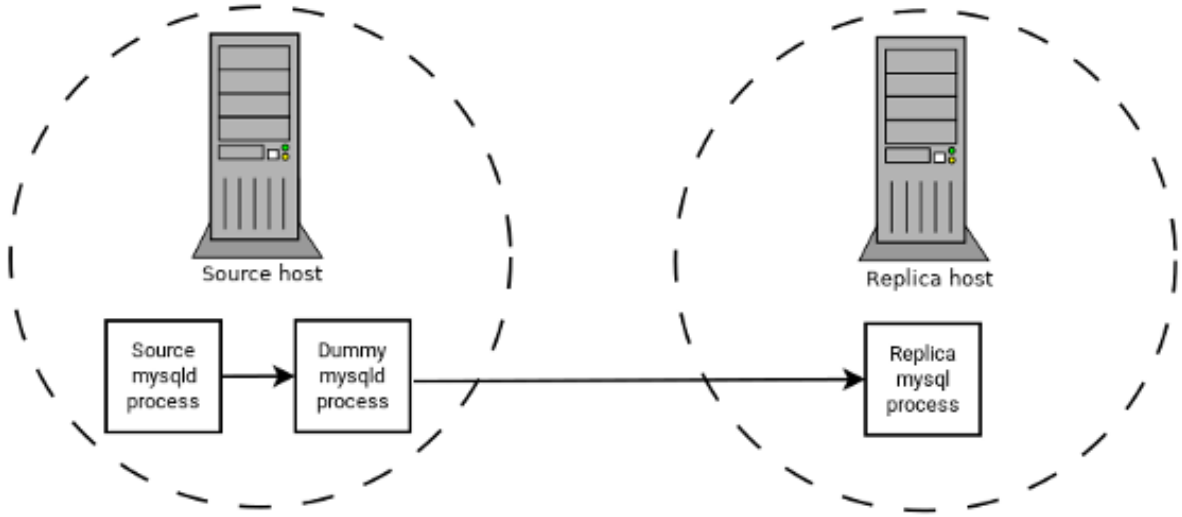
The BLACKHOLE storage engine does not support partitioning.

You can check whether the BLACKHOLE storage engine is available with the SHOW ENGINES statement.

Inserts into a BLACKHOLE table do not store any data, but if statement based binary logging is enabled, the SQL statements are logged and replicated to replica servers. This can be useful as a repeater or filter mechanism.

Suppose that your application requires replica-side filtering rules, but transferring all binary log data to the replica first results in too much traffic. In such a case, it is possible to set up on the replication source server a “dummy” replica process whose default storage engine is BLACKHOLE, depicted as follows:

Figure 16.1 Replication using BLACKHOLE for Filtering



The source writes to its binary log. The “dummy” mysqld process acts as a replica, applying the desired combination of replicate-do-* and replicate-ignore-* rules, and writes a new, filtered binary log of its own. (See Section 17.1.6, “Replication and Binary Logging Options and Variables”.) This filtered log is provided to the replica.

The dummy process does not actually store any data, so there is little processing overhead incurred by running the additional mysqld process on the replication source server. This type of setup can be repeated with additional replicas.

INSERT triggers for BLACKHOLE tables work as expected. However, because the BLACKHOLE table does not actually store any data, UPDATE and DELETE triggers are not activated: The FOR EACH ROW clause in the trigger definition does not apply because there are no rows.

Other possible uses for the BLACKHOLE storage engine include:

- Verification of dump file syntax.
- Measurement of the overhead from binary logging, by comparing performance using BLACKHOLE with and without binary logging enabled.
- BLACKHOLE is essentially a “no-op” storage engine, so it could be used for finding performance bottlenecks not related to the storage engine itself.

The BLACKHOLE engine is transaction-aware, in the sense that committed transactions are written to the binary log and rolled-back transactions are not.

Blackhole Engine and Auto Increment Columns

The BLACKHOLE engine is a no-op engine. Any operations performed on a table using BLACKHOLE have no effect. This should be borne in mind when considering the behavior of primary key columns that auto increment. The engine does not automatically increment field values, and does not retain auto increment field state. This has important implications in replication.

Consider the following replication scenario where all three of the following conditions apply:

1. On a source server there is a blackhole table with an auto increment field that is a primary key.
2. On a replica the same table exists but using the MyISAM engine.
3. Inserts are performed into the source's table without explicitly setting the auto increment value in the INSERT statement itself or through using a SET INSERT_ID statement.

In this scenario replication fails with a duplicate entry error on the primary key column.

In statement based replication, the value of INSERT_ID in the context event is always the same. Replication therefore fails due to trying insert a row with a duplicate value for a primary key column.

In row based replication, the value that the engine returns for the row always be the same for each insert. This results in the replica attempting to replay two insert log entries using the same value for the primary key column, and so replication fails.

Column Filtering

When using row-based replication, (binlog_format=ROW), a replica where the last columns are missing from a table is supported, as described in the section Section 17.5.1.9, “Replication with Differing Table Definitions on Source and Replica”.

This filtering works on the replica side, that is, the columns are copied to the replica before they are filtered out. There are at least two cases where it is not desirable to copy the columns to the replica:

1. If the data is confidential, so the replica server should not have access to it.
2. If the source has many replicas, filtering before sending to the replicas may reduce network traffic.

Source column filtering can be achieved using the BLACKHOLE engine. This is carried out in a way similar to how source table filtering is achieved - by using the BLACKHOLE engine and the --replicate-do-table or --replicate-ignore-table option.

The setup for the source is:

```
CREATE TABLE t1 (public_col_1, ..., public_col_N,
                 secret_col_1, ..., secret_col_M) ENGINE=MyISAM;
```

The setup for the trusted replica is:

```
CREATE TABLE t1 (public_col_1, ..., public_col_N) ENGINE=BLACKHOLE;
```

The setup for the untrusted replica is:

```
CREATE TABLE t1 (public_col_1, ..., public_col_N) ENGINE=MyISAM;
```

16.7 The MERGE Storage Engine

The MERGE storage engine, also known as the MRG_MyISAM engine, is a collection of identical MyISAM tables that can be used as one. “Identical” means that all tables have identical column data types and index information. You cannot merge MyISAM tables in which the columns are listed in a different order, do not have exactly the same data types in corresponding columns, or have the indexes in different order. However, any or all of the MyISAM tables can be compressed with myisampack. See

Section 4.6.6, “myisampack — Generate Compressed, Read-Only MyISAM Tables”. Differences between tables such as these do not matter:

- Names of corresponding columns and indexes can differ.
- Comments for tables, columns, and indexes can differ.
- Table options such as `AVG_ROW_LENGTH`, `MAX_ROWS`, or `PACK_KEYS` can differ.

An alternative to a `MERGE` table is a partitioned table, which stores partitions of a single table in separate files and enables some operations to be performed more efficiently. For more information, see Chapter 24, *Partitioning*.

When you create a `MERGE` table, MySQL creates a `.MRG` file on disk that contains the names of the underlying `MyISAM` tables that should be used as one. The table format of the `MERGE` table is stored in the MySQL data dictionary. The underlying tables do not have to be in the same database as the `MERGE` table.

You can use `SELECT`, `DELETE`, `UPDATE`, and `INSERT` on `MERGE` tables. You must have `SELECT`, `DELETE`, and `UPDATE` privileges on the `MyISAM` tables that you map to a `MERGE` table.

Note

The use of `MERGE` tables entails the following security issue: If a user has access to `MyISAM` table *`t`*, that user can create a `MERGE` table *`m`* that accesses *`t`*. However, if the user's privileges on *`t`* are subsequently revoked, the user can continue to access *`t`* by doing so through *`m`*.

Use of `DROP TABLE` with a `MERGE` table drops only the `MERGE` specification. The underlying tables are not affected.

To create a `MERGE` table, you must specify a `UNION=(`*`list-of-tables`*`)` option that indicates which `MyISAM` tables to use. You can optionally specify an `INSERT_METHOD` option to control how inserts into the `MERGE` table take place. Use a value of `FIRST` or `LAST` to cause inserts to be made in the first or last underlying table, respectively. If you specify no `INSERT_METHOD` option or if you specify it with a value of `NO`, inserts into the `MERGE` table are not permitted and attempts to do so result in an error.

The following example shows how to create a `MERGE` table:

```
mysql> CREATE TABLE t1 (
    ->    a INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->    message CHAR(20)) ENGINE=MyISAM;
mysql> CREATE TABLE t2 (
    ->    a INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
    ->    message CHAR(20)) ENGINE=MyISAM;
mysql> INSERT INTO t1 (message) VALUES ('Testing'),('table'),('t1');
mysql> INSERT INTO t2 (message) VALUES ('Testing'),('table'),('t2');
mysql> CREATE TABLE total (
    ->    a INT NOT NULL AUTO_INCREMENT,
    ->    message CHAR(20), INDEX(a))
    ->    ENGINE=MERGE UNION=(t1,t2) INSERT_METHOD=LAST;
```

Column `a` is indexed as a `PRIMARY KEY` in the underlying `MyISAM` tables, but not in the `MERGE` table. There it is indexed but not as a `PRIMARY KEY` because a `MERGE` table cannot enforce uniqueness over the set of underlying tables. (Similarly, a column with a `UNIQUE` index in the underlying tables should be indexed in the `MERGE` table but not as a `UNIQUE` index.)

After creating the `MERGE` table, you can use it to issue queries that operate on the group of tables as a whole:

```
mysql> SELECT * FROM total;
+---+---------+
| a | message |
+---+---------+
| 1 | Testing |
```

```
| 2 | table   |
| 3 | t1      |
| 1 | Testing |
| 2 | table   |
| 3 | t2      |
+---+---------+
```

To remap a `MERGE` table to a different collection of `MyISAM` tables, you can use one of the following methods:

- `DROP` the `MERGE` table and re-create it.
- Use `ALTER TABLE tbl_name UNION=(...)` to change the list of underlying tables.

  It is also possible to use `ALTER TABLE ... UNION=()` (that is, with an empty `UNION` clause) to remove all of the underlying tables. However, in this case, the table is effectively empty and inserts fail because there is no underlying table to take new rows. Such a table might be useful as a template for creating new `MERGE` tables with `CREATE TABLE ... LIKE`.

The underlying table definitions and indexes must conform closely to the definition of the `MERGE` table. Conformance is checked when a table that is part of a `MERGE` table is opened, not when the `MERGE` table is created. If any table fails the conformance checks, the operation that triggered the opening of the table fails. This means that changes to the definitions of tables within a `MERGE` may cause a failure when the `MERGE` table is accessed. The conformance checks applied to each table are:

- The underlying table and the `MERGE` table must have the same number of columns.
- The column order in the underlying table and the `MERGE` table must match.
- Additionally, the specification for each corresponding column in the parent `MERGE` table and the underlying tables are compared and must satisfy these checks:
  - The column type in the underlying table and the `MERGE` table must be equal.
  - The column length in the underlying table and the `MERGE` table must be equal.
  - The column of the underlying table and the `MERGE` table can be `NULL`.
- The underlying table must have at least as many indexes as the `MERGE` table. The underlying table may have more indexes than the `MERGE` table, but cannot have fewer.

  Note

  A known issue exists where indexes on the same columns must be in identical order, in both the `MERGE` table and the underlying `MyISAM` table. See Bug #33653.

  Each index must satisfy these checks:

  - The index type of the underlying table and the `MERGE` table must be the same.
  - The number of index parts (that is, multiple columns within a compound index) in the index definition for the underlying table and the `MERGE` table must be the same.
  - For each index part:
    - Index part lengths must be equal.
    - Index part types must be equal.
    - Index part languages must be equal.
    - Check whether index parts can be `NULL`.

If a MERGE table cannot be opened or used because of a problem with an underlying table, CHECK TABLE displays information about which table caused the problem.

Additional Resources

- A forum dedicated to the MERGE storage engine is available at https://forums.mysql.com/list.php?93.

16.7.1 MERGE Table Advantages and Disadvantages

MERGE tables can help you solve the following problems:

- Easily manage a set of log tables. For example, you can put data from different months into separate tables, compress some of them with myisampack, and then create a MERGE table to use them as one.
- Obtain more speed. You can split a large read-only table based on some criteria, and then put individual tables on different disks. A MERGE table structured this way could be much faster than using a single large table.
- Perform more efficient searches. If you know exactly what you are looking for, you can search in just one of the underlying tables for some queries and use a MERGE table for others. You can even have many different MERGE tables that use overlapping sets of tables.
- Perform more efficient repairs. It is easier to repair individual smaller tables that are mapped to a MERGE table than to repair a single large table.
- Instantly map many tables as one. A MERGE table need not maintain an index of its own because it uses the indexes of the individual tables. As a result, MERGE table collections are *very* fast to create or remap. (You must still specify the index definitions when you create a MERGE table, even though no indexes are created.)
- If you have a set of tables from which you create a large table on demand, you can instead create a MERGE table from them on demand. This is much faster and saves a lot of disk space.
- Exceed the file size limit for the operating system. Each MyISAM table is bound by this limit, but a collection of MyISAM tables is not.
- You can create an alias or synonym for a MyISAM table by defining a MERGE table that maps to that single table. There should be no really notable performance impact from doing this (only a couple of indirect calls and memcpy() calls for each read).

The disadvantages of MERGE tables are:

- You can use only identical MyISAM tables for a MERGE table.
- Some MyISAM features are unavailable in MERGE tables. For example, you cannot create FULLTEXT indexes on MERGE tables. (You can create FULLTEXT indexes on the underlying MyISAM tables, but you cannot search the MERGE table with a full-text search.)
- If the MERGE table is nontemporary, all underlying MyISAM tables must be nontemporary. If the MERGE table is temporary, the MyISAM tables can be any mix of temporary and nontemporary.
- MERGE tables use more file descriptors than MyISAM tables. If 10 clients are using a MERGE table that maps to 10 tables, the server uses (10 × 10) + 10 file descriptors. (10 data file descriptors for each of the 10 clients, and 10 index file descriptors shared among the clients.)
- Index reads are slower. When you read an index, the MERGE storage engine needs to issue a read on all underlying tables to check which one most closely matches a given index value. To read the next index value, the MERGE storage engine needs to search the read buffers to find the next value. Only when one index buffer is used up does the storage engine need to read the next index block. This makes MERGE indexes much slower on eq_ref searches, but not much slower on ref searches. For more information about eq_ref and ref, see Section 13.8.2, “EXPLAIN Statement”.

16.7.2 MERGE Table Problems

The following are known problems with `MERGE` tables:

- In versions of MySQL Server prior to 5.1.23, it was possible to create temporary merge tables with nontemporary child MyISAM tables.

  From versions 5.1.23, MERGE children were locked through the parent table. If the parent was temporary, it was not locked and so the children were not locked either. Parallel use of the MyISAM tables corrupted them.

- If you use `ALTER TABLE` to change a `MERGE` table to another storage engine, the mapping to the underlying tables is lost. Instead, the rows from the underlying `MyISAM` tables are copied into the altered table, which then uses the specified storage engine.

- The `INSERT_METHOD` table option for a `MERGE` table indicates which underlying `MyISAM` table to use for inserts into the `MERGE` table. However, use of the `AUTO_INCREMENT` table option for that `MyISAM` table has no effect for inserts into the `MERGE` table until at least one row has been inserted directly into the `MyISAM` table.

- A `MERGE` table cannot maintain uniqueness constraints over the entire table. When you perform an `INSERT`, the data goes into the first or last `MyISAM` table (as determined by the `INSERT_METHOD` option). MySQL ensures that unique key values remain unique within that `MyISAM` table, but not over all the underlying tables in the collection.

- Because the `MERGE` engine cannot enforce uniqueness over the set of underlying tables, `REPLACE` does not work as expected. The two key facts are:

  - `REPLACE` can detect unique key violations only in the underlying table to which it is going to write (which is determined by the `INSERT_METHOD` option). This differs from violations in the `MERGE` table itself.

  - If `REPLACE` detects a unique key violation, it changes only the corresponding row in the underlying table it is writing to; that is, the first or last table, as determined by the `INSERT_METHOD` option.

  Similar considerations apply for `INSERT ... ON DUPLICATE KEY UPDATE`.

- `MERGE` tables do not support partitioning. That is, you cannot partition a `MERGE` table, nor can any of a `MERGE` table's underlying `MyISAM` tables be partitioned.

- You should not use `ANALYZE TABLE`, `REPAIR TABLE`, `OPTIMIZE TABLE`, `ALTER TABLE`, `DROP TABLE`, `DELETE` without a `WHERE` clause, or `TRUNCATE TABLE` on any of the tables that are mapped into an open `MERGE` table. If you do so, the `MERGE` table may still refer to the original table and yield unexpected results. To work around this problem, ensure that no `MERGE` tables remain open by issuing a `FLUSH TABLES` statement prior to performing any of the named operations.

  The unexpected results include the possibility that the operation on the `MERGE` table reports table corruption. If this occurs after one of the named operations on the underlying `MyISAM` tables, the corruption message is spurious. To deal with this, issue a `FLUSH TABLES` statement after modifying the `MyISAM` tables.

- `DROP TABLE` on a table that is in use by a `MERGE` table does not work on Windows because the `MERGE` storage engine's table mapping is hidden from the upper layer of MySQL. Windows does not permit open files to be deleted, so you first must flush all `MERGE` tables (with `FLUSH TABLES`) or drop the `MERGE` table before dropping the table.

- The definition of the `MyISAM` tables and the `MERGE` table are checked when the tables are accessed (for example, as part of a `SELECT` or `INSERT` statement). The checks ensure that the definitions of the tables and the parent `MERGE` table definition match by comparing column order, types, sizes and associated indexes. If there is a difference between the tables, an error is returned and the statement fails. Because these checks take place when the tables are opened, any changes to the

definition of a single table, including column changes, column ordering, and engine alterations cause the statement to fail.

- The order of indexes in the `MERGE` table and its underlying tables should be the same. If you use `ALTER TABLE` to add a `UNIQUE` index to a table used in a `MERGE` table, and then use `ALTER TABLE` to add a nonunique index on the `MERGE` table, the index ordering is different for the tables if there was already a nonunique index in the underlying table. (This happens because `ALTER TABLE` puts `UNIQUE` indexes before nonunique indexes to facilitate rapid detection of duplicate keys.) Consequently, queries on tables with such indexes may return unexpected results.

- If you encounter an error message similar to `ERROR 1017 (HY000): Can't find file: 'tbl_name.MRG' (errno: 2)`, it generally indicates that some of the underlying tables do not use the `MyISAM` storage engine. Confirm that all of these tables are `MyISAM`.

- The maximum number of rows in a `MERGE` table is 2^{64} (~1.844E+19; the same as for a `MyISAM` table). It is not possible to merge multiple `MyISAM` tables into a single `MERGE` table that would have more than this number of rows.

- Use of underlying `MyISAM` tables of differing row formats with a parent `MERGE` table is currently known to fail. See Bug #32364.

- You cannot change the union list of a nontemporary `MERGE` table when `LOCK TABLES` is in effect. The following does *not* work:

```
CREATE TABLE m1 ... ENGINE=MRG_MYISAM ...;
LOCK TABLES t1 WRITE, t2 WRITE, m1 WRITE;
ALTER TABLE m1 ... UNION=(t1,t2) ...;
```

  However, you can do this with a temporary `MERGE` table.

- You cannot create a `MERGE` table with `CREATE ... SELECT`, neither as a temporary `MERGE` table, nor as a nontemporary `MERGE` table. For example:

```
CREATE TABLE m1 ... ENGINE=MRG_MYISAM ... SELECT ...;
```

  Attempts to do this result in an error: *`tbl_name`* is not `BASE TABLE`.

- In some cases, differing `PACK_KEYS` table option values among the `MERGE` and underlying tables cause unexpected results if the underlying tables contain `CHAR` or `BINARY` columns. As a workaround, use `ALTER TABLE` to ensure that all involved tables have the same `PACK_KEYS` value. (Bug #50646)

16.8 The FEDERATED Storage Engine

The `FEDERATED` storage engine lets you access data from a remote MySQL database without using replication or cluster technology. Querying a local `FEDERATED` table automatically pulls the data from the remote (federated) tables. No data is stored on the local tables.

To include the `FEDERATED` storage engine if you build MySQL from source, invoke `CMake` with the `-DWITH_FEDERATED_STORAGE_ENGINE` option.

The `FEDERATED` storage engine is not enabled by default in the running server; to enable `FEDERATED`, you must start the MySQL server binary using the `--federated` option.

To examine the source for the `FEDERATED` engine, look in the `storage/federated` directory of a MySQL source distribution.

16.8.1 FEDERATED Storage Engine Overview

When you create a table using one of the standard storage engines (such as `MyISAM`, `CSV` or `InnoDB`), the table consists of the table definition and the associated data. When you create a

FEDERATED table, the table definition is the same, but the physical storage of the data is handled on a remote server.

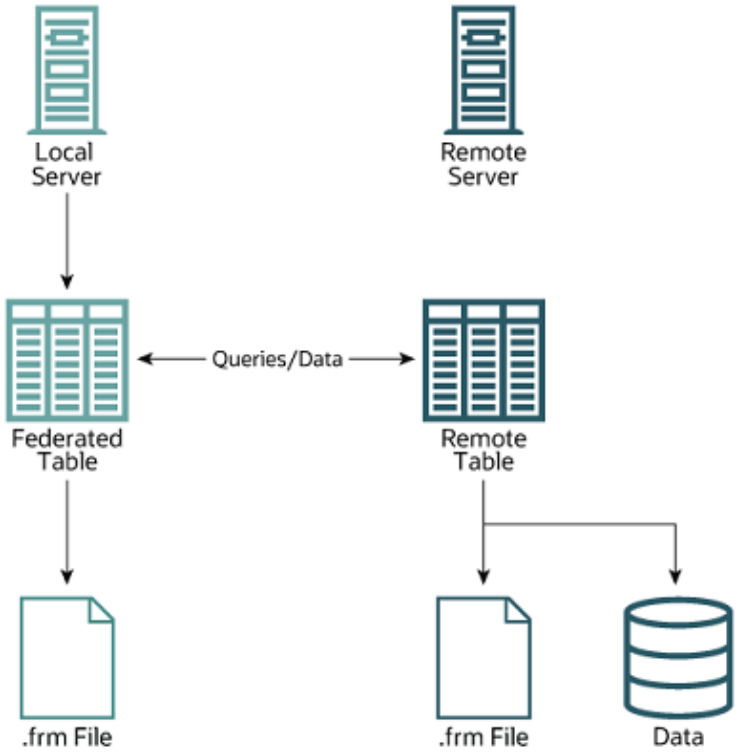
A FEDERATED table consists of two elements:

- A *remote server* with a database table, which in turn consists of the table definition (stored in the MySQL data dictionary) and the associated table. The table type of the remote table may be any type supported by the remote mysqld server, including MyISAM or InnoDB.
- A *local server* with a database table, where the table definition matches that of the corresponding table on the remote server. The table definition is stored in the data dictionary. There is no data file on the local server. Instead, the table definition includes a connection string that points to the remote table.

When executing queries and statements on a FEDERATED table on the local server, the operations that would normally insert, update or delete information from a local data file are instead sent to the remote server for execution, where they update the data file on the remote server or return matching rows from the remote server.

The basic structure of a FEDERATED table setup is shown in Figure 16.2, “FEDERATED Table Structure”.

Figure 16.2 FEDERATED Table Structure



When a client issues an SQL statement that refers to a FEDERATED table, the flow of information between the local server (where the SQL statement is executed) and the remote server (where the data is physically stored) is as follows:

1. The storage engine looks through each column that the FEDERATED table has and constructs an appropriate SQL statement that refers to the remote table.
2. The statement is sent to the remote server using the MySQL client API.
3. The remote server processes the statement and the local server retrieves any result that the statement produces (an affected-rows count or a result set).

4. If the statement produces a result set, each column is converted to internal storage engine format that the FEDERATED engine expects and can use to display the result to the client that issued the original statement.

The local server communicates with the remote server using MySQL client C API functions. It invokes mysql_real_query() to send the statement. To read a result set, it uses mysql_store_result() and fetches rows one at a time using mysql_fetch_row().

16.8.2 How to Create FEDERATED Tables

To create a FEDERATED table you should follow these steps:

1. Create the table on the remote server. Alternatively, make a note of the table definition of an existing table, perhaps using the SHOW CREATE TABLE statement.
2. Create the table on the local server with an identical table definition, but adding the connection information that links the local table to the remote table.

For example, you could create the following table on the remote server:

```
CREATE TABLE test_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=MyISAM
DEFAULT CHARSET=utf8mb4;
```

To create the local table that is federated to the remote table, there are two options available. You can either create the local table and specify the connection string (containing the server name, login, password) to be used to connect to the remote table using the CONNECTION, or you can use an existing connection that you have previously created using the CREATE SERVER statement.

Important

When you create the local table it *must* have an identical field definition to the remote table.

Note

You can improve the performance of a FEDERATED table by adding indexes to the table on the host. The optimization occurs because the query sent to the remote server includes the contents of the WHERE clause and is sent to the remote server and subsequently executed locally. This reduces the network traffic that would otherwise request the entire table from the server for local processing.

16.8.2.1 Creating a FEDERATED Table Using CONNECTION

To use the first method, you must specify the CONNECTION string after the engine type in a CREATE TABLE statement. For example:

```
CREATE TABLE federated_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=FEDERATED
```

```
DEFAULT CHARSET=utf8mb4
CONNECTION='mysql://fed_user@remote_host:9306/federated/test_table';
```

Note

`CONNECTION` replaces the `COMMENT` used in some previous versions of MySQL.

The `CONNECTION` string contains the information required to connect to the remote server containing the table in which the data physically resides. The connection string specifies the server name, login credentials, port number and database/table information. In the example, the remote table is on the server `remote_host`, using port 9306. The name and port number should match the host name (or IP address) and port number of the remote MySQL server instance you want to use as your remote table.

The format of the connection string is as follows:

```
scheme://user_name[:password]@host_name[:port_num]/db_name/tbl_name
```

Where:

- *`scheme`*: A recognized connection protocol. Only `mysql` is supported as the *`scheme`* value at this point.
- *`user_name`*: The user name for the connection. This user must have been created on the remote server, and must have suitable privileges to perform the required actions (`SELECT`, `INSERT`, `UPDATE`, and so forth) on the remote table.
- *`password`*: (Optional) The corresponding password for *`user_name`*.
- *`host_name`*: The host name or IP address of the remote server.
- *`port_num`*: (Optional) The port number for the remote server. The default is 3306.
- *`db_name`*: The name of the database holding the remote table.
- *`tbl_name`*: The name of the remote table. The name of the local and the remote table do not have to match.

Sample connection strings:

```
CONNECTION='mysql://username:password@hostname:port/database/tablename'
CONNECTION='mysql://username@hostname/database/tablename'
CONNECTION='mysql://username:password@hostname/database/tablename'
```

16.8.2.2 Creating a FEDERATED Table Using CREATE SERVER

If you are creating a number of `FEDERATED` tables on the same server, or if you want to simplify the process of creating `FEDERATED` tables, you can use the `CREATE SERVER` statement to define the server connection parameters, just as you would with the `CONNECTION` string.

The format of the `CREATE SERVER` statement is:

```
CREATE SERVER
server_name
FOREIGN DATA WRAPPER wrapper_name
OPTIONS (option [, option] ...)
```

The *`server_name`* is used in the connection string when creating a new `FEDERATED` table.

For example, to create a server connection identical to the `CONNECTION` string:

```
CONNECTION='mysql://fed_user@remote_host:9306/federated/test_table';
```

You would use the following statement:

```
CREATE SERVER fedlink
FOREIGN DATA WRAPPER mysql
OPTIONS (USER 'fed_user', HOST 'remote_host', PORT 9306, DATABASE 'federated');
```

To create a `FEDERATED` table that uses this connection, you still use the `CONNECTION` keyword, but specify the name you used in the `CREATE SERVER` statement.

```
CREATE TABLE test_table (
    id     INT(20) NOT NULL AUTO_INCREMENT,
    name   VARCHAR(32) NOT NULL DEFAULT '',
    other  INT(20) NOT NULL DEFAULT '0',
    PRIMARY KEY  (id),
    INDEX name (name),
    INDEX other_key (other)
)
ENGINE=FEDERATED
DEFAULT CHARSET=utf8mb4
CONNECTION='fedlink/test_table';
```

The connection name in this example contains the name of the connection (`fedlink`) and the name of the table (`test_table`) to link to, separated by a slash. If you specify only the connection name without a table name, the table name of the local table is used instead.

For more information on `CREATE SERVER`, see Section 13.1.18, “CREATE SERVER Statement”.

The `CREATE SERVER` statement accepts the same arguments as the `CONNECTION` string. The `CREATE SERVER` statement updates the rows in the `mysql.servers` table. See the following table for information on the correspondence between parameters in a connection string, options in the `CREATE SERVER` statement, and the columns in the `mysql.servers` table. For reference, the format of the `CONNECTION` string is as follows:

```
scheme://user_name[:password]@host_name[:port_num]/db_name/tbl_name
```

Description	`CONNECTION` string	`CREATE SERVER` option	`mysql.servers` column
Connection scheme	*`scheme`*	`wrapper_name`	`Wrapper`
Remote user	*`user_name`*	`USER`	`Username`
Remote password	*`password`*	`PASSWORD`	`Password`
Remote host	*`host_name`*	`HOST`	`Host`
Remote port	*`port_num`*	`PORT`	`Port`
Remote database	*`db_name`*	`DATABASE`	`Db`

16.8.3 FEDERATED Storage Engine Notes and Tips

You should be aware of the following points when using the `FEDERATED` storage engine:

- `FEDERATED` tables may be replicated to other replicas, but you must ensure that the replica servers are able to use the user/password combination that is defined in the `CONNECTION` string (or the row in the `mysql.servers` table) to connect to the remote server.

The following items indicate features that the `FEDERATED` storage engine does and does not support:

- The remote server must be a MySQL server.
- The remote table that a `FEDERATED` table points to *must* exist before you try to access the table through the `FEDERATED` table.
- It is possible for one `FEDERATED` table to point to another, but you must be careful not to create a loop.

- A `FEDERATED` table does not support indexes in the usual sense; because access to the table data is handled remotely, it is actually the remote table that makes use of indexes. This means that, for a query that cannot use any indexes and so requires a full table scan, the server fetches all rows from the remote table and filters them locally. This occurs regardless of any `WHERE` or `LIMIT` used with this `SELECT` statement; these clauses are applied locally to the returned rows.

  Queries that fail to use indexes can thus cause poor performance and network overload. In addition, since returned rows must be stored in memory, such a query can also lead to the local server swapping, or even hanging.

- Care should be taken when creating a `FEDERATED` table since the index definition from an equivalent `MyISAM` or other table may not be supported. For example, creating a `FEDERATED` table fails if the table uses an index prefix on any `VARCHAR`, `TEXT` or `BLOB` columns. The following definition using `MyISAM` is valid:

  ```
  CREATE TABLE `T1`(`A` VARCHAR(100),UNIQUE KEY(`A`(30))) ENGINE=MYISAM;
  ```

  The key prefix in this example is incompatible with the `FEDERATED` engine, and the equivalent statement fails:

  ```
  CREATE TABLE `T1`(`A` VARCHAR(100),UNIQUE KEY(`A`(30))) ENGINE=FEDERATED
    CONNECTION='MYSQL://127.0.0.1:3306/TEST/T1';
  ```

  If possible, you should try to separate the column and index definition when creating tables on both the remote server and the local server to avoid these index issues.

- Internally, the implementation uses `SELECT`, `INSERT`, `UPDATE`, and `DELETE`, but not `HANDLER`.

- The `FEDERATED` storage engine supports `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `TRUNCATE TABLE`, and indexes. It does not support `ALTER TABLE`, or any Data Definition Language statements that directly affect the structure of the table, other than `DROP TABLE`. The current implementation does not use prepared statements.

- `FEDERATED` accepts `INSERT ... ON DUPLICATE KEY UPDATE` statements, but if a duplicate-key violation occurs, the statement fails with an error.

- Transactions are not supported.

- `FEDERATED` performs bulk-insert handling such that multiple rows are sent to the remote table in a batch, which improves performance. Also, if the remote table is transactional, it enables the remote storage engine to perform statement rollback properly should an error occur. This capability has the following limitations:

  - The size of the insert cannot exceed the maximum packet size between servers. If the insert exceeds this size, it is broken into multiple packets and the rollback problem can occur.

  - Bulk-insert handling does not occur for `INSERT ... ON DUPLICATE KEY UPDATE`.

- There is no way for the `FEDERATED` engine to know if the remote table has changed. The reason for this is that this table must work like a data file that would never be written to by anything other than the database system. The integrity of the data in the local table could be breached if there was any change to the remote database.

- When using a `CONNECTION` string, you cannot use an '@' character in the password. You can get round this limitation by using the `CREATE SERVER` statement to create a server connection.

- The `insert_id` and `timestamp` options are not propagated to the data provider.

- Any `DROP TABLE` statement issued against a `FEDERATED` table drops only the local table, not the remote table.

- User-defined partitioning is not supported for `FEDERATED` tables.

16.8.4 FEDERATED Storage Engine Resources

The following additional resources are available for the FEDERATED storage engine:

- A forum dedicated to the FEDERATED storage engine is available at https://forums.mysql.com/list.php?105.

16.9 The EXAMPLE Storage Engine

The EXAMPLE storage engine is a stub engine that does nothing. Its purpose is to serve as an example in the MySQL source code that illustrates how to begin writing new storage engines. As such, it is primarily of interest to developers.

To enable the EXAMPLE storage engine if you build MySQL from source, invoke CMake with the -DWITH_EXAMPLE_STORAGE_ENGINE option.

To examine the source for the EXAMPLE engine, look in the storage/example directory of a MySQL source distribution.

When you create an EXAMPLE table, no files are created. No data can be stored into the table. Retrievals return an empty result.

```
mysql> CREATE TABLE test (i INT) ENGINE = EXAMPLE;
Query OK, 0 rows affected (0.78 sec)

mysql> INSERT INTO test VALUES(1),(2),(3);
ERROR 1031 (HY000): Table storage engine for 'test' doesn't »
                    have this option

mysql> SELECT * FROM test;
Empty set (0.31 sec)
```

The EXAMPLE storage engine does not support indexing.

The EXAMPLE storage engine does not support partitioning.

16.10 Other Storage Engines

Other storage engines may be available from third parties and community members that have used the Custom Storage Engine interface.

Third party engines are not supported by MySQL. For further information, documentation, installation guides, bug reporting or for any help or assistance with these engines, please contact the developer of the engine directly.

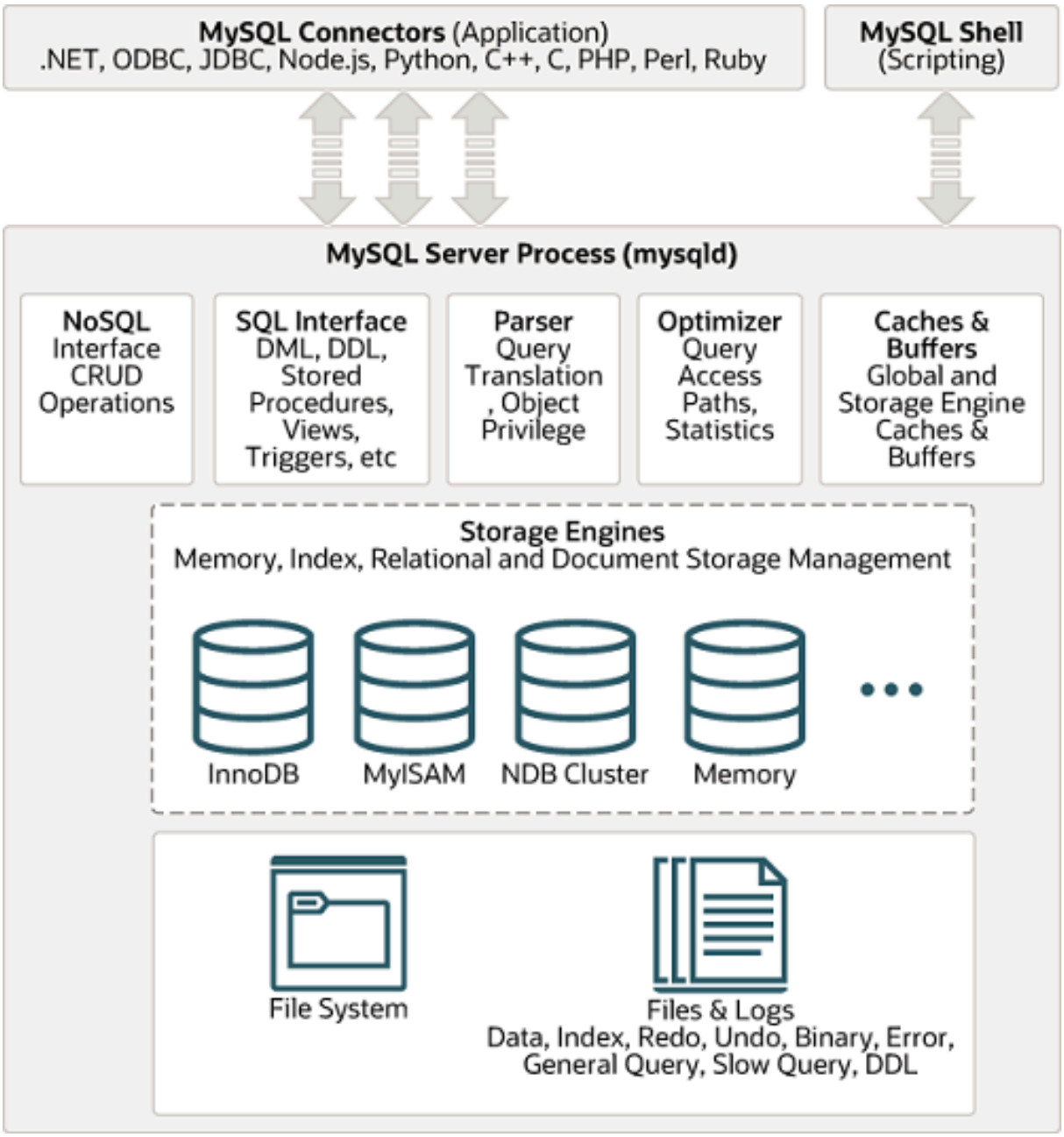
For more information on developing a customer storage engine that can be used with the Pluggable Storage Engine Architecture, see MySQL Internals: Writing a Custom Storage Engine.

16.11 Overview of MySQL Storage Engine Architecture

The MySQL pluggable storage engine architecture enables a database professional to select a specialized storage engine for a particular application need while being completely shielded from the need to manage any specific application coding requirements. The MySQL server architecture isolates the application programmer and DBA from all of the low-level implementation details at the storage level, providing a consistent and easy application model and API. Thus, although there are different capabilities across different storage engines, the application is shielded from these differences.

The MySQL pluggable storage engine architecture is shown in Figure 16.3, “MySQL Architecture with Pluggable Storage Engines”.

Figure 16.3 MySQL Architecture with Pluggable Storage Engines



The pluggable storage engine architecture provides a standard set of management and support services that are common among all underlying storage engines. The storage engines themselves are the components of the database server that actually perform actions on the underlying data that is maintained at the physical server level.

This efficient and modular architecture provides huge benefits for those wishing to specifically target a particular application need—such as data warehousing, transaction processing, or high availability situations—while enjoying the advantage of utilizing a set of interfaces and services that are independent of any one storage engine.

The application programmer and DBA interact with the MySQL database through Connector APIs and service layers that are above the storage engines. If application changes bring about requirements that demand the underlying storage engine change, or that one or more storage engines be added to support new needs, no significant coding or process changes are required to make things work. The MySQL server architecture shields the application from the underlying complexity of the storage engine by presenting a consistent and easy-to-use API that applies across storage engines.

16.11.1 Pluggable Storage Engine Architecture

MySQL Server uses a pluggable storage engine architecture that enables storage engines to be loaded into and unloaded from a running MySQL server.

Plugging in a Storage Engine

Before a storage engine can be used, the storage engine plugin shared library must be loaded into MySQL using the `INSTALL PLUGIN` statement. For example, if the `EXAMPLE` engine plugin is named `example` and the shared library is named `ha_example.so`, you load it with the following statement:

```
INSTALL PLUGIN example SONAME 'ha_example.so';
```

To install a pluggable storage engine, the plugin file must be located in the MySQL plugin directory, and the user issuing the `INSTALL PLUGIN` statement must have `INSERT` privilege for the `mysql.plugin` table.

The shared library must be located in the MySQL server plugin directory, the location of which is given by the `plugin_dir` system variable.

Unplugging a Storage Engine

To unplug a storage engine, use the `UNINSTALL PLUGIN` statement:

```
UNINSTALL PLUGIN example;
```

If you unplug a storage engine that is needed by existing tables, those tables become inaccessible, but are still present on disk (where applicable). Ensure that there are no tables using a storage engine before you unplug the storage engine.

16.11.2 The Common Database Server Layer

A MySQL pluggable storage engine is the component in the MySQL database server that is responsible for performing the actual data I/O operations for a database as well as enabling and enforcing certain feature sets that target a specific application need. A major benefit of using specific storage engines is that you are only delivered the features needed for a particular application, and therefore you have less system overhead in the database, with the end result being more efficient and higher database performance. This is one of the reasons that MySQL has always been known to have such high performance, matching or beating proprietary monolithic databases in industry standard benchmarks.

From a technical perspective, what are some of the unique supporting infrastructure components that are in a storage engine? Some of the key feature differentiations include:

- *Concurrency*: Some applications have more granular lock requirements (such as row-level locks) than others. Choosing the right locking strategy can reduce overhead and therefore improve overall performance. This area also includes support for capabilities such as multi-version concurrency control or “snapshot” read.
- *Transaction Support*: Not every application needs transactions, but for those that do, there are very well defined requirements such as ACID compliance and more.
- *Referential Integrity*: The need to have the server enforce relational database referential integrity through DDL defined foreign keys.
- *Physical Storage*: This involves everything from the overall page size for tables and indexes as well as the format used for storing data to physical disk.
- *Index Support*: Different application scenarios tend to benefit from different index strategies. Each storage engine generally has its own indexing methods, although some (such as B-tree indexes) are common to nearly all engines.

- *Memory Caches*: Different applications respond better to some memory caching strategies than others, so although some memory caches are common to all storage engines (such as those used for user connections), others are uniquely defined only when a particular storage engine is put in play.
- *Performance Aids*: This includes multiple I/O threads for parallel operations, thread concurrency, database checkpointing, bulk insert handling, and more.
- *Miscellaneous Target Features*: This may include support for geospatial operations, security restrictions for certain data manipulation operations, and other similar features.

Each set of the pluggable storage engine infrastructure components are designed to offer a selective set of benefits for a particular application. Conversely, avoiding a set of component features helps reduce unnecessary overhead. It stands to reason that understanding a particular application's set of requirements and selecting the proper MySQL storage engine can have a dramatic impact on overall system efficiency and performance.

Chapter 17 Replication

Table of Contents

Replication enables data from one MySQL database server (known as a source) to be copied to one or more MySQL database servers (known as replicas). Replication is asynchronous by default; replicas do not need to be connected permanently to receive updates from a source. 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 replicas to improve performance. In this environment, all writes and updates must take place on the source server. Reads, however, may take place on one or more replicas. This model can improve the performance of writes (since the source is dedicated to updates), while dramatically increasing read speed across an increasing number of replicas.

- Data security - because the replica can pause the replication process, it is possible to run backup services on the replica without corrupting the corresponding source data.
- Analytics - live data can be created on the source, while the analysis of the information can take place on the replica without affecting the performance of the source.
- 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 source.

For information on how to use replication in such scenarios, see Section 17.4, “Replication Solutions”.

MySQL 8.0 supports different methods of replication. The traditional method is based on replicating events from the source's binary log, and requires the log files and positions in them to be synchronized between source and replica. 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 source and replica as long as all transactions committed on the source have also been applied on the replica. For more information about GTIDs and GTID-based replication in MySQL, see Section 17.1.3, “Replication with Global Transaction Identifiers”. For information on using binary log file position based replication, see Section 17.1, “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 source, while one or more other servers act as replicas. This is in contrast to the *synchronous* replication which is a characteristic of NDB Cluster (see Chapter 23, *MySQL NDB Cluster 8.0*). In MySQL 8.0, semisynchronous replication is supported in addition to the built-in asynchronous replication. With semisynchronous replication, a commit performed on the source blocks before returning to the session that performed the transaction until at least one replica acknowledges that it has received and logged the events for the transaction; see Section 17.4.10, “Semisynchronous Replication”. MySQL 8.0 also supports delayed replication such that a replica deliberately lags behind the source by at least a specified amount of time; see Section 17.4.11, “Delayed Replication”. For scenarios where *synchronous* replication is required, use NDB Cluster (see Chapter 23, *MySQL NDB Cluster 8.0*).

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 17.1.2, “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 17.2.1, “Replication Formats”.

Replication is controlled through a number of different options and variables. For more information, see Section 17.1.6, “Replication and Binary Logging Options and Variables”. Additional security measures can be applied to a replication topology, as described in Section 17.3, “Replication Security”.

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 Section 17.4, “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 Section 17.5, “Replication Notes and Tips”. For answers to some questions often asked by those who are new to MySQL Replication, see Section A.14, “MySQL 8.0 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 Section 17.2, “Replication Implementation”.

17.1 Configuring Replication

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 17.1.2, “Setting Up Binary Log File Position Based Replication”, deals with the configuration of the servers and provides methods for copying data between the source and replicas.
- For a guide to setting up two or more servers for replication using GTID transactions, Section 17.1.3, “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 17.2.1, “Replication Formats”.
- Detailed information on the different configuration options and variables that apply to replication is provided in Section 17.1.6, “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 17.1.7, “Common Replication Administration Tasks”.

17.1.1 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 source (where the database changes take place) 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. Replicas are configured to read the binary log from the source and to execute the events in the binary log on the replica's local database.

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

Important

You cannot configure the source to log only certain events.

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

The source and each replica must be configured with a unique ID (using the `server_id` system variable). In addition, each replica must be configured with information about the source's host name, log file name, and position within that file. These details can be controlled from within a MySQL session using a `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) on the replica. The details are stored within the replica's connection metadata repository (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”).

17.1.2 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 in the database on the source that you want to replicate.

Tip

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

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

- On the source, you must ensure that binary logging is enabled, and configure a unique server ID. This might require a server restart. See Section 17.1.2.1, “Setting the Replication Source Configuration”.
- On each replica that you want to connect to the source, you must configure a unique server ID. This might require a server restart. See Section 17.1.2.2, “Setting the Replica Configuration”.
- Optionally, create a separate user for your replicas to use during authentication with the source when reading the binary log for replication. See Section 17.1.2.3, “Creating a User for Replication”.
- Before creating a data snapshot or starting the replication process, on the source you should record the current position in the binary log. You need this information when configuring the replica so that the replica knows where within the binary log to start executing events. See Section 17.1.2.4, “Obtaining the Replication Source Binary Log Coordinates”.
- If you already have data on the source and want to use it to synchronize the replica, you need to create a data snapshot to copy the data to the replica. 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 source to obtain a read-lock, then obtain its current binary log coordinates and dump its data, before permitting the source to continue executing statements. If you do not stop the execution of statements, the data dump and the source status information become mismatched, resulting in inconsistent or corrupted databases on the replicas. For more information on replicating a `MyISAM` source, see Section 17.1.2.4, “Obtaining the Replication Source 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 Section 15.19, “InnoDB and MySQL Replication”.
- Configure the replica with settings for connecting to the source, such as the host name, login credentials, and binary log file name and position. See Section 17.1.2.7, “Setting the Source Configuration on the Replica”.
- Implement replication-specific security measures on the sources and replicas as appropriate for your system. See Section 17.3, “Replication Security”.

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 source and replicas that contain no data, see Setting Up Replication with New Source and Replicas.
- To set up replication of a new source using the data from an existing MySQL server, see Setting Up Replication with Existing Data.
- To add replicas to an existing replication environment, see Section 17.1.2.8, “Adding Replicas to a Replication Environment”.

Before administering MySQL replication servers, read this entire chapter and try all statements mentioned in Section 13.4.1, “SQL Statements for Controlling Source Servers”, and Section 13.4.2, “SQL Statements for Controlling Replica Servers”. Also familiarize yourself with the replication startup options described in Section 17.1.6, “Replication and Binary Logging Options and Variables”.

17.1.2.1 Setting the Replication Source Configuration

To configure a source to use binary log file position based replication, you must ensure that binary logging is enabled, and establish a unique server ID.

Each server within a replication topology must be configured with a unique server ID, which you can specify using the `server_id` system variable. This server ID is used to identify individual servers within the replication topology, and must be a positive integer between 1 and (2^{32})−1. The default `server_id` value from MySQL 8.0 is 1. You can change the `server_id` value dynamically by issuing a statement like this:

```
SET GLOBAL server_id = 2;
```

How you organize and select the server IDs is your choice, so long as each server ID is different from every other server ID in use by any other server in the replication topology. Note that if a value of 0 (which was the default in earlier releases) was set previously for the server ID, you must restart the server to initialize the source with your new nonzero server ID. Otherwise, a server restart is not needed when you change the server ID, unless you make other configuration changes that require it.

Binary logging is required on the source because the binary log is the basis for replicating changes from the source to its replicas. Binary logging is enabled by default (the `log_bin` system variable is set to ON). The `--log-bin` option tells the server what base name to use for binary log files. It is recommended that you specify this option to give the binary log files a non-default base name, so that if the host name changes, you can easily continue to use the same binary log file names (see Section B.3.7, “Known Issues in MySQL”). If binary logging was previously disabled on the source using the `--skip-log-bin` option, you must restart the server without this option to enable it.

Note

The following options also have an impact on the source:

- For the greatest possible durability and consistency in a replication setup using `InnoDB` with transactions, you should use `innodb_flush_log_at_trx_commit=1` and `sync_binlog=1` in the source's `my.cnf` file.
- Ensure that the `skip_networking` system variable is not enabled on the source. If networking has been disabled, the replica cannot communicate with the source and replication fails.

17.1.2.2 Setting the Replica Configuration

Each replica must have a unique server ID, as specified by the `server_id` system variable. If you are setting up multiple replicas, each one must have a unique `server_id` value that differs from that of the source and from any of the other replicas. If the replica's server ID is not already set, or the current value conflicts with the value that you have chosen for the source or another replica, you must change it.

The default `server_id` value is 1. You can change the `server_id` value dynamically by issuing a statement like this:

```
SET GLOBAL server_id = 21;
```

Note that a value of 0 for the server ID prevents a replica from connecting to a source. If that server ID value (which was the default in earlier releases) was set previously, you must restart the server to initialize the replica with your new nonzero server ID. Otherwise, a server restart is not needed when you change the server ID, unless you make other configuration changes that require it. For example, if binary logging was disabled on the server and you want it enabled for your replica, a server restart is required to enable this.

If you are shutting down the replica server, you can edit the `[mysqld]` section of the configuration file to specify a unique server ID. For example:

```
[mysqld]
server-id=21
```

Binary logging is enabled by default on all servers. A replica is not required to have binary logging enabled for replication to take place. However, binary logging on a replica means that the replica's binary log can be used for data backups and crash recovery. Replicas that have binary logging enabled can also be used as part of a more complex replication topology. For example, you might want to set up replication servers using this chained arrangement:

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

Here, `A` serves as the source for the replica `B`, and `B` serves as the source for the replica `C`. For this to work, `B` must be both a source *and* a replica. Updates received from `A` must be logged by `B` to its binary log, in order to be passed on to `C`. In addition to binary logging, this replication topology requires the system variable `log_replica_updates` (from MySQL 8.0.26) or `log_slave_updates` (before MySQL 8.0.26) to be enabled. With replica updates enabled, the replica writes updates that are received from a source and performed by the replica's SQL thread to the replica's own binary log. The `log_replica_updates` or `log_slave_updates` system variable is enabled by default.

If you need to disable binary logging or replica update logging on a replica, you can do this by specifying the `--skip-log-bin` and `--log-replica-updates=OFF` or `--log-slave-updates=OFF` options for the replica. If you decide to re-enable these features on the replica, remove the relevant options and restart the server.

17.1.2.3 Creating a User for Replication

Each replica connects to the source using a MySQL user name and password, so there must be a user account on the source that the replica can use to connect. The user name is specified by the `SOURCE_USER` | `MASTER_USER` option of the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) when you set up a replica. 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 replica, or connect to the source using the same account for each replica.

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 replica's connection metadata repository `mysql.slave_master_info` (see Section 17.2.4.2, “Replication Metadata Repositories”). 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 source:

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

See Section 13.7.1, “Account Management Statements”, for more information on statements for manipulation of user accounts.

Important

To connect to the source using a user account that authenticates with the `caching_sha2_password` plugin, you must either set up a secure connection as described in Section 17.3.1, “Setting Up Replication to Use Encrypted Connections”, or enable the unencrypted connection to support password exchange using an RSA key pair. The `caching_sha2_password` authentication plugin is the default for new users created from MySQL 8.0 (for details, see Section 6.4.1.2, “Caching SHA-2 Pluggable Authentication”). If the user account that you create or use for replication (as specified by the `MASTER_USER` option) uses this authentication plugin, and you are not using a secure connection, you must enable RSA key pair-based password exchange for a successful connection.

17.1.2.4 Obtaining the Replication Source Binary Log Coordinates

To configure the replica to start the replication process at the correct point, you need to note the source's current coordinates within its binary log.

Warning

This procedure uses `FLUSH TABLES WITH READ LOCK`, which blocks `COMMIT` operations for `InnoDB` tables.

If you are planning to shut down the source 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 situation, the source creates a new binary log file on restart. The source binary log coordinates where the replica must start the replication process are therefore the start of that new file, which is the next binary log file on the source following after the files that are listed in the copied binary log index file.

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

1. Start a session on the source 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 source, use the `SHOW MASTER STATUS` statement to determine the current binary log file name and position:

   ```
   mysql> SHOW MASTER STATUS\G
   *************************** 1. row ***************************
                File: mysql-bin.000003
            Position: 73
        Binlog_Do_DB: test
    Binlog_Ignore_DB: manual, mysql
   Executed_Gtid_Set: 3E11FA47-71CA-11E1-9E33-C80AA9429562:1-5
   1 row in set (0.00 sec)
   ```

   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 replica. They represent the replication coordinates at which the replica should begin processing new updates from the source.

If the source has been running previously with binary logging disabled, 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 source's binary log file and position are the empty string (`''`) and `4`.

You now have the information you need to enable the replica to start reading from the source's binary log in the correct place to start replication.

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

- If you have existing data that needs be to synchronized with the replica 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 replica is in synchrony with the source. Proceed to Section 17.1.2.5, “Choosing a Method for Data Snapshots”.
- If you are setting up a new source and replica combination, you can exit the first session to release the read lock. See Setting Up Replication with New Source and Replicas for how to proceed.

17.1.2.5 Choosing a Method for Data Snapshots

If the source database contains existing data it is necessary to copy this data to each replica. There are different ways to dump the data from the source 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`.
- If your database is stored in binary portable files, you can copy the raw data files to a replica. This can be more efficient than using `mysqldump` and importing the file on each replica, because it skips the overhead of updating indexes as the `INSERT` statements are replayed. With storage engines such as `InnoDB` this is not recommended.
- Use MySQL Server's clone plugin to transfer all the data from an existing replica to a clone. For instructions to use this method, see Section 5.6.7.7, “Cloning for Replication”.

Tip

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

Creating a Data Snapshot Using mysqldump

To create a snapshot of the data in an existing source database, use the `mysqldump` tool. Once the data dump has been completed, import this data into the replica 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 REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement required on the replica to start the replication process:

```
$> 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 17.1.2.4, “Obtaining the Replication Source 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.

Note

By default, if GTIDs are in use on the source (`gtid_mode=ON`), `mysqldump` includes the GTIDs from the `gtid_executed` set on the source in the dump output to add them to the `gtid_purged` set on the replica. If you are dumping only specific databases or tables, it is important to note that the value that is included by `mysqldump` includes the GTIDs of all transactions in the `gtid_executed` set on the source, even those that changed suppressed parts of the database, or other databases on the server that were not included in the partial dump. Check the description for mysqldump's `--set-gtid-purged` option to find the outcome of the default behavior for the MySQL Server versions you are using, and how to change the behavior if this outcome is not suitable for your situation.

For more information, see Section 4.5.4, “mysqldump — A Database Backup Program”.

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

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 replica. MySQL Enterprise Backup is a commercial product that is included as part of a MySQL Enterprise subscription. See Section 30.2, “MySQL Enterprise Backup Overview” for detailed information.

This method also does not work reliably if the source and replica 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.

The following files are not required for replication:

- Files relating to the `mysql` database.
- The replica's connection metadata repository file `master.info`, if used; the use of this file is now deprecated (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”).
- The source's binary log files, with the exception of the binary log index file if you are going to use this to locate the source binary log coordinates for the replica.
- 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 source server during the process, as follows:

1. Acquire a read lock and get the source's status. See Section 17.1.2.4, “Obtaining the Replication Source Binary Log Coordinates”.
2. In a separate session, shut down the source server:

   ```
   $> 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:

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

4. Restart the source server.

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

1. Acquire a read lock and get the source's status. See Section 17.1.2.4, “Obtaining the Replication Source 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:

   ```
   $> tar cf /tmp/db.tar ./data
   $> zip -r /tmp/db.zip ./data
   $> 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 replica before starting the replication process.

17.1.2.6 Setting Up Replicas

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

- Configured the source with the necessary configuration properties. See Section 17.1.2.1, “Setting the Replication Source Configuration”.
- Obtained the source status information, or a copy of the source's binary log index file made during a shutdown for the data snapshot. See Section 17.1.2.4, “Obtaining the Replication Source Binary Log Coordinates”.
- On the source, released the read lock:

  ```
  mysql> UNLOCK TABLES;
  ```

- On the replica, edited the MySQL configuration. See Section 17.1.2.2, “Setting the Replica Configuration”.

The next steps depend on whether you have existing data to import to the replica or not. See Section 17.1.2.5, “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 Setting Up Replication with New Source and Replicas.
- If you have a snapshot of a database to import, see Setting Up Replication with Existing Data.

Setting Up Replication with New Source and Replicas

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

To set up replication between a source and a new replica:

1. Start up the replica.
2. Execute a `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement on the replica to set the source configuration. See Section 17.1.2.7, “Setting the Source Configuration on the Replica”.

Perform these replica setup steps on each replica.

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 source, the data is automatically replicated to the replicas.

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

```
$> mysql -h source < fulldb.dump
```

Setting Up Replication with Existing Data

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

Tip

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

Note

If the replication source server or existing replica that you are copying to create the new replica has any scheduled events, ensure that these are disabled on the new replica before you start it. If an event runs on the new replica that has already run on the source, the duplicated operation causes an error. The Event Scheduler is controlled by the `event_scheduler` system variable, which defaults to `ON` from MySQL 8.0, so events that are active on the original

server run by default when the new replica starts up. To stop all events from running on the new replica, set the `event_scheduler` system variable to `OFF` or `DISABLED` on the new replica. Alternatively, you can use the `ALTER EVENT` statement to set individual events to `DISABLE` or `DISABLE ON SLAVE` to prevent them from running on the new replica. You can list the events on a server using the `SHOW` statement or the Information Schema `EVENTS` table. For more information, see Section 17.5.1.16, “Replication of Invoked Features”.

As an alternative to creating a new replica in this way, MySQL Server's clone plugin can be used to transfer all the data and replication settings from an existing replica to a clone. For instructions to use this method, see Section 5.6.7.7, “Cloning for Replication”.

Follow this procedure to set up replication with existing data:

1. If you used MySQL Server's clone plugin to create a clone from an existing replica (see Section 5.6.7.7, “Cloning for Replication”), the data is already transferred. Otherwise, import the data to the replica using one of the following methods.

   a. If you used `mysqldump`, start the replica server, ensuring that replication does not start by using the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable. Then import the dump file:

   ```
   $> mysql < fulldb.dump
   ```

   b. If you created a snapshot using the raw data files, extract the data files into your replica's data directory. For example:

   ```
   $> tar xvf dbdump.tar
   ```

   You may need to set permissions and ownership on the files so that the replica server can access and modify them. Then start the replica server, ensuring that replication does not start by using the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable.

2. Configure the replica with the replication coordinates from the source. This tells the replica the binary log file and position within the file where replication needs to start. Also, configure the replica with the login credentials and host name of the source. For more information on the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement required, see Section 17.1.2.7, “Setting the Source Configuration on the Replica”.

3. Start the replication threads by issuing a `START REPLICA` (or before MySQL 8.0.22, `START SLAVE`) statement.

After you have performed this procedure, the replica connects to the source and replicates any updates that have occurred on the source since the snapshot was taken. Error messages are issued to the replica's error log if it is not able to replicate for any reason.

The replica uses information logged in its connection metadata repository and applier metadata repository to keep track of how much of the source's binary log it has processed. From MySQL 8.0, by default, these repositories are tables named `slave_master_info` and `slave_relay_log_info` in the `mysql` database. Do *not* remove or edit these tables unless you know exactly what you are doing and fully understand the implications. Even in that case, it is preferred that you use the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement to change replication parameters. The replica uses the values specified in the statement to update the replication metadata repositories automatically. See Section 17.2.4, “Relay Log and Replication Metadata Repositories”, for more information.

Note

The contents of the replica's connection metadata repository override some of the server options specified on the command line or in `my.cnf`. See

Section 17.1.6, “Replication and Binary Logging Options and Variables”, for more details.

A single snapshot of the source suffices for multiple replicas. To set up additional replicas, use the same source snapshot and follow the replica portion of the procedure just described.

17.1.2.7 Setting the Source Configuration on the Replica

To set up the replica to communicate with the source for replication, configure the replica with the necessary connection information. To do this, on the replica, execute the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23), replacing the option values with the actual values relevant to your system:

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

Or from MySQL 8.0.23:
mysql> CHANGE REPLICATION SOURCE TO
    ->     SOURCE_HOST='source_host_name',
    ->     SOURCE_USER='replication_user_name',
    ->     SOURCE_PASSWORD='replication_password',
    ->     SOURCE_LOG_FILE='recorded_log_file_name',
    ->     SOURCE_LOG_POS=recorded_log_position;
```

Note

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

The `CHANGE REPLICATION SOURCE TO` | `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 Section 13.4.2.1, “CHANGE MASTER TO Statement”.

Important

As noted in Section 17.1.2.3, “Creating a User for Replication”, if you are not using a secure connection and the user account named in the `SOURCE_USER` | `MASTER_USER` option authenticates with the `caching_sha2_password` plugin (the default from MySQL 8.0), you must specify the `SOURCE_PUBLIC_KEY_PATH` | `MASTER_PUBLIC_KEY_PATH` or `GET_SOURCE_PUBLIC_KEY` | `GET_MASTER_PUBLIC_KEY` option in the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement to enable RSA key pair-based password exchange.

17.1.2.8 Adding Replicas to a Replication Environment

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

Note

If the replication source server or existing replica that you are copying to create the new replica has any scheduled events, ensure that these are disabled on the new replica before you start it. If an event runs on the new replica that has already run on the source, the duplicated operation causes an error. The

> Event Scheduler is controlled by the `event_scheduler` system variable, which defaults to `ON` from MySQL 8.0, so events that are active on the original server run by default when the new replica starts up. To stop all events from running on the new replica, set the `event_scheduler` system variable to `OFF` or `DISABLED` on the new replica. Alternatively, you can use the `ALTER EVENT` statement to set individual events to `DISABLE` or `DISABLE ON SLAVE` to prevent them from running on the new replica. You can list the events on a server using the `SHOW` statement or the Information Schema `EVENTS` table. For more information, see Section 17.5.1.16, “Replication of Invoked Features”.

As an alternative to creating a new replica in this way, MySQL Server's clone plugin can be used to transfer all the data and replication settings from an existing replica to a clone. For instructions to use this method, see Section 5.6.7.7, “Cloning for Replication”.

To duplicate an existing replica without cloning, follow these steps:

1. Stop the existing replica and record the replica status information, particularly the source binary log file and relay log file positions. You can view the replica status either in the Performance Schema replication tables (see Section 27.12.11, “Performance Schema Replication Tables”), or by issuing `SHOW REPLICA STATUS` as follows:

   ```
   mysql> STOP SLAVE;
   mysql> SHOW SLAVE STATUS\G
   Or from MySQL 8.0.22:
   mysql> STOP REPLICA;
   mysql> SHOW REPLICA STATUS\G
   ```

2. Shut down the existing replica:

   ```
   $> mysqladmin shutdown
   ```

3. Copy the data directory from the existing replica to the new replica, 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 replica 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 replica might be in their own directories outside the data directory. Check through the system variables that are set for the existing replica 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 replication metadata repositories (see Section 17.2.4, “Relay Log and Replication Metadata Repositories”), ensure that you also copy these files from the existing replica to the new replica. If tables have been used for the repositories, which is the default from MySQL 8.0, the tables are in the data directory.
   - After copying, delete the `auto.cnf` file from the copy of the data directory on the new replica, so that the new replica is started with a different generated server UUID. The server UUID must be unique.

   A common problem that is encountered when adding new replicas is that the new replica 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 replica and has his hostname
   ```

```
changed!! Please use '--relay-log=new_replica_hostname-relay-bin' to avoid this problem.
071118 16:44:10 [ERROR] Failed to open the relay log './old_replica_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 17.1.6, “Replication and Binary Logging Options and Variables”.

To avoid this problem, use the same value for `relay_log` on the new replica that was used on the existing replica. If this option was not set explicitly on the existing replica, use *`existing_replica_hostname`*`-relay-bin`. If this is not possible, copy the existing replica's relay log index file to the new replica and set the `relay_log_index` system variable on the new replica to match what was used on the existing replica. If this option was not set explicitly on the existing replica, use *`existing_replica_hostname`*`-relay-bin.index`. Alternatively, if you have already tried to start the new replica 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 REPLICA` on the new replica.

   If you have already started the existing replica again, issue `STOP REPLICA` on the existing replica as well.

b. Copy the contents of the existing replica's relay log index file into the new replica'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 replica.

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

6. Start the new replica server, ensuring that replication does not start yet by specifying the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable. Use the Performance Schema replication tables or issue `SHOW REPLICA STATUS` to confirm that the new replica has the correct settings when compared with the existing replica. Also display the server ID and server UUID and verify that these are correct and unique for the new replica.

7. Start the replica threads by issuing a `START REPLICA` statement. The new replica now uses the information in its connection metadata repository to start the replication process.

17.1.3 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 replicas; this means that it is not necessary when using GTIDs to refer to log files or positions within those files when starting a new replica or failing over to a new source, which greatly simplifies these tasks. Because GTID-based replication is completely transaction-based, it is simple to determine whether sources and replicas are consistent; as long as all transactions committed on a source are also committed on a replica, consistency between the two is guaranteed. You can use either statement-based or row-based replication with GTIDs (see Section 17.2.1, “Replication Formats”); however, for best results, we recommend that you use the row-based format.

GTIDs are always preserved between source and replica. This means that you can always determine the source for any transaction applied on any replica 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 source can be applied no more than once on the replica, 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 17.1.3.1, “GTID Format and Storage”).
- The life cycle of a GTID (see Section 17.1.3.2, “GTID Life Cycle”).
- The auto-positioning function for synchronizing a replica and source that use GTIDs (see Section 17.1.3.3, “GTID Auto-Positioning”).
- A general procedure for setting up and starting GTID-based replication (see Section 17.1.3.4, “Setting Up Replication Using GTIDs”).
- Suggested methods for provisioning new replication servers when using GTIDs (see Section 17.1.3.5, “Using GTIDs for Failover and Scaleout”).
- Restrictions and limitations that you should be aware of when using GTID-based replication (see Section 17.1.3.7, “Restrictions on Replication with GTIDs”).
- Stored functions that you can use to work with GTIDs (see Section 17.1.3.8, “Stored Function Examples to Manipulate GTIDs”).

For information about MySQL Server options and variables relating to GTID-based replication, see Section 17.1.6.5, “Global Transaction ID System Variables”. See also Section 12.19, “Functions Used with Global Transaction Identifiers (GTIDs)”, which describes SQL functions supported by MySQL 8.0 for use with GTIDs.

17.1.3.1 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 source). 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 source, and replicated transactions, which are reproduced on a replica. When a client transaction is committed on the source, 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 replica, or is filtered out on the replica. 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 source can be applied no more than once on the replica, 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 blocks. 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:

```
GTID = source_id:transaction_id
```

The *`source_id`* identifies the originating server. Normally, the source'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 source. 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 upper limit for sequence numbers for GTIDs on a server instance is the number of non-negative values for a signed 64-bit integer (2 to the power of 63 minus 1, or 9,223,372,036,854,775,807). If the server runs out of GTIDs, it takes the action specified by `binlog_error_action`. From MySQL 8.0.23, a warning message is issued when the server instance is approaching the limit.

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 `gtid_next` system variable (`@@GLOBAL.gtid_next`) 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 REPLICA` (or before MySQL 8.0.22, `START SLAVE`) clauses `UNTIL SQL_BEFORE_GTIDS` and `UNTIL SQL_AFTER_GTIDS` can be used to make a replica 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:

```
3E11FA47-71CA-11E1-9E33-C80AA9429562:1-3:11:47-49
```

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 replica that has applied transactions from more than one source:

```
2174B383-5441-11E8-B90A-C80AA9429562:1-3, 24DA167-0C0C-11E8-8442-00059A3C7B00:1-19
```

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:
    hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh

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 replica to use GTIDs when binary logging is disabled on the replica, 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`. If binary logging is disabled (`log_bin` is `OFF`), or if `log_replica_updates` or `log_slave_updates` is disabled, the server stores the GTID belonging to each transaction together with the transaction in the `mysql.gtid_executed` table at transaction commit time. In addition, the table is compressed periodically at a user-configurable rate, as described in mysql.gtid_executed Table Compression.

If binary logging is enabled (`log_bin` is `ON`), from MySQL 8.0.17 for the `InnoDB` storage engine only, the server updates the `mysql.gtid_executed` table in the same way as when binary logging or replica update logging is disabled, storing the GTID for each transaction at transaction commit time. However, in releases before MySQL 8.0.17, and for other storage engines, the server only updates the `mysql.gtid_executed` table when the binary log is rotated or the server is shut down. At these times, 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 source prior to MySQL 8.0.17, or on a replica prior to MySQL 8.0.17 where binary logging is enabled, or with storage engines other than `InnoDB`, it has the following consequences:

- 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 so that replication can continue. The exception to this is if you disable

binary logging when the server is restarted (using `--skip-log-bin` or `--disable-log-bin`). In that case, the server cannot access the binary log file to recover the GTIDs, so replication cannot be started.

- 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. In releases before MySQL 8.0.17 and with storage engines other than `InnoDB`, always use `@@GLOBAL.gtid_executed`, which is updated after every commit, to represent the GTID state for the MySQL server, instead of querying the `mysql.gtid_executed` table.

The MySQL server can write to the `mysql.gtid_executed` table even when the server is in read only or super read only mode. In releases before MySQL 8.0.17, this ensures that the binary log file can still be rotated in these modes. If the `mysql.gtid_executed` table cannot be accessed for writes, and the binary log file is rotated for any reason other than reaching the maximum file size (`max_binlog_size`), the current binary log file continues to be used. An error message is returned to the client that requested the rotation, and a warning is logged on the server. If the `mysql.gtid_executed` table cannot be accessed for writes and `max_binlog_size` is reached, the server responds according to its `binlog_error_action` setting. If `IGNORE_ERROR` is set, an error is logged on the server and binary logging is halted, or if `ABORT_SERVER` is set, the server shuts down.

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:

```
+--------------------------------------+----------------+--------------+
| source_uuid                          | interval_start | interval_end |
|--------------------------------------+----------------+--------------|
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 37             | 37           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 38             | 38           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 39             | 39           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 40             | 40           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 41             | 41           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 42             | 42           |
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 43             | 43           |
...
```

To save space, the MySQL server can compress 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:

```
+--------------------------------------+----------------+--------------+
| source_uuid                          | interval_start | interval_end |
|--------------------------------------+----------------+--------------|
| 3E11FA47-71CA-11E1-9E33-C80AA9429562 | 37             | 43           |
...
```

The server can carry out compression using 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 '%gtid%'\G
*************************** 1. row ***************************
          THREAD_ID: 26
               NAME: thread/sql/compress_gtid_table
               TYPE: FOREGROUND
     PROCESSLIST_ID: 1
   PROCESSLIST_USER: NULL
   PROCESSLIST_HOST: NULL
     PROCESSLIST_DB: NULL
PROCESSLIST_COMMAND: Daemon
   PROCESSLIST_TIME: 1509
  PROCESSLIST_STATE: Suspending
   PROCESSLIST_INFO: NULL
   PARENT_THREAD_ID: 1
```

```
              ROLE: NULL
      INSTRUMENTED: YES
           HISTORY: YES
   CONNECTION_TYPE: NULL
      THREAD_OS_ID: 18677
```

When binary logging is enabled on the server, this compression method is not used, and instead the `mysql.gtid_executed` table is compressed on each binary log rotation. However, when binary logging is disabled on the server, the `thread/sql/compress_gtid_table` thread sleeps until a specified number of transactions have been executed, then wakes up to perform compression of the `mysql.gtid_executed` table. It then sleeps until the same number of transactions have taken place, then wakes up to perform the compression again, repeating this loop indefinitely. The number of transactions that elapse before the table is compressed, and thus the compression rate, is controlled by the value of the `gtid_executed_compression_period` system variable. Setting that value to 0 means that the thread never wakes up, meaning that this explicit compression method is not used. Instead, compression occurs implicitly as required.

From MySQL 8.0.17, `InnoDB` transactions are written to the `mysql.gtid_executed` table by a separate process to non-`InnoDB` transactions. This process is controlled by a different thread, `innodb/clone_gtid_thread`. This GTID persister thread collects GTIDs in groups, flushes them to the `mysql.gtid_executed` table, then compresses the table. If the server has a mix of `InnoDB` transactions and non-`InnoDB` transactions, which are written to the `mysql.gtid_executed` table individually, the compression carried out by the `compress_gtid_table` thread interferes with the work of the GTID persister thread and can slow it significantly. For this reason, from that release it is recommended that you set `gtid_executed_compression_period` to 0, so that the `compress_gtid_table` thread is never activated.

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

For releases before MySQL 8.0.17, the default value of 1000 for `gtid_executed_compression_period` can be used, meaning that compression of the table is performed after each 1000 transactions, or you can choose an alternative value. In those releases, if you set a value of 0 and binary logging is disabled, explicit compression is not performed on the `mysql.gtid_executed` table, and you should be prepared for a potentially large increase in the amount of disk space that may be required by the table if you do this.

When a server instance is started, if `gtid_executed_compression_period` is set to a nonzero value and the `thread/sql/compress_gtid_table` thread is launched, in most server configurations, explicit compression is performed for the `mysql.gtid_executed` table. In releases before MySQL 8.0.17 when binary logging is enabled, compression is triggered by the fact of the binary log being rotated at startup. In releases from MySQL 8.0.20, compression is triggered by the thread launch. In the intervening releases, compression does not take place at startup.

17.1.3.2 GTID Life Cycle

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

1. A transaction is executed and committed on the source. This client transaction is assigned a GTID composed of the source's UUID and the smallest nonzero transaction sequence number not yet used on this server. The GTID is written to the source'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 source), 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 replica and stored in the replica's relay log (using established mechanisms for this process, see Section 17.2, “Replication Implementation”, for details), the replica reads the GTID and sets the value of its `gtid_next` system variable as this GTID. This tells the replica that the next transaction must be logged using this GTID. It is important to note that the replica sets `gtid_next` in a session context.

5. The replica 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 replica guarantees not only that no previous transaction having this GTID has been applied on the replica, 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 replica 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 replica applies the replicated transaction. Because `gtid_next` is set to the GTID already assigned by the source, the replica 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 replica, 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 replica, 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. From MySQL 8.0, this operation is atomic for DDL statements as well as for DML statements. In this situation, the `mysql.gtid_executed` table is a complete record of the transactions applied on the replica.

9. Very shortly after the replicated transaction is committed on the replica, 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 replica. As for the source, this GTID set contains a representation of the set of all committed GTID transactions. If binary logging is disabled on the replica, the `mysql.gtid_executed` table is also a complete record of the transactions applied on the replica. If binary logging is enabled on the replica, 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 source 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 replica are persisted. If binary logging is enabled on the replica, 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 replica reconnects to the source, as explained in Section 17.1.3.3, “GTID Auto-Positioning”.

On a multithreaded replica (with `replica_parallel_workers > 0` or `slave_parallel_workers > 0`), transactions can be applied in parallel, so replicated transactions can commit out of order (unless `replica_preserve_commit_order=1` or `slave_preserve_commit_order=1` is set). When that happens, the set of GTIDs in the `gtid_executed` system variable contains multiple GTID ranges with gaps between them. (On a source or a single-threaded replica, there are monotonically increasing GTIDs without gaps between the numbers.) Gaps on multithreaded replicas only occur among the most recently applied transactions, and are filled in as replication progresses. When replication threads are stopped cleanly using the `STOP REPLICA` 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 replica begins to apply events from a source 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. Multiple GTIDs can be generated if any of the tables use storage engines that do not support atomic DDL, or if any of the tables are temporary tables.
- 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 replica, 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 replica. It also means the GTID is added to gtid_executed on the replica even when binary logging or replica update logging is disabled on the replica, or when the transaction is a no-op or is filtered out on the replica.

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

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. Before MySQL 8.0, you could only change the value of gtid_purged when gtid_executed (and therefore gtid_purged) was empty. From MySQL 8.0, this restriction does not apply, and you can also choose whether to replace the whole GTID set in gtid_purged with a specified GTID set, or to add a specified GTID set to the GTIDs already in gtid_purged. 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. This step gives the set of GTIDs that are currently, or were once, recorded in a binary log on the server (`gtids_in_binlog`). Next, the GTIDs in `Previous_gtids_log_event` in the oldest binary log file are subtracted from `gtids_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, `gtids_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 replica where binary logging is disabled. `RESET REPLICA` has no effect on the GTID execution history.

17.1.3.3 GTID Auto-Positioning

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

To start a replica using GTID-based replication, you need to enable the `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option in the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). The alternative `SOURCE_LOG_FILE` | `MASTER_LOG_FILE` and `SOURCE_LOG_POS` | `MASTER_LOG_POS` options specify the name of the log file and the starting position within the file, but with GTIDs the replica does not need this nonlocal data.. For full instructions to configure and start sources and replicas using GTID-based replication, see Section 17.1.3.4, “Setting Up Replication Using GTIDs”.

The `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option is disabled by default. If multi-source replication is enabled on the replica, you need to set the option for each applicable replication channel. Disabling the `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option again makes the replica revert to file-based replication, in which case you must also specify one or both of the `SOURCE_LOG_FILE` | `MASTER_LOG_FILE` or `SOURCE_LOG_POS` | `MASTER_LOG_POS` options.

When a replica 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 source. The source must have `GTID_MODE=ON` set in order for the connection to succeed. In the initial handshake, the replica 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 source responds by sending all transactions recorded in its binary log whose GTID is not included in the GTID set sent by the replica. To do this, the source first identifies the appropriate binary log file to begin working with, by checking the `Previous_gtids_log_event` in the header of each of its binary log files, starting with the most recent. When the source finds the first `Previous_gtids_log_event` which contains no transactions that the replica is missing, it begins with that binary log file. This method is efficient and only takes a significant amount of time if the replica is behind the source by a large number of binary log files. The source then reads the transactions in that binary log file and subsequent files up to the current one, sending the transactions with GTIDs that the replica is missing, and skipping the transactions that were in the GTID set sent by the replica. The elapsed time until the replica receives the first missing transaction depends on its offset in the binary log file. This exchange ensures that the source only sends the transactions with a GTID that the replica has not already received or committed. If the replica receives transactions from more than one source, 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 source have been purged from the source's binary log, or added to the set of GTIDs in the `gtid_purged` system variable by another method, the source sends the error `ER_MASTER_HAS_PURGED_REQUIRED_GTIDS` to the replica, and replication does not start. The GTIDs of the missing purged transactions are identified and listed in the source's error log in the warning message `ER_FOUND_MISSING_GTIDS`. The replica cannot recover automatically from this error because parts of the transaction history that are needed to catch up with the source have been purged. Attempting to reconnect without the `MASTER_AUTO_POSITION` option enabled only results in the loss of the purged transactions on the replica. The correct approach to recover from this situation is for the replica to replicate the missing transactions listed in the `ER_FOUND_MISSING_GTIDS` message from another source, or for the replica to be replaced by a new replica created from a more recent backup. Consider revising the binary log expiration period (`binlog_expire_logs_seconds`) on the source to ensure that the situation does not occur again.

If during the exchange of transactions it is found that the replica has received or committed transactions with the source's UUID in the GTID, but the source itself does not have a record of them, the source sends the error `ER_SLAVE_HAS_MORE_GTIDS_THAN_MASTER` to the replica and replication does not start. This situation can occur if a source 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 replica. The source and replica can diverge if any clients commit transactions on the source after it is restarted, which can lead to the situation where the source and replica are using the same GTID for different transactions. The correct approach to recover from this situation is to check manually whether the source and replica 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 source or the replica from the replication topology. If the issue is only missing transactions on the source, you can make the source into a replica instead, allow it to catch up with the other servers in the replication topology, and then make it a source again if needed.

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

17.1.3.4 Setting Up Replication Using GTIDs

This section describes a process for configuring and starting GTID-based replication in MySQL 8.0. This is a "cold start" procedure that assumes either that you are starting the source server for the first time, or that it is possible to stop it; for information about provisioning replicas using GTIDs from a running source server, see Section 17.1.3.5, "Using GTIDs for Failover and Scaleout". For information about changing GTID mode on servers online, see Section 17.1.4, "Changing GTID Mode on Online Servers".

The key steps in this startup process for the simplest possible GTID replication topology, consisting of one source and one replica, 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 replica to use the source 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 replica, then disable read-only mode on both servers, so that they can accept updates.

In the following example, two servers are already running as source and replica, using MySQL's binary log position-based replication protocol. If you are starting with new servers, see Section 17.1.2.3, “Creating a User for Replication” for information about adding a specific user for replication connections and Section 17.1.2.1, “Setting the Replication Source 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 Section 4.2.2.2, “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 replica to catch up with the source. *It is extremely important that you make sure the replica 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:

```
$> mysqladmin -uusername -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
enforce-gtid-consistency=ON
```

Start each replica with the `--skip-slave-start` option, or from MySQL 8.0.24, the `skip_slave_start` system variable, to ensure that replication does not start until you have configured the replica settings. From MySQL 8.0.26, use `--skip-replica-start` or `skip_replica_start` instead. For more information on GTID related options and variables, see Section 17.1.6.5, “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. Source servers must always have binary logging enabled in order to be able to replicate. However, replica servers can use GTIDs but without binary logging. If you need to disable binary logging on a replica server, you can do this by specifying the `--skip-log-bin` and `--log-replica-updates=OFF` or `--log-slave-updates=OFF` options for the replica.

Step 4: Configure the replica to use GTID-based auto-positioning. Tell the replica to use the source with GTID based transactions as the replication data source, and to use GTID-based auto-positioning rather than file-based positioning. Issue a `CHANGE REPLICATION SOURCE TO` statement

(from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) on the replica, including the `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option in the statement to tell the replica that the source's transactions are identified by GTIDs.

You may also need to supply appropriate values for the source'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 replica to connect to the source; 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;

Or from MySQL 8.0.23:

mysql> CHANGE REPLICATION SOURCE TO
     >     SOURCE_HOST = host,
     >     SOURCE_PORT = port,
     >     SOURCE_USER = user,
     >     SOURCE_PASSWORD = password,
     >     SOURCE_AUTO_POSITION = 1;
```

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 replica and disable read-only mode. Start the replica like this:

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

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 source as before. Section 17.1.3.5, “Using GTIDs for Failover and Scaleout”, discusses creation of new replicas when using GTIDs.

17.1.3.5 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 replica which can then be used for scaleout, being promoted to source as necessary for failover. This section describes the following techniques:

- Simple replication
- Copying data and transactions to the replica
- Injecting empty transactions
- Excluding transactions with gtid_purged
- Restoring GTID mode replicas

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 necessary for restoring a replica that is immediately available as a candidate to become a new source 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 replica of a source that has the entire execution history, and enable global transaction identifiers on both servers. See Section 17.1.3.4, “Setting Up Replication Using GTIDs”, for more information.

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

This method is simple and effective, but requires the replica to read the binary log from the source; it can sometimes take a comparatively long time for the new replica to catch up with the source, 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 source by copying binary log files to the new server.

Copying data and transactions to the replica. 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 replica. 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 replica. The server where the snapshot is taken can be either the source or one of its replicas, but you must ensure that the server 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 transferred to the replica, as outlined here:

Data Set

1. Create a dump file using `mysqldump` on the source server. Set the `mysqldump` option `--master-data` (with the default value of 1) to include a `CHANGE REPLICATION SOURCE TO` | `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 17.1.2.5, “Choosing a Method for Data Snapshots”. 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 replica. MySQL Enterprise Backup is a commercial product that is included as part of a MySQL Enterprise subscription. See Section 30.2, “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 replica's

data directory, then restart the replica. If you use this method, the replica must be configured for GTID-based replication, in other words with `gtid_mode=ON`. For instructions and important information for this method, see Section 17.1.2.8, “Adding Replicas 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 replica using `mysqlbinlog`, with the `--read-from-remote-server`, `--read-from-remote-source`, and `--read-from-remote-master` options.

2. Alternatively, copy the source server's binary log files to the replica. You can make copies from the replica using `mysqlbinlog` with the `--read-from-remote-server` and `--raw` options. These can be read into the replica 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:

   ```
   $> mysqlbinlog copied-binlog.000001 copied-binlog.000002 | mysql -u roo
   ```

   For more information, see Section 4.6.9.3, “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 source. This means that the replica's availability is not instantaneous, but only a relatively short amount of time should be required for the replica 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 source 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 replica discussed in this section use other means to transfer information about transactions to the new replica.

Injecting empty transactions. The source's global `gtid_executed` variable contains the set of all transactions executed on the source. 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 source's `gtid_executed`, like this:

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

BEGIN;
COMMIT;

SET GTID_NEXT='AUTOMATIC';
```

Once all transaction identifiers have been reinstated in this way using empty transactions, you must flush and purge the replica's binary logs, as shown here, where *`N`* is the nonzero suffix of the current binary log file name:

```
FLUSH LOGS;
PURGE BINARY LOGS TO 'source-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 the source. (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 source as its binary log history converges with that of the replication stream (that is, as it catches up with the source or sources). 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 source's global `gtid_purged` variable contains the set of all transactions that have been purged from the source'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 replica 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 source as its binary log history converges with that of the source and other replicas.

Restoring GTID mode replicas. When restoring a replica 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 replica and run the transaction you copied. The `mysqlbinlog` output sets the delimiter to `/*!*/;`, so set it back:

```
mysql> DELIMITER ;
```

Restart replication from the correct position automatically:

```
mysql> SET GTID_NEXT=automatic;
mysql> RESET SLAVE;
mysql> START SLAVE;
Or from MySQL 8.0.22:
mysql> SET GTID_NEXT=automatic;
mysql> RESET REPLICA;
mysql> START REPLICA;
```

17.1.3.6 Replication From a Source Without GTIDs to a Replica With GTIDs

From MySQL 8.0.23, you can set up replication channels to assign a GTID to replicated transactions that do not already have one. This feature enables replication from a source server that does not have GTIDs enabled and does not use GTID-based replication, to a replica that has GTIDs enabled. If it is possible to enable GTIDs on the replication source server, as described in Section 17.1.4, “Changing GTID Mode on Online Servers”, use that approach instead. This feature is designed for replication source servers where you cannot enable GTIDs. Note that as is standard for MySQL replication, this feature does not support replication from MySQL source servers earlier than the previous release series, so MySQL 5.7 is the earliest supported source for a MySQL 8.0 replica.

You can enable GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement. `LOCAL` assigns a GTID including the replica's own UUID (the `server_uuid` setting). *`uuid`* assigns a GTID including the specified UUID, such as the `server_uuid` setting for the replication source server. Using a nonlocal UUID lets you differentiate between transactions that originated on the replica and transactions that originated on the source, and for a multi-source replica, between

transactions that originated on different sources. If any of the transactions sent by the source do have a GTID already, that GTID is retained.

Important

A replica set up with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel cannot be promoted to replace the replication source server in the event that a failover is required, and a backup taken from the replica cannot be used to restore the replication source server. The same restriction applies to replacing or restoring other replicas that use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on any channel.

The replica must have `gtid_mode=ON` set, and this cannot be changed afterwards, unless you remove the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS=ON` setting. If the replica server is started without GTIDs enabled and with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` set for any replication channels, the settings are not changed, but a warning message is written to the error log explaining how to change the situation.

For a multi-source replica, you can have a mix of channels that use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, and channels that do not. Channels specific to Group Replication cannot use `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, but an asynchronous replication channel for another source on a server instance that is a Group Replication group member can do so. For a channel on a Group Replication group member, do not specify the Group Replication group name as the UUID for creating the GTIDs.

Using `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` on a replication channel is not the same as introducing GTID-based replication for the channel. The GTID set (`gtid_executed`) from a replica set up with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` should not be transferred to another server or compared with another server's `gtid_executed` set. The GTIDs that are assigned to the anonymous transactions, and the UUID you choose for them, only have significance for that replica's own use. The exception to this is any downstream replicas of the replica where you enabled `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, and any servers that were created from a backup of that replica.

If you set up any downstream replicas, these servers do not have `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` enabled. Only the replica that is receiving transactions directly from the non-GTID source server needs to have `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` set on the relevant replication channel. Among that replica and its downstream replicas, you can compare GTID sets, fail over from one replica to another, and use backups to create additional replicas, as you would in any GTID-based replication topology. `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` is used where transactions are received from a non-GTID server outside this group.

A replication channel using `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` has the following behavior differences to GTID-based replication:

- GTIDs are assigned to the replicated transactions when they are applied (unless they already had a GTID). A GTID would normally be assigned on the replication source server when the transaction is committed, and sent to the replica along with the transaction. On a multi-threaded replica, this means the order of the GTIDs does not necessarily match the order of the transactions, even if `slave-preserve-commit-order=1` is set.
- The `SOURCE_LOG_FILE` and `SOURCE_LOG_POS` options of the `CHANGE REPLICATION SOURCE TO` statement are used to position the replication I/O (receiver) thread, rather than the `SOURCE_AUTO_POSITION` option.
- The `SET GLOBAL sql_replica_skip_counter` or `SET GLOBAL sql_slave_skip_counter` statement is used to skip transactions on a replication channel set up with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS`, rather than the method of committing empty transactions. For instructions, see Section 17.1.7.3, “Skipping Transactions”.

- The `UNTIL SQL_BEFORE_GTIDS` and `UNTIL_SQL_AFTER_GTIDS` options of the `START REPLICA` statement cannot be used for the channel.
- The function `WAIT_UNTIL_SQL_THREAD_AFTER_GTIDS()`, which is deprecated from MySQL 8.0.18, cannot be used with the channel. Its replacement `WAIT_FOR_EXECUTED_GTID_SET()`, which works across the server, can be used to wait for any downstream replicas of the server that has `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` enabled. To wait for the channel with `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` enabled to catch up with the source, which does not use GTIDs, use the `SOURCE_POS_WAIT()` function (from MySQL 8.0.26) or the `MASTER_POS_WAIT()` function.

The Performance Schema `replication_applier_configuration` table shows whether GTIDs are assigned to anonymous transactions on a replication channel, what the UUID is, and whether it is the UUID of the replica server (`LOCAL`) or a user-specified UUID (`UUID`). The information is also recorded in the applier metadata repository. A `RESET REPLICA ALL` statement resets the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` setting, but a `RESET REPLICA` statement does not.

17.1.3.7 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 source and the replica 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. Prior to MySQL 8.0.21, `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 source used `STATEMENT` format and a replica used `ROW` format, the replica would be unable to handle the transaction correctly, therefore the `CREATE TABLE ... SELECT` statement is disallowed with GTIDs to prevent this scenario. This restriction is lifted in MySQL 8.0.21 on storage engines that support atomic DDL. In this case, `CREATE TABLE ... SELECT` is recorded in the binary log as one transaction. For more information, see Section 13.1.1, “Atomic Data Definition Statement Support”.

Temporary tables. When `binlog_format` is set to `STATEMENT`, `CREATE TEMPORARY TABLE` and `DROP TEMPORARY TABLE` statements cannot be used inside transactions, procedures, functions, and triggers when GTIDs are in use on the server (that is, when the `enforce_gtid_consistency` system variable is set to `ON`). They can be used outside these contexts when GTIDs are in use, provided that `autocommit=1` is set. From MySQL 8.0.13, when `binlog_format` is set to `ROW` or `MIXED`, `CREATE TEMPORARY TABLE` and `DROP TEMPORARY TABLE` statements are allowed inside a transaction, procedure, function, or trigger when GTIDs are in use. The statements are not written to the binary log and are therefore not replicated to replicas. The use of row-based replication means

that the replicas remain in sync without the need to replicate temporary tables. If the removal of these statements from a transaction results in an empty transaction, the transaction is not written to the binary log.

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 17.1.3.4, "Setting Up Replication Using GTIDs".

Skipping transactions. `sql_replica_skip_counter` or `sql_slave_skip_counter` is not available when using GTID-based replication. If you need to skip transactions, use the value of the source's `gtid_executed` variable instead. If you have enabled GTID assignment on a replication channel using the `ASSIGN_GTIDS_TO_ANONYMOUS_TRANSACTIONS` option of the `CHANGE REPLICATION SOURCE TO` statement, `sql_replica_skip_counter` or `sql_slave_skip_counter` is available. For more information, see Section 17.1.7.3, "Skipping Transactions".

Ignoring servers. The IGNORE_SERVER_IDS option of the `CHANGE REPLICATION SOURCE TO` | `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 REPLICA 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 mysql_upgrade. Prior to MySQL 8.0.16, 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). As of MySQL 8.0.16, the server performs the entire MySQL upgrade procedure, but disables binary logging during the upgrade, so there is no issue.

17.1.3.8 Stored Function Examples to Manipulate GTIDs

MySQL includes some built-in (native) functions for use with GTID-based replication. These functions are as follows:

`GTID_SUBSET(`*`set1`*`,`*`set2`*`)`	Given two sets of global transaction identifiers *`set1`* and *`set2`*, returns true if all GTIDs in *`set1`* are also in *`set2`*. Returns false otherwise.
`GTID_SUBTRACT(`*`set1`*`,`*`set2`*`)`	Given two sets of global transaction identifiers *`set1`* and *`set2`*, returns only those GTIDs from *`set1`* that are not in *`set2`*.
`WAIT_FOR_EXECUTED_GTID_SET(`*`gtid_set`*`[, `*`timeout`*`])`	Wait until the server has applied all of the transactions whose global transaction identifiers are contained in *`gtid_set`*. The optional timeout stops the function from waiting after the specified number of seconds have elapsed.

For details of these functions, see Section 12.19, "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 Chapter 25, *Stored Objects*. The following examples show some useful stored functions that can be created based on the built-in `GTID_SUBSET()` and `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:

```
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.

```
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.

```
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 `sum` is the union of the two sets.

```
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.

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

This function returns the union of two GTID sets.

```
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 `gtid_set_1` but not in `gtid_set_2`, and also the GTIDs that exist in `gtid_set_2` but not in `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 `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 17.1 Verifying that a replica is up to date

The built-in functions `GTID_SUBSET` and `GTID_SUBTRACT` can be used to check that a replica has applied at least every transaction that a source has applied.

To perform this check with `GTID_SUBSET`, execute the following statement on the replica:

```
SELECT GTID_SUBSET(source_gtid_executed, replica_gtid_executed)
```

If this returns 0 (false), some GTIDs in *`source_gtid_executed`* are not present in *`replica_gtid_executed`*, so the source has applied some transactions that the replica has not applied, and the replica is therefore not up to date.

To perform the check with `GTID_SUBTRACT`, execute the following statement on the replica:

```
SELECT GTID_SUBTRACT(source_gtid_executed, replica_gtid_executed)
```

This statement returns any GTIDs that are in *`source_gtid_executed`* but not in *`replica_gtid_executed`*. If any GTIDs are returned, the source has applied some transactions that the replica has not applied, and the replica is therefore not up to date.

Example 17.2 Backup and restore scenario

The stored functions `GTID_IS_EQUAL`, `GTID_IS_DISJOINT`, and `GTID_IS_DISJOINT_UNION` could be used to verify backup and restore operations involving multiple databases and servers. In this example scenario, `server1` contains database `db1`, and `server2` contains database `db2`. The goal is to copy database `db2` to `server1`, and the result on `server1` should be the union of the two databases. The procedure used is to back up `server2` using `mysqlpump` or `mysqldump`, then restore this backup on `server1`.

Provided the backup program's option `--set-gtid-purged` was set to `ON` or the default of `AUTO`, the program's output contains a `SET @@GLOBAL.gtid_purged` statement which adds the `gtid_executed` set from `server2` to the `gtid_purged` set on `server1`. The `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 `db2` is copied to `server1`, the GTIDs of the transactions committed on `server2`, which are not in the binary log files on `server1`, must be added to `gtid_purged` for `server1` 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`. Having identical GTIDs present on both servers causes errors when copying database `db2` to `server1`. To check, on `server1`, extract and store `gtid_purged` 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 `gtid_executed` cause it to fail.

Example 17.3 Selecting the most up-to-date replica for manual failover

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

On each replica, 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 replica to see which one has the most up-to-date record of transactions, and use this replica as the new source.

Example 17.4 Checking for extraneous transactions on a replica

The stored function `GTID_SUBTRACT_UUID` could be used to check whether a replica has received transactions that did not originate from its designated source or sources. 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 replica with a single source, issue the following statement, giving the identifier of the originating source, which is normally the same as `server_uuid`:

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

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

For a replica in a multisource topology, include the server UUID of each source in the function call, like this:

```
SELECT GTID_SUBTRACT_UUID(GTID_SUBTRACT_UUID(@@GLOBAL.gtid_executed,
                                             server_uuid_of_source_1),
                                             server_uuid_of_source_2);
```

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

Example 17.5 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 listed in `gtid_executed` from this server use 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 17.6 Validating an additional replica in multisource replication

The stored function `GTID_INTERSECTION_WITH_UUID()` could be used to find out if a replica attached to a multisource replication setup has applied all the transactions originating from one particular source. In this scenario, `source1` and `source2` are both sources and replicas and replicate to each other. `source2` also has its own replica. The replica also receives and applies transactions from `source1` if `source2` is configured with `log_replica_updates=ON` or `log_slave_updates=ON`, but it does not do so if `source2` uses `log_replica_updates=OFF` or `log_slave_updates=OFF`. Whatever the case, we currently only want to find out if the replica is up to date with `source2`. In this situation, the stored function `GTID_INTERSECTION_WITH_UUID` can be used to identify the transactions that `source2` originated, discarding the transactions that `source2` has replicated from `source1`. The built-in function `GTID_SUBSET` can then be used to compare the result to the `gtid_executed` set on the replica. If the replica is up to date with `source2`, the `gtid_executed` set on the replica contains all the transactions in the intersection set (the transactions that originated from `source2`).

To carry out this check, store the values of `gtid_executed` and the server UUID from `source2` and the value of `gtid_executed` from the replica into user variables as follows:

```
source2> SELECT @@GLOBAL.gtid_executed INTO @source2_gtid_executed;

source2> SELECT @@GLOBAL.server_uuid INTO @source2_server_uuid;

replica> SELECT @@GLOBAL.gtid_executed INTO @replica_gtid_executed;
```

Then use `GTID_INTERSECTION_WITH_UUID` and `GTID_SUBSET` with these variables as input, as follows:

```
SELECT GTID_SUBSET(
          GTID_INTERSECTION_WITH_UUID(@source2_gtid_executed,
                                      @source2_server_uuid),
                                      @replica_gtid_executed);
```

The server identifier from `source2` (`@source2_server_uuid`) is used with `GTID_INTERSECTION_WITH_UUID` to identify and return only those GTIDs from the set of GTIDs that originated on `source2`, omitting those that originated on `source1`. The resulting GTID set is then compared with the set of all executed GTIDs on the replica, using `GTID_SUBSET()`. If this statement returns nonzero (true), all the identified GTIDs from `source2` (the first set input) are also found in `gtid_executed` from the replica, meaning that the replica has received and executed all the transactions that originated from `source2`.

17.1.4 Changing GTID Mode on Online Servers

This section describes how to change the mode of replication from and to GTID mode without having to take the server offline.

17.1.4.1 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 GTID 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.

Transactions in a relay log that was received from a source running a previous version of MySQL may not be preceded by any particular event at all, but after being replayed and logged in the replica'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 Section 5.1.9.1, “System Variable Privileges”. In MySQL 5.6 and earlier, 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 `ON` or `OFF`, 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. 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 source with `gtid_mode=ON` could be replicating to a replica 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`.

The fields related to GTIDs 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 REPLICA STATUS` (or before MySQL 8.0.22, `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 source using `gtid_mode=ON` provides the ability to use GTID auto-positioning, configured using the `SOURCE_AUTO_POSITION` of the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23), or the `MASTER_AUTO_POSITION` option of the `CHANGE MASTER TO` statement (before MySQL 8.0.23). 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. It is strongly recommended to ensure there are no anonymous transactions remaining in the topology before enabling auto-positioning, see Section 17.1.4.2, “Enabling GTID Transactions Online”.

The valid combinations of `gtid_mode` and auto-positioning on source and replica are shown in the following table, where the source's `gtid_mode` is shown on the horizontal and the replica's `gtid_mode` is on the vertical. The meaning of each entry is as follows:

- `Y`: the `gtid_mode` of source and replica is compatible
- `N`: the `gtid_mode` of source and replica is not compatible
- `*`: auto-positioning can be used with this combination

Table 17.1 Valid Combinations of Source and Replica gtid_mode

`gtid_mode`	**Source `OFF`**	**Source `OFF_PERMISSIVE`**	**Source `ON_PERMISSIVE`**	**Source `ON`**
Replica `OFF`	Y	Y	N	N
Replica `OFF_PERMISSIVE`	Y	Y	Y	Y*
Replica `ON_PERMISSIVE`	Y	Y	Y	Y*
Replica `ON`	N	N	Y	Y*

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`. The meaning of each entry is as follows:

- `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.

Table 17.2 Valid Combinations of gtid_mode and gtid_next

	`gtid_next` AUTOMATIC binary log on	**`gtid_next` AUTOMATIC binary log off**	**`gtid_next` ANONYMOUS**	**`gtid_next` UUID:NUMBER**
`gtid_mode OFF`	ANONYMOUS	ANONYMOUS	ANONYMOUS	Error
`gtid_mode OFF_PERMISSIVE`	ANONYMOUS	ANONYMOUS	ANONYMOUS	UUID:NUMBER
`gtid_mode ON_PERMISSIVE`	New GTID	ANONYMOUS	ANONYMOUS	UUID:NUMBER
`gtid_mode ON`	New GTID	ANONYMOUS	Error	UUID: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.

17.1.4.2 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.

From MySQL 8.0.23, you can set up replication channels to assign a GTID to replicated transactions that do not already have one. This feature enables replication from a source server that does not use GTID-based replication, to a replica that does. If it is possible to enable GTIDs on the replication source server, as described in this procedure, use this approach instead. Assigning GTIDs is designed for replication source servers where you cannot enable GTIDs. For more information on this option, see Section 17.1.3.6, “Replication From a Source Without GTIDs to a Replica With GTIDs”.

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 17.1.4.3, “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:

   ```
   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:

   ```
   SET @@GLOBAL.ENFORCE_GTID_CONSISTENCY = ON;
   ```

3. On each server, execute:

   ```
   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:

   ```
   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 replica, 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 17.1.4.4, “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 replica. 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'];

Or from MySQL 8.0.22 / 8.0.23:
STOP REPLICA [FOR CHANNEL 'channel'];
CHANGE REPLICATION SOURCE TO SOURCE_AUTO_POSITION = 1 [FOR CHANNEL 'channel'];
START REPLICA [FOR CHANNEL 'channel'];
```

17.1.4.3 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:

- *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`.
- The `--replicate-same-server-id` option is not set on any server. You cannot disable GTID transactions if this option is set together with the `--log-slave-updates` option (which is the default) and binary logging is enabled (which is also the default). Without GTIDs, this combination of options causes infinite loops in circular replication.

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

```
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'];

Or from MySQL 8.0.22 / 8.0.23:
STOP REPLICA [FOR CHANNEL 'channel'];
CHANGE REPLICATION SOURCE TO SOURCE_AUTO_POSITION = 0, SOURCE_LOG_FILE = file, \
SOURCE_LOG_POS = position [FOR CHANNEL 'channel'];
START REPLICA [FOR CHANNEL 'channel'];
```

2. On each server, execute:

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

3. On each server, execute:

```
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:

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

   On a replica, 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 replicas. See Section 17.1.4.4, “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:

```
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.

17.1.4.4 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 replica 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 source with one or more replicas, do the following:

1. On the source, execute:

   ```
   SHOW MASTER STATUS;
   ```

   Note down the values in the `File` and `Position` column.

2. On every replica, use the file and position information from the source to execute:

   ```
   SELECT MASTER_POS_WAIT(file, position);

   Or from MySQL 8.0.26:
   SELECT SOURCE_POS_WAIT(file, position);
   ```

If you have a source and multiple levels of replicas, or in other words you have replicas of replicas, repeat step 2 on each level, starting from the source, then all the direct replicas, then all the replicas of replicas, and so on.

If you use a circular replication topology where multiple servers may have write clients, perform step 2 for each source-replica 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.

17.1.5 MySQL Multi-Source Replication

MySQL multi-source replication enables a replica to receive transactions from multiple immediate sources in parallel. In a multi-source replication topology, a replica creates a replication channel for each source that it should receive transactions from. For more information on how replication channels function, see Section 17.2.2, “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 replica must replicate from a different source. You cannot set up multiple replication channels from a single replica to a single source. This is because the server IDs of replicas must be unique in a replication topology. The source distinguishes replicas only by their server IDs, not by the names of the replication channels, so it cannot recognize different replication channels from the same replica.

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

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

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

17.1.5.1 Configuring Multi-Source Replication

A multi-source replication topology requires at least two sources and one replica configured. In these tutorials, we assume that you have two sources `source1` and `source2`, and a replica `replicahost`. The replica replicates one database from each of the sources, `db1` from `source1` and `db2` from `source2`.

Sources in a multi-source replication topology can be configured to use either GTID-based replication, or binary log position-based replication. See Section 17.1.3.4, “Setting Up Replication Using GTIDs” for how to configure a source using GTID-based replication. See Section 17.1.2.1, “Setting the Replication Source Configuration” for how to configure a source using file position based replication.

Replicas in a multi-source replication topology require `TABLE` repositories for the replica's connection metadata repository and applier metadata repository, which are the default in MySQL 8.0. Multi-source replication is not compatible with the deprecated alternative file repositories.

Create a suitable user account on all the sources that the replica can use to connect. You can use the same account on all the sources, 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 replica `replicahost`, use the `mysql` client to issue these statements on each of the sources:

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

For more details, and important information on the default authentication plugin for new users from MySQL 8.0, see Section 17.1.2.3, “Creating a User for Replication”.

17.1.5.2 Provisioning a Multi-Source Replica for GTID-Based Replication

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

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 source to the replica, and the replica requires this information. However, for any case more complex than provisioning one new, empty replica from one source, you need to check what effect the statement has in the version of MySQL used by the replica, and handle the statement accordingly. The following guidance summarizes suitable actions, but for more details, see the `mysqldump` documentation.

The behavior of the `SET @@GLOBAL.gtid_purged` statement written by `mysqldump` is different in releases from MySQL 8.0 compared to MySQL 5.6 and 5.7. In MySQL 5.6 and 5.7, the statement replaces the value of `gtid_purged` on the replica, and also in those releases that value can only be changed when the replica'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 cannot apply a second or subsequent dump file including this statement. Also note that for MySQL 5.6 and 5.7, this limitation means all the dump files from the sources must be applied in a single operation on a replica with an empty `gtid_executed` set. You can clear a replica's GTID execution history by issuing `RESET MASTER` on the replica, but if you have other, wanted transactions with GTIDs on the replica, choose an alternative method of provisioning from those described in Section 17.1.3.5, “Using GTIDs for Failover and Scaleout”.

From MySQL 8.0, the `SET @@GLOBAL.gtid_purged` statement adds the GTID set from the dump file to the existing `gtid_purged` set on the replica. The statement can therefore potentially be left in the dump output when you replay the dump files on the replica, and the dump files can be replayed at different times. However, it is important to note that the value that is included by `mysqldump` for the `SET @@GLOBAL.gtid_purged` statement includes the GTIDs of all transactions in the `gtid_executed` set on the source, even those that changed suppressed parts of the database, or other databases on the server that were not included in a partial dump. If you replay a second or subsequent dump file on the replica that contains any of the same GTIDs (for example, another partial dump from the same source, or a dump from another source that has overlapping transactions), any `SET @@GLOBAL.gtid_purged` statement in the second dump file fails, and must therefore be removed from the dump output.

For sources from MySQL 8.0.17, as an alternative to removing the `SET @@GLOBAL.gtid_purged` statement, you may set `mysqldump`'s `--set-gtid-purged` option to `COMMENTED` to include the statement but commented out, so that it is not actioned when you load the dump file. If you are provisioning the replica with two partial dumps from the same source, and the GTID set in the second dump is the same as the first (so no new transactions have been executed on the source 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.

In the following provisioning example, we assume that the `SET @@GLOBAL.gtid_purged` statement cannot be left in the dump output, and must be removed from the files and handled manually. We also assume that there are no wanted transactions with GTIDs on the replica before provisioning starts.

1. To create dump files for a database named `db1` on `source1` and a database named `db2` on `source2`, run `mysqldump` for `source1` as follows:

   ```
   mysqldump -u<user> -p<password> --single-transaction --triggers --routines --set-gtid-purged=ON --databa
   ```

   Then run `mysqldump` for `source2` as follows:

```
mysqldump -u<user> -p<password> --single-transaction --triggers --routines --set-gtid-purged=ON --da
```

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

   From MySQL 8.0, where the format has changed, you can extract the value like this:

```
cat dumpM1.sql | grep GTID_PURGED | perl -p0 -e 's#/\*.*?\*/##sg' | cut -f2 -d'=' | cut -f2 -d$'\''
cat dumpM2.sql | grep GTID_PURGED | perl -p0 -e 's#/\*.*?\*/##sg' | cut -f2 -d'=' | cut -f2 -d$'\''
```

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

```
source1:   2174B383-5441-11E8-B90A-C80AA9429562:1-1029
source2:   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 replica. For example:

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

5. On the replica, 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 replica). 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-11E8-8
```

   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 17.1.3.8, “Stored Function Examples to Manipulate GTIDs” to check this beforehand and to calculate the union of all the GTID sets.

17.1.5.3 Adding GTID-Based Sources to a Multi-Source Replica

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

Use the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) to configure a replication channel for each source on the replica (see Section 17.2.2, “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 source (see Section 17.1.3.3, “GTID Auto-Positioning”). The `SOURCE_AUTO_POSITION` | `MASTER_AUTO_POSITION` option is set to specify the use of auto-positioning.

For example, to add `source1` and `source2` as sources to the replica, use the `mysql` client to issue the statement twice on the replica, like this:

```
mysql> CHANGE MASTER TO MASTER_HOST="source1", MASTER_USER="ted", \
MASTER_PASSWORD="password", MASTER_AUTO_POSITION=1 FOR CHANNEL "source_1";
mysql> CHANGE MASTER TO MASTER_HOST="source2", MASTER_USER="ted", \
MASTER_PASSWORD="password", MASTER_AUTO_POSITION=1 FOR CHANNEL "source_2";

Or from MySQL 8.0.23:
```

```
mysql> CHANGE REPLICATION SOURCE TO SOURCE_HOST="source1", SOURCE_USER="ted", \
SOURCE_PASSWORD="password", SOURCE_AUTO_POSITION=1 FOR CHANNEL "source_1";
mysql> CHANGE REPLICATION SOURCE TO SOURCE_HOST="source2", SOURCE_USER="ted", \
SOURCE_PASSWORD="password", SOURCE_AUTO_POSITION=1 FOR CHANNEL "source_2";
```

To make the replica replicate only database `db1` from `source1`, and only database `db2` from `source2`, use the `mysql` client to issue the `CHANGE REPLICATION FILTER` statement for each channel, like this:

```
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db1.%') FOR CHANNEL "source_1";
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db2.%') FOR CHANNEL "source_2";
```

For the full syntax of the `CHANGE REPLICATION FILTER` statement and other available options, see Section 13.4.2.2, “CHANGE REPLICATION FILTER Statement”.

17.1.5.4 Adding Binary Log Based Replication Sources to a Multi-Source Replica

These steps assume that binary logging is enabled on the source (which is the default), the replica is using `TABLE` based replication applier metadata repositories (which is the default in MySQL 8.0), and that you have enabled a replication user and noted the current binary log file name and position.

Use the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23) to configure a replication channel for each source on the replica (see Section 17.2.2, “Replication Channels”). The `FOR CHANNEL` clause is used to specify the channel. For example, to add `source1` and `source2` as sources to the replica, use the `mysql` client to issue the statement twice on the replica, like this:

```
mysql> CHANGE MASTER TO MASTER_HOST="source1", MASTER_USER="ted", MASTER_PASSWORD="password", \
MASTER_LOG_FILE='source1-bin.000006', MASTER_LOG_POS=628 FOR CHANNEL "source_1";
mysql> CHANGE MASTER TO MASTER_HOST="source2", MASTER_USER="ted", MASTER_PASSWORD="password", \
MASTER_LOG_FILE='source2-bin.000018', MASTER_LOG_POS=104 FOR CHANNEL "source_2";

Or from MySQL 8.0.23:
mysql> CHANGE REPLICATION SOURCE TO SOURCE_HOST="source1", SOURCE_USER="ted", SOURCE_PASSWORD="password", \
SOURCE_LOG_FILE='source1-bin.000006', SOURCE_LOG_POS=628 FOR CHANNEL "source_1";
mysql> CHANGE REPLICATION SOURCE TO SOURCE_HOST="source2", SOURCE_USER="ted", SOURCE_PASSWORD="password", \
SOURCE_LOG_FILE='source2-bin.000018', SOURCE_LOG_POS=104 FOR CHANNEL "source_2";
```

To make the replica replicate only database `db1` from `source1`, and only database `db2` from `source2`, use the `mysql` client to issue the `CHANGE REPLICATION FILTER` statement for each channel, like this:

```
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db1.%') FOR CHANNEL "source_1";
mysql> CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE = ('db2.%') FOR CHANNEL "source_2";
```

For the full syntax of the `CHANGE REPLICATION FILTER` statement and other available options, see Section 13.4.2.2, “CHANGE REPLICATION FILTER Statement”.

17.1.5.5 Starting Multi-Source Replicas

Once you have added channels for all of the replication sources, issue a `START REPLICA` (or before MySQL 8.0.22, `START SLAVE`) statement to start replication. When you have enabled multiple channels on a replica, 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:

```
mysql> START SLAVE FOR CHANNEL "source_1";
mysql> START SLAVE FOR CHANNEL "source_2";
Or from MySQL 8.0.22:
mysql> START REPLICA FOR CHANNEL "source_1";
mysql> START REPLICA FOR CHANNEL "source_2";
```

For the full syntax of the `START REPLICA` command and other available options, see Section 13.4.2.8, “START REPLICA Statement”.

To verify that both channels have started and are operating correctly, you can issue `SHOW REPLICA STATUS` statements on the replica, for example:

```
mysql> SHOW SLAVE STATUS FOR CHANNEL "source_1"\G
mysql> SHOW SLAVE STATUS FOR CHANNEL "source_2"\G
Or from MySQL 8.0.22:
mysql> SHOW REPLICA STATUS FOR CHANNEL "source_1"\G
mysql> SHOW REPLICA STATUS FOR CHANNEL "source_2"\G
```

17.1.5.6 Stopping Multi-Source Replicas

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

- To stop all currently configured replication channels:

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

- To stop only a named channel, use a `FOR CHANNEL channel` clause:

```
mysql> STOP SLAVE FOR CHANNEL "source_1";
Or from MySQL 8.0.22:
mysql> STOP REPLICA FOR CHANNEL "source_1";
```

For the full syntax of the `STOP REPLICA` command and other available options, see Section 13.4.2.10, “STOP REPLICA Statement”.

17.1.5.7 Resetting Multi-Source Replicas

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

- To reset all currently configured replication channels:

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

- To reset only a named channel, use a `FOR CHANNEL channel` clause:

```
mysql> RESET SLAVE FOR CHANNEL "source_1";
Or from MySQL 8.0.22:
mysql> RESET REPLICA FOR CHANNEL "source_1";
```

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

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

For the full syntax of the `RESET REPLICA` command and other available options, see Section 13.4.2.5, “RESET REPLICA Statement”.

17.1.5.8 Monitoring Multi-Source Replication

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 Section 27.12.11, “Performance Schema Replication Tables”.

- Using `SHOW REPLICA STATUS FOR CHANNEL channel`. By default, if the `FOR CHANNEL channel` clause is not used, this statement shows the replica 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 migrated to the replication performance tables. Using a `SHOW VARIABLES` statement in a topology with multiple channels shows the status of only the default channel.

The error codes and messages that are issued when multi-source replication is enabled specify the channel that generated the error.

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: source_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: source_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 `source_1` and `source_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='source_1'\G
*************************** 1. row ***************************
CHANNEL_NAME: source_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 Section 27.12.11, “Performance Schema Replication Tables”.

17.1.6 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 sources and replicas 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.

Command-Line Format	`--server-id=#`
System Variable	`server_id`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1`
Minimum Value	`0`
Maximum Value	`4294967295`

This variable specifies the server ID. `server_id` is set to 1 by default. The server can be started with this default ID, but when binary logging is enabled, an informational message is issued if you did not set `server_id` explicitly to specify a server ID.

For servers that are used in a replication topology, you must specify a unique server ID for each replication server, 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 source or replica in the replication topology. For additional information, see Section 17.1.6.2, “Replication Source Options and Variables”, and Section 17.1.6.3, “Replica Server Options and Variables”.

If the server ID is set to 0, binary logging takes place, but a source with a server ID of 0 refuses any connections from replicas, and a replica with a server ID of 0 refuses to connect to a source. 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 replica.

For more information, see Section 17.1.2.2, “Setting the Replica Configuration”.

`server_uuid`

The MySQL server generates a true UUID in addition to the default or user-supplied server ID set in the `server_id` system variable. This is available as the global, read-only variable `server_uuid`.

Note

The presence of the `server_uuid` system variable 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.

System Variable	server_uuid
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	String

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. 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
```

Important

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

When using MySQL replication, sources and replicas know each other's UUIDs. The value of a replica's UUID can be seen in the output of SHOW REPLICAS (or before MySQL 8.0.22, SHOW SLAVE HOSTS). Once START REPLICA has been executed, the value of the source's UUID is available on the replica in the output of SHOW REPLICA STATUS. As of 8.0.22, the keyword SLAVE was replaced by REPLICA.

Note

Issuing a STOP REPLICA or RESET REPLICA statement does *not* reset the source's UUID as used on the replica.

A server's server_uuid is also used in GTIDs for transactions originating on that server. For more information, see Section 17.1.3, “Replication with Global Transaction Identifiers”.

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

- No source having the expected server_uuid exists.
- The source's server_uuid has changed, although no CHANGE REPLICATION SOURCE TO | CHANGE MASTER TO statement has ever been executed.

17.1.6.1 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 source servers and replicas. Section 17.1.6.2, “Replication Source Options and Variables” provides more detailed information about options and variables relating to replication source servers. For more information about options and variables relating to replicas, see Section 17.1.6.3, “Replica Server 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.
- `Com_change_master`: Count of CHANGE REPLICATION SOURCE TO and CHANGE MASTER TO statements.
- `Com_change_replication_source`: Count of CHANGE REPLICATION SOURCE TO and CHANGE MASTER TO statements.
- `Com_replica_start`: Count of START REPLICA and START SLAVE statements.
- `Com_replica_stop`: Count of STOP REPLICA and STOP SLAVE statements.
- `Com_show_master_status`: Count of SHOW MASTER STATUS statements.
- `Com_show_replica_status`: Count of SHOW REPLICA STATUS and SHOW SLAVE STATUS statements.
- `Com_show_replicas`: Count of SHOW REPLICAS and SHOW SLAVE HOSTS statements.
- `Com_show_slave_hosts`: Count of SHOW REPLICAS and SHOW SLAVE HOSTS statements.
- `Com_show_slave_status`: Count of SHOW REPLICA STATUS and SHOW SLAVE STATUS statements.
- `Com_slave_start`: Count of START REPLICA and START SLAVE statements.
- `Com_slave_stop`: Count of STOP REPLICA and 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 transactionally safe manner.
- `expire_logs_days`: Purge binary logs after this many days.
- `gtid_executed`: Global: All GTIDs in 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 logs can contain.
- `gtid_next`: Specifies GTID for next statement to execute; see documentation for details.
- `gtid_owned`: Set of GTIDs owned by this client (session), or by all clients, together with thread ID of owner (global). Read-only.
- `gtid_purged`: Set of all GTIDs that have been purged from binary log.
- `immediate_server_version`: MySQL Server release number of server which is immediate replication source.
- `init_replica`: Statements that are executed when replica connects to source.
- `init_slave`: Statements that are executed when replica connects to source.

- `log_bin_trust_function_creators`: If equal to 0 (default), then when --log-bin is used, stored function creation is allowed only to users having SUPER privilege and only if function created does not break binary logging.
- `log_statements_unsafe_for_binlog`: Disables error 1592 warnings being written to error log.
- `master-info-file`: Location and name of file that remembers source and where I/O replication thread is in source's binary log.
- `master-retry-count`: Number of tries replica makes to connect to source before giving up.
- `master_info_repository`: Whether to write connection metadata repository, containing source information and replication I/O thread location in source's binary log, to 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 value of max_binlog_size.
- `original_commit_timestamp`: Time when transaction was committed on original source.
- `original_server_version`: MySQL Server release number of server on which transaction was originally committed.
- `relay_log`: Location and base name to use for relay logs.
- `relay_log_basename`: Complete path to relay log, including file name.
- `relay_log_index`: Location and name to use for file that keeps list of last relay logs.
- `relay_log_info_file`: File name for applier metadata repository in which replica records information about relay logs.
- `relay_log_info_repository`: Whether to write location of replication SQL thread in relay logs to file or table.
- `relay_log_purge`: Determines whether relay logs are purged.
- `relay_log_recovery`: Whether automatic recovery of relay log files from source at startup is enabled; must be enabled for crash-safe replica.
- `relay_log_space_limit`: Maximum space to use for all relay logs.
- `replica_checkpoint_group`: Maximum number of transactions processed by multithreaded replica before checkpoint operation is called to update progress status. Not supported by NDB Cluster.
- `replica_checkpoint_period`: Update progress status of multithreaded replica and flush relay log info to disk after this number of milliseconds. Not supported by NDB Cluster.
- `replica_compressed_protocol`: Use compression of source/replica protocol.
- `replica_exec_mode`: Allows for switching replication thread between IDEMPOTENT mode (key and some other errors suppressed) and STRICT mode; STRICT mode is default, except for NDB Cluster, where IDEMPOTENT is always used.
- `replica_load_tmpdir`: Location where replica should put its temporary files when replicating LOAD DATA statements.
- `replica_max_allowed_packet`: Maximum size, in bytes, of packet that can be sent from replication source server to replica; overrides max_allowed_packet.
- `replica_net_timeout`: Number of seconds to wait for more data from source/replica connection before aborting read.
- `Replica_open_temp_tables`: Number of temporary tables that replication SQL thread currently has open.

- `replica_parallel_type`: Tells replica to use timestamp information (LOGICAL_CLOCK) or database partitioning (DATABASE) to parallelize transactions.
- `replica_parallel_workers`: Number of applier threads for executing replication transactions in parallel; 0 or 1 disables replica multithreading. NDB Cluster: see documentation.
- `replica_pending_jobs_size_max`: Maximum size of replica worker queues holding events not yet applied.
- `replica_preserve_commit_order`: Ensures that all commits by replica workers happen in same order as on source to maintain consistency when using parallel applier threads.
- `Replica_rows_last_search_algorithm_used`: Search algorithm most recently used by this replica to locate rows for row-based replication (index, table, or hash scan).
- `replica_skip_errors`: Tells replication thread to continue replication when query returns error from provided list.
- `replica_transaction_retries`: Number of times replication SQL thread retries transaction in case it failed with deadlock or elapsed lock wait timeout, before giving up and stopping.
- `replica_type_conversions`: Controls type conversion mode on replica. Value is list of zero or more elements from this list: ALL_LOSSY, ALL_NON_LOSSY. Set to empty string to disallow type conversions between source and replica.
- `replicate-do-db`: Tells replication SQL thread to restrict replication to specified database.
- `replicate-do-table`: Tells replication SQL thread to restrict replication to specified table.
- `replicate-ignore-db`: Tells replication SQL thread not to replicate to specified database.
- `replicate-ignore-table`: Tells replication SQL thread not to replicate to specified table.
- `replicate-rewrite-db`: Updates to database with different name from original.
- `replicate-same-server-id`: In replication, if enabled, do not skip events having our server id.
- `replicate-wild-do-table`: Tells replication SQL thread to restrict replication to tables that match specified wildcard pattern.
- `replicate-wild-ignore-table`: Tells replication SQL thread not to replicate to tables that match given wildcard pattern.
- `replication_optimize_for_static_plugin_config`: Shared locks for semisynchronous replication.
- `replication_sender_observe_commit_only`: Limited callbacks for semisynchronous replication.
- `report_host`: Host name or IP of replica to be reported to source during replica registration.
- `report_password`: Arbitrary password which replica server should report to source; not same as password for replication user account.
- `report_port`: Port for connecting to replica reported to source during replica registration.
- `report_user`: Arbitrary user name which replica server should report to source; not same as name used for replication user account.
- `rpl_read_size`: Set minimum amount of data in bytes which is read from binary log files and relay log files.
- `Rpl_semi_sync_master_clients`: Number of semisynchronous replicas.

- `rpl_semi_sync_master_enabled`: Whether semisynchronous replication is enabled on source.
- `Rpl_semi_sync_master_net_avg_wait_time`: Average time source has waited for replies from replica.
- `Rpl_semi_sync_master_net_wait_time`: Total time source has waited for replies from replica.
- `Rpl_semi_sync_master_net_waits`: Total number of times source waited for replies from replica.
- `Rpl_semi_sync_master_no_times`: Number of times source 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 source.
- `Rpl_semi_sync_master_timefunc_failures`: Number of times source failed when calling time functions.
- `rpl_semi_sync_master_timeout`: Number of milliseconds to wait for replica acknowledgment.
- `rpl_semi_sync_master_trace_level`: Semisynchronous replication debug trace level on source.
- `Rpl_semi_sync_master_tx_avg_wait_time`: Average time source waited for each transaction.
- `Rpl_semi_sync_master_tx_wait_time`: Total time source waited for transactions.
- `Rpl_semi_sync_master_tx_waits`: Total number of times source waited for transactions.
- `rpl_semi_sync_master_wait_for_slave_count`: Number of replica acknowledgments source must receive per transaction before proceeding.
- `rpl_semi_sync_master_wait_no_slave`: Whether source waits for timeout even with no replicas.
- `rpl_semi_sync_master_wait_point`: Wait point for replica transaction receipt acknowledgment.
- `Rpl_semi_sync_master_wait_pos_backtraverse`: Total number of times source has waited for event with binary coordinates lower than events waited for previously.
- `Rpl_semi_sync_master_wait_sessions`: Number of sessions currently waiting for replica replies.
- `Rpl_semi_sync_master_yes_tx`: Number of commits acknowledged successfully.
- `rpl_semi_sync_replica_enabled`: Whether semisynchronous replication is enabled on replica.
- `Rpl_semi_sync_replica_status`: Whether semisynchronous replication is operational on replica.
- `rpl_semi_sync_replica_trace_level`: Semisynchronous replication debug trace level on replica.
- `rpl_semi_sync_slave_enabled`: Whether semisynchronous replication is enabled on replica.
- `Rpl_semi_sync_slave_status`: Whether semisynchronous replication is operational on replica.
- `rpl_semi_sync_slave_trace_level`: Semisynchronous replication debug trace level on replica.
- `Rpl_semi_sync_source_clients`: Number of semisynchronous replicas.
- `rpl_semi_sync_source_enabled`: Whether semisynchronous replication is enabled on source.

- `Rpl_semi_sync_source_net_avg_wait_time`: Average time source has waited for replies from replica.
- `Rpl_semi_sync_source_net_wait_time`: Total time source has waited for replies from replica.
- `Rpl_semi_sync_source_net_waits`: Total number of times source waited for replies from replica.
- `Rpl_semi_sync_source_no_times`: Number of times source turned off semisynchronous replication.
- `Rpl_semi_sync_source_no_tx`: Number of commits not acknowledged successfully.
- `Rpl_semi_sync_source_status`: Whether semisynchronous replication is operational on source.
- `Rpl_semi_sync_source_timefunc_failures`: Number of times source failed when calling time functions.
- `rpl_semi_sync_source_timeout`: Number of milliseconds to wait for replica acknowledgment.
- `rpl_semi_sync_source_trace_level`: Semisynchronous replication debug trace level on source.
- `Rpl_semi_sync_source_tx_avg_wait_time`: Average time source waited for each transaction.
- `Rpl_semi_sync_source_tx_wait_time`: Total time source waited for transactions.
- `Rpl_semi_sync_source_tx_waits`: Total number of times source waited for transactions.
- `rpl_semi_sync_source_wait_for_replica_count`: Number of replica acknowledgments source must receive per transaction before proceeding.
- `rpl_semi_sync_source_wait_no_replica`: Whether source waits for timeout even with no replicas.
- `rpl_semi_sync_source_wait_point`: Wait point for replica transaction receipt acknowledgment.
- `Rpl_semi_sync_source_wait_pos_backtraverse`: Total number of times source has waited for event with binary coordinates lower than events waited for previously.
- `Rpl_semi_sync_source_wait_sessions`: Number of sessions currently waiting for replica replies.
- `Rpl_semi_sync_source_yes_tx`: Number of commits acknowledged successfully.
- `rpl_stop_replica_timeout`: Number of seconds that STOP REPLICA waits before timing out.
- `rpl_stop_slave_timeout`: Number of seconds that STOP REPLICA or STOP SLAVE waits before timing out.
- `server_uuid`: Server's globally unique ID, automatically (re)generated at server start.
- `show-replica-auth-info`: Show user name and password in SHOW REPLICAS on this source.
- `show-slave-auth-info`: Show user name and password in SHOW REPLICAS and SHOW SLAVE HOSTS on this source.
- `skip-replica-start`: If set, replication is not autostarted when replica server starts.
- `skip-slave-start`: If set, replication is not autostarted when replica server starts.
- `slave-skip-errors`: Tells replication thread to continue replication when query returns error from provided list.

- `slave_checkpoint_group`: Maximum number of transactions processed by multithreaded replica before checkpoint operation is called to update progress status. Not supported by NDB Cluster.
- `slave_checkpoint_period`: Update progress status of multithreaded replica and flush relay log info to disk after this number of milliseconds. Not supported by NDB Cluster.
- `slave_compressed_protocol`: Use compression of source/replica protocol.
- `slave_exec_mode`: Allows for switching replication thread between IDEMPOTENT mode (key and some other errors suppressed) and STRICT mode; STRICT mode is default, except for NDB Cluster, where IDEMPOTENT is always used.
- `slave_load_tmpdir`: Location where replica should put its temporary files when replicating LOAD DATA statements.
- `slave_max_allowed_packet`: Maximum size, in bytes, of packet that can be sent from replication source server to replica; overrides max_allowed_packet.
- `slave_net_timeout`: Number of seconds to wait for more data from source/replica connection before aborting read.
- `Slave_open_temp_tables`: Number of temporary tables that replication SQL thread currently has open.
- `slave_parallel_type`: Tells replica to use timestamp information (LOGICAL_CLOCK) or database partioning (DATABASE) to parallelize transactions.
- `slave_parallel_workers`: Number of applier threads for executing replication transactions in parallel; 0 or 1 disables replica multithreading. NDB Cluster: see documentation.
- `slave_pending_jobs_size_max`: Maximum size of replica worker queues holding events not yet applied.
- `slave_preserve_commit_order`: Ensures that all commits by replica workers happen in same order as on source to maintain consistency when using parallel applier threads.
- `Slave_rows_last_search_algorithm_used`: Search algorithm most recently used by this replica to locate rows for row-based replication (index, table, or hash scan).
- `slave_rows_search_algorithms`: Determines search algorithms used for replica update batching. Any 2 or 3 from this list: INDEX_SEARCH, TABLE_SCAN, HASH_SCAN.
- `slave_transaction_retries`: Number of times replication SQL thread retries transaction in case it failed with deadlock or elapsed lock wait timeout, before giving up and stopping.
- `slave_type_conversions`: Controls type conversion mode on replica. Value is list of zero or more elements from this list: ALL_LOSSY, ALL_NON_LOSSY. Set to empty string to disallow type conversions between source and replica.
- `sql_log_bin`: Controls binary logging for current session.
- `sql_replica_skip_counter`: Number of events from source that replica should skip. Not compatible with GTID replication.
- `sql_slave_skip_counter`: Number of events from source that replica should skip. Not compatible with GTID replication.
- `sync_master_info`: Synchronize source information 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.

- `sync_source_info`: Synchronize source information after every #th event.
- `terminology_use_previous`: Use terminology before specified version where changes are incompatible.
- `transaction_write_set_extraction`: Defines algorithm used to hash writes extracted during transaction.

For a listing of all command-line options, system variables, and status variables used with `mysqld`, see Section 5.1.4, “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 17.1.6.4, “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 Section 5.4.4, “The Binary Log”.

- `binlog-checksum`: Enable/disable binary log checksums.
- `binlog-do-db`: Limits binary logging to specific databases.
- `binlog-ignore-db`: Tells source that updates to given database should not be written to binary log.
- `binlog-row-event-max-size`: Binary log max event size.
- `Binlog_cache_disk_use`: Number of transactions which used temporary file instead of binary log cache.
- `binlog_cache_size`: Size of cache to hold SQL statements for binary log during transaction.
- `Binlog_cache_use`: Number of transactions that used 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_encryption`: Enable encryption for binary log files and relay log files on this server.
- `binlog_error_action`: Controls what happens when server cannot write to binary log.
- `binlog_expire_logs_auto_purge`: Controls automatic purging of binary log files; can be overridden when enabled, by setting both binlog_expire_logs_seconds and expire_logs_days to 0.
- `binlog_expire_logs_seconds`: Purge binary logs after this many seconds.
- `binlog_format`: Specifies format of binary log.
- `binlog_group_commit_sync_delay`: Sets number of microseconds to wait before synchronizing transactions to disk.
- `binlog_group_commit_sync_no_delay_count`: Sets maximum number of transactions to wait for before aborting current delay specified by binlog_group_commit_sync_delay.
- `binlog_gtid_simple_recovery`: Controls how binary logs are iterated during GTID recovery.
- `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_rotate_encryption_master_key_at_startup`: Rotate binary log master key at server startup.

- `binlog_row_image`: Use full or minimal images when logging row changes.
- `binlog_row_metadata`: Whether to record all or only minimal table related metadata to binary log when using row-based logging.
- `binlog_row_value_options`: Enables binary logging of partial JSON updates for row-based replication.
- `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 replicas/readers.
- `Binlog_stmt_cache_disk_use`: Number of nontransactional statements that used temporary file instead of binary log statement cache.
- `binlog_stmt_cache_size`: Size of cache to hold nontransactional statements for binary log during transaction.
- `Binlog_stmt_cache_use`: Number of statements that used temporary binary log statement cache.
- `binlog_transaction_compression`: Enable compression for transaction payloads in binary log files.
- `binlog_transaction_compression_level_zstd`: Compression level for transaction payloads in binary log files.
- `binlog_transaction_dependency_history_size`: Number of row hashes kept for looking up transaction that last updated some row.
- `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 replica's multithreaded applier.
- `Com_show_binlog_events`: Count of SHOW BINLOG EVENTS statements.
- `Com_show_binlogs`: Count of SHOW BINLOGS statements.
- `log-bin`: Base name for binary log files.
- `log-bin-index`: Name of binary log index file.
- `log_bin`: Whether binary log is enabled.
- `log_bin_basename`: Path and base name for binary log files.
- `log_bin_use_v1_row_events`: Whether server is using version 1 binary log row events.
- `log_replica_updates`: Whether replica should log updates performed by its replication SQL thread to its own binary log.
- `log_slave_updates`: Whether replica should log updates performed by its replication SQL thread to its own binary log.
- `master_verify_checksum`: Cause source to examine checksums when reading from 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 total size used to cache multi-statement transaction.
- `max_binlog_size`: Binary log is rotated automatically when size exceeds this value.
- `max_binlog_stmt_cache_size`: Can be used to restrict total size used to cache all nontransactional statements during transaction.

- `replica_sql_verify_checksum`: Cause replica to examine checksums when reading from relay log.
- `slave-sql-verify-checksum`: Cause replica to examine checksums when reading from relay log.
- `slave_sql_verify_checksum`: Cause replica to examine checksums when reading from relay log.
- `source_verify_checksum`: Cause source to examine checksums when reading from binary 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 Section 5.1.4, “Server Option, System Variable, and Status Variable Reference”.

17.1.6.2 Replication Source Options and Variables

This section describes the server options and system variables that you can use on replication source 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 source and each replica, 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 source or replica in the replication topology. Example: `server-id=3`.

For options used on the source for controlling binary logging, see Section 17.1.6.4, “Binary Logging Options and Variables”.

Startup Options for Replication Source Servers

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

- `--show-replica-auth-info`

Command-Line Format	`--show-replica-auth-info[={OFF|ON}]`
Introduced	8.0.26
Type	Boolean
Default Value	`OFF`

From MySQL 8.0.26, use `--show-replica-auth-info`, and before MySQL 8.0.26, use `--show-slave-auth-info`. Both options have the same effect. The options display replication user names and passwords in the output of `SHOW REPLICAS` (or before MySQL 8.0.22, `SHOW SLAVE HOSTS`) on the source for replicas started with the `--report-user` and `--report-password` options.

- `--show-slave-auth-info`

Command-Line Format	`--show-slave-auth-info[={OFF|ON}]`
Deprecated	8.0.26
Type	Boolean
Default Value	`OFF`

Use this option before MySQL 8.0.26 rather than `--show-replica-auth-info`. Both options have the same effect.

System Variables Used on Replication Source Servers

The following system variables are used for or by replication source servers:

- auto_increment_increment

Command-Line Format	--auto-increment-increment=#
System Variable	auto_increment_increment
Scope	Global, Session
Dynamic	Yes
SET_VAR Hint Applies	Yes
Type	Integer
Default Value	1
Minimum Value	1
Maximum Value	65535

auto_increment_increment and auto_increment_offset are intended for use with circular (source-to-source) 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.

As of MySQL 8.0.18, setting the session value of this system variable is no longer a restricted operation.

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. From MySQL 8.0, the system variables are also not modified when Group Replication is in single-primary mode, where only one server writes.

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 (0.01 sec)

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

mysql> SELECT col FROM autoinc1;
+-----+
| col |
+-----+
|   1 |
|  11 |
|  21 |
|  31 |
+-----+
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`:

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

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

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

mysql> SELECT col FROM autoinc2;
+-----+
| col |
+-----+
|   5 |
|  15 |
|  25 |
|  35 |
+-----+
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:

`auto_increment_offset` + *N* × `auto_increment_increment`

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

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

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

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

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* × 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 17.5.1.1, “Replication and AUTO_INCREMENT”.

- `auto_increment_offset`

Command-Line Format	`--auto-increment-offset=#`
System Variable	`auto_increment_offset`
Scope	Global, Session

Dynamic	Yes
SET_VAR Hint Applies	Yes
Type	Integer
Default Value	1
Minimum Value	1
Maximum Value	65535

This variable has a default value of 1. If it is left with its default value, and Group Replication is started on the server in multi-primary mode, 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.

As of MySQL 8.0.18, setting the session value of this system variable is no longer a restricted operation.

- immediate_server_version

Introduced	8.0.14
System Variable	immediate_server_version
Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	999999
Minimum Value	0
Maximum Value	999999

For internal use by replication. This session system variable holds the MySQL Server release number of the server that is the immediate source in a replication topology (for example, 80014 for a MySQL 8.0.14 server instance). If this immediate server is at a release that does not support the session system variable, the value of the variable is set to 0 (UNKNOWN_SERVER_VERSION).

The value of the variable is replicated from a source to a replica. With this information the replica can correctly process data originating from a source at an older release, by recognizing where syntax changes or semantic changes have occurred between the releases involved and handling these appropriately. The information can also be used in a Group Replication environment where one or more members of the replication group is at a newer release than the others. The value of the variable can be viewed in the binary log for each transaction (as part of the Gtid_log_event, or Anonymous_gtid_log_event if GTIDs are not in use on the server), and could be helpful in debugging cross-version replication issues.

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

- original_server_version

Introduced	8.0.14

System Variable	original_server_version
Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	999999
Minimum Value	0
Maximum Value	999999

For internal use by replication. This session system variable holds the MySQL Server release number of the server where a transaction was originally committed (for example, 80014 for a MySQL 8.0.14 server instance). If this original server is at a release that does not support the session system variable, the value of the variable is set to 0 (UNKNOWN_SERVER_VERSION). Note that when a release number is set by the original server, the value of the variable is reset to 0 if the immediate server or any other intervening server in the replication topology does not support the session system variable, and so does not replicate its value.

The value of the variable is set and used in the same ways as for the immediate_server_version system variable. If the value of the variable is the same as that for the immediate_server_version system variable, only the latter is recorded in the binary log, with an indicator that the original server version is the same.

In a Group Replication environment, view change log events, which are special transactions queued by each group member when a new member joins the group, are tagged with the server version of the group member queuing the transaction. This ensures that the server version of the original donor is known to the joining member. Because the view change log events queued for a particular view change have the same GTID on all members, for this case only, instances of the same GTID might have a different original server version.

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

- rpl_semi_sync_master_enabled

Command-Line Format	--rpl-semi-sync-master-enabled[={OFF|ON}]
System Variable	rpl_semi_sync_master_enabled
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

Controls whether semisynchronous replication is enabled on the source server. 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 source-side semisynchronous replication plugin is installed.

- rpl_semi_sync_master_timeout

Command-Line Format	--rpl-semi-sync-master-timeout=#

System Variable	rpl_semi_sync_master_timeout
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000
Minimum Value	0
Maximum Value	4294967295
Unit	milliseconds

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

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

- rpl_semi_sync_master_trace_level

Command-Line Format	--rpl-semi-sync-master-trace-level=#
System Variable	rpl_semi_sync_master_trace_level
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	32
Minimum Value	0
Maximum Value	4294967295

The semisynchronous replication debug trace level on the source server. 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 source-side semisynchronous replication plugin is installed.

- rpl_semi_sync_master_wait_for_slave_count

Command-Line Format	--rpl-semi-sync-master-wait-for-slave-count=#
System Variable	rpl_semi_sync_master_wait_for_slave_count
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	1

Minimum Value	1
Maximum Value	65535

The number of replica acknowledgments the source 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 replica 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 replicas must acknowledge receipt of the transaction before the timeout period configured by rpl_semi_sync_master_timeout for semisynchronous replication to proceed. If fewer replicas acknowledge receipt of the transaction during the timeout period, the source reverts to normal replication.

Note

This behavior also depends on rpl_semi_sync_master_wait_no_slave

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

- rpl_semi_sync_master_wait_no_slave

Command-Line Format	--rpl-semi-sync-master-wait-no-slave[={OFF|ON}]
System Variable	rpl_semi_sync_master_wait_no_slave
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

Controls whether the source waits for the timeout period configured by rpl_semi_sync_master_timeout to expire, even if the replica count drops to less than the number of replicas 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 replica count to drop to less than rpl_semi_sync_master_wait_for_slave_count during the timeout period. As long as enough replicas 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 replica 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 source reverts to normal replication.

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

- rpl_semi_sync_master_wait_point

Command-Line Format	--rpl-semi-sync-master-wait-point=value
System Variable	rpl_semi_sync_master_wait_point
Scope	Global
Dynamic	Yes

SET_VAR Hint Applies	No
Type	Enumeration
Default Value	AFTER_SYNC
Valid Values	AFTER_SYNC AFTER_COMMIT

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

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

The replication characteristics of these settings differ as follows:

- With AFTER_SYNC, all clients see the committed transaction at the same time: After it has been acknowledged by the replica and committed to the storage engine on the source. Thus, all clients see the same data on the source.

  In the event of source failure, all transactions committed on the source have been replicated to the replica (saved to its relay log). An unexpected exit of the source server and failover to the replica is lossless because the replica is up to date. Note, however, that the source 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 replica 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 replica acknowledgment. After the commit and before replica acknowledgment, other clients can see the committed transaction before the committing client.

  If something goes wrong such that the replica does not process the transaction, then in the event of an unexpected source server exit and failover to the replica, it is possible for such clients to see a loss of data relative to what they saw on the source.

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

With the addition of rpl_semi_sync_master_wait_point in MySQL 5.7, a version compatibility constraint was created because it increments the semisynchronous interface version: Servers for MySQL 5.7 and higher 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 and higher.

- rpl_semi_sync_source_enabled

Command-Line Format	--rpl-semi-sync-source-enabled[={OFF|ON}]
Introduced	8.0.26
System Variable	rpl_semi_sync_source_enabled

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

rpl_semi_sync_source_enabled is available when the rpl_semi_sync_source (semisync_source.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_master plugin (semisync_master.so library) was installed, rpl_semi_sync_master_enabled is available instead.

rpl_semi_sync_source_enabled controls whether semisynchronous replication is enabled on the source server. To enable or disable the plugin, set this variable to ON or OFF (or 1 or 0), respectively. The default is OFF.

- rpl_semi_sync_source_timeout

Command-Line Format	--rpl-semi-sync-source-timeout=#
Introduced	8.0.26
System Variable	rpl_semi_sync_source_timeout
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10000
Minimum Value	0
Maximum Value	4294967295
Unit	milliseconds

rpl_semi_sync_source_timeout is available when the rpl_semi_sync_source (semisync_source.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_master plugin (semisync_master.so library) was installed, rpl_semi_sync_master_timeout is available instead.

rpl_semi_sync_source_timeout controls how long the source waits on a commit for acknowledgment from a replica before timing out and reverting to asynchronous replication. The value is specified in milliseconds, and the default value is 10000 (10 seconds).

- rpl_semi_sync_source_trace_level

Command-Line Format	--rpl-semi-sync-source-trace-level=#
Introduced	8.0.26
System Variable	rpl_semi_sync_source_trace_level
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	32
Minimum Value	0
Maximum Value	4294967295

`rpl_semi_sync_source_trace_level` is available when the `rpl_semi_sync_source` (`semisync_source.so` library) plugin was installed on the replica to set up semisynchronous replication. If the `rpl_semi_sync_master` plugin (`semisync_master.so` library) was installed, `rpl_semi_sync_master_trace_level` is available instead.

`rpl_semi_sync_source_trace_level` specifies the semisynchronous replication debug trace level on the source server. 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)

- `rpl_semi_sync_source_wait_for_replica_count`

Command-Line Format	`--rpl-semi-sync-source-wait-for-replica-count=#`
Introduced	8.0.26
System Variable	`rpl_semi_sync_source_wait_for_replica_count`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1`
Minimum Value	`1`
Maximum Value	`65535`

`rpl_semi_sync_source_wait_for_replica_count` is available when the `rpl_semi_sync_source` (`semisync_source.so` library) plugin was installed on the replica to set up semisynchronous replication. If the `rpl_semi_sync_master` plugin (`semisync_master.so` library) was installed, `rpl_semi_sync_master_wait_for_slave_count` is available instead.

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

For example, if `rpl_semi_sync_source_wait_for_replica_count` is `2`, then 2 replicas must acknowledge receipt of the transaction before the timeout period configured by `rpl_semi_sync_source_timeout` for semisynchronous replication to proceed. If fewer replicas acknowledge receipt of the transaction during the timeout period, the source reverts to normal replication.

Note

This behavior also depends on `rpl_semi_sync_source_wait_no_replica`.

- `rpl_semi_sync_source_wait_no_replica`

Command-Line Format	`--rpl-semi-sync-source-wait-no-replica[={OFF|ON}]`
Introduced	8.0.26
System Variable	`rpl_semi_sync_source_wait_no_replica`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

`rpl_semi_sync_source_wait_no_replica` is available when the `rpl_semi_sync_source` (`semisync_source.so` library) plugin was installed on the replica to set up semisynchronous replication. If the `rpl_semi_sync_master` plugin (`semisync_master.so` library) was installed, `rpl_semi_sync_source_wait_no_replica` is available instead.

`rpl_semi_sync_source_wait_no_replica` controls whether the source waits for the timeout period configured by `rpl_semi_sync_source_timeout` to expire, even if the replica count drops to less than the number of replicas configured by `rpl_semi_sync_source_wait_for_replica_count` during the timeout period.

When the value of `rpl_semi_sync_source_wait_no_replica` is `ON` (the default), it is permissible for the replica count to drop to less than `rpl_semi_sync_source_wait_for_replica_count` during the timeout period. As long as enough replicas acknowledge the transaction before the timeout period expires, semisynchronous replication continues.

When the value of `rpl_semi_sync_source_wait_no_replica` is `OFF`, if the replica count drops to less than the number configured in `rpl_semi_sync_source_wait_for_replica_count` at any time during the timeout period configured by `rpl_semi_sync_source_timeout`, the source reverts to normal replication.

- `rpl_semi_sync_source_wait_point`

Command-Line Format	`--rpl-semi-sync-source-wait-point=value`
Introduced	8.0.26
System Variable	`rpl_semi_sync_source_wait_point`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`AFTER_SYNC`

Valid Values	`AFTER_SYNC`
	
	`AFTER_COMMIT`

`rpl_semi_sync_source_wait_point` is available when the `rpl_semi_sync_source` (`semisync_source.so` library) plugin was installed on the replica to set up semisynchronous

replication. If the `rpl_semi_sync_master` plugin (`semisync_master.so` library) was installed, `rpl_semi_sync_master_wait_point` is available instead.

`rpl_semi_sync_source_wait_point` controls the point at which a semisynchronous replication source server waits for replica acknowledgment of transaction receipt before returning a status to the client that committed the transaction. These values are permitted:

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

The replication characteristics of these settings differ as follows:

- With `AFTER_SYNC`, all clients see the committed transaction at the same time: After it has been acknowledged by the replica and committed to the storage engine on the source. Thus, all clients see the same data on the source.

  In the event of source failure, all transactions committed on the source have been replicated to the replica (saved to its relay log). An unexpected exit of the source server and failover to the replica is lossless because the replica is up to date. Note, however, that the source 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 replica 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 replica acknowledgment. After the commit and before replica acknowledgment, other clients can see the committed transaction before the committing client.

  If something goes wrong such that the replica does not process the transaction, then in the event of an unexpected source server exit and failover to the replica, it is possible for such clients to see a loss of data relative to what they saw on the source.

17.1.6.3 Replica Server Options and Variables

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

- Startup Options for Replica Servers
- System Variables Used on Replica Servers

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 REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). Specify system variable values using `SET`.

Server ID. On the source and each replica, 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 source or replica in the replication topology. Example `my.cnf` file:

```
[mysqld]
server-id=3
```

Startup Options for Replica Servers

This section explains startup options for controlling replica servers. Many of these options can be set while the server is running by using the `CHANGE REPLICATION SOURCE TO` statement (from MySQL 8.0.23) or `CHANGE MASTER TO` statement (before MySQL 8.0.23). Others, such as the `--replicate-*` options, can be set only when the replica server starts. Replication-related system variables are discussed later in this section.

- `--master-info-file=file_name`

Command-Line Format	`--master-info-file=file_name`
Deprecated	8.0.18
Type	File name
Default Value	`master.info`

The use of this option is now deprecated. It was used to set the file name for the replica's connection metadata repository if `master_info_repository=FILE` was set. `--master-info-file` and the use of the `master_info_repository` system variable are deprecated because the use of a file for the connection metadata repository has been superseded by crash-safe tables. For information about the connection metadata repository, see Section 17.2.4.2, “Replication Metadata Repositories”.

- `--master-retry-count=count`

Command-Line Format	`--master-retry-count=#`
Deprecated	Yes
Type	Integer
Default Value	`86400`
Minimum Value	`0`
Maximum Value (64-bit platforms)	`18446744073709551615`
Maximum Value (32-bit platforms)	`4294967295`

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

This option is deprecated; expect it to be removed in a future MySQL release. Use the `SOURCE_RETRY_COUNT` | `MASTER_RETRY_COUNT` option of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement instead.

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

Command-Line Format	`--max-relay-log-size=#`
System Variable	`max_relay_log_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`

Minimum Value	0
Maximum Value	1073741824
Unit	bytes
Block Size	4096

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 17.2.4.1, “The Relay Log”.

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

Command-Line Format	`--relay-log-purge[={OFF|ON}]`
System Variable	`relay_log_purge`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`ON`

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 risks data consistency and is therefore not crash-safe.

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

Command-Line Format	`--relay-log-space-limit=#`
System Variable	`relay_log_space_limit`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	0
Minimum Value	0
Maximum Value	18446744073709551615
Unit	bytes

This option places an upper limit on the total size in bytes of all relay logs on the replica. A value of 0 means “no limit”. This is useful for a replica server host that has limited disk space. When the limit is reached, the I/O (receiver) thread stops reading binary log events from the source 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 (applier) thread needs more events before it can delete relay logs. In that case, the receiver thread exceeds the limit until it becomes possible for the applier 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 receiver thread waits for free space because `--relay-log-space-limit` is exceeded, but the applier thread has no relay log to purge and is unable to satisfy the receiver thread. This forces the receiver thread to ignore `--relay-log-space-limit` temporarily.

- `--replicate-do-db=db_name`

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

Creates a replication filter using the name of a database. Such filters can also be created using `CHANGE REPLICATION FILTER REPLICATE_DO_DB`.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-do-db:`*`channel_1`*`:`*`db_name`*. In this case, the first colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, "Replication Channel Based Filters" for more information.

Note

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

The precise effect of this replication filter depends on whether statement-based or row-based replication is in use.

Statement-based replication. Tell the replication 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 is 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 replica is started with `--replicate-do-db=sales` and you issue the following statements on the source, the `UPDATE` statement is *not* replicated:

```
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 replication 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 replica is started with `--replicate-do-db=sales` and row-based replication is in effect, and then the following statements are run on the source:

```
USE prices;
```

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

The `february` table in the `sales` database on the replica 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 source has no effect on the replica when using row-based replication and `--replicate-do-db=sales`:

```
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 replica is started with `--replicate-do-db=db1`, and the following statements are executed on the source:

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

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

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

In this case, the `UPDATE` statement would have no effect on the replica when using statement-based replication. However, if you are using row-based replication, the `UPDATE` would change `table1` on the replica, but not `table2`—in other words, only tables in the database named by `--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 `--replicate-wild-do-table=db_name.%` instead. See Section 17.2.5, “How Servers Evaluate Replication Filtering Rules”.

Note

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

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

- `--replicate-ignore-db=db_name`

Command-Line Format	`--replicate-ignore-db=name`
Type	String

Creates a replication filter using the name of a database. Such filters can also be created using `CHANGE REPLICATION FILTER REPLICATE_IGNORE_DB`.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-ignore-db:channel_1:db_name`. In this case, the first

colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, "Replication Channel Based Filters" for more information.

Note

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

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, it is treated as the name of a single database.

As with `--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.

Statement-based replication. Tells the replication SQL thread not to replicate any statement where the default database (that is, the one selected by `USE`) is *`db_name`*.

Row-based replication. Tells the replication SQL thread not to update any tables in the database *`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 replica is started with `--replicate-ignore-db=sales` and you issue the following statements on the source:

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

The `UPDATE` statement *is* replicated in such a case because `--replicate-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 replication, the `UPDATE` statement's effects are *not* propagated to the replica, and the replica's copy of the `sales.january` table is unchanged; in this instance, `--replicate-ignore-db=sales` causes *all* changes made to tables in the source's copy of the `sales` database to be ignored by the replica.

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 17.2.5, "How Servers Evaluate Replication Filtering Rules".

If you need cross-database updates to work, use `--replicate-wild-ignore-table=`*`db_name`*`.%` instead. See Section 17.2.5, "How Servers Evaluate Replication Filtering Rules".

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`*

Command-Line Format	`--replicate-do-table=name`

Type	String

Creates a replication filter by telling the replication 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 17.2.5, “How Servers Evaluate Replication Filtering Rules”. You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_DO_TABLE` statement.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-do-table=`*`channel_1:db_name.tbl_name`*. In this case, the first colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, “Replication Channel Based Filters” for more information.

Note

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

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`*

Command-Line Format	`--replicate-ignore-table=name`
Type	String

Creates a replication filter by telling the replication 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 17.2.5, “How Servers Evaluate Replication Filtering Rules”. You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_IGNORE_TABLE` statement.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-ignore-table=`*`channel_1:db_name.tbl_name`*. In this case, the first colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, “Replication Channel Based Filters” for more information.

Note

Global replication filters cannot be used on a MySQL server instance that is configured for Group Replication, because filtering transactions on some servers would make the group unable to reach agreement on a consistent state. Channel specific replication filters can be used on replication channels that are not directly involved with Group Replication, such as where a group member also acts as a replica to a source that is outside the

group. They cannot be used on the `group_replication_applier` or `group_replication_recovery` channels.

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`

Command-Line Format	`--replicate-rewrite-db=old_name->new_name`
Type	String

Tells the replica to create a replication filter that translates the specified database to *`to_name`* if it was *`from_name`* on the source. Only statements involving tables are affected, not statements such as `CREATE DATABASE`, `DROP DATABASE`, and `ALTER DATABASE`.

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.

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

```
$> mysqld --replicate-rewrite-db="olddb->newdb"
```

The effect of the `--replicate-rewrite-db` option differs depending on whether statement-based or row-based binary logging format is used for the query. With statement-based format, DML statements are translated based on the current database, as specified by the `USE` statement. With row-based format, DML statements are translated based on the database where the modified table exists. DDL statements are always filtered based on the current database, as specified by the `USE` statement, regardless of the binary logging format.

To ensure that rewriting produces the expected results, particularly in combination with other replication filtering options, follow these recommendations when you use the `--replicate-rewrite-db` option:

- Create the *`from_name`* and *`to_name`* databases manually on the source and the replica with different names.
- If you use statement-based or mixed binary logging format, do not use cross-database queries, and do not specify database names in queries. For both DDL and DML statements, rely on the `USE` statement to specify the current database, and use only the table name in queries.
- If you use row-based binary logging format exclusively, for DDL statements, rely on the `USE` statement to specify the current database, and use only the table name in queries. For DML statements, you can use a fully qualified table name (*`db.table`*) if you want.

If these recommendations are followed, it is safe to use the `--replicate-rewrite-db` option in combination with table-level replication filtering options such as `--replicate-do-table`.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. Specify the channel name followed by a colon, followed by the filter specification. The first colon is interpreted as a separator, and any subsequent colons are interpreted

as literal colons. For example, to configure a channel specific replication filter on a channel named *`channel_1`*, use:

```
$> mysqld --replicate-rewrite-db=channel_1:db_name1->db_name2
```

If you use a colon but do not specify a channel name, the option configures the replication filter for the default replication channel. See Section 17.2.5.4, “Replication Channel Based Filters” for more information.

Note

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

- `--replicate-same-server-id`

Command-Line Format	`--replicate-same-server-id[={OFF|ON}]`
Type	Boolean
Default Value	`OFF`

This option is for use on replicas. The default is 0 (`FALSE`). With this option set to 1 (`TRUE`), the replica does not skip events that have its own server ID. This setting is normally useful only in rare configurations.

When binary logging is enabled on a replica, the combination of the `--replicate-same-server-id` and `--log-slave-updates` options on the replica can cause infinite loops in replication if the server is part of a circular replication topology. (In MySQL 8.0, binary logging is enabled by default, and replica update logging is the default when binary logging is enabled.) However, the use of global transaction identifiers (GTIDs) prevents this situation by skipping the execution of transactions that have already been applied. If `gtid_mode=ON` is set on the replica, you can start the server with this combination of options, but you cannot change to any other GTID mode while the server is running. If any other GTID mode is set, the server does not start with this combination of options.

By default, the replication I/O (receiver) thread does not write binary log events to the relay log if they have the replica's server ID (this optimization helps save disk usage). If you want to use `--replicate-same-server-id`, be sure to start the replica with this option before you make the replica read its own events that you want the replication SQL (applier) thread to execute.

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

Command-Line Format	`--replicate-wild-do-table=name`
Type	String

Creates a replication filter by telling the replication SQL (applier) 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 17.2.5, “How Servers Evaluate

Replication Filtering Rules". You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_WILD_DO_TABLE` statement.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-wild-do-table=`*`channel_1`*`:`*`db_name.tbl_name`*. In this case, the first colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, "Replication Channel Based Filters" for more information.

Important

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

The replication filter specified by the `--replicate-wild-do-table` 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%`.

Important

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

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 `my1ownAABCdb` 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`.

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

Command-Line Format	`--replicate-wild-ignore-table=name`
Type	String

Creates a replication filter which keeps the replication SQL 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 17.2.5, “How Servers Evaluate Replication Filtering Rules”. You can also create such a filter by issuing a `CHANGE REPLICATION FILTER REPLICATE_WILD_IGNORE_TABLE` statement.

This option supports channel specific replication filters, enabling multi-source replicas to use specific filters for different sources. To configure a channel specific replication filter on a channel named *`channel_1`* use `--replicate-wild-ignore=`*`channel_1`*`:`*`db_name.tbl_name`*. In this case, the first colon is interpreted as a separator and subsequent colons are literal colons. See Section 17.2.5.4, “Replication Channel Based Filters” for more information.

Important

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

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.

Important

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

If you need to filter out `GRANT` statements or other administrative statements, a possible workaround is to use the `--replicate-ignore-db` filter. This filter operates on the default database that is currently in effect, as determined by the `USE` statement. You can therefore create a filter to ignore statements for a database that is not replicated, then issue the `USE` statement to switch the default database to that one immediately before issuing any administrative statements that you want to ignore. In the administrative statement, name the actual database where the statement is applied.

For example, if `--replicate-ignore-db=nonreplicated` is configured on the replica server, the following sequence of statements causes the `GRANT` statement to be ignored, because the default database `nonreplicated` is in effect:

```
USE nonreplicated;
GRANT SELECT, INSERT ON replicated.t1 TO 'someuser'@'somehost';
```

- `--skip-replica-start`

Command-Line Format	`--skip-replica-start[={OFF|ON}]`
Introduced	8.0.26
System Variable	`skip_replica_start`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Boolean

Default Value	OFF

From MySQL 8.0.26, use `--skip-replica-start` in place of `--skip-slave-start`, which is deprecated from that release. In releases before MySQL 8.0.26, use `--skip-slave-start`.

`--skip-replica-start` tells the replica server not to start the replication I/O (receiver) and SQL (applier) threads when the server starts. To start the threads later, use a `START REPLICA` statement.

You can use the `skip_replica_start` system variable in place of the command line option to allow access to this feature using MySQL Server's privilege structure, so that database administrators do not need any privileged access to the operating system.

- `--skip-slave-start`

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

From MySQL 8.0.26, `--skip-slave-start` is deprecated and the alias `--skip-replica-start` should be used instead. In releases before MySQL 8.0.26, use `--skip-slave-start`.

Tells the replica server not to start the replication I/O (receiver) and SQL (applier) threads when the server starts. To start the threads later, use a `START REPLICA` statement.

From MySQL 8.0.24, you can use the `skip_slave_start` system variable in place of the command line option to allow access to this feature using MySQL Server's privilege structure, so that database administrators do not need any privileged access to the operating system.

- `--slave-skip-errors=[err_code1,err_code2,...|all|ddl_exist_errors]`

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

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

	`ddl_exist_errors`

Normally, replication stops when an error occurs on the replica, which gives you the opportunity to resolve the inconsistency in the data manually. This option causes the replication 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 replicas becoming hopelessly out of synchrony with the source, with you having no idea why this has occurred.

For error codes, you should use the numbers provided by the error message in your replica's error log and in the output of `SHOW REPLICA STATUS`. Appendix B, *Error Messages 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 replica 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 replica's data is not anywhere close to what it is on the source. *You have been warned*.

This option does not work in the same way when replicating between NDB Clusters, due to the internal `NDB` mechanism for checking epoch sequence numbers; normally, as soon as `NDB` detects an epoch number that is missing or otherwise out of sequence, it immediately stops the replica applier thread. Beginning with NDB 8.0.28, you can override this behavior by also specifying `--ndb-applier-allow-skip-epoch` together with `--slave-skip-errors`; doing so causes `NDB` to ignore skipped epoch transactions.

Examples:

```
--slave-skip-errors=1062,1053
--slave-skip-errors=all
--slave-skip-errors=ddl_exist_errors
```

- `--slave-sql-verify-checksum={0|1}`

Command-Line Format	`--slave-sql-verify-checksum[={OFF|ON}]`
Type	Boolean
Default Value	`ON`

When this option is enabled, the replica examines checksums read from the relay log. In the event of a mismatch, the replica 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`

Command-Line Format	`--abort-slave-event-count=#`
Deprecated	8.0.29
Type	Integer
Default Value	`0`
Minimum Value	`0`

When this option is set to some positive integer *`value`* other than 0 (the default) it affects replication behavior as follows: After the replication SQL thread has started, *`value`* log events are permitted to be executed; after that, the replication SQL thread does not receive any more events, just as if the network connection from the source were cut. The replication SQL thread continues to run, and the output from `SHOW REPLICA STATUS` displays `Yes` in both the `Replica_IO_Running` and the `Replica_SQL_Running` columns, but no further events are read from the relay log.

This option is used internally by the MySQL test suite for replication testing and debugging. It is not intended for use in a production setting. Beginning with MySQL 8.0.29, it is deprecated, and subject to removal in a future version of MySQL.

- `--disconnect-slave-event-count`

Command-Line Format	`--disconnect-slave-event-count=#`
Deprecated	8.0.29
Type	Integer
Default Value	`0`

This option is used internally by the MySQL test suite for replication testing and debugging. It is not intended for use in a production setting. Beginning with MySQL 8.0.29, it is deprecated, and subject to removal in a future version of MySQL.

System Variables Used on Replica Servers

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

- `init_replica`

Command-Line Format	`--init-replica=name`
Introduced	8.0.26
System Variable	`init_replica`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String

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

`init_replica` is similar to `init_connect`, but is a string to be executed by a replica server each time the replication 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 REPLICA` statements.

Note

The replication SQL thread sends an acknowledgment to the client before it executes `init_replica`. Therefore, it is not guaranteed that `init_replica` has been executed when `START REPLICA` returns. See Section 13.4.2.8, “START REPLICA Statement” for more information.

- `init_slave`

Command-Line Format	`--init-slave=name`

Deprecated	8.0.26
System Variable	`init_slave`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String

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

`init_slave` is similar to `init_connect`, but is a string to be executed by a replica server each time the replication 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 REPLICA` statements.

Note

The replication 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 REPLICA` returns. See Section 13.4.2.8, "START REPLICA Statement" for more information.

- `log_slow_replica_statements`

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

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

When the slow query log is enabled, `log_slow_replica_statements` enables logging for queries that have taken more than `long_query_time` seconds to execute on the replica. Note that if row-based replication is in use (`binlog_format=ROW`), `log_slow_replica_statements` has no effect. Queries are only added to the replica'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 replica's slow query log, even if `log_slow_replica_statements` is enabled.

Setting `log_slow_replica_statements` has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` 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 replication SQL thread to implement the change there (for example, by issuing `STOP REPLICA` and `START REPLICA` statements with the `SQL_THREAD` option).

- `log_slow_slave_statements`

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

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

When the slow query log is enabled, `log_slow_slave_statements` enables logging for queries that have taken more than `long_query_time` seconds to execute on the replica. 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 replica'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 replica'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 REPLICA` 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 replication SQL thread to implement the change there (for example, by issuing `STOP REPLICA` and `START REPLICA` statements with the `SQL_THREAD` option).

- `master_info_repository`

Command-Line Format	`--master-info-repository={FILE|TABLE}`
Deprecated	8.0.23
System Variable	`master_info_repository`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String
Default Value	`TABLE`

Valid Values	`FILE`
	
	`TABLE`

The use of this system variable is now deprecated. The setting `TABLE` is the default, and is required when multiple replication channels are configured. The alternative setting `FILE` was previously deprecated.

With the default setting, the replica records metadata about the source, consisting of status and connection information, to an `InnoDB` table in the `mysql` system database named

`mysql.slave_master_info`. For more information on the connection metadata repository, see Section 17.2.4, “Relay Log and Replication Metadata Repositories”.

The `FILE` setting wrote the replica's connection metadata repository to a file, which was named `master.info` by default. The name could be changed using the `--master-info-file` option.

- `max_relay_log_size`

Command-Line Format	`--max-relay-log-size=#`
System Variable	`max_relay_log_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`1073741824`
Unit	bytes
Block Size	`4096`

If a write by a replica to its relay log causes the current log file size to exceed the value of this variable, the replica 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 17.2.3, “Replication Threads”.

- `relay_log`

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

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.

The relay log and relay log index on a replication server cannot be given the same names as the binary log and binary log index, whose names are specified by the `--log-bin` and `--log-bin-index` options. The server issues an error message and does not start if the binary log and relay log file base names would be the same.

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 Section 4.2.2, "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.

- `relay_log_basename`

System Variable	`relay_log_basename`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	File name
Default Value	`datadir + '/' + hostname + '-relay-bin'`

Holds the base name and complete path to the relay log file. The maximum variable length is 256. This variable is set by the server and is read only.

- `relay_log_index`

Command-Line Format	`--relay-log-index=file_name`
System Variable	`relay_log_index`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	File name
Default Value	`*host_name*-relay-bin.index`

The name for the relay log index file. The maximum variable length is 256. If you do not specify this variable, but the `relay_log` system variable is specified, its value is used as the default base name for the relay log index file. If `relay_log` is also not specified, then for the default replication channel, the default name is *`host_name`*`-relay-bin.index`, using the name of the host machine. 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 default location for relay log files is the data directory, or any other location that was specified using the `relay_log` system variable. You can use the `relay_log_index` system variable to specify an alternative location, by adding a leading absolute path name to the base name to specify a different directory.

The relay log and relay log index on a replication server cannot be given the same names as the binary log and binary log index, whose names are specified by the `--log-bin` and `--log-bin-index` options. The server issues an error message and does not start if the binary log and relay log file base names would be the same.

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 Section 4.2.2, “Specifying Program Options”.

- `relay_log_info_file`

Command-Line Format	`--relay-log-info-file=file_name`
Deprecated	8.0.18
System Variable	`relay_log_info_file`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	File name
Default Value	`relay-log.info`

The use of this system variable is now deprecated. It was used to set the file name for the replica's applier metadata repository if `relay_log_info_repository=FILE` was set. `relay_log_info_file` and the use of the `relay_log_info_repository` system variable are deprecated because the use of a file for the applier metadata repository has been superseded by crash-safe tables. For information about the applier metadata repository, see Section 17.2.4.2, “Replication Metadata Repositories”.

- `relay_log_info_repository`

Command-Line Format	`--relay-log-info-repository=value`
Deprecated	8.0.23
System Variable	`relay_log_info_repository`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String
Default Value	`TABLE`
Valid Values	`FILE`

	TABLE

The use of this system variable is now deprecated. The setting TABLE is the default, and is required when multiple replication channels are configured. The TABLE setting for the replica's applier metadata repository is also required to make replication resilient to unexpected halts. See Section 17.4.2, “Handling an Unexpected Halt of a Replica” for more information. The alternative setting FILE was previously deprecated.

With the default setting, the replica stores its applier metadata repository as an InnoDB table in the mysql system database named mysql.slave_relay_log_info. For more information on the applier metadata repository, see Section 17.2.4, “Relay Log and Replication Metadata Repositories”.

The FILE setting wrote the replica's applier metadata repository to a file, which was named relay-log.info by default. The name could be changed using the relay_log_info_file system variable.

- relay_log_purge

Command-Line Format	--relay-log-purge[={OFF|ON}]
System Variable	relay_log_purge
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	ON

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

Command-Line Format	--relay-log-recovery[={OFF|ON}]
System Variable	relay_log_recovery
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

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 (applier) thread position to this new relay log, and initializes the I/O (receiver) thread to the applier thread position. Reading of the relay log from the source then continues. If SOURCE_AUTO_POSITION=1 was set for the replication channel using the CHANGE REPLICATION SOURCE TO option, the source position used to start replication might be the one received in the connection and not the ones assigned in this process.

This global variable is read-only at runtime. Its value can be set with the --relay-log-recovery option at replica server startup, which should be used following an unexpected halt of a replica to ensure that no possibly corrupted relay logs are processed, and must be used in order to guarantee a crash-safe replica. The default value is 0 (disabled). For information on the combination of settings

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

For a multithreaded replica (where `replica_parallel_workers` or `slave_parallel_workers` is greater than 0), setting `--relay-log-recovery` at startup automatically handles any inconsistencies and gaps in the sequence of transactions that have been executed from the relay log. These gaps can occur when file position based replication is in use. (For more details, see Section 17.5.1.34, “Replication and Transaction Inconsistencies”.) The relay log recovery process deals with gaps using the same method as the `START REPLICA UNTIL SQL_AFTER_MTS_GAPS` statement would. When the replica reaches a consistent gap-free state, the relay log recovery process goes on to fetch further transactions from the source beginning at the SQL (applier) thread position. When GTID-based replication is in use, from MySQL 8.0.18 a multithreaded replica checks first whether `MASTER_AUTO_POSITION` is set to `ON`, and if it is, omits the step of calculating the transactions that should be skipped or not skipped, so that the old relay logs are not required for the recovery process.

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 source or another group.

- `relay_log_space_limit`

Command-Line Format	`--relay-log-space-limit=#`
System Variable	`relay_log_space_limit`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`0`
Minimum Value	`0`
Maximum Value	`18446744073709551615`
Unit	bytes

The maximum amount of space to use for all relay logs.

- `replica_checkpoint_group`

Command-Line Format	`--replica-checkpoint-group=#`
Introduced	8.0.26
System Variable	`replica_checkpoint_group`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`512`
Minimum Value	`32`

Maximum Value	524280
Block Size	8

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

`replica_checkpoint_group` sets the maximum number of transactions that can be processed by a multithreaded replica before a checkpoint operation is called to update its status as shown by `SHOW REPLICA STATUS`. Setting this variable has no effect on replicas for which multithreading is not enabled. Setting this variable has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` commands.

Note

Multithreaded replicas are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See Section 23.7.3, “Known Issues in NDB Cluster Replication”, for more information.

This variable works in combination with the `replica_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.

- `replica_checkpoint_period`

Command-Line Format	`--replica-checkpoint-period=#`
Introduced	8.0.26
System Variable	`replica_checkpoint_period`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	300
Minimum Value	1
Maximum Value	4294967295
Unit	milliseconds

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

`replica_checkpoint_period` sets the maximum time (in milliseconds) that is allowed to pass before a checkpoint operation is called to update the status of a multithreaded replica as shown by `SHOW REPLICA STATUS`. Setting this variable has no effect on replicas for which multithreading

is not enabled. Setting this variable takes effect for all replication channels immediately, including running channels.

Note

Multithreaded replicas are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See Section 23.7.3, "Known Issues in NDB Cluster Replication", for more information.

This variable works in combination with the `replica_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 milliseconds, and the maximum possible value is 4294967295 milliseconds (approximately 49.7 days).

- `replica_compressed_protocol`

Command-Line Format	`--replica-compressed-protocol[={OFF|ON}]`
Introduced	8.0.26
System Variable	`replica_compressed_protocol`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

From MySQL 8.0.26, use `replica_compressed_protocol` in place of `slave_compressed_protocol`, which is deprecated. In releases before MySQL 8.0.26, use `slave_compressed_protocol`.

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

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

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

- `replica_exec_mode`

Command-Line Format	`--replica-exec-mode=mode`

Introduced	8.0.26
System Variable	`replica_exec_mode`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`IDEMPOTENT` (NDB) `STRICT` (Other)

Valid Values	`STRICT`
	
	`IDEMPOTENT`

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

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

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

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

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

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

- `replica_load_tmpdir`

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

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

`replica_load_tmpdir` specifies the name of the directory where the replica creates temporary files. Setting this variable takes effect for all replication channels immediately, including running

channels. The variable value is by default equal to the value of the `tmpdir` system variable, or the default that applies when that system variable is not specified.

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

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

- `replica_max_allowed_packet`

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

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

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

This global variable always has a value that is a positive integer multiple of 1024; if you set it to some value that is not, the value is rounded down to the next highest multiple of 1024 for it is stored or used; setting `replica_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.

- `replica_net_timeout`

Command-Line Format	`--replica-net-timeout=#`
Introduced	8.0.26
System Variable	`replica_net_timeout`

Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	60
Minimum Value	1
Maximum Value	31536000
Unit	seconds

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

replica_net_timeout specifies the number of seconds to wait for more data or a heartbeat signal from the source before the replica considers the connection broken, aborts the read, and tries to reconnect. Setting this variable has no immediate effect. The state of the variable applies on all subsequent START REPLICA commands.

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

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

- replica_parallel_type

Command-Line Format	--replica-parallel-type=value
Introduced	8.0.26
Deprecated	8.0.29
System Variable	replica_parallel_type
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value (≥ 8.0.27)	LOGICAL_CLOCK
Default Value (8.0.26)	DATABASE
Valid Values	DATABASE

	LOGICAL_CLOCK

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

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

- LOGICAL_CLOCK: Transactions are applied in parallel on the replica, based on timestamps which the replication source writes to the binary log. Dependencies between transactions are tracked based on their timestamps to provide additional parallelization where possible.
- DATABASE: Transactions that update different databases are applied in parallel. This value is only appropriate if data is partitioned into multiple databases which are being updated independently and concurrently on the source. There must be no cross-database constraints, as such constraints may be violated on the replica.

When replica_preserve_commit_order or slave_preserve_commit_order is enabled, you must use LOGICAL_CLOCK. Before MySQL 8.0.27, DATABASE is the default. From MySQL 8.0.27, multithreading is enabled by default for replica servers (replica_parallel_workers=4 by default), and LOGICAL_CLOCK is the default. (In MySQL 8.0.27 and later, replica_preserve_commit_order is also enabled by default.)

When the replication topology uses multiple levels of replicas, LOGICAL_CLOCK may achieve less parallelization for each level the replica is away from the source. To compensate for this effect, you should set binlog_transaction_dependency_tracking to WRITESET or WRITESET_SESSION on the source *as well as on every intermediate replica* to specify that write sets are used instead of timestamps for parallelization where possible.

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

replica_parallel_type is deprecated beginning with MySQL 8.0.29, as is support for parallelization of transactions using database partitioning. Expect support for these to be removed in a future release, and for LOGICAL_CLOCK to be used exclusively thereafter.

- replica_parallel_workers

Command-Line Format	--replica-parallel-workers=#
Introduced	8.0.26
System Variable	replica_parallel_workers
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value (≥ 8.0.27)	4
Default Value (8.0.26)	0
Minimum Value	0

Maximum Value	1024

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

`replica_parallel_workers` enables multithreading on the replica and sets the number of applier threads for executing replication transactions in parallel. When the value is a number greater than 1, the replica is a multithreaded replica with the specified number of applier threads, plus a coordinator thread to manage them. If you are using multiple replication channels, each channel has this number of threads.

Before MySQL 8.0.27, the default for this system variable is 0, so replicas are single-threaded by default. From MySQL 8.0.27, the default is 4, so replicas are multithreaded by default.

Beginning with MySQL 8.0.30, setting this variable to 0 is deprecated, and doing so raises a warning; 0 as a permitted value for `replica_parallel_workers` is subject to removal in a future MySQL release; set it to 1 instead, which has the same effect.

Retrying of transactions is supported when multithreading is enabled on a replica. When `replica_preserve_commit_order=ON` or `slave_preserve_commit_order=ON` is set, transactions on a replica are externalized on the replica in the same order as they appear in the replica's relay log. The way in which transactions are distributed among applier threads is configured by `replica_parallel_type` (from MySQL 8.0.26) or `slave_parallel_type` (before MySQL 8.0.26). From MySQL 8.0.27, these system variables also have appropriate defaults for multithreading.

To disable parallel execution, set `replica_parallel_workers` to 1, which gives the replica a single applier thread and no coordinator thread. With this setting, the `replica_parallel_type` or `slave_parallel_type` and `replica_preserve_commit_order` or `slave_preserve_commit_order` system variables have no effect and are ignored. From MySQL 8.0.27, if parallel execution is disabled when the `CHANGE REPLICATION SOURCE TO` option `GTID_ONLY` is enabled on a replica, the replica actually uses one parallel worker to take advantage of the method for retrying transactions without accessing the file positions. With one parallel worker, the `replica_preserve_commit_order` or `slave_preserve_commit_order` system variable also has no effect.

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

Note

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

- `replica_pending_jobs_size_max`

Command-Line Format	`--replica-pending-jobs-size-max=#`
Introduced	8.0.26
System Variable	`replica_pending_jobs_size_max`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer

Default Value	128M
Minimum Value	1024
Maximum Value	16EiB
Unit	bytes
Block Size	1024

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

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

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

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

- `replica_preserve_commit_order`

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

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

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

Before MySQL 8.0.27, the default for this system variable is `OFF`, meaning that transactions may be committed out of order. From MySQL 8.0.27, multithreading is enabled by default for replica servers (`replica_parallel_workers=4` by default), so `replica_preserve_commit_order=ON` is the default, and the setting `replica_parallel_type=LOGICAL_CLOCK` is also the default. Also from MySQL 8.0.27, the setting for `replica_preserve_commit_order` is ignored if

`replica_parallel_workers` is set to 1, because in that situation the order of transactions is preserved anyway.

Binary logging and replica update logging are not required on the replica to set `replica_preserve_commit_order=ON`, and can be disabled if wanted. Setting `replica_preserve_commit_order=ON` requires that `replica_parallel_type` is set to `LOGICAL_CLOCK`, which is *not* the default setting before MySQL 8.0.27. Before changing the value of `replica_preserve_commit_order` and `replica_parallel_type`, the replication SQL thread (for all replication channels if you are using multiple replication channels) must be stopped.

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

When `replica_preserve_commit_order=ON` is set, the executing worker thread waits until all previous transactions are committed before committing. While a given thread is waiting for other worker threads to commit their transactions, it reports its status as `Waiting for preceding transaction to commit`. With this mode, a multithreaded replica never enters a state that the source was not in. This supports the use of replication for read scale-out. See Section 17.4.5, “Using Replication for Scale-Out”.

Note

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

- `replica_sql_verify_checksum`

Command-Line Format	`--replica-sql-verify-checksum[={OFF|ON}]`
Introduced	8.0.26
System Variable	`replica_sql_verify_checksum`

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

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

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

Note

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

- replica_transaction_retries

Command-Line Format	--replica-transaction-retries=#
Introduced	8.0.26
System Variable	replica_transaction_retries
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	10
Minimum Value	0
Maximum Value	18446744073709551615

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

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

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

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

- `replica_type_conversions`

Command-Line Format	`--replica-type-conversions=set`
Introduced	8.0.26
System Variable	`replica_type_conversions`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Set
Default Value	

Valid Values	`ALL_LOSSY`
	`ALL_NON_LOSSY`
	`ALL_SIGNED`
	`ALL_UNSIGNED`

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

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

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

- `replication_optimize_for_static_plugin_config`

Command-Line Format	`--replication-optimize-for-static-plugin-config[={OFF|ON}]`
Introduced	8.0.23
System Variable	`replication_optimize_for_static_plugin_config`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Use shared locks, and avoid unnecessary lock acquisitions, to improve performance for semisynchronous replication. This setting and `replication_sender_observe_commit_only` help as the number of replicas increases, because contention for locks can slow down performance. While this system variable is enabled, the semisynchronous replication plugin cannot be uninstalled, so you must disable the system variable before the uninstall can complete.

This system variable can be enabled before or after installing the semisynchronous replication plugin, and can be enabled while replication is running. Semisynchronous replication source servers

can also get performance benefits from enabling this system variable, because they use the same locking mechanisms as the replicas.

`replication_optimize_for_static_plugin_config` can be enabled when Group Replication is in use on a server. In that scenario, it might benefit performance when there is contention for locks due to high workloads.

- `replication_sender_observe_commit_only`

Command-Line Format	`--replication-sender-observe-commit-only[={OFF|ON}]`
Introduced	8.0.23
System Variable	`replication_sender_observe_commit_only`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Boolean
Default Value	`OFF`

Limit callbacks to improve performance for semisynchronous replication. This setting and `replication_optimize_for_static_plugin_config` help as the number of replicas increases, because contention for locks can slow down performance.

This system variable can be enabled before or after installing the semisynchronous replication plugin, and can be enabled while replication is running. Semisynchronous replication source servers can also get performance benefits from enabling this system variable, because they use the same locking mechanisms as the replicas.

- `report_host`

Command-Line Format	`--report-host=host_name`
System Variable	`report_host`
Scope	Global
Dynamic	No
`SET_VAR` Hint Applies	No
Type	String

The host name or IP address of the replica to be reported to the source during replica registration. This value appears in the output of `SHOW REPLICAS` on the source server. Leave the value unset if you do not want the replica to register itself with the source.

Note

It is not sufficient for the source to simply read the IP address of the replica server from the TCP/IP socket after the replica connects. Due to NAT and other routing issues, that IP may not be valid for connecting to the replica from the source or other hosts.

- `report_password`

Command-Line Format	`--report-password=name`
System Variable	`report_password`
Scope	Global

Dynamic	No
SET_VAR Hint Applies	No
Type	String

The account password of the replica to be reported to the source during replica registration. This value appears in the output of SHOW REPLICAS on the source server if the source was started with --show-replica-auth-info or --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

Command-Line Format	--report-port=port_num
System Variable	report_port
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Integer
Default Value	[slave_port]
Minimum Value	0
Maximum Value	65535

The TCP/IP port number for connecting to the replica, to be reported to the source during replica registration. Set this only if the replica is listening on a nondefault port or if you have a special tunnel from the source or other clients to the replica. If you are not sure, do not use this option.

The default value for this option is the port number actually used by the replica. This is also the default value displayed by SHOW REPLICAS.

- report_user

Command-Line Format	--report-user=name
System Variable	report_user
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	String

The account user name of the replica to be reported to the source during replica registration. This value appears in the output of SHOW REPLICAS on the source server if the source was started with --show-replica-auth-info or --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_read_size

Command-Line Format	--rpl-read-size=#
System Variable	rpl_read_size

Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	8192
Minimum Value	8192
Maximum Value	4294959104
Unit	bytes
Block Size	8192

The rpl_read_size system variable controls the minimum amount of data in bytes that is read from the binary log files and relay log files. If heavy disk I/O activity for these files is impeding performance for the database, increasing the read size might reduce file reads and I/O stalls when the file data is not currently cached by the operating system.

The minimum and default value for rpl_read_size is 8192 bytes. The value must be a multiple of 4KB. Note that a buffer the size of this value is allocated for each thread that reads from the binary log and relay log files, including dump threads on sources and coordinator threads on replicas. Setting a large value might therefore have an impact on memory consumption for servers.

- rpl_semi_sync_replica_enabled

Command-Line Format	--rpl-semi-sync-replica-enabled[={OFF|ON}]
Introduced	8.0.26
System Variable	rpl_semi_sync_replica_enabled
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

rpl_semi_sync_replica_enabled is available when the rpl_semi_sync_replica (semisync_replica.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_slave plugin (semisync_slave.so library) was installed, rpl_semi_sync_slave_enabled is available instead.

rpl_semi_sync_replica_enabled controls whether semisynchronous replication is enabled on the replica server. 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 replica-side semisynchronous replication plugin is installed.

- rpl_semi_sync_replica_trace_level

Command-Line Format	--rpl-semi-sync-replica-trace-level=#
Introduced	8.0.26
System Variable	rpl_semi_sync_replica_trace_level
Scope	Global
Dynamic	Yes

SET_VAR Hint Applies	No
Type	Integer
Default Value	32
Minimum Value	0
Maximum Value	4294967295

rpl_semi_sync_replica_trace_level is available when the rpl_semi_sync_replica (semisync_replica.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_slave plugin (semisync_slave.so library) was installed, rpl_semi_sync_slave_trace_level is available instead.

rpl_semi_sync_replica_trace_level controls the semisynchronous replication debug trace level on the replica server. See rpl_semi_sync_master_trace_level for the permissible values.

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

- rpl_semi_sync_slave_enabled

Command-Line Format	--rpl-semi-sync-slave-enabled[={OFF|ON}]
System Variable	rpl_semi_sync_slave_enabled
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

rpl_semi_sync_slave_enabled is available when the rpl_semi_sync_slave (semisync_slave.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_replica plugin (semisync_replica.so library) was installed, rpl_semi_sync_replica_enabled is available instead.

rpl_semi_sync_slave_enabled controls whether semisynchronous replication is enabled on the replica server. 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 replica-side semisynchronous replication plugin is installed.

- rpl_semi_sync_slave_trace_level

Command-Line Format	--rpl-semi-sync-slave-trace-level=#
System Variable	rpl_semi_sync_slave_trace_level
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	32
Minimum Value	0

Maximum Value	4294967295

rpl_semi_sync_slave_trace_level is available when the rpl_semi_sync_slave (semisync_slave.so library) plugin was installed on the replica to set up semisynchronous replication. If the rpl_semi_sync_replica plugin (semisync_replica.so library) was installed, rpl_semi_sync_replica_trace_level is available instead.

rpl_semi_sync_slave_trace_level controls the semisynchronous replication debug trace level on the replica server. See rpl_semi_sync_master_trace_level for the permissible values.

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

- rpl_stop_replica_timeout

Command-Line Format	--rpl-stop-replica-timeout=#
Introduced	8.0.26
System Variable	rpl_stop_replica_timeout
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	31536000
Minimum Value	2
Maximum Value	31536000
Unit	seconds

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

You can control the length of time (in seconds) that STOP REPLICA waits before timing out by setting this variable. This can be used to avoid deadlocks between STOP REPLICA and other SQL statements using different client connections to the replica.

The maximum and default value of rpl_stop_replica_timeout is 31536000 seconds (1 year). The minimum is 2 seconds. Changes to this variable take effect for subsequent STOP REPLICA statements.

This variable affects only the client that issues a STOP REPLICA 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 replication I/O (receiver)and SQL (applier) threads to stop, but the replication threads continue to try to stop, and the STOP REPLICA instruction remains in effect. Once the replication threads are no longer busy, the STOP REPLICA statement is executed and the replica stops.

- rpl_stop_slave_timeout

Command-Line Format	--rpl-stop-slave-timeout=#
Deprecated	8.0.26
System Variable	rpl_stop_slave_timeout
Scope	Global
Dynamic	Yes

SET_VAR Hint Applies	No
Type	Integer
Default Value	31536000
Minimum Value	2
Maximum Value	31536000
Unit	seconds

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

You can control the length of time (in seconds) that STOP REPLICA waits before timing out by setting this variable. This can be used to avoid deadlocks between STOP REPLICA and other SQL statements using different client connections to the replica.

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 REPLICA statements.

This variable affects only the client that issues a STOP REPLICA 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 replication I/O (receiver) and SQL (applier) threads to stop, but the replication threads continue to try to stop, and the STOP REPLICA instruction remains in effect. Once the replication threads are no longer busy, the STOP REPLICA statement is executed and the replica stops.

- skip_replica_start

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

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

skip_replica_start tells the replica server not to start the replication I/O (receiver) and SQL (applier) threads when the server starts. To start the threads later, use a START REPLICA statement.

This system variable is read-only and can be set by using the PERSIST_ONLY keyword or the @@persist_only qualifier with the SET statement. The --skip-replica-start command line option also sets this system variable. You can use the system variable in place of the command line option to allow access to this feature using MySQL Server's privilege structure, so that database administrators do not need any privileged access to the operating system.

- skip_slave_start

Command-Line Format	--skip-slave-start[={OFF|ON}]
Deprecated	8.0.26

System Variable	skip_slave_start
Scope	Global
Dynamic	No
SET_VAR Hint Applies	No
Type	Boolean
Default Value	OFF

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

Tells the replica server not to start the replication I/O (receiver) and SQL (applier) threads when the server starts. To start the threads later, use a START REPLICA statement.

This system variable is available from MySQL 8.0.24. It is read-only and can be set by using the PERSIST_ONLY keyword or the @@persist_only qualifier with the SET statement. The --skip-slave-start command line option also sets this system variable. You can use the system variable in place of the command line option to allow access to this feature using MySQL Server's privilege structure, so that database administrators do not need any privileged access to the operating system.

- slave_checkpoint_group

Command-Line Format	--slave-checkpoint-group=#
Deprecated	8.0.26
System Variable	slave_checkpoint_group
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Integer
Default Value	512
Minimum Value	32
Maximum Value	524280
Block Size	8

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

slave_checkpoint_group sets the maximum number of transactions that can be processed by a multithreaded replica before a checkpoint operation is called to update its status as shown by SHOW REPLICA STATUS. Setting this variable has no effect on replicas for which multithreading is

not enabled. Setting this variable has no immediate effect. The state of the variable applies on all subsequent `START REPLICA` commands.

Note

Multithreaded replicas are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See Section 23.7.3, "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`

Command-Line Format	`--slave-checkpoint-period=#`
Deprecated	8.0.26
System Variable	`slave_checkpoint_period`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`300`
Minimum Value	`1`
Maximum Value	`4294967295`
Unit	milliseconds

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

`slave_checkpoint_period` sets the maximum time (in milliseconds) that is allowed to pass before a checkpoint operation is called to update the status of a multithreaded replica as shown by `SHOW REPLICA STATUS`. Setting this variable has no effect on replicas for which multithreading is not enabled. Setting this variable takes effect for all replication channels immediately, including running channels.

Note

Multithreaded replicas are not currently supported by NDB Cluster, which silently ignores the setting for this variable. See Section 23.7.3, "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