Galera Cluster Documentation

Release

Codership Oy

May 26, 2019

CONTENTS

Galera Cluster is a synchronous multi-master database cluster, based on synchronous replication and MySQL and InnoDB. When Galera Cluster is in use, database reads and writes can be directed to any node. Any individual node can be lost without interruption in operations and without using complex failover procedures.

At a high level, Galera Cluster consists of a database server—that is, MySQL, MariaDB or Percona XtraDB—that uses the *Galera Replication Plugin* to manage replication. To be more specific, the MySQL replication plugin API has been extended to provide all the information and hooks required for true multi-master, synchronous replication. This extended API is called the Write-Set Replication API, or wsrep API.

Through the wsrep API, Galera Cluster provides certification-based replication. A transaction for replication, the write-set not only contains the database rows to replicate, but also includes information on all of the locks that were held by the database during the transaction. Each node then certifies the replicated write-set against other write-sets in the applier queue. The write-set is then applied—if there are no conflicting locks. At this point, the transaction is considered committed, after which each node continues to apply it to the tablespace.

This approach is also called virtually synchronous replication, given that while it is logically synchronous, the actual writing and committing to the tablespace happens independently, and thus asynchronously on each node.

Benefits of Galera Cluster

Galera Cluster provides a significant improvement in high-availability for the MySQL system. The various ways to achieve high-availability have typically provided only some of the features available through Galera Cluster, making the choice of a high-availability solution an exercise in trade-offs.

The following features are available through Galera Cluster:

- **True Multi-Master**

  You can read and write to any node at any time. Changes to data on one node will be replicated on all.

- **Synchronous Replication**

  There is no slave lag, so no data is lost if a node crashes.

- **Tightly Coupled**

  All nodes hold the same state. There is no diverged data between nodes.

- **Multi-Threaded Slave**

  This allows for better performance and for any workload.

- **No Master-Slave Failover**

  There is no need for master-slave operations or to use VIP.

- **Hot Standby**

  There is no downtime related to failures or intentionally taking down a node for maintenance since there is no failover.

- **Automatic Node Provisioning**

  There's no need to backup manually the database and copy it to the new node.

- **Supports InnoDB.**

  The InnoDB storage engine allows for transactional tables, which is necessary for Galera.

- **Transparent to Applications**

  Generally, you won't have to change an application that will interface with the database as a result of Galera. If you do, it will be minimal changes.

- **No Read and Write Splitting Needed**

  There is no need to split read and write queries.

In summary, Galera Cluster is a high-availability solution that is both robust in terms of data integrity and provides high-performance with instant failovers.

Cloud Implementations with Galera Cluster

An additional benefit of Galera Cluster is good cloud support. Automatic node provisioning makes elastic scale-out and scale-in operations painless. Galera Cluster has been proven to perform extremely well in the cloud, such as when using multiple small node instances, across multiple data centers—AWS zones, for example—or even over Wider Area Networks.

Part I

Getting Started

Galera Cluster is a synchronous replication solution to improve availability and performance of the MySQL database service. All nodes in Galera Cluster are identical and fully representative of the cluster. They allow for unconstrained transparent client access, operating as a single, distributed MySQL server. It provides,

- Transparent client connections, so it's highly compatible with existing applications;
- Synchronous data safety semantics—if a client received confirmation, transactions will be committed on every node; and
- Automatic write conflict detection and resolution, so that nodes are always consistent.

Galera Cluster is well suited for LAN, WAN, container and cloud environments. The following chapters provide you with the basics to setting up and deploying Galera Cluster. Bear in mind before you get started that you need root access to at least three Linux or FreeBSD hosts and their IP addresses.

Note: With the latest release Glaerea Cluster begins the 4.x branch, introducing a number of new releases. For more information on these features, see *What's New in Galera Cluster 4.x* (page 9).

Node Initialization

Individual nodes in Galera Cluster are MySQL, MariaDB or Percona XtraDB. But, deploying a node is not exactly the same as the standard standalone instance of the database server. You need to take a few additional steps in order to properly install and configure the software. The software runs on any unix-like operating system. These chapters provides guides to installing and configuring nodes for Galera Cluster.

- *Installation* (page 11)

  Once you have your server hardware ready, including at least three hosts running either Linux or FreeBSD and their respective IP addresses. This chapter provides guides to preparing the server and installing Galera Cluster. When you install Galera Cluster, you must choose between three implementations available. For each implementations, you can install the software using Debian- and RPM-based binary packages or by building the node from source.

  - **Galera Cluster for MySQL**: The reference implementation from Codership, Oy. You can use either the *Binary Installation* (page 13) or the *Source Build* (page 17) methods.
  - **Percona XtraDB Cluster**: The Percona alternative implementation of Galera Cluster, which uses XtraDB in place of MySQL. You can use either the Binary Installation or the Source Build methods.
  - **MariaDB Galera Cluster**: The MariaDB Ab alternative implementation of Galera Cluster, which uses MariaDB in place of MySQL. You can use either the *Binary Installation* (page 19) or the *Source Build* (page 22).

- *System Configuration* (page 27)

  Before you can start the cluster, you need to configure the individual nodes. This chapter covers general parameters that you must set in the `my.cnf` configuration file in order to use Galera Cluster. It also provides a guide to configuring swap space in order to protect the node from crashing due to large write-sets requiring more memory than the server has available.

- *Replication Configuration* (page 31)

  With the system-level configurations complete, this chapter provides a guide to configuring the database server to connect and communicate with the cluster and explains the syntax format used in cluster addresses.

Cluster Initialization

With the software installed on the relevant servers in your your infrastructure, you can now initialize Galera Cluster, by bootstrapping the Primary Component then starting all the other nodes as you would any other database server instance. These chapters provide guides to starting the cluster, ways of testing that it's operational and, when you need to, how to restart the entire cluster.

- *Starting the Cluster* (page 33)

  In order to start Galera Cluster, you need to bootstrap the Primary Component. Afterwards, you can start nodes using the same command as you would the standard standalone MySQL, MariaDB or Percona XtraDB database server. This chapter provides a guide to initializing the cluster.

- *Testing a Cluster* (page 37)

  With your cluster online, you may want to test out some of the features in order to ensure it's working properly and to better see how it works in planning out your own deployment. This chapter provides a rough guide to testing replication and similar cluster operations.

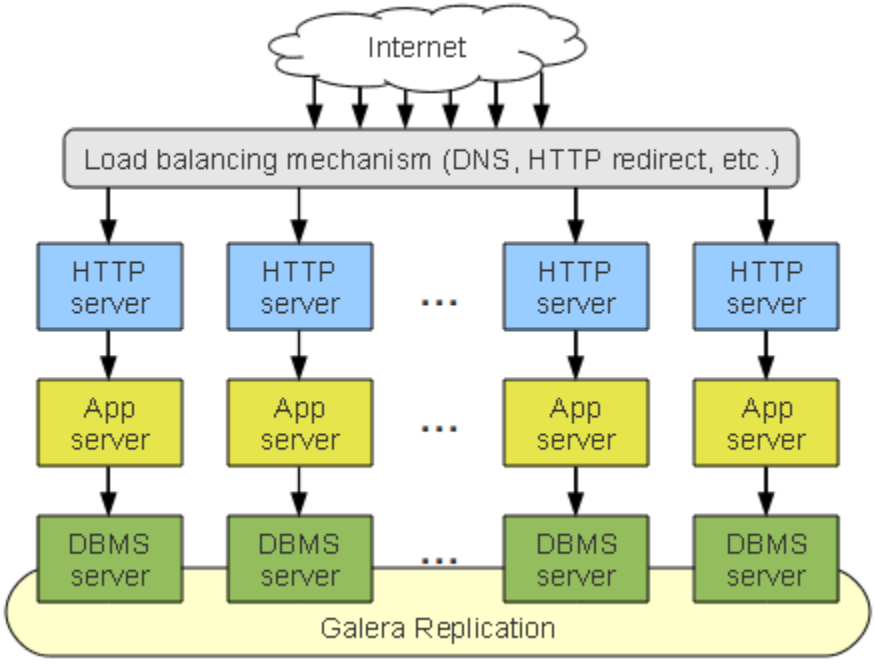
- *Restarting the Cluster* (page 39)

  On occasion, you may need to restart the entire cluster, such as in the case of a power failure where every node is shut down and there is no database server process left running. This chapter provides guides to finding your most advanced node and restarting the Primary Component on that node.

How Galera Cluster Works

The primary focus is data consistency. The transactions are either applied on every node or not all. So, the databases stay synchronized, provided that they were properly configured and synchronized at the beginning.

The *Galera Replication Plugin* differs from the standard MySQL Replication by addressing several issues, including multi-master write conflicts, replication lag and slaves being out of sync with the master.



In a typical instance of a Galera Cluster, applications can write to any node in the cluster and transaction commits, (RBR events), are then applied to all the servers, through certification-based replication.

Certification-based replication is an alternative approach to synchronous database replication, using group communication and transaction ordering techniques.

Note: For security and performance reasons, it's recommended that you run Galera Cluster on its own subnet.

CHAPTER

ONE

WHAT'S NEW IN GALERA CLUSTER 4.X

With the latest release of Galera Cluster in the 4.x branch, there are some new features available to you, including the following:

- **Streaming Replication** Under normal operation, the node initiates all replication and certification operations when the transaction commits. For large transactions, this can result in conflicts: smaller transactions can get in first and cause the large transactions to abort. With Streaming Replication, the node breaks the transaction into fragments, then certifies and replicates them on all slave nodes while the transaction is still in progress. Once certified, conflicting transactions can no longer abort the fragment.

  This provides an alternative replication method for handling large or long-running write transactions, or when working with hot records.

  For more information, see *Streaming Replication* (page 73) and *Using Streaming Replication* (page 115).

- **Synchronization Functions** This version introduces a series of SQL functions for use in wsrep synchronization operations. You can use them to obtain the *Global Transaction ID*, based on either the last write or last seen transaction, as well as setting the node to wait for a specific GTID to replicate and apply, before initiating the next transaction.

  For more information, see *Using Synchronization Functions* (page 229) and *MySQL wsrep Functions* (page 263).

- **Galera System Tables** In version 4 of Galera, three system tables were added to the `mysql` database: `wsrep_cluster`, `wsrep_cluster_members`, and `wsrep_streaming_log`. These tables may be used by database administrators to get a sense the current activity of the nodes in a cluster.

  For more information, see *System Tables* (page 305) and system-tables.

CHAPTER

TWO

INSTALLATION

Galera Cluster requires server hardware for a minimum of three nodes.

If your cluster runs on a single switch, use three nodes. If your cluster spans switches, use three switches. If your cluster spans networks, use three networks. If your cluster spans data centers, use three data centers. This ensures that the cluster can maintain a Primary Component in the event of network outages.

Hardware Requirements

For server hardware, each node requires at a minimum the following components:

- 1 GHz single core CPU;
- 512 MB RAM; and
- 100 Mbps network connectivity

Note: **See Also**: Galera Cluster may occasionally crash when run on limited hardware due to insufficient memory. To prevent this, make sure that you have allocated a sufficient amount of swap space. For more information on how to create swap space, see *Configuring Swap Space* (page 29).

Software Requirements

For software, each node in the cluster requires at a minimum the following:

- Linux or FreeBSD operating system installed;
- MySQL or MariaDB server with the wsrep API patch; and
- Galera Replication Plugin installed.

Note: Binary installation packages for Galera Cluster include the database server with the wsrep API patch. When building from source, though, you must apply this patch manually.

Preparing the Server

Before you begin the installation process, there are a few tasks that you need to do to prepare the servers for Galera Cluster. You must perform the following steps on each node in your cluster.

Disabling SELinux for mysqld

If SELinux (Security-Enhanced Linux) is enabled on the servers, it may block `mysqld` from performing required operations. You must either disable SELinux for `mysqld` or configure it to allow `mysqld` to run external programs and open listen sockets on unprivileged ports—that is, operations that an unprivileged user may do.

To disable SELinux for `mysqld`, execute the following from the command-line:

```
# semanage permissive -a mysqld_t
```

This command switches SELinux into permissive mode when it registers activity from the database server. While this is fine during the installation and configuration process, it is not in general a good policy to disable security applications.

Rather than disable SELinux, so that your may use it along with Galera Cluster, you will need to create an access policy. This will allow SELinux to understand and allow normal operations from the database server. For information on how to create such an access policy, see *SELinux Configuration* (page 183).

Note: **See Also**: For more information on writing SELinux policies, see SELinux and MySQL.

Firewall Configuration

Next, you will need to update the firewall settings on each node so that they may communicate with the cluster. How you do this varies depending upon your distribution and the particular firewall software that you use.

Note: If there is a NAT (Network Address Translation) firewall between the nodes, you must configure it to allow for direct connections between the nodes, such as through port forwarding.

As an example, to open ports between trusted hosts using `iptables`, you would execute something like the following on each node:

```
# iptables --append INPUT --protocol tcp \
      --source 64.57.102.34 --jump ACCEPT
# iptables --apend INPUT --protocol tcp \
      --source 193.166.33.20 --jump ACCEPT
# iptables --append INPUT --protocol tcp \
      --source 193.125.4.10 --jump ACCEPT
```

This causes packet filtering on the kernel to accept TCP (Transmission Control Protocol) connections between the given IP addresses.

Note: **Warning**: The IP addresses in the example are for demonstration purposes only. Use the real values from your nodes and netmask in the `iptables` configuration for your cluster.

The updated packet filtering rules take effect immediately, but are not persistent. When the server reboots, it reverts to default packet filtering rules, which do not include your updates. To use these rules after rebooting, you need to save them as defaults.

For systems that use `init`, run the following command:

```
# service save iptables
```

For systems that use `systemd`, you need to save the current packet filtering rules to the path that the `iptables` unit reads when it starts. This path can vary by distribution, but you can normally find it in the `/etc` directory.

- `/etc/sysconfig/iptables`
- `/etc/iptables/iptables.rules`

When you find the relevant file, you can save the rules using the `iptables-save` command, then redirecting the output to overwrite this file.

```
# iptables-save > /etc/sysconfig/iptables
```

When `iptables` starts it now reads the new defaults, with your updates to the firewall.

Note: See Also: For more information on setting up the firewall for Galera Cluster and other programs for configuring packet filtering in Linux and FreeBSD, see *Firewall Settings* (page 169).

Disabling AppArmor

By default, some servers—for instance, Ubuntu—include AppArmor, which may prevent `mysqld` from opening additional ports or running scripts. You must disable AppArmor or configure it to allow `mysqld` to run external programs and open listen sockets on unprivileged ports.

To disable AppArmor, run the following commands:

```
$ sudo ln -s /etc/apparmor.d/usr /etc/apparmor.d/disable/.sbin.mysqld
```

You will then need to restart AppArmor. If your system uses init scripts, run the following command:

```
$ sudo service apparmor restart
```

If instead, your system uses `systemd`, run the following command instead:

```
$ sudo systemctl restart apparmor
```

Installing Galera Cluster

There are two versions of Galera Cluster for MySQL: the original Codership reference implementation and MariaDB Galera Cluster. For each database server, binary packages are available for Debian- and RPM-based Linux distributions, or you can build them from source.

Galera Cluster for MySQL

Galera Cluster for MySQL - Binary Installation

Galera Cluster for MySQL is the reference implementation from Codership Oy. Binary installation packages are available for Linux distributions using `apt-get`, `yum` and `zypper` package managers through the Codership repository.

Enabling the Codership repository

In order to install Galera Cluster for MySQL through your package manager, you need to first enable the Codership repository on your system. There are different ways to accomplish this, depending on which Linux distribution and package manager you use.

Enabling the `apt` Repository

For Debian and Debian-based Linux distributions, the procedure for adding a repository requires that you first install the Software Properties. The package names vary depending on your distribution. For Debian, in the terminal run the following command:

```
# apt-get install python-software-properties
```

For Ubuntu or a distribution that derives from Ubuntu, instead run this command:

```
$ sudo apt-get install software-properties-common
```

In the event that you use a different Debian-based distribution and neither of these commands work, consult your distribution's package listings for the appropriate package name.

Once you have the Software Properties installed, you can enable the Codership repository for your system.

1. Add the GnuPG key for the Codership repository.

   ```
   # apt-key adv --keyserver keyserver.ubuntu.com \
         --recv BC19DDBA
   ```

2. Add the Codership repository to your sources list. Using your preferred text editor, create a *galera.list* file in the `/etc/apt/sources.list.d/` directory.

   ```
   # Codership Repository (Galera Cluster for MySQL)
   deb http://releases.galeracluster.com/DIST RELEASE main
   ```

   For the repository address, make the following changes:

   - `DIST` Indicates the name of your Linux distribution. For example, `ubuntu`.
   - `RELEASE` Indicates your distribution release. For example, `wheezy`.

     In the event that you do not know which release you have installed on your server, you can find out using the following command:

     ```
     $ lsb_release -a
     ```

3. Update the local cache.

   ```
   # apt-get update
   ```

Packages in the Codership repository are now available for installation through `apt-get`.

Enabling the `yum` Repository

For RPM-based distributions, such as CentOS, Red Hat and Fedora, you can enable the Codership repository by adding a `.repo` file to the `/etc/yum.repos.d/` directory.

Using your preferred text editor, create the `.repo` file.

```
[galera]
name = Galera
baseurl = http://releases.galeracluster.com/DIST/RELEASE/ARCH
gpgkey = http://releases.galeracluster.com/GPG-KEY-galeracluster.com
gpgcheck = 1
```

In the `baseurl` field, make the following changes to web address:

- `DIST` Indicates the distribution name. For example, `centos` or `fedora`.
- `RELEASE` indicates the distribution release number. For example, `6` for CentOS, `20` or `21` for Fedora.
- `ARCH` indicates the architecture of your hardware. For example, `x86_64` for 64-bit systems.

Packages in the Codership repository are now available for installation through `yum`.

Enabling the `zypper` Repository

For distributions that use `zypper` for package management, such as openSUSE and SUSE Linux Enterprise Server, you can enable the Codership repository by importing the GPG key and then creating a `.repo` file in the local directory.

1. Import the GPG key.

   ```
   $ sudo rpm --import "http://releases.galeracluster.com/GPG-KEY-galeracluster.com"
   ```

2. Create a `galera.repo` file in the local directory.

   ```
   [galera]
   name = Galera
   baseurl = http://releases.galeracluster.com/DIST/RELEASE
   ```

   For the `baseurl` repository address, make the following changes:

   - `DIST` indicates the distribution name. For example, `opensuse` or `sles`.
   - `RELEASE` indicates the distribution version number.

3. Add the Codership repository.

   ```
   $ sudo zypper addrepo galera.repo
   ```

4. Refresh `zypper`.

   ```
   $ sudo zypper refresh
   ```

Packages in the Codership repository are now available for installation through `zypper`.

Installing Galera Cluster for MySQL

There are two packages involved in the installation of Galera Cluster for MySQL: the MySQL database server, built to include the *wsrep API*; and the *Galera Replication Plugin*.

Note: For Debian-based distributions, you also need to include a third package, *Galera Arbitrator*. This is only necessary with `apt-get`. The `yum` and `zypper` repositories package Galera Arbitrator with the Galera Replication Plugin.

For Debian-based distributions, run the following command:

```
# apt-get install galera-3 \
             galera-arbitrator-3 \
             mysql-wsrep-5.6
```

For Red Hat, Fedora and CentOS distributions, instead run this command:

```
# yum install galera-3 \
             mysql-wsrep-5.6
```

Note: On CentOS 6 and 7, this command may generate a transaction check error. For more information on this error and how to fix it, see *MySQL Shared Compatibility Libraries* (page 16).

For openSUSE and SUSE Linux Enterprise Server, run this command:

```
# zypper install galera-3 \
             mysql-wsrep-5.6
```

Galera Cluster for MySQL is now installed on your server. You need to repeat this process for each node in your cluster.

Note: **See Also**: In the event that you installed Galera Cluster for MySQL over an existing standalone instance of MySQL, there are some additional steps that you need to take in order to update your system to the new database server. For more information, see *Migrating to Galera Cluster* (page 195).

MySQL Shared Compatibility Libraries

When installing Galera Cluster for MySQL on CentOS, versions 6 and 7, you may encounter a transaction check error that blocks the installation.

```
Transaction Check Error:
file /usr/share/mysql/czech/errmsg.sys from install
mysql-wsrep-server-5.6-5.6.23-25.10.e16.x86_64 conflicts
with file from package mysql-libs-5.1.73-.3.e16_5.x86_64
```

This relates to a dependency issue between the version of the MySQL shared compatibility libraries that CentOS uses and the one that Galera Cluster requires. Upgrades are available through the Codership repository and you can install them with `yum`.

There are two versions available for this package. The version that you need depends on which version of the MySQL wsrep database server that you want to install. Additionally, the package names themselves vary depending on the version of CentOS.

For CentOS 6, run the following command:

```
# yum upgrade -y mysql-wsrep-libs-compat-VERSION
```

Replace `VERSION` with `5.5` or `5.6`, depending upon the version of MySQL you want to use. For CentOS 7, to install MySQL version 5.6, run the following command:

```
# yum upgrade mysql-wsrep-shared-5.6
```

For CentOS 7, to install MySQL version 5.5, you also need to disable the 5.6 upgrade:

```
# yum upgrade -y mysql-wsrep-shared-5.5 \
      -x mysql-wsrep-shared-5.6
```

When `yum` finishes the upgrade, install the MySQL wsrep database server and the Galera Replication Plugin as described above.

Galera Cluster for MySQL - Source Installation

Galera Cluster for MySQL is the reference implementation from Codership Oy. Binary installation packages are available for Debian- and RPM-based distributions of Linux. In the event that your Linux distribution is based upon a different package management system, if your server uses a different unix-like operating system, such as Solaris or FreeBSD, you will need to build Galera Cluster for MySQL from source.

Note: See Also: In the event that you built Galera Cluster for MySQL over an existing standalone instance of MySQL, there are some additional steps that you need to take in order to update your system to the new database server. For more information, see *Migrating to Galera Cluster* (page 195).

Installing Build Dependencies

When building from source code, `make` cannot manage or install dependencies for either Galera Cluster or the build process itself. You need to install these first. For Debian-based systems, run the following command:

```
# apt-get build-dep mysql-server
```

For RPM-based distributions, instead run this command:

```
# yum-builddep MySQL-server
```

In the event that neither command works on your system or that you use a different Linux distribution or FreeBSD, the following packages are required:

- **MySQL Database Server with wsrep API**: Git, CMake, GCC and GCC-C++, Automake, Autoconf, and Bison, as well as development releases of libaio and ncurses.
- **Galera Replication Plugin**: SCons, as well as development releases of Boost, Check and OpenSSL.

Check with the repositories for your distribution or system for the appropriate package names to use during installation. Bear in mind that different systems may use different names and that some may require additional packages to run. For instance, to run CMake on Fedora you need both `cmake` and `cmake-fedora`.

Building Galera Cluster for MySQL

The source code for Galera Cluster for MySQL is available through GitHub. You can download the source code from the website or directly using `git`. In order to build Galera Cluster, you need to download both the database server with the wsrep API patch and the *Galera Replication Plugin*.

To download the database server, complete the following steps:

1. Clone the Galera Cluster for MySQL database server source code.

```
# git clone https://github.com/codership/mysql-wsrep
```

2. Checkout the branch for the version that you want to use.

```
# git checkout 5.6
```

The main branches available for Galera Cluster for MySQL are:

- `5.6`
- `5.5`

You now have the source files for the MySQL database server, including the wsrep API patch needed for it to function as a Galera Cluster node.

In addition to the database server, you need the wsrep Provider, also known as the Galera Replication Plugin. In a separator directory, run the following command:

```
# cd ..
# git clone https://github.com/codership/galera.git
```

Once Git finishes downloading the source files, you can start building the database server and the Galera Replication Plugin. The above procedures created two directories: `mysql-wsrep/` for the database server source and for the Galera source `galera/`

Building the Database Server

The database server for Galera Cluster is the same as that of the standard database servers for standalone instances of MySQL, with the addition of a patch for the wsrep API, which is packaged in the version downloaded from GitHub. You can enable the patch through the wsrep API, requires that you enable it through the `WITH_WSREP` and `WITH_INNODB_DISALLOW_WRITES` CMake configuration options.

To build the database server, `cd` into the `mysql-wsrep/` directory and run the following commands:

```
# cmake -DWITH_WSREP=ON -DWITH_INNODB_DISALLOW_WRITES=ON ./
# make
# make install
```

Building the wsrep Provider

The *Galera Replication Plugin* implements the *wsrep API* and operates as the wsrep Provider for the database server. What it provides is a certification layer to prepare write-sets and perform certification checks, a replication layer and a group communication framework.

To build the Galera Replicator plugin, `cd` into the `galera/` directory and run SCons:

```
# scons
```

This process creates the Galera Replication Plugin, (that is, the `libgalera_smm.so` file). In your `my.cnf` configuration file, you need to define the path to this file for the *wsrep_provider* (page 250) parameter.

Note: For FreeBSD users, building the Galera Replicator Plugin from source raises certain Linux compatibility issues. You can mitigate these by using the ports build at `/usr/ports/databases/galera`.

Post-installation Configuration

After the build completes, there are some additional steps that you must take in order to finish installing the database server on your system. This is over and beyond the standard configurations listed in *System Configuration* (page 27) and *Replication Configuration* (page 31).

Note: Unless you defined the `CMAKE_INSTALL_PREFIX` configuration variable when you ran `cmake` above, by default the database server installed to the path `/usr/local/mysql/`. If you chose a custom path, adjust the commands below to accommodate the change.

---

1. Create the user and group for the database server.

   ```
   # groupadd mysql
   # useradd -g mysql mysql
   ```

2. Install the database.

   ```
   # cd /usr/local/mysql
   # ./scripts/mysql_install_db --user=mysql
   ```

   This installs the database in the working directory. That is, at `/usr/local/mysql/data/`. If you would like to install it elsewhere or run it from a different directory, specify the desired path with the `--basedir` and `--datadir` options.

3. Change the user and group for the directory.

   ```
   # chown -R mysql /usr/local/mysql
   # chgrp -R mysql /usr/local/mysql
   ```

4. Create a system unit.

   ```
   # cp /usr/local/mysql/supported-files/mysql.server \
        /etc/init.d/mysql
   # chmod +x /etc/init.d/mysql
   # chkconfig --add mysql
   ```

   This allows you to start Galera Cluster using the `service` command. It also sets the database server to start during boot.

In addition to this procedure, bear in mind that any custom variables you enabled during the build process, such as a nonstandard base or data directory, requires that you add parameters to cover this in the configuration file, (that is, `my.cnf`).

Note: This tutorial omits MySQL authentication options for brevity.

MariaDB Galera Cluster

MariaDB Galera Cluster - Binary Installation

MariaDB Galera Cluster is the MariaDB implementation of Galera Cluster. Binary installation packages are available for Debian-based and RPM-based distributions of Linux through the MariaDB repository (MariaDB Repository Generator).

Enabling the MariaDB Repository

In order to install MariaDB Galera Cluster through your package manager, you need to enable the MariaDB repository on your server. There are two different ways to accomplish this, depending on which Linux distribution you use.

Enabling the `apt` Repository

For Debian and Debian-based Linux distributions, the procedure for adding a repository requires that you first install the software properties. The package names vary depending on your distribution. For Debian, at the command-line execute the following:

```
# apt-get install python-software-properties
```

For Ubuntu or a distribution derived from Ubuntu, execute instead this command:

```
$ sudo apt-get install software-properties-common
```

If you're use a different Debian-based distribution and neither of these lines above work, consult your distribution's package listings for the appropriate package name.

Once you have the software properties installed, you can enable the MariaDB repository for your server.

First, add the GnuPG key for the MariaDB repository by executing the following from the command-line:

```
# apt-key adv --recv-keys --keyserver \
      keyserver.ubuntu.com 0xcbcb082a1bb943db
```

Next, add the MariaDB repository to your sources list. You can do this by entering something like the following from the command-line:

```
# add-apt-repository 'deb http://mirror.jmu.edu/pub/mariadb/repo/version/distro␣
↪release main'
```

You wouldn't enter exactly the line above. You'll have to adjust the repository address:

- `version` indicates the version number of MariaDB that you want to use. (e.g., `5.6`).
- `distro` is the name of the Linux distribution you're using' (e.g., `ubuntu`).
- `release` should be changed to your distribution release (e.g., `wheezy`).

If you don't know which release is installed on your server, you can determine this by using the entering the following from the command-line:

```
$ lsb_release -a
```

1. You should also update the local cache on the server. You can do this by entering the following:

```
# apt-get update
```

For more information on the MariaDB repository, package names and available mirrors, see the MariaDB Repository Generator.

Packages in the MariaDB repository are now available for installation through `apt-get`.

Enabling the yum Repository

For RPM-based distributions (e.g., CentOS, Red Hat and Fedora), you can enable the MariaDB repository by creating a text file with `.repo` as the file extention to the `/etc/yum/repos.d/` directory.

Using a simple text editor, create a new `.repo` file containing something like the follwing:

```
# MariaDB.repo

[mariadb]
name = MariaDB
baseurl = http://yum.mariadb.org/version/package
gpgkey = https://yum.mariadb.org/RPM-GPG-KEY-MariaDB
gpgcheck = 1
```

For the value of `baseurl`, you'll have to adjust the web address:

- `version` should be changed to the version of MariaDB you want to use (e.g., `5.6`).
- `package` will have to be changed to the package name for your operating system distribution, release and architecture. For example, `rhel6-amd64` would reference packages for a Red Hat Enterprise Linux 6 server running on 64-bit hardware.

For more information on the repository, package names or available mirrors, see the MariaDB Repository Generator. It will generate the actual text you will need to put in your repository configuration file. In fact, by clicking through the choices presented, you can just copy the results and paste them into your configuration file without any modification.

Installing MariaDB Galera Cluster

There are three packages involved in the installation of MariaDB Galera Cluster: the MariaDB database client, a command-line tool for accessing the database; the MariaDB database server, built to include the *wsrep API* patch; and the *Galera Replication Plugin*.

For Debian-based distributions, from the command-line run the following command:

```
# apt-get install mariadb-client \
      mariadb-galera-server \
      galera
```

For RPM-based distributions, execute instead from the command line the following:

```
# yum install MariaDB-client \
      MariaDB-Galera-server \
      galera
```

Once you've done this, MariaDB Galera Cluster will be installed on your server. You'll need to repeat this process for each node in your cluster.

Note: See Also: If you installed MariaDB Galera Cluster over an existing stand-alone instance of MariaDB, there are some additional steps that you'll need to take to update your system to the new database server. For more information on this, see *Migrating to Galera Cluster* (page 195).

MariaDB Galera Cluster- Source Installation

MariaDB Galera Cluster is the MariaDB implementation of Galera Cluster for MySQL. Binary installation packages are available for Debian- and RPM-based distributions of Linux. In the event that your Linux distribution is based on a different package management system, or if it runs on a different unix-like operating system where binary installation packages are not available, such as Solaris or FreeBSD, you will need to build MariaDB Galera Cluster from source.

Note: See Also: In the event that you built MariaDB Galera Cluster over an existing standalone instance of MariaDB, there are some additional steps that you need to take in order to update your system to the new database server. For more information, see *Migrating to Galera Cluster* (page 195).

Preparing the Server

When building from source code, `make` cannot manage or install dependencies for either Galera Cluster or the build process itself. You need to install these packages first.

- For Debian-based distributions of Linux, if MariaDB is available in your repositories, you can run the following command:

```
# apt-get build-dep mariadb-server
```

- For RPM-based distributions, instead run this command:

```
# yum-builddep MariaDB-server
```

In the event that neither command works for your system or that you use a different Linux distribution or FreeBSD, the following packages are required:

- **MariaDB Database Server with wsrep API**: Git, CMake, GCC and GCC-C++, Automake, Autoconf, and Bison, as well as development releases of libaio and ncurses.
- **Galera Replication Plugin**: SCons, as well as development releases of Boost, Check and OpenSSL.

Check with the repositories for your distribution or system for the appropriate package names to use during installation. Bear in mind that different systems may use different names and that some may require additional packages to run. For instance, to run CMake on Fedora you need both `cmake` and `cmake-fedora`.

Building MariaDB Galera Cluster

The source code for MariaDB Galera Cluster is available through GitHub. Using Git you can download the source code to build MariaDB and the Galera Replicator Plugin locally on your system.

1. Clone the MariaDB database server repository.

```
# git clone https://github.com/mariadb/server
```

2. Checkout the branch for the version that you want to use.

```
# git checkout 10.0-galera
```

   The main branches available for MariaDB Galera Cluster are:

   - `10.1`
   - `10.0-galera`

- `5.5-galera`

Starting with version 10.1, MariaDB includes the wsrep API for Galera Cluster by default.

Note: **Warning**: MariaDB version 10.1 is still in beta.

You now have the source files for the MariaDB database server with the wsrep API needed to function as a Galera Cluster node.

In addition to the database server, you also need the wsrep Provider, also known as the Galera Replicator Plugin. In a separate directory run the following command:

```
# cd ..
# git clone https://github.com/codership/galera.git
```

Once Git finishes downloading the source files, you can start building the database server and the Galera Replicator Plugin. You now have the source files for the database server in a `server/` directory and the Galera source files in `galera/`.

Building the Database Server

The database server for Galera Cluster is the same as that of the standard database servers for standalone instances of MariaDB, with the addition of a patch for the wsrep API, which is packaged in the version downloaded from GitHub. You can enable the patch through the `WITH_WSREP` and `WITH_INNODB_DISALLOW_WRITES` CMake configuration options.

To build the database server, `cd` into the `server/` directory and run the following commands:

```
# cmake -DWITH_WSREP=ON -DWITH_INNODB_DISALLOW_WRITES=ON ./
# make
# make install
```

Note: In addition to compiling through `cmake` and `make`, there are also a number of build scripts in the `BUILD/` directory, which you may find more convenient to use. For example,

```
# ./BUILD/compile-pentium64-wsrep
```

This has the same effect as running the above commands with various build options pre-configured. There are several build scripts available in the directory, select the one that best suits your needs.

Building the wsrep Provider

The *Galera Replication Plugin* implements the *wsrep API* and operates as the wsrep Provider for the database server. What it provides is a certification layer to prepare write-sets and perform certification checks, a replication layer and a group communication framework.

To build the Galera Replication Plugin, `cd` into the `galera/` directory and run SCons.

```
# scons
```

This process creates the Galera Replication Pluigin, (that is, the `libgalera_smm.so` file). In your `my.cnf` configuration file, you need to define the path to this file for the *wsrep_provider* (page 250) parameter.

Note: For FreeBSD users, building the Galera Replication Plugin from source raises certain issues due to Linux dependencies. You can mitgate these by using the ports build available at `/usr/ports/databases/galera` or by installing the binary package:

```
# pkg install galera
```

Post-installation Configuration

After the build completes, there are some additional steps that you must take in order to finish installing the database server on your system. This is over and beyond the standard configuration process listed in *System Configuration* (page 27) and *Replication Configuration* (page 31).

Note: Unless you defined the `CMAKE_INSTALL_PREFIX` configuration variable when you ran `cmake` above, by default the database is installed to the path `/usr/local/mysql/`. If you chose a custom path, adjust the commands below to accommodate the change.

1. Create the user and group for the database server.

   ```
   # groupadd mysql
   # useradd -g mysql mysql
   ```

2. Install the database.

   ```
   # cd /usr/local/mysql
   # ./scripts/mysql_install_db --user=mysql
   ```

   This installs the database in the working directory, (that is, at `/usr/local/mysql/data`). If you would like to install it elsewhere or run the script from a different directory, specify the desired paths with the `--basedir` and `--datadir` options.

3. Change the user and group permissions for the base directory.

   ```
   # chown -R mysql /usr/local/mysql
   # chgrp -R mysql /usr/local/mysql
   ```

4. Create a system unit for the database server.

   ```
   # cp /usr/local/mysql/supported-files/mysql.server \
        /etc/init.d/mysql
   # chmod +x /etc/init.d/mysql
   # chkconfig --add mysql
   ```

   This allows you to start Galera Cluster using the `service` command. It also sets the database server to start during boot.

In addition to this procedure, bear in mind that any further customization variables you enabled during the build process, such as a nonstandard base or data directory, may require you to define additional parameters in the configuration file, (that is, `my.cnf`).

Note: This tutorial omits MariaDB authentication options for brevity.

Note: See Also: In the event that you build or install Galera Cluster over an existing standalone instance of MySQL or MariaDB, there are some additional steps that you need to take in order to update your system to the new database server. For more information, see *Migrating to Galera Cluster* (page 195).

CHAPTER

THREE

SYSTEM CONFIGURATION

After you finish installing Galera Cluster on your server, you're ready to configure the database itself to serve as a node in a cluster. To do this, you'll need to edit the MySQL configuration file.

Using a text editor, edit the `/etc/my.cnf` file. You'll need to include entries like the ones shown in this sample excerpt:

```
[mysqld]
datadir=/var/lib/mysql
socket=/var/lib/mysql/mysql.sock
user=mysql
binlog_format=ROW
bind-address=0.0.0.0
default_storage_engine=innodb
innodb_autoinc_lock_mode=2
innodb_flush_log_at_trx_commit=0
innodb_buffer_pool_size=122M
wsrep_provider=/usr/lib/libgalera_smm.so
wsrep_provider_options="gcache.size=300M; gcache.page_size=300M"
wsrep_cluster_name="example_cluster"
wsrep_cluster_address="gcomm://IP.node1,IP.node2,IP.node3"
wsrep_sst_method=rsync

[mysql_safe]
log-error=/var/log/mysqld.log
pid-file=/var/run/mysqld/mysqld.pid
```

Depending on your system and the location of your installation of MySQL or MariaDB, you will need to adjust the valuables for variables (e.g., the path to the data directory).

Configuring the Database Server

In addition to settings for the system, there are other basic configurations that you will need to set in the `/etc/my.cnf` file. Make these changes before starting the database server.

First, make sure that `mysqld` is not bound to 127.0.0.1. This is the IP address for localhost. If the `bind-address` variable is in the file, comment it out by adding a hash sign (i.e., `#`) at the start of the line:

```
# bind-address = 127.0.0.1
```

Next, ensure the configuration file includes the `conf.d/` by adding a line with `!includedir` at the start, followed by the file path:

```
!includedir /etc/mysql/conf.d/
```

Now, set the binary log format to use row-level replication, as opposed to statement-level replication. You'd do this by adding the following line:

```
binlog_format=ROW
```

Don't change this value later as it affects performance and consistency. The binary log can only use row-level replication for Galera Cluster.

Galera Cluster will not work with MyISAM or other non-transactional storage engines. So, make sure the default storage engine is InnoDB using the `default_storage_engine` variable like so:

```
default_storage_engine=InnoDB
```

Next, ensure the InnoDB locking mode for generating auto-increment values is set to interleaved lock mode. This is designated by a value of `2` for the appropriate variable:

```
innodb_autoinc_lock_mode=2
```

Don't change this value afterwards. Other modes may cause `INSERT` statements to fail on tables with `AUTO_INCREMENT` columns.

Note: Warning: When innodb_autoinc_lock_mode is set to traditional lock mode (i.e., a value of `0`) or to consecutive lock mode (i.e., a value of `1`) it can cause unresolved deadlocks and make the system unresponsive in Galera Cluster.

After all of that, make sure the InnoDB log buffer is written to file once per second, rather than on each commit, to improve performance. To do this, set the `innodb_flush_log_at_trx_commit` variable to 0 like so;

```
innodb_flush_log_at_trx_commit=0
```

Note: Warning: Although setting innodb_flush_log_at_trx_commit to a value of `0` or `2` improves performance, it also introduces potential problems. Operating system crashes or power outages can erase the last second of transaction. Although normally you can recover this data from another node, it can still be lost entirely in the event that the cluster goes down at the same time.

After you make all of these changes and additions to the configuration file, you're ready to configure the database privileges.

Configuring the InnoDB Buffer Pool

The InnoDB storage engine uses its own memory buffer to cache data and for indexes of tables. You can configure this memory buffer through the innodb_buffer_pool_size parameter. The default value is 128 MB. To compensate for the increased memory usage of Galera Cluster over a standalone MySQL database server, you should scale your usual value back by five percent.

```
innodb_buffer_pool_size=122M
```

Configuring Swap Space

Memory requirements for Galera Cluster are difficult to predict with any precision. The particular amount of memory it uses can vary significantly, depending upon the load the given node receives. In the event that Galera Cluster attempts to use more memory than the node has available, the `mysqld` instance will crash.

The way to protect a node from such crashes is to ensure that there is sufficient swap space available on the server. This can be either in the form of a swap partition or swap files. To check the available swap space, execute the following from the command-line:

```
# swapon --summary

Filename        Type        Size     Used    Priority
/dev/sda2       partition   3369980  0       -1
/swap/swap1     file        524284   0       -2
/swap/swap2     file        524284   0       -3
```

If swap is not configured, nothing will be returned from this command. If your system doesn't have swap space available or if the allotted space is insufficient, you can fix this by creating swap files.

First, create an empty file on your disk, set the file size to whatever size you require. You can do this with the `fallocate` tool like so:

```
# fallocate -l 512M /swapfile
```

Alternatively, you can manage the same using `dd` utility like this:

```
# dd if=/dev/zero of=/swapfile bs=1M count=512
```

Be sure to secure the swap file by changing the permissions on the filesystem with `chmod` like this:

```
# chmod 600 /swapfile

$ ls -a / | grep swapfile

-rw------- 1 root root 536870912 Feb 12 23:55 swapfile
```

This sets the file permissions so that only the root user can read and write to the file. No other user or group member can access it. Using the `ls` command command above shows the results.

Now you're read to format the swap file. You can do this with the `mkswap` utility. You'll then need to active the swap file.

```
# mkswap /swapfile
# swapon /swapfile
```

Using a text editor, update the `/etc/fstab` file to include the swap file by adding the following line to the bottom:

```
/swapfile none swap defaults 0 0
```

After you save the `/etc/fstab` file, you run `swapon` again to see the results:

```
$ swapon --summary
Filename        Type        Size     Used    Priority
/swapfile       file        524284   0       -1
```

CHAPTER

FOUR

REPLICATION CONFIGURATION

In addition to the configuration for the database server, there are some specific options that you need to set to enable write-set replication. You must apply these changes to the configuration file (i.e., `my.cnf`) for each node in the cluster.

- *wsrep_cluster_name* (page 238): Use this parameter to set the logical name for the cluster. You must use the same name for each node in the cluster. The connection will fail on nodes that have different values for this parameter.
- *wsrep_cluster_address* (page 237): Use this parameter to define the IP addresses for the cluster in a comma-separated list.

  Note: See Also: There are additional schemata and options available through this parameter. For more information on the syntax, see Understanding Cluster Addresses below.

- *wsrep_node_name* (page 247): Use this parameter to define the logical name for the individual node—for convenience.
- *wsrep_node_address* (page 246): Use this parameter to set explicitly the IP address for the individual node. It's used when auto-guessing doesn't produce desirable results.

```
[mysql]
wsrep_cluster_name=MyCluster
wsrep_cluster_address="gcomm://192.168.0.1,192.168.0.2,192.168.0.3"
wsrep_node_name=MyNode1
wsrep_node_address="192.168.0.1"
```

Backend Schema

There are two backend schemata available with Galera Cluster.

- `dummy`: This provides a pass-through back-end for testing and profiling purposes. It doesn't connect to other nodes and will ignore any values given to it.
- `gcomm`: This provides the group communications back-end for use in production. It accepts an address and has several settings that may be enabled through the option list, or by using the *wsrep_provider_options* (page 251) parameter.

Cluster Addresses

For the cluster address section, you have to provide a comma-separate list of IP addresses for all of the nodes in the cluster. You would do this using the *wsrep_cluster_address* (page 237) parameter. Cluster addresses are listed in the configuration file using a particular syntax, like so:

```
<backend schema>://<cluster address>[?<option1>=<value1>[&<option2>=<value2>]]
```

Below is an example of how this line from the configuration file might look:

```
wsrep_cluster_address="gcomm://192.168.0.1,192.168.0.2,192.168.0.3"
```

Here, the backend schema is `gcomm`. The cluster addresses (i.e., `192.168.0.1`, etc.) are listed next, separted by commas. You can add options after that, within the quotes. You would start with a question mark, followed by each option setting. Option key/value pairs are separated by an ampersand. This is covered in the Options section below.

The IP addresses given in the configuration file should include any current members of the cluster. The list may also include the IP addresses of any possible cluster members. Members can belong to no more than one Primary Component;

If you start a node without proving an IP address for this parameter, the node will assume that it's the first node of a new cluster. It will initialize the cluster as though you launched `mysqld` with the `--wsrep-new-cluster` option.

Options

When setting the IP address in the configuration file using the *wsrep_cluster_address* (page 237) parameter, you can also set some options. You can set backend parameters, such as the listen address and timeout values.

Note: **See Also**: The *wsrep_cluster_address* (page 237) options list is not durable. The node must resubmit the options on each connection to a cluster. To make these options durable, set them in the configuration file using the *wsrep_provider_options* (page 251) parameter.

The options set in the URL take precedent over parameters set elsewhere. Parameters you set through the options list are prefixed by `evs` (i.e., Extended Virtual Synchrony), `pc` (i.e., Primary Component) and `gmcast`.

Note: **See Also**: For more information on the available parameters, see *Galera Parameters* (page 265).

When listing options, start with a question mark after the IP address list. Then provide the options in a `key=value` format. Key/value pairs must be separated by an ampersand. Below is an example of how this might look:

```
wsrep_cluster_address="gcomm://192.168.0.1, 192.168.0.2, 192.168.0.3 ? gmcast.
↪segment=0 & evs.max_install_timeouts=1"
```

In this example, the `segment` option for `gcomm` and the `max_install_timeouts` option for `evs` are set.

Incidentally, if the listen address and port are not set in the parameter list, `gcomm` will listen on all interfaces. The listen port will be taken from the cluster address. If it's not specified in the cluster address, the default port is `4567`.

CHAPTER

FIVE

STARTING THE CLUSTER

After you finish installing MySQL (or MariaDB or Percona XtraDB to Galera Cluster) and Galera and have added the necessary settings for the configuration file needed for Galera Cluster, the next steps are to start the nodes that will form the cluster. To do this, you will need to start the `mysqld` daemon on one node, using the `--wsrep-new-cluster` option. This initializes the new *Primary Component* for the cluster. Each node you start after that will connect to the component and begin replication.

Before you attempt to initialize the cluster, there are a few things you should verify are in place on each node and related services:

- At least three servers with the same version of MySQL, MariaDB, or Percona XtraDB installed on each;
- If you're using firewalls, make sure the ports 4444, 4567, and 4568 for TCP traffic, and 4567 for UPD traffic are open between the hosts;
- SELinux and AppArmor, whichever your system uses or both, has to be set to allow access to `mysqld`; and,
- Set the parameter for *wsrep_provider* (page 250) to the location of `libgalera_smm.so`. That line in the configuration file might look like this:

```
wsrep_provider=/usr/lib64/libgalera_smm.so
```

Once you have at least three hosts ready, you can initialize the cluster.

Note: **See Also**: When migrating from an existing, stand-alone instance of MySQL, MariaDB or Percona XtraDB to Galera Cluster, there will be some additional steps that you must take. For more information on what you need to do, see *Migrating to Galera Cluster* (page 195).

Starting the First Node

By default, a node don't start as part of the *Primary Component*. Instead, it assumes that the Primary Component is already running and it is merely joining an existing cluster. For each node it encounters in the cluster, it checks whether or not it's a part of the Primary Component. When it finds the Primary Component, it requests a state transfer to bring its database into sync with the cluster. If it can't find the Primary Component, it will remains in a non-operational state.

The problem is that there is no Primary Component when a cluster starts, when the first node is initiated. Therefore, you need explicitly to tell that first node to do so with the `--wsrep-new-cluster` argument. Althought this initiate node is said to be the first node, it can fall behind and leave the cluster without necessarily affecting the Primary Component.

Note: See Also: When you start a new cluster, any node can serve as the first node, since all the databases are empty. When you migrate from MySQL to Galera Cluster, use the original master node as the first node. When restarting the cluster, use the most advanced node. For more information, see *Migrating to Galera Cluster* (page 195) and *Resetting the Quorum* (page 103).

To start the first node–which should have MySQL, MariaDB or Percona XtraDB, and Galera installed–you'll have to launch the database server on it with the `--wsrep-new-cluster` option. There are a few ways you might do this, depending on the operating system. For systems that use `init`, execute the following from the command-line:

```
# service mysql start --wsrep-new-cluster
```

For operating systems that use `systemd`, you would instead enter the following from the command-line:

```
# systemctl start mysql --wsrep-new-cluster
```

Both of these start the `mysqld` daemon on the node. Starting in MariaDB version 10.4, which includes Galera version 4, you can enter instead the following from the command-line to start MariaDB, Galera, and to establish the Primary Component:

```
# galera_new_cluster
```

Note: Warning: Use the `--wsrep-new-cluster` argument only when initializing the Primary Component. Don't use it to connect a new node to an existing cluster.

Once the first node starts the database server, verify that the cluster has started, albeit a one-node cluster, by checking *wsrep_cluster_size* (page 290). With the database client, execute the following SQL statement:

```
SHOW STATUS LIKE 'wsrep_cluster_size';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 1     |
+--------------------+-------+
```

This status variable indicates the number of nodes that are connected to the cluster. Since only the first node has been started, the value is `1` here. After you start other nodes that will be part of this same cluster, execute this SQL statement again–on thee first node or any node you've verified are in the cluster. The value should reflect the number of nodes in the cluster.

Once you get the first node started and the Primary Component initialized, don't restart `mysqld`. Instead, wait until you've added more nodes to the cluster so that it can stay viable without the first node. If you must restart the first node before adding other nodes, shutdown `mysqld` and then bootstrap start it again (e.g., execute `galera_new_cluster`). If it won't start as easily as it did the first time, you may have to edit the file containing the Galera Saved State (i.e., /var/lib/mysql/grastate.dat). The contents of that file will look something like this:

```
# GALERA saved state
version: 2.1
uuid:    bd5fe1c3-7d80-11e9-8913-4f209d688a15
seqno:   -1
safe_to_bootstrap: 0
```

The variable `safe_to_bootstrap` is set to 0 on the first node after it's been bootstrapped to protect against you inadvertently bootstrapping again while the cluster is runnning. You'll have to change the value to 1 to be able to

bootstrap anew.

Adding Nodes to the Cluster

Once you have successfully started the first node and thereby initialized a new cluster, the procedure for adding all the other nodes is even simpler. You just launch `mysqld` as you would normally–without the `--wsrep-new-cluster` option. You would enter something like the following from the command-line, depending on your operating system and database system (see above for other methods):

```
# systemctl start mariadb
```

When the database server initializes as a new node, it will try to connect to the cluster members. It knows where to find these other nodes based on the IP addresses listed in the *wsrep_cluster_address* (page 237) parameter in the configuration file.

You can verify that the node connection was successful checking the *wsrep_cluster_size* (page 290) status variable. In the database client of any node in the cluster, run the following SQL statement:

```
SHOW STATUS LIKE 'wsrep_cluster_size';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 2     |
+--------------------+-------+
```

This indicates that the two nodes are now connected to the cluster. When the nodes in the cluster agree on the membership state, they initiate state exchange. In state exchange, a new node will check the cluster state. If the state of a new node differs from the cluster state–which is normally the case–the new node requests a state snapshot transfer (SST) from the cluster and it installs it on its local database. After this is done, the new node is ready for use.

CHAPTER

SIX

TESTING A CLUSTER

When you have a cluster running, you may want to test certain features to ensure that they are working properly or to prepare yourself for handling actual problems that may arise.

Replication Testing

There are a few step to do to test that Galera Cluster is working as expected. First, using the database client, verify that all nodes have connected to each other. To do this, execute the `SHOW STATUS` statement like so:

```
SHOW STATUS LIKE 'wsrep_%';

+----------------------------+------------+
| Variable_name              | Value      |
+----------------------------+------------+
...
| wsrep_local_state_comment  | Synced (6) |
| wsrep_cluster_size         | 3          |
| wsrep_ready                | ON         |
+----------------------------+------------+
```

Because of the `LIKE` operator, only variables beginning with `wsrep_` are returned. The three variables pertinent here are the ones shown in the example above.

- *wsrep_local_state_comment* (page 300): The value `Synced` indicates that the node is connected to the cluster and operational.
- *wsrep_cluster_size* (page 290): The numeric value returned indicates the number of nodes in the cluster.
- *wsrep_ready* (page 302): A value of `ON` indicates that this node in which the SQL statement was executed is connected to the cluster and able to handle transactions.

For the next test, try creating a table and inserting data into it. Use a database client on `node1` to enter these SQL statements:

```
CREATE DATABASE galeratest;

USE galeratest;

CREATE TABLE test_table (
    id INT PRIMARY KEY AUTO_INCREMENT, msg TEXT ) ENGINE=InnoDB;

INSERT INTO test_table (msg) VALUES ("Hello my dear cluster.");
```

INSERT INTO test_table (msg) VALUES ("Hello, again, cluster dear.");

These statements will create the database `galeratest` and the table `test_table` within it. The last two SQL statements inserts data into that table. After doing this, log into `node2` and check that the data was replicated correctly. You would do this with by executing the following SQL statement on `node2`:

```
```

SELECT * FROM galeratest.test_table;

id	msg
1 2	Hello my dear cluster. Hello, again, cluster dear.

The results returned from the `SELECT` statement indicates that the data entered on `node1` was replicated on `node2`.

Split-Brain Testing

There are a few steps to test Galera Cluster for split-brain situations on a two-node cluster. First, disconnect the network connection between the two nodes. At this point, the quorum will be lost and the nodes won't serve requests.

Now, reconnect the network connection. The quorum will remain lost and the nodes still won't serve requests.

To reset the quorum, on one of the database clients, execute the following SQL statement:

```
```

SET GLOBAL wsrep_provider_options='pc.bootstrap=1';

At this point the quorum should be reset and the cluster recovered.

Failure Simulation

You can also test Galera Cluster by simulating various failure situations on three nodes. To simulate a crash of a single `mysqld` process, execute the following from the command-line on one of the nodes:

```
$ killall -9 mysqld
```

To simulate a network disconnection, use `iptables` or `netem` to block all TCP/IP traffic to a node.

To simulate an entire server crash, run each `mysqld` in a virtualized guest, and abrubtly terminate the entire virtual instance.

If you have three or more Galera Cluster nodes, the cluster should be able to survive the simulations.

CHAPTER

SEVEN

RESTARTING THE CLUSTER

Occasionally, you may have to restart an entire Galera Cluster. This may happen, for example, when there is a power failure in which every node is shut down and you have no `mysqld` process. Restarting a cluster, starting nodes in the wrong order, starting the wrong nodes first, can be devastating and lead to loss of data.

When restarting an entire Galera Cluster, you'll need to determine which node has the most advanced node state ID. This is covered in the next section. Once you've identified the most advanced node, you'll need to start that node first. Then you can start the rest of the nodes in any order. They will each look to the first node as the most up-to-date node.

Identifying the Most Advanced Node

Identifying the most advanced node state ID is done by comparing the *Global Transaction ID* values on each node in your cluster. You can find this in the `grastate.dat` file, located in the data directory for your database.

If the `grastate.dat` file looks like the example below, you have found the most advanced node state ID:

```
# GALERA saved state
version: 2.1
uuid:    5ee99582-bb8d-11e2-b8e3-23de375c1d30
seqno:   8204503945773
cert_index:
```

To find the sequence number of the last committed transaction, run `mysqld` with the `--wsrep-recover` option. This recovers the InnoDB table space to a consistent state, prints the corresponding Global Transaction ID value into the error log, and then exits. Here's an example of this:

```
130514 18:39:13 [Note] WSREP: Recovered position: 5ee99582-bb8d-11e2-b8e3-
23de375c1d30:8204503945771
```

This value is the node state ID. You can use it to update manually the `grastate.dat` file, by entering it in the value of the `seqno` field. As an alternative, you can just let `mysqld_safe` recover automatically and pass the value to your database server the next time you start it.

Identifying Crashed Nodes

You can easily determine if a node has crashed by looking at the contents of the `grastate.dat` file. If it looks like the example below, the node has either crashed during execution of a non-transactional operation (e.g., `ALTER TABLE`), or the node aborted due to a database inconsistency.

```
# GALERA saved state
version: 2.1
uuid:    5ee99582-bb8d-11e2-b8e3-23de375c1d30
seqno:   -1
cert_index:
```

It's possible for you to recover the *Global Transaction ID* of the last committed transaction from InnoDB, as described above. However, the recovery is rather meaningless. After the crash, the node state is probably corrupted and may not prove functional.

If there are no other nodes in the cluster with a well-defined state, there is no need to preserve the node state ID. You must perform a thorough database recovery procedure, similar to that used on stand-alone database servers. Once you recover one node, use it as the first node in a new cluster.

Part II

Technical Description

Galera Cluster is a synchronous certification-based replication solution for MySQL, MariaDB and Percona XtraDB. Cluster nodes are identical and fully representative of the cluster state. They allow for unconstrained transparent client access, acting as a single-distributed database server. In order to better understand Galera Cluster, this section provides detailed information on how it works and how you can benefit from it.

Understanding Replication

Replication in the context of databases refers to the frequent copying of data from one database server to another. These sections provide a high-level explanation of replication both in the general sense of how it works, as well as the particulars of how Galera Cluster implements these core concepts.

- *Database Replication* (page 45)

  This section explains how database replication works in general. It provides an overview of the problems inherent in the various replication implementations, including master-slave, asynchronous and synchronous replication.

- *Certification-Based Replication* (page 49)

  Using group communications and transaction ordering techniques, certification-based replication allows for synchronous replication.

Understanding Galera Cluster

With a better grasp on how replication works, these pages provide a more detailed explanation of how Galera Cluster implements certification-based replication, including the specific architecture of the nodes, how they communicate with each other, as well as replicate data and manage the replication process.

- *Replication API* (page 51)

  While the above sections explain the abstract concepts surrounding certification-based replication, this section covers the specific architecture used by Galera Cluster in implementing write-set replication, including the wsrep API and the Galera Replication and Group Communication plug-ins.

- *Isolation Levels* (page 55)

  In a database system, the server will process concurrent transactions in isolation from each other. The level of isolation determines whether and how these transactions affect one another. This section provides an overview of the isolation levels supported by Galera Cluster.

- *State Transfers* (page 57)

  The actual process that nodes use to replicate data into each other is called provisioning. Galera Cluster supports two provisioning methods: State Snapshot Transfers and Incremental State Transfers. This section presents an overview of each.

- *Flow Control* (page 61)

  Galera Cluster manages the replication process using a feedback mechanism called Flow Control. This allows the node to pause and resume replication according to its performance needs and to prevent any node from lagging too far behind the others in applying transaction. This section provides an overview of Flow Control and the different states nodes can hold.

- *Node Failure and Recovery* (page 65)

  Nodes fail to operate when they lose their connection with the cluster. This can occur for various reasons, such as hardware failures, software crashes, or the loss of network connectivity. This section provides an overview of how nodes and the cluster cope with failure and how they may recover.

- *Weighted Quorum* (page 67)

  When nodes connect to each other, they form components. The Primary Component is a component that has quorum: it carries the majority of nodes in the cluster. By default, each node represents one vote in quorum calculations. However, you can modify this feature in order to ensure certain stable nodes with strong connections carry a greater value. This section provides an overview of how Galera Cluster handles weighted values in quorum calculations.

- *Streaming Replication* (page 73)

  Normally, nodes transfer all replication and certification events on the transaction commit. With Streaming Replication, the nodes break the transaction into fragments. Then they certify, replicate and apply these replication fragments onto the slave nodes. This section describes Streaming Replication, how it works and the limitations of its use.

CHAPTER

EIGHT

DATABASE REPLICATION

Database replication refers to the frequent copying of data from one node—a database on a server—into another. Think of a database replication system as a distributed database, where all nodes share the same level of information. This system is also known as a *database cluster*.

The database clients, such as web browsers or computer applications, do not see the database replication system, but they benefit from close to native DBMS (Database Management System) behavior.

Masters and Slaves

Many DATABASE MANAGEMENT SYSTEMS (DBMS) replicate the database.

The most common replication setup uses a master/slave relationship between the original data set and the copies.

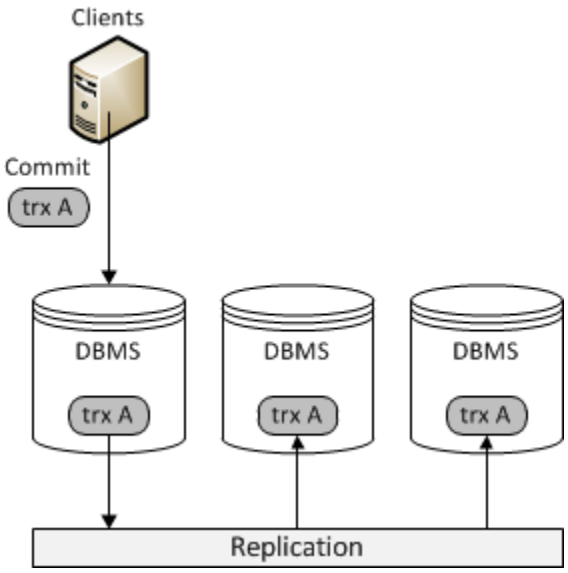


Fig. 8.1: *Master/Slave Replication*

In this system, the master database server logs the updates to the data and propagates those logs through the network to the slaves. The slave database servers receive a stream of updates from the master and apply those changes.

Another common replication setup uses mult-master replication, where all nodes function as masters.

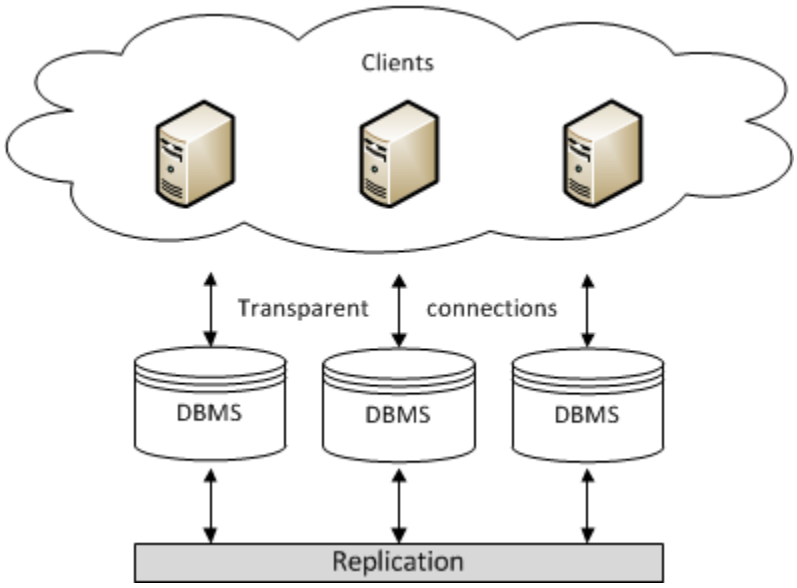


Fig. 8.2: *Multi-master Replication*

In a multi-master replication system, you can submit updates to any database node. These updates then propagate through the network to other database nodes. All database nodes function as masters. There are no logs available and the system provides no indicators sent to tell you if the updates were successful.

Asynchronous and Synchronous Replication

In addition to the setup of how different nodes relate to one another, there is also the protocol for how they propagate database transactions through the cluster.

- **Synchronous Replication** Uses the approach of eager replication. Nodes keep all replicas synchronized by updating all replicas in a single transaction. In other words, when a transaction commits, all nodes have the same value.
- **Asynchronous Replication** Uses the approach of lazy replication. The master database asynchronously propagates replica updates to other nodes. After the master node propagates the replica, the transaction commits. In other words, when a transaction commits, for at least a short time, some nodes hold different values.

Advantages of Synchronous Replication

In theory, there are several advantages that synchronous replication has over asynchronous replication. For instance:

- **High Availability** Synchronous replication provides highly available clusters and guarantees 24/7 service availability, given that:
  - No data loss when nodes crash.
  - Data replicas remain consistent.
  - No complex, time-consuming failovers.
- **Improved Performance** Synchronous replications allows you to execute transactions on all nodes in the cluster in parallel to each other, increasing performance.

- **Causality across the Cluster** Synchronous replication guarantees causality across the whole cluster. For example, a `SELECT` query issued after a transaction always sees the effects of the transaction, even if it were executed on another node.

Disadvantages of Synchronous Replication

Traditionally, eager replication protocols coordinate nodes one operation at a time. They use a two phase commit, or distributed locking. A system with n number of nodes due to process o operations with a throughput of t transactions per second gives you m messages per second with:

$$m = n \times o \times t$$

What this means that any increase in the number of nodes leads to an exponential growth in the transaction response times and in the probability of conflicts and deadlock rates.

For this reason, asynchronous replication remains the dominant replication protocol for database performance, scalability and availability. Widely adopted open source databases, such as MySQL and PostgreSQL only provide asynchronous replication solutions.

Solving the Issues in Synchronous Replication

There are several issues with the traditional approach to synchronous replication systems. Over the past few years, researchers from around the world have begun to suggest alternative approaches to synchronous database replication.

In addition to theory, several prototype implementations have shown much promise. These are some of the most important improvements that these studies have brought about:

- **Group Communication** This is a high-level abstraction that defines patterns for the communication of database nodes. The implementation guarantees the consistency of replication data.
- **Write-sets** This bundles database writes in a single write-set message. The implementation avoids the coordination of nodes one operation at a time.
- **Database State Machine** This processes read-only transactions locally on a database site. The implementation updates transactions are first executed locally on a database site, on shallow copies, and then broadcast as a read-set to the other database sites for certification and possibly commits.
- **Transaction Reordering** This reorders transactions before the database site commits and broadcasts them to the other database sites. The implementation increases the number of transactions that successfully pass the certification test.

The certification-based replication system that Galera Cluster uses is built on these approaches.

CHAPTER

NINE

CERTIFICATION-BASED REPLICATION

Certification-based replication uses group communication and transaction ordering techniques to achieve synchronous replication.

Transactions execute optimistically in a single node, or replica, and then at commit time, they run a coordinated certification process to enforce global consistency. It achieves global coordination with the help of a broadcast service that establishes a global total order among concurrent transactions.

Certification-Based Replication Requirements

It's not possible to implement certification-based replication for all database systems. It requires certain features of the database in order to work;

- **Transactional Database:** The database must be transactional. Specifically, it has to be able to rollback uncommitted changes.
- **Atomic Changes:** Replication events must be able to change the database, atomically. All of a series of database operations in a transaction must occur, else nothing occurs.
- **Global Ordering:** Replication events must be ordered globally. Specifically, they are applied on all instances in the same order.

How Certification-Based Replication Works

The main idea in certification-based replication is that a transaction executes conventionally until it reaches the commit point, assuming there is no conflict. This is called optimistic execution.

When the client issues a `COMMIT` command, but before the actual commit occurs, all changes made to the database by the transaction and primary keys of the changed rows, are collected into a write-set. The database then sends this write-set to all of the other nodes.

The write-set then undergoes a deterministic certification test, using the primary keys. This is done on each node in the cluster, including the node that originates the write-set. It determines whether or not the node can apply the write-set.

If the certification test fails, the node drops the write-set and the cluster rolls back the original transaction. If the test succeeds, though, the transaction commits and the write-set is applied to the rest of the cluster.

Certification-Based Replication in Galera Cluster

The implementation of certification-based replication in Galera Cluster depends on the global ordering of transactions.

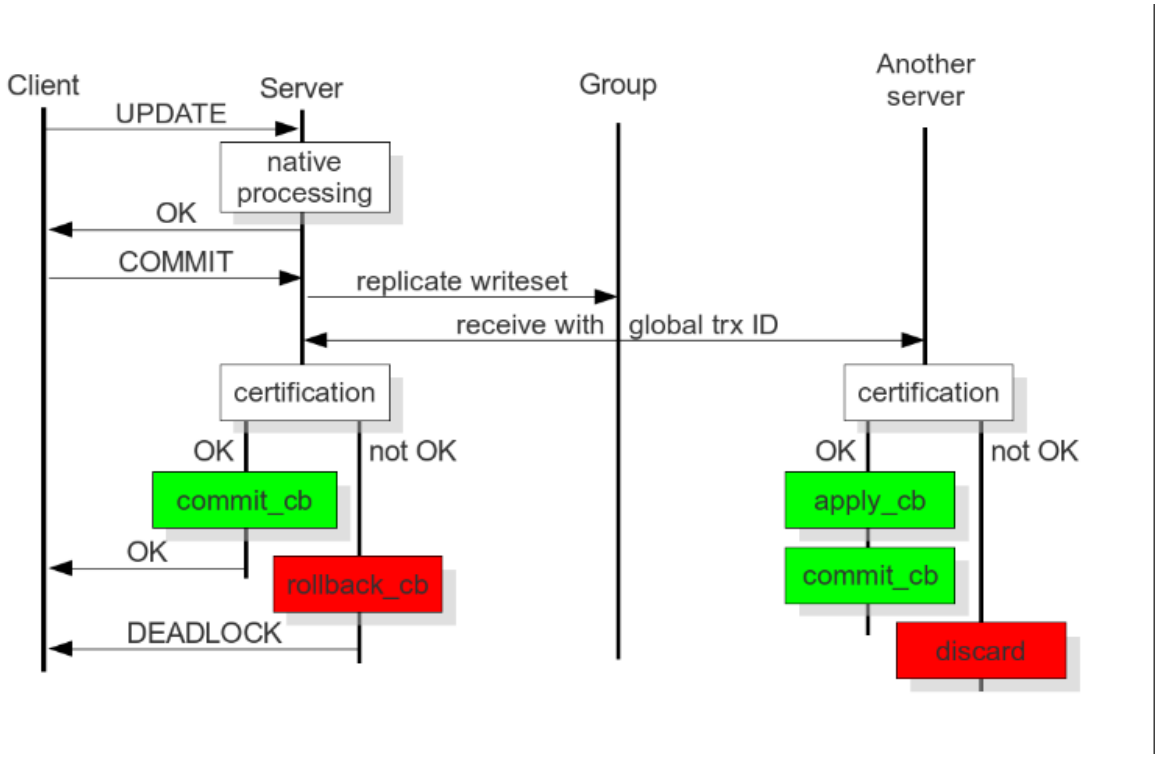


Fig. 9.1: *Certification Based Replication*

Galera Cluster assigns each transaction a global ordinal sequence number, or `seqno`, during replication. When a transaction reaches the commit point, the node checks the sequence number against that of the last successful transaction. The interval between the two is the area of concern, given that transactions that occur within this interval have not seen the effects of each other. All transactions in this interval are checked for primary key conflicts with the transaction in question. The certification test fails if it detects a conflict.

The procedure is deterministic and all replica receive transactions in the same order. Thus, all nodes reach the same decision about the outcome of the transaction. The node that started the transaction can then notify the client application whether or not it has committed the transaction.

CHAPTER

TEN

REPLICATION API

Synchronous replication systems generally use eager replication. Nodes in a cluster will synchronize with all of the other nodes by updating the replicas through a single transaction. This means that when a transaction commits, all of the nodes will have the same value. This process takes place using *write-set* replication through group communication.

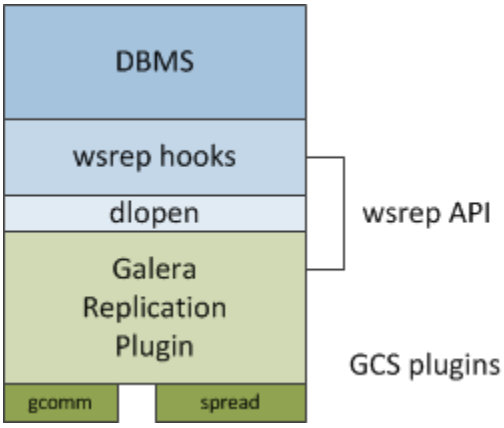


Fig. 10.1: *Replication API*

The internal architecture of Galera Cluster revolves around four components:

- **Database Management System (DBMS):** The database server that runs on an individual node. Galera Cluster can use MySQL, MariaDB or Percona XtraDB.
- **wsrep API:** This is the interface to the database server and it's the replication provider. It consists of two main elements:
- *wsrep Hooks:* This integrates with the database server engine for write-set replication.
- *dlopen():* This function makes the wsrep provider available to the wsrep hooks.
- **Galera Replication Plugin:** This plugin enables write-set replication service functionality.
- **Group Communication Plugins:** There several group communication systems available to Galera Cluster (e.g., *gcomm* and Spread).

wsrep API

The *wsrep API* is a generic replication plugin interface for databases. It defines a set of application callbacks and replication plugin calls.

The wsrep API uses a replication model that considers the database server to have a state. That state refers to the contents of the database. When a database is in use and clients modify the database content, its state is changed. The wsrep API represents changes in the database state as a series of atomic changes, or transactions.

In a database cluster, all of the nodes always have the same state. They synchronize with each other by replicating and applying state changes in the same serial order.

From a more technical perspective, Galera Cluster handles state changes in the following way:

1. On one node in the cluster, a state change occurs in the database.
2. In the database, the wsrep hooks translate the changes to the write-set.
3. `dlopen()` then makes the wsrep provider functions available to the wsrep hooks.
4. The Galera Replication plugin handles write-set certification and replication to the cluster.

For each node in the cluster, the application process occurs by high-priority transactions.

Global Transaction ID

In order to keep the state identical across the cluster, the wsrep API uses a *Global Transaction ID*, or GTID. This allows it to identify state changes and to identify the current state in relation to the last state change. Below is an example of a GTID:

```
45eec521-2f34-11e0-0800-2a36050b826b:94530586304
```

The Global Transaction ID consists of the following components:

- **State UUID** This is a unique identifier for the state and the sequence of changes it undergoes.
- **Ordinal Sequence Number:** The seqno is a 64-bit signed integer used to denote the position of the change in the sequence.

The Global Transaction ID allows you to compare the application state and establish the order of state changes. You can use it to determine whether or not a change was applied and whether the change is applicable to a given state.

Galera Replication Plugin

The *Galera Replication Plugin* implements the *wsrep API*. It operates as the wsrep Provider. From a more technical perspective, the Galera Replication Plugin consists of the following components:

- **Certification Layer:** This layer prepares the write-sets and performs the certification checks on them, ensuring that they can be applied.
- **Replication Layer:** This layer manages the replication protocol and provides the total ordering capability.
- **Group Communication Framework:** This layer provides a plugin architecture for the various group communication systems that connect to Galera Cluster.

Group Communication Plugins

The Group Communication Framework provides a plugin architecture for the various gcomm systems.

Galera Cluster is built on top of a proprietary group communication system layer, which implements a virtual synchrony QoS (Quality of Service). Virtual synchrony unifies the data delivery and cluster membership services, providing clear formalism for message delivery semantics.

While virtual synchrony guarantees consistency, it does not guarantee temporal synchrony, which is necessary for smooth multi-master operations. To address this, Galera Cluster implements its own runtime-configurable temporal flow control. Flow control keeps nodes synchronized to a fraction of a second.

Group Communication Framework also provides a total ordering of messages from multiple sources. It uses this to generate *Global Transaction ID*'s in a multi-master cluster.

At the transport level, Galera Cluster is a symmetric undirected graph. All database nodes connect to each other over a TCP connection. By default, TCP is used for both message replication and the cluster membership services. However, you can also use UDP (User Datagram Protocol) multicast for replication in a LAN (Local Area Network).

CHAPTER

ELEVEN

ISOLATION LEVELS

In a database system, concurrent transactions are processed in "isolation" from each other. The level of isolation determines how transactions can affect each other.

Intra-Node vs. Inter-Node Isolation in Galera Cluster

Before going into details about possible isolation levels which can be set for a client session in Galera Cluster it is important to make a distinction between single node and global cluster transaction isolation. Individual cluster nodes can provide any isolation level *to the extent* it is supported by MySQL/InnoDB. However isolation level *between* the nodes in the cluster is affected by replication protocol, so transactions issued on different nodes may not be isolated *identically* to transactions issued on the same node.

Overall isolation levels that are supported cluster-wide are

- *READ-UNCOMMITTED* (page 56)
- *READ-COMMITTED* (page 56)
- *REPEATABLE-READ* (page 56)

For transactions issued on different nodes, isolation is also strengthened by the "first committer wins" rule, which eliminates the "lost update anomaly" inherent to these levels, whereas for transactions issued on the same node this rule does not hold (as per original MySQL/InnoDB behavior). This makes for different outcomes depending on transaction origin (transaction issued on the same node may succeed, whereas the same transaction issued on another node would fail), but in either case it is no weaker than that isolation level on a standalone MySQL/InnoDB.

SERIALIZABLE (page 56) isolation level is honored only between transactions issued on the same node and thus should be avoided.

Data consistency between the nodes is always guaranteed regardless of the isolation level chosen by the client. However the client logic may break if it relies on an isolation level which is not not supported in the given configuration.

Understanding Isolation Levels

Note: Warning: When using Galera Cluster in master-slave mode, all four levels are available to you, to the extend that MySQL supports it. In multi-master mode, however, you can only use the `REPEATABLE-READ` level.

READ-UNCOMMITTED

Here transactions can see changes to data made by other transactions that are not yet committed.

In other words, transactions can read data that eventually may not exist, given that other transactions can always rollback the changes without commit. This is known as a dirty read. Effectively, `READ-UNCOMMITTED` has no real isolation at all.

READ-COMMITTED

Here dirty reads are not possible. Uncommitted changes remain invisible to other transactions until the transaction commits.

However, at this isolation level `SELECT` queries use their own snapshots of committed data, that is data committed before the `SELECT` query executed. As a result, `SELECT` queries, when run multiple times within the same transaction, can return different result sets. This is called a non-repeatable read.

REPEATABLE-READ

Here non-repeatable reads are not possible. Snapshots taken for the `SELECT` query are taken the first time the `SELECT` query runs during the transaction.

The snapshot remains in use throughout the entire transaction for the `SELECT` query. It always returns the same result set. This level does not take into account changes to data made by other transactions, regardless of whether or not they have been committed. IN this way, reads remain repeatable.

SERIALIZABLE

Here all records accessed within a transaction are locked. The resource locks in a way that also prevents you from appending records to the table the transaction operates upon.

`SERIALIZABLE` prevents a phenomenon known as a phantom read. Phantom reads occur when, within a transaction, two identical queries execute, and the rows the second query returns differ from the first.

CHAPTER

TWELVE

STATE TRANSFERS

The process of replicating data from the cluster to the individual node, bringing the node into sync with the cluster, is known as provisioning. There are two methods available in Galera Cluster to provision nodes:

- *State Snapshot Transfers (SST)* (page 57) Where a snapshot of the entire node state transfers.
- *Incremental State Transfers (IST)* (page 58) Where only the missing transactions transfer.

State Snapshot Transfer (SST)

In a *State Snapshot Transfer* (SST), the cluster provisions nodes by transferring a full data copy from one node to another. When a new node joins the cluster, the new node initiates a State Snapshot Transfer to synchronize its data with a node that is already part of the cluster.

You can choose from two conceptually different approaches in Galera Cluster to transfer a state from one database to another:

- **Logical** This method uses `mysqldump`. It requires that you fully initialize the receiving server and ready it to accept connections *before* the transfer.

  This is a blocking method. The donor node becomes `READ-ONLY` for the duration of the transfer. The State Snapshot Transfer applies the `FLUSH TABLES WITH READ LOCK` command on the donor node.

  `mysqldump` is the slowest method for State Snapshot Transfers. This can be an issue in a loaded cluster.

- **Physical** This method uses `rsync`, `rsync_wan`, `xtrabackup` and other methods and copies the data files directly from server to server. It requires that you initialize the receiving server *after* the transfer.

  This method is faster than `mysqldump`, but they have certain limitations. You can only use them on server startup. The receiving server requires very similar configurations to the donor, (for example, both servers must use the same innodb_file_per_table value).

  Some of these methods, such as `xtrabackup` can be made non-blocking on the donor. They are supported through a scriptable SST interface.

Note: See Also: For more information on the particular methods available for State Snapshot Transfers, see the *State Snapshot Transfers* (page 81).

You can set which State Snapshot Transfer method a node uses from the confirmation file. For example:

```
wsrep_sst_method=rsync_wan
```

Incremental State Transfer (IST)

In an *Incremental State Transfer* (IST), the cluster provisions a node by identifying the missing transactions on the joiner and sends them only, instead of the entire state.

This provisioning method is only available under certain conditions:

- Where the joiner node *state UUID* is the same as that of the group.
- Where all missing write-sets are available in the donor's write-set cache.

When these conditions are met, the donor node transfers the missing transactions alone, replaying them in order until the joiner catches up with the cluster.

For example, say that you have a node in your cluster that falls behind the cluster. This node carries a node state that reads:

```
5a76ef62-30ec-11e1-0800-dba504cf2aab:197222
```

Meanwhile, the current node state on the cluster reads:

```
5a76ef62-30ec-11e1-0800-dba504cf2aab:201913
```

The donor node on the cluster receives the state transfer request from the joiner node. It checks its write-set cache for the *sequence number* `197223`. If that seqno is not available in the *write-set cache*, a State Snapshot Transfer initiates. If that seqno is available in the write-set cache, the donor node sends the commits from `197223` through to `201913` to the joiner, instead of the full state.

The advantage of Incremental State Transfers is that they can dramatically speed up the reemerging of a node to the cluster. Additionally, the process is non-blocking on the donor.

Note: The most important parameter for Incremental State Transfers is `gcache.size` on the donor node. This controls how much space you allocate in system memory for caching write-sets. The more space available the more write-sets you can store. The more write-sets you can store the wider the seqno gaps you can close through Incremental State Transfers.

On the other hand, if the write-set cache is much larger than the size of your database state, Incremental State Transfers begun less efficient than sending a state snapshot.

Write-set Cache (GCache)

Galera Cluster stores write-sets in a special cache called the *Write-set Cache*, or GCache. GCache cache is a memory allocator for write-sets. Its primary purpose is to minimize the *write-set* footprint on the RAM (Random Access Memory). Galera Cluster improves upon this through the offload write-set storage to disk.

GCache employs three types of storage:

- **Permanent In-Memory Store** Here write-sets allocate using the default memory allocator for the operating system. This is useful in systems that have spare RAM. The store has a hard size limit.

  By default it is disabled.

- **Permanent Ring-Buffer File** Here write-sets pre-allocate to disk during cache initialization. This is intended as the main write-set store.

  By default, its size is 128Mb.

- **On-Demand Page Store** Here write-sets allocate to memory-mapped page files during runtime as necessary.

  By default, its size is 128Mb, but can be larger if it needs to store a larger write-set. The size of the page store is limited by the free disk space. By default, Galera Cluster deletes page files when not in use, but you can set a limit on the total size of the page files to keep.

  When all other stores are disabled, at least one page file remains present on disk.

Note: **See Also**: For more information on parameters that control write-set caching, see the `gcache.*` parameters on *Galera Parameters* (page 265).

Galera Cluster uses an allocation algorithm that attempts to store write-sets in the above order. That is, first it attempts to use permanent in-memory store. If there is not enough space for the write-set, it attempts to store to the permanent ring-buffer file. The page store always succeeds, unless the write-set is larger than the available disk space.

By default, the write-set cache allocates files in the working directory of the process. You can specify a dedicated location for write-set caching, using the *gcache.dir* (page 274) parameter.

Note: Given that all cache files are memory-mapped, the write-set caching process may appear to use more memory than it actually does.

CHAPTER

THIRTEEN

FLOW CONTROL

Galera Cluster manages the replication process using a feedback mechanism, called Flow Control. Flow Control allows a node to pause and resume replication according to its needs. This prevents any node from lagging too far behind the others in applying transactions.

How Flow Control Works

Galera Cluster achieves synchronous replication by ensuring that transactions copy to all nodes an execute according to a cluster-wide ordering. That said, the transaction applies and commits occur asynchronously as they replicate through the cluster.

Nodes receive write-sets and organize them into the global ordering. Transactions that the node receives from the cluster but which it has not applied and committed, are kept in the received queue.

When the received queue reaches a certain size the node triggers Flow Control. The node pauses replication, then works through the received queue. When it reduces the received queue to a more manageable size, the node resumes replication.

Understanding Node States

Galera Cluster implements several forms of Flow Control, depending on the node state. This ensures temporal synchrony and consistency—as opposed to logical, which virtual synchrony provides.

There are four primary kinds of Flow Control:

- *No Flow Control* (page 61)
- *Write-set Caching* (page 62)
- *Catching Up* (page 62)
- *Cluster Sync* (page 62)

No Flow Control

This Flow Control takes effect when nodes are in the `OPEN` or `PRIMARY` states.

When nodes hold these states, they are not considered part of the cluster. These nodes are not allowed to replicate, apply or cache any write-sets.

Write-set Caching

This Flow Control takes effect when nodes are in the `JOINER` and `DONOR` states.

Nodes cannot apply any write-sets while in this state and must cache them for later. There is no reasonable way to keep the node synchronized with the cluster, except for stopping all replication.

It is possible to limit the replication rate, ensuring that the write-set cache does not exceed the configured size. You can control the write-set cache with the following parameters:

- *gcs.recv_q_hard_limit* (page 277) Maximum write-set cache size (in bytes).
- *gcs.max_throttle* (page 277) Smallest fraction to the normal replication rate the node can tolerate in the cluster.
- *gcs.recv_q_soft_limit* (page 277) Estimate of the average replication rate for the node.

Catching Up

This Flow Control takes effect when nodes are in the `JOINED` state.

Nodes in this state can apply write-sets. Flow Control here ensures that the node can eventually catch up with the cluster. It specifically ensures that its write-set cache never grows. Because of this, the cluster wide replication rate remains limited by the rate at which a node in this state can apply write-sets. Since applying write-sets is usually several times faster than processing a transaction, nodes in this state hardly ever effect cluster performance.

The one occasion when nodes in the `JOINED` state do effect cluster performance is at the very beginning, when the buffer pool on the node in question is empty.

Note: You can significantly speed this up with parallel applying.

Cluster Sync

This Flow Control takes effect when nodes are in the `SYNCED` state.

When nodes enter this state Flow Control attempts to keep the slave queue to a minimum. You can configure how the node handles this using the following parameters:

- *gcs.fc_limit* (page 276) Used to determine the point where Flow Control engages.
- *gcs.fc_factor* (page 276) Used to determine the point where Flow Control disengages.

Changes in the Node State

The node state machine handles different state changes on different layers of Galera Cluster. These are the node state changes that occur at the top most layer:

1. The node starts and establishes a connection to the *Primary Component*.
2. When the node succeeds with a state transfer request, it begins to cache write-sets.
3. The node receives a *State Snapshot Transfer*. It now has all cluster data and begins to apply the cached write-sets.

   Here the node enables Flow Control to ensure an eventual decrease in the slave queue.

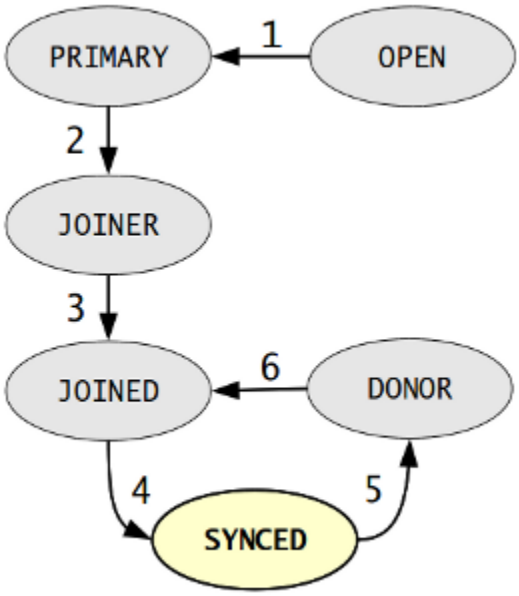


Fig. 13.1: *Galera Cluster Node State Changes*

4. The node finishes catching up with the cluster. Its slave queue is now empty and it enables Flow Control to keep it empty.

   The node sets the MySQL status variable *wsrep_ready* (page 302) to the value `1`. The node is now allowed to process transactions.

5. The node receives a state transfer request. Flow Control relaxes to `DONOR`. The node caches all write-sets it cannot apply.

6. The node completes the state transfer to joiner node.

For the sake of legibility, certain transitions were omitted from the above description. Bear in mind the following points:

- **Connectivity** Cluster configuration change events can send a node in any state to `PRIMARY` or `OPEN`. For instance, a node that is `SYNCED` reverts to `OPEN` when it loses its connection to the Primary Component due to network partition.
- **Missing Transitions** In the event that the joining node does not require a state transfer, the node state changes from the `PRIMARY` state directly to the `JOINED` state.

Note: **See Also**: For more information on Flow Control see Galera Flow Control in Percona XtraDB Cluster.

CHAPTER

FOURTEEN

NODE FAILURE AND RECOVERY

Individual nodes fail to operate when they lose touch with the cluster. This can occur due to various reasons. For instance, in the event of hardware failure or software crash, the loss of network connectivity or the failure of a state transfer. Anything that prevents the node from communicating with the cluster is generalized behind the concept of node failure. Understanding how nodes fail will help in planning for their recovery.

Detecting Single Node Failures

When a node fails the only sign is the loss of connection to the node processes as seen by other nodes. Thus nodes are considered failed when they lose membership with the cluster's *Primary Component*. That is, from the perspective of the cluster when the nodes that form the Primary Component can no longer see the node, that node is failed. From the perspective of the failed node itself, assuming that it has not crashed, it has lost its connection with the Primary Component.

Although there are third-party tools for monitoring nodes—such as ping, Heartbeat, and Pacemaker—they can be grossly off in their estimates on node failures. These utilities do not participate in the Galera Cluster group communications and remain unaware of the Primary Component.

If you want to monitor the Galera Cluster node status poll the *wsrep_local_state* (page 300) status variable or through the *Notification Command* (page 161).

Note: **See Also**: For more information on monitoring the state of cluster nodes, see the chapter on *Monitoring the Cluster* (page 153).

The cluster determines node connectivity from the last time it received a network packet from the node. You can configure how often the cluster checks this using the *evs.inactive_check_period* (page 270) parameter. During the check, if the cluster finds that the time since the last time it received a network packet from the node is greater than the value of the *evs.keepalive_period* (page 272) parameter, it begins to emit heartbeat beacons. If the cluster continues to receive no network packets from the node for the period of the *evs.suspect_timeout* (page 273) parameter, the node is declared suspect. Once all members of the Primary Component see the node as suspect, it is declared inactive—that is, failed.

If no messages were received from the node for a period greater than the *evs.inactive_timeout* (page 271) period, the node is declared failed regardless of the consensus. The failed node remains non-operational until all members agree on its membership. If the members cannot reach consensus on the liveness of a node, the network is too unstable for cluster operations.

The relationship between these option values is:

evs.keepalive_period (page 272)	<=	*evs.inactive_check_period* (page 270)
evs.inactive_check_period (page 270)	<=	*evs.suspect_timeout* (page 273)
evs.suspect_timeout (page 273)	<=	*evs.inactive_timeout* (page 271)
evs.inactive_timeout (page 271)	<=	*evs.consensus_timeout* (page 269)

Note: Unresponsive nodes that fail to send messages or heartbeat beacons on time—for instance, in the event of heavy swapping—may also be pronounced failed. This prevents them from locking up the operations of the rest of the cluster. If you find this behavior undesirable, increase the timeout parameters.

Cluster Availability vs. Partition Tolerance

Within the CAP theorem, Galera Cluster emphasizes data safety and consistency. This leads to a trade-off between cluster availability and partition tolerance. That is, when using unstable networks, such as WAN (Wide Area Network), low *evs.suspect_timeout* (page 273) and *evs.inactive_timeout* (page 271) values may result in false node failure detections, while higher values on these parameters may result in longer availability outages in the event of actual node failures.

Essentially what this means is that the *evs.suspect_timeout* (page 273) parameter defines the minimum time needed to detect a failed node. During this period, the cluster is unavailable due to the consistency constraint.

Recovering from Single Node Failures

If one node in the cluster fails, the other nodes continue to operate as usual. When the failed node comes back online, it automatically synchronizes with the other nodes before it is allowed back into the cluster.

No data is lost in single node failures.

Note: **See Also**: For more information on manually recovering nodes, see *Node Provisioning and Recovery* (page 79).

State Transfer Failure

Single node failures can also occur when a *state snapshot transfer* fails. This failure renders the receiving node unusable, as the receiving node aborts when it detects a state transfer failure.

When the node fails while using `mysqldump`, restarting may require you to manually restore the administrative tables. For the `rsync` method in state transfers this is not an issue, given that it does not require the database server to be in an operational state to work.

CHAPTER

FIFTEEN

WEIGHTED QUORUM

In addition to single node failures, the cluster may split into several components due to network failure. A component is a set of nodes that are connected to each other, but not to the nodes that form other components. In these situations, only one component can continue to modify the database state to avoid history divergence. This component is called the *Primary Component*.

Under normal operations, your Primary Component is the cluster. When cluster partitioning occurs, Galera Cluster invokes a special quorum algorithm to select one component as the Primary Component. This guarantees that there is never more than one Primary Component in the cluster.

Note: **See Also**: In addition to the individual node, quorum calculations also take into account a separate process called `garbd`. For more information on its configuration and use, see *Galera Arbitrator* (page 117).

Weighted Quorum

The current number of nodes in the cluster defines the current cluster size. There is no configuration setting that allows you to define the list of all possible cluster nodes. Every time a node joins the cluster, the total cluster size increases. When a node leaves the cluster, gracefully, the cluster size decreases. Cluster size determines the number of votes required to achieve quorum.

Galera Cluster takes a quorum vote whenever a node does not respond and is suspected of no longer being a part of the cluster. You can fine tune this no response timeout using the *evs.suspect_timeout* (page 273) parameter. The default setting is 5 seconds.

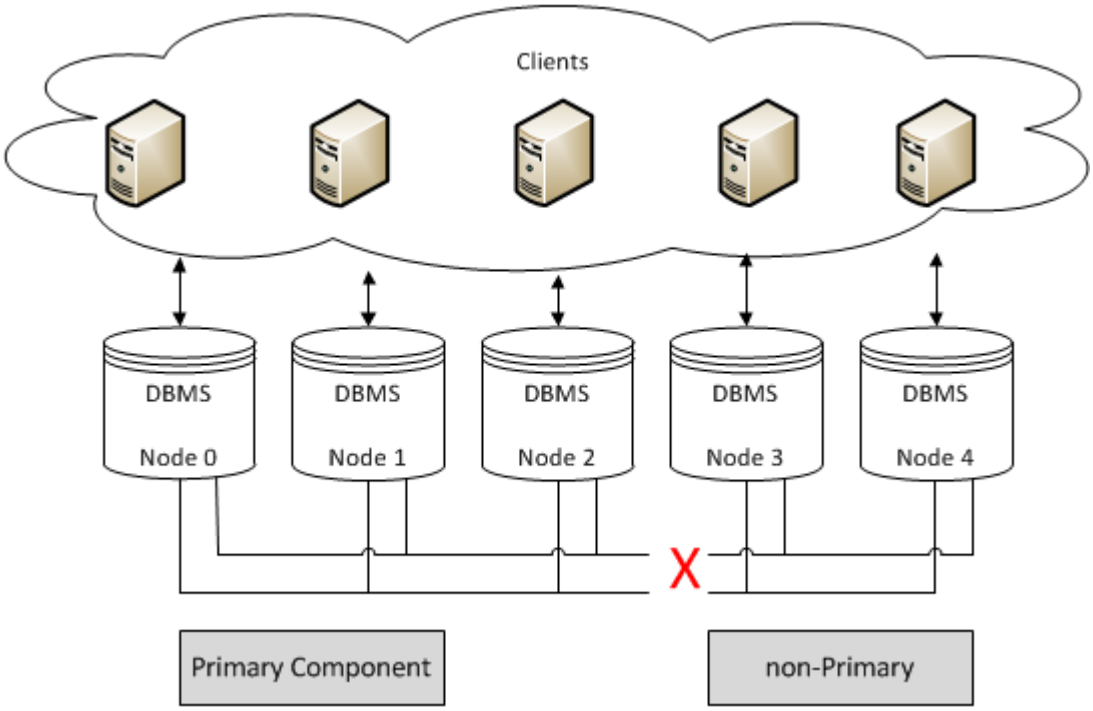
When the cluster takes a quorum vote, if the majority of the total nodes connected from before the disconnect remain, that partition stays up. When network partitions occur, there are nodes active on both sides of the disconnect. The component that has quorum alone continues to operate as the *Primary Component*, while those without quorum enter the non-primary state and begin attempt to connect with the Primary Component.

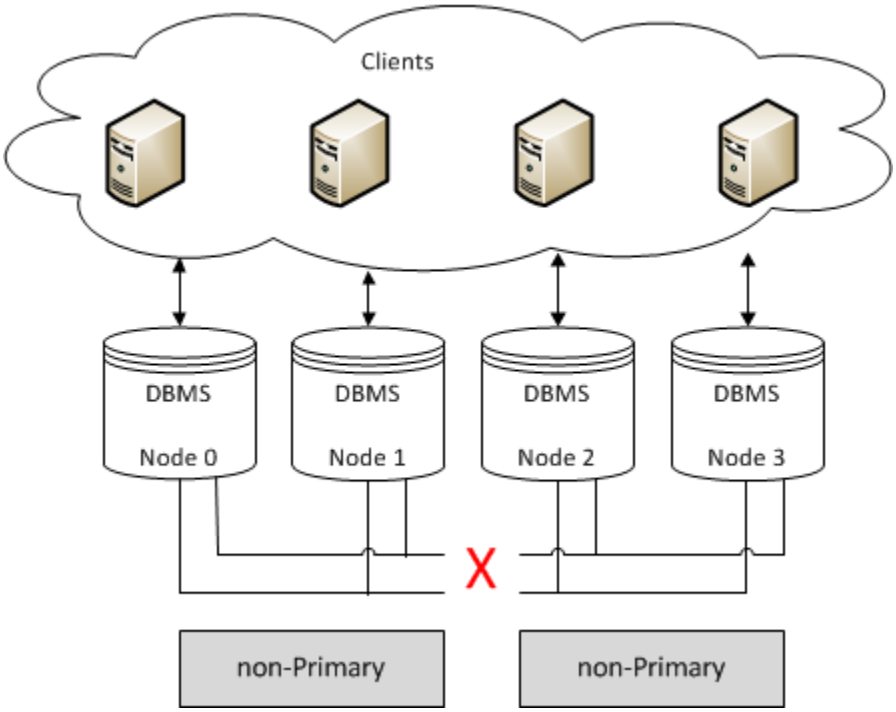
Quorum requires a majority, meaning that you cannot have automatic failover in a two node cluster. This is because the failure of one causes the remaining node automatically go into a non-primary state.

Clusters that have an even number of nodes risk split-brain conditions. If should you lose network connectivity somewhere between the partitions in a way that causes the number of nodes to split exactly in half, neither partition can retain quorum and both enter a non-primary state.

In order to enable automatic failovers, you need to use at least three nodes. Bear in mind that this scales out to other levels of infrastructure, for the same reasons.

- Single switch clusters should use a minimum of 3 nodes.
- Clusters spanning switches should use a minimum of 3 switches.


Clients
DBMS
Node 0
DBMS
Node 1
DBMS
Node 2
DBMS
Node 3
DBMS
Node 4
X
Primary Component
non-Primary


Clients
DBMS
Node 0
DBMS
Node 1
DBMS
Node 2
DBMS
Node 3
X
non-Primary
non-Primary

- Clusters spanning networks should use a minimum of 3 networks.
- Clusters spanning data centers should use a minimum of 3 data centers.

Split-brain Condition

Cluster failures that result in database nodes operating autonomous of each other are called split-brain conditions. When this occurs, data can become irreparably corrupted, such as would occur when two database nodes independently update the same row on the same table. As is the case with any quorum-based system, Galera Cluster is subject to split-brain conditions when the quorum algorithm fails to select a *Primary Component*.

For example, this can occur if you have a cluster without a backup switch in the event that the main switch fails. Or, when a single node fails in a two node cluster.

By design, Galera Cluster avoids split-brain condition. In the event that a failure results in splitting the cluster into two partitions of equal size, (unless you explicitly configure it otherwise), neither partition becomes a Primary Component.

To minimize the risk of this happening in clusters that do have an even number of nodes, partition the cluster in a way that one component always forms the Primary cluster section.

```
4 node cluster -> 3 (Primary) + 1 (Non-primary)
6 node cluster -> 4 (Primary) + 2 (Non-primary)
6 node cluster -> 5 (Primary) + 1 (Non-primary)
```

In these partitioning examples, it is very difficult for any outage or failure to cause the nodes to split exactly in half.

Note: **See Also**: For more information on configuring and managing the quorum, see *Resetting the Quorum* (page 103).

Quorum Calculation

Galera Cluster supports a weighted quorum, where each node can be assigned a weight in the 0 to 255 range, with which it will participate in quorum calculations.

The quorum calculation formula is

$$\frac{\sum_{p_i \times w_i} - \sum_{l_i \times w_i}}{2} < \sum_{m_i \times w_i}$$

Where:

- p_i Members of the last seen primary component;
- l_i Members that are known to have left gracefully;
- m_i Current component members; and,
- w_i Member weights.

What this means is that the quorum is preserved if (and only if) the sum weight of the nodes in a new component strictly exceeds half that of the preceding *Primary Component*, minus the nodes which left gracefully.

You can customize node weight using the *pc.weight* (page 282) parameter. By default, node weight is `1`, which translates to the traditional node count behavior.

Note: You can change node weight in runtime by setting the *pc.weight* (page 282) parameter.

```
SET GLOBAL wsrep_provider_options="pc.weight=3";
```

Galera Cluster applies the new weight on the delivery of a message that carries a weight. At the moment, there is no mechanism to notify the application of a new weight, but will eventually happen when the message is delivered.

Note: Warning: If a group partitions at the moment when the weight change message is delivered, all partitioned components that deliver weight change messages in the transitional view will become non-primary components. Partitions that deliver messages in the regular view will go through quorum computation with the applied weight when the following transitional view is delivered.

In other words, there is a corner case where the entire cluster can become non-primary component, if the weight changing message is sent at the moment when partitioning takes place. Recovering from such a situation should be done either by waiting for a re-merge or by inspecting which partition is most advanced and by bootstrapping it as a new Primary Component.

Weighted Quorum Examples

Now that you understand how quorum weights work, here are some examples of deployment patterns and how to use them.

Weighted Quorum for Three Nodes

When configuring quorum weights for three nodes, use the following pattern:

```
node1: pc.weight = 2
node2: pc.weight = 1
node3: pc.weight = 0
```

Under this pattern, killing `node2` and `node3` simultaneously preserves the *Primary Component* on `node1`. Killing `node1` causes `node2` and `node3` to become non-primary components.

Weighted Quorum for a Simple Master-Slave Scenario

When configuring quorum weights for a simple master-slave scenario, use the following pattern:

```
node1: pc.weight = 1
node2: pc.weight = 0
```

Under this pattern, if the master `node` dies, `node2` becomes a non-primary component. However, in the event that `node2` dies, `node1` continues as the *Primary Component*. If the network connection between the nodes fails, `node1` continues as the Primary Component while `node2` becomes a non-primary component.

Weighted Quorum for a Master and Multiple Slaves Scenario

When configuring quorum weights for a master-slave scenario that features multiple slave nodes, use the following pattern:

```
node1: pc.weight = 1
node2: pc.weight = 0
node3: pc.weight = 0
...
noden: pc.weight = 0
```

Under this pattern, if `node1` dies, all remaining nodes end up as non-primary components. If any other node dies, the *Primary Component* is preserved. In the case of network partitioning, `node1` always remains as the Primary Component.

Weighted Quorum for a Primary and Secondary Site Scenario

When configuring quorum weights for primary and secondary sites, use the following pattern:

```
Primary Site:
  node1: pc.weight = 2
  node2: pc.weight = 2

Secondary Site:
  node3: pc.weight = 1
  node4: pc.weight = 1
```

Under this pattern, some nodes are located at the primary site while others are at the secondary site. In the event that the secondary site goes down or if network connectivity is lost between the sites, the nodes at the primary site remain the *Primary Component*. Additionally, either `node1` or `node2` can crash without the rest of the nodes becoming non-primary components.

CHAPTER

SIXTEEN

STREAMING REPLICATION

Under normal operation, the node performs all replication and certification events when a transaction commits. When working with small transactions this is fine. However, it poses an issue with long-running writes and changes to large data-sets.

In *Streaming Replication*, the node breaks the transaction into fragments, then certifies and replicates them on the slaves while the transaction is still in progress. Once certified, the fragment can no longer be aborted by conflicting transactions.

Additionally, Streaming Replication allows the node to process transaction write-sets greater than 2Gb.

Note: Streaming Replication is a new feature introduced in version 4.0 of Galera Cluster. Older versions do not support these operations.

When to Use Streaming Replication

In most cases, the normal method Galera Cluster uses in replication is sufficient in transferring data from a node to a cluster. *Streaming Replication* provides you with an alternative for situations in which this is not the case. Keep in mind that there are some limitations to its use. It's recommended that you only enable it at a session-level, and then only on specific transactions that require the feature.

Note: For more information on the limitations to Streaming Replication, see *Limitations* (page 74).

Long-Running Write Transactions

When using normal replication, you may occasionally encounter issues with long-running write transactions.

The longer it takes for a node to commit a transaction, the greater the likelihood that the cluster will apply a smaller, conflicting transaction before the longer one can replicate to the cluster. When this happens, the cluster aborts the long-running transaction.

Using *Streaming Replication* on long-running transactions mitigates this situation. Once the node replicates and certifies a fragment, it is no longer possible for other transactions to abort it.

Large Data Write Transactions

When using normal replication, the node locally processes the transaction and doesn't replicate the data until you commit. This can create problems when updating a large volume of data, especially on nodes with slower network

connections.

Additionally, while slave nodes apply a large transaction, they cannot commit other transactions they receive, which may result in Flow Control throttling of the entire cluster.

With *Streaming Replication*, the node begins to replicate the data with each transaction fragment, rather than waiting for the commit. This allows you to spread the replication over the lifetime of the transaction.

In the case of the slave nodes, after the slave applies a fragment, it's free to apply and commit other, concurrent transactions without blocking. This allows the slave node to process incrementally the entire large transaction with a minimal impact on the cluster.

Hot Records

In situations in which an application frequently updates one and the same records from the same table (e.g., when implementing a locking scheme, a counter, or a job queue), you can use *Streaming Replication* to force critical updates to replicate to the entire cluster.

Running a transaction in this way effectively locks the hot record on all nodes, preventing other transactions from modifying the row. It also increases the chances that the transaction will commit successfully and that the client in turn will receive the desired outcome.

Note: For more information and an example of how to implement Streaming Replication in situations such as this, see *Using Streaming Replication with Hot Records* (page 116).

Limitations

In deciding whether you want to use *Streaming Replication* with your application, consider the following limitations.

Performance During a Transaction

When you enable *Streaming Replication*, each node in the cluster begin recording its write-sets to the `SR` table in the `wsrep_schema` database. The nodes do this to ensure the persistence of Streaming Replication updates in the event that they crash. However, this operation increases the load on the node, which may adversely affect its performance.

As such, it's recommended that you only enable Streaming Replication at a session-level and then only for transactions that would not run correctly without it.

Performance During Rollbacks

Occasionally, you may encounter situations in which the cluster needs to roll back a transaction while *Streaming Replication* is in use. In these situations, the rollback operation consumes system resources on all nodes.

When long-running write transactions frequently need to be rolled back, this can become a performance problem. Therefore, it's a good application design policy to use shorter transactions whenever possible. In the event that your application performs batch processing or scheduled housekeeping tasks, consider splitting these into smaller transactions in addition to using Streaming Replication.

Part III

Administration

With the basics of how the cluster works and how to install and initialize it covered, this part begins a five part series on the administration and management of Galera Cluster.

The chapters in this part relate to the administration of nodes and the cluster. *Deployment* (page 125), covers how to use Galera Cluster in relation to your wider infrastructure, how to configure load balancers to work with the cluster and edge case deployments, such as running nodes in containers. The chapters in *Cluster Monitoring* (page 151) show how to keep tabs on the status of the cluster and automate reporting. *Security* (page 167) covers configuring Galera Cluster to work with firewalls, SELinux and SSL encryption. *Migration* (page 189) how to transition from a standalone instance of MySQL, MariaDB or Percona XtraDB to Galera Cluster.

Node Administration

Managing and administering nodes in Galera Cluster is similar to the administration and management of the standard standalone MySQL, MariaDB and Percona XtraDB database servers, with some additional features used to manage its interaction with the cluster. These chapters cover the administration of individual nodes, how they handle write-set replication and schema updates, and the procedure for upgrading Galera Cluster software.

- *Node Provisioning* (page 79)

  The complete process of replicating data into a node so that it can operate as part of the Primary Component is called 'provisioning' the node. It ensures that the nodes update the local data, keeping it consistent with the state of the cluster. This chapter provides an overview to how nodes join the cluster and maintain their state through state transfers.

- *State Snapshot Transfers* (page 81)

  When a node falls too far behind the cluster, they request State Snapshot Transfers from another node in order to bring its local database up to date with the cluster. This chapter provides a guide to each state transfer method Galera Cluster supports.

- *Scriptable State Snapshot Transfers* (page 87)

  When nodes send and receive State Snapshot Transfers, they manage the process through external scripts that call the standard state transfer methods. In event that you require additional functionality than what is available by default, you can use scripts to implement your own custom state snapshot transfer methods.

- *System Databases* (page 91)

  When you install Galera Cluster, it creates a set of system databases, which it uses to store configuration information. Similar to how the underlying database server uses the `performance_schema` and `information_schema`, Galera Cluster uses `wsrep_schema` to record information relevant to replication. This chapter provides a guide to what you'll find in this database and how you might query it for useful information about the health of the node and the cluster.

- *Schema Upgrades* (page 93)

  Statements that update the database schema, (that is, DDL statements), are non-transactional and as such won't replicate to the cluster through write-sets. This chapter covers different methods for online schema upgrades and how to implement them in your deployment.

- *Upgrading Galera Cluster* (page 95)

  In order to upgrade Galera Cluster to a new version or increment of the software, there are a few additional steps you need to take in order to maintain the cluster during the upgrade. This chapter provides guides to different methods in handling this process.

Cluster Administration

In addition to node administration, Galera Cluster also provides interfaces for managing and administering the cluster. These chapters cover Primary Component recovery, managing Flow Control and Auto Eviction, as well as Galera Arbitrator and how to handle backups.

- *Recovering the Primary Component* (page 99)

  When nodes establish connections with each other, they form components. The operational component in the cluster is called the Primary Component. This chapter covers a new feature in version 3.6 of Galera Cluster, which sets the nodes to save the Primary Component state to disk. In the event of an outage, once all the nodes that previously formed the Primary Component reestablish network connectivity, they automatically restore themselves as the new Primary Component.

- *Resetting the Quorum* (page 103)

  The Primary Component maintains quorum when most of the nodes in the cluster are connected to it. This chapter provides a guide to resetting the quroum in the event that the cluster becomes non-operational due to a major network outage, the failure of more than half the nodes, or a split-brain situation.

- *Managing Flow Control* (page 107)

  When nodes fall too far behind, Galera Cluster uses a feedback mechanism called Flow Control, pausing replication to give the node to process transactions and catch up with the cluster. This chapter covers the monitoring and configuration of Flow Control, in order to improve node performance.

- *Auto-Eviction* (page 111)

  When Galera Cluster notices erratic behavior from a node, such as in the case of unusually delayed response times, it can initiate a process to remove the node permanently from the cluster. This chapter covers the configuration and management of how the cluster handles these Auto Evictions.

- *Using Streaming Replication* (page 115)

  When the node uses Streaming Replication, instead of waiting for the commit to replicate and apply transactions to the cluster, it breaks the transaction down into replication units, transferring and applying these on the slave nodes while the transaction is still open. This chapter provides a guide to how to enable, configure and utilize Streaming Replication.

- *Galera Arbitrator* (page 117)

  Galera Arbitrator is a separate application from Galera Cluster. It functions as an additional node in quorum calculations, receives the same data as other node, but does not participate in replications. You can use it to provide an odd node to help avoid split-brain situations, or use it in generating consistent application state snapshots, in generating backups.

- *Backing Up Cluster Data* (page 121)

  Standard backup methods available to MySQL database servers fail to preserve Global Transaction ID's used by Galera Cluster. You can recover data from these backups, but they're insufficient in restoring nodes to a well-defined state. This chapter shows how to use state transfers to properly perform backups in Galera Cluster.

CHAPTER

SEVENTEEN

NODE PROVISIONING

When the state of a new or failed node differs from that of the cluster's *Primary Component*, the new or failed node must be synchronized with the cluster. Because of this, the provisioning of new nodes and the recover of failed nodes are essentially the same process as that of joining a node to the cluster Primary Component.

Galera reads the initial node state ID from the **grastate.txt** file, found in the directory assigned by the `wsrep_data_dir` parameter. Each time the node gracefully shuts down, Galera saves to this file.

In the event that the node crashes while in *Total Order Isolation* mode, its database state is unknown and its initial node state remains undefined:

```
00000000-0000-0000-0000-000000000000:-1
```

Note: In normal transaction processing, only the seqno part of the GTID remains undefined, (that is, with a value of `-1`. The UUID, (that is, the remainder of the node state), remains valid. In such cases, you can recover the node through an *Incremental State Transfer*.

How Nodes Join the Cluster

When a node joins the cluster, it compares its own *state UUID* to that of the *Primary Component*. If the state UUID does not match, the joining node requests a state transfer from the cluster.

There are two options available to determining the state transfer donor:

- **Automatic** When the node attempts to join the cluster, the group communication layer determines the state donor it should use from those members available in the Primary Component.
- **Manual** When the node attempts to join the cluster, it uses the *wsrep_sst_donor* (page 255) parameter to determine which state donor it should use. If it finds that the state donor it is looking for is not part of the Primary Component, the state transfer fails and the joining node aborts. For *wsrep_sst_donor* (page 255), use the same name as you use on the donor node for the *wsrep_node_name* (page 247) parameter.

Note: A state transfer is a heavy operation. This is true not only for the joining node, but also for the donor. In fact, a state donor may not be able to serve client requests.

Thus, whenever possible: manually select the state donor, based on network proximity and configure the load balancer to transfer client connections to other nodes in the cluster for the duration of the state transfer.

When a state transfer is in process, the joining node caches write-sets that it receives from other nodes in a slave queue. Once the state transfer is complete, it applies the write-sets from the slave queue to catch up with the current Primary

Component state. Since the state snapshot carries a state UUID, it is easy to determine which write-sets the snapshot contains and which it should discard.

During the catch-up phase, flow control ensures that the slave queue shortens, (that is, it limits the cluster replication rates to the write-set application rate on the node that is catching up).

While there is no guarantee on how soon a node will catch up, when it does the node status updates to `SYNCED` and it begins to accept client connections.

State Transfers

There are two types of state transfers available to bring the node up to date with the cluster:

- *State Snapshot Transfer* (SST) Where donor transfers to the joining node a snapshot of the entire node state as it stands.
- *Incremental State Transfer* (IST) Where the donor only transfers the results of transactions missing from the joining node.

When using automatic donor selection, starting in Galera Cluster version 3.6, the cluster decides which state transfer method to use based on availability.

- If there are no nodes available that can safely perform an incremental state transfer, the cluster defaults to a state snapshot transfer.
- If there are nodes available that can safely perform an incremental state transfer, the cluster prefers a local node over remote nodes to serve as the donor.
- If there are no local nodes available that can safely perform an incremental state transfer, the cluster chooses a remote node to serve as the donor.
- Where there are several local or remote nodes available that can safely perform an incremental state transfer, the cluster chooses the node with the highest seqno to serve as the donor.

CHAPTER

EIGHTEEN

STATE SNAPSHOT TRANSFERS

When a node requires a state transfer from the cluster, by default it attempts the *Incremental State Transfer* (IST) method. In the event that there are no nodes available for this or if it finds a manual donor defined through the *wsrep_sst_donor* (page 255) parameter, uses a *State Snapshot Transfer* (SST) method.

Galera Cluster supports several back-end methods for use in state snapshot transfers. There are two types of methods available: Logical State Snapshots, which interface through the database server and client; and Physical State Snapshots, which copy the data files directly from node to node.

Method	Speed	Blocks Donor	Available on Live Node	Type	DB Root Access
mysqldump (page 82)	Slow	Blocks	Available	*Logical* (page 81)	Donor and Joiner
rsync (page 84)	Fastest	Blocks	Unavailable	*Physical* (page 83)	None
xtrabackup (page 84)	Fast	Briefly	Unavailable	*Physical* (page 83)	Donor only

To set the State Snapshot Transfer method, use the *wsrep_sst_method* (page 256) parameter. For example:

```
wsrep_sst_method = rsync
```

There is no single best method for State Snapshot Transfers. You must decide which best suits your particular needs and cluster deployment. Fortunately, you need only set the method on the receiving node. So long as the donor has support, it servers the transfer in whatever method the joiner requests.

Logical State Snapshot

There is one back-end method available for a Logical State Snapshots: `mysqldump`.

The *Logical State Transfer Method* has the following advantages:

- These transfers are available on live servers. In fact, only a fully initialized server can receive a Logical State Snapshot.
- These transfers do not require the receptor node to have the same configuration as the donor node. This allows you to upgrade storage engine options.

  For example, when using this transfer method you can migrate from the Antelope to the Barracuda file format, use compression resize, or move iblog* files from one partition into another.

The Logical State Transfer Method has the following disadvantages:

- These transfers are as slow as `mysqldump`.

- These transfers require that you configure the receiving database server to accept root connections from potential donor nodes.
- The receiving server must have a non-corrupted database.

mysqldump

The main advantage of `mysqldump` is that you can transfer a state snapshot to a working server. That is, you start the server standalone and then instruct it to join a cluster from within the database client command line. You can also use it to migrate from an older database format to a newer one.

`mysqldump` requires that the receiving node have a fully functional database, which can be empty. It also requires the same root credentials as the donor and root access from the other nodes.

This transfer method is several times slower than the others on sizable databases, but it may prove faster in cases of very small databases. For instance, on a database that is smaller than the log files.

Note: Warning: This transfer method is sensitive to the version of `mysqldump` each node uses. It is not uncommon for a given cluster to have installed several versions. A State Snapshot Transfer can fail if the version one node uses is older and incompatible with the newer server.

On occasion, `mysqldump` is the only option available. For instance, if you upgrade from a cluster using MySQL 5.1 with the built-in InnoDB support to MySQL 5.5, which uses the InnoDB plugin.

The `mysqldump` script only runs on the sending node. The output from the script gets piped to the MySQL client that connects to the joiner node.

Because `mysqldump` interfaces through the database client, configuring it requires several steps beyond setting the *wsrep_sst_method* (page 256) parameter. For more information on its configuration, see:

Enabling mysqldump

The *Logical State Transfer Method* `mysqldump` works by interfacing through the database server rather than the physical data. As such, it requires some additional configuration, besides setting the *wsrep_sst_method* (page 256) parameter.

Configuring SST Privileges

In order for `mysqldump` to interface with the database server, it requires root connections for both the donor and joiner nodes. You can enable this through the *wsrep_sst_auth* (page 254) parameter.

Using a text editor, open the `wsrep.cnf` file–it should be in the `/etc/mysql/conf.d/` directory. Add a line like the following to that file:

```
# wsrep SST Authentication
wsrep_sst_auth = wsrep_sst_username:password
```

You would use your own authentication parameters in place of `wsrep_sst_user` and `password`. This line will provide authentication information that the node will need to establish connections. Use the same values for every node in the cluster.

Granting SST Privileges

When the database server starts, it will read from the `wsrep.cnf` file to get the authentication information it needs to access another database server. In order for the node to accept connections from the cluster, you must also create and configure the State Snapshot Transfer user through the database client.

In order to do this, you need to start the database server. If you haven't used this node on the cluster before, start it with replication disabled. For servers that use `init`, execute the following from the command-line:

```
# service mysql start --wsrep-on=off
```

For servers that use `systemd`, instead execute this from the command-line:

```
# systemctl start mysql --wsrep-on=OFF
```

When the database server is running, log into the database using a client and execute the `GRANT ALL` statement for the IP address of each node in the cluster. You would do this like so:

```
GRANT ALL ON *.* TO 'wsrep_sst_user'@'node1_IP_address'
     IDENTIFIED BY 'password';
GRANT ALL ON *.* TO 'wsrep_sst_user'@'node2_IP_address'
     IDENTIFIED BY 'password';
GRANT ALL ON *.* TO 'wsrep_sst_user'@'node3_IP_address'
     IDENTIFIED BY 'password';
```

You would, of course, modify the text above to use your user names, IP addresses, and passwords. These SQL statements will grant each node in the cluster access to the database server on this node. You need to run these SQL statements on each node to allow `mysqldump` in state transfers among them.

If you have not yet created the cluster, you can stop the database server while you configure the other nodes. To stop MySQL on servers that use `init`, run the execute the following from the command-line:

```
# service mysql stop
```

For servers that use `systemd`, you would execute the following from the command-line to shutdown MySQL:

```
# systemctl stop mysql
```

Note: **See Also**: For more information on `mysqldump`, see mysqldump Documentation.

Physical State Snapshot

There are two back-end methods available for Physical State Snapshots: `rsync` and `xtrabackup`.

The *Physical State Transfer Method* has the following advantages:

- These transfers physically copy the data from one node to the disk of the other, and as such do not need to interact with the database server at either end.
- These transfers do not require the database to be in working condition, as the donor node overwrites what was previously on the joining node disk.
- These transfers are faster.

The Physical State Transfer Method has the following disadvantages:

- These transfers require the joining node to have the same data directory layout and the same storage engine configuration as the donor node. For example, you must use the same file-per-table, compression, log file size and similar settings for InnoDB.
- These transfers are not accepted by servers with initialized storage engines.

  What this means is that when your node requires a state snapshot transfer, the database server must restart to apply the changes. The database server remains inaccessible to the client until the state snapshot transfer is complete, since it cannot perform authentication without the storage engines.

rsync

The fastest back-end method for State Snapshot Transfers is `rsync`. It carries all the advantages and disadvantages of of the Physical Snapshot Transfer. While it does block the donor node during transfer, `rsync` does not require database configuration or root access, which makes it easier to configure.

When using terabyte-scale databases, `rsync` is considerably faster, (1.5 to 2 times faster), than `xtrabackup`. This translates to a reduction in transfer times by several hours.

`rsync` also features the rsync-wan modification, which engages the `rsync` delta transfer algorithm. However, given that this makes it more I/O intensive, you should only use it when the network throughput is the bottleneck, which is usually the case in WAN deployments.

Note: The most common issue encountered with this method is due to incompatibilities between the various versions of `rsync` on the donor and joining nodes.

The `rsync` script runs on both donor and joining nodes. On the joiner, it starts `rsync` in server-mode and waits for a connection from the donor. On the donor, it starts `rsync` in client-mode and sends the contents of the data directory to the joining node.

```
wsrep_sst_method = rsync
```

For more information about `rsync`, see the rsync Documentation.

xtrabackup

The most popular back-end method for State Snapshot Transfers is `xtrabackup`. It carries all the advantages and disadvantages of a Physical State Snapshot, but is virtually non-blocking on the donor node.

`xtrabackup` only blocks the donor for the short period of time it takes to copy the MyISAM tables, (for instance, the system tables). If these tables are small, the blocking time remains very short. However, this comes at the cost of speed: a state snapshot transfer that uses `xtrabackup` can be considerably slower than one that uses `rsync`.

Given that `xtrabackup` copies a large amount of data in the shortest possible time, it may also noticeably degrade donor performance.

Note: The most common issue encountered with this method is due to its configuration. `xtrabackup` requires that you set certain options in the configuration file, which means having local root access to the donor server.

```
[mysqld]
wsrep_sst_auth = <wsrep_sst_user>:<password>
wsrep_sst_method = xtrabackup
datadir = /path/to/datadir
```

```
[client]
socket = /path/to/socket
```

For more information on `xtrabackup`, see the Percona XtraBackup User Manual and XtraBackup SST Configuration.

CHAPTER

NINETEEN

SCRIPTABLE STATE SNAPSHOT TRANSFERS

When a node sends and receives a *State Snapshot Transfer*, it manage it through processes that run external to the database server. In the event that you need more from these processes that the default behavior provides, Galera Cluster provides an interface for custom shell scripts to manage state snapshot transfers on the node.

Using the Common SST Script

Galera Cluster includes a common script for managing a *State Snapshot Transfer*, which you can use as a starting point in building your own custom script. The filename is `wsrep_sst_common.sh`. For Linux users, the package manager typically installs it for you in `/usr/bin`.

The common SST script provides ready functions for parsing argument lists, logging errors, and so on. There are no constraints on the order or number of parameters it takes. You can add to it new parameters and ignore any of the existing as suits your needs.

It assumes that the storage engine initialization on the receiving node takes place only after the state transfer is complete. Meaning that it copies the contents of the source data directory to the destination data directory (with possible variations).

State Transfer Script Parameters

When Galera Cluster starts an external process for state snapshot transfers, it passes a number of parameters to the script, which you can use in configuring your own state transfer script.

General Parameters

These parameters are passed to all state transfer scripts, regardless of method or whether the node is sending or receiving:

- `--role` The script is given a string, either `donor` or `joiner`, to indicate whether the node is using it to send or receive a state snapshot transfer.
- `--address` The script is given the IP address of the joiner node.

  When the script is run by the joiner, the node uses the value of either the *wsrep_sst_receive_address* (page 258) parameter or a sensible default formatted as `<ip_address>:<port>`. When the script is run by the donor, the node uses the value from the state transfer request.
- `--auth` The script is given the node authentication information.

When the script is run by the joiner, the node uses the value given to the *wsrep_sst_auth* (page 254) parameter. When the script is run by the donor, it uses the value given by the state transfer request.

- `--datadir` The script is given the path to the data directory. The value is drawn from the `mysql_real_data_home` parameter.
- `--defaults-file` The script is given the path to the `my.cnf` configuration file.

The values the node passes to these parameters varies depending on whether the node calls the script to send or receive a state snapshot transfer. For more information, see *Calling Conventions* (page 88) below.

Donor-specific Parameters

These parameters are passed only to state transfer scripts initiated by a node serving as the donor node, regardless of the method being used:

- `--gtid` The node gives the *Global Transaction ID*, which it forms from the state UUID and the sequence number, or seqno, of the last committed transaction.
- `--socket` The node gives the local server socket for communications, if required.
- `--bypass` The node specifies whether the script should skip the actual data transfer and only pass the Global Transaction ID to the receiving node. That is, whether the node should initiate an *Incremental State Transfer*.

Logical State Transfer-specific Parameters

These parameters are passed only to the `wsrep_sst_mysqldump.sh` state transfer script by both the sending and receiving nodes:

- `--user` The node gives to the script the database user, which the script then uses to connect to both donor and joiner database servers. Meaning, this user must be the same on both servers, as defined by the *wsrep_sst_auth* (page 254) parameter.
- `--password` The node gives to the script the password for the database user, as configured by the *wsrep_sst_auth* (page 254) paraemter.
- `--host` The node gives to the script the IP address of the joiner node.
- `--port` The node gives to the script the port number to use with the joiner node.
- `--local-port` The node gives to the script the port number to use in sending the state transfer.

Calling Conventions

In writing your own custom script for state snapshot transfers, there are certain conventions that you need to follow in order to accommodate how Galera Cluster calls the script.

Receiver

When the node calls for a state snapshot transfer as a joiner, it begins by passing a number of arguments to the state transfer script, as defined in *General Parameters* (page 87) above. For your own script you can choose to use or ignore these arguments as suits your needs.

After the script receives these arguments, prepare the node to accept a state snapshot transfer. For example, in the case of `wsrep_sst_rsync.sh`, the script starts `rsync` in server mode.

To signal that the node is ready to receive the state transfer, print the following string to standard output: `ready <address>:port\n`. Use the IP address and port at which the node is waiting for the state snapshot. For example:

```
ready 192.168.1.1:4444
```

The node responds by sending a state transfer request to the donor node. The node forms the request with the address and port number of the joiner node, the values given to *wsrep_sst_auth* (page 254), and the name of your script. The donor receives the request and uses these values as input parameters in running your script on that node to send back the state transfer.

When the joiner node receives the state transfer and finishes applying it, print to standard output the *Global Transaction ID* of the received state. For example:

```
e2c9a15e-5485-11e0-0800-6bbb637e7211:8823450456
```

Then exit the script with a `0` status, to indicate that the state transfer was successful.

Sender

When the node calls for a state snapshot transfer as a donor, it begins by passing a number of arguments to the state transfer script, as defined in *General Parameters* (page 87) above. For your own script, you can choose to use or ignore these arguments as suits your needs.

While your script runs, Galera Cluster accepts the following signals. You can trigger them by printing to standard output:

- `flush tables\n` Optional signal that asks the database server to run `FLUSH TABLES`. When complete, the database server creates a `tables_flushed` file in the data directory.
- `continue\n` Optional signal that tells the database server that it can continue to commit transactions.
- `done\n` Mandatory signal that tells the database server that the state transfer is complete and successful.

After your script sends the `done\n` signal, exit with a `0` return code.

In the event of failure, Galera Cluster expects your script to return a code that corresponds to the error it encountered. The donor node returns this code to the joiner through group communication. Given that its data directory now holds an inconsistent state, the joiner node then leaves the cluster and aborts the state transfer.

Note: Without the `continue\n` signal, your script runs in Total Order Isolation, which guarantees that no further commits occur until the script exits.

Enabling Scriptable SST's

Whether you use `wsrep_sst_common.sh` directly or decide to write a script of your own from scratch, the process for enabling it remains the same. The filename must follow the convention of `wsrep_sst_<name>.sh`, with `<name>` being the value that you give for the *wsrep_sst_method* (page 256) parameter in the configuration file.

For example, if you write a script with the filename `wsrep_sst_galera-sst.sh`, you would add the following line to your `my.cnf`:

```
wsrep_sst_method = galera-sst
```

When the node starts, it uses your custom script for state snapshot transfers.

CHAPTER

TWENTY

SYSTEM DATABASES

When you install Galera Cluster, it creates a set of system databases that it uses to store configuration information. For instance, the underlying database server uses the `mysql` database for system tables, which record such things as user names, passwords and what databases and tables those users can access.

Note: Nodes begin using the `wsrep_schema` database in version 4.0 of Galera Cluster. This feature does not exist in older versions of Galera Cluster.

wsrep Schema Database

Similar to the `performance_schema` and `information_schema` databases, the node uses `wsrep_schema` to store information about the state of the cluster, including the nodes that current part of the *Primary Component* as well as a history of nodes that were previously in the cluster.

You may find this information useful in diagnosing issues or in checking the state of the cluster through a monitoring solution.

Note: For more information on monitoring, see *Cluster Monitoring* (page 151).

The database contains the following tables:

- `cluster` Table contains the current state of the cluster. It contains a single row:

Column	Description
`cluster_uuid`	Provides the current UUID of the cluster.
`view_id`	Provides a sequential ID that indicates the current topology of the cluster. This value increments with changes in cluster membership.

- `members` Table contains the current cluster membership.

Column	Description
`node_uuid`	Provides the node UUID.
`cluster_uuid`	Provides the cluster UUID.
`node_name`	Provides the logical name of the node.
`node_incoming_address`	Provides the node IP address and port on which the node listens for incoming SQL client connections.

- `member_history` Table contains the complete cluster membership, including a row for every node in the current cluster as well as nodes that currently are not members.

Column	Description
node_uuid	Provides the node UUID.
cluster_uuid	Provides the cluster UUID.
last_view_id	Provides the view ID that was in use the last time the node was in the cluster.
node_name	Provides the logical name of the node.
node_incoming_address	Provides the node IP address and port on which the node listens for incoming SQL client connections.

If the value of the `last_view_id` column is less than the `view_id` on the `wsrep_schema.cluster` table, the node is currently not a part of the cluster.

You can query these tables the same as any other. For instance,

```
SELECT node_name, node_incoming_address
FROM wsrep_schema.members;

+----------------------+-----------------------+
| node_name            | node_incoming_address |
+----------------------+-----------------------+
| localhost.localdomain | 127.0.0.1:13000       |
+----------------------+-----------------------+
| localhost.localdomain | 127.0.0.1.13004       |
+----------------------+-----------------------+
```

Would indicate that the cluster current has two nodes, both of which are running on localhost. Alternatively, you might query `members_history` for a more complete membership list:

```
SELECT node_name, node_incoming_address, last_view_id
FROM wsrep_schema.members_history;

+----------------------+-----------------------+--------------+
| node_name            | node_incoming_address | last_view_id |
+----------------------+-----------------------+--------------+
| localhost.localdomain | 127.0.0.1:13000       |            4 |
+----------------------+-----------------------+--------------+
| localhost.localdomain | 127.0.0.1:13004       |            4 |
+----------------------+-----------------------+--------------+
| localhost.localdomain | 127.0.0.1:13008       |            3 |
+----------------------+-----------------------+--------------+
```

Indicates that, previously, there was a third node running on localhost that is not present in the current cluster topology. This is indicated by the `last_view_id` on one node being less than the others.

CHAPTER

TWENTYONE

SCHEMA UPGRADES

Schema changes are of particular interest related to Galara Cluster. Schema changes are DDL (Data Definition Language) statement executed on a database (e.g., `CREATE TABLE`, `GRANT`). These DDL statements change the database itself and are non-transactional.

Galera Cluster processes schema changes by two different methods:

- *Total Order Isolation* (page 93): Abbreviated as TOI, these are schema changes made on all cluster nodes in the same total order sequence, preventing other transations from committing for the duration of the operation.
- *Rolling Schema Upgrade* (page 94) Known also as RSU, these are schema changes run locally, affecting only the node on which they are run. The changes do not replicate to the rest of the cluster.

You can set the method for online schema changes by using the `wsrep_OSU_method` parameter in the configuration file, (`my.ini` or `my.cnf`, depending on your build) or through the ``mysql` client. Galera Cluster defaults to the Total Order Isolation method.

Note: See Also: If you're using Galera Cluster for Percona XtraDB Cluster, see the the pt-online-schema-change in the Percona Toolkit.

Total Order Isolation

When you want an online schema change to replicate through the cluster and don't care that other transactions will be blocked while the cluster processes the DDL statements, use the *Total Order Isolation* method. You would do this with the `SET` statement like so:

```
SET GLOBAL wsrep_OSU_method='TOI';
```

In Total Order Isolation, queries that change the schema replicate as statements to all nodes in the cluster. The nodes wait for all preceding transactions to commit simultaneously, then they execute the schema change in isolation. For the duration of the DDL processing, no other transactions can commit.

The main advantage of Total Order Isolation is its simplicity and predictability, which guarantees data consistency. Additionally, when using Total Order Isolation, you should take the following particularities into consideration:

- From the perspective of certification, schema upgrades in Total Order Isolation never conflict with preceding transactions, given that they only execute after the cluster commits all preceding transactions. What this means is that the certification interval for schema changes using this method has a zero length. Therefore, schema changes will never fail certification and their execution is guaranteed.
- Transactions that were in progress while the DDL was running and that involved the same database resource will get a deadlock error at commit time and will be rolled back.

- The cluster replicates the schema change query as a statement before its execution. There is no way to know whether or not individual nodes succeed in processing the query. This prevents error checking on schema changes in Total Order Isolation.

Rolling Schema Upgrade

When you want to maintain high-availability during schema upgrades and can avoid conflicts between new and old schema definitions, use the *Rolling Schema Upgrade* method. You would do this with the `SET` statement like so:

```
SET GLOBAL wsrep_OSU_method='RSU';
```

In Rolling Schema Upgrade, queries that change the schema are only processed on the local node. While the node processes the schema change, it desynchronizes with the cluster. When it finishes processing the schema change, it applies delayed replication events and synchronizes itself with the cluster.

To change a schema cluster-wide, you must manually execute the query on each node in turn. Bear in mind that during a rolling schema change the cluster continues to operate, with some nodes using the old schema structure while others use the new schema structure.

The main advantage of the Rolling Schema Upgrade is that it only blocks one node at a time. The main disadvantage of the Rolling Schema Upgrade is that it is potentially unsafe, and may fail if the new and old schema definitions are incompatible at the replication event level.

Note: Warning: To avoid conflicts between new and old schema definitions, execute SQL statements such as `CREATE TABLE` and `DROP TABLE` using the *Total Order Isolation* (page 93) method.

CHAPTER

TWENTYTWO

UPGRADING GALERA CLUSTER

You have three methods available in upgrading Galera Cluster:

- *Rolling Upgrade* (page 95) Where you upgrade each node one at a time.
- *Bulk Upgrade* (page 96) Where you upgrade all nodes together.
- *Provider Upgrade* (page 97) Where you only upgrade the Galera Replication Plugin.

There are advantages and disadvantages to each method. For instance, while a rolling upgrade may prove time consuming, the cluster remains up. Similarly, while a bulk upgrade is faster, problems can result in longer outages. You must choose the best method to implement in upgrading your cluster.

Rolling Upgrade

When you need the cluster to remain live and do not mind the time it takes to upgrade each node, use rolling upgrades.

In rolling upgrades, you take each node down individually, upgrade its software and then restart the node. When the node reconnects, it brings itself back into sync with the cluster, as it would in the event of any other outage. Once the individual finishes syncing with the cluster, you can move to the next in the cluster.

The main advantage of a rolling upgrade is that in the even that something goes wrong with the upgrade, the other nodes remain operational, giving you time to troubleshoot the problem.

Some of the disadvantages to consider in rolling upgrades are:

- **Time Consumption** Performing a rolling upgrade can take some time, longer depending on the size of the databases and the number of nodes in the cluster, during which the cluster operates at a diminished capacity.

  Unless you use *Incremental State Transfer*, as you bring each node back online after an upgrade, it initiates a full *State Snapshot Transfer*, which can take a long time to process on larger databases and slower state transfer methods.

  During the State Snapshot Transfer, the node continues to accumulate catch-up in the replication event queue, which it will then have to replay to synchronize with the cluster. At the same time, the cluster is operational and continues to add further replication events to the queue.

- **Blocking Nodes** When the node comes back online, if you use **mysqldump** for State Snapshot Transfers, the donor node remains blocked for the duration of the transfer. In practice, this means that the cluster is short two nodes for the duration of the state transfer, one for the donor node and one for the node in catch-up.

  Using **xtrabackup** or **rsync** with the LVM state transfer methods, you can avoid blocking the donor, but doing so may slow the donor node down.

Note: Depending on the load balancing mechanism, you may have to configure the load balancer not to direct requests at joining and donating nodes.

- **Cluster Availability** Taking down nodes for a rolling upgrade can greatly diminish cluster performance or availability, such as if there are too few nodes in the cluster to begin with or where the cluster is operating at its maximum capacity.

  In such cases, losing access to two nodes during a rolling upgrade can create situations where the cluster can no longer serve all requests made of it or where the execution times of each request increase to the point where services become less available.

- **Cluster Performance** Each node you bring up after an upgrade, diminishes cluster performance until the node buffer pool warms back up. Parallel applying can help with this.

To perform a rolling upgrade on Galera Cluster, complete the following steps for each node:

Note: Transfer all client connections from the node you are upgrading to the other nodes for the duration of this procedure.

1. Shut down the node.
2. Upgrade the software.
3. Restart the node.

Once the node finishes synchronizing with the cluster and completes its catch-up, move on tot he next node in the cluster. Repeat the procedure until you have upgraded all nodes in the cluster.

Tip: If you are upgraded a node that is or will be part of a weighted quorum, set the initial node weight to zero. This guarantees that if the joining node should fail before it finishes synchronizing, it will not affect any quorum computations that follow.

Bulk Upgrade

When you want to avoid time-consuming state transfers and the slow process of upgrading each node, one at a time, use a bulk upgrade.

In bulk upgrades, you take all of the nodes down in an idle cluster, perform the upgrades, then bring the cluster back online. This allows you to upgrade your cluster quickly, but does mean a complete service outage for your cluster.

Note: Warning: Always use bulk upgrades when using a two-node cluster, as the rolling upgrade would result in a much longer service outage.

The main advantage of bulk upgrade is that when you are working with huge databases, it is much faster and results in better availability than rolling upgrades.

The main disadvantage is that it relies on the upgrade and restart being quick. Shutting down InnoDB may take a few minutes as it flushes dirty pages. If something goes wrong during the upgrade, there is little time to troubleshoot and fix the problem.

Note: To minimize any issues that might arise from an upgrade, do not upgrade all of the nodes at once. Rather, run the upgrade on a single node first. If it runs without issue, upgrade the rest of the cluster.

To perform a bulk upgrade on Galera Cluster, complete the following steps:

1. Stop all load on the cluster
2. Shut down all the nodes
3. Upgrade software
4. Restart the nodes. The nodes will merge to the cluster without state transfers, in a matter of seconds.
5. Resume the load on the cluster

Note: You can carry out steps 2-3-4 on all nodes in parallel, therefore reducing the service outage time to virtually the time needed for a single server restart.

Provider-only Upgrade

When you only need to upgrade the Galera provider, you can further optimize the bulk upgrade to only take a few seconds.

Important: In provider-only upgrade, the warmed up InnoDB buffer pool is fully preserved and the cluster continues to operate at full speed as soon as you resume the load.

Upgrading Galera Replication Plugin

If you installed Galera Cluster for MySQL using the binary package from the Codership repository, you can upgrade the Galera Replication Plugin through your package manager..

To upgrade the Galera Replicator Plugin on an RPM-based Linux distribution, run the following command for each node in the cluster:

```
$ yum update galera
```

To upgrade the Galera Replicator Plugin on a Debian-based Linux distribution, run the following commands for each node in the cluster:

```
$ apt-get update
$ apt-get upgrade galera
```

When `apt-get` or `yum` finish, you will have the latest version of the Galera Replicator Plugin available on the node. Once this process is complete, you can move on to updating the cluster to use the newer version of the plugin.

Updating Galera Cluster

After you upgrade the Galera Replicator Plugin package on each node in the cluster, you need to run a bulk upgrade to switch the cluster over to the newer version of the plugin.

1. Stop all load on the cluster.
2. For each node in the cluster, issue the following queries:

```
SET GLOBAL wsrep_provider='none';
SET GLOBAL wsrep_provider='/usr/lib64/galera/libgalera_smm.so';
```

3. One any one node in the cluster, issue the following query:

```
SET GLOBAL wsrep_cluster_address='gcomm://';
```

4. For every other node in the cluster, issue the following query:

```
SET GLOBAL wsrep_cluster_address='gcomm://node1addr';
```

   For `node1addr`, use the address of the node in step 3.

5. Resume the load on the cluster.

Reloading the provider and connecting it to the cluster typically takes less than ten seconds, so there is virtually no service outage.

CHAPTER

TWENTYTHREE

RECOVERING THE PRIMARY COMPONENT

Cluster nodes can store the *Primary Component* state to disk. The node records the state of the Primary Component and the UUID's of the nodes connected to it. In the event of an outage, once all nodes that were part of the last saved state achieve connectivity, the cluster recovers the Primary Component.

In the event that the write-set position differs between the nodes, the recovery process also requires a full state snapshot transfer.

Note: **See Also**: For more information on this feature, see the *pc.recovery* (page 280) parameter. By default, it is enabled starting in version 3.6.

Understanding the Primary Component State

When a node stores the *Primary Component* state to disk, it saves it as the `gvwstate.dat` file. The node creates and updates this file when the cluster forms or changes the Primary Component. This ensures that the node retains the latest Primary Component state that it was in. If the node loses connectivity, it has the file to reference. If the node shuts down gracefully, it deletes the file.

```
my_uuid: d3124bc8-1605-11e4-aa3d-ab44303c044a
#vwbeg
view_id: 3 0dae1307-1606-11e4-aa94-5255b1455aa0 12
bootstrap: 0
member: 0dae1307-1606-11e4-aa94-5255b1455aa0 1
member: 47bbe2e2-1606-11e4-8593-2a6d8335bc79 1
member: d3124bc8-1605-11e4-aa3d-ab44303c044a 1
#vwend
```

The `gvwstate.dat` file breaks into two parts:

- **Node Information** Provides the node's UUID, in the `my_uuid` field.
- **View Information** Provides information on the node's view of the Primary Component, contained between the `#vwbeg` and `#vwend` tags.
  - `view_id` Forms an identifier for the view from three parts:
    - *view_type* Always gives a value of `3` to indicate the primary view.
    - *view_uuid* and *view_seq* together form a unique value for the identifier.
  - `bootstrap` Displays whether or not the node is bootstrapped, but does not effect the Primary Component recovery process.
  - `member` Displays the UUID's of nodes in this primary component.

Modifying the Saved Primary Component State

In the event that you find yourself in the unusual situation where you need to force certain nodes to join each other specifically, you can do so by manually changing the saved *Primary Component* state.

Note: Warning: Under normal circumstances, for safety reasons, you should entirely avoid editing or otherwise modifying the `gvwstate.dat` file. Doing so may lead to unexpected results.

When a node starts for the first time or after a graceful shutdown, it randomly generates and assigns to itself a UUID, which serves as its identifier to the rest of the cluster. If the node finds a `gvwstate.dat` file in the data directory, it reads the `my_uuid` field to find the value it should use.

By manually assigning arbitrary UUID values to the respective fields on each node, you force them to join each other, forming a new Primary Component, as they start.

For example, assume that you have three nodes that you would like to start together to form a new Primary Component for the cluster. You will need to generate three UUID values, one for each node.

```
SELECT UUID();

+--------------------------------------+
| UUID()                               |
+--------------------------------------+
| 47bbe2e2-1606-11e4-8593-2a6d8335bc79 |
+--------------------------------------+
```

You would then take these values and use them to modify the `gwstate.dat` file on `node1`:

```
my_uuid: d3124bc8-1605-11e4-aa3d-ab44303c044a
#vwbeg
view_id: 3 0dae1307-1606-11e4-aa94-5255b1455aa0 12
bootstrap: 0
member: 0dae1307-1606-11e4-aa94-5255b1455aa0 1
member: 47bbe2e2-1606-11e4-8593-2a6d8335bc79 1
member: d3124bc8-1605-11e4-aa3d-ab44303c044a 1
#vwend
```

Then repeat the process for `node2`:

```
my_uuid: 47bbe2e2-1606-11e4-8593-2a6d8335bc79
#vwbeg
view_id: 3 0dae1307-1606-11e4-aa94-5255b1455aa0 12
bootstrap: 0
member: 0dae1307-1606-11e4-aa94-5255b1455aa0 1
member: 47bbe2e2-1606-11e4-8593-2a6d8335bc79 1
member: d3124bc8-1605-11e4-aa3d-ab44303c044a 1
#vwend
```

And, the same again for `node3`:

```
my_uuid: d3124bc8-1605-11e4-aa3d-ab44303c044a
#vwbeg
view_id: 3 0dae1307-1606-11e4-aa94-5255b1455aa0 12
bootstrap: 0
member: 0dae1307-1606-11e4-aa94-5255b1455aa0 1
member: 47bbe2e2-1606-11e4-8593-2a6d8335bc79 1
```

```
member: d3124bc8-1605-11e4-aa3d-ab44303c044a 1
#vwend
```

Then start all three nodes without the bootstrap flag. When they start, Galera Cluster reads the `gvwstate.dat` file for each. It pulls its UUID from the file and uses those of the `member` field to determine which nodes it should join in order to form a new Primary Component.

CHAPTER

TWENTYFOUR

RESETTING THE QUORUM

Occasionally, you may find your nodes no longer consider themselves part of the *Primary Component*. For instance, in the event of a network failure, the failure of more than half of the cluster, or a split-brain situation. In these cases, the node come to suspect that there is another Primary Component, to which they are no longer connected.

When this occurs, all nodes return an `Unknown command` error to all queries. You can check if this is happening using the *wsrep_cluster_status* (page 291) status variable. Run the following query on each node:

```
SHOW GLOBAL STATUS LIKE 'wsrep_cluster_status';

+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_cluster_status | Primary |
+----------------------+---------+
```

The return value `Primary` indicates that it the node is part of the Primary Component. When the query returns any other value it indicates that the node is part of a nonoperational component. If none of the nodes return the value `Primary`, it means that you need to reset the quorum.

Note: Bear in mind that situations where none of the nodes show as part of the Primary Component are very rare. In the event that you do find one or more nodes that return the value `Primary`, this indicates an issue with network connectivity rather than a need to reset the quorum. Troubleshoot the connection issue. Once the nodes regain network connectivity they automatically resynchronize with the Primary Component.

Finding the Most Advanced Node

Before you can reset the quorum, you need to identify the most advanced node in the cluster. That is, you must find the node whose local database committed the last transaction. Regardless of the method you use in resetting the quorum, this node serves as the starting point for the new *Primary Component*.

Identifying the most advanced node in the cluster requires that you find the node with the most advanced sequence number, or seqno. You can determine this using the *wsrep_last_committed* (page 295) status variable.

From the database client on each node, run the following query:

```
SHOW STATUS LIKE 'wsrep_last_committed';

+----------------------+--------+
| Variable_name        | Value  |
+----------------------+--------+
```

```
| wsrep_last_committed | 409745 |
+----------------------+--------+
```

The return value is the seqno for the last transaction the node committed. The node that provides the highest seqno is the most advanced node in your cluster. Use it as the starting point in the next section when bootstrapping the new Primary Component.

Resetting the Quorum

When you reset the quorum what you are doing is bootstrapping the *Primary Component* on the most advanced node you have available. This node then functions as the new Primary Component, bringing the rest of the cluster into line with its state.

There are two methods available to you in this process: automatic and manual.

Note: The preferred method for a quorum reset is the automatic method. Unlike the manual method, automatic bootstraps preserve the write-set cache, or GCache, on each node. What this means is that when the new Primary Component starts, some or all of the joining nodes can provision themselves using the *Incremental State Transfer* (IST) method, rather than the much slower *State Snapshot Transfer* (SST) method.

Automatic Bootstrap

Resetting the quorum bootstraps the *Primary Component* onto the most advanced node. In the automatic method this is done by enabling *pc.bootstrap* (page 280) under *wsrep_provider_options* (page 251) dynamically through the database client. This makes the node a new Primary Component.

To perform an automatic bootstrap, on the database client of the most advanced node, run the following command:

```
SET GLOBAL wsrep_provider_options='pc.bootstrap=YES';
```

The node now operates as the starting node in a new Primary Component. Nodes in nonoperational components that have network connectivity attempt to initiate incremental state transfers if possible, state snapshot transfers if not, with this node, bringing their own databases up-to-date.

Manual Bootstrap

Resetting the quorum bootstraps the *Primary Component* onto the most advanced node. In the manual method this is done by shutting down the cluster, then starting it up again beginning with the most advanced node.

To manually bootstrap your cluster, complete the following steps:

1. Shut down all cluster nodes. For servers that use `init`, run the following command from the console:

   ```
   # service mysql stop
   ```

   For servers that use `systemd`, instead run this command:

   ```
   # systemctl stop mysql
   ```

2. Start the most advanced node with the `--wsrep-new-cluster` option. For servers that use `init`, run the following command:

```
# service mysql start --wsrep-new-cluster
```

For servers that use `systemd`, instead run this command:

```
# systemctl start mysql --wsrep-new-cluster
```

3. Start every other node in the cluster. For servers that use `init`, run the following command:

```
# service mysql start
```

For servers that use `systemd`, instead run this command:

```
# systemctl start mysql
```

When the first node starts with the `--wsrep-new-cluster` option, it initializes a new cluster using the data from the most advanced state available from the previous cluster. As the other nodes start they connect to this node and request state snapshot transfers, to bring their own databases up-to-date.

CHAPTER

TWENTYFIVE

MANAGING FLOW CONTROL

The cluster replicates changes synchronously through global ordering, but applies these changes asynchronously from the originating node out. To prevent any one node from falling too far behind the cluster, Galera Cluster implements a feedback mechanism called Flow Control.

Nodes queue the write-sets they receive in the global order and begin to apply and commit them on the database. In the event that the received queue grows too large, the node initiates Flow Control. The node pauses replication while it works the received queue. Once it reduces the received queue to a more manageable size, the node resumes replication.

Monitoring Flow Control

Galera Cluster provides global status variables for use in monitoring Flow Control. These break down into those status variables that count Flow Control pause events and those that measure the effects of pauses.

```
SHOW STATUS LIKE 'wsrep_flow_control_%';
```

Running these status variables returns only the node's present condition. You are likely to find the information more useful by graphing the results, so that you can better see the points where Flow Control engages.

For instance, using myq_gadgets:

```
$ mysql -u monitor -p -e 'FLUSH TABLES WITH READ LOCK;' \
   example_database
$ myq_status wsrep

Wsrep    Cluster        Node           Queue  Ops       Bytes   Flow          Conflct
time     name     P cnf  # name  cmt  sta  Up  Dn  Up  Dn   Up   Dn pau snt dst lcf␣
↪bfa
09:22:17 cluster1 P   3  3 node3 Sync T/T   0   0   0   9    0  13K 0.0   0 101   0 ␣
↪0
09:22:18 cluster1 P   3  3 node3 Sync T/T   0   0   0  18    0  28K 0.0   0 108   0 ␣
↪0
09:22:19 cluster1 P   3  3 node3 Sync T/T   0   4   0   3    0 4.3K 0.0   0 109   0 ␣
↪0
09:22:20 cluster1 P   3  3 node3 Sync T/T   0  18   0   0    0    0 0.0   0 109   0 ␣
↪0
09:22:21 cluster1 P   3  3 node3 Sync T/T   0  27   0   0    0    0 0.0   0 109   0 ␣
↪0
09:22:22 cluster1 P   3  3 node3 Sync T/T   0  29   0   0    0    0 0.9   1 109   0 ␣
↪0
09:22:23 cluster1 P   3  3 node3 Sync T/T   0  29   0   0    0    0 1.0   0 109   0 ␣
↪0
```

You can find the slave queue under the `Queue Dn` column and `FC pau` refers to Flow Control pauses. When the slave queue rises to a certain point, Flow Control changes the pause value to `1.0`. The node will hold to this value until the slave queue is worked down to a more manageable size.

Note: **See Also**: For more information on status variables that relate to flow control, see *Galera Status Variables* (page 287).

Monitoring for Flow Control Pauses

When Flow Control engages, it notifies the cluster that it is pausing replication using an `FC_Pause` event. Galera Cluster provides two status variables that monitor for these events.

- *wsrep_flow_control_sent* (page 295) This status variable shows the number of Flow Control pause events sent by the local node since the last status query.
- *wsrep_flow_control_recv* (page 294) This status variable shows the number of Flow Control pause events on the cluster, both those from other nodes and those sent by the local node, since the last status query.

Measuring the Flow Control Pauses

In addition to tracking Flow Control pauses, Galera Cluster also allows you to track the amount of time since the last `SHOW STATUS` query during which replication was paused due to Flow Control.

You can find this using one of two status variables:

- *wsrep_flow_control_paused* (page 294) Provides the amount of time replication was paused as a fraction. Effectively, how much the slave lag is slowing the cluster. The value `1.0` indicates replication is paused now.
- *wsrep_flow_control_paused_ns* (page 294) Provides the amount of time replication was paused in nanoseconds.

Configuring Flow Control

Galera Cluster provides two sets of parameters that allow you to manage how nodes handle the replication rate and Flow Control. The first set controls the write-set cache, the second relates to the points at which the node engages and disengages Flow Control.

Managing the Replication Rate

These three parameters control how nodes respond to changes in the replication rate. They allow you to manage the write-set cache on an individual node.

- *gcs.recv_q_hard_limit* (page 277) This sets the maximum write-set cache size (in bytes). The parameter value depends on the amount of RAM, swap size and performance considerations.

  The default value is `SSIZE_MAX` minus 2 gigabytes on 32-bit systems. There is no practical limit on 64-bit systems.

  In the event that a node exceeds this limit and *gcs.max_throttle* (page 277) is not set at `0.0`, the node aborts with an out-of-memory error. If *gcs.max_throttle* (page 277) is set at `0.0`., replication in the cluster stops.

- *gcs.max_throttle* (page 277) This sets the smallest fraction to the normal replication rate the node can tolerate in the cluster. If you set the parameter to `1.0` the node does not throttle the replication rate. If you set the parameter for `0.0`, a complete replication stop is possible.

  The default value is `0.25`.

- *gcs.recv_q_soft_limit* (page 277) This serves to estimate the average replication rate for the node. It is a fraction of the *gcs.recv_q_hard_limit* (page 277). When the replication rate exceeds the soft limit, the node calculates the average replication rate (in bytes) during this period. After that, the node decreases the replication rate linearly with the cache size so that at the *gcs.recv_q_hard_limit* (page 277) it reaches the value of the *gcs.max_throttle* (page 277) times the average replication rate.

  The default value is `0.25`.

  Note: When the node estimates the average replication rate, it can reach a value that is way off from the sustained replication rate.

The write-set cache grows semi-logarithmically with time after the *gcs.recv_q_soft_limit* (page 277) and the time needed for a state transfer to complete.

Managing Flow Control

These parameters control the point at which the node triggers Flow Control and the factor used in determining when it should disengage Flow Control and resume replication.

- *gcs.fc_limit* (page 276) This parameter determines the point at which Flow Control engages. When the slave queue exceeds this limit, the node pauses replication.

  It is essential for multi-master configurations that you keep this limit low. The certification conflict rate is proportional to the slave queue length. In master-save setups, you can use a considerably higher value to reduce Flow Control intervention.

  The default value is `16`.

- *gcs.fc_factor* (page 276) This parameter is used in determining when the node can disengage Flow Control. When the slave queue on the node drops below the value of *gcs.fc_limit* (page 276) times that of *gcs.fc_factor* (page 276) replication resumes.

  The default value is `0.5`.

Bear in mind that, while it is critical for multi-master operations that you use as small a slave queue as possible, the slave queue length is not so critical in master-slave setups. Depending on your application and hardware, the node can apply even 1K of write-sets in a fraction of a second. The slave queue length has no effect on master-slave failover.

Note: **Warning**: Cluster nodes process transactions asynchronously with regards to each other. Nodes cannot anticipate in any way the amount of replication data. Because of this, Flow Control is always reactive. That is, it only comes into affect after the node exceeds certain limits. It cannot prevent exceeding these limits or, when they are exceeded, it cannot make any guarantee as to the degree they are exceeded.

Meaning, if you were to configure a node with:

```
gcs.recv_q_hard_limit=100Mb
```

That node can still exceed that limit from a 1Gb write-set.

CHAPTER

TWENTYSIX

AUTO-EVICTION

When Galera Cluster notices erratic behavior in a node (e.g., unusually delayed response times), it can initiate a process to remove the node permanently from the cluster. This process is called *Auto-Eviction*.

Configuring Auto-Eviction

Each node in a cluster monitors the group communication response times from all other nodes in the cluster. When a cluster registers delayed responses from a node, it makes an entry about the node to the delayed list.

If the delayed node becomes responsive again for a fixed period, entries for that node are removed from the delayed list. However, if the node receives enough delayed entries and it's found on the delayed list for the majority of the cluster, the delayed node is evicted permanently from the cluster. Evicted nodes cannot rejoin the cluster until restarted.

You can configure the parameters of Auto-Eviction by setting the following options through *wsrep_provider_options* (page 251):

- *evs.delayed_margin* (page 270): This sets the time period that a node can delay its response from expectations until the cluster adds it to the delayed list. You must set this parameter to a value higher than the round-trip delay time (RTT) between the nodes.

  The default value is `PT1S`.

- *evs.delayed_keep_period* (page 269): This sets the time period you require a node to remain responsive until it's removed from the delayed list.

  The default value is `PT30S`.

- *evs.evict* (page 270) This sets the point in which the cluster triggers manual eviction to a certain node value. Setting this parameter as an empty string causes it to clear the evict list on the node where it is set.

- *evs.auto_evict* (page 268): This sets the number of entries allowed for a delayed node before Auto-Eviction takes place. Setting this to `0` disables the Auto-Eviction protocol on the node, though the node will continue to monitor node response times.

  The default value is `0`.

- *evs.version* (page 274): This sets which version of the EVS Protocol the node uses. Galera Cluster enables Auto-Eviction starting with EVS Protocol version 1.

  The default value is version `0`, for backwards compatibility.

Checking Eviction Status

If you suspect a node is becoming delayed, you can check its eviction status through Galera status variables. You can do this by using the `SHOW STATUS` statement from the database client. You would enter something like this:

```
SHOW STATUS LIKE 'wsrep_evs_delayed';
```

Below are the Galera status variables available to you:

- *wsrep_evs_state* (page 294): This status variable gives the internal state of the EVS Protocol.
- *wsrep_evs_delayed* (page 293): This status variable gives a comma separated list of nodes on the delayed list. The format used in that list is `uuid:address:count`. The `count` refers to the number of entries for the given delayed node.
- *wsrep_evs_evict_list* (page 293): This status variable lists the UUID's of evicted nodes.

Upgrading from Previous Versions

Releases of Galera Cluster prior to version 3.8 use EVS Protocol version 0, which is not directly compatible with version 1. As such, when you upgrade Galera Cluster for a node, the node continues to use EVS Protocol version 0.

To update the EVS Protocol version, you must first update the Galera Cluster software on each node. Here are the steps to do that:

1. Choose a node to start the upgrade and stop `mysqld` on it. For systems that use `init`, run the following command:

   ```
   # service mysql stop
   ```

   For systems that run `systemd`, use instead this command:

   ```
   # systemctl stop mysql
   ```

2. Once you stop `mysqld`, update the Galera Cluster software for the node. This can vary depending on how you installed Galera Cluster and which database server and operating system distribution the server uses.

3. Using a text editor, edit the configuration file, `/etc/my.cnf`. Set the EVS Protocol version to `0`.

   ```
   wsrep_provider_options="evs.version=0"
   ```

4. After saving the configuration file, restart the node. For systems that use `init`, run the following command:

   ```
   # service mysql start
   ```

   For systems that run `systemd`, instead use this command:

   ```
   # systemctl start mysql
   ```

5. Using the database client, check the node state with the `SHOW STATUS` statement like so:

   ```
   SHOW STATUS LIKE 'wsrep_local_state_comment';

   +----------------------------+--------+
   | Variable_name              | Value  |
   +----------------------------+--------+
   ```

```
| wsrep_local_state_comment   | Joined |
+-----------------------------+--------+
```

When the node state reads as `Synced`, the node is back in sync with the cluster.

Repeat the above steps on each node in the cluster to update them. Once this process is finished, the cluster will have the latest version of Galera Cluster. You can then begin updating the EVS Protocol version for each node. Below are the steps to do that:

1. On the first node, edit the configuration file, `/etc/my.cnf` with a text editor. Change the EVS Protocol version in it like so:

   ```
   wsrep_provider_options="evs.version=1"
   ```

2. After saving, restart `mysqld`. If your system uses `init`, run the following command:

   ```
   # service mysql restart
   ```

   For system that run `systemd`, use instead this command:

   ```
   # systemctl restart mysql
   ```

3. Using the database clinet, execute the `SHOW STATUS` statement to see if the EVS Protocol is using version 1. This time give it the new *wsrep_evs_state* (page 294) status variable.

   ```
   SHOW STATUS LIKE 'wsrep_evs_state';
   ```

   If the `SHOW STATUS` statement returns an empty set, something went wrong and your database server is still using EVS Protocol version 0. If it returns a results set, the EVS Protocol is on the right version and you can proceed.

4. Once you confirm the server is using the right version, check the node state. Execute the `SHOW STATUS` statement like so:

   ```
   SHOW STATUS LIKE 'wsrep_local_state_comment';

   +-----------------------------+--------+
   | Variable_name               | Value  |
   +-----------------------------+--------+
   | wsrep_local_state_comment   | Joined |
   +-----------------------------+--------+
   ```

   When the node state reads as `Synced`, the node is back in sync with the cluster.

These steps will update the EVS Protocol version for one node in a cluster. Repeat the process on each of the remaining nodes so that they all use EVS Protocol version 1.

Note: See Also: For more information on upgrading in general, see *Upgrading Galera Cluster* (page 95).

CHAPTER

TWENTYSEVEN

USING STREAMING REPLICATION

When a node replicates a transaction under *Streaming Replication*, it breaks the transaction into fragments, and then certifies and applies the fragments to slave nodes while the transaction is still in progress.

This allows you to work with larger data-sets, manage hot records, and help avoid conflicts and hangs in the case of long-running transactions.

Note: Streaming Replication is a new feature introduced in version 4.0 of Galera Cluster. Older versions do not support these operations.

Enabling Streaming Replication

The best practice when working with *Streaming Replication* is to enable it at a session-level for specific transactions, or parts thereof. The reason is that Streaming Replication increases the load on all nodes when applying and rolling back transactions. You'll get better performance if you only enable Streaming Replication on those transactions that won't run correctly without it.

Note: For more information, see *When to Use Streaming Replication* (page 73).

Enabling Streaming Replication requires you tp define the replication unit and number of units to use in forming the transaction fragments. Two parameters control these variables: *wsrep_trx_fragment_unit* (page 260) and *wsrep_trx_fragment_size* (page 260).

Below is an example of how to set these two parameters:

```
SET SESSION wsrep_trx_fragment_unit='statement';
SET SESSION wsrep_trx_fragment_size=3;
```

In this example, the fragment is set to three statements. For every three statements from a transaction, the node will generate, replicate and certify a fragment.

You can choose between several replication units when forming fragments:

- **bytes** This defines the fragment size in bytes.
- **events** This defines the fragment size as the number of binary log events generated.
- **rows** This defines the fragment size as the number of rows the fragment updates.
- **statements** This defines the fragment size as the number of statements in a fragment.

Choose the replication unit and fragment size that best suits the specific operation you want to run.

Streaming Replication with Hot Records

When your application needs to update frequently the same records from the same table (e.g., implementing a locking scheme, a counter, or a job queue), Streaming Replication allows you to force critical changes to replicate to the entire cluster.

For instance, consider the use case of a web application that creates work orders for a company. When the transaction starts, it updates the table `work_orders`, setting the queue position for the order. Under normal replication, two transactions can come into conflict if they attempt to update the queue position at the same time.

You can avoid this with Streaming Replication. As an example of how to do this, you would first execute the following SQL statement to begin the transaction:

```
START TRANSACTION;
```

After reading the data that you need for the application, you would enable Streaming Replication by executing the following two `SET` statements:

```
SET SESSION wsrep_trx_fragment_unit='statement';
SET SESSION wsrep_trx_fragment_size=1;
```

Next, set the user's position in the queue like so:

```
UPDATE work_orders
SET queue_position = queue_position + 1;
```

Witht that done, you can disable Streaming Replication by executing one of the previous `SET` statemetns, but with a different value like so:

```
SET SESSION wsrep_trx_fragment_size=0;
```

You can now perform whatever additional tasks you need to prepare the work order, and then commit the transaction:

```
COMMIT;
```

During the work order transaction, the client initiates Streaming Replication for a single statement, which it uses to set the queue position. The queue position update then replicates throughout the cluster, which prevents other nodes from coming into conflict with the new work order.

CHAPTER

TWENTYEIGHT

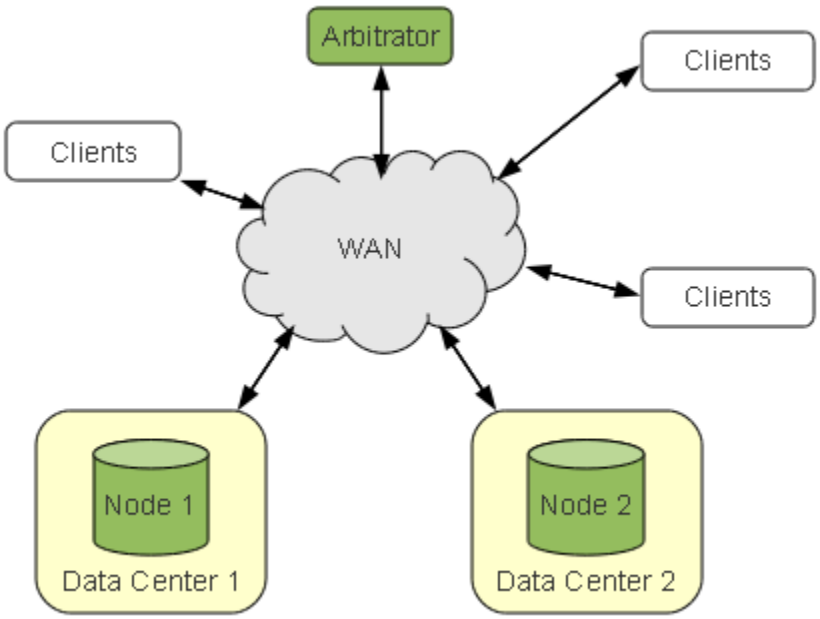
GALERA ARBITRATOR

It's recommended when deploying a Galera Cluster that you use a minimum of three instances: Three nodes, three datacenters and so on.

If the cost of adding resources (e.g., a third datacenter) is too much, you can use *Galera Arbitrator*. Galera Arbitrator is a member of a cluster that participates in voting, but not in the actual replication.

Note: **Warning** While Galera Arbitrator does not participate in replication, it does receive the same data as all other nodes. You must secure its network connection.

Galera Arbitrator serves two purposes: When you have an even number of nodes, it functions as an odd node, to avoid split-brain situations. It can also request a consistent application state snapshot, which is useful in making backups.



Galera Arbitrator

If one datacenter fails or loses its WAN connection, the node that sees the arbitrator—and by extension sees clients—continues operation.

Note: Even though Galera Arbitrator doesn't store data, it must see all replication traffic. Placing Galera Arbitrator

in a location with poor network connectivity to the rest of the cluster may lead to poor cluster performance.

In the event that Galera Arbitrator fails, it won't affect cluster operation. You can attach a new instance to the cluster at any time and there can be several instances running in the cluster.

Note: **See Also**: For more information on using Galera Arbitrator for making backups, see *Backing Up Cluster Data* (page 121).

Starting Galera Arbitrator

Galera Arbitrator is a separate daemon from Galera Cluster, called `garbd`. This means that you must start it separately from the cluster. It also means that you cannot configure Galera Arbitrator through the `my.cnf` configuration file.

How you configure Galera Arbitrator depends on how you start it. That is to say, whether it runs from the shell or as a service. These two methods are described in the next two sections.

Note: When Galera Arbitrator starts, the script executes a `sudo` statement as the user `nobody` during its process. There is a particular issue in Fedora and some other distributions of Linux, in which the default `sudo` configuration will block users that operate without `tty` access. To correct this, edit with a text editor the `/etc/sudoers` file and comment out this line:

```
Defaults requiretty
```

This will prevent the operating system from blocking Galera Arbitrator.

Starting Galera Arbitrator from the Shell

When starting Galera Arbitrator from the shell, you have two options as to how you may configure it. You can set the parameters through the command line arguments, as in the example here:

```
$ garbd --group=example_cluster \
    --address="gcomm://192.168.1.1,192.168.1.2,192.168.1.3" \
    --option="socket.ssl_key=/etc/ssl/galera/server-key.pem;socket.ssl_cert=/etc/ssl/
↪galera/server-cert.pem;socket.ssl_ca=/etc/ssl/galera/ca-cert.pem;socket.ssl_
↪cipher=AES128-SHA""
```

If you use SSL, it's necessary to specify the cipher. Otherwise, after initializing the ssl context an error will occur with a message saying, "Terminate called after throwing an instance of 'gu::NotSet"'.

If you don't want to enter the options every time you start Galera Arbitrator from the shell, you can set the options in the `arbtirator.config` configuration file:

```
# arbtirator.config
group = example_cluster
address = gcomm://192.168.1.1,192.168.1.2,192.168.1.3
```

Then, to enable those options when you start Galera Arbitrator, use the `--cfg` option like so:

```
$ garbd --cfg /path/to/arbitrator.config
```

For more information on the options available to Galera Arbitrator through the shell, run `garbd` with the `--help` argument.

```
$ garbd --help

Usage: garbd [options] [group address]

Configuration:
  -d [ --daemon ]       Become daemon
  -n [ --name ] arg     Node name
  -a [ --address ] arg  Group address
  -g [ --group ] arg    Group name
  --sst arg             SST request string
  --donor arg           SST donor name
  -o [ --options ] arg  GCS/GCOMM option list
  -l [ --log ] arg      Log file
  -c [ --cfg ] arg      Configuration file

Other options:
  -v [ --version ]      Print version
  -h [ --help ]         Show help message
```

In addition to the standard configuration, any parameter available to Galera Cluster also works with Galera Arbitrator, except for those prefixed by `repl`. When you start it from the shell, you can set those using the `--option` argument.

Note: **See Also**: For more information on the options available to Galera Arbitrator, see *Galera Parameters* (page 265).

Starting Galera Arbitrator as a Service

When starting Galera Aribtrator as a service, whether using `init` or `systemd`, you would use a different format for the configuration file than you would use when starting it from the shell. Below is an example of the configuration file:

```
# Copyright (C) 2013-2015 Codership Oy
# This config file is to be sourced by garbd service script.

# A space-separated list of node addresses (address[:port]) in the cluster:
GALERA_NODES="192.168.1.1:4567 192.168.1.2:4567"

# Galera cluster name, should be the same as on the rest of the node.
GALERA_GROUP="example_wsrep_cluster"

# Optional Galera internal options string (e.g. SSL settings)
# see http://galeracluster.com/documentation-webpages/galeraparameters.html
GALERA_OPTIONS="socket.ssl_cert=/etc/galera/cert/cert.pem;socket.ssl_key=/$"

# Log file for garbd. Optional, by default logs to syslog
LOG_FILE="/var/log/garbd.log"
```

In order for Galera Arbitrator to use the configuration file, you must place it in a file directory where your system looks for service configuration files. There is no standard location for this directory; it varies from distribution to distribution, though it usually in `/etc` and at least one sub-directory down. Some common locations include:

- `/etc/defaults/`
- `/etc/init.d/`

- /etc/systemd/
- /etc/sysconfig/

Check the documentation for the operating system distribution your server uses to determine where to place service configuration files.

Once you have the service configuration file in the right location, you can start the garb service. For systems that use init, run the following command:

```
# service garb start
```

For systems that run systemd, use instead this command:

```
# systemctl start garb
```

This starts Galera Arbitrator as a service. It uses the parameters set in the configuration file.

In addition to the standard configuration, any parameter available to Galera Cluster also works with Galera Arbitrator, excepting those prefixed by repl. When you start it as a service, you can set those using the GALERA_OPTIONS parameter.

Note: **See Also**: For more information on the options available to Galera Arbitrator, see *Galera Parameters* (page 265).

CHAPTER

TWENTYNINE

BACKING UP CLUSTER DATA

You can perform backups with Galera Cluster at the same regularity as with a standard database server, using a backup script. Since replication ensures that all nodes have the exact same data, running a backup script on one node will backup the data on all nodes in the cluster.

The problem with such a simple backup method, though, is that it lacks a *Global Transaction ID* (GTID). You can use backups of this kind to recover data, but they are insufficient for use in recovering nodes to a well-defined state. Furthermore, some backup procedures can block cluster operations during the backup.

Getting backups with the associated Global Transaction ID requires a different approach.

State Snapshot Transfer as Backup

Taking a full data backup is very similar to node provisioning through a *State Snapshot Transfer*. In both cases, the node creates a full copy of the database contents, using the same mechanism to associate a *Global Transaction ID* with the database state. Invoking backups through the state snapshot transfer mechanism has the following benefits:

- The node initiates the backup at a well-defined point.
- The node associates a Global Transaction ID with the backup.
- The node desyncs from the cluster to avoid throttling performance while making the backup, even if the backup process blocks the node.
- The cluster knows that the node is performing a backup and won't choose the node as a donor for another node.

In order to use this method for backups, you will need to use a script that implements both your preferred backup procedure and the Galera Arbitrator daemon, triggering it in a manner similar to a state snapshot transfer. You would execute such a script from the command-line like this:

```
$ garbd --address gcomm://192.168.1.2?gmcast.listen_addr=tcp://0.0.0.0:4444 \
  --group example_cluster --donor example_donor --sst backup
```

This command triggers donor node to invoke a script with the name `wsrep_sst_backup.sh`, which it looks for in the `PATH` for the `mysqld` process. When the donor reaches a well-defined point, a point where no changes are happening to the database, it runs the backup script passing the GTID corresponding to the current database state.

Note: In the command, '`?gmcast.listen_addr=tcp://0.0.0.0:4444`' is an arbitrary listen socket address that Galera Arbitrator opens to communicate with the cluster. You only need to specify this in the event that the default socket address (i.e., `0.0.0.0:4567` is busy).

Note: **See Also**: You may find it useful to create your backup script using a modified version of the standard state snapshot transfer script. For information on scripts of this kind, see *Scriptable State Snapshot Transfers* (page 87).

Part IV

Deployment

When you start Galera Cluster, you have to do so by initializing a series of nodes that are configured to communicate with each other and to replicate each other. Each node in the cluster is a particular instance of a MySQL, MariaDB, or Percona XtraDB database server. How your application servers interact with the cluster and how you manage the load and the individual nodes represents your deployment.

Cluster Deployment Variants (page 127)

Galera Cluster provides synchronous multi-master replication. This means that the nodes collectively operate as a single database server that listens across many interfaces. This section provides various examples of how you might deploy a cluster in relation to your application servers.

Load Balancing (page 133)

In high availability environments, you may sometimes encounter situations in which some nodes have a much greater load of traffic than others. If you discover such a situation, there may be some benefit in configuring and deploying load balancers between your application servers and Galera Cluster. Doing so will allow you to distribute client connections more evenly between the nodes, ensuring better performance.

This section provides guides to installing, configuring and deploying HAProxy, Pen and Galera Load Balancer, helping you to manage traffic between clients and the cluster.

Container Deployments (page 141)

When using the standard deployment methods of Galera Cluster, nodes run directly on the server hardware – interacting directly with the operating system (i.e., Linux, FreeBSD). By contrast, with container deployments nodes run in containerized virtual environments on the server. You may find containers useful in building portable deployments across numerous machines, when testing applications or scripting installations, or when isolating processes for security.

This section provides guides to installing, configuring and deploying Galera Cluster nodes in container instances using FreeBSD Jails and Docker.

CHAPTER

THIRTY

CLUSTER DEPLOYMENT VARIANTS

A Galera Cluster will consist of multiple nodes, preferably three or more. Each node is an instance of MySQL, MariaDB or Percona XtraDB that you convert to Galera Cluster, allowing you to use that node as a cluster base.

Galera Cluster provides synchronous multi-master replication. You can treat the cluster as a single database server that listens through many interfaces. To appreciate this, consider a typical *n*-tier application and the various benefits that would come from deploying it with Galera Cluster.

No Clustering

In the typical *n*-tier application cluster without database clustering, there's no concern for database replication or synchronization.

Internet traffic will be filtered down to your application servers, all of which read and write from the same DBMS server. Given that the upper tiers usually remain stateless, you can start as many instances as you need to meet the demand from the internet. Each instance stores its data in the data tier.

This solution is simple and easy to manage, but has a particular weakness in the data tier's lack of redundancy.

For example, if for any reason the DBMS server become unavailable, your application also becomes unavailable. This is the same whether the server crashes or it has been shut down for maintenance.

Similarly, this deployment also introduces performance concerns. While you can start as many instances as you need to meet the demands on your web and application servers, they can only so much load on the DBMS server can be handled before the load begins to slow end-user activities.

Whole Stack Clustering

In the typical *n*-tier application cluster you can avoid the performance bottleneck by building a whole stack cluster.

Internet traffic filters down to the application server, which stores data on its own dedicated DBMS server. Galera Cluster then replicates the data through to the cluster, ensuring it remains synchronous.

This solution is simple and easy to manage, especially if you can install the whole stack of each node on one physical machine. The direct connection from the application tier to the data tier ensures low latency.

There are, however, certain disadvantages to whole stack clustering that you should consider:

- **Lack of Redundancy**: When the database server fails, the whole stack fails. This is because the application server uses a dedicated database server. If the database server fails there's no alternative for the application server, so the whole stack goes down.

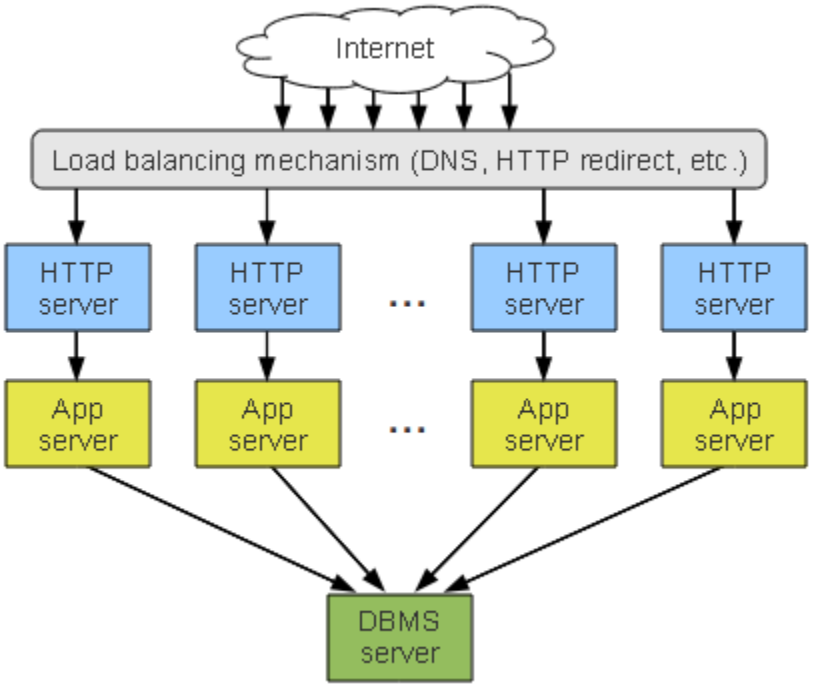


Fig. 30.1: *No Clustering*

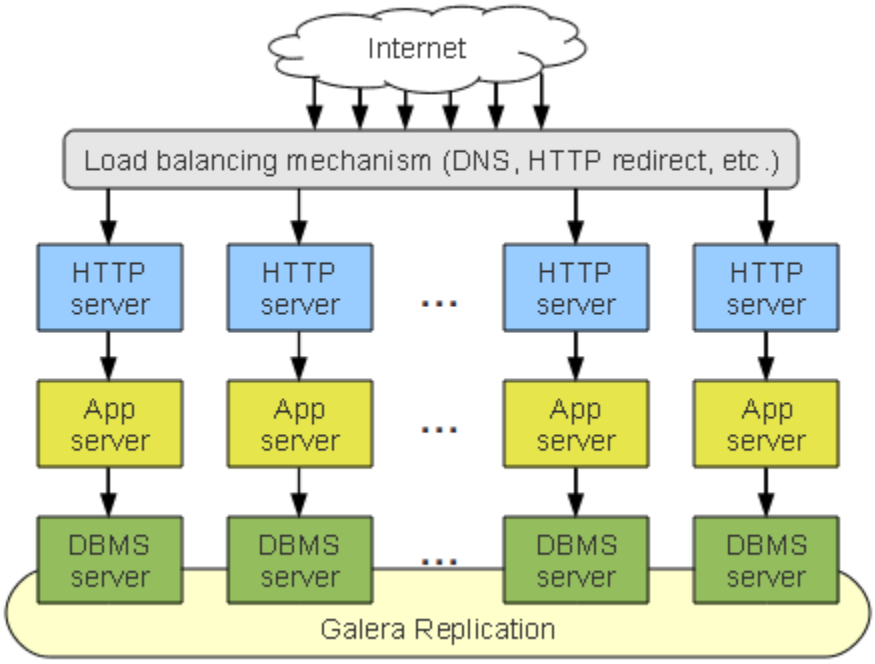


Fig. 30.2: *Whole Stack Cluster*

- **Inefficient Resource Usage**: A dedicated DBMS server for each application server will be overused. This is poor resource consolidation. For instance, one server with a 7 GB buffer pool is much faster than two servers with 4 GB buffer pools.
- **Increased Unproductive Overhead**: Each server reproduces the work of the other servers in the cluster. This redundancy is a drain on the server's resources.
- **Increased Rollback Rate**: Given that each application server writes to a dedicated database server, cluster-wide conflicts are more likely. This can increase the likelihood of corrective rollbacks.
- **Inflexibility**: There is no way for you to limit the number of master nodes or to perform intelligent load balancing.

Despite the disadvantages, however, this setup can prove very usable for several applications, depending on your needs.

Data Tier Clustering

To compensate for the shortcomings in whole stack clusters, you can cluster the data tier separately from your web and application servers.

With data tier clustering, the DBMS servers form a cluster distinct from your *n*-tier application cluster. The application servers treat the database cluster as a single virtual server, making calls through load balancers to the data tier.

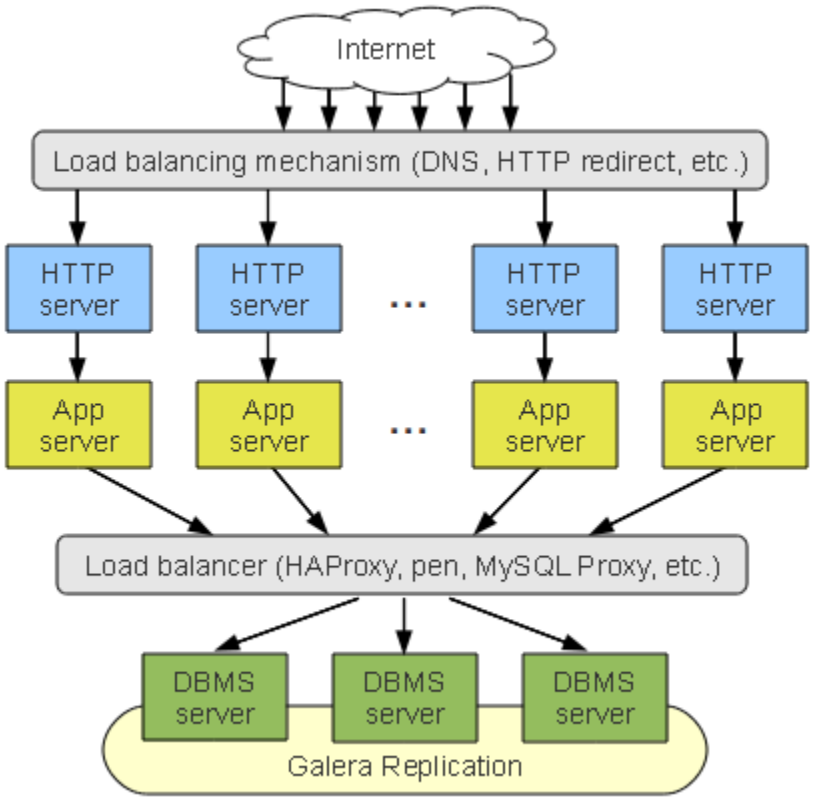


Fig. 30.3: *Data Tier Clustering*

In a data tier cluster, the failure of one node doesn't effect the rest of the cluster. Furthermore, resources are consolidated better and the setup is flexible. That is to say, you can assign nodes different roles using intelligent load balancing.

There are, however, certain disadvantages to consider in data tier clustering:

- **Complex Structure**: Since load balancers are involved, you must back them up in case of failure. This typically means that you have two more servers than you would otherwise, as well as a failover solution between them.
- **Complex Management**: You need to configure and reconfigure the load balancers whenever a DBMS server is added to the cluster or removed.
- **Indirect Connections**: The load balancers between the application cluster and the data tier cluster increase the latency for each query. As a result, this can easily become a performance bottleneck. You will need powerful load balancing servers to avoid this.
- **Scalability**: This setup doesn't scale well over several datacenters. Attempts to do so may reduce any benefits you gain from resource consolidation, given that each datacenter must include at least two DBMS servers.

Data Tier Clustering with Distributed Load Balancing

One solution to the limitations of data tier clustering is to deploy them with distributed load balancing. This method roughly follows the standard data tier cluster method, but includes a dedicated load balancer installed on each application server.

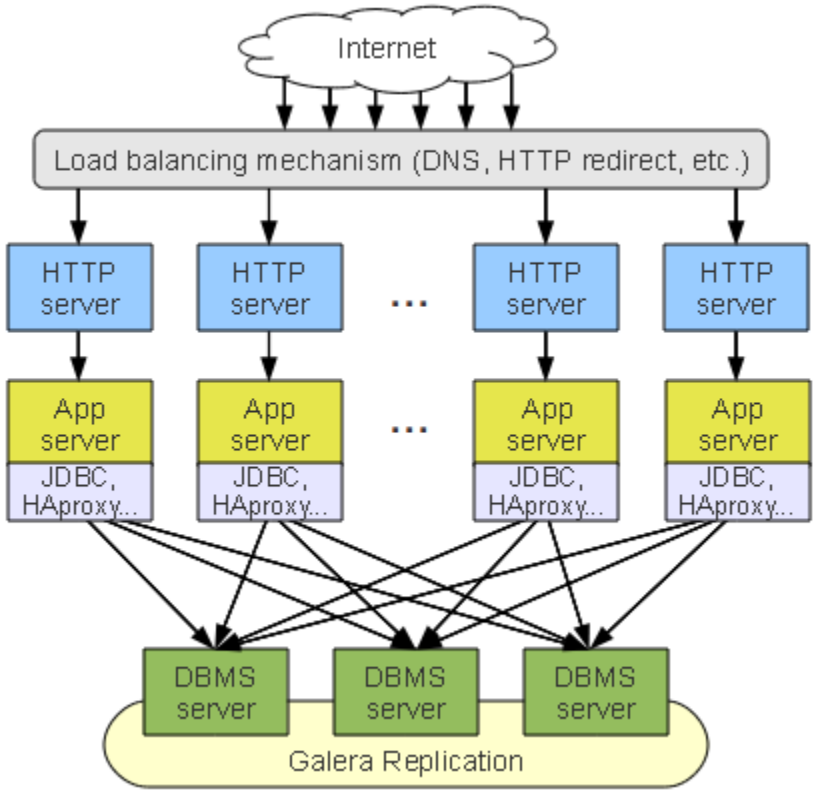


Fig. 30.4: *Data Tier Cluster with Distributed Load Balancing*

In this deployment, the load balancer is no longer a single point of failure. Furthermore, the load balancer scales

with the application cluster and thus is unlikely to become a bottleneck. Additionally, it minimizes the client-server communications latency.

Data tier clustering with distributed load balancing has the following disadvantage:

- **Complex Management**: Each application server deployed for an *n*-tier application cluster will require another load balancer that you need to set up, manage and reconfigure whenever you change or otherwise update the database cluster configuring.

Aggregated Stack Clustering

Besides the deployment methods already mentioned, you could set up a hybrid method that integrates whole stack and data tier clustering by aggregating several application stacks around single DBMS servers.

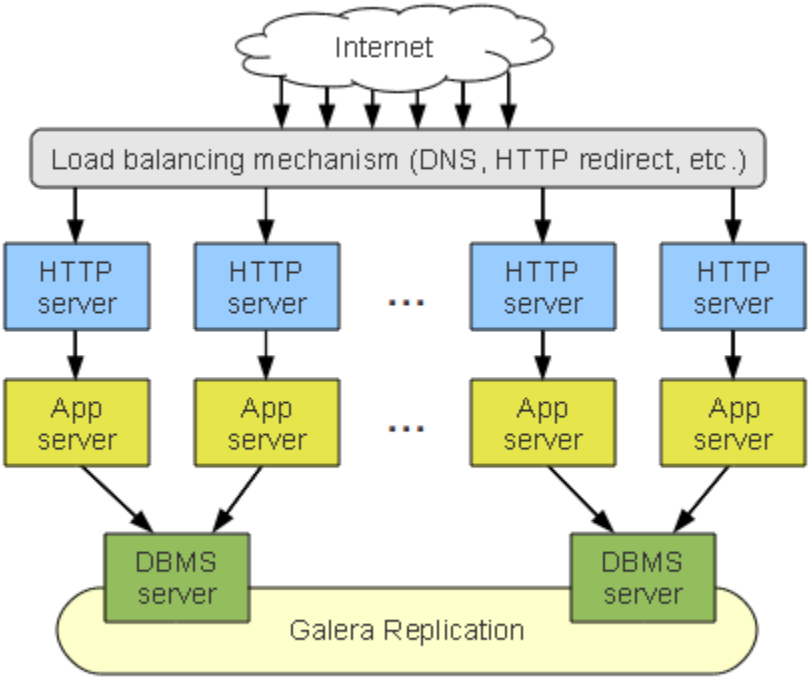


Fig. 30.5: *Aggregated Stack Clustering*

This layout improves on the resource utilization of the whole stack cluster, while maintaining its relative simplicity and direct DBMS connection benefits. It's also how a data tier cluster with distributed load balancing will look if you were to use only one DBMS server per datacenter.

The aggregated stack cluster is a good setup for sites that are not very large, but are hosted at more than one datacenter.

CHAPTER

THIRTYONE

LOAD BALANCING

Galera Cluster guarantees node consistency regardless of where and when the query is issued. In other words, you are free to choose a load-balancing approach that best suits your purposes. If you decide to place the load balancing mechanism between the database and the application, you can consider, for example, the following tools:

- **HAProxy** an open source TCP/HTTP load balancer.
- **Pen** another open source TCP/HTTP load balancer. Pen performs better than HAProxy on SQL traffic.
- **Galera Load Balancer** inspired by Pen, but is limited to balancing generic TCP connections only.

Note: For more information or ideas on where to use load balancers in your infrastructure, see *Cluster Deployment Variants* (page 127).

HAProxy

High Availability Proxy, or HAProxy is a single-threaded event-driven non-blocking engine that combines a fast I/O layer with a priority-based scheduler. You can use it to balance TCP connections between application servers and Galera Cluster.

Installation

HAProxy is available in the software repositories of most Linux distributions and it's the ports tree of FreeBSD. You can install it using the appropriate package manager.

- For DEB-based Linux distributions (e.g., Debian and Ubuntu), run the following from the command-line:

```
# apt-get install haproxy
```

- For RPM-based Linux distributions (e.g., Red Hat, Fedora and CentOS), execute the following from the command-line:

```
# yum install haproxy
```

- For SUSE-based Linux distributions (e.g., SUSE Enterprise Linux and openSUSE), execute instead this from the command-line:

```
# zypper install haproxy
```

- For FreeBSD and similar operating systems, HAProxy is available in the ports tree at */usr/ports/net/haproxy*. Alternatively, you can install it using the package manager like so:

```
# pkg install net/haproxy
```

Whichever method you use, it installs HAProxy on your server. In the event that the command for your Linux distribution or operating system doesn't work as expected, check your system's documentation or software repository for the correct procedure to install HAProxy.

Configuration

Configuration options for HAProxy are managed through an `haproxy.cfg` configuration file. The above package installations generally put this file in the `/etc/haproxy/` directory. However, it may have a different path depending on your operating system distribution.

To configure HAProxy to work with Galera Cluster, add the lines to the `haproxy.cfg` configuration file similar to the following:

```
# Load Balancing for Galera Cluster
listen galera 192.168.1.10:3306
     balance source
     mode tcp
     option tcpka
     option mysql-check user haproxy
     server node1 192.168.1.1:3306 check weight 1
     server node2 192.168.1.2:3306 check weight 1
     server node2 192.168.1.3:3306 check weight 1
```

You will create the proxy for Galera Cluster using the `listen` parameter. This gives HAProxy an arbitrary name for the proxy and defines the IP address and port you want it to listen on for incoming connections. Under this parameter, indent and define a series of options to tell HAProxy what you want it to do with these connections.

- `balance` defines the destination selection policy HAProxy should use in choosing which server it routes incoming connections.
- `mode tcp` defines the type of connections it should route. Galera Cluster uses TCP connections.
- `option tcpka` enables the keepalive function to maintain TCP connections.
- `option mysql-check user <username>` enables a database server check to determine whether the node is currently operational.
- `server <server-name> <IP_address> check weight 1` defines the nodes HAProxy should use in routing connections.

Destination Selection Policies

When HAProxy receives a new connection, there are a number of options available to define which algorithm it uses to choose where to route the connection. This algorithm is its destination selection policy. It's defined by the `balance` parameter.

- **Round Robin** directs new connections to the next destination in a circular order list, modified by the server's weight. Enable it with `balance roundrobin`.
- **Static Round Robin** directs new connections to the next destination in a circular order list, modified by the server's weight. Unlike the standard implementation of round robin, in static round robin you can't modify the server weight on the fly. Changing the server weight requires you to restart HAProxy. Enable it with `balance static-rr`.

- **Least Connected** directs new connections to the server with the smallest number of connections available, which is adjuted for the server's weight. Enable it with `balance leastconn`.
- **First** directs new connections to the first server with a connection slot available. They are chosen from the lowest numeric identifier to the highest. Once the server reaches its maximum connections value, HAProxy moves to the next in the list. Enable it with `balance first`.
- **Source Tracking** divides the source IP address by the total weight of running servers. Ensures that client connections from the same source IP always reach the same server. Enable it with `balance source`.

In the above configuration example, HAProxy is configured to use the source selection policy. For your implementation, choose the policy that works best with your infrastructure and load.

Enabling Database Server Checks

In addition to routing TCP connections to Galera Cluster, HAProxy can also perform basic health checks on the database server. When enabled, HAProxy attempts to establish a connection with the node and parses its response, or any errors, to determine if the node is operational.

For HAProxy, you can enable this through the `mysql-check` option. However, it requires that you also create a user in the cluster for HAProxy to use when connecting.

```
CREATE USER 'haproxy'@'192.168.1.10';
```

Make the user name the same as given in the `haproxy.cfg` configuration file for the `mysql-check` option. Replace the IP address with that of the server that runs HAProxy.

Using HAProxy

When you finish configuring HAProxy and the nodes to work with HAProxy, you can start it on the server. For servers that use `init`, run the following command:

```
# service haproxy start
```

For servers that use `systemd`, run instead this command:

```
# systemctl start haproxy
```

After doing this, the server will be running HAProxy. When new connections are made to this server, it routes them through to nodes in the cluster.

Pen

Pen is a high-scalability, high-availability, robust load balancer for TCP- and UDP-based protocols. You can use it to balance connections between application servers and Galera Cluster.

Installation

Pen is available in the software repositories of most Linux distributions. You can install it using a package manager.

- For DEB-based Linux distributions (i.e., Debian and Ubuntu), run the following from the command-line:

```
# apt-get install pen
```

- For RPM-based Linux distributions (i.e., Red Hat, Fedora and CentOS), use the `yum` utility instead by executing the following from the command-line:

```
# yum install pen
```

Whichever you use, they will install Pen on your system. In the event that the command for your distribution or operating system does not work as expected, check your system's documentation or software repository for information on the correct procedure to install Pen. For instance, on a RPM-based system, you may have to install the `yum` utility.

Using Pen

Once you've installed Pen on the load balancing server, you can launch it from the command-line by entering something like the following:

```
# pen -l pen.log -p pen.pid localhost:3306 \
      191.168.1.1:3306 \
      191.168.1.2:3306 \
      191.168.1.3:3306
```

When one of the application servers attempts to connect to the Pen server on port `3306`, Pen routes that connection to one of the Galera Cluster nodes.

Note: For more information on Pen configuration and use, see its manpage.

Server Selection

When Pen receives a new connection from the application servers, it first checks to see where the application was routed on the last connection and attempts to send traffic there. In the event that it cannot establish a connection, it falls back on a round-robin selection policy.

There are a number of options you can use to modify this behavior when you launch Pen.

- **Default Round Robin:** This directs all new connections to the next destination in a cirular order, without determining which server a client used the last time. You can enable this with the `-r` option.
- **Stubborn Selection:** In the event that the initial choice is unavailable, Pen closes the client connection. This is enabled with the `-s` option.
- **Hash Client IP Address:** Pen applies a hash on the client IP address for the initial server selection, making it more predictable where it routes client connections in the future.

Galera Load Balancer

Galera Load Balancer provides simple TCP connection balancing. It was developed with scalability and performance in mind. It draws on Pen for inspiration, but its functionality is limited to only balancing TCP connections. It provides several features:

- Support for configuring back-end servers at runtime.
- Support for draning servers.

- Support for the epoll API for routing performance.
- Support for multithreaded operations.
- Optional watchdog module to monitor destinations and adjust the routing table.

Installation

Unlike Galera Cluster, there is no binary installation available for Galera Load Balancer. Installing it on your system will requires you to build it from the source files. They're available on GitHub at glb.

To build Galera Load Balancer, you will need to complete a few steps. First, from a directory convenient for source builds (e.g., `/opt`), use the `git` utility to clone the GitHub repository for Galera Load Balancer. You would do this like so:

```
$ git clone https://github.com/codership/glb
```

Next, from within `glb` directory created by `git`, run the bootstrap script–which will be found in that directory.

```
$ cd glb/
$ ./bootstrap.sh
```

Now you will need to configure `make` to build on your system, then run `make` to build the application. After that, you'll use `make` to install it. This may seem like a lot, but it's simple. Just execute the following lines, one at a time, from the command-line:

```
$ ./configure

$ make

# make install
```

Note: Galera Load Balancer installs in the `/usr/sbin` directory. So you will need to run the last line above as root.

Once you've successfully execute everything above, Galera Load Balancer will be installed on your system. You can launch it from the command-line, using the `glbd` command.

In addition to the system daemon, you will also have installed `libglb`, a shared library for connection balancing with any Linux applications that use the `connect()` call from the C Standard Library.

Service Installation

The above installation procedure only installs Galera Load Balancer to be run manually from the command-line. However, you may find it more useful to run this application as a system service. To do this, you'll need to copy a couple of files to the appropriate directories.

In the source directory you cloned from GitHub, navigate into the `files` directory. Within that directory there is a configuration file and a service script that you need to copy to their relevant locations.

First, copy `glbd.sh` into `/etc/init.d` directory under a service name. You would execute the following from the command-line to do this:

```
# cp glbd.sh /etc/init.d/glb
```

Now, copy the default `glbd.cfg` file into the appropriate configuration directory. For Red Hat and its derivatives, this is `/etc/sysconfig/glbd.cfg`. For Debian and its derivatives, use `/etc/default/glbd.cfg`. For the former possibility, you would execute this from the command-line:

```
# cp glbd.cfg /etc/sysconfig/glbd.cfg
```

When you finish this, you will be able to manage Galera Load Balancer through the `service` command. For more information on available commands, see *Using Galera Load Balancer* (page 139).

Configuration

When you run Galera Load Balancer, you can configure its use through the command-line options. You can get a list of by exeduting `glb` with the `--help` option. For servers running Galera Load Balancer as a service, you can manage it through the `glbd.cfg` configuration file.

- *LISTEN_ADDR* (page 318): This is the address that Galera Load Balancer monitors for incoming client connections.
- *DEFAULT_TARGETS* (page 318): This specifies the default servers where Galera Load Balancer is to route incoming client connections. For this parameter, use the IP addresses for the nodes in your cluster.
- *OTHER_OPTIONS* (page 318): This is used to define additional Galera Load Balancer options. For example, you might want to set the balancing policy. Use the same format as you would from the command-line.

Below is an example of a *glbd.cfg* ' configuration file:

```
# Galera Load Balancer Configuration
LISTEN_ADDR="8010"
DEFAULT_TARGETS="192.168.1.1 192.168.1.2 192.168.1.3"
OTHER_OPTIONS="--random --top 3"
```

The `glbd.cfg` configuration file would be the one you copied into `/etc` as mentioned in the previous section.

Destination Selection Policies

Galera Load Balancer–both the system daemon and the shared library–supports five destination selection policies. When you run it from the command-line, you can define these using the command-line arguments. Otherwise, you'll have to add the arguments to the *OTHER_OPTIONS* (page 318) parameter in the `glbd.cfg` configuration file.

- **Least Connected**: This directs new connections to the server using the smallest number of connections possible. It will be adjusted for the server weight. This is the default policy.
- **Round Robin**: This sets new connections to the next destination in the circular order list. You can enable it with the *–round* (page 323) option.
- **Single**: This directs all connections to the single server with the highest weight of those available. Routing continues to that server until it fails, or until a server with a higher weight becomes available. You can enable it with the *–single* (page 323) option.
- **Random**: This will direct connections randomly to available servers. You can enable it using the *–random* (page 323) option.
- **Source Tracking**: This will direct connections originating from the same address to the same server. You can enable it with the *–source* (page 324) option.

Using Galera Load Balancer

The section on *Service Installation* (page 137) explained how to configure a system to run Galera Load Balancer as a service. If you do that, you can then manage common operations with the `service` command. The format for doing this is to enter `service`, followed by `glb`, and then an option.

Below is an example of how you might use `service` to get information on the Galera Load Balancer:

```
# service glb getinfo

Router:
-----------------------------------------
     Address       : weight   usage  cons
  192.168.1.1:4444 : 1.000    0.000    0
  192.168.1.2:4444 : 1.000    0.000    0
  192.168.1.3:4444 : 1.000    0.000    0
-----------------------------------------
Destinations: 3, total connections: 0
```

In the results shown here, you can see a list of servers available, their weight and usage, as well as the number of connections made to them.

The `service` script supports several operations. Below is a list of them and their uses:

- `start` is used to start `glb`, the Galera Load Balancer.
- `stop` will stop Galera Load Balancer.
- `restart` tells `glb` to stop and restart the Galera Load Balancer.
- `getinfo` is used as shown in the example above to retrieve the current routing information.
- `getstats` will provide performance statistics related to the cluster.
- `add <IP Address>` can be used to add an IP address from the routing table.
- `remove <IP Address>` will remove the designated IP address from the routing table.
- `drain <IP Address>` will sets the designated server to drain. When doing this, Galera Load Balancer won't send new connections to the given server, but it also won't kill existing connections. Instead, it waits for the connections to the specified server to end gracefully.

When adding an IP address to Galera Load Balancer at runtime, keep in mind that it must follow the convention, `IP Address:port:weight`. A hostname may be used instead of an IP address.

CHAPTER

THIRTYTWO

CONTAINER DEPLOYMENTS

In the standard deployment methods for Galera Cluster, a node runs on a server in the same manner as would an individual stand-alone instance of MySQL or MariaDB. In container deployments, a node runs in a containerized virtual environment on the server.

You may find these methods useful in portable deployments across numerous machines, testing applications that depend on Galera Cluster, process isolation for security, or scripting the installation and configuration process.

The configuration for a node running in a containerized environment remains primarily the same as a node running in the standard manner. However, there are some parameters that draw their defaults from the base system configurations. You will need to set these, manually. Otherwise, the jail will be unable to access the host file system.

- *wsrep_node_address* (page 246): A node determines the default address from the IP address on the first network interface. Jails cannot see the network interfaces on the host system. You need to set this parameter to ensure that the cluster is given the correct IP address for the node.
- *wsrep_node_name* (page 247): The node determines the default name from the system hostname. Jails have their own hostnames, distinct from that of the host system.

Bear in mind that the configuration file must be placed within the container `/etc` directory, not that of the host system.

Using Docker

Docker provides an open source platform for automatically deploying applications within software containers.

Galera Cluster can run from within a such a container, within Docker. You may find containers useful in portable deployment across numerous machines, testing applications that depend on Galera Cluster, or scripting the installation and configuration process.

Note: This guide assumes that you are only running one container node per server. For more information on running multiple nodes per server, see *Getting Started Galera with Docker, 'Part I <http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-1/>'_ and 'Part II <http://galeracluster.com/2015/05/getting-started-galera-with-docker-part-2-2/>'_.*

Configuring a Container

Images are the containers that Docker has available to run. There are a number of base images available through Docker Hub. You can pull these to your system through the `docker` command-line tool. You can also build new images.

When Docker builds a new image, it sources a `Dockerfile` to determine the steps that it needs to take in order to generate the image you want to use. This means you can script the installation and configuration process. Basically,

such a script would need to load the needed configuration files, run updates, and install packages when the image is built—all through a single command. Below is an example of how you might write such a script:

```
# Galera Cluster Dockerfile
FROM ubuntu:14.04
MAINTAINER your name <your.user@example.org>

ENV DEBIAN_FRONTEND noninteractive

RUN apt-get update
RUN apt-get install -y  software-properties-common
RUN apt-key adv --keyserver keyserver.ubuntu.com --recv BC19DDBA
RUN add-apt-repository 'deb http://releases.galeracluster.com/ubuntu trusty main'

RUN apt-get update
RUN apt-get install -y galera-3 galera-arbitrator-3 mysql-wsrep-5.6 rsync

COPY my.cnf /etc/mysql/my.cnf
ENTRYPOINT ["mysqld"]
```

This example follows the installation process for running Galera Cluster from within a Docker container running on Ubuntu. When you run the build command, Docker pulls down the Ubuntu 14.04 image from Docker Hub, if it's needed. It then runs each command in the `Dockerfile` to initialize the image for your use.

Configuration File

Before you build the container, you need to create the configuration file for the node. The `COPY` command in the `Dockerfile` example above copies `my.cnf`, the MySQL configuration file, from the build directory into the container.

For the most part, the configuration file for a node running within Docker is the same as when the node is running on a standard Linux server. However, there are some parameters that may not be included in the MySQL configuration file and instead use the default values from the underlying database system—or they may have been set manually, on-the-fly using the `SET` statement. For these parameters, since Docker can't access the host system, you may need to set them, manually.

- *wsrep_node_address* (page 246): A node will determine the default address from the IP address on the first network interface. Containers cannot see the network interfaces on the host system. Therefore, you will need to set this parameter to ensure the cluster is given the correct IP address for the node.
- *wsrep_node_name* (page 247): A node will determine the default host name from the system hostname. Containers have their own hostnames distinct from the host system.

Changes to the `my.cnf` file will not propagate into an existing container. Therefore, whenever you make changes to the configuration file, run the build again to create a new image with the updated configuration file. Docker caches each step of the build and only runs those steps that have changed when rebuilding. For example, using the `Dockerfile` example above, if you rebuild an image after changing `my.cnf`, Docker will run only the last two steps.

Note: If you need Docker to rerun the entire build, use the `--force-rm=true` option.

Building a Container Image

Building an image simplifies everyting—the node installation, the configuration and the deployment process—by reducing it to a single command. It will create a server instance where Galera Cluster is already installed, configured and ready to start.

You can build a container node using the `docker` command-line tool like so:

```
# docker build -t ubuntu:galera-node1 ./
```

When this command runs, Docker looks in the current working directory, (i.e., `./`), for the `Dockerfile`. It then follows each command in the `Dockerfile` to build the image. When the build is complete, you can view the addition among the available images by executing the following:

```
# docker images

REPOSITORY  TAG           IMAGE ID      CREATED         SIZE
ubuntu      galera-node-1 53b97c3d7740  2 minutes ago   362.7 MB
ubuntu      14.04         ded7cd95e059  5 weeks ago     185.5 MB
```

You can see in the results here that there is a working node image available for use as a container. You would launch it executing `docker run` at the command-line. You would repeat the build process on each server to create a node container image for Galera Cluster.

You would then update the container tag to help differentiate between each node by executing something like this:

```
[root@node2]# docker build -t ubuntu:galera-node2 ./
[root@node3]# docker build -t ubuntu:galera-node3 ./
```

Deploying a Container

When you finish building an image, you're ready to launch the node container. For each node, start the container using the Docker command-line tool with the `run` argument like so:

```
# docker run -i -d --name Node1 --host node1 \
      -p 3306:3306 -p 4567:4567 -p 4568:4568 -p 4444:4444 \
      -v /var/container_data/mysql:/var/lib/mysql \
      ubuntu:galera-node1
```

In this example, Docker launches a pre-built Ubuntu container tagged as `galera-node1`, which was built using the example `Dockerfile` from above. The `ENTRYPOINT` parameter is set to `/bin/mysqld` so that the container launches the database server when starting. You would modify the `--name` option in the example here for each node container you start.

You'll notice in the example here there are several `-p` options included. Those are described in the next section on Firewall Settings. The `-v` option is described in the section after it on Persistent Data.

Note: The above command starts a container node meant to be attached to an existing cluster. If you're starting the first node in a cluster, append the argument `--wsrep-new-cluster` to the end of the command. For more information, see *Starting the Cluster* (page 33).

Firewall Settings

When you launch the Docker container (i.e., `docker run` as shown above), the series of `-p` options connect the ports on the host system to those in the container. When the container is launched this way, nodes in the container have the same level of access to the network as the node would if it were running on the host system.

Use these settings, though, when you run only one container on the server. If you are running multiple containers on the server, you'll need a load balancer to handle and direct incoming connections to individual nodes.

For more information on configuring the firewall for Galera Cluster, see *Firewall Settings* (page 169).

Persistent Data

Docker containers are not meant to carry persistent data. When you close a container, the data it carries is lost. To avoid this problem, you can link volumes in the container to directories on the host file system. This is done with the `-v` option when you launch the container.

In the launch example above (i.e., the `docker run` lines), the `-v` argument connects the `/var/container_data/mysql/` directory to `/var/lib/mysql/` in the container. This replaces the local datadir inside the container with a symbolic link to a directory on the host system. This ensures that you won't lose data when the container restarts.

Database Client

Once you have a container node running, you can execute additional commands on the container using the `docker exec` command with the container name given with the `--name` parameter.

Using the example above, if you want access to the database client, you would run the following command:

```
# docker exec -ti Node1 /bin/mysql -u root -p
```

Notice here that `Node1` is the name given with the `--name` parameter in the example earlier.

Using Jails

In FreeBSD, jails provides a platform for securely deploying applications within virtual instances. You may find it useful in portable deployments across numerous machines for testing and security.

Galera Cluster can run from within a jail instance.

Preparing the Server

Jails exist as isolated file systems within, but unaware of, the host server. In order to grant the node running within the jail network connectivity with the cluster, you need to configure the network interfaces and firewall to redirect from the host into the jail.

Network Configuration

To begin, create a second loopback interface for the jail. this allows you to isolate jail traffic from `lo0`, the host loopback interface.

Note: For the purposes of this guide, the jail loopback is called `lo1`, if `lo1` already exists on your system, increment the digit to create one that does not already exist, (for instance, `lo2`).

To create a loopback interface, complete the following steps:

1. Using your preferred text editor, add the loopback interface to `/etc/rc.conf`:

   ```
   # Network Interface
   cloned_interfaces="${cloned_interfaces} lo1"
   ```

2. Create the loopback interface:

```
# service netif cloneup
```

This creates `lo1`, a new loopback network interface for your jails. You can view the new interface in the listing using the following command:

```
$ ifconfig
```

Firewall Configuration

FreeBSD provides packet filtering support at the kernel level. Using PF you can set up, maintain and inspect the packet filtering rule sets. For jails, you can route traffic from external ports on the host system to internal ports within the jail's file system. This allows the node running within the jail to have network access as though it were running on the host system.

To enable PF and create rules for the node, complete the following steps:

1. Using your preferred text editor, make the following additions to `/etc/rc.conf`:

   ```
   # Firewall Configuration
   pf_enable="YES"
   pf_rules="/etc/pf.conf"
   pflog_enable="YES"
   pflog_logfile="/var/log/pf.log"
   ```

2. Create the rules files for PF at `/etc/pf.conf`

   ```
   # External Network Interface
   ext_if="vtnet0"

   # Internal Network Interface
   int_if="lo1"

   # IP Addresses
   external_addr="host_IP_address"
   internal_addr="jail_IP_address_range"

   # Variables for Galera Cluster
   wsrep_ports="{3306,4567,4568,4444}"
   table <wsrep_cluster_address> persist {192.168.1.1,192.168.1.2,192.168.1.3}

   # Translation
   nat on $ext_if from $internal_addr to any -> ($ext_if)

   # Redirects
   rdr on $ext_if proto tcp from any to $external_addr/32 port 3306 -> jail_IP_
   ↪address port 3306
   rdr on $ext_if proto tcp from any to $external_addr/32 port 4567 -> jail_IP_
   ↪address port 4567
   rdr on $ext_if proto tcp from any to $external_addr/32 port 4568 -> jail_IP_
   ↪address port 4568
   rdr on $ext_if proto tcp from any to $external_addr/32 port 4444 -> jail_IP_
   ↪address port 4444

   pass in proto tcp from <wsrep_cluster_address> to any port $wsrep_ports keep state
   ```

   Replace `host_IP_address` with the IP address of the host server and `jail_IP_address` with the IP address you want to use for the jail.

3. Using `pfctl`, check for any typos in your PF configurations:

```
# pfctl -v -nf /etc/pf.conf
```

4. If `pfctl` runs without throwing any errors, start PF and PF logging services:

```
# service pf start
# service pflog start
```

The server now uses PF to manage its firewall. Network traffic directed at the four ports Galera Cluster uses is routed to the comparable ports within the jail.

Note: See Also: For more information on firewall configurations for FreeBSD, see *Firewall Configuration with PF* (page 172).

Creating the Node Jail

While FreeBSD does provide a manual interface for creating and managing jails on your server, (`jail(8)`), it can prove cumbersome in the event that you have multiple jails running on a server.

The application `ezjail` facilitates this process by automating common tasks and using templates and symbolic links to reduce the disk space usage per jail. It is available for installation through `pkg`. Alternative, you can build it through ports at `sysutils/ezjail`.

To create a node jail with `ezjail`, complete the following steps:

1. Using your preferred text editor, add the following line to `/etc/rc.conf`:

```
ezjail_enable="YES"
```

   This allows you to start and stop jails through the `service` command.

2. Initialize the `ezjail` environment:

```
# ezjail-admin install -sp
```

   This install the base jail system at `/usr/jails/`. It also installs a local build of the ports tree within the jail.

   Note: While the database server is not available for FreeBSD in ports or as a package binary, a port of the *Galera Replication Plugin* is available at `databases/galera`.

   ---

3. Create the node jail.

```
# ezjail-admin create galera-node 'lo1|192.168.68.1'
```

   This creates the particular jail for your node and links it to the `lo1` loopback interface and IP address. Replace the IP address with the local IP for internal use on your server. It is the same address as you assigned in the firewall redirects above for `/etc/pf.conf`.

   Note: Bear in mind that in the above command `galera-node` provides the hostname for the jail file system. As Galera Cluster draws on the hostname for the default node name, you need to either use a unique jail name for each node, or manually set *wsrep_node_name* (page 247) in the configuration file to avoid confusion.

   ---

4. Copy the `resolve.conf` file from the host file system into the node jail.

   ```
   # cp /etc/resolv.conf /usr/jails/galera-node/etc/
   ```

   This allows the network interface within the jail to resolve domain names in connecting to the internet.

5. Start the node jail.

   ```
   # ezjail-admin start galera-node
   ```

The node jail is now running on your server. You can view running jails using the `ezjail-admin` command:

```
# ezjail-admin list
STA JID  IP             Hostname     Root Directory
--- ---- -------------- ------------ ----------------------
DR  2    192.168.68.1   galera-node  /usr/jails/galera-node
```

While on the host system, you can access and manipulate files and directories in the jail file system from `/usr/jails/galera-node/`. Additionally, you can enter the jail directly and manipulate processes running within using the following command:

```
root@FreeBSDHost:/usr/jails # ezjail-admin console galera-node
root@galera-node:~ #
```

When you enter the jail file system, note that the hostname changes to indicate the transition.

Installing Galera Cluster

Regardless of whether you are on the host system or working from within a jail, currently, there is no binary package or port available to fully install Galera Cluster on FreeBSD. You must build the database server from source code.

The specific build process that you need to follow depends on the database server that you want to use:

- *Galera Cluster for MySQL* (page 17)
- Percona XtraDB Cluster
- *MariaDB Galera Cluster* (page 22)

Due to certain Linux dependencies, the *Galera Replication Plugin* cannot be built from source on FreeBSD. Instead you can use the port at `/usr/ports/databases/galera` or install it from a binary package within the jail:

```
# pkg install galera
```

This install the wsrep Provider file in `/usr/local/lib`. Use this path in the configuration file for the *wsrep_provider* (page 250) parameter.

Configuration File

For the most part, the configuration file for a node running in a jail is the same as when the node runs on a standard FreeBSD server. But, there are some parameters that draw their defaults from the base system. These you need to set manually, as the jail is unable to access the host file system.

- *wsrep_node_address* (page 246) The node determines the default address from the IP address on the first network interface. Jails cannot see the network interfaces on the host system. You need to set this parameter to ensure that the cluster is given the correct IP address for the node.

- *wsrep_node_name* (page 247) The node determines the default name from the system hostname. Jails have their own hostnames, distinct from that of the host system.

```
[mysqld]
user=mysql
#bind-address=0.0.0.0

# Cluster Options
wsrep_provider=/usr/lib/libgalera_smm.so
wsrep_cluster_address="gcomm://192.168.1.1, 192.168.1.2, 192.16.1.3"
wsrep_node_address="192.168.1.1"
wsrep_node_name="node1"
wsrep_cluster_name="example_cluster"

# InnoDB Options
default_storage_engine=innodb
innodb_autoinc_lock_mode=2
innodb_flush_log_at_trx_commit=0

# SST
wsrep_sst_method=rsync
```

If you are logged into the jail console, place the configuration file at `/etc/my.cnf`. If you are on the host system console, place it at `/usr/jails/galera-node/etc/my.cnf`. Replace `galera-node` in the latter with the name of the node jail.

Starting the Cluster

When running the cluster from within jails, you create and manage the cluster in the same manner as you would in the standard deployment of Galera Cluster on FreeBSD. The exception being that you must obtain console access to the node jail first.

To start the initial cluster node, run the following commands:

```
# ezjail-admin console galera-node
# service mysql start --wsrep-new-cluster
```

To start each additional node, run the following commands:

```
# ezjail-admin console galera-node
# service mysql start
```

Each node you start after the initial will attempt to establish network connectivity with the *Primary Component* and begin syncing their database states into one another.

Part V

Cluster Monitoring

Occasionally, you may want or need to check on the status of the cluster. For instance, you may want to check the state of nodes. You may want to check for network connectivity problems amongst nodes.

There are three methods available in monitoring cluster activity and replication health: query status variables through the database client; use a notification script; or through a third-party monitoring application (e.g., Nagios).

- *Using Status Variables* (page 153)

  In addition to the standard status variables in MySQL, Galera Cluster provides a series of its own status variables. These will allow you to check node and cluster states, as well as replication health through the database client.

- *Database Server Logs* (page 159)

  Queries entered through the database client will provide information on the current state of the cluster. However, to check its history for more systemic issues, you need to check the logs. This section provides a guide to the Galera Cluster parameter used to configure database logging to ensure it records the information you need.

- *Notification Command* (page 161)

  Although checking logs and status variables may give you the information you need while logged into a node, getting information from them is a manual process. Using the notification command, you can set the node to call a script in response to changes in cluster membership or node status. You can use this to raise alerts and adjust load balances. You can use it in a script for any situation in which you need the infrastructure to respond to changes in the cluster.

Note: You can also use Nagios for monitoring Galera Cluster. For more information, see Galera Cluster Nagios Plugin

---

CHAPTER

THIRTYTHREE

USING STATUS VARIABLES

From the database client, you can check the status of write-set replication throughout the cluster using standard queries. Status variables that relate to write-set replication have the prefix `wsrep_`, meaning that you can display them all using the following query:

```
SHOW GLOBAL STATUS LIKE 'wsrep_%';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_protocol_version | 5     |
| wsrep_last_committed   | 202   |
| ...                    | ...   |
| wsrep_thread_count     | 2     |
+------------------------+-------+
```

Note: **See Also**: In addition to checking status variables through the database client, you can also monitor for changes in cluster membership and node status through `wsrep_notify_cmd.sh`. For more information on its use, see *Notification Command* (page 161).

Checking Cluster Integrity

The cluster has integrity when all nodes in it receive and replicate write-sets from all other nodes. The cluster begins to lose integrity when this breaks down, such as when the cluster goes down, becomes partitioned, or experiences a split-brain situation.

You can check cluster integrity using the following status variables:

- *wsrep_cluster_state_uuid* (page 291) shows the cluster *state UUID*, which you can use to determine whether the node is part of the cluster.

  ```
  SHOW GLOBAL STATUS LIKE 'wsrep_cluster_state_uuid';

  +--------------------------+--------------------------------------+
  | Variable_name            | Value                                |
  +--------------------------+--------------------------------------+
  | wsrep_cluster_state_uuid | d6a51a3a-b378-11e4-924b-23b6ec126a13 |
  +--------------------------+--------------------------------------+
  ```

  Each node in the cluster should provide the same value. When a node carries a different value, this indicates that it is no longer connected to rest of the cluster. Once the node reestablishes connectivity, it realigns itself with

the other nodes.

- *wsrep_cluster_conf_id* (page 290) shows the total number of cluster changes that have happened, which you can use to determine whether or not the node is a part of the *Primary Component*.

```
SHOW GLOBAL STATUS LIKE 'wsrep_cluster_conf_id';

+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| wsrep_cluster_conf_id | 32    |
+-----------------------+-------+
```

  Each node in the cluster should provide the same value. When a node carries a different, this indicates that the cluster is partitioned. Once the node reestablish network connectivity, the value aligns itself with the others.

- *wsrep_cluster_size* (page 290) shows the number of nodes in the cluster, which you can use to determine if any are missing.

```
SHOW GLOBAL STATUS LIKE 'wsrep_cluster_size';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 15    |
+--------------------+-------+
```

  You can run this check on any node. When the check returns a value lower than the number of nodes in your cluster, it means that some nodes have lost network connectivity or they have failed.

- *wsrep_cluster_status* (page 291) shows the primary status of the cluster component that the node is in, which you can use in determining whether your cluster is experiencing a partition.

```
SHOW GLOBAL STATUS LIKE 'wsrep_cluster_status';

+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_cluster_status | Primary |
+----------------------+---------+
```

  The node should only return a value of `Primary`. Any other value indicates that the node is part of a nonoperational component. This occurs in cases of multiple membership changes that result in a loss of quorum or in cases of split-brain situations.

Note: **See Also**: In the event that you check all nodes in your cluster and find none that return a value of `Primary`, see *Resetting the Quorum* (page 103).

When these status variables check out and return the desired results on each node, the cluster is up and has integrity. What this means is that replication is able to occur normally on every node. The next step then is *checking node status* (page 155) to ensure that they are all in working order and able to receive write-sets.

Checking the Node Status

In addition to checking cluster integrity, you can also monitor the status of individual nodes. This shows whether nodes receive and process updates from the cluster write-sets and can indicate problems that may prevent replication.

- *wsrep_ready* (page 302) shows whether the node can accept queries.

```
SHOW GLOBAL STATUS LIKE 'wsrep_ready';

+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| wsrep_ready   | ON    |
+---------------+-------+
```

  When the node returns a value of `ON` it can accept write-sets from the cluster. When it returns the value `OFF`, almost all queries fail with the error:

```
ERROR 1047 (08501) Unknown Command
```

- *wsrep_connected* (page 292) shows whether the node has network connectivity with any other nodes.

```
SHOW GLOBAL STATUS LIKE 'wsrep_connected';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| wsrep_connected | ON    |
+-----------------+-------+
```

  When the value is `ON`, the node has a network connection to one or more other nodes forming a cluster component. When the value is `OFF`, the node does not have a connection to any cluster components.

  Note: The reason for a loss of connectivity can also relate to misconfiguration. For instance, if the node uses invalid values for the *wsrep_cluster_address* (page 237) or *wsrep_cluster_name* (page 238) parameters.

  Check the error log for proper diagnostics.

- *wsrep_local_state_comment* (page 300) shows the node state in a human readable format.

```
SHOW GLOBAL STATUS LIKE 'wsrep_local_state_comment';

+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| wsrep_local_state_comment | Joined |
+---------------------------+--------+
```

  When the node is part of the *Primary Component*, the typical return values are `Joining`, `Waiting on SST`, `Joined`, `Synced` or `Donor`. In the event that the node is part of a nonoperational component, the return value is `Initialized`.

  Note: If the node returns any value other than the one listed here, the state comment is momentary and transient. Check the status variable again for an update.

In the event that each status variable returns the desired values, the node is in working order. This means that it is receiving write-sets from the cluster and replicating them to tables in the local database.

Checking the Replication Health

Monitoring cluster integrity and node status can show you issues that may prevent or otherwise block replication. These status variables will help in identifying performance issues and identifying problem areas so that you can get the most from your cluster.

Note: Unlike other the status variables, these are differential and reset on every `SHOW STATUS` command. Execute the query a second time, about a minute after the first to get the current value.

Galera Cluster triggers a feedback mechanism called Flow Control to manage the replication process. When the local received queue of write-sets exceeds a certain threshold, the node engages Flow Control to pause replication while it catches up.

You can monitor the local received queue and Flow Control using the following status variables:

- *wsrep_local_recv_queue_avg* (page 297) shows the average size of the local received queue since the last status query.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue_avg';

+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| wsrep_local_recv_que_avg | 3.348452 |
+----------------------------+----------+
```

  When the node returns a value higher than `0.0` it means that the node cannot apply write-sets as fast as it receives them, which can lead to replication throttling.

  Note: In addition to this status variable, you can also use *wsrep_local_recv_queue_max* (page 298) and *wsrep_local_recv_queue_min* (page 298) to see the maximum and minimum sizes the node recorded for the local received queue.

- *wsrep_flow_control_paused* (page 294) shows the fraction of the time, since the status variable was last called, that the node paused due to Flow Control.

```
SHOW STATUS LIKE 'wsrep_flow_control_paused';

+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| wsrep_flow_control_paused | 0.184353 |
+----------------------------+----------+
```

  When the node returns a value of `0.0`, it indicates that the node did not pause due to Flow Control during this period. When the node returns a value of `1.0`, it indicates that the node spent the entire period paused. When the period between calls is one minute and the node returns `0.25`, it indicates that the node was paused for 15 seconds.

Ideally, the return value should stay as close to `0.0` as possible, since this means the node is not falling behind the cluster. In the event that you find that the node is pausing frequently, you can adjust the *wsrep_slave_threads* (page 253) parameter or you can exclude the node from the cluster.

- *wsrep_cert_deps_distance* (page 289) shows the average distance between the lowest and highest sequence number, or seqno, values that the node can possibly apply in parallel.

```
SHOW STATUS LIKE 'wsrep_cert_deps_distance';

+--------------------------+---------+
| Variable_name            | Value   |
+--------------------------+---------+
| wsrep_cert_deps_distance | 23.8889 |
+--------------------------+---------+
```

This represents the node's potential degree for parallelization. In other words, the optimal value you can use with the *wsrep_slave_threads* (page 253) parameter, given that there is no reason to assign more slave threads than transactions you can apply in parallel.

Detecting Slow Network Issues

While checking the status of Flow Control and the received queue can tell you how the database server copes with incoming write-sets, you can check the send queue to monitor for outgoing connectivity issues.

Note: Unlike other the status variables, these are differential and reset on every `SHOW STATUS` command. Execute the query a second time, about a minute after the first to get the current value.

---

wsrep_local_send_queue_avg (page 299) show an average for the send queue length since the last status query.

```
SHOW STATUS LIKE 'wsrep_local_send_queue_avg';

+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| wsrep_local_send_queue_avg | 0.145000 |
+----------------------------+----------+
```

Values much greater than `0.0` indicate replication throttling or network throughput issues, such as a bottleneck on the network link. The problem can occur at any layer from the physical components of your server to the configuration of the operating system.

Note: In addition to this status variable, you can also use *wsrep_local_send_queue_max* (page 299) and *wsrep_local_send_queue_min* (page 299) to see the maximum and minimum sizes the node recorded for the local send queue.

CHAPTER

THIRTYFOUR

DATABASE SERVER LOGS

Galera Cluster provides the same database server logging features available to MySQL, MariaDB and Percona XtraDB, depending on which you use. By default, it writes errors to a `<hostname>.err` file in the data directory. You can change this in the `my.cnf` configuration file using the log_error option, or by using the `--log-error` parameter.

Log Parameters

Galera Cluster provides parameters and wsrep options that allow you to enable error logging on events that are specific to the replication process. If you have a script monitoring the logs, these entires can give you information on conflicts occurring in the replication process.

- *wsrep_log_conflicts* (page 244): This parameter enables conflict logging for error logs. An example would be when two nodes attempt to write to the same row of the same table at the same time.
- *cert.log_conflicts* (page 268): This wsrep Provider option enables logging of information on certification failures during replication.
- *wsrep_debug* (page 240): This parameter enables debugging information for the database server logs.

  Note: Warning: In addition to useful debugging information, this parameter also causes the database server to print authentication information, (i.e., passwords) to the error logs. Don't enable it in production environments as it's a security vulnerability.

You can enable these through the `my.cnf` configuration file. The excerpt below is an example of these options and how they are enabled:

```
# wsrep Log Options
wsrep_log_conflicts=ON
wsrep_provider_options="cert.log_conflicts=ON"
wsrep_debug=ON
```

Additional Log Files

Whenever a node fails to apply an event on a slave node, the database server creates a special binary log file of the event in the data directory. The naming convention the node uses for the filename is `GRA_*.log`.

CHAPTER

THIRTYFIVE

NOTIFICATION COMMAND

While you can use the database client to check the status of your cluster, the individual nodes and the health of replication, you may find it counterproductive to log into the client on each node to run these checks. Galera Cluster provides a notification script and interface for customization, allowing you to automate the monitoring process for your cluster.

Notification Command Parameters

When the node registers a change in the cluster or itself that triggers the notification command, it passes a number of parameters in calling the script.

- `--status` The node passes a string indicating it's current state. For a list of the strings it uses, see *Node Status Strings* (page 161) below.
- `--uuid` The node passes a string of either *yes* or *no*, indicating whether it considers itself part of the *Primary Component*.
- `--members` The node passes a list of the current cluster members. For more information on the format of these listings, see *Member List Format* (page 162) below.
- `--index` The node passes a string that indicates its index value in the membership list.

Note: Only those nodes that in the `Synced` state accept connections from the cluster. For more information on node states, see *Node State Changes* (page 62).

Node Status Strings

The notification command passes one of six values with the `--status` parameter to indicate the current status of the node:

- `Undefined` Indicates a starting node that is not part of the Primary Component.
- `Joiner` Indicates a node that is part of the Primary Component that is receiving a state snapshot transfer.
- `Donor` Indicates a node that is part of the Primary Component that is sending a state snapshot transfer.
- `Joined` Indicates a node that is part of the Primary Component that is in a complete state and is catching up with the cluster.
- `Synced` Indicates a node that is syncrhonized with the cluster.
- `Error` Indicates that an error has occurred. This status string may provide an error code with more information on what occurred.

Members List Format

The notification command passes with the `--member` parameter a list containing entries for each node that is connected to the cluster component to which the node belongs. For each entry in the list the node uses this format:

```
<node UUID> / <node name> / <incoming address>
```

- **Node UUID** Refers to the unique identifier the node receives from the wsrep Provider.
- **Node Name** Refers to the node name, as you define it for the *wsrep_node_name* (page 247) parameter, in the configuration file.
- **Incoming Address** Refers to the IP address for client connections, as set for the *wsrep_node_incoming_address* (page 246) parameter, in the configuration file.

Example Notification Script

Nodes can call a notification script when changes happen in the membership of the cluster, that is when nodes join or leave the cluster. You can specify the name of the script the node calls using the *wsrep_notify_cmd* (page 247). While you can use whatever script meets the particular needs of your deployment, you may find it helpful to consider the example below as a starting point.

```
#!/bin/sh -eu

# This is a simple example of wsrep notification script (wsrep_notify_cmd).
# It will create 'wsrep' schema and two tables in it: 'membeship' and 'status'
# and fill them on every membership or node status change.
#
# Edit parameters below to specify the address and login to server.

USER=root
PSWD=rootpass
HOST=<host_IP_address>
PORT=3306

SCHEMA="wsrep"
MEMB_TABLE="$SCHEMA.membership"
STATUS_TABLE="$SCHEMA.status"

BEGIN="
   SET wsrep_on=0;
   DROP SCHEMA IF EXISTS $SCHEMA; CREATE SCHEMA $SCHEMA;
   CREATE TABLE $MEMB_TABLE (
      idx  INT UNIQUE PRIMARY KEY,
      uuid CHAR(40) UNIQUE, /* node UUID */
      name VARCHAR(32),     /* node name */
      addr VARCHAR(256)     /* node address */
   ) ENGINE=MEMORY;
   CREATE TABLE $STATUS_TABLE (
      size   INT,      /* component size   */
      idx    INT,      /* this node index  */
      status CHAR(16), /* this node status */
      uuid   CHAR(40), /* cluster UUID */
      prim   BOOLEAN   /* if component is primary */
   ) ENGINE=MEMORY;
   BEGIN;
```

```
   DELETE FROM $MEMB_TABLE;
   DELETE FROM $STATUS_TABLE;
"
END="COMMIT;"

configuration_change()
{
   echo "$BEGIN;"

   local idx=0

   for NODE in $(echo $MEMBERS | sed s/,/\ /g)
   do
      echo "INSERT INTO $MEMB_TABLE VALUES ( $idx, "
      # Don't forget to properly quote string values
      echo "'$NODE'" | sed  s/\\//\',\'/g
      echo ");"
      idx=$(( $idx + 1 ))
   done

   echo "
      INSERT INTO $STATUS_TABLE
      VALUES($idx, $INDEX,'$STATUS', '$CLUSTER_UUID', $PRIMARY);
   "

   echo "$END"
}

status_update()
{
   echo "
      SET wsrep_on=0;
      BEGIN;
      UPDATE $STATUS_TABLE SET status='$STATUS';
      COMMIT;
   "
}

COM=status_update # not a configuration change by default

while [ $# -gt 0 ]
do
   case $1 in
      --status)
         STATUS=$2
         shift
         ;;
      --uuid)
         CLUSTER_UUID=$2
         shift
         ;;
      --primary)
         [ "$2" = "yes" ] && PRIMARY="1" || PRIMARY="0"
         COM=configuration_change
         shift
         ;;
      --index)
         INDEX=$2
```

```
            shift
            ;;
        --members)
            MEMBERS=$2
            shift
            ;;
            esac
            shift
    done

# Undefined means node is shutting down
if [ "$STATUS" != "Undefined" ]
then
    $COM | mysql -B -u$USER -p$PSWD -h$HOST -P$PORT
fi

exit 0
```

When you finish editing the script to fit your needs, you need to move it into a directory in the `$PATH` environment variable or the binaries directory for your system. On Linux, the binaries directory is typically at `/usr/bin`, while on FreeBSD it is at `/usr/local/bin`.

```
# mv my-wsrep-notify.sh /usr/bin
```

In addition to this, given that the notification command contains your root password, change the ownership to the `mysql` user and make the script executable only to that user.

```
# chown mysql:mysql /usr/bin/my-wsrep-notify.sh
# chmod 700 /usr/bin/my-wsrep-notify.sh.
```

This ensures that only the `mysql` user executes and can read the notification script, preventing all other users from seeing your root password.

Enabling the Notification Command

You can enable the notification command through the *wsrep_notify_cmd* (page 247) parameter in the configuration file.

```
wsrep_notify_cmd=/path/to/wsrep_notify.sh
```

The node then calls the script for each change in cluster membership and node status. You can use these status changes in configuring load balancers, raising alerts or scripting for any other situation where you need your infrastructure to respond to changes to the cluster.

Galera Cluster provides a default script, `wsrep_notify.sh`, for you to use in handling notifications or as a starting point in writing your own custom notification script.

Part VI

Security

On occasion, you may want or need to enable degrees of security that go beyond the basics of Unix file permissions and secure database management. For situations such as these, you can secure both node communications and client connections between the application servers and the cluster.

- *Firewall Settings* (page 169)

  In order to use Galera Cluster, nodes must have access to a number of ports to maintain network connectivity with the cluster. While it was touched upon briefly in the *Installation* (page 12) chapter, this chapter provides more detailed guides on configuring a system firewall using `iptables`, FirewallD and PF.

- *SSL Settings* (page 175)

  To secure communications between nodes and from the application severs, you can enable encryption through the SSL protocol for client connections, replication traffic and State Snapshot Transfers. This chapter provides guidance to configuring SSL on Galera Cluster.

- *SELinux Configuration* (page 183)

  Without proper configuration, SELinux can either block nodes from communicating or it can block the database server from starting at all. When it does so, it causes the given process to fail silently, without any notification sent to standard output or error as to why. While you can configure SELinux to permit all activity from the database server, (as was explained in the *Installation* (page 12) chapter, this is not a good long-term solution.

  This chapter provides a guide to creating an SELinux security policy for Galera Cluster.

CHAPTER

THIRTYSIX

FIREWALL SETTINGS

Galera Cluster requires a number of ports to maintain network connectivity between the nodes. Depending on your deployment, you may not require all of these ports, but a cluster might require all of them on each node. Below is a list of these ports and their purpose:

- `3306` is the default port for MySQL client connections and *State Snapshot Transfer* using `mysqldump` for backups.
- `4567` is reserved for Galera Cluster replication traffic. Multicast replication uses both TCP and UDP transport on this port.
- `4568` is the port for *Incremental State Transfer*.
- `4444` is used for all other *State Snapshot Transfer*.

How these ports are enabled for Galera Cluster can vary depending upon your operating system distribution and what you use to configure the firewall.

Firewall Configuration with `iptables`

Linux provides packet filtering support at the kernel level. Using `iptables` and `ip6tables` you can set up, maintain and inspect tables of IPv4 and IPv6 packet filtering rules.

There are several tables that the kernel uses for packet filtering and within these tables are chains that it match specific kinds of traffic. In order to open the relevant ports for Galera Cluster, you need to append new rules to the `INPUT` chain on the filter table.

Opening Ports for Galera Cluster

Galera Cluster requires four ports for replication. There are two approaches to configuring the firewall to open these `iptables`. The method you use depends on whether you deploy the cluster in a LAN environment, such as an office network, or if you deploy the cluster in a WAN environment, such as on several cloud servers over the internet.

LAN Configuration

When configuring packet filtering rules for a LAN environment, such as on an office network, there are four ports that you need to open to TCP for Galera Cluster and one to UDP transport to enable multicast replication. This means five commands that you must run on each cluster node:

```
# iptables --append INPUT --in-interface eth0 \
   --protocol tcp --match tcp --dport 3306 \
   --source 192.168.0.1/24 --jump ACCEPT
```

```
# iptables --append INPUT --in-interface eth0 \
   --protocol tcp --match tcp --dport 4567 \
   --source 192.168.0.1/24 --jump ACCEPT
# iptables --append INPUT --in-interface eth0 \
   --protocol tcp --match tcp --dport 4568 \
   --source 192.168.0.1/24 --jump ACCEPT
# iptables --append INPUT --in-interface eth0 \
   --protocol tcp --match tcp --dport 4444 \
   --source 192.168.0.1/24 --jump ACCEPT
# iptables --append INPUT --in-interface eth0 \
   --protocol udp --match udp --dport 4567 \
   --source 192.168.0.1/24 --jump ACCEPT
```

These commands open the relevant ports to TCP and UDP transport. It assumes that the IP addresses in your network begin with 192.168.0.

Note: Warning: The IP addresses in the example are for demonstration purposes only. Use the real values from your nodes and netmask in your `iptables` configuration.

Galera Cluster can now pass packets through the firewall to the node, but the configuration reverts to default on reboot. In order to update the default firewall configuration, see *Making Firewall Changes Persistent* (page 171).

WAN Configuration

While the configuration shown above for LAN deployments offers the better security, only opening those ports necessary for cluster operation, it does not scale well into WAN deployments. The reason is that in a WAN environment the IP addresses are not in sequence. The four commands to open the relevant ports to TCP would grow to four commands per node on each node. That is, for ten nodes you would need to run four hundred `iptables` commands across the cluster in order to set up the firewall on each node.

Without much loss in security, you can instead open a range of ports between trusted hosts. This reduces the number of commands to one per node on each node. For example, firewall configuration in a three node cluster would look something like:

```
# iptables --append INPUT --protocol tcp \
    --source 64.57.102.34 --jump ACCEPT
# iptables --append INPUT --protocol tcp \
    --source 193.166.3.20  --jump ACCEPT
# iptables --append INPUT --protocol tcp \
    --source 193.125.4.10  --jump ACCEPT
```

When these commands are run on each node, they set the node to accept TCP connections from the IP addresses of the other cluster nodes.

Note: Warning: The IP addresses in the example are for demonstration purposes only. Use the real values from your nodes and netmask in your `iptables` configuration.

Galera Cluster can now pass packets through the firewall to the node, but the configuration reverts to default on reboot. In order to update the default firewall configuration, see *Making Firewall Changes Persistent* (page 171).

Making Firewall Changes Persistent

Whether you decide to open ports individually for LAN deployment or in a range between trusted hosts for a WAN deployment, the tables you configure in the above sections are not persistent. When the server reboots, the firewall reverts to its default state.

For systems that use `init`, you can save the packet filtering state with one command:

```
# service save iptables
```

For systems that use `systemd`, you need to save the current packet filtering rules to the path the `iptables` unit reads from when it starts. This path can vary by distribution, but you can normally find it in the `/etc` directory. For example:

- `/etc/sysconfig/iptables`
- `/etc/iptables/iptables.rules`

Once you find where your system stores the rules file, use `iptables-save` to update the file:

```
# iptables-save > /etc/sysconfig/iptables
```

When your system reboots, it now reads this file as the default packet filtering rules.

Firewall Configuration with FirewallD

The firewall daemon, or FirewallD, is an interface for dynamically managing firewalls on Linux operating systems. It allows you to set up, maintain and inspect IPv4 and IPv6 firewall rules.

FirewallD includes support for defining zones. This allows you to set the trust level of a given network connection or interface. For example, when deploying nodes that connect to each other over the internet–rather than a private network–you might configure your firewall around the `public` zone. This assumes that other computers on the network are untrusted and only accept designated connections.

Note: For more information on FirewallD, see the Documentation.

Opening Ports for Galera Cluster

Galera Cluster requires four open ports for replication over TCP. To use multicast replication, it also requires one for UDP transport. In order for this to work over FirewallD, you also need to add the database service to the firewall rules.

To enable the database service for FirewallD, you would enter something like the following at the command-line:

```
# firewall-cmd --zone=public --add-service=mysql
```

Next, you will need to open the TCP ports for Galera Cluster. Do this by executing the following from the command-line:

```
# firewall-cmd --zone=public --add-port=3306/tcp
# firewall-cmd --zone=public --add-port=4567/tcp
# firewall-cmd --zone=public --add-port=4568/tcp
# firewall-cmd --zone=public --add-port=4444/tcp
```

Optionally, if you would like to use multicast replication, execute the following from the command-line to open UDP transport on `4567`:

```
# firewall-cmd --zone=public --add-port=4567/udp
```

These commands dynamically configure FirewallD. Your firewall will then permit the rest of the cluster to connect to the node hosted on the server. Repeat the above commands on each server. Keep in mind, changes to the firewall made by this method are not persistent. When the server reboots, FirewallD will return to its default state.

Making Firewall Changes Persistent

The commands given in the above section allow you to configure FirewallD on a running server and update the firewall rules without restarting. However, these changes are not persistent. When the server restarts, FirewallD reverts to its default configuration. To change the default configuration, a somewhat different approach is required:

First, enable the database service for FirewallD by entering the following from the command-line:

```
# firewall-cmd --zone=public --add-service=mysql \
      --permanent
```

Now, you'll need to open the TCP ports for Galera Cluster. To do so, enter the following lines from the command-line:

```
# firewall-cmd --zone=public --add-port=3306/tcp \
      --permanent
# firewall-cmd --zone=public --add-port=4567/tcp \
      --permanent
# firewall-cmd --zone=public --add-port=4568/tcp \
      --permanent
# firewall-cmd --zone=public --add-port=4444/tcp \
      --permanent
```

If you would like to use multicast replication, execute the following command. It will open UDP transport on `4567`.

```
# firewall-cmd --zone=public --add-port=4567/udp \
      --permanent
```

Now you just need to reload the firewall rules, maintaining the current state information. To do this, executing the following:

```
# firewall-cmd --reload
```

These commands modify the default FirewallD settings and then cause the new settings to take effect, immediately. FirewallD will then be configured to allow the rest of the cluster to access the node. The configuration remains in effect after reboots. You'll have to repeat these commands on each server.

Firewall Configuration with PF

FreeBSD provides packet filtering support at the kernel level. Using PF you can set up, maintain and inspect the packet filtering rule sets.

Note: Warning: Different versions of FreeBSD use different versions of PF. Examples here are from FreeBSD 10.1, which uses the same version of PF as OpenBSD 4.5.

Enabling PF

In order to use PF on FreeBSD, you must first set the system up to load its kernel module. Additionally, you need to set the path to the configuration file for PF.

Using your preferred text editor, add the following lines to `/etc/rc.conf`:

```
pf_enable="YES"
pf_rules="/etc/pf.conf"
```

You may also want to enable logging support for PF and set the path for the log file. This can be done by adding the following lines to `/etc/rc.conf`:

```
pflog_enable="YES"
pflog_logfile="/var/log/pflog"
```

FreeBSD now loads the PF kernel module with logging features at boot.

Configuring PF Rules

In the above section, the configuration file for PF was set to `/etc/pf.conf`. This file allows you to set up the default firewall configuration that you want to use on your server. The settings you add to this file are the same for each cluster node.

There are two variables that you need to define for Galera Cluster in the PF configuration file: a list for the ports it needs open for TCP and a table for the IP addresses of nodes in the cluster.

```
# Galera Cluster Macros
wsrep_ports="{ 3306, 4567, 4568,4444}"
table <wsrep_cluster_address> persist {192.168.1.1 192.168.1.2 192.168.1.3}"
```

Once you have these defined, you can add the rule to allow cluster packets to pass through the firewall.

```
# Galera Cluster TCP Filter Rule
pass in proto tcp from <wsrep_cluster_address> to any port $wsrep_ports keep state
```

In the event that you deployed your cluster in a LAN environment, you need to also create on additional rule to open port `4568` to UDP transport for mutlicast replication.

```
# Galera Cluster UDP Filter Rule
pass in proto udp from <wsrep_cluster_address> to any port 4568 keep state
```

This defines the packet filtering rules that Galera Cluster requires. You can test the new rules for syntax errors using `pfctl`, with the `-n` options to prevent it from trying to load the changes.

```
# pfctl -v -nf /etc/pf.conf

wsrep_ports = "{ 3306, 4567, 4568, 4444 }"
table <wsrep_cluster_address> persist { 192.168.1.1 192.168.1.2 192.168.1.3 }
pass in proto tcp from <wsrep_cluster_address> to any port = mysql flags S/A/ keep␣
↪state
pass in proto tcp from <wsrep_cluster_address> to any port = 4567 flags S/SA keep␣
↪state
pass in proto tcp from <wsrep_cluster_address> to any port = 4568 flags S/SA keep␣
↪state
```

```
pass in proto tcp from <wsrep_cluster_address> to any port = krb524 falgs S/SA keep␣
↪state
pass in proto udp from <wsrep_cluster_address> to any port = 4568 keep state
```

If there are no syntax errors, `pfctl` prints each of the rules it adds to the firewall, (expanded, as in the example above). If there are syntax errors, it notes the line near where the errors occur.

Note: Warning: The IP addresses in the example are for demonstration purposes only. Use the real values from your nodes and netmask in your PF configuration.

Starting PF

When you finish configuring packet filtering for Galera Cluster and for any other service you may require on your FreeBSD server, you can start the service. This is done with two commands: one to start the service itself and one to start the logging service.

```
# service pf start
# service pflog start
```

In the event that you have PF running already and want to update the rule set to use the settings in the configuration file for PF, (for example, the rules you added for Galera Cluster), you can load the new rules through the `pfctl` command.

```
# pfctl -f /etc/pf.conf
```

CHAPTER

THIRTYSEVEN

SSL SETTINGS

Galera Cluster supports secure encrypted connections between nodes using SSL (Secure Socket Layer) protocol. This includes connections between database clients and servers through the standard SSL support in MySQL. It also includes encrypting replication traffic particular to Galera Cluster itself.

The SSL implementation is cluster-wide and doesn't support authentication for replication traffic. You must enable SSL for all nodes in the cluster or none of them.

SSL Certificates

Before you can enable encryption for your cluster, you first need to generate the relevant certificates for the nodes to use. This procedure assumes that you are using OpenSSL.

Note: See Also: This chapter only covers certificate generation. For information on its use in Galera Cluster, see *SSL Configuration* (page 177).

Generating Certificates

There are three certificates that you need to create in order to secure Galera Cluster: the Certificate Authority (CA) key and cert; the server certificate, to secure `mysqld` activity and replication traffic; and the client certificate to secure the database client and `stunnel` for state snapshot transfers.

Note: When certificates expire there is no way to update the cluster without a complete shutdown. You can minimize the frequency of this downtime by using large values for the `-days` parameter when generating your certificates.

CA Certificate

The node uses the Certificate Authority to verify the signature on the certificates. As such, you need this key and cert file to generate the server and client certificates.

To create the CA key and cert, complete the following steps:

1. Generate the CA key.

```
# openssl genrsa 2048 > ca-key.pem
```

2. Using the CA key, generate the CA certificate.

```
# openssl req -new -x509 -nodes -days 365000 \
      -key ca-key.pem -out ca-cert.pem
```

This creates a key and certificate file for the Certificate Authority. They are in the current working directory as `ca-key.pem` and `ca-cert.pem`. You need both to generate the server and client certificates. Additionally, each node requires `ca-cert.pem` to verify certificate signatures.

Server Certificate

The node uses the server certificate to secure both the database server activity and replication traffic from Galera Cluster.

1. Create the server key.

```
# openssl req -newkey rsa:2048 -days 365000 \
      -nodes -keyout server-key.pem -out server-req.pem
```

2. Process the server RSA key.

```
# openssl rsa -in server-key.pem -out server-key.pem
```

3. Sign the server certificate.

```
# openssl x509 -req -in server-req.pem -days 365000 \
      -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 \
      -out server-cert.pem
```

This creates a key and certificate file for the server. They are in the current working directory as `server-key.pem` and `server-cert.pem`. Each node requires both to secure database server activity and replication traffic.

Client Certificate

The node uses the client certificate to secure client-side activity. In the event that you prefer physical transfer methods for state snapshot transfers, `rsync` for instance, the node also uses this key and certificate to secure `stunnel`.

1. Create the client key.

```
# openssl req -newkey rsa:2048 -days 365000 \
      -nodes -keyout client-key.pem -out client-req.pem
```

2. Process client RSA key.

```
# openssl rsa -in client-key.pem -out client-key.pem
```

3. Sign the client certificate.

```
# openssl x509 -req -in client-req.pem -days 365000 \
      -CA ca-cert.pem -CAkey ca-key.pem -set_serial 01 \
      -out client-cert.pem
```

This creates a key and certificate file for the database client. They are in the current working directory as `client-key.pem` and `client-cert.pem`. Each node requires both to secure client activity and state snapshot transfers.

Verifying the Certificates

When you finish creating the key and certificate files, use `openssl` to verify that they were generated correctly:

```
# openssl verify -CAfile ca-cert.pem \
      server-cert.pem client-cert.pem

server-cert.pem: OK
client-cert.pem: OK
```

In the event that this verification fails, repeat the above process to generate replacement certificates.

Once the certificates pass verification, you can send them out to each node. Use a secure method, such as `scp` or `sftp`. The node requires the following files:

- Certificate Authority: `ca-cert.pem`.
- Server Certificate: `server-key.pem` and `server-cert.pem`.
- Client Certificate: `client-key.pem` and `client-cert.pem`.

Place these files in the `/etc/mysql/certs` directory of each node, or a similar location where you can find them later in configuring the cluster to use SSL.

SSL Configuration

When you finish generating the SSL certificates for your cluster, you need to enable it for each node. If you have not yet generated the SSL certificates, see *SSL Certificates* (page 175) for a guide on how to do so.

Note: For Gelera Cluster, SSL configurations are not dynamic. Since they must be set on every node in the cluster, if you are enabling this feature with a running cluster you need to restart the entire cluster.

Enabling SSL

There are three vectors that you can secure through SSL: traffic between the database server and client, replication traffic within Galera Cluster, and the *State Snapshot Transfer*.

Note: The configurations shown here cover the first two. The procedure for securing state snapshot transfers through SSL varies depending on the SST method you use. For more information, see *SSL for State Snapshot Transfers* (page 179).

Securing the Database

For securing database server and client connections, you can use the internal MySQL SSL support. In the event that you use logical transfer methods for state snapshot transfer, such as `mysqldump`, this is the only step you need to take in securing your state snapshot transfers.

In the configuration file, (`my.cnf`), add the follow parameters to each unit:

```
# MySQL Server
[mysqld]
ssl-ca = /path/to/ca-cert.pem
ssl-key = /path/to/server-key.pem
ssl-cert = /path/to/server-cert.pem

# MySQL Client Configuration
[mysql]
ssl-ca = /path/to/ca-cert.pem
ssl-key = /path/to/client-key.pem
ssl-cert = /path/to/client-cert.pem
```

These parameters tell the database server and client which files to use in encrypting and decrypting their interactions through SSL. The node will begin to use them once it restarts.

Securing Replication Traffic

In order to enable SSL on the internal node processes, you need to define the paths to the key, certificate and certificate authority files that you want the node to use in encrypting replication traffic.

- *socket.ssl_key* (page 285) The key file.
- *socket.ssl_cert* (page 284) The certificate file.
- *socket.ssl_ca* (page 284) The certificate authority file.

You can configure these options through the *wsrep_provider_options* (page 251) parameter in the configuration file, (that is, `my.cnf`).

```
wsrep_provider_options="socket.ssl_key=/path/to/server-key.pem;socket.ssl_cert=/path/
↪to/server-cert.pem;socket.ssl_ca=/path/to/cacert.pem"
```

This tells Galera Cluster which files to use in encrypting and decrypting replication traffic through SSL. The node will begin to use them once it restarts.

Configuring SSL

In the event that you want or need to further configure how the node uses SSL, Galera Cluster provides some additional parameters, including defining the cyclic redundancy check and setting the cryptographic cipher algorithm you want to use.

Note: See Also: For a complete list of available configurations available for SSL, see the options with the `socket.` prefix at *Galera Parameters* (page 265).

Configuring the Socket Checksum

Using the *socket.checksum* (page 284) parameter, you can define whether or which cyclic redundancy check the node uses in detecting errors. There are three available settings for this parameter, which are defined by an integer:

- `0` Disables the checksum.
- `1` Enables the CRC-32 checksum.
- `2` Enables the CRC-32C checksum.

The default configuration for this parameter is `1` or `2` depending upon your version. CRC-32C is optimized for and potentially hardware accelerated on Intel CPU's.

```
wsrep_provider_options = "socket.checksum=2"
```

Configuring the Encryption Cipher

Using the *socket.ssl_cipher* (page 285) parameter, you define which cipher the node uses in encrypting replication traffic. Galera Cluster uses whatever ciphers are available to the SSL implementation installed on the nodes. For instance, if you install OpenSSL on your node, Galera Cluster can use any cryptographic algorithms OpenSSL uses in ciphers.

The SSL configuration for Galera Cluster defaults to AES128-SHA, as this setting is considerably faster and no less secure than AES256.

```
wsrep_provider_options = "socket.ssl_cipher=AES128-SHA"
```

SSL for State Snapshot Transfers

When you finish generating the SSL certificates for your cluster, you can begin configuring the node for their use. Where *SSL Configuration* (page 177) covers how to enable SSL for replication traffic and the database client, this page covers enabling it for *State Snapshot Transfer* scripts.

The particular method you use to secure the State Snapshot Transfer through SSL depends upon the method you use in state snapshot transfers: `mysqldump` or `xtrabackup`.

Note: For Gelera Cluster, SSL configurations are not dynamic. Since they must be set on every node in the cluster, if you want to enable this feature with an existing cluster you need to restart the entire cluster.

Enabling SSL for `mysqldump`

The procedure for securing `mysqldump` is fairly similar to that of securing the database server and client through SSL. Given that `mysqldump` connects through the database client, you can use the same SSL certificates you created for replication traffic.

Before you shut down the cluster, you need to create a user for `mysqldump` on the database server and grant it privileges through the cluster. This ensures that when the cluster comes back up, the nodes have the correct privileges to execute the incoming state snapshot transfers. In the event that you use the *Total Order Isolation* online schema upgrade method, you only need to execute the following commands on a single node.

1. From the database client, check that you use Total Order Isolation for online schema upgrades.

   ```
   SHOW VARIABLES LIKE 'wsrep_OSU_method';

   +------------------+-------+
   | Variable_name    | Value |
   +------------------+-------+
   | wsrep_OSU_method | TOI   |
   +------------------+-------+
   ```

   If *wsrep_OSU_method* (page 249) is set to *Rolling Schema Upgrade*, or `ROI`, then you need to execute the following commands on each node individually.

2. Create a user for `mysqldump`.

```
CREATE USER 'sst_user'$'%' IDENTIFIED BY PASSWORD 'sst_password';
```

   Bear in mind that, due to the manner in which the SST script is called, the user name and password must be the same on all nodes.

3. Grant privileges to this user and require SSL.

```
GRANT ALL ON *.* TO 'sst_user'@'%' REQUIRE SSL;
```

4. From the database client on a different node, check to ensure that the user has replicated to the cluster.

```
SELECT User, Host, ssl_type FROM mysql.user WHERE User='sst_user';

+----------+------+----------+
| User     | Host | ssl_type |
+----------+------+----------+
| sst_user | %    | Any      |
+----------+------+----------+
```

This configures and enables the `mysqldump` user for the cluster.

Note: In the event that you find, *wsrep_OSU_method* (page 249) set to `ROI`, you need to manually create the user on each node in the cluster. For more information on rolling schema upgrades, see *Schema Upgrades* (page 93).

With the user now on every node, you can shut the cluster down to enable SSL for `mysqldump` State Snapshot Transfers.

1. Using your preferred text editor, update the `my.cnf` configuration file to define the parameters the node requires to secure state snapshot transfers.

```
# MySQL Server
[mysqld]
ssl-ca = /path/to/ca-cert.pem
ssl-key = /path/to/server-key.pem
ssl-cert = /path/to/server-cert.pem

# MySQL Client Configuration
[client]
ssl-ca = /path/to/ca-cert.pem
ssl-key = /path/to/client-key.pem
ssl-cert = /path/to/client-cert.pem
```

2. Additionally, configure *wsrep_sst_auth* (page 254) with the SST user authentication information.

```
[mysqld]
# mysqldump SST auth
wsrep_sst_auth = sst_user:sst_password
```

This configures the node to use `mysqldump` for state snapshot transfers over SSL. When all nodes are updated to SSL, you can begin restarting the cluster. For more information on how to do this, see *Starting the Cluster* (page 33).

Enabling SSL for `xtrabackup`

The *Physical State Transfer Method* for state snapshot transfers, uses an external script to copy the physical data directly from the file system on one cluster node into another. Unlike `rsync`, `xtrabackup` includes support for SSL encryption built in.

Configurations for `xtrabackup` are handled through the `my.cnf` configuration file, in the same as the database server and client. Use the `[sst]` unit to configure SSL for the script. You can use the same SSL certificate files as the node uses on the database server, client and with replication traffic.

```
# xtrabackup Configuration
[sst]
encrypt = 3
tca = /path/to/ca.pem
tkey = /path/to/key.pem
tcert = /path/to/cert.pem
```

When you finish editing the configuration file, restart the node to apply the changes. `xtrabackup` now sends and receives state snapshot transfers through SSL.

Note: In order to use SSL with `xtrabackup`, you need to set *wsrep_sst_method* (page 256) to `xtrabackup-v2`, instead of `xtrabackup`.

CHAPTER

THIRTYEIGHT

SELINUX CONFIGURATION

Security-Enhanced Linux, or SELinux, is a kernel module for improving security of Linux operating systems. It integrates support for access control security policies, including mandatory access control (MAC), that limit user applications and system daemons access to files and network resources. Some Linux distributions, such as Fedora, ship with SELinux enabled by default.

In the context of Galera Cluster, systems with SELinux may block the database server, keeping it from starting or preventing the node from establishing connections with other nodes in the cluster. To prevent this, you need to configure SELinux policies to allow the node to operate.

Generating an SELinux Policy

In order to create an SELinux policy for Galera Cluster, you need to first open ports and set SELinux to permissive mode. Then, after generating various replication events, state transfers and notifications, create a policy from the logs of this activity and reset SELinux from to enforcing mode.

Setting SELinux to Permissive Mode

When SELinux registers a system event, there are three modes that define its response: enforcing, permissive and disabled. While you can set it to permit all activity on the system, this is not a good security practice. Instead, set SELinux to permit activity on the relevant ports and to ignore the database server.

To set SELinux to permissive mode, complete the following steps:

1. Using `semanage`, open the relevant ports:

   ```
   # semanage port -a -t mysqld_port_t -p tcp 4567
   # semanage port -a -t mysqld_port_t -p tcp 4568
   # semanage port -a -t mysqld_port_t -p tcp 4444
   ```

   SELinux already opens the standard MySQL port `3306`. In the event that you use UDP in your cluster, you also need to open `4567` to those connections.

   ```
   # semanage port -a -t mysqld_port_t -p udp 4567
   ```

2. Set SELinux to permissive mode for the database server.

   ```
   # semanage permissive -a mysqld_t
   ```

SELinux now permits the database server to function on the server and no longer blocks the node from network connectivity with the cluster.

Defining the SELinux Policy

While SELinux remains in permissive mode, it continues to log activity from the database server. In order for it to understand normal operation for the database, you need to start the database and generate routine events for SELinux to see.

For servers that use `init`, start the database with the following command:

```
# service mysql start
```

For servers that use `systemd`, instead run this command:

```
# systemctl mysql start
```

You can now begin to create events for SELinux to log. There are many ways to go about this, including:

- Stop the node, then make changes on another node before starting it again. Not being that far behind, the node updates itself using an *Incremental State Transfer*.
- Stop the node, delete the `grastate.dat` file in the data directory, then restart the node. This forces a *State Snapshot Transfer*.
- Restart the node, to trigger the notification command as defined by *wsrep_notify_cmd* (page 247).

When you feel you have generated sufficient events for the log, you can begin work creating the policy and turning SELinux back on.

Note: In order to for your policy to work you must generate both State Snapshot and Incremental State transfers.

Enabling an SELinux Policy

Generating an SELinux policy requires that you search log events for the relevant information and pipe it to the `audit2allow` utility, creating a `galera.te` file to load into SELinux.

To generate and load an SELinux policy for Galera Cluster, complete the following steps:

1. Using `fgrep` and `audit2allow`, create a textease file with the policy information.

   ```
   # fgrep "mysqld" /var/log/audit/audit.log | audit2allow -m MySQL_galera -o galera.
   ↪te
   ```

   This creates a `galera.te` file in your working directory.

2. Compile the audit logs into an SELinux policy module.

   ```
   # checkmodule -M -m galera.te -o galera.mod
   ```

   This creates a `galera.mod` file in your working directory.

3. Package the compiled policy module.

   ```
   # semodule_package -m galera.mod -o galera.pp.
   ```

   This creates a `galera.pp` file in your working directory.

4. Load the package into SELinux.

```
semodule -i galera.pp
```

5. Disable permissive mode for the database server.

```
# semanage permissive -d mysql_t
```

SELinux returns to enforcement mode, now using new policies that work with Galera Cluster.

Part VII

Migration

In *Getting Started* (page 5), the installation guides were written on the assumption that Galera Cluster is the first database in your infrastructure. However, this won't always be the case. Sometimes, you may need to migrate to Galera Cluster from an existing database infrastructure: such as a single MySQL server or several database servers that were part of a MySQL master-slave cluster.

- *Differences from a Standalone MySQL Server* (page 191)

  While clusters operate as distributed database servers, there are some key differences between them and the standard stand-alone MySQL implementations.

- *Migrating to Galera Cluster* (page 195)

  When migrating from the standard MySQL implementations to Galera Cluster, there are some additional steps that you need to take to ensure that you safely transfer the data from the existing database to the new cluster. This section covers the procedure to update system tables and how to migrate from stand-alone MySQL, as well as from MySQL master-slave replication clusters.

Note: See Also: For more information on the installation and basic management of Galera Cluster, see the *Getting Started Guide* (page 5).

CHAPTER

THIRTYNINE

DIFFERENCES FROM A STANDALONE MYSQL SERVER

Although Galera Cluster is built on providing write-set replication to MySQL and related database systems, there are certain key differences between how it handles and the standard standalone MySQL server.

Server Differences

Using a server with Galera Cluster is not the same as one with MySQL. Galera Cluster does not support the same range of operating systems as MySQL, and there are differences in how it handles binary logs and character sets.

Operating System Support

Galera Cluster requires that you use Linux or a similar UNIX-like operating system. Binary packages are not supplied for FreeBSD, Solaris and Mac OS X. There is no support available for Microsoft Windows.

Binary Log Support

Do not use the `binlog-do-db` and `binlog-ignore-db` options.

These binary log options are only supported for DML (Data Manipulation Language) statements. They provide no support for DDL statements. This creates a discrepancy in the binary logs and will cause replication to abort.

Unsupported Character Sets

Do not use the `character_set_server` with UTF-16, UTF-32 or UCS-2.

When you use `rsync` for *State Snapshot Transfer*, the use of these unsupported character sets can cause the server to crash.

Note: This is also a problem when you use automatic donor selection in your cluster, as the cluster may choose to use `rsync` on its own.

Differences in Table Configurations

There are certain features and configurations available in MySQL that do not work as expected in Galera Cluster, such as storage engine support, certain queries and the query cache.

Storage Engine Support

Galera Cluster requires the InnoDB storage engine. Writes made to tables of other types, including the system `mysql-*` tables, do not replicate to the cluster.

That said, DDL statements do replicate at the statement level, meaning that changes made to the `mysql-*` tables do replicate that way.

What this means is that if you were to issue a statement like

```
CREATE USER 'stranger'@'localhost'
  IDENTIFIED BY 'password';
```

or, like

```
GRANT ALL ON strangedb.* TO 'stranger'@'localhost';
```

the changes made to the `mysql-*` tables would replicate to the cluster. However, if you were to issue a statement like

```
INSERT INTO mysql.user (Host, User, Password)
   VALUES ('localhost', 'stranger', 'password');
```

the changes would not replicate.

Note: In general, non-transactional storage engines cannot be supported in multi-master replication.

Tables without Primary Keys

Do not use tables without a primary key.

When tables lack a primary key, rows can appear in different order on different nodes in your cluster. As such, queries like `SELECT...LIMIT...` can return different results. Additionally, on such tables the `DELETE` statement is unsupported.

Note: If you have a table without a primary key, it is always possible to add an `AUTO_INCREMENT` column to the table without breaking your application.

Table Locking

Galera Cluster does not support table locking, as they conflict with multi-master replication. As such, the `LOCK TABLES` and `UNLOCK TABLES` queries are not supported. This also applies to lock functions, such as `GET_LOCK()` and `RELEASE_LOCK()...` for the same reason.

Query Logs

You cannot direct query logs to a table. If you would like to enable query logging in Galera Cluster, you must forward the logs to a file.

```
log_output = FILE
```

Use `general_log` and `general_log_file` to choose query logging and to set the filename for your log file.

Differences in Transactions

There are some differences in how Galera Cluster handles transactions from MySQL, such as XA (eXtended Architecture) transactions and limitations on transaction size.

Distributed Transaction Processing

The standard MySQL server provides support for distributed transaction processing using the Open Group XA standard. This feature is *not* available for Galera Cluster, given that it can lead to possible rollbacks on commit.

Transaction Size

Although Galera Cluster does not explicitly limit the transaction size, the hardware you run it on does impose a size limitation on your transactions. Nodes process write-sets in a single memory-resident buffer. As such, extremely large transactions, such as `LOAD DATA` can adversely effect node performance.

You can avoid situations of this kind using the *wsrep_max_ws_rows* (page 245) and the *wsrep_max_ws_size* (page 245) parameters. Limit the transaction rows to 128 KB and the transaction size to 1 GB.

If necessary, you can increase these limits.

Transaction Commits

Galera Cluster uses at the cluster-level optimistic concurrency control, which can result in transactions that issue a `COMMIT` aborting at that stage.

For example, say that you have two transactions that will write to the same rows, but commit on separate nodes in the cluster and that only one of them can successfully commit. The commit that fails is aborted, while the successful one replicates.

When aborts occur at the cluster level, Galera Cluster gives a deadlock error.

```
code (Error: 1213 SQLSTATE: 40001 (ER_LOCK_DEADLOCK)
```

If you receive this error, restart the failing transaction. It will then issue on its own, without another to put it into conflict.

CHAPTER

FORTY

MIGRATING TO GALERA CLUSTER

For systems that already have instances of the standalone versions of MySQL, MariaDB or Percona XtraDB, the Galera Cluster installation replaces the existing database server with a new one that includes the *wsrep API* patch. This only affects the database server, not the data.

When upgrading from a standalone database server, you must take some additional steps in order to subsequently preserve and use your data with Galera Cluster.

Note: **See Also**: For more information on installing Galera Cluster, see *Installation* (page 11).

Upgrading System Tables

When you finish upgrading a standalone database server to Galera Cluster, but before you initialize your own cluster, you need to update the system tables to take advantage of the new privileges and capabilities. You can do this with `mysql_upgrade`.

In order to use `mysql_upgrade`, you need to first start the database server, but start it without initializing replication. For systems that use `init`, run the following command:

```
# service mysql start --wsrep_on=OFF
```

For servers that use `systemd`, instead use this command:

```
# systemctl start mysql --wsrep_on=OFF
```

The command starts `mysqld` with the *wsrep_on* (page 249) parameter set to `OFF`, which disables replication. With the database server running, you can update the system tables:

```
# mysql_upgrade
```

If this command generates any errors, check the MySQL Reference Manual for more information related to the particular error message. Typically, these errors are not critical and you can usually ignore them, unless they relate to specific functionality that your system requires.

When you finish upgrading the system tables, you need to stop the `mysqld` process until you are ready to initialize the cluster. For servers that use `init`, run the following command:

```
# service mysql stop
```

For servers that use `systemd`, instead use this command:

```
# systemctl stop mysql
```

Running this command stops database server. When you are ready to initialize your cluster, choose this server as your starting node.

Note: **See Also**: For more information on initializing and adding nodes to a cluster, see *Starting the Cluster* (page 33).

Migrating from MySQL to Galera Cluster

In the event that you have an existing database server that uses the MyISAM storage engine or the stock MySQL master-slave replication, there are some additional steps that you need to take. The *Galera Replication Plugin* requires a transactional storage engine in order to function. As MyISAM is non-transactional, you need to migrate your data to InnoDB, in addition to installing the new software packages.

There are three types of database servers referred to in this guide:

- **Master Server** Refers to the MySQL master server.
- **Slave Server** Refers to a MySQL slave server.
- **Cluster Node** Refers to a node in Galera Cluster.

For the sake of simplicity, slave servers and cluster nodes are referenced collectively, rather than individually. In production, you may have several slave servers and must have at least three cluster nodes.

Infrastructure Preparation

For your existing infrastructure, you have a MySQL master server as well as several slave servers that form a master-slave cluster. Before you can begin migration, you first need to prepare your infrastructure for the change.

1. Launch at least three new servers, outside of and unconnected to your existing database infrastructure.
2. On each new server, install Galera Cluster. For information on how to do this, see *Installation* (page 11).
3. Configure the database server. In addition to the IP addresses of each node, on the *wsrep_cluster_address* (page 237) parameter, include the IP addresses of the MySQL master server and each instance of the slave servers.

   For more information on configuring Galera Cluster, see *System Configuration* (page 27) and *Replication Configuration* (page 31).
4. When you finish the installation and configuration, start the cluster. For more information on how to start the cluster, see *Starting the Cluster* (page 33).

To check that it is running properly, log into one of the database clients and run the *wsrep_cluster_size* (page 290) status variable:

```
SHOW STATUS LIKE 'wsrep_cluster_size';

+--------------------+-------+
| Varialbe_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 3     |
+--------------------+-------+
```

Galera Cluster is now running in parallel to your MySQL master-slave cluster. It contains no data and remains unused by your application servers. You can now begin migrating your data.

Data Migration

In order to migrate data from a MySQL master-slave cluster to Galera Cluster, you need to manually transfer it from your existing infrastructure to the new one.

1. Stop the load of the master server.
2. On the master server, run `mysqldump`:

   ```
   $ mysqldump -u root -p --skip-create-options --all-databases > migration.sql
   ```

   The `--skip-create-options` ensures that the database server uses the default storage engine when loading the data, instead of MyISAM.
3. Transfer the `migration.sql` output file to one of your new cluster nodes.

   ```
   $ scp migration.sql user@galera-node-IP
   ```
4. On the cluster node, load the data from the master server.

   ```
   mysql -u root -p < migration.sql
   ```
5. Restart the load from the application servers, this time directing it towards your cluster nodes instead of the master server.

Your application now uses Galera Cluster, instead of your previous MySQL master-slave cluster.

Note: Bear in mind that your application will experience downtime at this stage of the process. The length of the downtime varies depending on the amount of data you have to migrate, specifically how long it takes `mysqldump` to create a snapshot of the master server, then transfer and upload it onto a cluster node.

Database Migration

With your application server now using the new cluster nodes, you now need to migrate your master and slave servers from stock MySQL to Galera Cluster.

1. Using the same process described in *Installation* (page 11), install and configure Galera Cluster on the server.
2. Start the node with replication disabled. For servers that use `init`, run the following command:

   ```
   # service mysql start --wsrep-on=OFF
   ```

   For servers that use `systemd`, instead run this command:

   ```
   # systemctl start mysql --wsrep-on=OFF
   ```
3. From the database client, manually switch the storage engine on each table from MyISAM to InnoDB:

   ```
   ALTER TABLE table_name ENGINE = InnoDB;
   ```
4. Update the system tables:

```
# mysql_upgrade
```

Note: For more information, see *Upgrading System Tables* (page 195).

5. From one of the running Galera Cluster nodes, copy the `grastate.dat` file into the data directory of the former MySQL master server.

```
$ scp grastate.dat user@server-master-ip:/path/to/datadir
```

6. Using your preferred text editor, on the former MySQL master server update the sequence number (that is, the seqno) in the `grastate.dat` file from `-1` to `0`.

7. Restart the master and slave servers. For servers that use `init`, run the following command:

```
# service mysql restart
```

   For servers that use `systemd`, instead run this command:

```
# systemctl restart mysql
```

8. Resume load on these servers.

When the former MySQL master and slave servers come back after restarting, they establish network connectivity with the cluster and begin catching up with recent changes. All of the servers now function as nodes in Galera Cluster.

Part VIII

Support

Troubleshooting

When you experience difficulties with a Galera Cluster deployment, these sections may provide some assistance. They include frequently asked questions, as well as guides to diagnosing and addressing various performance and replication issues.

- *Frequently Asked Questions* (page 203)

  This section provides an FAQ for Galera Cluster. The questions range from the meaning behind the name Galera, to common issues like failover and how to handle crashes during rsync.

- *Server Error Log* (page 207)

  Given that Galera Cluster enables a number of new features not present in the standard MySQL implementation, it may on occasion log server errors that are unfamiliar to you. This section lists common error messages and explains them, as well as what to do about them.

- *Unknown Command Errors* (page 209)

  When a node encounters issues in loading the wsrep Provider, statements run in a console will generate an unknown command error. This section covers identifying the source of the problem and how to solve it.

- *User Changes not Replicating* (page 211)

  Galera Cluster doesn't support direct modification of MySQL system tables since they use the MyISAM engine and are thus non-transactional. For instance, if you add or modify rows in `mysql.user`, these changes will not replicate to the cluster. This section explains the correct methods to use in making changes to systems tables in a cluster.

- *Cluster Stalls on ALTER* (page 213)

  Sometimes, when executing `ALTER` statements that take a long time, you may encounter issues in which other nodes in the cluster stall. This section provides information on how to identify this problem and what to do about it.

- *Detecting a Slow Node* (page 215)

  By design, a cluster's performance is limited by its slowest node. This section shows you how to identify which node is the slowest and to help determine the cause. It also explains how thereby to improve performance.

- *Dealing with Multi-Master Conflicts* (page 217)

  Multi-master conflicts occur in as a result of application servers writing to different nodes. This can lead to two nodes attempting to update the same row with different data. This section provides information on diagnosing and correcting these conflicts.

- *Two-Node Clusters* (page 221)

  Optimally, Galera Cluster requires a minimum of three nodes. If you have a cluster that uses only two nodes, you may sometimes encounter issues in which single-node failures cause the cluster to stop working. This section gives some pointers in how to manage these cases.

Tutorials

Whereas the above sections relate to handling problems with a cluster, these sections provide additional information and guides on improving performance and optimizing configuration.

- *Performance* (page 223)

  This section provides a series of guides to improving performance, such as optimizing write-set caching for state transfers, configuring parallel slave threads and handling large transactions.

- *Configuration Tips* (page 227)

  This section provides a series of guides for optimizing your configuration, including WAN replication, single-master and multi-master deployments and working with SELinux.

CHAPTER

FORTYONE

FREQUENTLY ASKED QUESTIONS

This page lists a number of frequently asked questions on Galera Cluster and other related matters.

What is Galera Cluster?

Galera Cluster is a write-set replication service provider in the form of the *dlopenable* library. It provides synchronous replication and supports multi-master replication. Galera Cluster is capable of unconstrained parallel applying (i.e., "parallel replication"), multicast replication and automatic node provisioning.

The primary focus of Galera Cluster is data consistency. Transactions are either applied to every node or not at all. Galera Cluster is not a cluster manager, a load balancer or a cluster monitor. What it does is keep databases synchronized, provided they were properly configured and synchronized in the beginning.

What is Galera?

The word *galera* is the Italian word for *galley*. The galley is a class of naval vessel used in the Mediterranean Sea from the second millennium B.C.E. until the Renaissance. Although it used sails when the winds were favorable, its principal method of propulsion came from banks of oars.

In order to manage the vessel effectively, rowers had to act synchronously, lest the oars become intertwined and became blocked. Captains could scale the crew up to hundreds of rowers, making the galleys faster and more maneuverable in combat.

Note: **See Also**: For more information on galleys, see Wikipedia.

How are Failovers Managed?

Galera Cluster is a true synchronous multi-master replication system, which allows the use of any or all of the nodes as master at any time without any extra provisioning. What this means is that there is no failover in the traditional MySQL master-slave sense.

The primary focus of Galera Cluster is data consistency across the nodes. This doesn't allow for any modifications to the database that may compromise consistency. For instance, the node rejects write requests until the joining node synchronizes with the cluster and is ready to process requests.

The results of this is that you can safely use your favorite approach to distribute or migrate connections between the nodes without the risk of causing inconsistency.

Note: **See Also**: For more information on connection distribution, see *Cluster Deployment Variants* (page 127).

How do I Upgrade the Cluster?

Periodically, updates will become available for Galera Cluster–for the database server itself or the *Galera Replication Plugin*. To update the software for a node, you would redirect client connections away from it and then stop the node. Then upgrade the node's software. When finished, just restart the node.

Note: **See Also**: For more information on upgrade process, see *Upgrading Galera Cluster* (page 95).

What InnoDB Isolation Levels does Galera Cluster Support?

You can use all isolation levels. Locally, in a given node, transaction isolation works as it does natively with InnoDB.

Globally, with transactions processing in separate nodes, Galera Cluster implements a transaction-level called `SNAPSHOT ISOLATION`. The `SNAPSHOT ISOLATION` level is between the `REPEATABLE READ` and `SERIALIZABLE` levels.

The `SERIALIZABLE` level cannot be guaranteed in the multi-master use case because Galera Cluster replication does not carry a transaction read set. Also, `SERIALIZABLE` transaction is vulnerable to multi-master conflicts. It holds read locks and any replicated write to read locked row will cause the transaction to abort. Hence, it is recommended not to use it in Galera Cluster.

Note: **See Also**: For more information, see *Isolation Levels* (page 55).

How are DDL's Handled by Galera Cluster?

For DDL statements and similar queries, Galera Cluster has two modes of execution:

- *Total Order Isolation*: A query is replicated in a statement before executing on the master. The node waits for all preceding transactions to commit and then all nodes simultaneously execute the transaction in isolation.
- *Rolling Schema Upgrade*: Schema upgrades run locally, blocking only the node on which they are run. The changes do not replicate to the rest of the cluster.

Note: **See Also**: For more information, see *Schema Upgrades* (page 93).

What if Connections return an `Unknown Command` Error?

This may happen when a cluster experiences a temporary split, during which a portion of the nodes loses connectivity to the *Primary Component*. When they reconnect, nodes from the former non-operational component drop their client connections. New connections to the database client will return `Unknown command` errors.

Basically, the node does not yet consider itself a part of the Primary Component. While it has restored network connectivity, it still has to resynchronize itself with the cluster. MySQL doesn't have an error code for the node lacking Primary status. So it defaults to an `Unknown command` message.

Nodes in a non-operational component must regain network connectivity with the Primary Component, process a state transfer, and catch up with the cluster before they can resume normal operation.

Is GCache a Binlog?

The *Write-set Cache*, which is also called GCache, is a memory allocator for write-sets. Its primary purpose is to minimize the write-set footprint in RAM. It is not a log of events, but rather a cache.

- GCache is not persistent.
- Not every entry in GCache is a write-set.
- Not every write-set in GCache will be committed.
- Write-sets in GCache are not allocated in commit order.
- Write-sets are not an optimal entry for the binlog, since they contain extra information.

Nevertheless, it is possible to construct a binlog out of the write-set cache.

What if a Node Crashes during `rsync` SST

You can configure *wsrep_sst_method* (page 256) to use `rsync` for *State Snapshot Transfer*. If the node crashes before the state transfer is complete, it may cause the `rsync` process to hang forever, occupying the port and not allowing you to restart the node. If this occurs, the error logs for the database server will show that the port is in use.

To correct the problem, kill the orphaned `rsync` process. For example, if the process had a pid of `501`, you might enter the following from the command-line:

```
# kill 501
```

Once you kill the orphaned process, it will free the relevant ports and allow you to restart the node.

CHAPTER

FORTYTWO

SERVER ERROR LOG

`Node 0 (XXX) requested state transfer from '*any*'. Selected 1 (XXX) as donor.`

The node is attempting to initiate a *State Snapshot Transfer*.

In the event that you do not explicitly set the donor node through *wsrep_sst_donor* (page 255), the *Group Communication* module will select a donor based on the information available about the node states.

Group Communication monitors node states for the purposes of flow control, state transfers and quorum calculations. It ensures that a node that shows as `JOINING` doesn't count towards flow control and quorum.

A node can serve as a donor when it is in the `SYNCED` state. The joiner node selects a donor from the available synced nodes. It shows preference to synced nodes that have the same *gmcast.segment* (page 279) wsrep Provider option, or it selects the first in the index. When a donor node is chosen, its state changes immediately to `DONOR`. It's no longer available for requests.

If a node can find no free nodes that show as `SYNCED`, the joining node reports the following:

```
Requesting state transfer failed: -11(Resource temporarily
unavailable).  Will keep retrying every 1 second(s).
```

The joining node will continue to retry the state transfer request.

`SQL SYNTAX` Errors

When a *State Snapshot Transfer* fails using `mysqldump` for any reason, the node will write a `SQL SYNTAX` message into the server error logs.

This is a pseudo-statement, though. You can find the actual error message the state transfer returned within the `SQL SYNTAX` entry. It will provide the information you need to correct the problem.

`Commit failed for reason: 3`

When you have *wsrep_debug* (page 240) turned `ON`, you may occasionally see a message noting that a commit has failed due to reason `3`. Below is an example of this:

```
110906 17:45:01 [Note] WSREP: BF kill (1, seqno: 16962377), victim:  (140588996478720␣
↪4) trx: 35525064
110906 17:45:01 [Note] WSREP: Aborting query: commit
110906 17:45:01 [Note] WSREP: kill trx QUERY_COMMITTING for 35525064
110906 17:45:01 [Note] WSREP: commit failed for reason: 3, seqno: -1
```

When attempting to apply a replicated write-set, slave threads occasionally encounter lock conflicts with local transactions, which may already be in the commit phase. In such cases, the node aborts the local transaction, allowing the slave thread to proceed.

This is a consequence of optimistic transaction execution. The database server executes transaction under the expectation that there will be no row conflicts. It is an expected issue in a multi-master configuration.

To mitigate such conflicts, there are a couple of things you can do:

- Use the cluster in a master-slave configuration. Direct all writes to a single node.
- Use the same approach as master-slave read/write splitting.

CHAPTER

FORTYTHREE

UNKNOWN COMMAND ERRORS

Every query returns the `Unknown Command` error.

Situation

You log into a node and try to execute a query from a database client. Every query returns the same error message:

```
SELECT * FROM example_table;

ERROR: Unknown command '\\'
```

The reason for the error is that the node considers itself out of sync with the global state of the cluster. It is unable to handle SQL statements except for `SET` and `SHOW` statements.

This occurs when you have explicitly set the wsrep Provider (i.e., the *wsrep_provider* (page 250)), but the wsrep Provider rejects service. This happens in situations in which the node is unable to connect to the *Primary Component*. It happen when the *wsrep_cluster_address* (page 237) parameter becomes unset. It can also happen due to networking issues.

Solution

Using the *wsrep_on* (page 249) variable dynamically, you can bypass the wsrep Provider check. However, this disables replication.

```
SET wsrep_on=OFF;
```

This SQL statement tells `mysqld` to ignore the *wsrep_provider* (page 250) setting and behave as a standard stand-alone database server. Doing this can lead to data inconsistency with the rest of the cluster, but that may be the desired result for modifying the "local" tables.

If you know or suspect that a cluster doesn't have a *Primary Component*, you need to bootstrap a new one. There are a couple of queries you'll need to run on each node in the cluster.

First, confirm that the node is not part of the Primary Component by checking the *wsrep_cluster_status* (page 291) status variable. Do this by executing the following `SHOW STATUS` statement on each node:

```
SHOW STATUS LIKE 'wsrep_cluster_status';

+----------------------+-------------+
| Variable_name        | Value       |
+----------------------+-------------+
| wsrep_cluster_status | Non_primary |
+----------------------+-------------+
```

If this query returns a value of `Primary`, the node is part of the Primary Component. If it returns any other value, that indicates the node is part of a non-operational component.

Next, find the sequence number of the last committed transaction on each node by getting the value of the *wsrep_last_committed* (page 295) status variable. Do this by executing `SHOW STATUS` statement on each node like this:

```
SHOW STATUS LIKE 'wsrep_last_committed';

+----------------------+--------+
| Variable_name        | Value  |
+----------------------+--------+
| wsrep_last_committed | 409745 |
+----------------------+--------+
```

In the event that none of the nodes are the Primary Component, you will need to bootstrap a new one. The node that returned the largest sequence number is the most advanced in the cluster. On that node, run the following `SET` statement:

```
SET GLOBAL wsrep_provider_options='pc.bootstrap=YES';
```

The node on which you executed this will now operate as the starting point in a new Primary Component. Nodes that are part of non-operational components and have network connectivity will attempt to initiate a state transfer to bring their own databases up-to-date with this node. At this point, the cluster will begin accepting SQL requests.

CHAPTER

FORTYFOUR

USER CHANGES NOT REPLICATING

User changes do not replicate to the cluster.

Situation

You have made some changes to database users, but on inspection find that these changes are only present on the node in which you made them and have not replicated to the cluster.

For instance, say that you want to add a new user to your cluster. You log into a node and use an `INSERT` statement to update the `mysql.user` table.

```
INSERT INTO mysql.user (User,Host, Password)
   VALUES ('user1','localhost', password('my_password'));
```

When finished, you check your work by running a `SELECT` query, to make sure that `user1` does in fact exist on the node:

```
SELECT User, Host, Password FROM mysql.user WHERE User='user1';

+-------+-------------+-------------------------------------------+
| User  | Host        | Password                                  |
+-------+-------------+-------------------------------------------+
| user1 | localhost   | *00A60C0186D8740829671225B7F5694EA5C08EF5 |
+-------+-------------+-------------------------------------------+
```

This checks out fine. However, when you run the same query on a different node, you receive different results:

```
SELECT User, Host, Password FROM mysql.user WHERE User='user1';

Empty set (0.00 sec)
```

The changes you made to the `mysql.user` table on the first node do not replicate to the others. The new user you created can only function when accessing the database on the node where you created it.

Replication currently only works with the InnoDB and XtraDB storage engines. Multi-master replication cannot support non-transactional storage engines, such as MyISAM. Writes made to tables that use non-transactional storage engines do not replicate.

The system tables use MyISAM. This means that any changes you make to the system tables directly, such as in the above example with an `INSERT` statement, remain on the node in which they were issued.

Solution

While direct modifications to the system tables do not replicate, DDL statements replicate at the statement level. Meaning, changes made to the system tables in this manner are made to the entire cluster.

For instance, consider the above example where you added a user to node. If instead of `INSERT` you used `CREATE USER` or `GRANT` you would get very different results:

```
CREATE USER user1 IDENTIFIED BY 'my_password';
```

This creates `user1` in a way that replicates through the cluster. If you run `SELECT` query to check the `mysql.user` table on any node, it returns the same results:

```
SELECT User, Host, Password FROM mysql.user WHERE User='user1';

+-------+-------------+-------------------------------------------+
| User  | Host        | Password                                  |
+-------+-------------+-------------------------------------------+
| user1 | localhost   | *00A60C0186D8740829671225B7F5694EA5C08EF5 |
+-------+-------------+-------------------------------------------+
```

You can now `user1` on any node in the cluster.

CHAPTER

FORTYFIVE

CLUSTER STALLS ON `ALTER`

There may be times in which a cluster will stall when an `ALTER` statement is executed on an unused table.

Situation

There may be a situation in which you attempt to execute an `ALTER` statement on one node, but it takes a long time to execute–longer than expected. During that period all of the other nodes may stall, leading to performance problems throughout the cluster.

What's happening is a side effect of a multi-master cluster with several appliers. The cluster needs to control when a DDL statement ends in relation to other transactions, in order deterministically to detect conflicts and then schedule parallel appliers. Basically, the DDL statement must execute in isolation.

Galera Cluster has a 65K window of tolerance for transactions applied in parallel, but the cluster must wait when `ALTER` statements take too long.

Solution

Given that stalling due to an `ALTER` statement is a consequence of something intrinsic to how replication works in Galera Cluster, there is no direct solution to the problem. However, you can implement a workaround.

In the event that you can sure that no other session will try to modify the table and that there are no other DDL statements running, you can shift the schema upgrade method from *Total Order Isolation* to *Rolling Schema Upgrade* for the duration of the `ALTER` statement. This applies the changes to each node individually, without affecting cluster performance.

To run an `ALTER` statement in this manner, you will need to execute the `ALTER` statement between a paid of `SET` statements on each node like so:

```
SET wsrep_OSU_method='RSU';

ALTER TABLE table1 ADD COLUMN col8 INT;

SET wsrep_OSU_method='TOI';
```

The first SQL statement here will change the Schema Upgrade method to *Rolling Schema Upgrade* (i.e., `RSU`). The second SQL statement represents an `ALTER` statement you want to execute. Once that's finished, the last `SET` statement will reset the Schema Upgrade method back to *Total Order Isolation* (i.e., `TOI`). After you've done this on each node, the cluster will now run with the desired updates.

CHAPTER

FORTYSIX

DETECTING A SLOW NODE

By design, the performance of a cluster cannot be higher than the performance of the slowest node in the cluster. Even if you have only one node, its performance can be considerably slower when compared with running the same server in a standalone mode (i.e., without a wsrep Provider).

This is particularly true for large transactions—even if they are within transaction size limits. This is why it's important to be able to detect a slow node on a cluster.

Finding Slow Nodes

There are two status variables used in finding slow nodes in a cluster: `wsrep_flow_control_sent` and `wsrep_local_recv_queue_avg`. Check these status variables on each node in a cluster. The node that returns the highest value is the slowest one. Lower values are preferable.

wsrep_flow_control_sent (page 295): This variable provides the number of times a node sent a pause event due to flow control since the last status query.

```
SHOW STATUS LIKE 'wsrep_flow_control_sent';

+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| wsrep_flow_control_sent | 7     |
+--------------------------+-------+
```

wsrep_local_recv_queue_avg (page 297): This varaible returns an average of the received queue length since the last status query.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue_avg';

+----------------------------+---------+
| Variable_name              | Value   |
+----------------------------+---------+
| wsrep_local_recv_queue_avg | 3.34852 |
+----------------------------+---------+
```

A node that return values much higher than `0.0` indicates it cannot apply write-sets as fast as they're received and can generate replication throttling.

CHAPTER

FORTYSEVEN

DEALING WITH MULTI-MASTER CONFLICTS

These types of conflicts relate to multi-master database environments and typically involve inconsistencies of row amongst nodes. To understand this better, consider a situation in a multi-master replication system in which users can submit updates to any database node. There may be an instance in which two nodes attempt to change the same row in a database, but with different values. Galera Cluster copes with situations such as this by using certification-based replication.

Note: See Also: For more information, see *Certification-Based Replication* (page 49).

Diagnosing Multi-Master Conflicts

There are a few techniques available to log and monitor problems that may indicate multi-master conflicts. They can be enabled with the *wsrep_debug* (page 240) option. This instructs the node to include additional debugging information in the server output log. You can enable it through the configuration file with a line like so:

```
# Enable Debugging Output to Server Error Log
wsrep_debug=ON
```

Once you turn debugging on, you can use monitoring software to watch for row conflicts. Below is an example of a log entry that indicates a conflict as described above:

```
110906 17:45:01 [Note] WSREP: BF kill (1, seqno: 16962377), victim:  (140588996478720␣
↪4) trx: 35525064
110906 17:45:01 [Note] WSREP: Aborting query: commit
110906 17:45:01 [Note] WSREP: kill trx QUERY_COMMITTING for 35525064
110906 17:45:01 [Note] WSREP: commit failed for reason: 3, seqno: -1
```

Note: Warning: In addition to useful debugging information, this parameter also causes the database server to print authentication information, (that is, passwords), to the error logs. Do not enable it in production environments.

If you develop your own notification system, you can use status variables to watch for conflicts. Below is an example of how you might manually retrieve this information. You would simply incorporate something similar into your scripts or customized program.

```
SHOW STATUS LIKE 'wsrep_local_bf_aborts';

+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
```

```
| wsrep_local_bf_aborts | 333   |
+-----------------------+-------+

SHOW STATUS LIKE 'wsrep_local_cert_failures';

+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| wsrep_local_cert_failures | 333   |
+---------------------------+-------+
```

wsrep_local_bf_aborts (page 296) returns the total number of local transactions aborted by slave transactions while in execution. *wsrep_local_cert_failures* (page 296) provides the total number of transactions that have failed certification tests.

You can enable conflict logging features with *wsrep_log_conflicts* (page 244) and *cert.log_conflicts* (page 268). Just add the following lines to the configuration file (i.e., `my.cnf`):

```
# Enable Conflict Logging
wsrep_log_conflicts=ON
wsrep_provider_options="cert.log_conflicts=YES"
```

These parameters enable different forms of conflict logging on the database server. When turned on, the node logs additional information about the conflicts it encounters. For instance, it will log the name of the table and schema where the conflict occurred and the actual values for the keys that produced the conflict. Below is an example of such a log entry:

```
7:51:13 [Note] WSREP: trx conflict for key (1,FLAT8)056eac38 0989cb96:
source: cdeae866-d4a8-11e3-bd84-479ea1a1e941 version: 3 local: 1 state:
MUST_ABORT flags: 1 conn_id: 160285 trx_id: 29755710 seqnos (l: 643424,
g: 8749173, s: 8749171, d: 8749171, ts: 12637975935482109) <--X-->
source: 5af493da-d4ab-11e3-bfe0-16ba14bdca37 version: 3 local: 0 state:
APPLYING flags: 1 conn_id: 157852 trx_id: 26224969 seqnos (l: 643423,
g: 8749172, s: 8749171, d: 8749170, ts: 12637839897662340)
```

Auto-Committing Transactions

When two transactions are conflicting, the later of the two is rolled back by the cluster. The client application registers this rollback as a deadlock error. Ideally, the client application should retry the deadlocked transaction. However, not all client applications have this logic built in.

If you encounter this problem, you can set the node to attempt to auto-commit the deadlocked transactions on behalf of the client application. You would do this with the *wsrep_retry_autocommit* (page 252) parameter. Just enter the following to the configuration file:

```
wsrep_retry_autocommit=4
```

When a transaction fails the certification test due to a cluster-wide conflict, this parameter tells the node how many times you want it to retry the transaction before returning a deadlock error. In the example line above, it's set to four times.

Note: Retrying only applies to auto-commit transactions, as retrying is not safe for multi-statement transactions.

Multi-Master Conflict Work-Around

While Galera Cluster resolves multi-master conflicts automatically, there are steps you can take to minimize the frequency of their occurrence.

- First, analyze the hot-spot and see if you can change the application logic to catch deadlock exceptions.
- Next, enable retrying logic at the node level using the *wsrep_retry_autocommit* (page 252) parameter.
- Last, limit the number of master nodes or switch to a master-slave model.

If you can filter out access to the hot-spot table, it may be enough to treat writes only to the hot-spot table as master-slave.

CHAPTER

FORTYEIGHT

TWO-NODE CLUSTERS

Although it may seem simple to maintain a cluster of only two nodes, there is an inherent potential problem. In a two-node cluster, when one node fails, it will cause the other to stop.

Situation

Suppose you have a cluster composed of only two nodes. Suppose further that one of the nodes leaves the cluster, ungracefully. For instance, instead of being shut down through `init` or `systemd`, it crashes or loses network connectivity. The node that remains becomes non-operational. It will remain so until some additional information is provided by a third entity, such as another node or a person.

In a two-node cluster, if one node loses its network connection and other node is still on the network, both will think itself as being the *Primary Component*. Each will be unaware that the other is still running. This could cause problems once network connectivity is restored. To prevent this, the nodes become non-operational.

Solutions

There are two solutions available. You can bootstrap the surviving node (i.e., the one with network connectivity) to form a new *Primary Component*. You would do this by using the *pc.boostrap* (page 280) wsrep Provider option. To do so, log into the database client and run the following SQL statement:

```
SET GLOBAL wsrep_provider_options='pc.bootstrap=YES';
```

This will bootstrap the surviving node as a new Primary Component. When the other node comes back online or regains network connectivity with this node, it will recognize that it's behind and initiate a state transfer to catch up.

If you want the node to continue to operate, you can use the *pc.ignore_sb* (page 281) wsrep Provider option. To do so, log into the database client and run the following SQL statement:

```
SET GLOBAL wsrep_provider_options='pc.ignore_sb=TRUE';
```

The node will resume processing updates, even if it suspects a split-brain situation.

Note: Warning: Enabling *pc.ignore_sb* (page 281) is dangerous in a multi-master setup due to the aforementioned risk for split-brain situations. However, it does simplify things in master-slave clusters–especially in situations with only two nodes.

In addition to the solutions provided here, you can avoid this situation entirely by using Galera Arbitrator. Galera Arbitrator functions as an odd node in quorum calculations. If you enable Galera Arbitrator on one node in a two-node cluster, that node will remain the Primary Component, even if the other node fails or loses network connectivity.

CHAPTER

FORTYNINE

PERFORMANCE

Write-set Caching during State Transfers

Under normal operations, nodes do not consume much more memory than the regular standalone MySQL database server. The certification index and uncommitted write-sets do cause some additional usage, but in typical applications this is not usually noticeable.

Write-set caching during state transfers is the exception.

When a node receives a state transfer, it cannot process or apply incoming write-sets as it does not yet have a state to apply them to. Depending on the state transfer method, (`mysqldump`, for instance), the sending node may also be unable to apply write-sets.

The Write-set Cache, (or GCache), caches write-sets on memory-mapped files to disk and Galera Cluster allocates these files as needed. In other words, the only limit for the cache is the available disk space. Writing to disk in turn reduces memory consumption.

Note: **See Also**: For more information on configuring write-set caching to improve performance, see *Configuring Flow Control* (page 108).

Customizing the Write-set Cache Size

You can define the size of the write-set cache using the *gcache.size* (page 275) parameter. The set the size to one less than that of the data directory.

If you have storage issues, there are some guidelines to consider in adjusting this issue. For example, your preferred state snapshot method. `rsync` and `xtrabackup` copy the InnoDB log files, while `mysqldump` does not. So, if you use `mysqldump` for state snapshot transfers, you can subtract the size of the log files from your calculation of the data directory size.

Note: Incremental State Transfers (IST) copies the database five times faster over `mysqldump` and about 50% faster than `xtrabackup`. Meaning that your cluster can handle relatively large write-set caches. However, bear in mind that you cannot provision a server with Incremental State Transfers.

As a general rule, start with the data directory size, including any possible links, then subtract the size of the ring buffer storage file, which is called `galera.cache` by default.

In the event that storage remains an issue, you can further refine these calculations with the database write rate. The write rate indicates the tail length that the cluster stores in the write-set cache.

You can calculate this using the *wsrep_received_bytes* (page 302) status variable.

1. Determine the size of the write-sets the node has received from the cluster:

```
SHOW STATUS LIKE 'wsrep_received_bytes';

+------------------------+-----------+
| Variable name          | Value     |
+------------------------+-----------+
| wsrep_received_bytes   | 6637093   |
+------------------------+-----------+
```

   Note the value and time, respective as $recv_1$ and $time_1$.

2. Run the same query again, noting the value and time, respectively, as $recv_2$ and $time_2$.

3. Apply these values to the following equation:

$$writerate = \frac{recv_2 - recv_1}{time_2 - time_1}$$

From the write rate you can determine the amount of time the cache remains valid. When the cluster shows a node as absent for a period of time less than this interval, the node can rejoin the cluster through an incremental state transfer. Node that remains absent for longer than this interval will likely require a full state snapshot transfer to rejoin the cluster.

You can determine the period of time the cache remains valid using this equation:

$$period = \frac{cachesize}{writerate}$$

Conversely, if you already know the period in which you want the write-set cache to remain valid, you can use instead this equation:

$$cachesize = writerate \times time$$

This equation can show how the size of the write-set cache can improve performance. For instance, say you find that cluster nodes frequently request state snapshot transfers. Increasing the *gcache.size* (page 275) parameter extends the period in which the write-set remains valid, allowing the nodes to update instead through incremental state transfers.

Note: Consider these configuration tips as guidelines only. For example, in cases where you must avoid state snapshot transfers as much as possible, you may end up using a much larger write-set cache than suggested above.

Setting Parallel Slave Threads

There is no rule about how many slave threads you need for replication. Parallel threads do not guarantee better performance. But, parallel applying does not impair regular operation performance and may speed up the synchronization of new nodes with the cluster.

You should start with four slave threads per CPU core:

```
wsrep_slave_threads=4
```

The logic here is that, in a balanced system, four slave threads can typically saturate a CPU core. However, I/O performance can increase this figure several times over. For example, a single-core ThinkPad R51 with a 4200 RPM drive can use thirty-two slave threads.

Parallel applying requires the following settings:

```
innodb_autoinc_lockmode=2
innodb_locks_unsafe_For_binlog=1
```

You can use the *wsrep_cert_deps_distance* (page 289) status variable to determine the maximum number of slave threads possible. For example:

```
SHOW STATUS LIKE 'wsrep_cert_deps_distance';

+----------------------------+-----------+
| Variable name              | Value     |
+----------------------------+-----------+
| wsrep_cert_deps_distance   | 23.88889  |
+----------------------------+-----------+
```

This value essentially determines the number of write-sets that the node can apply in parallel on average.

Note: Warning: Do not use a value for *wsrep_slave_threads* (page 253) that is higher than the average given by the *wsrep_cert_deps_distance* (page 289) status variable.

Dealing with Large Transactions

Large transactions, for instance the transaction caused by a `DELETE` query that removes millions of rows from a table at once, can lead to diminished performance. If you find that you must perform frequently transactions of this scale, consider using `pt-archiver` from the Percona Toolkit.

For example, if you want to delete expired tokens from their table on a database called `keystone` at `dbhost`, you might run something like this:

```
$ pt-archiver --source h=dbhost,D=keystone,t=token \
   --purge --where "expires < NOW()" --primary-key-only \
   --sleep-coef 1.0 --txn-size 500
```

This allows you to delete rows efficiently from the cluster.

Note: See Also: For more information on `pt-archiver`, its syntax and what else it can do, see the manpage.

CHAPTER

FIFTY

CONFIGURATION TIPS

This page contains some advanced tips on configuring a replication, networking, and servers related to Galera Cluster.

WAN Replication

When running the cluster over a WAN, you may frequently experience transient network connectivity failures. To prevent this from partitioning the cluster, you may want to increase the *keepalive* timeouts.

The following parameters can tolerate 30 second connectivity outages:

```
wsrep_provider_options = "evs.keepalive_period = PT3S;
                          evs.suspect_timeout = PT30S;
                          evs.inactive_timeout = PT1M;
                          evs.install_timeout = PT1M"
```

In configuring these parameters, consider the following:

- You want the *evs.suspect_timeout* (page 273) parameter set as high as possible to avoid partitions. Partitions cause state transfers, which can effect performance.
- You must set the *evs.inactive_timeout* (page 271) parameter to a value higher than that of the *evs.suspect_timeout* (page 273) parameter.
- You must set the *evs.install_timeout* (page 271) parameter to a value higher than the value of the *evs.inactive_timeout* (page 271) parameter.

WAN Latency

When using Galera Cluster over a WAN, bear in mind that WAN links can have exceptionally high latency. You can check this by taking Round-Trip Time (RTT) measurements between cluster nodes. If there is a latency, you correct for this by adjusting all of the temporal parameters.

To take RTT measurements, use `ping` on each cluster node to ping the others. For example, if you were to log into the node at `192.168.1.1`, you might execute something like the following from the command-line:

```
$ ping -c 3 192.168.1.2

  PING 192.168.1.2 (192.168.1.2) 58(84) bytes of data.
  64 bytes from 192.168.1.2: icmp_seq=1 ttl=64 time=0.736 ms
  64 bytes from 192.168.1.2: icmp_seq=2 ttl=64 time=0.878 ms
  64 bytes from 192.168.1.2: icmp_seq=3 ttl=64 time=12.7 ms

  --- 192.168.1.2 ---
```

```
3 packets transmitted, 3 received, 0% packet loss, time 2002ms
rtt min/avg/max/mdev = 0.736/4.788/12.752/5.631 ms
```

Repeat this on each node in the cluster and note the highest value among them.

Parameters that relate to periods and timeouts, such as *evs.join_retrans_period* (page 272), must all use values that exceed the highest RTT measurement in the cluster.

```
wsrep_provider_options="evs.join_retrans_period=PT0.5S"
```

This allows the cluster to compensate for the latency issues of the WAN links between the cluster nodes.

Multi-Master Setup

A master is a node that can simultaneously process writes from clients. The more masters in a cluster, the higher the probability of certification conflicts. This can lead to undesirable rollbacks and performance degradation.

If you find you experience frequent certification conflicts, consider reducing the number of nodes the cluster uses as masters.

Single Master Setup

In the event that a cluster uses only one node as a master, there are certain requirements (e.g., the slave queue size) that can be relaxed.

To relax flow control, you might use the settings below:

```
wsrep_provider_options = "gcs.fc_limit = 256;
                          gcs.fc_factor = 0.99;
                          gcs.fc_master_slave = YES"
```

By reducing the rate of flow control events, these settings may improve replication performance.

Note: You can also use this setting as sub-optimal in a multi-master setup.

Using Galera Cluster with SELinux

When you first enable Galera Cluster on a node that runs SELinux, it will prohibit all cluster activities. In order to enable replication on the node, you need a policy so that SELinux can recognize cluster activities as legitimate.

To create a policy for Galera Cluster, set SELinux to run in permissive mode. Permissive mode does not block cluster activity, but it does log the actions as warnings. By collecting these warnings, you can iteratively create a policy for Galera Cluster. You can make this change generally by editing the SELinux configuration file (e.g., `/etc/selinux/config`) to include an uncommented line like so:

```
SELINUX=permissive
```

Once SELinux no longer registers warnings from Galera Cluster, you can switch it back into enforcing mode. SELinux then uses the new policy to allow the cluster access to the various ports and files it needs.

Note: Almost all Linux distributions ship with a MySQL policy for SELinux. You can use this policy as a starting point for Galera Cluster and extend it, using the above procedure.

Using Synchronization Functions

Occasionally, an application may need to perform a critical read. Critical reads are queries that require that the local database reaches the most up-to-date state possible before the query is executed.

In Galera Cluster prior to 4.x, you could manage critical reads using the *wsrep_sync_wait* (page 259) session variable. This would cause the node to enable causality checks, holding new queries until the database server catches up with all updates that were made prior to the start of the current transaction. While this method does ensure that the node reaches the most up-to-date state before executing the query, it also means that the node may wait to receive updates that might have nothing to do with the query at hand.

Beginning with Galera Cluster 4.0, though, you can use synchronization functions. This allows you to tie the synchronization process to specific transactions so that the node waits only until a specific transaction is applied before executing the query. Here is an example of how this might work:

Suppose on `node1`, you begin a transaction, make changes to a table and then commit the transaction like so:

```
START TRANSACTION;

UPDATE table1 SET col4 = col4 * 1.2;

COMMIT;
```

After that, using the *WSREP_LAST_WRITTEN_GTID()* (page 263) function, say you obtain the *Global Transaction ID* of the transaction and save it to the `$transaction_1_gtid` variable like this:

```
$transaction_1_gtid = SELECT WSREP_LAST_WRITTEN_GTID();
```

Now, on `node2`, suppose you set it to wait until it replicates and applies the transaction from `node1` before starting a new transaction:

```
SELECT WSREP_SYNC_WAIT_UPTO_GTID($transaction_1_gtid);

START TRANSACTION;
```

Next, you execute your critical reads.

Using the *WSREP_SYNC_WAIT_UPTO_GTID()* (page 264) function, the node waits until it has replicated and applied the given Global Transaction ID before starting a new transaction.

Note: Synchronization Functions were introduced in Galera Cluster 4. If you have an older version, you won't be able to use these features. To determine which version is installed on a server, use the `SHOW STATUS` statement and look for the *wsrep_provider_version* (page 302) status variable.

```
SHOW STATUS LIKE 'wsrep_provider_version';

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

```
| Variable_name          | Value                |
+------------------------+----------------------+
| wsrep_provider_version | 25.3.5-wheezy(rXXXX) |
+------------------------+----------------------+
```

The digits after the second and third decimal places are the version. The results here indicate that Galera Cluster version 3.5 is installed on the server.

Part IX

Reference

In the event that you need more information about particular variables or parameters or status variable or would like a clearer explanation about various terms used in the documentation, these chapters provide general reference material to Galera Cluster configuration and use.

Variable Reference

Defining persistent configurations in Galera Cluster is done through the underlying database server, using the `[mysqld]` unit in the `my.cnf` configuration file. These chapters provide reference guides to the base replication status and configuration variables as well as the specific wsrep Provider options implemented through the Galera Replication Plugin.

- *MySQL wsrep Options* (page 235)

  In addition to the standard configuration variables available through the database server, Galera Cluster implements several that are unique and specific to fine-tuning database replication. This chapter provides a reference guide to these new configuration variables in Galera Cluster

- *MySQL wsrep Functions* (page 263)

  There are a few Galera specific functions. This page lists and explains them, as well as gives examples of their use.

- *Galera Parameters* (page 265)

  The Galera Replication Plugin, which acts as the wsrep Provider, includes a number of parameters specific to its operation. This chapter provides a reference guide to the various wsrep Provider options available.

- *Galera Status Variables* (page 287)

  In addition to the standard status variables available through the database server, Galera Cluster also implements several that you can use in determining the status and health of the node and the cluster. This chapter provides a reference guide to these new status variables in Galera Cluster.

Utility Reference

In some cases your configuration or implementation may require that you work with external utilities in your deployment of Galera Cluster. These chapters provide reference guides for two such utilities: XtraBackup and Galera Load Balancer.

- *XtraBackup Parameters* (page 309)

  When you manage State Snapshot Transfers using Percona XtraBackup, it allows you to set various parameters on the state transfer script the node uses from the `my.cnf` configuration file. This chapter provides a reference guide to options available to XtraBackup.

- *Galera Load Balancer Parameters* (page 317)

  In high availability situations or similar cases where nodes are subject to high traffic situations, you may find it beneficial to set up a load balancer between your application servers and the cluster. This chapter provides a reference guide to the Codership implementation: the Galera Load Balancer.

- *Galera System Tables* (page 305)

  This page provides information on the Galera specific system tables. These were added as of version 4 of Galera.

Miscellaneous References

- *Versioning Information* (page 327)

  While the documentation follows a convention of 3.x in speaking of release versions for Galera Cluster, in practice the numbering system is somewhat more complicated: covering releases of the underlying database server, the wsrep Provider and the wsrep API. This chapter provides a more thorough explanation of versioning with Galera Cluster.

- *Legal Notice* (page 329)

  This page provides information on the documentation copyright.

- *Glossary* (page 331)

  In the event that you would like clarification on particular topics, this chapter provides reference information on core concepts in Galera Cluster.

- genindex

  In the event that you would like to check these concepts against related terms to find more information in the docs, this chapter provides a general index of the site.

CHAPTER

FIFTYONE

MYSQL WSREP OPTIONS

These are MySQL system variables introduced by wsrep API patch version 0.8. Almost all of the variables are global except for a few. Those are session variables. If you click on a particular variable in this table, your web browser will scroll down to the entry for it with more details and an explanation.

Option	Default	Global	Support	Dynamic
wsrep_auto_increment_control (page 236)	ON	Yes	1+	
wsrep_causal_reads (page 237)	OFF		1 - 3.6	
wsrep_certify_nonPK (page 237)	ON	Yes	1+	
wsrep_cluster_address (page 237)		Yes	1+	
wsrep_cluster_name (page 238)	example_cluster	Yes	1+	
wsrep_convert_LOCK_to_trx (page 239)	OFF	Yes	1+	
wsrep_data_home_dir (page 240)	/path/to/data_home	Yes	1+	
wsrep_dbug_option (page 240)		Yes	1+	
wsrep_debug (page 240)	OFF	Yes	1+	
wsrep_desync (page 241)	OFF	Yes	1+	
wsrep_dirty_reads (page 242)	OFF	Yes		Yes
wsrep_drupal_282555_workaround (page 242)	ON	Yes	1+	
wsrep_forced_binlog_format (page 243)	NONE	Yes	1+	
wsrep_load_data_splitting (page 244)	ON	Yes	1+	
wsrep_log_conflicts (page 244)	OFF	Yes	1+	
wsrep_max_ws_rows (page 245)	0	Yes	1+	
wsrep_max_ws_size (page 245)	1G	Yes	1+	
wsrep_node_address (page 246)	*host address:default port*	Yes	1+	
wsrep_node_incoming_address (page 246)	*host address:mysqld port*	Yes	1+	
wsrep_node_name (page 247)	<hostname>	Yes	1+	
wsrep_notify_cmd (page 247)		Yes	1+	
wsrep_on (page 249)	ON		1+	
wsrep_OSU_method (page 249)	TOI		3+	
wsrep_preordered (page 250)	OFF	Yes	1+	
wsrep_provider (page 250)	NONE	Yes	1+	
wsrep_provider_options (page 251)		Yes	1+	
wsrep_reject_queries (page 251)	NONE	Yes		Yes
wsrep_restart_slave (page 252)	OFF	Yes	1+	Yes
wsrep_retry_autocommit (page 252)	1	Yes	1+	
wsrep_slave_FK_checks (page 253)	ON	Yes	1+	Yes
wsrep_slave_threads (page 253)	1	Yes	1+	
wsrep_slave_UK_checks (page 254)	OFF	Yes	1+	Yes
wsrep_sst_auth (page 254)		Yes	1+	
wsrep_sst_donor (page 255)		Yes	1+	

Continued on next page

Table 51.1 – continued from previous page

Option	Default	Global	Support	Dynamic
wsrep_sst_donor_rejects_queries (page 256)	OFF	Yes	1+	
wsrep_sst_method (page 256)	mysqldump	Yes	1+	
wsrep_sst_receive_address (page 258)	*node IP address*	Yes	1+	
wsrep_start_position (page 258)	*see reference entry*	Yes	1+	
wsrep_sync_wait (page 259)	0	Yes	3.6+	Yes
wsrep_trx_fragment_size (page 260)	0	Yes	4+	Yes
wsrep_trx_fragment_unit (page 260)	bytes	Yes	4+	Yes
wsrep_ws_persistency (page 261)		Yes	1	

You can execute the SHOW VARIABLES statement with the LIKE operator as shown below to get list of all Galera related variables on your server:

```
SHOW VARIABLES LIKE 'wsrep%';
```

The results will vary depending on which version of Galera is running on your server. All of the parameters and variables possible are listed above, but they're listed below with explanations of each.

wsrep_auto_increment_control

This parameter enables the automatic adjustment of auto increment system variables with changes in cluster membership.

Command-line Format	--wsrep-auto-increment-control	
System Variable	*Name:*	wsrep_auto_increment_control
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	ON
Support	*Introduced:*	1

The node manages auto-increment values in a table using two variables: auto_increment_increment and auto_increment_offset. The first relates to the value auto-increment rows count from the offset. The second relates to the offset it should use in moving to the next position.

The *wsrep_auto_increment_control* (page 236) parameter enables additional calculations to this process, using the number of nodes connected to the *Primary Component* to adjust the increment and offset. This is done to reduce the likelihood that two nodes will attempt to write the same auto-increment value to a table.

It significantly reduces the rate of certification conflicts for INSERT statements. You can execute the following SHOW VARIABLES statement to see how its set:

```
SHOW VARIABLES LIKE 'wsrep_auto_increment_control';

 +------------------------------+-------+
 | Variable_name                | Value |
 +------------------------------+-------+
 | wsrep_auto_increment_control | ON    |
 +------------------------------+-------+
```

wsrep_causal_reads

This parameter enables the enforcement of strict cluster-wide `READ COMMITTED` semantics on non-transactional reads. It results in larger read latencies.

Command-line Format	`--wsrep-causal-reads`	
System Variable	*Name:*	`wsrep_causal_reads`
	Variable Scope:	Session
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`
Support	*Introduced:*	1
	Deprecated:	3.6

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_causal_reads';
```

Note: Warning: This feature has been **deprecated**. It has been replaced by *wsrep_sync_wait* (page 259).

wsrep_certify_nonPK

This parameter is used to define whether the node should generate primary keys on rows without them for the purposes of certification.

Command-line Format	`--wsrep-certify-nonpk`	
System Variable	*Name:*	`wsrep_certify_nonpk`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	`ON`
Support	*Introduced:*	1

Galera Cluster requires primary keys on all tables. The node uses the primary key in replication to allow for the parallel applying of transactions to a table. This parameter tells the node that when it encounters a row without a primary key, it should create one for replication purposes. However, as a rule don't use tables without primary keys.

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_certify_nonpk';

+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| wsrep_certify_nonpk | ON    |
+---------------------+-------+
```

wsrep_cluster_address

This parameter sets the back-end schema, IP addresses, ports and options the node uses in connecting to the cluster.

Command-line Format	--wsrep-cluster-address	
System Variable	*Name:*	wsrep_cluster_address
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	String
	Default Value:	
Support	*Introduced:*	1

Galera Cluster uses this parameter to determine the IP addresses for the other nodes in the cluster, the back-end schema to use and additional options it should use in connecting to and communicating with those nodes. Currently, the only back-end schema supported for production is `gcomm`.

Below is the syntax for this the values of this parameter:

```
<backend schema>://<cluster address>[?option1=value1[&option2=value2]]
```

Here's an example of how that might look:

```
wsrep_cluster_address="gcomm://192.168.0.1:4567?gmcast.listen_addr=0.0.0.0:5678"
```

Changing this variable while Galera is running will cause the node to close the connection to the current cluster, and reconnect to the new address. Doing this at runtime may not be possible, though, for all SST methods. As of Galera Cluster 23.2.2, it is possible to provide a comma-separated list of other nodes in the cluster as follows:

```
gcomm://node1:port1,node2:port2,...[?option1=value1&...]
```

Using the string `gcomm://` without any address will cause the node to startup alone, thus initializing a new cluster–that the other nodes can join to. Using `--wsrep-new-cluster` is the newer, preferred way.

Note: Warning: Never use an empty `gcomm://` string in the `my.cnf` configuration file. If a node restarts, that will cause the node not to rejoin the cluster of which it was a member. Instead, it will initialize a new one-node cluster and cause a split brain. To bootstrap a cluster, you should only pass the `--wsrep-new-cluster` string at the command-line–instead of using `--wsrep-cluster-address="gcomm://"`. For more information, see *Starting the Cluster* (page 33).

You can execute the following SQL statement to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_cluster_address';

+-----------------------+-------------------------------------------+
| Variable_name         | Value                                     |
+-----------------------+-------------------------------------------+
| wsrep_cluster_address | gcomm://192.168.1.1,192.168.1.2,192.168.1.3 |
+-----------------------+-------------------------------------------+
```

wsrep_cluster_name

This parameter defines the logical cluster name for the node.

Command-line Format	--wsrep-cluster-name	
System Variable	*Name:*	wsrep_cluster_name
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	String
	Default Value:	exmaple_cluster
Support	*Introduced:*	1

This parameter allows you to define the logical name the node uses for the cluster. When a node attempts to connect to a cluster, it checks the value of this parameter against that of the cluster. The connection is only made if the names match. If they don't match, the connection fails. Because of this, the cluster name must be the same on all nodes.

You can execute the following SHOW VARIABLES statement with a LIKE operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_cluster_name';

+--------------------+-----------------+
| Variable_name      | Value           |
+--------------------+-----------------+
| wsrep_cluster_name | example_cluster |
+--------------------+-----------------+
```

wsrep_convert_lock_to_trx

This parameter is used to set whether the node converts LOCK/UNLOCK TABLES statements into BEGIN/COMMIT statements.

Command-line Format	--wsrep-convert-lock-to-trx	
System Variable	*Name:*	wsrep_convert_lock_to_trx
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	OFF
Support	*Introduced:*	1

This parameter determines how the node handles LOCK/UNLOCK TABLES statements, specifically whether or not you want it to convert these statements into BEGIN/COMMIT statements. It tells the node to convert implicitly locking sessions into transactions within the database server. By itself, this is not the same as support for locking sections, but it does prevent the database from resulting in a logically inconsistent state.

This parameter may help sometimes to get old applications working in a multi-master setup.

Note: Loading a large database dump with LOCK statements can result in abnormally large transactions and cause an out-of-memory condition.

You can execute the following SHOW VARIABLES statement with a LIKE operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_convert_lock_to_trx';

+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| wsrep_convert_lock_to_trx | OFF   |
+---------------------------+-------+
```

wsrep_data_home_dir

Use this parameter to set the directory the wsrep Provider uses for its files.

System Variable	*Name:*	`wsrep_data_home_dir`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Directory
	Default Value:	/path/to/mysql_datahome
Support	*Introduced:*	1

During operation, the wsrep Provider needs to save various files to disk that record its internal state. This parameter defines the path to the directory that you want it to use. If not set, it defaults the MySQL `datadir` path.

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_data_home_dir';

+----------------------+-----------------+
| Variable_name        | Value           |
+----------------------+-----------------+
| wsrep_data_home_dir | /var/lib/mysql |
+----------------------+-----------------+
```

wsrep_dbug_option

You can set debug options to pass to the wsrep Provider with this parameter.

Command-line Format	`--wsrep-dbug-option`	
System Variable	*Name:*	`wsrep_dbug_option`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	String
	Default Value:	
Support	*Introduced:*	1

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see how this variable is set, if its set:

```
SHOW VARIABLES LIKE 'wsrep_dbug_option';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_dbug_option |       |
+--------------------+-------+
```

wsrep_debug

This parameter enables additional debugging output for the database server error log.

Command-line Format	--wsrep-debug	
System Variable	*Name:*	wsrep_debug
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	OFF
Support	*Introduced:*	1

Under normal operation, error events are logged to an error log file for the database server. By default, the name of this file is the server hostname with the `.err` extension. You can define a custom path using the log_error parameter. When you enable *wsrep_debug* (page 240), the database server logs additional events surrounding these errors to help in identifying and correcting problems.

Note: Warning: In addition to useful debugging information, this parameter also causes the database server to print authentication information (i.e., passwords) to the error logs. Don't enable it in production environments.

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see if this variable is enabled:

```
SHOW VARIABLES LIKE 'wsrep_debug';

+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| wsrep_debug   | OFF   |
+---------------+-------+
```

wsrep_desync

This parameter is used to set whether or not the node participates in Flow Control.

System Variable	*Name:*	wsrep_desync
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	OFF
Support	*Introduced:*	1

When a node receives more write-sets than it can apply, the transactions are placed in a received queue. In the event that the node falls too far behind, it engages Flow Control. The node takes itself out of sync with the cluster and works through the received queue until it reaches a more manageable size.

Note: See Also: For more information on Flow Control and how to configure and manage it in a cluster, see *Flow Control* (page 61) and *Managing Flow Control* (page 107).

When set to `ON`, this parameter disables Flow Control for the node. The node will continue to receive write-sets and fall further behind the cluster. The cluster doesn't wait for desynced nodes to catch up, even if it reaches the `fc_limit` value.

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see if this variable is enabled:

```
SHOW VARIABLES LIKE 'wsrep_desync';

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

```
| Variable_name | Value |
+---------------+-------+
| wsrep_desync  | OFF   |
+---------------+-------+
```

wsrep_dirty_reads

This parameter defines whether the node accepts read queries when in a non-operational state.

Command-line Format	`--wsrep-dirty-reads`	
System Variable	*Name:*	`wsrep_dirty_reads`
	Variable Scope:	Global, Session
	Dynamic Variable:	Yes
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`
Support	*Introduced:*	

When a node loses its connection to the *Primary Component*, it enters a non-operational state. Given that it can't keep its data current while in this state, it rejects all queries with an `ERROR: Unknown command` message. This parameter determines whether or not the node permits reads while in a non-operational state.

Note: Remember that by its nature, data reads from nodes in a non-operational state are stale. Current data in the Primary Component remains inaccessible to these nodes until they rejoin the cluster.

When enabling this parameter, the node only permits reads. It still rejects any command that modifies or updates the database. When in this state, the node allows `USE`, `SELECT`, `LOCK TABLE` and `UNLOCK TABLES` statements. It doesn't allow DDL statements. It also rejects DML statements (i.e., `INSERT`, `DELETE` and `UPDATE`).

You must set the *wsrep_sync_wait* (page 259) parameter to `0` when using this parameter, else it raises a deadlock error.

You can execute the following `SHOW VARIABLES` statement with a `LIKE` operator to see if this variable is enabled:

```
SHOW VARIABLES LIKE 'wsrep_dirty_reads';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| wsrep_dirty_reads | ON    |
+-------------------+-------+
```

Note: This is a MySQL wsrep parameter. It was introduced in version 5.6.29.

---

wsrep_drupal_282555_workaround

This parameter enables workaround for a bug in MySQL InnoDB that affects Drupal installations.

Command-line Format	--wsrep-drupal-282555-workaround	
System Variable	*Name:*	wsrep_drupal_282555_workaround
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	ON
Support	*Introduced:*	1

Drupal installations using MySQL are subject to a bug in InnoDB, tracked as MySQL Bug 41984 and Drupal Issue 282555. Specifically, inserting a *DEFAULT* value into an *AUTO_INCREMENT* column may return duplicate key errors.

This parameter enables a workaround for the bug on Galera Cluster.

You can execute the following SHOW VARIABLES statement with a LIKE operator to see if this variable is enabled:

```
SHOW VARIABLES LIKE 'wsrep_drupal_28255_workaround';

+-------------------------------+-------+
| Variable_name                 | Value |
+-------------------------------+-------+
| wsrep_drupal_28255_workaround | ON    |
+-------------------------------+-------+
```

wsrep_forced_binlog_format

This parameter defines the binary log format for all transactions.

Command-line Format	--wsrep-forced-binlog-format	
System Variable	*Name:*	wsrep_forced_binlog_format
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	enumeration
	Default Value:	NONE
	Valid Values:	ROW
		STATEMENT
		MIXED
		NONE
Support	*Introduced:*	1

The node uses the format given by this parameter regardless of the client session variable binlog_format. Valid choices for this parameter are: ROW, STATEMENT, and MIXED. Additionally, there is the special value NONE, which means that there is no forced format in effect for the binary logs. When set to a value other than NONE, this parameter forces all transactions to use a given binary log format.

This variable was introduced to support STATEMENT format replication during *Rolling Schema Upgrade*. In most cases, however, ROW format replication is valid for asymmetric schema replication.

You can execute the following SHOW VARIABLES statement with a LIKE operator to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_forced_binlog_format';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| wsrep_forced_binlog_format | NONE  |
+----------------------------+-------+
```

wsrep_load_data_splitting

This parameter defines whether the node splits large `LOAD DATA` commands into more manageable units.

Command-line Format	`--wsrep-load-data-splitting`	
System Variable	*Name:*	`wsrep_load_data_splitting`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	`ON`
Support	*Introduced:*	1

When loading huge amounts of data creates problems for Galera Cluster, in that they eventually reach a size that is too large for the node to rollback completely the operation in the event of a conflict and whatever gets committed stays committed.

This parameter tells the node to split `LOAD DATA` commands into transactions of 10,000 rows or less, making the data more manageable for the cluster. This deviates from the standard behavior for MySQL.

You can execute the following `SHOW VARIABLES` statement to see how this variable is set:

```
SHOW VARIABLES LIKE 'wsrep_load_data_splitting';

+---------------------------+-------+
| Variable_name             | Value |
+---------------------------+-------+
| wsrep_load_data_splitting | ON    |
+---------------------------+-------+
```

wsrep_log_conflicts

This parameter defines whether the node logs additional information about conflicts.

Command-line Format	`--wsrep-log-conflicts`	
System Variable	*Name:*	`wsrep_log_conflicts`
	Variable Scope:	Global
	Dynamic Variable:	No
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`
Support	*Introduced:*	1

In Galera Cluster, the database server uses the standard logging features of MySQL, MariaDB and Percona XtraDB. This parameter enables additional information for the logs pertaining to conflicts. You may find this useful in troubleshooting replication problems. You can also log conflict information with the wsrep Provider option *cert.log_conflicts* (page 268).

The additional information includes the table and schema where the conflict occurred, as well as the actual values for the keys that produced the conflict.

You can execute the following `SHOW VARIABLES` statement to see if this feature is enabled:

```
SHOW VARIABLES LIKE 'wsrep_log_conflicts';

+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
```

```
| wsrep_log_conflicts | OFF   |
+---------------------+-------+
```

wsrep_max_ws_rows

With this parameter you can set the maximum number of rows the node allows in a write-set.

Command-line Format	`--wsrep-max-ws-rows`	
System Variable	*Name:*	`wsrep_max_ws_rows`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	`0`
Support	*Introduced:*	1

If set to a value greater than `0`, this parameter sets the maximum number of rows that the node allows in a write-set.

You can execute the following `SHOW VARIABLES` statement to see the current value of this parameter:

```
SHOW VARIABLES LIKE 'wsrep_max_ws_rows';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| wsrep_max_ws_rows | 128   |
+-------------------+-------+
```

wsrep_max_ws_size

You can set the maximum size the node allows for write-sets with this parameter.

Command-line Format	`--wsrep-max-ws-size`	
System Variable	*Name:*	`wsrep_max_ws_size`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	`1G`
Support	*Introduced:*	1

This parameter sets the maximum size that the node allows for a write-set. Currently, this value limits the supported size of transactions and of `LOAD DATA` statements.

The maximum allowed write-set size is `2G`. You can execute the following `SHOW VARIABLES` statement to see the current value of this parameter:

```
SHOW VARIABLES LIKE 'wsrep_max_ws_size';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| wsrep_max_ws_size | 1G    |
+-------------------+-------+
```

wsrep_node_address

This parameter is used to note the IP address and port of the node.

Command-line Format	--wsrep-node-address	
System Variable	*Name:*	wsrep_node_address
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	server IP address, port 4567
Support	*Introduced:*	1

The node passes its IP address and port number to the *Galera Replication Plugin*, where it's used as the base address in cluster communications. By default, the node pulls the address of the first network interface and uses the default port for Galera Cluster. Typically, this is the address of `eth0` or `enp2s0` on port `4567`.

While the default behavior is often sufficient, there are situations in which this auto-guessing function produces unreliable results. Some common reasons are the following:

- Servers with multiple network interfaces;
- Servers that run multiple nodes;
- Network Address Translation (NAT);
- Clusters with nodes in more than one region;
- Container deployments, such as with Docker and jails; and
- Cloud deployments, such as with Amazon EC2 and OpenStack.

In these scenarios, since auto-guess of the IP address does not produce the correct result, you will need to provide an explicit value for this parameter.

Note: See Also: In addition to defining the node address and port, this parameter also provides the default values for the *wsrep_sst_receive_address* (page 258) parameter and the *ist.recv_addr* (page 279) option.

In some cases, you may need to provide a different value. For example, Galera Cluster running on Amazon EC2 requires that you use the global DNS name instead of the local IP address.

You can execute the `SHOW VARIABLES` statement as shown below to get the current value of this parameter:

```
SHOW VARIABLES LIKE 'wsrep_node_address';

+--------------------+-------------+
| Variable_name      | Value       |
+--------------------+-------------+
| wsrep_node_address | 192.168.1.1 |
+--------------------+-------------+
```

wsrep_node_incoming_address

This parameter is used to provide the IP address and port from which the node should expect client connections.

Command-line Format	--wsrep-node-incoming-address	
System Variable	*Name:*	wsrep_node_incoming_address
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	String
	Default Value:	
Support	*Introduced:*	1

This parameter defines the IP address and port number at which the node should expect to receive client connections. It's intended for integration with load balancers. For now, it's otherwise unused by the node.

You can execute the `SHOW VARIABLES` statement with the `LIKE` operator as shown below to get the IP address and port setting of this parameter:

```
SHOW VARIABLES LIKE 'wsrep_node_incoming_address';

+----------------------------+------------------+
| Variable_name              | Value            |
+----------------------------+------------------+
| wsrep_node_incoming_address | 192.168.1.1:3306 |
+----------------------------+------------------+
```

wsrep_node_name

You can set the logical name that the node uses for itself with this parameter.

Command-line Format	--wsrep-node-name	
System Variable	*Name:*	wsrep_node_name
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	server hostname
Support	*Introduced:*	1

This parameter defines the logical name that the node uses when referring to itself in logs and in the cluster. It's for convenience, to help you in identifying nodes in the cluster by means other than the node address.

By default, the node uses the server hostname. In some situations, you may need explicitly to set it. You would do this when using container deployments with Docker or FreeBSD jails, where the node uses the name of the container rather than the hostname.

You can execute the `SHOW VARIABLES` statement with the `LIKE` operator as shown below to get the node name:

```
SHOW VARIABLES LIKE 'wsrep_node_name';

+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| wsrep_node_name | GaleraNode1 |
+-----------------+-------------+
```

wsrep_notify_cmd

Defines the command the node runs whenever cluster membership or the state of the node changes.

Command-line Format	--wsrep-notify-cmd	
System Variable	*Name:*	wsrep_notify_cmd
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	
Support	*Introduced:*	1

Whenever the node registers changes in cluster membership or its own state, this parameter allows you to send information about that change to an external script defined by the value. You can use this to reconfigure load balancers, raise alerts and so on, in response to node and cluster activity.

Note: See Also: For an example script that updates two tables on the local node, with changes taking place at the cluster level, see the *Notification Command* (page 161).

When the node calls the command, it passes one or more arguments that you can use in configuring your custom notification script and how it responds to the change. The options are:

--status <status str> The status of this node. The possible statuses are:

- `Undefined` The node has just started up and is not connected to any *Primary Component*.
- `Joiner` The node is connected to a primary component and now is receiving state snapshot.
- `Donor` The node is connected to primary component and now is sending state snapshot.
- `Joined` The node has a complete state and now is catching up with the cluster.
- `Synced` The node has synchronized itself with the cluster.
- `Error(<error code if available>)` The node is in an error state.

--uuid <state UUID> The cluster state UUID.

--primary <yes/no> Whether the current cluster component is primary or not.

--members <list> A comma-separated list of the component member UUIDs. The members are presented in the following syntax:

- `<node UUID>` A unique node ID. The wsrep Provider automatically assigns this ID for each node.
- `<node name>` The node name as it is set in the `wsrep_node_name` option.
- `<incoming address>` The address for client connections as it is set in the `wsrep_node_incoming_address` option.

--index The index of this node in the node list.

```
SHOW VARIABLES LIKE 'wsrep_notify_cmd';

+------------------+--------------------------+
| Variable_name    | Value                    |
+------------------+--------------------------+
```

```
| wsrep_notify_cmd | /usr/bin/wsrep_notify.sh |
+------------------+--------------------------+
```

wsrep_on

Defines whether replication takes place for updates from the current session.

System Variable	*Name:*	wsrep_on
	Variable Scope:	Session
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	ON
Support	*Introduced:*	1

This parameter defines whether or not updates made in the current session replicate to the cluster. It does not cause the node to leave the cluster and the node continues to communicate with other nodes. Additionally, it is a session variable. Defining it through the SET GLOBAL syntax also affects future sessions.

```
SHOW VARIABLES LIKE 'wsrep_on';

+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| wsrep_on      | ON    |
+---------------+-------+
```

wsrep_OSU_method

Defines the Online Schema Upgrade method the node uses to replicate DDL statements.

Command-line Format	--wsrep-OSU-method	
System Variable	*Name:*	wsrep_OSU_method
	Variable Scope:	Global, Session
	Dynamic Variable:	Yes
Permitted Values	*Type:*	enumeration
	Default Value:	TOI
	Valid Values:	TOI
		ROI
Support	*Introduced:*	Patch v. 3 (5.5.17-22.3)

DDL statements are non-transactional and as such do not replicate through write-sets. There are three methods available that determine how the node handles replicating these statements:

- TOI In the *Total Order Isolation* method, the cluster runs the DDL statement on all nodes in the same total order sequence, blocking other transactions from committing while the DDL is in progress.
- RSU In the *Rolling Schema Upgrade* method, the node runs the DDL statements locally, thus blocking only the one node where the statement was made. While processing the DDL statement, the node is not replicating and may be unable to process replication events due to a table lock. Once the DDL operation is complete, the node catches up and syncs with the cluster to become fully operational again. The DDL statement or its effects are not replicated; the user is responsible for manually executing this statement on each node in the cluster.

Note: **See Also**: For more information on DDL statements and OSU methods, see *Schema Upgrades* (page 93).

```
SHOW VARIABLES LIKE 'wsrep_OSU_method';

+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| wsrep_OSU_method | TOI   |
+------------------+-------+
```

wsrep_preordered

Defines whether the node uses transparent handling of preordered replication events.

Command-line Format	`--wsrep-preordered`	
System Variable	*Name:*	`wsrep_preordered`
	Variable Scope:	Global
	Dynamic Variable:	Yes
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`
Support	*Introduced:*	1

This parameter enables transparent handling of preordered replication events, such as replication events arriving from traditional asynchronous replication. When this option is `ON`, such events will be applied locally first before being replicated to the other nodes of the cluster. This could increase the rate at which they can be processed which would be otherwise limited by the latency between the nodes in the cluster.

Preordered events should not interfere with events that originate on the local node. Therefore, you should not run local update queries on a table that is also being updated through asynchronous replication.

```
SHOW VARIABLES LIKE 'wsrep_preordered';

+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| wsrep_preordered | OFF   |
+------------------+-------+
```

wsrep_provider

Defines the path to the *Galera Replication Plugin*.

Command-line Format	`--wsrep-provider`	
System Variable	*Name:*	`wsrep_provider`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	File
	Default Value:	
Support	*Introduced:*	1

When the node starts, it needs to load the wsrep Provider in order to enable replication functions. The path defined in this parameter tells it what file it needs to load and where to find it. In the event that you do not define this path or you give it an invalid value, the node bypasses all calls to the wsrep Provider and behaves as a standard standalone instance of MySQL.

```
SHOW VARIABLES LIKE 'wsrep_provider';

+----------------+-----------------------------------+
| Variable_name  | Value                             |
+----------------+-----------------------------------+
| wsrep_provider | /usr/lib/galera/libgalera_smm.so |
+----------------+-----------------------------------+
```

wsrep_provider_options

Defines optional settings the node passes to the wsrep Provider.

Command-line Format	`--wsrep-provider-options`	
System Variable	*Name:*	`wsrep_provider_options`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	String
	Default Value:	
Support	*Introduced:*	1

When the node loads the wsrep Provider, there are several configuration options available that affect how it handles certain events. These allow you to fine tune how it handles various situations.

For example, you can use *gcache.size* (page 275) to define how large a write-set cache the node keeps or manage group communications timeouts.

Note: See Also: For more information on the wsrep Provider options, see *Galera Parameters* (page 265).

```
SHOW VARIABLES LIKE 'wsrep_provider_options';

+------------------------+-------------------------------------------------+
| Variable_name          | Value                                           |
+------------------------+-------------------------------------------------+
| wsrep_provider_options | ... evs.user_send_window=2,gcache.size=128Mb    |
|                        | evs.auto_evict=0,debug=OFF, evs.version=0 ... |
+------------------------+-------------------------------------------------+
```

wsrep_reject_queries:

Defines whether the node rejects client queries while participating in the cluster.

System Variable	*Name:*	`wsrep_reject_queries`
	Variable Scope:	Global
	Dynamic Variable:	Yes
Permitted Values	*Type:*	array
	Default Value:	`NONE`
	Valid Values:	`NONE`
		`ALL`
		`ALL_KILL`
Support	*Introduced:*	

When in use, this parameter causes the node to reject queries from client connections. The node continues to participate in the cluster and apply write-sets, but client queries generate `Unknown command` errors. For instance,

```
SELECT * FROM my_table;

Error 1047: Unknown command
```

You may find this parameter useful in certain maintenance situations. In enabling it, you can also decide whether or not the node maintains or kills any current client connections.

- `NONE` The node disables this feature.
- `ALL` The node enables this feature. It rejects all queries, but maintains any existing client connections.
- `ALL_KILL` The node enables this feature. It rejects all queries and kills existing client connections without waiting, including the current connection.

```
SHOW VARIABLES LIKE 'wsrep_reject_queries';

+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| wsrep_reject_queries | NONE  |
+----------------------+-------+
```

Note: This is a MySQL wsrep parameter. It was introduced in version 5.6.29.

wsrep_restart_slave

Defines whether the replication slave restarts when the node joins the cluster.

Command-line Format	`--wsrep-restart-slave`	
System Variable	*Name:*	`wsrep_restart_slave`
	Variable Scope:	Global
	Dynamic Variable:	Yes
Permitted Values	*Type:*	boolean
	Default Value:	`OFF`
Support	*Introduced:*	

Enabling this parameter tells the node to restart the replication slave when it joins the cluster.

```
SHOW VARIABLES LIKE 'wsrep_restart_slave';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_restart_slave | OFF   |
+--------------------+-------+
```

wsrep_retry_autocommit

Defines the number of retries the node attempts when an autocommit query fails.

Command-line Format	--wsrep-retry-autocommit	
System Variable	*Name:*	wsrep_retry_autocommit
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	integer
	Default Value:	1
Support	*Introduced:*	1

When an autocommit query fails the certification test due to a cluster-wide conflict, the node can retry it without returning an error to the client. This parameter defines how many times the node retries the query. It is analogous to rescheduling an autocommit query should it go into deadlock with other transactions in the database lock manager.

```
SHOW VARIABLES LIKE 'wsrep_retry_autocommit';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_retry_autocommit | 1     |
+------------------------+-------+
```

wsrep_slave_FK_checks

Defines whether the node performs foreign key checking for applier threads.

Command-line Format	--wsrep-slave-FK-checks	
System Variable	*Name:*	wsrep_slave_FK_checks
	Variable Scope:	Global
	Dynamic Variable:	Yes
Permitted Values	*Type:*	boolean
	Default Value:	ON
Support	*Introduced:*	

This parameter enables foreign key checking on applier threads.

```
SHOW VARIABLES LIKE 'wsrep_slave_FK_checks';

+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| wsrep_slave_FK_checks | ON    |
+-----------------------+-------+
```

wsrep_slave_threads

Defines the number of threads to use in applying slave write-sets.

Command-line Format	--wsrep-slave-threads	
System Variable	*Name:*	wsrep_slave_threads
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	integer
	Default Value:	1
Support	*Introduced:*	1

This parameter allows you to define how many threads the node uses when applying slave write-sets. Performance on the underlying system and hardware, the size of the database, the number of client connections, and the load your application puts on the server all factor in the need for threading, but not in a way that makes the scale of that need easy to predict. Because of this, there is no strict formula to determine how many slave threads your node actually needs.

Instead of concrete recommendations, there are some general guidelines that you can use as a starting point in finding the value that works best for your system:

- It is rarely beneficial to use a value that is less than twice the number of CPU cores on your system.
- Similarly, it is rarely beneficial to use a value that is more than one quarter the total number of client connections to the node. While it is difficult to predict the number of client connections, being off by as much as 50% over or under is unlikely to make a difference.
- From the perspective of resource utilization, it's recommended that you keep to the lower end of slave threads.

```
SHOW VARIABLES LIKE 'wsrep_slave_threads';

+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| wsrep_slave_threads | 1     |
+---------------------+-------+
```

wsrep_slave_UK_checks

Defines whether the node performs unique key checking on applier threads.

Command-line Format	`--wsrep-slave-UK-checks`	
System Variable	*Name:*	`wsrep_slave_UK_checks`
	Variable Scope:	Global
	Dynamic Variable:	Yes
Permitted Values	*Type:*	boolean
	Default Value:	`OFF`
Support	*Introduced:*	

This parameter enables unique key checking on applier threads.

```
SHOW VARIABLES LIKE 'wsrep_slave_UK_checks';

+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| wsrep_slave_UK_checks | OFF   |
+-----------------------+-------+
```

wsrep_sst_auth

Defines the authentication information to use in *State Snapshot Transfer*.

Command-line Format	--wsrep-sst-auth	
System Variable	*Name:*	wsrep_sst_auth
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	
	Valid Values:	username:password
Support	*Introduced:*	1

When the node attempts a state snapshot transfer using the *Logical State Transfer Method*, the transfer script uses a client connection to the database server in order to obtain the data it needs to send. This parameter provides the authentication information, (that is, the username and password), that the script uses to access the database servers of both sending and receiving nodes.

Note: Galera Cluster only uses this parameter for State Snapshot Transfers that use the Logical transfer method. Currently, the only method to use the Logical transfer method is `mysqldump`. For all other methods, the node doesn't need this parameter.

Format this value to the pattern: `username:password`.

```
SHOW VARIABLES LIKE 'wsrep_sst_auth'

+----------------+--------------------------+
| Variable_name  | Value                    |
+----------------+--------------------------+
| wsrep_sst_auth | wsrep_sst_user:mypassword |
+----------------+--------------------------+
```

wsrep_sst_donor

Defines the name of the node that this node uses as a donor in state transfers.

Command-line Format	--wsrep-sst-donor	
System Variable	*Name:*	wsrep_sst_donor
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	
Support	*Introduced:*	1

When the node requires a state transfer from the cluster, it looks for the most appropriate one available. The group communications module monitors the node state for the purposes of Flow Control, state transfers and quorum calculations. The node can be a donor if it is in the `SYNCED` state. The first node in the `SYNCED` state in the index becomes the donor and is made unavailable for requests while serving as such.

If there are no free `SYNCED` nodes at the moment, the joining node reports in the logs:

```
Requesting state transfer failed: -11(Resource temporarily unavailable).
 Will keep retrying every 1 second(s)
```

It continues retrying the state transfer request until it succeeds. When the state transfer request does succeed, the node makes the following entry in the logs:

```
Node 0 (XXX) requested state transfer from '*any*'. Selected 1 (XXX) as donor.
```

Using this parameter, you can tell the node which cluster node it should use instead for state transfers. The name given to the receiving node with this parameter must match the name given for *wsrep_node_name* (page 247) on the donor node.

```
SHOW VARIABLES LIKE 'wsrep_sst_donor';

+-----------------+---------------+
| Variable_name   | Value         |
+-----------------+---------------+
| wsrep_sst_donor | my_donor_node |
+-----------------+---------------+
```

wsrep_sst_donor_rejects_queries

Defines whether the node rejects blocking client sessions on a node when it is serving as a donor in a blocking state transfer method, such as `mysqldump` and `rsync`.

Command-line Format	`--wsrep-sst-donor-rejects-queries`	
System Variable	*Name:*	`wsrep_sst_donor_rejects_queries`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`
Support	*Introduced:*	1

This parameter determines whether the node rejects blocking client sessions while it is sending state transfers using methods that block it as the donor. In these situations, all queries return the error `ER_UNKNOWN_COM_ERROR`, that is they respond with `Unknown command`, just like the joining node does.

Given that a *State Snapshot Transfer* is scriptable, there is no way to tell whether the requested method is blocking or not. You may also want to avoid querying the donor even with non-blocking state transfers. As a result, when this parameter is enabled the donor node rejects queries regardless the state transfer and even if the initial request concerned a blocking-only transfer, (meaning, it also rejects during `xtrabackup`).

Note: Warning: The `mysqldump` state transfer method does not work with this setting, given that `mysqldump` runs queries on the donor and there is no way to differentiate its session from the regular client session.

```
SHOW VARIABLES LIKE 'wsrep_sst_donor_rejects_queries';

+---------------------------------+-------+
| Variable_name                   | Value |
+---------------------------------+-------+
| wsrep_sst_donor_rejects_queries | OFF   |
+---------------------------------+-------+
```

wsrep_sst_method

Defines the method or script the node uses in a *State Snapshot Transfer*.

Command-line Format	--wsrep-sst-method	
System Variable	*Name:*	wsrep_sst_method
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	mysqldump
Support	*Introduced:*	1

When the node makes a state transfer request it calls on an external shell script to establish a connection a with the donor node and transfer the database state onto the local database server. This parameter allows you to define what script the node uses in requesting state transfers.

Galera Cluster ships with a number of default scripts that the node can use in state snapshot transfers. The supported methods are:

- `mysqldump` This is slow, except for small data-sets, but is the most tested option.
- `rsync` This option is much faster than `mysqldump` on large data-sets.

  Note: You can only use `rsync` when anode is starting. You cannot use it with a running InnoDB storage engine.

- `rsync_wan` This option is almost the same as `rsync`, but uses the `delta-xfer` algorithm to minimize network traffic.
- `xtrabackup` This option is a fast and practically non-blocking state transfer method based on the Percona `xtrabackup` tool. If you want to use it, the following settings must be present in the `my.cnf` configuration file on all nodes:

```
[mysqld]
wsrep_sst_auth=YOUR_SST_USER:YOUR_SST_PASSWORD
wsrep_sst_method=xtrabackup
datadri=/path/to/datadir

[client]
socket=/path/to/socket
```

In addition to the default scripts provided and supported by Galera Cluster, you can also define your own custom state transfer script. The naming convention that the node expects is for the value of this parameter to match `wsrep_%.sh`. For instance, giving the node a transfer method of `MyCustomSST` causes it to look for `wsrep_MyCustomSST.sh` in `/usr/bin`.

Bear in mind, the cluster uses the same script to send and receive state transfers. If you want to use a custom state transfer script, you need to place it on every node in the cluster.

Note: See Also: For more information on scripting state snapshot transfers, see *Scriptable State Snapshot Transfers* (page 87).

```
SHOW VARIABLES LIKE 'wsrep_sst_method';

+------------------+-----------+
| Variable_name    | Value     |
+------------------+-----------+
| wsrep_sst_method | mysqldump |
+------------------+-----------+
```

wsrep_sst_receive_address

Defines the address from which the node expects to receive state transfers.

Command-line Format	--wsrep-sst-receive-address	
System Variable	*Name:*	wsrep_sst_receive_address
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	*wsrep_node_address* (page 246)
Support	*Introduced:*	1

This parameter defines the address from which the node expects to receive state transfers. It is dependent on the *State Snapshot Transfer* method the node uses.

For example, `mysqldump` uses the address and port on which the node listens, which by default is set to the value of *wsrep_node_address* (page 246).

Note: Check that your firewall allows connections to this address from other cluster nodes.

```
SHOW VARIABLES LIKE 'wsrep_sst_receive_address';

+---------------------------+-------------+
| Variable_name             | Value       |
+---------------------------+-------------+
| wsrep_sst_receive_address | 192.168.1.1 |
+---------------------------+-------------+
```

wsrep_start_position

Defines the node start position.

Command-line Format	--wsrep-start-position	
System Variable	*Name:*	wsrep_start_position
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	00000000-0000-0000-0000-000000000000:-1
Support	*Introduced:*	1

This parameter defines the node start position. It exists for the sole purpose of notifying the joining node of the completion of a state transfer.

Note: **See Also**: For more information on scripting state snapshot transfers, see *Scriptable State Snapshot Transfers* (page 87).

```
SHOW VARIABLES LIKE 'wsrep_start_position';

+----------------------+----------------------------------------+
| Variable_name        | Value                                  |
+----------------------+----------------------------------------+
| wsrep_start_position | 00000000-0000-0000-0000-000000000000:-1 |
+----------------------+----------------------------------------+
```

wsrep_sync_wait

Defines whether the node enforces strict cluster-wide causality checks.

Command-line Format	--wsrep-sync-wait	
System Variable	*Name:*	wsrep_sync_wait
	Variable Scope:	Session
	Dynamic Variable:	Yes
Permitted Values	*Type:*	bitmask
	Default Value:	0
Support	*Introduced:*	3.6

When you enable this parameter, the node triggers causality checks in response to certain types of queries. During the check, the node blocks new queries while the database server catches up with all updates made in the cluster to the point where the check was begun. Once it reaches this point, the node executes the original query.

Note: Causality checks of any type can result in increased latency.

This value of this parameter is a bitmask, which determines the type of check you want the node to run.

Bitmask	Checks
0	Disabled.
1	Checks on READ statements, including SELECT, SHOW, and BEGIN / START TRANSACTION.
2	Checks made on UPDATE and DELETE statements.
3	Checks made on READ, UPDATE and DELETE statements.
4	Checks made on INSERT and REPLACE statements.

For example, say that you have a web application. At one point in its run, you need it to perform a critical read. That is, you want the application to access the database server and run a SELECT query that must return the most up to date information possible.

```
SET SESSION wsrep_sync_wait=1;
SELECT * FROM example WHERE field = "value";
SET SESSION wsrep_sync_wait=0
```

In the example, the application first runs a SET command to enable *wsrep_sync_wait* (page 259) for READ statements, then it makes a SELECT query. Rather than running the query, the node initiates a causality check, blocking incoming queries while it catches up with the cluster. When the node finishes applying the new transaction, it executes the SELECT query and returns the results to the application. The application, having finished the critical read, disables *wsrep_sync_wait* (page 259), returning the node to normal operation.

Note: Setting *wsrep_sync_wait* (page 259) to 1 is the same as *wsrep_causal_reads* (page 237) to ON. This deprecates *wsrep_causal_reads* (page 237).

```
SHOW VARIABLES LIKE 'wsrep_sync_wait';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| wsrep_sync_wait | 0     |
+-----------------+-------+
```

wsrep_trx_fragment_size

Defines the number of replication units needed to generate a new fragment in Streaming Replication.

Command-line Format	--wsrep-trx-fragment-size	
System Variable	*Name:*	wsrep_trx_fragment_size
	Variable Scope:	Session
	Dynamic Variable:	Yes
Permitted Values	*Type:*	integer
	Default Value:	0
Support	*Introduced:*	4.0

In *Streaming Replication*, the node breaks transactions down into fragments, then replicates and certifies them while the transaction is in progress. Once certified, a fragment can no longer be aborted due to conflicting transactions. This parameter determines the number of replication units to include in a fragment. To define what these units represent, use *wsrep_trx_fragment_unit* (page 260). A value of 0 indicates that streaming replication will not be used.

```
SHOW VARIABLE LIKE 'wsrep_trx_fragment_size';

+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| wsrep_trx_fragment_size | 5     |
+-------------------------+-------+
```

wsrep_trx_fragment_unit

Defines the replication unit type to use in Streaming Replication.

Command-line Format	--wsrep-trx-fragment-unit	
System Variable	*Name:*	wsrep_trx_fragment_unit
	Variable Scope:	Session
	Dynamic Variable:	Yes
Permitted Values	*Type:*	string
	Default Value:	bytes
	Valid Values:	bytes
		events
		rows
		statements
Support	*Introduced:*	4.0

In *Streaming Replication*, the node breaks transactions down into fragments, then replicates and certifies them while the transaction is in progress. Once certified, a fragment can no longer be aborted due to conflicting transactions. This parameter determines the unit to use in determining the size of the fragment. To define the number of replication units to use in the fragment, use *wsrep_trx_fragment_size* (page 260).

Supported replication units are:

- **bytes**: Refers to the fragment size in bytes.
- **events**: Refers to the number of binary log events in the fragment.
- **rows**: Refers to the number of rows updated in the fragment.
- **statements**: Refers to the number of SQL statements in the fragment.

```
SHOW VARIABLE LIKE 'wsrep_trx_fragment_unit';

+--------------------------+--------+
| Variable_name            | Value  |
+--------------------------+--------+
| wsrep_trx_fragment_unit | bytes  |
+--------------------------+--------+
```

wsrep_ws_persistency

Defines whether the node stores write-sets locally for debugging.

Command-line Format	`--wsrep-ws-persistency`	
System Variable	*Name:*	`wsrep_ws_persistency`
	Variable Scope:	Global
	Dynamic Variable:	
Permitted Values	*Type:*	string
	Default Value:	
Support	*Introduced:*	
	Deprecated:	0.8

This parameter defines whether the node stores write-sets locally for debugging purposes.

```
SHOW VARIABLES LIKE 'wsrep_ws_persistency';

+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| wsrep_ws_persistency | ON    |
+----------------------+-------+
```

CHAPTER

FIFTYTWO

MYSQL WSREP FUNCTIONS

Function	Arguments	Support
WSREP_LAST_SEEN_GTID() (page 263)		4+
WSREP_LAST_WRITTEN_GTID() (page 263)		4+
WSREP_SYNC_WAIT_UPTO_GTID() (page 264)	`gtid [timeout]`	4+

WSREP_LAST_SEEN_GTID()

Much like `LAST_INSERT_ID()` for getting the identification number of the last row inserted in MySQL, this function returns the *Global Transaction ID* of the last write transaction observed by the client.

Function	`WSREP_LAST_SEEN_GTID()`
Arguments	None
Support	4+

This function returns the *Global Transaction ID* of the last write transaction observed by the client. It can be useful in combination with *WSREP_SYNC_WAIT_UPTO_GTID()* (page 264). You can use this parameter to identify the transaction upon which it should wait before unblocking the client.

Below is an example of how you might use the `WSREP_LAST_SEEN_GTID()` function to get the Global Transaction ID of the last write transaction observed:

```
SELECT WSREP_LAST_SEEN_GTID();
```

WSREP_LAST_WRITTEN_GTID()

This function returns the *Global Transaction ID* of the last write transaction made by the client.

Function	`WSREP_LAST_WRITTEN_GTID()`
Arguments	None
Support	4+

This function returns the Global Transaction ID of the last write transaction made by the client. This can be useful in combination with *WSREP_SYNC_WAIT_UPTO_GTID()* (page 264). You can use this parameter to identify the transaction upon which it should wait before unblocking the client.

Below is an example of how you might use the `WSREP_LAST_SEEN_GTID()` function to get the Global Transaction ID of the last write transaction observed:

```
BEGIN;

UPDATE table_name SET id = 0
WHERE field = 'example';
```

```
COMMIT;

SELECT WSREP_LAST_WRITTEN_GTID();
```

WSREP_SYNC_WAIT_UPTO_GTID()

This function blocks the client until the node applies and commits the given transaction.

Function	WSREP_LAST_WRITTEN_GTID()	
Arguments	*Required Arguments*	Global Transaction ID
	Optional Arguments	timeout
Support	4+	

This function blocks the client until the node applies and commits the given *Global Transaction ID*. If you don't provide a timeout, it defaults to the value of *repl.causal_read_timeout* (page 283). It the following return values:

- `1`: The node applied and committed the given Global Transaction ID.
- `ER_LOCAL_WAIT_TIMEOUT` Error: The function times out before the node can apply the transaction.
- `ER_WRONG_ARGUMENTS` Error: The function is given an incorrect Global Transaction ID.

Below is an example of how you might use the `WSREP_SYNC_WAIT_UPTO_GTID()` function:

```
$transaction_gtid = SELECT WSREP_LAST_SEEN_GTID();
...
SELECT WSREP_SYNC_WAIT_UPTO_GTID($transaction_gtid);
```

CHAPTER

FIFTYTHREE

GALERA PARAMETERS

As of version 0.8, Galera Cluster accepts parameters as semicolon-separated key value pair lists, such as `key1 = value1; key2 = value2`. In this way, you can configure an arbitrary number of Galera Cluster parameters in one call. A key consists of parameter group and parameter name:

```
<group>.<name>
```

Where `<group>` roughly corresponds to some Galera module.

Table legend:

- **Numeric values** Galera Cluster understands the following numeric modifiers: `K`, `M`, `G`, `T` standing for 2^{10}, 2^{20}, 2^{30} and 2^{40} respectively.
- **Boolean values** Galera Cluster accepts the following boolean values: `0`, `1`, `YES`, `NO`, `TRUE`, `FALSE`, `ON`, `OFF`.
- Time periods must be expressed in the ISO8601 format. See also the examples below.
- **T** indicates parameters that are strictly for use in troubleshooting problems. You should not implement these in production environments.

Parameter	Default	Support	Dynamic
base_host (page 268)	detected network address	1+	
base_port (page 268)	`4567`	1+	
cert.log_conflicts (page 268)	`NO`	2+	Yes
debug (page 268)	`NO`	2+	Yes
evs.auto_evict (page 268)	`0`	3.8+	No
evs.causal_keepalive_period (page 269)		1+	No
evs.consensus_timeout (page 269) T	`PT30S`	1 - 2	No
evs.debug_log_mask (page 269)	`0x1`	1+	Yes
evs.delayed_keep_period (page 269)	`PT30S`	3.8+	No
evs.delayed_margin (page 270)	`PT1S`	3.8+	No
evs.evict (page 270)		3.8	No
evs.inactive_check_period (page 270)	`PT1S`	1+	No
evs.inactive_timeout (page 271)	`PT15S`	1+	No

Continued on next page

Table 53.1 – continued from previous page

Parameter	Default	Support	Dynamic
evs.info_log_mask (page 271)	0	1+	No
evs.install_timeout (page 271)	PT15S	1+	Yes
evs.join_retrans_period (page 272)	PT1S	1+	Yes
evs.keepalive_period (page 272)	PT1S	1+	No
evs.max_install_timeouts (page 272)	1	1+	No
evs.send_window (page 272)	4	1+	Yes
evs.stats_report_period (page 273)	PT1M	1+	No
evs.suspect_timeout (page 273)	PT5S	1+	No
evs.use_aggregate (page 273)	TRUE	1+	No
evs.user_send_window (page 273)	2	1+	Yes
evs.view_forget_timeout (page 274)	PT5M	1+	No
evs.version (page 274) [T]	0	1+	No
gcache.dir (page 274)	working directory	1.0	No
gcache.name (page 275)	galera.cache	1+	No
gcache.keep_pages_size (page 275)	0	1+	No
gcache.page_size (page 275)	128Mb	1+	No
gcache.size (page 275)	128Mb	1+	No
gcomm.thread_prio (page 276)		3+	No
gcs.fc_debug (page 276)	0	1+	No
gcs.fc_factor (page 276)	0.5	1+	Yes
gcs.fc_limit (page 276)	16	1+	Yes
gcs.fc_master_slave (page 277)	NO	1+	No
gcs.max_packet_size (page 277)	32616	1+	No
gcs.max_throttle (page 277)	0.25	1+	No
gcs.recv_q_hard_limit (page 277)	LLONG_MAX	1+	No
gcs.recv_q_soft_limit (page 277)	0.25	1+	No
gcs.sync_donor (page 278)	NO	1+	No
gmcast.listen_addr (page 278)	tcp://0.0.0.0:4567	1+	No

Continued on next page

Table 53.1 – continued from previous page

Parameter	Default	Support	Dynamic
gmcast.mcast_addr (page 278)		1+	No
gmcast.mcast_ttl (page 278)	1	1+	No
gmcast.peer_timeout (page 279)	PT3S	1+	No
gmcast.segment (page 279)	0	3+	No
gmcast.time_wait (page 279)	PT5S	1+	No
gmcast.version (page 279) T	n/a		
ist.recv_addr (page 279)		1+	No
ist.recv_bind (page 280)		3+	No
pc.recovery (page 280)	TRUE	3+	No
pc.bootstrap (page 280)	n/a	2+	Yes
pc.announce_timeout (page 280)	PT3S	2+	No
pc.checksum (page 281)	TRUE	1+	No
pc.ignore_sb (page 281)	FALSE	1+	Yes
pc.ignore_quorum (page 281)	FALSE	1+	Yes
pc.linger (page 281)	PT2S	1+	No
pc.npvo (page 281)	FALSE	1+	No
pc.wait_prim (page 282)	FALSE	1+	No
pc.wait_prim_timeout (page 282)	P30S	2+	No
pc.weight (page 282)	1	2.4+	Yes
pc.version (page 282) T	n/a	1+	
protonet.backend (page 282)	asio	1+	No
protonet.version (page 282) T	n/a	1+	
repl.commit_order (page 283)	3	1+	No
repl.causal_read_timeout (page 283)	PT30S	1+	No
repl.key_format (page 283)	FLAT8	3+	No
repl.max_ws_size (page 283)	2147483647	3+	No
repl.proto_max (page 284)	5	2+	No
socket.ssl_ca (page 284)		1+	No
socket.ssl_cert (page 284)		1+	No
socket.checksum (page 284)	1 (for version 2) 2 (for version 3+)	2+	No
socket.ssl_cipher (page 285)	AES128-SHA	1+	No

Continued on next page

Table 53.1 – continued from previous page

Parameter	Default	Support	Dynamic
socket.ssl_compression (page 285)	YES	1+	No
socket.ssl_key (page 285)		1+	No
socket.ssl_password_file (page 285)		1+	No

base_host

Global variable for internal use.

Note: Warning: Do not manually set this variable.

Default Values	Dynamic	Introduced | Deprecated	
detected network address			

base_port

Global variable for internal use.

Note: Warning: Do not manually set this variable.

Default Value	Dynamic	Introduced	Deprecated
4567			

cert.log_conflicts

Log details of certification failures.

```
wsrep_provider_options="cert.log_conflicts=NO"
```

Default Value	Dynamic	Introduced	Deprecated
NO	Yes	2.0	

debug

Enable debugging.

```
wsrep_provider_options="debug=NO"
```

Default Value	Dynamic	Introduced	Deprecated
NO	Yes	2.0	

evs.auto_evict

Defines how many entries the node allows for given a delayed node before it triggers the Auto Eviction protocol.

```
wsrep_provider_options="evs.auto_evict=5"
```

Each cluster node monitors the group communication response times from all other nodes. When the cluster registers delayed response from a given node, it adds an entry for that node to its delayed list. If the majority of the cluster nodes show the node as delayed, the node is permanently evicted from the cluster.

This parameter determines how many entries a given node can receive before it triggers Auto Eviction.

When this parameter is set to `0`, it disables the Auto Eviction protocol for this node. Even when you disable Auto Eviction, though; the node continues to monitor response times from the cluster.

Note: **See Also**: For more information on the Auto Eviction process, see *Auto-Eviction* (page 111).

Default Value	Dynamic	Introduced	Deprecated
`0`	No	3.8	

evs.causal_keepalive_period

For developer use only. Defaults to `evs.keepalive_period`.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

evs.consensus_timeout

Timeout on reaching the consensus about cluster membership.

```
wsrep_provider_options="evs.consensus_timeout=PT30S"
```

This variable is mostly used for troubleshooting purposes and should not be implemented in a production environment.

Note: **See Also**: This feature has been **deprecated**. It is succeeded by *evs.install_timeout* (page 271).

Default Value	Dynamic	Introduced	Deprecated
`PT30S`	No	1.0	2.0

evs.debug_log_mask

Control EVS debug logging, only effective when `wsrep_debug` is in use.

```
wsrep_provider_options="evs.debug_log_mask=0x1"
```

Default Value	Dynamic	Introduced	Deprecated
`0x1`	Yes	1.0	

evs.delayed_keep_period

Defines how long this node requires a delayed node to remain responsive before it removes an entry from the delayed list.

```
wsrep_provider_options="evs.delayed_keep_period=PT45S"
```

Each cluster node monitors the group communication response times from all other nodes. When the cluster registered delayed responses from a given node, it adds an entry for that node to its delayed list. Nodes that remain on the delayed list can trigger Auto Eviction, which removes them permanently from the cluster.

This parameter determines how long a node on the delayed list must remain responsive before it removes one entry. The number of entries on the delayed list and how long it takes before the node removes all entries depends on how long the delayed node was unresponsive.

Note: See Also: For more information on the delayed list and the Auto Eviction process, see *Auto-Eviction* (page 111).

Default Value	Dynamic	Introduced	Deprecated
PT30S	No	3.8	

evs.delayed_margin

Defines how long the node allows response times to deviate before adding an entry to the delayed list.

```
wsrep_provider_options="evs.delayed_margin=PT5S"
```

Each cluster node monitors group communication response times from all other nodes. When the cluster registers a delayed response from a given node, it adds an entry for that node to its delayed list. Delayed nodes can trigger Auto Eviction, which removes them permanently from the cluster.

This parameter determines how long a delay can run before the node adds an entry to the delayed list. You must set this parameter to a value higher than the round-trip delay time (RTT) between the nodes.

Note: See Also: For more information on the delayed list and the Auto Eviction process, see *Auto-Eviction* (page 111).

Default Value	Dynamic	Introduced	Deprecated
PT1S	No	3.8	

evs.evict

Defines the point at which the cluster triggers manual eviction to a certain node value. Setting this parameter as an empty string causes it to clear the eviction list on the node where it is set.

Note: See Also: For more information on the eviction and Auto Eviction process, see *Auto-Eviction* (page 111).

Default Value	Dynamic	Introduced	Deprecated
	No	3.8	

evs.inactive_check_period

Defines how often you want the node to check for peer inactivity.

```
wsrep_provider_options="evs.inactive_check_period=PT1S"
```

Each cluster node monitors group communication response times from all other nodes. When the cluster registers a delayed response from a given node, it adds an entry for that node to its delayed list, which can lead to the delayed node's eviction from the cluster.

This parameter determines how often you want the node to check for delays in the group communication responses from other cluster nodes.

Default Value	Dynamic	Introduced	Deprecated
PT1S	No	1.0	

evs.inactive_timeout

Defines a hard limit on node inactivity.

Hard limit on the inactivity period, after which the node is pronounced dead.

```
wsrep_provider_options="evs.inactive_timeout=PT15S"
```

Each cluster node monitors group communication response times from all other nodes. When the cluster registers a delayed response from a given node, it add an entry for that node to its delayed list, which can lead tot he delayed node's eviction from the cluster.

This parameter sets a hard limit for node inactivity. If a delayed node remains unresponsive for longer than this period, the node pronounces the delayed node as dead.

Default Value	Dynamic	Introduced	Deprecated
PT15S	No	1.0	

evs.info_log_mask

Defines additional logging options for the EVS Protocol.

```
wsrep_provider_options="evs.info_log_mask=0x4"
```

The EVS Protocol monitors group communication response times and controls the node eviction and auto eviction processes. This parameter allows you to enable additional logging options, through a bitmask value.

- `0x1` Provides extra view change info.
- `0x2` Provides extra state change info
- `0x4` Provides statistics
- `0x8` Provides profiling (only in builds with profiling enabled)

Default Value	Dynamic	Introduced	Deprecated
0	No	1.0	

evs.install_timeout

Defines the timeout for install message acknowledgments.

```
wsrep_provider_options="evs.install_timeout=PT15S"
```

Each cluster node monitors group communication response times from all other nodes, checking whether they are responsive or delayed. This parameter determines how long you want the node to wait on install message acknowledgments.

Note: See Also: This parameter replaces *evs.consensus_timeout* (page 269).

Default Value	Dynamic	Introduced	Deprecated
PT15S	Yes	1.0	

evs.join_retrans_period

Defines how often the node retransmits EVS join messages when forming cluster membership.

```
wsrep_provider_options="evs.join_retrans_period=PT1S"
```

Default Value	Dynamic	Introduced	Deprecated
PT1S	Yes	1.0	

evs.keepalive_period

Defines how often the node emits keepalive signals.

```
wsrep_provider_options="evs.keepalive_period=PT1S"
```

Each cluster node monitors group communication response times from all other nodes. When there is no traffic going out for the cluster to monitor, nodes emit keepalive signals so that other nodes have something to measure. This parameter determines how often the node emits a keepalive signal, absent any other traffic.

Default Value	Dynamic	Introduced	Deprecated
PT1S	No	1.0	

evs.max_install_timeouts

Defines the number of membership install rounds to try before giving up.

```
wsrep_provider_options="evs.max_install_timeouts=1"
```

This parameter determines the maximum number of times that the node tries for a membership install acknowledgment, before it stops trying. The total number of rounds it tries is this value plus 2.

Default Value	Dynamic	Introduced	Deprecated
1	No	1.0	

evs.send_window

Defines the maximum number of packets at a time in replication.

```
wsrep_provider_options="evs.send_window=4"
```

This parameter determines the maximum number of packets the node uses at a time in replication. For clusters implemented over WAN, you can set this value considerably higher, (for example, 512), than for clusters implemented over LAN.

You must use a value that is greater than *evs.user_send_window* (page 273). The recommended value is double *evs.user_send_window* (page 273).

Default Value	Dynamic	Introduced	Deprecated
4	Yes	1.0	

evs.stats_report_period

Control period of EVS statistics reporting. The node is pronounced dead.

```
wsrep_provider_options="evs.stats_report_period=PT1M"
```

Default Value	Dynamic	Introduced	Deprecated
PT1M	No	1.0	

evs.suspect_timeout

Defines the inactivity period after which a node is *suspected* as dead.

```
wsrep_provider_options="evs.suspect_timeout=PT5S"
```

Each node in the cluster monitors group communications from all other nodes in the cluster. This parameter determines the period of inactivity before the node suspects another of being dead. If all nodes agree on that, the cluster drops the inactive node.

Default Value	Dynamic	Introduced	Deprecated
PT5S	No	1.0	

evs.use_aggregate

Defines whether the node aggregates small packets into one when possible.

```
wsrep_provider_options="evs.use_aggregate=TRUE"
```

Default Value	Dynamic	Introduced	Deprecated
TRUE	No	1	

evs.user_send_window

Defines the maximum number of data packets at a time in replication.

```
wsrep_provider_options="evs.user_send_window=2"
```

This parameter determines the maximum number of data packets the node uses at a time in replication. For clusters implemented over WAN, you can set this to a value considerably higher than cluster implementations over LAN, (for example, 512).

You must use a value that is smaller than *evs.send_window* (page 272). The recommended value is half *evs.send_window* (page 272).

Note: **See Also**: *evs.send_window* (page 272).

Default Value	Dynamic	Introduced	Deprecated
2	Yes	1.0	

evs.view_forget_timeout

Defines how long the node saves past views from the view history.

```
wsrep_provider_options="evs.view_forget_timeout=PT5M"
```

Each node maintains a history of past views. This parameter determines how long you want the node to save past views before dropping them from the table.

Default Value	Dynamic	Introduced	Deprecated
PT5M	No	1.0	

evs.version

Defines the EVS Protocol version.

```
wsrep_provider_options="evs.version=1"
```

This parameter determines which version of the EVS Protocol the node uses. In order to ensure backwards compatibility, the parameter defaults to 0. Certain EVS Protocol features, such as Auto Eviction, require you to upgrade to more recent versions.

Note: **See Also**: For more information on the procedure to upgrade from one version to another, see *Upgrading the EVS Protocol* (page 112).

Default Value	Dynamic	Introduced	Deprecated
0	No	1.0	

gcache.dir

Defines the directory where the write-set cache places its files.

```
wsrep_provider_options="gcache.dir=/usr/share/galera"
```

When nodes receive state transfers they cannot process incoming write-sets until they finish updating their state. Under certain methods, the node that sends the state transfer is similarly blocked. To prevent the database from falling further behind, GCache saves the incoming write-sets on memory mapped files to disk.

This parameter determines where you want the node to save these files for write-set caching. By default, GCache uses the working directory for the database server.

Default Value	Dynamic	Introduced	Deprecated
/path/to/working_dir	No	1.0	

gcache.keep_pages_size

Total size of the page storage pages to keep for caching purposes. If only page storage is enabled, one page is always present.

```
wsrep_provider_options="gcache.keep_pages_size=0"
```

Default Value	Dynamic	Introduced	Deprecated
0	No	1.0	

gcache.name

Defines the filename for the write-set cache.

```
wsrep_provider_options="gcache.name=galera.cache"
```

When nodes receive state transfers they cannot process incoming write-sets until they finish updating their state. Under certain methods, the node that sends the state transfer is similarly blocked. To prevent the database from falling further behind, GCache saves the incoming write-sets on memory-mapped files to disk.

This parameter determines the name you want the node to use for this ring buffer storage file.

Default Value	Dynamic	Introduced	Deprecated
galera.cache	No	1.0	

gcache.page_size

Size of the page files in page storage. The limit on overall page storage is the size of the disk. Pages are prefixed by `gcache.page`.

```
wsrep_provider_options="gcache.page_size=128Mb"
```

Default Value	Dynamic	Introduced	Deprecated
128M	No	1.0	

gcache.size

Defines the disk space you want to node to use in caching write-sets.

```
wsrep_provider_options="gcache.size=128Mb"
```

When nodes receive state transfers they cannot process incoming write-sets until they finish updating their state. Under certain methods, the node that sends the state transfer is similarly blocked. To prevent the database from falling further behind, GCache saves the incoming write-sets on memory-mapped files to disk.

This parameter defines the amount of disk space you want to allocate for the present ring buffer storage. The node allocates this space when it starts the database server.

Note: See Also: For more information on customizing the write-set cache, see *Performance* (page 223).

Default Value	Dynamic	Introduced	Deprecated
128M	No	1.0	

gcomm.thread_prio

Defines the policy and priority for the gcomm thread.

```
wsrep_provider_options="gcomm.thread_prio=rr:2"
```

Using this option, you can raise the priority of the gcomm thread to a higher level than it normally uses. You may find this useful in situations where Galera Cluster threads do not receive sufficient CPU time, due to competition with other MySQL threads. In these cases, when the thread scheduler for the operating system does not run the Galera threads frequently enough, timeouts may occur, causing the node to drop from the cluster.

The format for this option is: `<policy>:<priority>`. The priority value is an integer. The policy value supports the following options:

- `other` Designates the default time-sharing scheduling in Linux. They can run until they are blocked by an I/O request or preempted by higher priorities or superior scheduling designations.
- `fifo` Designates first-in out scheduling. These threads always immediately preempt any currently running other, batch or idle threads. They can run until they are either blocked by an I/O request or preempted by a FIFO thread of a higher priority.
- `rr` Designates round-robin scheduling. These threads always preempt any currently running other, batch or idle threads. The scheduler allows these threads to run for a fixed period of a time. If the thread is still running when this time period is exceeded, they are stopped and moved to the end of the list, allowing another round-robin thread of the same priority to run in their place. They can otherwise continue to run until they are blocked by an I/O request or are preempted by threads of a higher priority.

Default Value	Dynamic	Introduced	Deprecated
	No	3.0	

gcs.fc_debug

Post debug statistics about SST flow every this number of writesets.

```
wsrep_provider_options="gcs.fc_debug=0"
```

Default Value	Dynamic	Introduced	Deprecated
0	No	1.0	

gcs.fc_factor

Resume replication after recv queue drops below this fraction of `gcs.fc_limit`.

```
wsrep_provider_options="gcs.fc_factor=0.5"
```

Default Value	Dynamic	Introduced	Deprecated
0.5	Yes	1.0	

gcs.fc_limit

Pause replication if recv queue exceeds this number of writesets. For master-slave setups this number can be increased considerably.

```
wsrep_provider_options="gcs.fc_limit=16"
```

Default Value	Dynamic	Introduced	Deprecated
16	Yes	1.0	

gcs.fc_master_slave

Defines whether there is only one master node in the group.

```
wsrep_provider_options="gcs.fc_master_slave=NO"
```

Default Value	Dynamic	Introduced	Deprecated
NO	No	1.0	

gcs.max_packet_size

All writesets exceeding that size will be fragmented.

```
wsrep_provider_options="gcs.max_packet_size=32616"
```

Default Value	Dynamic	Introduced	Deprecated
32616	No	1.0	

gcs.max_throttle

How much to throttle replication rate during state transfer (to avoid running out of memory). Set the value to 0.0 if stopping replication is acceptable for completing state transfer.

```
wsrep_provider_options="gcs.max_throttle=0.25"
```

Default Value	Dynamic	Introduced	Deprecated
0.25	No	1.0	

gcs.recv_q_hard_limit

Maximum allowed size of recv queue. This should normally be half of (RAM + swap). If this limit is exceeded, Galera Cluster will abort the server.

```
wsrep_provider_options="gcs.recv_q_hard_limit=LLONG_MAX"
```

Default Value	Dynamic	Introduced	Deprecated
LLONG_MAX	No	1.0	

gcs.recv_q_soft_limit

The fraction of *gcs.recv_q_hard_limit* (page 277) after which replication rate will be throttled.

```
wsrep_provider_options="gcs.recv_q_soft_limit=0.25"
```

The degree of throttling is a linear function of recv queue size and goes from 1.0 (`full rate`) at *gcs.recv_q_soft_limit* (page 277) to *gcs.max_throttle* (page 277) at *gcs.recv_q_hard_limit* (page 277) Note that `full rate`, as estimated between 0 and *gcs.recv_q_soft_limit* (page 277) is a very imprecise estimate of a regular replication rate.

Default Value	Dynamic	Introduced	Deprecated
`0.25`	No	1.0	

gcs.sync_donor

Should the rest of the cluster keep in sync with the donor? `YES` means that if the donor is blocked by state transfer, the whole cluster is blocked with it.

```
wsrep_provider_options="gcs.sync_donor=NO"
```

If you choose to use value `YES`, it is theoretically possible that the donor node cannot keep up with the rest of the cluster due to the extra load from the SST. If the node lags behind, it may send flow control messages stalling the whole cluster. However, you can monitor this using the *wsrep_flow_control_paused* (page 294) status variable.

Default Value	Dynamic	Introduced	Deprecated
`NO`	No	1.0	

gmcast.listen_addr

Address at which *Galera Cluster* listens to connections from other nodes. By default the port to listen at is taken from the connection address. This setting can be used to overwrite that.

```
wsrep_provider_options="gmcast.listen_addr=tcp://0.0.0.0:4567"
```

Default Value	Dynamic	Introduced	Deprecated
`tcp://0.0.0.0"4567`	No	1.0	

gmcast.mcast_addr

If set, UDP multicast will be used for replication, for example:

```
wsrep_provider_options="gmcast.mcast_addr=239.192.0.11"
```

The value must be the same on all nodes.

If you are planning to build a large cluster, we recommend using UDP.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

gmcast.mcast_ttl

Time to live value for multicast packets.

```
wsrep_provider_options="gmcast.mcast_ttl=1"
```

Default Value	Dynamic	Introduced	Deprecated
`1`	No	1.0	

gmcast.peer_timeout

Connection timeout to initiate message relaying.

```
wsrep_provider_options="gmcast.peer_timeout=PT3S"
```

Default Value	Dynamic	Introduced	Deprecated
PT3S	No	1.0	

gmcast.segment

Define which network segment this node is in. Optimisations on communication are performed to minimise the amount of traffic between network segments including writeset relaying and IST and SST donor selection. The *gmcast.segment* (page 279) value is an integer from 0 to 255. By default all nodes are placed in the same segment (0).

```
wsrep_provider_options="gmcast.segment=0"
```

Default Value	Dynamic	Introduced	Deprecated
0	No	3.0	

gmcast.time_wait

Time to wait until allowing peer declared outside of stable view to reconnect.

```
wsrep_provider_options="gmcast.time_wait=PT5S"
```

Default Value	Dynamic	Introduced	Deprecated
PT5S	No	1.0	

gmcast.version

This status variable is used to check which gmcast protocol version is used.

This variable is mostly used for troubleshooting purposes and should not be implemented in a production environment.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

ist.recv_addr

Address to listen on for Incremental State Transfer. By default this is the <address>:<port+1> from *wsrep_node_address* (page 246).

```
wsrep_provider_options="ist.recv_addr=192.168.1.1"
```

Default Value	Dynamic	Introduced	Deprecated
	No	2.0	

ist.recv_bind

Defines the address that the node binds on for receiving an *Incremental State Transfer*.

```
wsrep_provider_options="ist.recv_bind=192.168.1.1"
```

This option defines the address to which the node will bind in order to receive Incremental State Transfers. When this option is not set, it takes its value from *ist.recv_addr* (page 279) or, in the event that that is also not set, from *wsrep_node_address* (page 246). You may find it useful when the node runs behind a NAT or in similar cases where the public and private addresses differ.

Default Value	Dynamic	Introduced	Deprecated
	No	3.16	

pc.recovery

When set to `TRUE`, the node stores the Primary Component state to disk, in the `gvwstate.dat` file. The Primary Component can then recover automatically when all nodes that were part of the last saved state reestablish communications with each other.

```
wsrep_provider_options="pc.recovery=TRUE"
```

This allows for:

- Automatic recovery from full cluster crashes, such as in the case of a data center power outage.
- Graceful full cluster restarts without the need for explicitly bootstrapping a new Primary Component.

Note: In the event that the wsrep position differs between nodes, recovery also requires a full State Snapshot Transfer.

Default Value	Dynamic	Introduced	Deprecated
`TRUE`	No	3.0	

pc.bootstrap

If you set this value to `TRUE` is a signal to turn a `NON-PRIMARY` component into `PRIMARY`.

```
wsrep_provider_options="pc.bootstrap=TRUE"
```

Default Value	Dynamic	Introduced	Deprecated
	Yes	2.0	

pc.announce_timeout

Cluster joining announcements are sent every $\frac{1}{2}$ second for this period of time or less if the other nodes are discovered.

```
wsrep_provider_options="pc.announce_timeout=PT3S"
```

Default Value	Dynamic	Introduced	Deprecated
`PT3S`	No	2.0	

pc.checksum

Checksum replicated messages.

```
wsrep_provider_options="pc.checksum=TRUE"
```

Default Value	Dynamic	Introduced	Deprecated
TRUE	No	1.0	

pc.ignore_sb

Should we allow nodes to process updates even in the case of split brain? This is a dangerous setting in multi-master setup, but should simplify things in master-slave cluster (especially if only 2 nodes are used).

```
wsrep_provider_options="pc.ignore_sb=FALSE"
```

Default Value	Dynamic	Introduced	Deprecated
FALSE	Yes	1.0	

pc.ignore_quorum

Completely ignore quorum calculations. For example if the master splits from several slaves it still remains operational. Use with extreme caution even in master-slave setups, as slaves will not automatically reconnect to master in this case.

```
wsrep_provider_options="pc.ignore_quorum=FALSE"
```

Default Value	Dynamic	Introduced	Deprecated
FALSE	Yes	1.0	

pc.linger

The period for which the PC protocol waits for the EVS termination.

```
wsrep_provider_options="pc.linger=PT2S"
```

Default Value	Dynamic	Introduced	Deprecated
PT2S	No	1.0	

pc.npvo

If set to TRUE, the more recent primary component overrides older ones in the case of conflicting primaries.

```
wsrep_provider_options="pc.npvo=FALSE"
```

Default Value	Dynamic	Introduced	Deprecated
FALSE	No	1.0	

pc.wait_prim

If set to TRUE, the node waits for the *pc.wait_prim_timeout* (page 282) time period. Useful to bring up a non-primary component and make it primary with *pc.bootstrap* (page 280).

```
wsrep_provider_options="pc.wait_prim=FALSE"
```

Default Value	Dynamic	Introduced	Deprecated
FALSE	No	1.0	

pc.wait_prim_timeout

The period of time to wait for a primary component.

```
wsrep_provider_options="pc.wait_prim_timeout=PT30S"
```

Default Value	Dynamic	Introduced	Deprecated
PT30S	No	2.0	

pc.weight

As of version 2.4. Node weight for quorum calculation.

```
wsrep_provider_options="pc.weight=1"
```

Default Value	Dynamic	Introduced	Deprecated
1	Yes	2.4	

pc.version

This status variable is used to check which pc protocol version is used.

This variable is mostly used for troubleshooting purposes and should not be implemented in a production environment.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

protonet.backend

Which transport backend to use. Currently only ASIO is supported.

```
wsrep_provider_options="protonet.backend=asio"
```

Default Value	Dynamic	Introduced	Deprecated
asio	No	1.0	

protonet.version

This status variable is used to check which transport backend protocol version is used.

This variable is mostly used for troubleshooting purposes and should not be implemented in a production environment.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

repl.commit_order

Whether to allow Out-Of-Order committing (improves parallel applying performance).

```
wsrep_provider_options="repl.commit_order=2"
```

Possible settings:

- `0` or `BYPASS` All commit order monitoring is switched off (useful for measuring performance penalty).
- `1` or `OOOC` Allows out of order committing for all transactions.
- `2` or `LOCAL_OOOC` Allows out of order committing only for local transactions.
- `3` or `NO_OOOC` No out of order committing is allowed (strict total order committing)

Default Value	Dynamic	Introduced	Deprecated
`3`	No	1.0	

repl.causal_read_timeout

Sometimes causal reads need to timeout.

```
wsrep_provider_options="repl.causal_read_timeout=PT30S"
```

Default Value	Dynamic	Introduced	Deprecated
`PT30S`	No	1.0	

repl.key_format

The hash size to use for key formats (in bytes). An `A` suffix annotates the version.

```
wsrep_provider_options="repl.key_format=FLAT8"
```

Possible settings:

- `FLAT8`
- `FLAT8A`
- `FLAT16`
- `FLAT16A`

Default Value	Dynamic	Introduced	Deprecated
`FLAT8`	No	3.0	

repl.max_ws_size

The maximum size of a write-set in bytes. This is limited to 2G.

```
wsrep_provider_options="repl.max_ws_size=2147483647"
```

Default Value	Dynamic	Introduced	Deprecated
2147483647	No	3.0	

repl.proto_max

The maximum protocol version in replication. Changes to this parameter will only take effect after a provider restart.

```
wsrep_provider_options="repl.proto_max=5"
```

Default Value	Dynamic	Introduced	Deprecated
5	No	2.0	

socket.ssl_ca

Defines the path to the SSL Certificate Authority (CA) file.

The node uses the CA file to verify the signature on the certificate. You can use either an absolute path or one relative to the working directory. The file must use PEM format.

```
wsrep_provider_options='socket.ssl_ca=/path/to/ca-cert.pem'
```

Note: **See Also**: For more information on generating SSL certificate files for your cluster, see *SSL Certificates* (page 175).

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

socket.ssl_cert

Defines the path to the SSL certificate.

The node uses the certificate as a self-signed public key in encrypting replication traffic over SSL. You can use either an absolute path or one relative to the working directory. The file must use PEM format.

```
wsrep_provider_options="socket.ssl_cert=/path/to/server-cert.pem"
```

Note: **See Also**: For more information on generating SSL certificate files for your cluster, see *SSL Certificates* (page 175).

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

socket.checksum

Checksum to use on socket layer:

- 0 - disable checksum
- 1 - CRC32

- 2 - CRC-32C (optimized and potentially HW-accelerated on Intel CPUs)

```
wsrep_provider_options="socket.checksum=2"
```

Default Value	Dynamic	Introduced	Deprecated
version 1 : 1	No	2.0	
version 3+: 2			

socket.ssl_cipher

Symmetric cipher to use. AES128 is used by default it is considerably faster and no less secure than AES256.

```
wsrep_provider_options="socket.ssl_cipher=AES128-SHA"
```

Default Value	Dynamic	Introduced	Deprecated
AES128-SHA	No	1.0	

socket.ssl_compression

Whether to enable compression on SSL connections.

```
wsrep_provider_options="socket.ssl_compression=YES"
```

Default Value	Dynamic	Introduced	Deprecated
YES	No	1.0	

socket.ssl_key

Defines the path to the SSL certificate key.

The node uses the certificate key a self-signed private key in encrypting replication traffic over SSL. You can use either an absolute path or one relative to the working directory. The file must use PEM format.

```
wsrep_provider_options="socket.ssl_key=/path/to/server-key.pem"
```

Note: **See Also**: For more information on generating SSL certificate files for your cluster, see *SSL Certificates* (page 175).

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

socket.ssl_password_file

Defines a password file for use in SSL connections.

```
wsrep_provider_options="socket.ssl_password_file=/path/to/password-file"
```

In the event that you have your SSL key file encrypted, the node uses the SSL password file to decrypt the key file.

Default Value	Dynamic	Introduced	Deprecated
	No	1.0	

Setting Galera Parameters in MySQL

You can set *Galera Cluster* parameters in the `my.cnf` configuration file as follows:

```
wsrep_provider_options="gcs.fc_limit=256;gcs.fc_factor=0.9"
```

This is useful in master-slave setups.

You can set Galera Cluster parameters through a MySQL client with the following query:

```
SET GLOBAL wsrep_provider_options="evs.send_window=16";
```

This query only changes the *evs.send_window* (page 272) value.

To check which parameters are used in Galera Cluster, enter the following query:

```
SHOW VARIABLES LIKE 'wsrep_provider_options';
```

CHAPTER

FIFTYFOUR

GALERA STATUS VARIABLES

These variables are *Galera Cluster* 0.8.x status variables. There are two types of wsrep-related status variables:

- Galera Cluster-specific variables exported by Galera Cluster
- Variables exported by MySQL. These variables are for the general wsrep provider.

This distinction is of importance for developers only. For convenience, all status variables are presented as a single list below. Variables exported by MySQL are indicated by an *M* in superscript.

Status Variable	Example	Support
wsrep_apply_oooe (page 288)	0.671120	1+
wsrep_apply_oool (page 288)	0.195248	1+
wsrep_apply_window (page 289)	5.163966	1+
wsrep_cert_deps_distance (page 289)	23.88889	1+
wsrep_cert_index_size (page 289)	30936	1+
wsrep_cert_interval (page 290)		1+
wsrep_cluster_conf_id (page 290) M	34	1+
wsrep_cluster_size (page 290) M	3	1+
wsrep_cluster_state_uuid (page 291) M		1+
wsrep_cluster_status (page 291) M	Primary	1+
wsrep_commit_oooe (page 291)	0.000000	1+
wsrep_commit_oool (page 291)	0.000000	1+
wsrep_commit_window (page 292)	0.000000	1+
wsrep_connected (page 292)	ON	1+
wsrep_desync_count (page 292)	0	3+
wsrep_evs_delayed (page 293)		3.8+
wsrep_evs_evict_list (page 293)		3.8+
wsrep_evs_repl_latency (page 293)		3.0+
wsrep_evs_state (page 294)		3.8+
wsrep_flow_control_paused (page 294)	0.184353	1+
wsrep_flow_control_paused_ns (page 294)	20222491180	1+
wsrep_flow_control_recv (page 294)	11	1+
wsrep_flow_control_sent (page 295)	7	1+
wsrep_gcomm_uuid (page 295)		1+
wsrep_incoming_addresses (page 295)		1+
wsrep_last_committed (page 295)	409745	1+
wsrep_local_bf_aborts (page 296)	960	1+
wsrep_local_cached_downto (page 296)		1+
wsrep_local_cert_failures (page 296)	333	1+
wsrep_local_commits (page 297)	14981	1+

Continued on next page

Table 54.1 – continued from previous page

Status Variable	Example	Support
wsrep_local_index (page 297)	1	1+
wsrep_local_recv_queue (page 297)	0	1+
wsrep_local_recv_queue_avg (page 297)	3.348452	1+
wsrep_local_recv_queue_max (page 298)	10	1+
wsrep_local_recv_queue_min (page 298)	0	1+
wsrep_local_replays (page 298)	0	1+
wsrep_local_send_queue (page 299)	1	1+
wsrep_local_send_queue_avg (page 299)	0.145000	1+
wsrep_local_send_queue_max (page 299)	10	1+
wsrep_local_send_queue_min (page 299)	0	1+
wsrep_local_state (page 300)	4	1+
wsrep_local_state_comment (page 300)	Synced	1+
wsrep_local_state_uuid (page 300)		1+
wsrep_protocol_version (page 301)	4	1+
wsrep_provider_name (page 301) M	Galera	1+
wsrep_provider_vendor (page 301) M		1+
wsrep_provider_version (page 302) M		1+
wsrep_ready (page 302) M	ON	1+
wsrep_received (page 302)	17831	1+
wsrep_received_bytes (page 302)	6637093	1+
wsrep_repl_data_bytes (page 303)	265035226	1+
wsrep_repl_keys (page 303)	797399	1+
wsrep_repl_keys_bytes (page 303)	11203721	1+
wsrep_repl_other_bytes (page 304)	0	1+
wsrep_replicated (page 304)	16109	1+
wsrep_replicated_bytes (page 304)	6526788	1+

wsrep_apply_oooe

How often applier started write-set applying out-of-order (parallelization efficiency).

```
SHOW STATUS LIKE 'wsrep_apply_oooe';

+------------------+----------+
| Variable_name    | Value    |
+------------------+----------+
| wsrep_apply_oooe | 0.671120 |
+------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.671120	Galera		

wsrep_apply_oool

How often write-set was so slow to apply that write-set with higher seqno's were applied earlier. Values closer to 0 refer to a greater gap between slow and fast write-sets.

```
SHOW STATUS LIKE 'wsrep_apply_oool';

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

```
| Variable_name    | Value    |
+------------------+----------+
| wsrep_apply_oool | 0.195248 |
+------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.195248	Galera		

wsrep_apply_window

Average distance between highest and lowest concurrently applied seqno.

```
SHOW STATUS LIKE 'wsrep_apply_window';

+--------------------+----------+
| Variable_name      | Value    |
+--------------------+----------+
| wsrep_apply_window | 5.163966 |
+--------------------+----------+
```

Example Value	Location	Introduced	Deprecated
5.163966	Galera		

wsrep_cert_deps_distance

Average distance between highest and lowest seqno value that can be possibly applied in parallel (potential degree of parallelization).

```
SHOW STATUS LIKE 'wsrep_cert_deps_distance';

+--------------------------+----------+
| Variable_name            | Value    |
+--------------------------+----------+
| wsrep_cert_deps_distance | 23.88889 |
+--------------------------+----------+
```

Example Value	Location	Introduced	Deprecated
23.888889	Galera		

wsrep_cert_index_size

The number of entries in the certification index.

```
SHOW STATUS LIKE 'wsrep_certs_index_size';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_certs_index_size | 30936 |
+------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
30936	Galera		

wsrep_cert_interval

Average number of transactions received while a transaction replicates.

```
SHOW STATUS LIKE 'wsrep_cert_interval';

+----------------------+-------+
| Variable_name        | Value |
+----------------------+-------+
| wsrep_cert_interval | 1.0   |
+----------------------+-------+
```

When a node replicates a write-set to the cluster, it can take some time before all the nodes in the cluster receive it. By the time a given node receives, orders and commits a write-set, it may receive and potentially commit others, changing the state of the database from when the write-set was sent and rendering the transaction inapplicable.

To prevent this, Galera Cluster checks write-sets against all write-sets within its certification interval for potential conflicts. Using the *wsrep_cert_interval* (page 290) status variable, you can see the average number of transactions with the certification interval.

This shows you the number of write-sets concurrently replicating to the cluster. In a fully synchronous cluster, with one write-set replicating at a time, *wsrep_cert_interval* (page 290) returns a value of `1.0`.

Example Value	Location	Introduced	Deprecated
1.0	Galera		

wsrep_cluster_conf_id

Total number of cluster membership changes happened.

```
SHOW STATUS LIKE 'wsrep_cluster_conf_id';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_cluster_conf_id | 34    |
+------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
34	MySQL		

wsrep_cluster_size

Current number of members in the cluster.

```
SHOW STATUS LIKE 'wsrep_cluster_size';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_cluster_size | 15    |
+--------------------+-------+
```

Example Value	Location	Introduced	Deprecated
3	MySQL		

wsrep_cluster_state_uuid

Provides the current State UUID. This is a unique identifier for the current state of the cluster and the sequence of changes it undergoes.

```
SHOW STATUS LIKE 'wsrep_cluster_state_uuid';

+--------------------------+--------------------------------------+
| Variable_name            | Value                                |
+--------------------------+--------------------------------------+
| wsrep_cluster_state_uuid | e2c9a15e-5485-11e0-0800-6bbb637e7211 |
+--------------------------+--------------------------------------+
```

Note: **See Also**: For more information on the state UUID, see *wsrep API* (page 51).

Example Value	Location	Introduced	Deprecated
e2c9a15e-5485-11e0 0900-6bbb637e7211	MySQL		

wsrep_cluster_status

Status of this cluster component. That is, whether the node is part of a PRIMARY or NON_PRIMARY component.

```
SHOW STATUS LIKE 'wsrep_cluster_status';

+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_cluster_status | Primary |
+----------------------+---------+
```

Example Value	Location	Introduced	Deprecated
Primary	MySQL		

wsrep_commit_oooe

How often a transaction was committed out of order.

```
SHOW STATUS LIKE 'wsrep_commit_oooe';

+-------------------+----------+
| Variable_name     | Value    |
+-------------------+----------+
| wsrep_commit_oooe | 0.000000 |
+-------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.000000	Galera		

wsrep_commit_oool

No meaning.

```
SHOW STATUS LIKE 'wsrep_commit_oool';

+--------------------+----------+
| Variable_name      | Value    |
+--------------------+----------+
| wsrep_commit_oool | 0.000000 |
+--------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.000000	Galera		

wsrep_commit_window

Average distance between highest and lowest concurrently committed seqno.

```
SHOW STATUS LIKE 'wsrep_commit_window';

+---------------------+----------+
| Variable_name       | Value    |
+---------------------+----------+
| wsrep_commit_window | 0.000000 |
+---------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.000000	Galera		

wsrep_connected

If the value is OFF, the node has not yet connected to any of the cluster components. This may be due to misconfiguration. Check the error log for proper diagnostics.

```
SHOW STATUS LIKE 'wsrep_connected';

+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| wsrep_connected | ON    |
+-----------------+-------+
```

Example Value	Location	Introduced	Deprecated
ON	Galera		

wsrep_desync_count

Returns the number of operations in progress that require the node to temporarily desync from the cluster.

```
SHOW STATUS LIKE 'wsrep_desync_count';

+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| wsrep_desync_count | 1     |
+--------------------+-------+
```

Certain operations, such as DDL statements issued when *wsrep_OSU_method* (page 249) is set to Rolling Schema Upgrade or when you enable *wsrep_desync* (page 241), cause the node to desync from the cluster. This status variable shows how many of these operations are currently running on the node. When all of these operations complete, the counter returns to its default value `0` and the node can sync back to the cluster.

Example Value	Location	Introduced	Deprecated
`0`	Galera	3.8	

wsrep_evs_delayed

Provides a comma separated list of all the nodes this node has registered on its delayed list.

The node listing format is

```
uuid:address:count
```

This refers to the UUID and IP address of the delayed node, with a count of the number of entries it has on the delayed list.

Example Value	Location	Introduced	Deprecated
	Galera	3.8	

wsrep_evs_evict_list

Lists the UUID's of all nodes evicted from the cluster. Evicted nodes cannot rejoin the cluster until you restart their `mysqld` processes.

Example Value	Location	Introduced	Deprecated
	Galera	3.8	

wsrep_evs_repl_latency

This status variable provides figures for the replication latency on group communication. It measures latency from the time point when a message is sent out to the time point when a message is received. As replication is a group operation, this essentially gives you the slowest ACK and longest RTT in the cluster.

For example,

```
SHOW STATUS LIKE 'wsrep_evs_repl_latency';

+------------------------+-------------------------------------------+
| Variable_name          | Value                                     |
+------------------------+-------------------------------------------+
| wsrep_evs_repl_latency | 0.00243433/0.144022/0.591963/0.215824/13 |
+------------------------+-------------------------------------------+
```

The units are in seconds. The format of the return value is:

```
Minimum / Average / Maximum / Standard Deviation / Sample Size
```

This variable periodically resets. You can control the reset interval using the *evs.stats_report_period* (page 273) parameter. The default value is 1 minute.

Example Value	Location	Introduced	Deprecated
`0.00243433/0.144033/ 0.581963/0.215724/13`	Galera	3.0	

wsrep_evs_state

Shows the internal state of the EVS Protocol.

Example Value	Location	Introduced	Deprecated
	Galera	3.8	

wsrep_flow_control_paused

The fraction of time since the last `FLUSH STATUS` command that replication was paused due to flow control.

In other words, how much the slave lag is slowing down the cluster.

```
SHOW STATUS LIKE 'wsrep_flow_control_paused';

+--------------------------+-----------+
| Variable_name            | Value     |
+--------------------------+-----------+
| wsrep_flow_control_paused | 0.184353 |
+--------------------------+-----------+
```

Example Value	Location	Introduced	Deprecated
0.174353	Galera		

wsrep_flow_control_paused_ns

The total time spent in a paused state measured in nanoseconds.

```
SHOW STATUS LIKE 'wsrep_flow_control_paused_ns';

+-----------------------------+-------------+
| Variable_name               | Value       |
+-----------------------------+-------------+
| wsrep_flow_control_paused_ns | 20222491180 |
+-----------------------------+-------------+
```

Example Value	Location	Introduced	Deprecated
20222491180	Galera		

wsrep_flow_control_recv

Returns the number of `FC_PAUSE` events the node has received, including those the node has sent. Unlike most status variables, the counter for this one does not reset every time you run the query.

```
SHOW STATUS LIKE 'wsrep_flow_control_recv';

+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| wsrep_flow_control_recv | 11    |
+-------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
11	Galera		

wsrep_flow_control_sent

Returns the number of FC_PAUSE events the node has sent. Unlike most status variables, the counter for this one does not reset every time you run the query.

```
SHOW STATUS LIKE 'wsrep_flow_control_sent';

+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| wsrep_flow_control_sent | 7     |
+-------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
7	Galera		

wsrep_gcomm_uuid

Displays the group communications UUID.

```
SHOW STATUS LIKE 'wsrep_gcomm_uuid';

+------------------+--------------------------------------+
| Variable_name    | Value                                |
+------------------+--------------------------------------+
| wsrep_gcomm_uuid | 7e729708-605f-11e5-8ddd-8319a704b8c4 |
+------------------+--------------------------------------+
```

Example Value	Location	Introduced	Deprecated
7e729708-605f-11e5-8ddd-8319a704b8c4	Galera	1	

wsrep_incoming_addresses

Comma-separated list of incoming server addresses in the cluster component.

```
SHOW STATUS LIKE 'wsrep_incoming_addresses';

+--------------------------+---------------------------------------+
| Variable_name            | Value                                 |
+--------------------------+---------------------------------------+
| wsrep_incoming_addresses | 10.0.0.1:3306,10.0.02:3306,undefined |
+--------------------------+---------------------------------------+
```

Example Value	Location	Introduced	Deprecated
10.0.0.1:3306, 10.0.0.2:3306, undefined	Galera		

wsrep_last_committed

The sequence number, or seqno, of the last committed transaction. See *wsrep API* (page 51).

```
SHOW STATUS LIKE 'wsrep_last_committed';

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

```
| Variable_name        | Value  |
+----------------------+--------+
| wsrep_last_committed | 409745 |
+----------------------+--------+
```

Note: **See Also**: For more information, see *wsrep API* (page 51).

Example Value	Location	Introduced	Deprecated
409745	Galera		

wsrep_local_bf_aborts

Total number of local transactions that were aborted by slave transactions while in execution.

```
SHOW STATUS LIKE 'wsrep_local_bf_aborts';

+-----------------------+-------+
| Variable_name         | Value |
+-----------------------+-------+
| wsrep_local_bf_aborts | 960   |
+-----------------------+-------+
```

Example Value	Location	Introduced	Deprecated
960	Galera		

wsrep_local_cached_downto

The lowest sequence number, or seqno, in the write-set cache (GCache).

```
SHOW STATUS LIKE 'wsrep_local_cached_downto';

+--------------------------+----------------------+
| Variable_name            | Value                |
+--------------------------+----------------------+
| wsrep_local_cached_downto | 18446744073709551615 |
+--------------------------+----------------------+
```

Example Value	Location	Introduced	Deprecated
18446744073709551615	Galera		

wsrep_local_cert_failures

Total number of local transactions that failed certification test.

```
SHOW STATUS LIKE 'wsrep_local_cert_failures';

+--------------------------+-------+
| Variable_name            | Value |
+--------------------------+-------+
| wsrep_local_cert_failures | 333   |
+--------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
333	Galera		

wsrep_local_commits

Total number of local transactions committed.

```
SHOW STATUS LIKE 'wsrep_local_commits';

+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| wsrep_local_commits | 14981 |
+---------------------+-------+
```

Example Value	Location	Introduced	Deprecated
14981	Galera		

wsrep_local_index

This node index in the cluster (base 0).

```
SHOW STATUS LIKE 'wsrep_local_index';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| wsrep_local_index | 1     |
+-------------------+-------+
```

Example Value	Location	Introduced	Deprecated
1	MySQL		

wsrep_local_recv_queue

Current (instantaneous) length of the recv queue.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_local_recv_queue | 0     |
+------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
0	Galera		

wsrep_local_recv_queue_avg

Recv queue length averaged over interval since the last `FLUSH STATUS` command. Values considerably larger than `0.0` mean that the node cannot apply write-sets as fast as they are received and will generate a lot of replication throttling.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue_avg';

+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| wsrep_local_recv_queue_avg | 3.348452 |
+----------------------------+----------+
```

Example Value	Location	Introduced	Deprecated
3.348452	Galera		

wsrep_local_recv_queue_max

The maximum length of the recv queue since the last FLUSH STATUS command.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue_max';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| wsrep_local_recv_queue_max | 10    |
+----------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
10	Galera		

wsrep_local_recv_queue_min

The minimum length of the recv queue since the last FLUSH STATUS command.

```
SHOW STATUS LIKE 'wsrep_local_recv_queue_min';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| wsrep_local_recev_queue_min | 0     |
+----------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
0	Galera		

wsrep_local_replays

Total number of transaction replays due to *asymmetric lock granularity*.

```
SHOW STATUS LIKE 'wsrep_local_replays';

+---------------------+-------+
| Variable_name       | Value |
+---------------------+-------+
| wsrep_lcoal_replays | 0     |
+---------------------+-------+
```

Example Value	Location	Introduced	Deprecated
0	Galera		

wsrep_local_send_queue

Current (instantaneous) length of the send queue.

```
SHOW STATUS LIKE 'wsrep_local_send_queue';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_local_send_queue | 1     |
+------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
1	Galera		

wsrep_local_send_queue_avg

Send queue length averaged over time since the last `FLUSH STATUS` command. Values considerably larger than 0.0 indicate replication throttling or network throughput issue.

```
SHOW STATUS LIKE 'wsrep_local_send_queue_avg';

+----------------------------+----------+
| Variable_name              | Value    |
+----------------------------+----------+
| wsrep_local_send_queue_avg | 0.145000 |
+----------------------------+----------+
```

Example Value	Location	Introduced	Deprecated
0.145000	Galera		

wsrep_local_send_queue_max

The maximum length of the send queue since the last `FLUSH STATUS` command.

```
SHOW STATUS LIKE 'wsrep_local_send_queue_max';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| wsrep_local_send_queue_max | 10    |
+----------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
10	Galera		

wsrep_local_send_queue_min

The minimum length of the send queue since the last `FLUSH STATUS` command.

```
SHOW STATUS LIKE 'wsrep_local_send_queue_min';

+----------------------------+-------+
| Variable_name              | Value |
+----------------------------+-------+
| wsrep_local_send_queue_min | 0     |
+----------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
0	Galera		

wsrep_local_state

Internal Galera Cluster FSM state number.

```
SHOW STATUS LIKE 'wsrep_local_state';

+-------------------+-------+
| Variable_name     | Value |
+-------------------+-------+
| wsrep_local_state | 4     |
+-------------------+-------+
```

Note: **See Also**: For more information on the possible node states, see *Node State Changes* (page 62).

Example Value	Location	Introduced	Deprecated
4	Galera		

wsrep_local_state_comment

Human-readable explanation of the state.

```
SHOW STATUS LIKE 'wsrep_local_state_comment';

+---------------------------+--------+
| Variable_name             | Value  |
+---------------------------+--------+
| wsrep_local_state_comment | Synced |
+---------------------------+--------+
```

Example Value	Location	Introduced	Deprecated
Synced	Galera		

wsrep_local_state_uuid

The UUID of the state stored on this node.

```
SHOW STATUS LIKE 'wsrep_local_state_uuid';

+-----------------------+--------------------------------------+
| Variable_name         | Value                                |
+-----------------------+--------------------------------------+
```

```
| wsrep_local_state_uuid | e2c9a15e-5485-11e0-0800-6bbb637e7211 |
+------------------------+--------------------------------------+
```

Note: **See Also**: For more information on the state UUID, see *wsrep API* (page 51).

Example Value	Location	Introduced	Deprecated
e2c9a15e-5385-11e0- 0800-6bbb637e7211	Galera		

wsrep_protocol_version

The version of the wsrep Protocol used.

```
SHOW STATUS LIKE 'wsrep_protocol_version';

+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| wsrep_protocol_version | 4     |
+-------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
4	Galera		

wsrep_provider_name

The name of the wsrep Provider.

```
SHOW STATUS LIKE 'wsrep_provider_name';

+----------------------+--------+
| Variable_name        | Value  |
+----------------------+--------+
| wsrep_provider_name | Galera |
+----------------------+--------+
```

Example Value	Location	Introduced	Deprecated
Galera	MySQL		

wsrep_provider_vendor

The name of the wsrep Provider vendor.

```
SHOW STATUS LIKE 'wsrep_provider_vendor';

+-----------------------+-------------------------------------+
| Variable_name         | Value                               |
+-----------------------+-------------------------------------+
| wsrep_provider_vendor | Codership Oy <info@codership.com> |
+-----------------------+-------------------------------------+
```

Example Value	Location	Introduced	Deprecated
Codership Oy <info@codership.com>	MySQL		

wsrep_provider_version

The name of the wsrep Provider version string.

```
SHOW STATUS LIKE 'wsrep_provider_version';

+------------------------+----------------------+
| Variable_name          | Value                |
+------------------------+----------------------+
| wsrep_provider_version | 25.3.5-wheezy(rXXXX) |
+------------------------+----------------------+
```

Example Value	Location	Introduced	Deprecated
25.3.5-wheezy(rXXXX)	MySQL		

wsrep_ready

Whether the server is ready to accept queries. If this status is OFF, almost all of the queries will fail with:

```
ERROR 1047 (08S01) Unknown Command
```

unless the wsrep_on session variable is set to 0.

```
SHOW STATUS LIKE 'wsrep_ready';

+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| wsrep_ready   | ON    |
+---------------+-------+
```

Example Value	Location	Introduced	Deprecated
ON	MySQL		

wsrep_received

Total number of write-sets received from other nodes.

```
SHOW STATUS LIKE 'wsrep_received';

+----------------+-------+
| Variable_name  | Value |
+----------------+-------+
| wsrep_received | 17831 |
+----------------+-------+
```

Example Value	Location	Introduced	Deprecated
17831	Galera		

wsrep_received_bytes

Total size of write-sets received from other nodes.

```
SHOW STATUS LIKE 'wsrep_received_bytes';

+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_received_bytes | 6637093 |
+----------------------+---------+
```

Example Value	Location	Introduced	Deprecated
6637093	Galera		

wsrep_repl_data_bytes

Total size of data replicated.

```
SHOW STATUS LIKE 'wsrep_repl_data_bytes';

+----------------------+---------+
| Variable_name        | Value   |
+----------------------+---------+
| wsrep_repl_data_bytes | 6526788 |
+----------------------+---------+
```

Example Value	Location	Introduced	Deprecated
6526788	Galera		

wsrep_repl_keys

Total number of keys replicated.

```
SHOW STATUS LIKE 'wsrep_repl_keys';

+-----------------+--------+
| Variable_name   | Value  |
+-----------------+--------+
| wsrep_repl_keys | 797399 |
+-----------------+--------+
```

Example Value	Location	Introduced	Deprecated
797399	Galera		

wsrep_repl_keys_bytes

Total size of keys replicated.

```
SHOW STATUS LIKE 'wsrep_repl_keys_bytes';

+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| wsrep_repl_keys_bytes | 11203721 |
+----------------------+----------+
```

Example Value	Location	Introduced	Deprecated
11203721	Galera		

wsrep_repl_other_bytes

Total size of other bits replicated.

```
SHOW STATUS LIKE 'wsrep_repl_other_bytes';

+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| wsrep_repl_other_bytes | 0     |
+------------------------+-------+
```

Example Value	Location	Introduced	Deprecated
0	Galera		

wsrep_replicated

Total number of write-sets replicated (sent to other nodes).

```
SHOW STATUS LIKE 'wsrep_replicated';

+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| wsrep_replicated | 16109 |
+------------------+-------+
```

Example Value	Location	Introduced	Deprecated
16109	Galera		

wsrep_replicated_bytes

Total size of write-sets replicated.

```
SHOW STATUS LIKE 'wsrep_replicated_bytes';

+------------------------+---------+
| Variable_name          | Value   |
+------------------------+---------+
| wsrep_replicated_bytes | 6526788 |
+------------------------+---------+
```

Example Value	Location	Introduced	Deprecated
6526788	Galera		

CHAPTER

FIFTYFIVE

GALERA SYSTEM TABLES

Starting with version 4 of Galera, three system tables were added to the `mysql` database related to Galera replication: `wsrep_cluster`, `wsrep_cluster_members`, and `wsrep_streaming_log`. These system tables may be used by database administrators to get a sense of the current activity of the nodes in a cluster.

To verify the version of Galera installed on your servers has these tables, execute the following SQL statement one of your servers using the `mysql` client or a similar client:

```
SHOW TABLES FROM mysql LIKE 'wsrep%';

+----------------------------+
| Tables_in_mysql (wsrep%)   |
+----------------------------+
| wsrep_cluster              |
| wsrep_cluster_members      |
| wsrep_streaming_log        |
+----------------------------+
```

Database administrators and clients with the access to the `mysql` database may read these tables, but they may not modify them: the database itself will make modifications, as needed.

Cluster View

One of the new Galera related system tables is the `wsrep_cluster` table. This new table, starting in version 4 of Galera, contains a current view of the cluster. That is to say, it stores the UUID of the cluster and some other identification information, as well as the cluster's capabilities.

To see the names of the columns in this table, either use the `DESCRIBE` statement or execute the following SQL statement from the `mysql` client on one of the nodes in the cluster:

```
SELECT COLUMN_NAME FROM information_schema.columns
WHERE table_schema='mysql'
AND table_name='wsrep_cluster';

+------------------+
| COLUMN_NAME      |
+------------------+
| cluster_uuid     |
| view_id          |
| view_seqno       |
| protocol_version |
| capabilities     |
+------------------+
```

The `cluster_uuid` contains the UUID of the cluster.

The `view_id` corresponds to the status value of the `wsrep_cluster_conf_id`, the number of cluster configuration changes which have occurred in the cluster. The `view_seqno` on the other hand, corresponds to Galera sequence number associated with the cluster view. The protocol version is the same value as contained in the `wsrep_protocol_version` variable. It's the protocol version of the MySQL-wsrep or the MariaDB wsrep patch. Last, the `capabilities` column contains the capabilities bitmask provided by the Galera library. It's metadata that will be needed to recover node state during crash recovery.

If you execute the following SQL statement from any node in a cluster, you can see the contents of this table:

```
SELECT * FROM mysql.wsrep_cluster \G

*************************** 1. row ***************************
    cluster_uuid: bd5fe1c3-7d80-11e9-8913-4f209d688a15
         view_id: 3
      view_seqno: 2956
protocol_version: 4
    capabilities: 184703
```

In the results here, you can see the cluster UUID. This can also be found by using the SQL statement, `SHOW STATUS` for the variable, `wsrep_local_state_uuid`.

Cluster Members

Another Galera related system tables is the `wsrep_cluster_members` table. This system table will provide the current membership of the cluster; it will contain a row for each node in the cluster. That is to say, each node in the cluster known to the node upon which the table is queried.

To see the names of columns in this table, either use the `DESCRIBE` statement or execute the following SQL statement from the `mysql` client on one of the nodes in the cluster:

```
SELECT COLUMN_NAME FROM information_schema.columns
WHERE table_schema='mysql'
AND table_name='wsrep_cluster_members';

+-----------------------+
| COLUMN_NAME           |
+-----------------------+
| node_uuid             |
| cluster_uuid          |
| node_name             |
| node_incoming_address |
+-----------------------+
```

The `node_uuid` records the UUID of each node in the cluster. The `cluster_uuid` is the UUID of the cluster for which the node belongs–the one on which the table has been queried. This is currently the same as what's contained in the `wsrep_cluster table`. The `node_name` contains the human readable name of each node, Last, the `node_incoming_address` stores the IP address and port on which each node is listening for client connections.

If you execute the following SQL statement from any node in a cluster, you can see the contents of this table:

```
SELECT * FROM mysql.wsrep_cluster_members ORDER BY node_name \G

*************************** 1. row ***************************
            node_uuid: e39d1774-7e2b-11e9-b5b2-7696f81d30fb
         cluster_uuid: bd5fe1c3-7d80-11e9-8913-4f209d688a15
```

```
            node_name: galera1
node_incoming_address: AUTO
*************************** 2. row ***************************
            node_uuid: eb8fc512-7e2b-11e9-bb74-3281cf207f60
         cluster_uuid: bd5fe1c3-7d80-11e9-8913-4f209d688a15
            node_name: galera2
node_incoming_address: AUTO
*************************** 3. row ***************************
            node_uuid: 2347a8ac-7e2c-11e9-b6f0-da90a2d0a563
         cluster_uuid: bd5fe1c3-7d80-11e9-8913-4f209d688a15
            node_name: galera3
node_incoming_address: AUTO
```

In the results of this example you can see that this cluster is composed of three nodes. The node UUIDs are unique for each node. Notice that the cluster UUID is the same for all three and corresponds to the related value found in the `wsrep_cluster` table shown in the example earlier. Each node has a unique name (e.g., galera1). They were named in the configuration file using the `wsrep_node_name` parameter. The incoming node address is set to `AUTO` for all of these nodes, but they can be set individual to specific nodes with the `wsrep-node-address` or the `bind-address` parameter in each node's configuration file.

Cluster Streaming Log

The last Galera related system tables is the `wsrep_streaming_log` table. This system table contains meta data and row events for ongoing streaming transactions, write set fragment per row.

The `node_uuid` column contains the node UUID of the hosting node for the transaction (i.e. node where the client is executing the transaction). The `trx_id` column stores the transaction identifier, whereas the `seqno` stores the sequence number of the write set fragment. Last, the `flags` columns records flags associated with the write set fragment, and `frag` contains the binary log replication events contained in the write set fragment.

To see the names of columns in this table, either use the `DESCRIBE` statement or execute the following SQL statement from the `mysql` client on one of the nodes in the cluster:

```
SELECT COLUMN_NAME FROM information_schema.columns
WHERE table_schema='mysql'
AND table_name='wsrep_streaming_log';

+-------------+
| COLUMN_NAME |
+-------------+
| node_uuid   |
| trx_id      |
| seqno       |
| flags       |
| frag        |
+-------------+
```

If you execute the following SQL statement from any node in a cluster, you can see the contents of this table:

```
SELECT * FROM mysql.wsrep_streaming_log \G
```

Typically, you won't see any results since it will contain entries only for transactions which have streaming replication enabled. For example:

```
CREATE TABLE table1 (col1 INT PRIMARY KEY);

SET SESSION wsrep_trx_fragment_size=1;

START TRANSACTION;

INSERT INTO table1 VALUES (100);

SELECT node_uuid, trx_id, seqno, flags
FROM mysql.wsrep_streaming_log;

+--------------------------------------+--------+-------+-------+
| node_uuid                            | trx_id | seqno | flags |
+--------------------------------------+--------+-------+-------+
| a006244a-7ed8-11e9-bf00-867215999c7c |     26 |     4 |     1 |
+--------------------------------------+--------+-------+-------+
```

You can see in the results from the example here that the node UUID matches that of the third node (i.e., `galera3`) in the results for the example above related to the `wsrep_cluster_members` table. In this example, the `frag` column was omitted from the `SELECT` statement since it contains binary characters that don't format well.

CHAPTER

FIFTYSIX

XTRABACKUP PARAMETERS

When using `xtrabackup-v2` as your *State Snapshot Transfer* method, you can fine tune how the script operates using the `[sst]` unit in the `my.cnf` configuration file.

```
[mysqld]
wsrep_sst_method=xtrabackup-v2

[sst]
compressor="gzip"
decompressor="gzip -dc"
rebuild=ON
compact=ON
encrypt=3
tkey="/path/to/key.pem"
tcert="/path/to/cert.pem"
tca="/path/to/ca.pem"
```

Bear in mind, some XtraBackup parameters require that you match the configuration on donor and joiner nodes, (as designated in the table below).

Option	Default	Match
compressor (page 309)		
cpat (page 310)	`0`	
decompressor (page 310)		
encrypt (page 311)	`0`	Yes
encrypt-algo (page 311)		
progress (page 311)		
rebuild (page 312)	`0`	
rlimit (page 312)		
sst_initial_timeout (page 312)	`100`	
sst_special_dirs (page 313)	`1`	
sockopt (page 313)		
streamfmt (page 313)	`xbstream`	Yes
tca (page 314)		
tcert (page 314)		
time (page 314)	`0`	
transferfmt (page 315)	`socat`	Yes

compressor

Defines the compression utility the donor node uses to compress the state transfer.

System Variable	*Name:*	`compressor`
	Match:	Yes
Permitted Values	*Type:*	String
	Default Value:	

This parameter defines whether the donor node performs compression on the state transfer stream. It also defines what compression utility it uses to perform the operation. You can use any compression utility which works on a stream, such as `gzip` or `pigz`. Given that the joiner node must decompress the state transfer before attempting to read it, you must match this parameter with the *decompressor* (page 310) parameter, using the appropriate flags for each.

```
compression="gzip"
```

compact

Defines whether the joiner node performs compaction when rebuilding indexes after applying a *State Snapshot Transfer*.

System Variable	*Name:*	`compact`
	Match:	No
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`

This parameter operates on the joiner node with the *rebuild* (page 312) parameter. When enabled, the node performs compaction when rebuilding indexes after applying a state transfer.

```
rebuild=ON
compact=ON
```

cpat

Defines what files to clean up from the datadir during state transfers.

System Variable	*Name:*	`cpat`
	Match:	No
Permitted Values	*Type:*	String
	Default Value:	

When the donor node begins a *State Snapshot Transfer*, it cleans up various files from the datadir. This ensures that the joiner node can cleanly apply the state transfer. With this parameter, you can define what files you want the node to delete before the state transfer.

```
cpat=".*glaera\.cache$\|.*sst_in_progress$\|.*grastate\.dat$\|.*\.err"
```

decompressor

Defines the decompression utility the joiner node uses to decompress the state transfer.

System Variable	*Name:*	`decompressor`
	Match:	No
Permitted Values	*Type:*	String
	Default Value:	

This parameter defines whether the joiner node performs decompression on the state transfer stream. It also defines what decompression utility it uses to perform the operation. You can use any compression utility which works on a

stream, such as `gzip` or `pigz`. Given that the donor node must compress the state transfer before sending it, you must match this parameter with the *compressor* (page 309) parameter, using the appropriate flags for each.

```
decompressor="gzip -dc"
```

encrypt

Defines whether the node uses SSL encryption for XtraBackup and what kind of encryption it uses.

System Variable	*Name:*	`encrypt`
	Match:	Yes
Permitted Values	*Type:*	Integer
	Default Value:	`0`

This parameter determines the type of SSL encryption the node uses when sending state transfers through xtrabackup. The recommended type is `2` when using the cluster over WAN.

Value	Description
`0`	No encryption.
`1`	The node encrypts State Snapshot Transfers through XtraBackup.
`2`	The node encrypts State Snapshot Transfers through OpenSSL, using Socat.
`3`	The node encrypts State Snapshot Transfers through the key and certificate files implemented for Galera Cluster.

```
encrypt=3
tkey="/path/to/key.pem"
tcert="/path/to/cert.pem"
tca="/path/to/ca.pem"
```

encrypt-algo

Defines the SSL encryption type the node uses for XtraBackup state transfers.

System Variable	*Name:*	`encrypt-algo`
	Match:	No
Permitted Values	*Type:*	Integer
	Default Value:	`0`

When using the *encrypt* (page 311) parameter in both the `[xtrabackup]` and `[sst]` units, there is a potential issue in it having different meanings according to the unit under which it occurs. That is, in `[xtrabackup]`, it turns encryption on while in `[sst]` it both turns it on as specifies the algorithm.

In the event that you need to clarify the meaning, this parameter allows you to define the encryption algorithm separately from turning encryption on. It is only read in the event that *encrypt* (page 311) is set to `1`

```
encrypt=1
encrypt-algo=3
```

progress

Defines whether where the node reports *State Snapshot Transfer* progress.

System Variable	*Name:*	`progress`
	Match:	No
Permitted Values	*Type:*	String
	Default Value:	
	Valid Values:	`1`
		/path/to/file

When you set this parameter, the node reports progress on XtraBackup progress in state transfers. If you set the value to `1`, the node makes these reports to the database server stderr. If you set the value to a file path, it writes the progress to that file.

Note: Bear in mind, that a `0` value is invalid. If you want to disable this parameter, delete or comment it out.

```
progress="/var/log/mysql/xtrabackup-progress.log"
```

rebuild

Defines whether the joiner node rebuilds indexes during a *State Snapshot Transfer*.

System Variable	*Name:*	`rebuild`
	Match:	No
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`

This parameter operates on the joiner node. When enabled, the node rebuilds indexes when applying the state transfer. Bear in mind, this operation is separate from compaction. Due to Bug #1192834, it is recommended that you use this parameter with *compact* (page 310).

```
rebuild=ON
compact=ON
```

rlimit

Defines the rate limit for the donor node.

System Variable	*Name:*	`rlimit`
	Match:	No
Permitted Values	*Type:*	Integer
	Default Value:	

This parameter allows you to definite the rate-limit the donor node. This allows you to keep state transfers from blocking regular cluster operations.

```
rlimit=300M
```

sst_initial_timeout

Defines the initial timeout to receive the first state transfer packet.

System Variable	*Name:*	`sst_initial_timeout`
	Match:	No
Permitted Values	*Type:*	Integer
	Default Value:	`100`

This parameter determines the initial timeout in seconds for the joiner to receive the first packet in a *State Snapshot Transfer*. This keeps the joiner node from hanging in the event that the donor node crashes while starting the operation.

```
sst_initial_timeout=130
```

sst_special_dirs

Defines whether the node uses special InnoDB home and log directories.

System Variable	*Name:*	`sst_special_dirs`
	Match:	No
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`

This parameter enables support for `innodb_data_home_dir` and `innodb_log_home_dir` parameters for XtraBackup. It requires that you define `innodb_data_home_dir` and `innodb_log_group_home_dir` in the `[mysqld]` unit.

```
[mysqld]
innodb_data_home_dir="/var/mysqld/innodb"
innodb_log_group_home_dir="/var/log/innodb"
wsrep_sst_method="xtrabackup-v2"

[sst]
sst_special_dirs=TRUE
```

sockopt

Defines socket options.

System Variable	*Name:*	`sockopt`
	Match:	No
Permitted Values	*Type:*	String
	Default Value:	

This parameter allows you to define one or more socket options for XtraBackup using the Socat transfer format.

streamfmt

Defines the stream formatting utility.

System Variable	*Name:*	`streamfmt`
	Match:	Yes
Permitted Values	*Type:*	String
	Default Value:	`xbstream`
	Valid Values:	`tar`
		`xbstream`

This parameter defines the utility the node uses to archive the node state before the transfer is sent and how to unarchive the state transfers that is receives. There are two methods available: `tar` and `xbstream`. Given that the receiving node needs to know how to read the stream, it is necessary that both nodes use the same values for this parameter.

The default and recommended utility is `xbstream` given that it supports encryption, compression, parallel streaming, incremental backups and compaction. `tar` does not support these features.

```
streamfmt='xbstream'
```

tca

Defines the Certificate Authority (CA) to use in SSL encryption.

System Variable	*Name:*	`tca`
	Match:	No
Permitted Values	*Type:*	path
	Default Value:	

This parameter defines the Certificate Authority (CA) file that the node uses with XtraBackup state transfers. In order to use SSL encryption with XtraBackup, you must configure the *transferfmt* (page 315) parameter to use `socat`.

Note: For more information on using Socat with encryption, see Securing Traffic between Two Socat Instances using SSL.

```
transferfmt="socat"
tca="/path/to/ca.pem"
```

tcert

Defines the certificate to use in SSL encryption.

System Variable	*Name:*	`tcert`
	Match:	No
Permitted Values	*Type:*	String
	Default Value:	

This parameter defines the SSL certificate file that the node uses with SSL encryption on XtraBackup state transfers. In order to use SSL encryption with XtraBackup, you must configure the *transferfmt* (page 315) parameter to use Socat.

Note: For more information on using Socat with encryption, see Securing Traffic between Two Socat Instances using SSL.

```
transferfmt="socat"
tcert="/path/to/cert.pem"
```

time

Defines whether XtraBackup instruments key stages in the backup and restore process for state transfers.

System Variable	*Name:*	`time`
	Match:	No
Permitted Values	*Type:*	Boolean
	Default Value:	`OFF`

This parameter instruments key stages of the backup and restore process for state transfers.

```
time=ON
```

transferfmt

Defines the transfer stream utility.

System Variable	*Name:*	`transferfmt`
	Match:	Yes
Permitted Values	*Type:*	String
	Default Value:	`socat`
	Valid Values:	`socat`
		`nc`

This parameter defines the utility that the node uses to format transfers sent from donor to joiner nodes. There are two methods supported: Socat and `nc`. Given that the receiving node needs to know how to interpret the transfer, it is necessary that both nodes use the same values for this parameter.

The default and recommended utility is Socat, given that it allows for socket options, such as transfer buffer size. For more information, see the socat Documentation.

```
transferfmt="socat"
```

CHAPTER

FIFTYSEVEN

GALERA LOAD BALANCER PARAMETERS

Galera Load Balancer provides simple TCP connection balancing developed with scalability and performance in mind. It draws on Pen for inspiration, but its functionality is limited to only balancing TCP connections.

It can be run either through the `service` command or the command-line interface of `glbd`. Configuration for Galera Load Balancer depends on which you use to run it.

Configuration Parameters

When Galera Load Balancer starts as a system service, it reads the `glbd.cfg` configuration file for default parameters you want to use. Only the *LISTEN_ADDR* (page 318) parameter is mandatory.

Parameter	Default Configuration
CONTROL_ADDR (page 317)	`127.0.0.1:8011`
CONTROL_FIFO (page 317)	`/var/run/glbd.fifo`
DEFAULT_TARGETS (page 318)	`127.0.0.1:80 10.0.1:80 10.0.0.2:80`
LISTEN_ADDR (page 318)	`8010`
MAX_CONN (page 318)	
OTHER_OPTIONS (page 318)	
THREADS (page 319)	`2`

CONTROL_ADDR

Defines the IP address and port for controlling connections.

Command-line Argument	*–control* (page 320)
Default Configuration	`127.0.0.1:8011`
Mandatory Parameter	No

This is an optional parameter. Use it to define the server used in controlling client connections. When using this parameter you must define the port. In the event that you do not define this parameter, Galera Load Balancer does not open the relevant socket.

```
CONTROL_ADDR="127.0.0.1:8011"
```

CONTROL_FIFO

Defines the path to the FIFO control file.

Command-line Argument	*–fifo* (page 321)
Default Configuration	`/var/run/glbd.fifo`
Mandatory Parameter	No

This is an optional parameter. It defines the path to the FIFO control file as is always opened. In the event that there is already a file at this path, Galera Load Balancer fails to start.

```
CONTROL_FIFO="/var/run/glbd.fifo"
```

DEFAULT_TARGETS

Defines the IP addresses and ports of the destination servers.

Default Configuration	`127.0.0.1:80 10.0.0.1:80 10.0.0.2:80:2`
Mandatory Parameter	No

This parameter defines that IP addresses that Galera Load Balancer uses as destination servers. Specifically, in this case the Galera Cluster nodes that it routes application traffic onto.

```
DEFAULT_TARGETS="192.168.1.1 192.168.1.2 192.168.1.3"
```

LISTEN_ADDR

Defines the IP address and port used for client connections.

Default Configuration	`8010`
Mandatory Parameter	Yes

This parameter defines the IP address and port that Galera Load Balancer listens on for incoming client connections. The IP address is optional, the port mandatory. In the event that you define a port without an IP address, Galera Load Balancer listens on that port for all available network interfaces.

```
LISTEN_ADDR="8010"
```

MAX_CONN

Defines the maximum allowed client connections.

Command-line Argument	*–max_conn* (page 322)
Mandatory Parameter	No

This parameter defines the maximum number of client connections that you want to allow to Galera Load Balancer. It modifies the system open files limit to accommodate at least this many connections, provided sufficient privileges. It is recommend that you define this parameter if you expect the number of client connections to exceed five hundred.

```
MAX_CONN="135"
```

This option defines the maximum number of client connections that you want allow to Galera Load Balancer. Bear in mind, that it can be operating system dependent.

OTHER_OPTIONS

Defines additional options that you want to pass to Galera Load Balancer.

Mandatory Parameter	No

This parameter defines various additional options that you would like to pass to Galera Load Balancer, such as a destination selection policy or Watchdog configurations. Use the same syntax as you would for the command-line arguments. For more information on the available options, see *Configuration Options* (page 319).

```
OTHER_OPTIONS="--random --watchdog exec:'mysql -utest -ptestpass' --discover"
```

THREADS

Defines the number of threads you want to use.

Command-line Argument	*–threads* (page 324)
Mandatory Parameter	No

This parameter allows you to define the number of threads (that is, connection pools), which you want to allow Galera Load Balancer to use. It is advisable that you have at least a few per CPU core.

```
THREADS="6"
```

Configuration Options

When Galera Load Balancer starts as a daemon process, through the `/sbin/glbd` command, it allows you to pass a number of command-line arguments to configure how it operates. It uses the following syntax:

```
/usr/local/sbin/glbd [OPTIONS] LISTEN_ADDRESS [DESTINATION_LIST]
```

In the event that you would like to set any of these options when you run Galera Load Balancer as a service, you can define them through the *OTHER_OPTIONS* (page 318) parameter.

Long Argument	Short	Type	Parameter
–control (page 320)	`-c`	IP address	*CONTROL_ADDR* (page 317)
–daemon (page 320)	`-d`	Boolean	
–defer-accept (page 320)	`-a`	Boolean	
–discover (page 320)	`-D`	Boolean	
–extra (page 321)	`-x`	Decimal	
–fifo (page 321)	`-f`	File Path	*CONTROL_FIFO* (page 317)
–interval (page 321)	`-i`	Decimal	
–keepalive (page 321)	`-K`	Boolean	
–latency (page 322)	`-L`	Integer	
–linger (page 322)	`-l`	Boolean	
–max_conn (page 322)	`-m`	Integer	*MAX_CONN* (page 318)
–nodelay (page 323)	`-n`	Boolean	
–random (page 323)	`-r`	Boolean	
–round (page 323)	`-b`	Boolean	
–single (page 323)	`-S`	Boolean	
–source (page 324)	`-s`	Boolean	
–threads (page 324)	`-t`	Integer	*THREADS* (page 319)
–top (page 324)	`-T`	Boolean	
–verbose (page 325)	`-v`	Boolean	
–watchdog (page 325)	`-w`	String	

--control

Defines the IP address and port for control connections.

Short Argument	-c
Syntax	--control [IP|Hostname:]port
Type	IP Address
Configuration Parameter	*CONTROL_ADDR* (page 317)

For more information on defining the controlling connections, see the *CONTROL_ADDR* (page 317) parameter.

```
# glbd --control 192.168.1.1:80 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--daemon

Defines whether you want Galera Load Balancer to run as a daemon process.

Short Argument	-d
Syntax	--daemon
Type	Boolean

This option defines whether you want to start `glbd` as a daemon process. That is, if you want it to run in the background, instead of claiming the current terminal session.

```
# glbd --daemon 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--defer-accept

Enables TCP deferred acceptance on the listening socket.

Short Argument	-a
Syntax	--defer-accept
Type	Boolean

Enabling `TCP_DEFER_ACCEPT` allows Galera Load Balancer to awaken only when data arrives on the listening socket. It is disabled by default.

```
# glbd --defer-accept 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--discover

Defines whether you want to use watchdog results to discover and set new destinations.

Short Argument	-D
Syntax	--discover
Type	Boolean

When you define the *–watchdog* (page 325) option, this option defines whether Galera Load Balancer uses the return value in discovering and setting new addresses for destination servers. For instance, after querying for the *wsrep_cluster_address* (page 237) parameter.

```
# glbd --discover -w exec:"mysql.sh -utest -ptestpass" 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--extra

Defines whether you want to perform an extra destination poll on connection attempts.

Short Argument	-x
Syntax	--extra D.DDD
Type	Decimal

This option defines whether and when you want Galera Load Balancer to perform an additional destination poll on connection attempts. The given value indicates how many seconds after the previous poll that you want it to run the extra poll. By default, the extra polling feature is disabled.

```
# glbd --extra 1.35 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--fifo

Defines the path to the FIFO control file.

Short Argument	-f
Syntax	--fifo /path/to/glbd.fifo
Type	File Path
Configuration Parameter	*CONTROL_FIFO* (page 317)

For more information on using FIFO control files, see the *CONTROL_FIFO* (page 317) parameter.

```
# glbd --fifo /var/run/glbd.fifo 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--interval

Defines how often to probe destinations for liveliness.

Short Argument	-i
Syntax	--interval D.DDD
Type	Decimal

This option defines how often Galera Load Balancer checks destination servers for liveliness. It uses values given in seconds. By default, it checks every second.

```
# glbd --interval 2.013 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--keepalive

Defines whether you want to disable the SO_KEEPALIVE socket option on server-side sockets.

Short Argument	-K
Syntax	--keepalive
Type	Boolean

Linux systems feature the socket option SO_KEEPALIVE, which causes the server to send packets to a remote system in order to main the client connection with the destination server. This option allows you to disable SO_KEEPALIVE on server-side sockets. It allows SO_KEEPALIVE by default.

```
# glbd --keepalive 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--latency

Defines the number of samples to take in calculating latency for watchdog.

Short Argument	-L
Syntax	--latency N
Type	Integer

When the Watchdog module tests a destination server to calculate latency, it sends a number of packets through to measure its responsiveness. This option configures how many packets it sends in sampling latency.

```
# glbd --latency 25 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--linger

Defines whether Galera Load Balancer disables sockets lingering after they are closed.

Short Argument	-l
Syntax	--linger
Type	Boolean

When Galera Load Balancer sends the close() command, occasionally sockets linger in a TIME_WAIT state. This options defines whether or not you want Galera Load Balancer to disable lingering sockets.

```
# glbd --linger 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--max_conn

Defines the maximum allowed client connections.

Short Argument	-m
Syntax	--max_conn N
Type	Integer

For more information on defining the maximum client connections, see the *MAX_CONN* (page 318) parameter.

```
# glbd --max_conn 125 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--nodelay

Defines whether it disables the TCP no-delay socket option.

Short Argument	-n
Syntax	--nodelay
Type	Boolean

Under normal operation, TCP connections automatically concatenate small packets into larger frames through the Nagle algorithm. In the event that you want Galera Load Balancer to disable this feature, this option causes it to open TCP connections with the TCP_NODELAY feature.

```
# glbd --nodelay 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--random

Defines the destination selection policy as Random.

Short Argument	-r
Syntax	--random
Type	Boolean

The destination selection policy determines how Galera Load Balancer determines which servers to route traffic to. When you set the policy to Random, it randomly chooses a destination from the pool of available servers. You can enable this feature by default through the *OTHER_OPTIONS* (page 318) parameter.

For more information on other policies, see *Destination Selection Policies* (page 138).

```
# glbd --random 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--round

Defines the destination selection policy as Round Robin.

Short Argument	-b
Syntax	--round
Type	Boolean

The destination selection policy determines how Galera Load Balancer determines which servers to route traffic to. When you set the policy to Round Robin, it directs new connections to the next server in a circular order list. You can enable this feature by default through the *OTHER_OPTIONS* (page 318) parameter.

For more information on other policies, see *Destination Selection Policies* (page 138).

```
# glbd --round 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--single

Defines the destination selection policy as Single.

Short Argument	-S
Syntax	--single
Type	Boolean

The destination selection policy determines how Galera Load Balancer determines which servers to route traffic to.

When you set the policy to Single, all connections route to the server with the highest weight value. You can enable this by default through the *OTHER_OPTIONS* (page 318) parameter.

```
# glbd --single 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--source

Defines the destination selection policy as Source Tracking.

Short Argument	-s
Syntax	--source
Type	Boolean

The destination selection policy determines how Galera Load Balancer determines which servers to route traffic to. When you set the policy to Source Tracking, connections that originate from one address are routed to the same destination. That is, you can ensure that certain IP addresses always route to the same destination server. You can enable this by default through the *OTHER_OPTIONS* (page 318) parameter.

Bear in mind, there are some limitations to this selection policy. When the destination list changes, the destination choice for new connections changes as well, while established connections remain in place. Additionally, when a destination is marked as unavailable, all connections that would route to it fail over to another, randomly chosen destination. When the original target becomes available again, routing to it for new connections resumes. In other words, Source Tracking works best with short-lived connections.

For more information on other policies, see *Destination Selection Policies* (page 138).

```
# glbd --source 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--threads

Defines the number of threads that you want to use.

Short Argument	-t
Syntax	--threads N
Type	Integer

For more information on threading in Galera Load Balancer, see *THREADS* (page 319).

```
# glbd --threads 6 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--top

Enables balancing to top weights only.

Short Argument	-T
Syntax	--top
Type	Boolean

This option restricts all balancing policies to a subset of destination servers with the top weight. For instance, if you have servers with weights 1, 2 and 3, balancing occurs only on servers with weight 3, while they remain available.

```
# glbd --top 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--verbose

Defines whether you want Galera Load Balancer to run as verbose.

Short Argument	-v
Syntax	--verbose
Type	Boolean

This option enables verbose output for Galera Load Balancer, which you may find useful for debugging purposes.

```
# glbd --verbose 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

--watchdog

Defines specifications for watchdog operations.

Short Argument	-w
Syntax	--watchdog SPEC_STR
Type	String

Under normal operation, Galera Load Balancer checks destination availability by attempting to establish a TCP connection to the server. For most use cases, this is insufficient. If you want to establish a connection with web server, you need to know if it is able to serve web pages. If you want to establish a connection with a database server, you need to know if it is able to execute queries. TCP connections don't provide that kind of information.

The Watchdog module implements asynchronous monitoring of destination servers through back-ends designed to service availability. This option allows you to enable it by defining the back-end ID string, optionally followed by a colon and the configuration options.

```
# glbd -w exec:"mysql.sh -utest -ptestpass" 3306 \
      192.168.1.1 192.168.1.2 192.168.1.3
```

This initializes the `exec` back-end to execute external programs. It runs the `mysql.sh` script on each destination server in order to determine it's availability. You can find the `mysql.sh` in the Galera Load Balancer build directory, under `files/`.

Note: The Watchdog module remains a work in progress. Neither its functionality nor terminology is final.

CHAPTER

FIFTYEIGHT

VERSIONING INFORMATION

Galera Cluster for MySQL is available in binary software packages for several different Linux distributions, as well as in source code for other distributions and other Unix-like operating systems, such as FreeBSD and Solaris.

For Linux distributions, binary packages in 32-bit and 64-bit for both the MySQL database server with the wsrep API patch and the *Galera Replication Plugin* are available from the Codership Repository. These include support for:

- Red Hat Enterprise Linux
- Fedora
- CentOS
- SUSE Linux Enterprise Server
- openSUSE
- Debian
- Ubuntu

By installing and configuring the Codership Repository on any of these systems, you can install and update Galera Cluster for MySQL through your package manager. In the event that you use a distribution of Linux that is not supported, or if you use another Unix-like operating system, source files are available on GitHub, at:

- MySQL Server with the wsrep API patch.
- Galera Replication Plugin.
- glb, the Galera Load Balancer.

For users of FreeBSD and similar operating systems, the Galera Replication Plugin is also available in ports, at `/usr/ports/databases/galera`, which corrects for certain compatibility issues with Linux dependencies.

Note: For more information on the installation process, see *Installation* (page 11).

Release Numbering Schemes

Software packages for Galera Cluster have their own release numbering schemas. There are two schemas to consider in version numbering:

- **Galera wsrep Provider** Also, referred to as the *Galera Replication Plugin*. The wsrep Provider uses the following versioning schema: `<wsrep API main version>.<Galera version>`. For example, release 24.2.4 indicates wsrep API version 24.x.x with Galera wsrep Provider version 2.4.

- **MySQL Server with wsrep API patch** The second versioning schema relates to the database server. Here, the MySQL server uses the following versioning schema `<MySQL server version>-<wsrep API version>`. For example, release 5.5.29-23.7.3 indicates a MySQL database server in 5.5.29 with wsrep API version 23.7.3.

For instances of Galera Cluster that use the MariaDB or Percona XtraDB database servers, consult their respective documentation for version and release information.

Third-party Implementations of Galera Cluster

In addition to the Galera Cluster for MySQL, the reference implementation from Codership Oy, there are two third party implementations of Galera Cluster. These are,

- Percona XtraDB Cluster is a high-availability and high-scalability solution for MySQL users. Percona XtraDB CLuster integrates Percona XtraDB Server with the Galera library of high-availability solutions in a single product package.
- MariaDB Galera Cluster uses the Galera library for the replication implementation. To interface with the Galera Replication Plugin, MariaDB is enhanced to support the replication API definition in the wsrep API project. Additionally, releases of MariaDB Server from version 10.1 on are packaged with Galera Cluster.

  For more information, see What is MariaDB Galera Cluster.

CHAPTER

FIFTYNINE

LEGAL NOTICE

CHAPTER

SIXTY

GLOSSARY

Galera Arbitrator An external process that functions as an additional node in certain cluster operations, such as quorum calculations and generating consistent application state snapshots.

For example, consider a situation where your cluster becomes partitioned due to a loss of network connectivity that results in two components of equal size. Each component initiates quorum calculations to determine which should remain the *Primary Component* and which should become a non-operational component. If the components are of equal size, it risks a split-brain condition. Galera Arbitrator provides an addition vote in the quorum calculation, so that one component registers as larger than the other. The larger component then remains the Primary Component.

Unlike the main `mysqld` process, `garbd` doesn't generate replication events of its own and doesn't store replication data. It does, however, acknowledge all replication events. Furthermore, you can route replication through Galera Arbitrator, such as when generating a consistent application state snapshot for backups.

Note: **See Also**: For more information, see *Galera Arbitrator* (page 117) and *Backing Up Cluster Data* (page 121).

Galera Replication Plugin Galera Replication Plugin is a general purpose replication plugin for any transactional system. It can be used to create a synchronous multi-master replication solution to achieve high availability and scale-out.

Note: **See Also**: See *Galera Replication Plugin* (page 52) for more details.

GCache See *Write-set Cache*.

Global Transaction ID To keep the state identical on all nodes, the *wsrep API* uses global transaction IDs (GTID), which are used to identify the state change and to identify the state itself by the ID of the last state change

The GTID consists of a state UUID, which uniquely identifies the state and the sequence of changes it undergoes, and an ordinal sequence number (seqno, a 64-bit signed integer) to denote the position of the change in the sequence.

Note: **See Also**: For more information on Global Transaction ID's, see *wsrep API* (page 51).

Incremental State Transfer In an Incremental State Transfer (IST) a node only receives the missing write-sets and catchs up with the group by replaying them. See also the definition for State Snapshot Transfer (SST).

Note: **See Also**: For more information on IST's, see *Incremental State Transfer (IST)* (page 58).

IST See *Incremental State Transfer*.

Logical State Transfer Method This is a type of back-end state transfer method that operates through the database server (e.g., `mysqldump`).

Note: **See Also**: For more information, see *Logical State Snapshot* (page 81).

Physical State Transfer Method This is another type of back-end state transfer method, but it operates on the physical media in the datadir (e.g., `rsync` and `xtrabackup`).

Note: **See Also**: For more information, see *Physical State Snapshot* (page 83).

Primary Component In addition to single-node failures, the cluster may be split into several components due to network failure. In such a situation, only one of the components can continue to modify the database state to avoid history divergence. This component is called the Primary Component (PC).

Note: **See Also**: For more information on the Primary Component, see *Weighted Quorum* (page 67).

Rolling Schema Upgrade The rolling schema upgrade is a DDL processing method in which the DDL will only be processed locally on the node. The node is desynchronized from the cluster for the duration of the DDL processing in a way that it doesn't block the other nodes. When the DDL processing is complete, the node applies the delayed replication events and synchronizes with the cluster.

Note: **See Also**: For more information, see *Rolling Schema Upgrade* (page 94).

RSU See *Rolling Schema Upgrade*.

seqno See *Sequence Number*.

Sequence Number This is a 64-bit signed integer that the node uses to denote the position of a given transaction in the sequence. The seqno is second component to the *Global Transaction ID*.

SST See *State Snapshot Transfer*.

State Snapshot Transfer State Snapshot Transfer refers to a full data copy from one cluster node (i.e., a donor) to the joining node (i.e., a joiner). See also the definition for Incremental State Transfer (IST).

Note: **See Also**: For more information, see *State Snapshot Transfer (SST)* (page 57).

State UUID Unique identifier for the state of a node and the sequence of changes it undergoes. It's the first component of the *Global Transaction ID*.

Streaming Replication This provides an alternative replication method for handling large or long-running write transactions. It's a new feature in version 4.0 of Galera Cluster. In older versions, the feature is unsupported.

Under normal operation, the node performs all replication and certification operations when the transaction commits. With large transactions this can result in conflicts if smaller transactions are committed first. With Streaming Replication, the node breaks the transaction into fragments, then certifies and replicates them to all nodes while the transaction is still in progress. Once certified, a fragment can no longer be aborted by a conflicting transaction.

Note: For more information see *Streaming Replication* (page 73) and *Using Streaming Replication* (page 115).

TOI See *Total Order Isolation*.

Total Order Isolation By default, DDL statements are processed by using the Total Order Isolation (TOI) method. In TOI, the query is replicated to the nodes in a statement form before executing on the master. The query waits for all preceding transactions to commit and then gets executed in isolation on all nodes, simultaneously.

Note: **See Also**: For more information, see *Total Order Isolation* (page 93).

write-set Transaction commits the node sends to and receives from the cluster.

Write-set Cache Galera stores write-sets in a special cache called, Write-set Cache (GCache). GCache is a memory allocator for write-sets. Its primary purpose is to minimize the write set footprint on the RAM.

Note: **See Also**: For more information, see *Write-set Cache (GCache)* (page 58).

wsrep API The wsrep API is a generic replication plugin interface for databases. The API defines a set of application callbacks and replication plugin calls.

Note: **See Also**: For more information, see *wsrep API* (page 51).

- genindex
- search

INDEX

F

G

I

R

S

T

V

W