

Migrating to MySQL 5.7

The Database Upgrade Guide



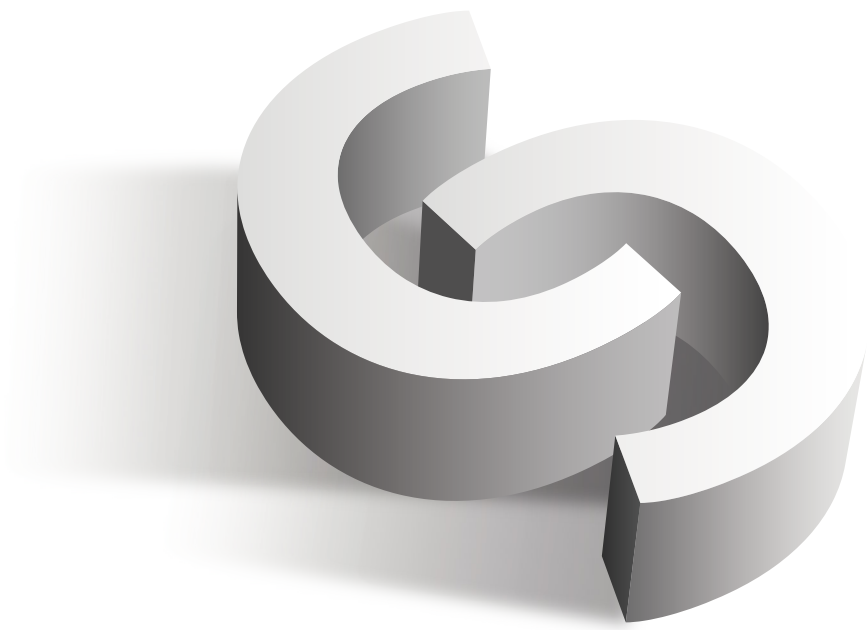




Table of Contents

1. Introduction	4
2. Changes between MySQL 5.6 and MySQL 5.7	5
2.1. Information schema changes	5
2.2. Systemd introduction to RPM-based distros	6
2.3. SQL modes	6
2.4. Authentication changes	6
2.5. Changes in InnoDB	7
2.6. Other changes introduced in MySQL 5.7	8
3. Overview of test environment	9
4. Pre-upgrade testing	10
4.1. First step - build a test environment	10
4.2. Collect data for regression tests	16
4.3. Regression tests using pt-upgrade	17
4.4. Regression tests of application	20
4.5. Bring back the node into replication	21
5. Upgrade	24
5.1. Slave upgrade process	24
5.2. Switchover process and upgrade of the old master	25
6. Graceful upgrade process using ProxySQL	27
6.1. Installation of ProxySQL	27
6.2. Configuring ProxySQL for graceful switchover with the ClusterControl	27
7. About Severalnines	35
8. Related resources from Severalnines	36



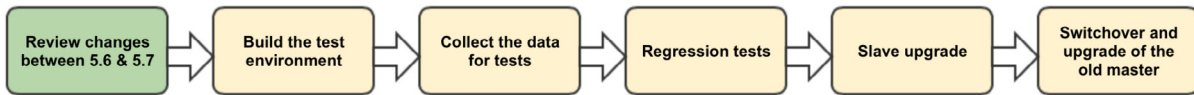
Introduction

MySQL 5.7 has been GA since October 2015. At the time of writing, it is still a very new release. But more and more companies are looking into upgrading, as it has a list of great new features. Schema changes can be performed with less downtime, with more online configuration options. Multi-source and parallel replication improvements make replication more flexible and scalable. Native support for JSON data type allows for storage, search and manipulation of schema-less data.

An upgrade, however, is a complex process - no matter which major MySQL version you are upgrading to. There are a few things you need to keep in mind when planning this, such as important changes between versions 5.6 and 5.7 as well as detailed testing that needs to precede any upgrade process. This is especially important if you would like to maintain availability for the duration of the upgrade.

Upgrading to a new major version involves risk, and it is important to plan the whole process carefully. In this document, we'll look at the important new changes in 5.7 and show you how to plan the test process. Then, we'll look at how to do a live system upgrade without downtime. For those who want to avoid connection failures during slave restarts and switchover, we'll go even further and show you how to leverage ProxySQL to achieve a graceful upgrade process.

Changes between MySQL 5.6 and MySQL 5.7



To upgrade to 5.7, you would need to be on 5.6. The officially supported upgrade process disallows upgrading from a version that is older than 5.6. So if you are on MySQL 5.5, you would first upgrade to 5.6 and then to 5.7. In MySQL 5.6 and earlier versions, dump/reload was the recommended way of performing an upgrade. But starting from MySQL 5.7, binary in-place upgrade is supported on the same level as logical dump/reload. This change has huge impact on how the upgrade is performed. Instead of dumping the data, which takes a lot of time, we can just upgrade RPMs or DEB binary packages.

No matter which way the upgrade is performed, we first need to go over any possible incompatibilities between 5.6 and 5.7. This is an important step that you should never skip.

2.1. Information schema changes

One of the incompatibilities between MySQL 5.6 and 5.7 is the fact that some of the views which were stored in the information_schema are not supported in MySQL 5.7. Examples include global_status, session_status, global_variables and session_variables. When running a query against them, you'll end up with the following error:

```
1 | mysql> select * from information_schema.global_variables;
2 | ERROR 3167 (HY000): The 'INFORMATION_SCHEMA.GLOBAL_VARIABLES' feature is disabled; see the documentation for 'show_compatibility_56'
```

You can restore original behavior by setting show_compatibility_56 to '1' but it's time to change the queries and use the performance_schema instead:

```
1 | mysql> select count(*) from performance_schema.global_variables;
2 | +-----+
3 | | count(*) |
4 | +-----+
5 | |      496 |
6 | +-----+
7 | 1 row in set (0.00 sec)
```

This issue can hit monitoring and trending systems which may use those queries to collect MySQL metrics.

2.2. Systemd introduction to RPM-based distros

MySQL 5.7 uses systemd init processes for startup and shutdown. This may cause some troubles in how you pass variables to MySQL via the command line. We do not want to go into details here, but we'd like to share one of the not-that-well-documented gotchas. By default, on Centos 7, MySQL is executed using this line:

```
1 | ExecStart=/usr/sbin/mysqld --daemonize --pid-file=/var/run/  
  | mysqld/mysqld.pid $MYSQLD_OPTS
```

In general, if you want to override any of the settings located (by default) in `/usr/lib/systemd/system/mysqld.service` you need to create following file:

`/etc/systemd/system/mysqld.service.d/override.conf`

and pass any overrides there. Exception is the ExecStart directive. It cannot be overridden by adding it to the override.conf file - you'll end up with an error saying you can't have two ExecStart directives. It is possible to do that, but you need to first set it to an empty value and then set it again to the desired setting. Below is an override.conf which changes location for the MySQL pid file. Note that we override two directives here: PIDFile and ExecStart. With PIDFile, it is enough to just add a directive and set it to whatever you want. With ExecStart, we had to put it there twice.

```
1 | [Service]  
2 | PIDFile=/var/lib/mysql/mysql.pid  
3 | ExecStart=  
4 | ExecStart=/usr/sbin/mysqld --daemonize --pid-file=/var/lib/  
  | mysql/mysql.pid $MYSQLD_OPTS
```

2.3. SQL modes

With MySQL 5.7, STRICT_TRANS_TABLES mode is used by default. This makes MySQL behavior much less forgiving when it comes to handling invalid data like zeroed date or skipping column in INSERT when column doesn't have an explicit DEFAULT value. This change can significantly impact applications which do not stick to 'good practices'. You can learn more about this setting in the [MySQL documentation](#).

What is also important to keep in mind is that SQL modes can be changed. Therefore it is possible that you create a table with, let's say, DEFAULT 0 for a DATETIME column, change the SQL mode to more strict and then see how INSERTs fail to execute against this table. In fact, sql_mode can be set dynamically on session level, so such behavior is not only related to databases upgraded from older MySQL versions.

Changing from non-GTID to GTID requires reconfiguration of all the nodes in your topology. You can prepare all your replica nodes to enable GTID, however as the master node is the one that generates the transactions, the master should be reconfigured before the GTID becomes effective. This means you will certainly have downtime for all nodes.

2.4. Authentication changes

A significant number of changes were introduced around how MySQL stores authentication data, and how it implements authentication. Some of them were made in the mysql.user table. For example, the 'password' column has been removed and all

authentication data along with passwords, have been moved to the 'authentication_string' column. Another change is that in MySQL 5.7, the 'plugin' column has to be non-empty, otherwise the account will be disabled.

When upgrading from earlier versions, *mysql_upgrade* should be able to fix those problems. It examines all tables in all databases for incompatibilities with the current version of MySQL Server, you should not forget to execute it.

The pre-4.1 password format has (finally) been removed from MySQL, along with related configuration variables (*old_passwords*, *secure_auth*). Function `OLD_PASSWORD()` has also been removed. This is a step in a good direction, removing those unsecure passwords.

Last, but not least, in 5.7 MySQL introduced password expiration. This is a great way to achieve a better level of security - you can force periodic password change on users. Unfortunately, this can also have some undesired side effects after an upgrade. Password expiration data is stored in the `mysql.user` table, in 'password_lifetime' column. When you perform an upgrade, this column is set to 'NULL', which means there's no per-user setting in use. The thing is, MySQL introduced global setting: `default_password_lifetime`. By default it is set to '360', which means that all accounts on your newly upgraded MySQL 5.7 will expire after 360 days. This introduces a time-delayed bomb - in a year, your application won't be able to query your database. As a result you'll see errors like:

```
1 | mysql> select 1;
2 | ERROR 1820 (HY000): You must reset your password using ALTER
   | USER statement before executing this statement.
```

To avoid this problem, you have to ensure that you've altered every user with the desired password expiration settings. You can, for example, set it to x days:

```
1 | ALTER USER backupuser@localhost PASSWORD EXPIRE INTERVAL 10
   | DAY;
```

or disable the password expiration for a particular host:

```
1 | ALTER USER backupuser@localhost PASSWORD EXPIRE NEVER;
```

Of course, you can also change the global setting (`SET GLOBAL default_password_lifetime=0`) but this is not the best option to choose - ideally you want your passwords to be rotated every now and then, maybe just not for all of the users.

This behavior has changed in MySQL 5.7.11 - in that version 'default_password_lifetime' defaults to 0 which helps in solving the problem we've just described. Still, [it has been said this variable's default setting may change in the future](#). So we are going to propose sticking to the process we described as the best way to solve password expiration issue once and for all.

2.5. Changes in InnoDB

A couple of changes introduced in MySQL 5.7 affected InnoDB. In short, both redo log and undo log formats changed a little bit between MySQL 5.6 and MySQL 5.7. Some of the changes also affect earlier MySQL 5.7 versions. We are not going to discuss those changes in detail, what's enough to say is that you want to use `innodb_fast_`

shutdown=0 when stopping the previous MySQL version to ensure all data has been flushed correctly before you attempt the upgrade.

With MySQL 5.7, the default row format has changed to DYNAMIC. If you want to retain the previous (COMPACT) format as default one, you need to make changes in MySQL configuration (innodb_default_row_format).

2.6. Other changes introduced in MySQL 5.7

One of the remaining changes is the removal of YEAR(2) format - you have to convert such columns to YEAR(4) data type, and mysql_upgrade can handle this conversion for you.

If you make use of user-defined locks in your application, you want to modify it to play by the new rules introduced in MySQL 5.7. The most important change is that GET_LOCK() doesn't implicitly release the currently held lock. In the past, you could only hold one lock at a time - MySQL 5.7 allows to grab multiple locks.

Along with the implementation of derived_merge optimization, you might see some SQL incompatibility. In some cases of UPDATE or DELETE where you use the same table for both DML and the derived subquery, you may get an error as the derived query is merged in the outer query. You'll end up updating a table based on the select executed on the same table. In the past, the derived subquery was materialized which, de facto, made it a separate table from the one you've been updating. To restore this behavior, you can disable this optimization (i.e. on a session level) by running:

```
1 | SET optimizer_switch = 'derived_merge=off';
```

You can find more details about this issue in the MySQL documentation:

<https://dev.mysql.com/doc/refman/5.7/en/subquery-optimization.html#derived-table-optimization>

Last but not least, as usual with new MySQL releases, the list of reserved keywords has changed. You'll want to confirm it's not affecting your queries and table structure. More details in the MySQL documentation:

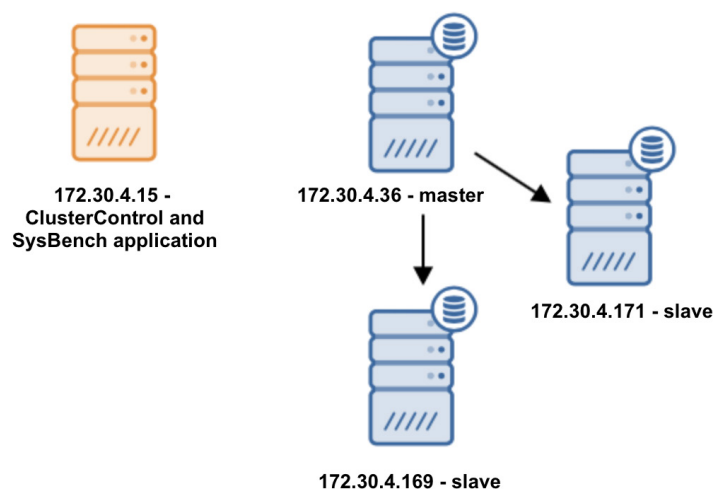
<https://dev.mysql.com/doc/refman/5.7/en/keywords.html>

As you can see, the changes affect a lot of areas and some databases may be significantly impacted - up to the point where it's not possible to operate on MySQL 5.7 without going through a process of finding workarounds for the parts in the application that are affected by the new MySQL behavior. That's why it is so important to perform detailed tests of your application before you proceed with the upgrade process. This is out of scope of this document, as each application is different and there's no way to give good generic suggestions on how to check your app.

Overview of test environment

In the previous chapter, we started our upgrade process by going through the incompatible changes which were introduced in MySQL 5.7. From now on, we'll do some more hands-on work. We'll set up a testing environment for MySQL 5.7 and do some preliminary testing.

Let's start with a short tour of our fairly typical MySQL replication setup. It consists of four hosts at the moment:



172.30.4.15 - ClusterControl node, colocated with our proxy, MaxScale and our "application" (sysbench)

172.30.4.36 - Percona Server 5.6 - master

172.30.4.169 - Percona Server 5.6 - slave

172.30.4.171 - Percona Server 5.6 - slave

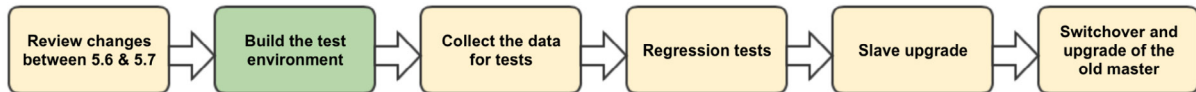
The whole setup was deployed using ClusterControl. As proxy, we use MaxScale and its RW connector - this is because our "application" does not support read-write split and MaxScale can handle it for us.

We use the following sysbench command to generate traffic. It's connected to the MaxScale proxy on the RW router (172.30.4.15:4008).

```
1 while true ; do sysbench --test=/root/sysbench/sysbench/
  tests/db/oltp.lua --num-threads=6 --max-requests=0 --max-
  time=0 --mysql-host=172.30.4.15 --mysql-user=sbtest
  --mysql-password=sbtest --mysql-port=4008 --oltp-tables-
  count=32 --report-interval=10 --oltp-skip-trx=on --oltp-ta-
  ble-size=1000000 run ; done
```

Pre-upgrade testing

4.1. First step - build a test environment



In order to perform tests, we need an upgraded database instance. Ideally, our test database contains actual production data. If you just use a subset of data from production, or if your data in test differs from the data in production, you may end up with results which may not be comparable to production.

Our approach will be as follows:

1. We are going to set up a new slave, running on MySQL 5.6. If you can afford to remove one of your slaves from rotation, you can reuse it.
2. Once the slave is up and running, we'll stop the slave thread to maintain a consistent state of the database.
3. We will then set up another host with a MySQL 5.6 instance, using the data from the stopped slave
4. Once the second host is up, we'll upgrade it to MySQL 5.7

We should have two database instances by now, one running MySQL 5.6, one running MySQL 5.7, both having the same dataset. This will be our starting point for performance regression tests. Of course, it goes without saying, both hosts have to have exactly the same hardware configuration, otherwise our results would not be meaningful.

For our purposes we've created a new EC2 instance with IP of 172.30.4.220. We will use ClusterControl to provision it but first we need to setup passwordless SSH communication between the ClusterControl node and this new host - this is required for ClusterControl to be able to connect and add this node to the cluster.

What we do is:

Check the SSH key for root user on ClusterControl node

```
1 root@ip-172-30-4-15:~# cat /root/.ssh/authorized_keys
2 ssh-rsa AAAAB3NzaC1yc2EAAAADAQABAAQDLPeKbHoF5tCJLUg16cWX-
  G1KBcUWJSQsmWPwzY80Mpc4xTgk7dhrZFrF+VH5njoHnGomIUwAzUFxRfeK-
  T7Ydx7fWLoHC27jTdeofSNMikz6GjJfMQf7tF/X2YhTHIXC7bSywgR43Mk-
  4jZN
3 ...
```

Remove `authorized_keys` on the new host, create new one with our SSH key, change access rights to 600:

```
1 root@ip-172-30-4-220:~# rm /root/.ssh/authorized_keys ; vim
  /root/.ssh/authorized_keys ; chmod 600 /root/.ssh/author-
  ized_keys
```

Once it's done, we should be able to connect to the server.

```
1 root@ip-172-30-4-15:~# ssh 172.30.4.220
2 The authenticity of host '172.30.4.220 (172.30.4.220)' can't
  be established.
3 ECDSA key fingerprint is 6e:80:23:e5:54:75:1f:28:56:94:b8:
  ae:fc:c7:e5:44.
4 Are you sure you want to continue connecting (yes/no)? yes
5 Warning: Permanently added '172.30.4.220' (ECDSA) to the
  list of known hosts.
6 Welcome to Ubuntu 14.04 LTS (GNU/Linux 3.13.0-24-generic
  x86_64)
7
8 * Documentation:  https://help.ubuntu.com/
9
10 System information as of Fri Jan 15 08:42:14 UTC 2016
11
12 System load:  0.16                Processes:            113
13 Usage of /:   10.5% of 7.74GB     Users logged in:    0
14 Memory usage: 2%                 IP address for eth0:
15 172.30.4.220
16 Swap usage:   0%
17
18 Graph this data and manage this system at:
19   https://landscape.canonical.com/
20
21 Get cloud support with Ubuntu Advantage Cloud Guest:
22   http://www.ubuntu.com/business/services/cloud
23
24 *** /dev/xvda1 should be checked for errors ***
25
26
27 The programs included with the Ubuntu system are free soft-
28 ware;
29 the exact distribution terms for each program are described
30 in the
31 individual files in /usr/share/doc/*/copyright.
32
33 Ubuntu comes with ABSOLUTELY NO WARRANTY, to the extent per-
34 mitted by
35 applicable law.
36
37 root@ip-172-30-4-220:~#
```

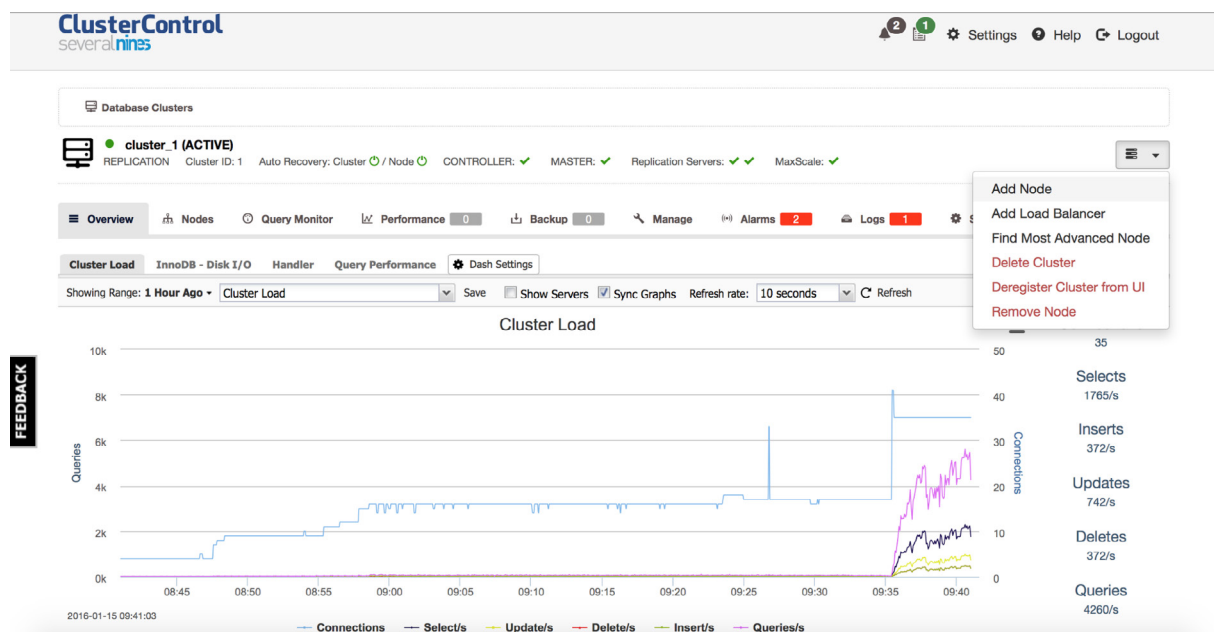
We also want to copy our ssh key to the new node:

```

1 root@ip-172-30-4-15:~# scp /root/.ssh/id_rsa 172.30.4.220:/
  root/.ssh/ ; ssh 172.30.4.220 'chmod 600 /root/.ssh/id_rsa'
2 id_rsa
  100% 1675      1.6KB/s   00:00

```

Once this is done, we should be fine to provision our new node as a slave to the existing cluster. To do that, we need to open the ClusterControl UI and navigate to our existing cluster. On the right hand side, there's a drop down menu from which we pick 'Add Node'.



Next step will be to file the necessary details for ClusterControl to create this new node:

The screenshot shows the 'Add Node' dialog box in the ClusterControl UI. The dialog is open, showing fields for Hostname (172.30.4.220), Configuration (my.cnf.gtid_replication), Install Software (Yes), Disable Firewall (Yes), Disable AppArmor/SELinux (checked), Include in Loadbalancer set (unchecked), and Do you want to delay the slave? (No). The dialog also includes a 'Software Package' section indicating it installs from the percona repository.

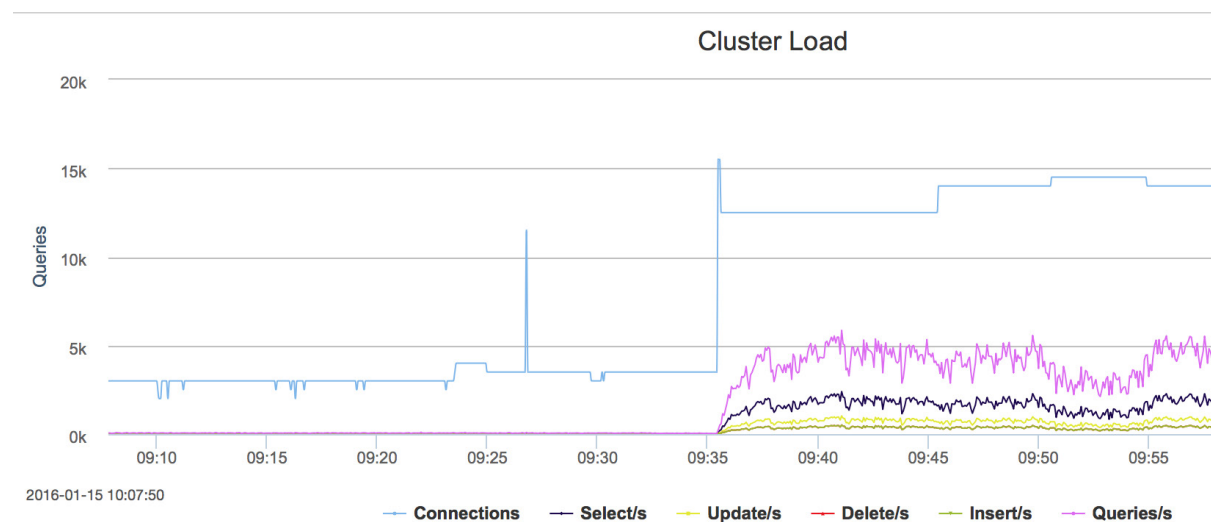
We are passing here the IP of our new node, we also definitely want ClusterControl to install the database software for us (you can see at the bottom it will be installed from the Percona repository). We do not want to add this node to existing load balancers, because we don't want to see production traffic hitting our test box. Once you start the

job, you can watch the progress in the 'Logs' section of the ClusterControl UI:

The screenshot shows the ClusterControl UI interface. The top navigation bar includes Overview, Nodes, Query Monitor, Performance, Backup, Manage, Alarms (2), Logs (0), and Settings. The left sidebar has a 'FEEDBACK' button and sections for Logs, Jobs, CMON Logs, Error Reports, and System Logs. The main content area displays a job titled 'Jobs in the Cluster' with a table of job details. The selected job is 'Add node started.' with a status of 'FINISHED' and a completion time of 15 Jan 2016 08:57:54. Below the table, a message log shows the following steps:

- 172.30.4.220: Installing software. Exit Code: 0
- 172.30.4.220: Setting up repositories. Exit Code: 0
- 172.30.4.220: Installing replication software, vendor percona, version 5.6, Exit Code: 0
- Using dataDir: '/var/lib/mysql'
- 172.30.4.220: un-installing existing mysql packages. Exit Code: 0

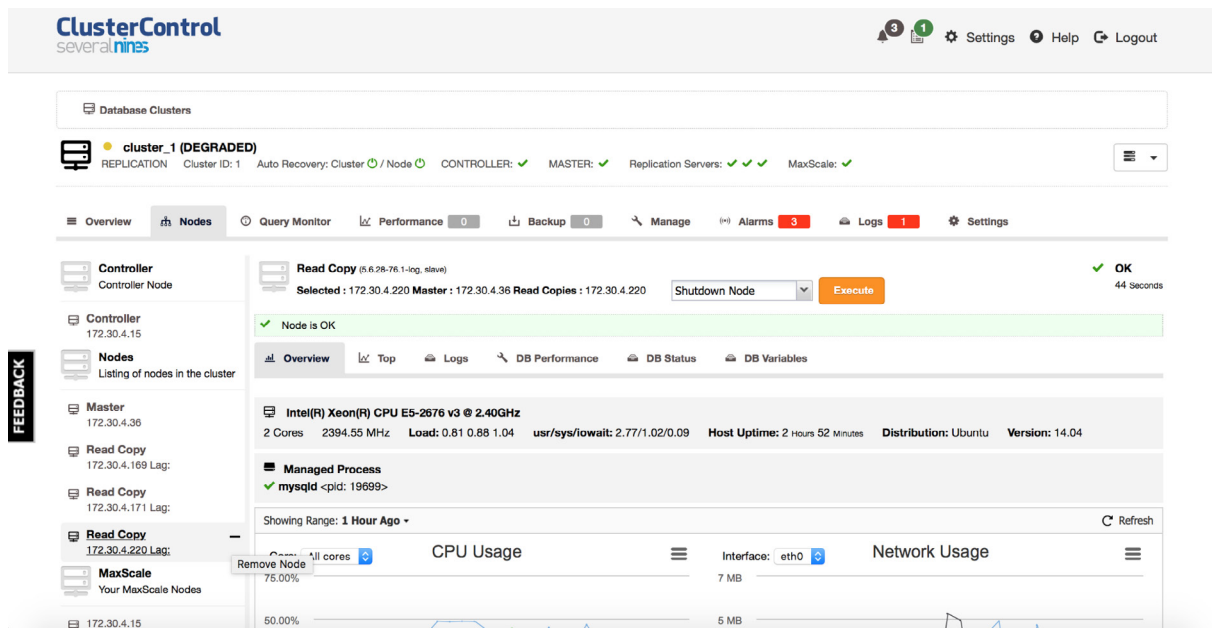
In this job, ClusterControl connects to the new node using the SSH connection we've just set, installs new software, in our case Percona Server 5.6, and configures it using the configuration template we selected in the 'Add Node' window. Once MySQL is ready, ClusterControl uses xtrabackup to stream data from the master node. This process takes some time, depending on your I/O performance, network throughput and size of the dataset. It also, unfortunately, impacts performance of the production master as Xtrabackup has to read a lot of data in order to transfer it to the new slave. The impact can be seen in the graph below - a dip was caused by the provisioning of our new node.



There are many ways to provision a slave. Some of them can cause less impact. It all depends on the exact environment you are working in. You could take a snapshot of slave's volume, you could stop one of the slaves (if you can afford that) and rsync data from it to the new slave.

Ok, by now, we have a new slave created and (eventually) synced with the master. Now it's time to deregister our 172.30.4.220 node from ClusterControl. We'll do it from the

ClusterControl UI to avoid automated recovery.



Once it's done, we want to make a copy of its data on some other host. We'll use another node, 172.30.4.236 for that. We'll set it up in the same way as we did with 172.30.4.220 - setup ssh keys. This time we are not going to use CC to provision that node (although we could have done that - provision using data from master and then replace it with data from our stopped slave), we'll do it manually. For that we need to setup Percona repositories.

```
1 root@ip-172-30-4-236:~# wget https://repo.percona.com/apt/
  percona-release_0.1-3.%(lsb_release -sc)_all.deb
2 root@ip-172-30-4-236:~# dpkg -i percona-release_0.1-3.%(lsb_
  release -sc)_all.deb
3 root@ip-172-30-4-236:~# apt-get update
```

and install Percona Server 5.6

```
1 root@ip-172-30-4-236:~# apt-get install percona-server-serv-
  er-5.6
```

To finalize setting up MySQL, we need to copy the my.cnf from our test slave:

```
1 root@ip-172-30-4-220:~# scp /etc/mysql/my.cnf 172.30.4.236:/
  etc/mysql/
```

and then edit it on the new node so IP's will point correctly:

```
1 root@ip-172-30-4-236:~# sed --in-place
  's/172.30.4.220/172.30.4.236/g' /etc/mysql/my.cnf
```

On both nodes we want to add 'skip_slave_start=1' directive to my.cnf - we don't want replication to jump in and change our data.

MySQL is ready, we still need to transfer data though. Again, it can be done in multiple

ways - EBS snapshots, xtrabackup, scp, rsync. This time, to make it more interesting, we'll use rsync to copy the dataset. You need to keep in mind that MySQL has to be stopped in order to copy the data correctly using rsync. First things first - we need to disable `innodb_fast_shutdown` on the host which we'll be upgrading to MySQL 5.7. This is related to incompatibilities in InnoDB which may impact our upgrade. We'll choose 172.30.4.220 for upgrade so we need to run on it:

```
1 | mysql> set global innodb_fast_shutdown=0;
```

We'll stop MySQL on both 'donor' node and new one, then we can start rsync'ing data using the `rsync-over-ssh`.

```
1 | root@ip-172-30-4-220:~# service mysql stop
2 | root@ip-172-30-4-236:~# service mysql stop
3 | root@ip-172-30-4-220:~# rsync -avz -e "ssh -o StrictHostKey-
  | Checking=no -o UserKnownHostsFile=/dev/null" --progress /
  | var/lib/mysql/ 172.30.4.236:/var/lib/mysql/
```

After a while (longer or shorter - it depends on your I/O and network performance and a size of your data), data should be transferred to the other node. We need to make sure that owner is set correctly on transferred files and then we can try to start MySQL and see if it works.

```
1 | root@ip-172-30-4-236:~# chown -R mysql:mysql /var/lib/mysql
2 | root@ip-172-30-4-236:~# service mysql start
```

When we are ready with two nodes, we can upgrade 172.30.4.220 to MySQL 5.7. First we need to setup repositories. We will follow the steps described on the MySQL website:

<https://dev.mysql.com/doc/mysql-apt-repo-quick-guide/en/>

```
1 | root@ip-172-30-4-220:~# wget http://dev.mysql.com/get/mysql-
  | apt-config_0.6.0-1_all.deb
2 | root@ip-172-30-4-220:~# dpkg -i mysql-apt-config_0.6.0-1_all.
  | deb
3 | root@ip-172-30-4-220:~# apt-get update
4 | root@ip-172-30-4-220:~# apt-get install mysql-server
```

One of the gotchas we've found while doing it is how MySQL 5.7 looks for some server variables. For example, to find pid file, it executes:

```
1 | my_print_defaults mysqld_safe | sed -n 's/--pid-file=//p'
```

In order for this command to work, you need to have a line like this:

```
1 | pid-file=/var/run/mysqld/mysql.pid
```

in `[mysqld_safe]` section of your `my.cnf`. If you do not have it or if you have it put in different (but correct) way like, let's say:

```
1 | pid_file=/var/run/mysqld/mysql.pid
```

MySQL 5.7 init script will not be able to start/stop the service correctly (i.e. MySQL will start but the init script won't detect it is running) and apt-get will fail.

Once MySQL 5.7 is installed and started, we need to execute 'mysql_upgrade' script, which is supposed to fix some of the known incompatibilities in schema and file formats.

```
1 | root@ip-172-30-4-220:~# mysql_upgrade -ppass
```

4.2. Collect data for regression tests



In the previous chapter, we accomplished the upgrade to MySQL 5.7. We also have a MySQL 5.6 node with the same dataset. This allows us to do some regression testing - we will use pt-upgrade, a tool from Percona Toolkit. The main idea behind this tool is to replay a set of queries on two different MySQL instances and compare the results. The tool is looking for performance regressions along with changes in the result set (different number of rows or even different row order). To be able to start tests, we need to have a set of queries to replay. One of the formats pt-upgrade understands is the MySQL slow query log but it can also understand binary logs, general log or even tcpdump data.

Collecting slow logs is the most obvious and easy way of gathering the data for pt-upgrade. Unfortunately, it doesn't go without performance impact. We've done some testing in the past and we shared results in one of our blog posts:

<http://severalnines.com/blog/become-mysql-dba-blog-series-query-tuning-process>

In short, enabling slow log can introduce transient slowdowns which make database performance and user experience not as stable as it should be.

Instead of using the slow query log, we can use data collected by tcpdump. It also has its impact on the system (as discussed in the blog post mentioned above), but in our case we don't have to execute it on our database - we should be able use it on our "application" server.

This choice has some repercussions. There will be some impact on this server. However, most of the time, you have a number of application servers - usually more than databases. Using tcpdump on one of the application hosts will impact less users than using it directly on a database server. When doing that, you need to remember that you will be able to collect only those queries which were executed by this particular server. If your infrastructure design is more complex (i.e. you have many 'types' of application servers, each type serving different traffic), you may need to collect the data on many hosts. Otherwise you'll not cover all of your queries - and this is the goal we are aiming at.

Another question you need to answer is - how long do you need to collect the data? It all depends on your workload. If your workload doesn't change in time, you probably don't need to spend too much time collecting. On the other side, you have cases when the workload varies during the day and you may need to collect multiple samples over the 24 hours. The goal here is to collect all queries you execute against your database. If you have an ETL process or batch load job which runs every now and then (i.e. on weekly/monthly basis), you want to ensure those queries will also be collected.

In our case, the "benchmark application" executes pretty much the same queries over

and over again so there's no need to collect data for too long. Unfortunately, we ran into an issue with MaxScale at the time of writing - for some reason data exchanged between proxy and the application cannot be parsed properly by pt-upgrade. So we resorted to a less ideal solution - run tcpdump on the proxy node and collect data exchanged between it and database servers.

We can use tcpdump to grab this traffic with the following line. Please note port 3306 - it will collect data exchanged by MaxScale and MySQL servers.

```
1 | root@ip-172-30-4-15:~# tcpdump -s 65535 -x -nn -q -tttt -i  
any port 3306 > mysql.tcp.txt
```

4.3. Regression tests using pt-upgrade



Once we've captured enough data (in our case it was ~5 minutes worth of data - as we said, sysbench repeats queries very often), we can start our pt-upgrade process. The first run will consist of running the queries against MySQL 5.6. Please note we used '--save-results' argument. In short, pt-upgrade allows you to either store your data as a baseline and then compare other nodes against this result. You can also pass two DSN's and run queries on both hosts at once. We'd suggest to always create baseline results and reuse them, though. Thanks to that, you will need to run queries just on the one host, making the process twice as fast. One run may take hours if your query log is large and you also never know how many times you'll have to rerun pt-upgrade - you may want to test, for example, different configuration tweaks.

Another important thing to remember - when running pt-upgrade, you need to ensure both databases are at the same state buffer pool-wise. You don't want to run queries when one of the databases is warm and has data in its caches - your results will not be correct. To avoid impact from cache, you should clear them before you start the pt-upgrade process:

```
1 | root@ip-172-30-4-236:~# service mysql restart ; echo "3" > /  
proc/sys/vm/drop_caches  
2 | root@ip-172-30-4-220:~# service mysql restart ; echo "3" > /  
proc/sys/vm/drop_caches
```

This should be enough to make sure our databases are in 'cold' state.

To ensure we test both best and worse case scenarios, we'll use two runs - first on a cold database and then we'll execute queries on a warm database - immediately after our first run, data should be in the buffer pool.

Yet another thing you want to keep in mind is network latency - even if you use the same source host for your pt-upgrade, it still may happen that one of the test servers is further away in terms of network communication. This may have an impact on the load. To avoid this, we'd suggest to execute pt-upgrade directly on the node, using socket - it should minimize the latency. Once we have results, we can copy the baseline directory to the 5.7 host and, again, use local socket to connect and test performance of the queries.

Our first run:

```

1  root@ip-172-30-4-236:~# ./pt-upgrade h=localhost,p=pass,u=-
   root,D=sbtest --type=tcpdump --save-results 56_results_cold/
   mysql.tcp.txt
2
3  #-----
   -----
4  # Logs
5  #-----
   -----
6
7  File: mysql.tcp.txt
8  Size: 1003693001
9
10 #-----
   -----
11 # Hosts
12 #-----
   -----
13
14 host1:
15
16     DSN:          h=localhost
17     hostname:    ip-172-30-4-236
18     MySQL:       Percona Server (GPL), Release 76.1, Revision
   5759e76 5.6.28
19
20 Saving results in 56_results_cold/
21 TCP session 172.30.4.15:45870 had errors, will save them in
   /tmp/pt-upgrade-errors.NvLv7YU
22 mysql.tcp.txt:  2% 21:34 remain
23
24 . . .
25 mysql.tcp.txt:  99% 00:05 remain
26
27 #-----
   -----
28 # Stats
29 #-----
   -----
30
31 failed_queries          0
32 not_query               2150
33 not_select              29380
34 queries_filtered       0
35 queries_no_diffs       0
36 queries_read           133104
37 queries_with_diffs     0
38 queries_with_errors    0
39 queries_written        101574

```

As you can see, pt-upgrade processed 133k queries, 101k queries were written in the results output. By default, pt-upgrade runs only SELECT queries - data on both of our nodes will not be affected. Of course, you may want to also run DML's but this will require restoring databases to their original state and makes the process more time-consuming.

Second warm run:

```
1 root@ip-172-30-4-236:~# ./pt-upgrade h=localhost,p=pass,u=-
  root,D=sbtest --type=tcpdump --save-results 56_results_warm/
  mysql.tcp.txt
```

Next, we need to copy both baseline directories to our 5.7 host and run pt-upgrade on it.

```
1 root@ip-172-30-4-236:~# scp -r 56_results_* 172.30.4.220:
2 root@ip-172-30-4-220:~# ./pt-upgrade 56_results_cold/ h=lo-
  calhost,p=pass,u=root,D=sbtest > 57.cold
3 root@ip-172-30-4-220:~# ./pt-upgrade 56_results_warm/ h=lo-
  calhost,p=pass,u=root,D=sbtest > 57.warm
```

In our case, warm workload was clean - no discrepancies and regressions have been found. Cold workload, on the other hand, reported some of the queries to be significantly slower on MySQL 5.7. For example:

```
1 #####
  #####
2 # Query class 558CAEF5F387E929
3 #####
  #####
4
5 Reporting class because there are 3 query diffs.
6
7 Total queries      3
8 Unique queries    3
9 Discarded queries  0
10
11 select c from sbtest? where id=?
12
13 ##
14 ## Query time diffs: 3
15 ##
16
17 -- 1.
18
19 0.000114 vs. 0.007351 seconds (64.5x increase)
20
21 SELECT c FROM sbtest9 WHERE id=496353
22
23 -- 2.
24
```

```

25 | 0.000113 vs. 0.009107 seconds (80.6x increase)
26 |
27 | SELECT c FROM sbtest18 WHERE id=495601
28 |
29 | -- 3.
30 |
31 | 0.000107 vs. 0.029717 seconds (277.7x increase)
32 |
33 | SELECT c FROM sbtest18 WHERE id=502477

```

Usually, it is good to confirm such results manually (well, not necessarily manually - automation is better, it's just important to compare results outside of pt-upgrade as well) - we've seen false positives in pt-upgrade results. In this case, regressions have been confirmed. Next step would be to try and identify what caused those regressions. It can be anything - query execution plan, some MySQL settings which need tuning. We checked the query execution plans as this is the most obvious and common reason for such discrepancies. All looked good and we are going to skip further checks because we found those regressions to only impact a cold database and we will strive to avoid running in production with a cold database.

4.4. Regression tests of application

Another level of database testing involves running a regular suite of acceptance tests. Every application should have a set of tests that are executed, for example, when a new release is getting close to go live. Such tests should cover as much of the functionality as possible - you want to test all processes that are used in the application. What needs to be tested, it depends on the application, of course. If you run a e-commerce website, you may want to check if the process of buying items works flawlessly. Can you add item to a cart? Can you check out? Is the order processed correctly? Is the payment processed correctly? You should see a pattern here - test everything, but at a minimum, test the core functionality that drives your business.

Such tests, most likely, require interaction between DBA's and developers and the required coordination can slow things down. That's why we've executed generic tests with the help of pt-upgrade - to catch the most obvious issues without needing to wait for the development team.

Our 'application' is pretty simple, therefore we are just going to point it to our MySQL 5.7 test instance and run it there. The exact command which we will execute is:

```

1 | root@ip-172-30-4-15:~# sysbench --test=/root/sysbench/sys-
  | bench/tests/db/oltp.lua --num-threads=6 --max-requests=0
  | --max-time=600 --mysql-host=172.30.4.220 --mysql-user=sbtest
  | --mysql-password=sbtest --mysql-port=3306 --oltp-tables-
  | count=32 --report-interval=10 --oltp-skip-trx=on --oltp-ta-
  | ble-size=1000000 run

```

We are going to run our tests for 10 minutes and see how it behaves when using MySQL 5.7. Of course, longer is better but again, we are using a simple application which doesn't do anything complex. Let's take a look at the results of our run:

```

1  OLTP test statistics:
2      queries performed:
3          read:                1754942
4          write:               501412
5          other:                0
6          total:               2256354
7      transactions:           0      (0.00 per
8      sec.)
9      read/write requests:    2256354 (3760.51
10     per sec.)
11     other operations:       0      (0.00 per
12     sec.)
13     ignored errors:         0      (0.00 per
14     sec.)
15     reconnects:             0      (0.00 per
16     sec.)
17
18     General statistics:
19     total time:              600.0122s
20     total number of events:  125353
21     total time taken by event execution: 3599.6573s
22     response time:
23         min:                 14.56ms
24         avg:                 28.72ms
25         max:                 586.51ms
26         approx. 95 percentile: 73.54ms

```

We didn't get any errors, which means our queries work just fine on MySQL 5.7. Latency looks also good.

At this stage, more data is better. If you have good profiling of database-related code in your application, you should use it. If you use tools like New Relic, it's also great to keep an eye on it. You should have trending data on your MySQL installation - it can be a good idea to include new MySQL node in your monitoring and trending systems. You want to figure out if there are any issues with the code and any SQL used. But you should also look into performance metrics, and perhaps spot something which may have been missed by previous tests with pt-upgrade.

4.5. Bring back the node into replication

It's time to conclude our tests with one final check - we want to bring the new MySQL 5.7 instance into our replication topology. It may happen that, for some reason, old and new MySQL versions won't work correctly with your query mix. In general, old master and new slave should work just fine, and it indeed works most of the time. We need to check that this is true in our case. We will use a mixed replication environment for a while, with a 5.6 master and 5.7 slaves - unless we can accept downtime and upgrade all nodes at the same time.

The easiest way in our case will be to start replication on our MySQL 5.7 instance, and then bring it into the cluster using the ClusterControl UI. Please note that we've changed its data so replication will most likely fail at some point. Additionally, at the moment of writing, Percona's Xtrabackup didn't work correctly with MySQL 5.7 (some limited support for it has been added in the first release candidate for xtrabackup 2.4).

Thus it was not possible for ClusterControl to automatically rebuild this node using xtrabackup. We need to do it manually, using rsync. We'll stop our MySQL 5.7 node, stop one of the slaves, making sure we disabled fast shutdown before that. Then we'll rsync data from 5.6 to the 5.7 node. Once this is done, we can start the MySQL on our MySQL slave. On our MySQL 5.7 host, we'll make sure the owner is correctly set. We'll remove the auto.cnf file, start MySQL, and finally, run mysql_upgrade. Please note that we've just copied data from MySQL 5.6 so that's why we need to run the mysql_upgrade again. After that has finished, we need to restart our MySQL 5.7 to reload the upgraded tables.

```
1 | root@ip-172-30-4-220:~# service mysql stop
```

On our slave, 172.30.4.169:

```
1 | mysql> set global innodb_fast_shutdown=0;
2 | Query OK, 0 rows affected (0.00 sec)
```

And then, the rest of the process:

```
1 | root@ip-172-30-4-169:~# service mysql stop
2 | root@ip-172-30-4-169:~# rsync -avz -e "ssh -o StrictHost-
  | KeyChecking=no -o UserKnownHostsFile=/dev/null" --progress
  | --delete-after /var/lib/mysql/ 172.30.4.220:/var/lib/mysql/
3 | root@ip-172-30-4-169:~# service mysql start
4 | root@ip-172-30-4-220:~# chown -R mysql:mysql /var/lib/mysql/
5 | root@ip-172-30-4-220:~# rm /var/lib/mysql/auto.cnf
6 | root@ip-172-30-4-220:~# service mysql start
7 | root@ip-172-30-4-220:~# mysql_upgrade -ppass
8 | root@ip-172-30-4-220:~# service mysql restart
```

In the process above, we removed auto.cnf file - this is because we've copied the whole contents of the data directory from our slave and as a result, our new, 5.7 host would have the same UUID as that slave. This would break the replication, therefore we had to remove that file. This will force the generation of a new UUID.

Once this whole process is done, we need to setup replication from the 5.6 master to our 5.7 slave. For that, we need to check the current replication status on our MySQL 5.7. The exact steps may depend on how you have your replication configured. If you store your replication data in InnoDB tables, no further steps other than maybe 'START SLAVE' should be required. If you use files instead, things may be a bit different (although they should not).

In our case, no action was needed as our 5.7 host started the replication as soon as it was restarted after running mysql_upgrade.

So, we have our MySQL 5.7 slaving off a 5.6 master. Now, we'll use the ClusterControl 'Add Node' job to register that new node to our cluster.

Add Node ✕

Create and add a new DB Node
 Add an existing DB Node

Hostname:

Port:

Include in Loadbalancer set(if exist):

Do you want to delay the slave?:

Once the job completes, we should see our new node in ClusterControl.

We don't want to put any production traffic on our upgraded node yet. Therefore we don't want to add it to any of proxies or load balancers. You have to make sure this is the case also in your environment. At the moment, all we want is to observe replication traffic on our 5.7 node and let it run for couple of days - just to make sure the replication works correctly with new MySQL version. Obviously, working replication is a requirement to make the upgrade process as non-impacting as possible.

The screenshot shows the ClusterControl interface for a MySQL replication cluster named 'cluster_1 (ACTIVE)'. The 'Nodes' tab is selected, showing a list of nodes on the left and a detailed view of a 'Read Copy' node on the right. The 'Read Copy' node is selected, and its status is 'OK'. The node details include hardware information (Intel(R) Xeon(R) CPU E5-2676 v3 @ 2.40GHz), host uptime (3 Hours 4 Minutes), and network usage (195 KB). The CPU usage graph shows a peak around 20%.

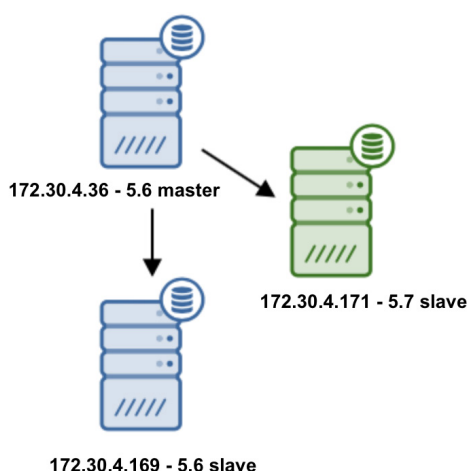
Upgrade

5.1. Slave upgrade process



After testing the new MySQL version, and verifying that our 5.7 slave keeps up with the replication as it should, we can now start upgrading our slaves. We are going to use the same process as for upgrading our test node:

- Disable `innodb_fast_shutdown`
- Shut down MySQL
- Upgrade binaries to MySQL 5.7
- Start MySQL
- Run `mysql_upgrade`
- Restart MySQL
- Make sure replication is working and if needed, fix it



Such process has to be executed on every slave in our topology.

```
1 root@ip-172-30-4-169:~# mysql -ppass -e "set global innodb_
  fast_shutdown=0;"
2 root@ip-172-30-4-169:~# service mysql stop
3 root@ip-172-30-4-169:~# wget http://dev.mysql.com/get/mysql-
  apt-config_0.6.0-1_all.deb
4 root@ip-172-30-4-169:~# dpkg -i mysql-apt-config_0.6.0-1_all.
  deb
```

In this step, we need to confirm that we have an entry like `'pid-file=/path/to/mysql.pid'` in `[mysqld_safe]` section of `my.cnf`

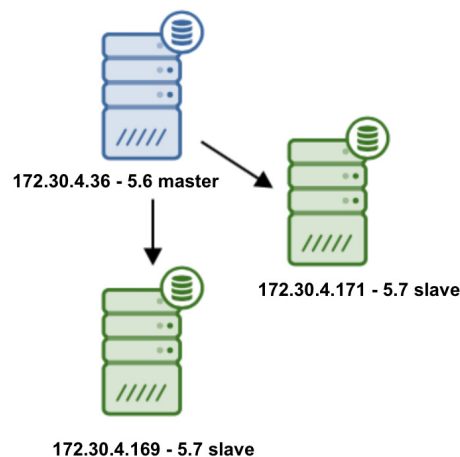

```

1 root@ip-172-30-4-169:~# apt-get update
2 root@ip-172-30-4-169:~# apt-get install mysql-server
3 root@ip-172-30-4-169:~# mysql_upgrade -ppass
4 root@ip-172-30-4-169:~# service mysql restart

```

We are using MaxScale in our setup, therefore some impact may be visible in the application - there will be some short periods of lack of access to data. More information about this issue can be found at <https://mariadb.atlassian.net/browse/MXS-579>.

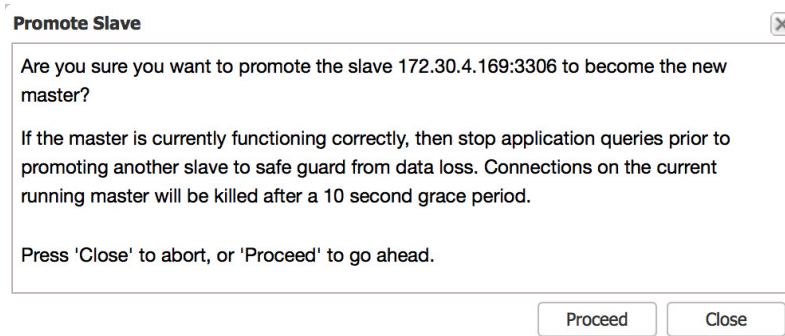
Once all slaves have been upgraded to MySQL 5.7, the last step will be to failover from current master to one of upgraded slaves.



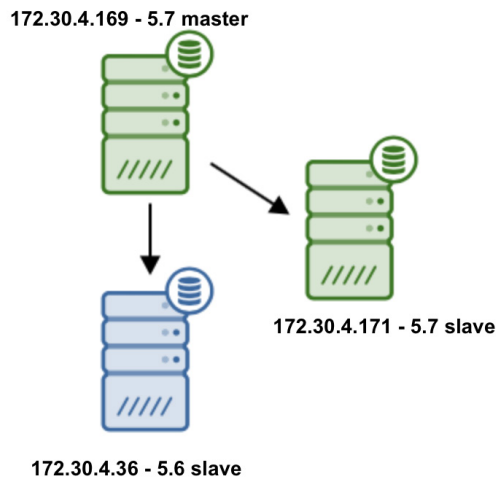
5.2. Switchover process and upgrade of the old master



Switchover can be executed in numerous ways but as long as we use GTID in our replication, we can use ClusterControl perform the master switchover. This is useful as ClusterControl will perform all required steps, including issuing lacking grants and running CHANGE MASTER commands. What we need to do is to decide which slave to promote, and then run a 'Promote Slave' job on it.



After we confirm that we indeed want to promote this node, ClusterControl will execute a master switch.



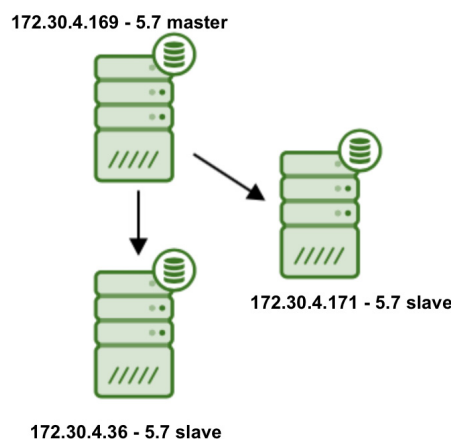
When the switch completes, a couple of steps are still to be executed. First, we need to upgrade the old master to MySQL 5.7 using the process we've discussed earlier. Once all nodes are running MySQL 5.7, the last step will be to make sure that no password expiration will surprise us. As we mentioned at the beginning, there are couple of ways to do that. We can, for example, set a fixed expiration period for the password

```
1 | ALTER USER backupuser@localhost PASSWORD EXPIRE INTERVAL 10 DAY;
```

or disable the password expiration for a particular user.

```
1 | ALTER USER backupuser@localhost PASSWORD EXPIRE NEVER;
```

You need to go over the users and make sure all of them have a correct password expiration policy. Changes should be introduced on the master.



Graceful upgrade process using ProxySQL

As one may have noticed, the process we described above is not graceful - restarts of slaves and final switchover caused connections to break and our application had to retry them in order to make them work. Slave restarts and failovers may happen at any time, therefore it is something the application should be able to handle. It's not an elegant solution, though.

In our case, the problem was related to the proxy - MaxScale. In its current version, it cannot handle nodes going down - the connection is terminated and an error is returned to the application. This would not be a problem if our application can catch these errors and retry transactions as needed. In case this is a problem for the application, there's a solution for that - you can use a proxy that fails over the connections. One of the proxies which can perform a graceful failover is ProxySQL, an SQL-aware load balancer created by René Cannao. We'll guide you through the process of installing the software and setting up a simple read-write split configuration.

6.1. Installation of ProxySQL

Initial installation is pretty simple - all you need to do is to grab the latest binaries from the site:

<https://github.com/sysown/proxysql/releases/>

and install them using rpm or dpkg:

```
1 | root@cmon:~# wget https://github.com/sysown/proxysql/releases/download/v1.1.1-beta.5/proxysql_1.1.1-ubuntu14_amd64.deb
2 |
3 | root@cmon:~# dpkg -i proxysql_1.1.1-ubuntu14_amd64.deb
```

Finally, we can start the service.

```
1 | root@cmon:~# service proxysql start
2 | Starting ProxySQL: DONE!
```

6.2. Configuring ProxySQL for graceful switchover with ClusterControl

The first time you start ProxySQL, it loads an initial configuration which allows you to connect to it and configure it using MySQL client and SQL-like language. We are not going to cover all aspects of the setup, we are going to focus on the steps required to configure read-write split in our cluster. Let's start by connecting to the administrative console and defining some servers.

```

1 root@cmon:~# mysql -u admin -padmin -h 127.0.0.1 -P6032
2
3 mysql> INSERT INTO mysql_servers(hostgroup_id, hostname,
4 port) VALUES (0,'172.30.4.36',3306);
5 Query OK, 1 row affected (0.00 sec)
6
7 mysql> INSERT INTO mysql_servers(hostgroup_id, hostname,
8 port) VALUES (1,'172.30.4.169',3306);
9 Query OK, 1 row affected (0.00 sec)
10
11 mysql> INSERT INTO mysql_servers(hostgroup_id, hostname,
12 port) VALUES (1,'172.30.4.171',3306);
13 Query OK, 1 row affected (0.00 sec)

```

What's important to mention - hostgroups are used to aggregate different types of hosts. They can be hosts which accept similar type of traffic or are used by a particular application. In our case, hostgroup '0' will be used for 'master' and hostgroup '1' will be used for slaves.

```

1 mysql> LOAD MYSQL SERVERS TO RUNTIME;
2 Query OK, 0 rows affected (0.00 sec)
3
4 mysql> SAVE MYSQL SERVERS TO DISK;
5 Query OK, 0 rows affected (0.10 sec)

```

ProxySQL has been built with a concept similar to what's used in the CLI of network devices - changes in configuration are not applied immediately to the system. You have to explicitly load them 'to runtime' to make them work. You also need to save the changes to disk, to make them permanent.

Once we're done with this, next step is to configure users that ProxySQL will use to connect to the backend:

```

1 mysql> INSERT INTO mysql_users (username, password, default_
2 hostgroup) VALUES ('sctest', 'sctest', 0);
3 Query OK, 1 row affected (0.00 sec)
4
5 mysql> LOAD MYSQL USERS TO RUNTIME;
6 Query OK, 0 rows affected (0.00 sec)
7
8 mysql> SAVE MYSQL USERS TO DISK;
9 Query OK, 0 rows affected (0.13 sec)

```

This step is needed for every user that you plan to use to connect to the database via the proxy. Please note the default hostgroup is set to hostgroup 0 - we'll use this in our next step.

Next things to configure are the query rules. ProxySQL uses them to decide how queries should be routed - this is done by specifying a pattern (that the query should match) and a hostgroup, to which the query should be routed.

```

1  | mysql> INSERT INTO mysql_query_rules(active,match_pat-
    | tern,destination_hostgroup,apply) VALUES(1,'^SELECT.*FOR
    | UPDATE$',0,1);
2  | Query OK, 1 row affected (0.00 sec)
3  |
4  | mysql> INSERT INTO mysql_query_rules(active,match_pat-
    | tern,destination_hostgroup,apply) VALUES(1,'^SELECT',1,1);
5  | Query OK, 1 row affected (0.01 sec)

```

In our case we are going to set two rules. We'll leverage the fact that, by default, all queries executed as user 'sbtst' are set to the hostgroup 0, the 'master' in our setup - that's how we set it when adding an user entry. What's left is to route SELECTs to hostgroup 1 (slaves) and explicitly route queries like SELECT ... FOR UPDATE to the master.

Those two inserts do the trick. The rest is up to ProxySQL mechanisms to make sure that whole transactions will end up in the correct place, session settings are set correctly and so on.

It's worth mentioning that ProxySQL loads query rules in the order in which they were created - in our case we first check 'SELECT ... FOR UPDATE' type of queries and then, any query which haven't matched that rule will be tested against second pattern. Those which did not match neither of rules will be routed to the user's default hostgroup.

```

1  | mysql> LOAD MYSQL QUERY RULES TO RUNTIME;
2  | Query OK, 0 rows affected (0.01 sec)
3  |
4  | mysql> SAVE MYSQL QUERY RULES TO DISK;
5  | Query OK, 0 rows affected (0.13 sec)

```

As usual, we finish by loading the changes to runtime and saving them to a persistent storage.

ProxySQL's design doesn't require having MySQL backends actively monitored - it keeps trying to route the queries to the desired hostgroup as long as timeout is not exceeded (10 seconds by default). Health is constantly monitored, not only during connection time but also during the execution of queries - in case of failures, the traffic is routed somewhere else. Having said that, in some cases it's beneficial to setup active monitoring - ProxySQL uses it to determine the replication lag on the node or to check the read_only state on the node. We are going to leverage this feature, therefore we need to set things up.

```

1  | mysql> UPDATE global_variables SET variable_value="cmon"
    | WHERE variable_name="mysql-monitor_username";
2  | Query OK, 1 row affected (0.00 sec)
3  |
4  | mysql> UPDATE global_variables SET variable_value="cmon"
    | WHERE variable_name="mysql-monitor_password";
5  | Query OK, 1 row affected (0.01 sec)
6  |
7  | mysql> LOAD MYSQL VARIABLES TO RUNTIME;
8  | Query OK, 0 rows affected (0.00 sec)

```

```

9 | mysql> SAVE MYSQL VARIABLES TO DISK;
10 |
11 | Query OK, 54 rows affected (0.13 sec)

```

What we did was to update 'global_variables' table and set 'mysql-monitor_password' and 'mysql-monitor_username' variables to a user who will be able to connect and collect the data.

We are getting closer to the end, but there are still few things remaining. When it comes to handling switchovers, ProxySQL does not do that automatically. By default, ProxySQL does not track replication topology - some external input is required to reconfigure ProxySQL whenever a change in the replication topology has been made. You may need to extend your scripts which handle failover to connect to the ProxySQL and make necessary changes in its configuration to, for example, promote a slave to a master. This is not as hard as it sounds - when done right, it's possible to implement a graceful master switch in less than one second.

For MySQL replication, though, ProxySQL has an exception. A very common pattern is to use the 'read_only' variable to differentiate between master and slaves. Slaves are, obviously, read only so they have read_only=1. The Master has this variable disabled. This pattern is frequently used by different tools that manage failovers. ProxySQL can leverage this standard and automatically perform the topology changes. This functionality requires that you have MySQL monitoring set correctly in ProxySQL variables (what we did in the previous step). You also want to make sure it works correctly by, for example, by running select like the one below:

```

1 | mysql> SELECT hostname, port, DATETIME(time_
2 | start/1000/1000,'unixepoch') `time` , connect_success_time ,
3 | connect_error FROM monitor.mysql_server_connect_log ORDER BY
4 | time_start DESC LIMIT 10;
5 | +-----+-----+-----+-----+-----+
6 | | hostname      | port | time                | connect_suc-
7 | |              |      |                      | cess_time | connect_error |
8 | +-----+-----+-----+-----+-----+
9 | | 172.30.4.171 | 3306 | 2016-02-17 08:16:05 | 1001
10 | | NULL         |      |                      |          |                |
11 | | 172.30.4.169 | 3306 | 2016-02-17 08:16:05 | 999
12 | | NULL         |      |                      |          |                |
13 | | 172.30.4.36  | 3306 | 2016-02-17 08:16:05 | 970
14 | | NULL         |      |                      |          |                |
15 | | 172.30.4.171 | 3306 | 2016-02-17 08:14:05 | 999
16 | | NULL         |      |                      |          |                |
17 | | 172.30.4.169 | 3306 | 2016-02-17 08:14:05 | 1095
18 | | NULL         |      |                      |          |                |
19 | | 172.30.4.36  | 3306 | 2016-02-17 08:14:05 | 1232
20 | | NULL         |      |                      |          |                |
21 | | 172.30.4.171 | 3306 | 2016-02-17 08:12:05 | 1537
22 | | NULL         |      |                      |          |                |
23 | | 172.30.4.169 | 3306 | 2016-02-17 08:12:05 | 1360
24 | | NULL         |      |                      |          |                |

```

```

13 | 172.30.4.36 | 3306 | 2016-02-17 08:12:05 | 1497
    | NULL      |      |      |      |
14 | 172.30.4.171 | 3306 | 2016-02-17 08:10:05 | 864
    | NULL      |      |      |      |
15 +-----+-----+-----+-----+-----+-----+-----+-----+
    |-----+-----+-----+-----+
16 | 10 rows in set (0.00 sec)

```

As you can see, we've listed some recent connections from the monitor. There are no errors so we are good to go. Of course, you should also confirm that `read_only` variables are set correctly on your MySQL instances - enabled on slaves, disabled on the master.

The feature we've been talking about can be enabled by a single insert into the `mysql_replication_hostgroups` table.

```

1 | mysql> insert into mysql_replication_hostgroups values (0,
  | 1);
2 | Query OK, 1 row affected (0.00 sec)
3 |
4 | mysql> select * from mysql_replication_hostgroups\G
5 | ***** 1. row
  | *****
6 | writer_hostgroup: 0
7 | reader_hostgroup: 1
8 | 1 row in set (0.00 sec)

```

SELECT should explain what it is all about - we've inserted two values - one for writer hostgroup and one for reader hostgroup. In our case, we've configured the proxy such that whenever a node is determined as a writer (node has `read_only` set to 0 - we expect it to be a single node, master), it will be assigned to hostgroup 0 - the one which handles write traffic in our setup. Nodes determined as readers (nodes with `read_only` set to 1 - we expect them to be our slaves) will be assigned to hostgroup 1 - again, the one which handles SELECT (except for SELECT ... FOR UPDATE).

```

1 | mysql> LOAD MYSQL SERVERS TO RUNTIME;
2 | Query OK, 0 rows affected (0.00 sec)
3 |
4 | mysql> SAVE MYSQL SERVERS TO DISK;
5 | Query OK, 0 rows affected (0.14 sec)

```

Of course, we need to save the changes once more and update the running configuration.

There's one final step and we will be ready to perform graceful switchover.

```

1 | mysql> UPDATE global_variables SET variable_value=20000
  | WHERE variable_name='mysql-connect_timeout_server_max';
2 | Query OK, 1 row affected (0.00 sec)
3 |
4 | mysql> SAVE MYSQL VARIABLES TO DISK;

```

```

5 | Query OK, 55 rows affected (0.08 sec)
6 |
7 | mysql> LOAD MYSQL VARIABLES TO RUNTIME;
8 | Query OK, 0 rows affected (0.00 sec)

```

What we just did was to increase the maximum timeout to 20000 milliseconds (from default of 10000 milliseconds) - as we mentioned earlier, ProxySQL determines the health of the node the moment it opens a connection. If a connection cannot be opened, another node from the hostgroup is used. If there's no available node, ProxySQL will try for 'mysql-connect_timeout_server_max' milliseconds to find another reachable node before an error is thrown back to the application. Switchover process executed by ClusterControl involves 10 seconds of waiting to allow for existing transactions to complete and then the failover itself takes couple more seconds. This make whole process longer than default timeout of 10 seconds in ProxySQL. Once we made this change and updated the running configuration, we are ready to initiate switchover from ClusterControl. There's one more thing worth mentioning. By default, after switchover, ProxySQL configures the new master also as a member of the 'slaves' hostgroup - in our case it means that new master will be a part of both hostgroup 0 and 1. This is intentional - if you have only single slave to failover to, you want it to handle reads too. You can change this behavior in two ways. First method - you can disable it completely by running:

```

1 | mysql> UPDATE global_variables SET variable_value='false'
  | WHERE variable_name='mysql-monitor_writer_is_also_reader';
2 | Query OK, 1 row affected (0.00 sec)
3 |
4 | mysql> LOAD MYSQL VARIABLES TO RUNTIME;
5 | Query OK, 0 rows affected (0.01 sec)
6 |
7 | mysql> SAVE MYSQL VARIABLES TO DISK;
8 | Query OK, 54 rows affected (0.08 sec)

```

But this may cause troubles if you'll end up with a single slave to promote after a master failure.

Second option is, basically, to manually remove the entry from the mysql_servers table when you decide that master should not receive reads. In our case, the table may look like this:

```

1 | mysql> select hostgroup_id, hostname, port, status from
  | mysql_servers;
2 | +-----+-----+-----+-----+
3 | | hostgroup_id | hostname      | port | status |
4 | +-----+-----+-----+-----+
5 | | 0           | 172.30.4.171 | 3306 | ONLINE |
6 | | 1           | 172.30.4.169 | 3306 | ONLINE |
7 | | 1           | 172.30.4.36  | 3306 | ONLINE |
8 | | 1           | 172.30.4.171 | 3306 | ONLINE |
9 | +-----+-----+-----+-----+
10 | 4 rows in set (0.00 sec)

```

We'd need to run the following SQL to remove read traffic from the master:


```

1  mysql> DELETE FROM mysql_servers WHERE host-
   name='172.30.4.171' AND hostgroup_id=1;
2  Query OK, 1 row affected (0.00 sec)
3
4  mysql> LOAD MYSQL SERVERS TO RUNTIME;
5  Query OK, 0 rows affected (0.00 sec)
6
7  mysql> SAVE MYSQL SERVERS TO DISK;
8  Query OK, 0 rows affected (0.12 sec)

```

Failover executed from ClusterControl under a ProxySQL setup the way we did it will look like below from the application point of view:

```

1  [ 60s] threads: 1, tps: 0.00, reads: 846.00, writes:
   240.00, response time: 40.23ms (95%), errors: 0.00, recon-
   nects: 0.00
2  [ 61s] threads: 1, tps: 0.00, reads: 814.00, writes:
   232.00, response time: 40.12ms (95%), errors: 0.00, recon-
   nects: 0.00
3  [ 62s] threads: 1, tps: 0.00, reads: 878.00, writes:
   252.00, response time: 35.81ms (95%), errors: 0.00, recon-
   nects: 0.00
4  [ 63s] threads: 1, tps: 0.00, reads: 841.00, writes:
   240.00, response time: 40.18ms (95%), errors: 0.00, recon-
   nects: 0.00
5  [ 64s] threads: 1, tps: 0.00, reads: 914.98, writes:
   259.99, response time: 26.58ms (95%), errors: 0.00, recon-
   nects: 0.00
6  [ 65s] threads: 1, tps: 0.00, reads: 4.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
7  [ 66s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
8  [ 67s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
9  [ 68s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
10 [ 69s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
11 [ 70s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
12 [ 71s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
13 [ 72s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
14 [ 73s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
15 [ 74s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
16 [ 75s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
   response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00

```

```

17 [ 76s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
    response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
18 [ 77s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
    response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
19 [ 78s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
    response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
20 [ 79s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
    response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
21 [ 80s] threads: 1, tps: 0.00, reads: 0.00, writes: 0.00,
    response time: 0.00ms (95%), errors: 0.00, reconnects: 0.00
22 [ 81s] threads: 1, tps: 0.00, reads: 504.00, writes:
    144.00, response time: 75.61ms (95%), errors: 0.00, recon-
    nects: 0.00
23 [ 82s] threads: 1, tps: 0.00, reads: 602.01, writes:
    172.00, response time: 42.03ms (95%), errors: 0.00, recon-
    nects: 0.00
24 [ 83s] threads: 1, tps: 0.00, reads: 559.98, writes:
    162.99, response time: 51.58ms (95%), errors: 0.00, recon-
    nects: 0.00
25 [ 84s] threads: 1, tps: 0.00, reads: 504.01, writes:
    143.00, response time: 55.62ms (95%), errors: 0.00, recon-
    nects: 0.00

```

There's a period of inactivity which potentially may cause some stalls for users, but the application won't be affected as much as it would have been if the proxy had terminated some transactions on the fly. Additionally, slave restarts (needed in the process of upgrading slaves to 5.7) do not affect our application when using ProxySQL while they do that when using latest (1.2.1) stable build of MaxScale.

As you can see, the process of upgrading MySQL from MySQL 5.6 to MySQL 5.7 is not a complex one. It is definitely a time-consuming process, though - you need to spend a lot of time understanding the changes which may affect your environment, build a solid test environment and do tests in order to make sure your application won't be affected by the upgrade. There are some mines you can step on, like SQL mode changes or password expiration which potentially may change its current behavior and impact you long after the upgrade. On the pro side, at the end you'll be using the best MySQL version to date - both in terms of performance under high concurrency and a great new feature set.

About Severalnines

Severalnines provides automation and management software for database clusters. We help companies deploy their databases in any environment, and manage all operational aspects to achieve high-scale availability.

Severalnines' products are used by developers and administrators of all skills levels to provide the full 'deploy, manage, monitor, scale' database cycle, thus freeing them from the complexity and learning curves that are typically associated with highly available database clusters. The company has enabled over 8,000 deployments to date via its popular ClusterControl solution. Currently counting BT, Orange, Cisco, CNRS, Technicolour, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with offices in Singapore and Tokyo, Japan. To see who is using Severalnines today visit, <http://severalnines.com/customers>.



Deploy



Manage



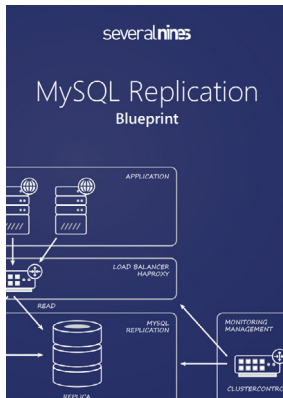
Monitor



Scale

Related resources from Severalnines

Whitepapers



MySQL Replication Blueprint

The MySQL Replication Blueprint whitepaper includes all aspects of a Replication topology with the ins and outs of deployment, setting up replication, monitoring, upgrades, performing backups and managing high availability using proxies.

[Download here](#)



MySQL Replication for High Availability

This tutorial covers information about MySQL Replication, with information about the latest features introduced in 5.6 and 5.7. There is also a more hands-on, practical section on how to quickly deploy and manage a replication setup using ClusterControl.

[Download here](#)



Management and Automation of Open Source Databases

Proprietary databases have been around for decades with a rich third party ecosystem of management tools. But what about open source databases? This whitepaper discusses the various aspects of open source database automation and management as well as the tools available to efficiently run them.

[Download here](#)

several**nines**



Deploy



Manage



Monitor



Scale