Setting up MySQL replication with existing data Master – Slave

Angelo Schalley (https://www.schalley.eu/author/admin/) / Jan, 16, 2017 /
html / php (https://www.schalley.eu/category/html-php/), joomla (https://www.schalley.eu/category/joomla/), Linux (https://www.schalley.eu/category/linux/), MySQL (https://www.schalley.eu/category/mysql/), Plesk Linux (https://www.schalley.eu/category/plesk-linux/)
2 Comments (https://www.schalley.eu/2017/01/16/setting-up-mysql-replication-with-existing-data-master-slave/#comments) /
(https://www.schalley.eu/2017/01/16/setting-up-mysql-replication-with-existing-data-master-slave/) /

This is a step-by-step guide on how to replicate an existing MySQL server. The server is live and contains data and needs a constant backup companion.

Many tutorials focus on how to setup replication when no data is present on the system. That's an ideal solution if you're building a new setup, but in case you've got a server that already has data present then here's how to accomplish the this:

1. setup your existing MySQL server (with data) as a Master
2. export all your databases and user accounts
3. create a slave and import all your data
4. start replication

I've done this several times and always forgot to take some notes – until today. Without further ado, let's replicate MySQL.

Prerequisites

We need the following ingredients for our project:

- a currently running MySQL Server we want to use as a master – it's live so we want to avoid downtime as much as we can
- a new MySQL server which contains no data – soon to be the slave
- root access to both servers, both for the OS and MySQL

Note that any MySQL data on the slave we're creating will be wiped out. There are ways to keep several databases unsynchronized, but this is not covered here.

I'm working on CentOS 6.4 here with MySQL 5.1. Whenever I issue commands to the OS the cursor prefix is "root# ", and when I'm issuing MySQL commands my cursor prefix is "mysql# " – hope this makes sense.

Note that if you have access to phpMyAdmin on both servers, you'll be pleased to hear that there's a much easier and faster way to setup replication. This article however will focus on doing all this via the command line.

Setting up the Master

Our current server needs to write a log of everything he's doing from now on. This log is read later by the slave and contains a collection of statements of what has been executed by the master. Therefore the slave just remembers the position of where he last read the log, and from then onwards just does what the master does. Hence the slave ends up being in sync.

Login to your Master Server as root and edit the MySQL configuration file called ***/etc/my.cnf***.

We need to add the following statements to give our master an ID and setup logging:

```
[mysqld]
log-bin = mysql-bin
server-id = 1
```

Save your file and restart MySQL using

```
service mysqld restart
```

To check that your server is logging things, head over to

```
/var/lib/mysql
```

and see if a file by the name of *mysql-bin.000001* has been created. MySQL will write these files in sizes of up to 1GB, then start the next one. Hence it's worth cleaning out those logs from time to time.

You can also check which file is currently being used by checking *mysql-bin.index* which contains a list of all log files that have been used over time.

Note that you may find the following statements in your configuration file:

```
skip-networking
bind-address = 127.0.0.1
```

These need to be deleted or commented out for replication to work properly. Add a # (hash) in front of each line to do this. If they're already commented out or not even present – even better.

Getting the Master Coordinates

Before we continue, we need to know where the master's last statement was written in the log. We also need to make sure nobody is writing anything to the master while we look at this.

To do this, login to MySQL from a different session and issue the following command:

```
mysql# flush tables with read lock;
```

Just so that we're on the same page here: you now have TWO SSH sessions open, one still connected to your physical server's OS, and another which shows your MySQL prompt.

Thanks to the above command, database writes cannot be committed to disk for now. Notice that MySQL still works fine and all writes are committed to memory – so there's no downtime. We like!

Let's see where the master's last log entry was:

```
mysql# show master status;
```

You'll see something like this:

```
+------------------+----------+--------------+------------------+
| File             | Position | Binlog_Do_DB | Binlog_Ignore_DB |
+------------------+----------+--------------+------------------+
| mysql-bin.000037 |    14462 |              |                  |
+------------------+----------+--------------+------------------+
1 row in set (0.00 sec)
```

My server has been logging for some time, hence the .000037 at the end of the log fie. Make a note of these coordinates. If all goes well you won't need them – but if you run into trouble, these will come in handy.

Backing up the Master

With the lock still in place, let's go back to the first SSH session and dump all databases. Issue the following to create a dump of everything MySQL currently knows:

```
root# mysqldump --all-databases --user=root --password --master-data > everything.sql
```

Depending on how much data you have and how fast your server is, this can take some time… in which you'll get no feedback whatsoever. Sit tight and grab a coffee.

This command will create a file called everything.sql which can be several gigabytes in size. Since we need to copy it later, let's ZIP it up to make it smaller:

```
root# zip -r everything.zip everything.sql
```

That should bring it down by about 80% of its previous size. Feel free to use another compression utility, or leave this step out if you feel the file is small enough for your taste.

If you're an avid Linux command line user you can probably pipe the output of the mysqldump into zip or gunzip, but sadly this knowledge eludes me. Besides, I'm a big believer in smaller, more readable steps.

Now that we have everything from the Master, go back to the MySQL session and unlock the tables so writes can be committed to disk again:

```
mysql# unlock tables;
```

Now the Master will start writing to disk again. You can verify this by issuing the "show master status" command again, and you should see that it's at a different position now. Is this exciting or what?

Creating a Replication User

It is customary to use a dedicated MySQL user to read replication statements from the Master. All this user needs is the "replication slave" privilege. Let's do this while we're in this session:

I'll name mine "replicator" and give it a good password. Since I may want to replicate my data to different slaves, I'll also specify that any host can do this. To make this more secure you can specify a single host if you like.

Still on the MySQL prompt, the following command should take care of this:

```
mysql# GRANT REPLICATION SLAVE ON * . * TO 'replicator'@'%'
IDENTIFIED BY '***' WITH MAX_QUERIES_PER_HOUR 0
MAX_CONNECTIONS_PER_HOUR 0 MAX_UPDATES_PER_HOUR 0
MAX_USER_CONNECTIONS 0 ;
```

This is a long command and for readability I've split it across 4 lines here. Replace '***' with your actual password. If you don't like creating users on the MySQL prompt you can also use phpMyAdmin or Sequel Pro for this (if you have access to either).

You can now quit both sessions; we're done with the master.

Telling your Slave that he's a Slave

Just like we configured the Master, we also need to configure the slave. Open a new SSH session and edit /etc/my.cnf, then give your slave a unique ID:

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

It doesn't have to be just 1 or 2, you can go crazy and use a 10 digit number if you like. Just make sure the master and slave to not have the same ID. Just as before, restart MySQL on the Slave for the change to take effect:

```
root# service mysqld restart
```

Importing your data on the Slave

The (potentially zipped) snapshot we've created on the Master needs to be copied over to our slave. I like using rsync for this, but scp or even FTP will do just fine. Use whatever you're comfortable with, as long as the file ends up on your Slave machine.

For example, to copy everything.sql to my slave in its current directory I can issue this on the slave:

```
root# rsync -av root@master:/everything.sql .
```

But as I said, use whatever you know best. We'll assume that everything.sql has made it onto your slave, and that we're in the directory where the file currently resides. We'll also assume that you've un-zipped it so we end up with everything.sql.

We can now import the file, wiping out all existing databases on the slave. This shouldn't be a problem if it's a new machine that's built for the purpose of being a slave, but if you have any data you may want to keep, this is the time to export it.

Issue the following command:

```
root# mysql -p < everything.sql
```

This will read in all your databases and user accounts, line by line, from the backup file. Since writes take longer than reads this can take some time again – time in which you will get no feedback whatsoever (again). I guess it's time for another coffee/meal/movie/trip to New York.

A note on user accounts: each database has its own directory in /var/lib/mysql – including a database called "mysql". This is MySQL's own database in which it stores users and their privileges. Hence, when you import all your previous databases into the slave, you'll automatically copy all user accounts from the master too.

At the same time, all information about your slave's user accounts prior to the import will be eliminated. Therefore your previous MySQL root password for your Slave will no longer work – as it's been replaced with that of the master.

When the command has finished (without any feedback if all went well) you can verify that it's done a good job: login to MySQL with the Master's root credentials:

```
root# mysql -u root -p
```

Then have a look at all your server's handy work:

```
mysql# show databases;
```

You should see a list of all your Master's databases. Neat, huh?

Starting the Replication Station

With the Master and Slave now more or less in sync, we need to make sure the Slave picks up where the Master's last statement was. Before we go ahead, restart your MySQL Slave – trust me, this will avoid syncing issues:

```
root# service mysqld restart
```

Remember when we dumped all our databases, we used a switch called "–master-data"? This is something that will automatically set the log coordinates on the Slave to where the Master left off writing statements. There's no need to give our Slave this data again. Keep in mind that for troubleshooting purposes you can set those manually if you wish.

So all we have to do now is tell the Slave who the Master is and start it. Still on the MySQL prompt, issue the following:

```
mysql# change master to
-> MASTER_HOST='masterip-or-domain',
-> MASTER_USER='replicator',
-> MASTER_PASSWORD='slavepassword';

mysql# start slave;
```

With a bit of luck, this will start your Slave, which will in turn add everything that's happened on the Master since you took the snapshot. If all went well, this is the end of your setup. Go celebrate!

Quick notes about the above statement:

You can hit return at any time on the MySQL prompt, in which case a "->" is displayed so you can add more. Only when you enter a semicolon will your statement be executed. So don't add the "->", MySQL will do this when you hit enter.

CaSe sEnsiTiviTY doesn't matter to MySQL: "MASTER_PASSWORD" is the same as "master_password".

You can add the following if you wish – you see this in many tutorials:

```
-> MASTER_LOG_FILE = 'mysql-bin.000037',
-> MASTER_LOG_POS = 14462;
```

This is necessary if you take a mysqldump without the "–master-data" option, or if you want to point the Slave manually. If you set "–master-data", then these two coordinates will be added automatically upon import.

Verify that Replication is working

I don't like leaving things to chance and need "proof" that everything is working as intended. The simplest way to test your replication setup is by adding a new database on the master, then check if it exists on your slave.

You can do this via the command line, phpMyAdmin, Sequel Pro – or any other utility. Let's create two MySQL sessions, one on the Master and one on the Slave. On both, issue the following:

```
mysql# show databases;
```

This shows you all databases – hopefully an identical list on both servers. Now create a new database on the Master:

```
mysql# create database zzz;
```

List your databases again and you should see "zzz" on either server. You can create as many as you like – it's magic seeing them come to life almost instantly on your Slave. Everything works this way: tables, rows, users all change in a flash.

If your test databases only show up on the Master, then you have a problem (see below). Remember that if you've only just started the Slave, he may have some catching up to do – so give it a few minutes, as he may still be working through earlier statements from the Master.

When you've finished testing, delete those test databases databases with

```
mysql# drop database zzz;
```

MySQL Replication Troubleshooting

Sometimes things don't work out with replication. When I first started experimenting with it I thought this was a "setup and forget about it" kind of job.

Experience has shown though that you have to regularly triple check and see if things may have broken (despite a good and once working setup).

Master Troubleshooting

Replication problems on your Slave do not usually indicate a fault on the Master; as long as it's creating a log, all is well. You can check this by issuing a couple of "show master status" commands in 30 second intervals and see the log position change.

Provided data is being created or changed, the log positions should be different. If no log status shows up, then you're being told that "binary loggin is disabled" – in which case you need to update your my.cnf file.

Slave Troubleshooting

More often replication fails because of something which isn't working well on the Slave. Common scenarios include:

- login problems: wrong user/password/host combination
- data import/synchronization problems
- problematic statements
- corrupt tables or databases

Let's go through these one by one. I intend to update this article if I find other problems (and solutions). Feel free to share your experiences in a comment at the bottom.

How to check what's bothering your Slave

Your best place to see where something's gone wrong is the MySQL log. On CentOS and many other Linux distributions that's in ***/var/log/mysqld.log*** and you can check the last 10 lines with

```
tail /var/log/mysqld.log
```

Not the easiest thing to read, but you get the basic idea of what the problem is.

Slave can't login to the Master

Any issues pertaining to log data not being read, or connections not being possible are most certainly due to login problems.

See if you yourself can login to your Master as the replication user if you've set one up. Log in like this:

```
mysql -h domain+or+ip -u replicationuser -p
```

If this isn't possible then you need to look at your credentials on the Master. Perhaps the replication password is wrong, perhaps the user doesn't have the "replication slave" privilege, or is not allowed to connect from your host.

You can always change your slave to use your root user momentarily to see if connections in general are possible. Note that this is not meant as a long term solution.

Master and Slave are out of sync

If you CAN login and your Slave still throws up errors then it's usually because the Slave is trying to execute a command from the Master log and fails. For example, he may try to delete a database that doesn't exist. Such things can happen if you've (accidentally) changed data on the Slave. That's a big no-no.

It's "read only", all else is just a replica of the Master. Even though you can introduce new data onto the Slave, changing data that has been replicated from the Master will cause problems.

Sync problems can also occur when you didn't restart MySQL on the Slave just after the big data import.

Lucky for us this is not a biggie though: simply stop the Slave, restart MySQL and give the your Slave the Master Log Coordinates manually (that's why it's handy to make a note of those).

You do it like this, obviously replacing the values with your own:

```
stop slave;

change master to
MASTER_LOG_FILE = 'mysql-bin.000037',
MASTER_LOG_POS = 14462;

start slave;
```

Corrupt Data on the Slave

Sometimes tables or databases can get corrupt when they're not properly closed. These things can happen on the Master – and as a result find their way onto the Slave, or something bad can happen on the Slave.

In either case, the Slave may not be able to process a statement and as a result will also stop and complain in his log.

To overcome this, you have two options:

- skip the bad statement and carry on
- repairing the corrupt tables and carry on

Let's discuss how to tackle both.

Skip a Statement or two (not recommended)

This is the quick and dirty way of making the Slave ignore one (or several) Master log statements. This is not a problem if we're dealing with disposable or temporary data – but it is a problem if you're not (and frankly, who can really tell).

Be aware that skipping statements on the Slave will cause data inconsistencies – if these are noticeable or not depends entirely on your data and applications.

Here's how you tell the Slave to skip the next 1 statement:

```
stop slave;
set global sql_slave_skip_counter = 1;
start slave;
```

You can skip as many events as you like by increasing the value. Once the Slave is running again, go back to the log and see if processing goes well. Use this approach with extreme caution!

There's an interesting little script here by Vito Botta (http://vitobotta.com/recovering-mysql-replication) on how to do this automatically.

Repairing your Tables and Databases

Ignoring a problem and hoping it doesn't bite you later is never a good idea – repairing what's broken on the other hand sounds like a good plan.

You can repair single tables or databases with the "repair table" and "repair database" commands respectively – but what happens when you have hundreds of databases containing several thousand tables? You'd sit there all night issuing commands.

Lucky for us there's a handy tool that does it for us:

- mysqlcheck – check it out in the MySQL Manual (http://dev.mysql.com/doc/refman/5.1/en/mysqlcheck.html).

From the OS command prompt, issue this:

```
mysqlcheck -p -A --auto-repair
```

This can take some time, but it will check and repair every table in every database you have. You can even perform this on a live server without taking it offline, how cool is that.

Once done, start your Slave again, give it some time to catch up and will hopefully resume its replication duties.

Last Resort

If none of the above works for you, then you can really only do one thing: lock your master again, take a new snapshot and re-import it to the Slave.

Ref : wpguru . co . uk

« Convert apache htaccess rewrite rules to nginx rewrite rules automatically (https://www.schalley.eu/2017/01/06/convert-apache-htaccess-rewrite-rules-to-nginx-rewrite-rules-automatically/)

Netscaler check saml error's on console (https://www.schalley.eu/2017/03/07/netscaler-check-saml-errors-on-console/) »