severalnines

MySQL on Docker

How to Containerize Your Database

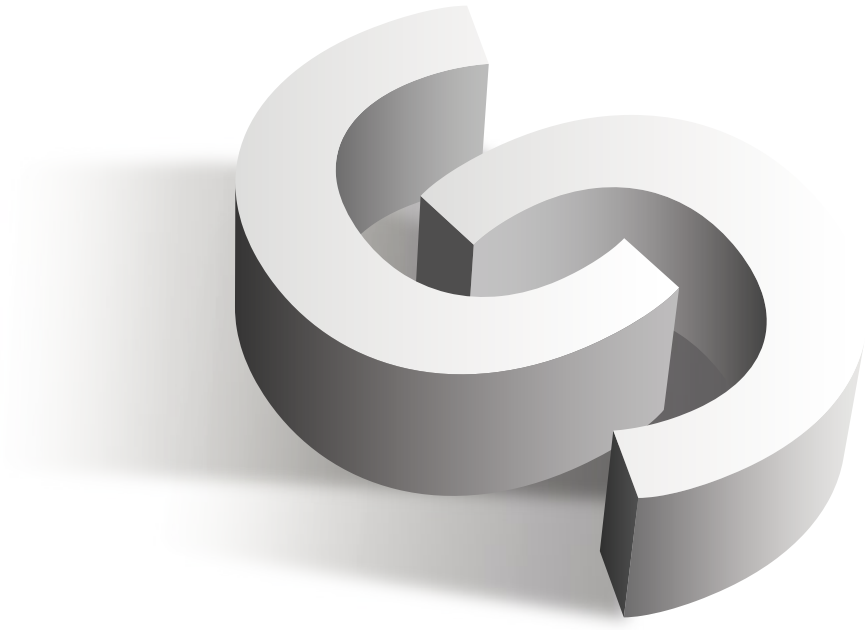


Table of Contents

Introduction

Docker is quickly becoming mainstream, as a method to package and deploy self-sufficient applications in primarily stateless Linux containers. Yet, for a stateful service like a database, this might be bit of a headache. How do we best configure MySQL in a container environment? What can go wrong? Should we even run our databases in a container environment? How does performance compare with e.g. running on virtual machines or bare-metal servers? How do we manage replicated or clustered setups, where multiple containers need to be created, upgraded and made highly available?

This whitepaper covers the basics you need to understand considering to run a MySQL service on top of Docker container virtualization. Take note that this whitepaper does not cover MySQL container orchestration on multiple hosts.

Introduction to Docker

The key benefit of Docker is that it allows users to package an application with all of its dependencies into a standardized unit (container). Running many containers allows each one to focus on a specific task; multiple containers then work in concert to implement a distributed system.

2.1. Concepts

Think about a container as a "lightweight virtual machine". Unlike virtual machines though, containers do not require an entire operating system, or all required libraries including the actual application binaries. The same Linux kernel and libraries can be shared between multiple containers running on the host. Docker makes it easy to package software in self-contained images, where all software dependencies are bundled and deployed in a repeatable manner. An image will have exactly the same software installed, whether we run it on a laptop or on a server.

Every container has its own file system, process space and also a network stack, where each container will be assigned with at least one network interface. Due to this isolation, containers won't affect the machine host process or file system. More on this in ***Chapter 5 - MySQL Container and Volume***.

2.2. Components

Since version 17.x, Docker has split into two editions:

- Docker Community Edition (CE)
- Docker Enterprise Edition (EE)

Docker Community Edition (CE) is a free version, ideal for developers and small teams looking to get started with Docker and experimenting with container-based apps. Docker CE has two update channels, stable and edge:

- Stable - New updates every quarter
- Edge - New features every month

Docker Enterprise Edition (EE) is a commercial subscription, and includes software, support and certification.

- Regardless of the edition, Docker consists of the following components out-of-the-box:
- Docker Client - The command line tool that allows the user to interact with the Docker daemon.
- Docker Engine - The background service running on the host that manages building, running and distributing Docker containers.

- Docker Image - The file system and configuration of our application which are used to create containers.
- Docker Container - Running instances of Docker images. A container includes an application and all of its dependencies. It shares the kernel with other containers, and runs as an isolated process in user space on the host OS.
- Registry - A registry of Docker images, where you can find trusted and enterprise ready containers, plugins, and Docker editions.

2.3. Benefits

Generally, containerization allows you to get a greater density on the physical host. It reduces conflicts between various deployment environments, like development, testing, staging, production, and also speeds up deployment.

The following points in particular are benefits of running MySQL as container:

- Cost savings - Docker can help facilitate cost savings by dramatically reducing infrastructure resources. The nature of Docker is that fewer resources are necessary to run the same application, in this case MySQL.
- Standardization - Docker provides repeatable development, build, test, and production environments. Standardizing service infrastructure across the entire pipeline allows every team member to work on a production parity environment. You can easily create an immutable MySQL image to streamline the database configuration and setup.
- Compatibility - Less time is spent setting up environments, debugging environment-specific issues, and a more portable and easy-to-set-up codebase, achieving similar consistency and functionality. Images built on one Linux distribution need zero customization to run on any other Linux distributions.
- Simplicity - Docker can be used in a wide variety of environments, the requirements of the infrastructure are no longer linked with the environment of MySQL.
- Rapid Deployment - Docker creates a container for every process and does not boot an OS. Data can be created, persisted and destroyed without worry that the cost to bring it up again would be higher than affordable.
- Multi-cloud platforms - Infrastructure in the cloud with dynamic usage is much more cost-efficient. All major cloud providers have embraced Docker's availability, where some of them have included support for full automation container orchestration platform.
- Isolation - By default, no Docker container can look into processes running inside another container. From an architectural point of view, each container gets its own set of resources ranging from processing to network stacks.
- Security - A container can be ran as read-only, reducing the attack vector significantly. All communications between Docker and clients are encrypted using TLS.

There are real benefits of running MySQL on Docker. Some of the management and administration practices are different to the conventional day-to-day operational practices. More on this in ***Chapter 6 - Monitoring and Management***.

2.4. Installation

Docker can run on most platforms, from your workstation or servers to cloud infrastructure. For desktop, it supports Windows and Mac. AWS and Azure support Docker on their cloud infrastructure. For servers, it supports Windows Server, and popular Linux distribution - RedHat, CentOS, Debian, Ubuntu, SUSE, Fedora and Oracle Linux. Take note that containers aren't VMs. They offer isolation, not virtualization. The host and container OSs must be the same. You cannot use a Linux container on a Windows machine or vice-versa.

In this paper, we are going to use Docker Community Edition (CE) on an Ubuntu 16.04 LTS. The installation steps are pretty straightforward, as shown below:

1. Update the apt package index:

```
$ sudo apt-get update
```

2. Install packages to allow apt to use a repository over HTTPS:

```
$ sudo apt-get install \
    apt-transport-https \
    ca-certificates \
    curl \
    software-properties-common
```

3. Add Docker's official GPG key:

```
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg |
sudo apt-key add -
```

4. Add Docker CE stable repo:

```
$ sudo add-apt-repository \
   "deb [arch=amd64] https://download.docker.com/linux/ubun-
tu \
   $(lsb_release -cs) \
   stable"
```

5. Update the apt package index and install the latest version of Docker CE:

```
$ sudo apt-get update
$ sudo apt-get install docker-ce
```

At this point, Docker engine is running as a daemon. You can verify this with the `systemctl status` command. For more details, please refer to the documentation.

MySQL Images

In order to run a Docker container, you need to have an image to start with. An image is like a template. It has all the required things you need to run the container. That includes software, including the code or the binaries, dependencies, libraries, interpreter, environment variables and config files. There are two ways to create an image:

- Commit all the changes of a container.
- Use a more declarative approach defined inside a text file, called Dockerfile, together with the docker build command.

3.1. To commit changes and create Docker images

To building an image by committing the changes is similar to deploying MySQL on a standard host. The following table compares the simplest steps required to install and run a MySQL server on an Ubuntu 16.04, both on a standard host and with the equivalent steps for Docker:

Steps	Host	Docker
Start a container		`$ docker run -d --name=abc ubuntu:16.04`
Attach inside a container		`$ docker exec -it abc /bin/bash`

Install MySQL and configure MySQL parameters inside my.cnf	`$ apt-get update`
	`$ DEBIAN_ FRONTEND=noninteractive apt-get install -y mysql-server`
	`$ vi /etc/mysql/my.cnf`
		`root@abc:/# apt-get update`
		`root@abc:/# DEBIAN_ FRONTEND=noninteractive apt-get install -y mysql-server`
		`root@abc:/# vi /etc/mysql/ my.cnf`

Build image		`$ docker commit abc myimage/ mysql:5.7`
Run MySQL	`$ mysqld --socket=/var/lib/ mysql/mysqld.sock --pid-file=/ var/lib/mysql/mysqld.pid`	`$ docker run -d myimage/ mysql:5.7 mysqld --socket=/var/ lib/mysql/mysqld.sock --pid-file=/var/lib/mysql/mysqld.pid`

One would start by pulling a base image, and then start a container based on this image. Then, perform the required changes directly into the container, like installing MySQL packages and editing the MySQL configuration file. Once everything is in place, commit the container into a single image with the following formatting:

- Username: “myimage”
- Followed by a “/”
- Followed by an image name: “mysql”

- End with an optional tag: “5.7”. If undefined, the tag will default to “latest”.

When the container is being committed, the running processes will be paused to reduce the risk of data corruption.

To run a container based on this image, one would use the **`docker run`** command with the image name, followed by the execution command which is “mysqld” followed by the configuration variables for MySQL.

3.2. Using Dockerfile

Using Dockerfile to create an image is much simpler and declarative. Dockerfile is a text file that contains all the instructions to build an image. The build process is executed by the Docker daemon.

Dockerfile is the preferred way to maintain your Docker images. It can be integrated with a registry service like Docker Hub to provide continuous integration and it is easier to see what is provided by the image. The following table compares the equivalent steps with the standard deployment vs Dockerfile approach:

Steps	Host	Docker

Install MySQL	`$ apt-get update`
	`$ DEBIAN_ FRONTEND=noninteractive apt-get install -y mysql-server`
		`# Dockerfile content`
		`FROM ubuntu:16.04`
		`RUN apt-get update`
		`RUN DEBIAN_ FRONTEND=noninteractive apt-get install -y mysql-server`
		`COPY my.cnf /etc/mysql/ my.cnf`
		`EXPOSE 3306`

Configure MySQL	`$ vi /etc/mysql/my.cnf`	`$ vi ~/my.cnf`
Build image		`$ docker build -t myimage/ mysql:5.7 .`
Run MySQL	`$ mysqld --socket=/var/lib/ mysql/mysqld.sock --pid-file=/var/lib/mysql/mysqld.pid`	`$ docker run -d myimage/ mysql:5.7 mysqld --socket=/ var/lib/mysql/mysqld.sock --pid-file=/var/lib/mysql/ mysqld.pid`

By looking at the content of our Dockerfile, one can easily tell that the Docker image runs on Ubuntu 16.04. It has a MySQL server installed through the Debian OS package repository, with a custom my.cnf that will be copied over inside the container under **`/etc/mysql/my.cnf`**. By default, Docker will expose port 3306 when the container is started. To build the image, use **`docker build`** command with a proper naming and tag.

For more details on Dockerfile reference, please refer to Dockerfile documentation page.

3.3. Building the Image

When building a custom image with Docker, each action taken, for example `apt-get install mysql-server`, forms a new layer on top of the previous one. An image is simply a collection of layers stacked on top of each other and this layer will be read-only once it is built. These base images can then be used to create new containers. The following diagram illustrates building an image based on the Dockerfile from the previous section:

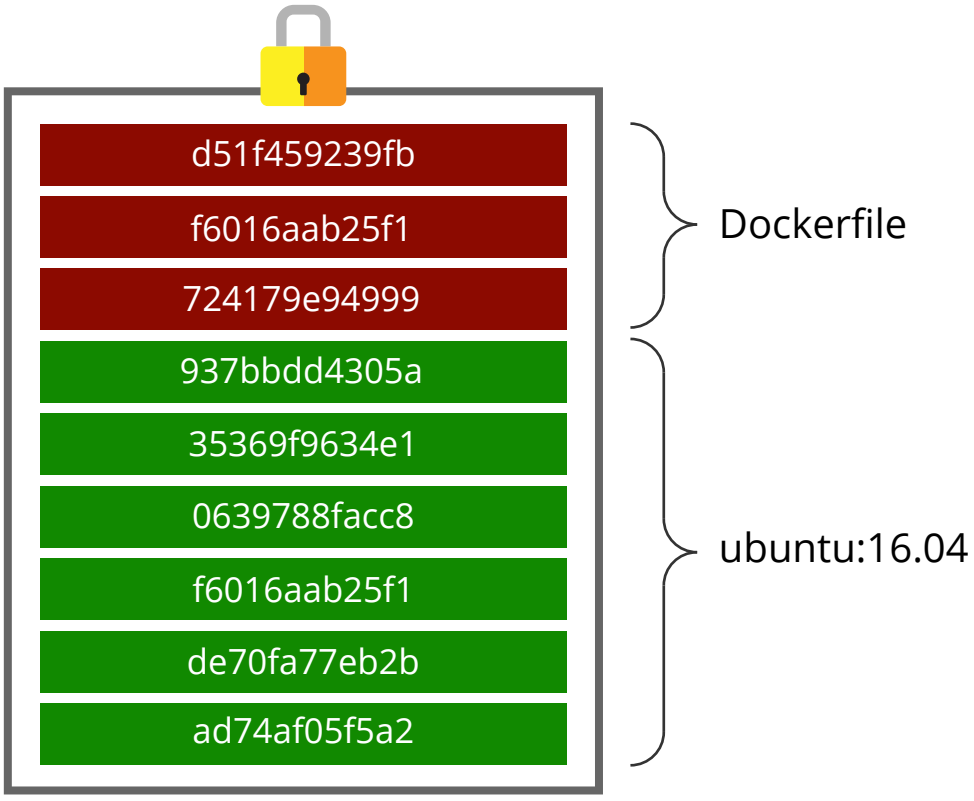


Image: myimage/mysql:5.7

The layers in red are related to our Dockerfile commands, while the green layers are inherited from the Ubuntu 16.04 base image that we have defined in the Dockerfile. You can see the layers by using the `docker history` command:

```
$ docker history myimage/mysql:5.7
CREATED BY                                      SIZE
/bin/sh -c #(nop) COPY file:e8163ef656b6da...   101B
/bin/sh -c DEBIAN_FRONTEND=noninteractive ...   370MB
/bin/sh -c apt-get update                       39.1MB
/bin/sh -c #(nop)  CMD ["/bin/bash"]            0B
/bin/sh -c mkdir -p /run/systemd && echo '...   7B
/bin/sh -c sed -i 's/^#\s*\(deb.*universe\...   2.76kB
/bin/sh -c rm -rf /var/lib/apt/lists/*          0B
/bin/sh -c set -xe   && echo '#!/bin/sh' >...   745B
/bin/sh -c #(nop) ADD file:39d3593ea220e68...   120MB
```

From the above output, we can see that the second layer is larger in size, where we defined `apt-get install mysql-server` with 370MB in size. This layer consists of MySQL components and all of its dependencies.

3.4. Sharing the Image

Once the image is built, you can distribute it to other hosts manually using a tarball, or push the image to the Docker image repository. The following example shows how to save a built MySQL image named 'myimage/mysql:5.7' into a TAR file:

```
1 | $ docker save -o mysql.tar myimage/mysql:5.7
```

The other method is to push the image directly to the Docker registry. The registry is a stateless, highly scalable server side application that stores and distributes Docker images. Docker Hub is the most popular public registry. It stores thousands of Docker images and supports automated build for continuous integration (CI) where it can hook to a code repository like Github and trigger a build when it sees a new commit being pushed. You can use an image published and maintained by the application vendor itself, or use other user contributed image, or you can create your own images to suit your needs.

```
1 | $ docker push myimage/mysql:5.7
```

To pull the image to the Docker host:

```
1 | $ docker pull myimage/mysql:5.7
```

A public registry like Docker Hub is very helpful for public and open-source images. However, for a organisation's private images, you should go for a private registry, either by subscribing to hosted services like quay.io, Amazon EC2 Container Registry (ECR), Google Cloud Registry (GCR) and Bintray, or you can also opt for a self-hosted private registry if you have enough resources for processing, bandwidth and storage. The following command can be used to start your own Docker Registry service running on port 5000 on the Docker host:

```
1 | $ docker run -d -p 5000:5000 --name registry registry:2
```

3.5. Best practices when building Docker images for MySQL

There are hundreds of MySQL container images available on Docker Hub that we can reuse, and enhance with more functionality. Inheriting a MySQL image allows us to leverage the work done by the maintainer. For example, you can use the image that is built and maintained by the application vendor of your choice; Oracle MySQL (mysql/mysql-server), Percona (percona/percona-server), MariaDB (mariadb/mariadb-server) or the Docker team itself (mysql or docker.io/mysql).

Try to avoid unnecessary packages inside the image to increase the deployment speed. Only install packages required by the MySQL server and client. There is also a popular tiny image called "alpine", which comes with its own package manager, and includes MySQL and MariaDB packages as well. You can use alpine as base image and build your own MySQL server around it. For the sake of comparison, the base image is just 5MB in size while the Ubuntu 16.04 base image is around 190MB in size.

When you want to build your own custom MySQL image, and you want to run an entrypoint script at container startup, make sure the last command of the script starts with "exec" followed by the main process of the container, in this case is "mysqld". This process will hold PID 1 during container runtime, and is critical when stopping the container later on. Details on this in ***Chapter 6.1 - Service Control***.

Finally, try to follow the general guidelines when writing a Dockerfile, especially when building a public image. In this way, anyone else can easily understand the content of the image and contribute to it.

Networking in Docker

Networking is critical in MySQL, it is fundamental to manage access to the database server from client applications and other database servers (e.g. in master-slave replication setups). The behaviour of a containerized MySQL service is determined by how the MySQL container is started with the `docker run` command. A MySQL container can be run in an isolated environment (only reachable by containers in the same network), or an open environment (where the MySQL service is totally exposed to the outside world) or the instance simply runs with no network at all.

Once Docker Engine is installed, Docker will create 3 types of networks by default:

- host
- none
- bridge

The **bridge** network also known as docker0, is the default network when you create a container without specifying any networking related parameter (`--net` or `-n`). The **none** network is using the "null" driver and no interface will be configured inside the container, other than localhost. Bear in mind that **host** and **none** networks are not directly configurable in Docker.

You can get a list of created networks in Docker with the `docker network` command:

```
$ docker network ls
NAME                DRIVER              SCOPE
bridge              bridge              local
db_multi            overlay             swarm
db_single           bridge              local
docker_gwbridge     bridge              local
host                host                local
ingress             overlay             swarm
none                null                local
```

Docker also supports multi-host network out-of-the-box through the Docker Swarm overlay network. You can also use other network plugins to extend the Docker network capabilities, as described in the upcoming chapters.

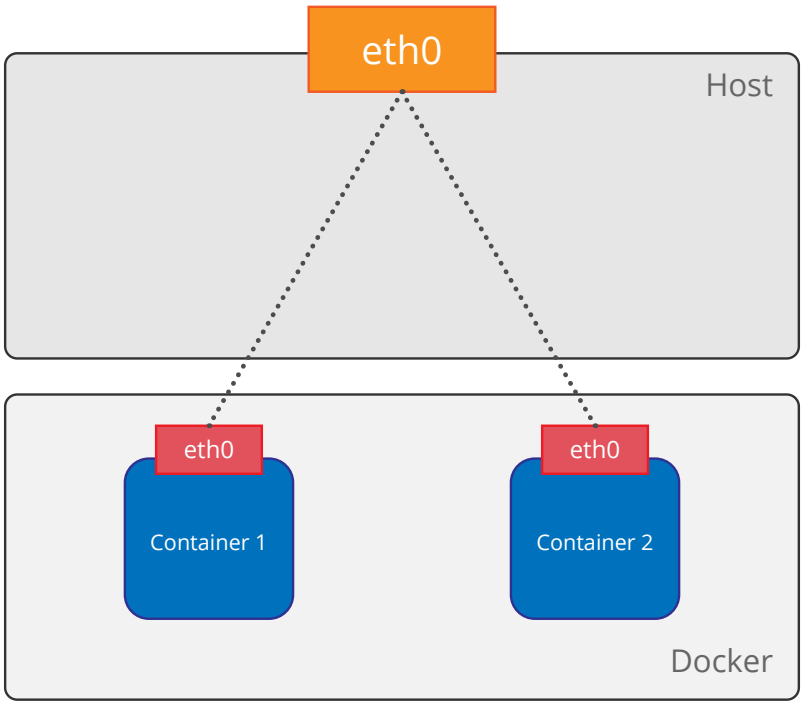
4.1. Host Network

Host network adds the container to the host's network stack, which means the container will have the exact same network characteristics as the host interface. Only one host network per machine host is allowed.

In order to run a container in this network, specify `--net=host` in the docker run command:

```
$ docker run -d \
--name mysql-host \
--net host \
-e MYSQL_ROOT_PASSWORD=mypassword \
mysql:5.7
```

The container will be attached directly to the network stack of the hosts. If the host has two network interfaces, the container will also have them. There is no isolation in this network, which means containers created on this network are reachable by containers created inside the bridge network.



The container does not need any forwarding rules using iptables since it is already attached to the same network as the host. Hence, port mapping and container linking are not supported and Docker will not manage the firewall rules of containers that run in this type of network. That means, you can only run one MySQL container per host under this network.

This network is only helpful if you want to dedicate the host machine as a MySQL server, and manage it under Docker.

4.2. Bridge Network

Docker bridge network creates a virtual Ethernet bridge host network interface, a single aggregate network from the host network. There are two types of bridge networks supported by Docker:

1. Default bridge (docker0)
2. User-defined bridge

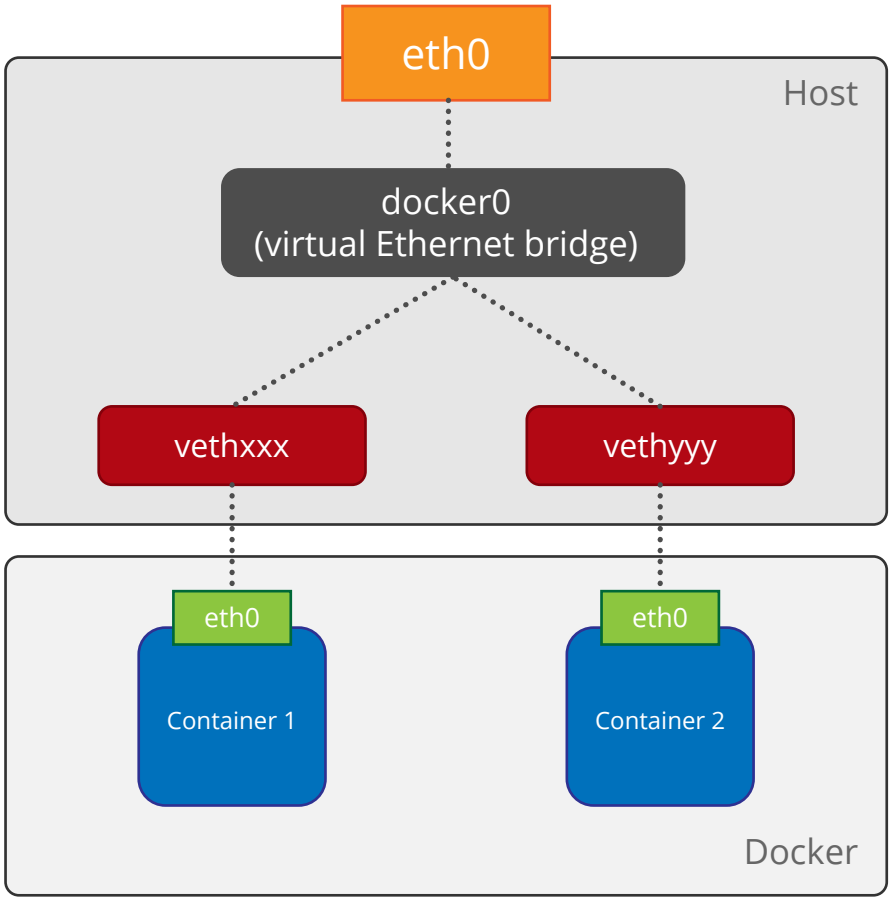
Each outgoing connection will appear to originate from the host machine's IP address.

4.2.1. Default Bridge

The default bridge network, also known as docker0, will be automatically created by Docker upon installation. Basically, if you do not specify `--net` parameter in the `docker run` command, Docker will create the container under this docker0 network:

```
$ docker run -d \
--name mysql-bridge \
-p 3308:3306 \
-e MYSQL_ROOT_PASSWORD=mypassword \
mysql:5.7
```

On the machine host, you will see a new virtual Ethernet interface, "veth" as illustrated by the red box in the diagram below, and one Ethernet interface, "eth0" inside the container:



Docker utilises iptables to manage packet forwarding to the bridge network via docker-proxy. It redirects connections to the correct container through NAT. Thus, it supports `--publish` parameters where you can run multiple containers with the same image through different ports. Container linking is also supported, where you can expose environment variables and auto-configured host mapping through `/etc/hosts` file inside the linked containers.

To access the MySQL service, simply point the MySQL client to the Docker host and the assigned port of the container, 3308 in this case. Only one default bridge network is allowed per Docker host.

4.2.2. User-defined Bridge

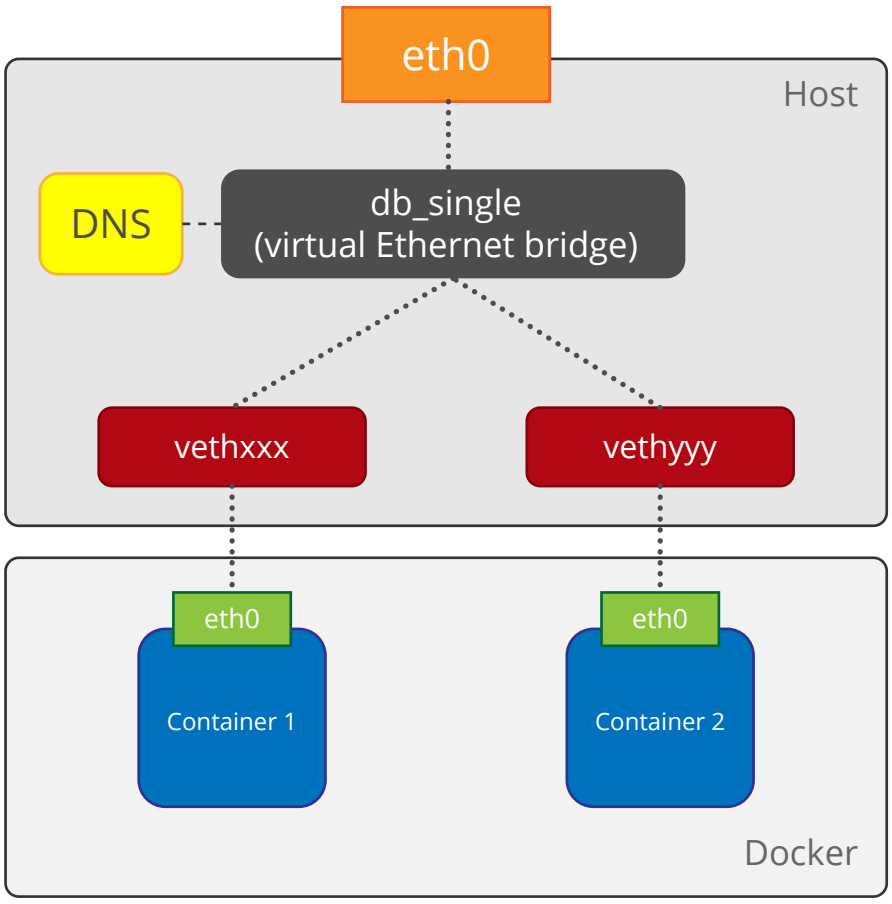
In order to use user-defined bridge, the user has to create the network beforehand:

```
$ docker network create subnet=192.168.10.0/24 db_single
```

Then, pass the network name using `--net` or `-n` parameter to run the container under this network:

```
$ docker run -d \
--name mysql1 \
--net db_single \
-p 3308:3306 \
--ip 192.168.10.10 \
--hostname mysql1 \
-e MYSQL_ROOT_PASSWORD=mypassword \
mysql:5.7
```

The main difference between the default bridge network and user-defined network is the embedded DNS service. You can resolve a container's name in the same network to an IP address, which is practical for simple service discovery among database containers. This network also provides a sticky IP address and hostname, using `--ip` and `--hostname` parameters.



This is the recommended network to run single-host MySQL containers, because MySQL relies on a proper hostname or IP address for authentication and authorization. There is also a performance consideration. By using IP address when granting a user, MySQL does not need to perform reverse DNS lookup when a user is authenticating. The reason is that resolving a name can potentially cause a problem if the DNS resolver does not respond in a timely manner. But this is rarely happening in Docker's networking.

You can have multiple user-defined bridge networks in a host and you can run a container connected to multiple networks as well by repeating the `--net` parameter for each network:

```
$ docker run -d \
--name mysql1 \
--net db_single \
--net backend \
--net database \
-p 3308:3306 \
--ip 192.168.10.10 \
--hostname mysql1 \
-e MYSQL_ROOT_PASSWORD=mypassword \
mysql:5.7
```

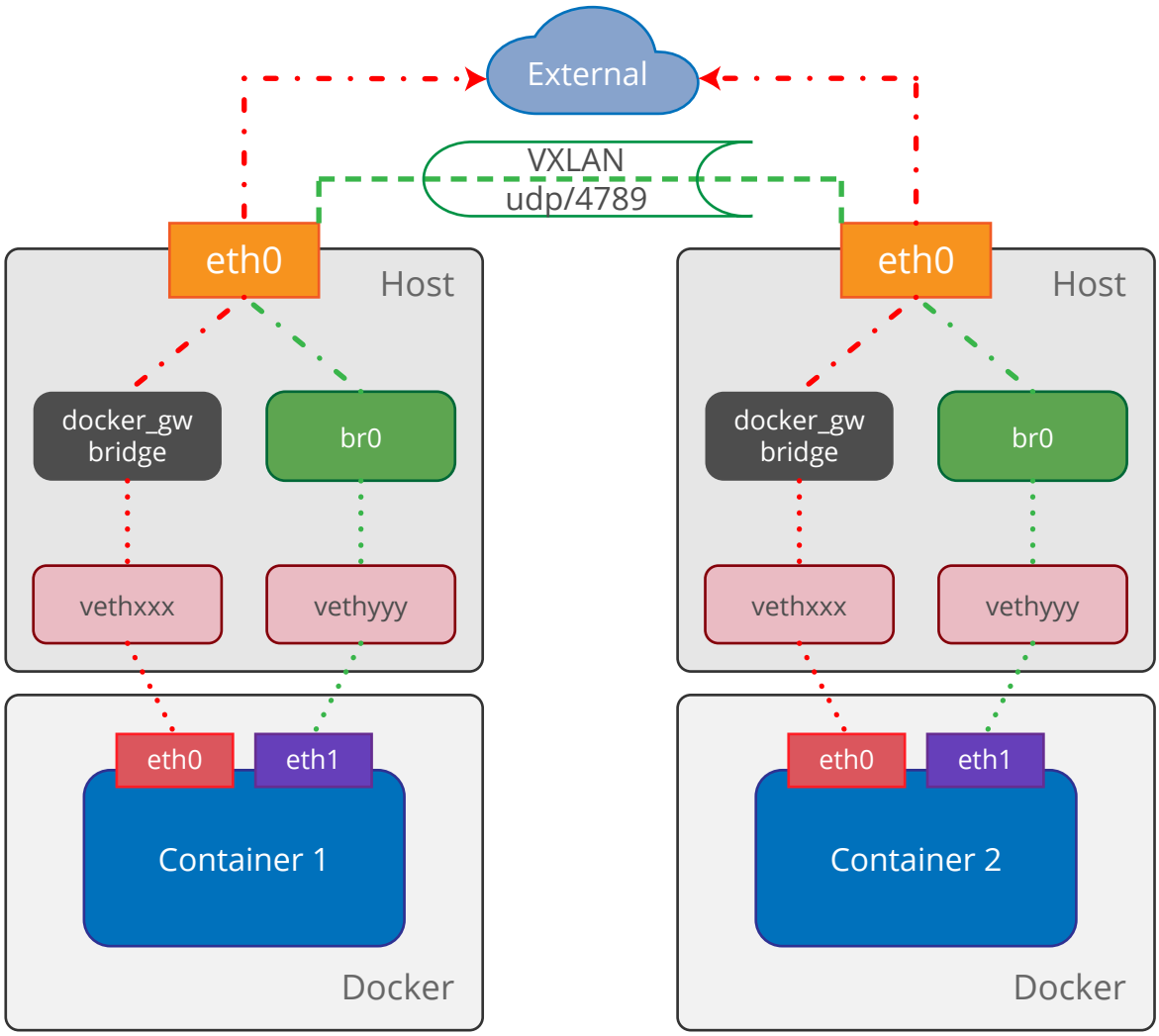
Only containers within the same network can communicate with each other.

4.3. Multi-host Network

Docker has its own multi-host networking out-of-the-box called overlay network, which comes together with Docker Swarm starting version 1.12.

4.3.1. Default Overlay

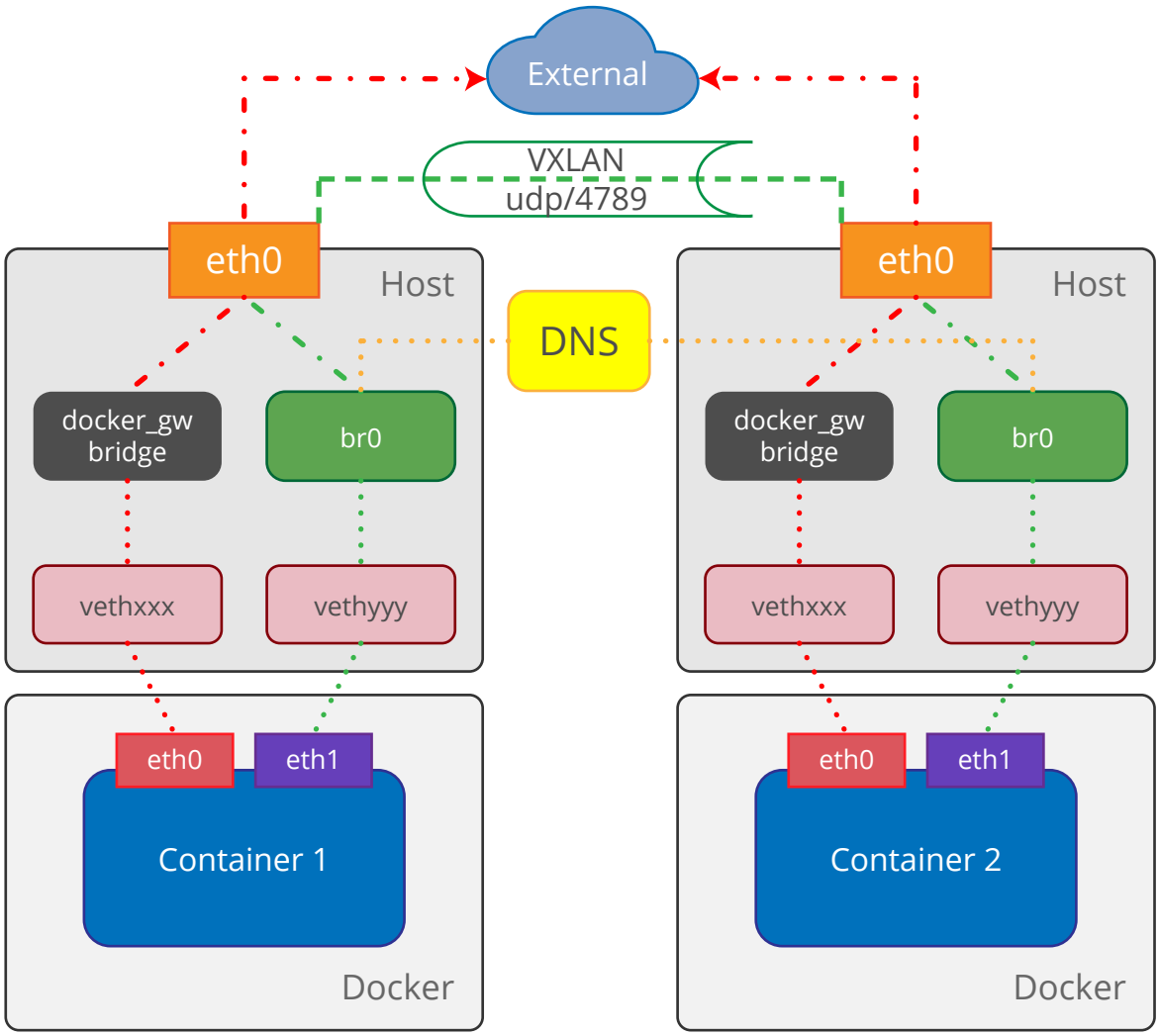
Overlay network is only available if you activate the Swarm mode. You can do that by running the **`docker swarm init`** command, then create a service using **`docker service`** command.



In the default overlay network, each container will have two network interfaces, eth0 and eth1. One is connected to ***docker_gwbridge*** as shown in black, while the other is connected to VXLAN tunnel network through another bridge, represented in green. The one that is connected to ***docker_gwbridge*** is the interface that faces the public network. If you were trying to ping google.com, the container would go through eth0. If from Container 1, you try to ping container 2 through eth0, you won't get any reply. The other interface eth1 is solely for container-to-container communication. You would usually get two IP addresses for the overlay network interface. One is the container IP address or in Swarm term "Task IP address", and another one is the Service VIP or virtual IP address.

4.3.2. User-defined Overlay

User defined overlay network is similar to the default overlay, plus it comes with an embedded DNS resolver.



This is the preferred network when deploying a MySQL container as a Swarm service in a multiple Docker host environment due to its isolation and it's ability to resolve the container's name. Docker orchestration is out of the scope of this whitepaper.

4.4. Network Plugins

Similar to Docker volume plugins, Docker network can be extended to support a wide range of networking technologies. The network can be extended to run in multi-host networking mode, to support name resolver and service discovery, and to even support encryption. The following are the most popular network plugins for Docker, that you can use in a clustered system:

- Contiv
- Calico
- Weave
- Flannel

To summarize our options, Calico is the winner performance wise because it operates on Layer 3 of the OSI layer. Weave is easy to configure and comes with DNS service. Flannel and Contiv have been a long time on the market, and are pretty stable. Each of

them has its advantages and disadvantages, and it is strongly recommended to test and compare, based on your workload.

4.5. Accessing the MySQL Containers

There are several ways to access the created MySQL containers:

- via container's exposed port,
- via container's published port,
- by attaching to the active container.

4.5.1. Container's exposed port

From the example in the previous section, we did not specify any networking parameters which means Docker will create the container as per default bridge network (docker0) without host-port mapping (`--publish`, or `-p`). To verify, use the `docker ps` command:

```
1  $ docker ps
2  CONTAINER ID        IMAGE               COMMAND
   CREATED             STATUS              PORTS
   NAMES
3  8864688449d4        mysql:5.7.20        "docker-entry-
   point..."   About an hour ago   Up About an hour    3306/tcp
   my-test
```

Under PORTS column, we can tell that port 3306 is exposed for this container (as defined inside Dockerfile for this image, or through `--expose` flag) without any mapping to the host network. The exposed port is only accessible by containers within the same Docker network (docker0).

4.5.2. Container's published port

To access a MySQL container created inside the bridge network to the outside world, you have to publish the exposed MySQL port accordingly. Publish in Docker's term means Docker will create a one-to-one mapping between the host port and the container port. Run the following command instead when running the container:

```
1  $ docker run -d \
2  --name my-test \
3  --publish 8000:3306 \
4  --env MYSQL_ROOT_PASSWORD=mypassword \
5  mysql
```

To verify the port mapping from the host to the container, use `docker ps` command:

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND
CREATED             STATUS              PORTS
NAMES
2198fb16b979        mysql:5.7.20        “docker-en-
trypoint...”   24 seconds ago      Up 23 seconds
0.0.0.0:8000->3306/tcp   my-test
```

Under PORTS column, we can see that port 3306 exposed for this container (as defined inside the Dockerfile for this image, or through `--expose` flag) is mapped to port 8000 of the host network. The MySQL instance in this container is now remotely accessible within Docker bridge network on port 3306 and port 8000 to the external network. You can also use `--publish-all` or `-P` to publish all exposed ports to random ports automatically. If the container is running on the host network with `--net=host`, the exposed port will automatically become the published port.

4.5.3. Attaching to the active container

You can also attach to the running container directly by using the `docker exec` command. The following command allows us to attach to the container as a bash terminal:

```
$ docker exec -it my-test /bin/bash
root@8864688449d4:/#
```

You can then execute the necessary commands just like the normal terminal access does. You can also pass any shell command to be running inside the container. The following command executes a MySQL client within the container directly from the host console:

```
$ docker exec -it mysql-test mysql -uroot -p -e ‘SELECT @@
innodb_buffer_pool_size’
Enter password:
+---------------------------+
| @@innodb_buffer_pool_size |
+---------------------------+
|                 134217728 |
+---------------------------+
```

This method is similar to treating the container as an independent host, and is usually used by the administrators who have local access to the Docker host. You don’t have to know the exposed or published ports for this container, it is enough to know the container’s name or ID.

MySQL Containers and Volumes

A container is a running instance of Docker images. MySQL is a disk-based relational database management system (RDBMS) and in order to run a MySQL container, one has to understand how Docker stores and manages its data.

5.1. Running a Single MySQL Container

Let's take a look at the simplest command to run a MySQL container:

```
$ docker run -d \
--name my-test \
--env MYSQL_ROOT_PASSWORD=mypassword \
mysql
```

The `docker run` command is basically a combination of three docker commands:

1. `docker pull` to pull the container image. In this example, we are using the standard MySQL image published and maintained by Docker, available on Docker Hub. The image name is "mysql" (a shortened alias of "docker.io/mysql:latest").
2. `docker create` to create a container based on the variable defined. This MySQL image is a general-purpose image, in which you can define a number of environment variables to suit your needs. This image requires at least a MYSQL_ROOT_PASSWORD environment variable to be set during startup (--env MYSQL_ROOT_PASSWORD). Details at Docker Image's documentation page at Docker Hub.
3. `docker start` to launch the container.

In this example, we did not define any volume (`--volume` or `-v`). If a container is running without a volume, all the directory or file modifications in the container will be written in the container layer, which is the upper layer on top of the image layers, as illustrated in the following diagram:

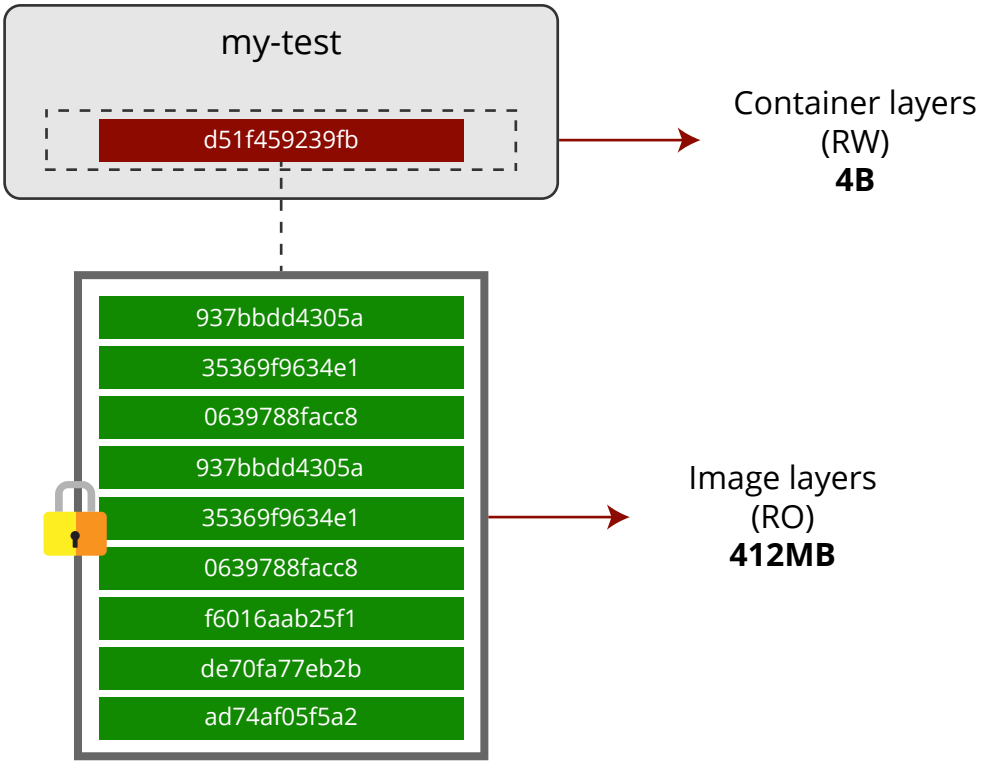


Image: docker.io/mysql:latest

The image layers (shown in green) are read-only while the container layers (red) are writable. The writable layer is also known as thin-pool layer. These two layers run on union filesystem, depending on the Docker host operating system. For Ubuntu, it defaults to AUFS. For RHEL/CentOS, it defaults to overlayfs or overlayfs2. If you use Docker EE on RHEL, CentOS, or Oracle Linux, you must use the devicemapper storage driver with direct-lvm approach, as this is the only supported storage driver by Docker in a production system.

The container layer only utilizes around 4 bytes of disk space once started and you can verify this using `docker ps -s` command:

```
$ docker ps -s
CONTAINER ID  IMAGE  NAMES           SIZE
d51f459239fb  mysql  my-test         4B (virtual 412MB)
```

5.2. Running Multiple MySQL Containers

By using the same image, we can create another MySQL container with a different name (the container name must be distinct in the same Docker host):

```
$ docker run -d \
--name my-test2 \
--env MYSQL_ROOT_PASSWORD=mypassword \
mysql
```

When creating another MySQL container using the same image, you are actually creating a new writable layer to store data for the container. Each container has its own writable container layer, which means multiple containers can share access to the same underlying image and yet have their own data state.

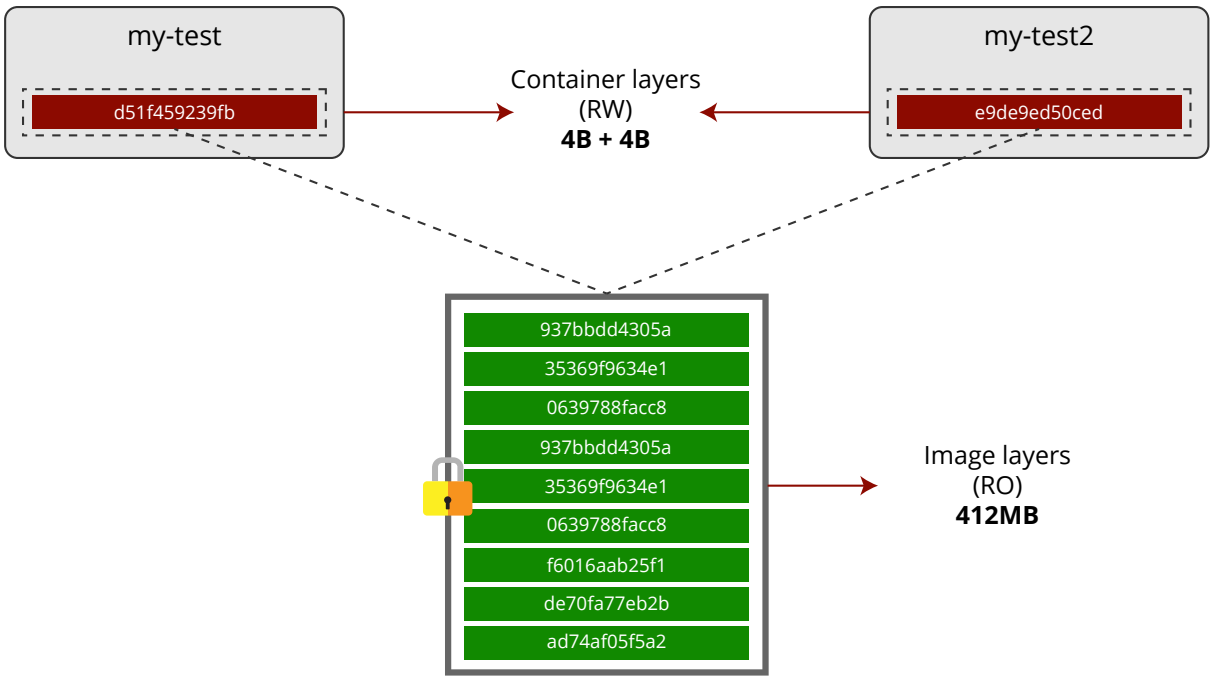


Image: docker.io/mysql:latest

In this example, these two containers occupy 8 bytes of disk space once started with two independent MySQL instances. The container layer will grow once we start to make changes to the container, for example, if we create a new database, add new tables and start inserting rows of data.

When you remove a container, you are basically removing the writable layer. Thus all the changes inside the container layer will be removed as well. This layering and unification technique is very space efficient, minimizes IO operations on the host and supports caching.

5.3. Container Layer Changes

You can imagine the layers as directories. When you modify an existing file, for example, `/etc/mysql/my.cnf`, the container will first look up this file in the container layer. If the file does not exist, it will go through the image layers starting from the top and look for this file. It then copies the file from the image layer to the writeable container layer. Docker will then write the changes to the new copy of the file in the container layer, as shown in operation 1 in the diagram below. This is known as Copy-on-Write (CoW) operation.

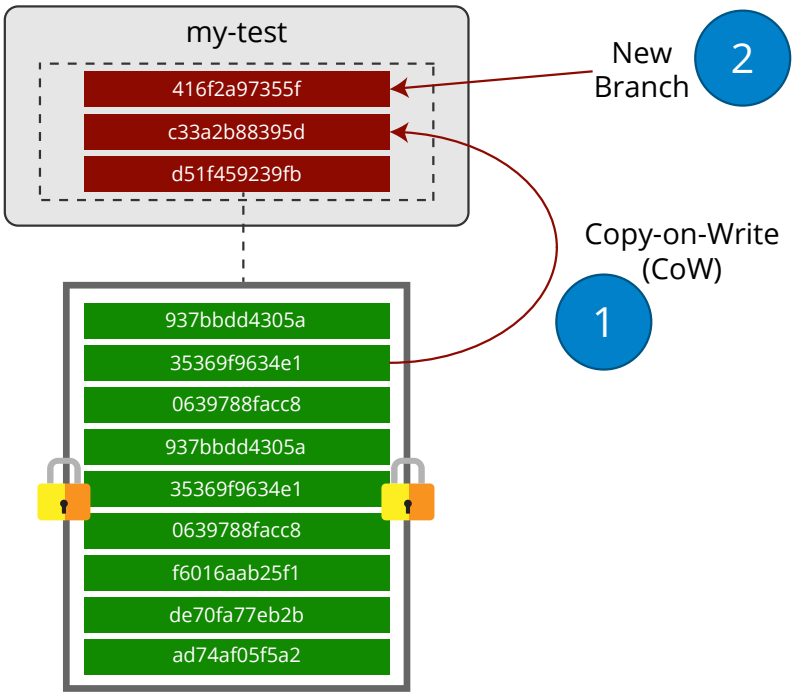


Image: docker.io/mysql:latest

In example operation number 2, we are trying to create a new schema and table, which means a new directory and file will be existed in the container layer. When you create a new file inside a container, Docker will create a directory, or "branch" in AUFS terminology with a random SHA-256 hash name. It will then place the file inside that directory with metadata including a pointer to the parent directory. You can see this file directly inside `/var/lib/docker/aufs`.

Docker supports a number of CoW filesystems and devices like overlayfs, btrfs, device mapper and ZFS. Every storage driver handles the implementation differently, but all drivers use stackable image layers with this Copy-on-Write strategy.

5.4. MySQL Persistency

In MySQL, the most important path is the datadir, and by default it is under `/var/lib/mysql`. This is the default data directory path in MySQL, and you can customize this at a later stage. When running MySQL on multiple containers, you would expect them to run in a more uniform and organized way, minimizing the complexity of your MySQL configuration options by sticking to the default options if possible.

The following table shows the path required by MySQL for persisting data in a container:

Description	Variable Name	Default Value (5.7)
MySQL data directory	datadir	/var/lib/mysql
MySQL plugin directory	plugin_dir	{basedir}/lib/plugin
MySQL configuration directory	config_dir	/etc/my.cnf.d
MySQL binary log	log_bin	{datadir}/{hostname}-bin

Description	Variable Name	Default Value (5.7)
Slow query log	slow_query_log_file	{datadir}/{hostname}-slow.log
Error log	log_error	{datadir}/{hostname}.err
General log	general_log_file	{datadir}/{hostname}.log

Most of the files or the directory generated by MySQL are default to datadir. Apart from those, if you would like to use your custom my.cnf file, you should preserve `/etc/my.cnf.d`, the MySQL configuration directory and store your my.cnf under this directory. If you are using any other MySQL plugins, you might also want to preserve `{basedir}/lib/plugin` directory. This directory can be mounted as RO using the `--mount` flag in Docker. This is the recommended way to prevent the MySQL plugin from executing any arbitrary code.

As you can see here, hostname is an important aspect of logging. Try to stick with the same container hostname, especially when running another container, to avoid confusion later on. Otherwise, you can use a custom filename for every logging option.

5.5. Docker Volume

Since Docker 1.9, it is possible to store data persistently in a named volume using the "local" volume driver. Before that, only bind mounts were supported. The problem using bind mounts is that it is very dependent on the host. If you wanted to mount existing data, you would need to make sure that path existed on the host. One good thing about bind mounts is that you can mount a file instead of a directory.

Using named volumes is much simpler and more independent. Docker creates a directory under `/var/lib/docker/volumes` and the volume can be referred to by other containers using the volume's name, so you don't have to remember the static physical path. The volume contents exist outside of the container lifecycle. Since Docker manages the directory's content of the named volume, it can be extended with an external volume plugin instead of relying on the local volume driver.

This opens the door for stateful applications to run efficiently in Docker. You don't have to worry about container upgrades, or the container being killed or crashing. If you run with persistent storage mounted inside the container, another container can pick up from the last stored state and continue from there.

You can also have another volume dedicated to storing non-persistent data in RAM (or swap if memory is low). MySQL can leverage this type of volume for temporary data directory (tmpdir).

5.5.1. Persistent Volume

To store data persistently, we would have to configure some extra options related to volume. You can create the volume first, using the `docker volume create` command. You can also just directly specify a volume flag when running the `docker run` command. Docker will then create the volume if it does not exist.

To run a container with a persistent named volume:

```
$ docker run -d \
--name mysql-local \
-p 3308:3306 \
-e MYSQL_ROOT_PASSWORD=mypassword \
-v local-datadir:/var/lib/mysql \
mysql:5.7
```

The command tells Docker to create a volume inside of the `/var/lib/docker/volumes` directory on the host filesystem if it does not already exist, and mount it inside the container under the MySQL data directory path, which is `/var/lib/mysql`. The data volume is accessible directly from the host, even while the container is running.

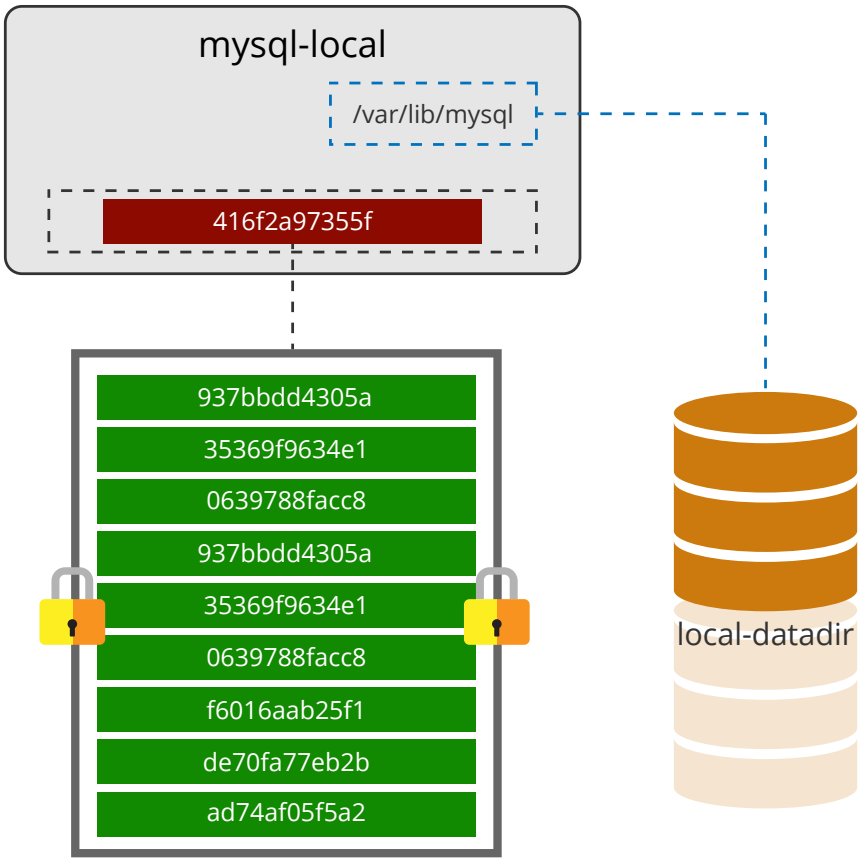


Image: docker.io/mysql:5.7

Volumes are often a better choice to persist your data, instead of using the container's writable layer, because using a volume does not increase the size of the containers using it. The contents of the volume exists outside the lifecycle of the container. Starting from version 17.06, you can also use `--mount` which is much more verbose. The following command does the same thing:

```
$ docker run -d \
--name mysql-local \
-p 3308:3306 \
-e MYSQL_ROOT_PASSWORD=mypassword \
-v local-datadir:/var/lib/mysql \
mysql:5.7
```

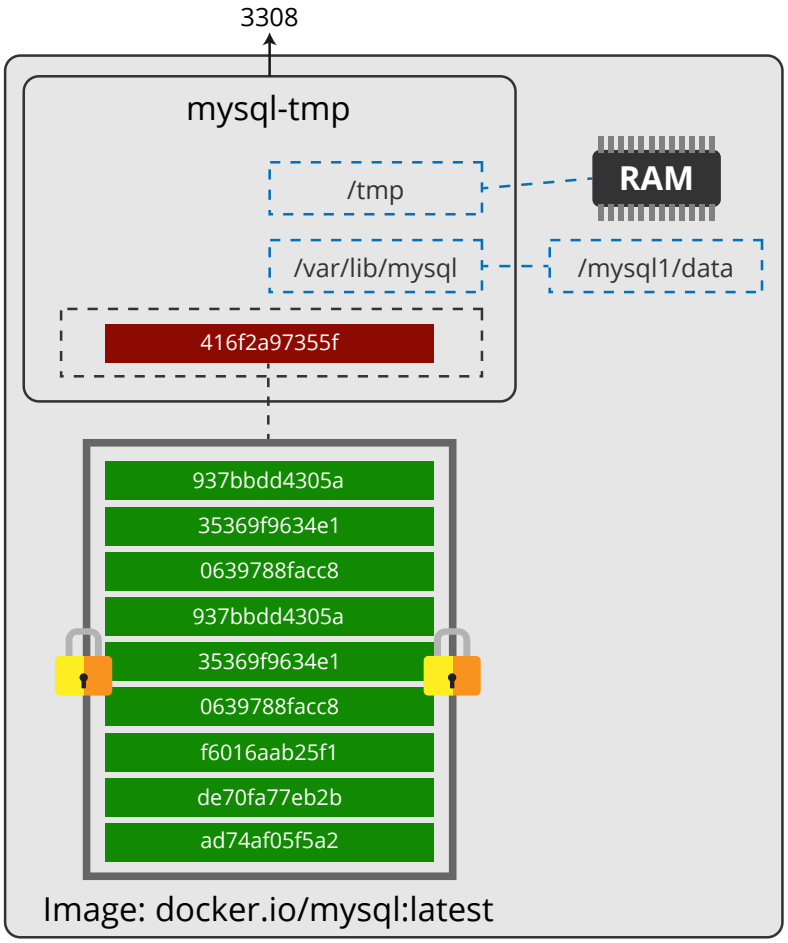
Originally, the `--mount` option was meant for Swarm services and not for standalone containers. With mount, the order of the key value pairs is not significant, and the value of the flag is easier to understand.

5.5.2. Non-Persistent Volume

MySQL also generates non-persistent data like temporary files with the TMPDIR variable. By default, MySQL will use `/tmp`, `/var/tmp` or `/usr/tmp`. Some SELECT statements, especially when sorting using GROUP BY or ORDER BY, create temporary tables. You would see a file prefixed with 'SQL_' created during the operation. You can mount this directory inside the container using the tmpfs volume instead, which is only stored in the host machine's memory (or swap, if memory is low) similar to mounting a ramdisk:

```
1  $ docker run -d \
2  --name mysql-tmp \
3  -p 3308:3306 \
4  -e MYSQL_ROOT_PASSWORD=mypassword \
5  -v /mysql1/data:/var/lib/mysql \
6  --tmpfs /tmp:rw,size=1g,mode=177 \
7  mysql
```

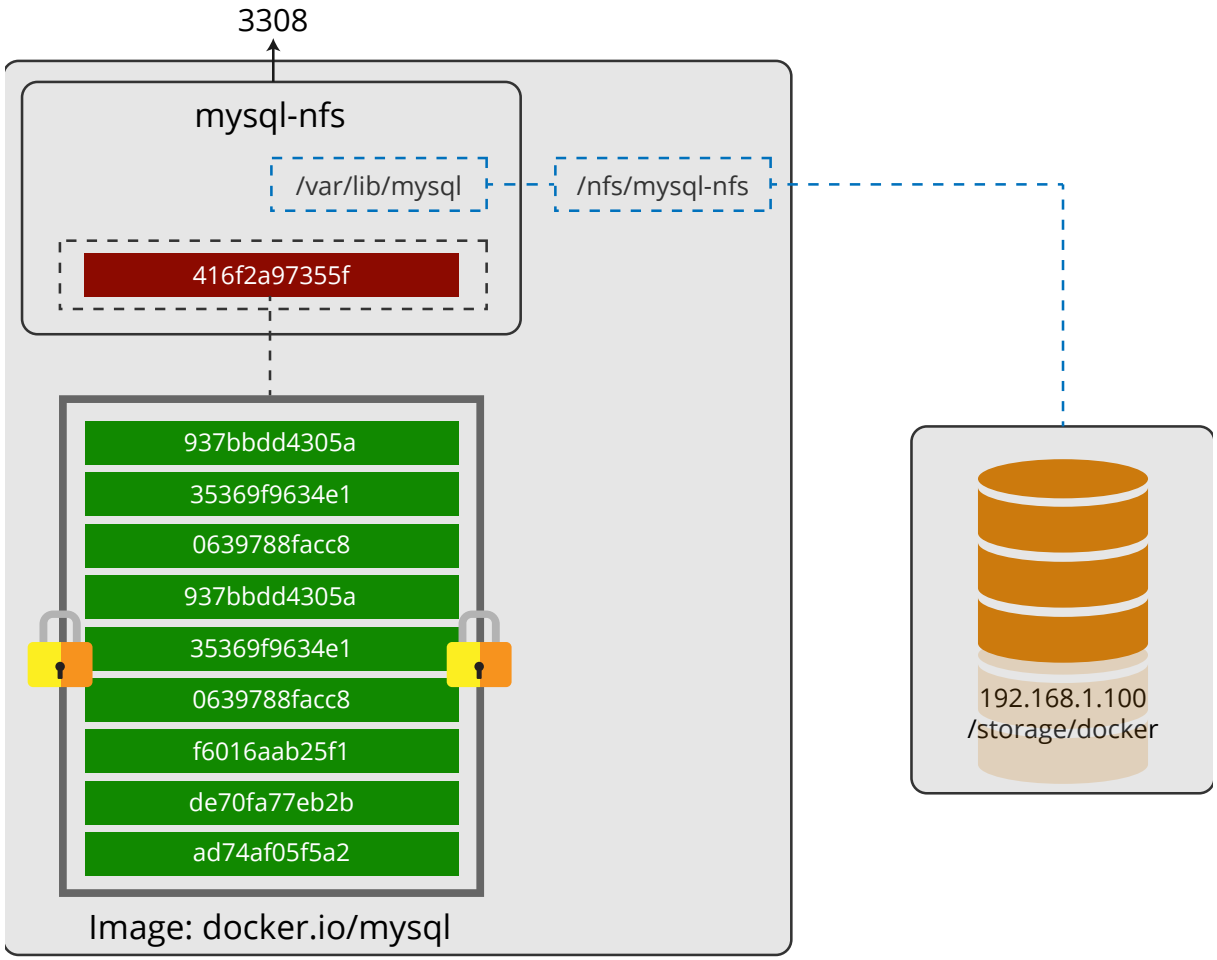
When the container stops, the tmpfs mount is removed. If a container is committed, the tmpfs mount is not saved. The following diagram illustrates a MySQL containers with both persistent and non-persistent volumes:



However, there is one caveat. Beware of large sort operations that could occupy more than the limit, because when this happens, the host would start swapping. To avoid this, you could also use the standard named volume for `/tmp`. This feature must be used with caution and only works for Linux containers, not for Windows.

5.5.3. Remote Volume

What if the physical host itself crashes and somehow gets corrupted? You would lose your container as well as the persistent volume. To recover from this scenario, we can use a volume outside of the local host. The simplest setup would be using network file system (NFS) which allows you to export a directory to another server via network. The following diagram illustrates a remote volume setup with NFS:



The remote directory can be mounted locally on the client host, which is the Docker host:

```
1   $ mount 192.168.1.100:/storage/docker /nfs -o noatime,nodi-
    ratime,hard,intr
```

You can then use the bind mount approach to mount the directory inside the Docker container:

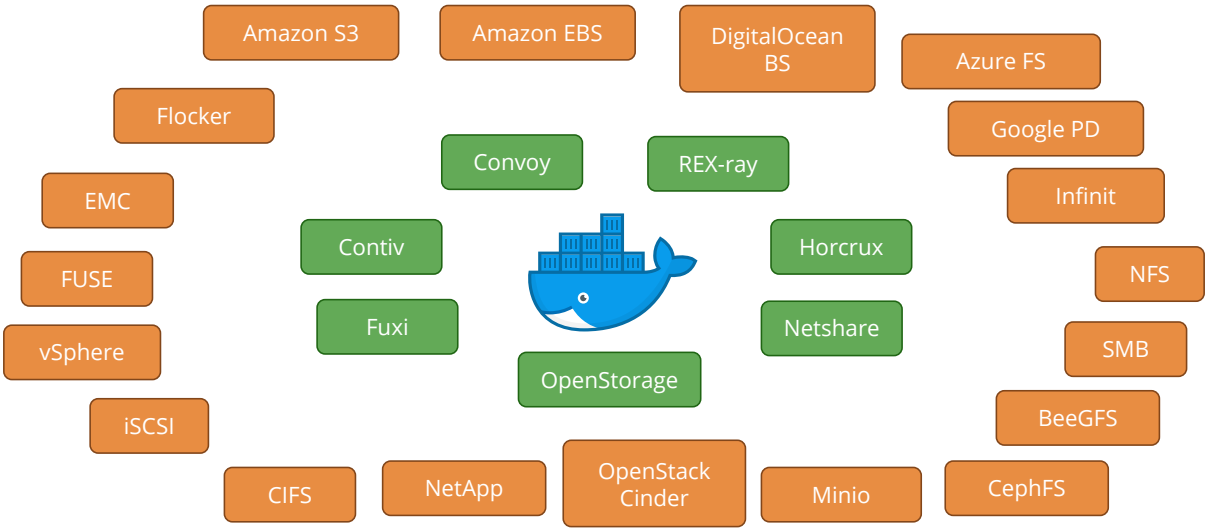
```
$ docker run -d \
--name mysql-nfs \
-p 3308:3306 \
-e MYSQL_ROOT_PASSWORD=mypassword \
-v /nfs/mysql-nfs:/var/lib/mysql \
mysql
```

In this example, we mounted the NFS path in the Docker host under `/nfs` directory, from a remote host, 192.168.1.100. Also note that there are a number of considerations when running MySQL on NFS. See Using NFS for MySQL for details.

5.6. Volume Drivers

By default, the storage driver is local, which means if you create a named volume, the volume will be created on the Docker host where the container resides. If you have multiple Docker hosts and you would expect them to scale out in the near future, you should use the volume drivers plugin to reduce the complexity of provisioning each Docker host as the storage client.

The following diagram shows Docker volume drivers and storage platforms:



The orange boxes in the outer layer are storage platforms or file systems that are supported by various Docker volume plugins, the latter are represented in green. There are lots more, you can view a more complete list on Docker documentation page.

Install the plugin you want to use. In this example, we'll use REX-Ray, an open-source volume driver plugin for Docker. First, install the volume plugin for EBS:

```
$ docker plugin install rexray/ebs EBS_ACCESSKEY=XXXX EBS_
SECRETKEY=YYYY
```

Then, verify if the driver is correctly installed. We use another tool here called 'jq', a JSON query tool to beautify the output:

```
$ docker info -f '{{json .Plugins.Volume}}' | jq
[
  "local",
  "rexray"
]
```

Run a container by specifying the `--volume-driver` flag for this driver:

```
$ docker run -d \
--name=mysql-ebs \
--volume-driver=rexray/ebs \
-v ebs-datadir:/var/lib/mysql \
mysql:5.7
```

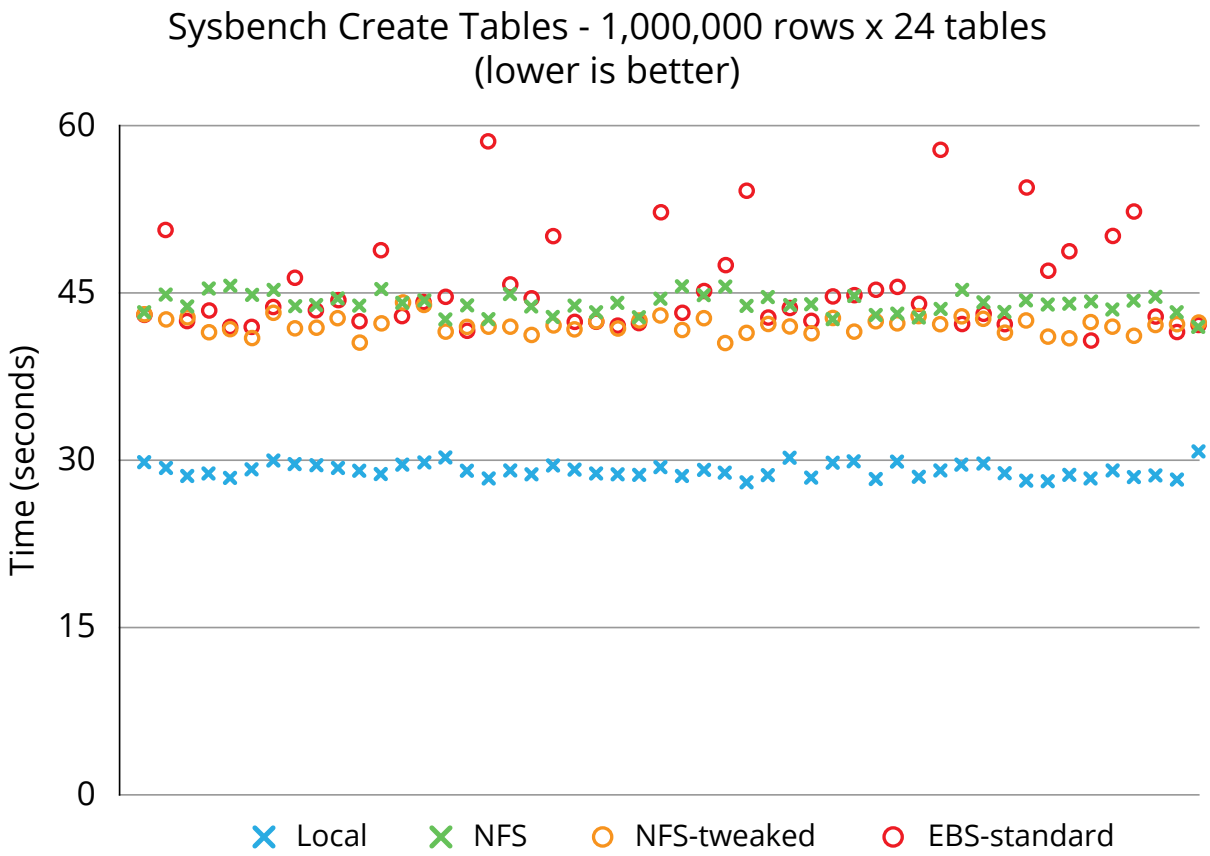
Verify the volumes maintained by Docker:

```
$ docker volume ls
DRIVER              VOLUME NAME
local               local-datadir1
local               local-datadir2
rexray              ebs-datadir
```

You can then manage the volume using the `docker volume` command directly. When you delete the volume, the volume driver itself will do the necessary cleanup for you. Volume drivers also allow you to extend Docker volume capabilities. For instance, some storage systems support encryption-at-rest, volume snapshots and backups.

5.7. Performance Tradeoff

What is the tradeoff of having MySQL database mounted through remote storage? The main one is ***performance***. Don't expect it to be as good as local disks or direct-attached storage on the Docker host, especially if the remote storage is shared among multiple hosts, which can make it a busy endpoint.



Here is a simple benchmark comparing the local disk and other remote storage with a simple create table job using sysbench. The tests were repeated for 50 times on every storage driver. You can see the green dots which represent performance of remote NFS storage is around 40% slower than the local disk. With some further tweaking using a couple of NFS mount options like `noatime`, `nodiratime`, `hard`, `intr`, we could get around 10% improvement over the default NFS as shown by the orange dots. The red dots represent Amazon Elastic Block Storage (EBS) volume without provisioned IOPS, where the results are pretty inconsistent.

As conclusion, using a remote volume should be sufficient if database performance is not critical, for instance, in testing and development environments. Do not forget to benchmark on your storage system and understand its performance.

Monitoring and Management

6.1. Service Control

If you look at the entrypoint script for the MySQL image, you should see it executes the mysqld process as a foreground process at the very last line with **exec** command. The following snippet is from the end of the entrypoint script used in the MySQL image:

```
...
...
exec mysqld "$@"
```

This is to ensure that the mysqld process holds process ID 1 when the container is started. The PID 1 has a special meaning in a container. When stopping a container, only the process with PID 1 will receive the signal. Take a look at this example:

```
$ docker exec -it mysql-test ps -e
   PID TTY          TIME CMD
     1 ?        00:00:00 mysqld
    62 pts/0    00:00:00 bash
    86 pts/0    00:00:00 ps
```

When we run **docker stop** command, Docker will send a signal to the container called SIGTERM, to indicate process termination. Only the mysqld process will receive it, so it will start terminating and perform any clean up required to shut down in a graceful manner. Other processes like bash (pid 62) and ps (pid 86) will not receive this signal.

However, using **docker stop** is risky, because there is a risk that Docker will not allow proper termination of the mysqld process. By default, **docker stop** command will send SIGTERM and wait for 10 seconds, before it sends a SIGKILL to the container to force for termination if it's still running. You can change the grace period to something higher.

In some cases, the MySQL termination process requires more time. MySQL, after receiving a SIGTERM, will:

1. stop receiving new connections,
2. then complete executing whatever queries are still pending (this can take a while, depending on your workload and number of concurrent clients)
3. then start flushing data to disk (this can take many seconds, again depending on your workload, configurations, available memory, and storage engine choices),
4. after all the data-flushing is done, MySQL will then deallocate whatever memory it allocated during the initialization stage (this can also take a while, depending on the amount of memory allocated by MySQL),
5. close all file handles still open, and then call "exit(0)".

To be safe when performing a graceful shutdown, use `docker kill` command with a proper termination signal. This command doesn't have a grace period so it won't interrupt the process cleanup operation by mysqld:

```
$ docker kill --signal=TERM mysql-test
```

You can then verify the termination status by looking at the container's log. Ensure you got the "`[Note] mysqld: Shutdown complete`" at the end of the line. Stopping a container won't delete the container, so you can start it up again using `docker start` command.

6.2. Resource Control

By default, there are no resource constraints when you run a Docker container. Docker allocates as much of a given resource as the host's kernel will allow.

Memory is a very important resource in MySQL. This is where the buffers and cache are stored. It is a critical resource for MySQL, as it reduces the impact of hitting the disk too often. On the other hand, swapping is bad for MySQL performance. If `--memory-swap` is set to the same value as `--memory`, and `--memory` is set to a positive integer, the container will not have access to swap. If `--memory` and `--memory-swap` are set to the same value, this will prevent containers from using any swap. This is because `--memory-swap` is the amount of combined memory and swap that can be used, while `--memory` is only the amount of physical memory that can be used. If `--memory-swap` is not set, container swap defaults to `--memory` multiply by two.

You can also increase the limit of open file descriptors, or "nofile" to something higher. This is to cater for the number of files that the MySQL server can open simultaneously with `--ulimit` parameter. Setting this number a bit high is recommended. Example of the `docker run` command with a recommended configuration for memory, swap and open files:

```
$ docker run -d \
--name mysql-staging \
--memory 4g \
--memory-swap 4g \
--ulimit nofile=16824:16824 \
mysql:5.7.6
```

Some of the container resources like memory and cpu can be controlled dynamically (without restart) through `docker update`, as shown below:

```
$ docker update \
--memory 6g \
--memory-swap 6g \
mysql-staging
```

It is important not to forget to tune the MySQL configuration parameters accordingly, so you can take advantage of the resource allocation to the container.

6.3. Resource Monitoring

Docker provides a way to summarize the container resource consumption through `docker stats` command. It can report in real-time similar to the `top` command. The following command shows example of using the `docker stats` command with custom output formatting:

```
$ docker stats --format “table {{.Name}}\t{{.CPUPerc}}\t{{.MemUsage}}”
NAME          CPU %               MEM USAGE / LIMIT
minio1        0.00%               6.078MiB / 7.78GiB
mysql2        0.12%               194.1MiB / 7.78GiB
mysql-local   0.14%               220MiB / 7.78GiB
```

The `docker stats` command is an interface to the stats application program interface (API) endpoint. The stats API exposes all of the information in the stats command and more. To see for yourself, run the following command on Docker host:

```
$ curl --unix-socket /var/run/docker.sock http:/containers/{container_name}/stats
```

The output of the above command is wrapped in a JavaScript Object Notation (JSON) array, which is ready to be ingested by third-party tools.

Inside a container, `free` and `top` commands are not accurate because these tools rely on the value reported under `/proc` on the host. To monitor these resources correctly, you have to see the cgroup directory. These paths are mounted as read-only in the container as well if you run the container in unprivileged mode:

- Memory - `/sys/fs/cgroup/memory/memory.*`
- CPU - `/sys/fs/cgroup/cpu/cpu.*`
- Disk IO - `/sys/fs/cgroup/blkio/blkio.*`

There are also external open-source command line tools like sysdig and dockviz. Sysdig hooks into the host’s kernel, which means it doesn’t entirely rely on getting metrics from the Docker daemon. It also allows you to add orchestration context by hooking directly into the orchestrator, thereby allowing you to troubleshoot by container resources like pod, cluster, namespace and service.

Container visibility and monitoring are much better with graphical user interface. There are a number of open-source tools you can use to monitor the Docker resources such as cAdvisor, Portainer, Shipyard, Rancher and Prometheus. cAdvisor and Portainer can be considered the most popular, super lightweight and easy to install. Most of them can be installed and running with a one-liner command. These tools make use of Docker unix socket file to connect to the daemon process and retrieve monitoring data.

You can also opt for paid solution vendors like New Relic, Dynatrace, Datadog and CoScale. Most of them provide full-stack monitoring and alerting service that monitors everything from the server and container resources, up to application performance.

6.4. Configuration Management

Most of MySQL configuration parameters can be changed during runtime, which means you don't need to restart to load up the changes. Check in the MySQL documentation page for details. If the parameter changes require restarting MySQL, set the changes inside `/etc/my.cnf.d/my.cnf` on the host, and map the directory to a named volume like in the example below:

```
$ docker run -d \
--name mysql2 \
-e MYSQL_ROOT_PASSWORD=mypassword \
-v mysql2-conf:/etc/my.cnf.d \
mysql:5.7.6
```

You can also use the bind mount approach to map a file on the host directly into the container:

```
$ docker run -d \
--name mysql2 \
-e MYSQL_ROOT_PASSWORD=mypassword \
-v /root/docker/mysql2/config/my.cnf:/etc/my.cnf \
mysql:5.7.6
```

Also, you can append the configuration options, right after the image name, passed as flags to the mysqld process inside the container:

```
$ docker run -d \
--name mysql3 \
-e MYSQL_ROOT_PASSWORD=mypassword \
mysql:5.7.6 \
--innodb_buffer_pool_size=1G \
--max_connections=100
```

Generally, standard MySQL and InnoDB optimization applies to the container, for example, ***innodb_buffer_pool_size***, ***innodb_log_file_size*** and ***max_connections***. This is highly dependent on the resource allocated to the containers (as explained in ***Chapter 6.2 - Resource Control***), as well as the underlying disk sub-system (as explained in ***Chapter 5 - MySQL Containers and Volumes***).

6.5. Security

Docker Secrets was introduced in v17.1 to handle sensitive information during container runtime. If you look at the previous example commands, we usually have to define an environment variable called `MYSQL_ROOT_PASSWORD` equal to your root password, in clear text format. This method exposed the password in clear text and can be traced through `docker inspect` or using Linux history command.

To create a secret, use the `docker secret create` command and send the sensitive data through stdin. Then, when running a container, specify the secret name to be mounted under `/run/secrets` directory as a file. To create a secret, simply:

```
$ echo 'MyP2s$w0rD' | docker secret mysql_root_passwd -
```

The image that we used here supports Docker Secrets, so we can define it through the `MYSQL_ROOT_PASSWORD_FILE` environment variable and the image will read the secret file and pass it as the MySQL root password when initializing the container:

```
$ docker service create \
--name mysql-container \
--publish 4406:3306 \
--secret mysql_root_passwd \
-e MYSQL_ROOT_PASSWORD_FILE=/run/secrets/mysql_root_passwd \
-v local-datadir:/var/lib/mysql \
mysql:5.7
```

Another security aspect that we have to look at is the runtime privileges. There are two types of runtime privileges:

- Unprivileged mode is default and recommended.
- Privileged mode.

6.5.1. Unprivileged Container

This is the default and recommended option. The container does not have access to other devices, nor is able to modify kernel features. All of these devices are mounted as read-only, you can verify from the mount list in the container using the following command:

```
$ docker exec -it mysql-container mount | grep ro,
sysfs on /sys type sysfs (ro,nosuid,nodev,noexec,relatime)
tmpfs on /sys/fs/cgroup type tmpfs (ro,nosuid,nodev,noex-
ec,relatime,mode=755)
...
...
cgroup on /sys/fs/cgroup/memory type cgroup (ro,no-
suid,nodev,noexec,relatime,memory)
cgroup on /sys/fs/cgroup/freezer type cgroup (ro,no-
suid,nodev,noexec,relatime,freezer)
proc on /proc/bus type proc (ro,relatime)
proc on /proc/fs type proc (ro,relatime)
proc on /proc/irq type proc (ro,relatime)
proc on /proc/sys type proc (ro,relatime)
proc on /proc/sysrq-trigger type proc (ro,relatime)
tmpfs on /proc/scsi type tmpfs (ro,relatime)
tmpfs on /sys/firmware type tmpfs (ro,relatime)
```

6.5.2. Privileged Container

On some occasions, you would probably need to modify the kernel parameters like sysctl.conf or `/proc` to suit your database needs and workloads. In that case, you have to start the container with `--privileged` flag. Docker will mount all devices with read-write, so you can make those changes accordingly. With `--privileged` mode, the container has almost all the capabilities of the host machine.

Use `--privileged` with caution, and only if you really need to modify kernel parameters (sysctl, /proc, /sys) or you want to run a container inside another container.

6.6. Backup and Restore

6.6.1. Backup

Taking a logical backup is pretty straightforward because the MySQL image also includes mysqldump and mysqlpump (MySQL 5.7.8+). You simply use the `docker exec` command to run mysqldump and redirect the stdout output to a path on the host:

```
$ docker exec -it mysql-prod mysqldump -uroot -p --single-transaction > /path/in/physical/host/dump.sql
```

Binary backup like Percona Xtrabackup and MariaDB Backup requires the process to access the MySQL data directory directly. You have to either install these tools inside the container, or through the machine host or use a dedicated image for it. The following example shows how to perform an xtrabackup backup, using "perconalab/percona-xtrabackup" image to create the backup and store it inside `/tmp/backup` on the host:

```
$ docker run --rm -it \
--net db-prod \
-v mysql-datadir:/var/lib/mysql \
-v /tmp/backup:/xtrabackup_backupfiles \
perconalab/percona-xtrabackup \
--backup --host=mysql-prod --user=root --password=mypassword
```

You can also apply a global lock or stop the container and just copy over the data volume to another location, as shown in the following example:

1. Flush all tables to disk:

```
$ docker exec -it mysql-prod /bin/bash
root@mysql-prod:# mysql -uroot -p
mysql> FLUSH TABLE WITH READ LOCK;
```

2. Copy the volume to another location in another terminal:

```
$ cp /var/lib/docker/volumes/mysql-datadir/_data /destination/in/physical/host
```

3. In the same terminal as in step 1, release the lock:

```
mysql> UNLOCK TABLES;
```

6.6.2. Restore

For mysqldump, restoration is pretty similar to the backup procedure. You can simply redirect the stdin into the container from the Docker host:

```
$ docker exec -it mysql-staging mysql -uroot -p < /path/in/physical/host/dump.sql
```

You can also use the standard mysql client command line remotely with a proper hostname and port value instead of using this `docker exec` command.

For Xtrabackup, you have to prepare the backup beforehand, and the prepared backup then can be used as the mysql datadir in another container:

```
$ docker run --rm -it \
-v mysql-datadir:/var/lib/mysql \
-v /tmp/backup:/xtrabackup_backupfiles \
perconalab/percona-xtrabackup \
--prepare --target-dir /xtrabackup_backupfiles
```

If the backup was taken using the snapshot, it's easy to duplicate the database state and run on multiple containers for testing purposes. With bind mount, you can directly mount the directory as a volume. If you would like to use a named volume, you need to:

1. Create a volume.
2. Copy the content to the volume's directory.
3. Start a new container by mounting the volume.

6.7. Upgrades

You can perform MySQL upgrade operation using two ways - logical or in-place upgrade. Keep in mind that MySQL only supports upgrade from one previous version. If you are on 5.5 and would like to upgrade to 5.7, you have to upgrade to MySQL 5.6 first, followed by another upgrade step to MySQL 5.7. You can use the in-place upgrade to achieve this pretty easily.

6.7.1. Logical Upgrade

Logical upgrade is where you use a MySQL logical backup, like mysqldump or mydumper, to export data from the existing database and load the dump file into the new MySQL version. You can start by pulling down the new MySQL image to the host. Then start a new container with that image. Export the data from the old container using mysqldump and load it into the new container. Once done, run the ***mysql_upgrade*** script inside the new container. If everything is OK, remove the old DB container, otherwise, you can roll back to the previous version by starting the old container.

The logical upgrade procedure is summarized in the following table:

#	Step	Command
1	Pull the new image, M.	`$ docker pull mysql:5.7.18`
2	Start new container, B with new image, M.	`$ docker run -d \ --name=mysql-new \ -e MYSQL_ROOT_PASSWORD=mypassword \ -v local-datadir:/var/lib/mysql \ mysql:5.7.18`
3	Export MySQL on A.	`$ docker exec -it mysql-old mysqldump -uroot -p > dump.sql`
4	Stop A.	`$ docker kill --signal=TERM mysql-old`
5	Import into B.	`$ docker exec -it mysql-new mysql -uroot -p < dump.sql`
6	Run mysql_upgrade inside B.	`$ docker exec -it mysql-new mysql_upgrade -uroot -p`
7	OK? Remove A.	`$ docker rm -f mysqld-old`
8	Fallback? Stop B, start A.	`$ docker stop mysqld-new` `$ docker start mysqld-old`

6.7.2. In-Place Upgrade

In-place upgrade is upgrading the MySQL server without removing the existing MySQL data, beyond normal precautions. The most important part is to perform ***innodb_fast_shutdown*** and to run the ***mysql_upgrade*** script afterwards.

The in-place upgrade procedures are simplified in the following table:

#	Step	Command
1	Pull the new image, M.	`$ docker pull mysql:5.7.18`
2	Set innodb_fast_shutdown=0 inside container.	`$ docker exec -it mysql-old mysql -uroot -p -e 'SET GLOBAL innodb_fast_shutdown = 0'`
3	Stop the container, A.	`$ docker kill --signal=TERM mysqld-old`
4	Start a new container, B with new image, M.	`$ docker run -d \ --name mysql-new \ --publish 3307:3306 \ -e MYSQL_ROOT_PASSWORD=mypassword \ -v local-datadir:/var/lib/mysql \ mysql:5.7.18`
5	Run mysql_upgrade inside B.	`$ docker exec -it mysql-new mysql_upgrade -uroot -p`
6	OK? Remove A.	`$ docker rm -f mysqld-old`
7	Fallback? Stop B, start A.	`$ docker stop mysqld-new` `$ docker start mysqld-old`

6.8. Housekeeping

When working with multiple MySQL containers, you might find that your host machine's disk space starts to run out, especially when you build the image directly on the host. Occasionally, some of the intermediate images might still there. It is safe to remove all dangling images with this command:

```
1 $ docker image prune
```

To clean up unused volumes, you can use the `docker volume prune` command to achieve this. For example, if you would like to remove all unused volumes (volumes that are not associated with any running container any longer):

```
1 $ docker volume prune
```

A similar result can be achieved by pruning the system altogether:

```
1 $ docker system prune -a
```

This will remove:

- all stopped containers,
- all volumes not used by at least one container,
- all networks not used by at least one container,
- all images without at least one container associated to them

Housekeeping has to be done regularly to ensure MySQL has enough resources to run. It's recommended to set up a resource monitoring and alerting system for the Docker host, so you can take proactive actions before you run out of resources.

ClusterControl on Docker

ClusterControl is a management and automation platform for open source databases. It can be used to deploy, manage, monitor and scale highly available clusters. It supports all types of MySQL high availability setups, including MySQL Replication, MySQL Cluster, MySQL Group Replication and Galera Cluster. It also supports MongoDB ReplicaSets and Sharded Clusters, as well as streaming replication for PostgreSQL.

As we discussed earlier, Docker is a credible alternative when running MySQL across multiple environments. A group of MySQL instances can be deployed with ease through Docker. Combine this with ClusterControl to automate the majority of the repetitive database administration tasks, and we have an advanced automation solution for our database infrastructure.

Severalnines, the team behind ClusterControl, has built an image specifically to make ClusterControl run well on Docker. This image comes with ClusterControl and all of its components and dependencies - ClusterControl 1.5 suite, including controller, REST API interface, web user interface, notification and web-ssh package installed via repository. The image is also pre-configured with all applications required by ClusterControl like MySQL, Apache, SSL certificates as well as an SSH key for ClusterControl usage.

7.1. Running ClusterControl as Docker Container

ClusterControl has a short release cycle, usually a new version after every couple of months. So you have to upgrade the image regularly to keep up to date with the latest version. Running with data persistence is the recommended way, to ensure the container's upgrade process works fine. The recommended way to run ClusterControl in a container is to use the user-defined bridge network. Create the network first:

```
$ docker network create --subnet=192.168.10.0/24 db-cluster
```

Then, run the container and specify it under this network, with a static IP address and hostname:

```
$ docker run -d --name clustercontrol \
--network db-cluster \
--ip 192.168.10.10 \
-h clustercontrol \
-p 5000:80 \
-p 5001:443 \
-v /storage/clustercontrol/cmon.d:/etc/cmon.d \
-v /storage/clustercontrol/datadir:/var/lib/mysql \
-v /storage/clustercontrol/.ssh:/root/.ssh \
-v /storage/clustercontrol/backups:/root/backups \
-e CMON_PASSWORD mysecr3t \
-e MYSQL_ROOT_PASSWORD mys3cret \
severalnines/clustercontrol
```

We use supervisord to manage daemons inside the container. So if you would like to perform service control without restarting the container, you have to use supervisorctl command line. The following services inside the container are managed by supervisord:

- sshd
- mysqld
- httpd
- cmon
- cmon-ssh
- cmon-events
- cmon-cloud
- cc-auto-deployment

The following example shows how we should restart ClusterControl Controller service (cmon) inside the container:

```
$ docker exec -it clustercontrol supervisorctl restart cmon
```

By default, the ClusterControl UI is exposed on port 80 and 443 of the Docker host. You can run it in the host network, so you can monitor and manage other DB nodes outside of the Docker network (e.g., DB nodes running on physical hosts connected to the same switch, or VM instances running in the cloud). Or you can run ClusterControl under the user-defined bridge network, and have your database cluster pack in one single host.

7.2. Automatic Database Deployment

To deploy a database automatically with ClusterControl, we have built a complementary image running on CentOS base image with SSH enabled. It's a general-purpose image that works well with ClusterControl due to its ability to setup passwordless SSH from the ClusterControl container to the database containers. More details in the Github page.

The following command launches a three-node Galera Cluster with automatic deployment:

```
$ for i in {1..3}; do
  docker run -d \
  --name galera${i} \
  -p 666{$i}:3306 \
  --link clustercontrol:clustercontrol \
  -e CLUSTER_TYPE=galera \
  -e CLUSTER_NAME=mygalera \
  -e INITIAL_CLUSTER_SIZE=3 \
  severalnines/centos-ssh
done
```

The database deployment flow is described as below:

1. Create the database containers through docker run command.
2. The *cc-auto-deployment* script picks up the registered containers and creates a deployment job.
3. ClusterControl deploys the cluster once the INITIAL_CLUSTER_SIZE is reached.

The image supports automatic database cluster deployment. One command is needed to create ClusterControl, and another command to create the database cluster. ClusterControl will then pick up the database containers' IP addresses and hostnames after they are started, and start the deployment once it reaches the `INITIAL_CLUSTER_SIZE`. The deployment script uses 's9s', the ClusterControl CLI, to interact with the controller.

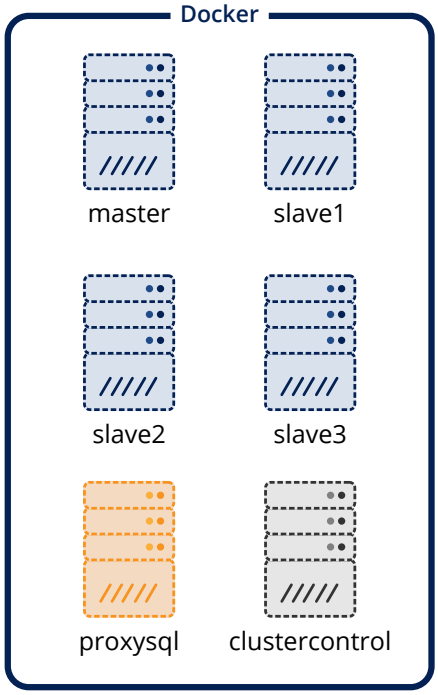
7.3. Manual Database Deployment

Using this method, the user creates the database containers through `docker create` or `docker service` commands. The IP addresses or hostnames of the created containers are then entered into the ClusterControl user interface in order to do the deployment. This extra step gives better control on the deployment process, as compared to the automated deployment process.

The basic steps are:

1. Run ClusterControl container
2. Run DB containers using severalnines/centos-ssh image with AUTO_DEPLOYMENT=0
3. Use ClusterControl UI to deploy the cluster according to the topology defined in the deployment wizard

Suppose we want to deploy a 4-node MySQL Replication with a ProxySQL on a single Docker host as illustrated in the following diagram:



1. Create a Docker network, 192.168.10.0/24 (db-cluster), for persistent IP addresses and hostnames:

```
$ docker network create --subnet=192.168.10.0/24
db-cluster
```

2. Run the ClusterControl container with dedicated IP address 192.168.10.10, publish HTTP port 5000 and create persistent volumes to survive across upgrade/restart/reschedule:

```
$ docker run -d --name=clustercontrol \
--network db-cluster \
--ip 192.168.10.10 \
-p 5000:80 \
-p 5001:443 \
-h clustercontrol \
-v /storage/clustercontrol/.ssh:/root/.ssh \
-v /storage/clustercontrol/datadir:/var/lib/mysql \
-v /storage/clustercontrol/cmon.d:/etc/cmon.d \
-v /storage/clustercontrol/backups:/backups \
severalnines/clustercontrol
```

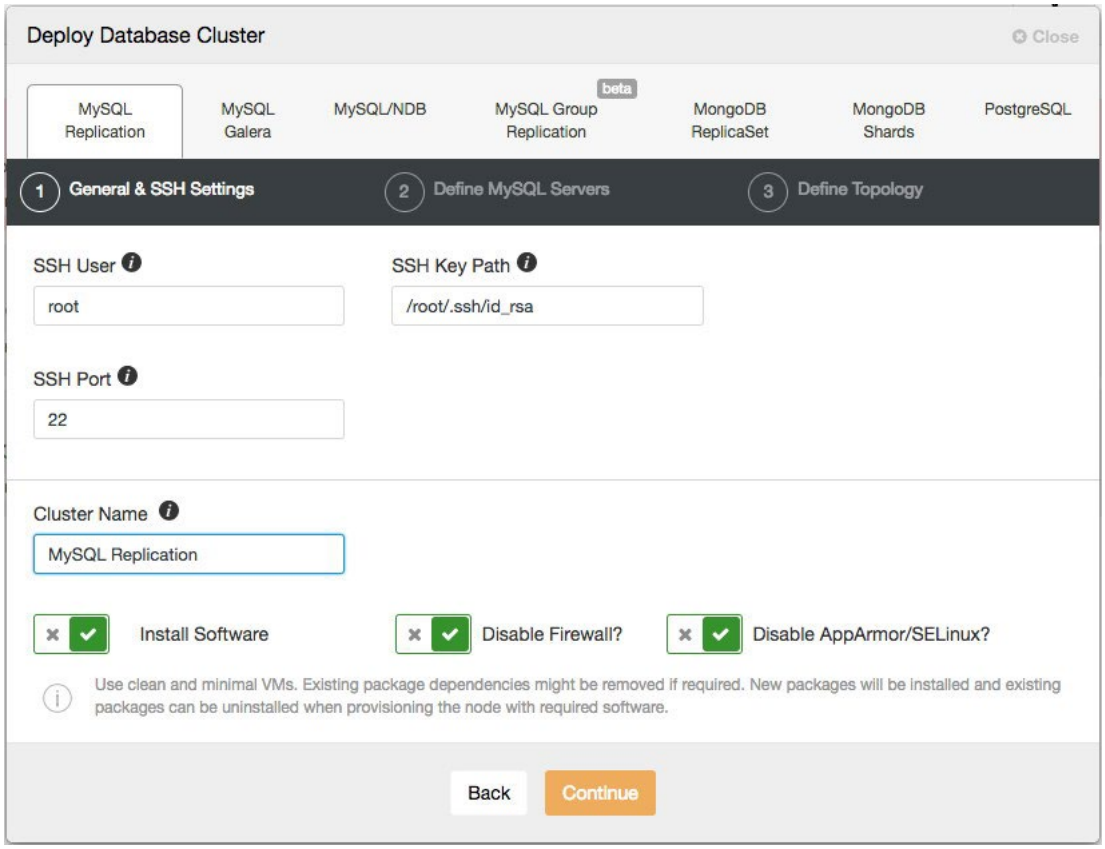
3. Run containers for the MySQL Replication instances. Make sure ClusterControl container starts first and then start the MySQL master, 192.168.10.100 on port 6000:

```
$ docker run -d --name master \
--network db-cluster \
--ip 192.168.10.100 \
-v /storage/master/datadir:/var/lib/mysql \
-h master \
-p 6000:3306 \
-e AUTO_DEPLOYMENT=0 \
-e CC_HOST=192.168.10.10 \
severalnines/centos-ssh
```

4. Then, create 3 MySQL slaves, 192.168.10.101 to 192.168.10.103, on port 6001 to 6003:

```
$ for i in {1..3}; do
        docker run -d --name slave${i} \
        -v /storage/slave${i}/datadir:/var/lib/mysql \
        -h slave${i} \
        -p 600${i}:3306 \
        --network db-cluster \
        --ip 192.168.10.10${i} \
        -e AUTO_DEPLOYMENT=0 \
        -e CC_HOST=192.168.10.10 \
        severalnines/centos-ssh
done
```

5. Log into the ClusterControl UI at https://{docker_host}:5001/clustercontrol, register the default admin user and go to "Deploy" -> "MySQL Replication" to start the deployment. Take note that AUTO_DEPLOYMENT is turned off, so we have better control of the installation via the ClusterControl UI. The centos-ssh image should already have pre-configured the SSH key, which is located at `/root/.ssh/id_rsa` inside the ClusterControl container. Enter the following details in the deployment wizard:



   Fill in the remaining input fields and click Deploy.

6. Run another container for ProxySQL, 192.168.10.201 port 6033:

```
$ docker run -d \
--name proxysql1 \
-v /storage/proxysql1/datadir:/var/lib/proxysql \
-p 6033:3306 \
--network db-cluster \
--ip 192.168.10.201 \
-e AUTO_DEPLOYMENT=0 \
-e CC_HOST=192.168.10.10 \
severalnines/centos-ssh
```

7. Go to ClusterControl -> your cluster -> Manage -> Load Balancers -> ProxySQL -> Deploy ProxySQL. Specify 192.168.10.201 as the ProxySQL Address and fill in the remaining input fields. Click Deploy to start the deployment process.

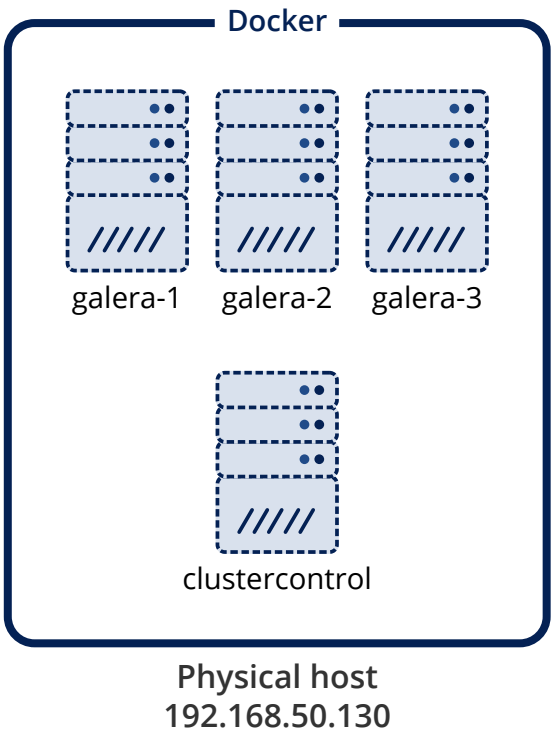
After the deployment completes, you will have a 4-node MySQL Replication setup with ProxySQL as load balancer.

7.4. Add Existing Database Containers

SSH is the main communication channel for ClusterControl, so having it run on the monitored host or container is mandatory. This can be a bit of a hassle though, because most of the MySQL images do not have SSH packages installed. This is one of the reasons we came up with the "severalnines/centos-ssh" image to simplify the deployment process.

If you already have a database cluster running on Docker, and you would like ClusterControl to manage it, you can simply run the ClusterControl container in the same Docker network as the database containers. The only requirement is to ensure the target containers have SSH related packages installed (openssh-server, openssh-clients). Then allow passwordless SSH from ClusterControl to the database containers. Once done, use the "Add Existing Server/Cluster" feature to import the database setup into ClusterControl.

Let's say we have a physical host, 192.168.50.130 installed with Docker, and assume there is a three-node Galera Cluster running under the standard Docker bridge network. We are going to import the cluster into ClusterControl, which is running in another container on the same host. The following is the high-level architecture diagram:



Install OpenSSH related packages on each of the database containers, allow root login, start it up and set the root password:

```
1  $ docker exec -ti [db-container] apt-get update
2  $ docker exec -ti [db-container] apt-get install -y
   openssh-server openssh-client
3  $ docker exec -it [db-container] sed -i 's|^PermitRootLog-
   in.*|PermitRootLogin yes|g' /etc/ssh/sshd_config
4  $ docker exec -ti [db-container] service ssh start
5  $ docker exec -it [db-container] passwd
```

Start the ClusterControl container as daemon and forward port 80 on the container to port 5000 on the host:

```
$ docker run -d --name=clustercontrol \
--network db-cluster \
-p 5000:80 \
-p 5001:443 \
-h clustercontrol \
-v /storage/clustercontrol/.ssh:/root/.ssh \
-v /storage/clustercontrol/datadir:/var/lib/mysql \
-v /storage/clustercontrol/cmon.d:/etc/cmon.d \
-v /storage/clustercontrol/backups:/backups \
severalnines/clustercontrol
```

Open a browser, go to http://{docker_host}:5000/clustercontrol and create a default admin user and password. You should now see the ClusterControl landing page.

The last step is setting up passwordless SSH to all database containers. Attach to the ClusterControl container interactive console:

```
$ docker exec -it clustercontrol /bin/bash
```

Copy the SSH key to all database containers:

```
$ ssh-copy-id 172.17.0.2
$ ssh-copy-id 172.17.0.3
$ ssh-copy-id 172.17.0.4
```

Start importing the cluster into ClusterControl. Open a web browser and go to Docker's physical host IP address with the mapped port e.g, http://192.168.50.130:5000/clustercontrol and click "Add Existing Cluster/Server" and specify the necessary details like MySQL vendor, version, MySQL root user and password and so on, as well as the IP address or hostname of the database containers.

Ensure you got the green tick when entering the hostname or IP address, indicating that ClusterControl is able to communicate with the node. Then, click the ***Import*** button and wait until ClusterControl finishes the job. The database cluster will be listed under the ClusterControl dashboard once it is imported.

Summary

Docker is probably the most talked about technology in the past few years. It has a very promising future and is currently well on its way to establish itself as a mainstream application deployment standard. Despite the challenges in using Docker for stateful database services, MySQL as well as PostgreSQL and MongoDB are typically in the top 10 deployed technologies in Docker surveys from different technology vendors. Both Docker and MySQL have huge user bases, each with their own vibrant community. We will most probably see more adoption of these technologies together, especially as the ecosystem of database tools and utilities for Docker expands.

Although Docker can help automate deployment of MySQL, the database still has to be managed and monitored. Crashes do not fix themselves, and care has to be taken in order to ensure data integrity. Operation teams need to be notified in case of performance issues or failures, backups need to be taken, administrators and developers need access monitoring metrics in order to understand and optimize performance. There is a lot to think about when managing a database environment, and combining with something like ClusterControl can provide a complete operational platform for production database workloads.

About ClusterControl

ClusterControl is the all-inclusive open source database management system for users with mixed environments that removes the need for multiple management tools. ClusterControl provides advanced deployment, management, monitoring, and scaling functionality to get your MySQL, MongoDB, and PostgreSQL databases up-and- running using proven methodologies that you can depend on to work. At the core of ClusterControl is it's automation functionality that let's you automate many of the database tasks you have to perform regularly like deploying new databases, adding and scaling new nodes, running backups and upgrades, and more. 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.

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. Severalnines is often called the "anti-startup" as it is entirely self-funded by its founders. The company has enabled over 12,000 deployments to date via its popular product ClusterControl. Currently counting BT, Orange, Cisco, CNRS, Technicolor, AVG, Ping Identity and Paytrail as customers. Severalnines is a private company headquartered in Stockholm, Sweden with o ces in Singapore, Japan and the United States. To see who is using Severalnines today visit:

https://www.severalnines.com/company

Deploy

Manage

Monitor

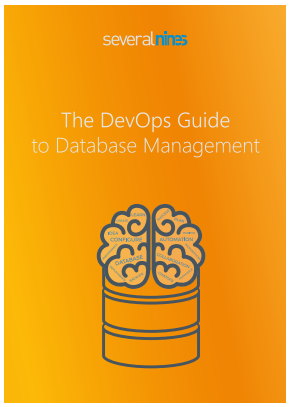
Scale

Related Whitepapers

DIY Cloud Database on Amazon Web Services: Best Practices

Over the course of this paper, we cover the details of AWS infrastructure deployment, considerations for deploying your database server(s) in the cloud, and finish with an example overview of how to automate the deployment and management of a MongoDB cluster using ClusterControl.

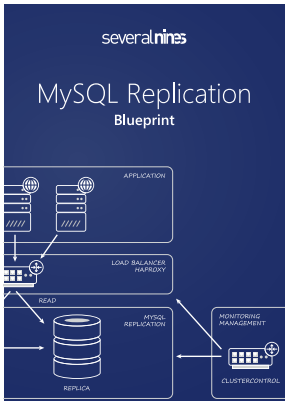
Download here



The DevOps Guide to Database Management

Relational databases are not very flexible by nature, while DevOps is all about flexibility. This creates many challenges that need to be overcome. This white paper discusses three core challenges faced by DevOps when it comes to managing databases. It also discusses how Severalnines ClusterControl can be used to address these challenges.

Download here



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

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

severalnines

This whitepaper covers the basics you need to understand when considering to run a MySQL service on top of Docker container virtualization. And although Docker can help automate deployment of MySQL, the database still has to be managed and monitored. ClusterControl by Severalnines can provide a complete operational platform for production database workloads.

Deploy

Manage

Monitor

Scale