Use the Device Mapper storage driver

Estimated reading time: 28 minutes

Device Mapper is a kernel-based framework that underpins many advanced volume management technologies on Linux. Docker's `devicemapper` storage driver leverages the thin provisioning and snapshotting capabilities of this framework for image and container management. This article refers to the Device Mapper storage driver as `devicemapper`, and the kernel framework as *Device Mapper*.

For the systems where it is supported, `devicemapper` support is included in the Linux kernel. However, specific configuration is required to use it with Docker.

The `devicemapper` driver uses block devices dedicated to Docker and operates at the block level, rather than the file level. These devices can be extended by adding physical storage to your Docker host, and they perform better than using a filesystem at the operating system (OS) level.

Prerequisites

- `devicemapper` storage driver is a supported storage driver for Docker EE on many OS distribution. See the Product compatibility matrix (https://success.docker.com/article/compatibility-matrix) for details.
- `devicemapper` is also supported on Docker CE running on CentOS, Fedora, Ubuntu, or Debian.
- Changing the storage driver makes any containers you have already created inaccessible on the local system. Use `docker save` to save containers, and push existing images to Docker Hub or a private repository, so you do not need to recreate them later.

Configure Docker with the `devicemapper` storage driver

Before following these procedures, you must first meet all the prerequisites (/storage/storagedriver/device-mapper-driver/#prerequisites).

Configure `loop-lvm` mode for testing

This configuration is only appropriate for testing. The `loop-lvm` mode makes use of a 'loopback' mechanism that allows files on the local disk to be read from and written to as if they were an actual physical disk or block device. However, the addition of the loopback mechanism, and interaction with the OS filesystem layer, means that IO operations can be slow and resource-intensive. Use of loopback devices can also introduce race conditions. However, setting up `loop-lvm` mode can help identify basic

issues (such as missing user space packages, kernel drivers, etc.) ahead of attempting the more complex set up required to enable `direct-lvm` mode. `loop-lvm` mode should therefore only be used to perform rudimentary testing prior to configuring `direct-lvm` .

For production systems, see Configure direct-lvm mode for production (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).

1. Stop Docker.

   ```
   $ sudo systemctl stop docker
   ```

2. Edit `/etc/docker/daemon.json` . If it does not yet exist, create it. Assuming that the file was empty, add the following contents.

   ```
   {
     "storage-driver": "devicemapper"
   }
   ```

   See all storage options for each storage driver:

   - Stable (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
   - Edge (https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

   Docker does not start if the `daemon.json` file contains badly-formed JSON.

3. Start Docker.

   ```
   $ sudo systemctl start docker
   ```

4. Verify that the daemon is using the `devicemapper` storage driver. Use the `docker info` command and look for `Storage Driver` .

```
$ docker info

  Containers: 0
    Running: 0
    Paused: 0
    Stopped: 0
  Images: 0
  Server Version: 17.03.1-ce
  Storage Driver: devicemapper
  Pool Name: docker-202:1-8413957-pool
  Pool Blocksize: 65.54 kB
  Base Device Size: 10.74 GB
  Backing Filesystem: xfs
  Data file: /dev/loop0
  Metadata file: /dev/loop1
  Data Space Used: 11.8 MB
  Data Space Total: 107.4 GB
  Data Space Available: 7.44 GB
  Metadata Space Used: 581.6 KB
  Metadata Space Total: 2.147 GB
  Metadata Space Available: 2.147 GB
  Thin Pool Minimum Free Space: 10.74 GB
  Udev Sync Supported: true
  Deferred Removal Enabled: false
  Deferred Deletion Enabled: false
  Deferred Deleted Device Count: 0
  Data loop file: /var/lib/docker/devicemapper/data
  Metadata loop file: /var/lib/docker/devicemapper/metadata
  Library Version: 1.02.135-RHEL7 (2016-11-16)
<output truncated>
```

This host is running in `loop-lvm` mode, which is **not** supported on production systems. This is indicated by the fact that the `Data loop file` and a `Metadata loop file` are on files under `/var/lib/docker/devicemapper`. These are loopback-mounted sparse files. For production systems, see Configure direct-lvm mode for production (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-for-production).

Configure direct-lvm mode for production

Production hosts using the `devicemapper` storage driver must use `direct-lvm` mode. This mode uses block devices to create the thin pool. This is faster than using loopback devices, uses system resources more efficiently, and block devices can grow as needed. However, more setup is required than in `loop-lvm` mode.

After you have satisfied the prerequisites (/storage/storagedriver/device-mapper-driver/#prerequisites), follow the steps below to configure Docker to use the `devicemapper` storage driver in `direct-lvm` mode.

> **Warning**: Changing the storage driver makes any containers you have already created inaccessible on the local system. Use `docker save` to save containers, and push existing images to Docker Hub or a private repository, so you do not need to recreate them later.

ALLOW DOCKER TO CONFIGURE DIRECT-LVM MODE

With Docker `17.06` and higher, Docker can manage the block device for you, simplifying configuration of `direct-lvm` mode. **This is appropriate for fresh Docker setups only.** You can only use a single block device. If you need to use multiple block devices, configure direct-lvm mode manually (/storage/storagedriver/device-mapper-driver/#configure-direct-lvm-mode-manually) instead. The following new configuration options have been added:

Option	Description	Required?	Default	Example
`dm.directlvm_device`	The path to the block device to configure for `direct-lvm`.	Yes		`dm.directlvm_device="/dev/xvdf"`
`dm.thinp_percent`	The percentage of space to use for storage from the passed in block device.	No	95	`dm.thinp_percent=95`
`dm.thinp_metapercent`	The percentage of space to use for metadata storage from the passed-in block device.	No	1	`dm.thinp_metapercent=1`
`dm.thinp_autoextend_threshold`	The threshold for when lvm should automatically extend the thin pool as a percentage of the total storage space.	No	80	`dm.thinp_autoextend_threshold=80`
`dm.thinp_autoextend_percent`	The percentage to increase the thin pool by when an autoextend is triggered.	No	20	`dm.thinp_autoextend_percent=20`

Option	Description	Required?	Default	Example
`dm.directlvm_device_force`	Whether to format the block device even if a filesystem already exists on it. If set to `false` and a filesystem is present, an error is logged and the filesystem is left intact.	No	false	`dm.directlvm_device_force=true`

Edit the `daemon.json` file and set the appropriate options, then restart Docker for the changes to take effect. The following `daemon.json` configuration sets all of the options in the table above.

```
{
  "storage-driver": "devicemapper",
  "storage-opts": [
    "dm.directlvm_device=/dev/xdf",
    "dm.thinp_percent=95",
    "dm.thinp_metapercent=1",
    "dm.thinp_autoextend_threshold=80",
    "dm.thinp_autoextend_percent=20",
    "dm.directlvm_device_force=false"
  ]
}
```

See all storage options for each storage driver:

- Stable (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options)
- Edge (https://docs.docker.com/edge/engine/reference/commandline/dockerd/#storage-driver-options)

Restart Docker for the changes to take effect. Docker invokes the commands to configure the block device for you.

> **Warning**: Changing these values after Docker has prepared the block device for you is not supported and causes an error.

You still need to perform periodic maintenance tasks (/storage/storagedriver/device-mapper-driver/#manage-devicemapper).

CONFIGURE DIRECT-LVM MODE MANUALLY

The procedure below creates a logical volume configured as a thin pool to use as backing for the storage pool. It assumes that you have a spare block device at `/dev/xvdf` with enough free space to complete the task. The device identifier and volume sizes may be different in your environment and you should substitute your own values throughout the procedure. The procedure also assumes that the Docker daemon is in the `stopped` state.

1. Identify the block device you want to use. The device is located under `/dev/` (such as `/dev/xvdf`) and needs enough free space to store the images and container layers for the workloads that host runs. A solid state drive is ideal.
2. Stop Docker.

   ```
   $ sudo systemctl stop docker
   ```

3. Install the following packages:
   - **RHEL / CentOS**: `device-mapper-persistent-data`, `lvm2`, and all dependencies
   - **Ubuntu / Debian**: `thin-provisioning-tools`, `lvm2`, and all dependencies
4. Create a physical volume on your block device from step 1, using the `pvcreate` command. Substitute your device name for `/dev/xvdf`.

   > **Warning**: The next few steps are destructive, so be sure that you have specified the correct device!

   ```
   $ sudo pvcreate /dev/xvdf

   Physical volume "/dev/xvdf" successfully created.
   ```

5. Create a `docker` volume group on the same device, using the `vgcreate` command.

   ```
   $ sudo vgcreate docker /dev/xvdf

   Volume group "docker" successfully created
   ```

6. Create two logical volumes named `thinpool` and `thinpoolmeta` using the `lvcreate` command. The last parameter specifies the amount of free space to allow for automatic expanding of the data or metadata if space runs low, as a temporary stop-gap. These are the recommended values.

```
$ sudo lvcreate --wipesignatures y -n thinpool docker -l 95%VG

Logical volume "thinpool" created.

$ sudo lvcreate --wipesignatures y -n thinpoolmeta docker -l 1%VG

Logical volume "thinpoolmeta" created.
```

7. Convert the volumes to a thin pool and a storage location for metadata for the thin pool, using the `lvconvert` command.

```
$ sudo lvconvert -y \
--zero n \
-c 512K \
--thinpool docker/thinpool \
--poolmetadata docker/thinpoolmeta

WARNING: Converting logical volume docker/thinpool and docker/thinpoolmeta to
thin pool's data and metadata volumes with metadata wiping.
THIS WILL DESTROY CONTENT OF LOGICAL VOLUME (filesystem etc.)
Converted docker/thinpool to thin pool.
```

8. Configure autoextension of thin pools via an `lvm` profile.

```
$ sudo vi /etc/lvm/profile/docker-thinpool.profile
```

9. Specify `thin_pool_autoextend_threshold` and `thin_pool_autoextend_percent` values.

   `thin_pool_autoextend_threshold` is the percentage of space used before `lvm` attempts to autoextend the available space (100 = disabled, not recommended).

   `thin_pool_autoextend_percent` is the amount of space to add to the device when automatically extending (0 = disabled).

   The example below adds 20% more capacity when the disk usage reaches 80%.

```
activation {
  thin_pool_autoextend_threshold=80
  thin_pool_autoextend_percent=20
}
```

   Save the file.

10. Apply the LVM profile, using the `lvchange` command.

```
$ sudo lvchange --metadataprofile docker-thinpool docker/thinpool

Logical volume docker/thinpool changed.
```

11. Ensure monitoring of the logical volume is enabled.

```
$ sudo lvs -o+seg_monitor

LV       VG     Attr       LSize  Pool Origin Data%  Meta%  Move Log Cpy%Sync Convert Mo
thinpool docker twi-a-t--- 95.00g             0.00   0.01                             no
```

If the output in the `Monitor` column reports, as above, that the volume is `not monitored`, then monitoring needs to be explicitly enabled. Without this step, automatic extension of the logical volume will not occur, regardless of any settings in the applied profile.

```
$ sudo lvchange --monitor y docker/thinpool
```

Double check that monitoring is now enabled by running the `sudo lvs -o+seg_monitor` command a second time. The `Monitor` column should now report the logical volume is being `monitored`.

12. If you have ever run Docker on this host before, or if `/var/lib/docker/` exists, move it out of the way so that Docker can use the new LVM pool to store the contents of image and containers.

```
$ sudo su -
# mkdir /var/lib/docker.bk
# mv /var/lib/docker/* /var/lib/docker.bk
# exit
```

If any of the following steps fail and you need to restore, you can remove `/var/lib/docker` and replace it with `/var/lib/docker.bk`.

13. Edit `/etc/docker/daemon.json` and configure the options needed for the `devicemapper` storage driver. If the file was previously empty, it should now contain the following contents:

```
{
    "storage-driver": "devicemapper",
    "storage-opts": [
    "dm.thinpooldev=/dev/mapper/docker-thinpool",
    "dm.use_deferred_removal=true",
    "dm.use_deferred_deletion=true"
    ]
}
```

14. Start Docker.

    systemd:

```
$ sudo systemctl start docker
```

service:

```
$ sudo service docker start
```

15. Verify that Docker is using the new configuration using `docker info`.

```
$ docker info

Containers: 0
 Running: 0
 Paused: 0
 Stopped: 0
Images: 0
Server Version: 17.03.1-ce
Storage Driver: devicemapper
 Pool Name: docker-thinpool
 Pool Blocksize: 524.3 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 19.92 MB
 Data Space Total: 102 GB
 Data Space Available: 102 GB
 Metadata Space Used: 147.5 kB
 Metadata Space Total: 1.07 GB
 Metadata Space Available: 1.069 GB
 Thin Pool Minimum Free Space: 10.2 GB
 Udev Sync Supported: true
 Deferred Removal Enabled: true
 Deferred Deletion Enabled: true
 Deferred Deleted Device Count: 0
 Library Version: 1.02.135-RHEL7 (2016-11-16)
<output truncated>
```

If Docker is configured correctly, the `Data file` and `Metadata file` is blank, and the pool name is `docker-thinpool`.

16. After you have verified that the configuration is correct, you can remove the `/var/lib/docker.bk` directory which contains the previous configuration.

```
$ sudo rm -rf /var/lib/docker.bk
```

Manage devicemapper

Monitor the thin pool

Do not rely on LVM auto-extension alone. The volume group automatically extends, but the volume can still fill up. You can monitor free space on the volume using `lvs` or `lvs -a`. Consider using a monitoring tool at the OS level, such as Nagios.

To view the LVM logs, you can use `journalctl`:

```
$ sudo journalctl -fu dm-event.service
```

If you run into repeated problems with thin pool, you can set the storage option `dm.min_free_space` to a value (representing a percentage) in `/etc/docker/daemon.json`. For instance, setting it to `10` ensures that operations fail with a warning when the free space is at or near 10%. See the storage driver options in the Engine daemon reference (https://docs.docker.com/engine/reference/commandline/dockerd/#storage-driver-options).

Increase capacity on a running device

You can increase the capacity of the pool on a running thin-pool device. This is useful if the data's logical volume is full and the volume group is at full capacity. The specific procedure depends on whether you are using a loop-lvm thin pool (/storage/storagedriver/device-mapper-driver/#resize-a-loop-lvm-thin-pool) or a direct-lvm thin pool (/storage/storagedriver/device-mapper-driver/#resize-a-direct-lvm-thin-pool).

RESIZE A LOOP-LVM THIN POOL

The easiest way to resize a `loop-lvm` thin pool is to use the device_tool utility (/storage/storagedriver/device-mapper-driver/#use-the-device_tool-utility), but you can use operating system utilities (/storage/storagedriver/device-mapper-driver/#use-operating-system-utilities) instead.

Use the device_tool utility

A community-contributed script called `device_tool.go` is available in the moby/moby (https://github.com/moby/moby/tree/master/contrib/docker-device-tool) Github repository. You can use this tool to resize a `loop-lvm` thin pool, avoiding the long process above. This tool is not guaranteed to work, but you should only be using `loop-lvm` on non-production systems.

If you do not want to use `device_tool`, you can resize the thin pool manually (/storage/storagedriver/device-mapper-driver/#use-operating-system-utilities) instead.

1. To use the tool, clone the Github repository, change to the `contrib/docker-device-tool`, and follow the instructions in the `README.md` to compile the tool.
2. Use the tool. The following example resizes the thin pool to 200GB.

   ```
   $ ./device_tool resize 200GB
   ```

Use operating system utilities

If you do not want to use the device-tool utility (/storage/storagedriver/device-mapper-driver/#use-the-device_tool-utility), you can resize a `loop-lvm` thin pool manually using the following procedure.

In `loop-lvm` mode, a loopback device is used to store the data, and another to store the metadata. `loop-lvm` mode is only supported for testing, because it has significant performance and stability drawbacks.

If you are using `loop-lvm` mode, the output of `docker info` shows file paths for `Data loop file` and `Metadata loop file`:

```
$ docker info |grep 'loop file'

 Data loop file: /var/lib/docker/devicemapper/data
 Metadata loop file: /var/lib/docker/devicemapper/metadata
```

Follow these steps to increase the size of the thin pool. In this example, the thin pool is 100 GB, and is increased to 200 GB.

1. List the sizes of the devices.

   ```
   $ sudo ls -lh /var/lib/docker/devicemapper/

   total 1175492
   -rw------- 1 root root 100G Mar 30 05:22 data
   -rw------- 1 root root 2.0G Mar 31 11:17 metadata
   ```

2. Increase the size of the `data` file to 200 G using the `truncate` command, which is used to increase or decrease the size of a file. Note that decreasing the size is a destructive operation.

   ```
   $ sudo truncate -s 200G /var/lib/docker/devicemapper/data
   ```

3. Verify the file size changed.

   ```
   $ sudo ls -lh /var/lib/docker/devicemapper/

   total 1.2G
   -rw------- 1 root root 200G Apr 14 08:47 data
   -rw------- 1 root root 2.0G Apr 19 13:27 metadata
   ```

4. The loopback file has changed on disk but not in memory. List the size of the loopback device in memory, in GB. Reload it, then list the size again. After the reload, the size is 200 GB.

   ```
   $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

   100

   $ sudo losetup -c /dev/loop0

   $ echo $[ $(sudo blockdev --getsize64 /dev/loop0) / 1024 / 1024 / 1024 ]

   200
   ```

5. Reload the devicemapper thin pool.

a. Get the pool name first. The pool name is the first field, delimited by `:`. This command extracts it.

```
$ sudo dmsetup status | grep ' thin-pool ' | awk -F ': ' {'print $1'}

docker-8:1-123141-pool
```

b. Dump the device mapper table for the thin pool.

```
$ sudo dmsetup table docker-8:1-123141-pool

0 209715200 thin-pool 7:1 7:0 128 32768 1 skip_block_zeroing
```

c. Calculate the total sectors of the thin pool using the second field of the output. The number is expressed in 512-k sectors. A 100G file has 209715200 512-k sectors. If you double this number to 200G, you get 419430400 512-k sectors.

d. Reload the thin pool with the new sector number, using the following three `dmsetup` commands.

```
$ sudo dmsetup suspend docker-8:1-123141-pool

$ sudo dmsetup reload docker-8:1-123141-pool --table '0 419430400 thin-pool 7:1 7:0

$ sudo dmsetup resume docker-8:1-123141-pool
```

RESIZE A DIRECT-LVM THIN POOL

To extend a `direct-lvm` thin pool, you need to first attach a new block device to the Docker host, and make note of the name assigned to it by the kernel. In this example, the new block device is `/dev/xvdg`.

Follow this procedure to extend a `direct-lvm` thin pool, substituting your block device and other parameters to suit your situation.

1. Gather information about your volume group.

   Use the `pvdisplay` command to find the physical block devices currently in use by your thin pool, and the volume group's name.

   ```
   $ sudo pvdisplay |grep 'VG Name'

   PV Name               /dev/xvdf
   VG Name               docker
   ```

   In the following steps, substitute your block device or volume group name as appropriate.

2. Extend the volume group, using the `vgextend` command with the `VG Name` from the previous step, and the name of your **new** block device.

```
$ sudo vgextend docker /dev/xvdg

Physical volume "/dev/xvdg" successfully created.
Volume group "docker" successfully extended
```

3. Extend the `docker/thinpool` logical volume. This command uses 100% of the volume right away, without auto-extend. To extend the metadata thinpool instead, use `docker/thinpool_tmeta`.

```
$ sudo lvextend -l+100%FREE -n docker/thinpool

Size of logical volume docker/thinpool_tdata changed from 95.00 GiB (24319 extents) to 1
Logical volume docker/thinpool_tdata successfully resized.
```

4. Verify the new thin pool size using the `Data Space Available` field in the output of `docker info`. If you extended the `docker/thinpool_tmeta` logical volume instead, look for `Metadata Space Available`.

```
Storage Driver: devicemapper
 Pool Name: docker-thinpool
 Pool Blocksize: 524.3 kB
 Base Device Size: 10.74 GB
 Backing Filesystem: xfs
 Data file:
 Metadata file:
 Data Space Used: 212.3 MB
 Data Space Total: 212.6 GB
 Data Space Available: 212.4 GB
 Metadata Space Used: 286.7 kB
 Metadata Space Total: 1.07 GB
 Metadata Space Available: 1.069 GB
<output truncated>
```

Activate the `devicemapper` after reboot

If you reboot the host and find that the `docker` service failed to start, look for the error, "Non existing device". You need to re-activate the logical volumes with this command:

```
sudo lvchange -ay docker/thinpool
```

How the `devicemapper` storage driver works

> **Warning**: Do not directly manipulate any files or directories within `/var/lib/docker/` . These files and directories are managed by Docker.

Use the `lsblk` command to see the devices and their pools, from the operating system's point of view:

```
$ sudo lsblk

NAME                    MAJ:MIN RM  SIZE RO TYPE MOUNTPOINT
xvda                    202:0    0    8G  0 disk
└─xvda1                 202:1    0    8G  0 part /
xvdf                    202:80   0  100G  0 disk
├─docker-thinpool_tmeta 253:0    0 1020M  0 lvm
│ └─docker-thinpool     253:2    0   95G  0 lvm
└─docker-thinpool_tdata 253:1    0   95G  0 lvm
  └─docker-thinpool     253:2    0   95G  0 lvm
```

Use the `mount` command to see the mount-point Docker is using:

```
$ mount |grep devicemapper
/dev/xvda1 on /var/lib/docker/devicemapper type xfs (rw,relatime,seclabel,attr2,inode64,noquo
```

When you use `devicemapper` , Docker stores image and layer contents in the thinpool, and exposes them to containers by mounting them under subdirectories of `/var/lib/docker/devicemapper/` .

Image and container layers on-disk

The `/var/lib/docker/devicemapper/metadata/` directory contains metadata about the Devicemapper configuration itself and about each image and container layer that exist. The `devicemapper` storage driver uses snapshots, and this metadata include information about those snapshots. These files are in JSON format.

The `/var/lib/devicemapper/mnt/` directory contains a mount point for each image and container layer that exists. Image layer mount points are empty, but a container's mount point shows the container's filesystem as it appears from within the container.

Image layering and sharing

The `devicemapper` storage driver uses dedicated block devices rather than formatted filesystems, and operates on files at the block level for maximum performance during copy-on-write (CoW) operations.

SNAPSHOTS

Another feature of `devicemapper` is its use of snapshots (also sometimes called *thin devices* or *virtual devices*), which store the differences introduced in each layer as very small, lightweight thin pools. Snapshots provide many benefits:

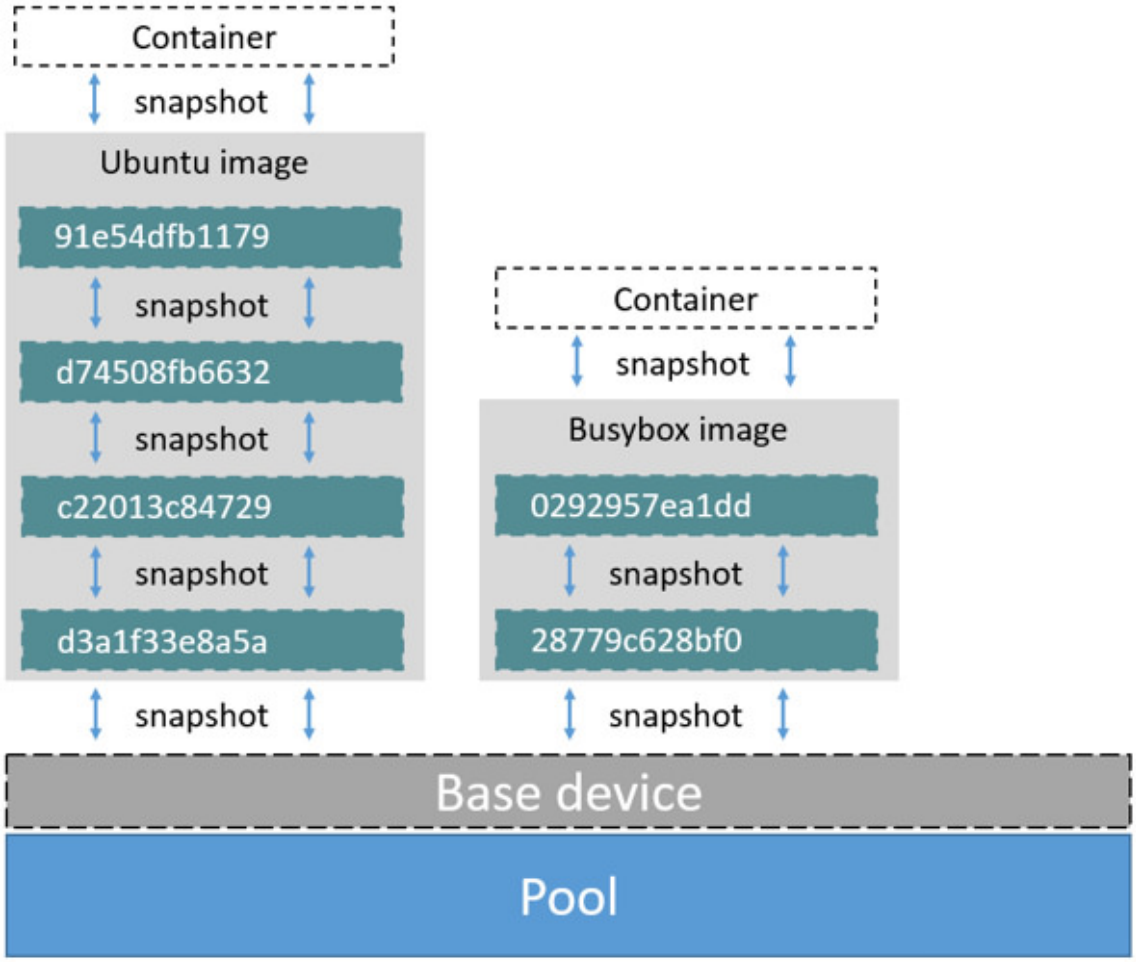
- Layers which are shared in common between containers are only stored on disk once, unless they are writable. For instance, if you have 10 different images which are all based on `alpine`, the `alpine` image and all its parent images are only stored once each on disk.
- Snapshots are an implementation of a copy-on-write (CoW) strategy. This means that a given file or directory is only copied to the container's writable layer when it is modified or deleted by that container.
- Because `devicemapper` operates at the block level, multiple blocks in a writable layer can be modified simultaneously.
- Snapshots can be backed up using standard OS-level backup utilities. Just make a copy of `/var/lib/docker/devicemapper/`.

DEVICEMAPPER WORKFLOW

When you start Docker with the `devicemapper` storage driver, all objects related to image and container layers are stored in `/var/lib/docker/devicemapper/`, which is backed by one or more block-level devices, either loopback devices (testing only) or physical disks.

- The *base device* is the lowest-level object. This is the thin pool itself. You can examine it using `docker info`. It contains a filesystem. This base device is the starting point for every image and container layer. The base device is a Device Mapper implementation detail, rather than a Docker layer.
- Metadata about the base device and each image or container layer is stored in `/var/lib/docker/devicemapper/metadata/` in JSON format. These layers are copy-on-write snapshots, which means that they are empty until they diverge from their parent layers.
- Each container's writable layer is mounted on a mountpoint in `/var/lib/docker/devicemapper/mnt/`. An empty directory exists for each read-only image layer and each stopped container.

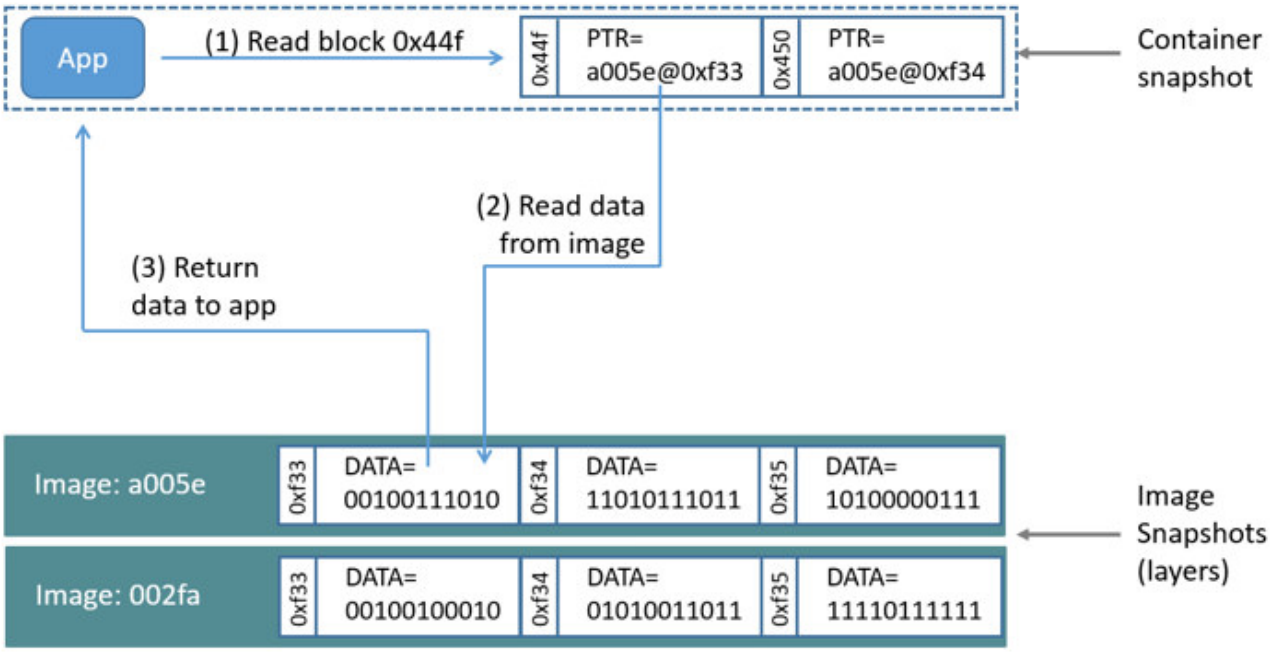
Each image layer is a snapshot of the layer below it. The lowest layer of each image is a snapshot of the base device that exists in the pool. When you run a container, it is a snapshot of the image the container is based on. The following example shows a Docker host with two running containers. The first is a `ubuntu` container and the second is a `busybox` container.



How container reads and writes work with `devicemapper`

Reading files

With `devicemapper`, reads happen at the block level. The diagram below shows the high level process for reading a single block (`0x44f`) in an example container.



An application makes a read request for block `0x44f` in the container. Because the container is a thin snapshot of an image, it doesn't have the block, but it has a pointer to the block on the nearest parent image where it does exist, and it reads the block from there. The block now exists in the container's memory.

Writing files

Writing a new file: With the `devicemapper` driver, writing new data to a container is accomplished by an *allocate-on-demand* operation. Each block of the new file is allocated in the container's writable layer and the block is written there.

Updating an existing file: The relevant block of the file is read from the nearest layer where it exists. When the container writes the file, only the modified blocks are written to the container's writable layer.

Deleting a file or directory: When you delete a file or directory in a container's writable layer, or when an image layer deletes a file that exists in its parent layer, the `devicemapper` storage driver intercepts further read attempts on that file or directory and responds that the file or directory does not exist.

Writing and then deleting a file: If a container writes to a file and later deletes the file, all of those operations happen in the container's writable layer. In that case, if you are using `direct-lvm`, the blocks are freed. If you use `loop-lvm`, the blocks may not be freed. This is another reason not to use `loop-lvm` in production.

Device Mapper and Docker performance

- **`allocate-on demand` performance impact**:

  The `devicemapper` storage driver uses an `allocate-on-demand` operation to allocate new blocks from the thin pool into a container's writable layer. Each block is 64KB, so this is the minimum amount of space that is used for a write.

- **Copy-on-write performance impact**: The first time a container modifies a specific block, that block is written to the container's writable layer. Because these writes happen at the level of the block rather than the file, performance impact is minimized. However, writing a large number of blocks can still negatively impact performance, and the `devicemapper` storage driver may actually perform worse than other storage drivers in this scenario. For write-heavy workloads, you should use data volumes, which bypass the storage driver completely.

Performance best practices

Keep these things in mind to maximize performance when using the `devicemapper` storage driver.

- **Use `direct-lvm`**: The `loop-lvm` mode is not performant and should never be used in production.
- **Use fast storage**: Solid-state drives (SSDs) provide faster reads and writes than spinning disks.
- **Memory usage**: the `devicemapper` uses more memory than some other storage drivers. Each launched container loads one or more copies of its files into memory, depending on how many blocks of the same file are being modified at the same time. Due to the memory pressure, the

`devicemapper` storage driver may not be the right choice for certain workloads in high-density use cases.

- Use volumes for write-heavy workloads: Volumes provide the best and most predictable performance for write-heavy workloads. This is because they bypass the storage driver and do not incur any of the potential overheads introduced by thin provisioning and copy-on-write. Volumes have other benefits, such as allowing you to share data among containers and persisting even when no running container is using them.
- Note: when using `devicemapper` and the `json-file` log driver, the log files generated by a container are still stored in Docker's dataroot directory, by default `/var/lib/docker`. If your containers generate lots of log messages, this may lead to increased disk usage or the inability to manage your system due to a full disk. You can configure a log driver (https://docs.docker.com/config/containers/logging/configure/) to store your container logs externally.

Related Information

- Volumes (https://docs.docker.com/storage/volumes/)
- Understand images, containers, and storage drivers (https://docs.docker.com/storage/storagedriver/imagesandcontainers/)
- Select a storage driver (https://docs.docker.com/storage/storagedriver/selectadriver/)

container (https://docs.docker.com/glossary/?term=container), storage (https://docs.docker.com/glossary/?term=storage), driver (https://docs.docker.com/glossary/?term=driver), device mapper (https://docs.docker.com/glossary/?term=device%20mapper)