Docker: Overview and Benchmark

Yiğit Çakmak - Mehmet Kubilay Karabağ

Docker is an open-source platform that automates the deployment, scaling, and management of applications within lightweight, portable containers. These containers package an application and its dependencies together, ensuring that the software runs consistently across different environments.

Core Concepts of Docker

Containers

Containers are the fundamental units in Docker. They encapsulate an application along with its dependencies, libraries, and configuration files. This isolation ensures that the application will run the same way, regardless of where it is deployed. Containers are lightweight and share the host system's kernel, making them more efficient than traditional virtual machines.

Images

Docker images are read-only templates used to create containers. They include the application's code, runtime, libraries, environment variables, and configuration files. Images are built using a `Dockerfile`, which is a script that contains a series of instructions to assemble the image.

Dockerfile

A `Dockerfile` is a text document that contains all the commands a user could call on the command line to assemble an image. It specifies the base image, the software packages to install, the configuration settings, and the commands to run. Docker reads this Dockerfile and builds an image accordingly.

Docker Engine

The Docker Engine is the core component of Docker, consisting of:

- A server (daemon process)
- A REST API interface
- A command-line interface (CLI) client

The daemon creates and manages Docker objects such as images, containers, networks, and volumes.

Docker Hub

Docker Hub is a cloud-based registry service for sharing Docker images. It allows users to publish and access images, making it easy to distribute applications. Public images are freely available, while private images require authentication.

How Docker Works

1. Building an Image

The process begins with a `Dockerfile`, where the user specifies the base image and additional layers needed for the application. Docker reads the Dockerfile and executes the commands to create an image.

2. Creating a Container

Once the image is built, it can be used to create a container. A container is an instance of an image running as a separate entity with its own filesystem, network interfaces, and isolated process tree.

3. Running a Container

The Docker Engine takes the image and runs it as a container. Containers can be started, stopped, moved, and deleted using Docker commands. They can also be scaled by running multiple instances to handle increased load.

4. Networking and Storage

Docker provides various networking models to connect containers with each other and with external networks. It also supports persistent storage, allowing containers to store and share data using volumes and bind mounts.

5. Management and Orchestration

Docker includes tools for managing and orchestrating containers, such as Docker Compose for defining multi-container applications and Docker Swarm for clustering and scaling across multiple Docker hosts.

Benefits of Docker

- **Consistency**: Containers ensure that the application runs the same way in development, testing, and production environments.
- **Portability**: Containers can run on any system that supports Docker, making it easier to migrate applications across different infrastructures.
- **Efficiency**: Containers are lightweight and use fewer resources compared to virtual machines, enabling higher density and better utilization of hardware.

- **Scalability**: Docker makes it easy to scale applications up or down by adding or removing containers as needed.

Conclusion

Docker revolutionizes the way applications are developed, shipped, and deployed by providing a consistent and efficient environment for running software across different platforms. Its use of containers ensures that applications are portable, scalable, and isolated, making it a powerful tool for modern software development and deployment.

Get Started

Get Docker

Download and install Docker on the platform of your choice, including Mac, Linux, or Windows.

https://docs.docker.com/get-started/get-docker/

Step-by-Step Guide to Installing Docker on Windows

Step 1: Enable Hyper-V and WSL 2

Docker on Windows requires two key components: **Hyper-V** (for virtualization) and the **Windows Subsystem for Linux (WSL 2)**. Follow these steps to enable them:

Enable Hyper-V:

1. Open the **Control Panel** and navigate to **Programs** > **Programs and Features**.
2. On the left, click **Turn Windows features on or off**.
3. Scroll through the list and check **Hyper-V** and **Windows Subsystem for Linux**.
4. Click **OK** and reboot your system to apply these changes.

Install WSL 2:

1. Open **PowerShell** as Administrator.
2. Run the following command to install WSL 2:

```
wsl --install
```

3. After running this, reboot your machine again. WSL 2 will be enabled, improving Docker's performance and compatibility.

Step 2: Download Docker Desktop for Windows

1. Head to the **Docker Hub** and download the Docker Desktop installer for Windows.
2. Choose the **stable** version for better reliability. Save the file to an easily accessible location.

Step 3: Install Docker Desktop

1. **Run the Installer**: Find the downloaded `.exe` file and double-click it to start the installation process.
2. **Follow Installation Prompts**:
   - Accept the license agreement.
   - Choose the installation location or leave the default settings.
   - During installation, enable the **WSL 2-based engine** when prompted. This is highly recommended for better performance.
3. **Complete Installation**: After installation, click **Close** to finish.

Step 4: Verify Docker Installation

Once Docker Desktop is installed:

1. **Launch Docker Desktop** from the Start Menu. Docker might take a few moments to fully start up, and you'll notice the Docker whale icon appear in the system tray when it's running.

2. **Verify Installation**: Open **Command Prompt** or **PowerShell** and type:

   ```
   docker --version
   ```

   You should see the version number of Docker, confirming a successful installation.

Step 5: Run Your First Docker Container

To make sure Docker is functioning correctly, try running a test container:

1. In a terminal, run:

   ```
   docker run hello-world
   ```

2. Docker will pull the test image and run it inside a container. If the installation was successful, you'll see a confirmation message indicating Docker is working correctly.

Basic Docker Commands to Get You Started

Here are a few essential Docker commands to help you begin:

- **Run a container from an image**:

  ```
  docker run [image-name]
  ```

- **List running containers**:

  ```
  docker ps
  ```

- **List locally stored images**:

  ```
  docker images
  ```

- **Pull an image from Docker Hub**:

```
docker pull [image-name]
```

- **Remove a container**:

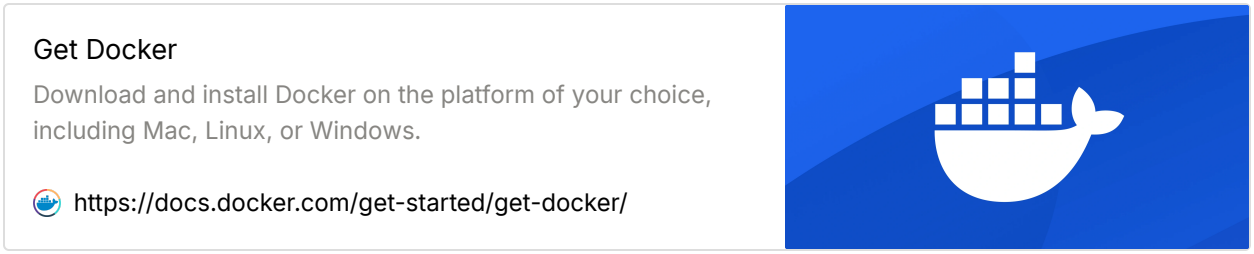
```
docker rm [container-id]
```

- **Remove an image**:

```
docker rmi [image-id]
```

This guide walks you through Docker's installation on Windows, from enabling WSL 2 and Hyper-V to running your first container. With these steps, you're set to explore Docker's containerization features on your system!

You can follow this document for other systems:



Our project was built using Apache Spark in all three ways, utilizing Docker to provide convenience to the users.

Installing PySpark environment with Docker

Important Note on Using `requirements.txt`

In Python projects, the `requirements.txt` file is used to list all the dependencies required by the project along with their specific versions. This file ensures consistency by installing the same versions of dependencies in different environments, avoiding potential issues caused by version incompatibilities.

Editing `requirements.txt` for Custom Versions

To change the versions of dependencies listed in `requirements.txt`, simply update the version numbers next to each package name.

First Way: Using .py File with Docker

1. Start Docker.
2. Create a `data` folder within the `pyspark` folder.
3. Upload the text file you want to use into the `pyspark` folder.
4. Right-click on the folder named `spark-env-main`. Then press "Open in the terminal."
5. Write `cd pyspark` in the terminal and press Enter.
6. Write `docker build -t pyspark-app .` in the terminal and press Enter. Wait for the build to complete.
7. Write `docker run pyspark-app` in the terminal.
8. Finish! You can see the result in the terminal.

Second Way: Using Jupyter with Docker

1. Start Docker.
2. Right-click on the folder named `spark-env-main`. Then press "Open in the terminal."
3. Write `cd pyspark` in the terminal and press Enter.
4. Write `docker-compose up --build` in the terminal and press Enter. Wait for the build to complete.
5. Go to this website: http://127.0.0.1:8888/lab
6. Click the relevant icon and upload your `.ipynb` and `.txt` files.
7. Finish! You can see the result.

Benchmark

1. Elapsed Time (elapsedTime)

- **Description**: This metric represents the total time taken to complete the entire job, including all tasks.
- **Importance**: It indicates how quickly the job completes, making it a fundamental metric for assessing overall performance.

2. Executor Run Time (executorRunTime)

- **Description**: This represents the total time that all executors spent actively running tasks.
- **Importance**: It shows the workload intensity and how long system resources were used. A higher runtime means more processing power was required.

3. Executor CPU Time (executorCpuTime)

- **Description**: This shows the actual CPU processing time consumed by the executors.
- **Importance**: The higher or lower CPU time reflects how computation-heavy the job is and how effectively the CPU is being utilized.

4. Executor Deserialize Time (executorDeserializeTime)

- **Description**: This metric reflects the time spent by executors deserializing the received data (converting it into a usable format).
- **Importance**: Deserialization is necessary to make data usable for processing. High deserialization times can indicate a delay in starting the processing of data.

5. Executor Deserialize CPU Time (executorDeserializeCpuTime)

- **Description**: This metric represents the CPU time spent during the deserialization process.
- **Importance**: It shows how much of the CPU is being consumed just to deserialize data. Ideally, this should be low to avoid bottlenecks before actual

data processing starts.

6. Result Serialization Time (resultSerializationTime)

- **Description**: This is the time spent serializing (converting to byte form) the task results before sending them back to the client.
- **Importance**: A high serialization time indicates the results are taking a long time to be converted, which may cause delays in transmitting the results to the client.

7. JVM Garbage Collection Time (jvmGCTime)

- **Description**: This shows the time spent by the Java Virtual Machine (JVM) performing garbage collection, which clears unused objects from memory.
- **Importance**: Garbage collection can impact performance if it takes too long, as it temporarily pauses the application to clean up memory. Keeping this time low is ideal for maintaining performance.

8. Shuffle Write Time (shuffleWriteTime)

- **Description**: This is the time spent writing shuffle data to disk during the process of exchanging data between tasks.
- **Importance**: Shuffling is a resource-heavy operation. The shuffle write time indicates how long the system spends transferring data, which can affect job performance.

9. Result Size (resultSize)

- **Description**: This indicates the total size of the results processed by the executor and sent to the client.
- **Importance**: The result size shows how much data is being handled at the output stage. Larger result sizes could impact memory and network performance.

10. Records Read (recordsRead)

- **Description**: This shows the total number of records read by the tasks during the job.

- **Importance**: This metric helps measure the volume of data being processed. Larger datasets typically require more computational resources.

11. Shuffle Remote Bytes Read to Disk (shuffleRemoteBytesReadToDisk)

- **Description**: The total number of bytes read from remote executors and written to disk during the shuffle phase.
- **Importance**: High amounts of remote data fetching can slow down performance, so minimizing this value where possible is ideal.

12. Shuffle Bytes Written (shuffleBytesWritten)

- **Description**: This is the total number of bytes written to disk during the shuffle phase.
- **Importance**: Shuffling involves transferring data across nodes or tasks. The amount of shuffle data written impacts disk I/O and overall shuffle efficiency.

13. Shuffle Records Written (shuffleRecordsWritten)

- **Description**: This represents the total number of records written during the shuffle process.
- **Importance**: It shows the volume of data being shuffled. A higher number of records written implies a more extensive shuffle, which could affect performance.

14. Average Number of Active Tasks (averageNumberOfActiveTasks)

- **Description**: This shows the average number of tasks that were active (running) at any given moment.
- **Importance**: This metric indicates the degree of parallelism and helps assess how effectively the system is using its resources.

15. Stage Duration (stageDuration)

- **Description**: This is the time each stage took to complete.

- **Importance**: Understanding the duration of individual stages helps identify which parts of the job are more time-consuming and may need optimization.

These metrics are crucial for gaining deep insights into system performance and are a good starting point for performance optimization in Spark jobs.

Our Test Results

Benchmark for Preprocess

- **Scheduling Mode**: FIFO
- **Spark Context Default Degree of Parallelism**: 12

Aggregated Spark Stage Metrics:

- **Number of Stages**: 1
- **Number of Tasks**: 31
- **Elapsed Time**: 15,063 ms (15 s)
- **Stage Duration**: 15,063 ms (15 s)
- **Executor Run Time**: 161,150 ms (2.7 min)
- **Executor CPU Time**: 719 ms (0.7 s)
- **Executor Deserialize Time**: 1,632 ms (2 s)
- **Executor Deserialize CPU Time**: 682 ms (0.7 s)
- **Result Serialization Time**: 29 ms
- **JVM Garbage Collection Time (jvmGCTime)**: 2,904 ms (3 s)
- **Shuffle Write Time**: 0 ms
- **Result Size**: 44.2 KB
- **Records Read**: 8,000,000
- **Shuffle Remote Bytes Read to Disk**: 0 Bytes
- **Shuffle Bytes Written**: 0 Bytes

- **Shuffle Records Written**: 0

Average Number of Active Tasks: 10.7

Stage Durations:

- **Stage 0 Duration**: 15,063 ms (15 s)

Benchmark for Word Count

- **Scheduling Mode**: FIFO
- **Spark Context Default Degree of Parallelism**: 12

Aggregated Spark Stage Metrics:

- **Number of Stages**: 2
- **Number of Tasks**: 62
- **Elapsed Time**: 60,218 ms (1.0 min)
- **Stage Duration**: 60,193 ms (1.0 min)
- **Executor Run Time**: 636,301 ms (11 min)
- **Executor CPU Time**: 7,218 ms (7 s)
- **Executor Deserialize Time**: 682 ms (0.7 s)
- **Executor Deserialize CPU Time**: 394 ms (0.4 s)
- **Result Serialization Time**: 180 ms
- **JVM Garbage Collection Time (jvmGCTime)**: 6,515 ms (7 s)
- **Shuffle Write Time**: 1,280 ms (1 s)
- **Result Size**: 63.3 KB
- **Records Read**: 8,000,000
- **Shuffle Remote Bytes Read to Disk**: 0 Bytes
- **Shuffle Bytes Written**: 39.7 MB
- **Shuffle Records Written**: 13,454

Average Number of Active Tasks: 10.6

Stage Durations:

- **Stage 1 Duration**: 57,418 ms (57 s)

Benchmark for Sorting

- **Scheduling Mode**: FIFO
- **Spark Context Default Degree of Parallelism**: 12

Aggregated Spark Stage Metrics:

- **Number of Stages**: 4
- **Number of Tasks**: 124
- **Elapsed Time**: 9,002 ms (9 s)
- **Stage Duration**: 8,900 ms (9 s)
- **Executor Run Time**: 92,457 ms (1.5 min)
- **Executor CPU Time**: 2,515 ms (3 s)
- **Executor Deserialize Time**: 683 ms (0.7 s)
- **Executor Deserialize CPU Time**: 449 ms (0.4 s)
- **Result Serialization Time**: 259 ms
- **JVM Garbage Collection Time (jvmGCTime)**: 2,100 ms (2 s)
- **Shuffle Write Time**: 207 ms (0.2 s)
- **Result Size**: 68.9 KB
- **Records Read**: 0
- **Shuffle Remote Bytes Read to Disk**: 0 Bytes
- **Shuffle Bytes Written**: 2.6 MB
- **Shuffle Records Written**: 2,381

Average Number of Active Tasks: 10.3

Stage Durations:

- **Stage 4 Duration**: 2,381 ms (2 s)
- **Stage 6 Duration**: 2,415 ms (2 s)
- **Stage 8 Duration**: 2,634 ms (3 s)
- **Stage 9 Duration**: 1,470 ms (1 s)