**Name: Suyash Tandon**

# Assignment 1-Report

## 1. Configuration used for experimental set-up of Google Instance:

No. of CPUs= 2

Memory= 4 GB

Disk Space= 20GB

OS= Ubuntu 16.04 LTS (Xenial image built)

Remote Access method: SSH

## 2. Steps to Enable Docker container:

After installing the Docker container to our native system, by using the steps already given, below are the steps need to enable/run the docker container on our native system:

**STEP 1:**

Run the below command that runs the docker image with sysbench pre-installed on it:

- sudo docker run -itd csminpp/ubuntu-sysbench

This command gives the container ID of the container running as an output.

For eg:

```
3552b7f4f43dd7ea5fe975f6f0a763864c35dd73522869666c46340df45b6cb4
```

**STEP 2:**

Now, using the below command, we attach our terminal's standard I/O and error to the already running container using its container ID.

- `sudo docker attach 355`

Other useful operations:

- docker ps
  This command is used to list the information related to all the the container running currently on the docker. From where we can collect the container ID of the required container.

- docker ps -all
  This command is a corollary to the above command. It shows both the running as well as exited container details.

## 3. Steps to enable/run QEMU VM container:

After installing the OS image on the native system and configuring the GUI, by using the steps already given, follow the below steps to enable the QUEM VM:

**Step 1:**

Run the below command on the native server every time you start the GCP Instance , to start the GUI service( for my setup, I have use VNC Viewer).:

- vncserver

**Step 2:**

Now, using the external IP of your instance, connect to the VM using the VNC Viewer.

Open terminal in the Google instance, and type the below command to connect the QEMU VM:

- $sudo qemu-system-x86_64 -hda ubuntu.img -m 1536

Now, enter your login details which you created during installation and your QEMU VM starts.

**QEMU VM Configuration:**

CPU(s): 1

Disk Space: 10  GB

Proxy Setup: None

## 4. Conducting performance measurements in different Scenarios:

Run the below command to measure the CPU performance and display it on the terminal every 5 seconds.

- iostat 5

NOTE: For all the different scenarios, only one common instance of iostat is required which will report the CPU statistics, using which we can measure the CPU performance for different tests.

**<u>On Native system:</u>**

On native system, install sysbench using:

- apt-get install sysbench

Then run the below steps to perform the tests.

**1. CPU Test**

- Run the below command to run the CPU test, which calculates the number of primes upto 25000.

  ```
  sysbench --num-threads=2  --test=cpu --cpu-max-prime=25000 run
  ```

**2. File IO Test**

- The below command prepares files of size 3GB

  ```
  sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rndrw prepare
  ```

- The below command is used to run the above created files.

  ```
  sysbench --num-threads=16 --test=fileio –file-t numotal-size=3G --file-test-mode=rndrw run
  ```

**<u>On Docker:</u>**

test Commands are same as Native.

**<u>On QEMU VM:</u>**

Test Commands are same as Native.

## 5. Performance Data for different scenarios for the two tests:

### 1. CPU Test

Using the same values for all three scenarios for optimum performance data:

```
sysbench --num-threads=1  --test=cpu --cpu-max-prime=35000 run
```

| Scenario\Output Data | % cpu usage | Total Time taken for execution | Per-request average |
|---|---|---|---|
| Native | 50 | 61.82s | 6.18s |
| Docker | 50 | 61.60s | 6.16s |
| QEMU | 50 | 148.25s | 14.82s |

Changing the number of threads and running the command :

```
sysbench --num-threads=2  --test=cpu --cpu-max-prime=35000 run
```

| Scenario\Output Data | % cpu usage | Total Time taken for execution | Per-request average |
|---|---|---|---|
| Native | 100 | 33.76s | 6.75s |
| Docker | 100 | 33.50s | 6.70s |
| QEMU | 50 | 140.83s | 29.75s |

**2. FileIO Test**

Using the below command for all the three scenarios.

sysbench --num-threads=1 --test=fileio –file-t numotal-size=3G --file-test-mode=rndrw run

| Scenario\ Output Data | Total Time/Latency | Kb read/sec | Kb write/sec | TPS | IOWait % |
|---|---|---|---|---|---|
| Native | 57.8s | 1075 | 1215 | 347 | 29.18 |
| Docker | 56.75 | 1091 | 941 | 329 | 28.16 |
| QEMU | 137.8 | 5964 | 664 | 281 | 30.70 |

Changing the number of threads to 2, we get;

sysbench --num-threads=2 --test=fileio –file-t numotal-size=3G --file-test-mode=rndrw run

| Scenario\ Output Data | Total Time/Latency | Kb read/sec | Kb write/sec | TPS | IOWait % |
|---|---|---|---|---|---|
| Native | 17.5s | 6347 | 3494 | 1193 | 63.22 |
| Docker | 26.3s | 2595 | 2770 | 842 | 64.04 |
| QEMU | 110.80s | 2288 | 838 | 316 | 32.01 |

## 6. Inference

Upon the analysis of the above extracted data, we see that the data for benchmark test on native system as well as the data for benchmark test on the docker is same.

Whereas, for QEMU the performance data is worse when compared to the other two.

**Reason:**

Because QEMU has 1 CPU to run all the computation, whereas the docker(which runs on the native system as a process) and the native system itself has 2 CPUs to perform the computation. Which gives the native system and docker more resources to perform the computation which in turn makes the faster and hence better performance data.

This can be proved by changing the number of threads. When changing the threads, we can see that changing the threads has little or no effect on the CPU usage(computed using IOSTAT) as it still uses the 1 CPU assigned to it out of the 2 CPUs of the native system. So, it does not matter how many threads are created, the % usage won't be more than 50% for QEMU.