Alla Topp
MSDS 696 – Data Science Practicum

# Testing Virtual machine performance with PostgreSQL database in Cloud Stack with 4 and 8GB of RAM, 4CPU cores and 40GB of hard drive.

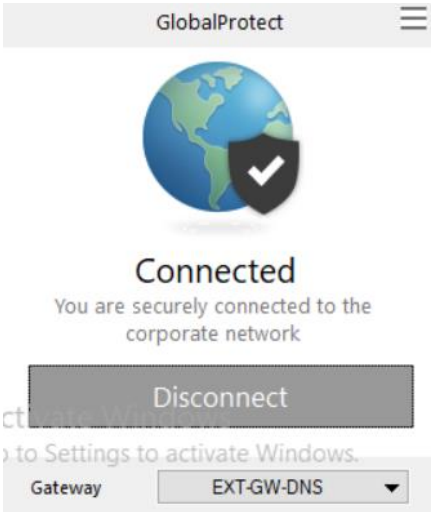
In this documentation I applied multi-processing to the PostgreSQL database on 2 virtual machines with different specs in CloudStack and tested their performance by analyzing queries run time. The first virtual machine specifications are 4 CPU cores, 4GB of RAM and 40GB of hard drive space. The second virtual machine specifications are 4 CPU cores, 8GB of RAM and 40 GB of hard drive space. Multi-processing is done in Jupyter Hub (Python) in Cobalt. In order to do multi-processing, I run 5 different queries simultaneously testing 1, 10, 20, 50 and 100 processes (users) and recorded their run times. The whole process took 3 steps:

- Connecting to PostgreSQL database (yelpdb) with psycopg2 python library by using VM`s IP address, port, username and database name.
- Creating and applying multi-processing function with multiprocessing python library and calling the function after running queries. To be able to simulate number of users I multiplied each query by the number of processes, then called the multiprocessing function. To time the run time of each query process I applied %%timeit magic function.
- Creating graphs when all average run times collected. For accurate result, I ran each query 7 times with each number of processes.

Since I tested performance of both VMs, I created comparison graphs to see how long it took to run each query with each number of processes.

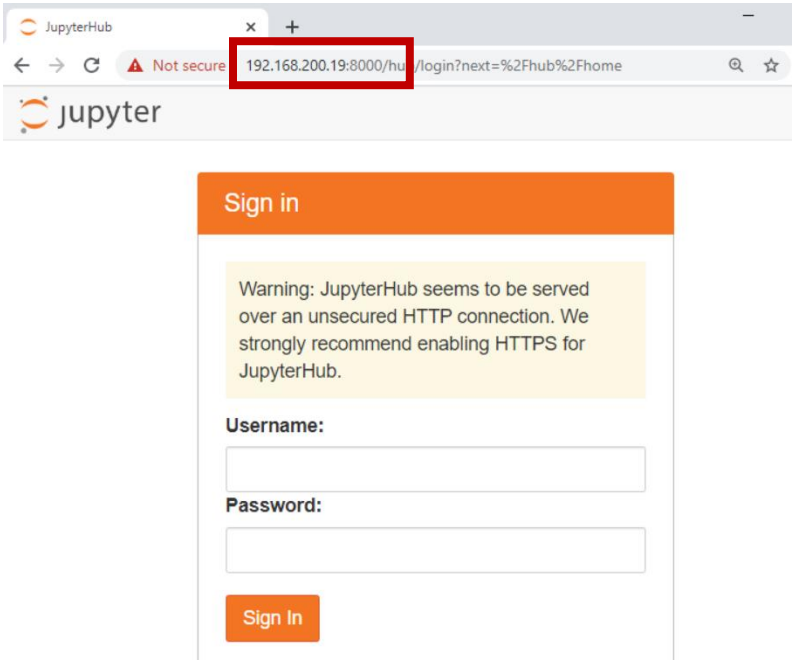
## Step 1. Connect to Global Protect

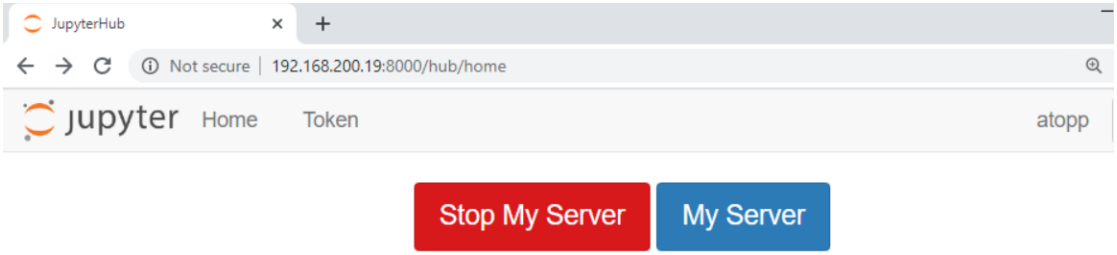
To connect to the virtual machine on Cloud Stack, connect to the private network Global Protect first on PC or Windows Virtual Machine like shown below.



## Step 2. Connect to Jupiter Hub (cobalt)

Open the browser and paste the IP address (192.168.200.19:8000) to be connected to the JupyterHub. Then log in with your username and password with cobalt account. Then proceed with "My Server" and the Jupyter environment will be ready.




## Step 3. Install python libraries

To work with PostgreSQL in Jupiter Lab, the psycopg2 and pygresql should be installed. Working in Jupyter Notebook, it can be done with pip install like on the print screen below (in Jupyter Notebook):

```
!pip install psycopg2
```

```
!pip install pygresql
```

Although, working in Jupiter Hub in cobalt, it should be installed in cobalt. In order to do it, follow the steps below:

- Log in to cobalt through ssh (PuTTY) with root user and its password, then follow 3 commands below for installation:
- [root@cobalt ~]# yum install python3-devel
- [root@cobalt ~]# pip3 install psycopg2
- [root@cobalt ~]# pip3 install pygresql

## Step 4. Connect to the postgres database on virtual machine in CloudStack in Python

First, import two libraries before writing the code in Jupyter Hub. To work with PostgreSQL and multi-processing import psycopg2 and multiprocessing:

```
import psycopg2
import sqlalchemy
import matplotlib as plt
import concurrent.futures
import multiprocessing
```

Second, create function which connects to PostgreSQL database with psycopg2 using host, port, database name and password (if the database was created with no password, it should not be included). Below print screen shows the connection function with an existing virtual machine and psql parameters that was used in the project:

```
def doCountQuery(query):
    conn = psycopg2.connect(host = "192.168.200.194", port = "5432", user = "postgres",
                                                                    database = "yelpdb")
    cur = conn.cursor()
    cur.execute(query)
    query_results = cur.fetchall()

#    print(query_results)
    conn.close()
```

## Step 5. Multi-processing

Function below calls queries with several users by including connection function and each query name in PSQL:

```
# define function multi-processing

def mLProcess(queries):
    with multiprocessing.Pool() as pool:
        pool.map(doCountQuery, queries)
```

## Step 6. Testing queries with multi-processing and documenting results (run time) in the comments

Since there are created functions with connection and multiprocessing function, define all 5 queries and test several processes by calling the functions like shown below for each query. Time the runs when calling the multiprocessing function with magic function %%time which shows wall time for documentation. By multiplying each query run by number of users (processes), the run time for that amount of processes will be received:

### Query 1 test

Count number of rows in the largest table ("review").

Define new name for the first query and include number of processes, then call multiprocessing function mLProcess(). After that CPU times and wall time will be shown. Wall time will be recorded after finding out the average. In this project, all queries were run without an output.

```
# Q1.Count number of rows in largest table "review"
queries = ["""SELECT COUNT(*) FROM review;"""]*20 #multiply by ammount of users (1, 10, 20, 50, 100)
```

```
%%time
mLProcess(queries)

#8GB RAM VM
# Running each amount of processes 7 times to calculate the average result
#1. 2.04s, 1.93s, 1.83s, 1.73s, 1.83s, 1.83s, 1.84s
#10. 11.4s, 11s, 11.1, 11.2s, 11.3s, 11.2, 11.3s
#20. 54.2s, 22.4s, 22.2s, 22.2s, 21.9s, 22.1s, 22.1s
#50. 51.1s, 56.1s, 55.3s, 55.8s, 55.3, 55.9s, 55.5s
#100. 1min 50s, 1min 50s, 1min 51s, 1min 52s, 1min 51s, 1min 53 s, 1min 53s
```

```
CPU times: user 109 ms, sys: 110 ms, total: 219 ms
Wall time: 55 s
```



### Query 2 test

Select all distinct names in the user table.

Define new name for the second query and include number of processes, then call multiprocessing function mLProcess():

```
#Q2. Select all distinc names in the user table
distinct = ["""SELECT DISTINCT(user_name) FROM y_user;"""] * 20
```

```
%%time
mLProcess(distinct)

# of Users:
#1. 1.64s, 1.63s, 1.73s, 1.73s, 1.73s, 1.64s, 1.63s
#10. 11.3s, 4.04s, 4.24s, 3.84s, 3.84s, 3.84s, 3.85s
#20. 7.37s, 7.37s, 7.07s, 6.97s, 7.26s, 7.38s, 6.96s
#50. 18.3s, 18.7s, 18.5s, 18.6s, 17.9s, 17.3s, 18.3s
#100. 37.1s, 37.5s, 36.7s, 36.6s, 36.5s, 36.9s, 37.1s
```

```
CPU times: user 44.7 ms, sys: 90.1 ms, total: 135 ms
Wall time: 7.37 s
```

**Query 3 test.**

Count occurrence of each name in the user table.

Define new name for the second query and include number of processes, then call multiprocessing function mLProcess():

```
#Q3. Count occurrence of each name in the user tabel
occurrence = ["""SELECT user_name, COUNT(*) FROM y_user GROUP BY user_name;"""] * 20
```

```
%%time
mLProcess(occurrence)

#1. 1.23s, 1.23s, 1.25s, 1.23s, 1.23s, 1.33s, 1.23s
#10. 5.16s, 5.04s, 5.05s, 5.15s, 4.95s, 5.25 s, 5.25s
#20. 11.1s, 9.19s, 8.98s, 9.07s, 8.98s, 9.37s, 9.08s
#50. 23.1s, 22.7s, 22.8s, 23.2s, 23s, 22.2s, 22.9s
#100. 48.8s, 45.2s, 45.1, 45.1s, 45.3, 45.3s, 45s
```

```
CPU times: user 46.2 ms, sys: 92.4 ms, total: 139 ms
Wall time: 10.1 s
```

**Query 4 test.**

Count number of reviews by each user using join.

Define new name for the second query and include number of processes, then call multiprocessing function mLProcess():

```
#Q4. Count number of reviews by each user
num_reviews = ["""SELECT y_user.user_name, COUNT(review.text)
                AS Number FROM y_user INNER JOIN review ON
                y_user.user_id = review.user_id GROUP BY y_user.user_name LIMIT 100;"""] * 3
```

```
%%time
mLProcess(num_reviews)

# 8GB RAM VM
#1. 4min 9s, 3min 52s, 3min 32s, 4min 41s, 3min 50s, 4min 32s, 4min 44s.
#3. 24min 47s, 29min 35s, 28min 25s, 28min 20s, 27min 59s, 29min 53s, 27min 59s
#5. 1hr 20s, 1hr 1min 49s, 1h 49s, 52min 56s, 1hr 2min 10s, 1hr 37s, 1h 3s
```

**Query 5 test**

Create any query to try including all the tables with multiple joins.

Define new name for the second query and include number of processes, then call multiprocessing function mLProcess():

```
#Q5.Multiple joins including all the tables
mul_joins = ["""SELECT y_user.user_name, review.stars, tip.text, business.name
               FROM y_user FULL OUTER JOIN review ON y_user.user_id = review.user_id
               FULL OUTER JOIN tip ON review.user_id = tip.user_id
               FULL OUTER JOIN business ON tip.business_id = business.business_id;"""]*3
```

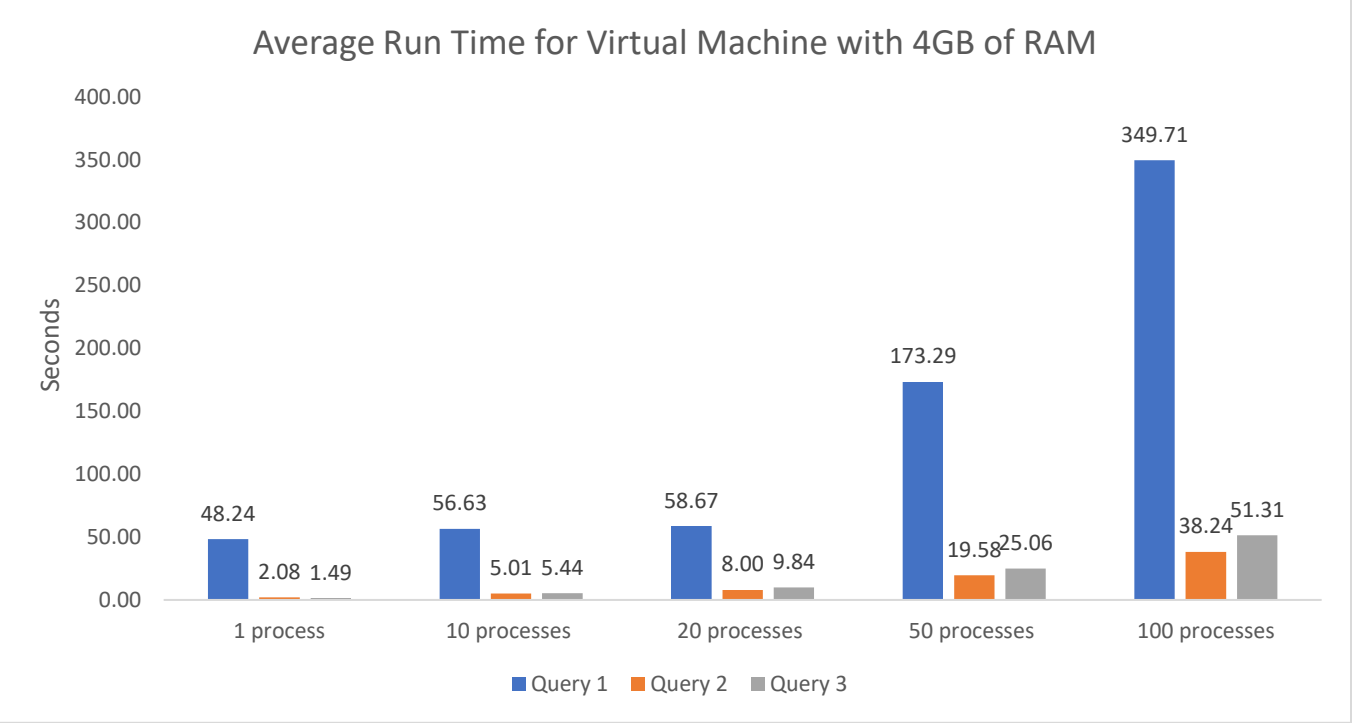
```
%%time
mLProcess(mul_joins)

#8GB RAM
#1. 40min 12s, 37 min 40s, 37min 11s, 38min 50s, 36 min 28s, 37min 16s, 36min 47s
```

## Step 7. Graphing the results in excel

Below there is a table with all the average run times for each query and processes that have been tested. Since the virtual machine contains only 40GB of hard drive, it is impossible to run queries 4 and 5 because they contain joins and require temporary space on the hard drive. To run those queries with 10, 20, 50 and 100 processes, the VM would need up to 500GB of free disk space.

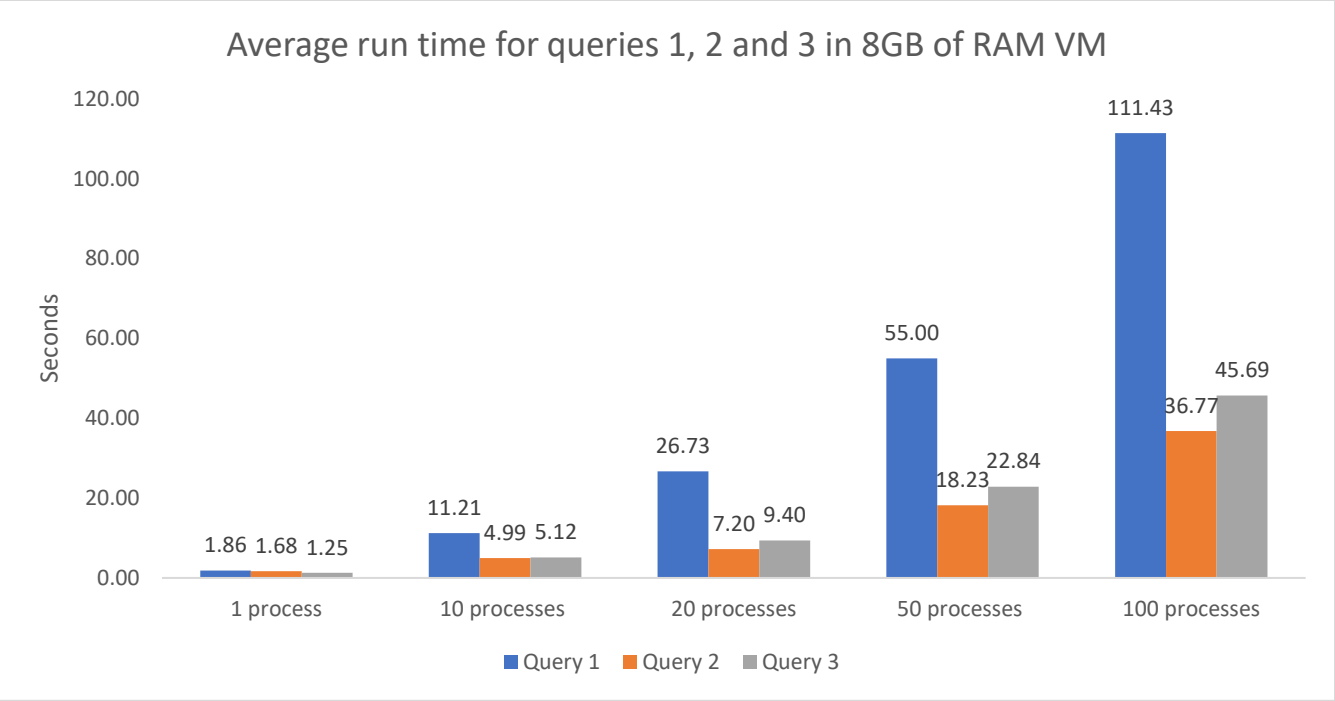
| | Run time for virtual machine with 4GB of RAM in seconds | | | | |
|---|---|---|---|---|---|
| | **1 process** | **10 processes** | **20 processes** | **50 processes** | **100 processes** |
| Query 1 | 48.24 | 56.63 | 58.67 | 173.29 | 349.71 |
| Query 2 | 2.08 | 5.01 | 8.00 | 19.58 | 38.24 |
| Query 3 | 1.49 | 5.44 | 9.84 | 25.06 | 51.31 |
| Query 4 | 409.29 | n/a | n/a | n/a | n/a |
| Query 5 | 3075.86 | n/a | n/a | n/a | n/a |




**Step 8. Creating new virtual machine and repeating the whole process with the new VM.**

The new virtual machine has the same amount CPU cores and hard drive space, but different RAM size (8GB). Run time (performance) of the same queries will be analyzed and compared to the 4GB RAM virtual machine. After that, the run time table and graph for the new VM will be compared to the data from the 4GB RAM VM .

| | Run time for virtual machine with 8GB of RAM in seconds | | | | |
|---|---|---|---|---|---|
| | **1 process** | **10 processes** | **20 processes** | **50 processes** | **100 processes** |
| Query 1 | 1.86 | 11.21 | 26.73 | 55.00 | 111.43 |
| Query 2 | 1.68 | 4.99 | 7.20 | 18.23 | 36.77 |
| Query 3 | 1.25 | 5.12 | 9.40 | 22.84 | 45.69 |
| Query 4 | 251.43 | n/a | n/a | n/a | n/a |
| Query 5 | 2266.29 | n/a | n/a | n/a | n/a |

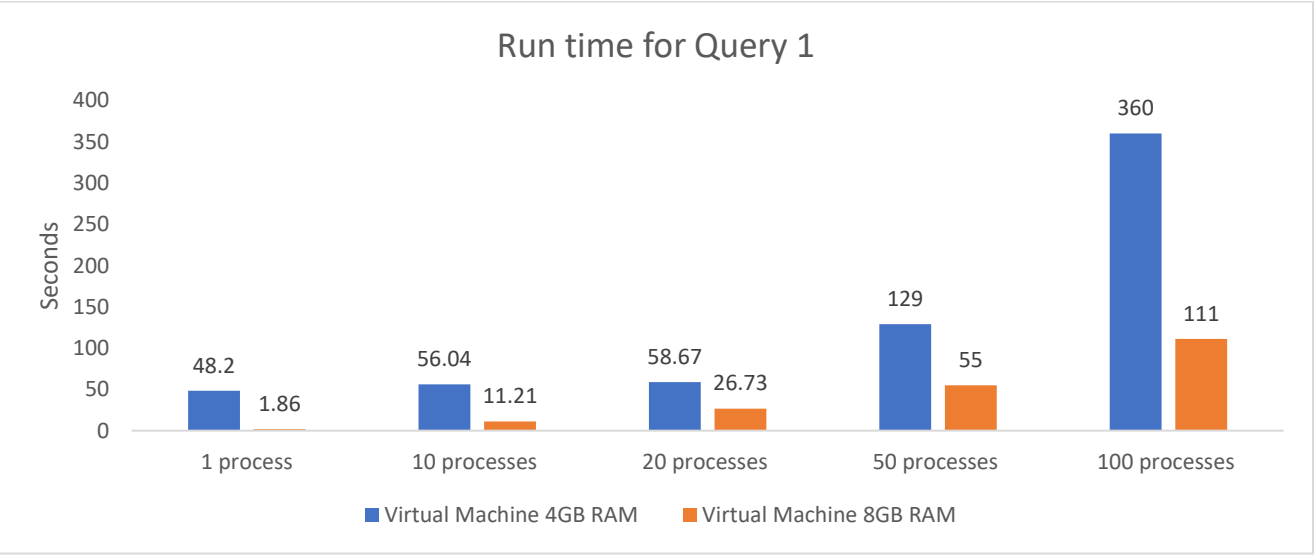



Looking at the graph, there is gradual raise of time while running query 2 and 3, but when running query 1 there is a rapid rise in time as the number of processes increase.

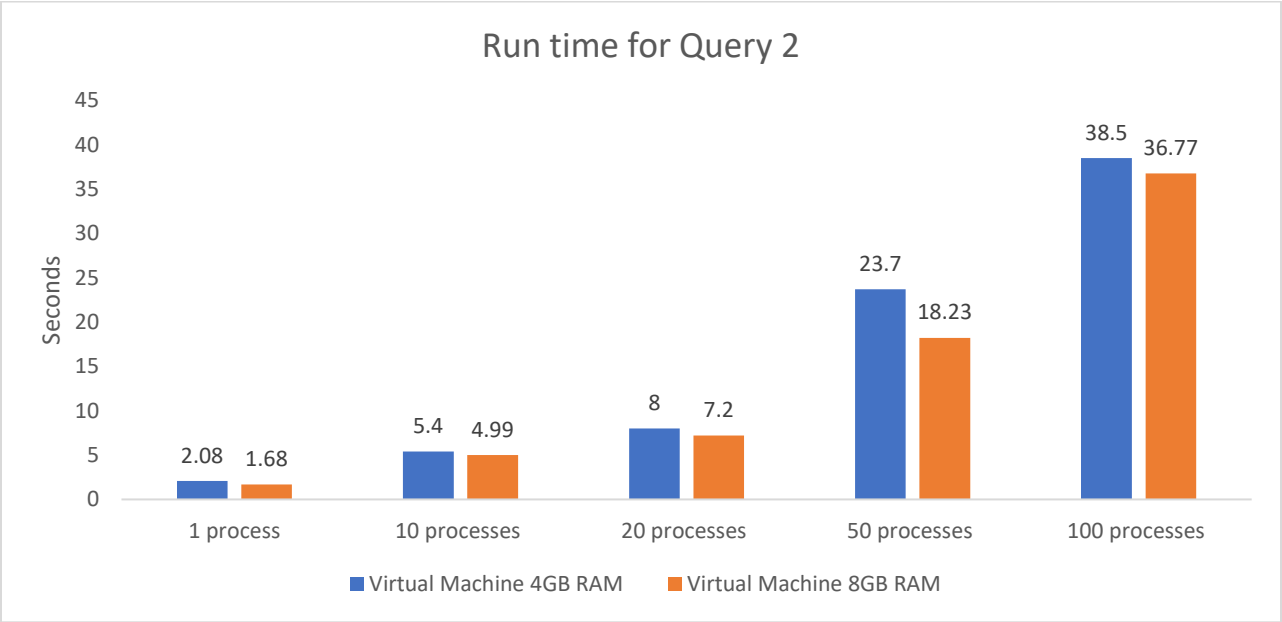
**Comparison of average run time between virtual machine with 4GB of RAM and virtual machine with 8 GB of RAM**

Since all 5 queries were run in both VMs, they could be compared for better performance. Each query will be compared for performance.

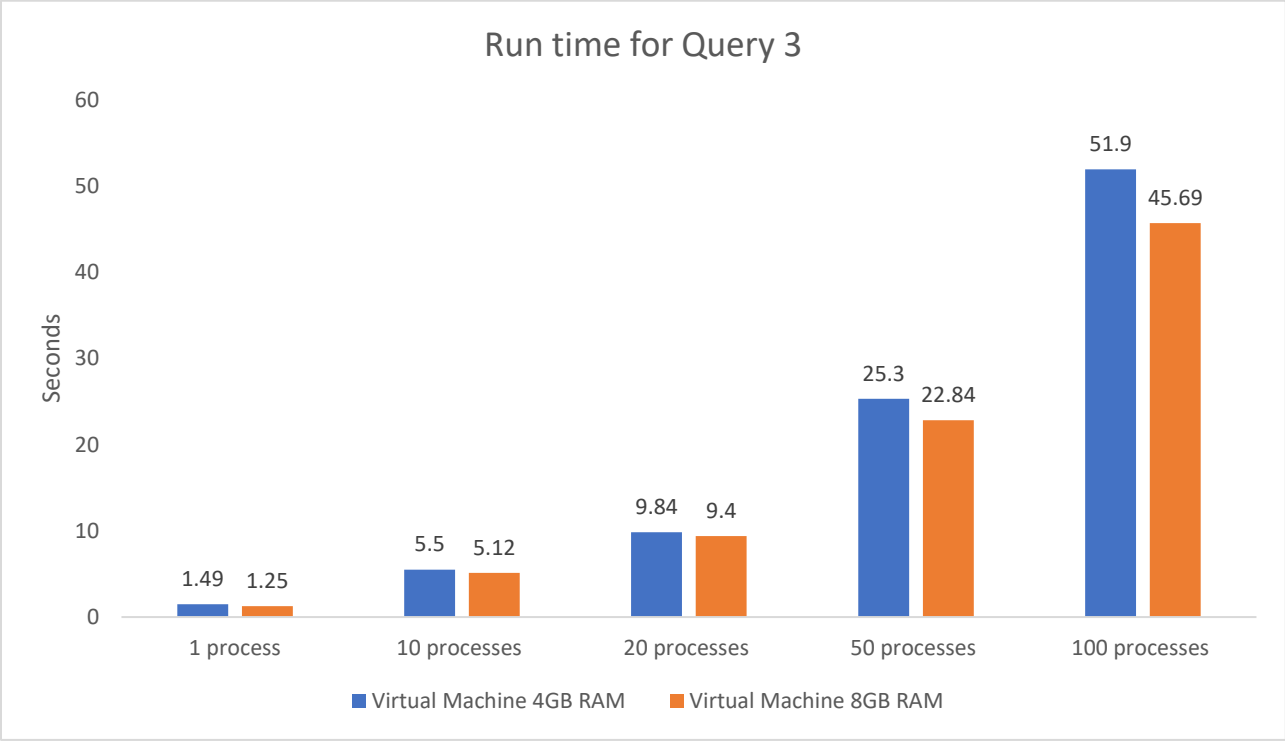
**Query 1:**



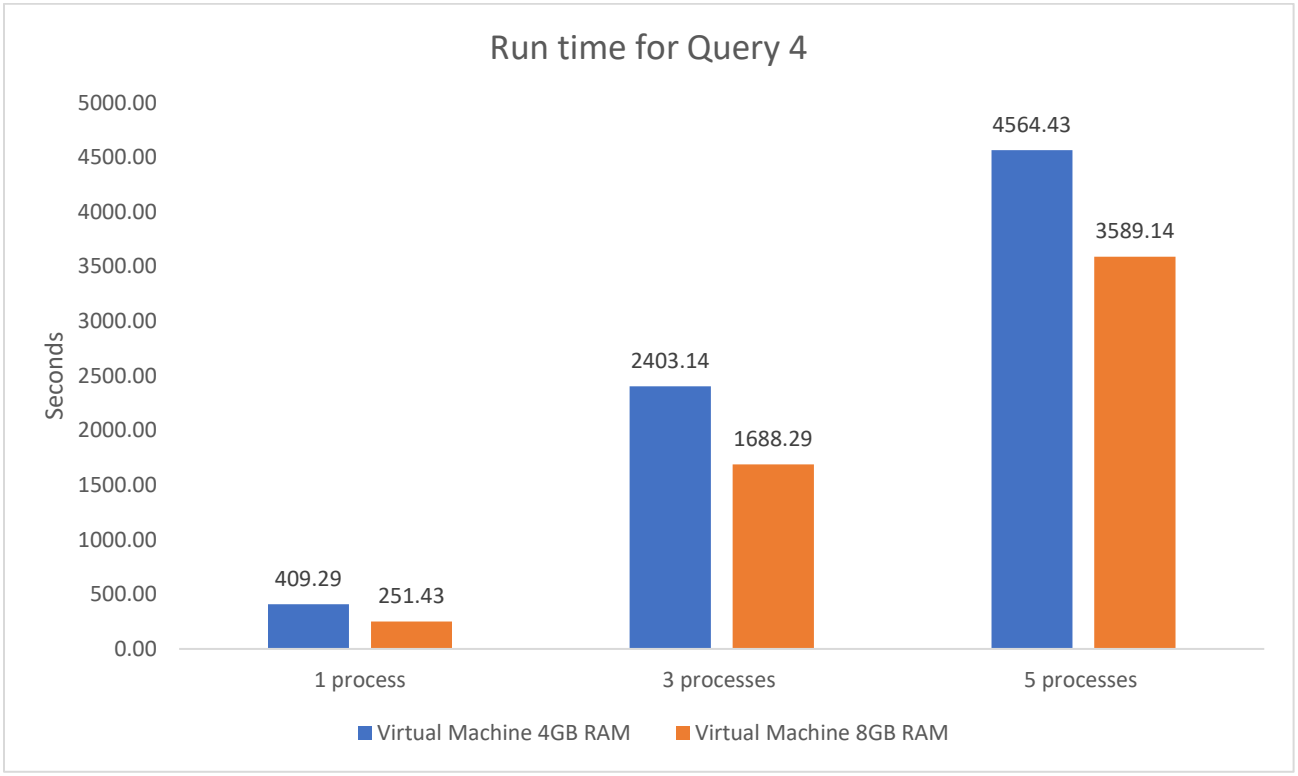
Looking at the query 1 graph between first and the second VMs, there is a huge difference. This query requires a lot of work because it is counting every rows of a 3.5 GB data in a table. There is a significant difference in the performance between the two VMs especially when running 1 vs 100 processes.



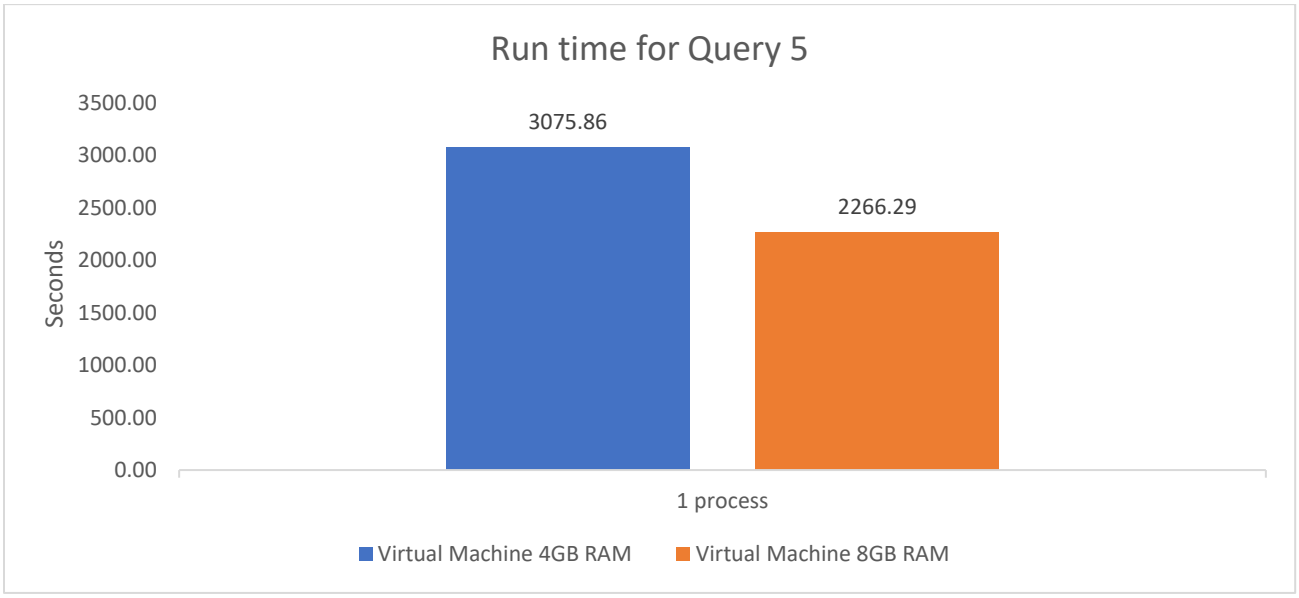

There is a noticeable difference running query 2, especially while running 50 and 100 processes.



To run this query, it takes the least amount of time in comparison with other queries, so the difference between processes and VMs is not very big. Although, it increases in 50 and 100 processes.




Since query 4 with joins is unable to run with 10, 20, 50 and 100 processes, run time with 1, 3 and 5 processes were recorded. The virtual machine with 8GB of RAM has a much better performance as expected, especially with larger tables and queries with joins. The more time it takes a query to run, the better the VM with 8GB of RAM will perform.

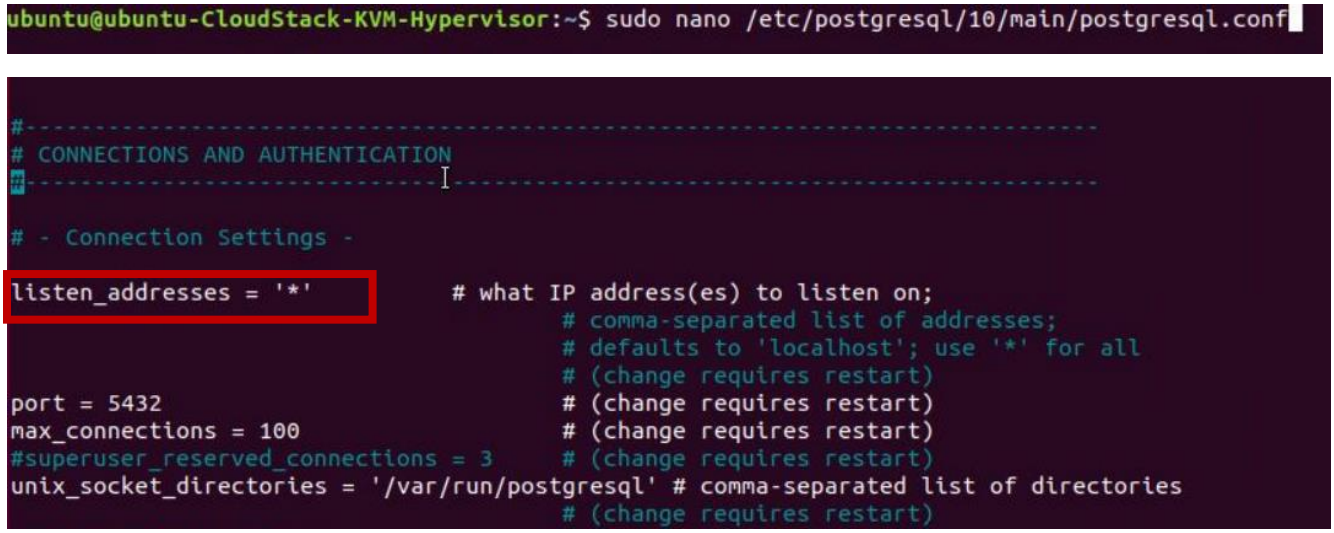


Again, there is a significant difference between running the same query in the two VMs. The 4GB RAM VM showed 51 min of run time approximately and the 8GB RAM VM showed 38 min of run time. The difference is over 10 minutes!

**Issues while connecting to the VM database in Python.**

When running the connection function with VM IP address, most likely an error connecting to the database will be received when working with Ubuntu VM. In this case, the problem is in the configuration PostgreSQL files on the virtual machine it is connecting to. In order to fix that problem, follow the steps below:

1. Open the virtual machine in CloudSack
2. In Linux terminal open the files postgresql.conf and change listen_addresses = '*' and uncomment it. It allows to listen to all IP addresses.




3. Then open file pg_hba.conf with sudo nano command and changes a few lines in the file:

```
#
# Database administrative login by Unix domain socket
local   all             postgres                                peer

# TYPE  DATABASE        USER            ADDRESS                 METHOD

# "local" is for Unix domain socket connections only
local   all             all                                     peer
# IPv4 local connections:
# host    all             all             127.0.0.1/32            md5
host    all             all             0.0.0.0/0               trust
# IPv6 local connections:
host    all             all             ::0/0                   trust
# Allow replication connections from localhost, by a user with the
# replication privilege.
local   replication     all                                     peer
host    replication     all             127.0.0.1/32            md5
host    replication     all             ::1/128                 md5
```

After updating both files, restart PostgreSQL and try to connect to the database in python.

```
ubuntu@ubuntu-CloudStack-KVM-Hypervisor:~$ sudo systemctl restart postgresql
```

# References

Claria, F., Claria, F., & Axones. (2018, December 04). An Introduction to Using SQL Aggregate Functions with JOINs. Retrieved from https://academy.vertabelo.com/blog/introduction-using-aggregate-functions-joins/

DaniloDanilo 2, WolphWolph 63.7k88 gold badges113113 silver badges138138 bronze badges, Raj MoreRaj More 42.6k2929 gold badges118118 silver badges188188 bronze badges, AdaTheDevAdaTheDev 119k2424 gold badges175175 silver badges180180 bronze badges, Marc_smarc_s 632k143143 gold badges12001200 silver badges13331333 bronze badges, & Alex HowanskyAlex Howansky 41k55 gold badges6262 silver badges8989 bronze badges. (1960, June 01). Count number of users from a certain country. Retrieved from https://stackoverflow.com/questions/3773351/count-number-of-users-from-a-certain-country

Mitrani, A. (2019, November 08). Python and PostgreSQL: How To Access A PostgreSQL Database Like A Data Scientist. Retrieved from https://towardsdatascience.com/python-and-postgresql-how-to-access-a-postgresql-database-like-a-data-scientist-b5a9c5a0ea43

Multi-Table Joins. (n.d.). Retrieved from http://etutorials.org/SQL/Postgresql/Part I General PostgreSQL Use/Chapter 1. Introduction to PostgreSQL and SQL/Multi-Table Joins/

PostgreSQL - JOINS. (n.d.). Retrieved from https://www.tutorialspoint.com/postgresql/postgresql_using_joins.htm

Python PostgreSQL - Limit. (n.d.). Retrieved from https://www.tutorialspoint.com/python_data_access/python_postgresql_limit.htm