Pgpool Installation + Connection Test With Python

Pgpool-II is a postgresql middleware tool. It provides load balancing between postgresql servers, additional features have been added over time. These features are as follows

- **Connection Pooling :** Pgpool-II saves connections to PostgreSQL servers and reuses them when a new connection with the same properties (e.g. username, database, protocol version) arrives. It reduces connection overhead and increases the overall throughput of the system

- **Replication:** Multiple PostgreSQL servers can be managed with Pgpool-II. The use of the Replication function allows the creation of a real-time backup on 2 or more physical disks, so that in case of a disk failure the service can continue without stopping the servers

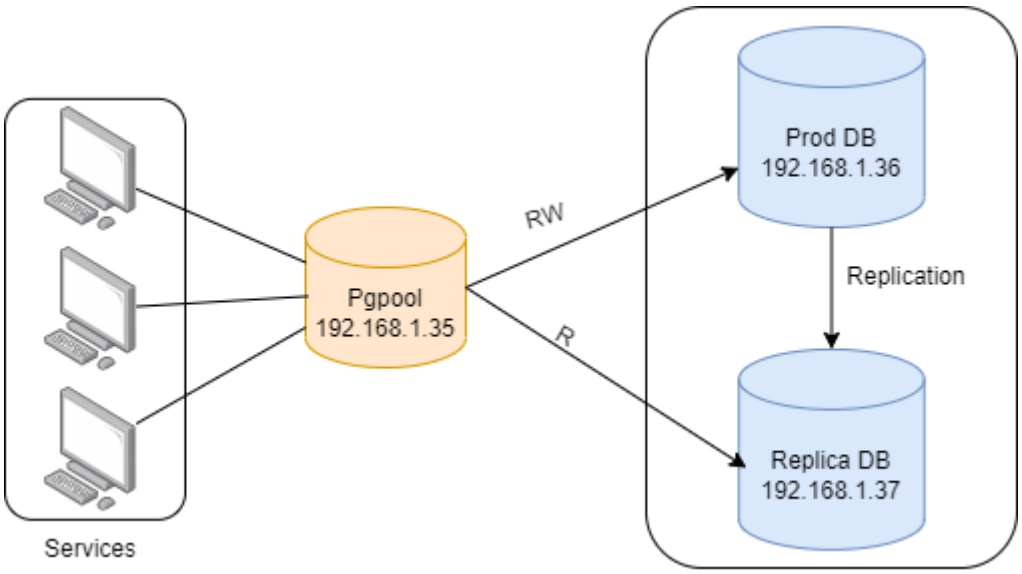
- **Load Balancing:** If a database is replicated, any SELECT query run on the master or replica database will return the same result because the two databases are synchronized. Pgpool-II takes advantage of load balancing to distribute SELECT queries across multiple servers to reduce the load on each PostgreSQL server

- **Limiting Exceeding Connections:** PostgreSQL has a limit on the maximum number of connections and connections are rejected after this limit is exceeded. Increasing the maximum number of connections increases resource consumption and negatively affects system performance. With Pgpool-II, a limit can be set on the maximum number of connections and when this limit is exceeded, connection requests are queued, not rejected

- **Watchdog:** Watchdog can coordinate multiple Pgpool-IIs and thanks to the cluster structure we can avoid the possibility of single point of failure. Watchdog can use other pgpool-II nodes to detect a Pgpoll-II failure. If one node goes down, the other pgpool node will detect it. If the active Pgpool-II goes down, the Pgpool-II in wait state can be promoted to active state and take over the Virtual IP

- **In Memory Query Cache:** The set SELECT queries and their results are stored in the cache. If the same SELECTs come back, Pgpool-II returns the value from the cache. Since no sql parsing or access to postgresql is involved, cache usage is extremely fast

Installation



Our prod and replica db's installed in Postgresql 15 are synchronized, we will distribute requests to these two db's from the server where we will install pgpool and we will test and observe this with a python script

```
--pgpool repo (192.168.1.35)
yum install -y https://www.pgpool.net/yum/rpms/4.4/redhat/rhel-7-x86_64/pgpool-II-
release-4.4-1.noarch.rpm
```

For pgpool installation, a postgresql database must also be installed on the server

```
--postgresql repo (192.168.1.35)
yum install -y https://download.postgresql.org/pub/repos/yum/reporpms/EL-7-
x86_64/pgdg-redhat-repo-latest.noarch.rpm

yum install -y postgresql15-server

yum install -y postgresql15-contrib

/usr/pgsql-15/bin/postgresql-15-setup initdb

yum install -y pgpool-II-pg15-debuginfo

yum install -y pgpool-II-pg15

yum install -y pgpool-II-pg15-devel

yum install -y pgpool-II-pg15-extensions

--create monitor user and databases(192.168.1.36)
CREATE USER monitor WITH  PASSWORD 'monitor@password';
CREATE DATABASE monitor;
```

At this point we have completed our pgpool installation. Now we will do the configurations

```
--pgpool conf file (192.168.1.35)

vi /etc/pgpool-II/pgpool.conf >>
listen_addresses='*'
port=9999
backend hostname0 = '192.168.1.36'
backend port0 = 5432
backend_weight0 = 1
backend flag0 = 'ALLOW TO FAILOVER'

backend hostname1 = '192.168.1.37'
backend port1 = 5432
backend_weight1 = 1
backend flag1 = 'ALLOW TO FAILOVER'

sr check period = 10
sr_check_user = 'monitor'
sr check password = 'monitor@password'
sr check database = 'monitor'
delay_threshold = 10240

connection_cache = on
load balance mode = on
master slave mode = on
master_slave_sub_mode = 'stream'

enable_pool_hba = on
pool_passwd = 'pool_passwd'
```

```
allow clear text frontend auth = on
ssl = on
ssl_ciphers = 'HIGH:MEDIUM:+3DES:!aNULL'
```

We have edited the pool_hba.conf file, where we manage Pgpool authentication, as follows to use the scram-sha-256 method for local access and external requests

```
[root@pgpool pgpool-II]# cat /etc/pgpool-II/pool_hba.conf
# "local" is for Unix domain socket connections only
local   all         all                                     scram-sha-256
# IPv4 local connections:
host    all         all         0.0.0.0/0             scram-sha-256
host    all         all         127.0.0.1/32          scram-sha-256
host    all         all         ::1/128               trust
```

We need to make a similar edit in pg_hba.conf file

```
-- add the following line to the hba file and reload it
host    all             all             192.168.1.35/32      scram-sha-256

postgres=# SELECT pg_reload_conf();
```

We can use the following command to observe which server ip it goes to at the moment

```
watch -n1 'psql -h 192.168.1.35 -U monitor -p 9999 postgres  -c "select
inet_server_addr();"'
```

The command that we instantly observe which database the requests are directed to

```
watch -n1 'psql -h 192.168.1.35 -U monitor -p 9999 postgres  -c "show pool_nodes;"'
```

Now we will use a python code to insert then select and observe which node it goes to. This example especially important because initially the session will connect to the prod because it will insert, but when it starts to select it will be redirected to the replica server and the prod

Let’s create the table that the Python script will insert

```
CREATE TABLE public.employee
(
    first name character(20) COLLATE pg catalog."default" NOT NULL,
    last_name character(20) COLLATE pg_catalog."default",
    age integer,
    sex character(1) COLLATE pg catalog."default",
    income double precision
);
```

And python code:

```
yum install python-psycopg2

vi test.py>>
import psycopg2

try:
    connection = psycopg2.connect(user="monitor",
                                  password="monitor@password",
                                  host="192.168.1.35",
                                  port="9999",
                                  database="monitor")
    cursor = connection.cursor()

    cursor.execute('''INSERT INTO EMPLOYEE(FIRST_NAME, LAST_NAME, AGE, SEX,INCOME)
VALUES ('Can', 'Can Ucanefe', 26, 'M', 9000)''')

    connection.commit()
    count = cursor.rowcount
    print(count, "Record inserted successfully into employee table")
    print("Table After updating record ")
    sql_select_query = """select * from public.employee"""
    cursor.execute(sql select query)
    record = cursor.fetchone()
    print(record)
except (Exception, psycopg2.Error) as error:
```

```
        print("Failed to insert record into employee table", error)

    finally:
        # closing database connection.
        if connection:
            cursor.close()
            connection.close()
            print("PostgreSQL connection is closed")
```

With python code, we can first execute an insert and then select to observe which nodes it goes to

```
-- run the script every second
watch -n1 python test.py

-- check select counts
watch -n1 'psql -h 192.168.1.35 -U monitor -p 9999 postgres  -c "show pool_nodes;"'
```

Conclusion:
If you are looking for a load balancing solution at the database layer, pgpool will do the job, but it is also possible to solve the management of read-only transactions at the software layer