How To setup Pgbouncer

PostgreSQL handles connections in a way that can be resource-intensive. Every time a new connection is made, PostgreSQL creates a separate process, which can be costly and inefficient, especially with many connections.

When a client requests something from PostgreSQL, it opens a connection to the database, and this connection stays open until the client gets a response. If connections are frequently created and then quickly discarded, it can waste server memory. This is where PgBouncer comes in—it helps manage and pool these connections more efficiently.

PgBouncer Description

PgBouncer is a lightweight, open-source tool that helps manage PostgreSQL connections. It works by pooling connections for each user-database pair on the server, using both TCP and Unix domain sockets.

When a client wants to connect to PostgreSQL, it first goes through PgBouncer. PgBouncer then provides a connection from its pool and returns it after the client disconnects. The key benefit is that PgBouncer reuses these connections, instead of creating new ones each time. This reduces the load on PostgreSQL and significantly improves query performance.

Configuration Parameters

pool_mode: In PgBouncer, the `pool_mode` setting determines how client connections are pooled and assigned to server connections. There are three modes:

1. **session**: Each client connection gets its own dedicated server connection for the entire duration of the session. Ideal for applications that require session-level state, such as session variables or prepared statements. Higher resource consumption since each client connection requires a separate server connection.
2. **transaction** : Workers will be assigned according to transactions. Which means it maybe considerable in multi-transactional environments. This configuration is more aggressive than session mode. This mode is a good balance between safety and resource efficiency. It's suitable for most applications that don't require session state across multiple transactions. More efficient than session mode, as server connections are reused more frequently.
3. **statement** :A server connection is assigned to a client connection for the duration of a single SQL statement. After the statement is executed, the server connection is immediately returned to the pool.This is the most efficient mode, as it maximizes server connection reuse. It's suitable for applications that execute independent SQL statements without any need for transaction or session state.

default_pool_size: This setting in PgBouncer controls the maximum number of server connections that can be allocated from the pool for each database or user. This setting effectively limits the number of active server connections that can be maintained by PgBouncer for a particular database or user. If a database or user reaches this limit, additional client connections will have to wait until a server connection becomes available. Setting `default_pool_size` too high can exhaust server resources like CPU and memory.

reserve_pool_size: This is a backup option for important business operations. If a client tries to connect but the connection pool is full, the client will wait for a certain number of seconds set by `reserve_pool_timeout`. After that, it can use extra connections provided by this configuration.

reserve_pool_timeout : If a client doesn't get a connection within this many seconds, it will use extra connections from the reserve pool. Setting it to 0 turns this feature off.

server_lifetime : The pooler will close any unused server connection that has been open longer than this time. If set to 0, the connection will be used just once and then closed.

server_idle_timeout : This setting in PgBouncer controls how long a server connection can remain idle before it's closed.

server_connect_timeout : This setting defines the maximum time PgBouncer will wait for a connection to the PostgreSQL server to be established. If PgBouncer cannot connect to the server within the specified time, the connection attempt will fail, and an error will be

returned to the client.

server_login_retry : If a login fails due to a connection or authentication issue, the pooler will wait this long before trying to connect again.

query_timeout: This setting in PgBouncer controls how long a query can run before it gets automatically canceled. It helps prevent long-running queries from using up too many resources or causing delays.

query_wait_timeout : This setting in PgBouncer controls how long a client can wait in the queue for a server connection before the waiting query is canceled. It also helps when the server is down or database rejects connections for any reason. If this is disabled, clients will be queued indefinitely.

client_idle_timeout: This setting in PgBouncer controls how long an idle client connection can remain open before it's closed. And only used for network problems.

client_login_timeout: This setting in PgBouncer controls how long a client has to successfully log in before the connection attempt is canceled. If `client_login_timeout` is set to `60` seconds, a client must complete the login process within 60 seconds, or PgBouncer will close the connection.

PGbouncer Installation

1. Pgbouncer packages installation

```
sudo yum install pgbouncer
```

2. Folder permissions for the PostgreSQL user

```
chown postgres:postgres /etc/pgbouncer/ -R
```

3. Create userlist.txt file

Generate an MD5 password for the PostgreSQL user and add it to the `userlist.txt` file.

```
postgres=# SELECT 'md5' || md5('test' || 'postgres');
              ?column?
-------------------------------------
 md5633bc3c3d823be2a52d3dff94031e2c2
(1 row)
```

```
touch /etc/pgbouncer/userlist.txt
"postgres":"md5633bc3c3d823be2a52d3dff94031e2c2"
```

4. Permissions for other directories

```
chmod 777 -R /var/log/pgbouncer/
sudo chown pgbouncer:pgbouncer /var/run/pgbouncer/
sudo chmod 777 -R /var/run/pgbouncer/
```

5. Create the PgBouncer service file

```
[Unit]
Description=A lightweight connection pooler for PostgreSQL
Documentation=man:pgbouncer(1)
After=syslog.target network.target
[Service]
RemainAfterExit=yes
User=postgres
Group=postgres
# Path to the init file
Environment=BOUNCERCONF=/etc/pgbouncer/pgbouncer.ini
ExecStart=/usr/bin/pgbouncer -q ${BOUNCERCONF}
```

```
ExecReload=/usr/bin/pgbouncer -R -q ${BOUNCERCONF}
# Give a reasonable amount of time for the server to start up/shut down
TimeoutSec=300
[Install]
WantedBy=multi-user.target

4. Open PgBouncer config/init file and paste the config below
[databases]
* = port=5432 auth_user=postgres
[pgbouncer]
logfile = pgbouncer.log
pidfile = pgbouncer.pid
listen_addr = 0.0.0.0
listen_port = 6432
auth_type = hba
auth_hba_file = /var/lib/pgsql/16/data/pg_hba.conf
admin_users = postgres
stats_users = postgres
pool_mode = session
ignore_startup_parameters = extra_float_digits
max_client_conn = 200
default_pool_size = 50
reserve_pool_size = 25
reserve_pool_timeout = 3
server_lifetime = 300
server_idle_timeout = 120
server_connect_timeout = 5
server_login_retry = 1
query_timeout = 60
```

6. Add this entry in pgbouncer.ini

```
postgres = host=localhost port=5432 dbname=postgres
```

7. Update the authentication method to MD5 in `pg_hba.conf`

```
 psql -U postgres -d postgres -h localhost -p 5432
Password for user postgres:
psql (16.4)
Type "help" for help.

postgres=# show password_encryption;
 password_encryption
---------------------
 md5
(1 row)
```

8. Start the pgbouncer service

```
[root@a125cb415207 data]# systemctl start pgbouncer
[root@a125cb415207 data]# systemctl status pgbouncer
● pgbouncer.service - A lightweight connection pooler for PostgreSQL
     Loaded: loaded (/usr/lib/systemd/system/pgbouncer.service; disabled; preset: disabled)
     Active: active (running) since Mon 2024-08-12 05:06:33 UTC; 3s ago
       Docs: man:pgbouncer(1)
   Main PID: 802 (pgbouncer)
      Tasks: 2 (limit: 75664)
     Memory: 1.3M
```

```
     CGroup:
/docker/a125cb415207ba986d1bcde058c3c274c8e8066eb689665877113e6286ba98f2/system.slice/pgbouncer.service
             └─802 /usr/bin/pgbouncer -q /etc/pgbouncer/pgbouncer.ini

Aug 12 05:06:33 a125cb415207 systemd[1]: Started A lightweight connection pooler for PostgreSQL.
[root@a125cb415207 data]# vi /usr/lib/systemd/system/pgbouncer.service
[root@a125cb415207 data]# vi /etc/pgbouncer/pgbouncer.ini
```

9. Connect the pgbouncer

```
 /usr/pgsql-16/bin/psql -h localhost -p 6432 -U postgres -d postgres
Password for user postgres:
psql (16.4)
Type "help" for help.

postgres=#
```

10. Conduct a basic benchmark test

After setting up, conduct a simple test run to observe the improvement in the TPS of the database server with PgBouncer. I've observed approximately a 24% improvement in TPS with PgBouncer.

For the TPS benchmark, I used `pgbench` initialized with a scale factor of 1. The benchmark was run with 5 threads and 10 clients over a duration of 60 seconds.

TPS without Pgbouncer

```
./pgbench -c 10 -j 5 -T 60 -h localhost -p 5432 -U postgres postgres
Password:
pgbench (16.4)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 5
maximum number of tries: 1
duration: 60 s
number of transactions actually processed: 73272
number of failed transactions: 0 (0.000%)
latency average = 8.188 ms
initial connection time = 15.287 ms
tps = 1221.329787 (without initial connection time)
```

TPS with Pgbouncer

```
./pgbench -c 10 -j 5 -T 60 -h localhost -p 6432 -U postgres postgres
Password:
pgbench (16.4)
starting vacuum...end.
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 1
query mode: simple
number of clients: 10
number of threads: 5
maximum number of tries: 1
duration: 60 s
number of transactions actually processed: 55843
number of failed transactions: 0 (0.000%)
latency average = 10.747 ms
```

```
initial connection time = 2.129 ms
tps = 930.493410 (without initial connection time)
```

To make it easier to understand, you can see the benchmark test results in the graphs below.