Now we will see what the dashboard looks like once our master is no longer available. Refer to the following screenshot:

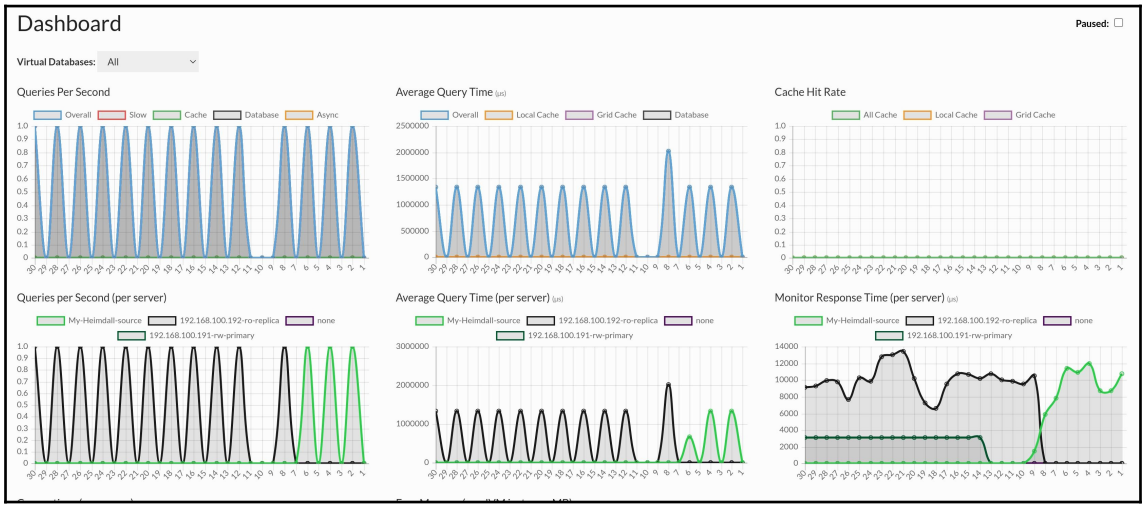

Figure 8.12 Heimdall dashboard without master

As you can see in the preceding screenshot, in the **Monitor Response Time** chart, there are a number of important issues that need to be highlighted:

1. The **192.168.100.191-rw-primary** (master) line in green has gone as a consequence of our artificial crash implemented with `Vagrant suspend`.
2. As a result of the event in point 1, Heimdall acts quickly, changing the role of the **192.168.100.192-ro-replica** line in black and placing at the front the **My-Heimdall-source** green line until a new slave or replica is added.
3. Let's not lose sight of the **Queries per Second** chart. This is perhaps the most important event because it means that end users have never lost their connection to the database.

Following these events, let's now continue monitoring and we will see that everything has returned to normal, but in a new context:

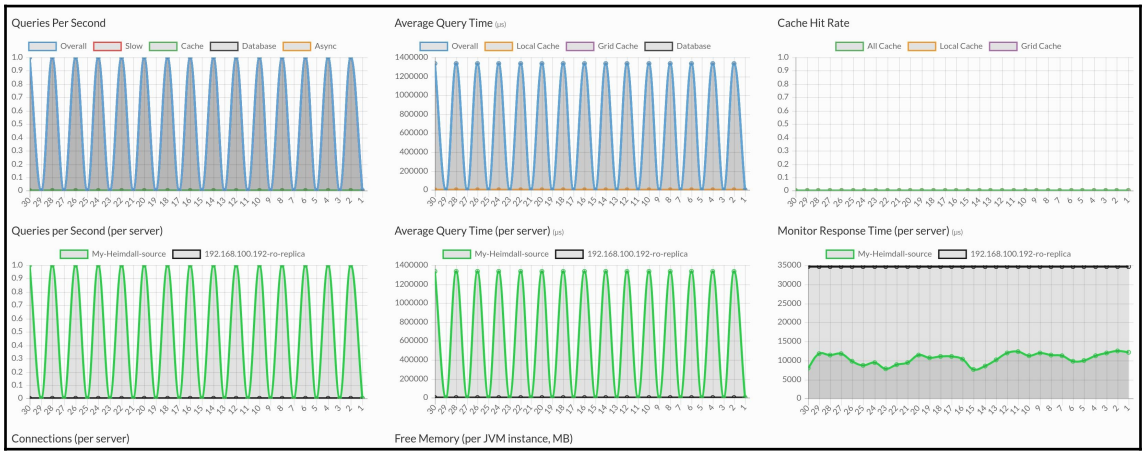

Figure 8.13 Heimdall dashboard post crash

As you can see in the preceding screenshot, in this new context, we can say that **192.168.100.192-ro-replica** is now the one that will receive all the connections and keep the existing ones. This is what is known as high availability, and for end users such as our psql client or web apps, this will be transparent. In the meantime, our technical staff will have taken care of this, rectifying issues on the master server or replacing it with a new one.

Everything that we have seen, and that Heimdall has solved fantastically behind the scenes, is amazing from the perspective of availability, and even more so if we have users concurrently accessing our services.

Summary

In this chapter, we learned about streaming replication with PostgreSQL and saw how straightforward it is to configure it. We stated that it is the base solution for numerous high availability architectures, and then we presented Heimdall, which, based on streaming replication and thanks to its load balancing and high availability features, demonstrated how, in the event of a failure in the master server, it can continue to provide a service without the end customer being aware of this. In the next chapter, we'll explore New Relic and how we can use it to create team dashboards for various applications.

9
High-Performance Team Dashboards Using PostgreSQL and New Relic

In this chapter, you will learn how to install and activate PostgreSQL integration and will gain an understanding of the data collected by the New Relic infrastructure. New Relic PostgreSQL on-host integration receives and sends inventory metrics from our PostgreSQL database to the New Relic platform, where we can aggregate and visualize key performance metrics. Data about the level of instances, databases, and clusters can help us to more easily monitor our PostgreSQL database. You will learn how to use New Relic to monitor a PostgreSQL RDS and you will be able to access, visualize, and troubleshoot entire PostgreSQL databases for yourself – and your software team.

With the help of the project demonstrated in this chapter, we will create a New Relic monitoring dashboard for the PostgreSQL 12 RDS from Amazon Web Services from the previous chapters.

The following topics will be covered in this chapter:

- Signing up for and installing New Relic
- Defining PostgreSQL RDS role permissions
- Configuring New Relic for PostgreSQL
- Adding new metric data for PostgreSQL
- Infrastructure inventory data collection

Technical requirements

This chapter will take developers around 6-8 hours of work to create a New Relic monitoring dashboard for the PostgreSQL 12 RDS. For this chapter, you only need a PostgreSQL RDS and a stable internet connection because New Relic is an online dashboard from the web.

Signing up for and installing New Relic

New Relic supplies a high-performance team dashboard for various systems, such as the following:

- Backend, frontend, and mobile applications
- Cloud and platform technologies
- Host operating systems
- Log ingestion
- Infrastructure
- Open source monitoring systems

We can plug New Relic into different cloud platforms such as **Amazon Web Services** (**AWS**), Microsoft Azure, and Google Cloud Platform. Hence, obviously, New Relic installation for a PostgreSQL RDS from AWS is a typical application of New Relic.

You can access New Relic through your browser, and we will learn how to make a New Relic dashboard by taking the following steps:

1. In a browser, open the New Relic website: `https://infrastructure.newrelic.com/`.

2. Next, you need to sign in. If you do not have an account, you can register with New Relic by clicking on **Create a free account**:

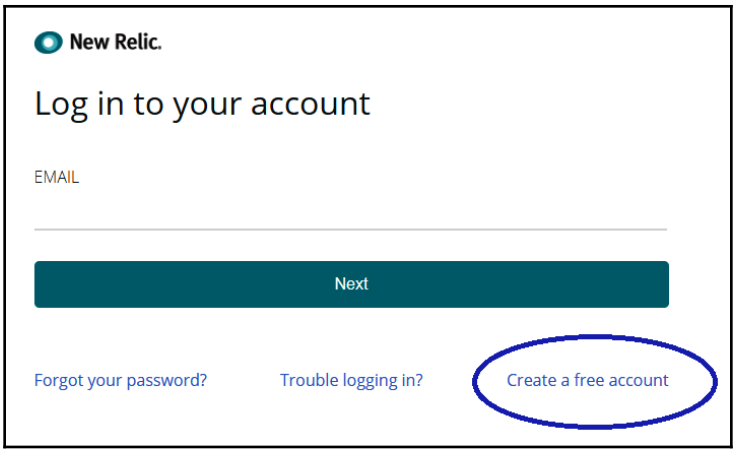

Figure 9-1. New Relic login

3. Enter the required information into the New Relic registration form as shown in *Figure 9-2* and *Figure 9-3*:

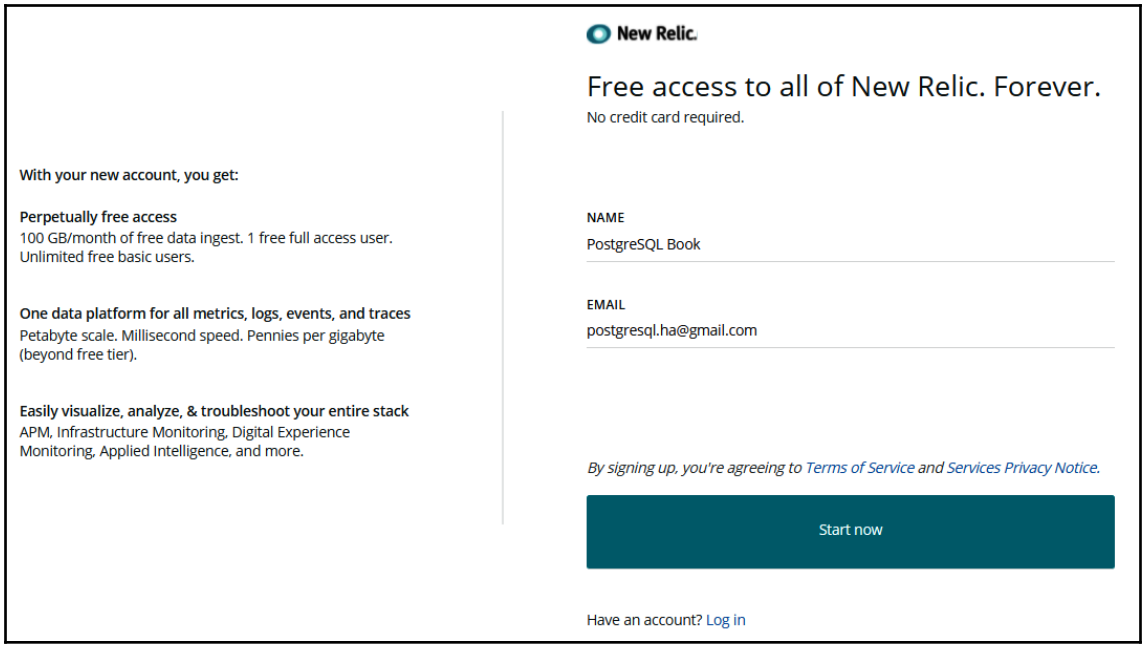

Figure 9-2. New Relic user registration

Scroll down to the end of the web page to check the **Terms of Service** and **Service Policy Notice** before you click the **Start now** button:

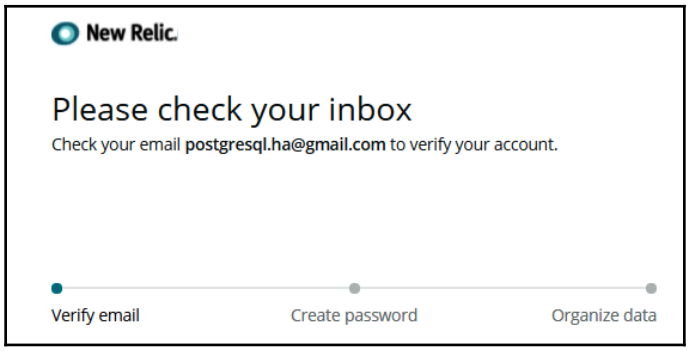

Figure 9-3. New Relic user registration (cont.)

4. You should receive a notification about verification as shown here:

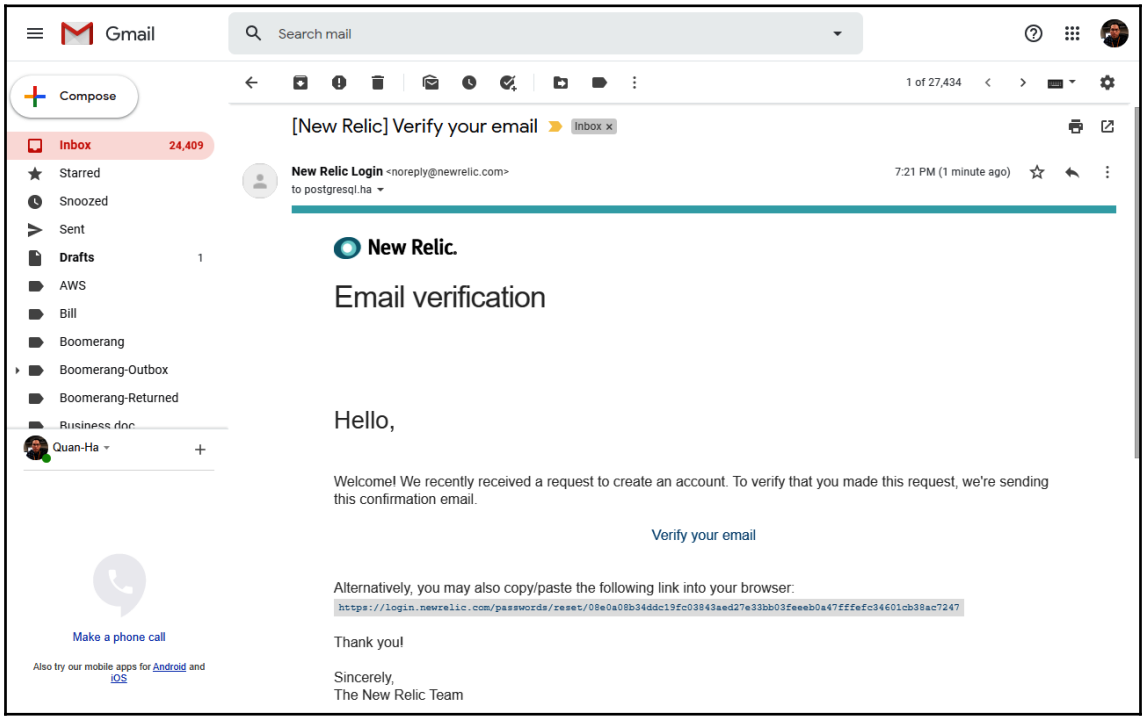

Figure 9-4. New Relic registration email

5. When you click on the link inside the New Relic email, it will pop up the **Create your password** screen for you, as shown in Figure 9-5:

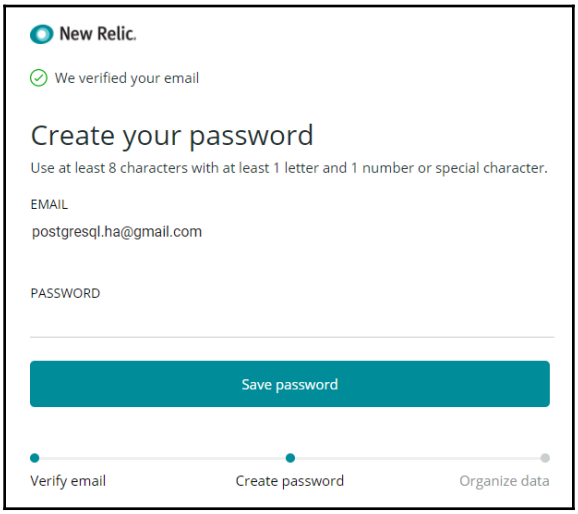

Figure 9-5. Setting a New Relic password

For example, I set up my password as `BookDemo@9`.

6. Now with your email ID and password, you need to sign in here:

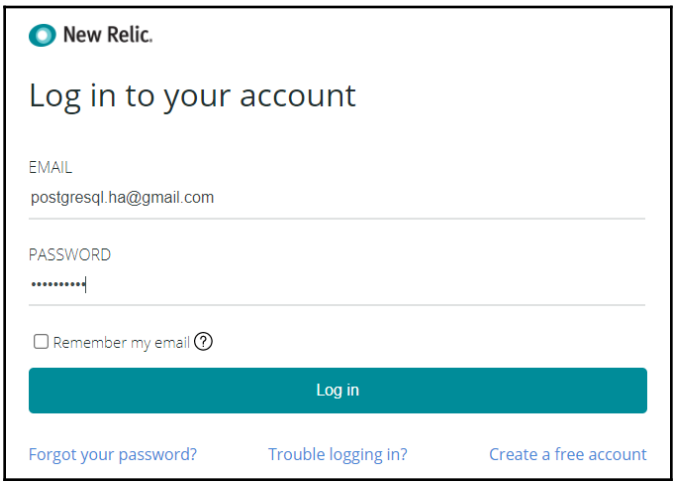

Figure 9-6. Signing in to New Relic as a newly registered user

7. After you click on the **Log in** button, the New Relic APM page will open. Select the Data Region as either **United States** or **Europe**:

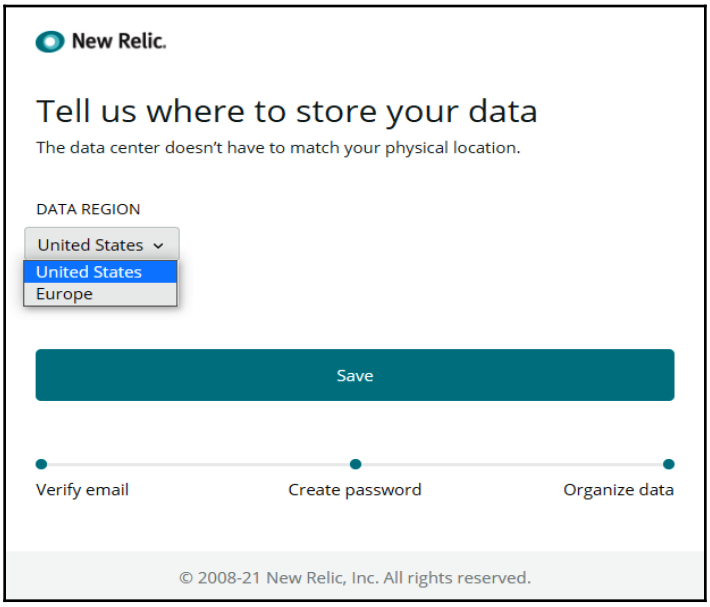

Figure 9-7. New Relic APM page

8. Click on the **Save** button:

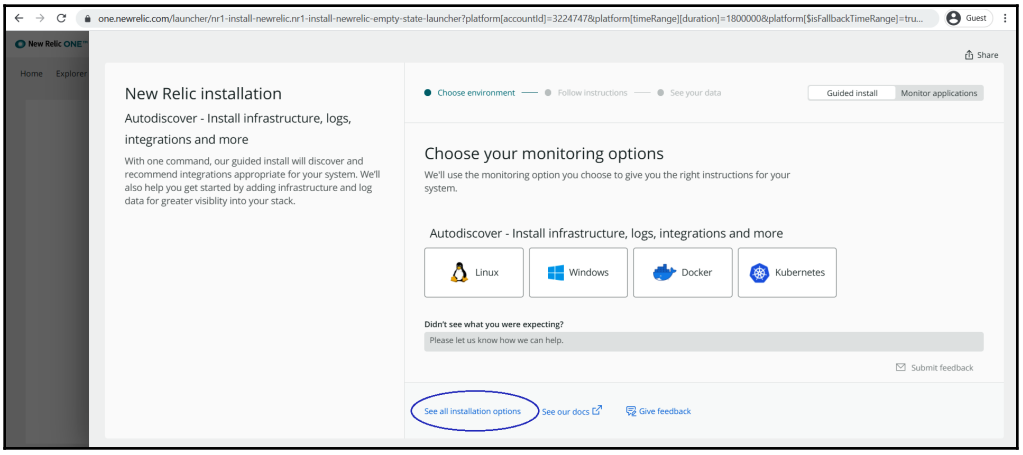

Figure 9-8. New Relic installation page

9. Click on the **See all installation options** link in the previous figure and select the infrastructure type as Amazon Web Services as seen here:

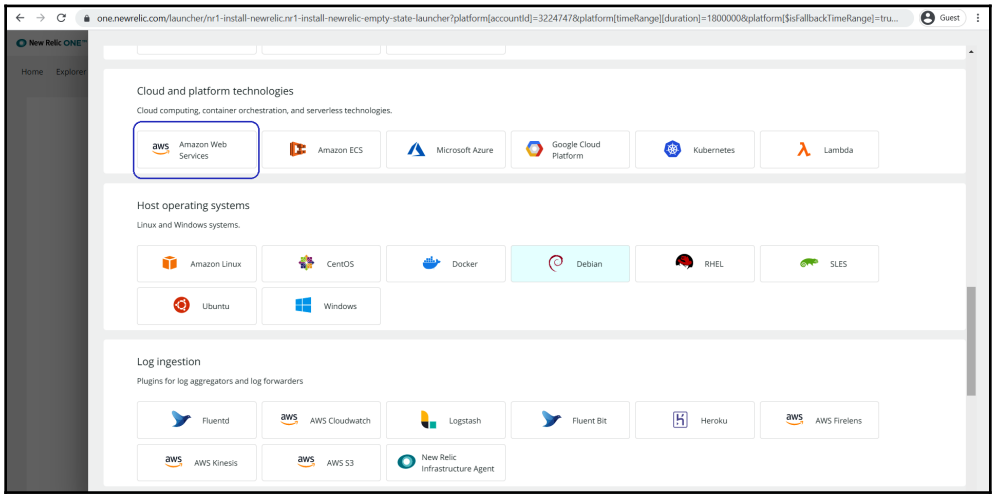

Figure 9-9. Different New Relic infrastructure types

10. Select the type of AWS services that you would like to set up. Here, I am going to select **RDS** for the PostgreSQL database:

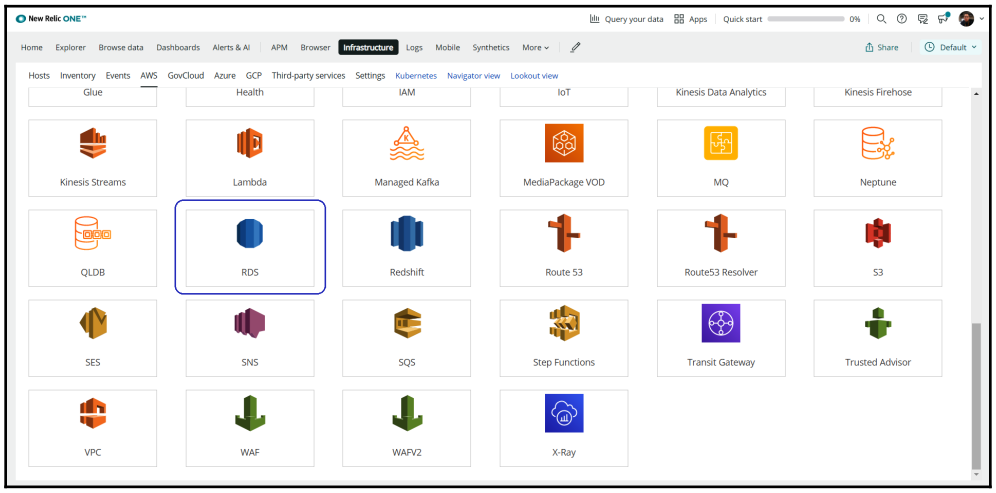

Figure 9-10. RDS inside New Relic infrastructure

Not only can New Relic offer a monitoring dashboard for PostgreSQL RDS but it can also visualize monitoring for many other AWS services, such as IoT, Kinesis, Lambda, Redshift, and VPC. By simply signing up for a new user account, you can start to monitor different AWS services with a few easy steps.

11. There are 2 integration modes for AWS, please press the **Use API Polling** button.

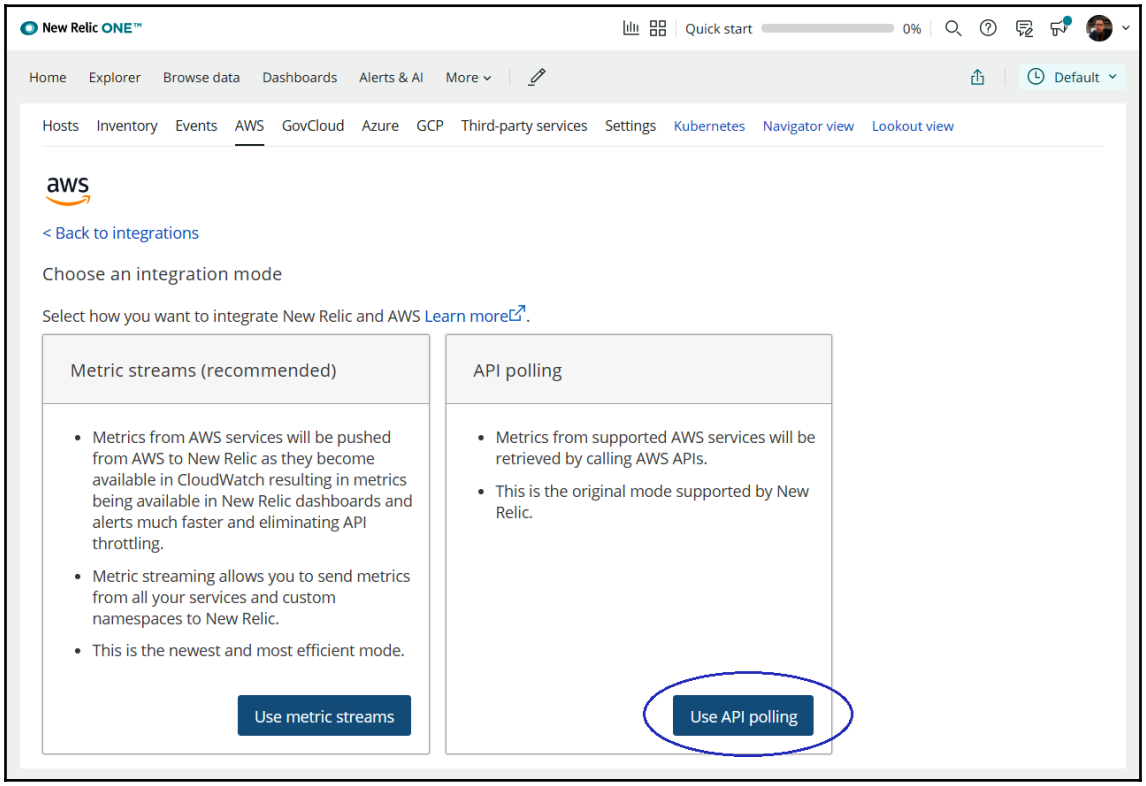

Fig 9.11 Integration modes for RDS

In order to connect your New Relic account to the PostgreSQL RDS, you have to grant permissions from your AWS account so that New Relic will be allowed to collect information about your RDS.

We will discuss the required role permissions that we need to set up for New Relic to access your RDS in the next section.

Defining PostgreSQL role permissions

AWS uses the **Identity and Access Management** (**IAM**) service to enable you to grant access to AWS services and resources. With IAM, you can create a new role using permissions to allow New Relic access to AWS resources. Therefore, you need to open two tabs at the same time on your browser for both AWS and New Relic, then you have to copy values from your New Relic account into the AWS IAM service.

The following are the steps to create an IAM policy:

1. Now we will link the New Relic user account to the PostgreSQL RDS. In Step 1 on the New Relic page called Trust, remember to copy the details on the page and open a new browser window/tab for AWS. You will then switch between the two browser windows – the browser window of New Relic and the browser window of AWS:

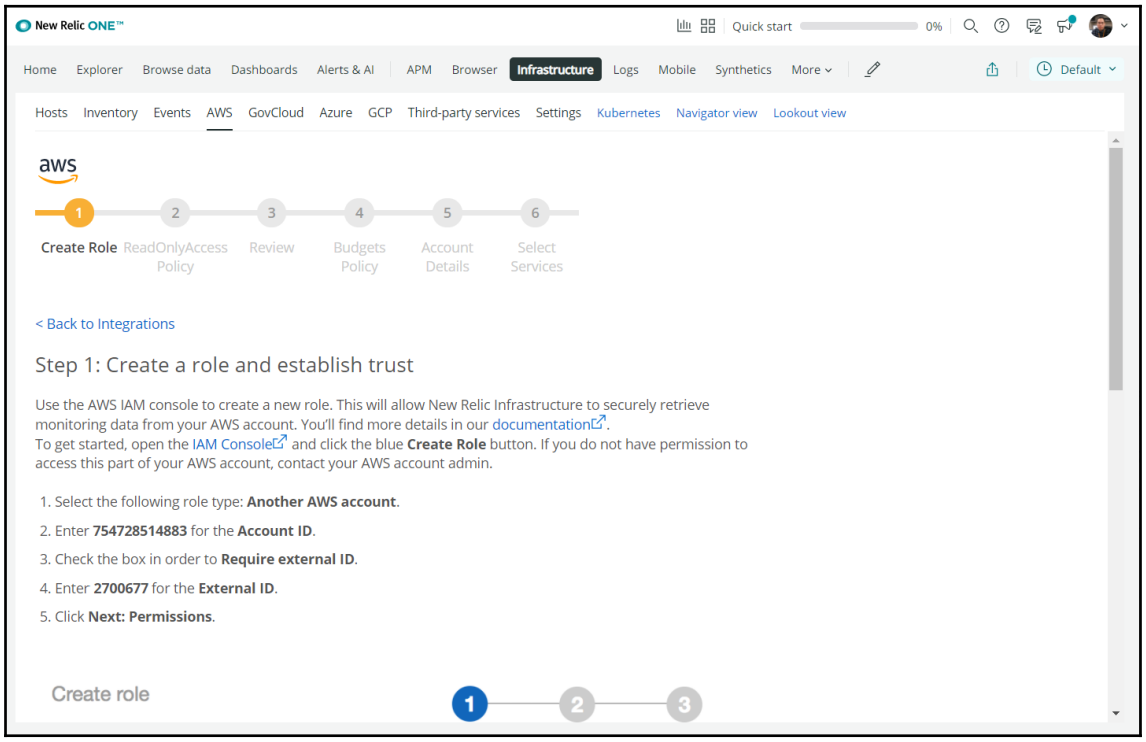

Figure 9-12. The first step – trust

2. In the second browser window, please sign in and navigate to the AWS Management Console using this link: `https://console.aws.amazon.com/iam/home#roles`. Click on **Create role** to set up an IAM role for granting permissions to New Relic to access our AWS services:

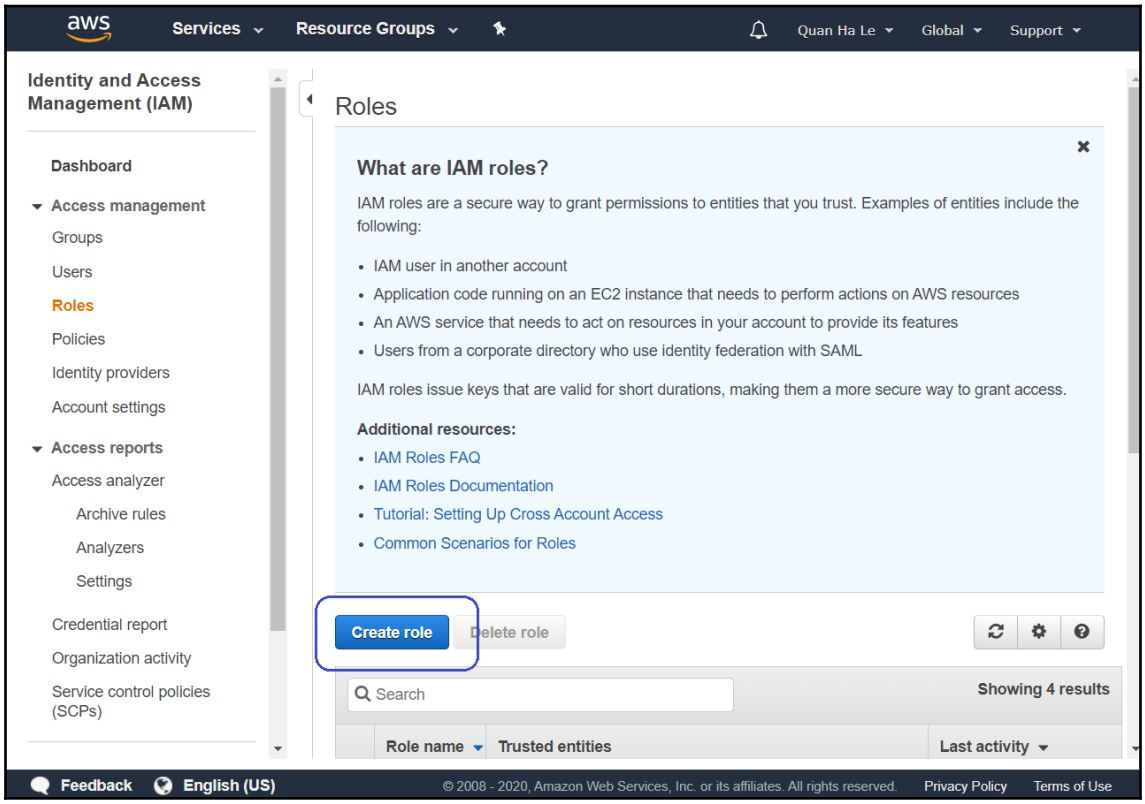

Figure 9-13. Repeat the New Relic details on the AWS IAM role page

3. Switch back to the first New Relic window and scroll down the page to see all the guidelines for step 1:

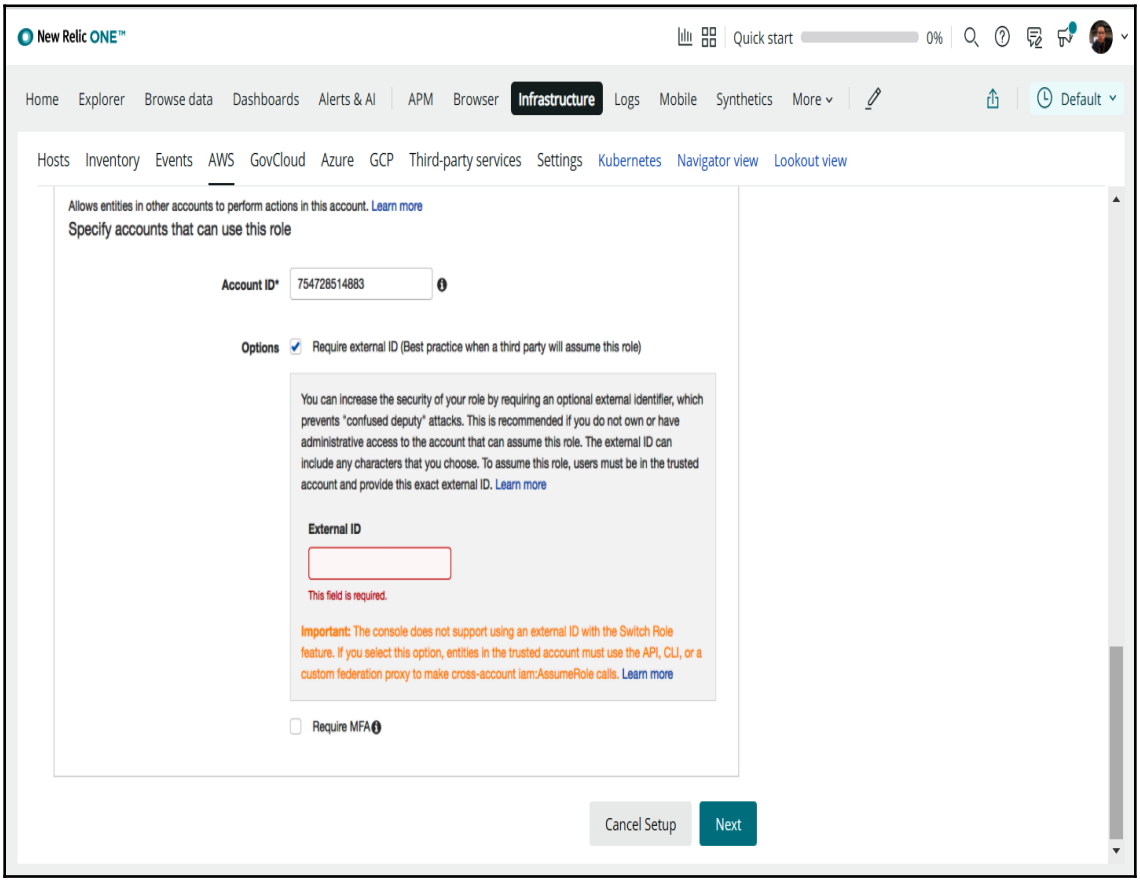

Figure 9-14. New Relic step 1 – trust (cont.)

4. Switch to the AWS browser to complete step 2. We'll now select the type of trusted entity as **Another AWS account** in the AWS browser window. Then copy the values from the first browser window of New Relic for the following values on the AWS page and then click on **Next: Permissions**:
    - **Account ID**: `754728514883`
    - **Require external ID**: Yes
    - **External ID**: `2700677`

We can see this in the following screenshot:

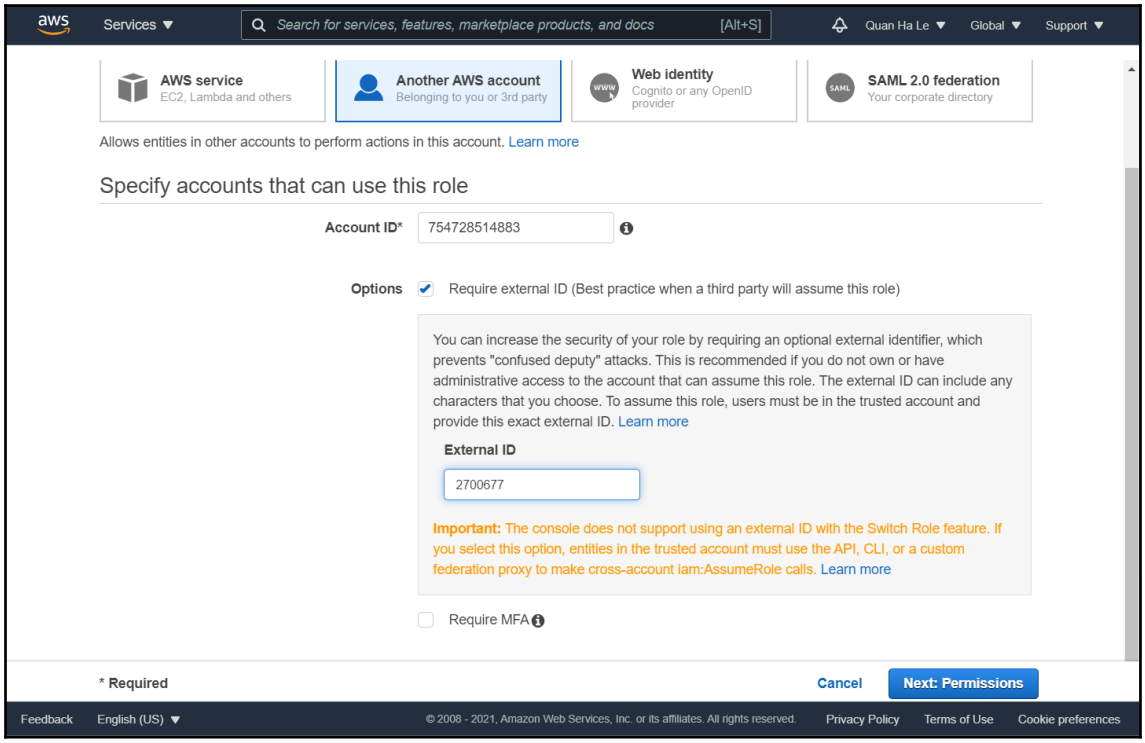

Figure 9-15. Type of trusted entity

5. In the New Relic window, click **Next** to move on to the next guideline step, **Permissions**:

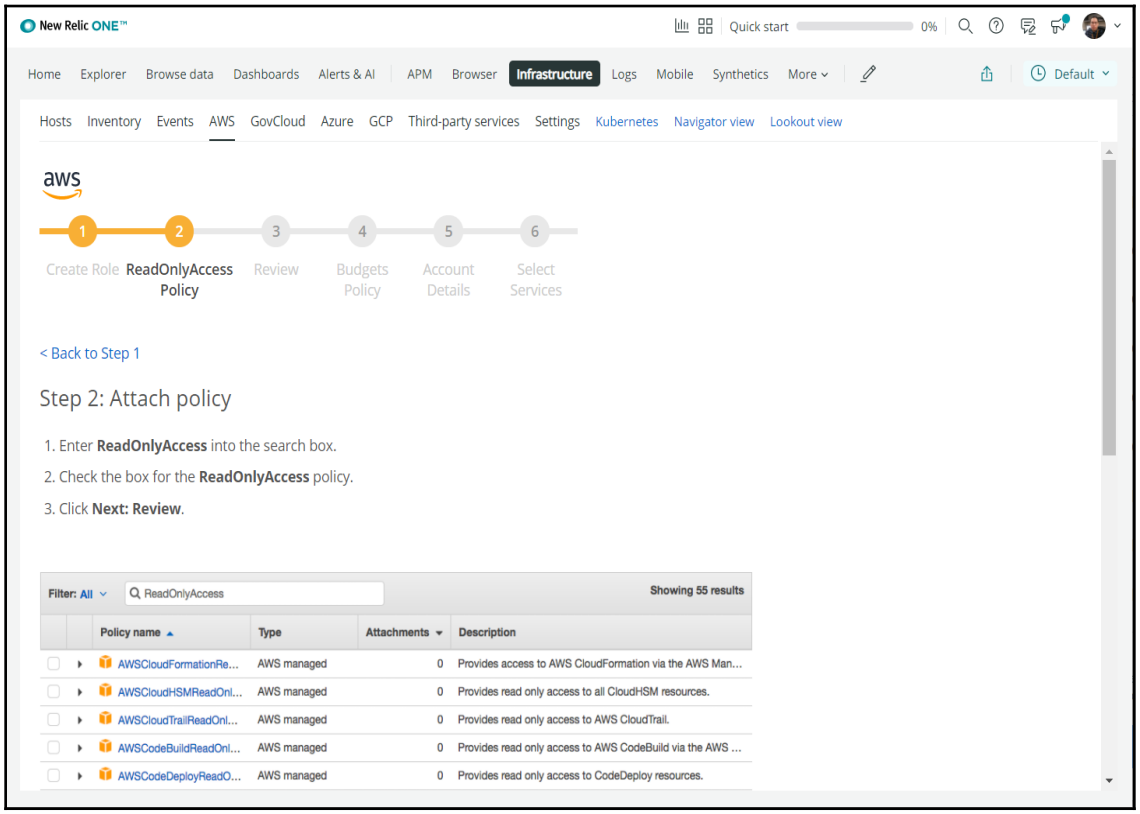

Figure 9-16. New Relic step 2 – Permissions

6. We will switch to the AWS window and will follow the New Relic guidelines from step 5 to proceed further:
    1. Enter `ReadOnlyAccess` into the search box.
    2. Scroll down until nearly the end of the list, then check the box for **ReadOnlyAccess**.
    3. Click the **Next: Tags** button:

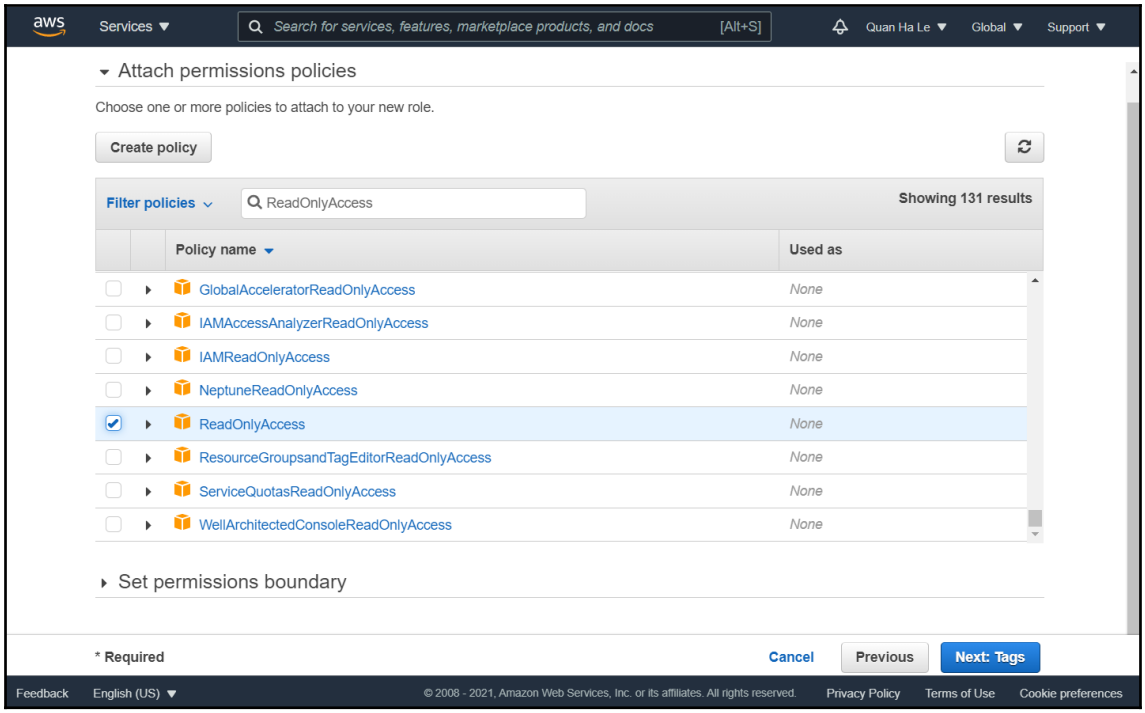

Figure 9-17. AWS permissions

7. Click the **Next: Review** button:

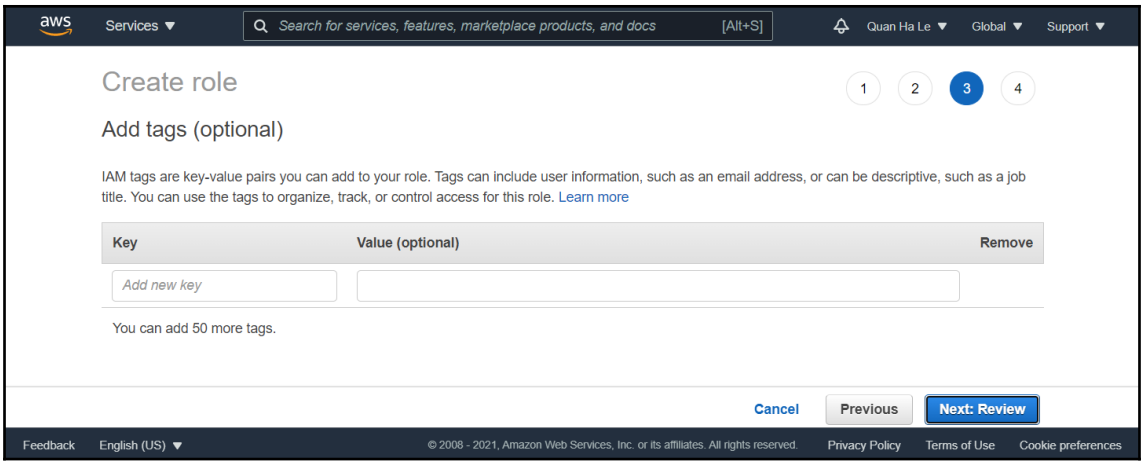

Figure 9-18. No tags are needed

8. In the New Relic window, click **Next** to move on to the next guidelines, step 3, called **Review**:

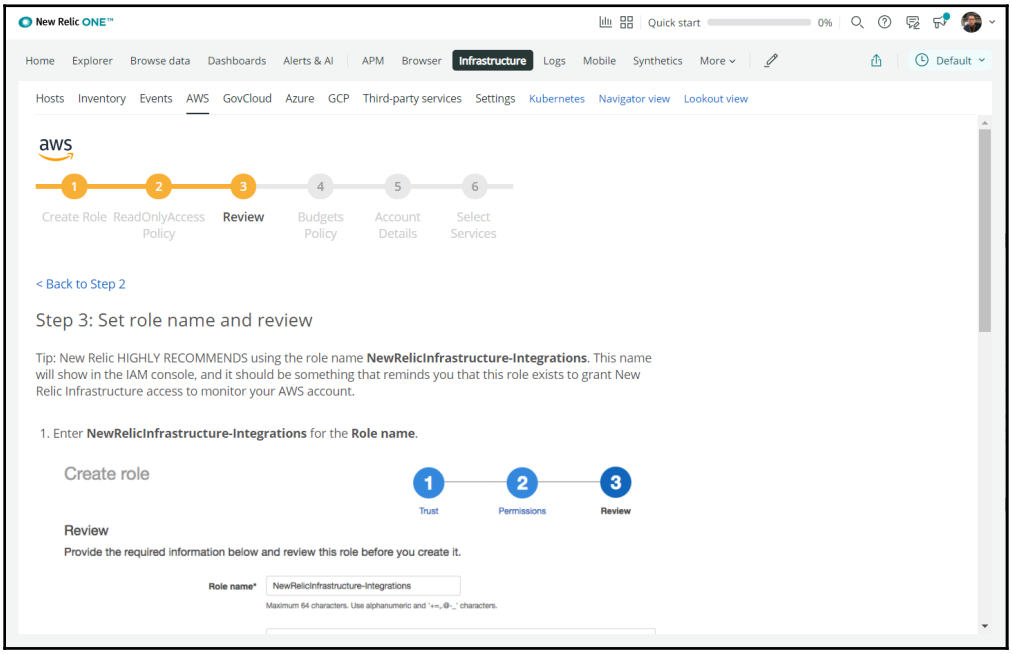

Figure 9-19. New Relic step 3 – Review

9. In the AWS window, set a name for the IAM role and click on **Create role**:

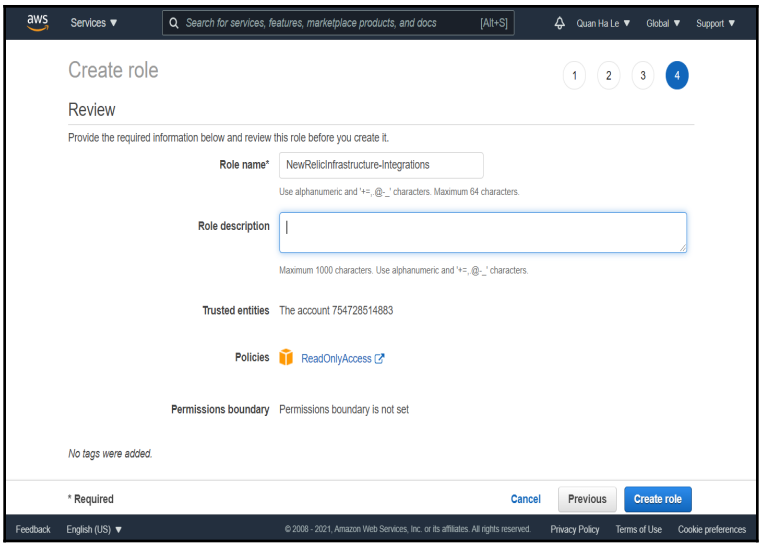

Figure 9-20. Integration of the AWS role and New Relic

10. Click on the new role name:

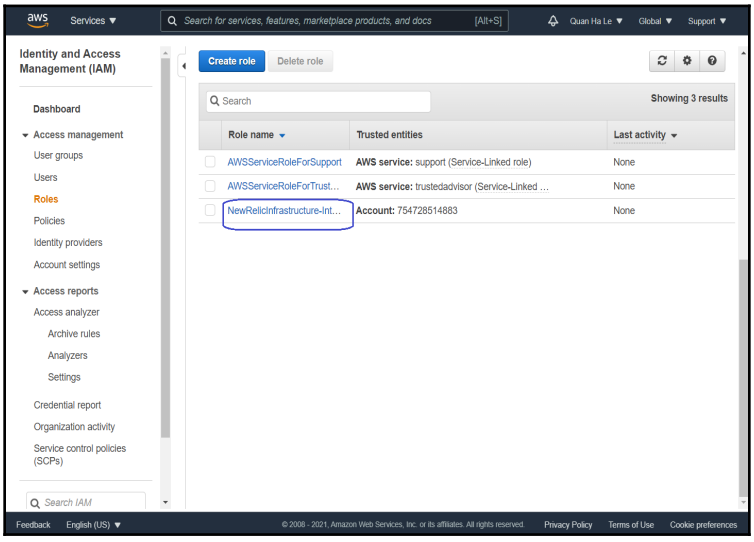

Figure 9-21. A New IAM role for New Relic has been created

11. Copy the IAM role ARN by clicking on the icon:

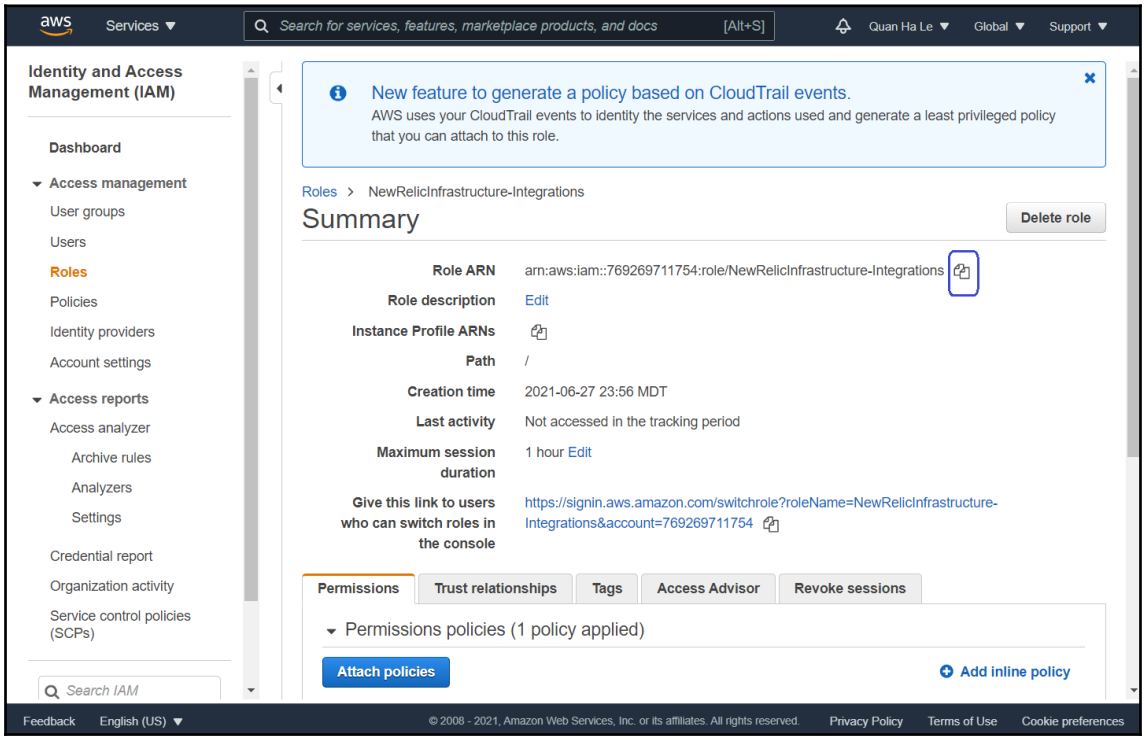

Figure 9-22. The New Relic role ARN

Keep the role ARN somewhere safe to use later.

12. Now switch to the New Relic window and click on **Next** to move on to step 4, which is **Budgets Policy**:

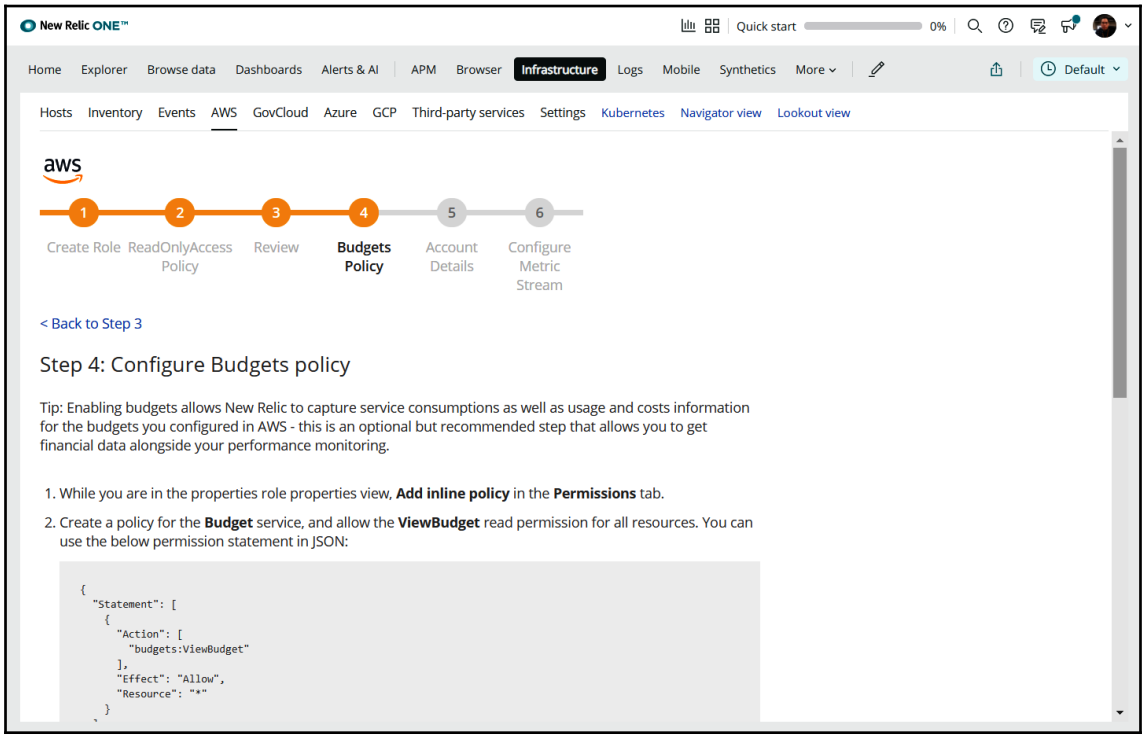

Figure 9-23. New Relic guideline for policy creation

13. In the AWS window, click on **Add inline policy** (Figure 9-22) and then click on **Choose a service**:

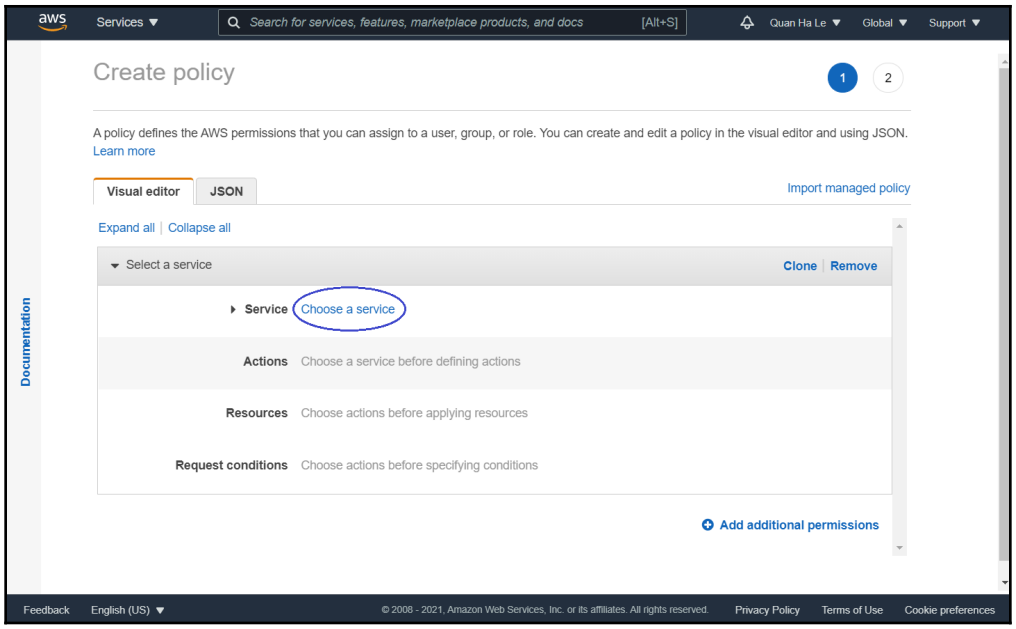

Figure 9-24. Create policy page

14. Enter `Budget` to select the **Budget** service:

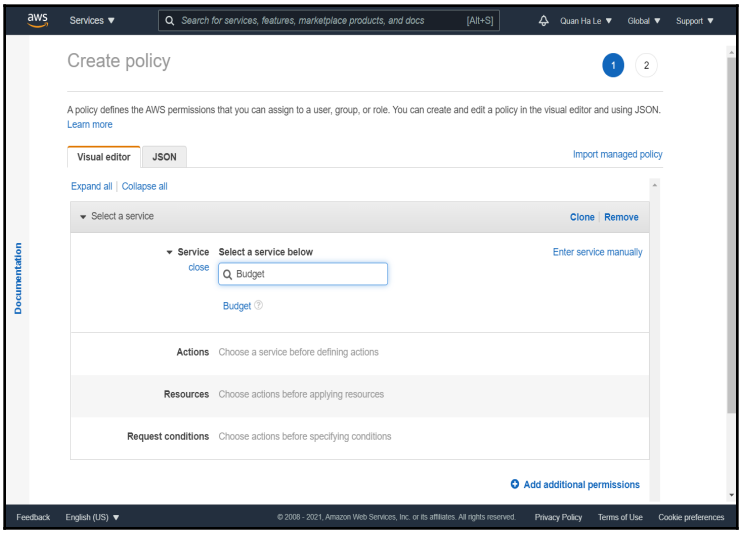

Figure 9-25. Select the Budget service

15. The **Actions** area will expand. For **Access level**, select **Read**:

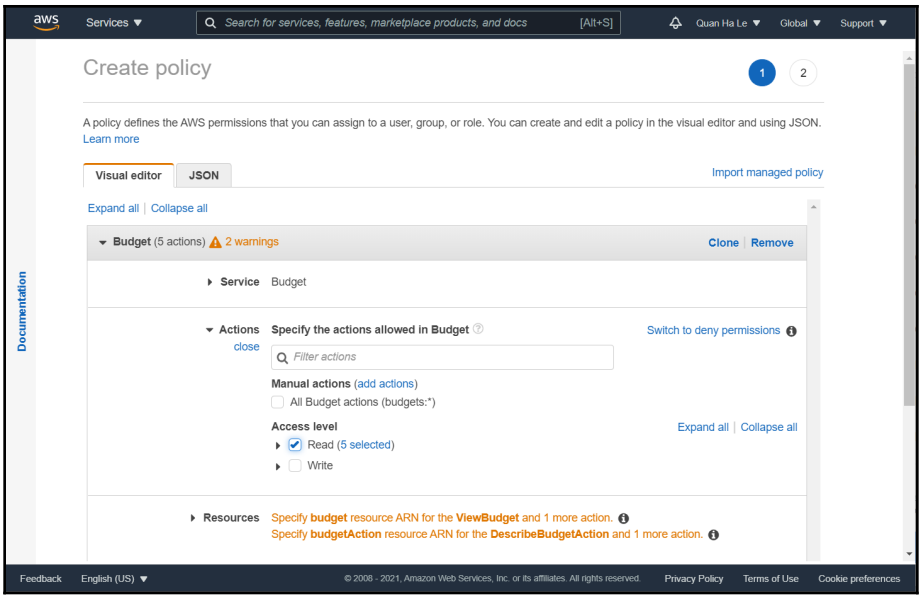

Figure 9-26. Policy access level

Then, for **Resources**, select **All resources** and click on the **Review policy** button:

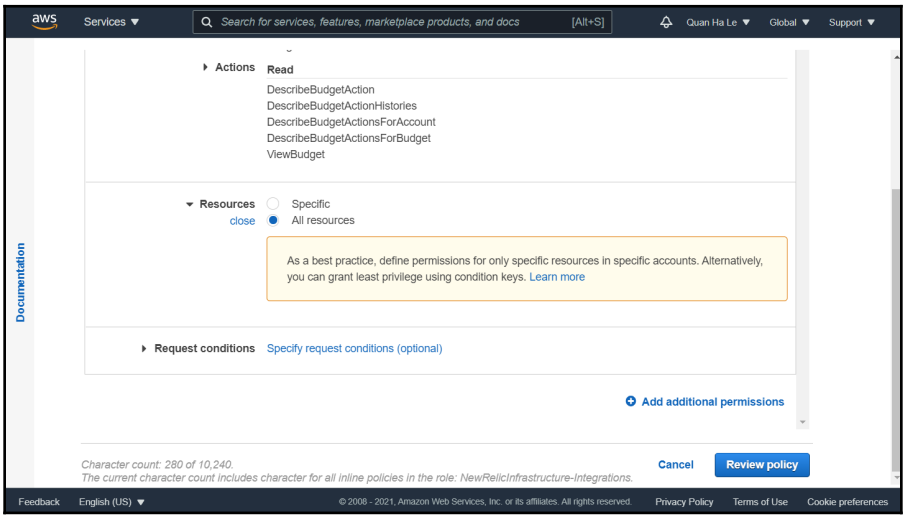

Figure 9-27. Budget resources

16. Enter the policy name in the **Name** field as **NewRelicBudget** and then click on the **Create policy** button:

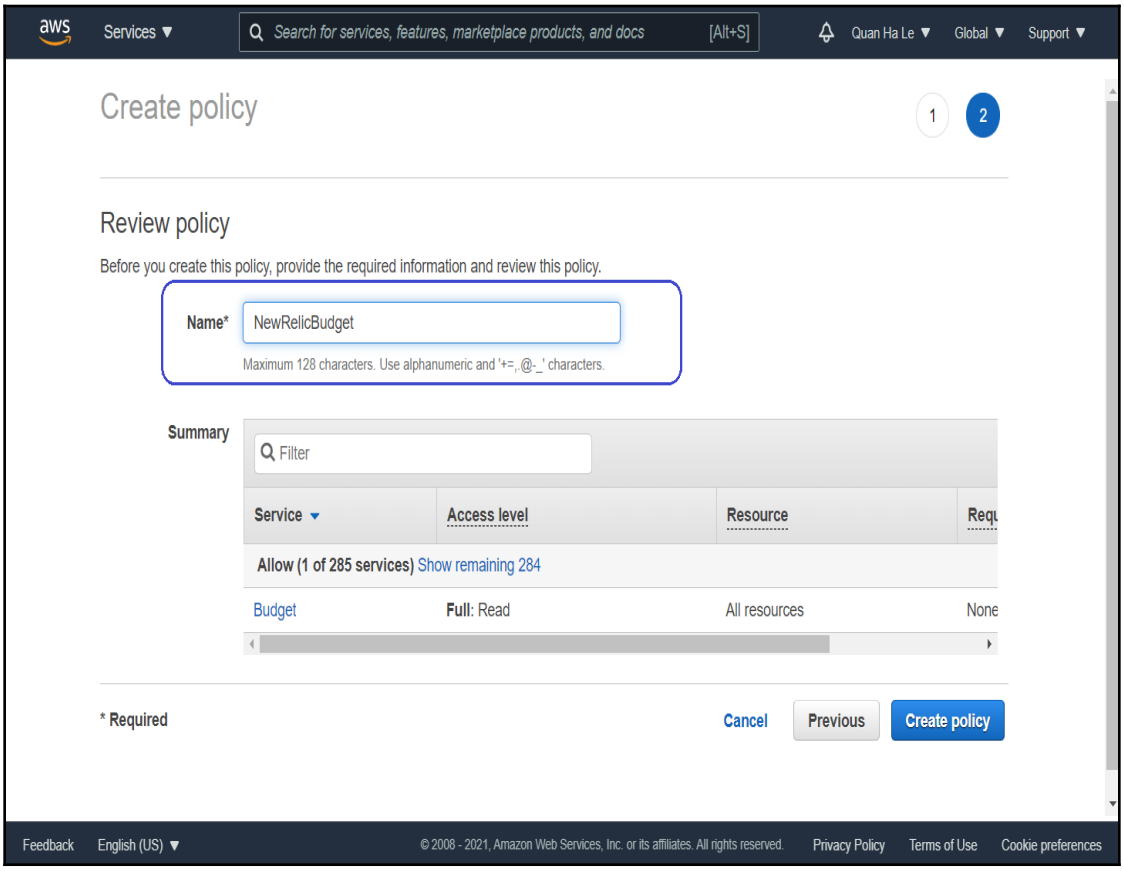

Figure 9-28. Create policy

In this section, you learned how to grant permissions for New Relic to access the AWS RDS service. The steps are straightforward using IAM roles of AWS. New Relic also provides step-by-step detailed guidance, which is very convenient for system administrators.

Now that you have prepared your AWS account, in the next section, you will learn how to link it with your New Relic account to use the created IAM **NewRelicBudget** policy.

Configuring New Relic for PostgreSQL

Now it is time to provide your AWS account details for New Relic. Once New Relic connects to your AWS RDS service using these details, because you have already set up IAM role permissions to authorize New Relic, the New Relic dashboard will be able to launch:

1. Switch to the New Relic window and click on **Next** to move on to the next step, **Account Details**, **i**n the New Relic browser:

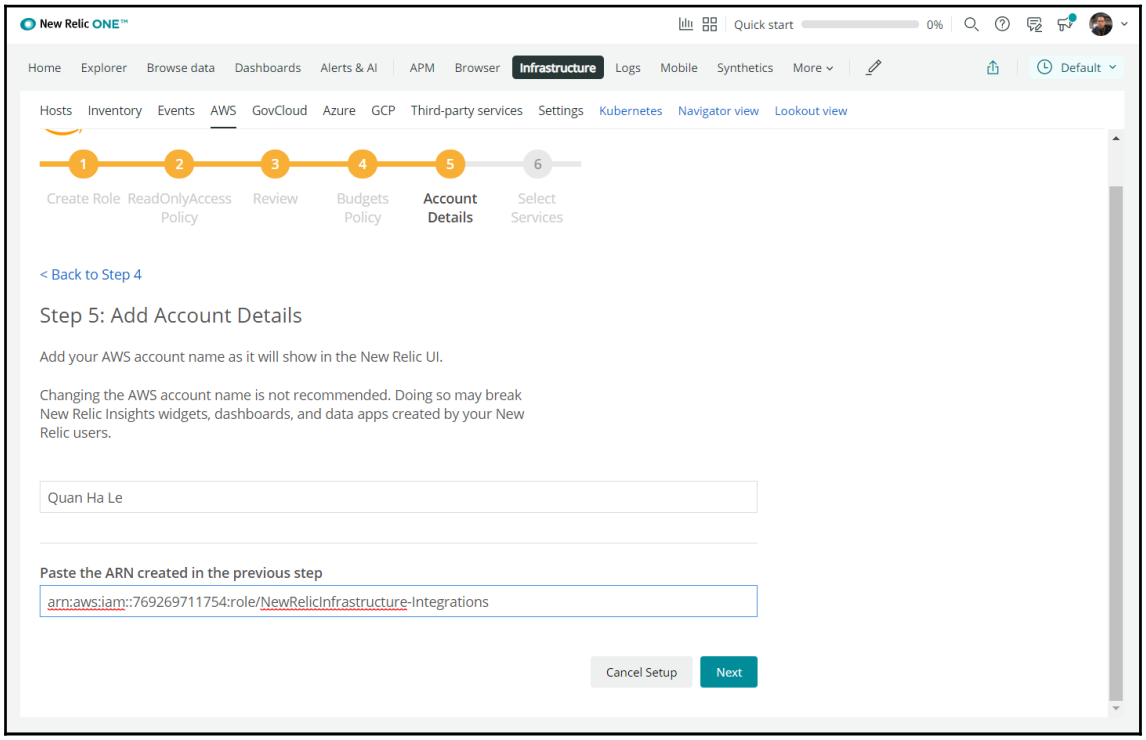

Figure 9-29. Step 5 Account Details

In Figure 9-30, we added the AWS account name in New Relic, and then for the IAM Role ARN, we have to find the correct values from the AWS window.

2. Switch to the AWS window and click on the **My Account** drop-down field to get the AWS account details for New Relic as shown in Figure 9-30:

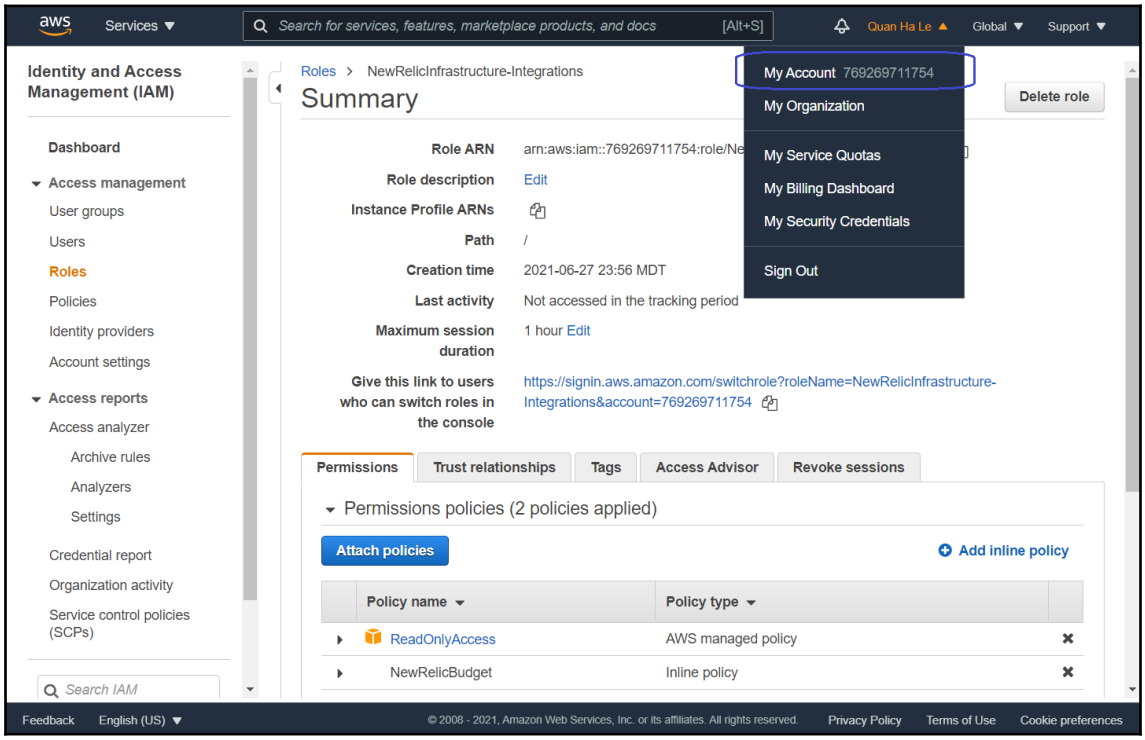

Figure 9-30. AWS account page

3. Copy the account name as shown in Figure 9-31:

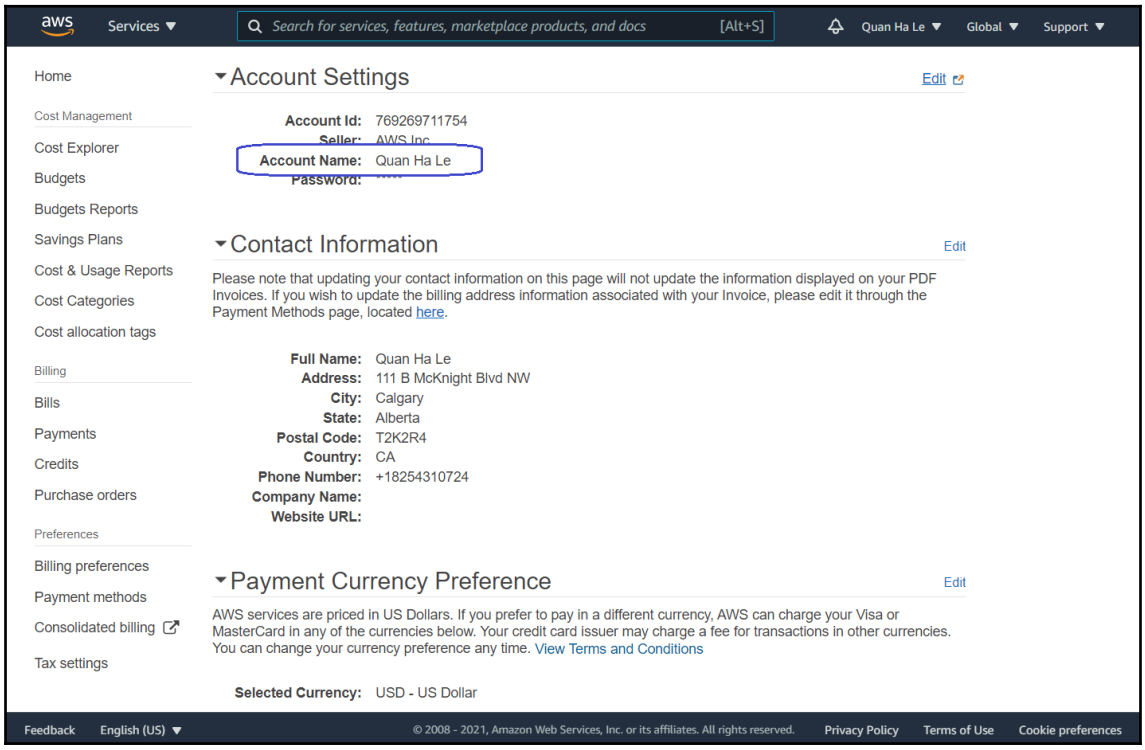

Figure 9-31. AWS My Account page

4. In the New Relic window, click **Next** to proceed to step 6, which is **Select Services**:

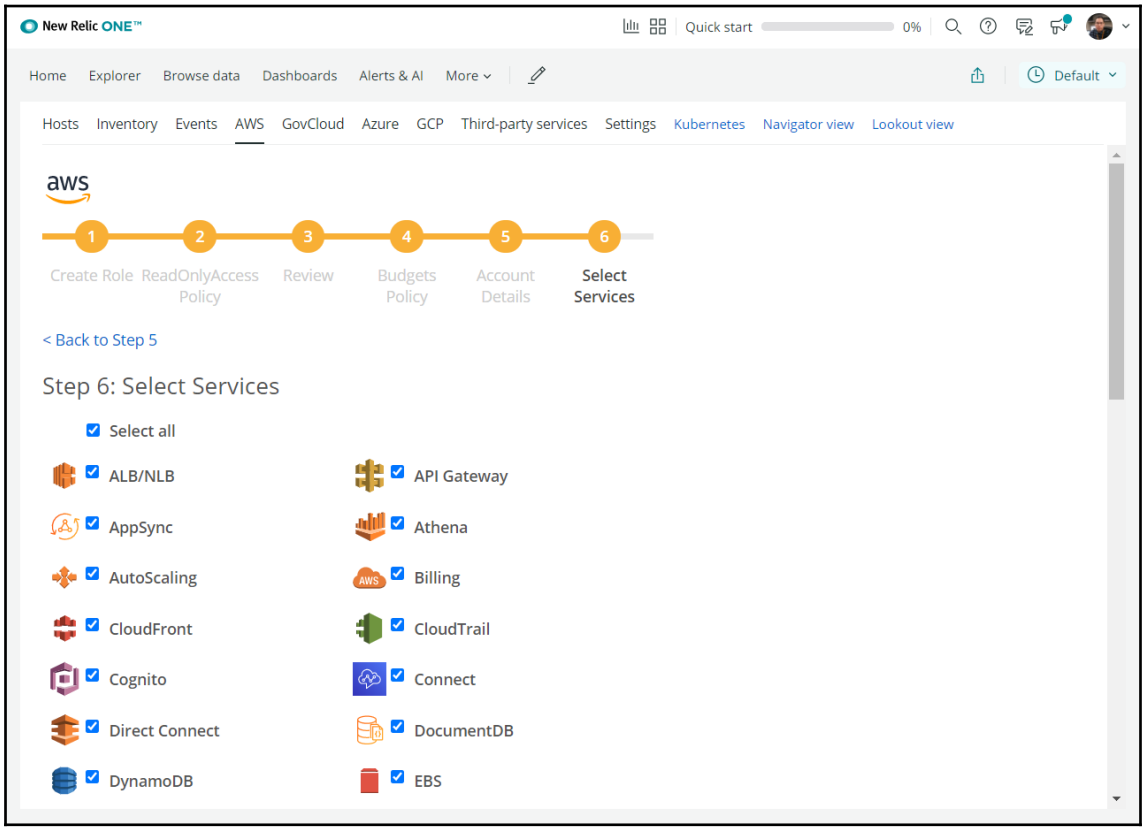

Figure 9-32. Step 6 – Select Services

5. Next, you need to de-select the **Select all** checkbox and then select the **RDS** option:

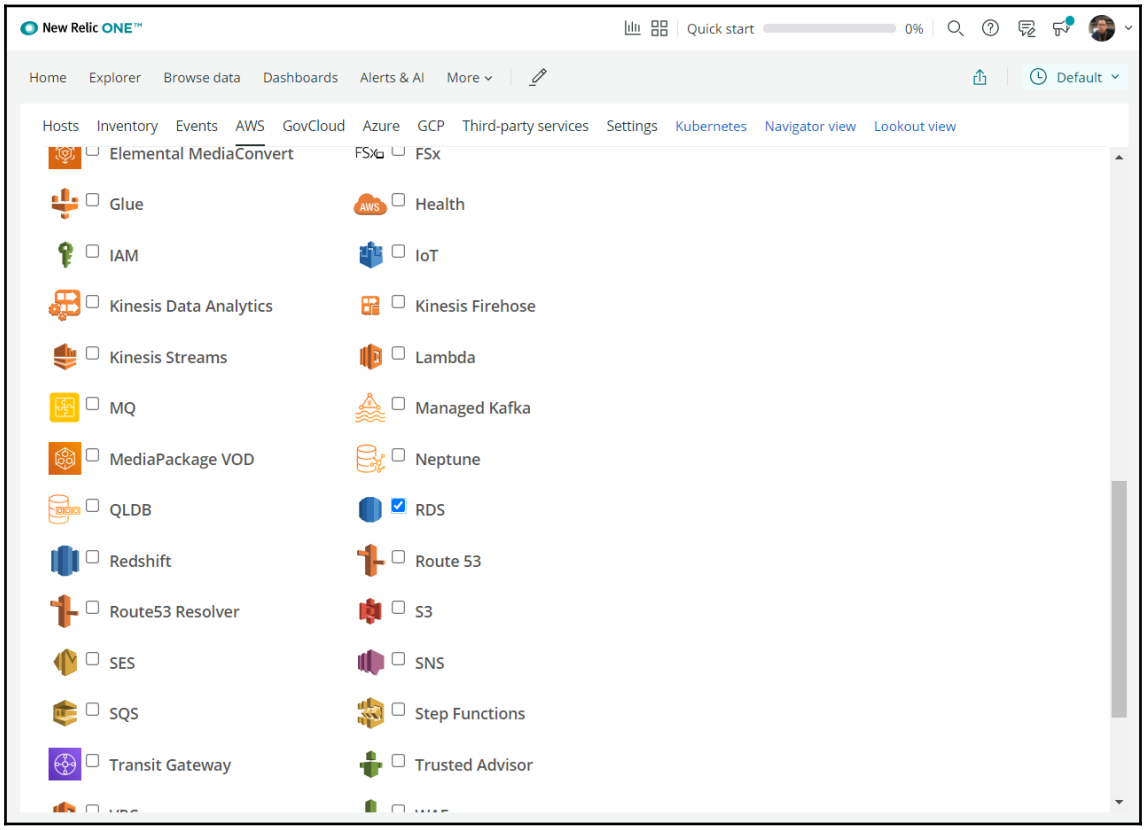

Figure 9-33. New Relic RDS option

6. Click on **Next** and then we will reach our final results:

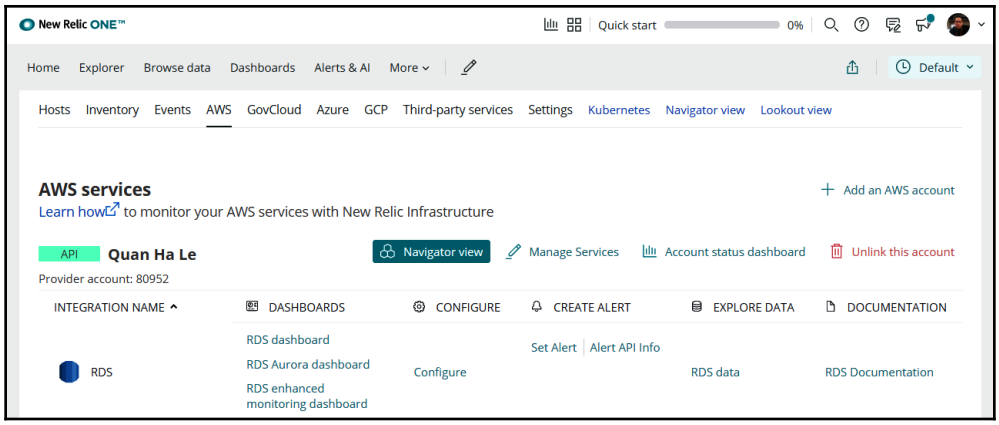

Figure 9-34. Your AWS services with a New Relic infrastructure

7. Click on **RDS dashboard** (Figure 9-34). After around 10 to 15 minutes, you will observe that the monitoring data will be captured from AWS PostgreSQL RDS in New Relic as shown in Figure 9-35:

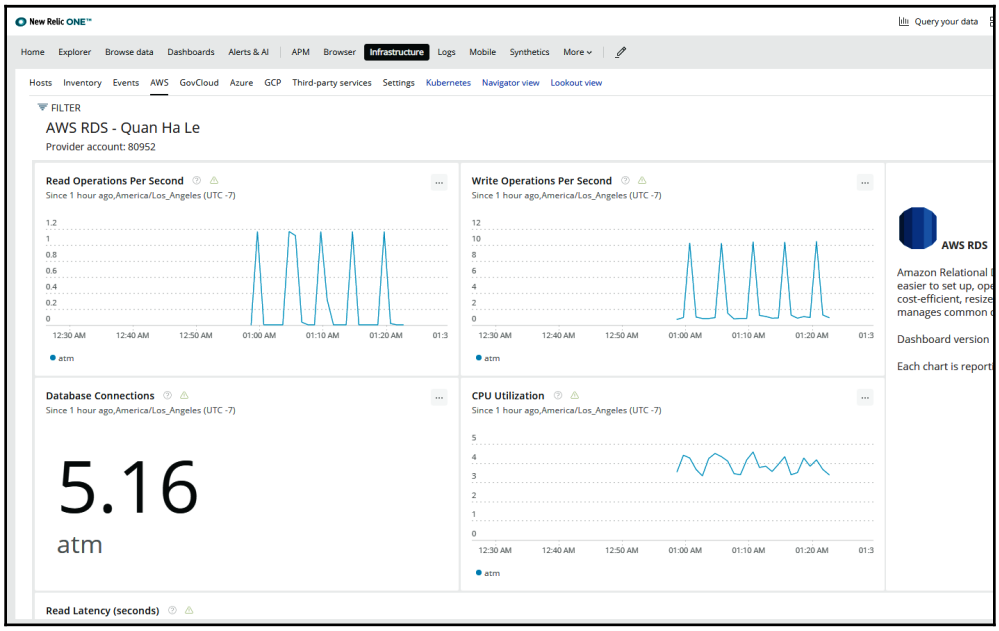

Figure 9-35. RDS dashboard of New Relic

New Relic uses an API as an intermediate layer between AWS RDS and the New Relic dashboard so that the charts in Figure 9-35 can be shown. The data sent from this API is called *metric data* hence the API can be called the *Metric API*. In the next section, we will go through some examples of New Relic metric data.

Adding new metric data for PostgreSQL

The Metric API is a way to get metric data from AWS PostgreSQL RDS into New Relic dashboards. A metric means a numeric measurement of an application, a system, or a database. Metrics are typically reported on a regular schedule. There are two types of metrics:

- Aggregated data: the rate of some event per minute
- A numeric status at a moment in time: for example, the `CPU% used` status

Metrics are a strong solution for monitoring dashboards. The metrics that are used in New Relic are as follows:

- `readIops`: This measures the average number of disk I/O operations per second.
- `writeIops`: This measures the average number of disk I/O operations per second.
- `databaseConnections`: This measures the number of connections to an instance.
- `cpuUtilization`: This measures the percentage of CPU used by a DB instance.
- `readLatency`: This measures the average amount of time taken per disk I/O operation, in seconds.
- `writeLatency`: This measures the average amount of time taken per disk I/O operation.
- `networkReceiveThroughput`: This measures the amount of network throughput received from clients by each instance in the Aurora MySQL DB cluster, in bytes per second.
- `networkTransmitThroughput`: This measures the amount of network throughput sent to clients by each instance in the Aurora DB cluster, in bytes per second.
- `swapUsageBytes`: This measures the amount of swap space used on the Aurora PostgreSQL DB instance, in bytes.

We will show you an example of `readIops` and `readLatency` in this section:

1. In order to see how New Relic uses the `readIops` metric, click on the **...** button on the **Read Operations Per Second** chart, then select **Copy to query builder**:

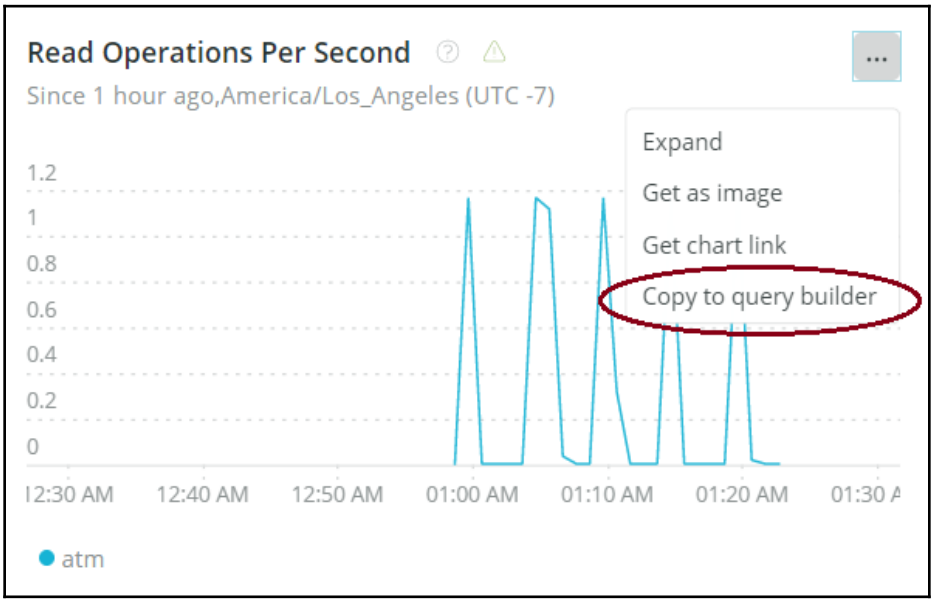

Figure 9-36. Read Operations Per Second chart

2. You will see the output of the following `SELECT` query in Figure 9-36. This query selects the average value of `readIops` measurements of our PostgreSQL RDS through the Metric API:

```
SELECT average(`provider.readIops.Average`) as 'Read Operations'
From DatastoreSample WHERE provider = 'RdsDbInstance' AND
providerAccountId = '34337' Since 355 minutes ago TIMESERIES Until
10 minutes ago facet displayName
```

Hence the values are an average of the read input/output operations per second:

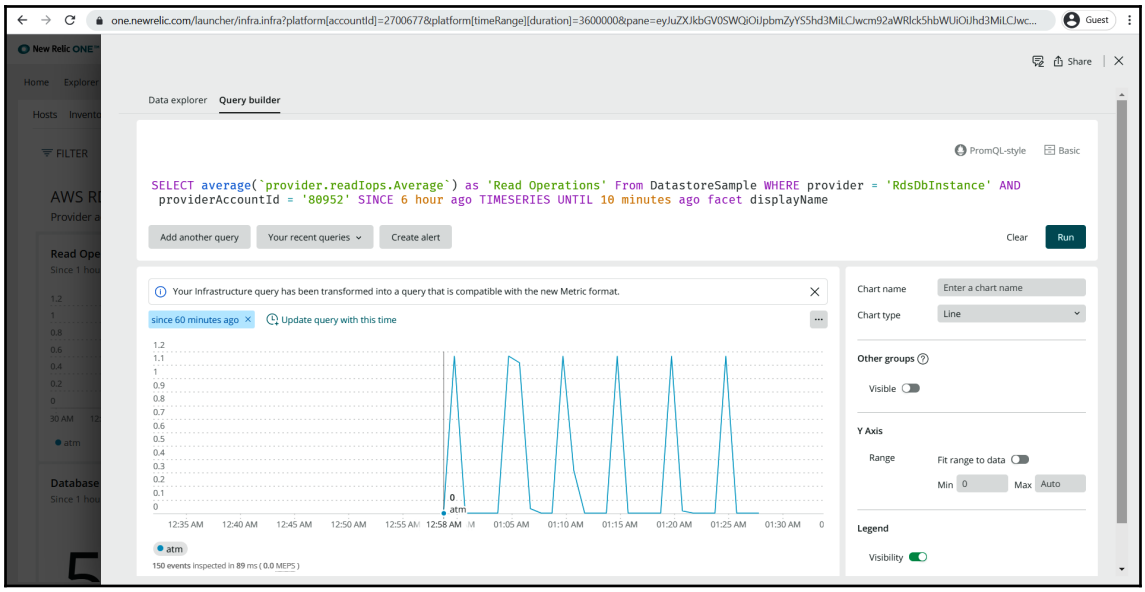

Figure 9-37. Average of readIops

3. Another example for the usage of the `readLatency` metric is with the `sum()` function. We will take the same steps but we will change the query:

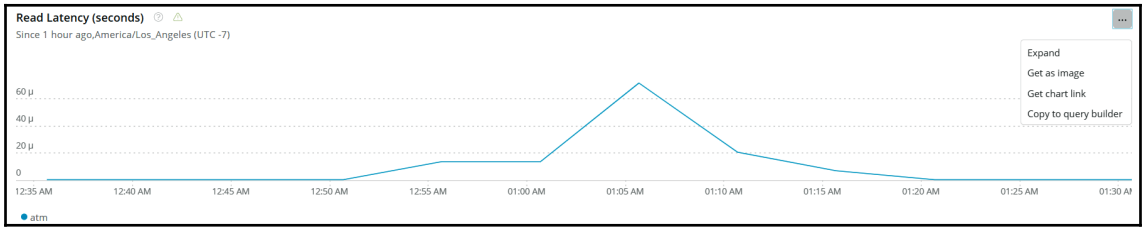

Figure 9-38. Read Latency chart

Here is the New Relic query that uses the `sum()` function:

```
SELECT sum(`provider.readLatency.Sum`) / 60 as 'seconds' From
DatastoreSample WHERE provider = 'RdsDbInstance' AND
providerAccountId = '34337' timeseries 5 minutes Until 10 minutes
ago Since 6 hours ago facet displayName
```

As for the previous query, we will copy it inside a modal window:

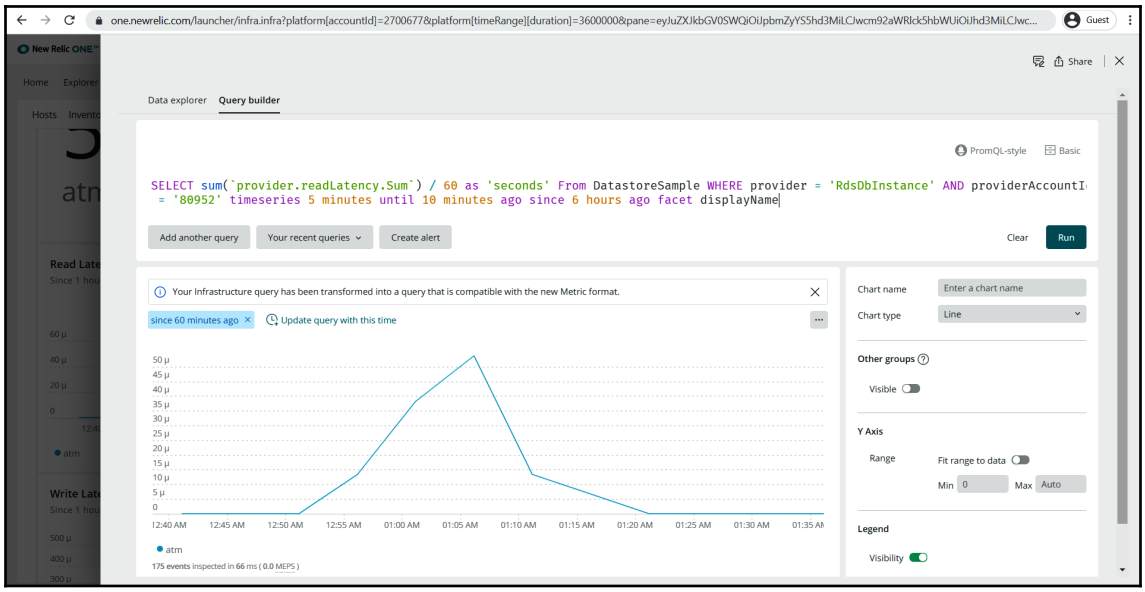

Figure 9-39. Sum of readLatency

In this section, you were introduced to the mechanism of how New Relic measures activities from your PostgreSQL RDS, which is metric data. You can open any charts from your New Relic dashboard of PostgreSQL RDS and try to view its query to get more experience with the usage of metric data.

Infrastructure inventory data collection

Each AWS service also has its own parameters and values defined by AWS settings. New Relic calls the AWS parameters as an *infrastructure inventory data collection*. So New Relic will also show the resource parameters on the dashboards so that the system administrator can easily look for the AWS settings and values directly in New Relic.

The **RDS Inventory** area is located at the end of the RDS dashboard of New Relic, as shown in Figure 9-40:

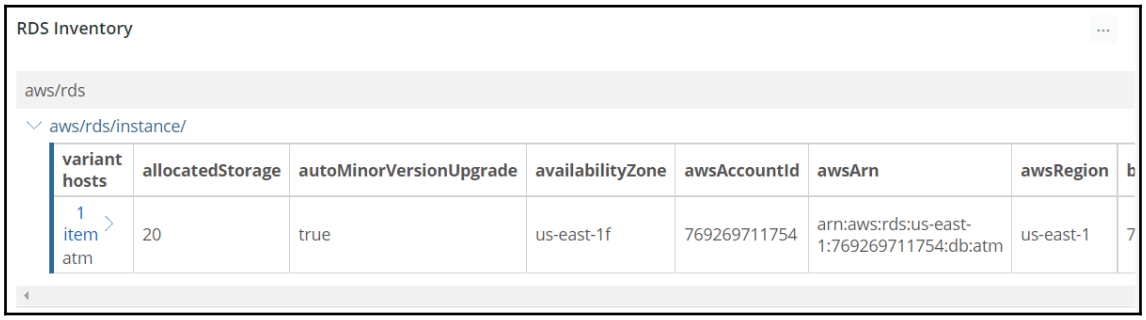

Figure 9-40. RDS Inventory

You can scroll right and left to see all of the inventory properties of the RDS. All of the property names and explanations are in the following table:

Name	Description
`awsRegion`	The AWS region that the instance was provisioned in.
`dbInstanceIdentifier`	Contains a user-supplied database identifier. This identifier is the unique key that identifies a DB instance.
`allocatedStorage`	Specifies the allocated storage size, in gibibytes.
`autoMinorVersionUpgrade`	Indicates that minor version patches are applied automatically.
`availabilityZone`	Specifies the name of the Availability Zone the DB instance is located in.
`backupRetentionPeriod`	Specifies the number of days for which automatic DB snapshots are retained.
`caCertificateIdentifier`	The identifier of the CA certificate for this DB instance.
`characterSetName`	If present, specifies the name of the character set that this instance is associated with.
`dbClusterIdentifier`	If the DB instance is a member of a DB cluster, contains the name of the DB cluster that the DB instance is a member of.
`dbInstanceClass`	Contains the name of the compute and memory capacity class of the DB instance.

Name	Description
`dbInstancePort`	Specifies the port that the DB instance listens on. If the DB instance is part of a DB cluster, this can be a different port than the DB cluster port.
`dbInstanceStatus`	Specifies the current state of this database.
`dbName`	Contains the name of the initial database of this instance that was provided at creation time, if one was specified when the DB instance was created. This same name is returned for the life of the DB instance.
`dbParameterGroups`	Provides the list of DB parameter groups applied to this DB instance.
`dbSecurityGroups`	Provides a list of DB security group elements.
`dbSubnetGroup`	Specifies information on the subnet group associated with the DB instance, including the name, description, and subnets in the subnet group.
`endpoint`	Specifies the connection endpoint.
`engine`	Provides the name of the database engine to be used for this DB instance.
`engineVersion`	Indicates the database engine version.
`kmsKeyId`	If `StorageEncrypted` is true, the AWS KMS key identifier for the encrypted DB instance.
`licenseModel`	License model information for this DB instance.
`masterUsername`	Contains the master username for the DB instance.
`multiAz`	Specifies whether the DB instance is a Multi-AZ deployment.
`optionGroupMemberships`	Provides the list of option group memberships for this DB instance.
`preferredBackupWindow`	Specifies the daily time range during which automated backups are created if automated backups are enabled, as determined by `BackupRetentionPeriod`.
`preferredMaintenanceWindow`	Specifies the weekly time range during which system maintenance can occur, in **Universal Coordinated Time** (**UTC**).
`publiclyAccessible`	Specifies the accessibility options for the DB instance. A value of true specifies an internet-facing instance with a publicly resolvable DNS name, which resolves to a public IP address. A value of false specifies an internal instance with a DNS name that resolves to a private IP address.

Name	Description
`readReplicaDdInstanceIdentifiers`	Contains one or more identifiers of the read replicas associated with this DB instance.
`readReplicaSourceDbInstanceIdentifier`	Contains the identifier of the source DB instance if this DB instance is a read replica.
`secondaryAvailabilityZone`	If present, specifies the name of the secondary Availability Zone for a DB instance with multi-AZ support.
`storageEncrypted`	Specifies whether the DB instance is encrypted.
`storageType`	Specifies the storage type associated with the DB instance.
`tdeCredentialArn`	The ARN from the key store with which the instance is associated for TDE encryption.
`vpcSecurityGroups`	Provides a list of VPC security group elements that the DB instance belongs to.
`clusterInstance`	Specifices whether the instance is a cluster or not.
`tags*`	Instance tags.

Table 9-1. RDS Inventory properties

The inventory area of New Relic provides a real-time, filterable, searchable view into the PostgreSQL RDS parameters. If you change any settings on your PostgreSQL RDS by using the AWS service, you will immediately see the new values updated on the New Relic inventory. New Relic is very convenient and easy to use, so whenever you need to, you can look directly at the New Relic dashboard instead of opening the AWS service to search for a database parameter.

Summary

In this chapter, we went through step-by-step PostgreSQL RDS database monitoring through New Relic, which is a popular DevOps tool. We learned how to register the Amazon RDS monitoring integration of New Relic. Going ahead, we learned how to prepare the IAM role for New Relic to collect PostgreSQL metrics. We also investigated the RDS dashboard of New Relic. Finally, we understood the metric data for PostgreSQL and learned how to read the RDS inventory data.

This chapter focused on PostgreSQL monitoring for high-performance teams. In the next chapter, we will move on to PostgreSQL database performance monitoring with PGBench and Apache JMeter.

10
Testing the Performance of Our Banking App with PGBench and JMeter

In this chapter, we will learn how to create a load test for a PostgreSQL database to benchmark PostgreSQL performance with pgbench and JMeter. The purpose of benchmarking a database is not only to check the capability of the database but also to check the behavior of a particular database against your application. Different hardware provides different results based on the benchmarking plan that you set. It is very important to isolate the server (the actual one being benchmarked) from other elements such as the servers driving the load or the servers used to collect and store performance metrics. After benchmarking, you will also get application characteristics.

With the help of a project in this chapter, we will create and run load tests to benchmark the performance of the PostgreSQL 12 RDS from Amazon Web Services from Chapter 2, *Creating a Geospatial Database Using PostGIS and PostgreSQL*, related to bank ATM locations in a typical city.

The following topics will be covered in this chapter:

- How to benchmark PostgreSQL performance
- pgbench 1 – Creating and initializing a benchmark database
- pgbench 2 – Running a baseline pgbench test
- pgbench 3 – Creating and testing a connection pool
- JMeter setup
- JMeter for AWS PostgreSQL RDS

Technical requirements

This chapter will take developers around 10-12 hours of work to develop PostgreSQL performance tests.

How to benchmark PostgreSQL performance

We usually have to benchmark our PostgreSQL database to make sure that before we deliver our products for customers, the capability of the PostgreSQL database will be able to cover the expected traffic loads in production.

Benchmarks are super important in our initial implementation stage since we can estimate and answer questions such as the following:

- How many users can we support?
- What response time will we have at peak usage time?
- Will the storage space have enough capacity?
- Will the current hardware meet expectations?

Based on the answers that we obtain, we can adjust tuning parameters in PostgreSQL and at the same time, estimate what hardware we will need in the future.

Having said all that, one of the common ways to benchmark PostgreSQL is through two typical tools:

- pgbench, a built-in benchmarking tool of PostgreSQL
- Apache JMeter software, a tool designed to load test functional behavior and measure performance

In the following topics, we will perform some strength tests on our ATM database on PostgreSQL RDS and we will see its behavior through those benchmarking tools.

pgbench 1 – Creating and initializing a benchmark database

We are going to use pgbench for the RDS on Amazon Web Services. We will set up pgbench from the Jenkins server at `192.168.0.200` that we set up with Vagrant in Chapter 7, *PostgreSQL with DevOps for Continuous Delivery*. We PuTTY into the Jenkins server:

1. First, open PowerShell as administrator:

```
PS C:\Windows\system32>
PS C:\Windows\system32> cd C:\Projects\Vagrant\Jenkins
PS C:\Projects\Vagrant\Jenkins> vagrant up --provider virtualbox
PS C:\Projects\Vagrant\Jenkins> vagrant ssh

vagrant@devopsubuntu1804:~$
```

2. Execute the following script to install pgbench:

```
vagrant@devopsubuntu1804:~$ wget --quiet -O -
https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key
add -

vagrant@devopsubuntu1804:~$ echo "deb
http://apt.postgresql.org/pub/repos/apt/ `lsb_release -cs`-pgdg
main" |sudo tee /etc/apt/sources.list.d/pgdg.list

vagrant@devopsubuntu1804:~$ sudo apt update
vagrant@devopsubuntu1804:~$ sudo apt -y install postgresql-contrib
vagrant@devopsubuntu1804:~$ pgbench -V
pgbench (PostgreSQL) 12.2 (Ubuntu 12.2-2.pgdg18.04+1)
```

3. Now we will initialize the benchmark database by using the following initialization syntax:

```
Syntax:
pgbench -i [-h hostname] [-p port] [-s scaling_factor] [-U login]
[-d] [dbname]
```

The parameter definitions are given as follows:

- `scaling_factor`: The number of tuples generated will be a multiple of the scaling factor. For example, `-s 1` will imply 100,000 tuples in the `pgbench_accounts` table.
- `port`: Default `5432`.
- `hostname`: Default localhost.
- `-d`: Shows debug information.

4. Hence we issue the `pgbench` command to our RDS endpoint with 1 million tuples:

```
vagrant@devopsubuntu1804:~$ pgbench -i -h atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com -p 5432 -s 10 -U dba -d atm
Password: (please enter the RDS password = bookdemo)
dropping old tables...
NOTICE: table "pgbench_accounts" does not exist, skipping
NOTICE: table "pgbench_branches" does not exist, skipping
NOTICE: table "pgbench_history" does not exist, skipping
NOTICE: table "pgbench_tellers" does not exist, skipping
creating tables...
generating data...
100000 of 1000000 tuples (10%) done (elapsed 1.93 s, remaining
17.34 s)
200000 of 1000000 tuples (20%) done (elapsed 4.27 s, remaining
17.09 s)
300000 of 1000000 tuples (30%) done (elapsed 6.82 s, remaining
15.91 s)
400000 of 1000000 tuples (40%) done (elapsed 9.46 s, remaining
14.19 s)
500000 of 1000000 tuples (50%) done (elapsed 12.12 s, remaining
12.12 s)
600000 of 1000000 tuples (60%) done (elapsed 14.49 s, remaining
9.66 s)
700000 of 1000000 tuples (70%) done (elapsed 17.13 s, remaining
7.34 s)
800000 of 1000000 tuples (80%) done (elapsed 19.80 s, remaining
4.95 s)
900000 of 1000000 tuples (90%) done (elapsed 22.16 s, remaining
2.46 s)
1000000 of 1000000 tuples (100%) done (elapsed 24.83 s, remaining
0.00 s)
vacuuming...
creating primary keys...
done.
```

Please look at *Figure 10.1* for an illustration:

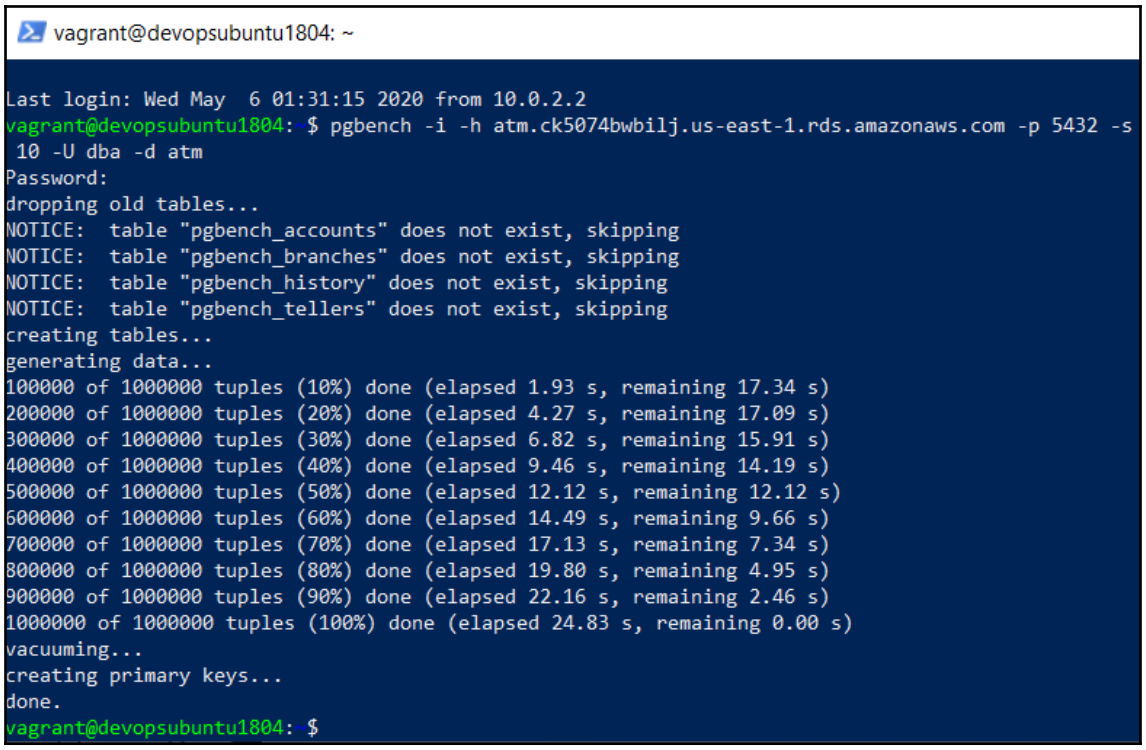

Figure 10.1. pgbench initialization

After the `pgbench` command has been completed with the parameters that we defined previously, we can open the RDS database by using pgAdmin and we will see that there are four more pgbench tables added – **pgbench_accounts**, **pgbench_branches**, **pgbench_history**, and **pgbench_tellers**:

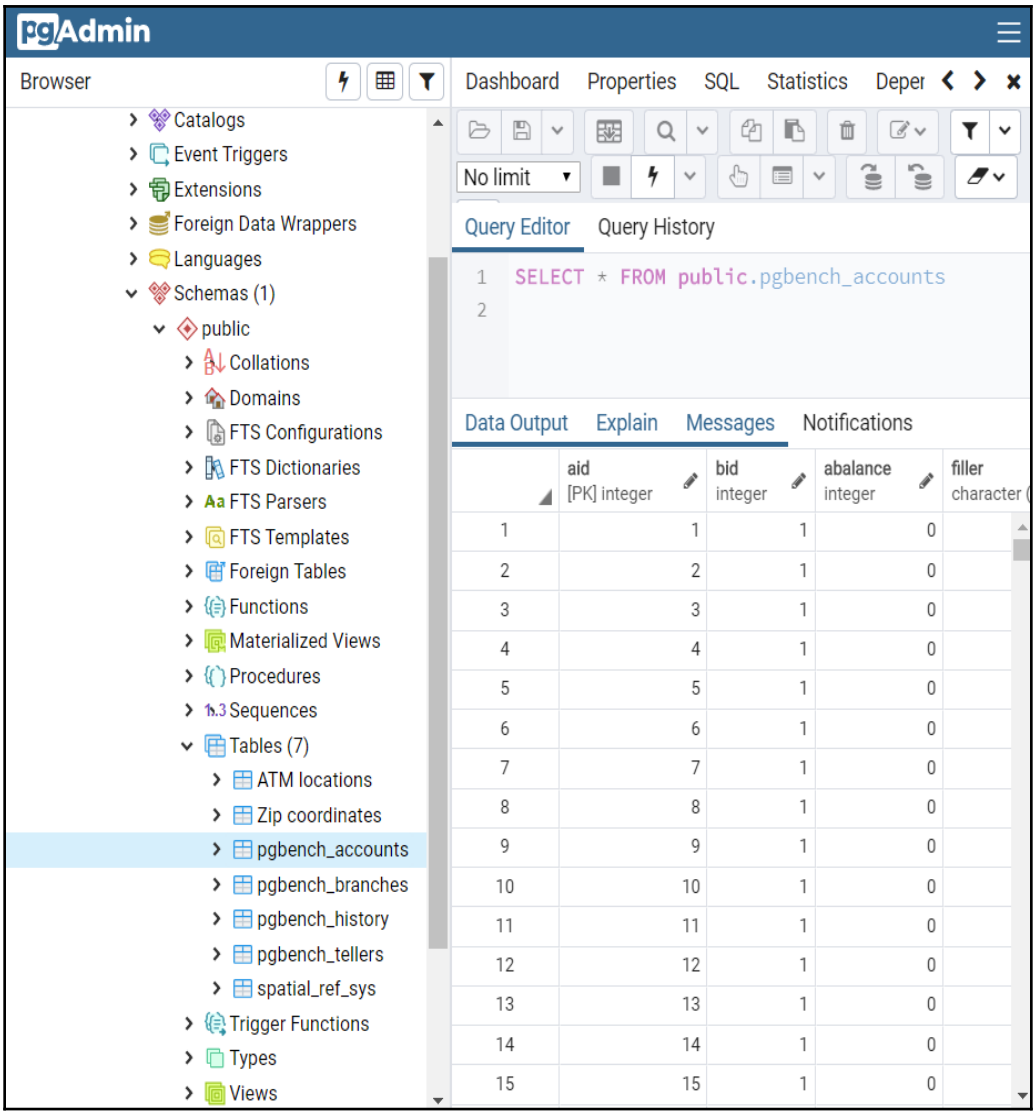

Figure 10.2. pgbench tables

Obviously, from pgAdmin, we can easily check that there are 1 million rows generated in the table `pgbench_accounts`.

pgbench 2 – Running a baseline pgbench test

1. This is the general syntax to run a pgbench benchmarking test:

```
Syntax:
pgbench  [-h hostname]  [-p port]  [-c nclients]  [-t
ntransactions]  [-s scaling_factor] [-D varname=value]  [-n]  [-C]
[-v] [-S]  [-N]  [-f filename]  [-j threads]  [-l]  [-U login]  [-
d]  [dbname]
```

We'll see the following parameters:

- `nclients`: Client number – default 1
- `ntransactions`: Transactions per client – default 10
- `varname=value`: Defined variable referred to by the file script
- `-n`: novacuuming – cleaning the history table before testing
- `-C`: One connection for each transaction
- `-v`: Vacuuming before testing
- `-S`: Only `SELECT` transactions
- `-N`: No updates on the `pgbench_branches` and `pgbench_tellers` tables
- `filename`: Read the transaction script from a specified SQL file
- `threads`: The number of worker threads
- `-l`: Write the time of each transaction to a `pgbench_log` file

2. Hence we can create a small load of 50 users. Each user runs 100 transactions with 2 threads on the RDS `atm` database:

```
vagrant@devopsubuntu1804:~$ pgbench -h atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com -U dba -c 50 -j 2 -t 100 -d atm
pghost: atm.ck5074bwbilj.us-east-1.rds.amazonaws.com pgport:
nclients: 50 nxacts: 100 dbName: atm
Password: (please enter the RDS password = bookdemo)
...
...
client 49 executing script "<builtin: TPC-B (sort of)>"
client 49 executing \set aid
client 49 executing \set bid
client 49 executing \set tid
```

```
client 49 executing \set delta
client 49 sending BEGIN;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_accounts SET abalance = abalance +
-1345 WHERE aid = 938376;
client 49 receiving
client 49 receiving
client 49 sending SELECT abalance FROM pgbench_accounts WHERE aid =
938376;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_tellers SET tbalance = tbalance +
-1345 WHERE tid = 63;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_branches SET bbalance = bbalance +
-1345 WHERE bid = 1;
client 49 receiving
client 49 receiving
client 49 sending INSERT INTO pgbench_history (tid, bid, aid,
delta, mtime) VALUES (63, 1, 938376, -1345, CURRENT_TIMESTAMP);
client 49 receiving
client 49 receiving
client 49 sending END;
client 49 receiving
client 49 receiving
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 50
number of threads: 2
number of transactions per client: 100
number of transactions actually processed: 5000/5000
latency average = 1008.361 ms
tps = 49.585429 (including connections establishing)
tps = 49.797092 (excluding connections establishing)
```

Therefore, the total number of transactions will be 50 users x 100 transactions = 5,000 transactions. The throughput of pgbench shows that less than 50 transactions are completed per second:

vagrant@devopsubuntu1804: ~

```
client 10 sending END;
client 10 receiving
client 49 receiving
client 49 sending END;
client 49 receiving
client 10 receiving
client 49 receiving
client 49 executing script "<builtin: TPC-B (sort of)>"
client 49 executing \set aid
client 49 executing \set bid
client 49 executing \set tid
client 49 executing \set delta
client 49 sending BEGIN;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_accounts SET abalance = abalance + -1345 WHERE aid = 938376;
client 49 receiving
client 49 receiving
client 49 sending SELECT abalance FROM pgbench_accounts WHERE aid = 938376;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_tellers SET tbalance = tbalance + -1345 WHERE tid = 63;
client 49 receiving
client 49 receiving
client 49 sending UPDATE pgbench_branches SET bbalance = bbalance + -1345 WHERE bid = 1;
client 49 receiving
client 49 receiving
client 49 sending INSERT INTO pgbench_history (tid, bid, aid, delta, mtime) VALUES (63, 1, 938376
, -1345, CURRENT_TIMESTAMP);
client 49 receiving
client 49 receiving
client 49 sending END;
client 49 receiving
client 49 receiving
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 50
number of threads: 2
number of transactions per client: 100
number of transactions actually processed: 5000/5000
latency average = 1008.361 ms
tps = 49.585429 (including connections establishing)
tps = 49.797092 (excluding connections establishing)
vagrant@devopsubuntu1804:~$
```

Figure 10.3. pgbench performance for 50 users

3. Now let's run the same test. This time, we will use a number of users higher than our RDS allowance. Hence, from the pgAdmin browser, please select the atm database and execute the following `SELECT` statement:

```
select * from pg_settings where name='max_connections';
```

The value of `max_connections` of the PostgreSQL RDS is `87` for the atm database, as shown in the following screenshot. That capacity is correct for our RDS size – `db.t2.micro`:

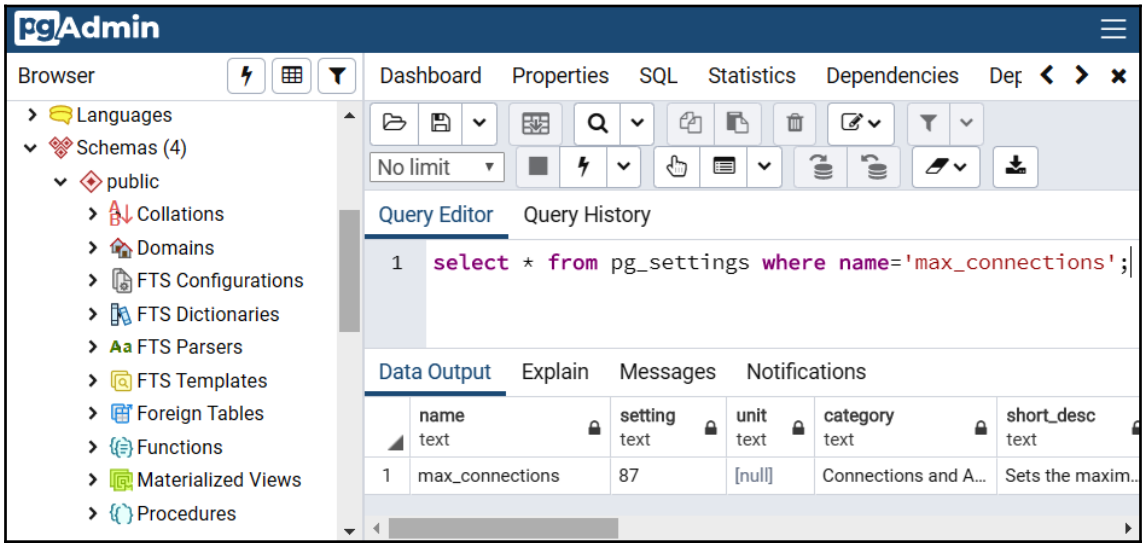

Figure 10.4. Connection allowance of the db.t2.micro PostgreSQL RDS

4. We will now use 150 concurrent users for our next pgbench test – a value that is higher than `max_connections=87` for this PostgreSQL database – to synthetically simulate a mass influx of client connections:

```
vagrant@devopsubuntu1804:~$ pgbench -h atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com -U dba -c 150 -j 2 -t 100 -d atm
pghost: atm.ck5074bwbilj.us-east-1.rds.amazonaws.com pgport:
nclients: 150 nxacts: 100 dbName: atm
Password: (please enter the RDS password = bookdemo)
starting vacuum...end.
connection to database "atm" failed:
FATAL: remaining connection slots are reserved for non-replication
superuser connections
...
...
```

```
client 37 executing script "<builtin: TPC-B (sort of)>"
client 37 executing \set aid
client 37 executing \set bid
client 37 executing \set tid
client 37 executing \set delta
client 37 sending BEGIN;
client 37 receiving
client 37 receiving
client 37 sending UPDATE pgbench_accounts SET abalance = abalance +
4605 WHERE aid = 560242;
client 37 receiving
client 37 receiving
client 37 sending SELECT abalance FROM pgbench_accounts WHERE aid =
560242;
client 37 receiving
client 37 receiving
client 37 sending UPDATE pgbench_tellers SET tbalance = tbalance +
4605 WHERE tid = 20;
client 37 receiving
client 37 receiving
client 37 sending UPDATE pgbench_branches SET bbalance = bbalance +
4605 WHERE bid = 7;
client 37 receiving
client 37 receiving
client 37 sending INSERT INTO pgbench_history (tid, bid, aid,
delta, mtime) VALUES (20, 7, 560242, 4605, CURRENT_TIMESTAMP);
client 37 receiving
client 37 receiving
client 37 sending END;
client 37 receiving
client 37 receiving
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 150
number of threads: 2
number of transactions per client: 100
number of transactions actually processed: 7500/15000
latency average = 3138.897 ms
tps = 47.787488 (including connections establishing)
tps = 47.853332 (excluding connections establishing)
```

Note the **FATAL** error, indicating that pgbench hits the 87-connection limit threshold set by `max_connections`, resulting in a refused connection. The test was still able to complete, with a throughput of less than 48 as follows:

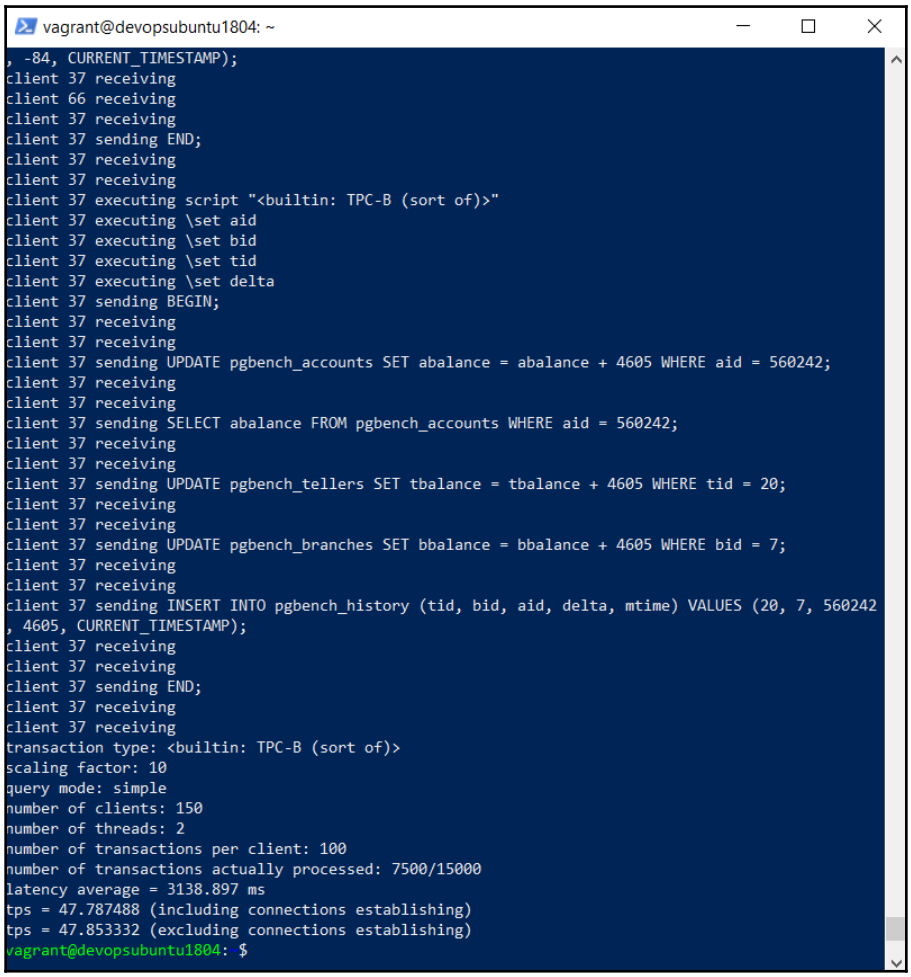

Figure 10.5. Errors of a higher load than the PostgreSQL max_connections allowance

We can check the preceding figure and see that pgbench requires 150 users x 100 transactions = 15,000 transactions, but this load test is only able to complete 7,500 transactions because the maximum connections with PostgreSQL is only 87. Hence, we conclude that our RDS performance is not good for 150 users.

Because 150 users is a small number, we will now move on to the next section to resolve the preceding error so that our RDS can improve the performance. In addition to that error, pgbench statements always ask us to enter the RDS password, which we can utilize to resolve errors in the next section.

pgbench 3 – Creating and testing a connection pool

At the moment, Amazon Web Services is doing an experiment with a database connection pool service called RDS Proxy. In order to access the RDS Proxy service, please open the RDS dashboard (`https://console.aws.amazon.com/rds`) and select the **Proxies** tab on the left panel as shown in the following figure:

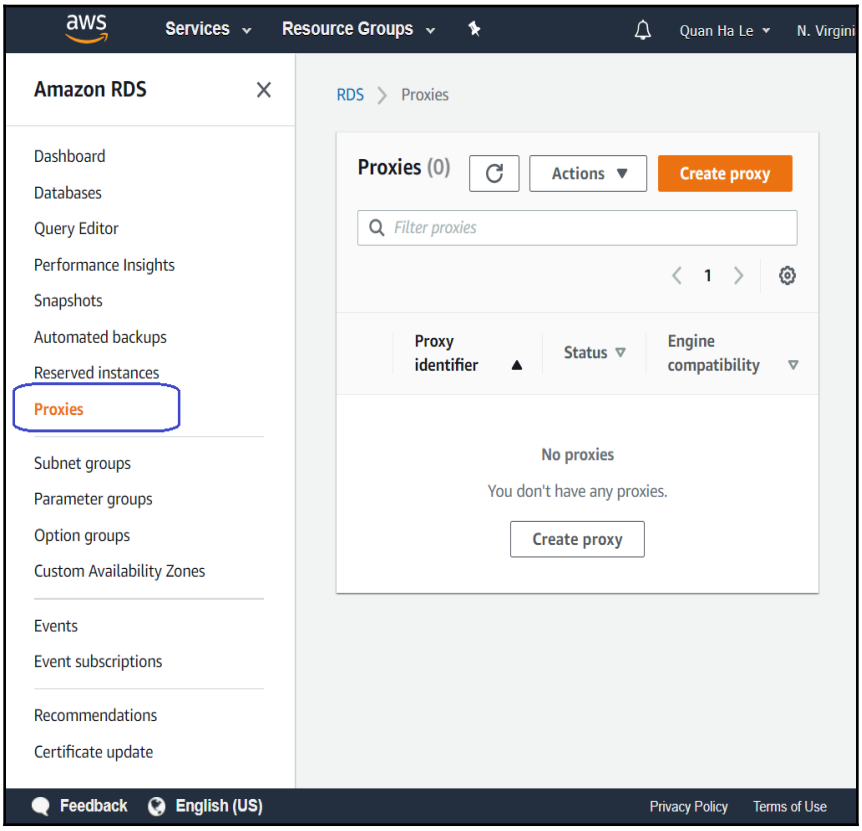

Figure 10.6. Experimental AWS RDS Proxy

Figure 10.7 shows how the RDS proxy works as a connection pool between applications and RDS databases:

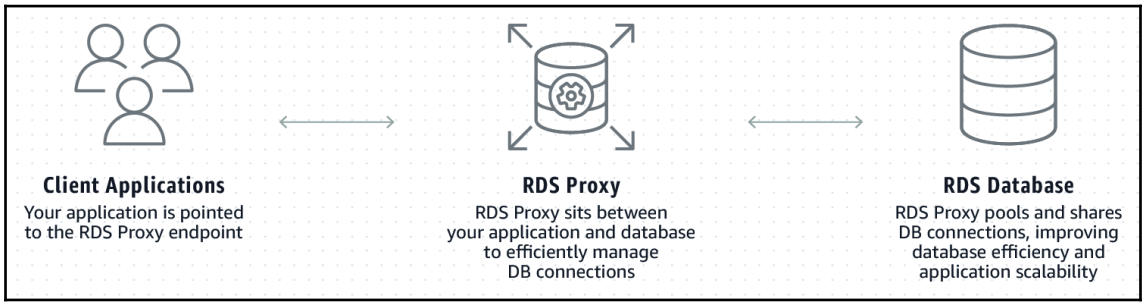

Figure 10.7. Principles of AWS RDS Proxy

However, currently, the RDS Proxy service is still in preview only. With this preview status, the RDS proxy service is able to create connection pools for Aurora MySQL RDS, MySQL RDS, and Aurora PostgreSQL RDS. This means that this AWS connection pool service is not available for our current PostgreSQL version 12.2.

Therefore, we are going to install a PostgreSQL connection pool on our own with either of the following:

- `pgpool`
- `pgbouncer`

I have already tried `pgpool` before, hence if the pgbench performance tool creates a higher number of user connections than the *max_connections = 87* allowance of our RDS, our test could show that `pgpool` uses the 2 parameters `max_pool` and `num_init_children`. The `max_pool` parameter configures how many connections `pgpool` can cache for each child process. For example, if `num_init_children` is configured to 100, and `max_pool` is configured to 3, then `pgpool` could potentially open 300 (=3*100) connections to the RDS.

However, `pgpool` also sets a restriction on the `max_connections` value of PostgreSQL RDS as you can see in the next formula:

```
max_pool*num_init_children <= (max_connections -
superuser_reserved_connections)
```

Our PostgreSQL RDS `superuser_reserved_connections` is 10.

Therefore, because of the preceding `pgpool` restriction, we cannot use `pgpool` to resolve our PostgreSQL connection error. Therefore, we will look towards `pgbouncer` and take the next steps:

1. We will install `pgbouncer` for the Jenkins server:

```
vagrant@devopsubuntu1804:~$ sudo apt update
vagrant@devopsubuntu1804:~$ echo "deb
http://apt.postgresql.org/pub/repos/apt/ `lsb_release -cs`-pgdg
main" | sudo tee  /etc/apt/sources.list.d/pgdg.list
vagrant@devopsubuntu1804:~$ wget --quiet -O -
https://www.postgresql.org/media/keys/ACCC4CF8.asc | sudo apt-key
add -
vagrant@devopsubuntu1804:~$ sudo apt update
vagrant@devopsubuntu1804:~$ sudo apt install -y pgbouncer
```

The installation of `pgbouncer` is easy and smooth, as we saw previously:

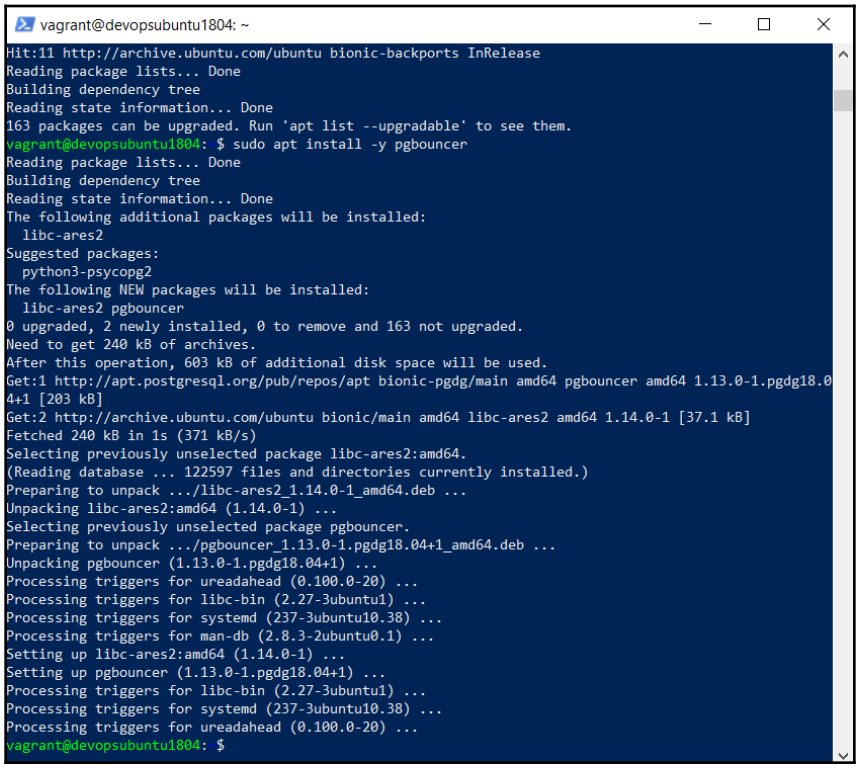

Figure 10.8. Installation of pgbouncer

2. We will edit the configuration file for `pgbouncer` to connect to the RDS endpoint and the `atm` database. Only For demonstration purposes, we will use it *without ssl* and password authentication. Please do not use it in production environments:

```
vagrant@devopsubuntu1804:~$ sudo nano /etc/pgbouncer/pgbouncer.ini
------------------------------------------------------------------------
--
[databases]
atm = host=atm.ck5074bwbilj.us-east-1.rds.amazonaws.com port=5432
dbname=atm

[pgbouncer]
listen_addr = *
listen_port = 6432
auth_type = trust
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 200
max_db_connections = 70
server_reset_query =
------------------------------------------------------------------------
--
```

Please use *Ctrl + O* to write the changes to the file:

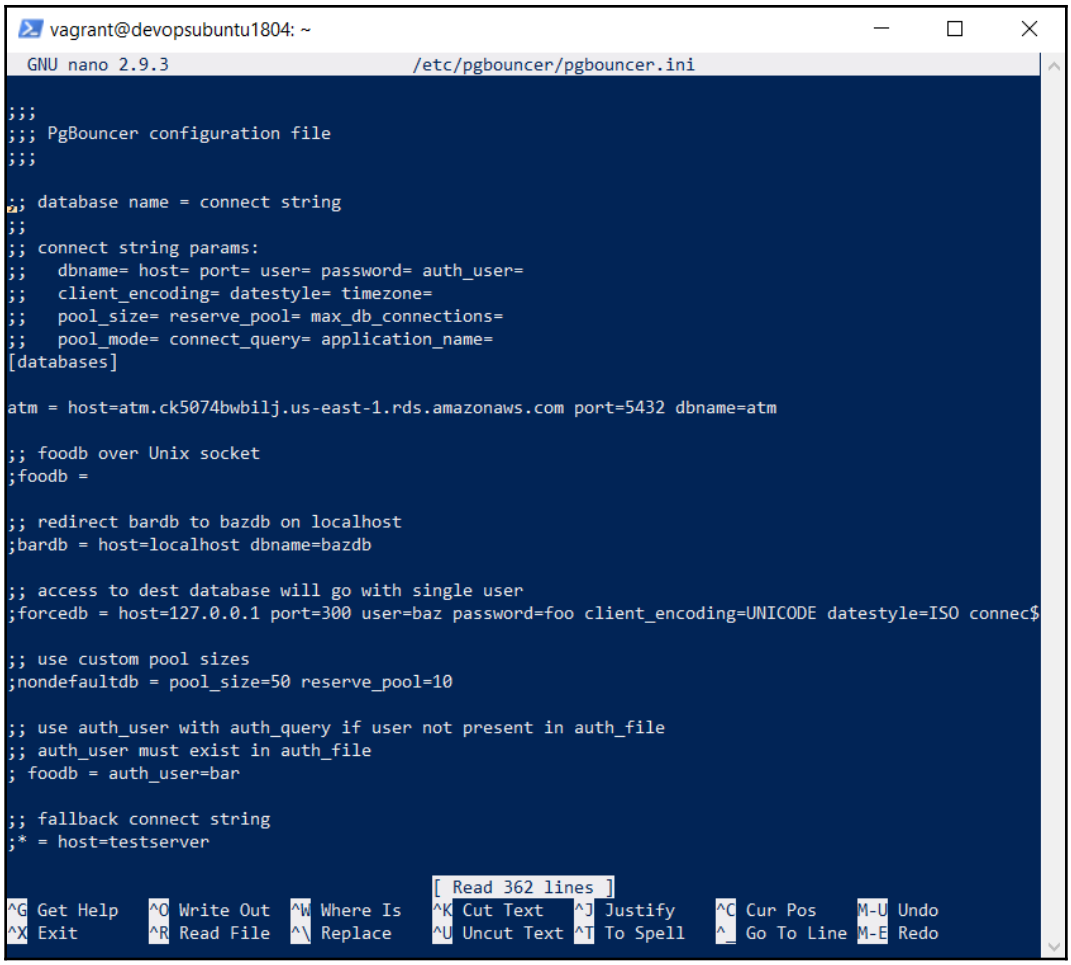

Figure 10.9. Configuration of pgbouncer

3. We will create the `pgbouncer` authorization file:

```
vagrant@devopsubuntu1804:~$ sudo nano /etc/pgbouncer/userlist.txt
------------------------------------------------------------------------
--
"dba"    "bookdemo"
------------------------------------------------------------------------
--
```

The user for `pgbouncer` to connect to the RDS is `dba` and the password is `bookdemo`.

4. At the end, we will restart the `pgbouncer` service:

```
vagrant@devopsubuntu1804:~$ sudo systemctl stop pgbouncer
vagrant@devopsubuntu1804:~$ sudo systemctl start pgbouncer
```

5. If all the settings are correct, we can call `psql` to connect from our Jenkins server to the RDS through port `6432` of pgbouncer:

```
vagrant@devopsubuntu1804:~$ sudo psql --dbname=atm --port=6432 --
username=dba
psql (12.2 (Ubuntu 12.2-2.pgdg18.04+1))
Type "help" for help.

atm=> SELECT * FROM public."ATM locations" LIMIT 10;
```

Hence, pgbouncer is working alright and it does a pooling task for the psql connection as follows:

Figure 10.10. psql connection calls to pgbouncer port 6432

6. We'll now use pgbouncer port `6432` to create a pgbench load of 150 users more than the `max_connections` = 87 allowance of our RDS:

```
vagrant@devopsubuntu1804:~$ pgbench -h atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com -p 5432 -U dba -c 150 -j 2 -t 100 -d atm

pghost: localhost pgport: 6432 nclients: 150 nxacts: 100 dbName:
atm
...
...
client 147 sending UPDATE pgbench_branches SET bbalance = bbalance
+ -4201 WHERE bid = 2;
client 147 receiving
client 148 receiving
client 110 receiving
client 110 sending INSERT INTO pgbench_history (tid, bid, aid,
delta, mtime) VALUES (29, 6, 647981, 3295, CURRENT_TIMESTAMP);
client 110 receiving
client 134 receiving
client 147 receiving
client 147 sending INSERT INTO pgbench_history (tid, bid, aid,
delta, mtime) VALUES (62, 2, 543082, -4201, CURRENT_TIMESTAMP);
client 147 receiving
client 110 receiving
client 110 sending END;
client 110 receiving
client 147 receiving
client 147 sending END;
client 147 receiving
client 110 receiving
client 147 receiving
transaction type: <builtin: TPC-B (sort of)>
scaling factor: 10
query mode: simple
number of clients: 150
number of threads: 2
number of transactions per client: 100
number of transactions actually processed: 15000/15000
latency average = 7540.436 ms
tps = 19.892748 (including connections establishing)
tps = 19.892775 (excluding connections establishing)
vagrant@devopsubuntu1804:~$
```

This time, pgbouncer makes a very good connection pool layer between pgbench and the RDS. We can see from the 150-user load that a total of 15,000 transactions are all completed with the throughput of roughly 20 completed transactions per second:

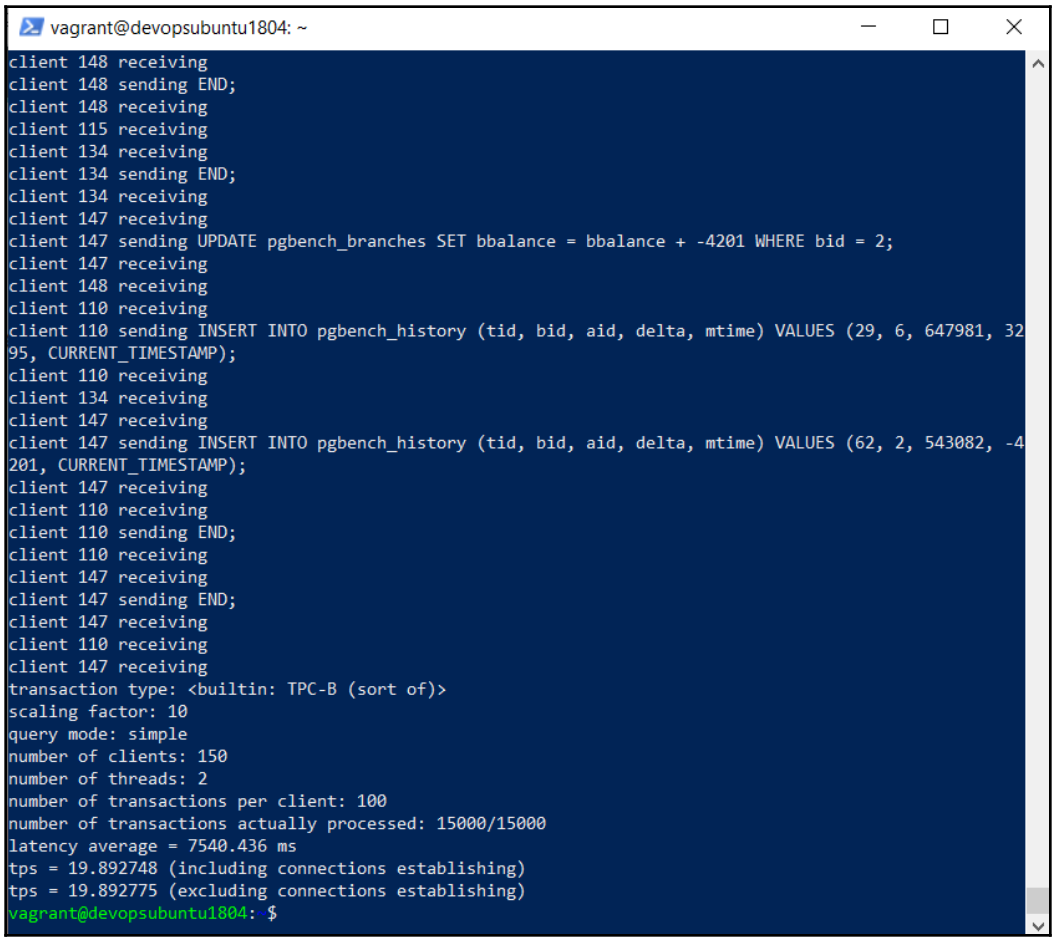

Figure 10.11. pgbouncer makes a good connection pool for pgbench

Also, pgbouncer can connect automatically into the RDS without it asking for a password.

These are our settings for pgbouncer:

```
max_client_conn = 200
max_db_connections = 70
```

We can explain this result as follows:

- At first, pgbouncer accepts when pgbench launches 150 users because pgbouncer is set up with `max_client_conn` = 200, which is more than 150.
- Then pgbouncer pools 70 user connections `(max_db_connections` = 70) for pgbench to our RDS. This does not cause any errors because the allowance of PostgreSQL RDS is equal to `(max_connections - superuser_reserved_connections) = (87 - 10) = 77` connections larger than 70.
- Whenever each of the first 70 connections inside the pgbouncer pool has done its transactions, then pgbouncer can reuse the same idle connection for one of the remaining (150 - 70) users = 80 users.
- Therefore, pgbouncer creates a good connection pool to allow clients to execute a larger number of users than the real RDS connection allowance.

pgbouncer has done provided a good PostgreSQL benchmark but it generates its own data to send its own transactions over its pgbench tables only, therefore we decide to move on by using the JMeter tool so that we can use our atm data for load tests.

JMeter setup

Because JMeter will be easier to understand with the GUI, we are going to reuse the "QGIS Windows" EC2 instance from `Chapter 5`, *Creating a Geospatial Database Using PostGIS and PostgreSQL*, in the QGIS section.

We repeat the steps from Chapter 5 as follows:

1. Choose the **Amazon Machine Image** (**AMI**) from Chapter 5:

   Microsoft Windows Server 2019 Base – `ami-04ca2d0801450d495`
   This AMI can be seen at the top of *Figure 5.10* in Chapter 5.

2. The EC2 instance was shown in *Figure 5.11* in Chapter 5.
3. Please open the "QGIS Windows" EC2 instance with Remote Desktop Connection as shown in *Figure 5.12* in Chapter 5, Creating a Geospatial Database Using PostGIS and PostgreSQL.

3. Download and install the newest JMeter version here:
   `https://jmeter.apache.org/download_jmeter.cgi`

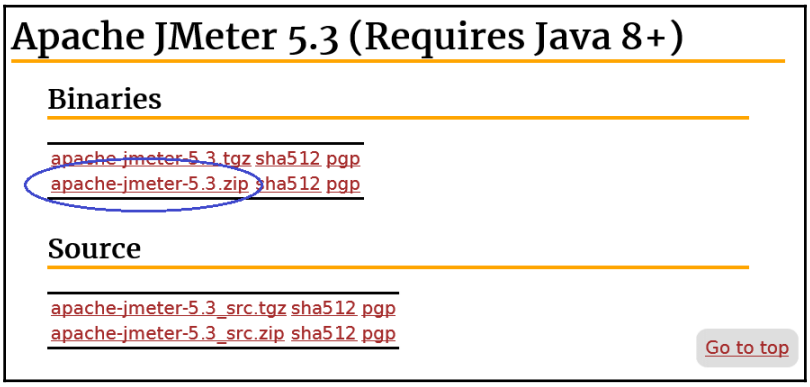

Figure 10.12. The newest JMeter versions

5. Let's save the JMeter ZIP file into a folder of EC2 such as `C:\JMeter\`. We then can extract the ZIP file into the folder `C:\JMeter\apache-jmeter-5.3\`.
6. We will then download the JDBC driver for PostgreSQL from the website `https://jdbc.postgresql.org/download.html`:

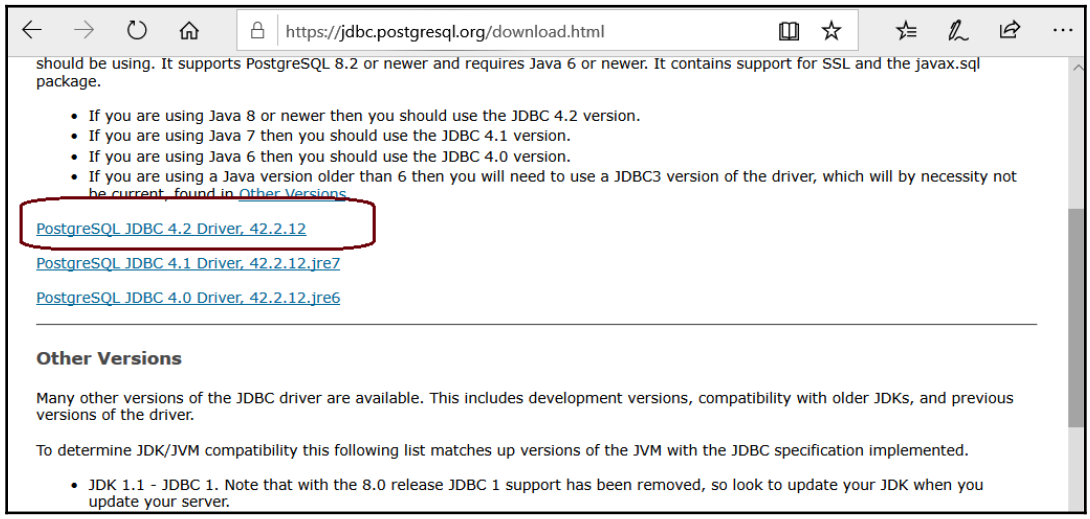

Figure 10.13. The PostgreSQL JDBC driver for JMeter

7. We have to save the JDBC driver file `postgresql-42.2.12.jar` to the JMeter lib folder, that is, `C:\JMeter\apache-jmeter-5.3\lib\`.
8. Now our JMeter is ready for PostgreSQL, we can execute the file `C:\JMeter\apache-jmeter-5.3\bin\jmeter.bat` to launch JMeter:

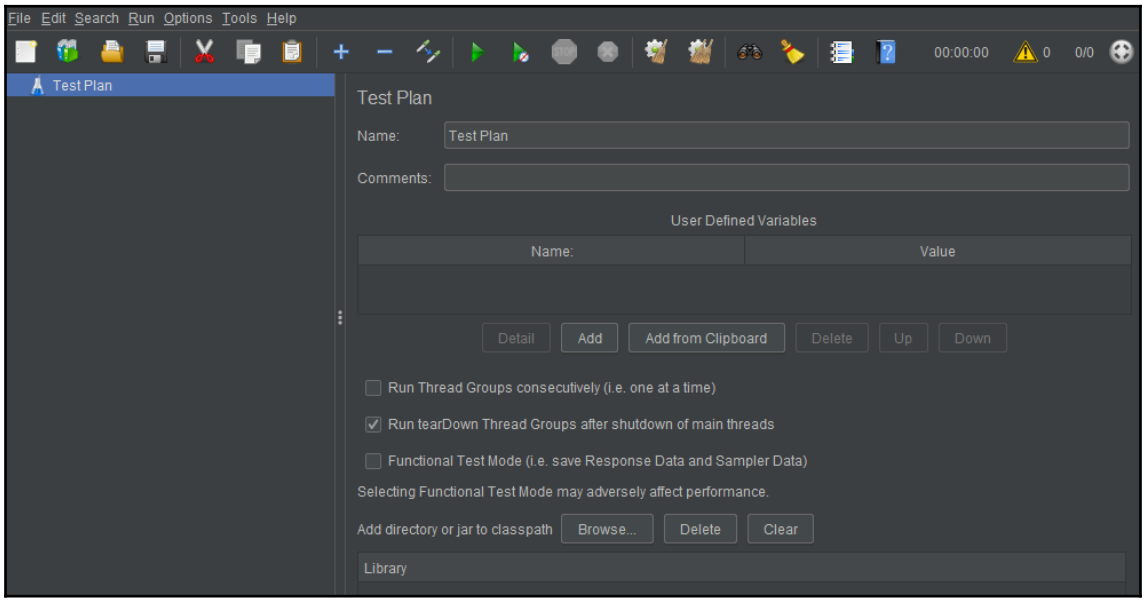

Figure 10.14. The first launch of Apache JMeter on Windows

In order to run a load test from JMeter, we will create a JMeter test plan in the next section.

JMeter for AWS PostgreSQL RDS

A JMeter test plan for AWS PostgreSQL RDS includes a user thread group; its thread group will include elements of variables, JDBC connections, JDBS requests, and performance reports:

1. Please create a thread group by using the menu **Edit** > **Add** > **Threads (Users)** > **Thread Grou**p. Or, we can simply right-click on **Test Plan** to pop up the **Add** > **Threads (Users)** > **Thread Group menu** options:

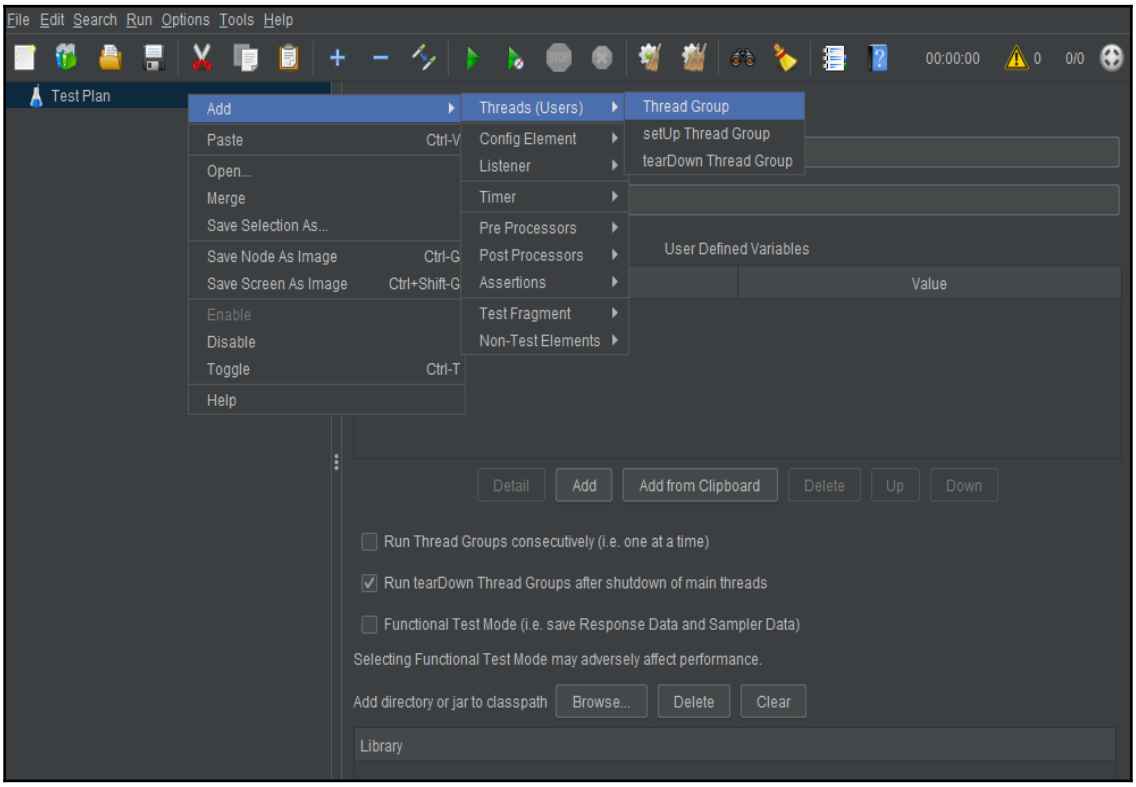

Figure 10.15. Creation of a new thread group by JMeter

We will see the default values of a thread group are as follows:

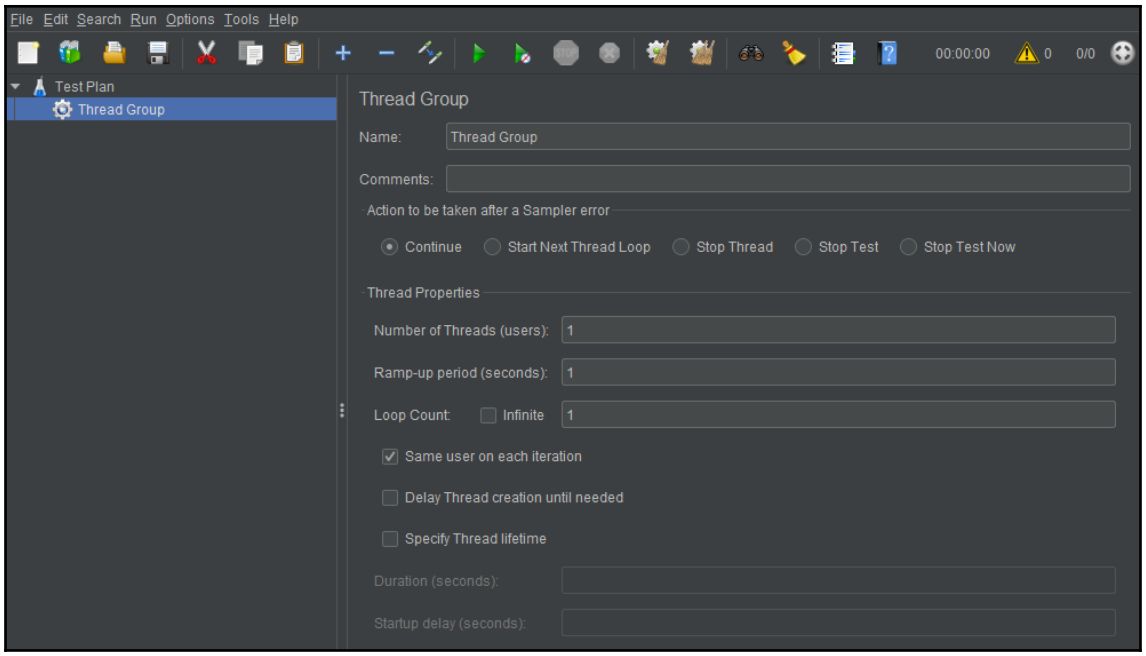

Figure 10.16. Thread Group default values

3. We will set up each value in **Thread Group** as follows:

- **Name**: `PostgreSQL Users`
- **Number of Threads (users)**: `50`
- **Ramp-up period (seconds)**: `10`
- **Loop Count**: `100`
- To have JMeter repeatedly run your test plan, select the **Infinite** checkbox.

The **Ramp-up period** value tells JMeter how long to delay between starting each user. Now our ramp-up period is 10 seconds, JMeter has to start all of your users in 10 seconds. In this case, the number of users is 50, so the delay between starting users is equal to 10 seconds / 50 users = 0.2 seconds per user:

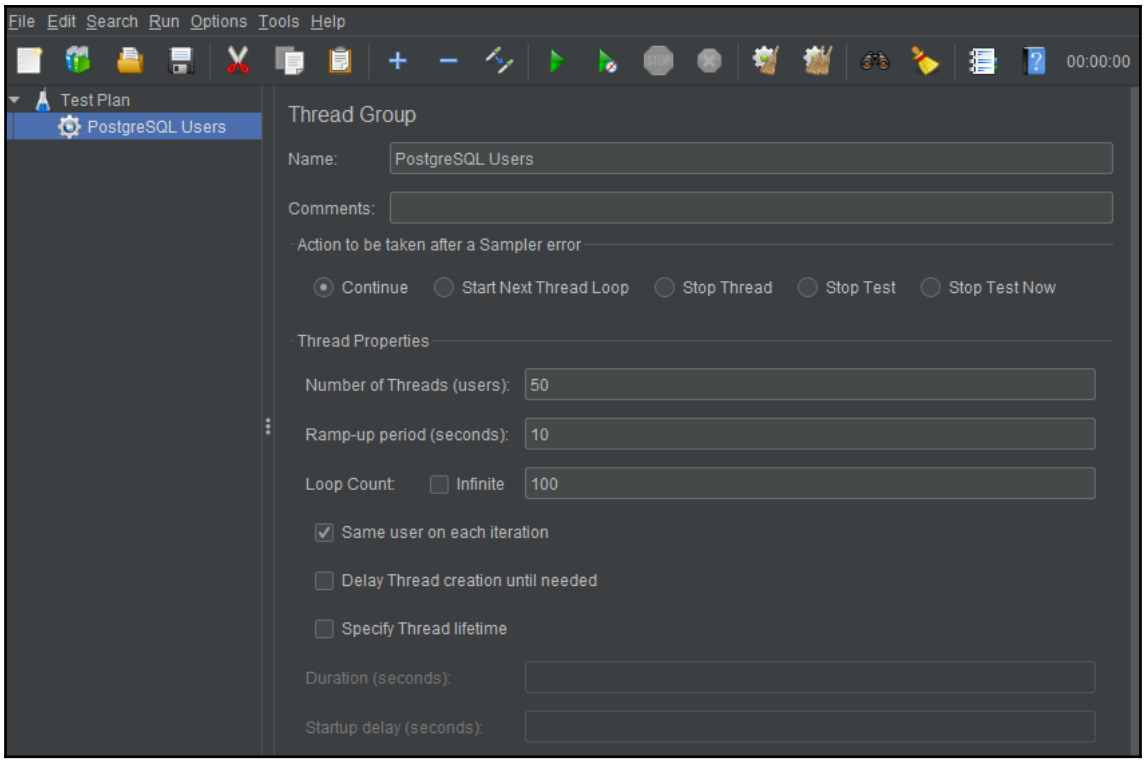

Figure 10.17. Number of users and Ramp-up period

4. We will add a random variable to generate a random value following the next steps:

   First, right-click to get the contextual **Add** menu, and then select **Add** > **Config Element** > **Random Variable**. Then, select this new element to view its Control Panel:

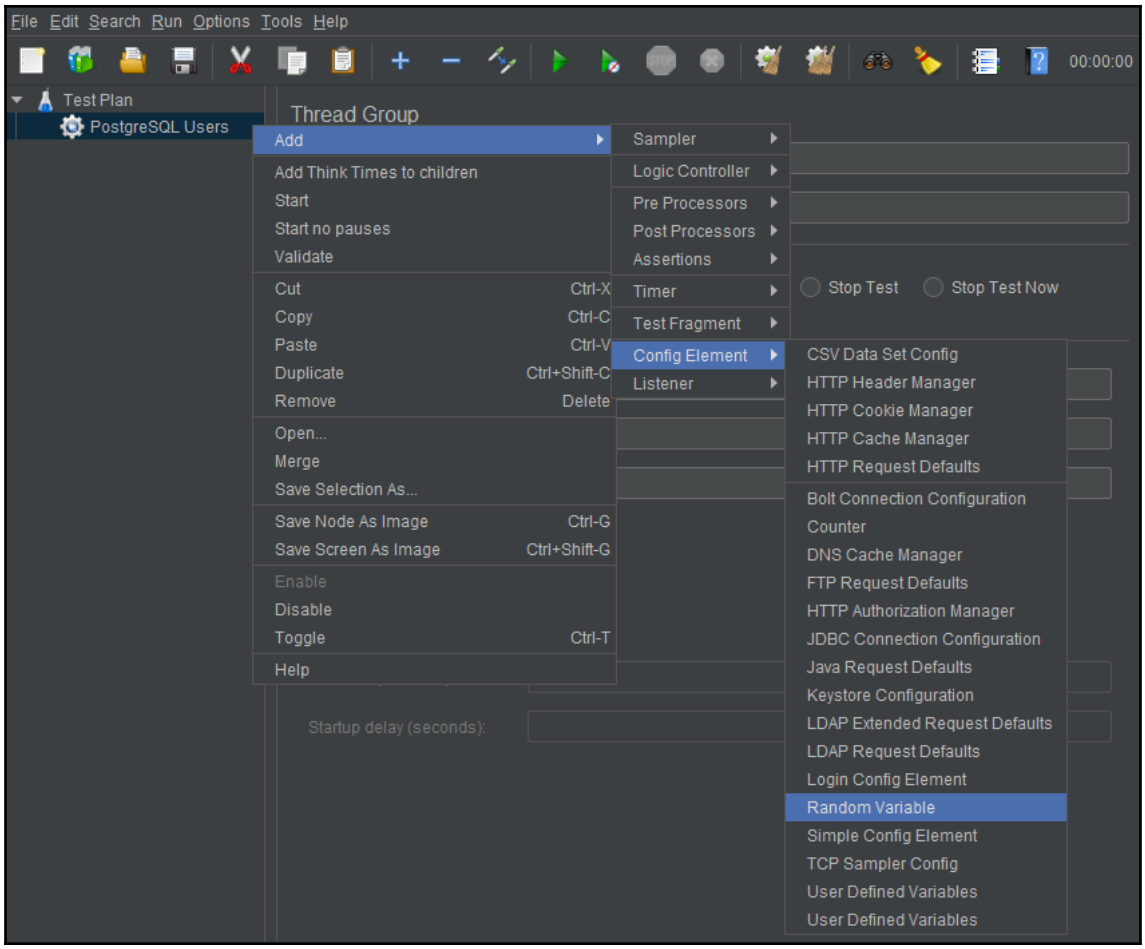

Figure 10.18. Adding a JMeter random variable

5. Once the variable is created, we will set up values for our random variable:

- **Name**: `Random ZipCode`
- **Variable Name**: `zip`
- **Minimum Value**: `10000`
- **Maximum Value**: `11011`

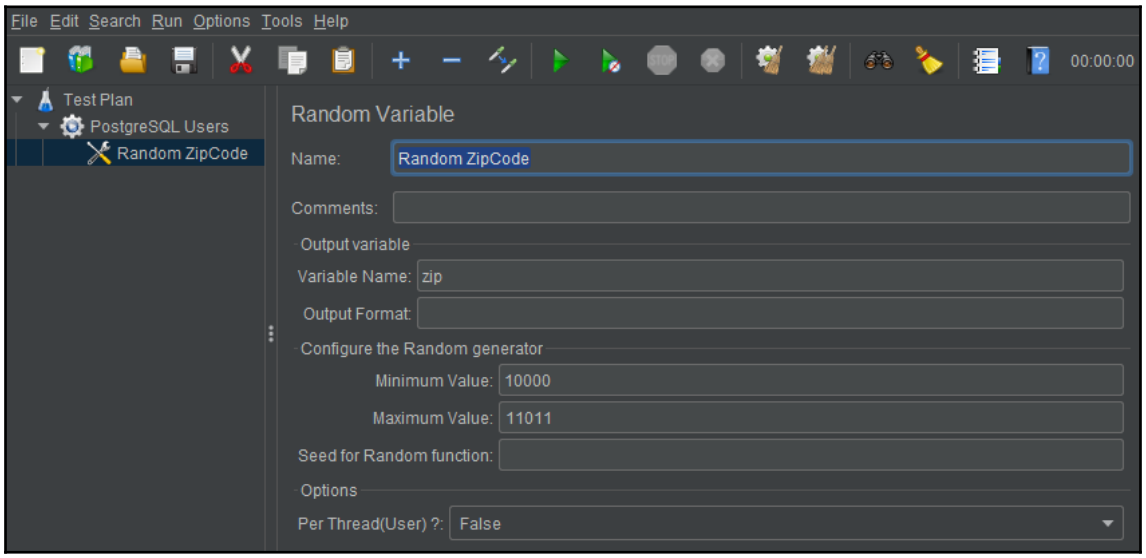

Figure 10.19. Random variable setup

6. We will set up a connection to the RDS on AWS.

   Right-click to get the **Add** menu, and then select **Add** > **Config Element** > **JDBC Connection Configuration.** Then, select this new element to view its Control Panel:

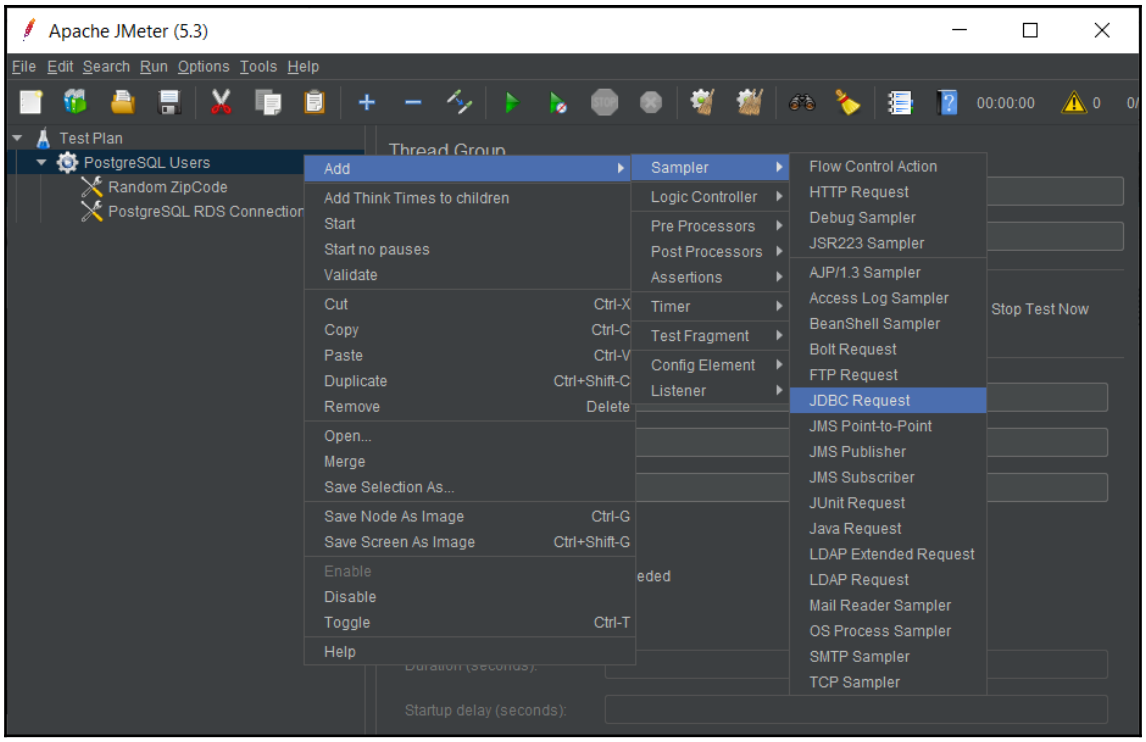

Figure 10.20. Adding a PostgreSQL connection

7. We will add values for the PostgreSQL connection:

- **Name**: `PostgreSQL RDS Connection`
- **Variable Name for created pool**: `ATM Database`
- **Database URL**: `jdbc:postgresql://atm.ck5074bwbilj.us-east-1.rds.amazonaws.com:5432/atm`

- **JDBC Driver class**: `org.postgresql.Driver`
- **Username**: `dba`
- **Password**: `bookdemo`

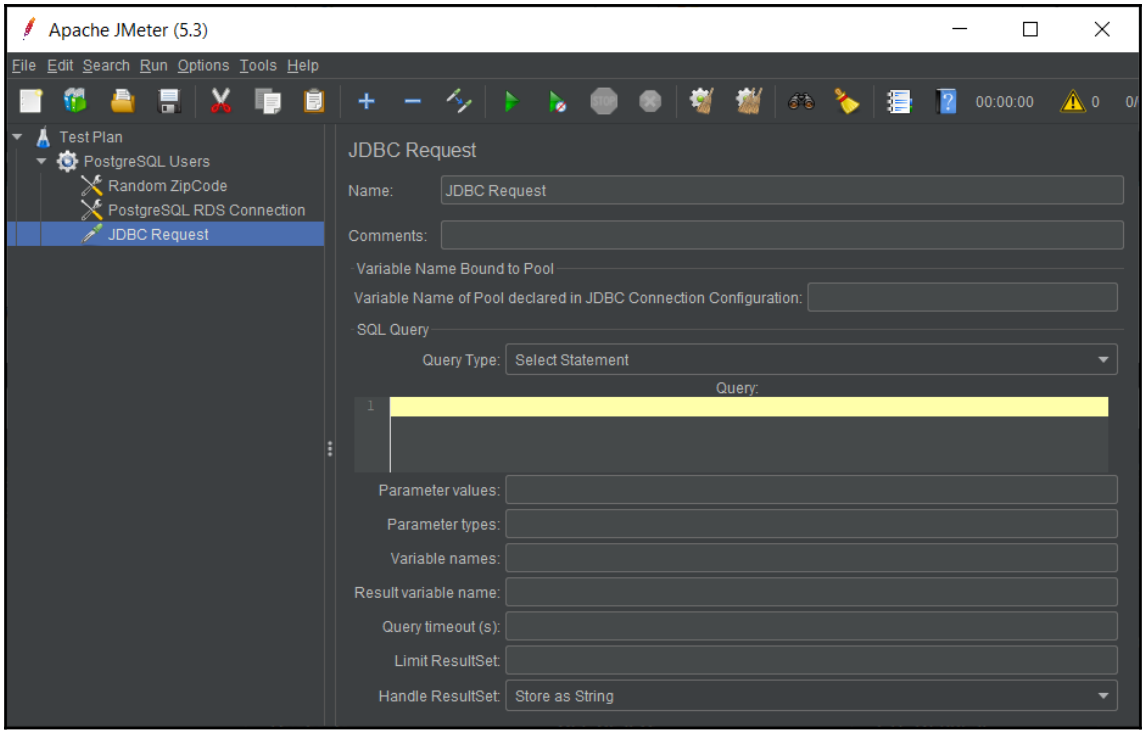

Figure 10.21. Connection details

8. We will add the first JDBC request.

   Click your right mouse button to get the **Add** menu, and then select **Add** > **Sampler** > **JDBC Request**. Then, select this new element to view its Control Panel:

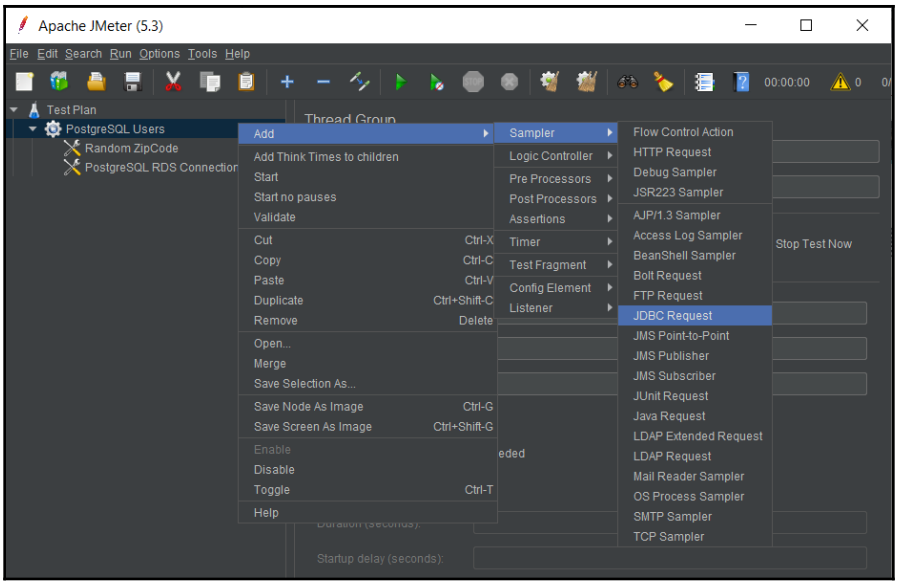

Figure 10.22. Adding a JDBC request

9. The **JDBC Request** form looks like this:

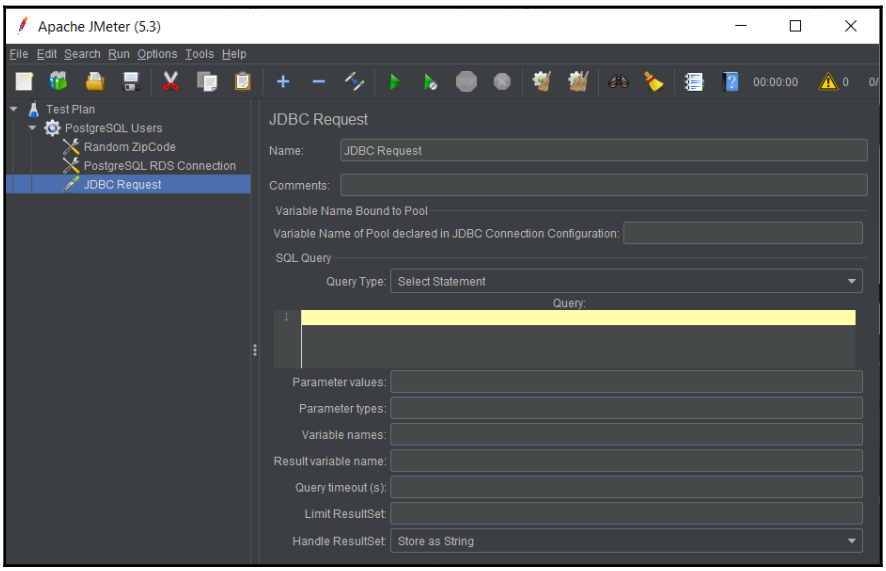

Figure 10.23. JDBC Request form

10. We will add our first `SELECT` query for the "ATM locations" table inside the RDS:

    - Change **Name** to `ATM locations`.
    - Enter the pool name: `ATM Database`.
    - Enter the **SQL Query** string field: `SELECT "BankName", "Address", "County", "City", "State" FROM "ATM locations" WHERE "ZipCode" = ${zip};`

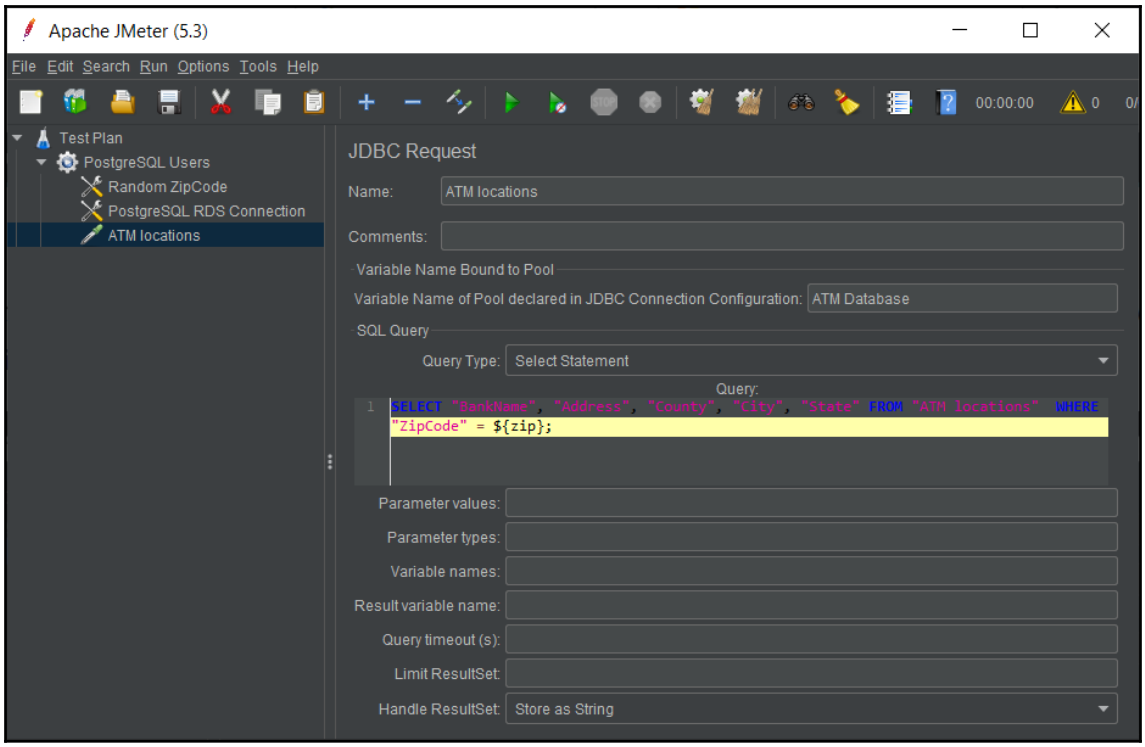

Figure 10.24. SELECT request

Our SELECT statement applies our random variable named **zip** that we set up in step 5.

11. Next, add the second JDBC request and edit the following properties:
    1. Change **Name** to `Zip coordinates`.
    2. Enter the pool name: `ATM Database`.
    3. Enter the **SQL Query** string field: `SELECT "city", "state", "geog" FROM "Zip coordinates" WHERE "zip" = ${zip};`

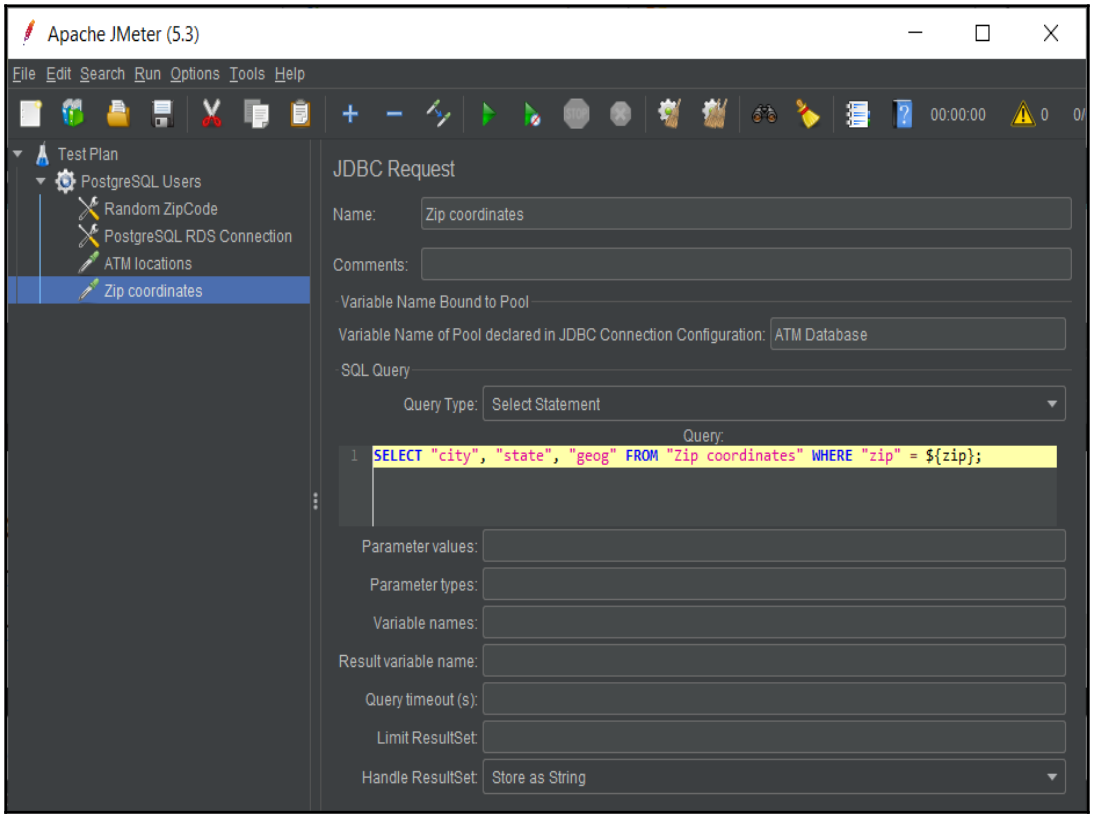

Figure 10.25. Second JDBC request

12. We will add two listeners to view/store the load test results:

    First, we will add a view results tree (by using the menu **Add** > **Listener** > **View Results Tree**):

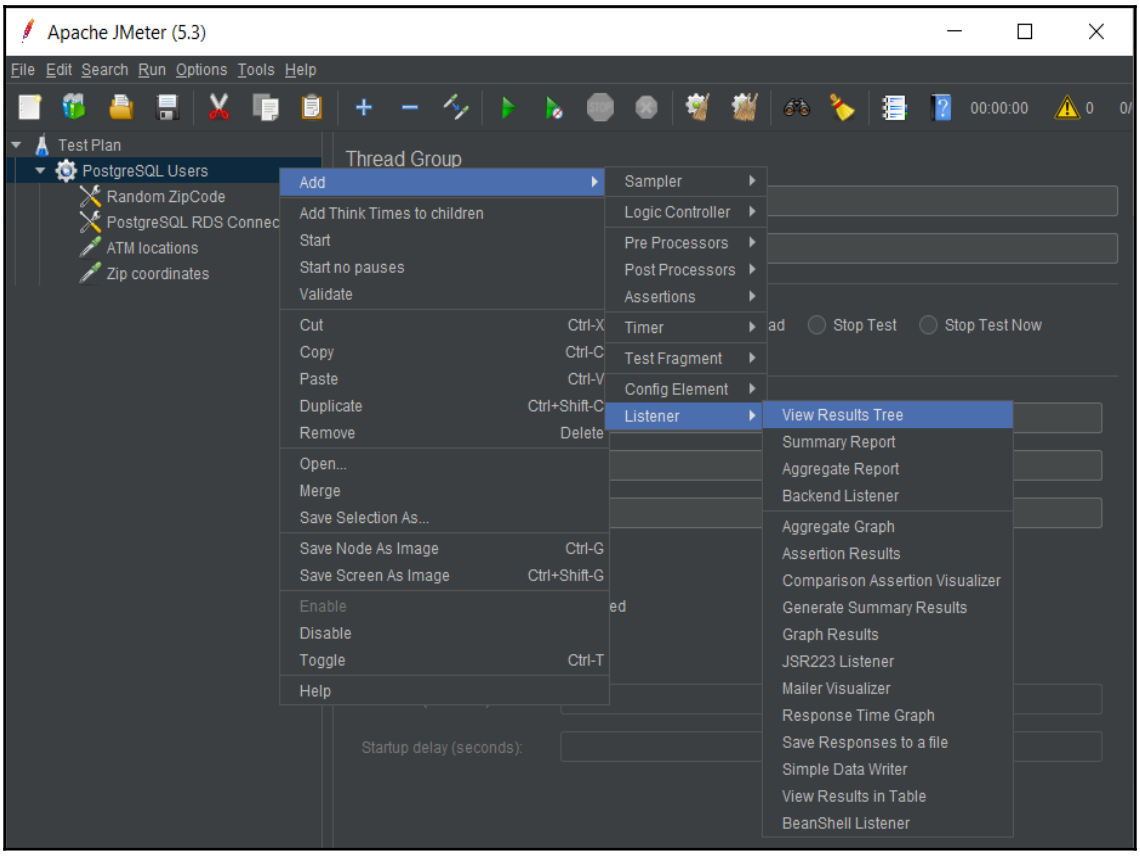

Figure 10.26. The View Results Tree listener

13. Then, add a summary report listener via the **Add** > **Listener** > **Summary Report** menu options:

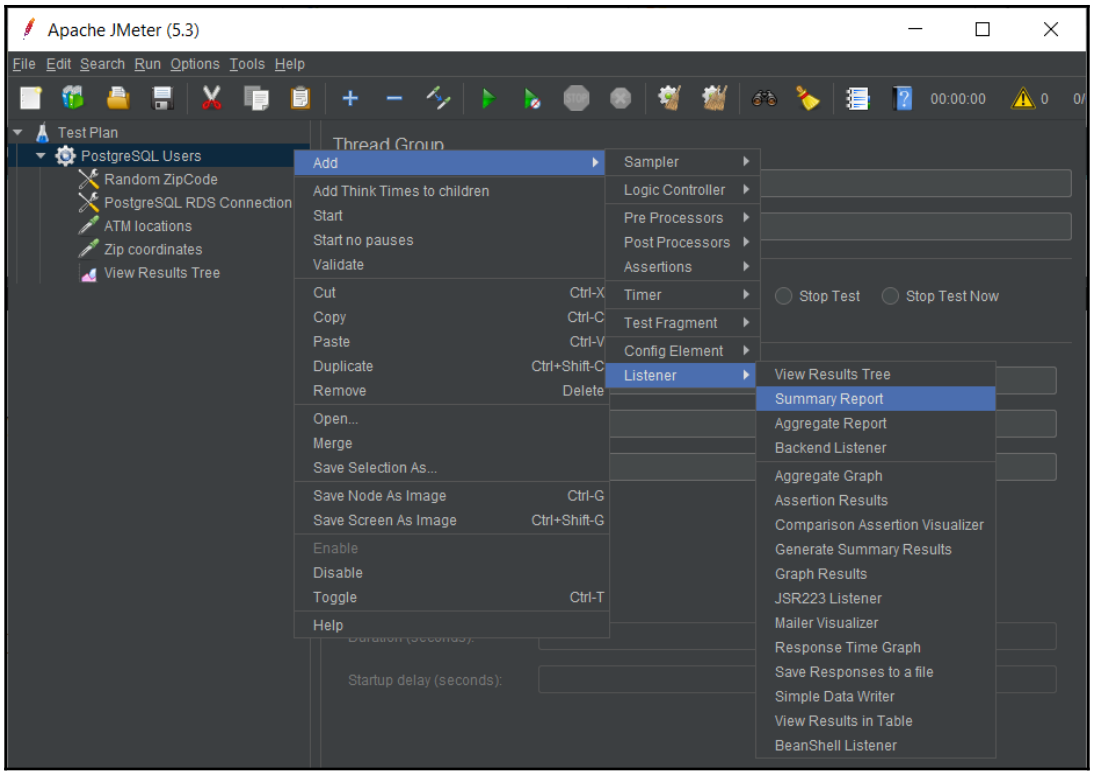

Figure 10.27. The Summary Report listener

14. Save the test plan by going to File -> Save or by clicking on the icon:

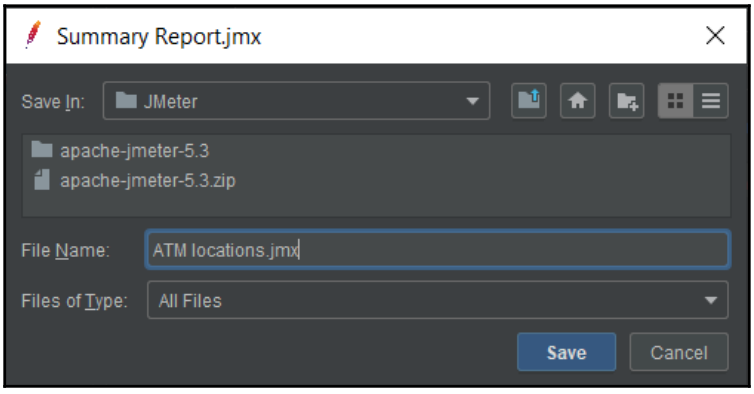

Figure 10.28. Saving the test plan file

15. Run the test with the menu **Run** > **Start** or *Ctrl* + *R*.

1. Select the **View Results Tree** tab.
2. Select the **Response data** tab.
3. Select **Response Body**.
4. Navigate down through each JDBC request.

You will be able to reach the retrieved data from the RDS:

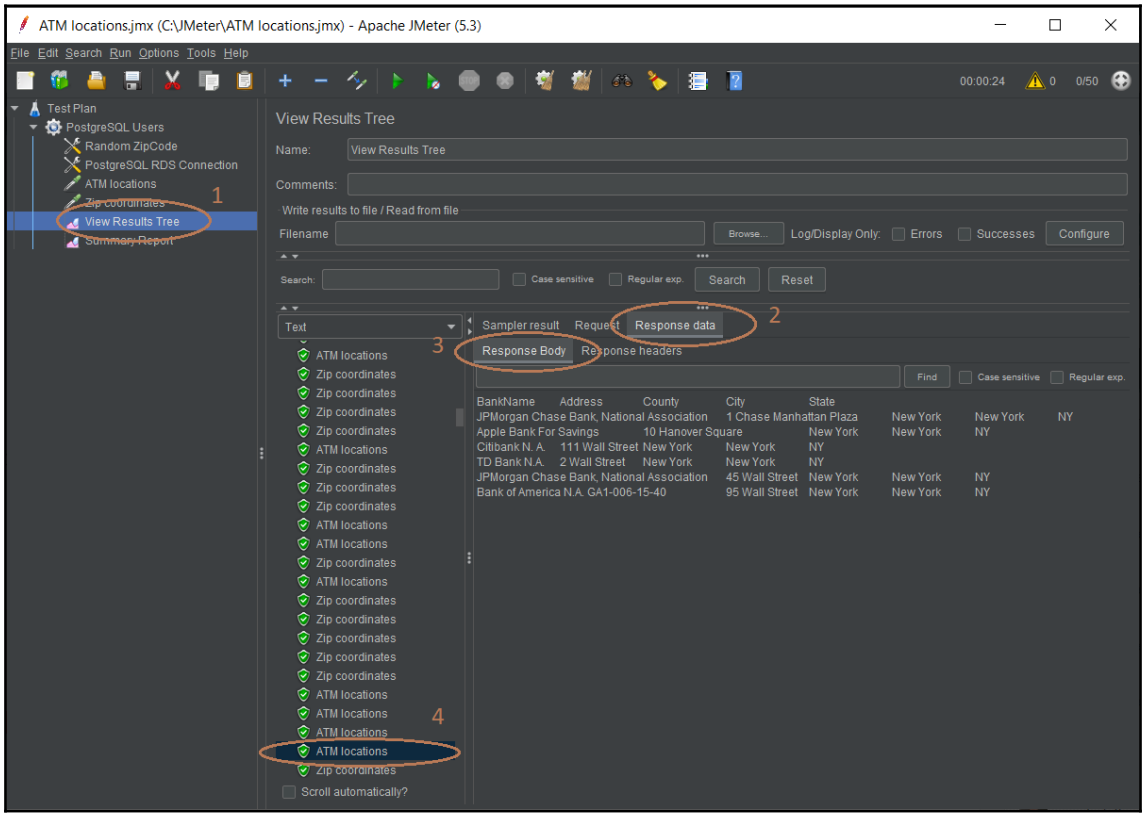

Figure 10.29. Checking each request execution with the View Results Tree listener

16. Select the **Summary Report** tab. You will be able to view the throughput. This time, we reached 404 JDBC requests per second:

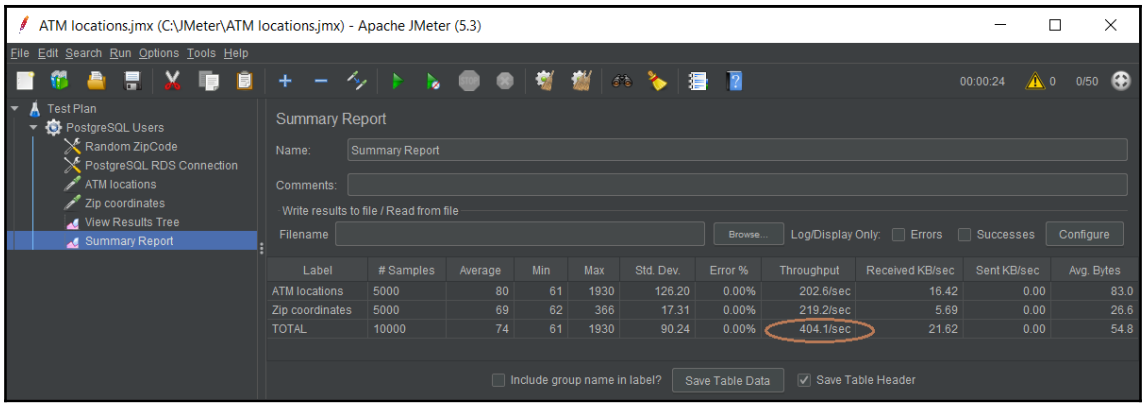

Figure 10.29. Final throughput in the Summary Report listener

JMeter is a good and stable tool. It is easy to learn and to use. Knowing JMeter gives system engineers the ability to complete load test requests for companies all over the world.

Summary

In this chapter, we have learned all about pgbench and how to configure its tests with pgbench parameters. We also learned about Pgbouncer and Pgpool, and then moved on to JMeter and JDBC configuration, threads, and listeners and also how to create JDBC requests, and how to use listener load reports.

In the next chapter, we will learn how to perform unit tests for PostgreSQL databases by using four test frameworks.

11
Test Frameworks for PostgreSQL

In this chapter, you will learn how to write automated tests for existing stored procedures or develop procedures using the concepts of unit tests and **Test-Driven Development** (**TDD**). All test cases are stored in the database, so we do not need any external resources (such as files, version control, and command-line utilities) to save tests. Database tests may be grouped into test cases; each test case may have its own environment initialization code. All tests are still executed independently because their effects are automatically rolled back after the execution.

With the help of the project in this chapter, we will set up and run automated unit tests to test whether possible mistakes exist in our PostgreSQL 12 RDS from **Amazon Web Services** (**AWS**) of Chapter 2, *Setting Up a PostgreSQL RDS for ATM Machines*, of our banking ATM machine locations within a typical city.

The following topics will be covered in the chapter:

- Making unit tests with pgTAP
- Making unit tests in a simpler way with PGUnit
- PGUnit – same name but a different approach
- Testing with Python – Tesgres

Technical requirements

This chapter will take developers around 16-20 hours of working to develop four PostgreSQL test frameworks.

Making unit tests with pgTAP

Through the different topics that we will develop here, we will be applying different techniques to carry out tests. We are going to start with one of the most used techniques in production environments: pgTAP, which has several functions for use case tests and also provides the output in TAP. The **Test Anything Protocol** (**TAP**) is well known for being suitable for harvesting, analysis, and reporting with a TAP harness.

The official website of pgTAP is `https://pgtap.org/`.

Setting up pgTAP for PostgreSQL RDS

We'll set up pgTAP using the following RDS:

1. Use pgAdmin to connect to our ATM RDS on AWS and select the ATM database to set up pgTAP.
2. Then, navigate to the top menu bar, go to **Tools** | **Query Tool**, and then execute the following SQL statements by pressing the  icon on the toolbar, as in the following figure, *Figure 11.1*:

```
CREATE SCHEMA IF NOT EXISTS pgTAP;
CREATE EXTENSION IF NOT EXISTS pgTAP WITH SCHEMA pgTAP;
```

This is better illustrated here:

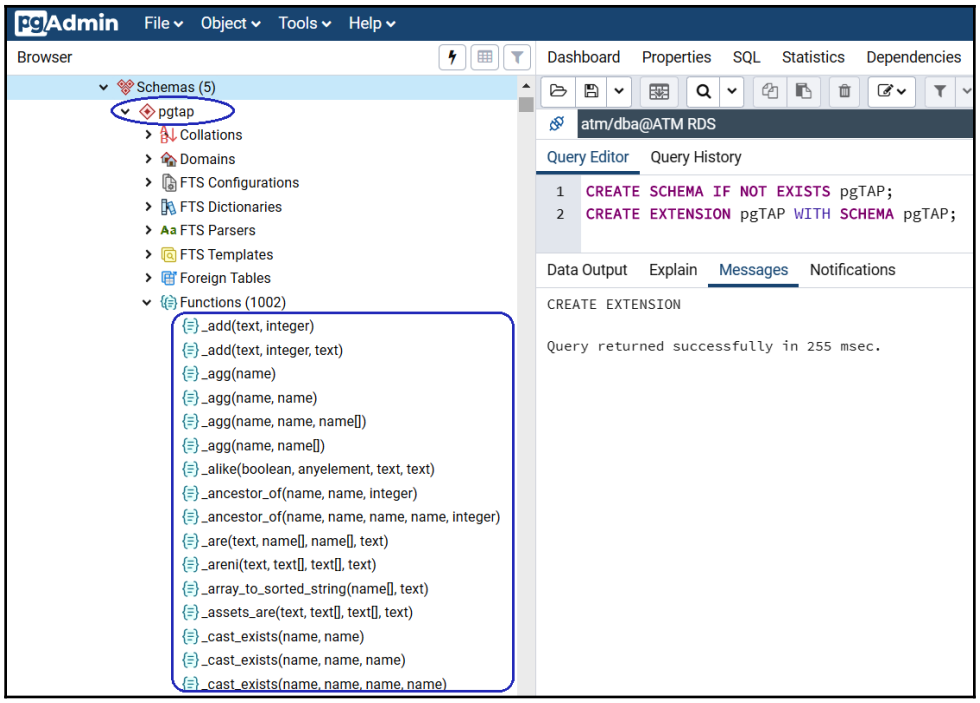

Figure 11.1 – pgTAP installation for AWS RDS

3. After that, you should be able to see the new pgTAP schema and the pgTAP stored procedures, as you can see in the preceding figure.

pgTAP test examples

Before anything else, you need a testing plan. This basically declares how many tests your script is going to run to protect against premature failure:

1. The preferred way to do this is to declare a plan by calling the `plan ()` function for tests; if you only have one test, then you will call `SELECT plan(1)`, but if you intend to implement *n* tests, you will declare `SELECT plan(n)`:

```
SET search_path TO pgTAP;

SELECT plan(1);

SELECT is_empty('select "ID" from public."ATM locations" where
"ZipCode" NOT IN (select "zip" from public."Zip coordinates") LIMIT
```

```
1;', 'Check if there are any Bank ZipCodes not included in Zip
coordinates table');

SELECT * FROM finish();
```

2. The preceding SQL code is a test to see whether there are any bank ZIP codes inside the ATM locations table that are not found in the ZIP coordinates table, hence we execute that code from pgAdmin:

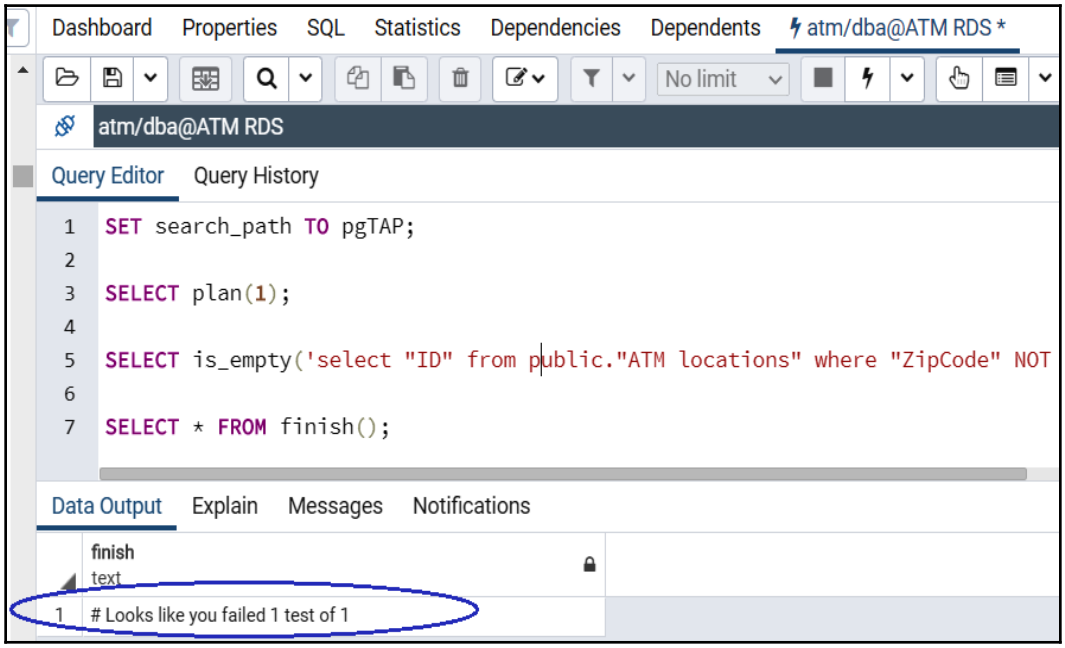

Figure 11.2 – pgTAP test 1

3. The output shows that the test fails because the first ATM has `ZipCode = 1000` but this value cannot be found in the ZIP coordinates table; the failure is `# Looks like you failed 1 test of 1`.
4. Now we test the second error, because the ATM database is used to store ATM locations of New York City, hence the ATM locations table must not be empty, so we have the following code for the next test:

```
SET search_path TO pgTAP;

SELECT plan(1);
```

```
SELECT isnt_empty('select distinct "ID" FROM public."ATM
locations";', 'Check if the ATM locations inside the ATM database
are not empty');

SELECT * FROM finish();
```

5. We now execute the second test with pgAdmin:

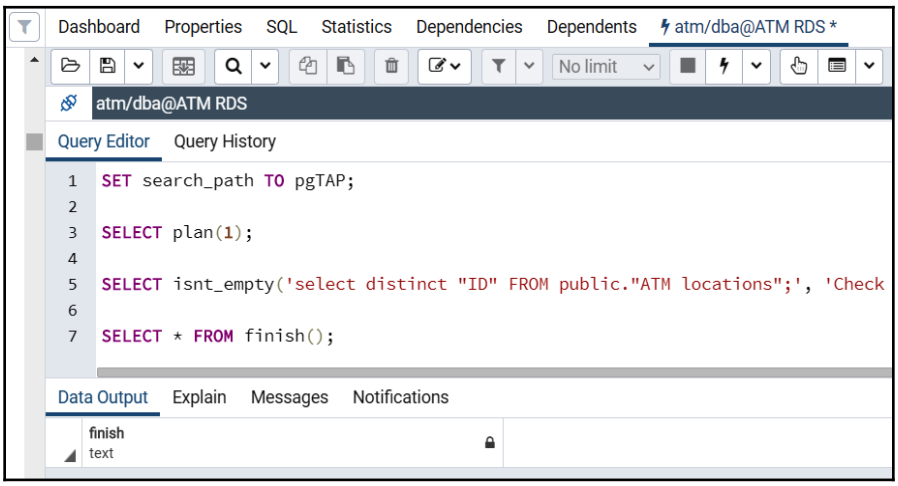

Figure 11.3 – pgTAP test 2

6. Our RDS passes the second test OK, which means our ATM locations table has 654 records and it is not empty.
7. Now, we are going to combine the two preceding tests together into one plan. So, our code will be designed for two tests with `SELECT plan(2)`, as follows:

```
SET search_path TO pgTAP;

SELECT plan(2);

SELECT is_empty('select "ID" from public."ATM locations" where
"ZipCode" NOT IN (select "zip" from public."Zip coordinates") LIMIT
1;', 'Check if there are any Bank ZipCodes not included in Zip
coordinates table');

SELECT isnt_empty('select distinct "ID" FROM public."ATM
locations";', 'Check if the ATM locations inside the ATM database
are not empty');

SELECT * FROM finish();
```

8. Hence, when we execute the two-test plan with pgAdmin, the result will show `# Looks like you failed 1 test of 2`:

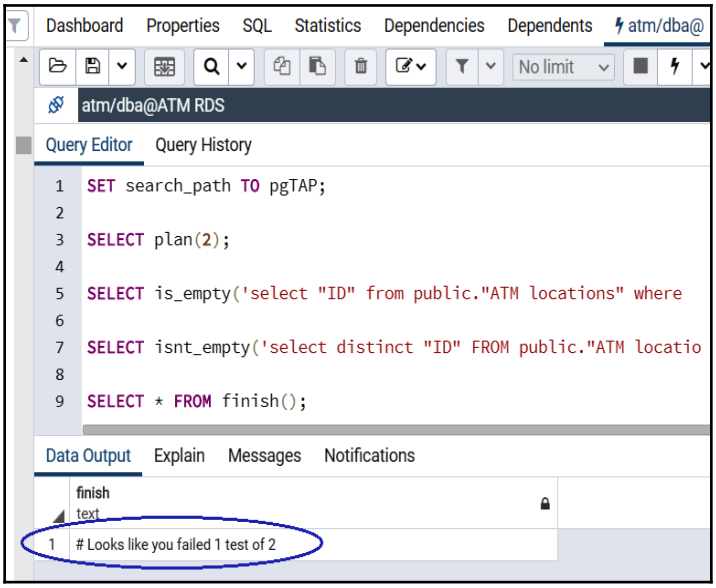

Figure 11.4 – The final pgTAP test plan

Uninstalling pgTAP for PostgreSQL RDS

In order to uninstall pgTAP, please use pgAdmin to execute the following two SQL statements:

```
DROP EXTENSION IF EXISTS pgTAP;
DROP SCHEMA IF EXISTS pgTAP;
```

Up to this point, we have seen one of the most important and updated test tools for PostgreSQL. It is easy to install and is available on different platforms such as Linux or Windows.

One of its greatest strengths lies in that TAP output is easily customizable to use in tools such as Jenkins for continuous integration.

Making unit tests in a simple way with PG_Unit

In the previous section, we have seen a tool that, thanks to its regular updates, has many functions to apply in test cases. Now we will continue with a tool that is unfortunately not so often updated but that, thanks to standardization and its simplicity, does not lose its validity: PG_Unit. We will see that when applying it in pgAdmin, we must make some fixes to the installation script but apart from that, we will be able to demonstrate a test case and thus have another option when carrying out test cases.

The website for PGUnit is here: `https://github.com/danblack/pgunit-sql/`.

Setting up PGUnit for PostgreSQL RDS

We'll start setting up pgunit for PostgreSQL using the following steps:

1. The SQL script to set up PGUnit is found here: `https://github.com/danblack/pgunit-sql/blob/master/sql/reinstall.sql`. Please open the preceding link in your browser. You will see the script shown in *Figure 11.5*:

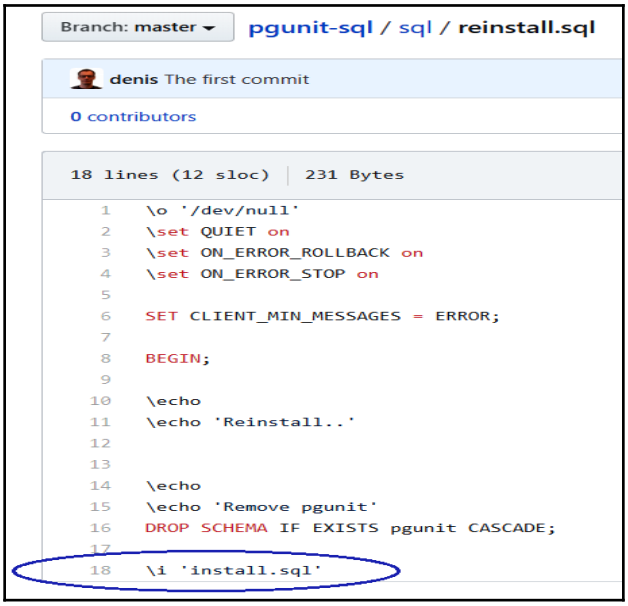

Figure 11.5 – PGUnit installation script

2. The `reinstall.sql` installation script calls to execute another `install.sql` file and in turn, the `install.sql` script calls to many other scrips, such as `assert_array_equals.sp.sql`, `assert_equals.sp.sql`, and `assert_false.sp.sql`, but we are not using `psql` as these scripts are intending:

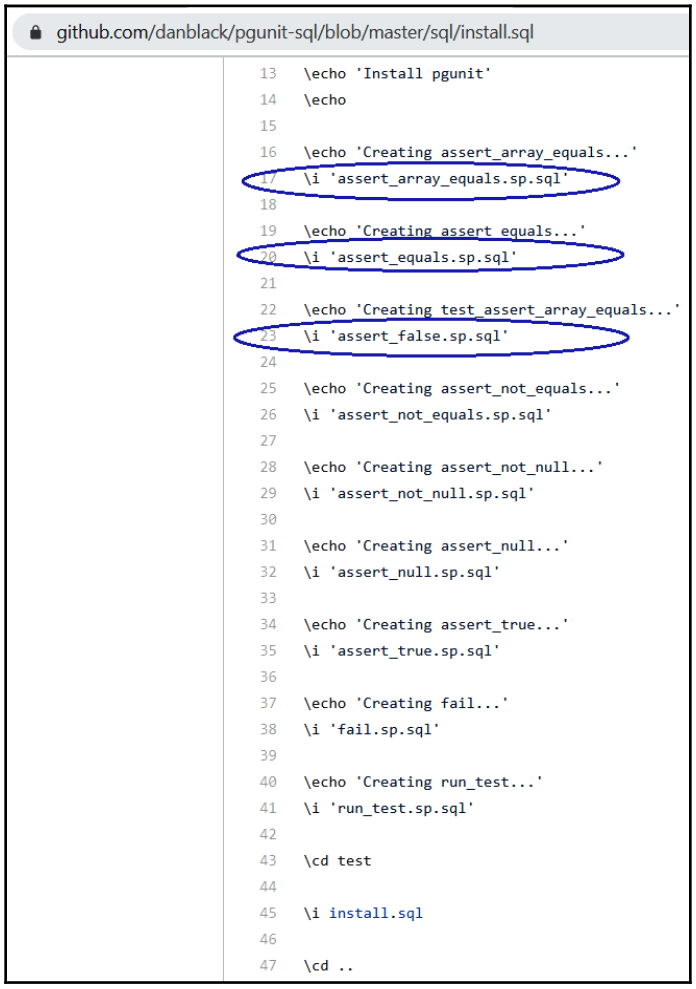

Figure 11.6 – The inner install.sql script

3. Because we are using pgAdmin for the AWS PostgreSQL RDS and we are not using `psql` command-line scripts, we have to comment out `\o` and `\echo` for command-line `psql` command usages and then we have to copy thee sub-files' scripts to replace in the `reinstall.sql` script. Also, the following three statements need commenting out:

- `\set QUIET on`
- `\set ON_ERROR_ROLLBACK on`
- `\set ON_ERROR_STOP on`

4. We copy and execute the script in pgAdmin as in the following screenshot:

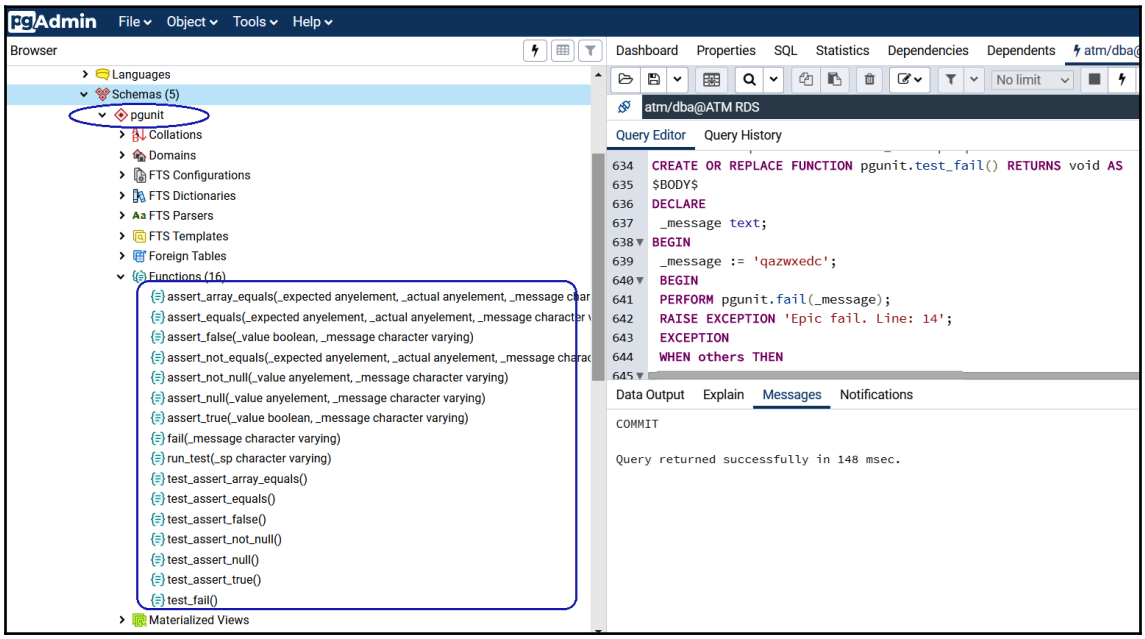

Figure 11.7 – Completion of PGUnit installation

5. The library is based on stored procedures that compare variables between themselves or compare variables with various constants:

- `pgunit. assert_equals (_expected anyelement, actual anyelement, custom_message varchar)`: Compares two elements. If they are not equal, then the `#assert_equalsn {custom_message}` exception is thrown.

- `pgunit. assert_not_equals (_expected anyelement, actual anyelement, custom_message varchar)`: Compares two elements. If they are equal, an exception is thrown: `#assert_not_equalsn {custom_message}`.
- `pgunit. assert_array_equals (_expected anyelement [ ], actual [] anyelement, custom_message varchar)`: Compares two arrays. Arrays are considered equal if these arrays have the same elements and the sizes of the arrays are equal. If the arrays are not equal, an exception is thrown with the text `#assert_array_equalsn {custom_message}`.
- `pgunit. assert_true (_value boolean, _custom_message varchar)`: Compares `_value` to `True`. If they are not equal, a `#assert_truen {custom_message}` exception is thrown.
- `pgunit. assert_false (_value boolean, _custom_message varchar)`: Compares `_value` to `False`. If they are not equal, an exception is thrown: `#assert_falsen {custom_message}`.
- `pgunit. assert_null (_value boolean, _custom_message varchar)`: Compares `_value` to `NULL`. If they are not equal, a `#assert_nulln {custom_message}` exception is thrown.
- `pgunit. assert_not_null (_value boolean, _custom_message varchar)`:
  Compares `_value` to `NULL`. If they are equal, an exception is thrown: `#assert_not_nulln {custom_message}`.
- `pgunit. fail (_custom_message varchar)`: Throws an exception with the text `#assert_failn {custom_message}`.
- `pgunit. run_test (_sp varchar)`: Runs the specified stored procedure inside the test infrastructure. After starting the test procedure, data is rolled back.

PGUnit test examples

We'll check out some pgunit test examples as follows:

1. This is the PGUnit script for the same first test of checking whether there are any bank ATM ZIP codes that are not included in the ZIP coordinate table:

```
create or replace function pgunit.__test_bank_zipcode() returns
void as $$
declare
    id INT;
begin
    select "ATM locations"."ID" from "ATM locations" where "ATM
```

```
locations"."ZipCode" NOT IN (select "Zip coordinates".zip from "Zip
coordinates") LIMIT 1 INTO id;
    perform pgunit.assert_null(id, 'Bank ZipCode is not included in
Zip coordinates table');
end;
$$ language plpgsql;
```

2. After creating the first test in pgAdmin, we can execute the following query to run the first test:

```
select * from pgunit.__test_bank_zipcode();
```

3. Because the first ATM location has `zipcode = 1000`, which does not exist in the ZIP coordinates table, an exception is raised with our defined message: `Bank ZipCode is not included in Zip coordinates table`:

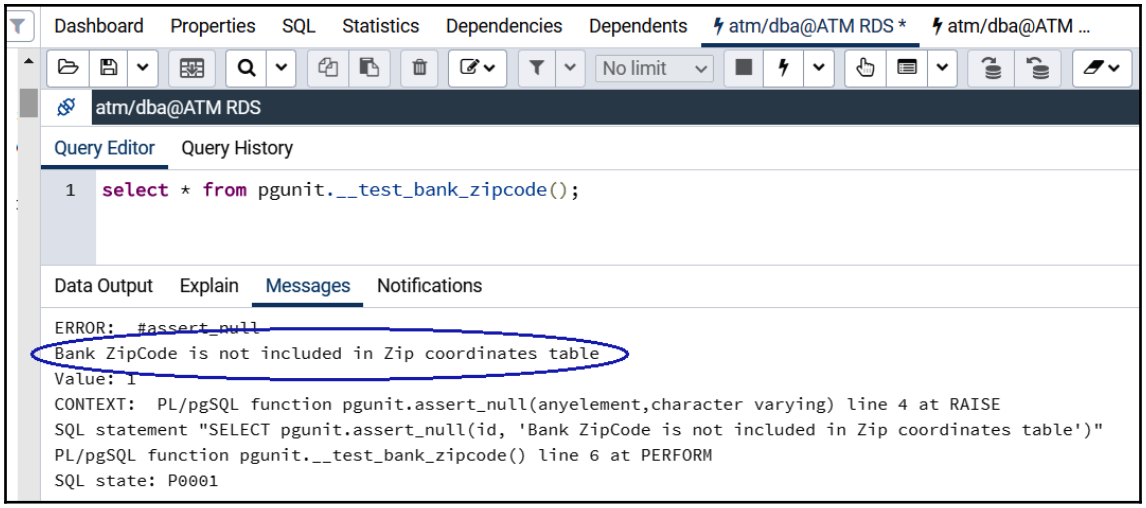

Figure 11.8 – PGUnit test 1

4. This is the PGUnit script for the same second test of checking that the ATM locations should not be empty:

```
create or replace function pgunit.__test_atm_locations() returns
void as $$
declare
    atmtotal INT;
begin
    SELECT count(distinct "ID") INTO atmtotal FROM public."ATM
locations";
```

```
    PERFORM pgunit.assert_true ((atmtotal IS NOT NULL) AND
(atmtotal > 0), 'There are no ATM locations inside the ATM
database');
end;
$$ language plpgsql;
```

5. After creating the second test in pgAdmin, we can execute the following query to run the second test:

```
select * from pgunit.__test_atm_locations();
```

6. Because we already stored 654 ATM locations, the second test should pass alright with no exceptions:

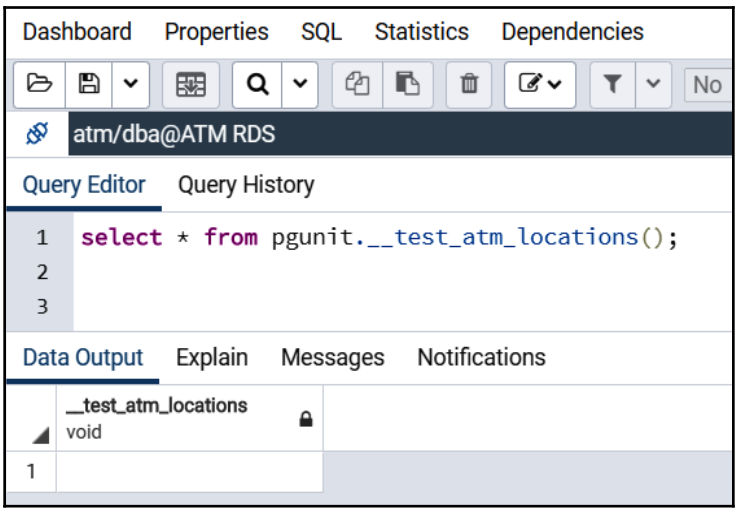

Figure 11.9 – PGUnit test 2

Uninstalling PGUnit for PostgreSQL RDS

In order to uninstall PGUnit, please use pgAdmin to execute the following SQL statement:

```
DROP SCHEMA IF EXISTS pgunit CASCADE;
//
```

As we have seen, PG_Unit is a simple tool for PostgreSQL tests; while it's not full of features for testing, it works in a modest way and we have verified it in the preceding sections.

PGUnit – same name but a different approach

When we decided to apply unit tests in PostgreSQL, we noticed there were two tools with the same name: PGUnit, but when we tried them, we realized that they share the fact of simplifying the tests but, unlike the previous one, the pgunit we are using has more recent updates. In the following steps, we will take care of applying it to our project and evaluating its behavior.

The website for simple pgunit is found here: `https://github.com/adrianandrei-ca/pgunit`.

Setting up simple pgunit for PostgreSQL RDS

1. The `plpgsql` code depends on the `dblink` extension being present in the database you run the tests on. We set up a simple pgunit in a dedicated schema such as `pgunit` and run these two lines of SQL:

```
CREATE SCHEMA pgunit;
CREATE EXTENSION DBLINK SCHEMA pgunit;
```

2. You should run the `PGUnit.sql` code using pgAdmin: `https://github.com/adrianandrei-ca/pgunit/blob/master/PGUnit.sql`.

3. We will copy the code from GitHub:

339 lines (317 sloc) | 11 KB

```
1
2  create type test_results as (
3    test_name varchar,
4    successful boolean,
5    failed boolean,
6    erroneous boolean,
7    error_message varchar,
8    duration interval);
9
10 --
11 -- Use select * from test_run_all() to execute all test cases
12 --
13 create or replace function test_run_all() returns setof test_results as $$
14 begin
15   return query select * from test_run_suite(NULL);
16 end;
17 $$ language plpgsql set search_path from current;
18
19 --
20 -- Executes all test cases part of a suite and returns the test results.
21 --
22 -- Each test case will have a setup procedure run first, then a precondition,
23 -- then the test itself, followed by a postcondition and a tear down.
24 --
25 -- The test case stored procedure name has to match 'test_case_<p_suite>%' patern.
26 -- It is assumed the setup and precondition procedures are in the same schema as
27 -- the test stored procedure.
28 --
29 -- select * from test_run_suite('my_test'); will run all tests that will have
30 -- 'test_case_my_test' prefix.
31 create or replace function test_run_suite(p_suite TEXT) returns setof test_results as $$
32 declare
33   l_proc RECORD;
34   l_sid INTEGER;
```

Figure 11.10 – Simple pgunit installation script

4. A convenient way to install the simple pgunit suite in our dedicated `pgunit` schema is to temporarily change the search path like this:

```
SET search_path TO pgunit;
```

We then revert `search_path` back to the public schema after the GitHub script.

5. Thereafter, please execute the whole of the script together in pgAdmin.

6. The result of a simple pgunit should be as follows:

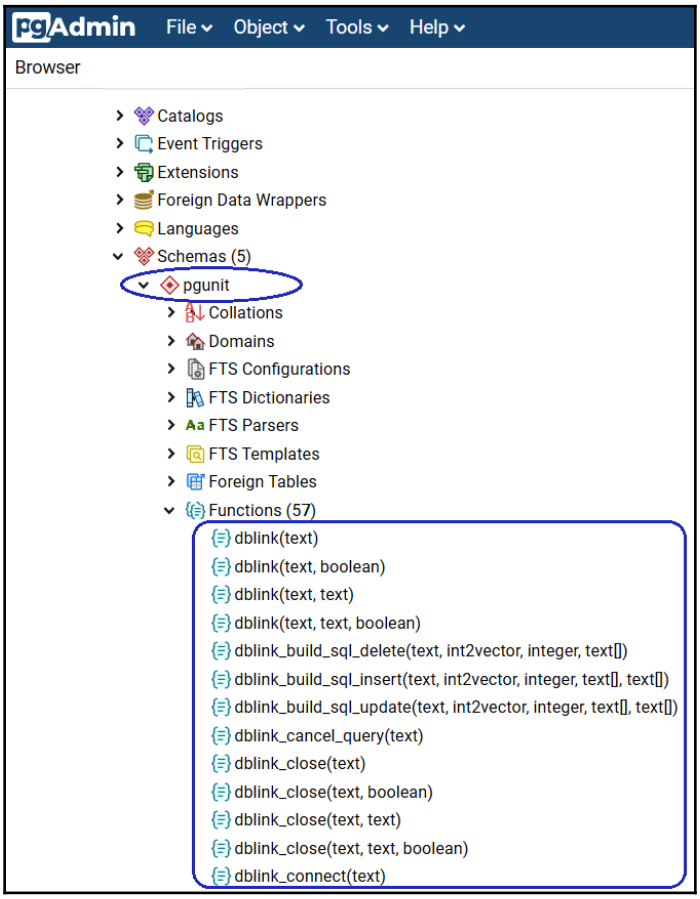

Figure 11.11 – Completion of the simple pgunit installation

Here is a list of prefixes for all tests:

- `"test_case_"`: It is a unit test procedure.
- `"test_precondition_"`: It is a test precondition function.
- `"test_postcondition_"`: It is a test postcondition function.
- `"test_setup_"`: It is a test setup procedure.
- `"test_teardown_"`: It is a test tear-down procedure.

Simple pgunit test examples

We'll now check out some pgunit test examples as follows:

1. This is the first test script to test whether there are any bank ATM ZIP codes not included in the ZIP coordinates table:

```
create or replace function pgunit.test_case_bank_zipcode() returns
void as $$
declare
    id INT;
begin
    select "ATM locations"."ID" from "ATM locations" where "ATM
locations"."ZipCode" NOT IN (select "Zip coordinates".zip from "Zip
coordinates") LIMIT 1 INTO id;
    perform pgunit.test_assertNull('Bank ZipCode is not included in
Zip coordinates table', id); end;
$$ language plpgsql;
```

2. After creating the first test by pgAdmin, please execute this query to proceed with the first test:

```
select * from pgunit.test_case_bank_zipcode();
```

3. The result is as follows:

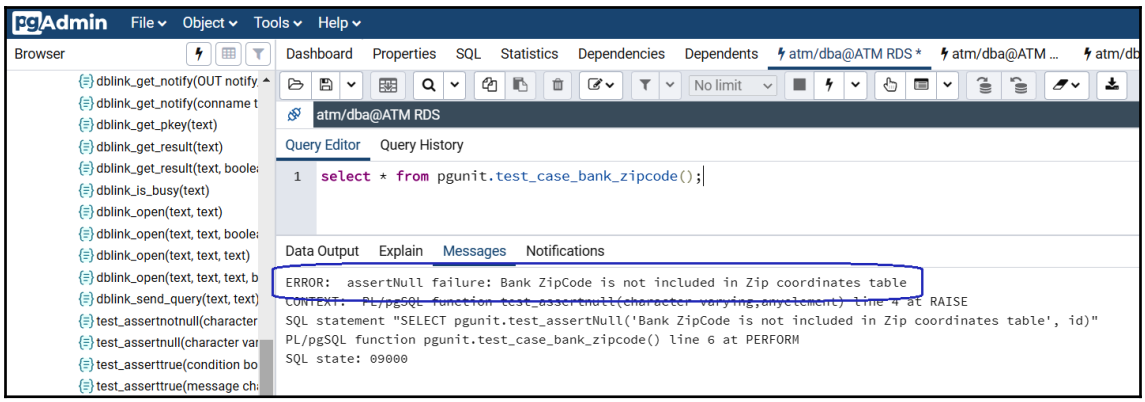

Figure 11.12 – Simple pgunit test 1

The first bank ATM location has `zipcode = 1000` but as this value, `1000`, does not exist in the ZIP coordinates table, the `Bank ZipCode is not included in Zip coordinates table` exception is raised.

4. The second test to make sure that the ATM locations table is not empty is a precondition function:

```
create or replace function pgunit.test_precondition_atm_locations()
returns void as $$
declare
    atmtotal INT;
begin
    SELECT count(distinct "ID") INTO atmtotal FROM public."ATM
locations";
    PERFORM pgunit.test_assertTrue('There are no ATM locations
inside the ATM database', (atmtotal IS NOT NULL) AND (atmtotal > 0)
);
end;
$$ language plpgsql;
```

5. After creating the first test by pgAdmin, please execute this query to proceed with the second test:

```
select * from pgunit.test_precondition_atm_locations();
```

6. The result of the second test is shown:

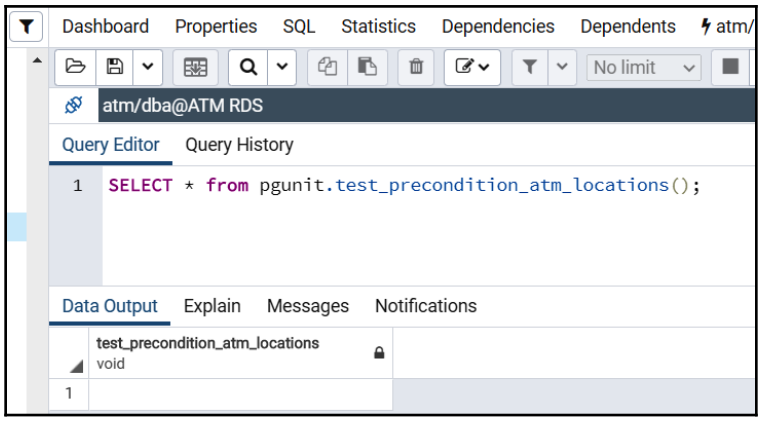

Figure 11.13 – Simple pgunit test 2

There are no exceptions raised because our ATM locations table has 654 records and it is not empty.

For general tests of simple pgunit, such as `select * from pgunit.test_run_all()`, because AWS does not support the `dblink_connect_u` function on RDS for PostgreSQL, these general tests are not supported in AWS RDS.

Uninstalling simple pgunit for PostgreSQL RDS

We can uninstall simple pgunit using the following steps:

1. `PGUnitDrop.sql` has the code you can use to remove all PGUnit code from the database: `https://github.com/adrianandrei-ca/pgunit/blob/master/PGUnitDrop.sql`.

2. We will copy the content from the browser:

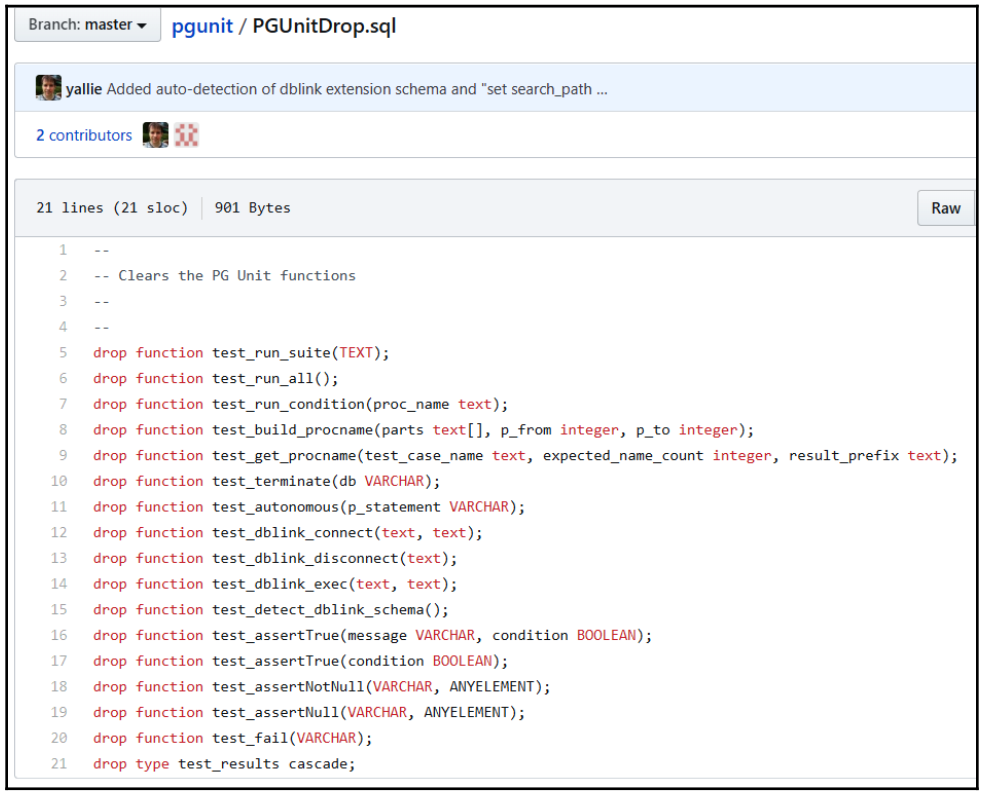

Figure 11.14 – Un-installation of the simple pgunit

3. Because we have installed all of the simple pgunit library inside our separate `pgunit` schema, we modify the preceding script to look like the following:

```
--
-- Clears the PG Unit functions
--
--
drop function pgunit.test_run_suite(TEXT);
drop function pgunit.test_run_all();
drop function pgunit.test_run_condition(proc_name text);
drop function pgunit.test_build_procname(parts text[], p_from
integer, p_to integer);
drop function pgunit.test_get_procname(test_case_name text,
expected_name_count integer, result_prefix text);
drop function pgunit.test_terminate(db VARCHAR);
drop function pgunit.test_autonomous(p_statement VARCHAR);
drop function pgunit.test_dblink_connect(text, text);
drop function pgunit.test_dblink_disconnect(text);
drop function pgunit.test_dblink_exec(text, text);
drop function pgunit.test_detect_dblink_schema();
drop function pgunit.test_assertTrue(message VARCHAR, condition
BOOLEAN);
drop function pgunit.test_assertTrue(condition BOOLEAN);
drop function pgunit.test_assertNotNull(VARCHAR, ANYELEMENT);
drop function pgunit.test_assertNull(VARCHAR, ANYELEMENT);
drop function pgunit.test_fail(VARCHAR);
drop type pgunit.test_results cascade;

DROP EXTENSION IF EXISTS DBLINK;
DROP SCHEMA IF EXISTS pgunit CASCADE;
```

We will execute the preceding script in pgAdmin to uninstall the simple pgunit.

As we have seen, using pgunit is pretty straightforward to install but at times uninstalling it is a bit more complicated. Despite this, we were able to run a case test successfully.

In the next section, we will work with another test approach based on Python.

Testing with Python – Testgres

Testgres was developed under the influence of the Postgres TAP test feature. As an extra feature, it can manage Postgres clusters: initialize, edit configuration files, start/stop the cluster, and execute queries. In the following steps, we will see how to install it, execute a case test, and properly uninstall it.

The website for Testgres is found here: `https://github.com/postgrespro/testgres`.

Setting up Testgres for PostgreSQL

Testgres is a Python test tool, hence we will set up Testgres on our Jenkins Ubuntu server `192.168.0.200` that we set up with Vagrant in Chapter 7, *PostgreSQL with DevOps for Continuous Delivery*, to connect to our RDS on AWS:

1. We open SSH into the Jenkins server. Please launch PowerShell as an administrator:

```
PS C:\Windows\system32>
PS C:\Windows\system32> cd C:\Projects\Vagrant\Jenkins
PS C:\Projects\Vagrant\Jenkins> vagrant up --provider virtualbox
PS C:\Projects\Vagrant\Jenkins> vagrant ssh

vagrant@devopsubuntu1804:~$
```

2. Install `pip3`:

```
vagrant@devopsubuntu1804:~$ sudo apt install -y python3-pip
```

3. Please answer `<Yes>` when you get to the **Package configuration** screen:

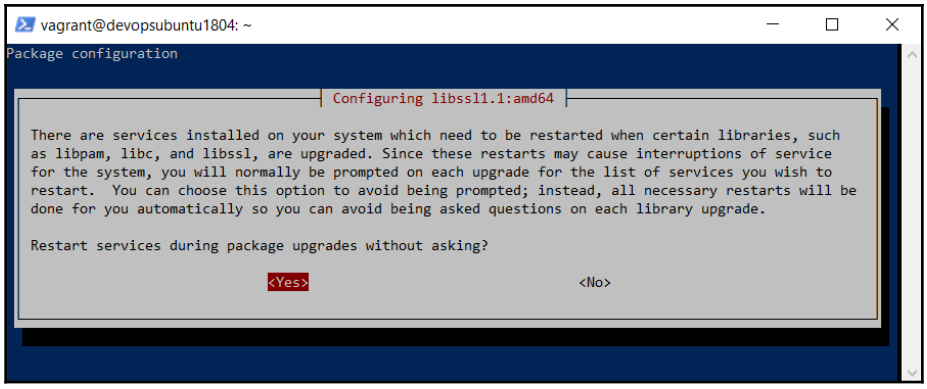

Figure 11.15 – Package configuration screen

4. You can check the version of `pip3`:

```
vagrant@devopsubuntu1804:~$ pip3 -V
pip 9.0.1 from /usr/lib/python3/dist-packages (python 3.6)
```

5. Install Testgres using `pip3`:

```
vagrant@devopsubuntu1804:~$ sudo pip3 install testgres
```

This is a screenshot of the Testgres installation:

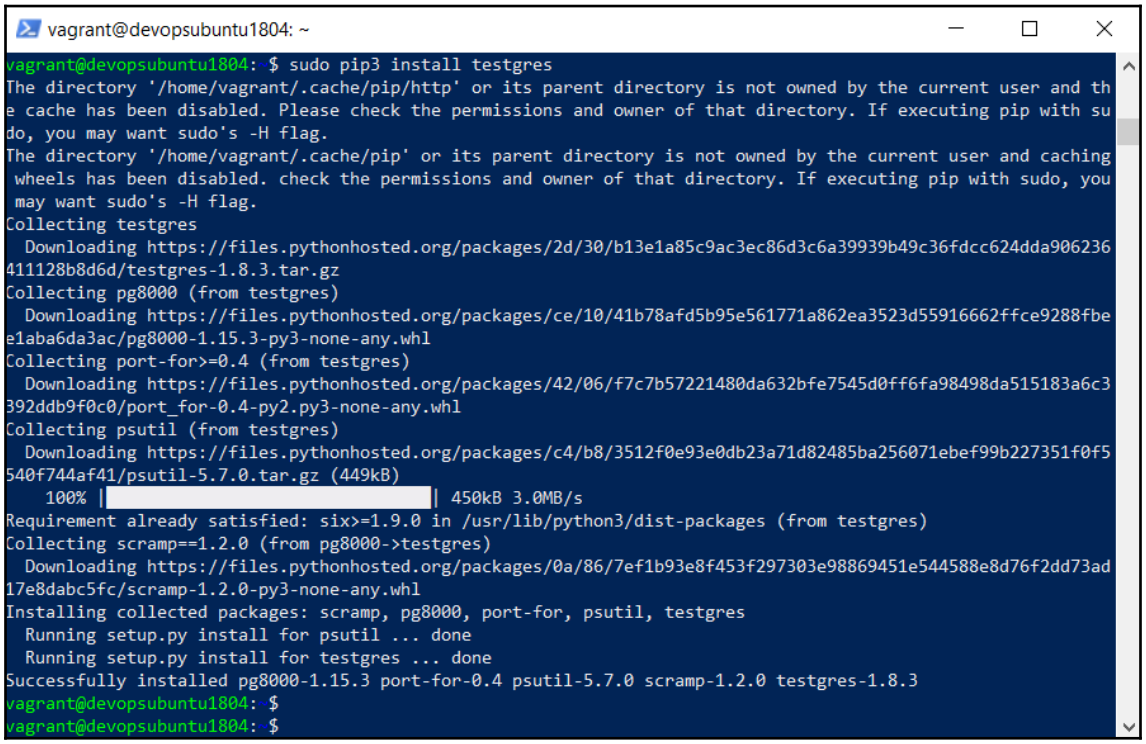

Figure 11.16 – Testgres installation

7. We complete the Testgres installation by setting the `PG_BIN` and `PYTHON_VERSION` environment variables and then storing them:

```
vagrant@devopsubuntu1804:~$ sudo su
root@devopsubuntu1804:~# export PG_BIN=/usr/lib/postgresql/12/bin
root@devopsubuntu1804:~# export PYTHON_VERSION=3
root@devopsubuntu1804:~# source ~/.profile
```

Testgres test examples

We will check out some examples from Testgres:

1. We create the same two tests for our RDS, as follows:

```
root@devopsubuntu1804:~# mkdir /usr/local/src/testgres
root@devopsubuntu1804:~# nano /usr/local/src/testgres/atmrds.py
----------------------------------------------------------------
#!/usr/bin/env python
# coding: utf-8

import unittest
import logging
import testgres

logging.basicConfig(filename='/tmp/testgres.log')

class RdsTest(unittest.TestCase):
    def test_bank_zipcode(self):
        #create a node with random name, port, etc
        with testgres.get_new_node() as node:
           # RDS endpoint
           node.host = 'atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com'
           node.port = 5432

           # run inidb
           node.init()

           with node.connect('atm', 'dba', 'bookdemo') as conn:
               conn.begin('serializable')

               # execute a query in the AWS RDS
               data = conn.execute('select "ATM locations"."ID"
from "ATM locations" where "ATM locations"."ZipCode" NOT IN (select
"Zip coordinates".zip from "Zip coordinates") LIMIT 1;')
               self.assertFalse(data, 'Bank ZipCode is not included
in Zip coordinates table')

    def test_atm_locations(self):
        #create a node with random name, port, etc
        with testgres.get_new_node() as node:
           # RDS endpoint
           node.host = 'atm.ck5074bwbilj.us-
east-1.rds.amazonaws.com'
           node.port = 5432
```

```
            # run inidb
            node.init()

            with node.connect('atm', 'dba', 'bookdemo') as conn:
                conn.begin('serializable')

                # execute a query in the AWS RDS
                data = conn.execute('SELECT count(distinct "ID")
FROM public."ATM locations";')
                self.assertTrue(data, 'There are no ATM locations
inside the ATM database')
                self.assertGreater(data[0][0], 0, 'There are no ATM
locations inside the ATM database')

if __name__ == '__main__':
    unittest.main()
----------------------------------------------------------------------
```

The `test_bank_zipcode` function is the first test, to check whether there are any bank ATM ZIP codes that not included in the ZIP coordinates table, and the second `test_atm_locations` function checks that the ATM locations table is not empty.

2. Please set the `755` permission for the test:

```
root@devopsubuntu1804:~# chmod 755
/usr/local/src/testgres/atmrds.py
```

3. With the correct permission, we can execute the test for our RDS as a Postgres user:

```
root@devopsubuntu1804:~# cd /usr/local/src/testgres/
root@devopsubuntu1804:/usr/local/src/testgres# su postgres -c
'python3 atmrds.py'
.F
======================================================================
===
FAIL: test_bank_zipcode (__main__.RdsTest)
----------------------------------------------------------------------
---
Traceback (most recent call last):
  File "atmrds.py", line 26, in test_bank_zipcode
    self.assertFalse(data, 'Bank ZipCode is not included in Zip
coordinates table')
AssertionError: [(1,)] is not false : Bank ZipCode is not included
in Zip coordinates table
```

```
----------------------------------------------------------------------
---
Ran 2 tests in 1.941s

FAILED (failures=1)
```

We can see this is in the screenshot here:

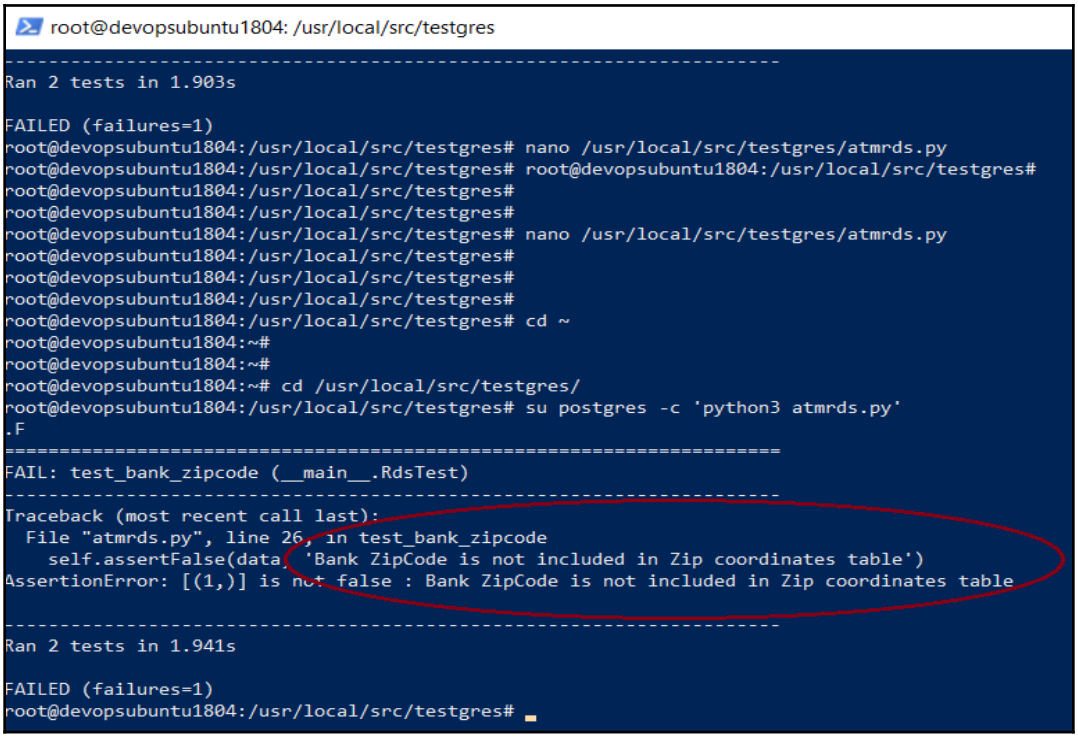

Figure 11.17 – Testgres tests

The two tests return one failure with the exception as `Bank ZipCode is not included in Zip coordinates table`.

5. *Steps 1-4* show how to run tests directly with `python3`. We can improve our testing by using `virtualenv`. Please install `virtualenv`:

```
root@devopsubuntu1804:/usr/local/src/testgres# cd /root
root@devopsubuntu1804:~# pip3 install virtualenv
```

6. A screenshot of the `virtualenv` installation is as follows:

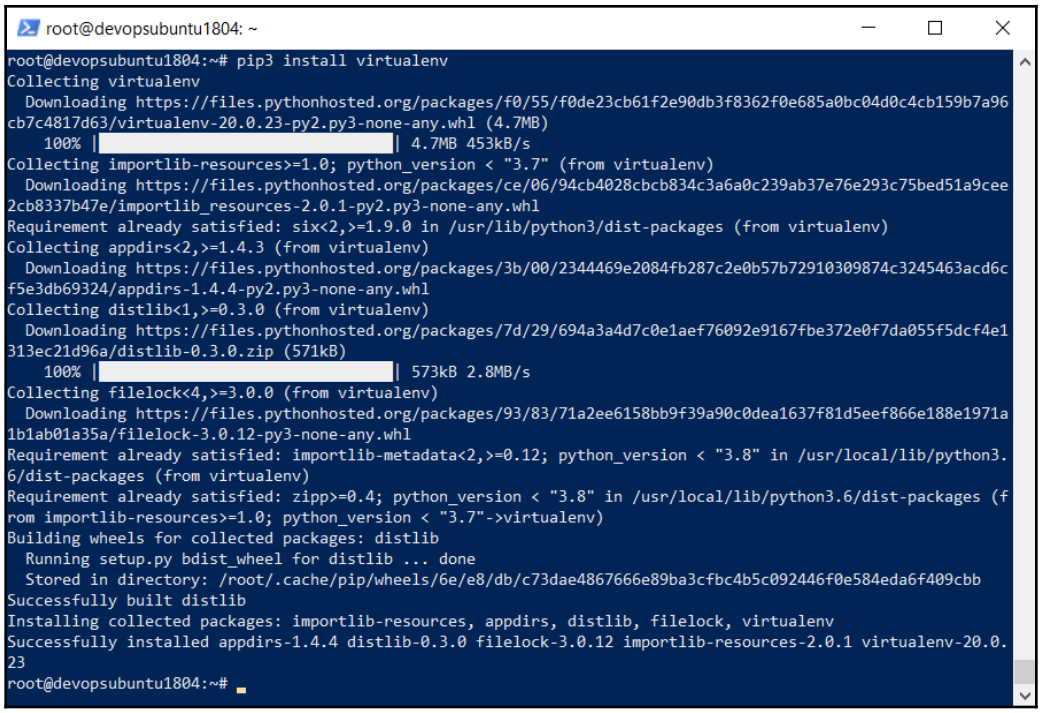

Figure 11.18 – virtualenv installation

7. Now we can create the Bash script for `virtualenv` as follows. This script calls to the `python3 '/usr/local/src/testgres/atmrds.py'` test file:

```
root@devopsubuntu1804:~# nano /usr/local/src/testgres/rds_tests.sh
-------------------------------------------------------------------
#!/usr/bin/env bash

VIRTUALENV="virtualenv --python=/usr/bin/python$PYTHON_VERSION"
PIP="pip$PYTHON_VERSION"

# prepare environment
VENV_PATH=/tmp/testgres_venv
rm -rf $VENV_PATH
$VIRTUALENV $VENV_PATH
export VIRTUAL_ENV_DISABLE_PROMPT=1
source $VENV_PATH/bin/activate

# Install local version of testgres
$PIP install testgres
```

```
# run tests (PG_BIN)
/usr/local/src/testgres/atmrds.py
----------------------------------------------------------------------
```

8. Please grant execution permission for the script:

```
root@devopsubuntu1804:~# chmod 755
/usr/local/src/testgres/rds_tests.sh
```

9. We navigate away from the current `/root` folder and then execute the `virtualenv` Bash script as a Postgres user:

```
root@devopsubuntu1804:~# cd /usr/local/src/testgres/

root@devopsubuntu1804:/usr/local/src/testgres# su postgres -c
'./rds_tests.sh'
created virtual environment CPython3.6.9.final.0-64 in 184ms
  creator CPython3Posix(dest=/tmp/testgres_venv, clear=False,
global=False)
  seeder FromAppData(download=False, pip=latest, setuptools=latest,
wheel=latest, via=copy,
app_data_dir=/var/lib/postgresql/.local/share/virtualenv/seed-app-
data/v1.0.1)
  activators
BashActivator,CShellActivator,FishActivator,PowerShellActivator,Pyt
honActivator,XonshActivator
Processing
/var/lib/postgresql/.cache/pip/wheels/a7/2c/a6/37870923d4e356392e53
abab3a242cc67535075480e6c177b0/testgres-1.8.3-py3-none-any.whl
Collecting pg8000
  Using cached pg8000-1.15.3-py3-none-any.whl (24 kB)
Processing
/var/lib/postgresql/.cache/pip/wheels/a1/d9/f2/b5620c01e9b3e858c687
7b1045fda5b115cf7df6490f883382/psutil-5.7.0-cp36-cp36m-
linux_x86_64.whl
Collecting port-for>=0.4
  Using cached port_for-0.4-py2.py3-none-any.whl (21 kB)
Collecting six>=1.9.0
  Using cached six-1.15.0-py2.py3-none-any.whl (10 kB)
Collecting scramp==1.2.0
  Using cached scramp-1.2.0-py3-none-any.whl (6.3 kB)
Installing collected packages: scramp, pg8000, psutil, port-for,
six, testgres
Successfully installed pg8000-1.15.3 port-for-0.4 psutil-5.7.0
scramp-1.2.0 six-1.15.0 testgres-1.8.3
.F
======================================================================
===
```

```
FAIL: test_bank_zipcode (__main__.RdsTest)
----------------------------------------------------------------------
---
Traceback (most recent call last):
  File "/usr/local/src/testgres/atmrds.py", line 26, in
test_bank_zipcode
    self.assertFalse(data, 'Bank ZipCode is not included in Zip
coordinates table')
AssertionError: [(1,)] is not false : Bank ZipCode is not included
in Zip coordinates table

----------------------------------------------------------------------
---
Ran 2 tests in 2.018s

FAILED (failures=1)
root@devopsubuntu1804:/usr/local/src/testgres#
```

10. The result confirms that we execute two tests and the first test has failed with the `Bank ZipCode is not included in Zip coordinates table` exception message:

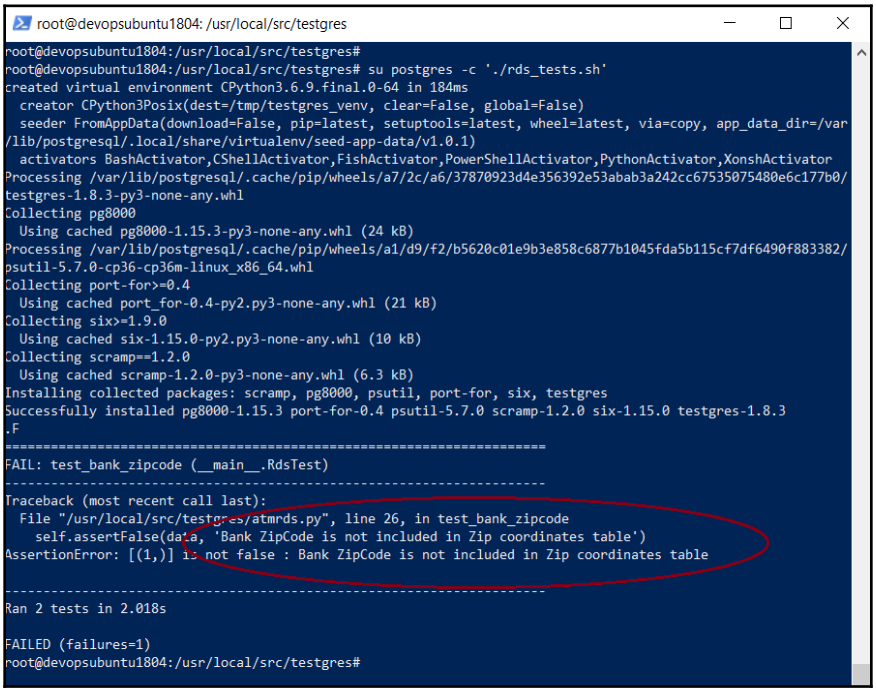

Figure 11.19 – Running Testgres tests with virtualenv

Uninstalling Testgres for PostgreSQL RDS

Because we have installed Testgres with the Vagrant user, now we can exit from the root user to perform Testgres uninstallation as follows:

```
root@devopsubuntu1804:/usr/local/src/testgres# exit
vagrant@devopsubuntu1804:~$ sudo -H pip3 uninstall -y testgres
Uninstalling testgres-1.8.3:
  Successfully uninstalled testgres-1.8.3
```

As we have seen, Testgres is a serious test framework. It should be taken into consideration when testing in a real production environment. A disadvantage could be that it is hard to install in a Windows environment.

Summary

In this chapter, with the help of a step-by-step project, we have learned PostgreSQL test skills such as schema validation and xUnit-style testing through pgTAP, automated tests for existing stored procedures or developing procedures with PGUnit and simple pgunit, and PostgreSQL node control with Testgres.

This book is now complete. We can proceed on to the book's *Appendix* to learn about PostgreSQL using other clouds that are different from AWS.