case_12.3

May 1, 2020

1 Analyzing Net Promoter Score (NPS) data with SQL

1.1 Introduction (10 mts)

Business Context. You are a data scientist at a new but fast-growing startup. The startup released its first product 12 months ago and has been tracking Net Promoter Score (NPS) over its growing customer base since the product's launch.

The team assumes that the NPS score is correlated to the product stability and feature-completeness and that the product has been getting more stable and complete over time. They also realize that there have been some hiccups along the way, and they assume that NPS has therefore fluctuated up and down.

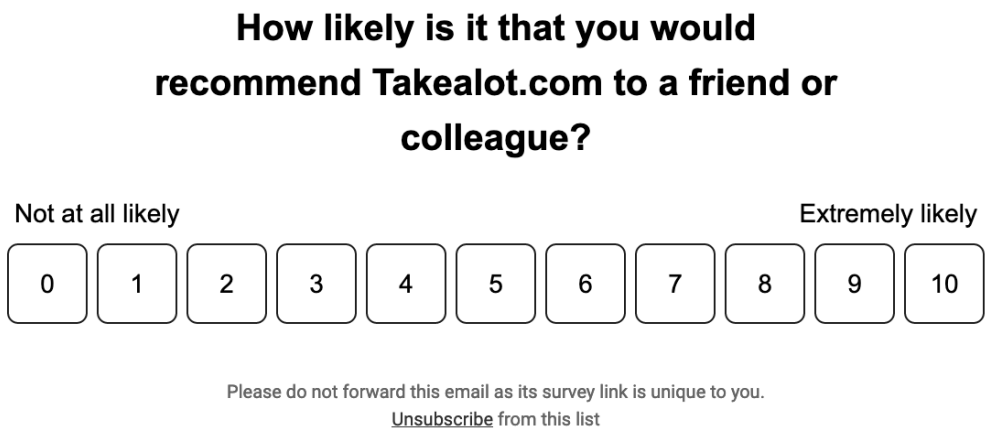
Business Problem. The startup wants you to investigate the data and answering the following question: **"Has our NPS improved over time? And has our average NPS decreased in specific periods over the last 12 months?"**

Analytical Context. In this case, you will be working with a large dataset – so large that your personal laptop is not powerful enough to run heavy SQL queries on it (the startup is stingy and doesn't provide employees with hardware – luckily they have free cloud credits though!). Instead, you will be working with a powerful PostgreSQL database in the cloud (on Amazon Web Services), and uploading the data there for remote processing. We'll connect to the remote database and have the remote machine run the resource-intensive queries.

Specifically, you will: (1) get familiarized with what NPS is and some of its properties; (2) set up Amazon RDS and import the given dataset into it; (3) use advanced SQL features to calculate NPS using the remote machine.

1.2 Understanding the Net Promoter Score (NPS)

NPS is a metric to measure customer satisfaction. You've probably seen pop-ups online, or received surveys via email, asking you "Would you recommend [product] to a friend or family member?" and giving you the option to respond with a number between 0 and 10. That's someone collecting information to calculate their NPS.



The basic idea is simple - customers who respond with high ratings are more likely to promote your product to other potential customers. Customers who give low ratings are unhappy and are unlikely to help you grow your customer base. If you ask enough people at different time periods, you can track customer satisfaction over time and see how this correlates to product development and other aspects of your business that are within your control.

NPS categorizes users into three groups based on the ratings that they leave. This is done as follows:

1. Users who leave a rating of 0 - 6 are regarded as ”detractors”
2. Users who leave a rating of 7 or 8 are regarded as ”passives”
3. Users who leave a rating of 9 or 10 are regarded as ”promoters”

The final NPS score for a given period is calculated as the percentage of total users who are promoters minus the percentage of total users who are detractors. This means that an NPS score can be anything from -100 to 100.

1.2.1 Exercise 1:

If you have the following scores left by your customers:

Date	CustomerId	Score	Group
1 January 2018	562	1	Detractor
1 January 2018	544	10	Promoter
2 January 2018	333	9	Promoter
2 January 2018	102	9	Promoter
4 January 2018	267	9	Promoter
5 January 2018	981	10	Promoter
6 January 2018	105	6	Detractor
6 January 2018	459	7	Passive
6 January 2018	188	10	Promoter
8 January 2018	982	8	Passive

What is your NPS? How would you adjust your calculation of NPS if instead users had many opportunities to rate you in a short time period? What would you consider to be a "good" NPS?

Answer. We have 10 responses: 6 promoters, 2 detractors and 2 passives. That is 60% promoters and 20% detractors, so our NPS is $60 - 20 = 40$.

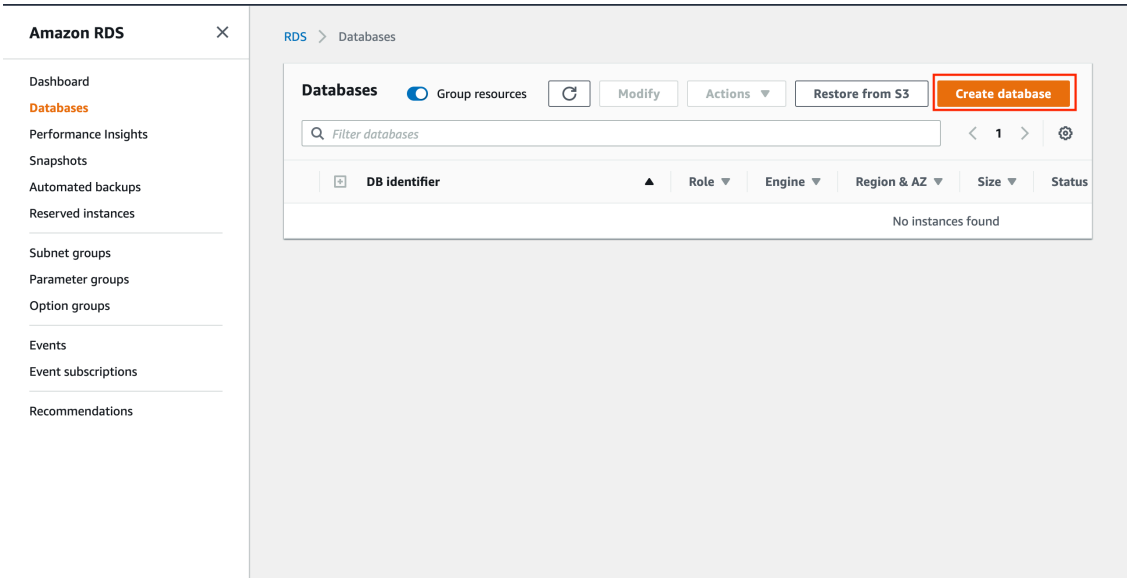
If users could rate us many times in a short time span, a sensible adjustment would be to first average all of the responses per user, and use this averaged response to group each user into promoter, detractor, or passive. This is because no matter how many times a single user interacts with our product, they are likely still only paying us once for it. Thus, since they are not weighted more heavily in our revenue streams, they should not be weighted more heavily in our customer satisfaction schemes either.

Defining what constitutes a good NPS depends on the specific line of business the company is in. It varies between different industries with internet providers generally getting far lower scores than technology companies.

1.3 Setting up a cloud database using RDS and importing data

Enough theory! Let's set up a database and load in some NPS data so that we can analyze it using SQL. We'll use the code at this repository to generate a large sample of fake NPS data and push it into a PostgreSQL instance running in the cloud. (Don't look at the source code that generates the data, as it will spoil the fun.)

1. Log into your AWS account and select "RDS" from the service list. You should see a screen like the one below, where you can hit the "Create database" button:

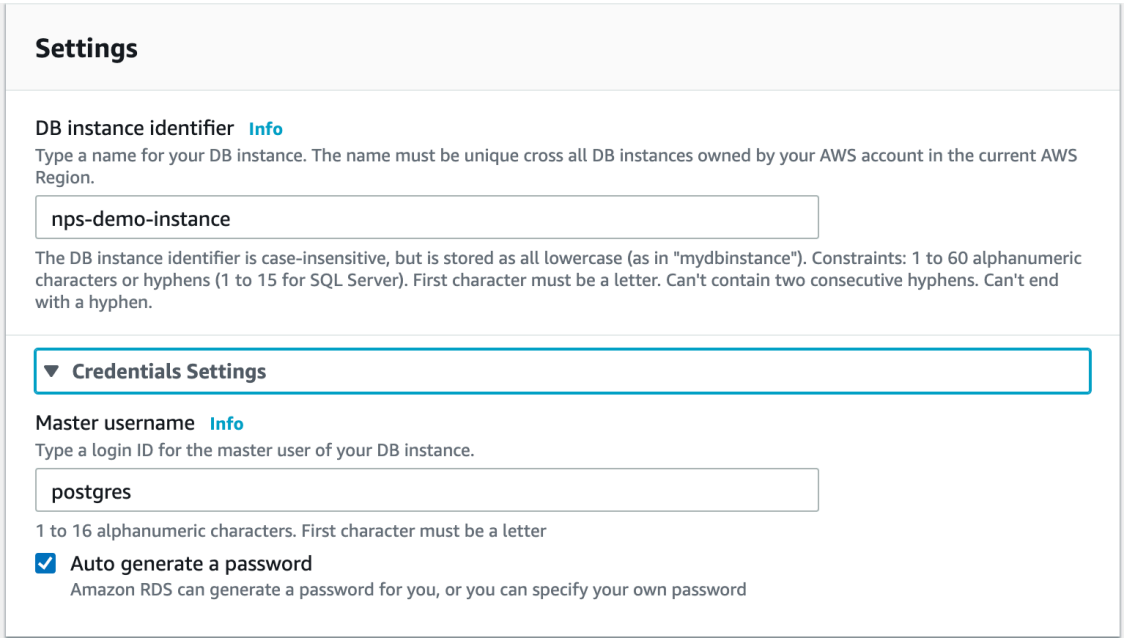


2. The next option you'll see asks you if you want to use "standard create" or "easy create". Easy might sound tempting, but **choose "standard"** as we'll have to set up our database for public use so we can connect to it locally.

3. Choose "PostgreSQL" as the database type, leave the version at the default AWS has chosen for you (10.6-R1 at the time of writing), and choose "Free Tier"

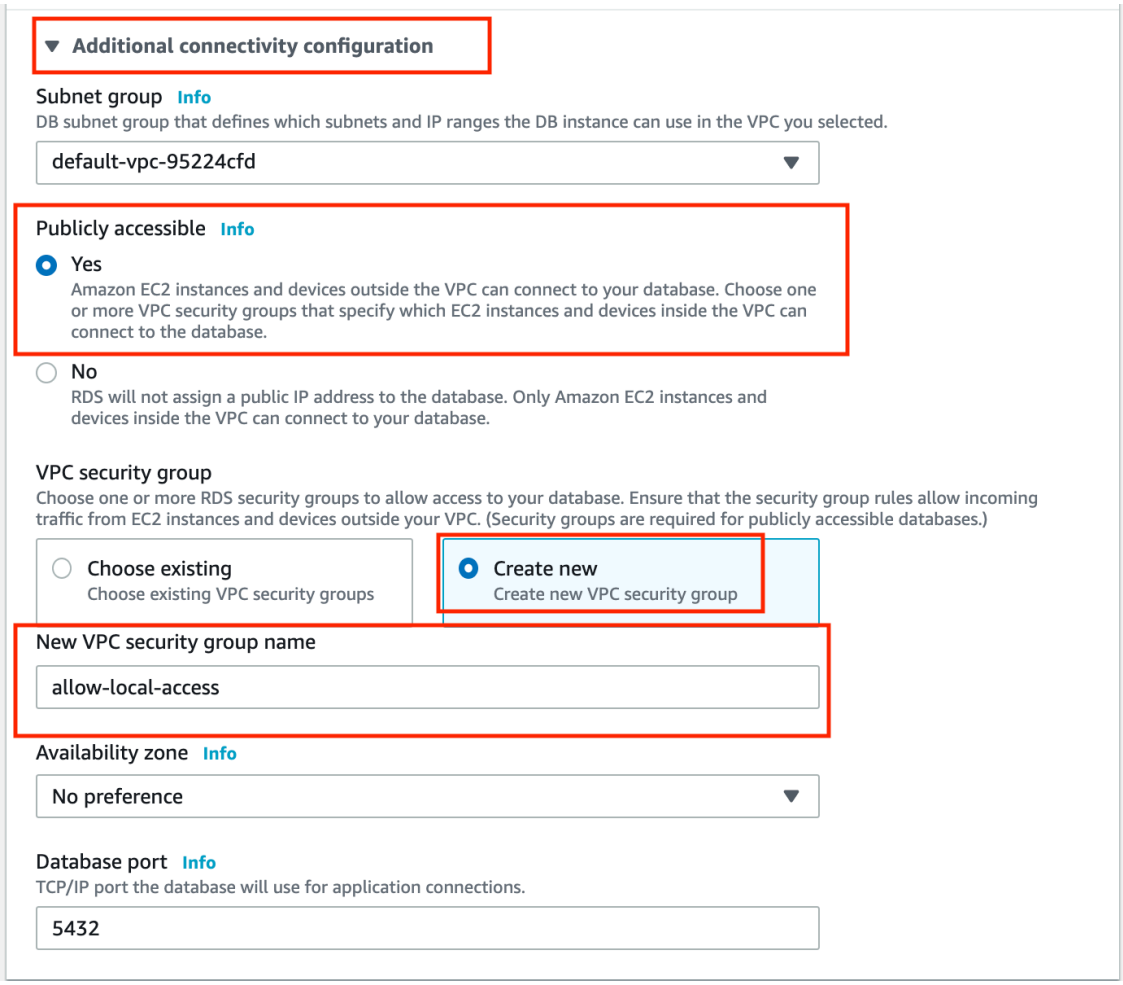
4. Under the next section, choose a name for your database instance. Remember this is the machine that is hosting the database software, not the database itself (one RDS instance can host many databases), so I'm calling mine `nps-demo-instance` to reflect this, although we'll only be creating a single database for now.

5. You can leave the master username as `postgres` and ask RDS to autogenerate a password (we'll be able to see this password at the next step):

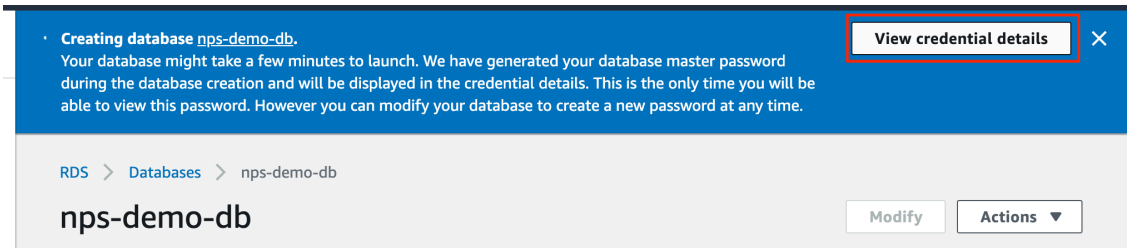


6. You can leave the next settings as their defaults until you get to the "Connectivity" section. Usually, you'll set up an RDS instance to play with other infrastructure within your AWS account, such as EC2 servers. In our case, we want to push data in and out of the database directly from our local machine as the client, so we'll have to set our database up for "public access". This is generally less secure, but we'll add some firewall rules in a bit to make sure that only we can access it:

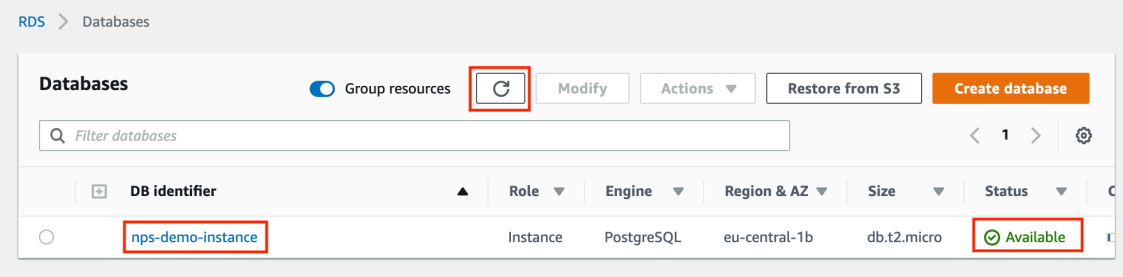
   - Expand the "Additional connectivity configuration" section
   - Set "publicly accessible" to "Yes"
   - Under "VPC security group", choose to "Create new", and give it a name like `allow-local-access`. This will create a firewall rule that will allow you to connect to your database on port 5432 (the default for PostgreSQL) using your current IP address. If you are using public WiFi, a hotspot, or if you think your IP address is likely to change soon for any reason, note that you'll have to modify this security group any time your IP address changes:



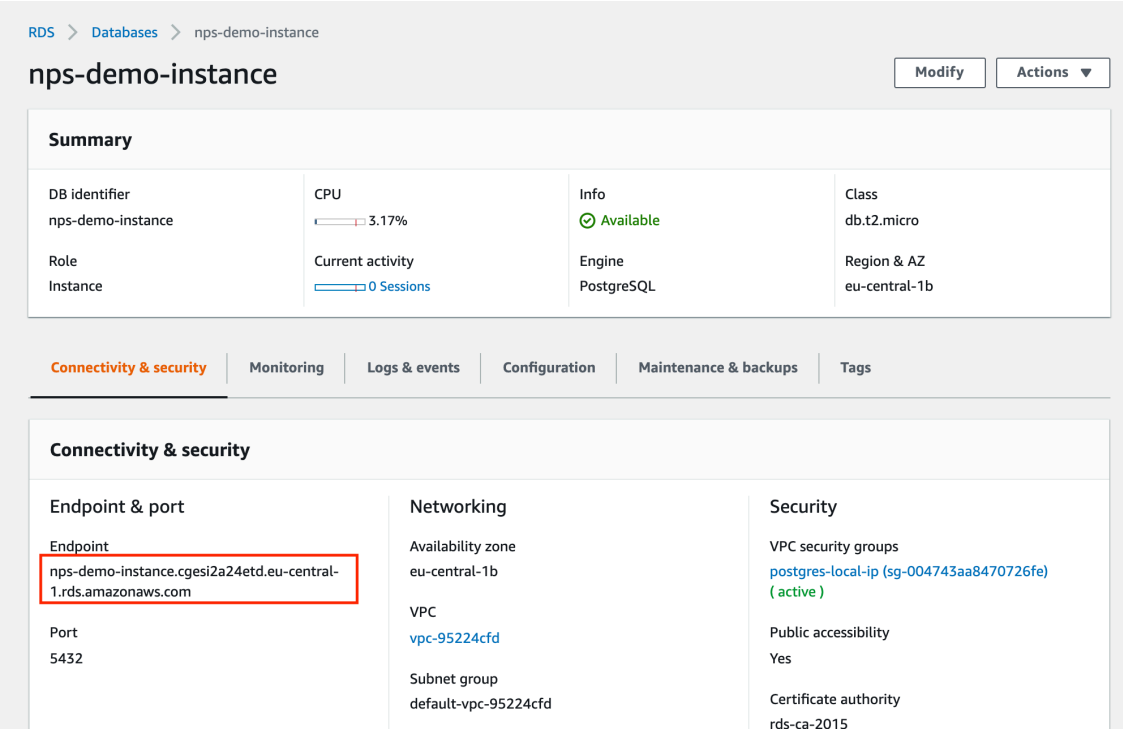
7. Press the "Create database" button in the bottom right, and you'll be taken back to the overview page where you can see your database being created. At the top, there'll be a notification where you can press "View credential details" to access your master password that was automatically generated. Take note of this as you can only see it once. NOTE: this creates a database in the default VPC. If your default VPC is not configured for DNS connections, you will need to create a new VPC. Please see 'Appendix 1: Troubleshooting RDS creation' for instructions on how to do achieve this.



8. Once your database becomes "available" (you might need to press the "refresh" button indicated below to see the change), you can connect to it. Click on the name of the database (`nps-demo-instance` in our example), to find out the connection details:



9. Once you click on the database, you should see the endpoint that you need on a screen similar to the one shown below. You need this endpoint to connect to the database from your local machine.



10. Locally, open a terminal and run the following command, substituting [endpoint] with the one that you noted from the RDS console above.

```
psql -h [endpoint] -U postgres
```

This will connect to our instance's default database using the master username. It will prompt you for the password and you can enter the autogenerated password from above. You should now see a SQL prompt, similar to the image below:

```
~/git/nps-sample-data $ psql -h nps-demo-instance.cgesi2a24etd.eu-central-1.rds.amazonaws.com -U postgres
Password for user postgres:
psql (11.5, server 10.6)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

postgres=>
```

We've successfully created a cloud database and connected to it!

1.3.1 Setting up our NPS database

Let's proceed by setting up our database in Amazon RDS:

1. In the SQL shell, run the following commands to create a database, create a user to manage our database, and give privileges on our new database to our new user. Replace [password] with your own choice of password:

```
create database nps_demo_db;
create user nps_demo_user with login encrypted password '[password]';
grant all privileges on database nps_demo_db to nps_demo_user;
\q
```

Here, `\q` closes the connection so you can re-open it under a different user.

2. Run the following command. It is similar to the one we used before to connect but now specifies both our custom user and our custom database. Once again, substitute [endpoint] with the one you see in the RDS console.

```
psql -h [endpoint] -U nps_demo_user -d nps_demo_db
```

3. Put in the new password that you entered in the SQL statement in step 1 instead of the master password that AWS automatically generated for us. You'll see a very similar prompt, but with the `nps_demo_db=>` prompt instead of `postgres=>`:

```
~/git/nps-sample-data $ psql -h nps-demo-instance.cgesi2a24etd.eu-central-1.rds.amazonaws.com -U nps_demo_user nps_demo_db;
Password for user nps_demo_user:
psql (11.5, server 10.6)
SSL connection (protocol: TLSv1.2, cipher: ECDHE-RSA-AES256-GCM-SHA384, bits: 256, compression: off)
Type "help" for help.

nps_demo_db=>
```

The next thing we need to do is to create tables to house our data. We'll use the data from this repository, consisting of two tables: `customer` and `score`. There are some extra fields on `customer` (`is_premier` and `is_spam`) that we won't use right away, but we'll create our tables to match that format anyway to make the import easier.

The important context is that we are imagining a scenario where:

- We have been running a new company for around one year.
- The product has gone through different stages of feature improvement and stability but has overall shown growth and improvement.
- Every day, new customers join and both new and old customers may or may not leave us a score between 0-10 to rate how likely they are to recommend our product to family and friends.
- At the start and at some key points during the year, the product is unstable or lacking features and this affects the customer rating.

1.3.2 Exercise 2:

Using the SQL prompt, create two tables: one for customers, and one for the scores that our customers leave.

Answer. To create the `customer` table, we do the following:

```
create table customer (id serial not null, created_at date, is_premier boolean, is_spam boolean
```

This creates a `customer` table with an ID, the date the customer first signed up (`created_at`), and two boolean flags that we don't need yet. It also adds a constraint to the ID field saying it is a primary key, meaning it has to be unique.

To create the table of scores that our customers leave, use the following command in the same prompt:

```
create table score (id serial not null, customer_id integer references customer(id), created_at
```

This is similar to the `customer` table, but has a `score` field to store the value between 0 and 10 that a customer leaves each time they complete a survey, and another date field to record when the survey was done. There is also a foreign key `customer_id` to link each score to a specific entity in the `customer` table.

You can close the connection again with `\q`.

1.3.3 Pushing sample data into RDS

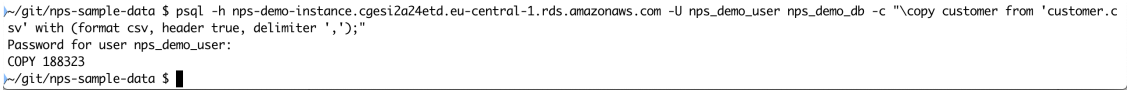
Let's now push the NPS data onto RDS:

1. Download the two CSV files (`score.csv` and `customer.csv`) from https://github.com/sixhobbits/nps-sample-data into your local working directory. Don't look at the README file.
2. Run the command below, again substituting [endpoint] with the actual endpoint you used above. Make sure that the `customer.csv` file is located in the same directory that you run the `psql` command from:

```
psql -h [endpoint] -U nps_demo_user -d nps_demo_db -c "\copy customer from 'customer.csv' with
```

The first part of the command is the same one we used before to open a SQL shell. Here we also pass the `-c` flag which allows us to specify a SQL command to be run on the database. Because our shell has permissions to access our local file system, but our database doesn't, running the command like this means we won't have problems with permissions. In the `\copy` command, we specify which table we want to populate (`customer`), where the local file is (`customer.csv`), that our file is in CSV format, that it has a header, and that we are using a comma as a delimiter.

This should prompt you for the password (again, use the one that you created for the `nps_demo_user`). It will then let you know how many rows it has successfully imported, similar to the image below:



3. Now we can add the scores data as well using the same method. The only things we need to change are the table name and the filename from which we source the data. The full command (don't forget to substitute your endpoint) is:

```
psql -h [endpoint] -U nps_demo_user -d nps_demo_db -c "\copy score from 'score.csv' with (forma
```

There are a lot more sample scores than customers (as each customer can respond to the survey more than once), so this will take a bit longer than the previous command:

```
~/git/nps-sample-data $ psql -h nps-demo-instance.cgesi2a24etd.eu-central-1.rds.amazonaws.com -U nps_demo_user nps_demo_db -c "\copy score from 'score.csv' wi
th (format csv, header true, delimiter ',');"
Password for user nps_demo_user:
COPY 1577578
~/git/nps-sample-data $
```

1.4 Analyzing our NPS data using SQL

Now we can proceed to the fun part. We have NPS scores left by a large number of customers over the past year, and we want to see how these scores change over time.

We only have raw data – numbers between 0 and 10 inclusive – so we'll use SQL to group this data in different ways and transform it into NPS data. If you remember how to define NPS from the first section, you can probably work out that the main things we need to do are:

1. Break down our scores per customer for any given time period (here, we will look at this per week)
2. Divide customers into promoters, passives or detractors, based on the scores they have left in that week
3. Calculate the NPS per week and look at how this value changes week-by-week

1.4.1 Counting customers and scores

We saw how many customers and scores we had when we did the import step above. However, in a real-world setting, you would have gathered this data slowly, over time, so let's start by counting out customers, our survey responses (`scores`), and looking at how many surveys each customer responds to. For each of the following, you'll need to be connected to the SQL shell, so run the following first (using your endpoint) and any time you need to.

We'll show the output of each SQL command directly below – you only need to enter the command shown in the first section in each of the following examples.

```
psql -h [endpoint] -U nps_demo_user -d nps_demo_db
```

Counting customers

```
SELECT COUNT(*) FROM customer;
```

```
 count
--------
 188323
(1 row)
```

We have nearly 200k customers, which is not bad for a product that's been running for one year!

Counting scores

```
SELECT COUNT(*) FROM score;
```

```
  count
---------
 1577578
(1 row)
```

And we have over 1.5 million survey responses. That's just over 8 responses per customer if we assume an equal distribution. Let's use SQL to look at that.

1.4.2 Exercise 3:

Write a SQL query that outputs a table showing the 10 customers with the highest number of responses and their total response count, in descending order (customer with most responses at the top).

Answer. One possible solution is given below:

```
SELECT customer_id, COUNT(score.id) AS cnt FROM score
INNER JOIN customer ON customer_id = customer.id
GROUP BY customer_id ORDER BY cnt DESC
LIMIT 10;
```

```
 customer_id | count
-------------+-------
          31 |    38
         928 |    38
        4271 |    38
        5333 |    37
        1253 |    37
        1259 |    36
        1030 |    36
        2327 |    36
         564 |    36
        2335 |    36
```

We can see that the top three places have customer IDs `31`, `928` and `4271`, each having left 38 survey responses.

You might also be used to doing SQL JOINs using commas and a `WHERE` clause as a shortcut. The above command is equivalent to the following one, but the earlier version is preferable in most contexts as it is more explicit:

```
SELECT customer_id, COUNT(score.id) AS cnt FROM score, customer
WHERE customer_id = customer.id
GROUP BY customer_id ORDER BY cnt DESC
LIMIT 10;
```

We can also look at customers who have left very few responses by ordering by `ASC` instead of `DESC`:

```
SELECT customer_id, COUNT(score.id) AS cnt FROM score
INNER JOIN customer ON customer_id = customer.id
GROUP BY customer_id ORDER BY cnt ASC
LIMIT 10;
```

```
 customer_id | cnt
-------------+-----
       57565 |   1
       62357 |   1
       49021 |   1
       57424 |   1
       61891 |   1
       62295 |   1
       44796 |   1
       44995 |   1
       57286 |   1
       62402 |   1
```

We can see there are at least 10 customers who have left only a single response. Let's do a 'count of counts' query to get a better idea of how many responses most customers leave. We want to count how many customers have left exactly x responses.

1.4.3 Exercise 4:

Write a SQL query that outputs a table showing how many customers leave x responses for any given integer x. Sort this table in descending order (x with highest number of customers leaving x responses at the top).

(Hint: Use a **nested** `SELECT` statement. A nested statement is when you treat the results of one query as the input to another one.)

Answer. One possible solution is given below:

```
SELECT cnt, COUNT(cnt) as count_of_count FROM
(SELECT customer_id, count(score.id) AS cnt FROM score
INNER JOIN customer ON customer_id = customer.id
GROUP BY customer_id ) a
GROUP BY cnt
ORDER BY count_of_count DESC
LIMIT 100;
```

Notice in the query above we have taken a query very similar to the one from Exercise 3 and nested it in parentheses. We have then given this intermediate query an **alias**, which comes immediately after the closing parenthesis; in this case we have chosen the alias `a`. It is a common convention to use aliases `a`, `b`, `c`, etc. as a shorthand if you are primarily interested only in the final result.

From our previous queries, we already know that all the values have to fall between 1 and 38, so there can be a maximum of 38 rows returned in this query. Therefore there is no real need to add a LIMIT clause, but we add a `LIMIT 100` anyway. This is a good habit in case you make a wrong

assumption to prevent the case where you accidentally try to pull thousands or millions of rows from a remote server. For brevity, we only included the first 15 rows of output below:

```
 cnt | count_of_count
-----+----------------
   6 |          18779
   5 |          17218
   7 |          17094
   4 |          15642
   8 |          14108
   3 |          12983
   9 |          11978
  10 |          10556
   2 |          10191
  11 |           9001
  12 |           7833
  13 |           6698
   1 |           6302
  14 |           5707
  15 |           4908
```

We can see that most customers leave between 2 and 10 responses so the maximum of 38 is an outlier. A fair number of people only leave one response.

1.4.4 Average scores per week

However, we still have not looked at how scores are *changing*. Let's average all scores in each week and see how the scores go up and down over time:

```
SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, AVG(score) AS avg_score
FROM score
GROUP BY week
ORDER BY week ASC
LIMIT 100;
```

Again, we did not need to add a limit clause as we know there will only be 52 rows (the number of weeks in a year, which is the span of our dataset), but we do anyway for good measure and again include only the first 15 rows of output below:

```
  week   |     avg_score
---------+--------------------
 2018-01 | 5.3618090452261307
 2018-02 | 6.1577181208053691
 2018-03 | 5.1405228758169935
 2018-04 | 5.2256097560975610
 2018-05 | 6.3962765957446809
 2018-06 | 7.2065359477124183
 2018-07 | 7.0110294117647059
 2018-08 | 6.9827490261547023
```

```
 2018-09 | 7.4689516129032258
 2018-10 | 7.9564362001124227
 2018-11 | 8.0201993704092340
 2018-12 | 7.8336310283235519
 2018-13 | 7.9298795180722892
 2018-14 | 7.9583184257602862
 2018-15 | 7.9876211782252051
```

We can see that the scores start low and generally trend up over time, although they go down again around week 36 (not shown above). We use the ISO Week through PostgreSQL's `TO_CHAR` function to break down each of our dates into a specific week number and average the scores per week.

There are a couple issues with the above query, though:

1. The `AVG` function shows a lot of decimal points by default which makes it more difficult to read the data
2. Many customers leave a different number of responses and some might leave more than one response per week

A good compromise is to calculate the average score per customer per week, then average all of these to get an average score across all customers per week. Let's do this and round off some decimal points to make our data easier to read.

1.4.5 Exercise 5:

Write a query to compute the average score across all customers per week, rounding off to two decimal places. (Hint: Use the `ROUND()` function, which takes two arguments: the quantity you are rounding, and how many decimals you are rounding off to.)

Answer. One possible solution is given below:

```
SELECT week, ROUND(AVG(avg_week_score),2) as avg_score FROM
(SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, customer_id, AVG(score) as avg_week_score
GROUP BY week, customer_id) a
GROUP BY week
ORDER BY week
LIMIT 100;
```

```
 week    | avg_score
---------+-----------
 2018-01 |      5.12
 2018-02 |      5.80
 2018-03 |      5.74
 2018-04 |      5.50
 2018-05 |      6.33
 2018-06 |      7.02
 2018-07 |      7.01
 2018-08 |      6.89
 2018-09 |      7.38
 2018-10 |      7.75
```

```
 2018-11 |      7.80
```

1.4.6 Classifying our customers as promoters, passives, or detractors

Now, let's proceed to classifying our customers so we can calculate the NPS per week. We used a similar `SELECT` (two deep this time!) and a `CASE` statement. The `CASE` keyword acts as an if statement and returns specific values in specific cases. For us, anything larger than an 8 (i.e. 9 or 10) is a promoter, otherwise, anything larger than a 6 (i.e. 7 or 8) is a passive and everything else is a detractor:

```
SELECT * FROM
(SELECT CASE
    WHEN avg_week_score > 8 THEN 'promoter'
    WHEN avg_week_score > 6 THEN 'passive'
    ELSE 'detractor'
END AS nps_class, week FROM
(SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, customer_id, AVG(score) as avg_week_score
GROUP BY week, customer_id) a) b
limit 10;
```

Which gives us the following output: a huge table with the nps_class and the week number:

```
 nps_class |  week
-----------+---------
 detractor | 2018-01
 detractor | 2018-01
 promoter  | 2018-01
 detractor | 2018-01
 promoter  | 2018-01
 detractor | 2018-01
 detractor | 2018-01
 detractor | 2018-01
 detractor | 2018-01
 promoter  | 2018-01
```

This is closer to what we need, but not very useful in its current form. We can confirm that there are still nearly a million rows by using another `COUNT`:

```
SELECT count(*) FROM
(SELECT CASE
    WHEN avg_week_score > 8 THEN 'promoter'
    WHEN avg_week_score > 6 THEN 'passive'
    ELSE 'detractor'
END AS nps_class, week FROM
(SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, customer_id, AVG(score) as avg_week_score
GROUP BY week, customer_id) a) b
limit 10;
```

```
 count
--------
```

```
 951289
(1 row)
```

Now that we've broken our customers into specific categories, we want to count them. It's useful to "pivot" this data so that we can see the count of each class of people as a separate column. In a spreadsheet program like Microsoft Excel or Google Sheets, we would think of this as a pivot table, and there are plugins for PostgreSQL to allow you to use it in a similar way. In our case, though, we can count the number of each class each week using some more `CASE` statements and the `SUM` function as follows:

```
SELECT week,
SUM(CASE WHEN nps_class = 'promoter' THEN 1 ELSE 0 END) AS "promoter",
SUM(CASE WHEN nps_class = 'passive' THEN 1 ELSE 0 END) AS "passive",
SUM(CASE WHEN nps_class = 'detractor' THEN 1 ELSE 0 END) AS "detractor",
    COUNT(*) AS "total" FROM
(SELECT CASE
    WHEN avg_week_score > 8 THEN 'promoter'
    WHEN avg_week_score > 6 THEN 'passive'
    ELSE 'detractor'
END AS nps_class, week FROM
(SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, customer_id, AVG(score) as avg_week_score
GROUP BY week, customer_id) a) b
GROUP BY week
ORDER BY week
limit 100;
```

Which results in (truncated for brevity):

```
  week   | promoter | passive | detractor | total
---------+----------+---------+-----------+-------
 2018-01 |       26 |       0 |        39 |    65
 2018-02 |       65 |       2 |        63 |   130
 2018-03 |       71 |       7 |        70 |   148
 2018-04 |       76 |       6 |        83 |   165
 2018-05 |      186 |      23 |       135 |   344
 2018-06 |      397 |      56 |       202 |   655
 2018-07 |      471 |      72 |       238 |   781
 2018-08 |      520 |      79 |       276 |   875
 2018-09 |      771 |     102 |       300 |  1173
 2018-10 |     1154 |     154 |       351 |  1659
 ...
```

Note that we also had to add another intermediate alias (`b`) to our SQL code, as we have yet another level of nested `SELECT`.

1.4.7 Calculating NPS per week

We now have all the pieces in place to calculate our NPS. To do this, we will have to use a *third* nested `SELECT` and yet another table alias `c`.

1.4.8 Exercise 6:

Given the above guidance, write the query to compute NPS per week.

Answer. One possible solution is given below:

```
SELECT *, ROUND(((CAST(promoter AS DECIMAL) / total) - (CAST(detractor AS DECIMAL) / total)) *
(SELECT week,
SUM(CASE WHEN nps_class = 'promoter' THEN 1 ELSE 0 END) AS "promoter",
SUM(CASE WHEN nps_class = 'passive' THEN 1 ELSE 0 END) AS "passive",
SUM(CASE WHEN nps_class = 'detractor' THEN 1 ELSE 0 END) AS "detractor",
    COUNT(*) AS "total" FROM
(SELECT CASE
    WHEN avg_week_score > 8 THEN 'promoter'
    WHEN avg_week_score > 6 THEN 'passive'
    ELSE 'detractor'
END AS nps_class, week FROM
(SELECT TO_CHAR(score.created_at, 'IYYY-IW') AS week, customer_id, AVG(score) as avg_week_score
GROUP BY week, customer_id) a) b
GROUP BY week
ORDER BY week) c
limit 100;
```

```
  week   | promoter | passive | detractor | total | nps
---------+----------+---------+-----------+-------+-----
 2018-01 |       26 |       0 |        39 |    65 | -20
 2018-02 |       65 |       2 |        63 |   130 |   2
 2018-03 |       71 |       7 |        70 |   148 |   1
 2018-04 |       76 |       6 |        83 |   165 |  -4
 2018-05 |      186 |      23 |       135 |   344 |  15
 2018-06 |      397 |      56 |       202 |   655 |  30
 2018-07 |      471 |      72 |       238 |   781 |  30
 2018-08 |      520 |      79 |       276 |   875 |  28
 2018-09 |      771 |     102 |       300 |  1173 |  40
 2018-10 |     1154 |     154 |       351 |  1659 |  48
 2018-11 |     1313 |     180 |       394 |  1887 |  49
 2018-12 |     1419 |     204 |       423 |  2046 |  49
```

That first line is not pretty, but it works! We can now see the NPS, correctly rounded, for any given week.

1.5 Conclusions

In this case, you learned about the Net Promoter Score (NPS) metric. You set up a cloud database using Amazon RDS, a service that makes it easy to manage and scale your databases with little to no work from your local machine. You also learned how to write complex queries in SQL that could be run directly on the cloud database. These queries used advanced features like nested `SELECT` statements and `CASE` statements which can be combined in intricate ways to get the results you need directly from your database.

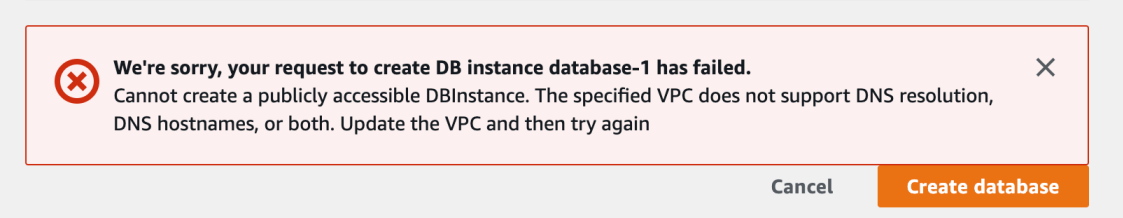
We found that there was a general increase in NPS over time; however, starting in September there was a significant downturn in average NPS score. It is likely that the product encountered some significant bugs or outages during this time and going forward we should check if anything was recorded by the startup's product team to confirm this.

1.6 Takeaways

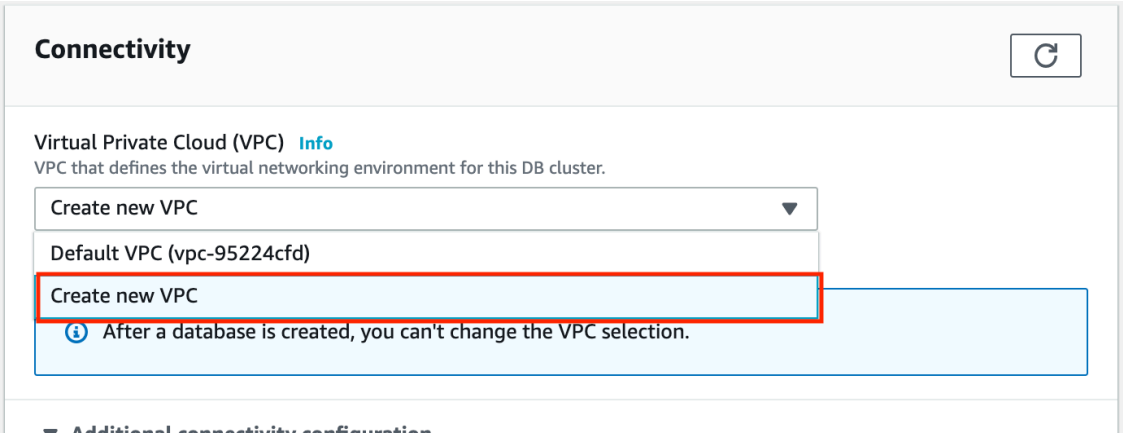
Cloud databases are a powerful and scalable way to analyze data if you have constraints on processor, memory or storage resources for your local hardware. You can do all sorts of things in-cloud that you could originally only do on your local machine, such as run complex SQL queries directly against a cloud database. Although SQL is often seen as "simple" and discarded in favor of new-age languages like Python, basic SQL building blocks, such as `SELECT`, `WHERE`, and `CASE` can be joined to build up sophisticated queries that are highly efficient in comparison to trying to do the same thing in Python.

1.7 Appendix 1: Troubleshooting RDS creation

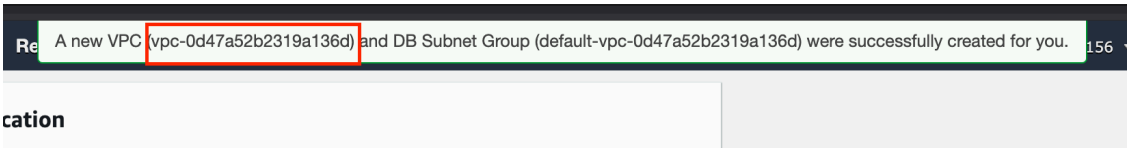
If you cannot create your database using the RDS service and instead see the error below, you will need to create a new VPC instead of using the default one.



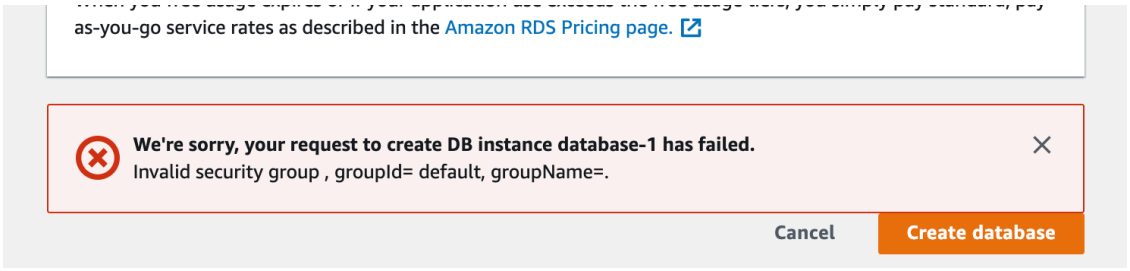
To do this, scroll back up to the 'Connectivity' section, and choose 'create new VPC' from the dropdown as shown in the image below



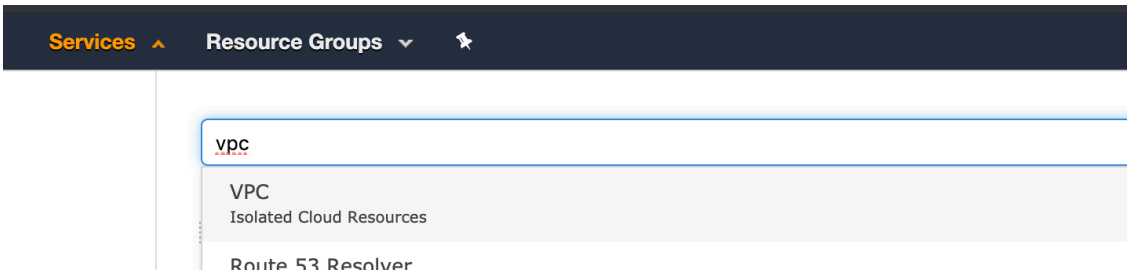
At the bottom of the page, press "Create Database" again, and you should see a notification briefly at the top of the page that confirms a new VPC has been created, as in the image below. Take a note of the ID.



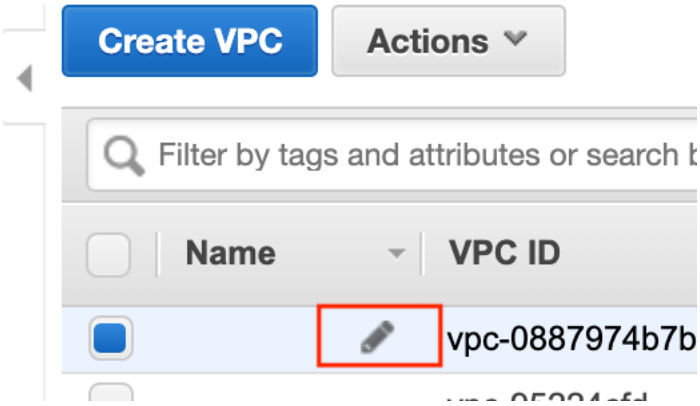
You might now see another error, as follows. This is because the VPC created from the RDS console has no name.



If this is the case, you need to name your VPC. From the services dropdown at the top of the page, search for "VPC" and open the VPC page in a new tab.



Find the VPC that was recently created (it will have the same ID as the one you noted above). Mouse over the 'name' field to see the pencil 'edit' option appear, click on this, and give the VPC a name.



Now that your VPC has a name, go back to the tab where you are creating the RDS instance, and scroll back up to the connectivity section, and choose the newly created VPC (you will see the name you chose displayed) from the dropdown.

Now you can finally press ”Create database” again (at the bottom of the page) and all should work.