Hands-on Lab: Monitoring and Optimizing Your Databases in PostgreSQL

Estimated time needed: 45 minutes

In this lab, you'll learn how to monitor and optimize your database in PostgreSQL with both the command line interface (CLI) and database administration tool, pgAdmin.

Objectives

After completing this lab, you will be able to:

1. Monitor the performance of your database with the command line interface and pgAdmin.
2. Identify optimal data types for your database.
3. Optimize your database via the command line with best practices.

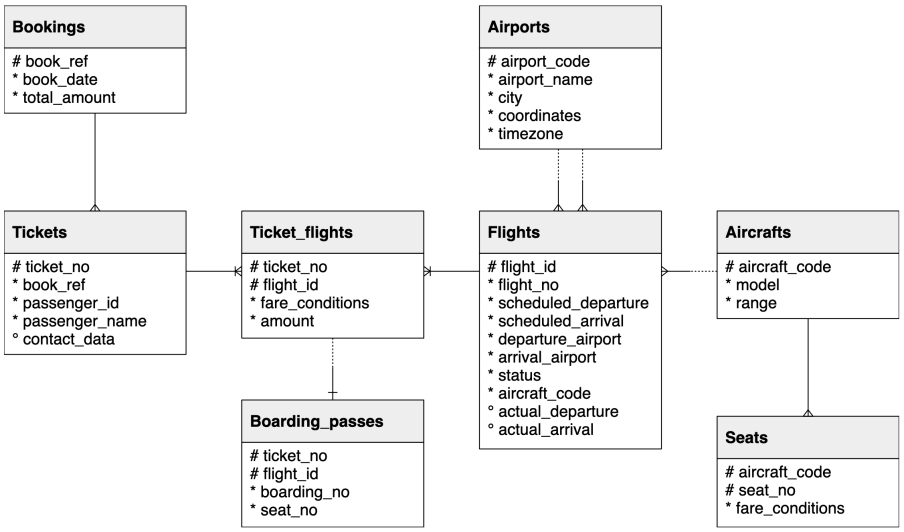
Software Used in this Lab

In this lab, you will be using PostgreSQL. It is a popular open source object relational database management system (RDBMS) capable of performing a wealth of database administration tasks, such as storing, manipulating, retrieving, and archiving data.

To complete this lab, you will be accessing the PostgreSQL service through the IBM Skills Network (SN) Cloud IDE, which is a virtual development environment you will use throughout this course.

Database Used in this Lab

In this lab, you will use a database from https://postgrespro.com/education/demodb distributed under the PostgreSQL licence. It stores a month of data about airline flights in Russia and is organized according to the following schema:



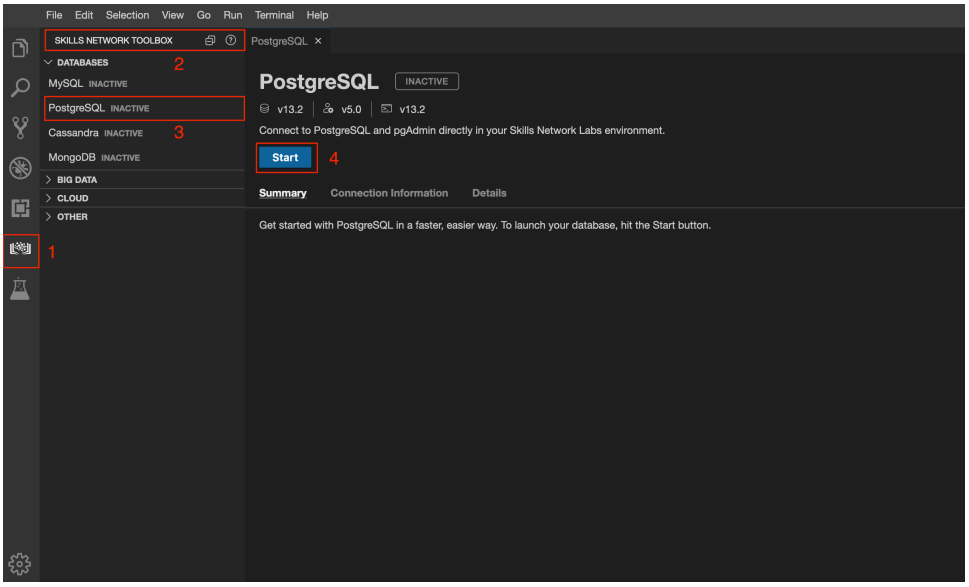
Exercise 1: Create Your Database

To get started with this lab, you'll launch PostgreSQL in Cloud IDE and create our database with the help of a SQL file.

Task A: Start PostgreSQL in Cloud IDE

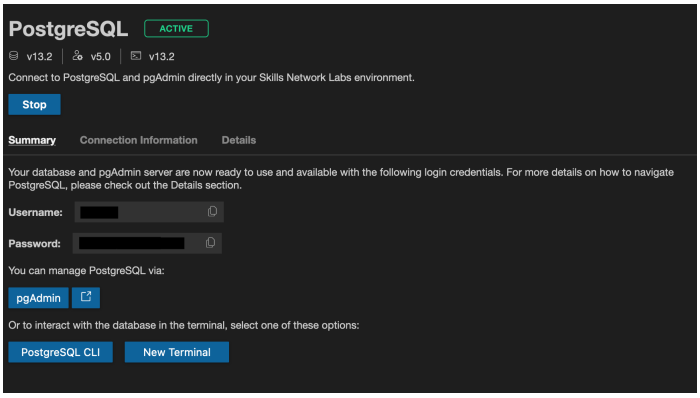
1. To start PostgreSQL, navigate to the **Skills Network Toolbox**, select **Databases**, and select **PostgreSQL**.

   Select **Start**. This will start a session of PostgreSQL in Skills Network Labs.



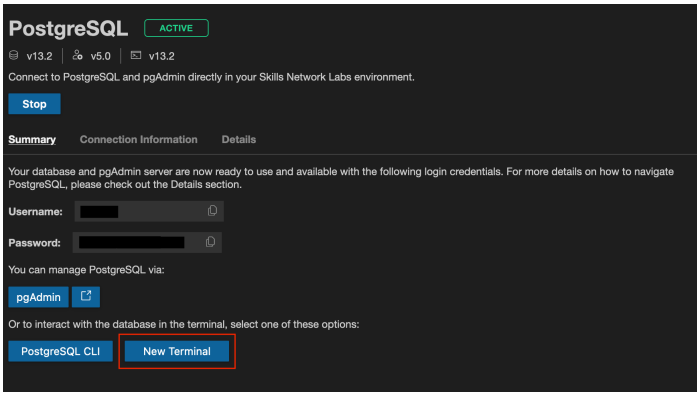
   The **Inactive** label will change to **Starting**. This may take a minute or so.

2. When the label changes to **Active**, it means your session has started.



Task B: Create Your Database

1. Open a new terminal by selecting the **New Terminal** button near the bottom of the PostgreSQL tab.



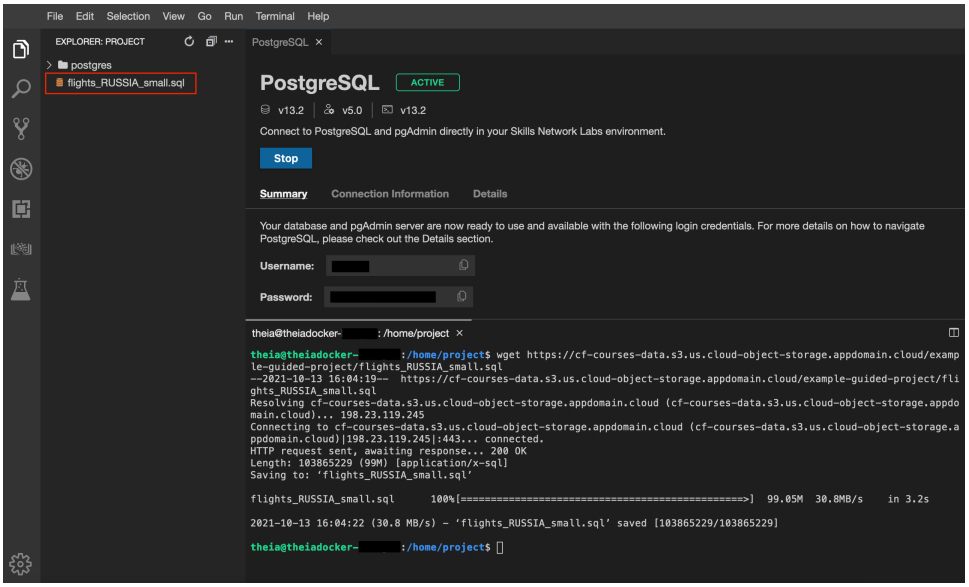
2. With the terminal, you'll want to download the **demo** database that you're using in this lab. This database contains a month of data about flights in Russia.

   To download it, you can use the following command:

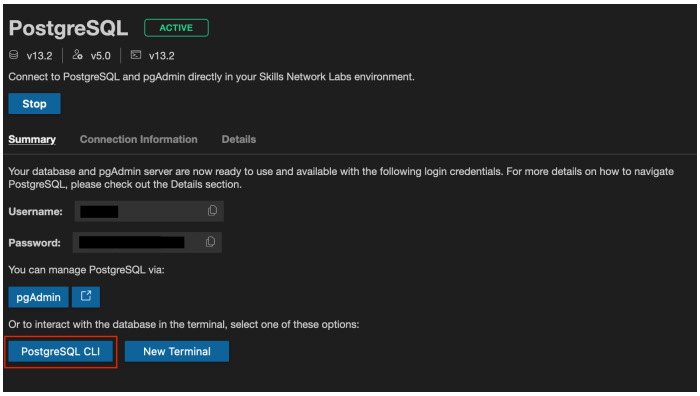
```
1. 1
1. wget https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/example-guided-project/flights_RUSSIA_small.sql
```

   Copied!

   You should now see the SQL file in your file explorer in Cloud IDE.



3. Let's return to the PostgreSQL tab and select the **PostgreSQL CLI** button near the bottom of the tab. This button will open a new PostgreSQL command line session.



4. Now, you want to import the data from the file that you downloaded.

   ▸ Hint (Click Here)
   ▸ Solution (Click Here)

5. With our created database, let's see what tables you have. How many tables are there?

   ▸ Hint (Click Here)
   ▸ Solution (Click Here)

Great! With your environment and database set up, let's take a look at how you can monitor and optimize this database!

Exercise 2: Monitor Your Database

Database monitoring refers to reviewing the operational status of your database and maintaining its health and performance. With proper and proactive monitoring, databases will be able to maintain a consistent performance. Any problems that emerge, such as sudden outages, can be identified and resolved in a timely manner.

Tools such as pgAdmin, an open source graphical user interface (GUI) tool for PostgreSQL, come with several features that can help monitor your database. The main focus in this lab will be using the command line interface to monitor your database, but we'll also take a quick look at how the monitoring process can be replicated in pgAdmin.

Monitoring these statistics can be helpful in understanding your server and its databases, detecting any anomalies and problems that may arise.

Task A: Monitor Current Activity

To start, let's take a look at how you can monitor current server and database activity in PostgreSQL.

Server Activity

You can take a look at the server activity by running the following query:

```
1. 1
1. SELECT pid, username, datname, state, state_change FROM pg_stat_activity;
```

Copied!

This query will retrieve the following:

Column	Description
pid	Process ID
username	Name of user logged in
datname	Name of database
state	Current state, with two common values being: active (executing a query) and idle (waiting for new command)
state_change	Time when the state was last changed

This information comes from the **pg_stat_activity**, one of the built-in statistics provided by PostgreSQL.

1. Copy the query and paste it into the terminal.

   You should see the following output:

```
demo=# SELECT pid, username, datname, state, state_change FROM pg_stat_activity;
 pid  | username  | datname  | state  |         state_change
------+-----------+----------+--------+-------------------------------
   42 |           |          |        |
   44 | postgres  |          |        |
   51 | postgres  | postgres | idle   | 2021-10-13 22:11:20.330154+00
 1090 | postgres  | demo     | active | 2021-10-13 22:11:20.725355+00
   40 |           |          |        |
   39 |           |          |        |
   41 |           |          |        |
(7 rows)
```

   As you can see, there are currently 7 active connections to the server, with two of them being connected to databases that you're familiar with. After all, you started in the default **postgres** database, which is now idle, and now you're actively querying in the **demo** database.

2. To see what other columns are available for viewing, feel free to take a look at the pg_stat_activity documentation!

   Let's say you wanted to see all the aforementioned columns, in addition to the actual text of the query that was last executed. Which column should you add to review that?

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   If you wanted to see which query was most recently executed, you can add the **query** column.

```
1. 1
```

```
1. SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity;
```
Copied!

This column returns the most recent query. If **state** is active, it'll show the currently executed query. If not, it'll show the last query that was executed.

Your result should look similar to the following:

```
demo=# SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity;
 pid  | usename  | datname  | state  |         state_change          |                                   query
------+----------+----------+--------+-------------------------------+---------------------------------------------------------------------------
   42 |          |          |        |                               |
   44 | postgres |          |        |                               |
   51 | postgres | postgres | idle   | 2021-10-13 22:24:41.289228+00 | COMMIT
 1090 | postgres | demo     | active | 2021-10-13 22:24:42.068464+00 | SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity;
   40 |          |          |        |                               |
   39 |          |          |        |                               |
   41 |          |          |        |                               |
(7 rows)
```

Notice how for the **demo** database, with a status of **active**, the current query you are executing is the one listed in the query column.

Please note, if your table looks strange or squished, you can resize the terminal window by dragging it out.

If your result shows the text **(END)**, then type in `q` to exit that view. Whenever you encounter this view, you can use `q` to return to your original view.

3. With queries, you can apply filtering. What if you only wanted to see the states that were **active**? How would you do that?

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   To see which processes are **active**, you use the following query:

   ```
   1. 1
   1. SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity WHERE state = 'active';
   ```
   Copied!

   If you recall, there was only one active process with the **demo** database.

   You can confirm that with the following result:

   ```
   demo=# SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity WHERE state = 'active';
    pid | usename  | datname | state  |         state_change          |                                          query
   -----+----------+---------+--------+-------------------------------+-----------------------------------------------------------------------------------------------------
   1090 | postgres | demo    | active | 2021-10-13 23:23:49.659362+00 | SELECT pid, usename, datname, state, state_change, query FROM pg_stat_activity WHERE state = 'active';
   (1 row)
   ```

Database Activity

When looking at database activity, you can use the following query:

```
1. 1
1. SELECT datname, tup_inserted, tup_updated, tup_deleted FROM pg_stat_database;
```
Copied!

This query will retrieve the following:

Column	Description
datname	Name of database
tup_inserted	Number of rows inserted by queries in this database
tup_updated	Number of rows updated by queries in this database
tup_deleted	Number of rows deleted by queries in this database

This information comes from the **pg_stat_database**, one of the statistics provided by PostgreSQL.

1. Copy the query and paste it into the terminal.

   You should see the following output:

   ```
   demo=# SELECT datname, tup_inserted, tup_updated, tup_deleted FROM pg_stat_database;
     datname   | tup_inserted | tup_updated | tup_deleted
   ------------+--------------+-------------+-------------
               |            2 |           1 |           0
    postgres   |            0 |           0 |           0
    demo       |      2290162 |          22 |           0
    template1  |            0 |           0 |           0
    template0  |            0 |           0 |           0
   (5 rows)
   ```

   As you can see, the two databases that are returned are the **postgres** and **demo**. These are databases that you are familiar with.

   The other two, **template1** and **template0** are default templates for databases, and can be overlooked in this analysis.

   Based on this output, you now know that **demo** had about 2,290,162 rows inserted and 22 rows updated.

2. To see what other columns are available for viewing, you can read through the [pg_stat_database documentation](#).

   Let's say you wanted to see the number or rows fetched and returned by this database.

   Note: The number of rows fetched is the number of rows that were returned. The number of rows returned is the number of rows that were read and scanned by the query.

   What query should you use to do that?

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   To see the number of rows fetched and returned, you use the following query:

   ```
   1. 1
   1. SELECT datname, tup_fetched, tup_returned FROM pg_stat_database;
   ```
   Copied!

   Your result should look similar to the following:

   ```
   demo=# SELECT datname, tup_fetched, tup_returned FROM pg_stat_database;
     datname   | tup_fetched | tup_returned
   ------------+-------------+--------------
               |        8513 |        65666
    postgres   |       76915 |      2554350
    demo       |      587944 |      7992392
    template1  |           0 |            0
    template0  |           0 |            0
   (5 rows)
   ```

   Notice how the rows returned tend to be greater than the rows fetched. If you consider how tables are read, this makes sense because not all the rows scanned may be the ones that are returned.

3. With queries, you can apply filtering. What if you only wanted to see the database details (rows inserted, updated, deleted, returned and fetched) for **demo**?

   ▾ Hint (Click Here)

   Recall that you can filter queries with the `WHERE` clause.

   ▾ Solution (Click Here)

   To filter the results so that only those from the **demo** database are shown, you use the following query:

   ```
   1. 1
   1. SELECT datname, tup_inserted, tup_updated, tup_deleted, tup_fetched, tup_returned FROM pg_stat_database WHERE datname = 'demo';
   ```
   Copied!

   Your result should look similar to the following:

   ```
   demo=# SELECT datname, tup_inserted, tup_updated, tup_deleted, tup_fetched, tup_returned FROM pg_stat_database WHERE datname = 'demo';
    datname | tup_inserted | tup_updated | tup_deleted | tup_fetched | tup_returned
   ---------+--------------+-------------+-------------+-------------+--------------
    demo    |      2290162 |          22 |           0 |      588014 |      7998912
   (1 row)
   ```

Later, we'll take a look at how you can monitor these activities in pgAdmin.

Task B: Monitor Performance Over Time

Extensions, which can enhance your PostgreSQL experience, can be helpful in monitoring your database. One such extension is **pg_stat_statements**, which gives you an aggregated view of query statistics.

1. To enable the extension, enter the following command:

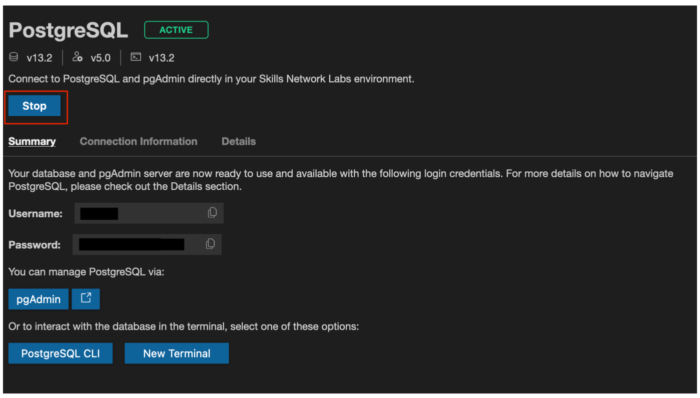
   ```
   1. 1
   1. CREATE EXTENSION pg_stat_statements;
   ```
   Copied!

   This will enable the **pg_stat_statements** extension, which will start to track the statistics for your database.

2. Now, let's edit the PostgreSQL configuration file to include the extension you just added:

   ```
   1. 1
   1. ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';
   ```
   Copied!

   For the changes to take effect, you will have to restart your database. You can do that by typing `exit` in the terminal to stop your current session. Close the terminal and return to the PostgreSQL tab. Select **Stop**.

   When the session has become **Inactive** once more, select **Start** to restart your session.

3. Once your session has started, open the **PostgreSQL CLI**.

   You'll need to reconnect to the **demo** database, which you can do by using the following command:

   ```
   1. 1
   1. \connect demo
   ```
   Copied!

4. You can see if this extension has been loaded by checking both the installed extensions and the **shared_preload_libraries**.

   First let's check the installed extensions:

   ```
   1. 1
   1. \dx
   ```
   Copied!

   ```
   demo=# \dx
                                        List of installed extensions
          Name        | Version |   Schema   |                              Description
   -------------------+---------+------------+------------------------------------------------------------------------
    pg_stat_statements | 1.8     | bookings   | track planning and execution statistics of all SQL statements executed
    plpgsql           | 1.0     | pg_catalog | PL/pgSQL procedural language
   (2 rows)
   ```

   Notice how **pg_stat_statements** has been installed.

   You can also check the **shared_preload_libraries** with:

   ```
   1. 1
   1. show shared_preload_libraries;
   ```
   Copied!

   ```
   demo=# show shared_preload_libraries;
    shared_preload_libraries
   --------------------------
    pg_stat_statements
   (1 row)
   ```

   pg_stat_statements is also shown under **shared_preload_libraries**.

5. Since the results returned by **pg_stat_statements** can be quite long, let's turn on expanded table formatting with the following command:

```
1. 1
1. \x
```

Copied!

This will display the output tables in an expanded table format.

```
demo=# \x
Expanded display is on.
demo=#
```

You can turn it off by repeating the `\x` command.

6. From the pg_stat_statements documentation, you'll see the various columns available to be retrieved.

Let's say you wanted to retrieve the database ID, the query, and total time that it took to execute the statement (in milliseconds).

▸ Hint (Click Here)
▾ Solution (Click Here)

Use the following query to extract the details you'd like to retrieve:

```
1. 1
1. SELECT dbid, query, total_exec_time FROM pg_stat_statements;
```

Copied!

You'll notice that you can scroll through the results, which may look similar to the following:

```
-[ RECORD 10 ]--+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
dbid            | 13442
query           | COMMIT
total_exec_time | 0.7378020000000001
-[ RECORD 11 ]--+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
dbid            | 16384
query           | SELECT e.extname AS "Name", e.extversion AS "Version", n.nspname AS "Schema", c.description AS "Description"
                                                                                                                  +
                | FROM pg_catalog.pg_extension e LEFT JOIN pg_catalog.pg_namespace n ON n.oid = e.extnamespace LEFT JOIN pg_catalog.pg_description c ON
 c.objoid = e.oid AND c.classoid = $1::pg_catalog.regclass                                                        +
                | ORDER BY 1
total_exec_time | 0.32195999999999997
-[ RECORD 12 ]--+-----------------------------------------------------------------------------------------------------------
-------------------------------------------------------------------------------------------------
dbid            | 16384
query           | show shared_preload_libraries
total_exec_time | 0.015136

(END)
```

Unlike **pg_stat_activity**, which showed the latest query that was run, **pg_stat_statements** shows an aggregated view of the queries that were run since the extension was installed.

7. What if you wanted to check which datbase name matches the database ID?

▸ Hint (Click Here)
▾ Solution (Click Here)

Use the following query to extract the database ID and database name:

```
1. 1
1. SELECT oid, datname FROM pg_database;
```

Copied!

```
demo=# SELECT oid, datname FROM pg_database
demo-# ;
-[ RECORD 1 ]------
oid     | 13442
datname | postgres
-[ RECORD 2 ]------
oid     | 16384
datname | demo
-[ RECORD 3 ]------
oid     | 1
datname | template1
-[ RECORD 4 ]------
oid     | 13441
datname | template0
```

Based on this, you can now see that database ID **16384** is the **demo** database. This makes sense because you performed the query `show shared_preload libraries` on the **demo** database, which appeared in **pg_stat_statements**.

It's important to note that adding these extensions can increase your server load, which may affect performance. If you need to drop the extension, you can achieve that with the following command:

```
1. 1
1. DROP EXTENSION pg_stat_statements;
```

Copied!

If you check the current extensions with `\dx`, you'll also see that **pg_stat_statements** no longer appears.

You should also reset the `shared_preload_libraries` in the configuration file:

```
1. 1
1. ALTER SYSTEM RESET shared_preload_libraries;
```

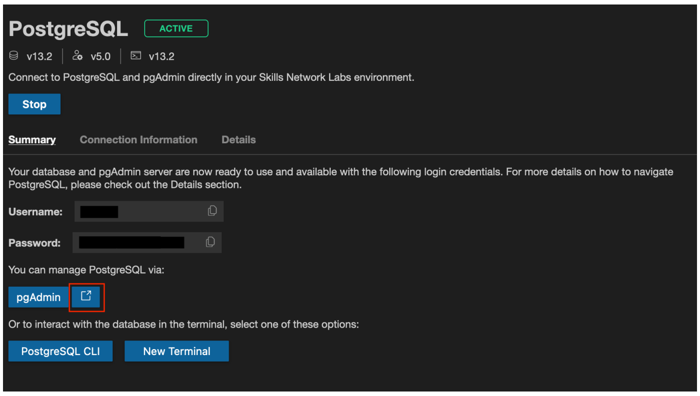
Copied!

After this, you'll need to exit the terminal and restart the PostgreSQL CLI to see the changes reflected in `show shared_preload_libraries;`.

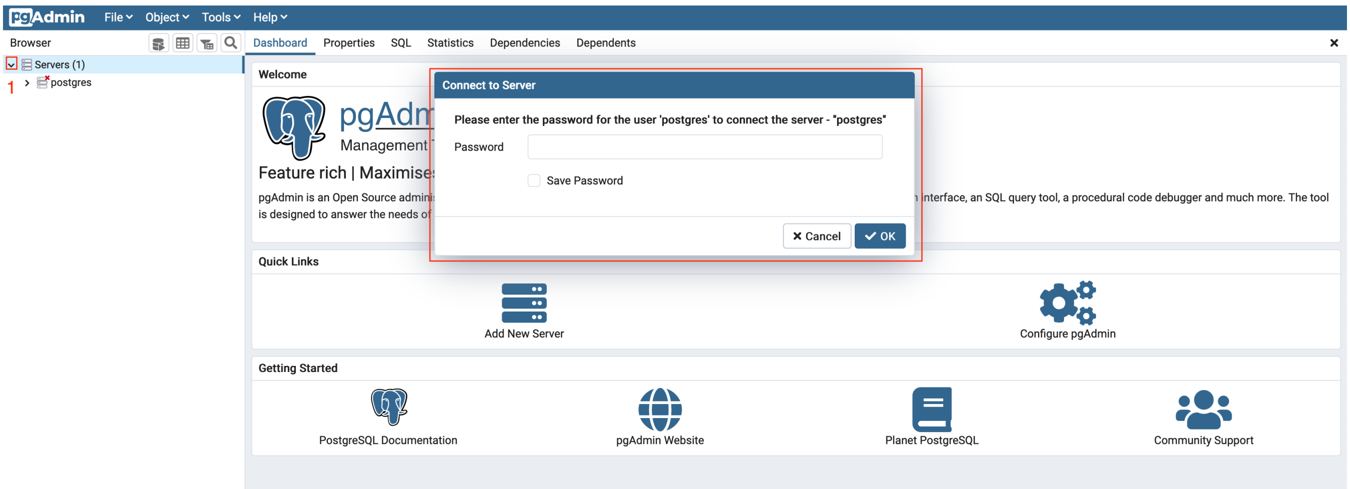
Task C: Monitor with pgAdmin

Another method of monitoring your database comes in the form of pgAdmin. In order to use this tool, you'll need to first launch it.

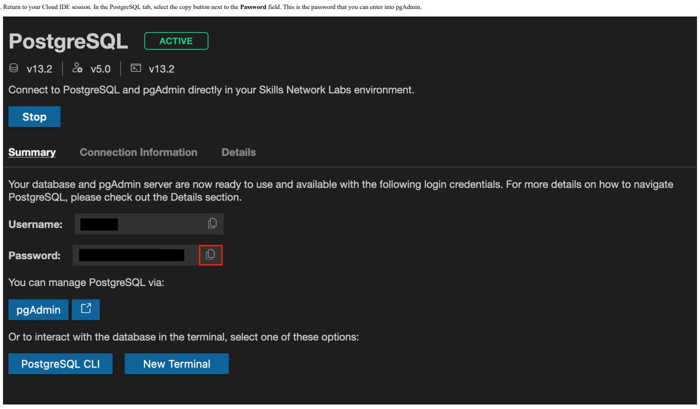
1. Open the PostgreSQL tab from the Skills Network Toolbox and select the pop-out button next to the **pgAdmin** button. This will open pgAdmin in a new tab.



2. In the left panel, select the dropdown next to **Servers**. You'll be prompted to enter a password.



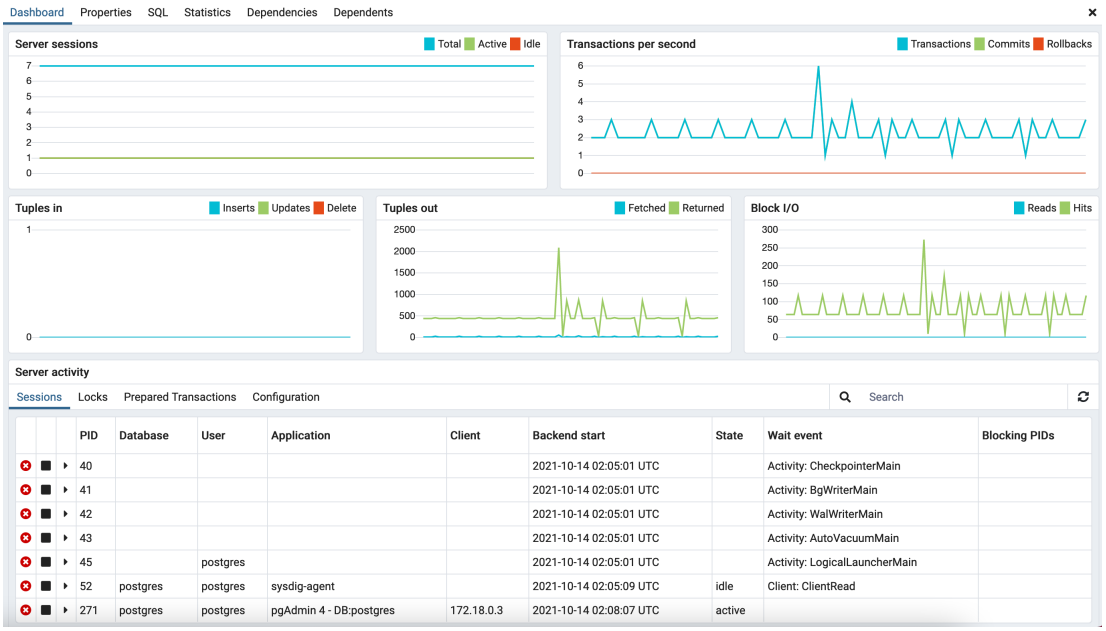
3. Return to your Cloud IDE session. In the PostgreSQL tab, select the copy button next to the **Password** field. This is the password that you can enter into pgAdmin.



Paste that password into pgAdmin. Then, click **OK**.

Your server will now load.

4. On the home page, under **Dashboard**, you will have access to server or database statistics, depending on which you are looking at.

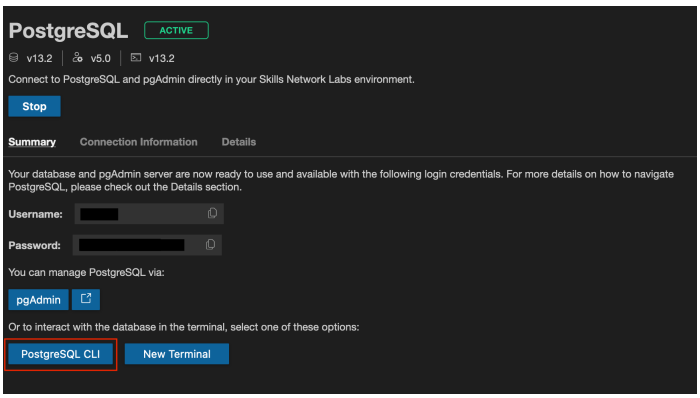


The table below lists the displayed statistics on the Dashboard that correspond with the statistics that you accessed with the CLI:

Chart	Description
Server/Database sessions	Displays the total sessions that are running. For servers, this is similar to the **pg_stat_activity**, and for databases, this is similar to the **pg_stat_database**.
Transactions per second	Displays the commits, rollbacks, and transactions taking place.
Tuples in	Displays the number of tuples (rows) that have been inserted, updated, and deleted, similar to the **tup_inserted**, **tup_updated**, and **tup_deleted** columns from **pg_stat_database**.
Tuples out	Displays the number of tuples (rows) that have been fetched (returned as output) or returned (read or scanned). This is similar to **tup_fetched** and **tup_returned** from **pg_stat_database**.
Server activity	Displays the sessions, locks, prepared transactions, and configuration for the server. In the Sessions tab, it offers a look at the breakdown of the sessions that are currently active on the server, similar to the view provided by **pg_stat_activity**. To check for any new processes, you can select the refresh button at the top-right corner.

5. You can test these charts out by starting another session.

Return to the tab with the Cloud IDE environment. On the PostgreSQL tab, select **PostgreSQL CLI**. This will start a new session of PostgreSQL with the CLI.



6. Once you have started that instance, switch back to the tab with pgAdmin.

What do you notice?

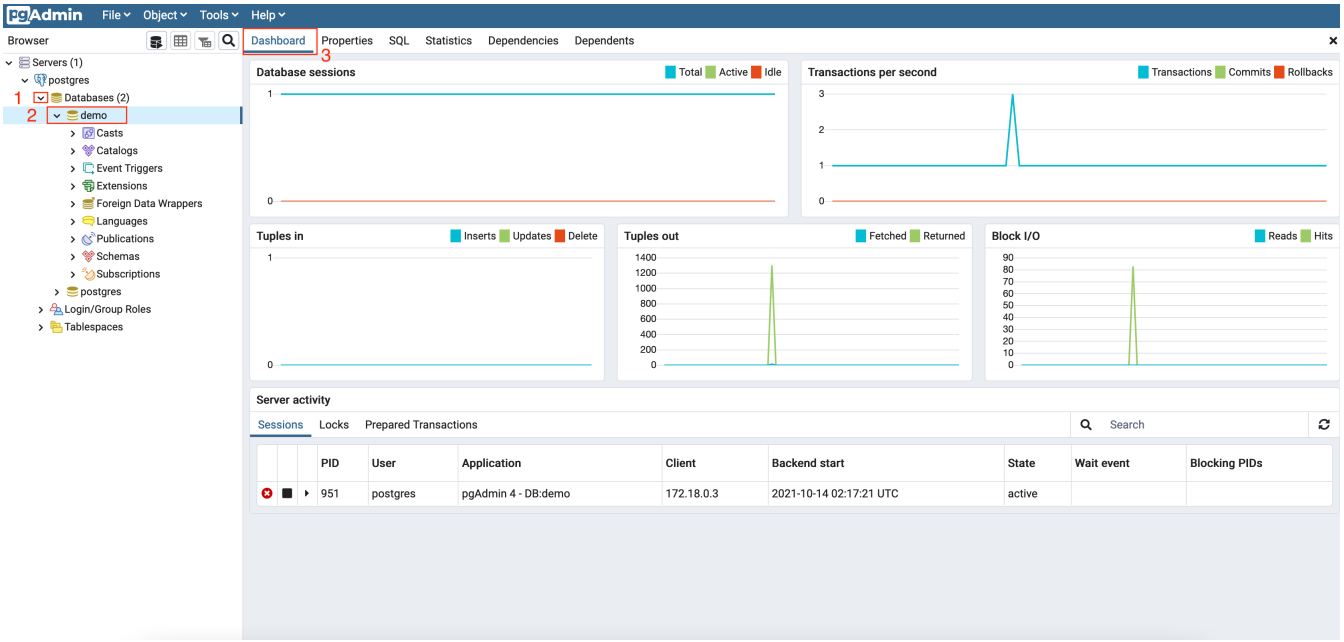
▸ Hint (Click Here)
▾ Solution (Click Here)

You may have noticed that the **Server sessions** saw an increase of sessions. It increased from 7 to 8 sessions. This makes sense since you started a new session with PostgreSQL CLI.

To see that change reflected in **Server Activity**, you'll have to click the refresh button to see that an additional **postgres** database session appeared.

7. To see the dashboard for the **demos** database, navigate to the left panel and select the **Databases** dropdown and then select the **demo** database to connect to it.

As you can see, similar statistics are displayed for the database.



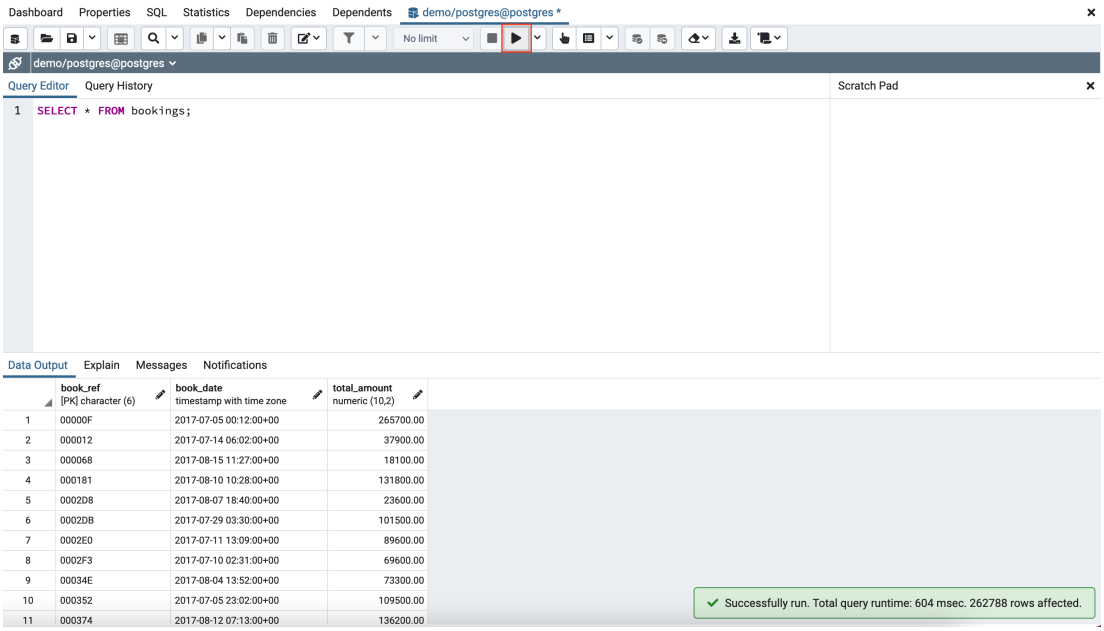
8. Let's run a query on the database! To do that, navigate to the menu bar and select `Tools > Query Tool`.

You can run any query. To keep things simple, let's run the following to select all the data from the **bookings** table:

```
1. 1
1. SELECT * FROM bookings;
```

Copied!

Select the run button. You will see that the query has successfully loaded.

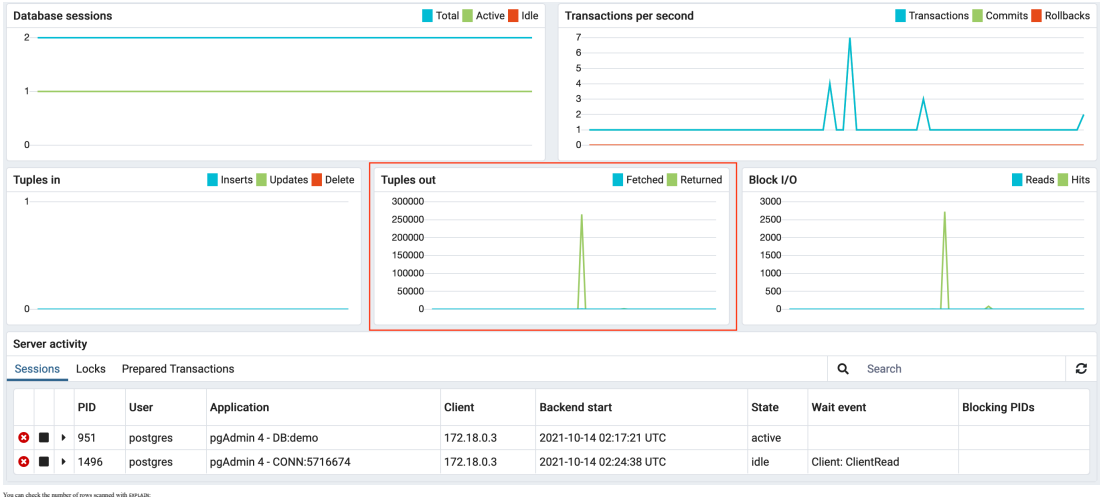


9. In pgAdmin, switch back to the database's **Dashboard** tab. You can refresh the **Server activity** and check to see if any of the charts have shown a spike since the data was retrieved.

What do you notice?

▸ Hint (Click Here)
▾ Solution (Click Here)

You may have noticed that the number of tuples (rows) returned (read/scanned) was greater than 250,000.



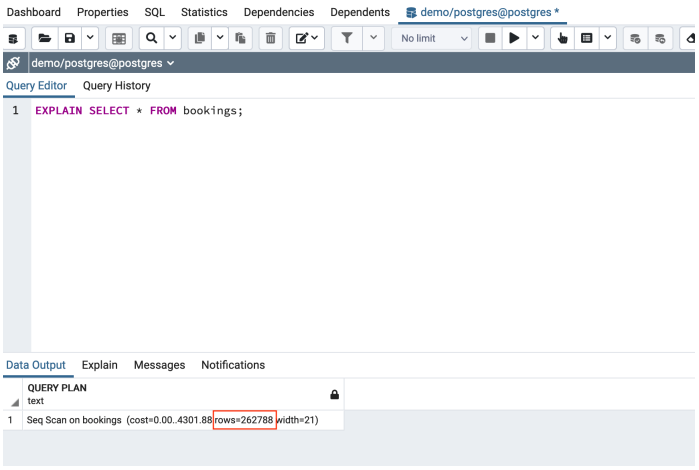
You can check the number of rows scanned with `EXPLAIN`:

```
1. 1
1. EXPLAIN SELECT * FROM bookings;
```

Copied!

If you can't see the full text, simply drag the **QUERY PLAN** column out.

This statement reveals that 262,788 rows were scanned, which is similar to the amount that was read/scanned based on the spike in the **Tuples out** chart.



While you can monitor your database through the command line alone, tools like pgAdmin can be helpful in providing a visual representation of how your server and its databases are performing.

PostgreSQL also offers logging capabilities to monitor and troubleshoot your lab, which will be further discussed in the Troubleshooting lab.

Exercise 3: Optimize Your Database

Data optimization is the maximization of the speed and efficiency of retrieving data from your database. Optimizing your database will improve its performance, whether that's inserting or retrieving data from your database. Doing this will improve the experience of anyone interacting with the database.

Similar to MySQL, there are optimal data types and maintenance (otherwise known as "vacuuming") that can be applied to optimize databases.

Task A: Optimize Data Types

When it comes to optimizing data types, understanding the data values will help in selecting the proper data type for the column.

Let's take a look at an example in the **demo** database.

1. Return to the CLI session that you opened previously (or open a new session if it has been closed).

   If you're no longer conected to the **demo** database, you can reconnect to it!

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   You can use the following command to connect to the **demo** database:

   ```
   1
   1. \connect demo
   ```
   Copied!

2. Let's list out the tables in the database with the following command:

   ```
   1
   1. \dt
   ```
   Copied!

```
demo=# \dt
                List of relations
  Schema  |       Name       | Type  |  Owner
----------+------------------+-------+----------
 bookings | aircrafts_data   | table | postgres
 bookings | airports_data    | table | postgres
 bookings | boarding_passes  | table | postgres
 bookings | bookings         | table | postgres
 bookings | flights          | table | postgres
 bookings | seats            | table | postgres
 bookings | ticket_flights   | table | postgres
 bookings | tickets          | table | postgres
(8 rows)
```

3. Now that you know which tables are in the database, select the first one, **aircrafts_data** and see what data you can pull from it. How can you select all of its data?

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   You can use the following query to select all the data from **aircrafts_data**:

   ```
   1
   1. SELECT * FROM aircrafts_data;
   ```
   Copied!

```
demo=# SELECT * FROM aircrafts_data;
 aircraft_code |              model               | range
---------------+----------------------------------+-------
 773           | {"en": "Boeing 777-300"}         | 11100
 763           | {"en": "Boeing 767-300"}         |  7900
 SU9           | {"en": "Sukhoi Superjet-100"}    |  3000
 320           | {"en": "Airbus A320-200"}        |  5700
 321           | {"en": "Airbus A321-200"}        |  5600
 319           | {"en": "Airbus A319-100"}        |  6700
 733           | {"en": "Boeing 737-300"}         |  4200
 CN1           | {"en": "Cessna 208 Caravan"}     |  1200
 CR2           | {"en": "Bombardier CRJ-200"}     |  2700
(9 rows)
```

   You can see that there are 9 entries in total with three columns: **aircraft_code**, **model**, and **range**.

4. For the purposes of this lab, we'll create a hypothetical situation that will potentially require changing the data types of columns to optimize them.

   Let's say that **aircraft_code** is always set to three characters, **model** will always be in a JSON format and **range** has a maximum value of 12,000 and minimum value of 1,000.

   In this case, what would be the best data types for each column?

   ▸ Hint (Click Here)
   ▾ Solution (Click Here)

   Based on the documentation, the following data types would be suitable for the following columns:

   1. **aircraft_code**: char(3), since you know that the aircraft code will always be fixed to three characters.
   2. **model**: json, which is a special data type that PostgreSQL supports.
   3. **range**: smallint, since the range of its numbers falls between -32,768 to 32,767.

•

You can check the current data types (and additional details such as the indexes and constraints) of the **aircrafts_data** table with the following:

```
1
1. \d aircrafts_data
```
Copied!

```
demo=# \d aircrafts_data;
                Table "bookings.aircrafts_data"
    Column     |     Type     | Collation | Nullable | Default
---------------+--------------+-----------+----------+---------
 aircraft_code | character(3) |           | not null |
 model         | jsonb        |           | not null |
 range         | integer      |           | not null |
Indexes:
    "aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)
Check constraints:
    "aircrafts_range_check" CHECK (range > 0)
Referenced by:
    TABLE "flights" CONSTRAINT "flights_aircraft_code_fkey" FOREIGN KEY (aircraft_code) REFERENCES aircrafts_data(aircraft_code)
    TABLE "seats" CONSTRAINT "seats_aircraft_code_fkey" FOREIGN KEY (aircraft_code) REFERENCES aircrafts_data(aircraft_code) ON DELETE CASCADE
```

•

Notice that most of the columns in this table have been optimized for our sample scenario, except for the **range**. This may be because the range was unknown in the original database.

For this lab, let's take the opportunity to optimize that column for your hypothetical situation. You can do this by changing the data type of the column.

Please note that in this lab you'll first need to drop a view, which is another way our data can be presented, in order to change the column's data type. Otherwise, you will encounter an error. This is a special case for this database because you loaded a SQL file that included commands to create views. In your own database, you may not need to drop a view.

To drop the **aircrafts** view, use the following command:

```
1
1. DROP VIEW aircrafts;
```
Copied!

•

To change the column's data type, you'll use the following command:

```
1
1. ALTER TABLE aircrafts_data ALTER COLUMN range TYPE smallint;
```
Copied!

aircrafts_data is the table you want to change and **range** is the column you want to change to data type **smallint**.

•

Now, let's check the table's columns and data types again!

▸ Hint (Click Here)
▾ Solution (Click Here)

With the following command, you can check the columns and data types of the **aircrafts_data** table:

```
1
1. \d aircrafts_data
```
Copied!

```
demo=# \d aircrafts_data;
                Table "bookings.aircrafts_data"
    Column     |     Type     | Collation | Nullable | Default
---------------+--------------+-----------+----------+---------
 aircraft_code | character(3) |           | not null |
 model         | jsonb        |           | not null |
 range         | smallint     |           | not null |
Indexes:
    "aircrafts_pkey" PRIMARY KEY, btree (aircraft_code)
Check constraints:
    "aircrafts_range_check" CHECK (range > 0)
Referenced by:
    TABLE "flights" CONSTRAINT "flights_aircraft_code_fkey" FOREIGN KEY (aircraft_code) REFERENCES aircrafts_data(aircraft_code)
    TABLE "seats" CONSTRAINT "seats_aircraft_code_fkey" FOREIGN KEY (aircraft_code) REFERENCES aircrafts_data(aircraft_code) ON DELETE CASCADE
```

You can see that the data type has successfully been changed, optimizing your table in this hypothetical situation.

Task B: Vacuum Your Databases

In your day-to-day life, you can vacuum our rooms to keep them neat and tidy. You can do the same with databases by maintaining and optimizing them with some vacuuming.

In PostgreSQL, vacuuming means to clean out your databases by reclaiming any storage from "dead tuples", otherwise known as rows that have been deleted but have not been cleaned out.

Generally, the **autovacuum** feature is automatically enabled, meaning that PostgreSQL will automate the vacuum maintenance process for you.

You can check if this is enabled with the following command:

```
1. 1
```

```
1. show autovacuum;
```

Copied!

```
demo=# show autovacuum;
 autovacuum
------------
 on
(1 row)
```

As you can see, **autovacuum** is enabled.

Since **autovacuum** is enabled, let's check to see when your database was last vacuumed.

To do that, you can use the **pg_stat_user_tables**, which displays statistics about each table that is a user table (instead of a system table) in the database. The columns that are returned are the same ones listed in pg_stat_all_tables documentation.

What if you wanted to check the table (by name), the estimated number of dead rows that it has, the last time it was autovacuumed, and how many times it has been autovacuumed?

▸ Hint (Click Here)
▾ Solution (Click Here)

To select the table name, number of dead rows, the last time it was autovacuumed, and the number of times this table has been autovacuumed, you can use the following query:

```
1. 1
```

```
1. SELECT relname, n_dead_tup, last_autoanalyze, autovacuum_count FROM pg_stat_user_tables;
```

Copied!

```
demo=# SELECT relname, n_dead_tup, last_autoanalyze, autovacuum_count FROM pg_stat_user_tables;
    relname     | n_dead_tup |       last_autoanalyze        | autovacuum_count
----------------+------------+-------------------------------+------------------
 tickets        |          0 | 2021-10-14 15:14:22.586399+00 |                1
 boarding_passes |          0 | 2021-10-14 15:14:19.709957+00 |                1
 seats          |          0 | 2021-10-14 15:14:19.967282+00 |                1
 aircrafts_data |          0 |                               |                0
 flights        |          0 | 2021-10-14 15:14:19.926375+00 |                1
 ticket_flights |          0 | 2021-10-14 15:14:16.945501+00 |                1
 bookings       |          0 | 2021-10-14 15:14:22.842128+00 |                1
 airports_data  |          0 | 2021-10-14 15:14:19.946403+00 |                0
(8 rows)
```

Notice that you currently don't have any "dead tuples" (deleted rows that haven't yet been cleaned out) and so far, these tables have been autovacuumed once. This makes sense given that the database was just created and based on the logs, autovacuumed then.

Conclusion

Congratulations! Now, not only do you know how to monitor and optimize your database with the CLI, but you can also do so with pgAdmin. You will now be able to apply this knowledge to any PostgreSQL databases you create and modify in the future.

Author(s)

Kathy An

Other Contributor(s)

Changelog

Date	Version	Changed by	Change Description
2021-10-14	1.0	Kathy An	Created initial version
2023-05-04	1.1	Rahul Jaideep	Updated Markdown file