PostgreSQL Instance Configuration and System Catalog

Estimated time needed: 30 minutes

In this lab, you will obtains hands-on experience in customizing the configuration of a PostgreSQL server instance, both through the command line interface (CLI) and by editing the configuration files. Furthermore, you will learn to navigate and query the PostgreSQL system catalog, which is a series of tables that store metadata about objects in the database.

Objectives

After completing this lab, you will be able to:

- Customize the configuration parameters of your PostgreSQL server instance
- Query the system catalog to retrieve metadata about database objects

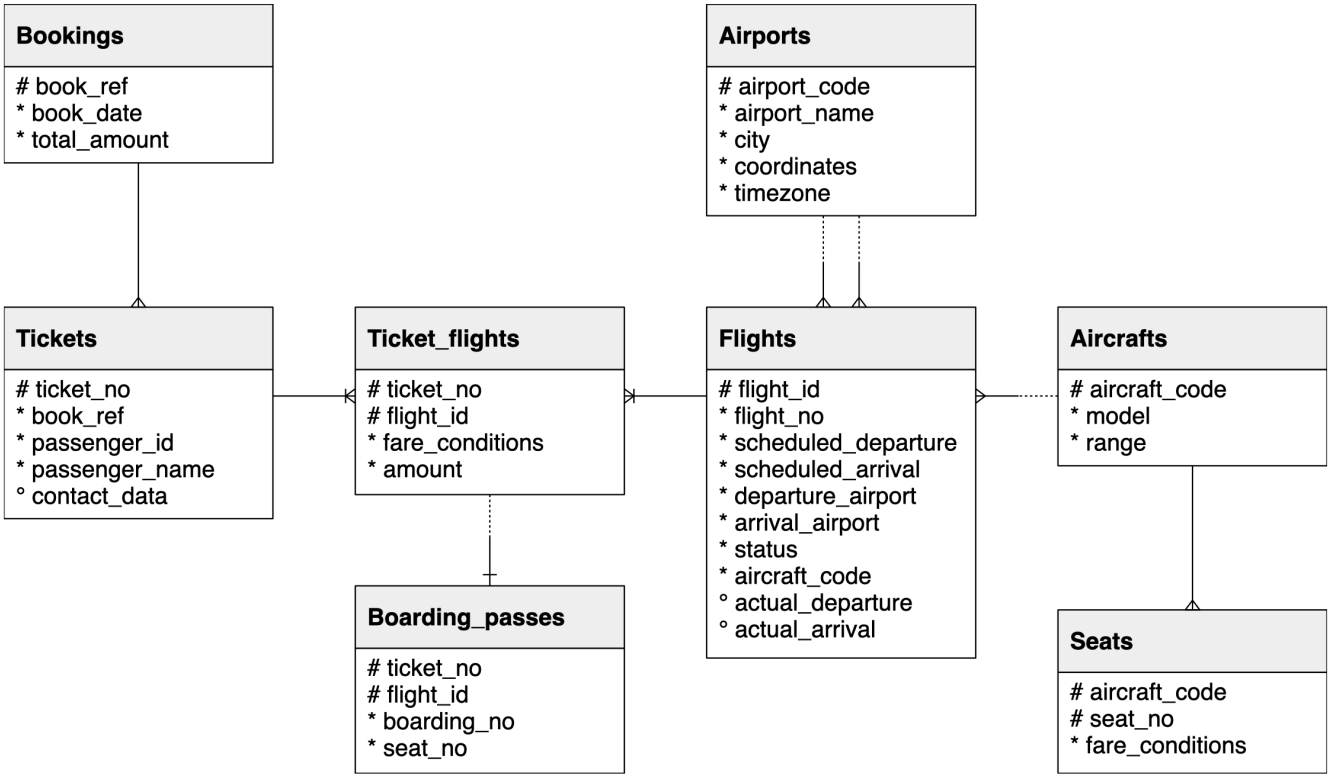
Software Used in This Lab

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

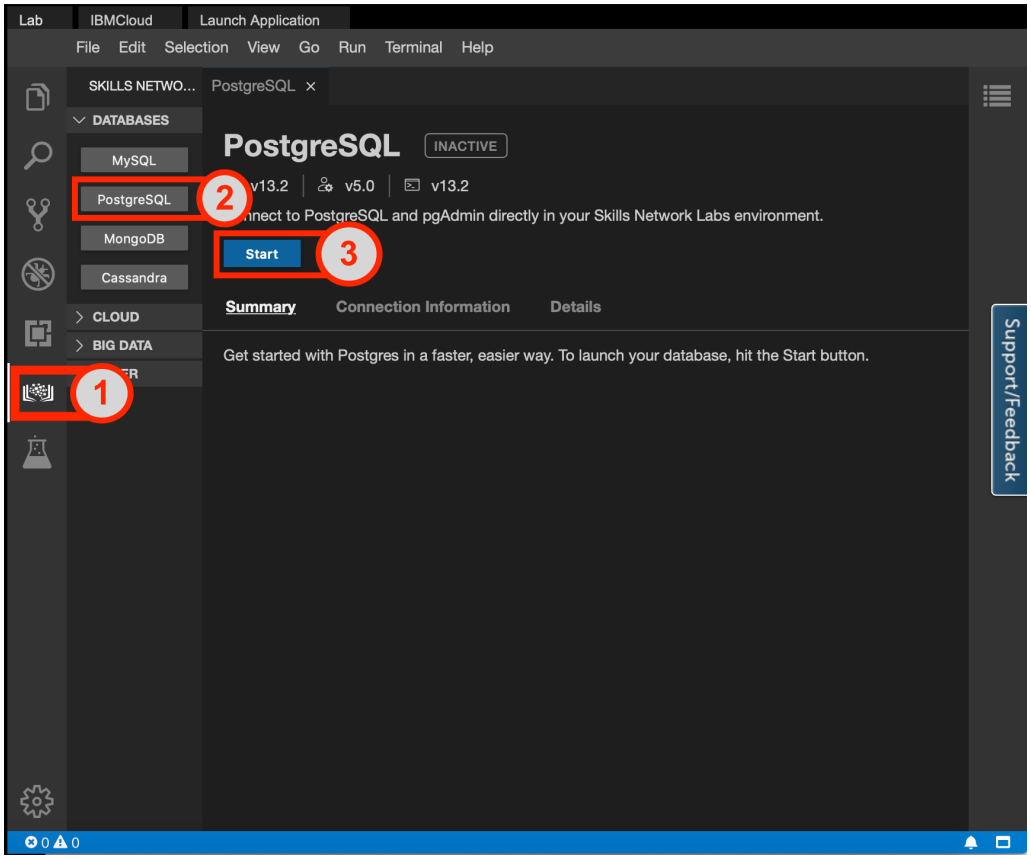
In this lab, you will use a database from https://postgrespro.com/education/demodb distributed under the PostgreSQL licence. It stores a month of data about airline flights in Russia and is organized according to the following schema:



Launching PostgreSQL in Cloud IDE

To get started with this lab, launch PostgreSQL using the Cloud IDE. You can do this by following these steps:

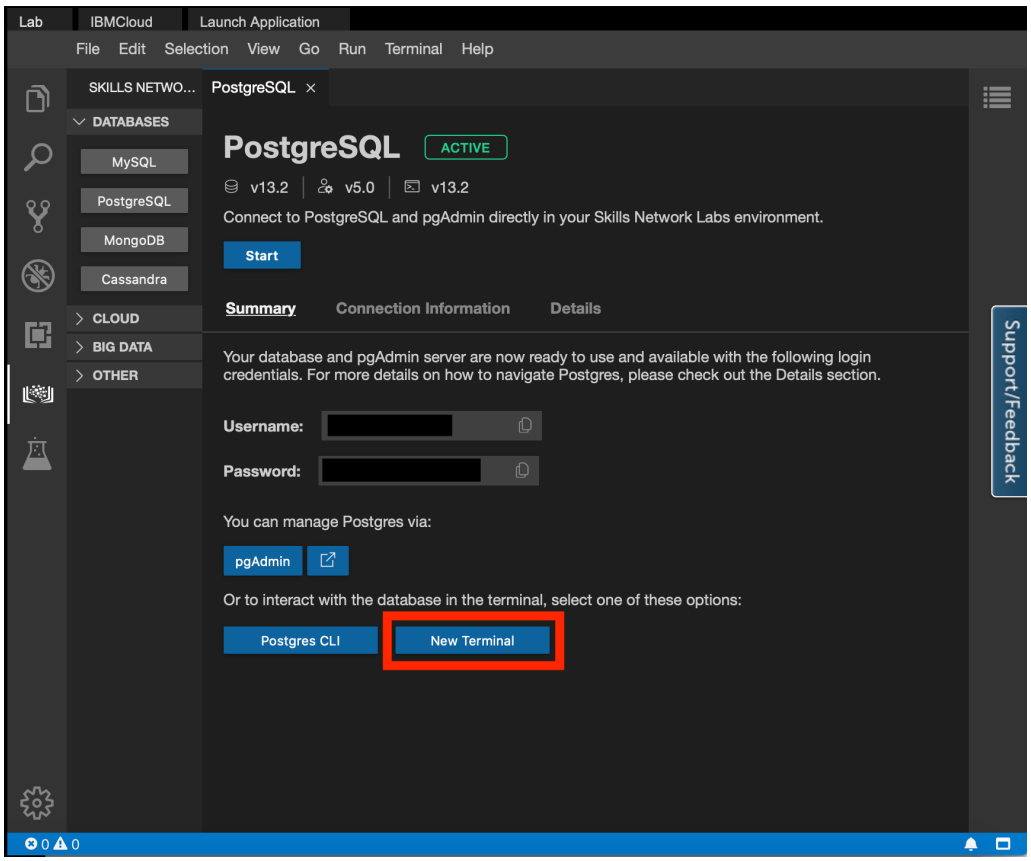
1. Click the Skills Network extension button in the left pane.
2. Open the “DATABASES” drop-down menu and click “PostgreSQL”
3. Click the “Start” button. PostgreSQL may take a few moments to start.



Downloading and Creating the Database

First, you will need to download the database.

1. Open a new terminal by clicking the "New Terminal" button near the bottom of the interface.



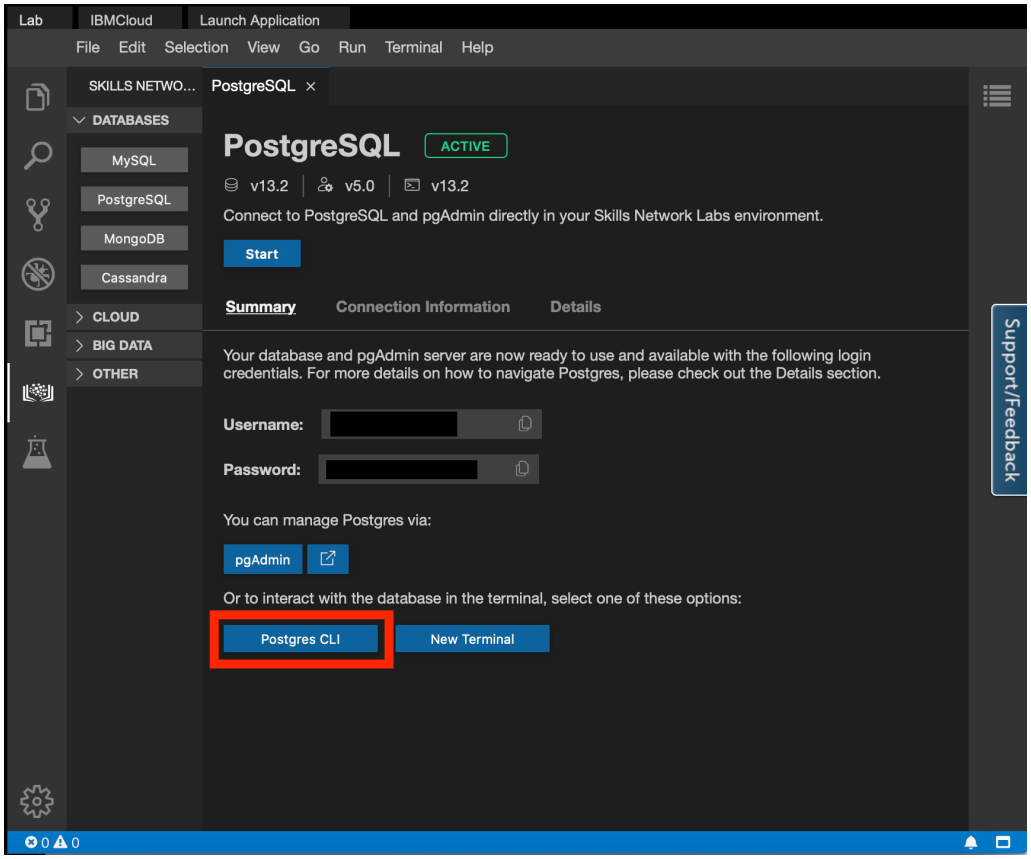
2. Run the following command in the terminal:

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

Copied!

The file you downloaded is a full database backup of a month of flight data in Russia. Now, you can perform a full restoration of the data set by first opening the PostgreSQL CLI.

3. Near the bottom of the window, click the "Postgres CLI" button to launch the command line interface.



4. In the PostgreSQL CLI, enter the command `\i <file_name>`. In your case, the file name will be the name of the file you downloaded, `flights_RUSSIA_small.sql`. This will restore the data into a new database called `demo`.

```
1. 1
1. \i flights_RUSSIA_small.sql
```
Copied!

The restorations may take a few moments to complete.

5. Verify that the database was properly created by entering the following command:

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

You should see the following output showing all the tables that are part of the `bookings` schema in the `demo` database.

Exercise 1: Configure Your PostgreSQL Server Instance

A PostgreSQL server instance has a corresponding file named `postgresql.conf` that contains the configuration parameters for the server. By modifying this file, you can enable, disable, or otherwise customize the settings of your PostgreSQL server instance to best suit your needs as a database administrator. While you can manually modify this `postgresql.conf` file and restart the server for the changes to take effect, you can also edit some configuration parameters directly from the command line interface (CLI).

In this exercise, you will customize the configuration settings for the PostgreSQL instance using the CLI.

1. First, let's take a look at the current setting of the `wal_level` parameter. You can do so by entering the following command into the CLI:

```
1. 1
1. SHOW wal_level;
```
Copied!

Without going into too much detail, the `wal_level` parameter dictates how much information is written to the write-ahead log (WAL), which can be used for continuous archiving. If you're interested, you can find further information in the PostgreSQL official documentation.

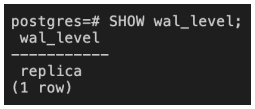
2. The `ALTER SYSTEM` command is a way to modify the global defaults of a PostgreSQL instance without having to manually edit the configuration file. Let's give it a try and change the `wal_level` parameter to `logical`. To change the parameter, enter the following command into the CLI:

```
1. 1
1. ALTER SYSTEM SET wal_level = 'logical';
```
Copied!

3. **Try it yourself:** Use the CLI to check the current setting of `wal_level`.

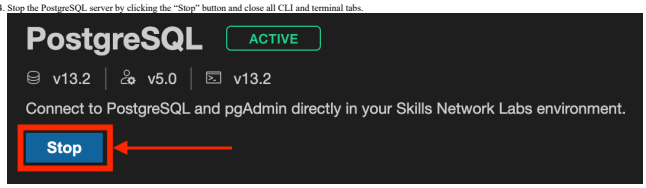
▸ **Hint** (Click Here)
▾ **Solution** (Click Here)

```
1. 1
1. SHOW wal_level;
```
Copied!

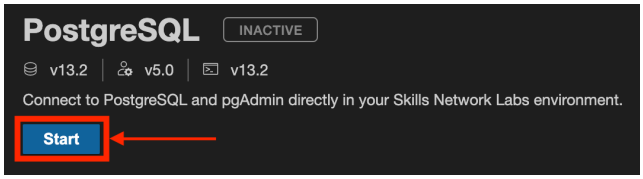


In Step 2, you changed the `wal_level` parameter from `replica` to `logical` yet the command you just entered shows that the parameter is still set to `replica`. Why would this be? It turns out that some configuration parameters require a server restart before any changes take effect - the `wal_level` is one such parameter.

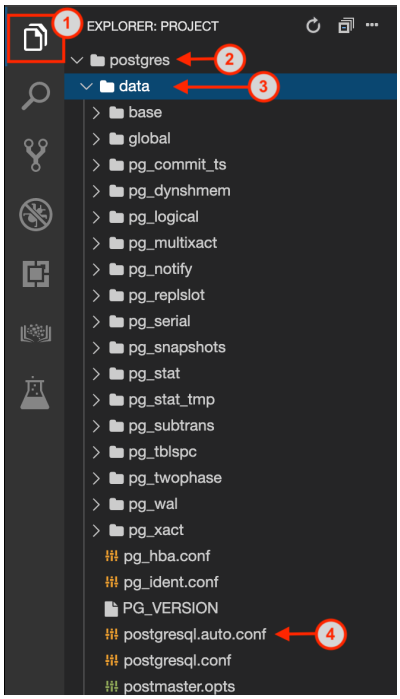
4. Stop the PostgreSQL server by clicking the "Stop" button and close all CLI and terminal tabs.

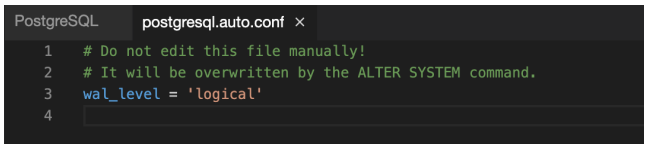


5. Now restart the PostgreSQL server by clicking the "Start" button. It may take a few moments to start up again. When does it so, reopen the PostgreSQL CLI.



6. When you executed the `ALTER SYSTEM` command in Step 2 of this exercise, a new file named `postgres.auto.conf` was created. You can open the file by first opening the file explorer on Cloud IDE then clicking `postgres` > `data` > `postgresql.auto.conf`.





This file was automatically modified to contain the new parameter you set using the `ALTER SYSTEM` command in Step 2. When you started up the PostgreSQL server again, it will read this file and set the `wal_level` parameter to `logical`.

7. Finally, and for the last time in this lab, let's confirm the current setting of the `wal_level` parameter. Enter the following into the CLI:

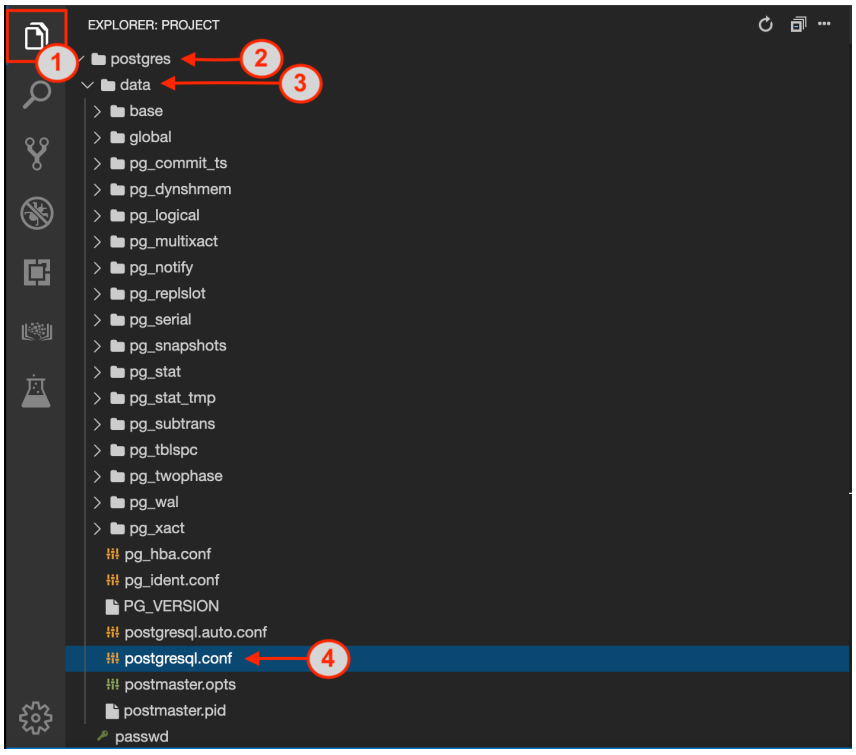
```
1. 1
1. SHOW wal_level;
```

Copied!

```
postgres=# SHOW wal_level;
 wal_level
-----------
 logical
(1 row)
```

You can see that the parameter was changed successfully after the restart.

8. For more advanced instance configuration where many parameter changes are required, using a series of `ALTER SYSTEM` commands may be cumbersome. Instead, you can edit the `postgresql.conf` file directly. You can once again use the Cloud IDE file explorer to open `postgres` > `data` > `postgresql.conf`.



You can edit the configuration file right in the Cloud IDE file explorer. ![view of postgresql.conf](https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/IBM-DB0231EN-SkillsNetwork/labs/PostgreSQL/Lab%20-%20PostgreSQL%20Instance%20Configuration%20and%20System%20Catalog/images/config_6.png)

Exercise 2: Navigate the System Catalog

The system catalog stores schema metadata, such as information about tables and columns and internal bookkeeping information. In PostgreSQL, the system catalogs are regular tables in which you can add columns and insert and update values. In directly modifying the system catalogs, you can cause severe problems in your system, so it is generally recommended to avoid doing so. Instead, the system catalogs are updated automatically when performing other SQL commands. For example, if you run a `CREATE DATABASE` command, a new database is created on the disk and a new row is automatically inserted into the `pg_database` system catalog table, storing metadata about that database.

Let's explore some of the system catalog tables in PostgreSQL.

1. Start with a simple query of `pg_tables`, which is a system catalog containing metadata about each table in the database. Let's query it to display metadata about all the tables belonging to the `bookings` schema in the `demo` database by entering the following command into the CLI:

```
1. 1
1. SELECT * FROM pg_tables WHERE schemaname = 'bookings';
```
Copied!

```
demo=# SELECT * FROM pg_tables WHERE schemaname = 'bookings';
 schemaname |    tablename     | tableowner | tablespace | hasindexes | hasrules | hastriggers | rowsecurity
------------+------------------+------------+------------+------------+----------+-------------+-------------
 bookings   | ticket_flights   | postgres   |            | t          | f        | t           | f
 bookings   | boarding_passes  | postgres   |            | t          | f        | t           | f
 bookings   | aircrafts_data   | postgres   |            | t          | f        | t           | f
 bookings   | flights          | postgres   |            | t          | f        | t           | f
 bookings   | airports_data    | postgres   |            | t          | f        | t           | f
 bookings   | seats            | postgres   |            | t          | f        | t           | f
 bookings   | tickets          | postgres   |            | t          | f        | t           | f
 bookings   | bookings         | postgres   |            | t          | f        | t           | f
(8 rows)
```

As you can see, the 8 tables belonging to the `bookings` schema are displayed with various pieces of metadata, such as the table owner and other parameters.

2. Suppose as the database administrator, you would like to enable row-level security for the `boarding_passes` table in the `demo` database. When row security is enabled on a table, all normal access to the table for selecting or modifying rows must be specified by a row security policy. Since row security policies are not the focus of this lab, we will not go in depth about specifying a policy but will simply enable it for demonstration purposes. However, if you wish to learn more about this topic, you can check out the PostgreSQL documentation. To enable row security on the `boarding_passes` table, enter the following command in the CLI:

```
1. 1
1. ALTER TABLE boarding_passes ENABLE ROW LEVEL SECURITY;
```
Copied!

3. **Try it yourself:** Use the CLI to query the `pg_tables` to display metadata about the tables belonging to the `bookings` schema and confirm that the row security for the `boarding_passes` was successfully enabled.

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

```
1. 1
1. SELECT * FROM pg_tables WHERE schemaname = 'bookings';
```
Copied!

```
demo=# ALTER TABLE boarding_passes ENABLE ROW LEVEL SECURITY;
ALTER TABLE
demo=# SELECT * FROM pg_tables WHERE schemaname = 'bookings';
 schemaname |    tablename     | tableowner | tablespace | hasindexes | hasrules | hastriggers | rowsecurity
------------+------------------+------------+------------+------------+----------+-------------+-------------
 bookings   | boarding_passes  | postgres   |            | t          | f        | t           | t
 bookings   | ticket_flights   | postgres   |            | t          | f        | t           | f
 bookings   | aircrafts_data   | postgres   |            | t          | f        | t           | f
 bookings   | flights          | postgres   |            | t          | f        | t           | f
 bookings   | airports_data    | postgres   |            | t          | f        | t           | f
 bookings   | seats            | postgres   |            | t          | f        | t           | f
 bookings   | tickets          | postgres   |            | t          | f        | t           | f
 bookings   | bookings         | postgres   |            | t          | f        | t           | f
(8 rows)
```

As you can see, the `boarding_passes` has `t`, for "true", under the `rowsecurity` column, which tells us that the row security was enabled successfully.

4. Let's connect your work in the previous section about PostgreSQL instance configuration to the system catalogs. Earlier, you used `SHOW` statements to display configuration parameters. There's also a system catalog called `pg_settings` that stores data about configuration parameters of the PostgreSQL server. Let's query with the following command:

```
1. 1
1. SELECT name, setting, short_desc FROM pg_settings WHERE name = 'wal_level';
```
Copied!

```
postgres=# SELECT name, setting, short_desc FROM pg_settings WHERE name = 'wal_level';
   name    | setting |                  short_desc
-----------+---------+-----------------------------------------------
 wal_level | logical | Set the level of information written to the WAL.
(1 row)
```

From the query, you see the same results from the `SHOW` statement in Exercise 1 and more. In fact, `pg_tables` contains much more data about a given parameter than is available from the `SHOW` statement (a full list can be found in the documentation) so, the somewhat more complicated SQL query has its benefits.

Exercise 3: Try it yourself!

Now that you have seen some examples of configuring a PostgreSQL instance and navigating the system catalogs, it's time to put what you learned to use and give it a go yourself.

In this practice exercise, suppose you wanted to change the name of the `aircrafts_data` to `aircraft_fleet`.

1. **Try it yourself:** First, try changing the name of the table by directly editing the `pg_tables` table from the system catalogs.

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

```
1. 1
1. UPDATE pg_tables SET tablename = 'aircraft_fleet' WHERE tablename = 'aircrafts_data';
```
Copied!

```
demo=# UPDATE pg_tables SET tablename = 'aircraft_fleet' WHERE tablename = 'aircrafts_data';
ERROR:  cannot update view "pg_tables"
DETAIL:  Views that do not select from a single table or view are not automatically updatable.
HINT:  To enable updating the view, provide an INSTEAD OF UPDATE trigger or an unconditional ON UPDATE DO INSTEAD rule.
```

As you can see, the SQL command to update a table from the system catalog directly results in an error. This is a good safeguard for you as a database administrator since as discussed earlier in the lab, changing individual values in a system catalog directly can severely mess up your database. Let's try a different approach.

2. To properly change the name of the `aircrafts_data`, enter the following command in the CLI:

```
1. 1
1. ALTER TABLE aircrafts_data RENAME TO aircraft_fleet;
```
Copied!

3. **Try it yourself:** To confirm that the table was successfully renamed, query `pg_tables` from the system catalog by `schemaname` 'bookings' to display the `tablename` column.

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

```
1. 1
1. SELECT tablename FROM pg_tables WHERE schemaname = 'bookings';
```
Copied!

```
demo=# ALTER TABLE aircrafts_data RENAME TO aircraft_fleet;
ALTER TABLE
demo=# SELECT tablename FROM pg_tables WHERE schemaname = 'bookings';
    tablename
-----------------
 ticket_flights
 boarding_passes
 flights
 airports_data
 seats
 tickets
 bookings
 aircraft_fleet
(8 rows)
```

As you can see, the table was successfully renamed to `aircraft_fleet` and the changes are automatically reflected in the system catalog.

Conclusion

Congratulations on completing this lab on database adminstration with PostgreSQL! You now have some foundational knowledge on how to configure a PostgreSQL instance and customize it for your specific use cases. In addition, you now have the ability to query the system catalog to retrieve metadata on database objects and you are ready to move on to the next lesson.

Author

David Pasternak

Other Contributors

Sandip Saha Joy, Rav Ahuja

Changelog

Date	Version	Changed by	Change Description
2021-09-20	0.1	David Pasternak	Initial version created
2022-07-27	0.2	Lakshmi Holla	Updated html tag
2023-05-05	0.3	Jaskomal Natt	Updated copyright date and removed empty cells