1 Introduction

OmniDB is an open source browser-based app designed to access and manage many different Database Management systems, e.g. PostgreSQL, Oracle and MySQL. OmniDB can run either as an App or via Browser, combining the flexibility needed for various access paths with a design that puts security first. OmniDB is actively developed, automatically tested on a variety of databases and browsers and comes with full documentation.

Since early development, OmniDB was designed as an browser-based app. Consequently, it runs in any browser, from any operational system. It can be accessed by several computers and multiple users, each one of them with his/her own group of connections. It also can be hosted in any operational system, without the need of install any dependencies. We will see further details on installation in the next chapters.

OmniDB's main objective is to offer an unified workspace with all functionalities needed to manipulate different DMBS. DBMS specific tools aren't required: in OmniDB, the context switch between different DBMS is done with a simple connection switch, without leaving the same page. The end-user's sensation is that there is no difference when he/she manipulates different DBMS, it just feels like different connections.

Despite this, OmniDB is built with simplicity in mind, designed to be a fast and lightweight browser-based application. OmniDB is also powered by the WebSocket technology, allowing the user to execute multiple queries and procedures in multiple databases in multiple hosts in background.

OmniDB is also secure. All OmniDB user data are stored encrypted, and no database password is stored at all. When the user first connects to a database, OmniDB asks for the password. This password is encrypted and stored in memory for a specific amount of time. When this time expires, OmniDB asks the password again. This ensures maximum security for the database OmniDB is connecting to.

1.0.0.1 History

OmniDB's creators, Rafael Thofehrn Castro and William Ivanski, worked in a company where they needed to deal with several different databases from customers on a daily basis. These databases were from different DBMS technologies, and so they needed to keep switching between database management tools (typically one for each DBMS). As they were not keen of the existing unified database management tools (that could manage different DBMS), they came up with OmniDB's main idea.

OmniDB's first version was presented as an undergrad final project in the Computer Science Course from the Federal University of Paraná, in Brazil. The objective was to trace a common line between popular DBMS, and to study deeply their *metadata*. The result was a tool written in ASP.NET/C# capable of connecting and identifying the main structures (tables, keys, indexes and constraints), in a generic way, from several DBMS:

- Firebird
- MariaDB / MySQL
- Oracle
- PostgreSQL
- SQLite
- Microsoft SQL Server

OmniDB's first version also allowed the conversion between all DBMSs supported by the tool. This feature was developed to be user friendly, requiring just a few steps: the user needs to select a source connection, the structures that will be converted (just tables and all their structures, along with their data) and the target connection.

2 Installation

OmniDB provides 2 kinds of packages to fit every user needs:

- **OmniDB Application**: Runs a web server on a random port behind, and provides a simplified web browser window to use OmniDB interface without any additional setup. Just feels like a desktop application.
- **OmniDB Server**: Runs a web server on a random port. User needs to connect to it through a web browser. Provides user management, ideal to be hosted on a server on users' networks.

Both application and server can be installed on the same machine.

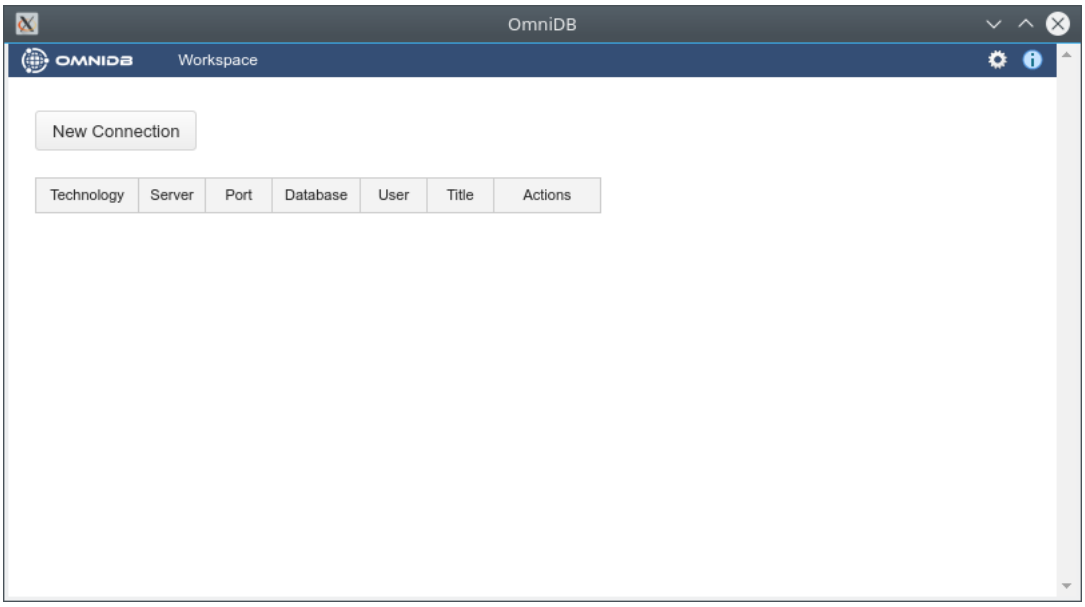
2.0.0.1 OmniDB Application

In order to run OmniDB app, you don't need to install any additional piece of software. Just head to omnidb.org and download the latest package for your specific operating system and architecture:

- Linux 32 bits / 64 bits
  - DEB installer
  - RPM installer
- Windows 32 bits / 64 bits
  - EXE installer

- Mac OSX
  - DMG installer

Use the specific installer for your Operational System and it will be available through your desktop environment application menu or via command line with `omnidb-app`.



2.0.0.2 OmniDB Server

Like OmniDB app, OmniDB server doesn't require any additional piece of software and the same options for operating system and architecture are provided.

Use the specific installer for your Operational System and it will be available through command line with `omnidb-server`:

```
user@machine:~$ omnidb-server
Starting OmniDB server...
Checking port availability...
Starting server OmniDB 2.4.0 at 0.0.0.0:8000.
Starting migration of user database from version 0.0.0 to version 2.4.0
OmniDB successfully migrated user database from version 0.0.0 to version 2.4.0
Press Ctrl+C to exit
```

Note how OmniDB starts a *web server* in port 8000. You can also specify web server port and listening address:

```
user@machine:~$ omnidb-server -p 8080 -H 127.0.0.1
Starting OmniDB server...
Checking port availability...
Starting server OmniDB 2.4.0 at 0.0.0.0:8080.
Starting migration of user database from version 0.0.0 to version 2.4.0
OmniDB successfully migrated user database from version 0.0.0 to version 2.4.0
Press Ctrl+C to exit
```

2.0.0.3 OmniDB with Oracle

OmniDB app and server does not require any piece of additional software, as explained above. But if you are going to connect to an *Oracle* database, then you need to download and extract *Oracle Instant Client* into OmniDB's folder (`/opt/omnidb-app` if you are using app version or `/opt/omnidb-server` if you are using server version) or into `~/lib`, depending on the operating system you use: - **MacOSX**: Download Oracle Instant Client (64-bit) and extract in `~/lib`; - **Linux**: Download Oracle Instant Client (32-bit) (64-bit) and extract it into OmniDB's folder; - **Windows**: Download Oracle Instant Client (32-bit) (64-bit) and extract it into OmniDB's folder.

If you already have an Oracle Instant Client installed and loaded into your lib path, then OmniDB will be able to use it, and you will not need to download it and extract it into a specific folder again.

2.0.0.4 OmniDB User Database

Since version 2.4.0, upon initialization both server and app will create a file `~/.omnidb/omnidb-app/omnidb.db` (for OmniDB app) or `~/.omnidb/omnidb-server/omnidb.db` (for OmniDB server) in the user home directory, if it does not exist. That can be confirmed by the message *OmniDB successfully migrated user database from version 0.0.0 to version 2.4.0* you saw above. This file is also called **user database** and contains user data. If it already exists, then OmniDB will check whether the version of the server matches the version of the user database:

```
user@machine:~$ omnidb-server
Starting OmniDB server...
Checking port availability...
Starting server OmniDB 2.4.0 at 0.0.0.0:8000.
User database version 2.4.0 is already matching server version.
Press Ctrl+C to exit
```

Future releases of OmniDB will contain the **user database migration** SQL commands required to upgrade the user database, if necessary. This way user data is not lost by upgrading OmniDB. Imagine the following scenario: you use OmniDB 2.4.0 now and you decide to upgrade it to newest release 2.5.0, for example. After the upgrade, when you start OmniDB server, it will apply the changes version 2.5.0 requires. So you will see something like that:

```
user@machine:~$ omnidb-server
Starting OmniDB server...
Checking port availability...
Starting server OmniDB 2.4.0 at 0.0.0.0:8080.
Starting migration of user database from version 2.4.0 to version 2.5.0
OmniDB successfully migrated user database from version 2.4.0 to version 2.5.0
Press Ctrl+C to exit
```

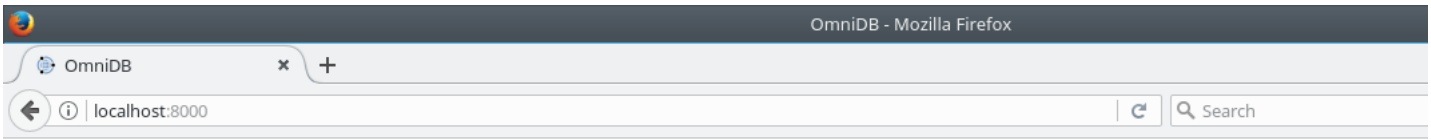
2.0.0.5 OmniDB configuration file

Starting on version 2.1.0, OmniDB server comes with a configuration file `omnidb.conf` that enables the user to specify parameters such as port and listening address. Also, 2.1.0 enables us to start the server with SSL, this requires a certificate and is configured in the same configuration file.

Starting on version 2.4.0, this file is located in `~/.omnidb/omnidb-server/omnidb.conf` in the user home directory.

2.0.0.6 OmniDB in the browser

Now that the web server is running, you may access OmniDB browser-based app on your favorite browser. Type in address bar: `localhost:8000` and hit `Enter`. If everything went fine, you shall see a page like this:

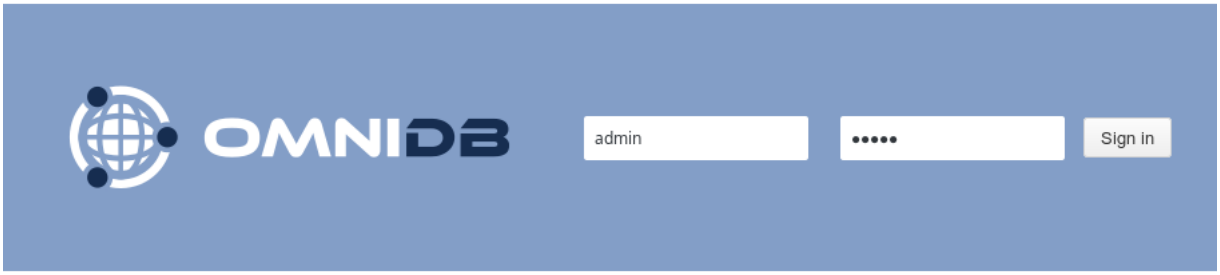


Now you know that OmniDB is running correctly. In the next chapters, we will see how to login for the first time, how to create an user and to utilize OmniDB.

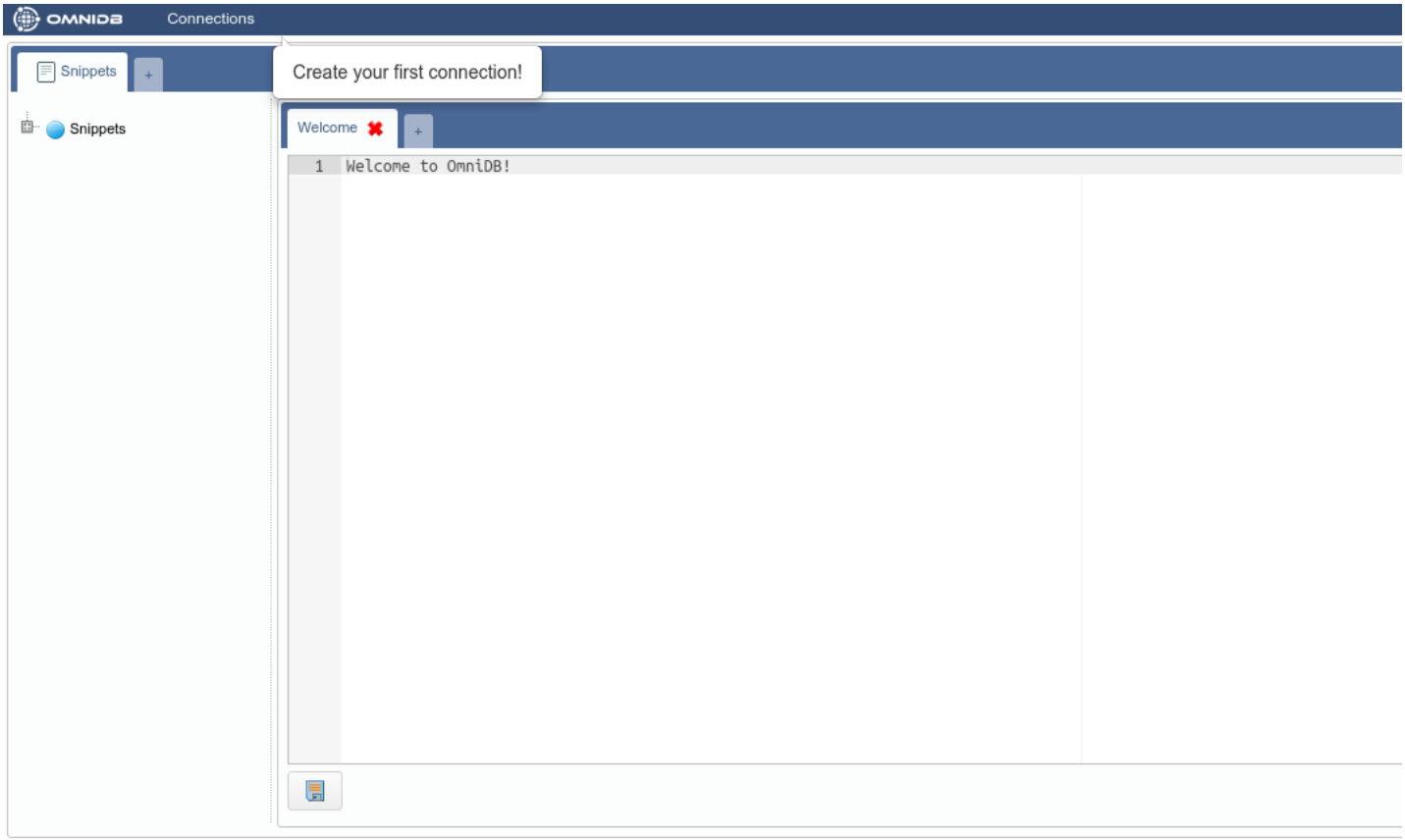
3 Creating Users and Connections

3.0.0.1 Logging in as user *admin*

OmniDB comes only with the user *admin*. If you are using the server version, the first thing to do is sign in as *admin*, the default password is *admin*. You don't need to login in the app version.

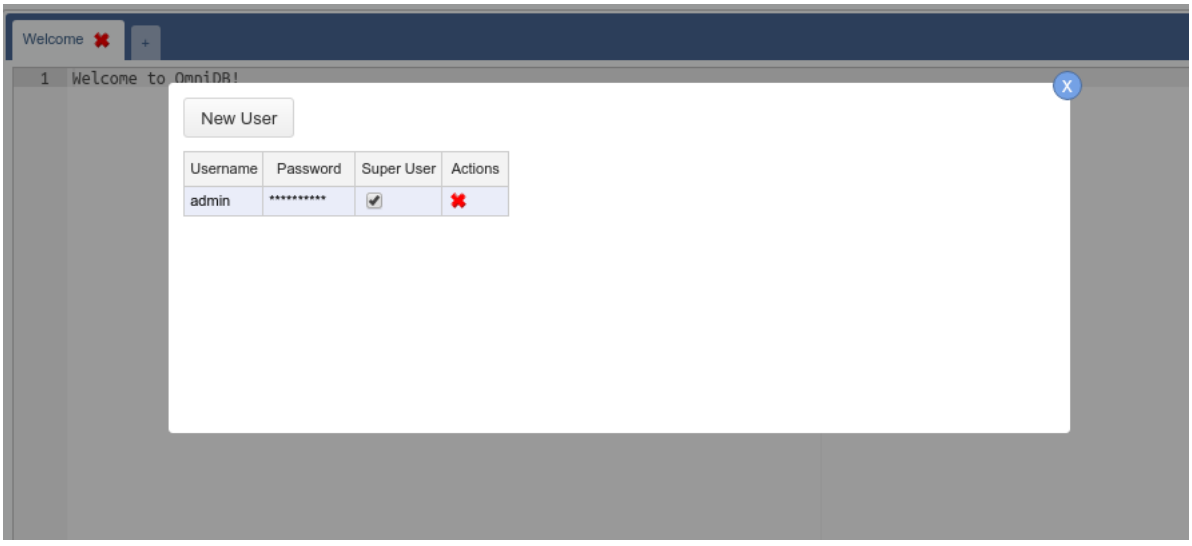


The next window is the initial window. We will talk about it later.

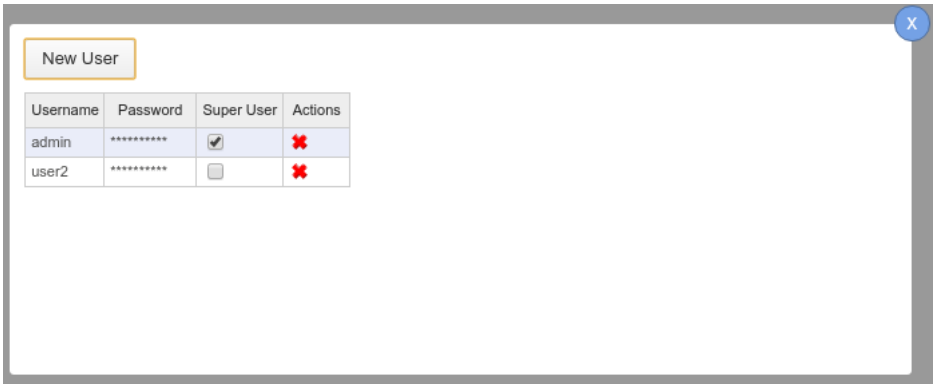


3.0.0.2 Creating another user

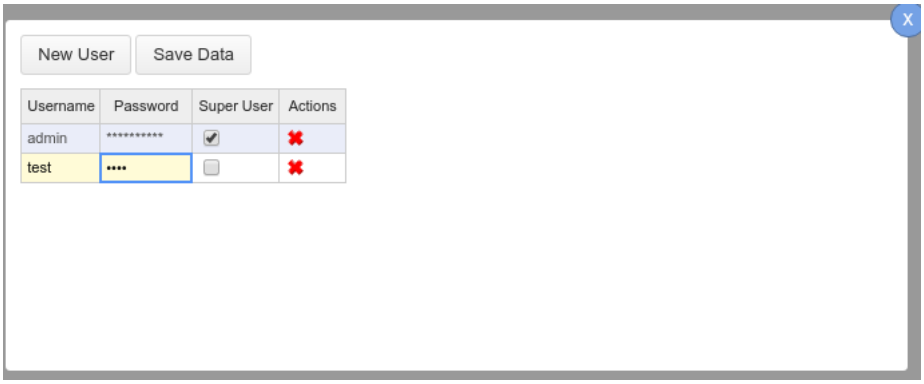
Click on the *Users* icon on the upper right corner. It will open a popup that allows the current OmniDB super user to create a new OmniDB user.



After clicking on the *Users* icon the tool inserts a new user called *user2* (if that is the first user after *admin*).



You will have to change the *username* and *password*. Check if you want this new user to be a *super user*. This user management window is only seem by super users. When you are done, click on the *Save Data* button inside the popup.

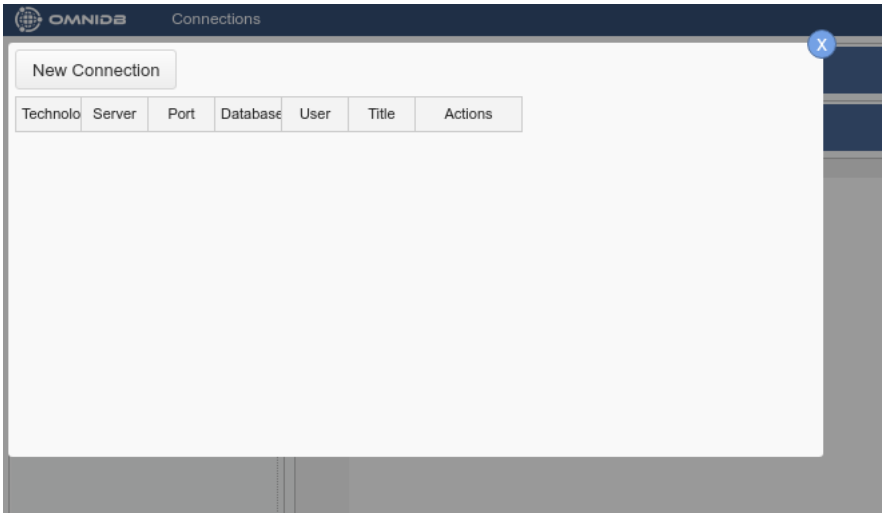


You can create as many users as you want, edit existing users and also delete users by clicking on the red cross at the actions column. Now you can logout.

3.0.0.3 Signing in as the new user

Let us sign in as the user we just created.

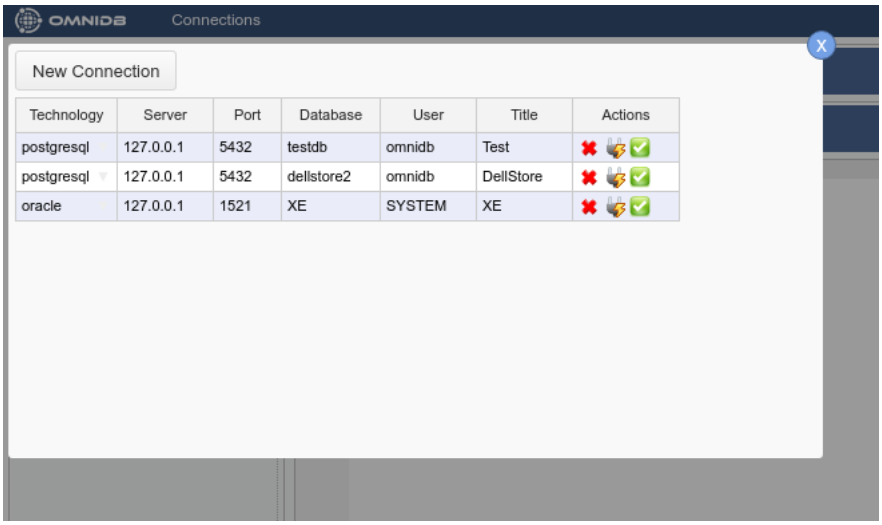
And we can see the window again. Note that now there is no *Users* icon, because the *test* user is not a super user. Go ahead and click on **Connections** on the upper left corner. You will see a popup like this:



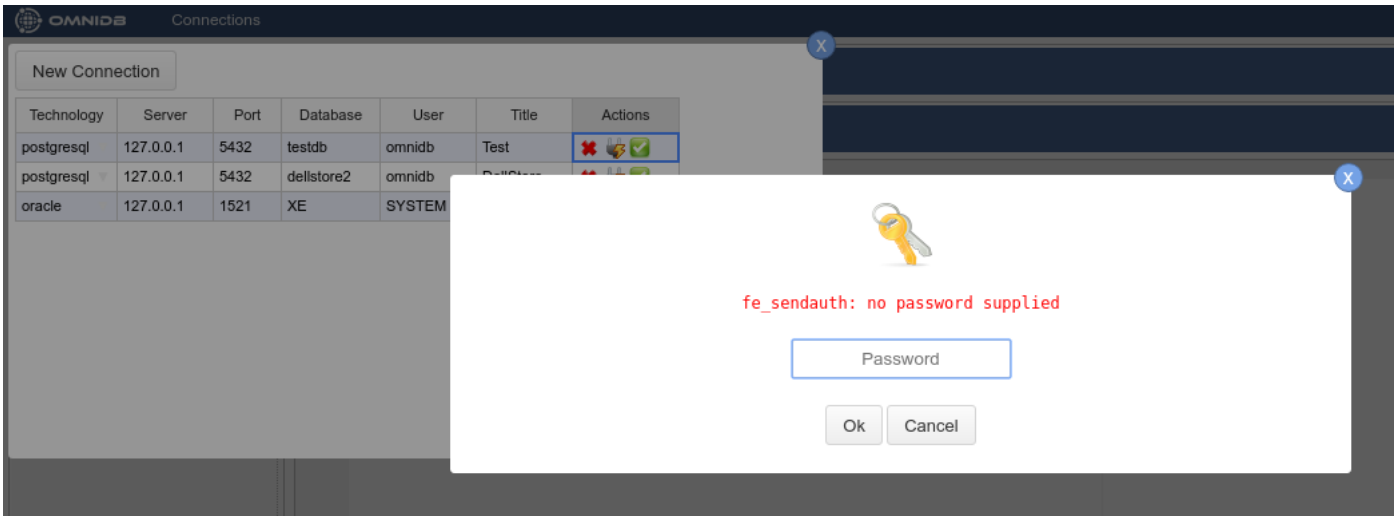
3.0.0.4 Creating connections

OmniDB C# version supported several DBMS. At the moment, OmniDB Python version, or `OmniDB 2.0`, supports only PostgreSQL and Oracle. More DBMS support is being added as you read this.

We will now create two connections to PostgreSQL databases and one connection to an Oracle database. To create the connections you have to click on the button *New Connection* and then choose the connection and fill the other fields. After filling all the fields for both connections, click on the *Save Data* button.

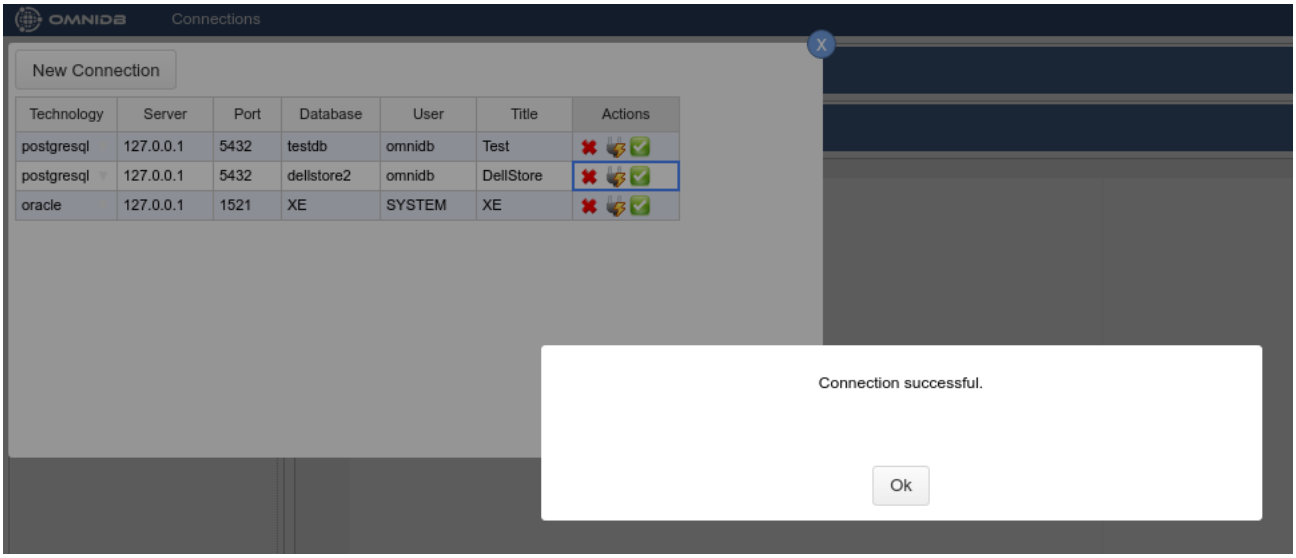


For each connection there is an *Actions* column where you can delete, test and select them. Go ahead and test one of the PostgreSQL connections.

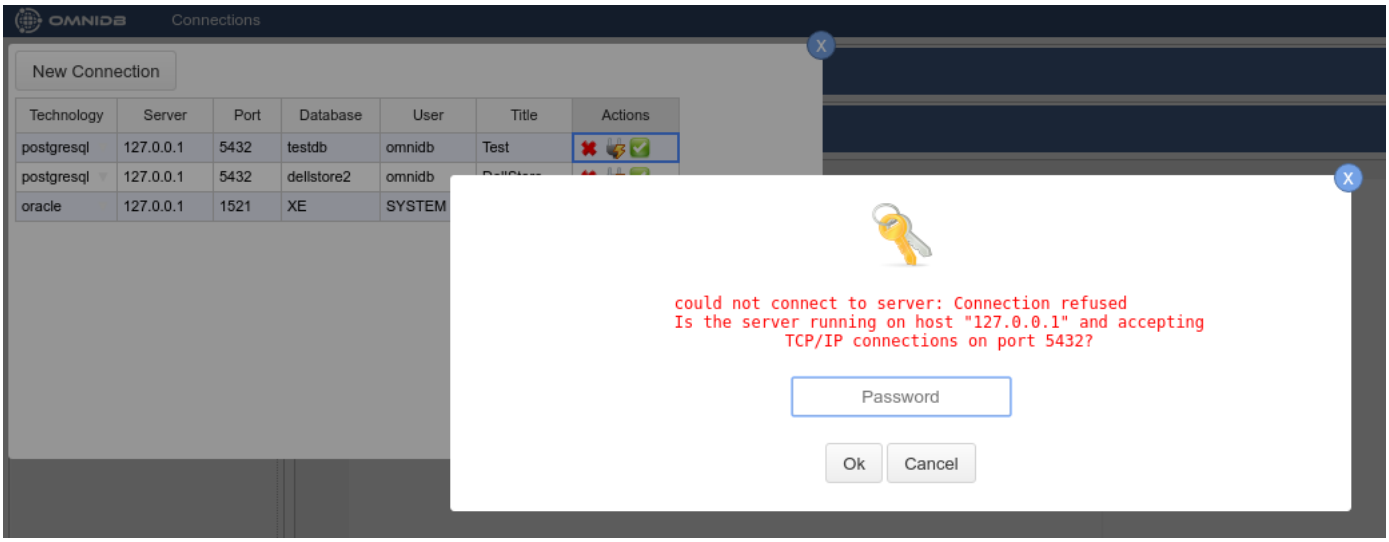


Notice a pop-up appears with the message *fe_sendauth: no password supplied*. This is happening because OmniDB does not store the database user password on disk. Not having any password at hand, OmniDB will try to connect without one, thus trying to take advantage of automatic authentication methods that might be in place: `trust` method, `.pgpass` file, and so on. As the database server replies with an error not allowing the user to connect, then OmniDB understands a password is required and asks it to the user. When the user types a password in this popup, the password is encrypted and stored in memory.

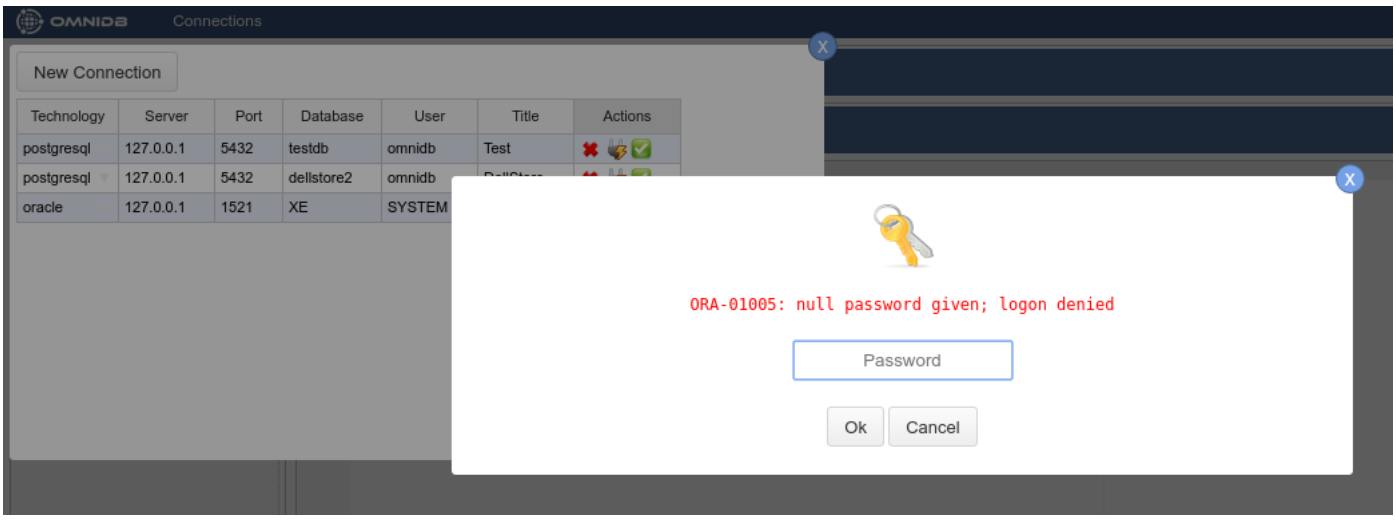
After you type the password and hit *Enter*, if the connection to the database is successful you will see a confirmation pop-up.



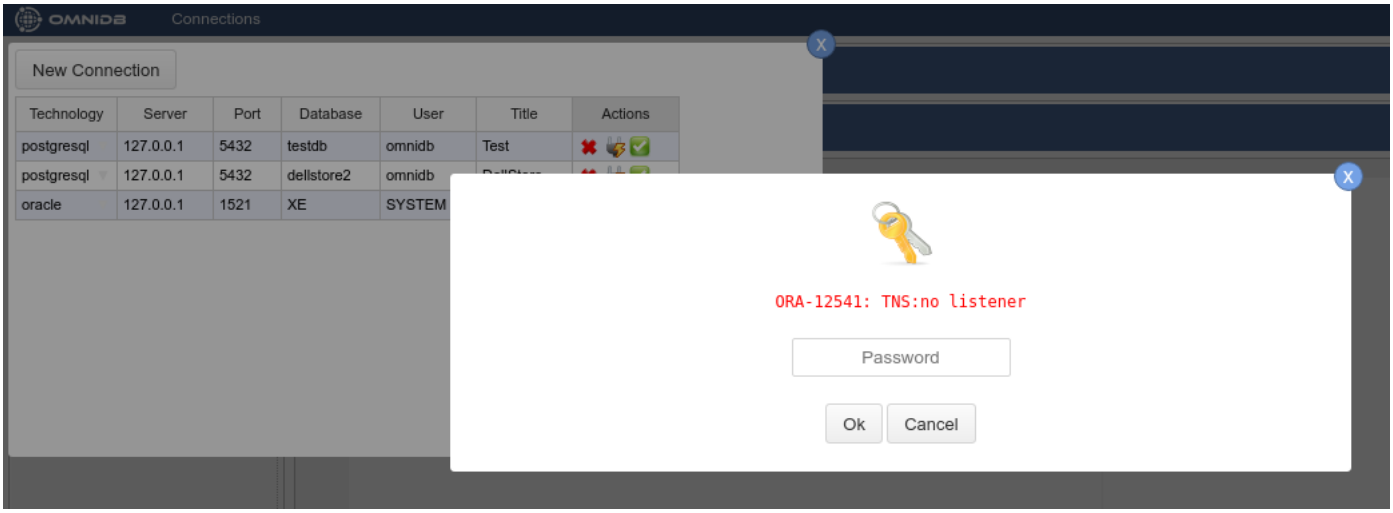
But, if you have trouble of any kind connecting to your PostgreSQL database, the same popup will remain showing the error OmniDB got.



For Oracle, the behavior is similar. When OmniDB first tries to connect to an Oracle database without a password, you will see a message like this:



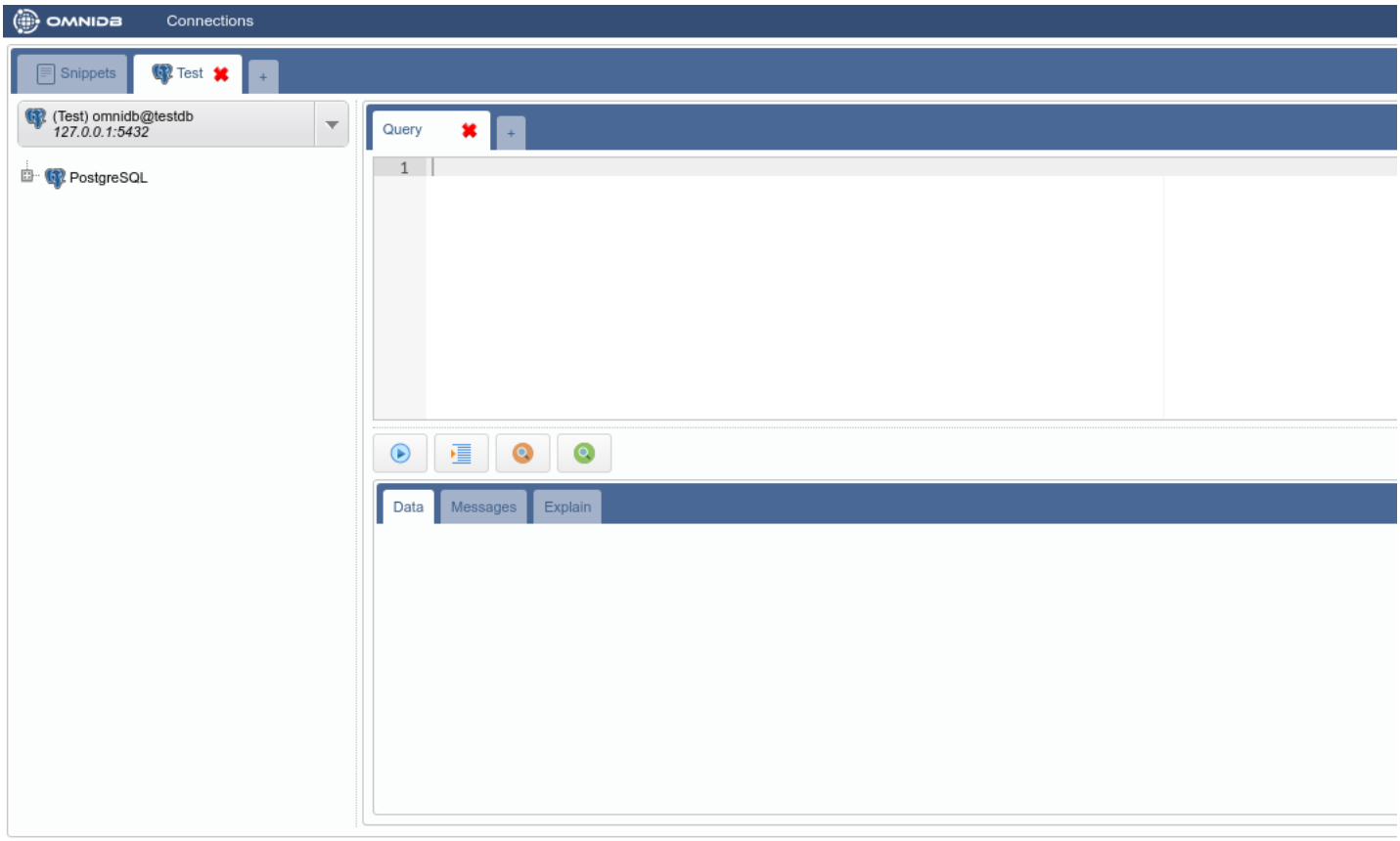
If you have any trouble connection to your Oracle database, the same popup will remain showing the error OmniDB got:



Finally, in the connections grid, if you click on the *Select Connection* action, OmniDB will open it in a new **Connection Outer Tab** as we can see in the next chapter.

4 Managing Databases

After creating a connection you can select it by clicking in the *Select Connection* action in the connections grid. You will see that the connection will be represented by a kind of outer tab called a *Connection Tab*. And this whole area is called the *Workspace Window*.



4.0.0.1 Sections of the *Workspace* window

This interface has several elements:

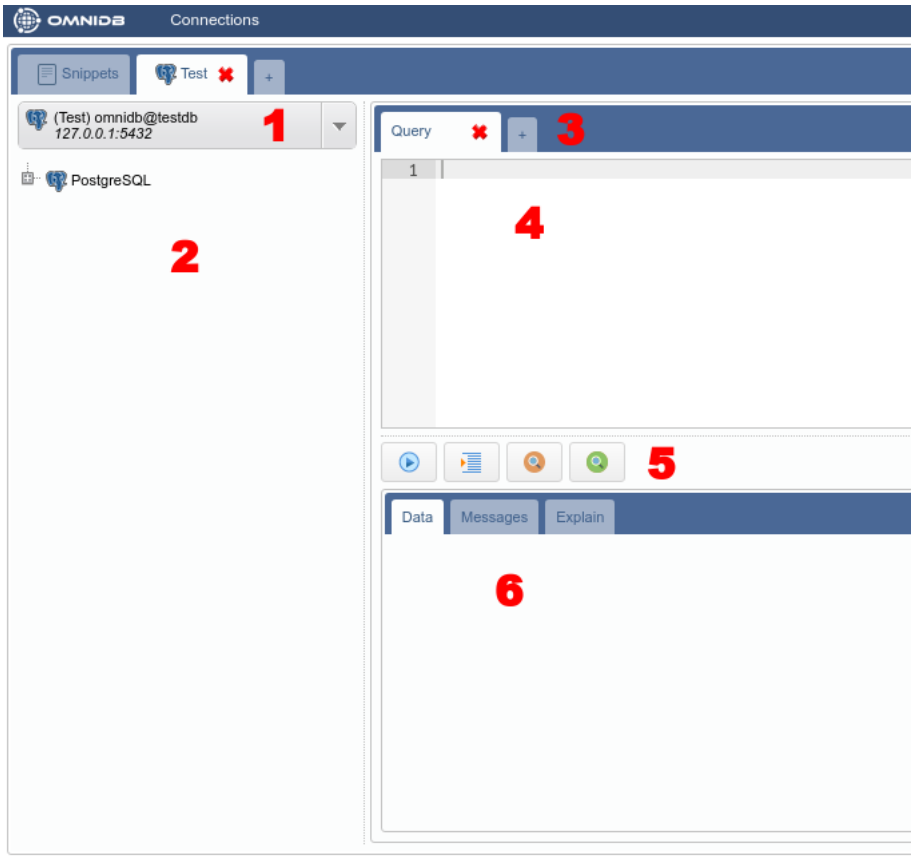
- **1) Connections**: Opens a popup with the *Connections* grid

- **2) Outer Tabs**: OmniDB lets you work with several databases at the same time. Each database will be accessible through an *outer tab*. Outer tabs also can host miscellaneous connection-independent features, like the *Snippets* feature
- **3) Options**: Shows the current user logged in, and also links for *user settings*, *query history*, *information* and *logout*.

4.0.0.2 Connection Outer Tab

So, the outer table named *Test* has this name because of the alias we put in the connection to the `testdb`. This tab is a *Connection Outer Tab*. Notice the little tab with a cross besides the *Test* outer tab. This allows you to create a new outer tab that will automatically be a *Connection Outer Tab*. However, the *Snippet Outer Tab* is fixed and will always be the first.

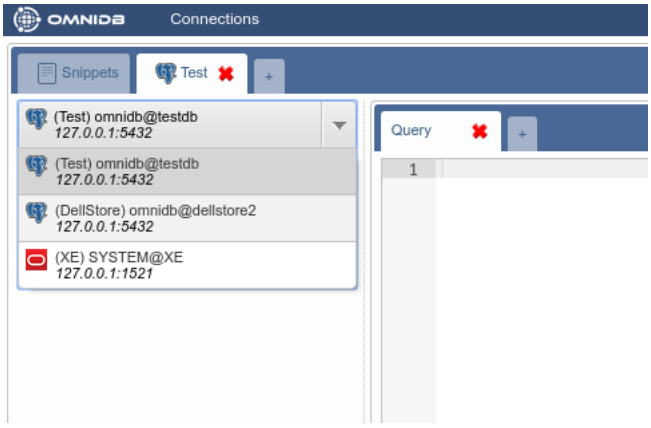
A new *Connection Outer Tab* will always automatically point to the first connection on your list of database connections. Or, if you clicked on the *Select Connection* action, it will point to the selected connection. Observe the elements inside of this tab:



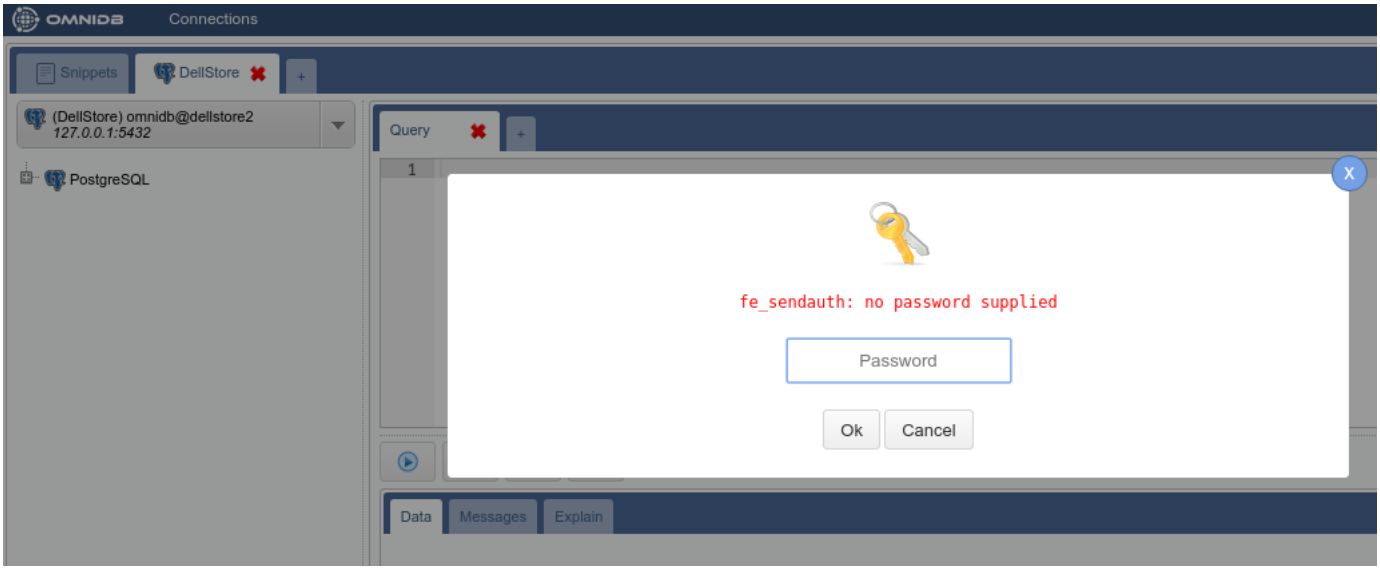
- **1) Connection Selector**: Shows all connections and lets the user select the current one
- **2) Tree of Structures**: Displays a hierarchical tree where you can navigate through the database elements
- **3) Inner Tabs**: Allows the user to execute actions in the current database. There are several kinds of inner tabs for the current database. By clicking on the last small tab with a cross, you can add a new tab. A new tab always will be a *Query Tab*, where you can write any kind of SQL statement
- **4) Inner Tab Content**: Can vary depending on the kind of inner tab. The figure shows a *Query Tab* and in this case the content will be an *SQL Editor*, with syntax highlight and autocomplete
- **5) Inner Tab Actions**: Can vary depending on the kind of inner tab. For a *Query Tab*, they are *Execute*, *Format*, *Explain* and *Explain Analyze*
- **6) Inner Tab Results**: A *Query Tab*, after you click in the *Execute Button* or type the execute shortcut (`Alt-Q`), will show a grid with the query results in the *Data* subtab. If the query calls a function that raises messages, those will be shown in the *Messages* subtab. If instead of *Execute* you clicked in *Explain* or *Explain Analyze*, the explain plan for the query will be shown in the *Explain* subtab.

4.0.0.3 Working with databases

Take a look at your connections selector. OmniDB always points to the first available connection but you can change it by clicking on the selector.

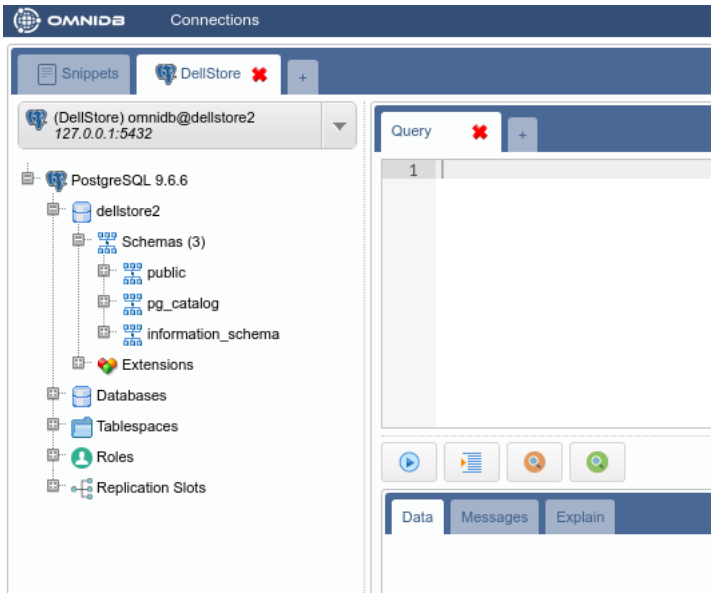


Select the *DellStore* connection. Now go to the tree right below the selector and click to expand the root node *PostgreSQL*.



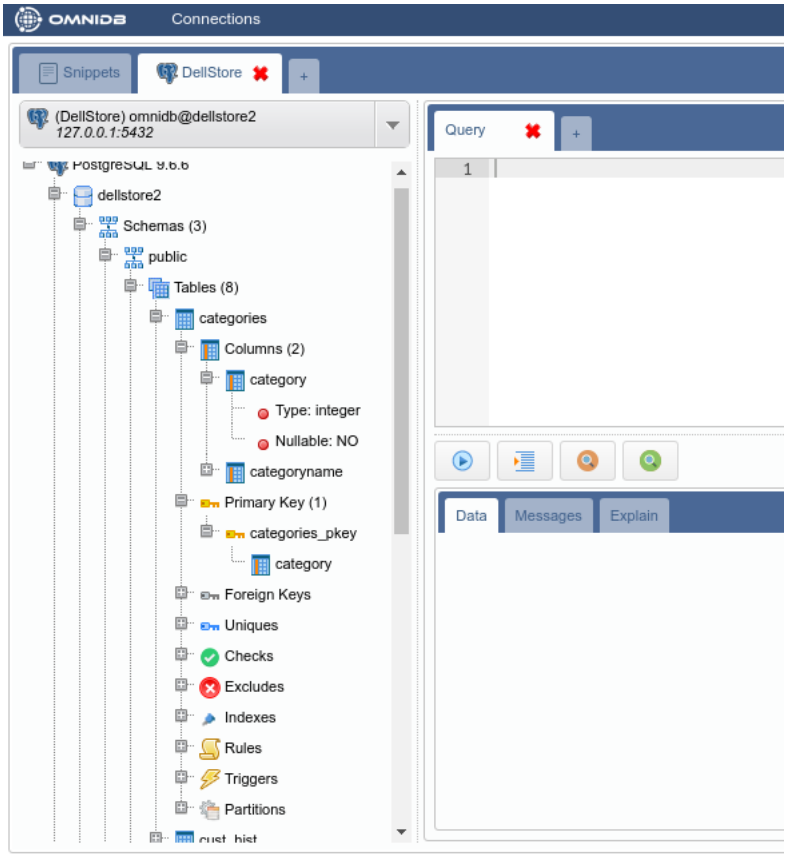
Bear in mind that every 30 minutes you keep without performing actions on the database, will trigger a *Authentication* popup, meaning that the password that OmniDB has encrypted and stored in memory is now expired. As explained before, this is important for your database security. After you type the correct password, you will see the PostgreSQL node now shows the PostgreSQL version and also was expanded, showing the current database connection and also instance wide elements: *Databases*, *Tablespaces*, *Roles* and *Replication Slots*.

Go ahead and expand the *Schemas* node. You will see all schemas in the current database (in case of PostgreSQL, `TOAST` and temp schemas are not shown).

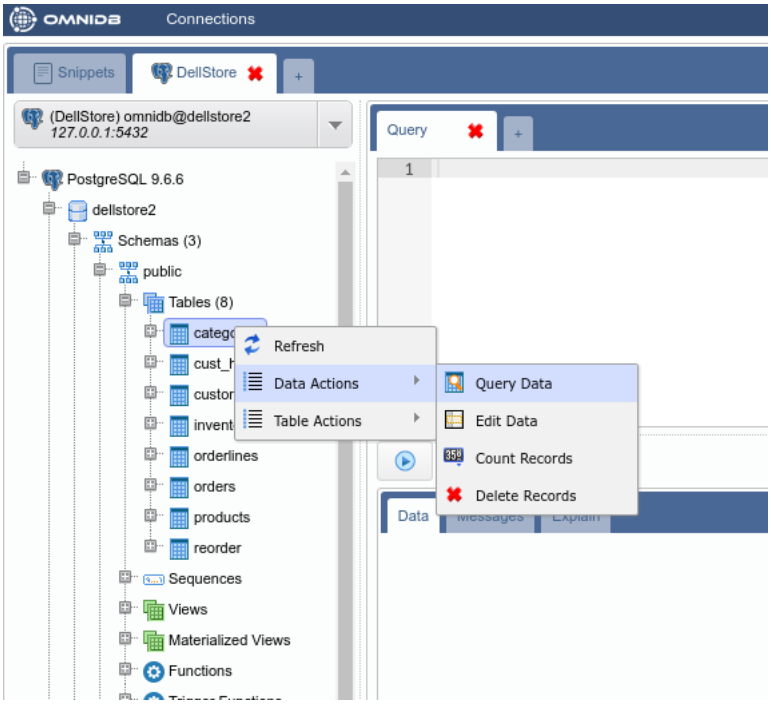


Now click to expand the schema `public`. You will see different kinds of elements contained in this schema.

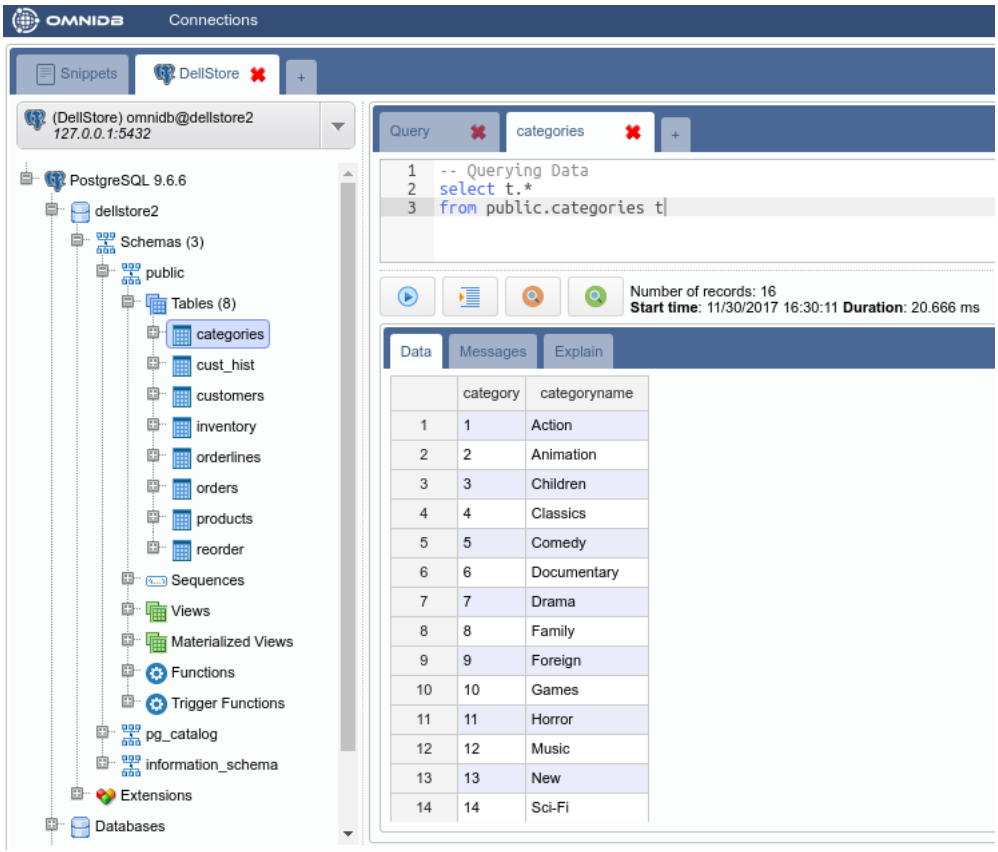
Now click to expand the node *Tables*, and you will see all tables contained in the schema `public`. Expand any table and you will see its columns, primary key, foreign keys, constraints, indexes, rules, triggers and partitions.



In order to view records inside a table, right click it and choose *Data Actions* > *Query Data*.



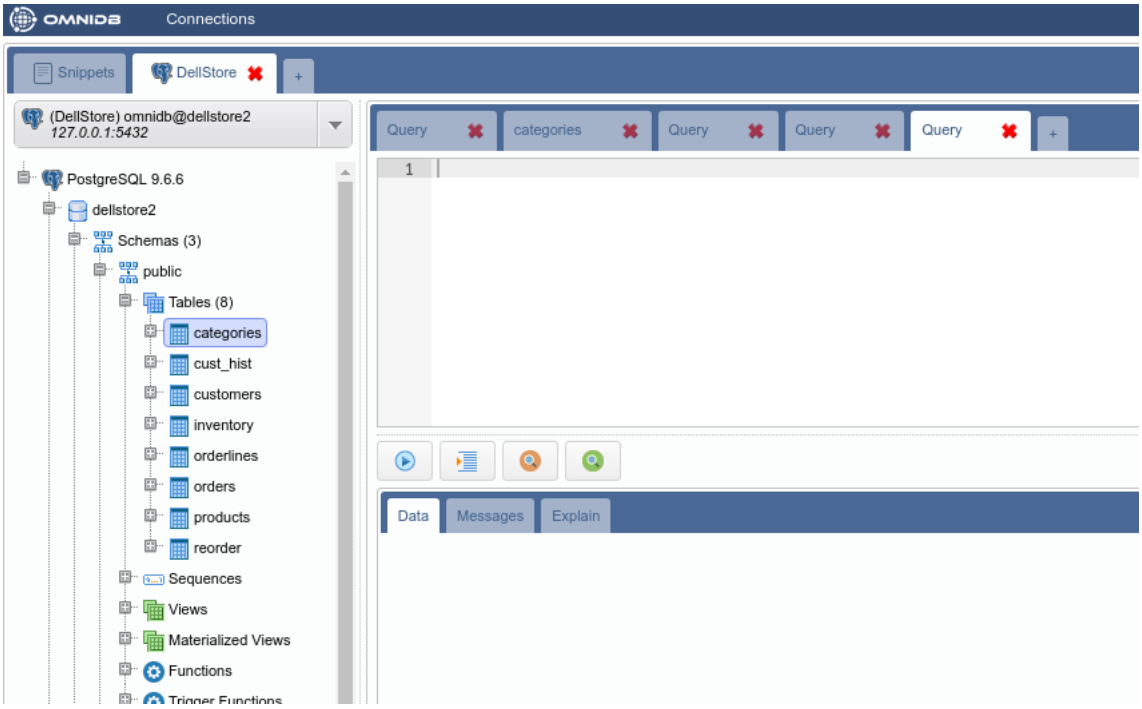
Notice that OmniDB opens a new SQL editor with a simple query to list table records. The records are displayed in a grid right below the editor. This grid can be controlled with keyboard as if you were using a spreadsheet manager. You can also copy data from single cells or block of cells (that can be selected with the keyboard or mouse) and paste on any spreadsheet manager.



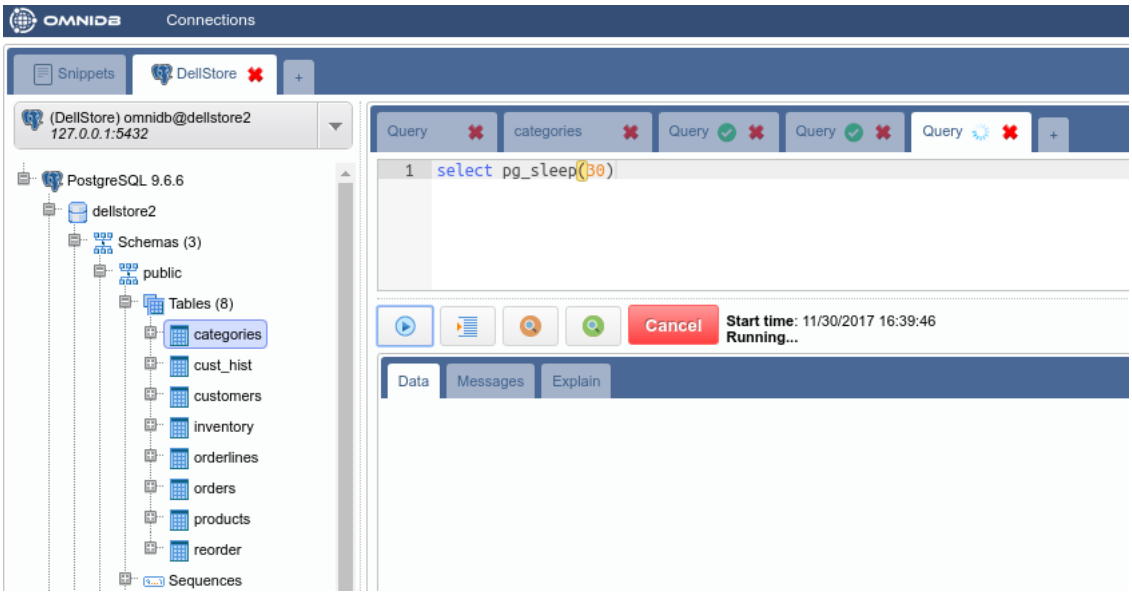
You can edit the query on the SQL editor, writing simple or more complex queries. To execute, click on the action button or hit the keystroke `Ctrl-Q`. If the results exceed 50 registers, then extra buttons *Fetch More* and *Fetch All* will appear. More details in the next chapters.

4.0.0.4 Working with multiple tabs inside the same connection

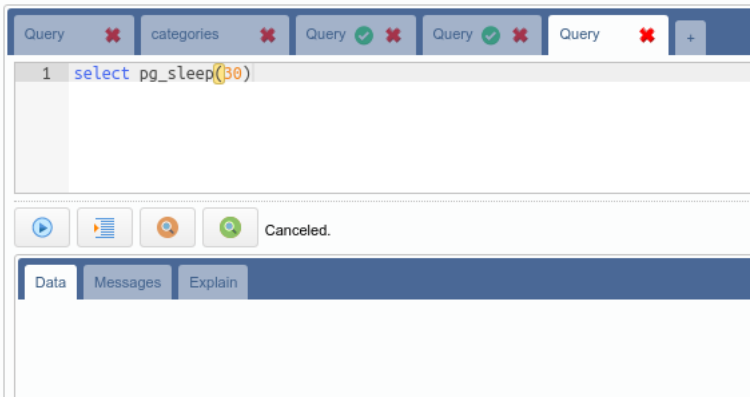
Inside a single connection, you can create several inner tabs by clicking on the last little tab with a cross. Each new inner tab will be a *Query Tab*.



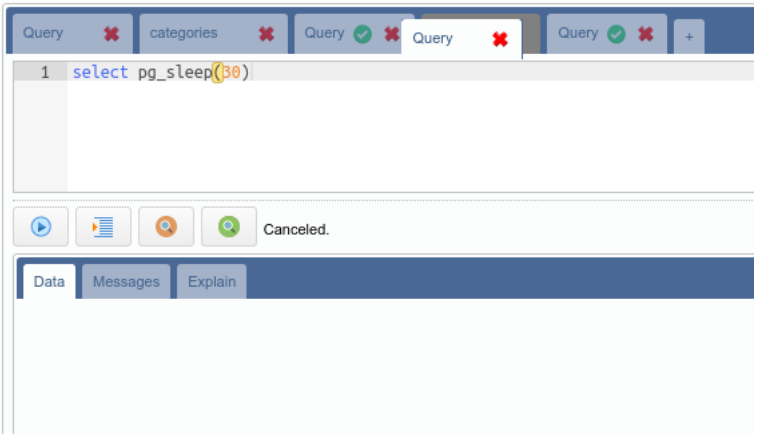
On OmniDB, you can execute several SQL statements and procedures in parallel. When it is executing, an icon will be shown in the tab to indicate its current state. If some process is finished and it is not in the current tab, that tab will show a green icon indicating the routine being executed there is now finished.



By clicking in the *Cancel button*, you can cancel a process running inside the database.



You can also drag and drop a tab to change its order. This works with both inner and outer tabs.



Additionally, you can use keyboard shortcuts to manage inner tabs (SQL Query) and outer tabs (Connection):

- **Ctrl-Insert**: Insert a new inner tab
- **Ctrl-Delete**: Removes an inner tab
- **Ctrl-<**: Change focus to inner tab at left
- **Ctrl->**: Change focus to inner tab at right
- **Ctrl-Shift-Insert**: Insert a new outer tab
- **Ctrl-Shift-Delete**: Removes an outer tab
- **Ctrl-Shift-<**: Change focus to outer tab at left
- **Ctrl-Shift->**: Change focus to outer tab at right

Starting from OmniDB version 2.3.0, all SQL Query tabs are automatically saved whenever you execute them. Even if you close OmniDB window or browser tab, they are already stored in OmniDB *User Database*. They will be automatically restored when you open OmniDB again (if you are using app), open it in another browser window (if you are using server), or even if you clicked in the *Connections* window or logged out. Removing an outer tab or inner tab by the interface makes it permanently deleted, so it will not be restored.

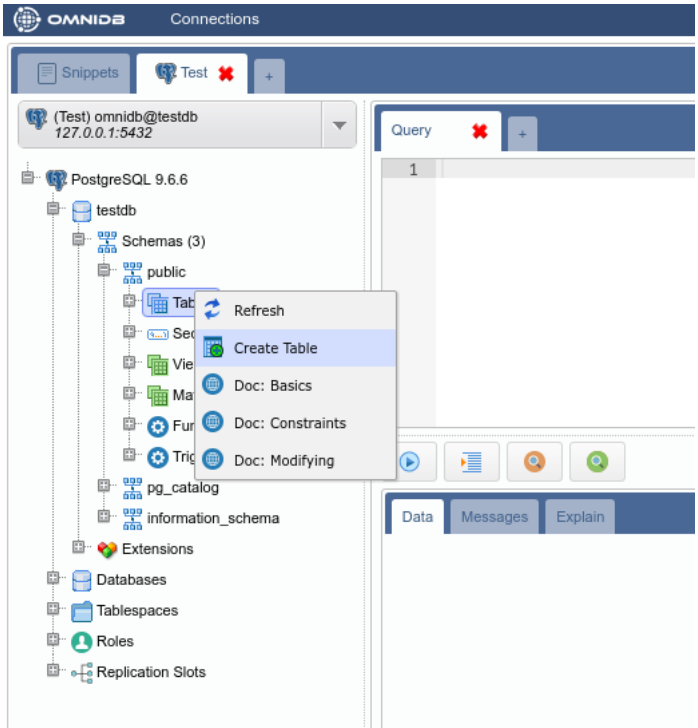
5 Creating, Changing and Removing Tables

5.0.0.1 Creating tables

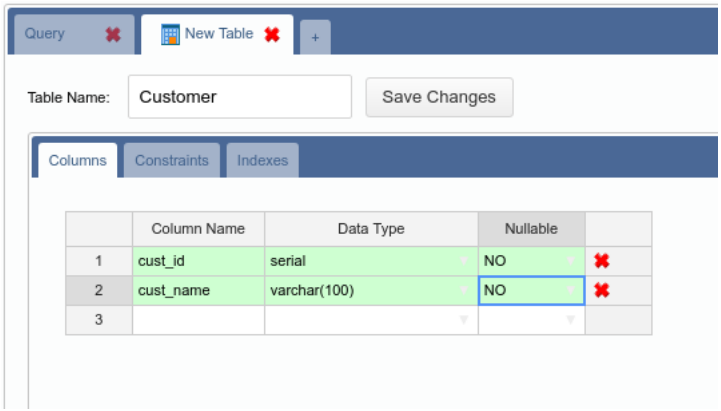
OmniDB has a table creation interface that lets you configure columns, constraints and indexes. A couple of observations should be mentioned:

- Most DBMS automatically create indexes when primary keys and unique constraints are created. Because of that, the indexes tab is only available after creating the table.
- Each DBMS has its unique characteristics and limitations regarding table creation and the OmniDB interface reflects these limitations. For instance, SQLite does not allow us to change existing columns and constraints. Because of that, the interface lets us change only table name and add new columns when dealing with SQLite databases (it is still not the case in OmniDB Python version, as it currently supports only PostgreSQL databases).

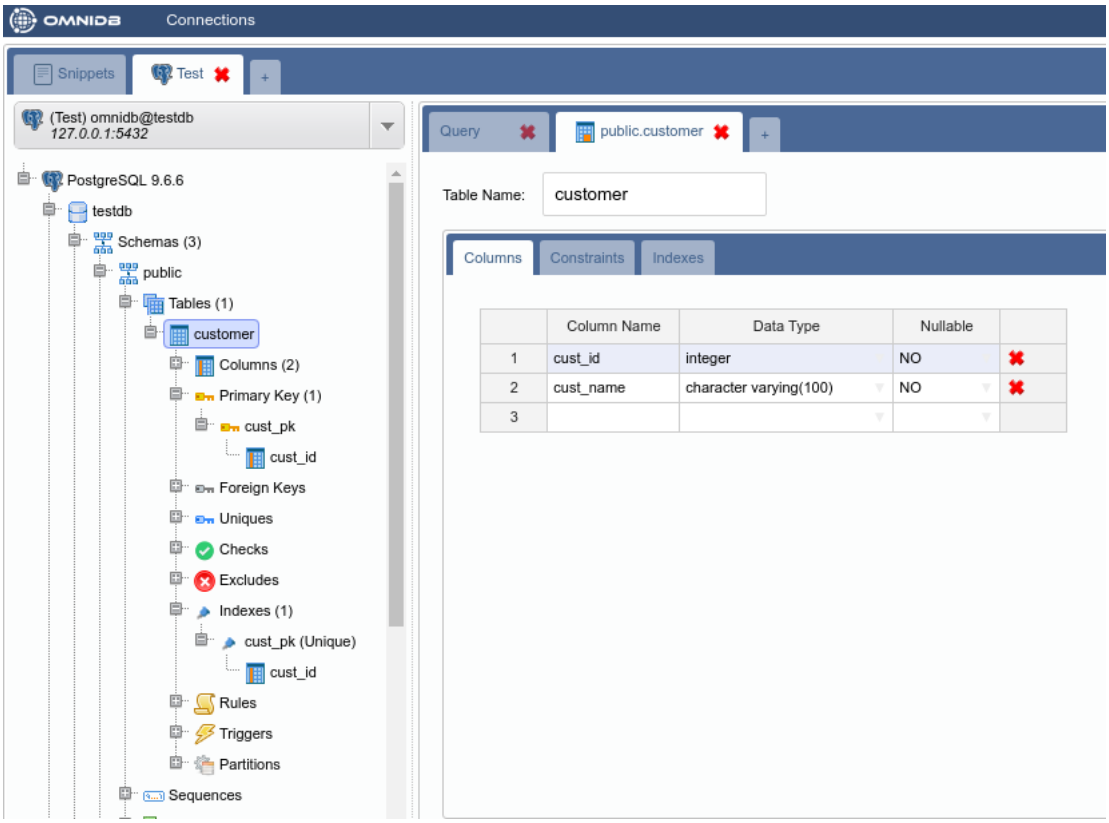
We will create example tables (*Customer* and *Address*) in the `testdb` database we connected to earlier. Right click on the **Tables** node and select the **Create Table** action:



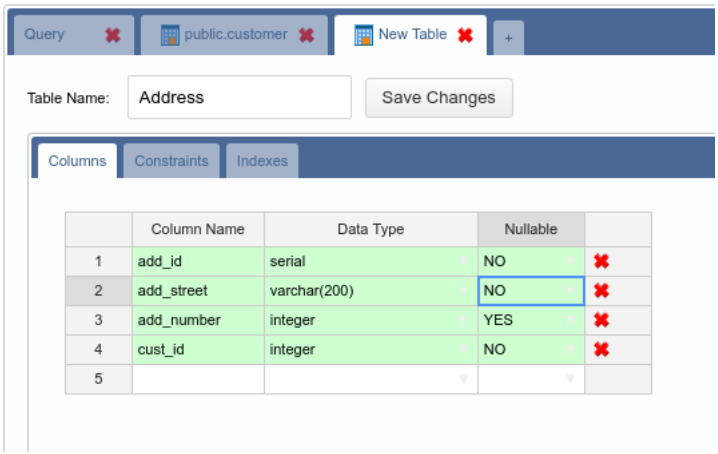
We will create the table *Customer* with a primary key that will be referenced by the table *Address*:

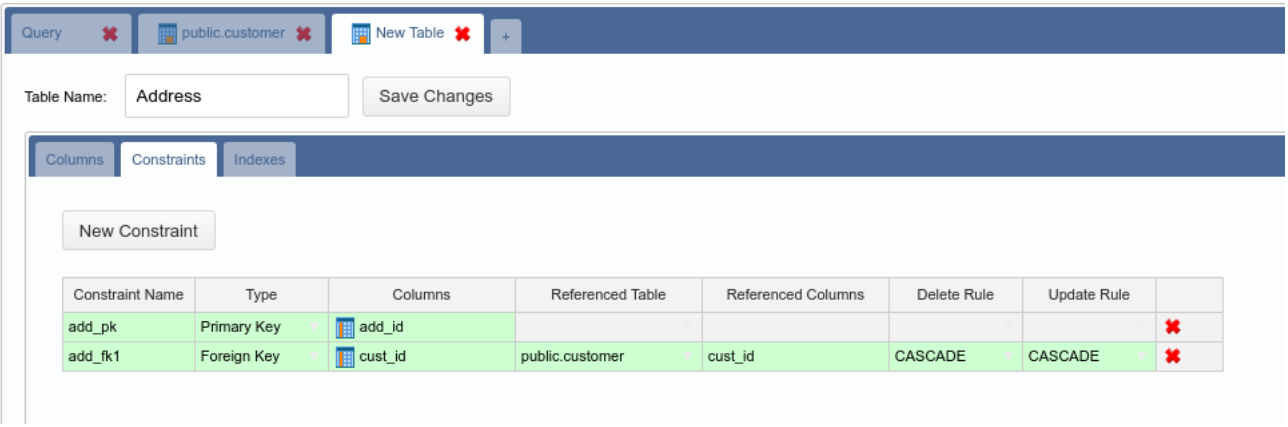


Click on the *Save Changes* button. Right-click the *Tables* tree node and click *Refresh*. Note how the table appers in the *Tables* tree node:

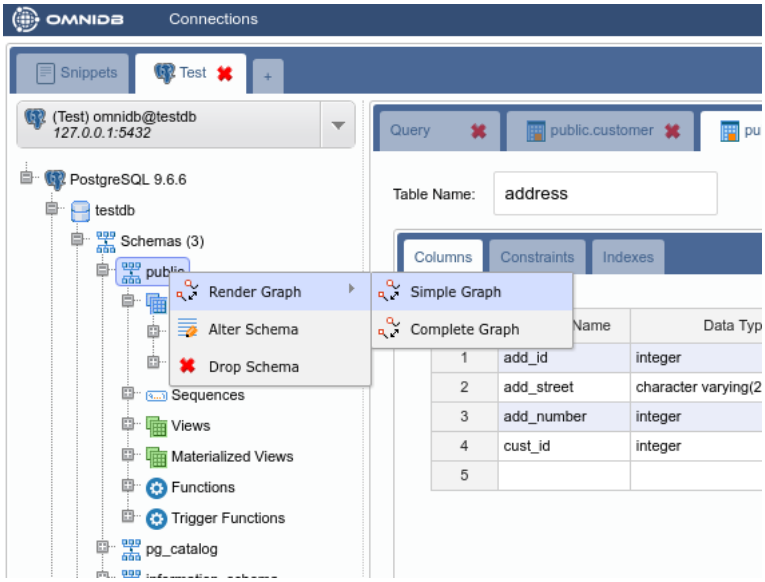


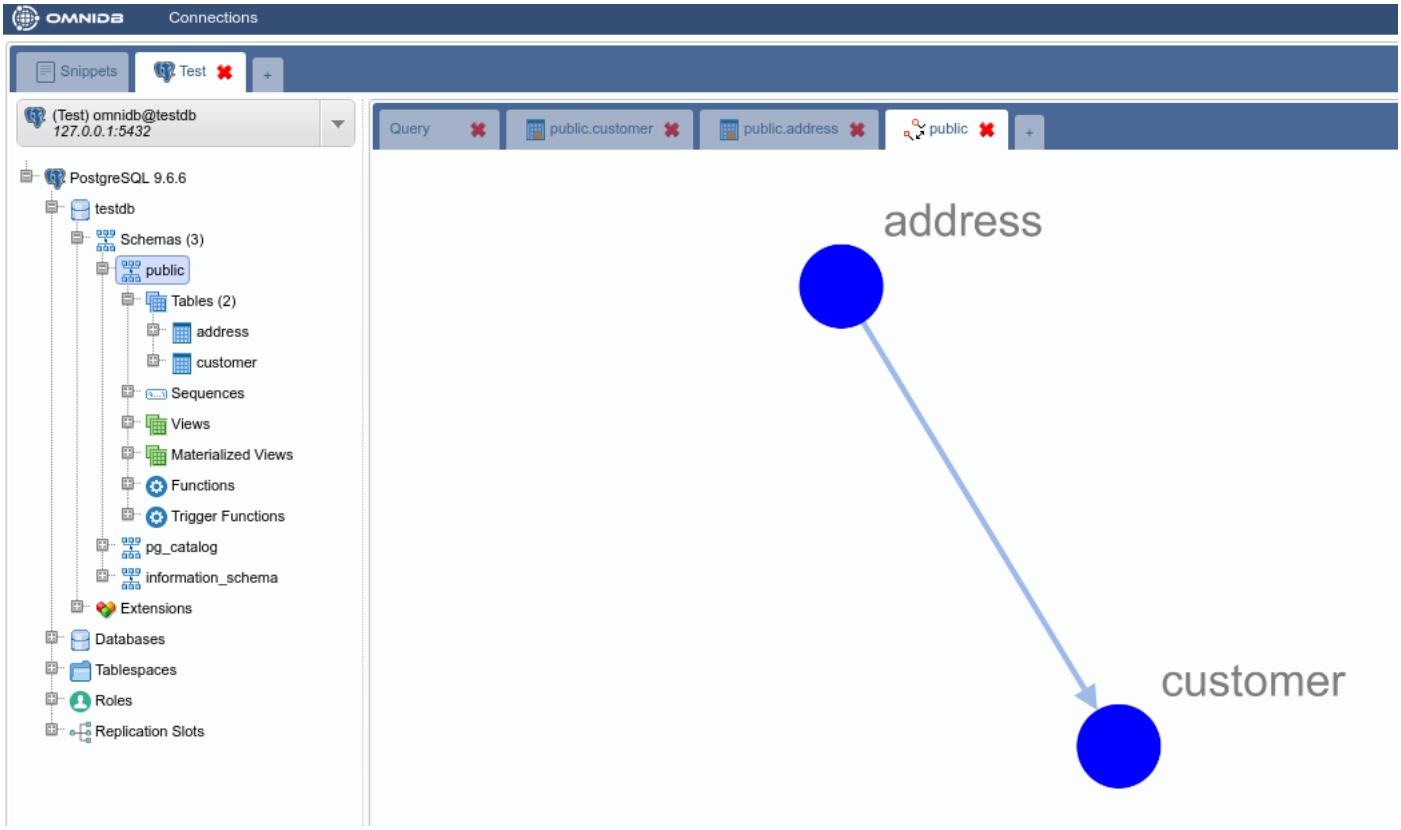
Now create the table *Address* with a primary key and a foreign key:



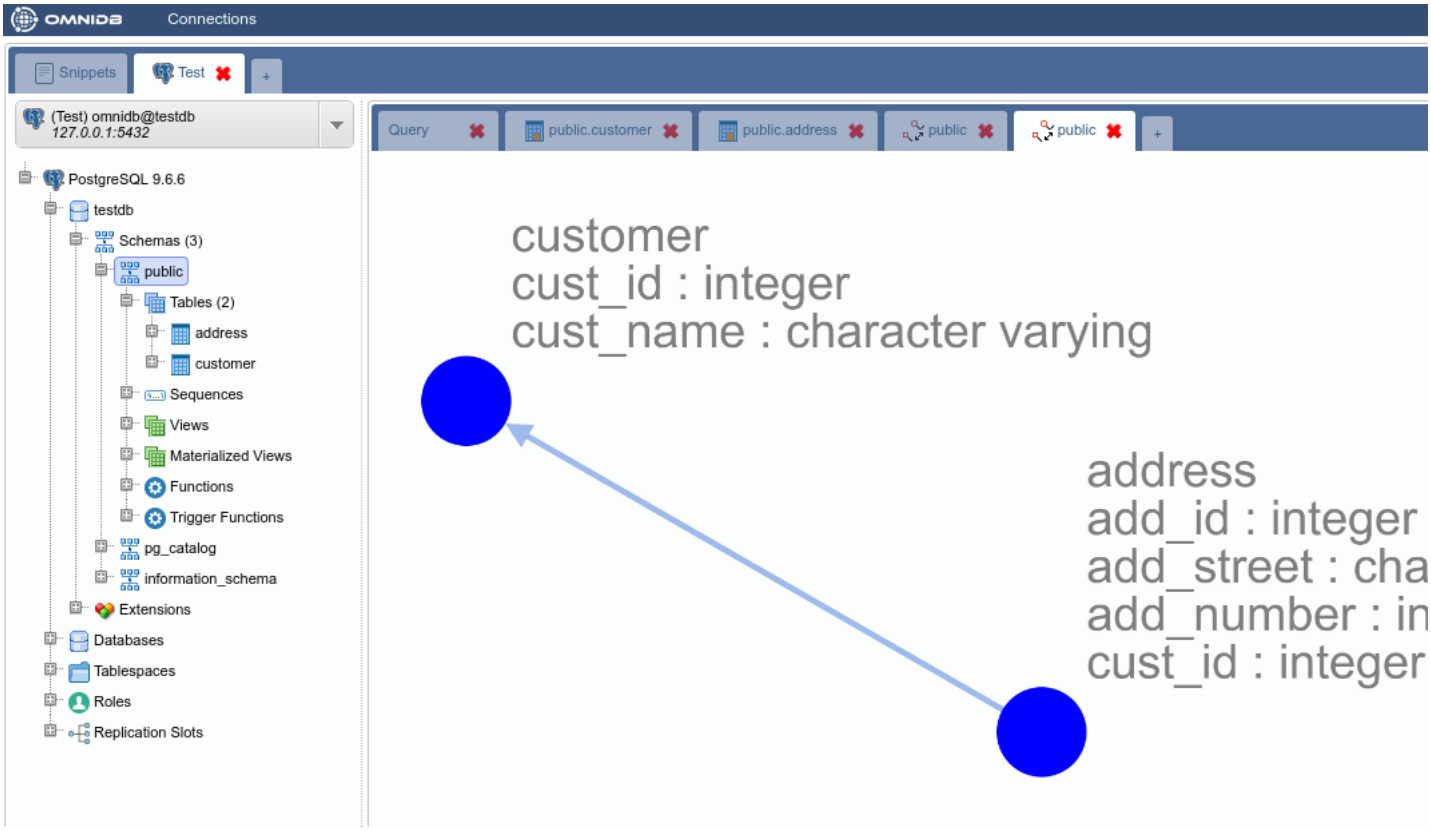


Don't forget to click on the *Save Changes* button when done. At this point we have two tables in schema `public`. The schema structure can be seen with the graph feature by right clicking on the schema `public` node of the tree and selecting *Render Graph* > *Simple Graph*:



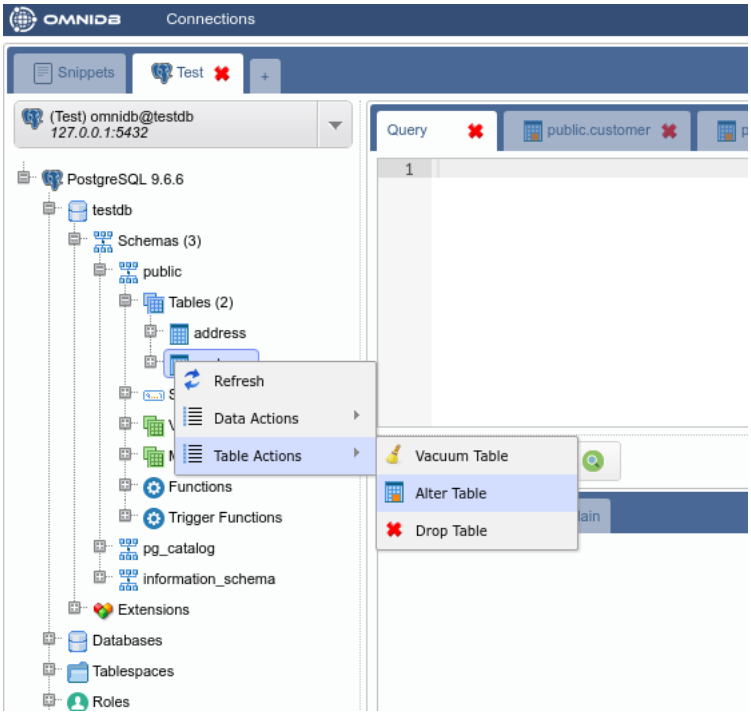


And this is what the *Complete Graph* looks like:

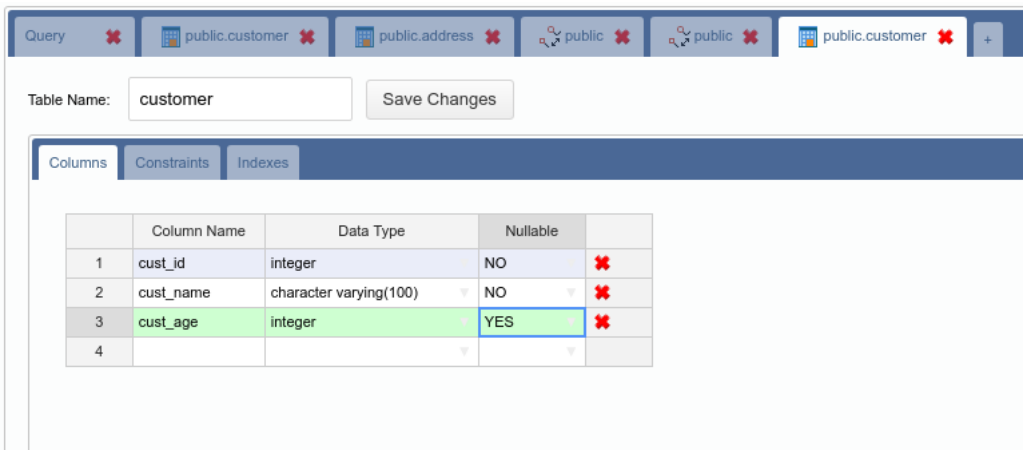


5.0.0.2 Editing tables

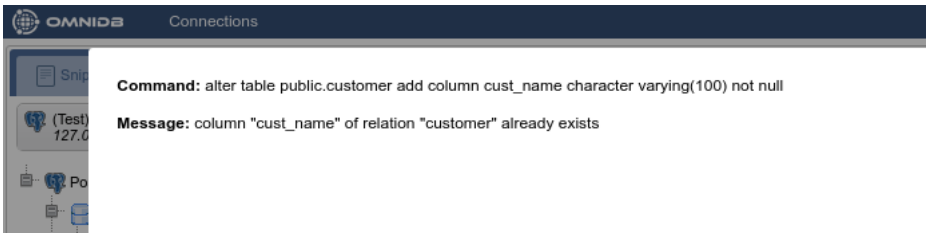
OmniDB also lets you edit existing tables (always following DBMS limitations). To test this feature we will add a new column to the table *Customer*. To access the alter table interface just right click the table node and select the action *Table Actions* > *Alter Table*:



Add the column *cust_age* and save:

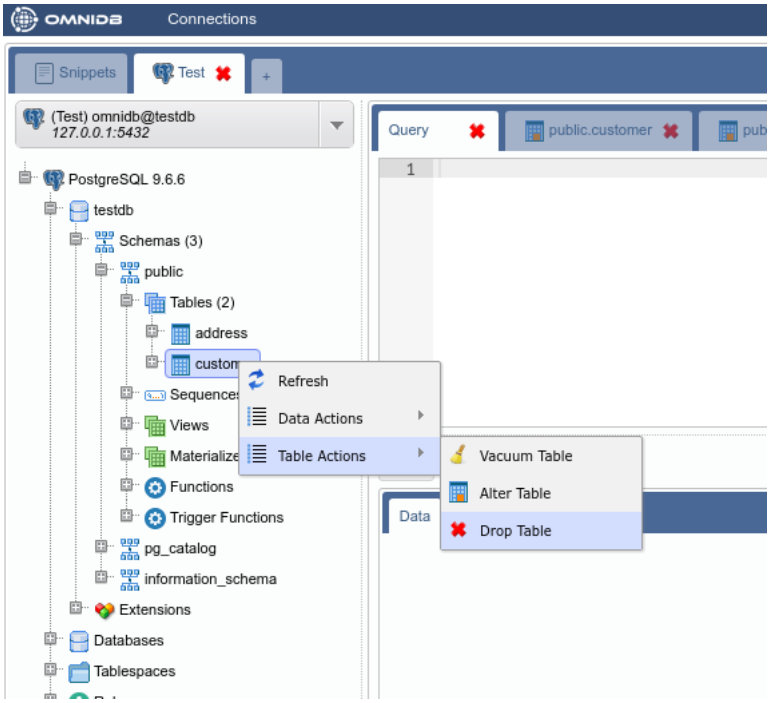


The interface is capable of detecting errors that may occur during alter table operations, showing the command and the error that occurred. To demonstrate it we will try to add the column *cust_name*, which already belongs to this table:



5.0.0.3 Removing tables

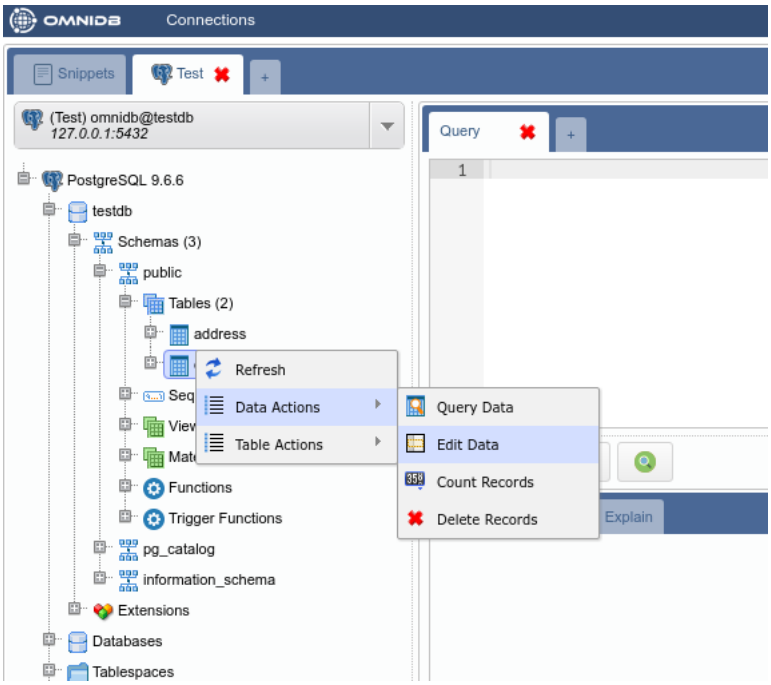
In order to remove a table just right click the table node and select the action *Table Actions* > *Drop Table*:

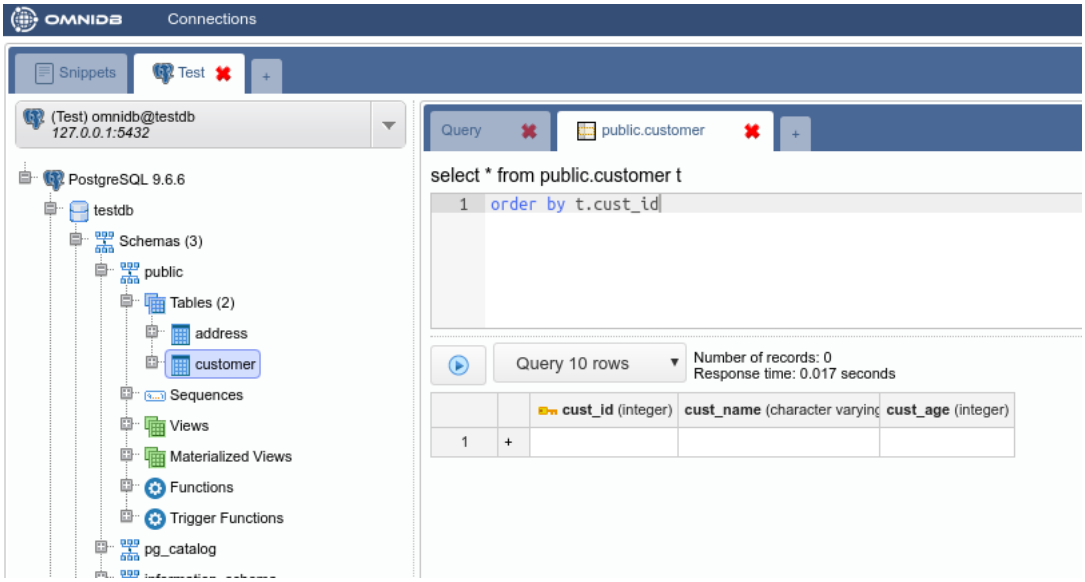


6 Managing Table Data

The tool allows us to edit records contained in tables through a very simple and intuitive interface. Given that only a few DBMS have unique identifiers for table records, we opted to allow data editing and removal only for tables that have a primary key. Tables that do not have it can only receive new records.

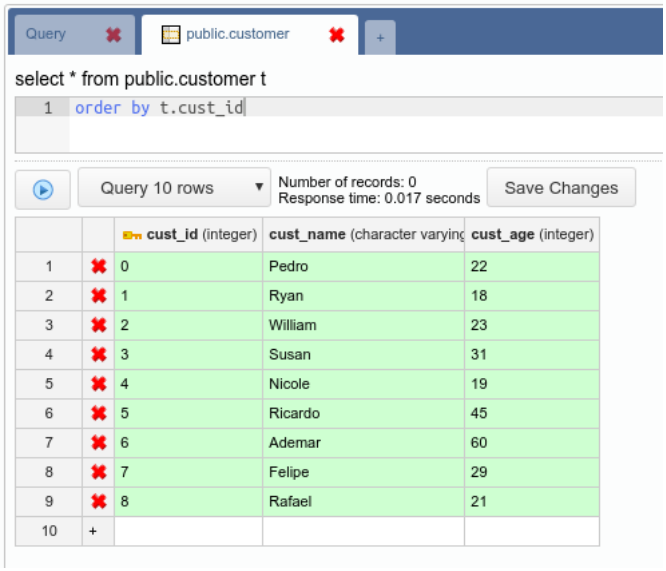
To access the record editing interface, right click the table node and select the action *Data Actions* > *Edit Data*:



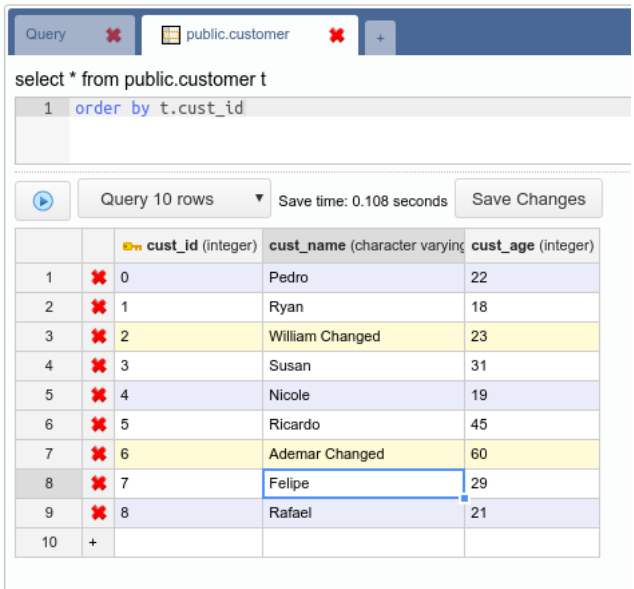


The interface has a SQL editor where you can filter and order records. To prevent that the interface requests too many records, there is a field that limits the number of records to be displayed. The records grid has column names and data types. Columns that belong to the primary key have a key icon next to their names.

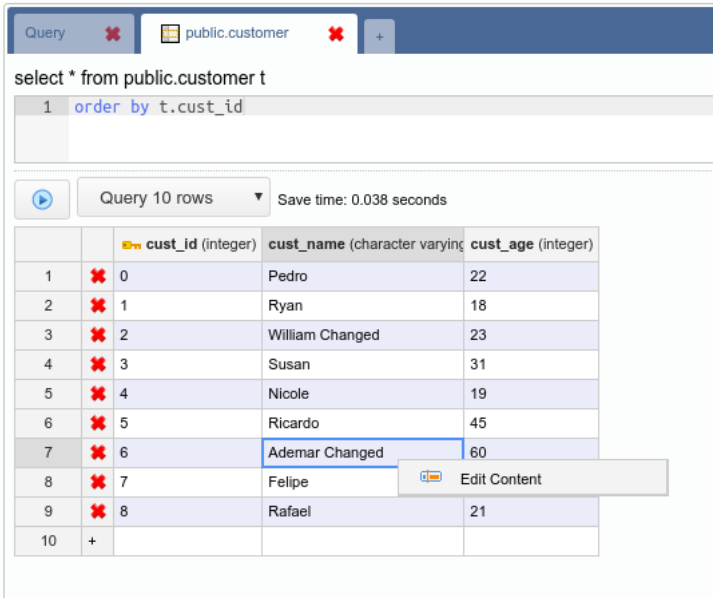
The row of the grid that have the symbol * is the row to add new records. Let us insert some records in the table `Customer`:

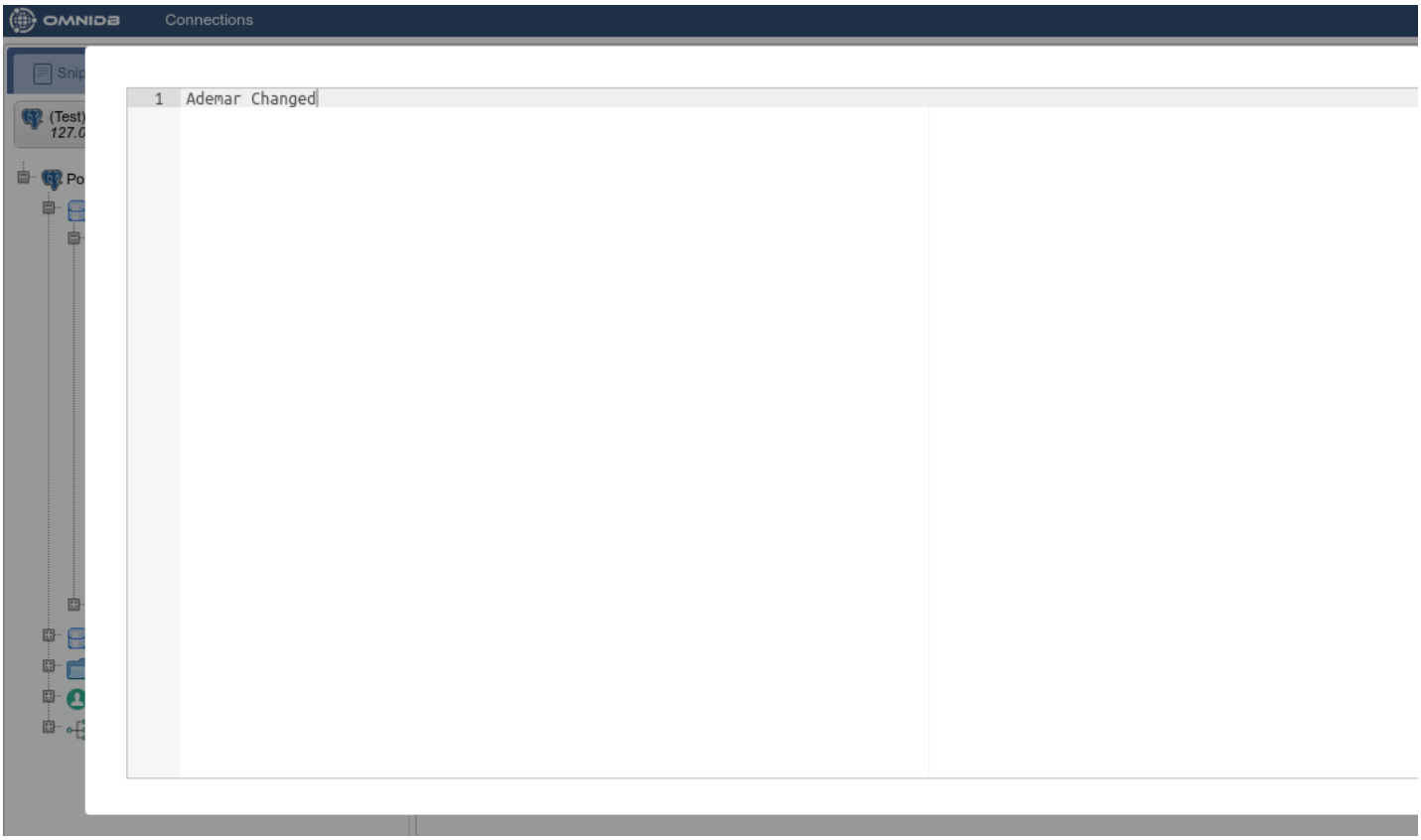


After saving, the records will be inserted and can be edited (only because this table has a primary key). Let's change the *cust_name* of some of the existing records:

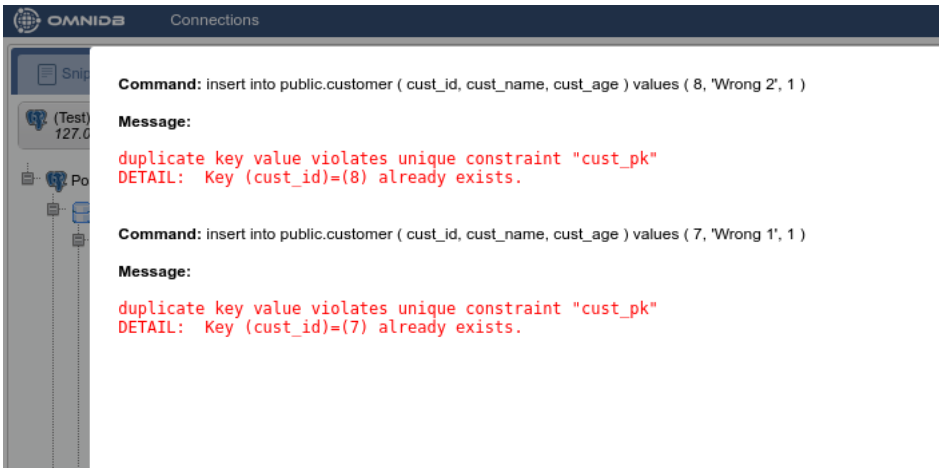


Tables can have fields with values represented by very long strings. To help edit these fields, OmniDB has an interface that can be accessed by right clicking the specific cell:



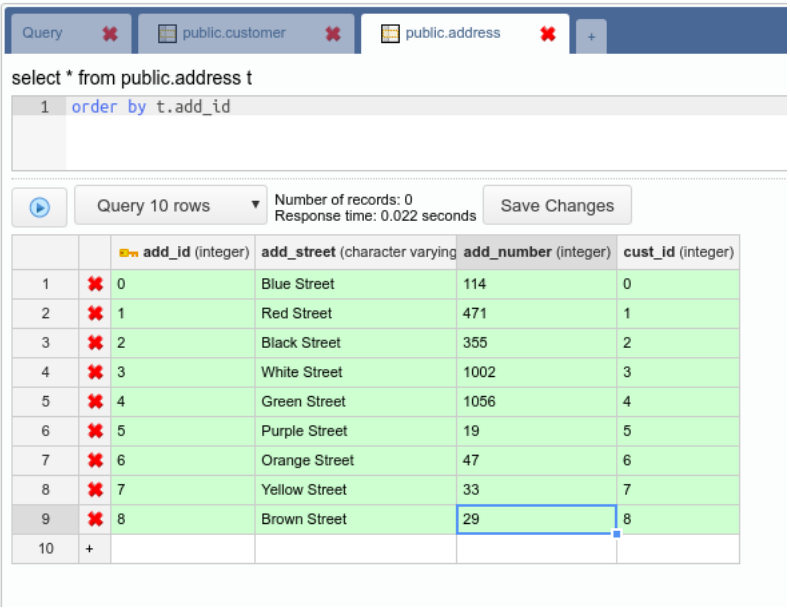


The interface detects errors that may occur during operations related to records. To demonstrate, let us insert two records with existing `cust_id` (primary key):



It shows which commands tried to be executed and the respective errors.

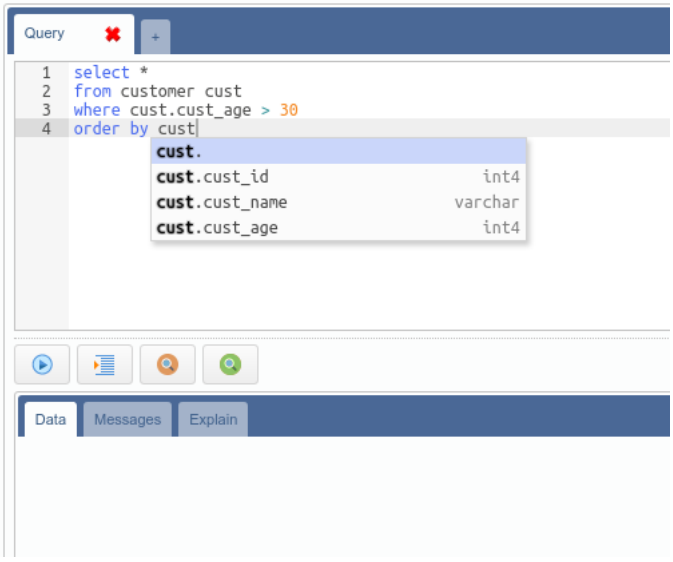
To complete this chapter, let's add some records to the *Address* table:



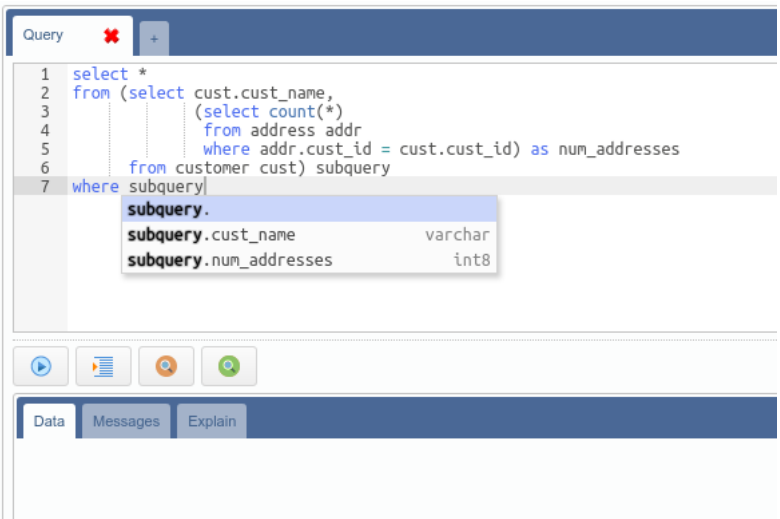
7 Writing SQL Queries

The tool comes with a tab system where each tab contains a SQL editor, an action button, an indent button, a field to select the type of command and a space to display the result.

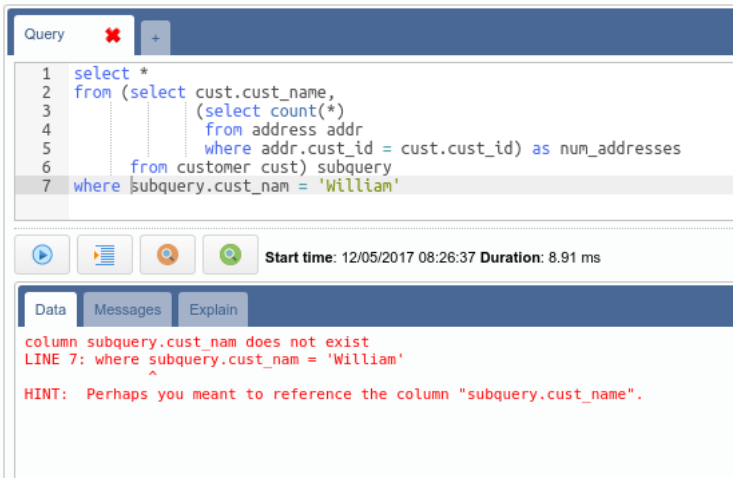
The SQL editor has a feature that helps a lot when creating new queries: SQL code completion. With this feature it is possible to autocomplete columns contained in a table referenced by an alias. To open the autocomplete interface you just have to type the alias and then the dot character:



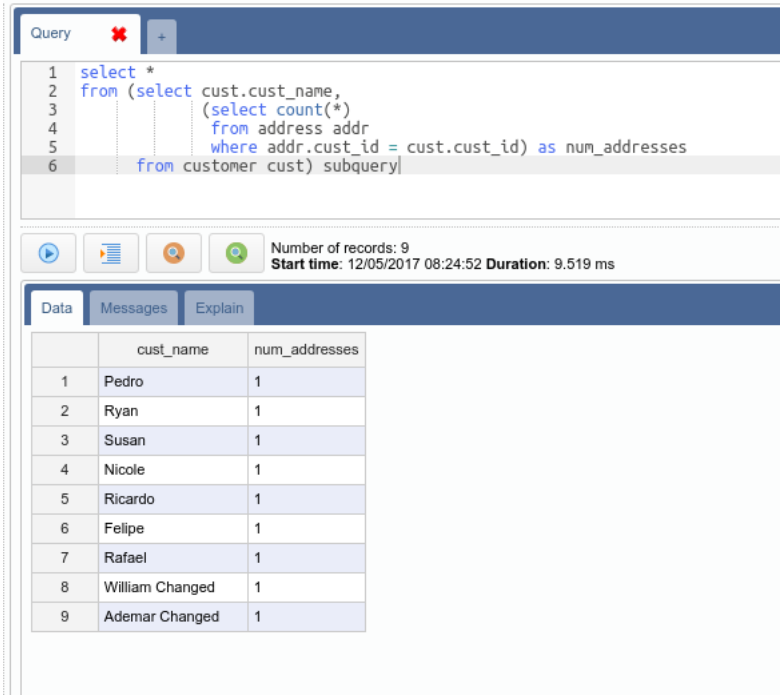
Besides autocompleting table columns the editor also searches for columns contained in subqueries:



If the query raises an error, OmniDB will show the error message in the *Data* tab and the cursor will be placed in the position indicated by the error message:



When the query is successful, OmniDB shows the number of records returned by the query, the start time and the duration of the query. The *Data* tab will show a data grid with the records records returned by the query.



Just like in the record editing interface each cell can be visualized separately by right clicking it.

If you click on the *Indent SQL* button, OmniDB will reorganize the SQL text to make it prettier:

8 Visualizing Query Plans

OmniDB 2.2.0 introduced a very useful feature: graphical query plan visualization. This may come in handy when writing or optimizing queries, since it allows you to easily identify performance bottlenecks in your SQL query.

For this feature, *SQL Query* inner tab shows 2 buttons: *Explain* (magnifier in orange circle button)and *Explain Analyze* (magnifier in green circle button).

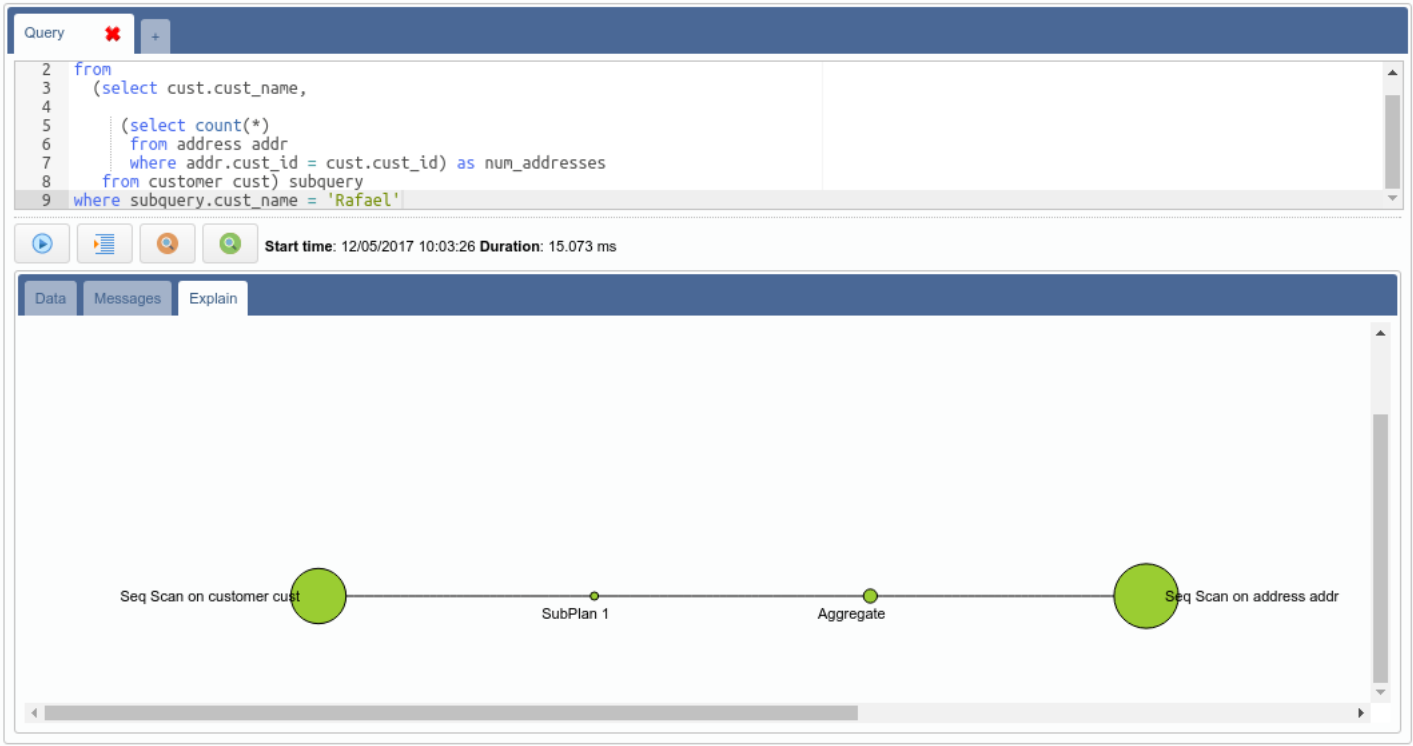
8.0.0.1 Textual visualization

When you click the *Explain* button, OmniDB will execute an `EXPLAIN` command in your query. Initial visualization is *textual* and will show exactly the output of the `EXPLAIN` command, but with colored bars representing the estimated cost. The higher the cost, the darker and wider the bar.

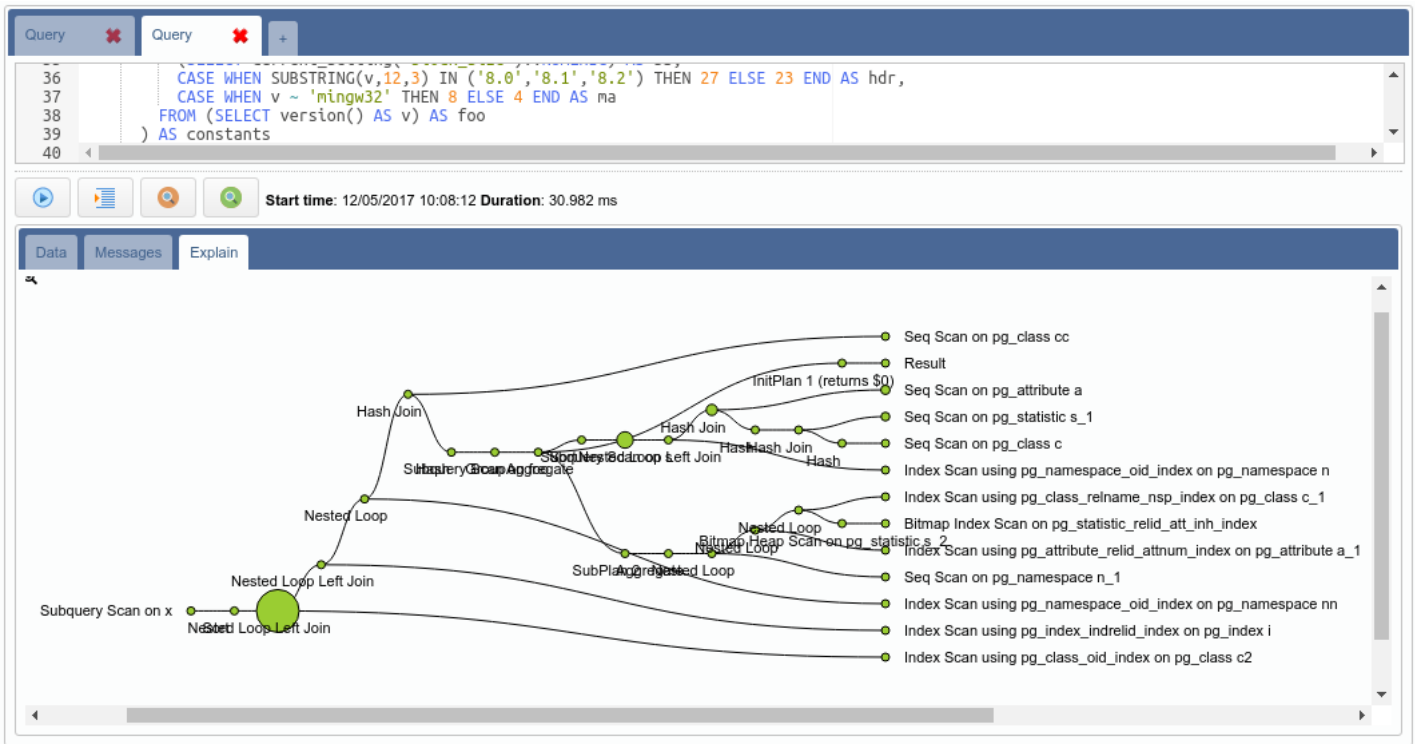
When you click the *Explain Analyze* button, OmniDB will execute an `EXPLAIN ANALYZE` command in your query. Beware that this command will really execute the query. Also, the textual visualization will show much more information, and the costs are not estimated as in those provided by the `EXPLAIN` command; they are real costs.

8.0.0.2 Tree visualization

Both *Explain* and *Explain Analyze* modes also can graphically represent the textual output into a *tree* diagram. Each circle represent a node executed by the query plan, and the larger the circle, the higher the cost.



When queries become more and more complex, also its query plan can be very complex. With such queries (like the *check bloat* query we executed below) the tree visualization can be very interesting:



The query plan visualization component allows you to easily switch between textual and 2 tree visualizations, which can be zoomed in and out.

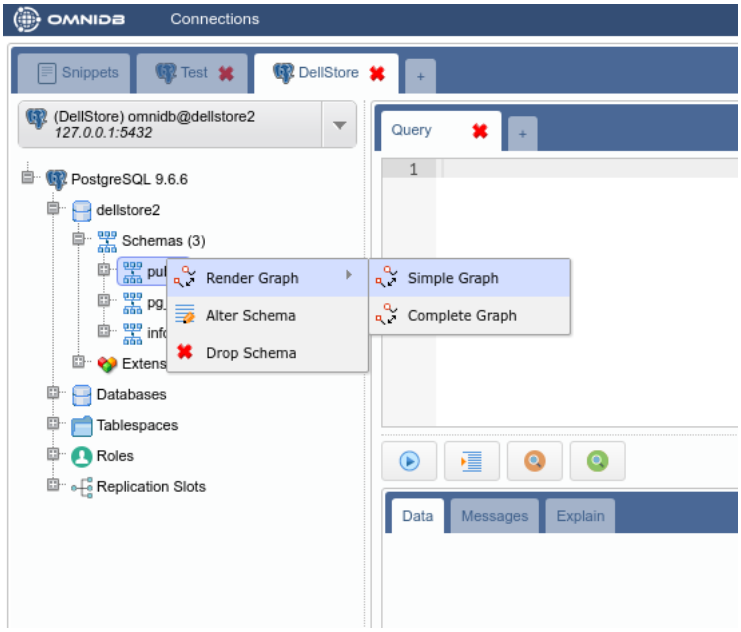
9 Visualizing Data

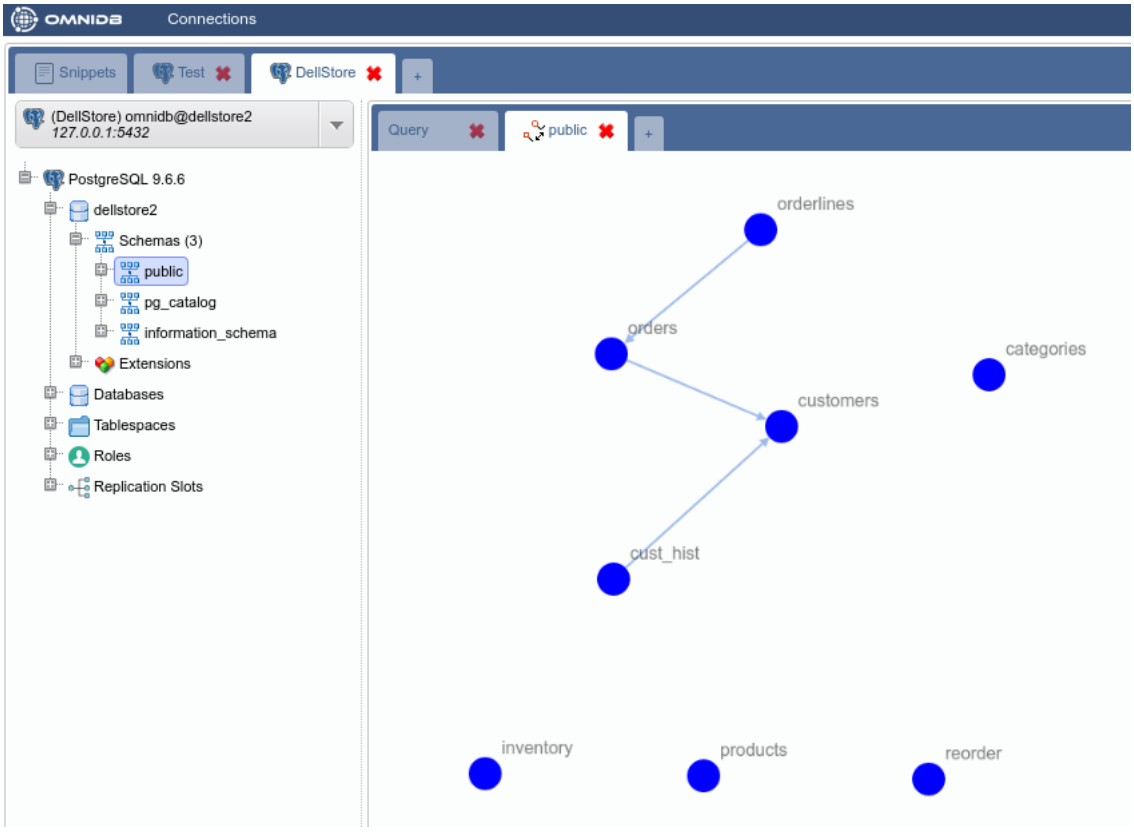
This feature displays a graph with nodes representing tables and edges representing table relationships with foreign keys. Using the mouse, the user is able to zoom in, zoom out, and drag and drop nodes to change its position.

There are two types of graphs: *Simple Graph* and *Complete Graph*.

9.0.0.1 Simple graph

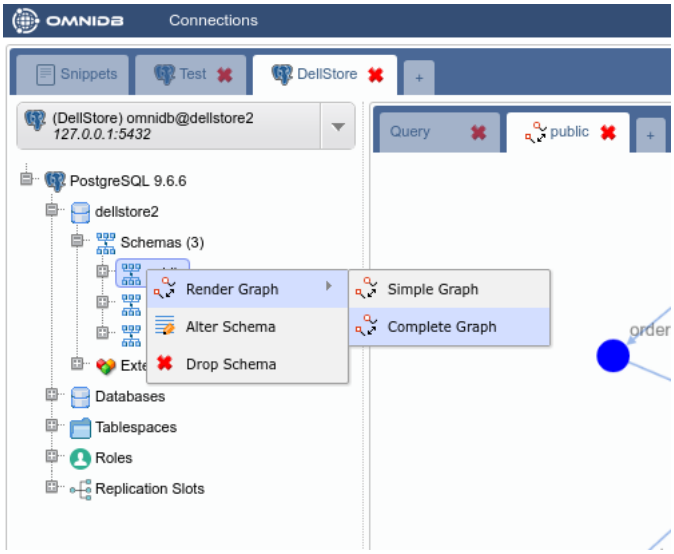
To access it just right click the root node of the tree and then select the action *Render Graph* > *Simple Graph*:

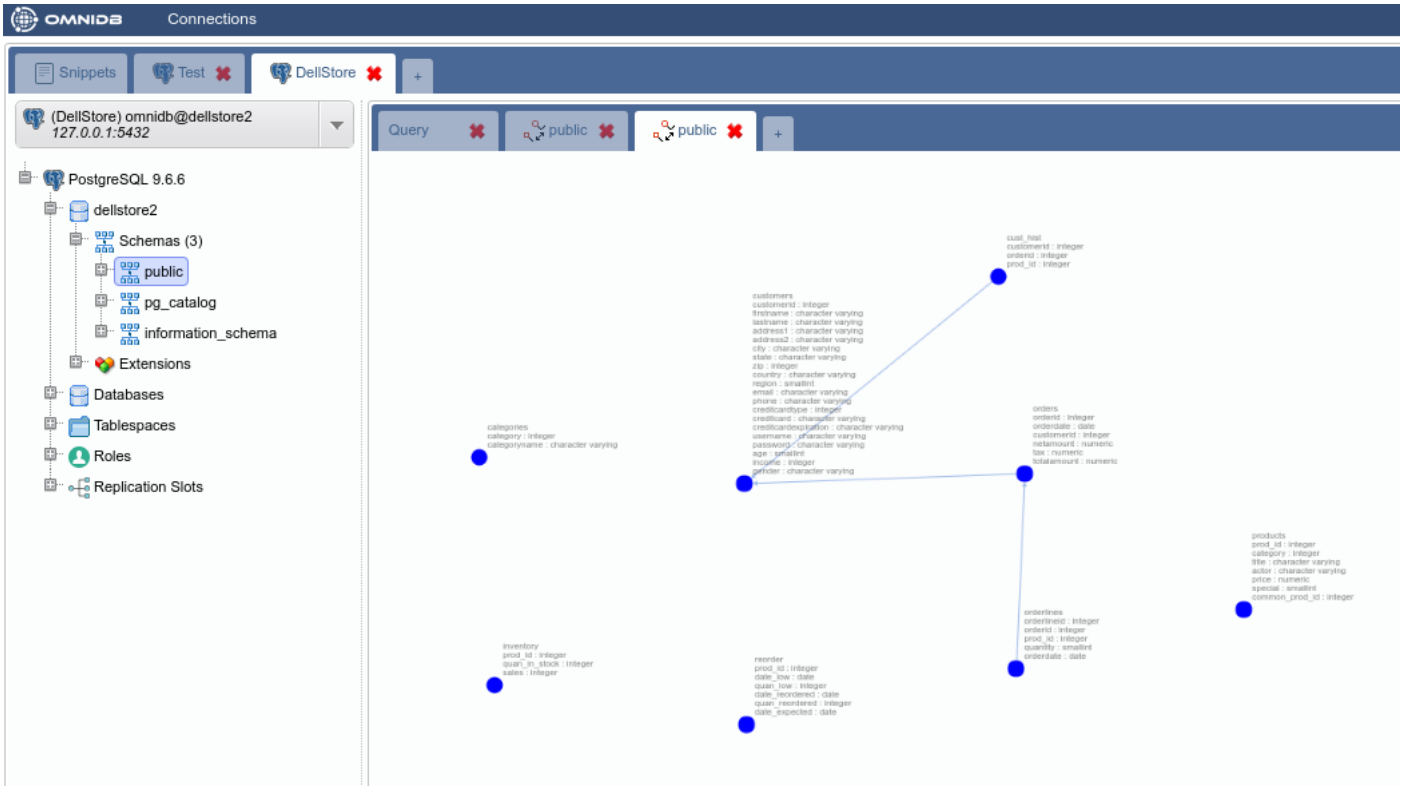




9.0.0.2 Complete graph

This graph displays tables with all its columns and respective data types. Additionally, edges now are labeled with information about the specific foreign key. To access it just right click the root node of the tree and then select the action *Render Graph* > *Complete Graph*:

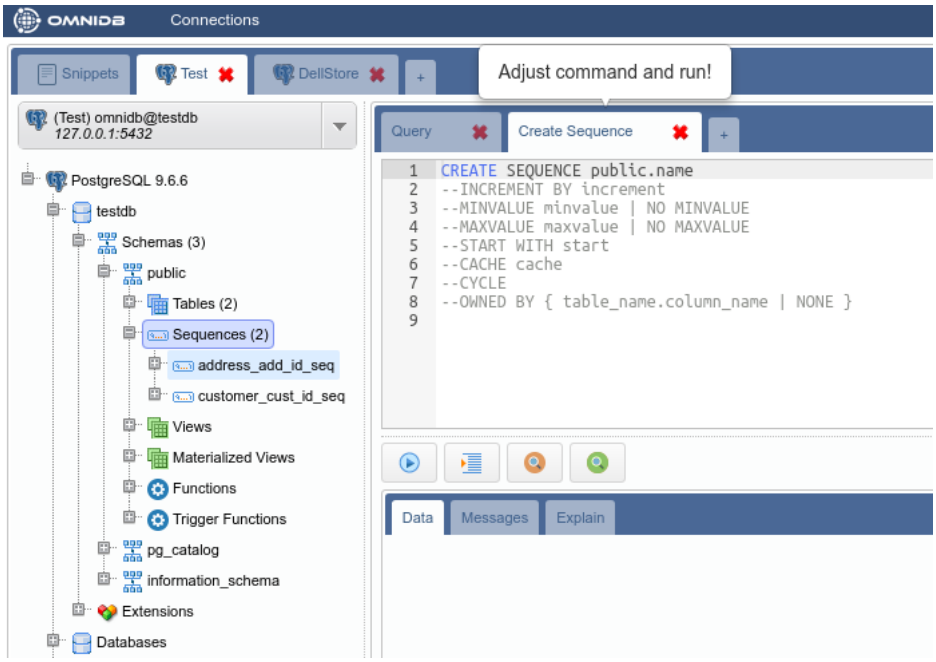




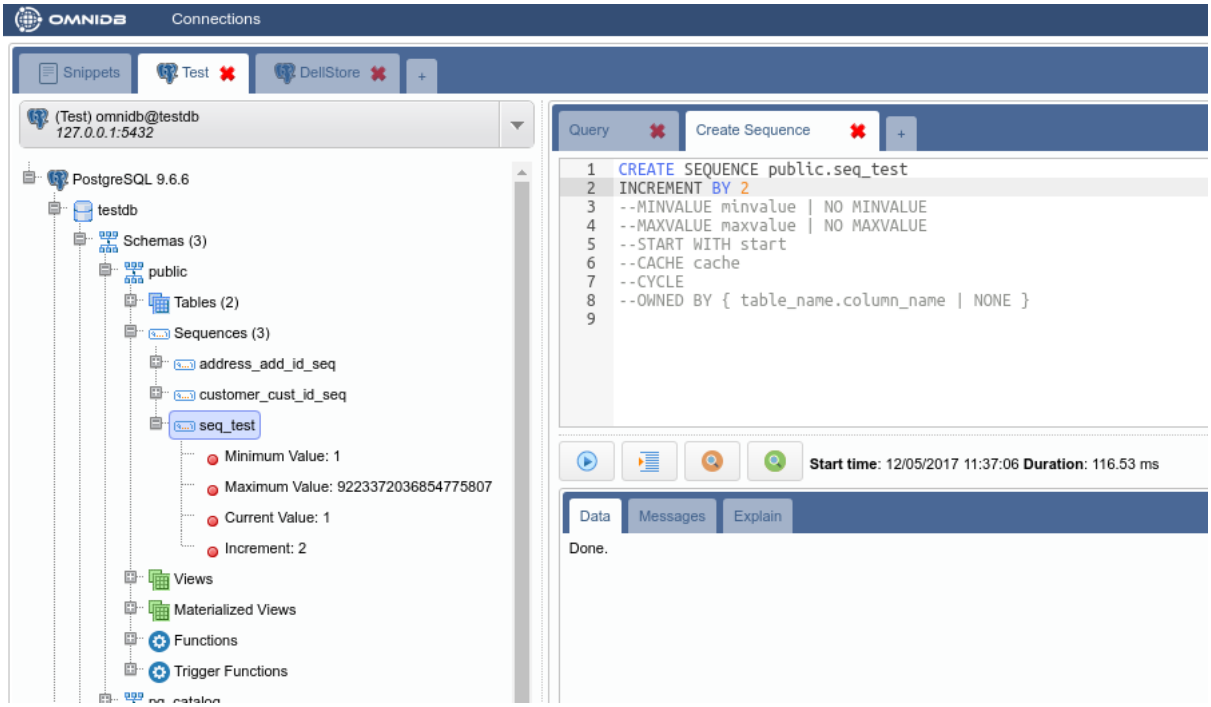
10 Managing other Elements

With the exception of tables, all PostgreSQL structures are possible to be managed with the use of *SQL templates*. This gives the user more power than using graphical forms to manipulate structures.

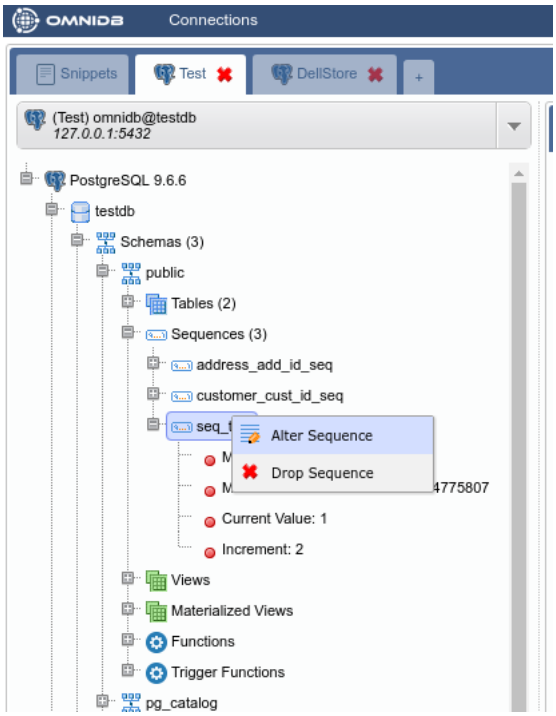
For example, let's consider the sequences inside the schema `public` of the database `testdb`. To create a new sequence, right click on the *Sequences* node, and choose *Create Sequence*.

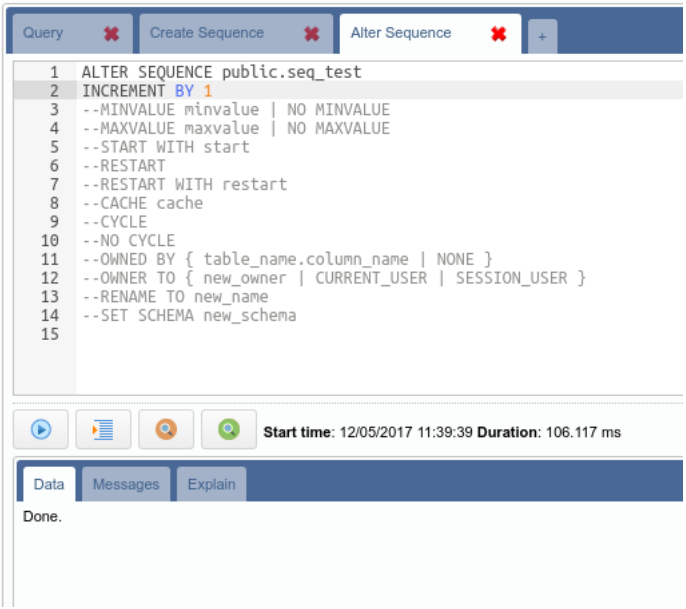


After you change the name of the sequence, you can uncomment other command options and set them accordingly to your needs. When the entire command looks fine, just execute it and the new sequence will be created:



With right click on an existing sequence, you can alter or drop it. It will work the same way as the creation, by using a SQL template for the user to change.

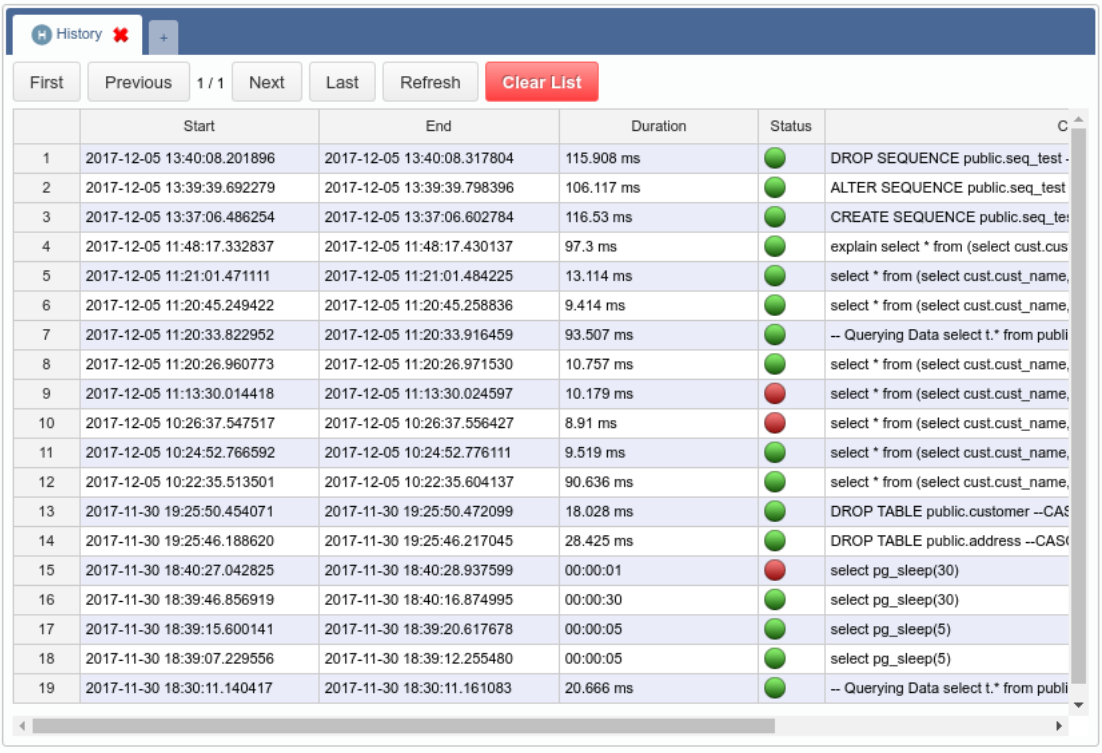




11 Additional Features

11.0.0.1 SQL History

Every interaction the user does with every database is logged in OmniDB's *SQL History*. To access it you need to click on the icon with an *H* on the upper right corner. OmniDB will open an inner tab with all actions in a paginated grid.



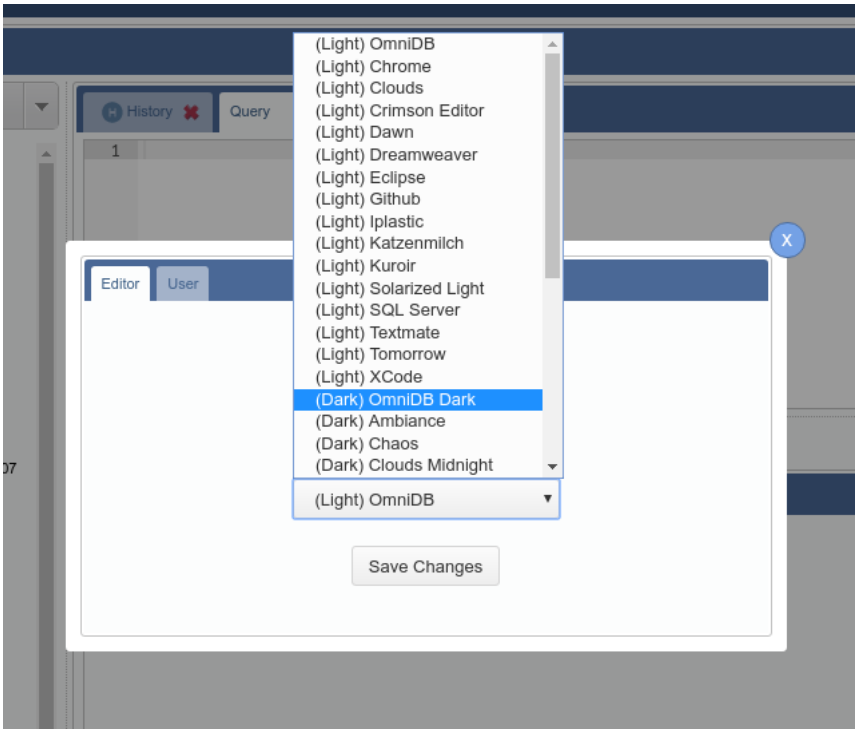
	Start	End	Duration	Status	C
1	2017-12-05 13:40:08.201896	2017-12-05 13:40:08.317804	115.908 ms		DROP SEQUENCE public.seq_test -
2	2017-12-05 13:39:39.692279	2017-12-05 13:39:39.798396	106.117 ms		ALTER SEQUENCE public.seq_test
3	2017-12-05 13:37:06.486254	2017-12-05 13:37:06.602784	116.53 ms		CREATE SEQUENCE public.seq_te
4	2017-12-05 11:48:17.332837	2017-12-05 11:48:17.430137	97.3 ms		explain select * from (select cust.cus
5	2017-12-05 11:21:01.471111	2017-12-05 11:21:01.484225	13.114 ms		select * from (select cust.cust_name,
6	2017-12-05 11:20:45.249422	2017-12-05 11:20:45.258836	9.414 ms		select * from (select cust.cust_name,
7	2017-12-05 11:20:33.822952	2017-12-05 11:20:33.916459	93.507 ms		-- Querying Data select t.* from publi
8	2017-12-05 11:20:26.960773	2017-12-05 11:20:26.971530	10.757 ms		select * from (select cust.cust_name,
9	2017-12-05 11:13:30.014418	2017-12-05 11:13:30.024597	10.179 ms		select * from (select cust.cust_name,
10	2017-12-05 10:26:37.547517	2017-12-05 10:26:37.556427	8.91 ms		select * from (select cust.cust_name,
11	2017-12-05 10:24:52.766592	2017-12-05 10:24:52.776111	9.519 ms		select * from (select cust.cust_name,
12	2017-12-05 10:22:35.513501	2017-12-05 10:22:35.604137	90.636 ms		select * from (select cust.cust_name,
13	2017-11-30 19:25:50.454071	2017-11-30 19:25:50.472099	18.028 ms		DROP TABLE public.customer --CAS
14	2017-11-30 19:25:46.188620	2017-11-30 19:25:46.217045	28.425 ms		DROP TABLE public.address --CAS
15	2017-11-30 18:40:27.042825	2017-11-30 18:40:28.937599	00:00:01		select pg_sleep(30)
16	2017-11-30 18:39:46.856919	2017-11-30 18:40:16.874995	00:00:30		select pg_sleep(30)
17	2017-11-30 18:39:15.600141	2017-11-30 18:39:20.617678	00:00:05		select pg_sleep(5)
18	2017-11-30 18:39:07.229556	2017-11-30 18:39:12.255480	00:00:05		select pg_sleep(5)
19	2017-11-30 18:30:11.140417	2017-11-30 18:30:11.161083	20.666 ms		-- Querying Data select t.* from publi

Each action shows date time it started, the time it ended, the duration, the status and the command. As every grid in OmniDB, you can right click on the command and click *View Content*, where another pop-up will open showing the content in a larger text editor.

11.0.0.2 User Settings

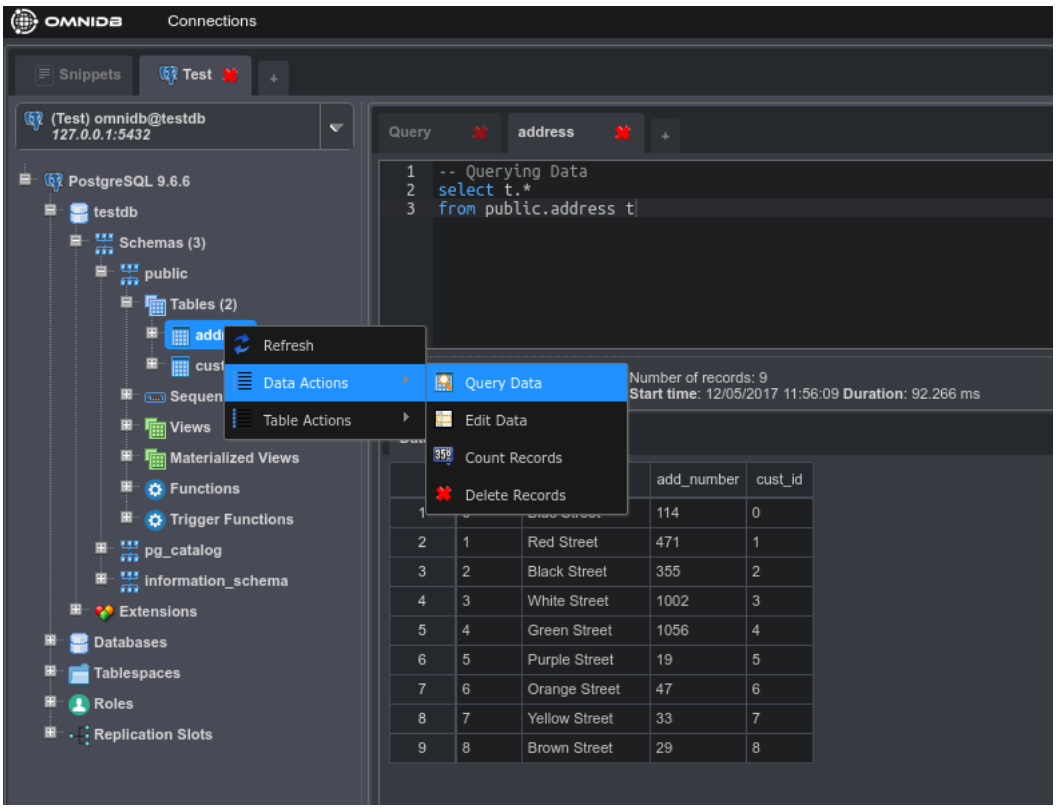
Also in the upper right corner, by clicking in the gear-like icon, OmniDB will open the *User Settings* pop-up. It is composed by two tabs:

- **User**: Allows the user to change its password. More user settings will be added in the future.
- **Editor**: Allows the user to change the font size of the SQL Editor, and also change the entire OmniDB theme. There are a lot of OmniDB themes, each of them change the syntax highlight color of the editor. They are also categorized in light and dark themes. A light theme is the default; a dark theme will change the entire interface of OmniDB.



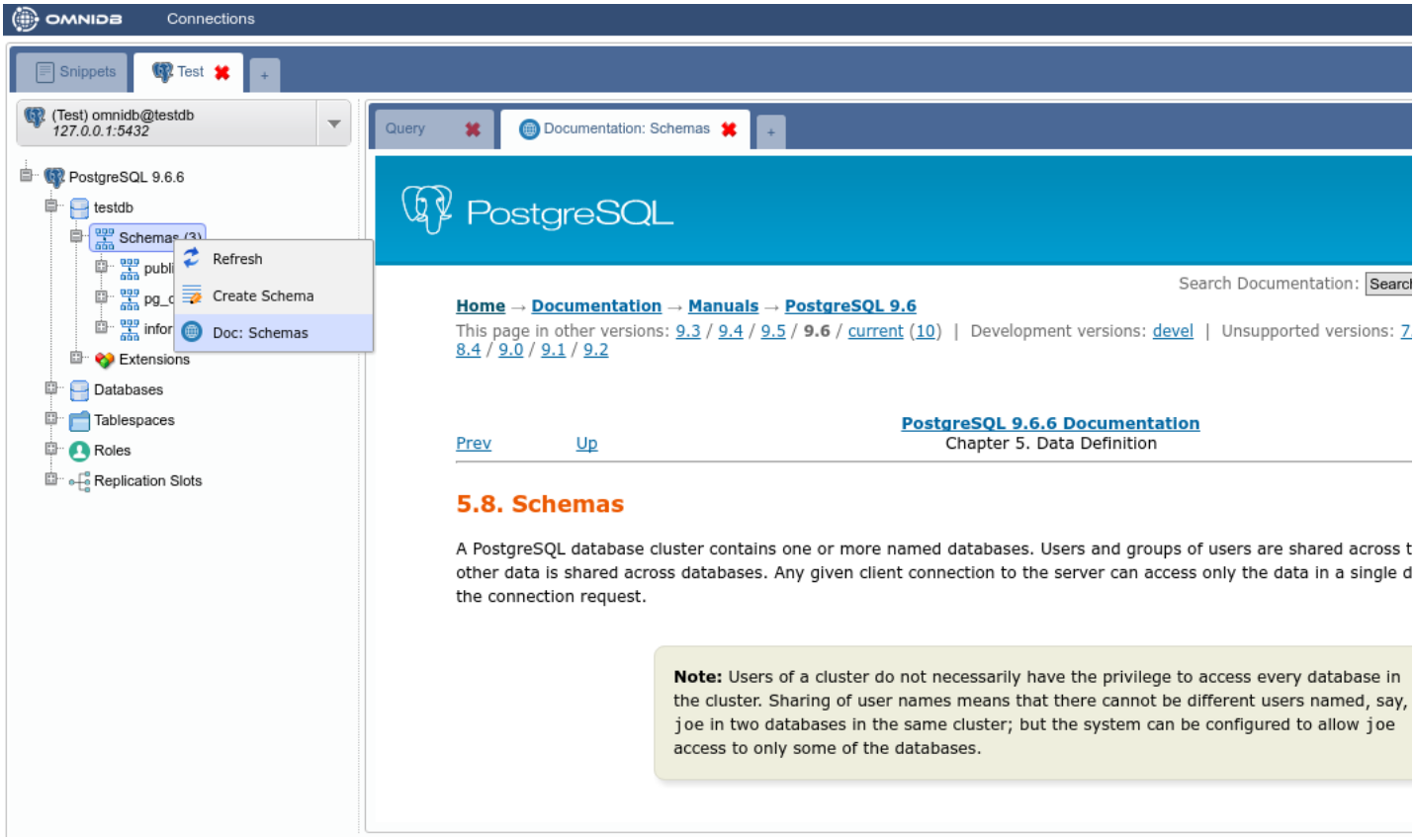
Every change in user settings require that you either:

- Refresh the page, if you are using OmniDB server and the interface through a web browser; or
- Open and close OmniDB, if you are using OmniDB app.



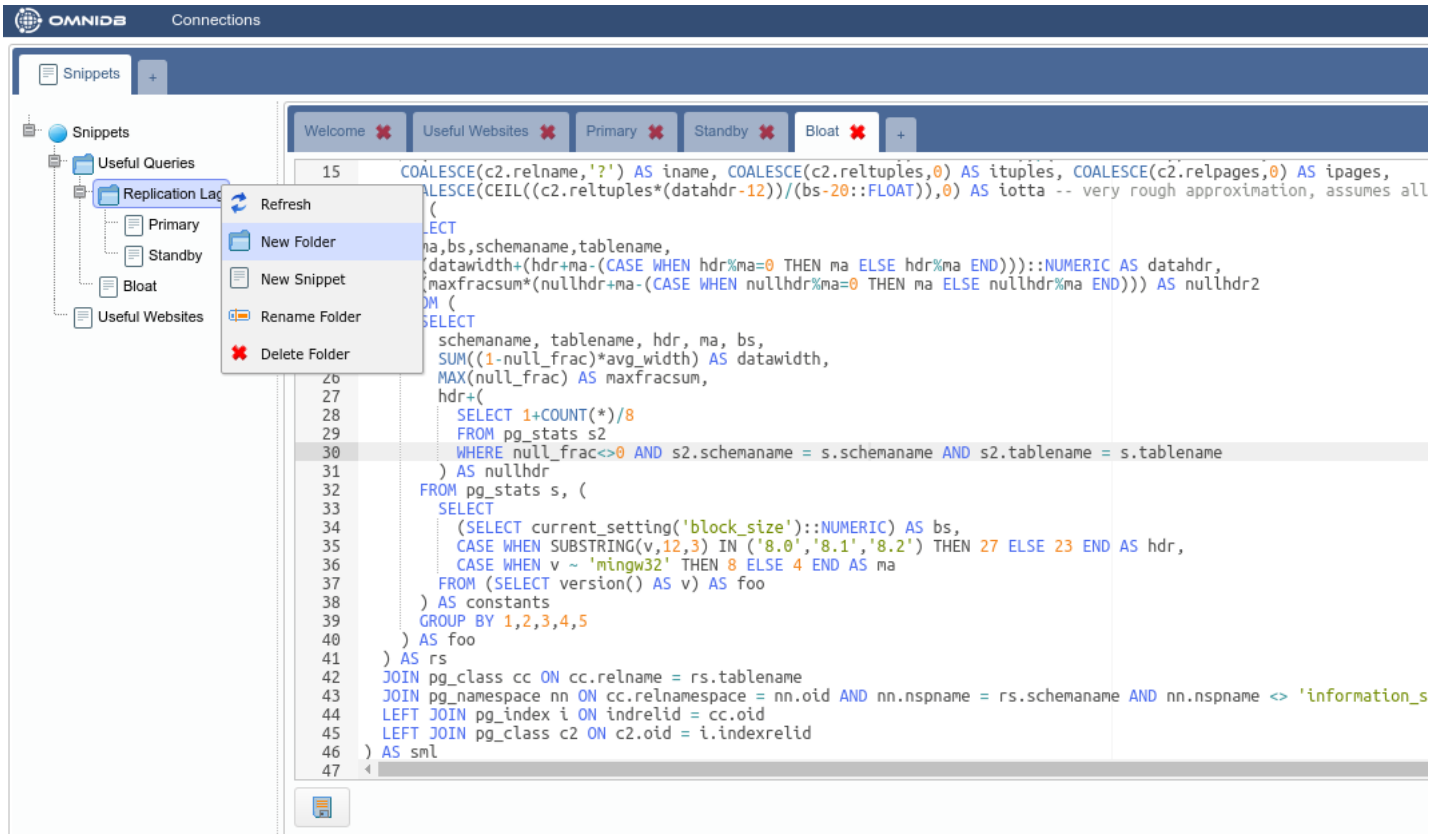
11.0.0.3 Contextual Help

Most of tree nodes (generally grouping ones like *Schemas* or *Tables*) offer contextual help. This feature can be accessed by right-clicking the tree node. When you click in the *Doc: ...* option, OmniDB will open an inner tab showing a web browser pointing to the specific page in the online **PostgreSQL Documentation**. Also, it will redirect to the specific page considering the PostgreSQL version you are connected to.



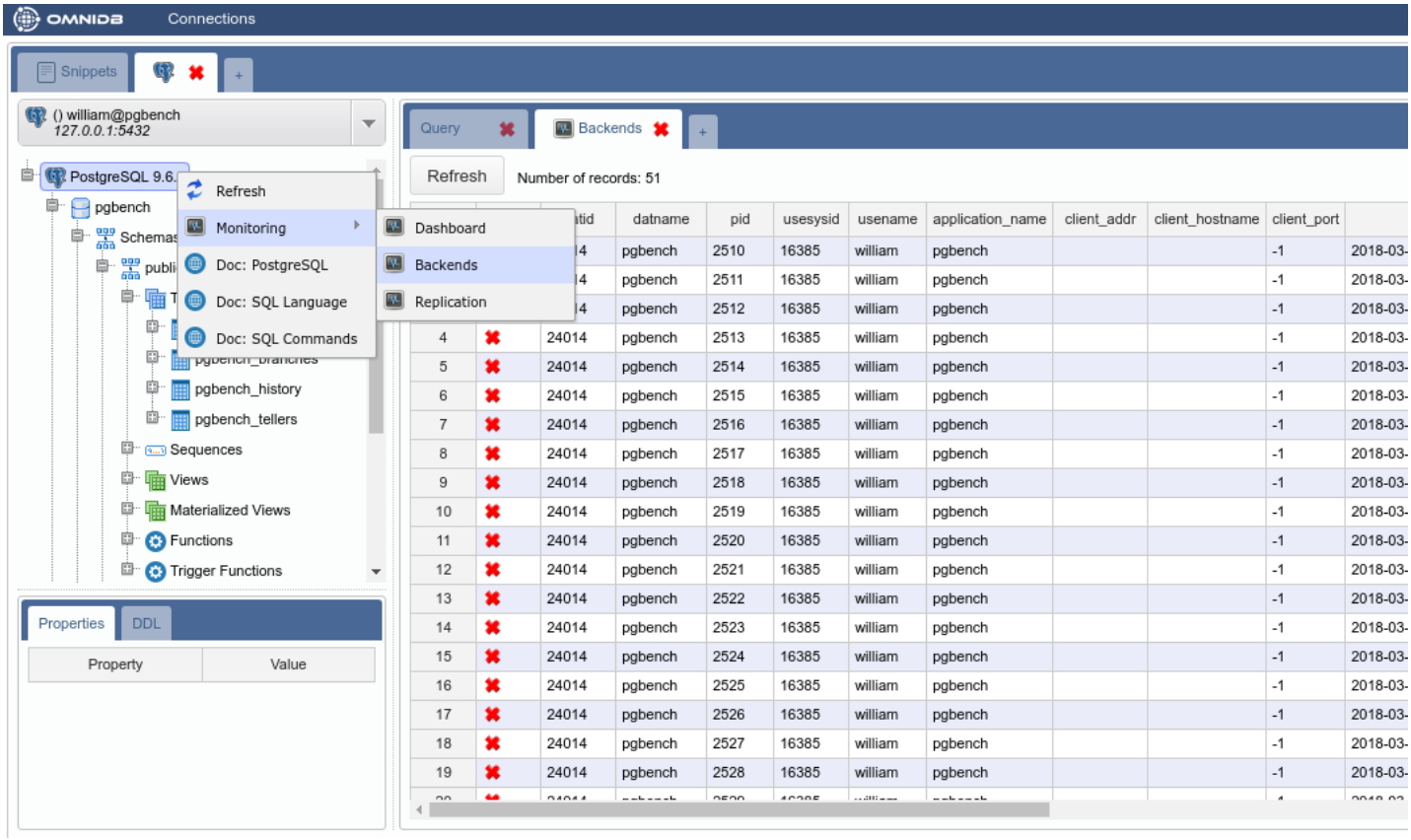
11.0.0.4 Snippets

Workspace Window has a fixed outer tab with an useful feature called *Snippets*. With this feature you can store queries, command instructions and any other kinds of text you want. You can also structure the snippets in a directory tree the way you want. All directories and snippets you create are stored inside of `omnidb.db` user database and persist when you upgrade OmniDB.

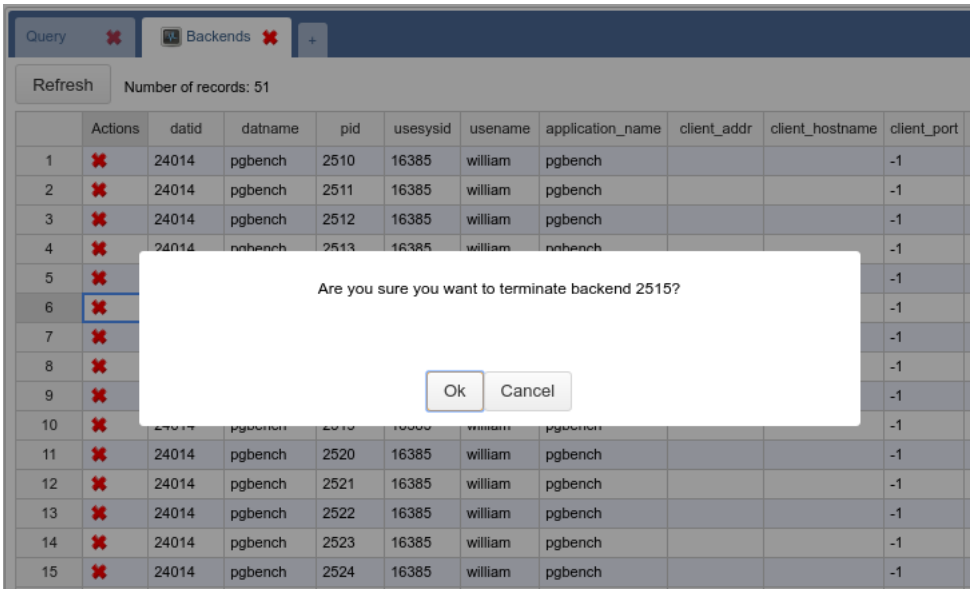


11.0.0.5 Backend Management

By right-clicking in the tree root node, then moving mouse pointer to *Monitoring* and then clicking on *Backends*, the user can see all activities going on in the database. Some information are hidden for normal users, only database superusers are allowed to see.

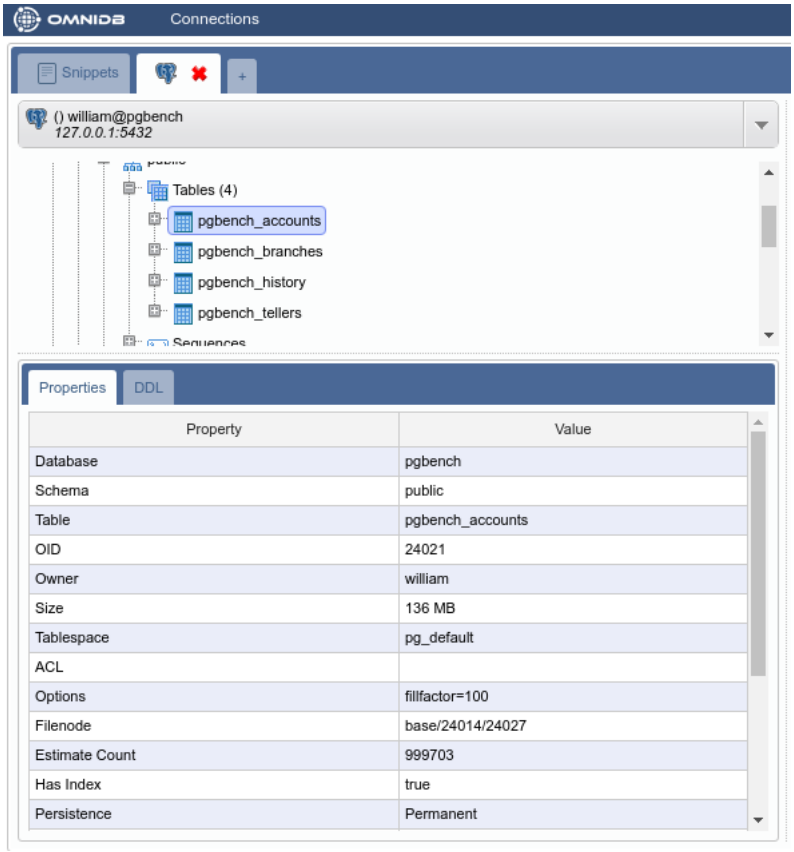


By clicking in the red *X* in the *Actions* column, you can terminate the backend. A confirmation popup will appear.

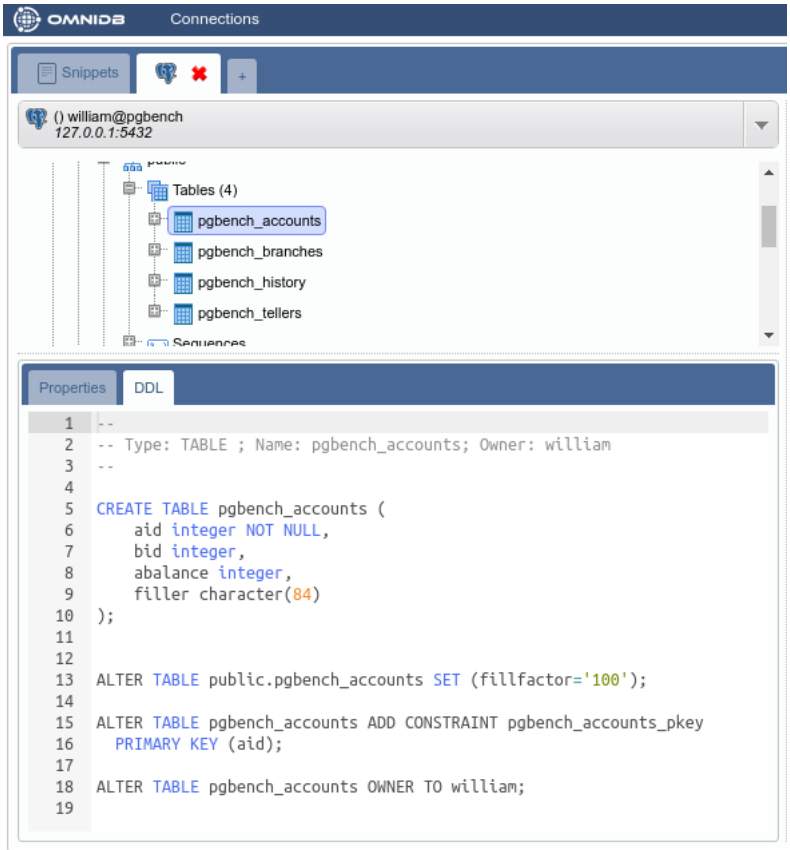


11.0.0.6 Properties and DDL

By clicking on most of objects in the treeview (tables, sequences, views, roles, databases, etc), the user will be able to see a very comprehensive list of properties of the object.



In the other panel called *DDL*, the user will be able to see the SQL DDL source code that can be used to re-create the object. The user can copy this text and paste it wherever he/she wants.



12 OmniDB Config Tool

Every installation of OmniDB also comes with a small CLI utility called *OmniDB Config*. It will have a different file name, depending on the way you installed OmniDB:

- If you are using a tarball or zip package, it is called **omnidb-config**, for both server and app versions;
- If you used an installer (like the .deb file) of server version, it is called **omnidb-config-server**;
- If you used an installer of app version, it is called **omnidb-config-app**.

Despite having different names, the utility does exactly the same. If you used an installer, it will be put in your `$PATH`.

```
user@machine:~$  omnidb-config-app --help
Usage: omnidb-config-app [options]

Options:
  --version             show program\'s version number and exit
  -h, --help            show this help message and exit
  -c username password, --createsuperuser=username password
                        create super user: -c username password
  -a, --vacuum          databases maintenance
  -r, --resetdatabase   reset databases
```

12.0.0.1 Create super user

Option `-c` allows you to create a new super user, without needing to open OmniDB interface.

```
user@machine:~$ omnidb-config-app -c william password
Creating superuser...
Superuser created.
```

12.0.0.2 Vacuum

OmniDB has two databases:

- `omnidb.db`: Stores all users and connections, and other OmniDB related stuff;
- *Sessions database*: Stores Django user sessions.

Both databases are SQLite, so it can be useful to vacuum them sometimes to reduce file size. This can be done with the `-a` option.

```
user@machine:~$ omnidb-config-app -a
Vacuuming OmniDB database...
Done.
Vacuuming Sessions database...
Done.
```

12.0.0.3 Reset database

If you wish to wipe out all OmniDB information and get a clean database as it was just installed, you can use the `-r` option. Use it with caution!

```
user@machine:~$ omnidb-config-app -r
*** ATENTION *** ALL USERS DATA WILL BE LOST
Would you like to continue? (y/n) y
Cleaning users...
Done.
Cleaning sessions...
Vacuuming OmniDB database...
Done.
Vacuuming Sessions database...
Done.
```

13 Writing and Debugging PL/pgSQL Functions

13.0.0.1 Introduction

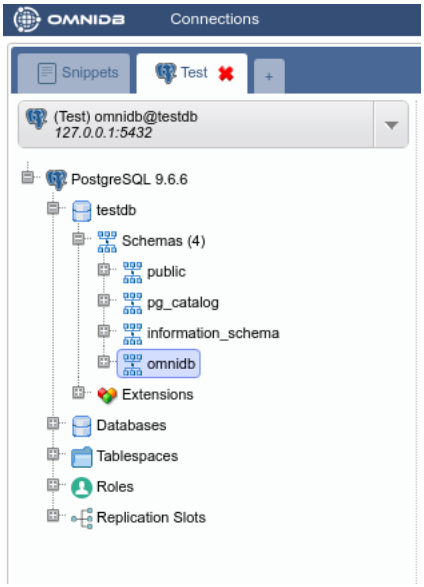
PostgreSQL is more than a RDBMS engine. It is a developing platform. It provides a very powerful and flexible programming language called `PL/pgSQL`. Using this language you can write your own *user-defined functions* to achieve abstraction levels and procedural calculations that would be difficult to achieve with plain SQL (and sometimes impossible to achieve without context-switching with the application). While you always could develop and manage your own functions within OmniDB, it is a recent feature that allows you to also *debug* your own functions.

OmniDB 2.3.0 introduces this great feature: a debugger for PL/pgSQL functions. It was implemented by scratch and takes advantage of hooks, an extensibility in PostgreSQL's source code that allows us to perform custom actions when specific events are triggered in the database. For the debugger we use hooks that are triggered when PL/pgSQL functions are called, and each statement is executed.

This requires the user to install a binary library called `omnidb_plugin` and enable it in PostgreSQL's config file. The debugger also uses a special schema with special tables to control the whole debugging process. This can be manually created or with an extension.

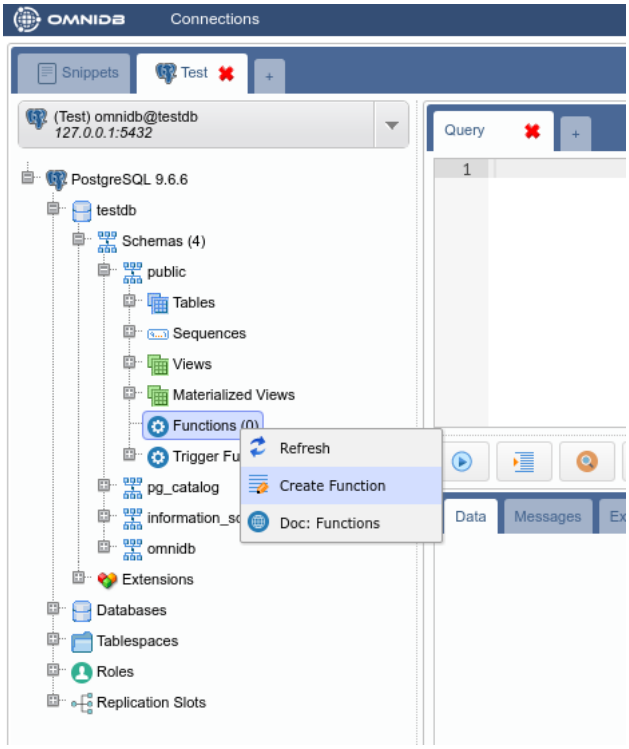
For more details on the installation, please refer to the instructions. Also please read the notes in this document, to be aware that currently there are some limitations.

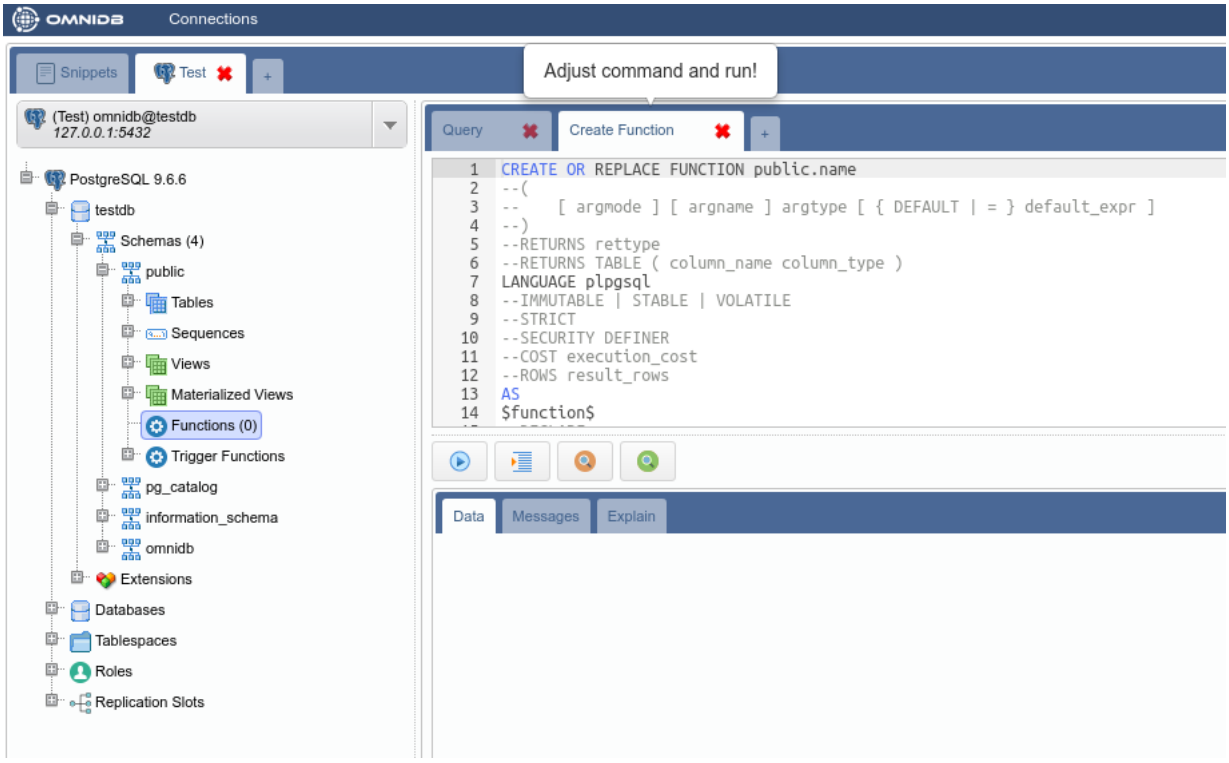
After successfully installing the debugger, you will se a schema called `omnidb` in your database. Also, if you compiled the debugger yourself, you can install it as a PostgreSQL extension, and in this case it will appear under the *Extensions* tree node.



13.0.0.2 Writing functions

In the `public` schema, right-click the `Functions` node and click on *Create Function*. It will open a *SQL Query* inner tab, already containing a SQL Template to help you create your first PL/pgSQL function.





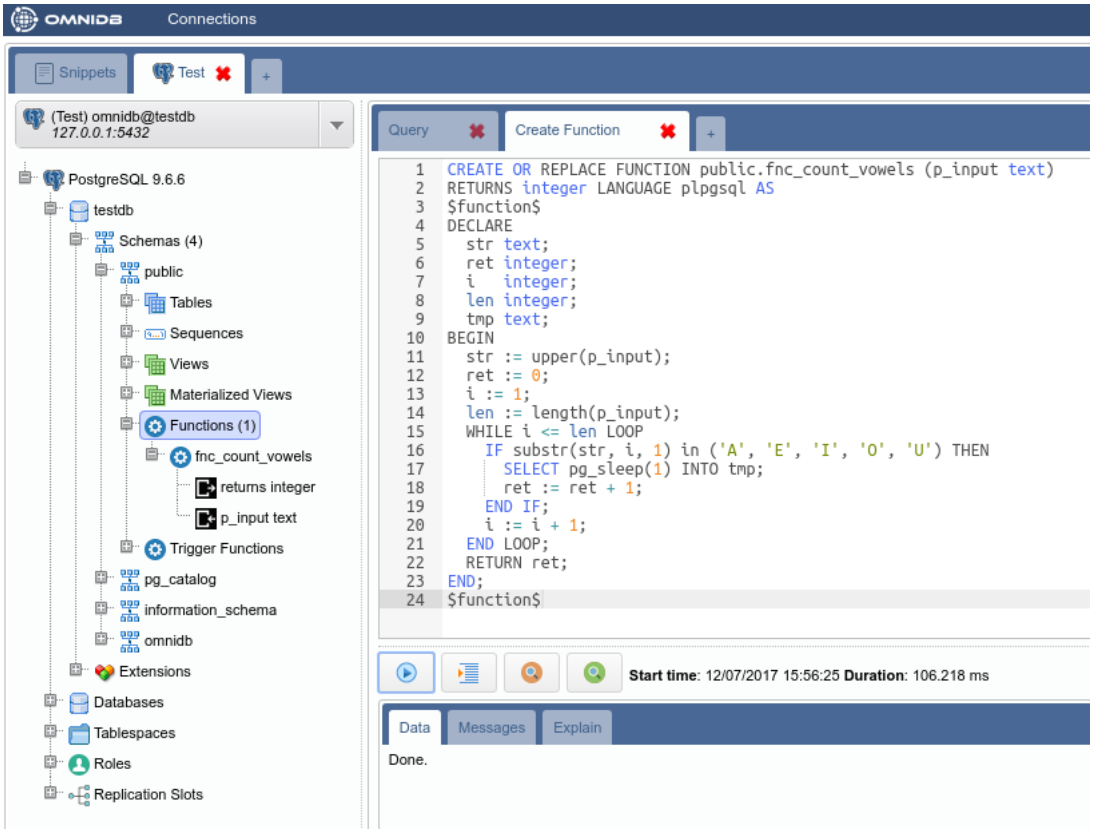
You can refer to PostgreSQL documentation on how to write user-defined functions. No need to open a new browser tab: just right-click the *Functions* node and click on *Doc: Functions* to view the documentation inside OmniDB.

For now, let us replace this SQL template entirely for the source code below:

```
CREATE OR REPLACE FUNCTION public.fnc_count_vowels (p_input text)
RETURNS integer LANGUAGE plpgsql AS
$function$
DECLARE
  str text;
  ret integer;
  i   integer;
  len integer;
  tmp text;
BEGIN
  str := upper(p_input);
  ret := 0;
  i := 1;
  len := length(p_input);
  WHILE i <= len LOOP
    IF substr(str, i, 1) in ('A', 'E', 'I', 'O', 'U') THEN
      SELECT pg_sleep(1) INTO tmp;
      ret := ret + 1;
    END IF;
    i := i + 1;
  END LOOP;
  RETURN ret;
END;
$function$
```

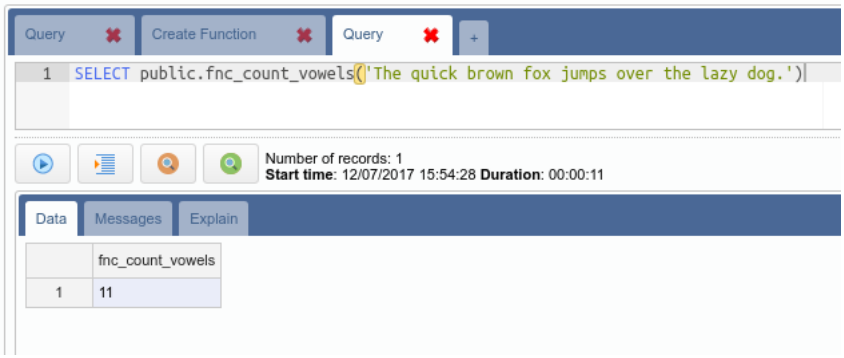
This will create a function called `fnc_count_vowels` inside the schema `public`. This function takes a text argument called `p_input` and counts how many vowels there are in this *string*. Then returns this count.

To create the function, execute the command in the SQL Query inner tab. If successful, the function will appear under the *Functions* tree node (you can refresh it by right-clicking and then clicking in *Refresh*). By expanding the function node as well, you can see its return type and its argument.



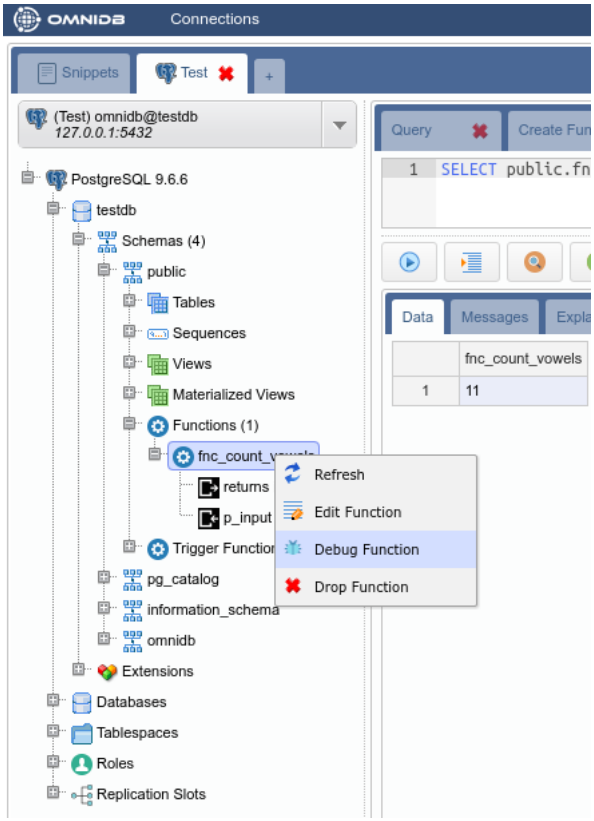
Now let us execute this new function for the first time. Open a simple SQL Query inner tab and execute the following SQL query:

```
SELECT public.fnc_count_vowels('The quick brown fox jumps over the lazy dog.')
```



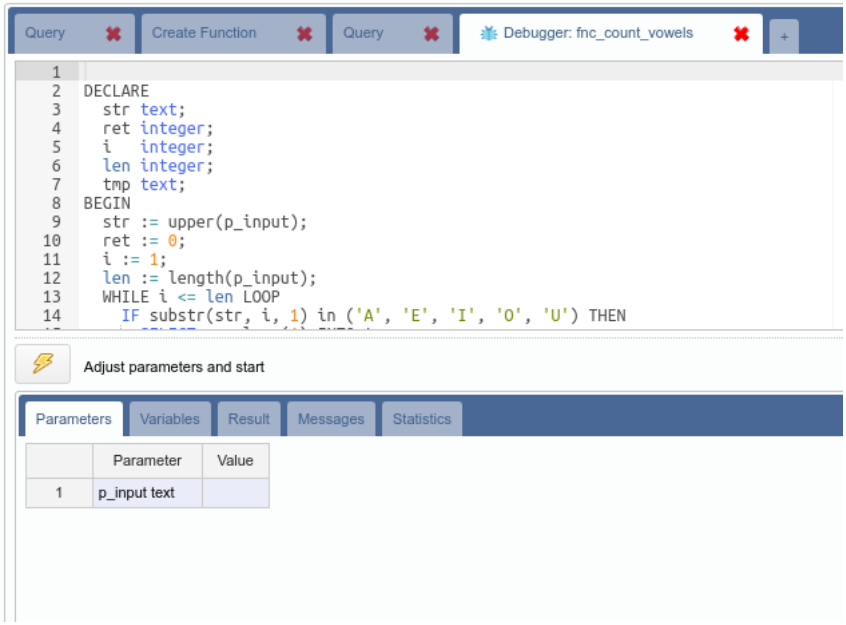
Note how the query returns a single value, containing the number of vowels in the text. Note also how the query took several seconds to finish; this is caused by the `pg_sleep` we put in the source code of the function `fnc_count_vowels`.

By right-clicking the function node, you can see there are actions to edit and drop it. As you probably guessed, each action will open SQL Query inner tabs with handy SQL templates in them. But the most interesting action right now is *Debug Function*. Go ahead and click it!



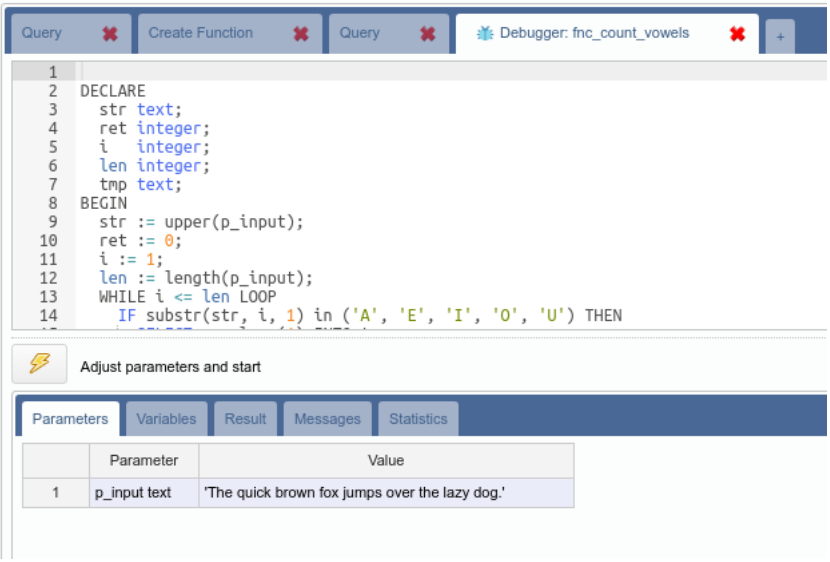
13.0.0.3 Debugging functions

The debugger is a specific inner tab composed of a SQL editor that will show the process step by step on top of the function source code, and 5 tabs to manage and view different parts of the debugger.



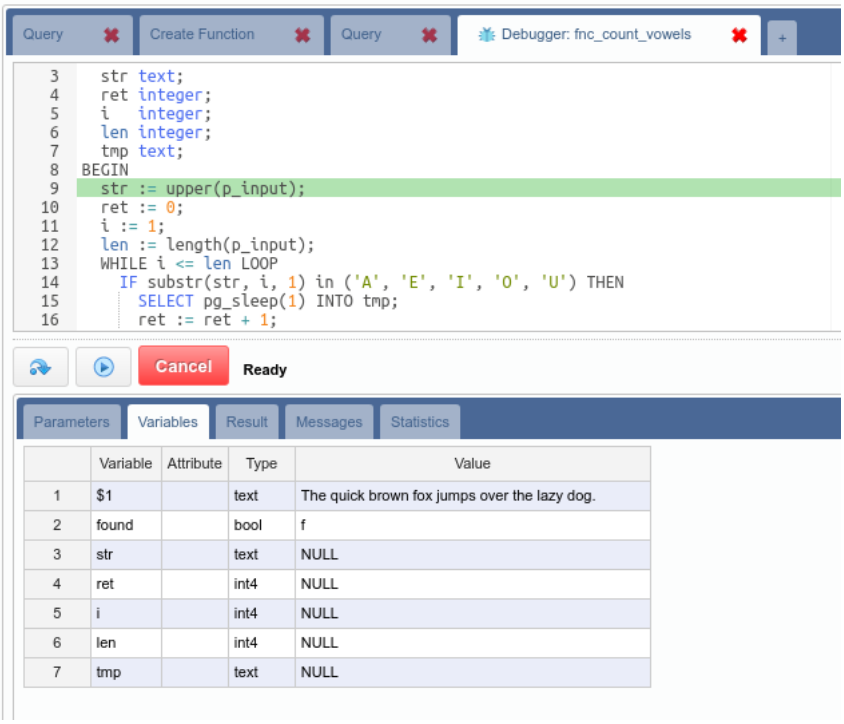
- **Parameters**: Before the debugging process starts, the user must provide all the parameters in this tab. Parameters must be provided exactly the same way you would provide them if you were executing the function in plain SQL, quoting strings for instance;
- **Variables**: This grid displays the current value of each variable that exists in the current execution context, it will be updated with every step;
- **Result**: When the function ends, this tab will show the result of the function call. It could be empty, a single value or even a set of rows;
- **Messages**: Messages returned explicitly by `RAISE` commands or even automatic messages from PostgreSQL will be presented in this tab;
- **Statistics**: At the end of the debugging process, a chart depicting execution times for each line in the function body will be presented in this tab. Additionally, the SQL editor will be updated with a set of colors representing a heat map, from blue to red, according to the max duration of each line.

Now let us start debugging this function. First thing to do is to fill *every* parameter in the *Parameters* tab:

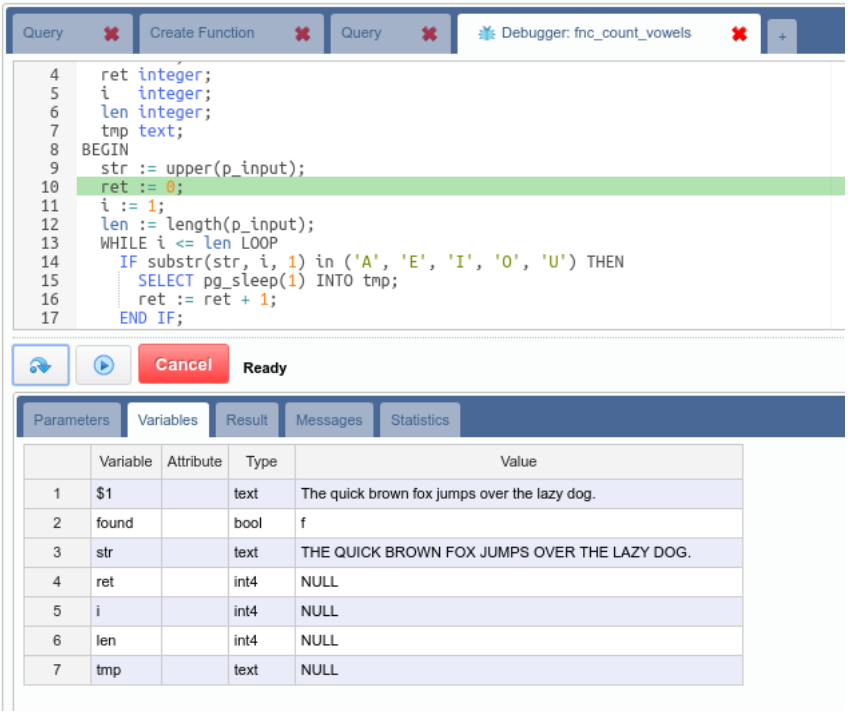


Then click on the *Start* button. Note how OmniDB automatically goes to the *Variables* tab, which is the interesting tab now that the function is being debugged. The argument `p_input` is now called `$1`, indicating the first argument of the function. Also note the variable `found`, which is a PostgreSQL reserved variable that indicates whether or not a query has returned values inside of the function.

Also note that OmniDB points to the first line of the source code of the function, highlighting it in green. This is the line that is about to be executed.

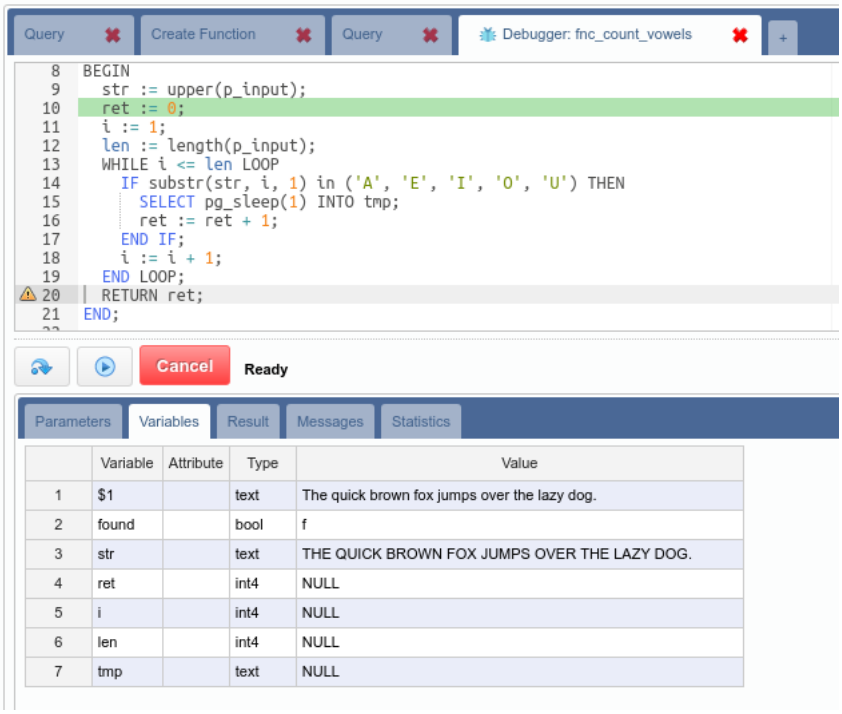


Now click in the first button below the SQL editor. It is the *Step Over* button, and it means that OmniDB will execute the next statement and stop right after it.

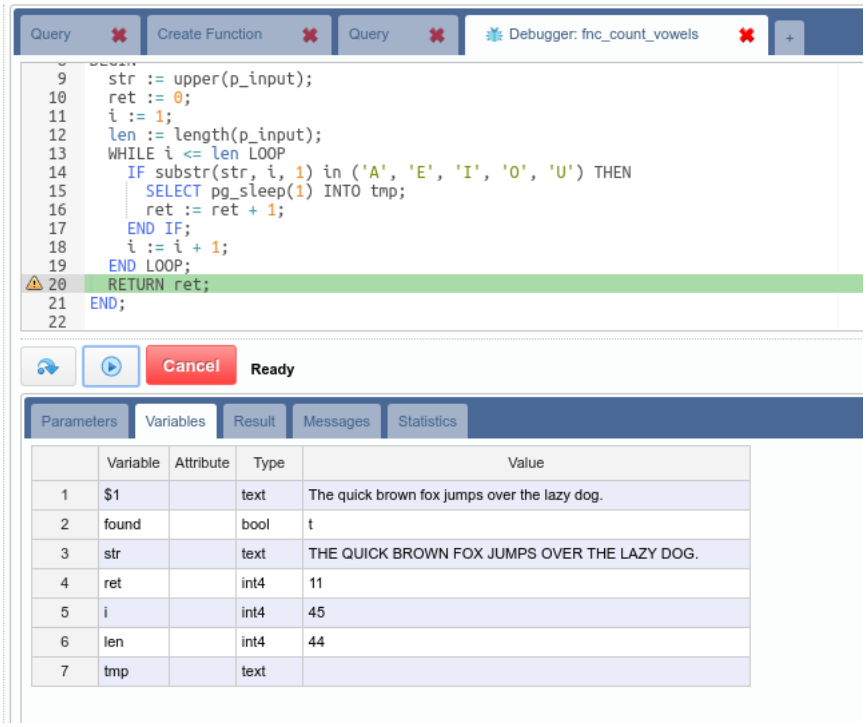


Note how the variable `str` has the value assigned to it during execution of line 9. Right now OmniDB is about to execute line 10, showing the current execution state.

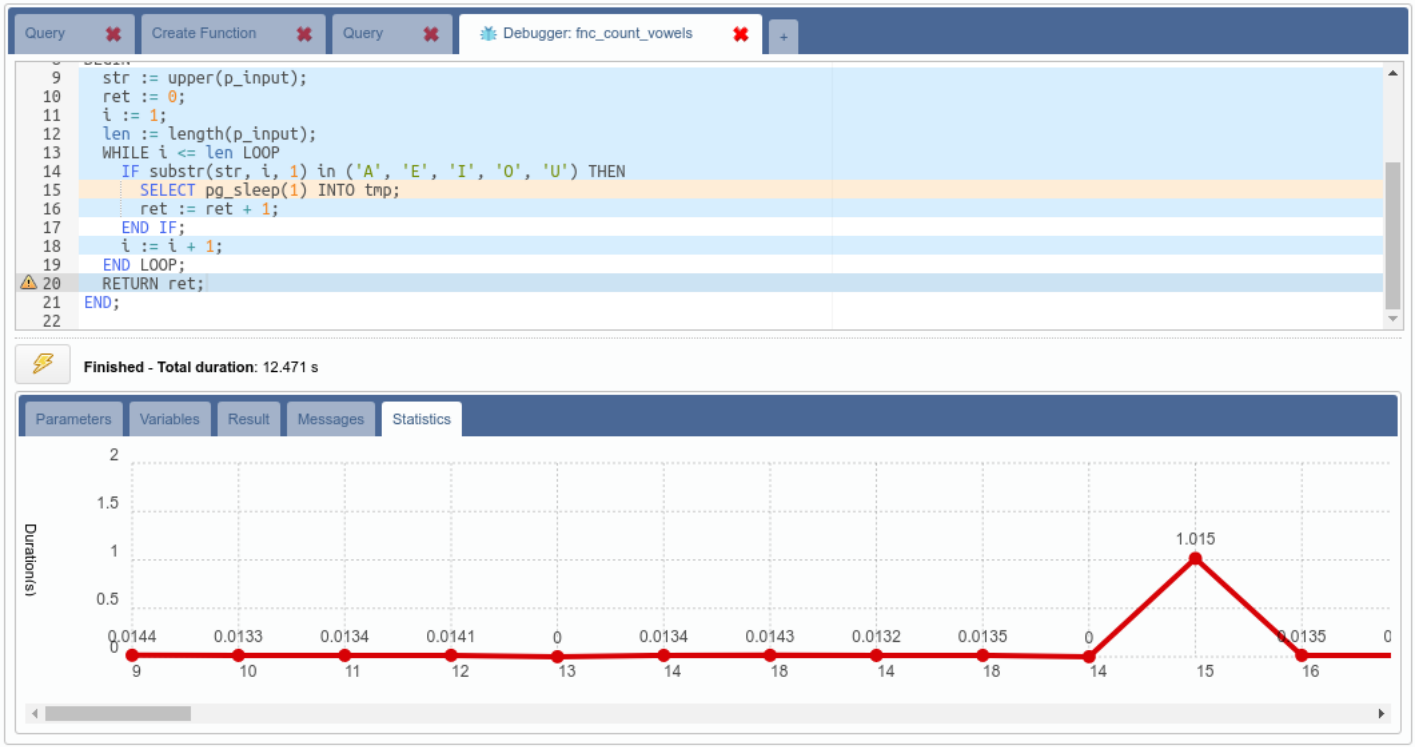
Now that you know how to step over, let us speed up things a little bit. Click on the header of the line 20, the last line of code. By doing this, you just placed a *breakpoint*. The debugger interface allows you to place one breakpoint at a time.



After setting a breakpoint, you can click in the second button, *Resume*. OmniDB will carry on with the debugging process until it reaches the line of code with the breakpoint. This may take a while because of the `pg_sleep` commands we put in the source code. Note that if you click this button without previously setting a breakpoint, OmniDB will execute the entire function to the end.



Observe the values for each variable. We can see that the value of `ret` is 11 even before the function finishes. Also note that OmniDB does not remove the breakpoint you placed. To do that, you can click in the breakpoint little icon. Now hit *Resume* again. Let us see now what happens when the function finishes.



OmniDB will go automatically to the *Statistics* tab, which shows 2 interesting features:

- *Sum of Duration per Line of Code Chart*: in the bottom, a chart represents total duration of the function distributed in the lines of code. With this chart, you can easily spot bottlenecks in your code. In our example, it was line 15, which we deliberately put a `pg_sleep(1)` call;
- *Colored lines of source code*: OmniDB colors the lines accordingly to the numbers seen in the chart. Colors vary from blue (small duration), passing through yellow (medium duration) until red (high duration), as in a *temperature* diagram.

Also note the *Total duration* message, which shows execution time of the function, without considering the time you spent analyzing it.

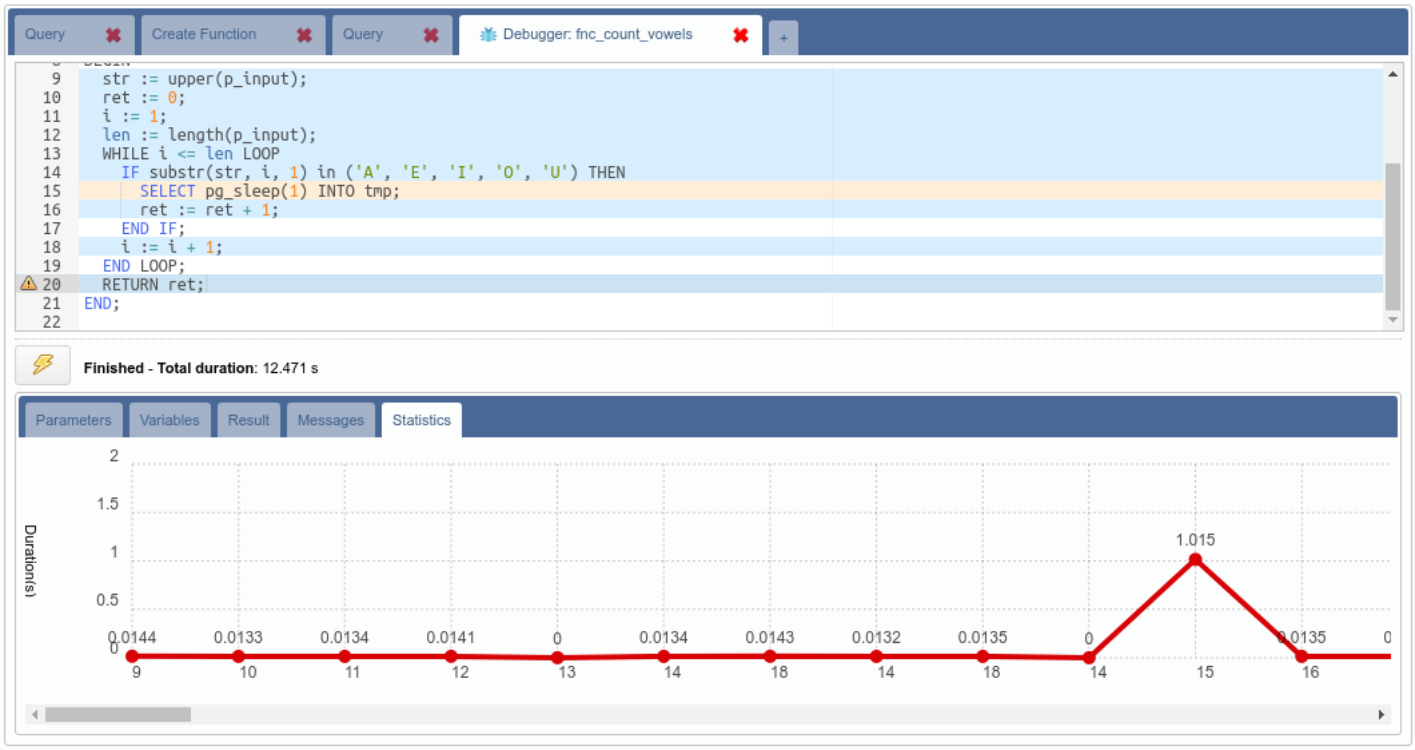
13.0.0.4 Inspecting record attribute values

An interesting feature that we do not usually see in other debuggers is the ability to inspect each attribute of a variable of type `record`. OmniDB debugger does that as it is split into different variables, allowing you to see the value and type of each attribute.

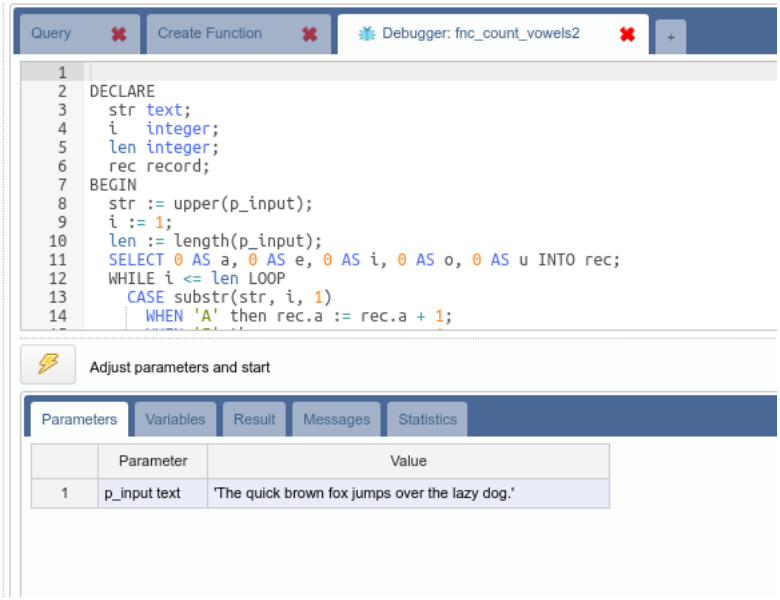
To illustrate that, let us create another function, similar to the previous one, but now called `fnc_count_vowels2`:

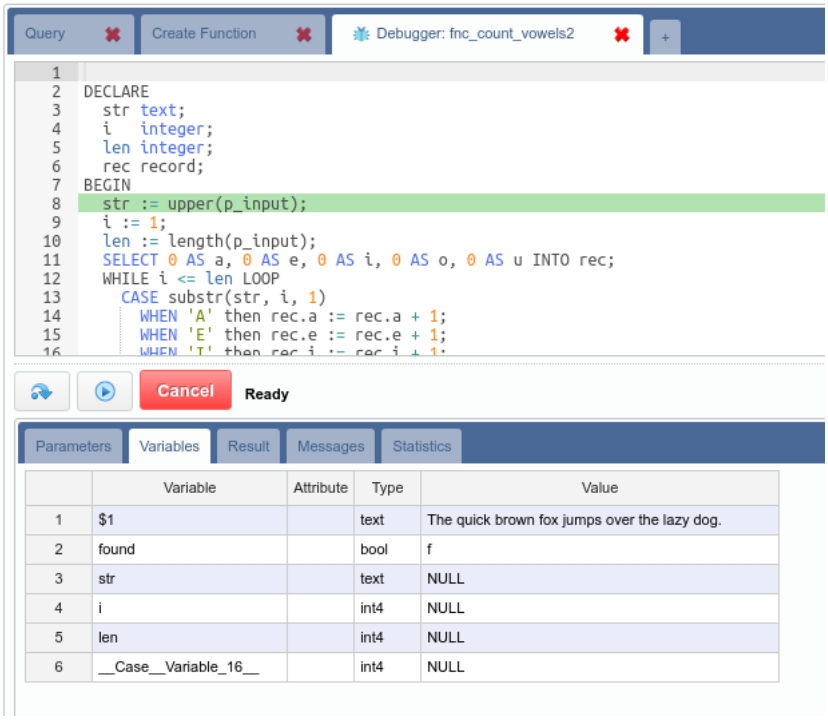
```
CREATE OR REPLACE FUNCTION public.fnc_count_vowels2 (p_input text)
RETURNS integer LANGUAGE plpgsql AS
$function$
DECLARE
  str text;
```

```
  i   integer;
  len integer;
  rec record;
BEGIN
  str := upper(p_input);
  i := 1;
  len := length(p_input);
  SELECT 0 AS a, 0 AS e, 0 AS i, 0 AS o, 0 AS u INTO rec;
  WHILE i <= len LOOP
    CASE substr(str, i, 1)
      WHEN 'A' then rec.a := rec.a + 1;
      WHEN 'E' then rec.e := rec.e + 1;
      WHEN 'I' then rec.i := rec.i + 1;
      WHEN 'O' then rec.o := rec.o + 1;
      WHEN 'U' then rec.u := rec.u + 1;
      ELSE NULL;
    END CASE;
    i := i + 1;
  END LOOP;
  RETURN rec.a + rec.e + rec.i + rec.o + rec.u;
END;
$function$
```

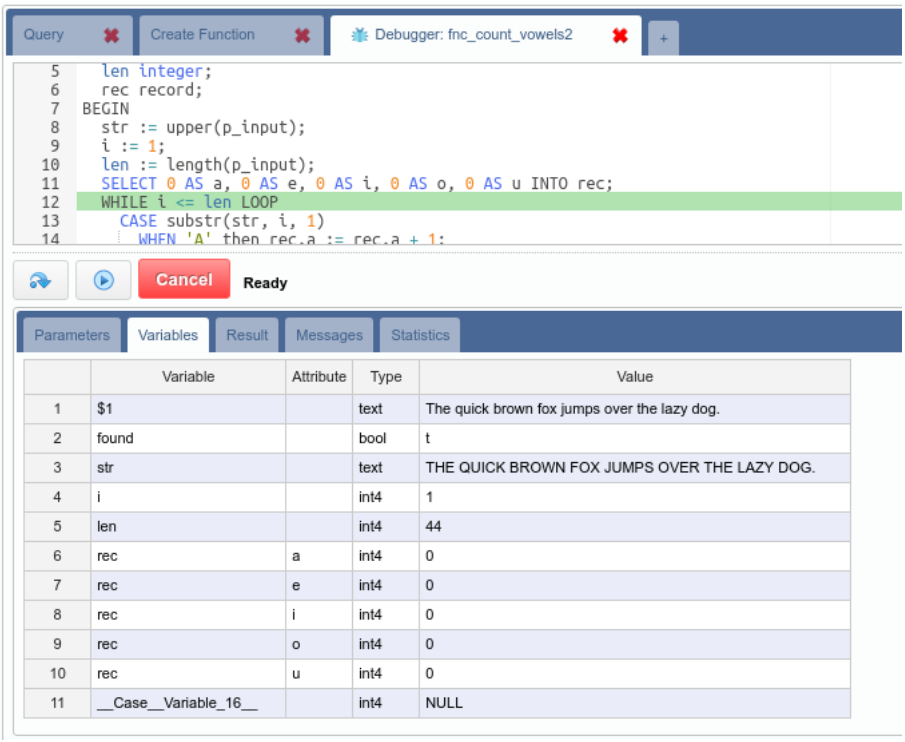


Observe how we keep track of every vowel count individually. Now let us start debugging it, using the same text as before (`'The quick brown fox jumps over the lazy dog.'`):



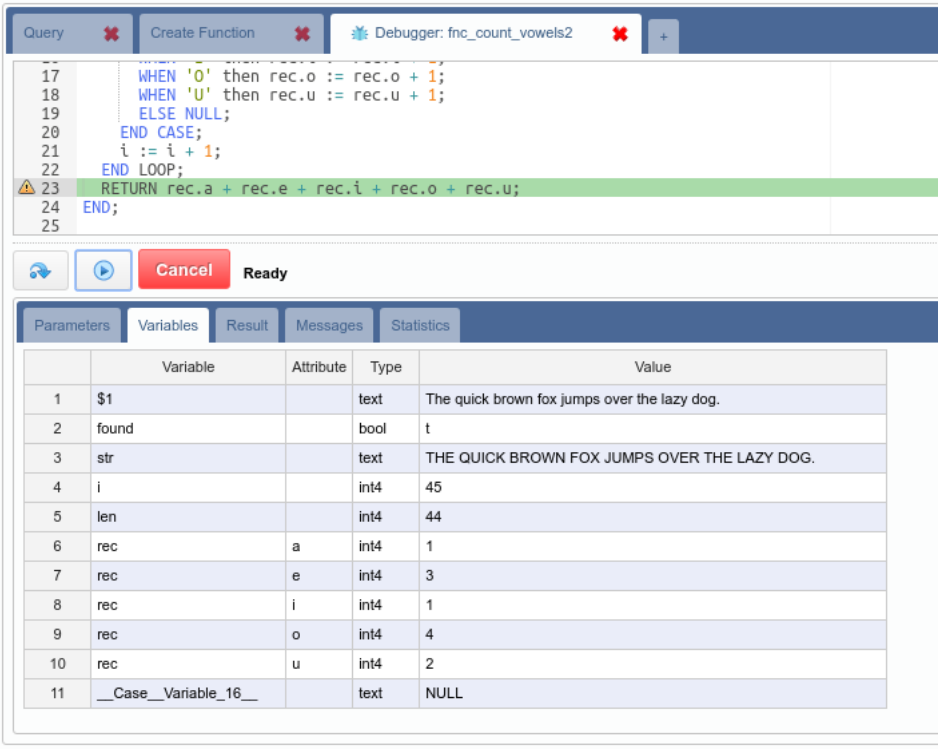


Note from the picture above that PostgreSQL created an internal *Case Variable*. Also note that the variable `rec` is not shown in the list of known variables. This is because PostgreSQL still does not know what attributes `rec` will contain. Let's step over some more steps.



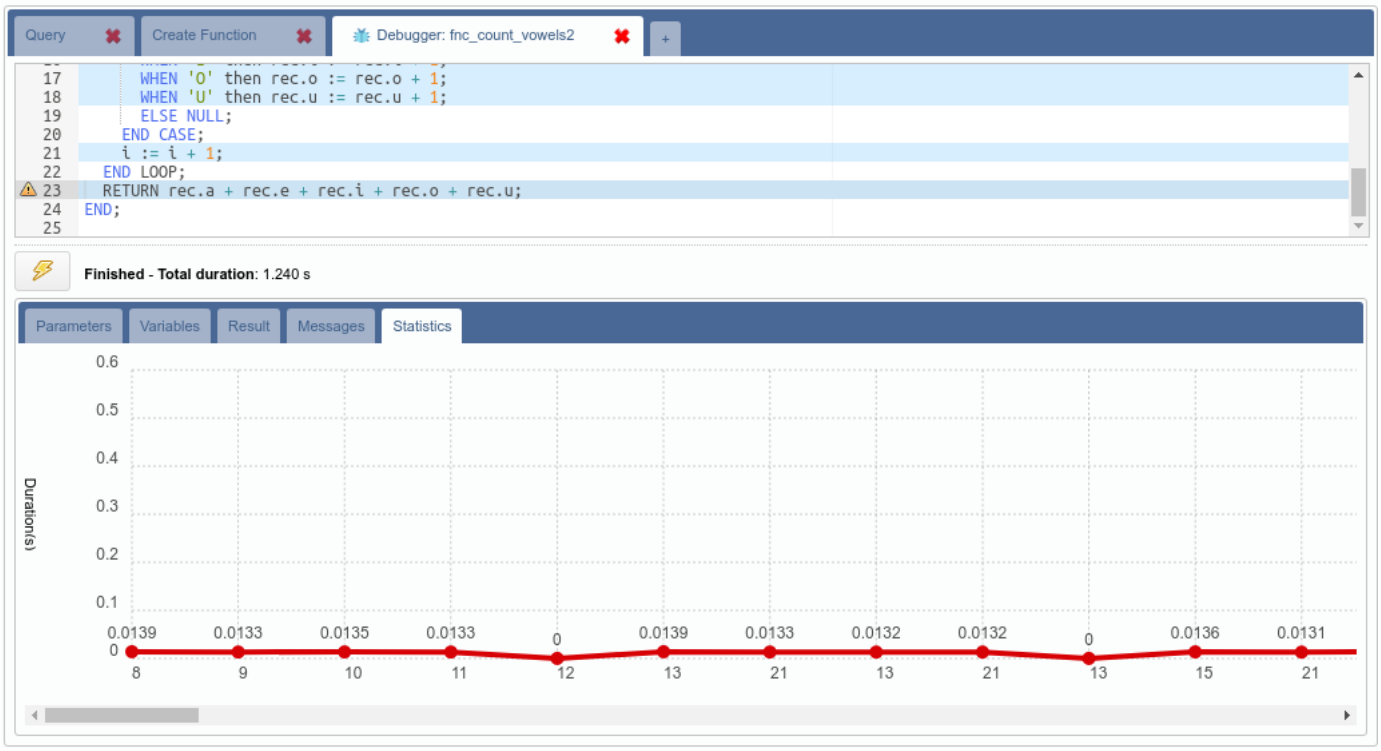
Right after the execution of line 11, `rec` variable comes to life and we can see it has 5 attributes: `a`, `e`, `i`, `o` and `u`, all of the type `int` and having initial value `0`.

Now set a breakpoint in line 23 and click the *Resume* button.



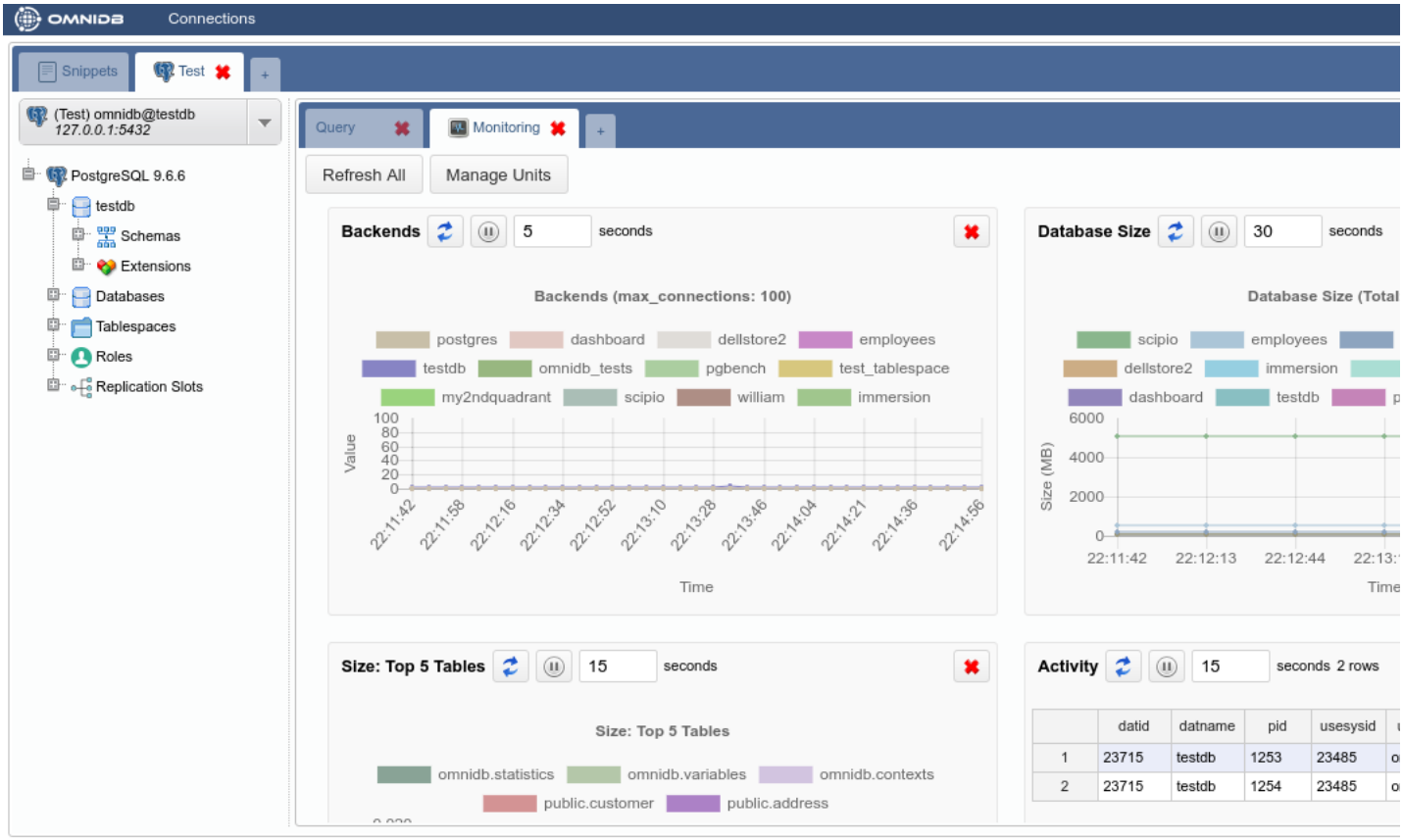
	Variable	Attribute	Type	Value
1	$1		text	The quick brown fox jumps over the lazy dog.
2	found		bool	t
3	str		text	THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
4	i		int4	45
5	len		int4	44
6	rec	a	int4	1
7	rec	e	int4	3
8	rec	i	int4	1
9	rec	o	int4	4
10	rec	u	int4	2
11	__Case__Variable_16__		text	NULL

See how we can inspect every attribute, observing how many of each vowel the text contain. Now let's finish this function.

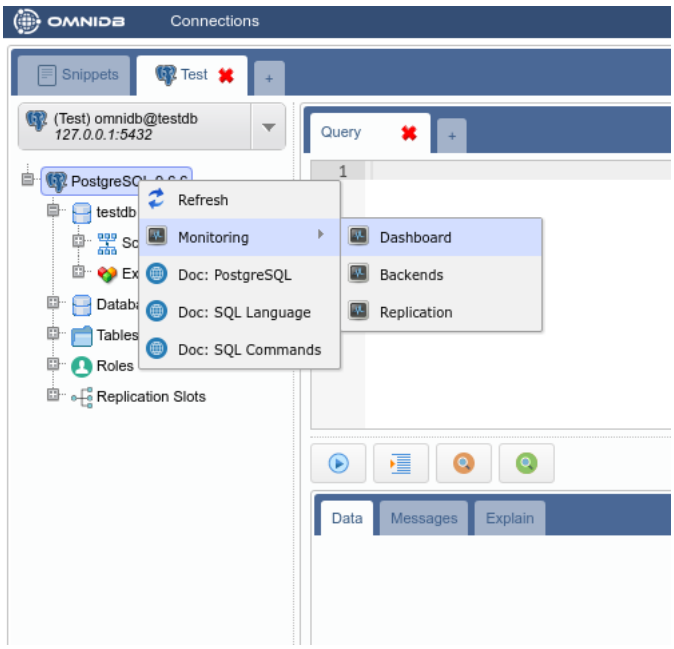


14 Monitoring Dashboard

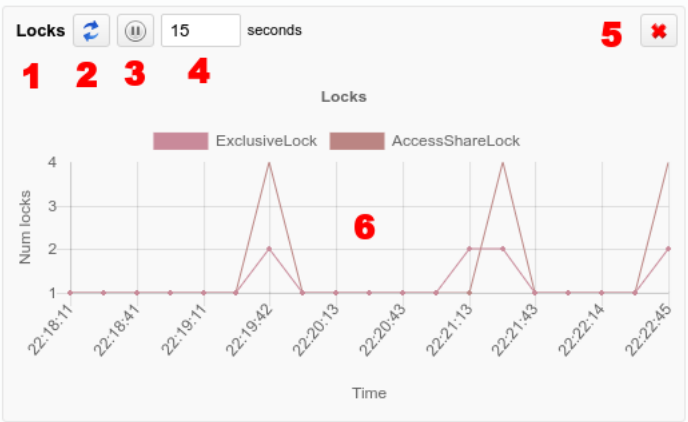
OmniDB 2.4.0 introduces a new cool feature called *Monitoring Dashboard*. We know a picture is worth a thousand words, so please take a look:



As you can see, this is a new kind of inner tab showing some charts and grids. This *Monitoring* inner tab is automatically opened once you expand the tree root node (the *PostgreSQL* node). You can keep it open or close it at any time. To open it again, right-click the root node and click on *Dashboard*.



The dashboard is composed of handy information rectangles called *Monitoring Units*. Here is an example of Monitoring Unit and its interface elements:

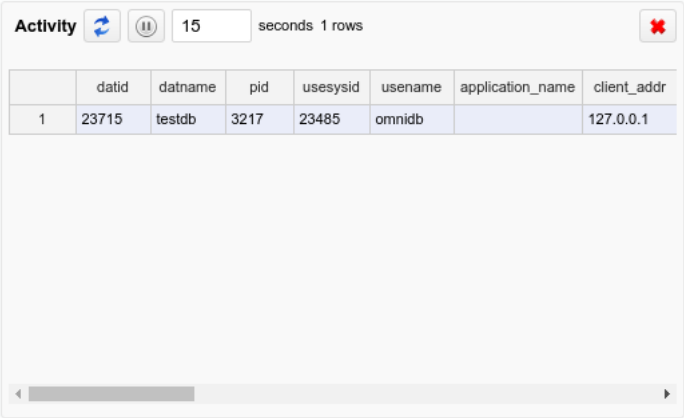


- **1**: Title of the Monitoring Unit;
- **2**: Refresh the Monitoring Unit. Depending on the type, clicking on this button will refresh the entire drawing area or just make the chart acquire a new set of values;
- **3**: Pause the Monitoring Unit;
- **4**: Interval in seconds for automatic refreshing;
- **5**: Remove the Monitoring Unit of the Monitoring Dashboard;
- **6**: Drawing area, that will be different depending on the type of the Monitoring Unit.

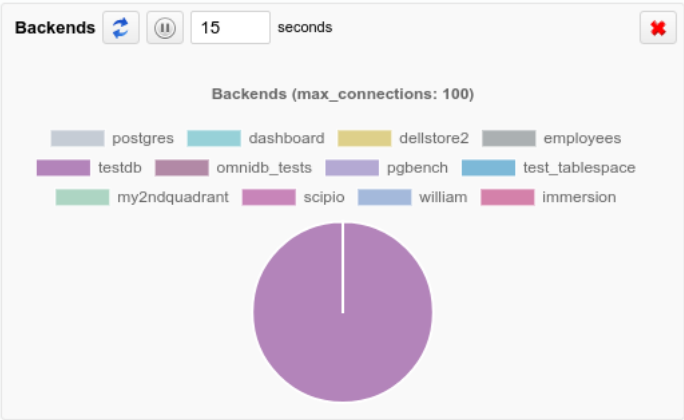
14.0.0.1 Types of Monitoring Units

Currently there are 3 types of Monitoring Units:

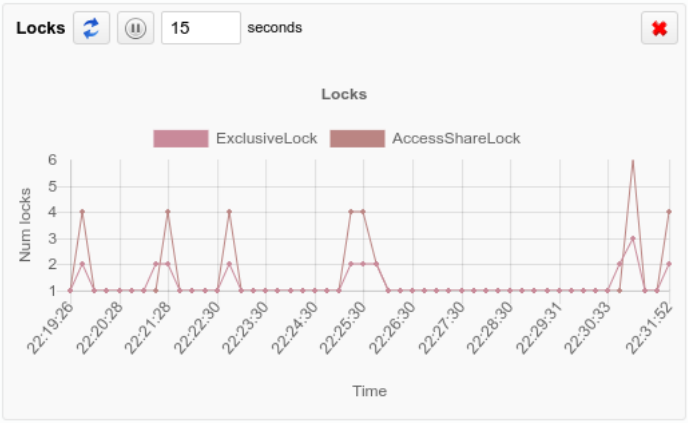
- **Grid**: The most simple kind, just executes a query from time to time and shows the results in a data grid.



- **Chart**: Every time it refreshes, it renders a new complete chart. The old set of values is lost. This is most useful for pie charts, but other kind of charts can be used too.

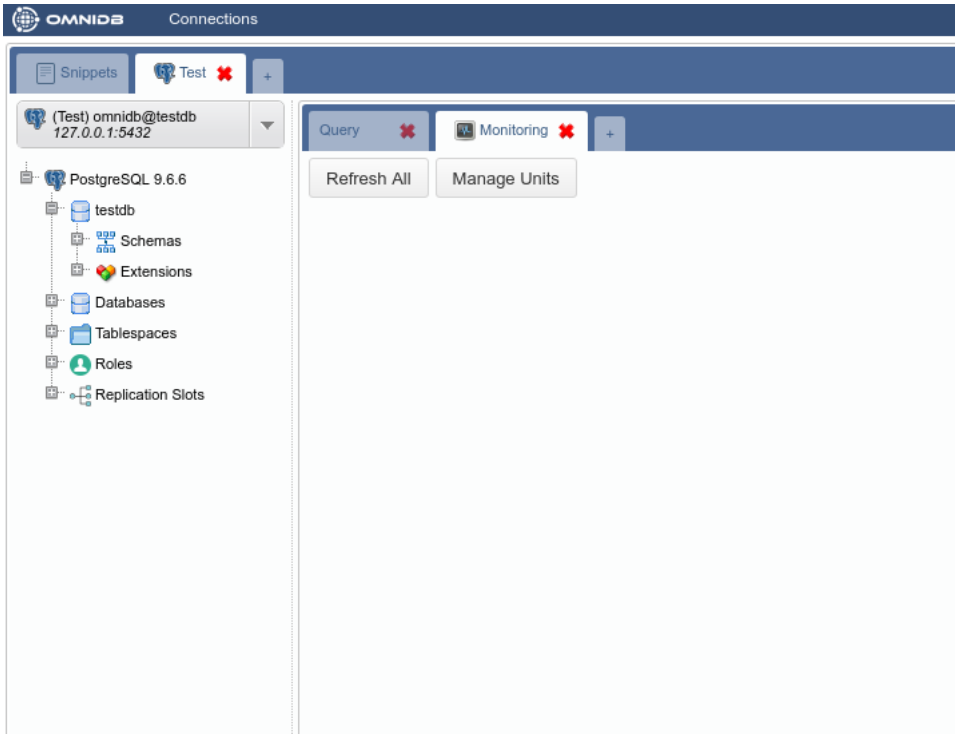


- **Chart-Append**: Perhaps this is the most useful kind of Monitoring Unit. It is a chart that appends a new set of values every time it refreshes. Line or bar charts fit best for this type. The last 50 set of values are kept by the component client-side to be viewed by the user.

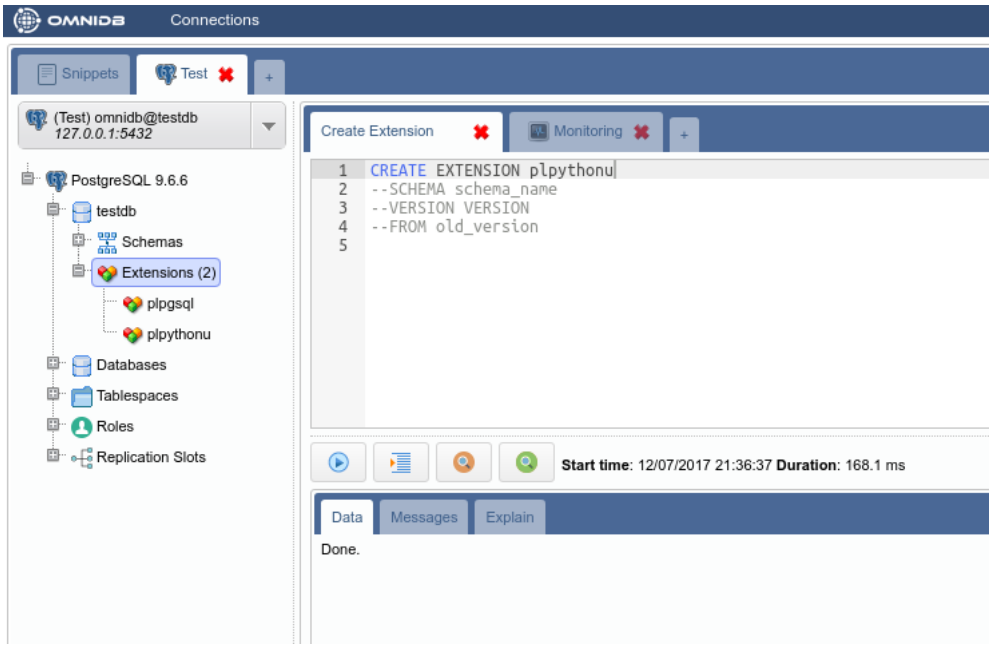


14.0.0.2 Showing and hiding units in the dashboard

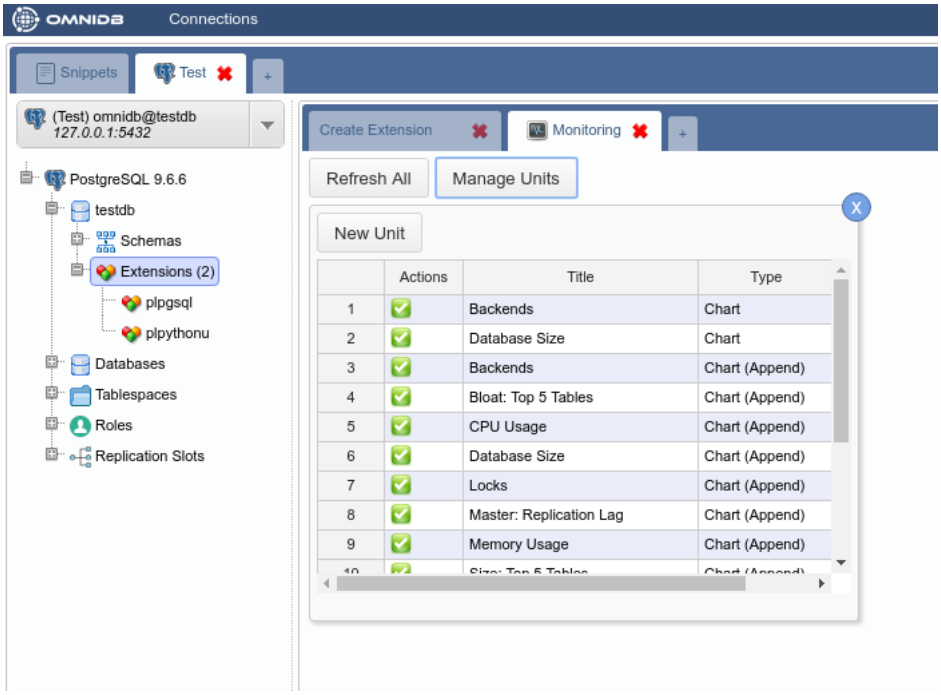
If you click in the button *Refresh All*, then all Monitoring Units will be refreshed at once. You can also remove undesired Monitoring Units by clicking in the *Remove* button. Let us go ahead and remove all units from the dashboard, making it empty:



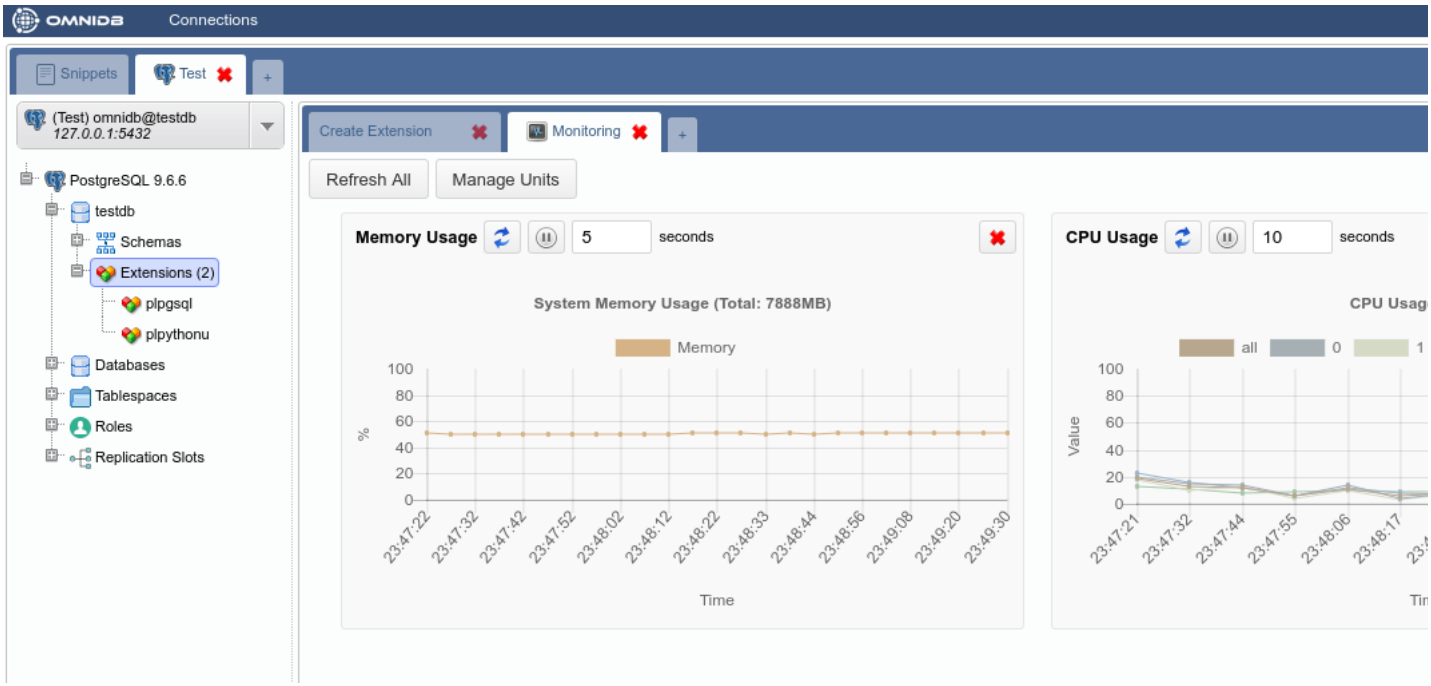
All Monitoring Units that come with OmniDB are open source and available in this repository (feel free to contribute). But be aware that some Monitoring Units require the `plpythonu` script to be installed in the database. Please refer to the instructions specific to your operating system on how to install `plpythonu` if you desire to use and create Monitoring Units that use `plpythonu`.



Now that our dashboard is empty, let us add some units. Click on the *Manage Units* button.



Click on the green action to add the Monitoring Units called *CPU Usage* and *Memory Usage*. Bear in mind that both units require `plpythonu` extension in the database. *CPU Usage* also requires that the tool `mpstat` should be installed in the server. Also both units are of type *Chart-Append*. Wait for some seconds and you will have a dashboard like this:

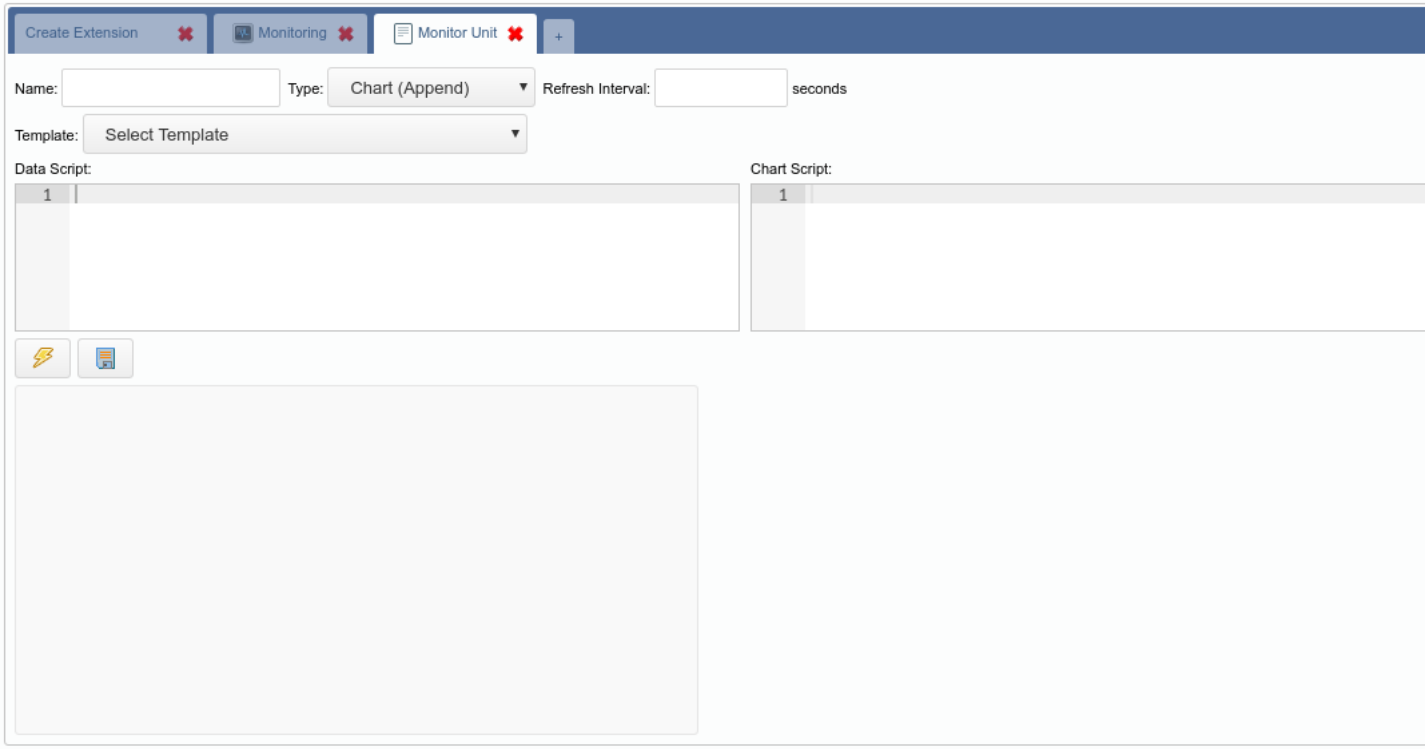


In a similar way, you can add and remove any unit you want to customize the dashboard the way you want.

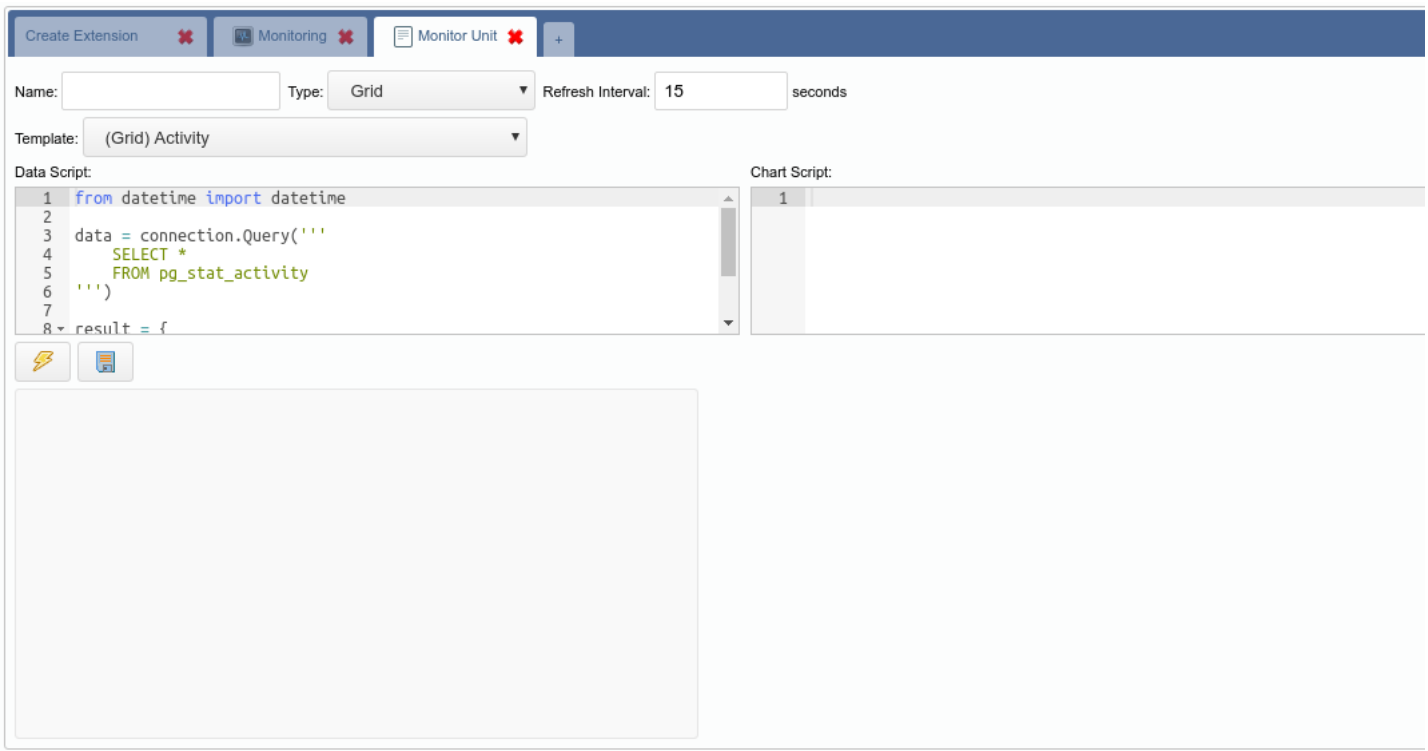
14.0.0.3 Writing custom Monitoring Units: Grid

OmniDB provides you the power to write your own units and customize existing ones. Everything is done through Python scripts that run inside a sandbox. Beware that units powered by `plpythonu` can have access to the file system the database user also has access to, and any Monitoring Unit have the same permission as the database user you configured in the Connection.

To create a new Monitoring Unit, click on the *Manage Units* button in the dashboard, then click on the *New Unit* button. It will open a new kind of inner tab like this:



The easiest way to write a custom unit is to use an existing one as template. Go ahead and select the *(Grid) Activity* template:



Note how OmniDB fills the *Data Script* source code. This script is responsible for generating data for the unit every time it refreshes. As a grid unit is nothing else but a grid of data, we can rely on only this script for now.

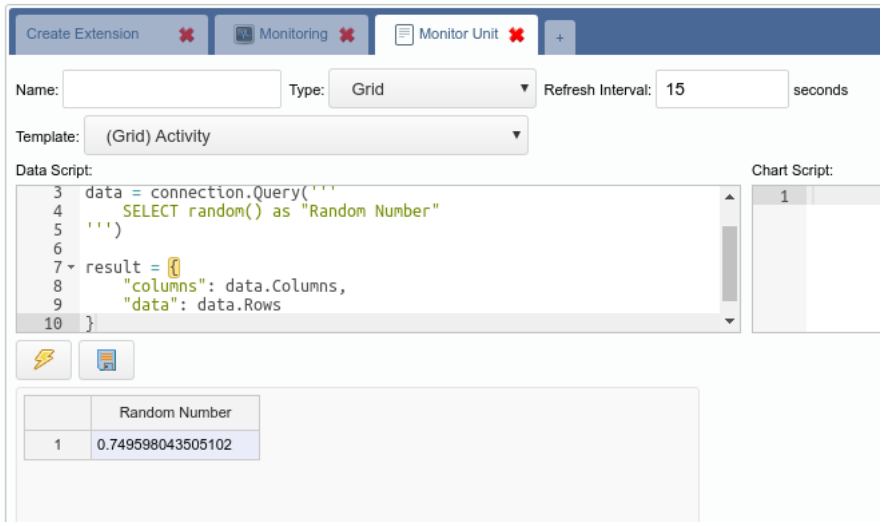
Now let us take a look at the source code of this template:

```
from datetime import datetime

data = connection.Query('''
    SELECT *
    FROM pg_stat_activity
''')

result = {
    "columns": data.Columns,
    "data": data.Rows
}
```

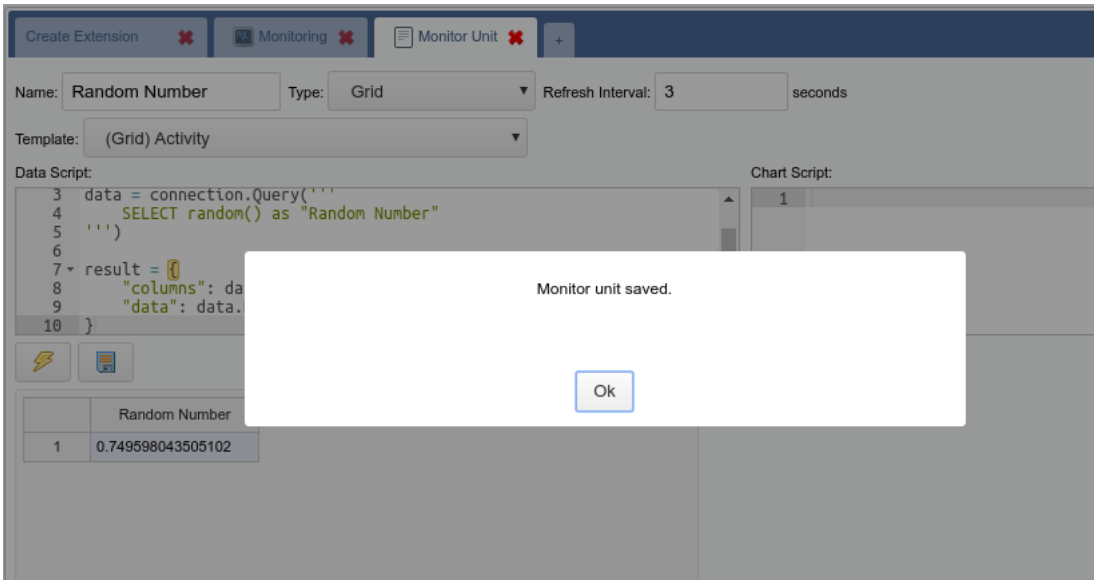
It is simple enough. It executes an SQL query into the current connection using the reserved `connection` variable. Also, the grid unit type expects its results in a JSON variable that must be called `result` and must have the attributes `"columns"` (an array of column names) and `"data"` (an array of rows, each row being an array of values). The `connection.Query()` function already does the job pretty well, so let us just change the SQL query this way:

```
from datetime import datetime

data = connection.Query('''
    SELECT random() as "Random Number"
''')

result = {
    "columns": data.Columns,
    "data": data.Rows
}
```

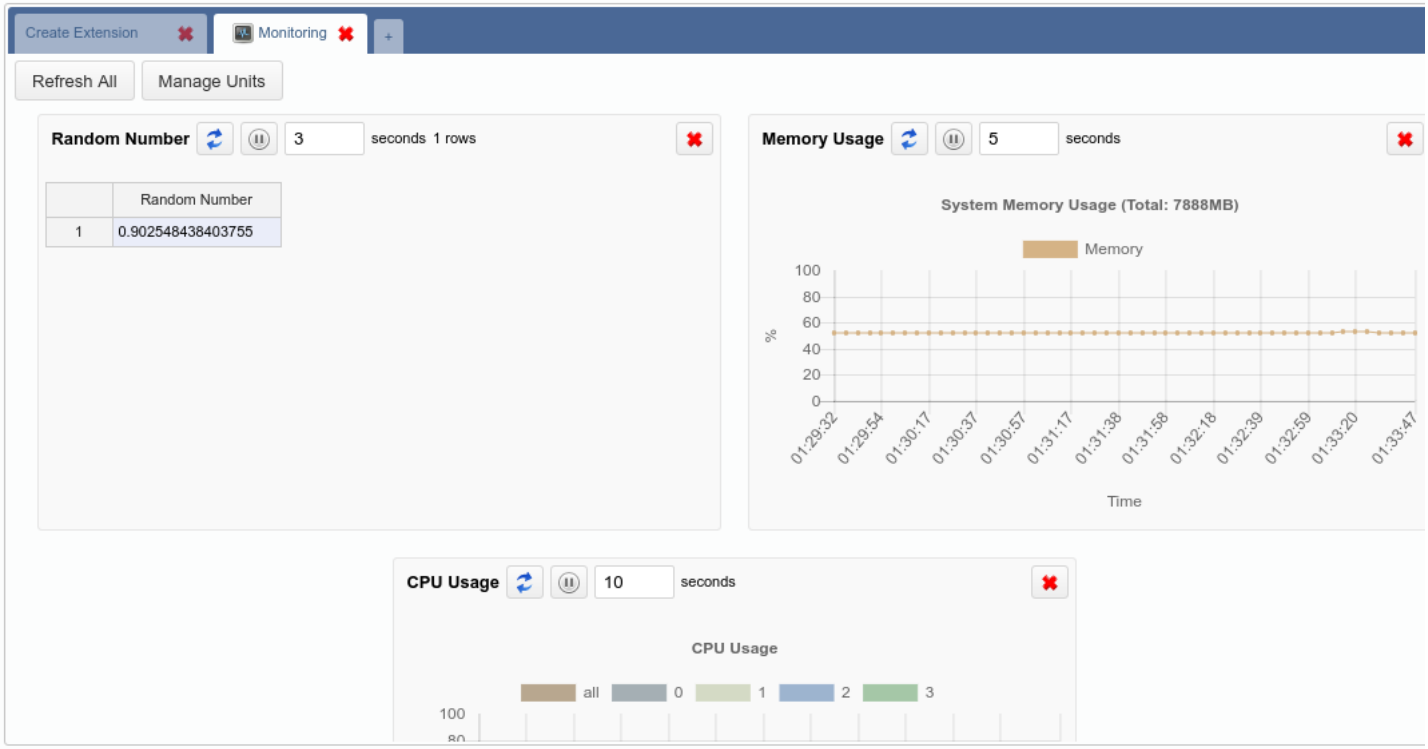
Copy and paste the above Python code into the *Data Script* text field and then click on the *Test* (lighting) button:



Note how the grid was rendered in the preview drawing area. You can click the *Test* button as many times as you want. Now we will give the unit a name, set a refresh interval and then hit the *Save* button:

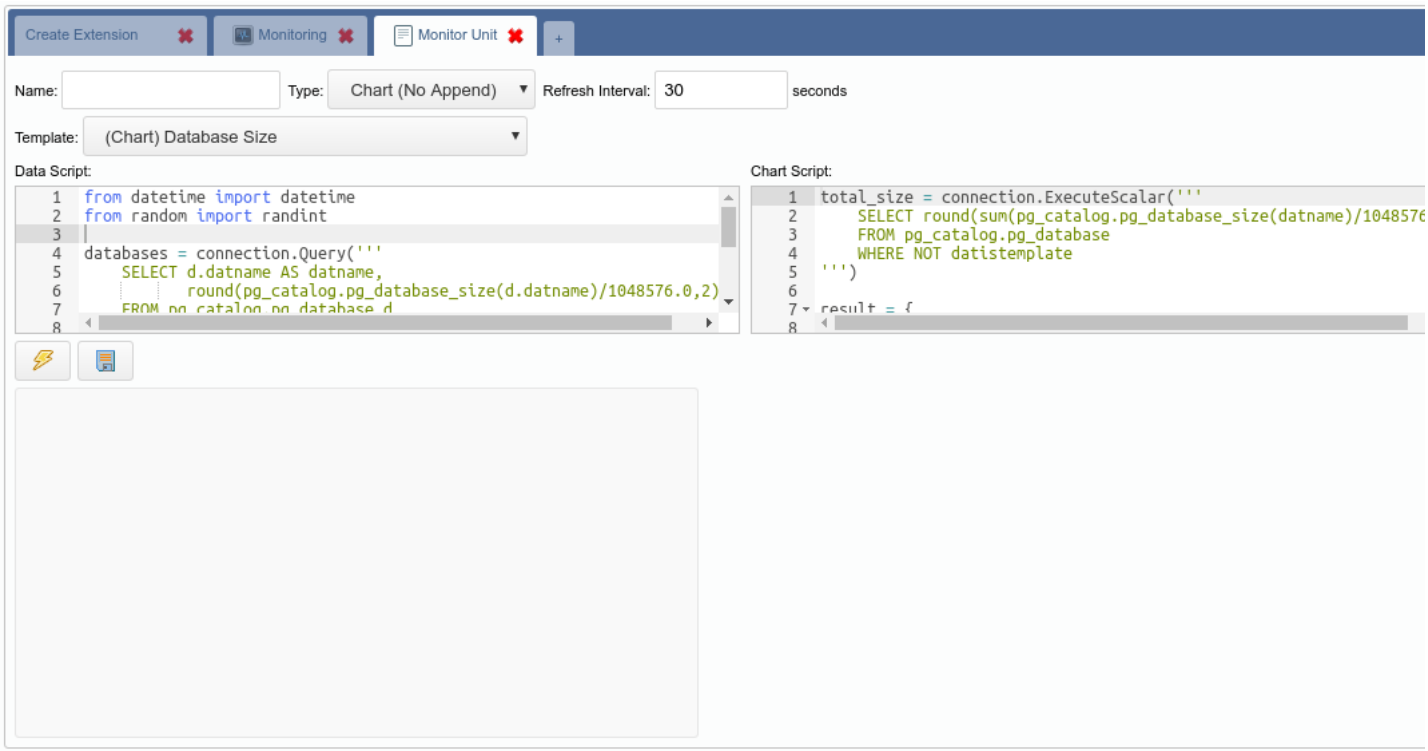


Click the *OK* button and then close the edit tab. Our new Monitoring Unit will be in the list of available units. As we created this unit, we can either add it to the dashboard, edit it or remove it. Let us add it to the dashboard (green action):



14.0.0.4 Writing custom Monitoring Units: Chart

Click in the *Manage Units* button and then in the *New Unit* button. This time we will create a Chart Monitoring Unit. So choose *(Chart) Database Size* as a template.



The source code of this kind of unit is more complex. There are two scripts:

- **Data Script**: Executed every time the unit is refreshed;
- **Chart Script**: Executed only at the beginning to build the chart.

The chart units are based in the component Chart.js and each chart type contains a specific JSON structure. The best approach to build new chart units is to start from a template and also check the Chart.js docs to see every property that can be added to make the output even better for each situation.

Let us take a look at the **Data Script**:

```
from datetime import datetime
from random import randint

databases = connection.Query('''
    SELECT d.datname AS datname,
           round(pg_catalog.pg_database_size(d.datname)/1048576.0,2) AS size
    FROM pg_catalog.pg_database d
    WHERE d.datname not in ('template0','template1')
''')
```

```
data = []
color = []
label = []

for db in databases.Rows:
    data.append(db["size"])
    color.append("rgb(" + str(randint(125, 225)) + "," + str(randint(125, 225)) + "," + str(randint(125, 225)) + ")")
    label.append(db["datname"])

result = {
    "labels": label,
    "datasets": [
        {
            "data": data,
            "backgroundColor": color,
            "label": "Dataset 1"
        }
    ]
}
```

Here we can see that the reserved variable `connection` is still being used to retrieve data from the database. Bear in mind that this variable is always pointing to the current Connection.

This template is for a Pie chart, which contains only one dataset and three arrays for the data:

- `data`: One value per slice;
- `color`: One color per slice;
- `label`: One label per slice.

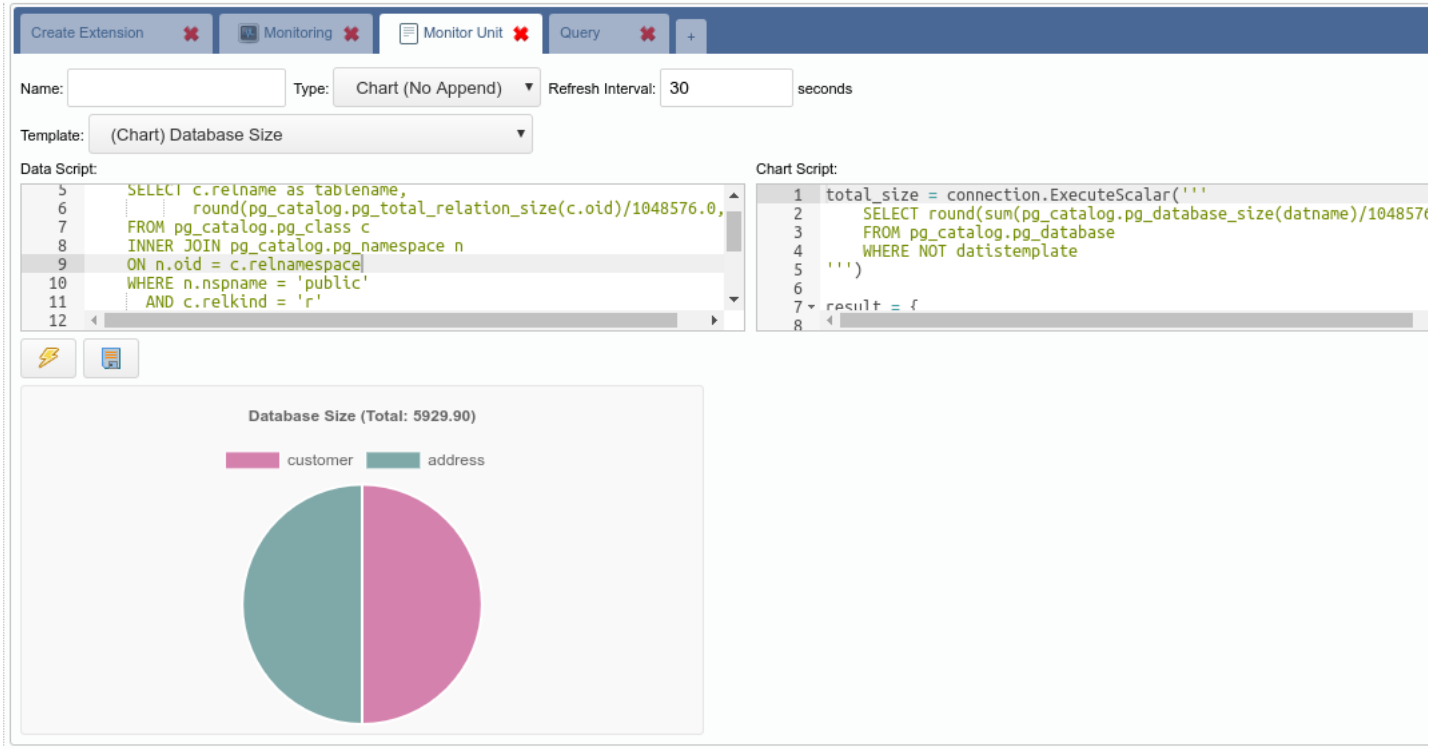
This way, `data[0]`, `color[0]` and `label[0]` refer to the first slice, while `data[1]`, `color[1]` and `label[1]` refer to the second slice, and so on.

This script must return a variable called `result` and also needs to be a JSON like in the above script.

So right now you are probably guessing that you just need to change the SQL query to make the chart behave different. Well, in terms of data and datasets, you guessed right. So let's change the SQL query of this chart to compare sizes of tables of schema `public`. Also change the references from `datname` to `tablename`, as we have changed the column name.

```
from datetime import datetime
from random import randint

databases = connection.Query('''
    SELECT c.relname as tablename,
           round(pg_catalog.pg_total_relation_size(c.oid)/1048576.0,2) AS size
    FROM pg_catalog.pg_class c
    INNER JOIN pg_catalog.pg_namespace n
    ON n.oid = c.relnamespace
    WHERE n.nspname = 'public'
      AND c.relkind = 'r'
''')

data = []
color = []
label = []

for db in databases.Rows:
    data.append(db["size"])
    color.append("rgb(" + str(randint(125, 225)) + "," + str(randint(125, 225)) + "," + str(randint(125, 225)) + ")")
    label.append(db["tablename"])

result = {
    "labels": label,
    "datasets": [
        {
            "data": data,
            "backgroundColor": color,
            "label": "Dataset 1"
        }
    ]
}
```

Copy and paste the above script into the *Data Script* field and then hit the *Test* button:



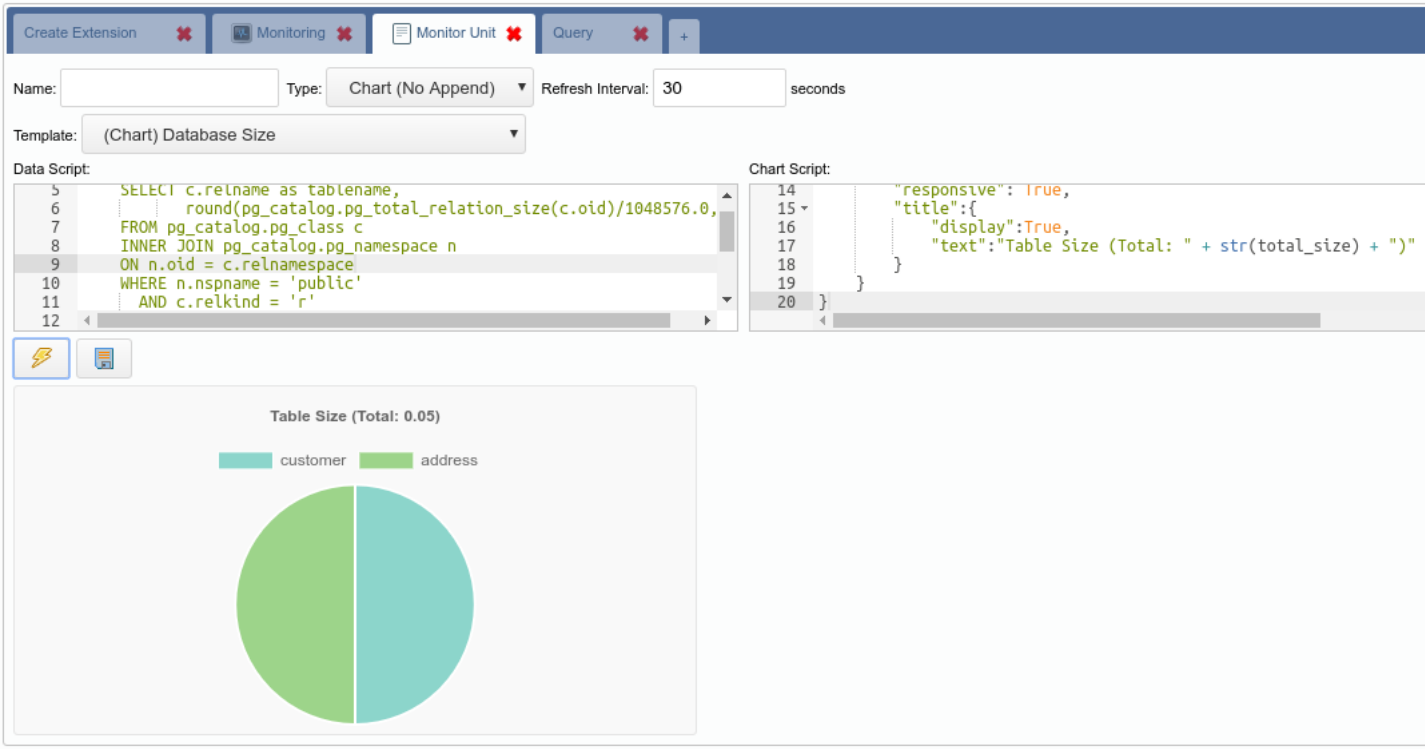
Apparently the chart is almost done. We need to fix the title, it still says *Database Size*, when this chart is about table size. Any information about the format of the chart itself is defined in the *Chart Script* text field. Let us understand the current source code:

```
total_size = connection.ExecuteScalar('''
    SELECT round(sum(pg_catalog.pg_database_size(datname)/1048576.0),2)
    FROM pg_catalog.pg_database
    WHERE NOT datistemplate
''')

result = {
    "type": "pie",
    "data": None,
    "options": {
        "responsive": True,
        "title":{
            "display":True,
            "text":"Database Size (Total: " + str(total_size) + ")"
        }
    }
}
```

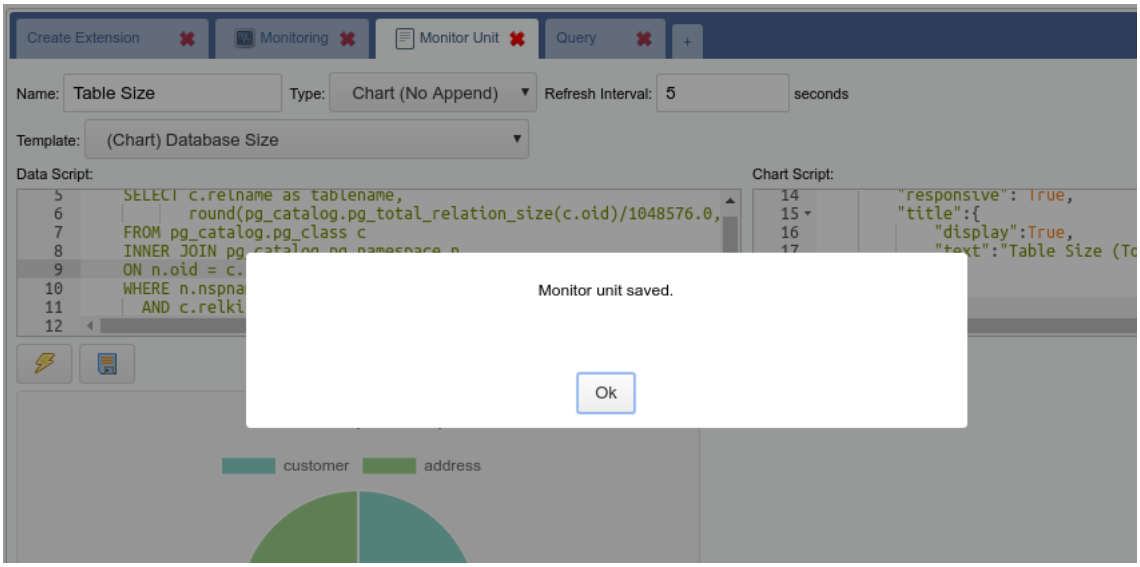
Easy enough. We can make use of the reserved variable `connection` to retrieve data in the *Chart Script* too. This is mainly used to put information in the chart title. The variable `result` must be defined here. Note how its JSON value defines a pie chart and the title. So we just need to change the query and adjust the title, this way:

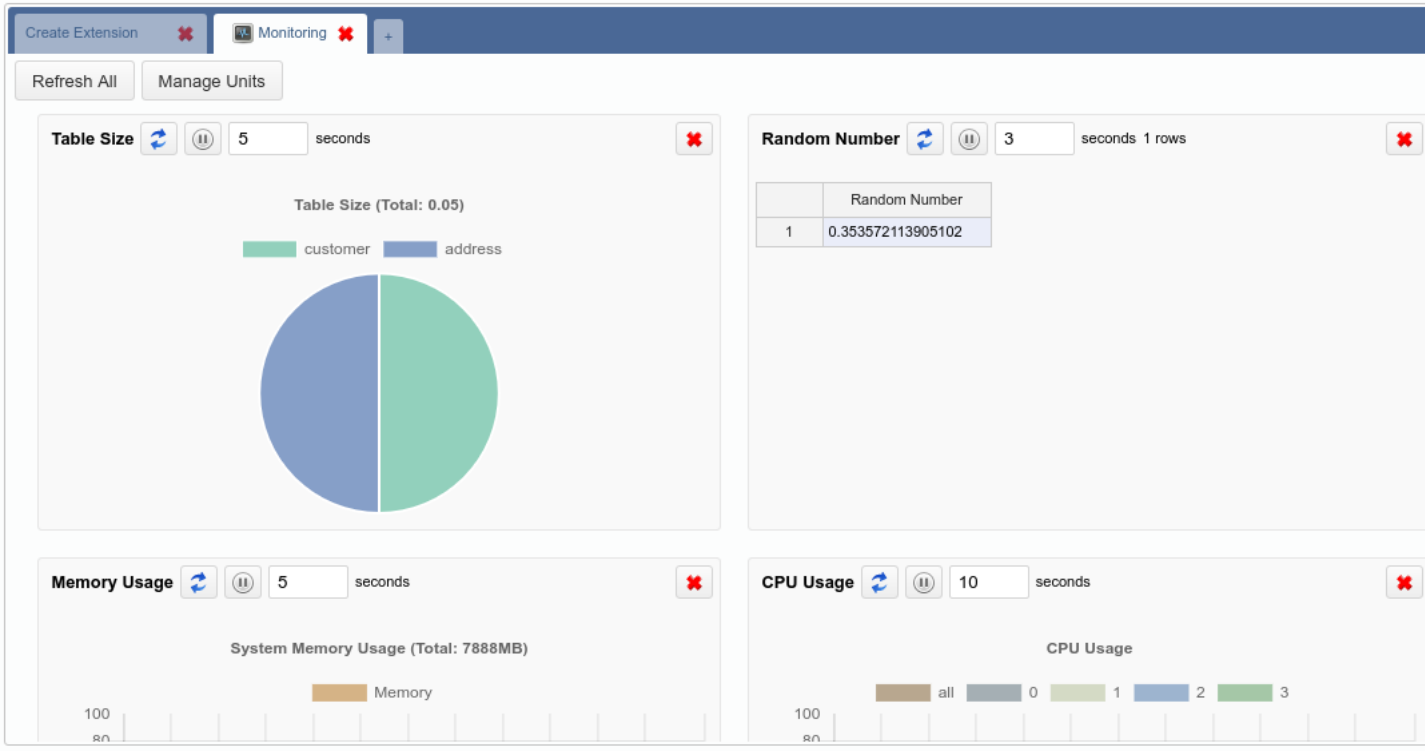
```
total_size = connection.ExecuteScalar('''
    SELECT round(sum(pg_catalog.pg_total_relation_size(c.oid)/1048576.0),2) AS size
    FROM pg_catalog.pg_class c
    INNER JOIN pg_catalog.pg_namespace n
    ON n.oid = c.relnamespace
    WHERE n.nspname = 'public'
      AND c.relkind = 'r'
''')

result = {
    "type": "pie",
    "data": None,
    "options": {
        "responsive": True,
        "title":{
            "display":True,
            "text":"Table Size (Total: " + str(total_size) + ")"
        }
    }
}
```

Copy and paste the above Python code into the *Chart Script*. Then click in the *Test* button:



Now that the chart finally works the way we want, we can give it a title, adjust the refresh interval and then click in the *Save* button. After that we can add it to the dashboard.

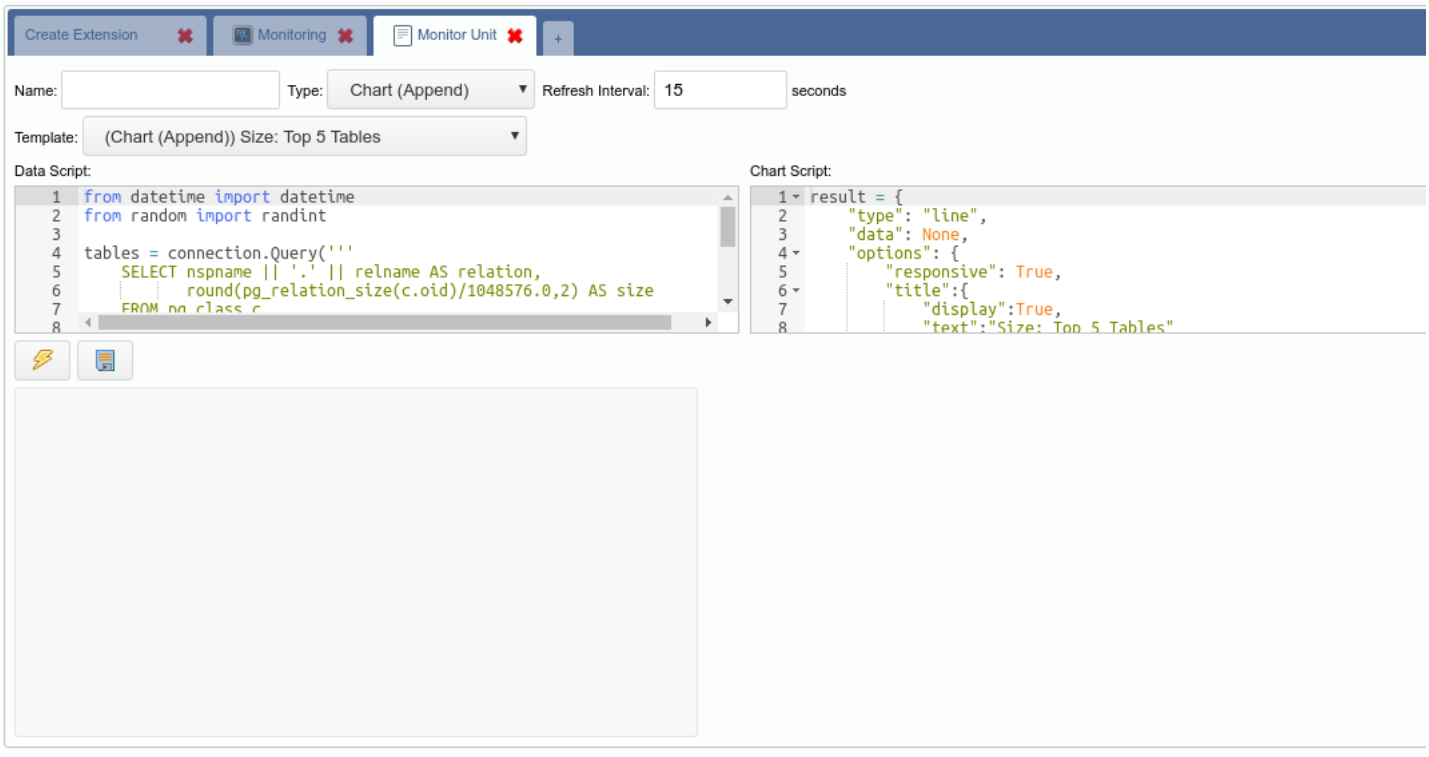




14.0.0.5 Writing custom Monitoring Units: Chart-Append

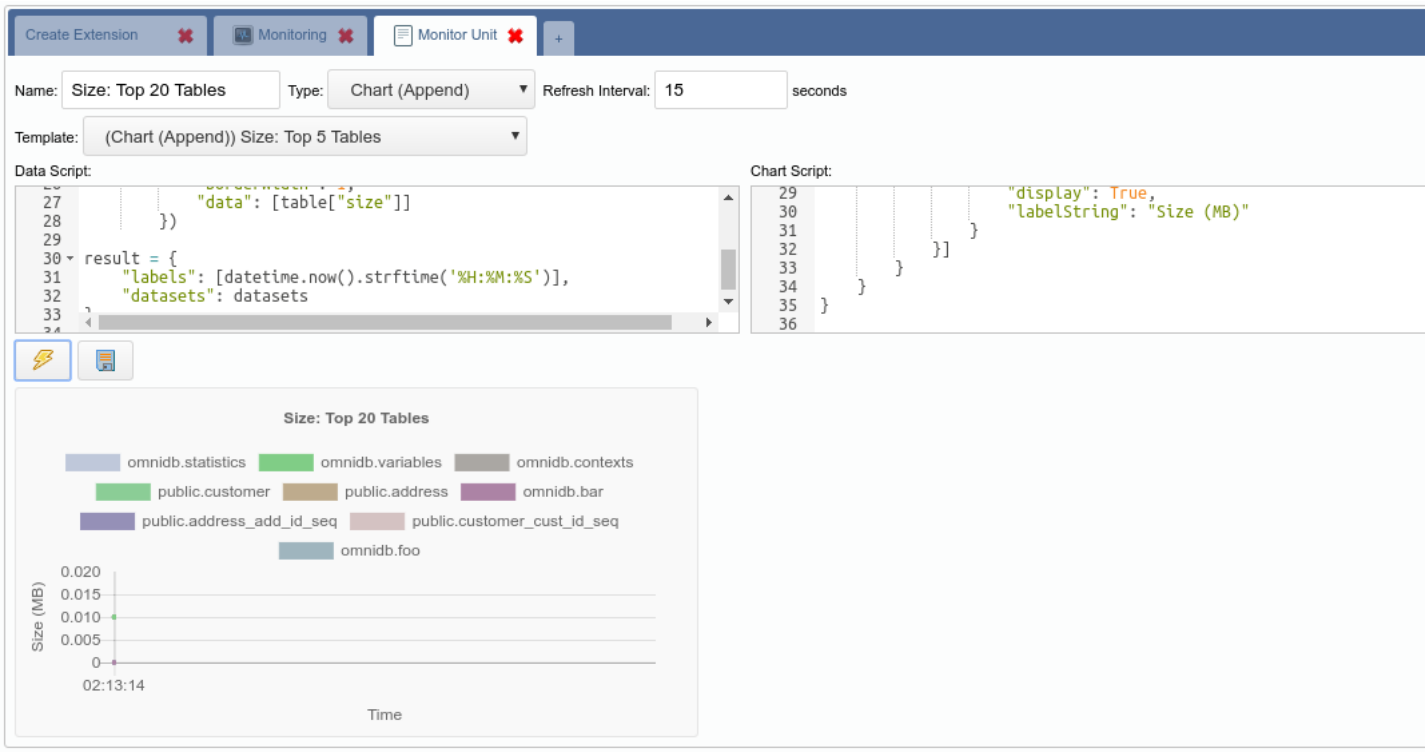
Now for the last, but most interesting kind of Monitoring Unit: *Chart-Append*. It is interesting because there is a wide range of applications for these units, since they keep recent historic data that allows us to see a comparison of values.

Go ahead and add a new chart using *(Chart (Append)) Size: Top 5 Tables* as template:

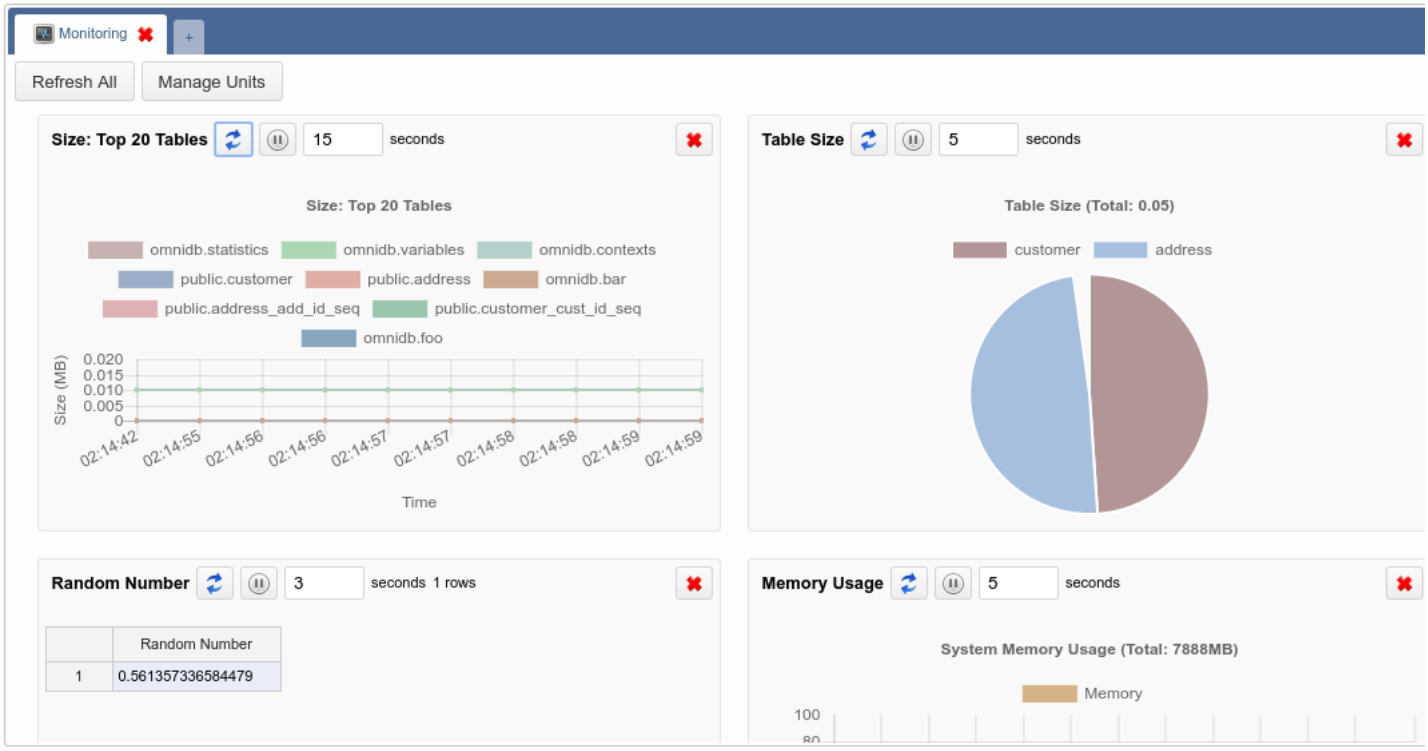


Now take a look at the source code of both *Data Script* and *Chart Script*. It is not too different from the Chart units. The dataset creation is a bit more complex as it involves other JSON settings, but that's all.

As an exercise, based on this chart, create another one called *Size: Top 20 Tables*. It should look like this:



Now save it and add it to your dashboard:



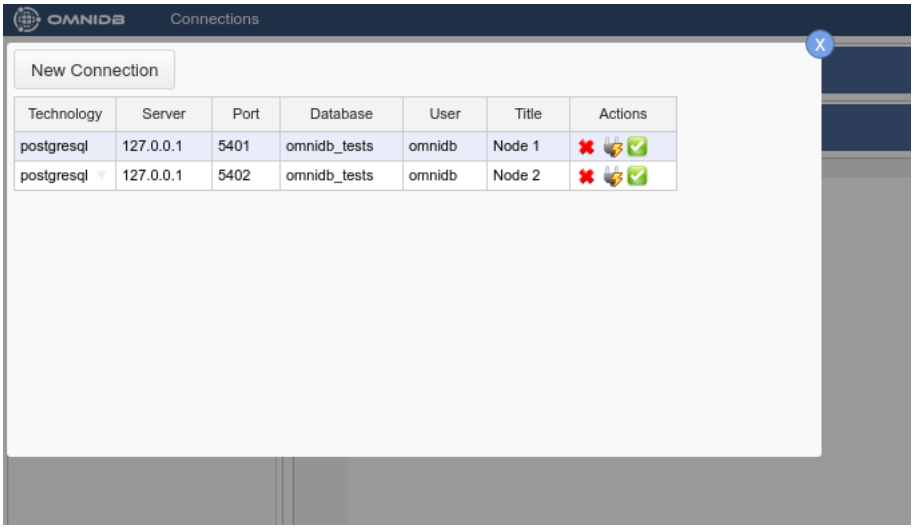
15 Logical Replication

PostgreSQL 10 introduces native logical replication, which uses a publish/subscribe model and so we can create publications on the upstream (or publisher) and subscriptions on downstream (or subscriber). For more details about it, please refer to the PostgreSQL documentation.

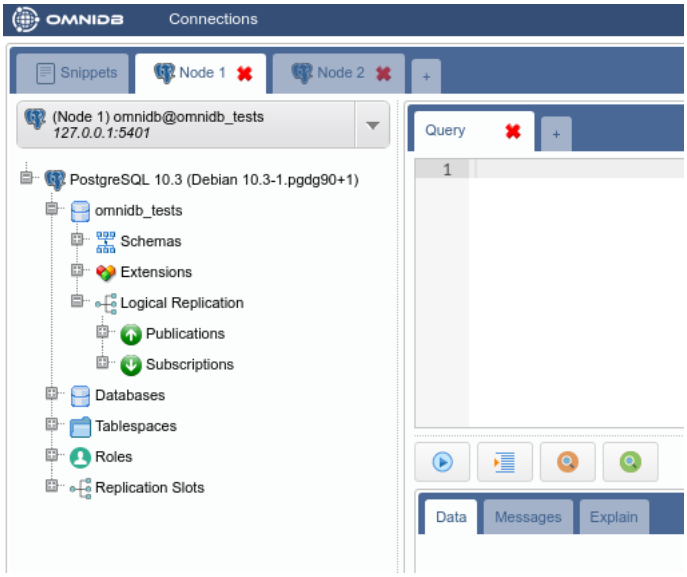
In this chapter, we will use a 2-node cluster to demonstrate PostgreSQL 10 native logical replication. Note that on each PostgreSQL instance, you need to configure `wal_level = logical` and also make sure to adjust file `pg_hba.conf` to grant access to `replication` between the 2 nodes.

15.0.0.1 Connecting to both nodes

Let's use OmniDB to connect to both PostgreSQL nodes. First of all, fill out connection info in the connection grid:



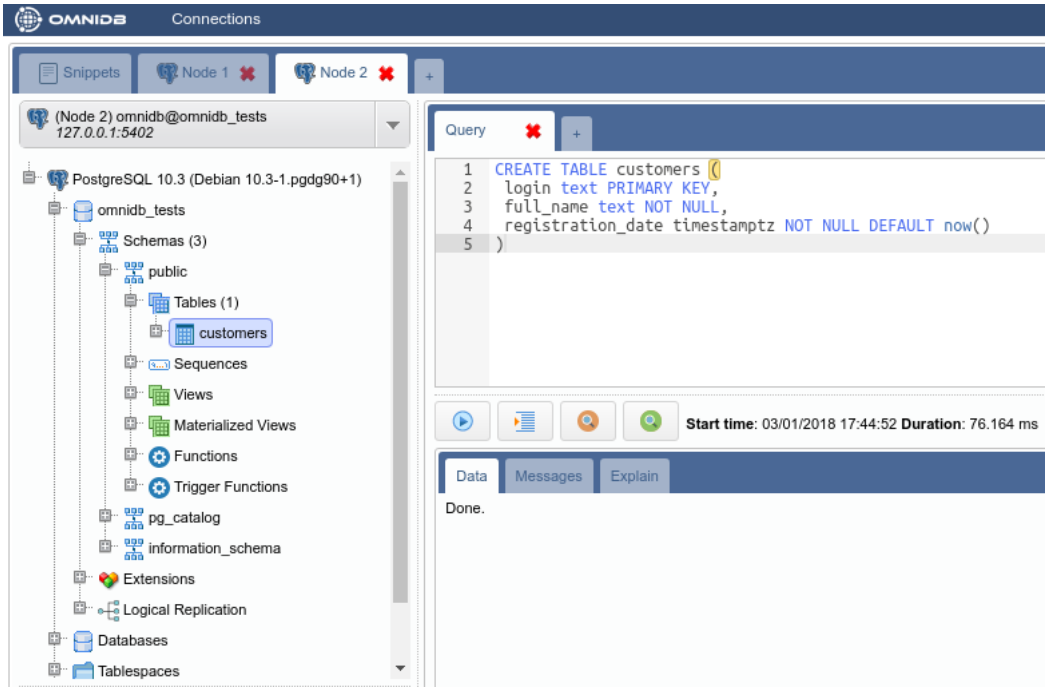
Then select both connections. Note how OmniDB understands it is connected to PostgreSQL 10 and enables a new node in the current connection tree view: it is called *Logical Replication*. Inside of it, we can see *Publications* and *Subscriptions*.



15.0.0.2 Creating a test table on both nodes

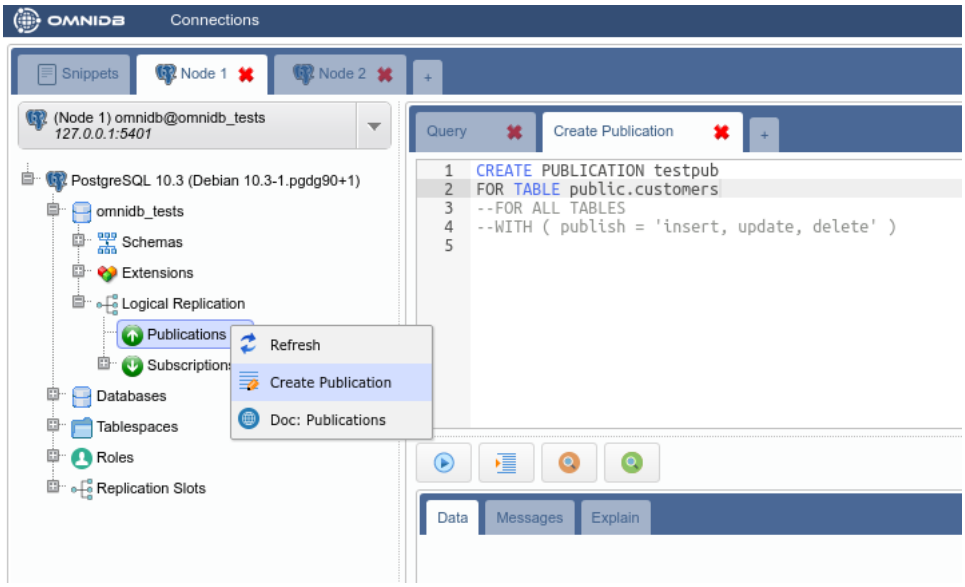
On both nodes, create a table like this:

```
CREATE TABLE customers (
 login text PRIMARY KEY,
 full_name text NOT NULL,
 registration_date timestamptz NOT NULL DEFAULT now()
)
```

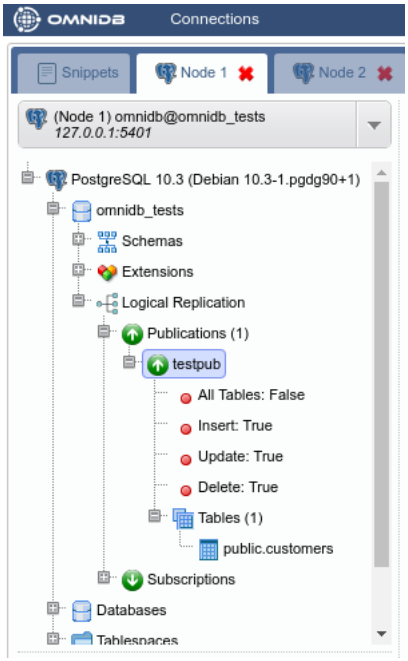


15.0.0.3 Create a publication on the first machine

Inside the connection node, expand the *Logical Replication* node, then right click in the *Publications* node, and choose the action *Create Publication*. OmniDB will open a SQL template tab with the `CREATE PUBLICATION` command ready for you to make some adjustments and run:

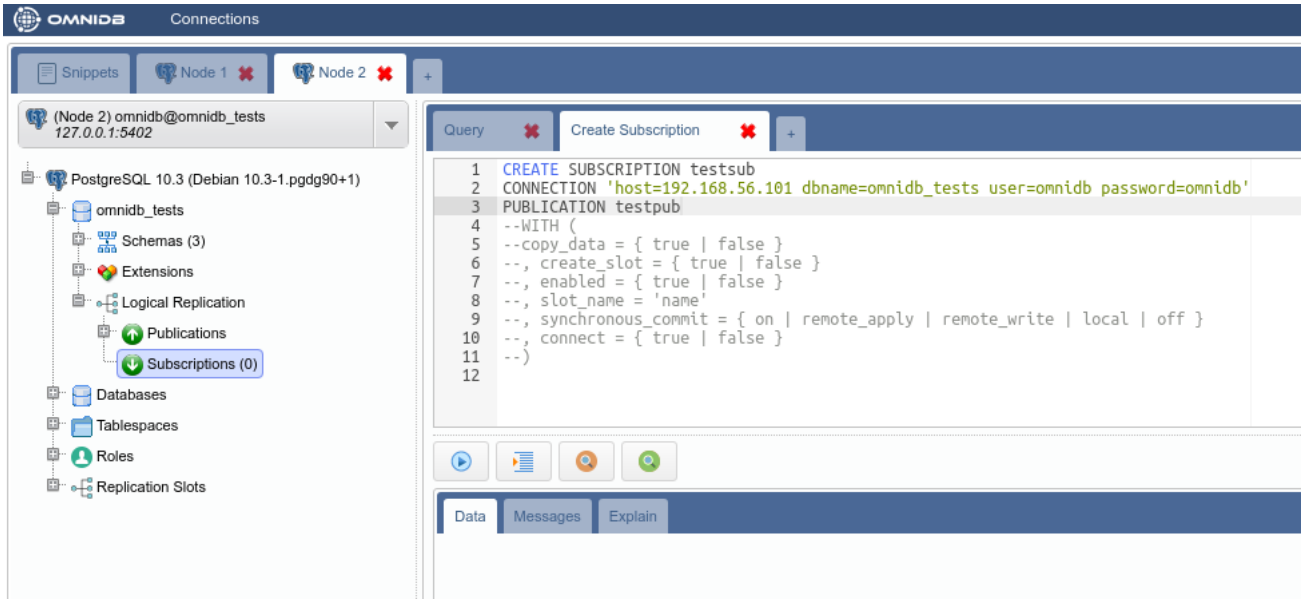


After adjusting and executing the command, you can right click the *Publications* node again and click on the *Refresh* action. You will see that will be created a new node with the same name you gave to the publication. Expanding this node, you will see the details and the tables for the publication:

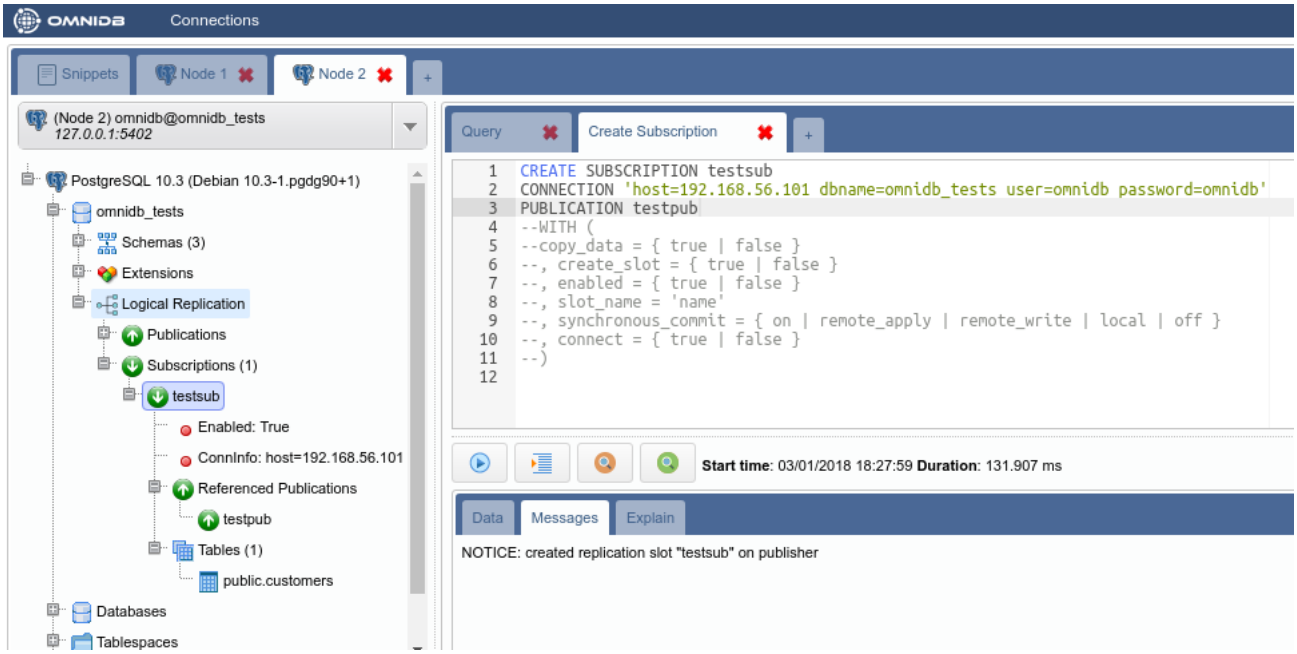


15.0.0.4 Create a subscription on the second machine

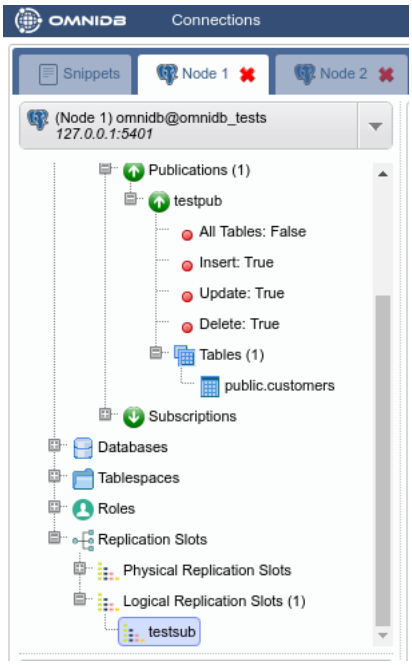
Inside the connection node, expand the *Logical Replication* node, then right click in the *Subscriptions* node, and choose the action *Create Subscription*. OmniDB will open a SQL template tab with the `CREATE SUBSCRIPTION` command ready for you to make some adjustments and run:



After adjusting and executing the command, you can right click the *Subscriptions* node again and click on the *Refresh* action. You will see that will be created a new node with the same name you gave to the subscription. Expanding this node, you will see the details, the referenced publications and the tables for the subscription:

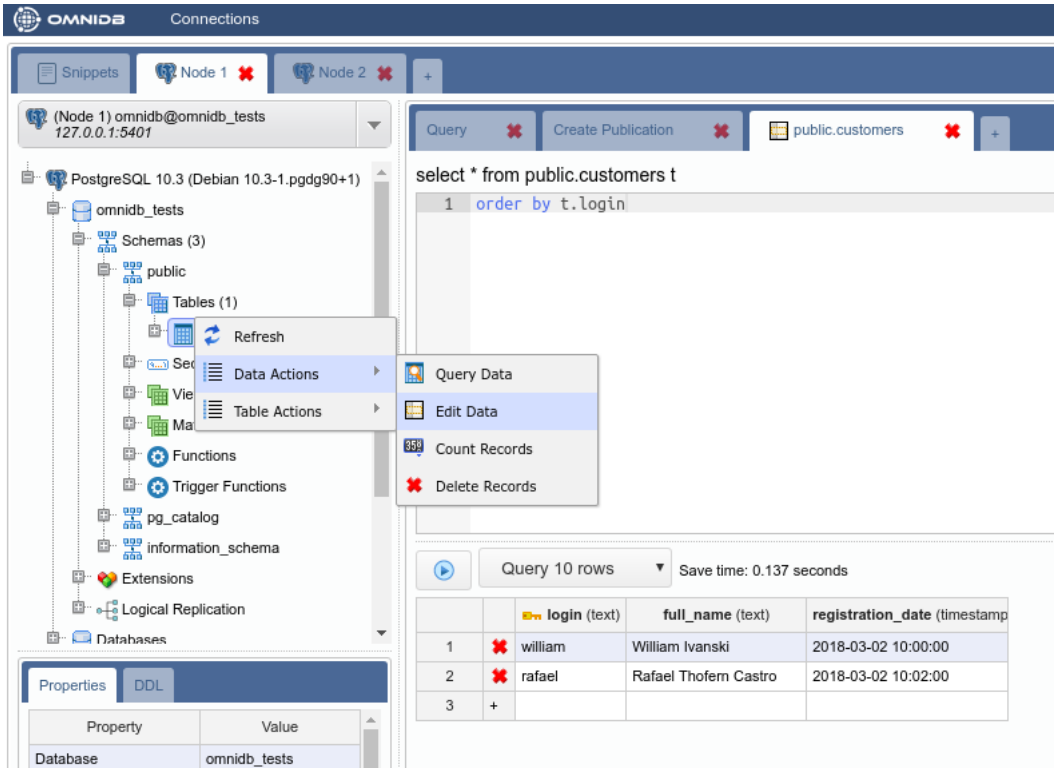


Also, the `CREATE SUBSCRIPTION` command created a *logical replication slot* called `testsub` (the same name as the subscription) in the first machine:

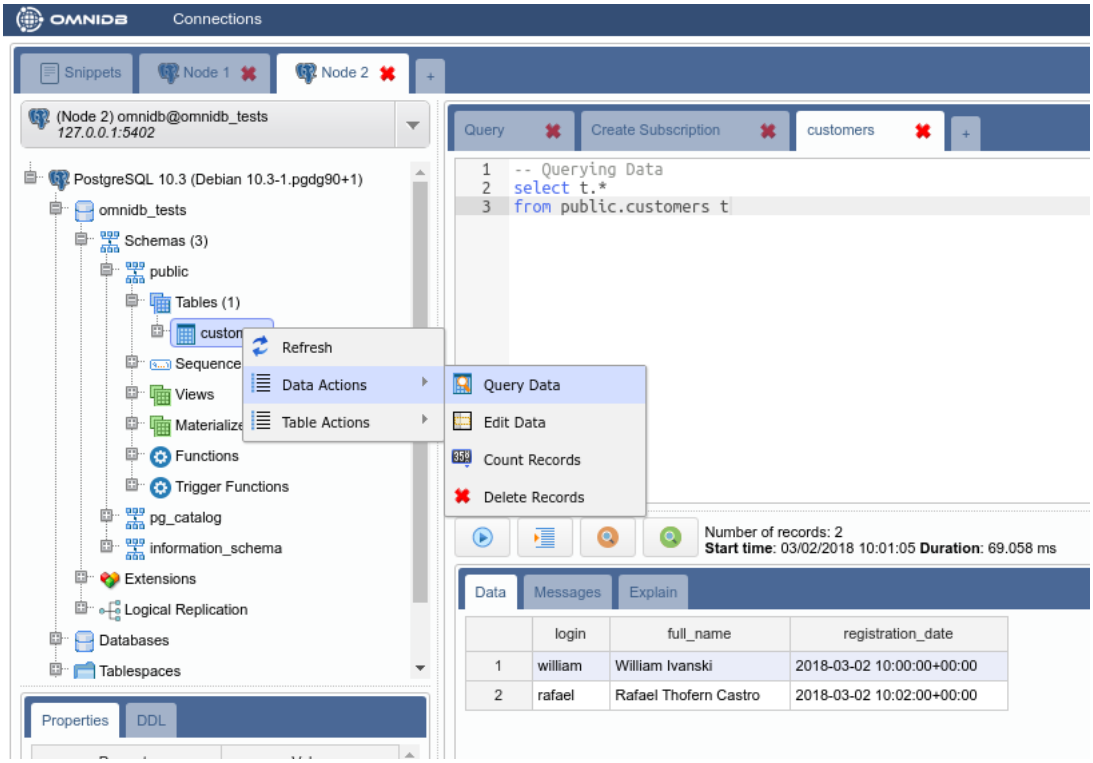


15.0.0.5 Testing the logical replication

To test the replication is working, let's create some data on the node 1. Right click on the table `public.customers`, then point to *Data Actions*, then click on the action *Edit Data*. In this grid, you are able to add, edit and remove data from the table. Add 2 sample rows, like this:



Then, on the other node, check if the table `public.customers` was automatically populated. Right click on the table `public.customers`, then point to *Data Actions*, then click on the action *Query Data*:



As we can see, both rows created in the first machine were replicated into the second machine. This tell us that the logical replication is working.

Now you can perform other actions, such as adding/removing tables to the publication and creating a new publication that publishes all tables.

16 pglogical

pglogical is a PostgreSQL extension that provides an advanced logical replication system that serves as a highly efficient method of replicating data as an alternative to physical replication.

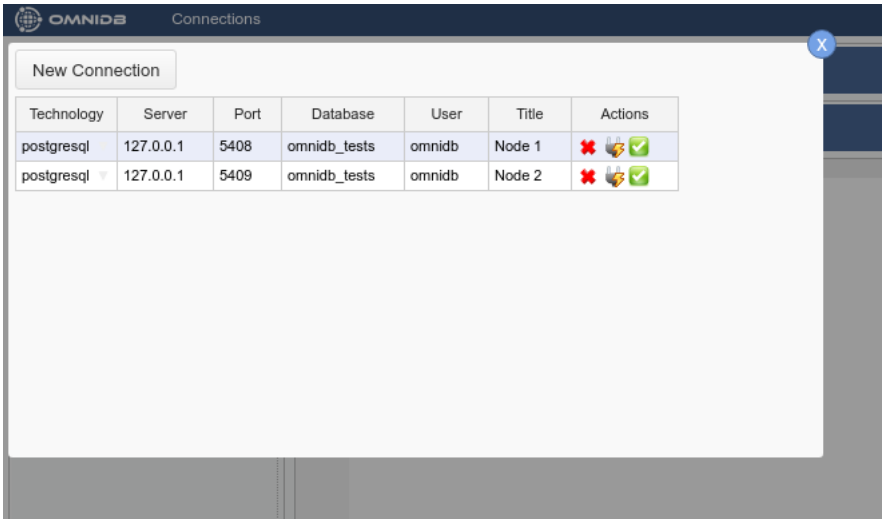
In this chapter, we will use a 2-node cluster to demonstrate pglogical with PostgreSQL 10. Note that on each PostgreSQL instance, you need to configure:

```
wal_level = 'logical'
track_commit_timestamp = on
max_worker_processes = 10   # one per database needed on provider node
                            # one per node needed on subscriber node
max_replication_slots = 10  # one per node needed on provider node
max_wal_senders = 10        # one per node needed on provider node
shared_preload_libraries = 'pglogical'
```

Also make sure to adjust file `pg_hba.conf` to grant access to `replication` between the 2 nodes.

16.0.0.1 Connecting to both nodes

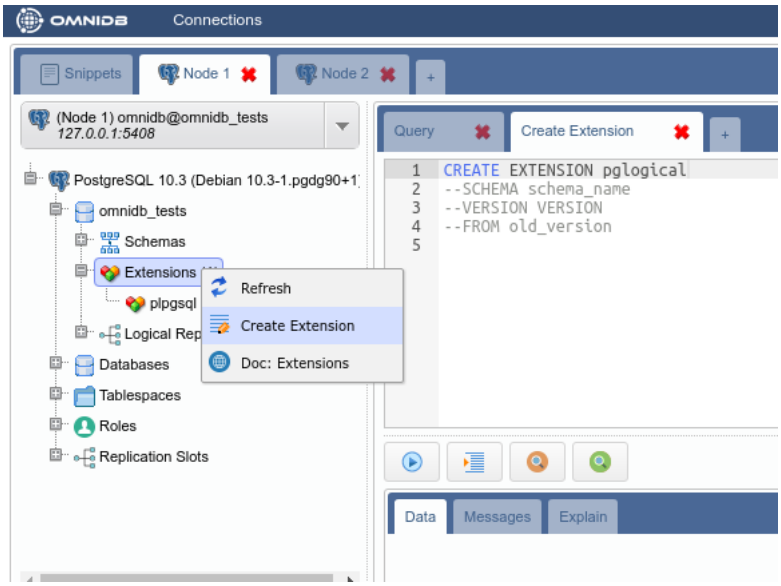
Let's use OmniDB to connect to both PostgreSQL nodes. First of all, fill out connection info in the connection grid:



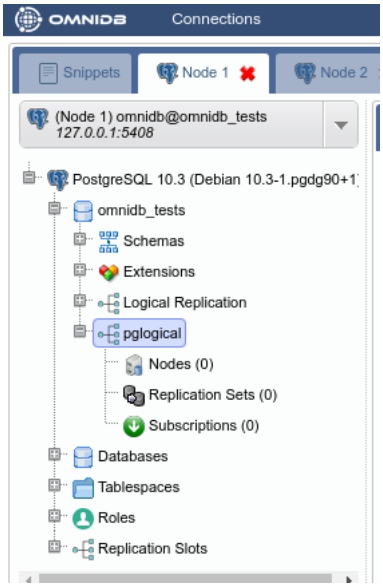
Then select both connections.

16.0.0.2 Create pglogical extension in both nodes

pglogical requires an extension to be installed in both nodes. Inside OmniDB, you can create the extension by right clicking on the *Extensions* node, and choosing the action *Create Extension*. OmniDB will open a SQL template tab with the `CREATE EXTENSION` command ready for you to make some adjustments and run:

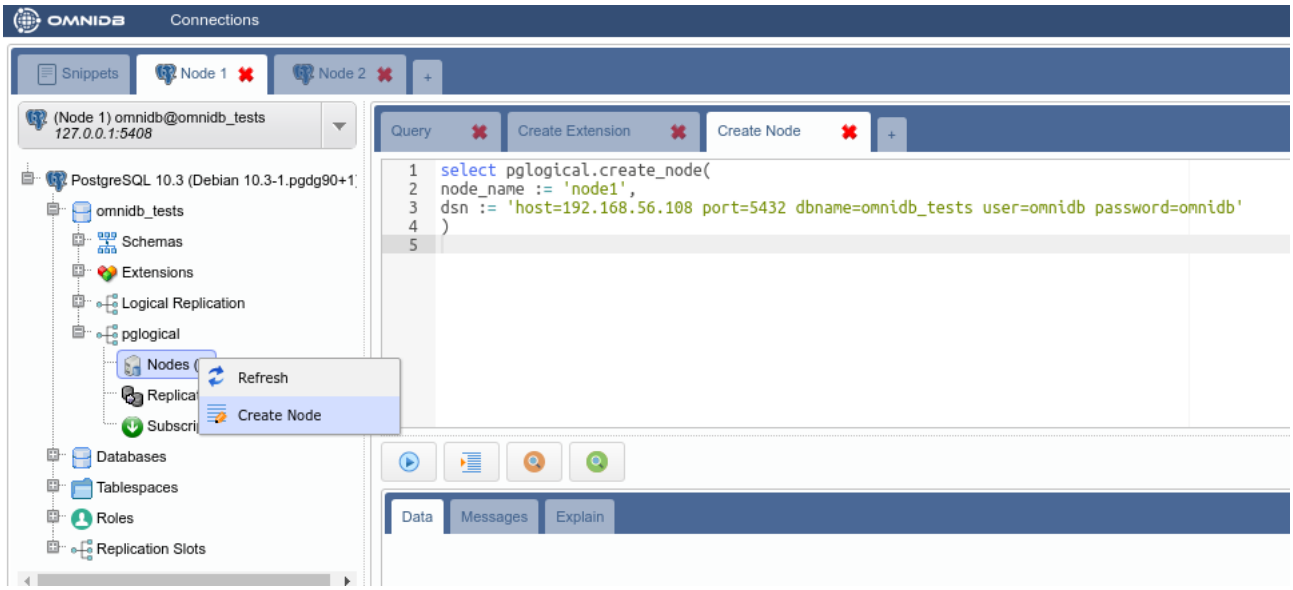


After you have created the extension, you need to refresh the root node of the treeview, by right-clicking on it and choosing *Refresh*. Then you will see that OmniDB already acknowledges the existence of pglogical in this database. However, pglogical is not active yet.



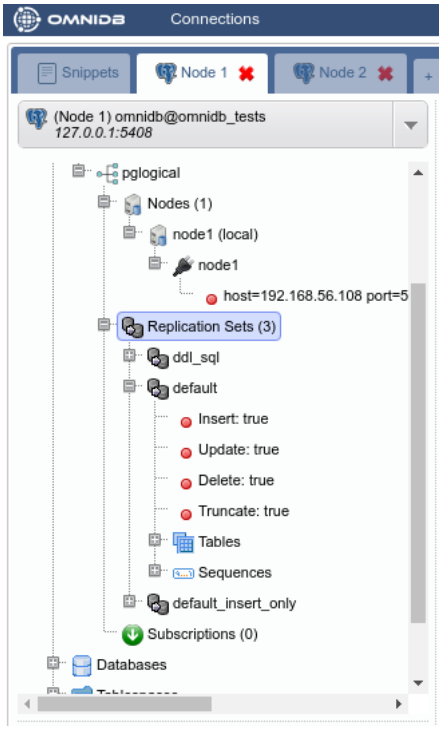
16.0.0.3 Create pglogical nodes

To activate pglogical in this database, we need to create a pglogical node on each machine. Inside the *pglogical* node of the treeview, right click *Nodes*, then choose *Create Node*. In the SQL template that will open, adjust the node name and the DSN and run the command.



Then right click *Nodes* again, but this time choose *Refresh*. You will see the node you just created. Note how OmniDB understands that this node is local. Expand the local node to see its interface inside. You can manage the interfaces of the nodes using OmniDB too.

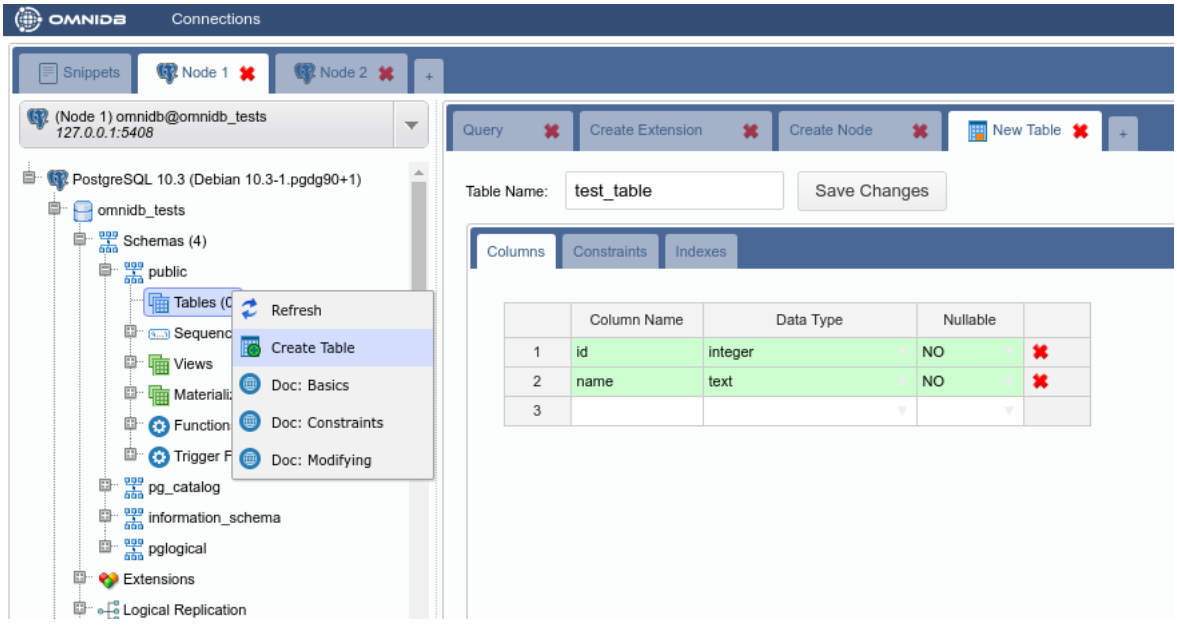
Go ahead and expand the *Replication Sets* node. You can see pglogical default replication sets are already created: *ddl_sql*, *default* and *default_insert_only*. You can also manage replication sets using OmniDB.

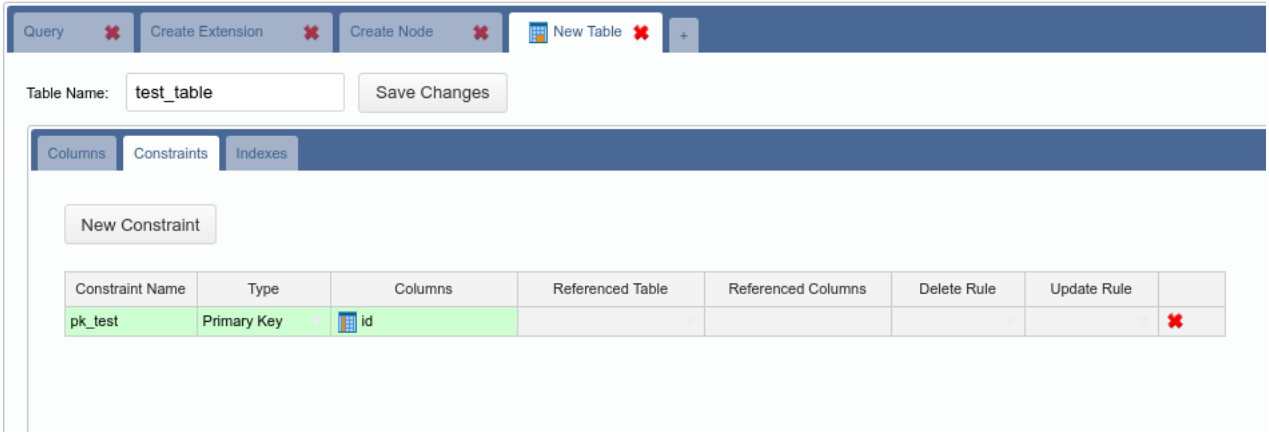


Now create a node on the other machine too. Choose a different name for the node.

16.0.0.4 Create a table on the first machine

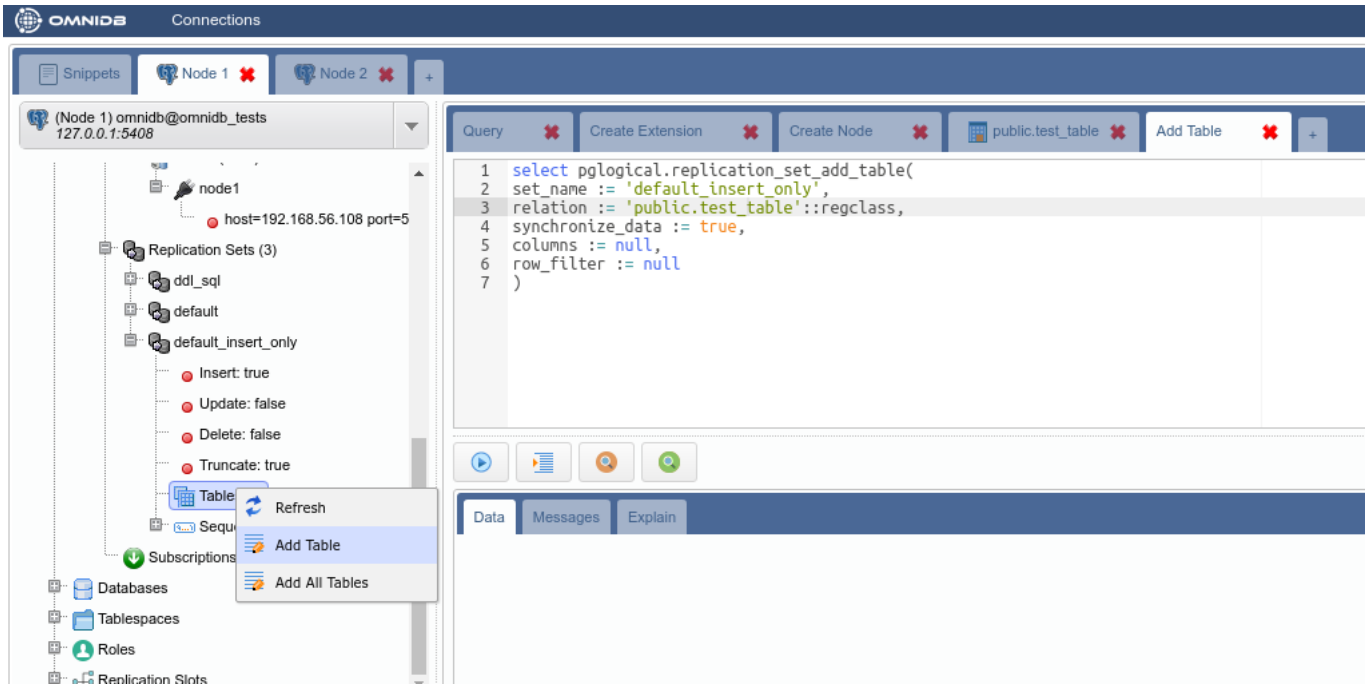
In the first machine, under the *Schemas* node, expand the *public* node, then right-click the *Tables* node and choose *Create Table*. In the form tab that will open, give the new table a name and some columns. Also add a primary key in the *Constraints* tab. When done, click in the *Save Changes* button.



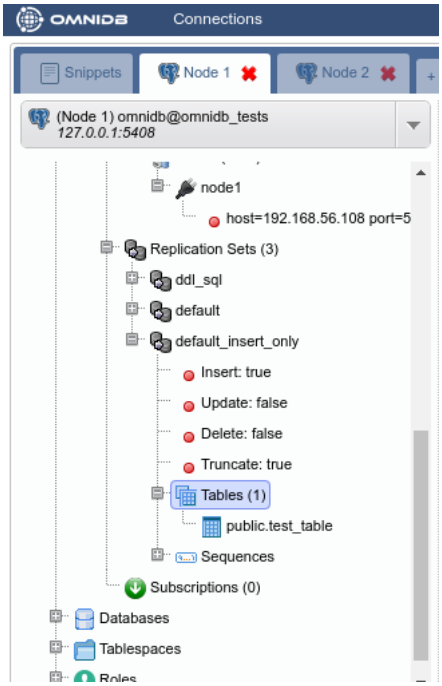


16.0.0.5 Add the new table to a replication set on the first machine

In the first machine, under the *default_insert_only* replication set, right click the *Tables* node and choose *Add Table*. In the SQL template tab that will open, change the table name in the *relation* argument and then execute the command.

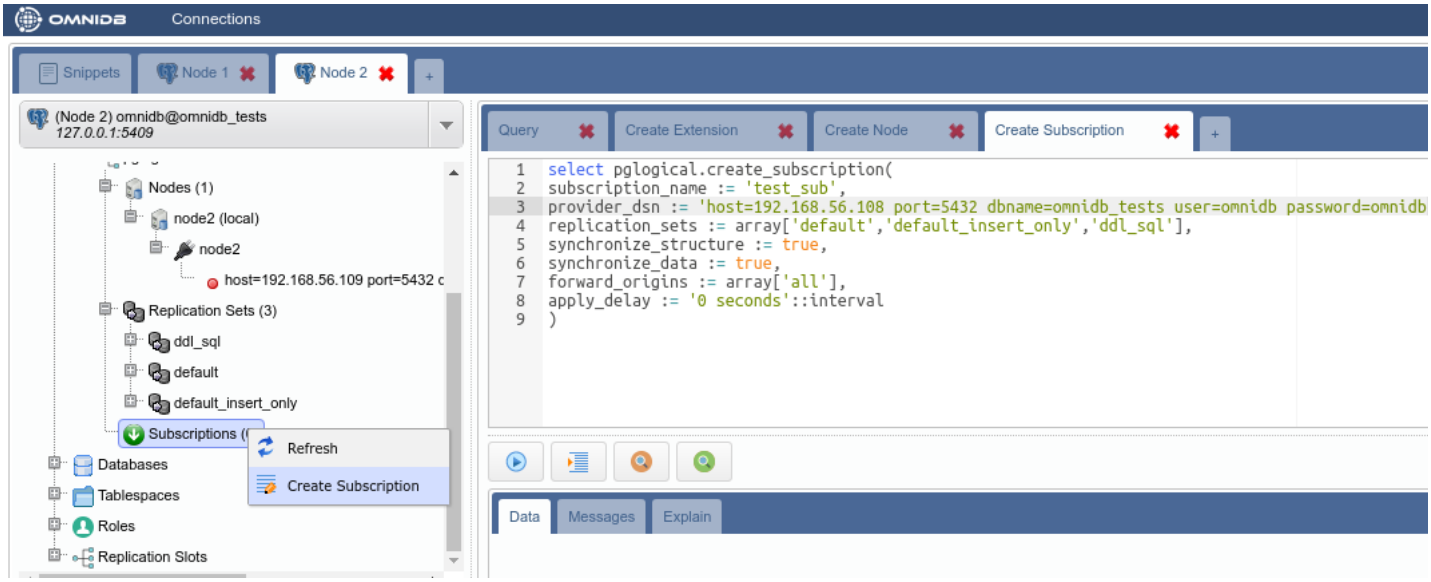


Refresh the *Tables* node to check the table was added to the replication set.

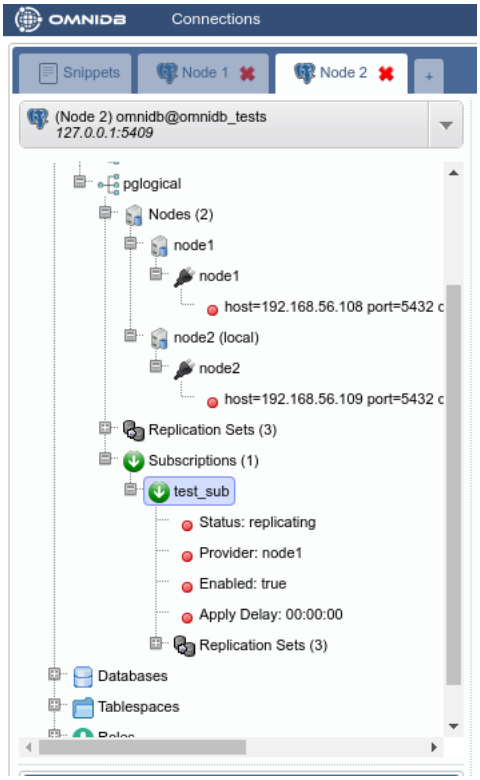


16.0.0.6 Add a subscription on the second machine

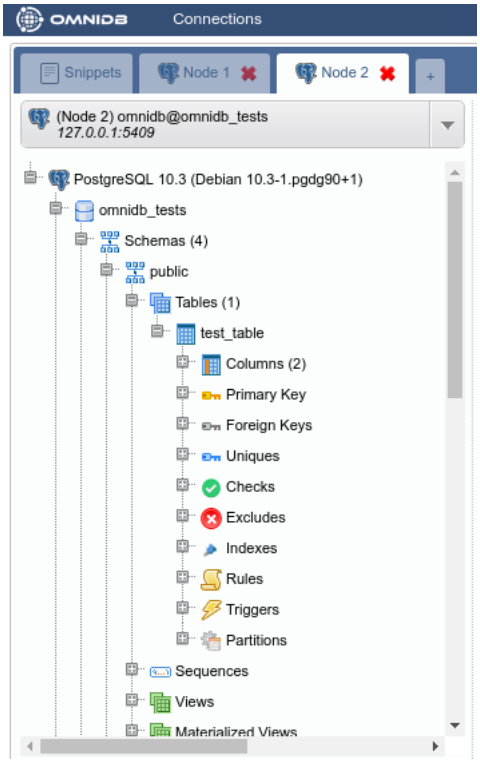
In the second machine, right-click the *Subscriptions* node and choose *Create Subscription*. In the SQL template tab that will open, change DSN of the first machine and then execute the command.



Refresh and expand both *Nodes* and *Subscriptions* nodes of the treeview. Note how now the second machine knows about the first machine. Also check the information OmniDB shows about the subscription we just created.

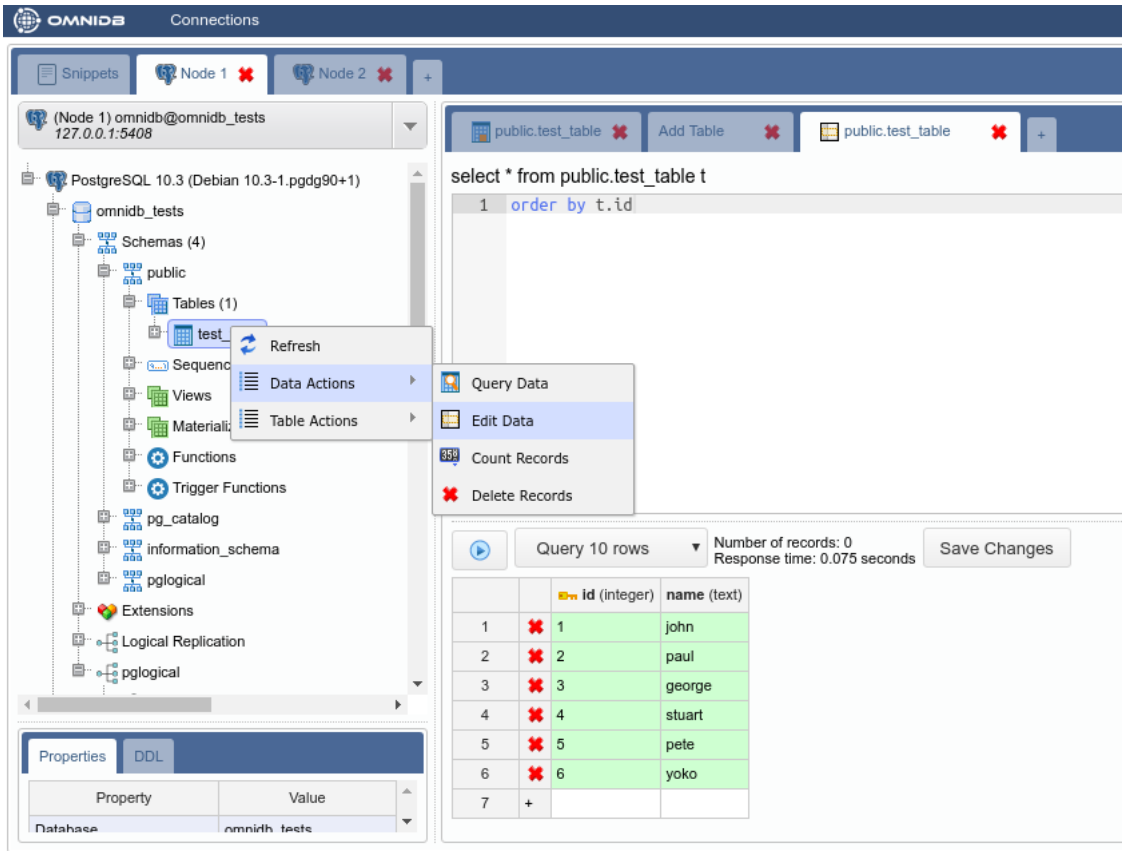


Also verify that the table *public.test_table* was created automatically in the second machine:

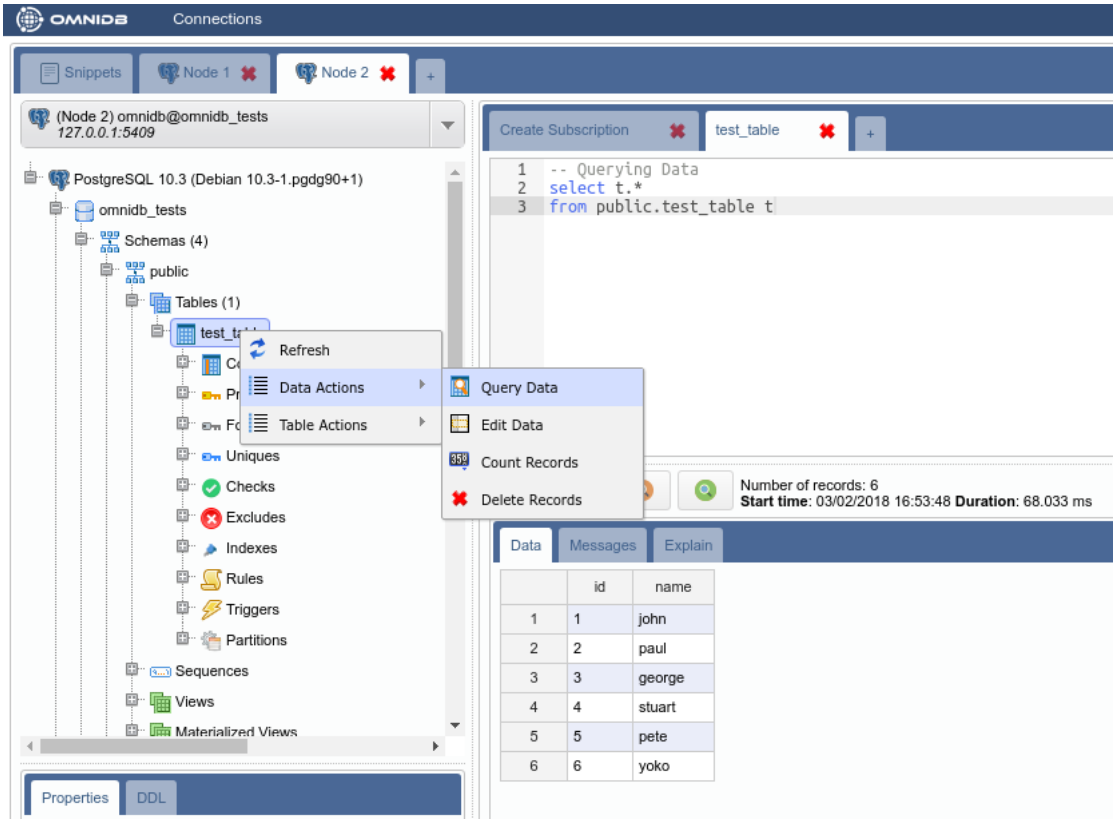


16.0.0.7 Add some data in the table on the first machine

In the first machine, under the *Schemas* node, expand the *public* node and the *Tables* node. Right-click in our table, *test_table*, move the mouse pointer to *Data Actions* and then click on *Edit Data*. Insert some data to the table. When finished, click on the *Save Changes* button.

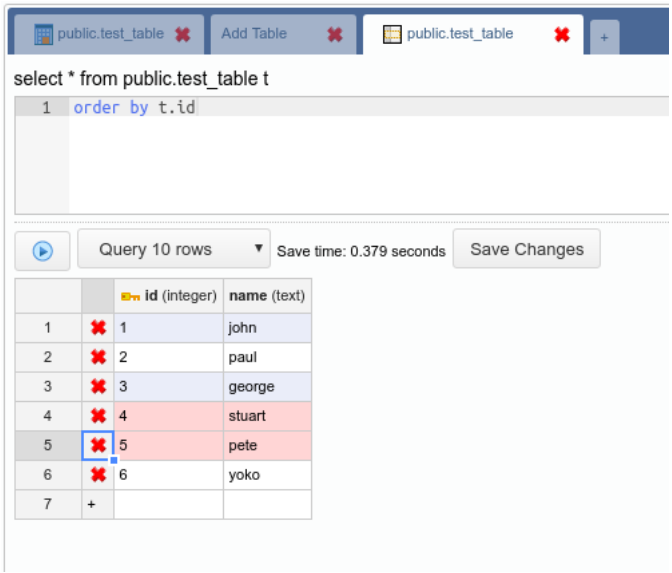


Now let us check the data was replicated. Go to the second machine and right-click the table, move the mouse pointer to *Data Actions* and then click on *Query Data*.

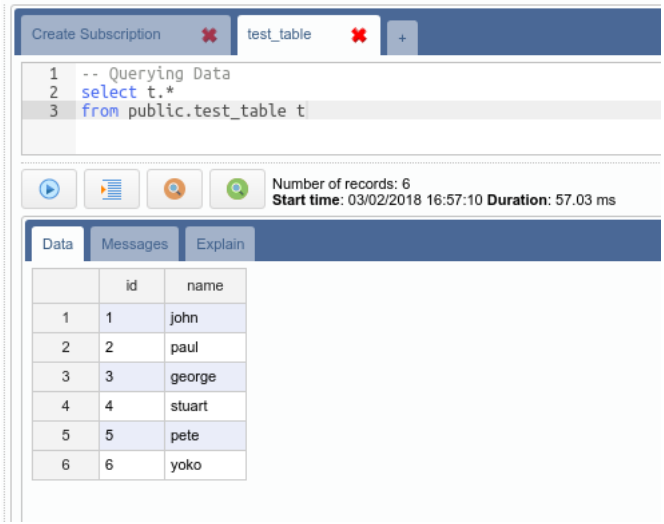


16.0.0.8 Check if delete is being replicated

In the *Edit Data* tab in the first machine, remove Pete and Stuart. Click on the button *Save Changes* when done.



Check if these 2 rows were deleted in the second machine.



They were not removed in the second machine because the table *public.test_table* is in the replication set *default_insert_only*, that does not replicate *updates* and *deletes*.

17 Postgres-BDR

Postgres-BDR (or just **BDR**, for short) is an open source project from 2ndQuadrant that provides multi-master features for PostgreSQL.

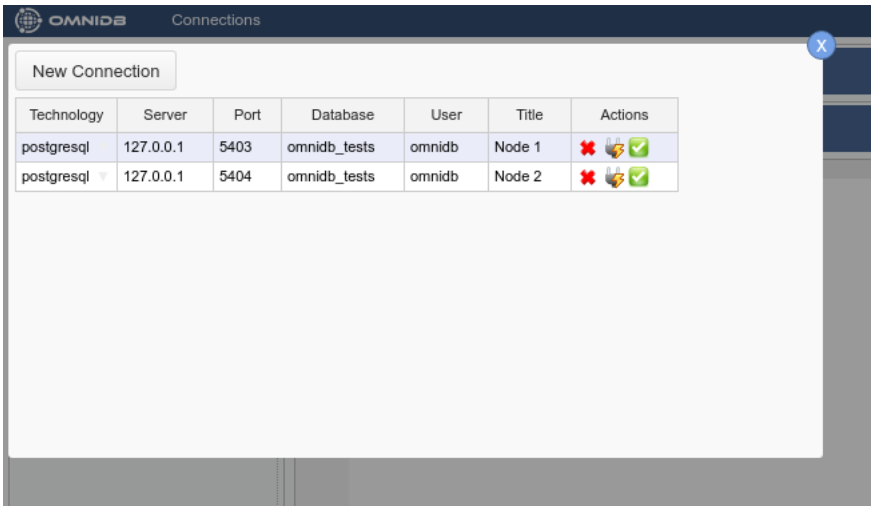
In this chapter, we will use a 2-node cluster to demonstrate Postgres-BDR 9.4. Note that on each PostgreSQL instance, you need to configure:

```
wal_level = 'logical'
track_commit_timestamp = on
max_worker_processes = 10   # one per database needed on provider node
                            # one per node needed on subscriber node
max_replication_slots = 10  # one per node needed on provider node
max_wal_senders = 10        # one per node needed on provider node
shared_preload_libraries = 'bdr'
```

Also make sure to adjust file `pg_hba.conf` to grant access to `replication` between the 2 nodes.

17.0.0.1 Connecting to both nodes

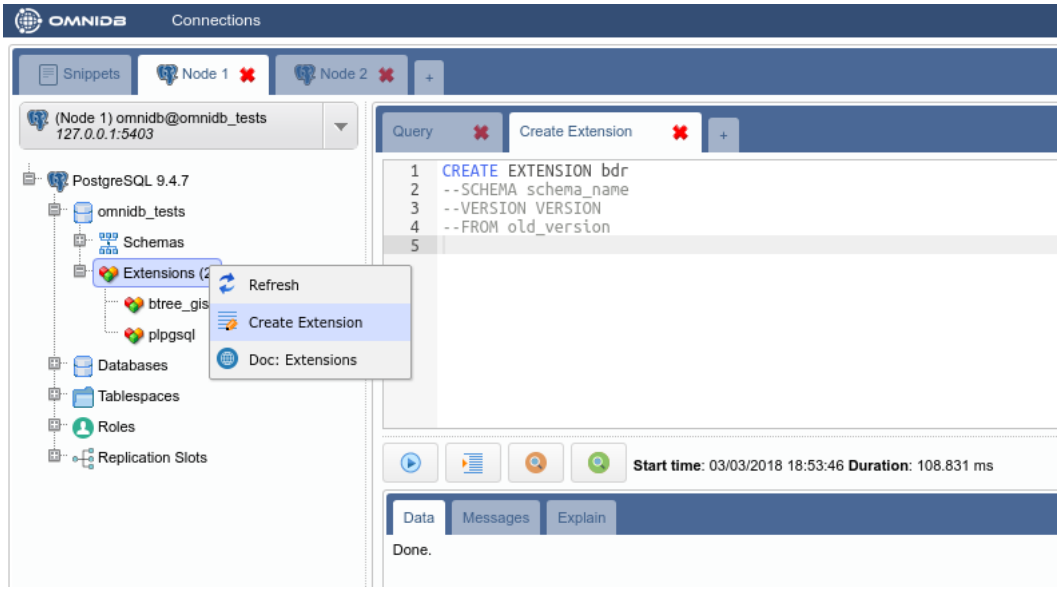
Let's use OmniDB to connect to both PostgreSQL nodes. First of all, fill out connection info in the connection grid:



Then select both connections.

17.0.0.2 Create required extensions

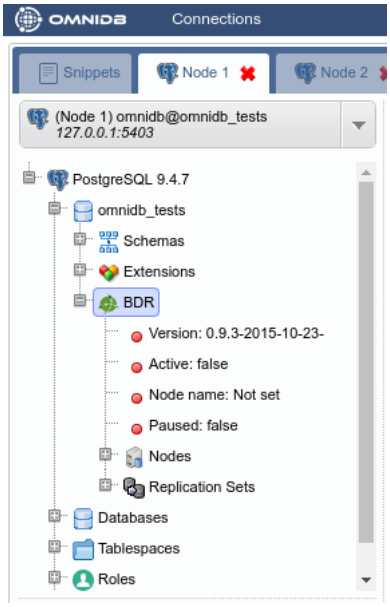
BDR requires 2 extensions to be installed on each database that should have multi-master capabilities: `btree_gist` and `bdr`. Inside OmniDB, you can create both extensions by right clicking on the *Extensions* node, and choosing the action *Create Extension*. OmniDB will open a SQL template tab with the `CREATE EXTENSION` command ready for you to make some adjustments and run:



You need to create both extensions `btree_gist` and `bdr` on both nodes.

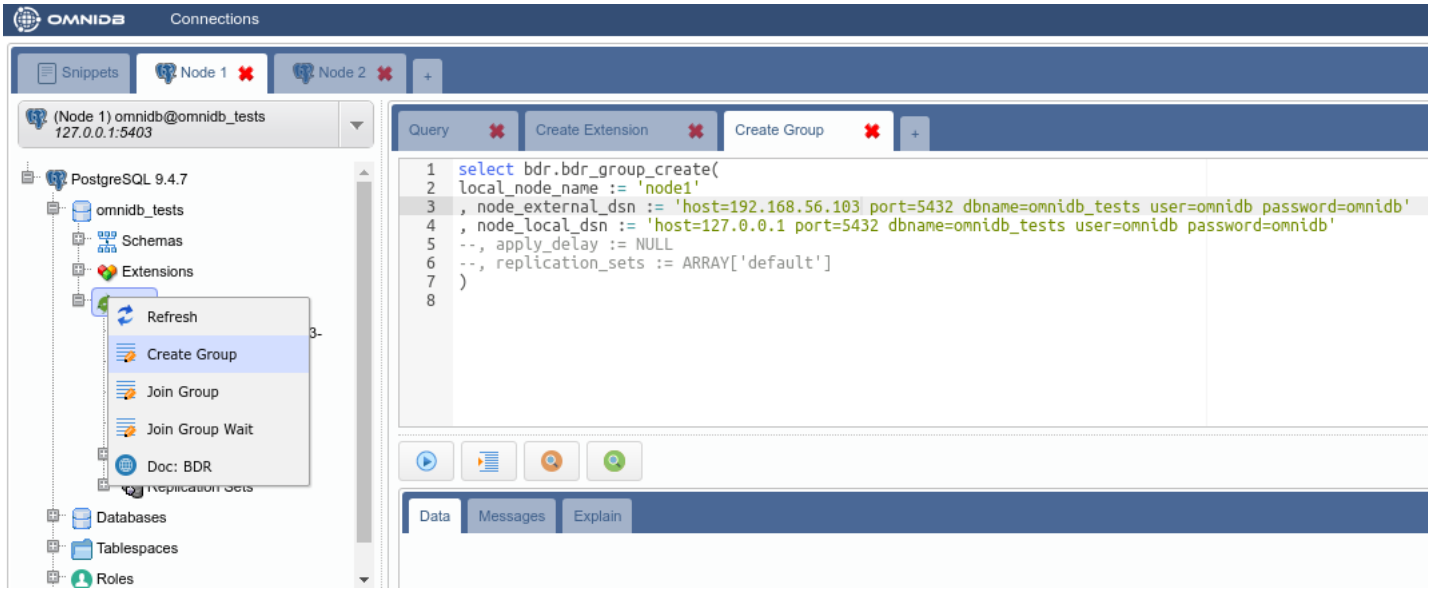
17.0.0.3 Create the BDR group in the first node

With both extensions installed, you can refresh the root node of the OmniDB tree view. A new *BDR* node will appear just inside your database. You can expand this node to see some informations about BDR:

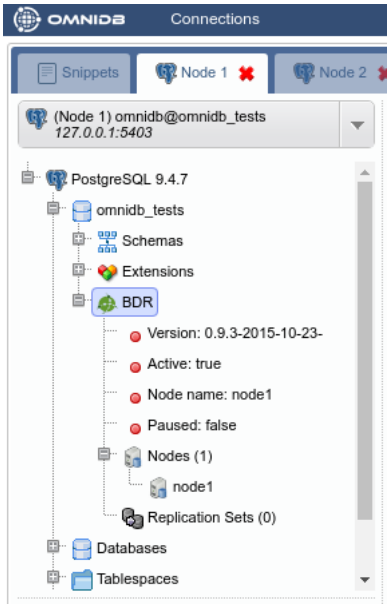


As you can see, BDR is not active yet. In the first node, we need to create a *BDR group*. The other nodes will join this group later.

To create a BDR group, right click in the *BDR* node. In the SQL template, adjust the node name and the node external connection info (the way other nodes will use to connect to this node):

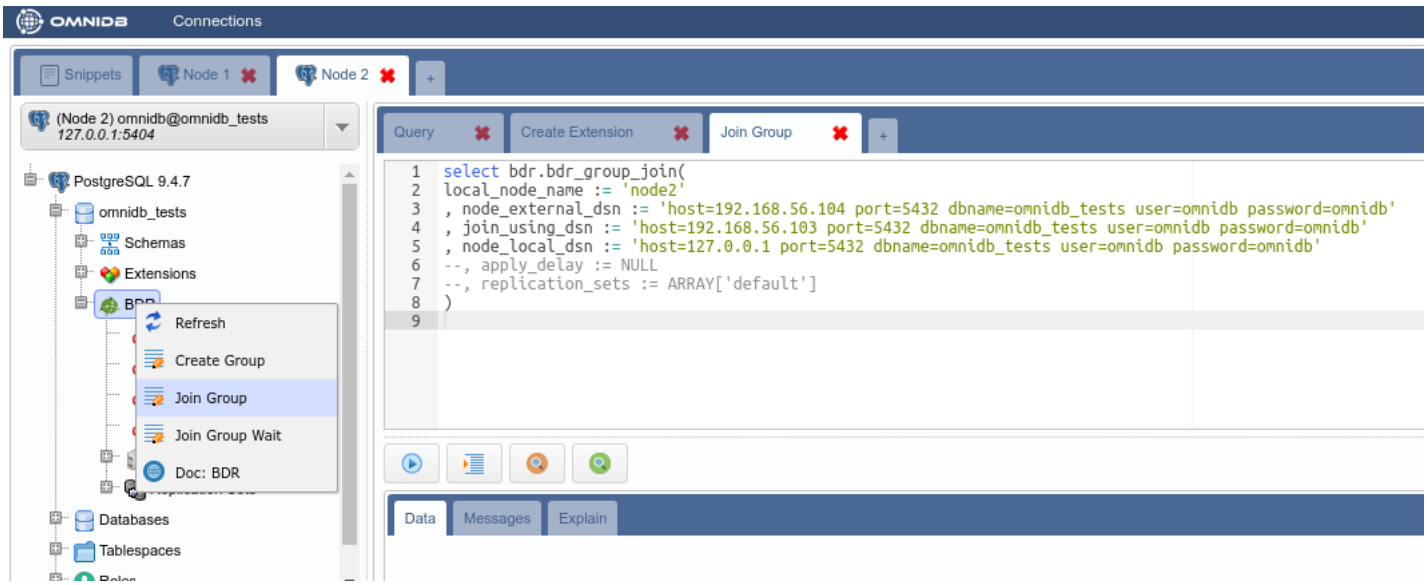


After you execute the above command, right click the *BDR* node and choose *Refresh*. You will see that now BDR is active in this node, now called `node1`. If you expand *Nodes*, you will see that this BDR group has only 1 node:

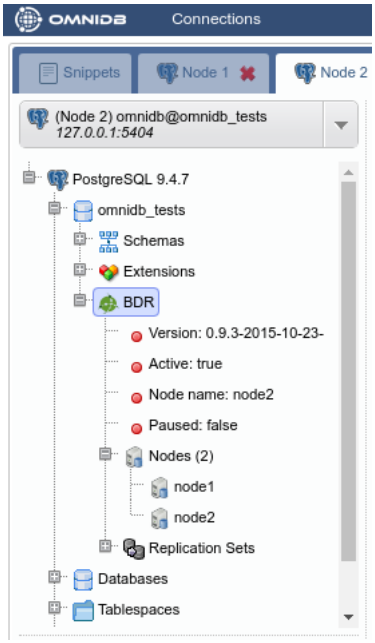


17.0.0.4 Join the BDR group in the second node

Now let's move to the other node. You can see that BDR is installed but not active yet. To link the two nodes, we will need to make this node join the BDR group that was previously created in the first node:

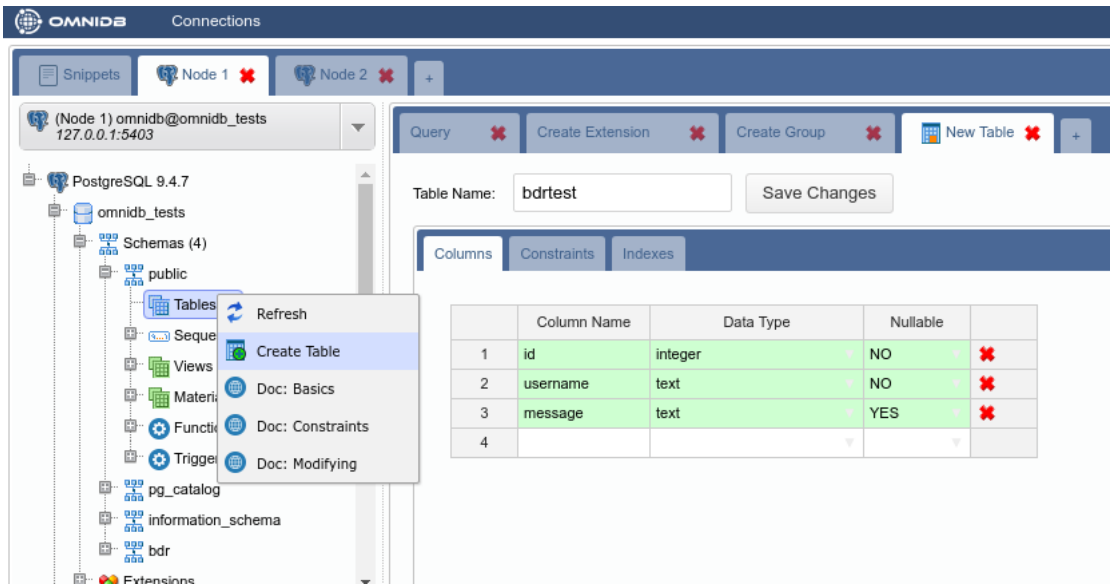


And now we can see that the second node has BDR active, his name in the BDR group is `node2`, and now the BDR group has 2 nodes:

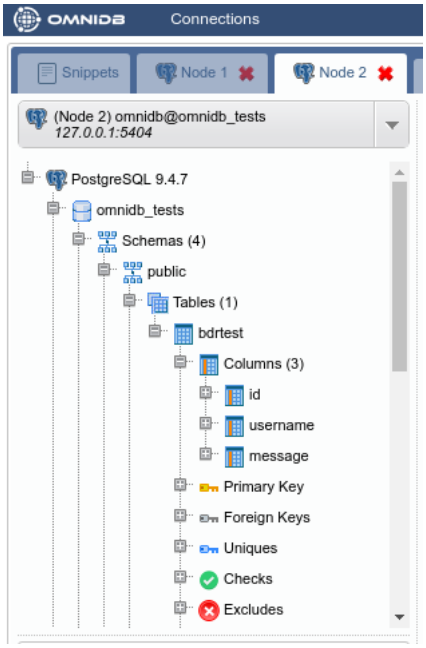


17.0.0.5 Creating a table in the first node

Let's create a table in the first node. Expand the `public` schema, right click the *Tables* node and choose *Create Table*. Give the new table a name and add some columns. When done, click in the button *Save Changes*:

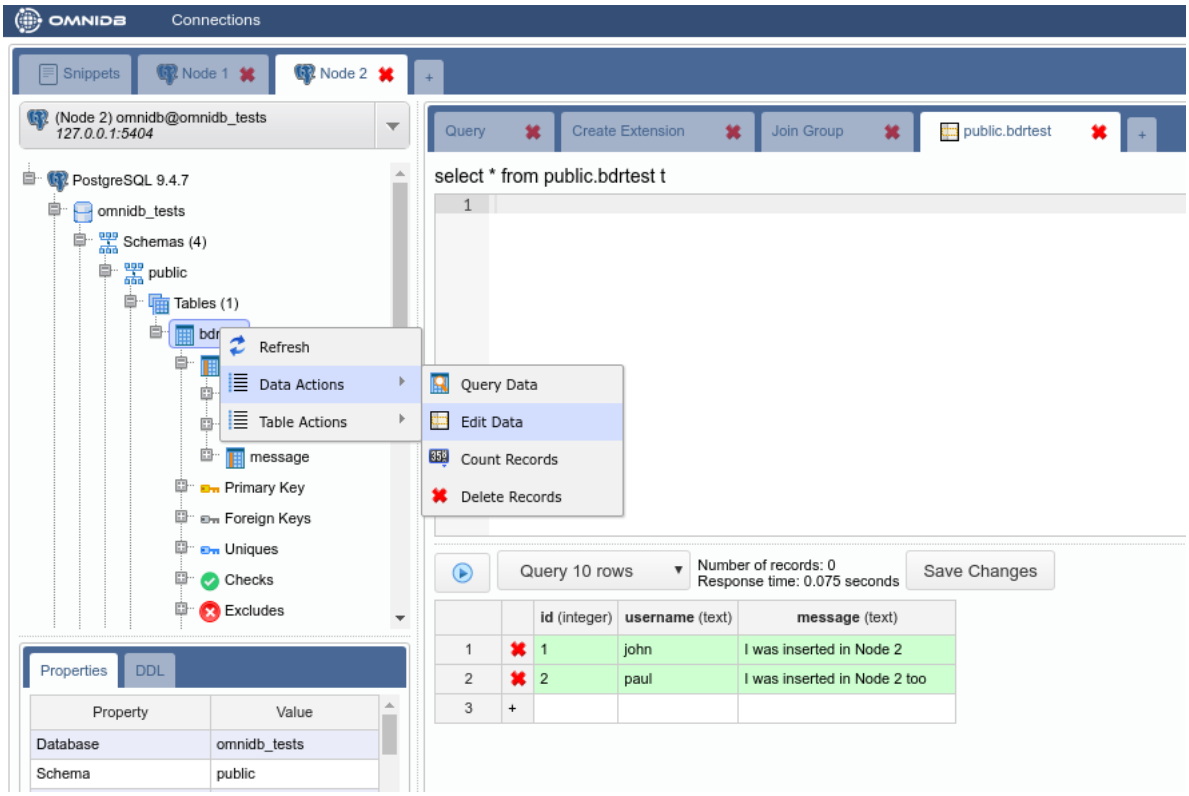


Now confirm that the table has been created in the first node by right clicking the *Tables* node and choosing *Refresh*. Go to the second node, expand the schema `public`, then expand the *Tables* node. Note that the table has been replicated from `node1` to `node2`. If the table was created in the second node, it would have been created in the first node as well, because in BDR all nodes are masters.

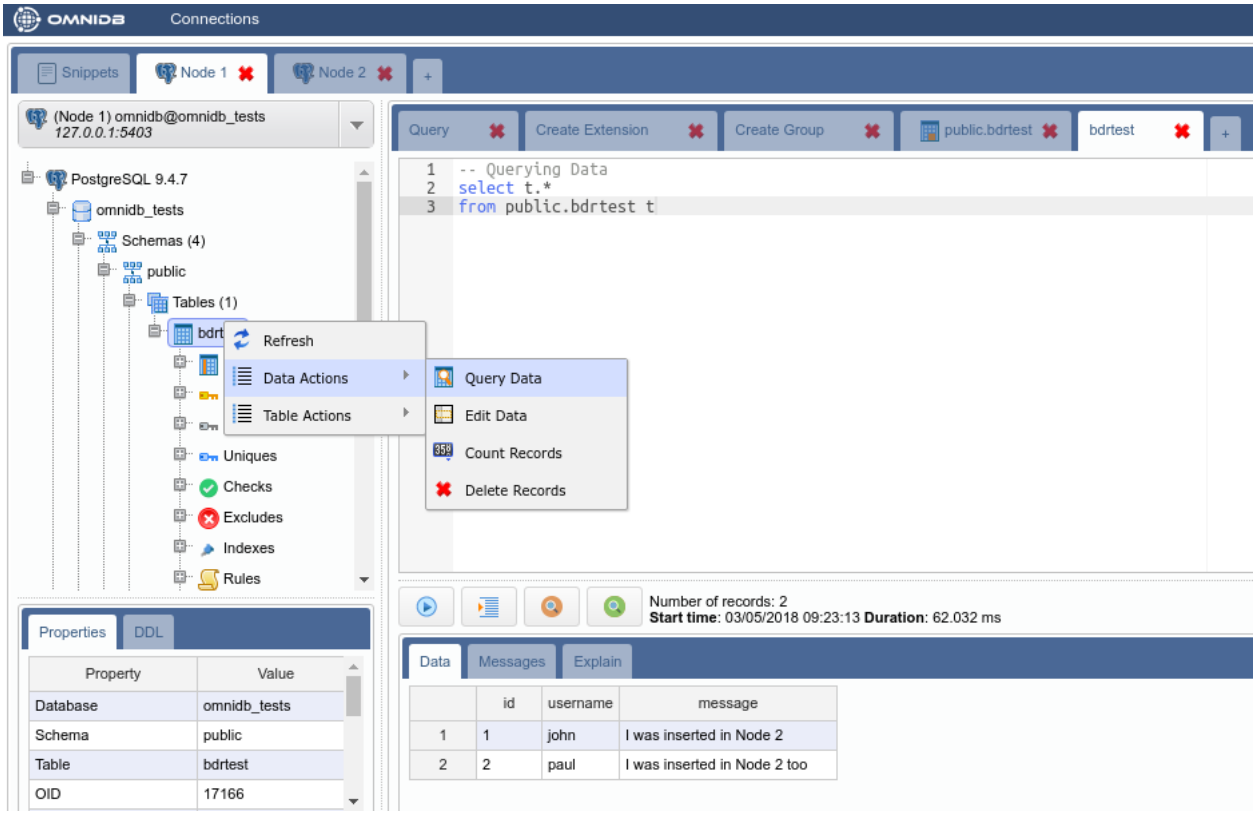


17.0.0.6 Adding some data in the second node

While you are at the second node, right click the table `bdrtest`, point to *Data Actions* and then click in *Edit Data*. Add some rows to this table. When finished, click in the *Save Changes* button.

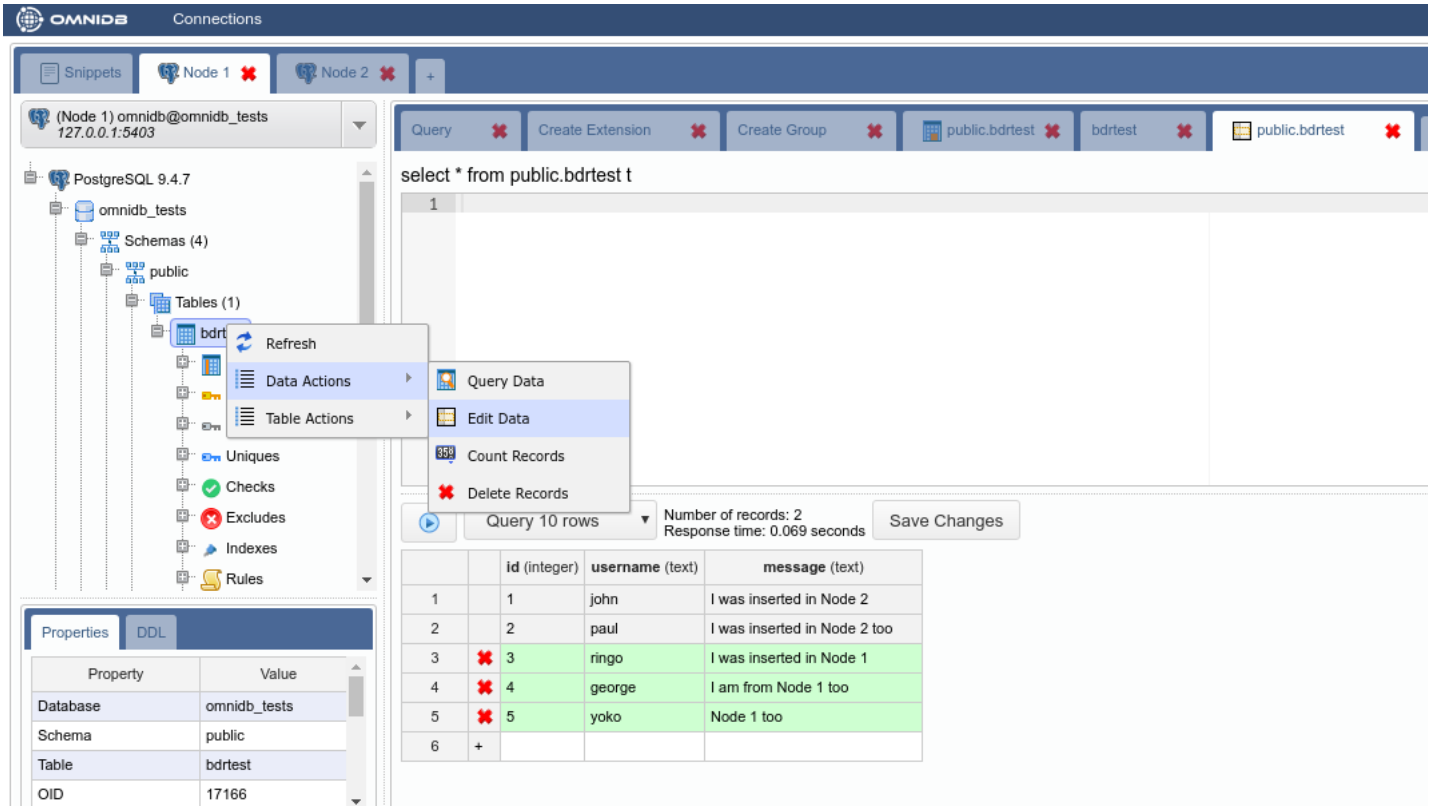


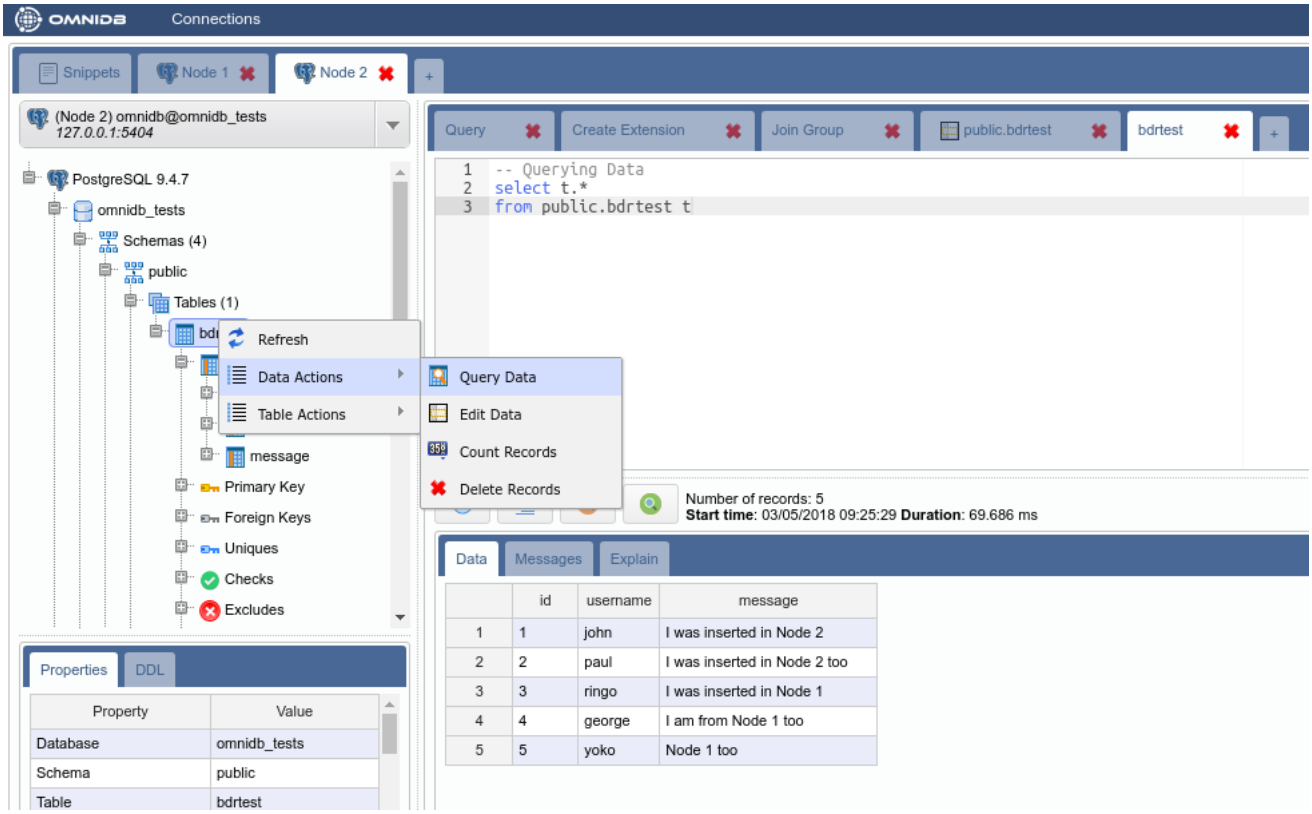
Now go to the first node, right click the table, point to *Data Actions* and then click in *Query Data*. See how the rows created in `node2` were automatically replicated into `node1`.



17.0.0.7 Adding some data in the first node

Let's repeat the same procedure above, but instead of inserting rows from the second node, let's insert some rows while connected to the first node. Note how they replicate into the second node in the same way.





18 Postgres-XL

Postgres-XL (or just **XL**, for short) is an open source project from 2ndQuadrant. It is a massively parallel database built on top of PostgreSQL, and it is designed to be horizontally scalable and flexible enough to handle various workloads.

In this chapter, we will use a cluster with 1 GTM and 1 coordinator on the same virtual machine, and 2 data nodes (each data node on a separate virtual machine).

Machine	IP	Role
xl_gtmcoord	192.168.56.105	GTM and coordinator
xl_datanode1	192.168.56.106	data node
xl_datanode2	192.168.56.107	data node

On each machine, you need to clone Postgres-XL repository and compile it. You also need to set specific XL parameters on file `postgresql.conf` and make sure all machines are communicating to each other by adjusting file `pg_hba.conf`. More information on how Postgres-XL works and how to install it on Postgres-XL documentation. You can also refer to this blog post.

18.0.0.1 Preparing the nodes

After you have XL up and running on all nodes, you need to let them know about their roles in the cluster and also about the other nodes. On the GTM/coordinator node, run the following as `postgres` user:

```
ALTER NODE coord1 WITH (TYPE = 'coordinator', HOST = 'localhost', PORT = 5432);
CREATE NODE datanode_1 WITH (TYPE = 'datanode', HOST = '192.168.56.106', PORT = 5432);
CREATE NODE datanode_2 WITH (TYPE = 'datanode', HOST = '192.168.56.107', PORT = 5432);
SELECT pgxc_pool_reload();
```

On the first data node, run:

```
ALTER NODE datanode_1 WITH (TYPE = 'datanode', HOST = 'localhost', PORT = 5432);
CREATE NODE coord1 WITH (TYPE = 'coordinator', HOST = '192.168.56.105', PORT = 5432);
CREATE NODE datanode_2 WITH (TYPE = 'datanode', HOST = '192.168.56.107', PORT = 5432);
SELECT pgxc_pool_reload();
```

On the second data node, run:

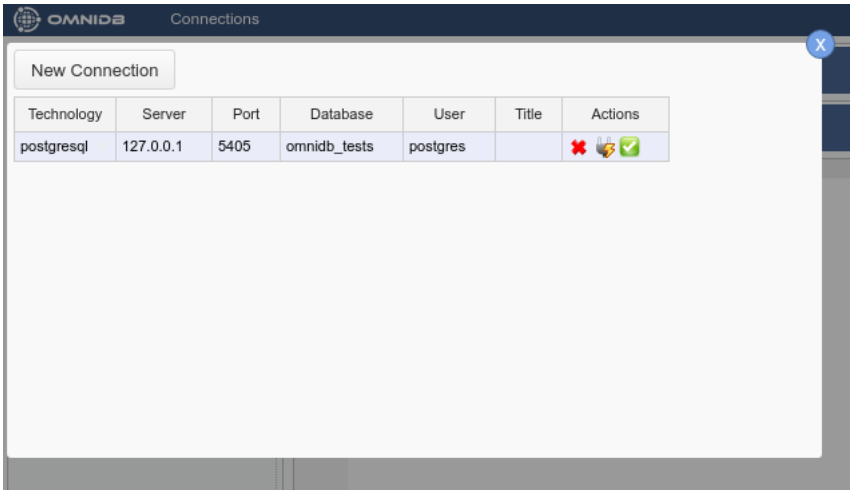
```
ALTER NODE datanode_2 WITH (TYPE = 'datanode', HOST = 'localhost', PORT = 5432);
CREATE NODE coord1 WITH (TYPE = 'coordinator', HOST = '192.168.56.105', PORT = 5432);
CREATE NODE datanode_1 WITH (TYPE = 'datanode', HOST = '192.168.56.106', PORT = 5432);
SELECT pgxc_pool_reload();
```

Finally, on the GTM/coordinator, change password of user `postgres` and create a database:

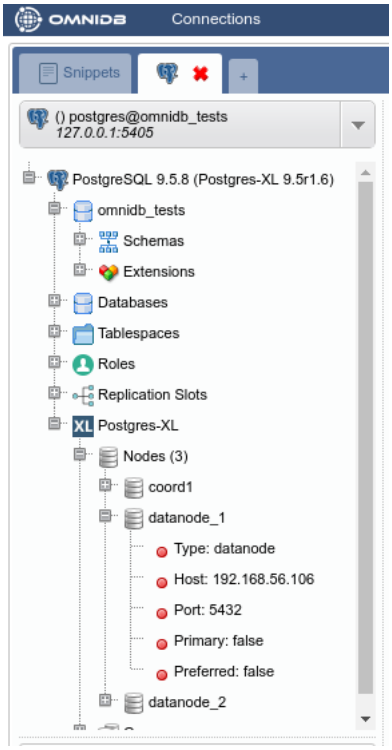
```
ALTER USER postgres WITH PASSWORD 'omnidb';
CREATE DATABASE omnidb_tests;
```

18.0.0.2 Connecting to the cluster

Let's use OmniDB to connect to the coordinator node. First of all, fill out connection info in the connection grid:

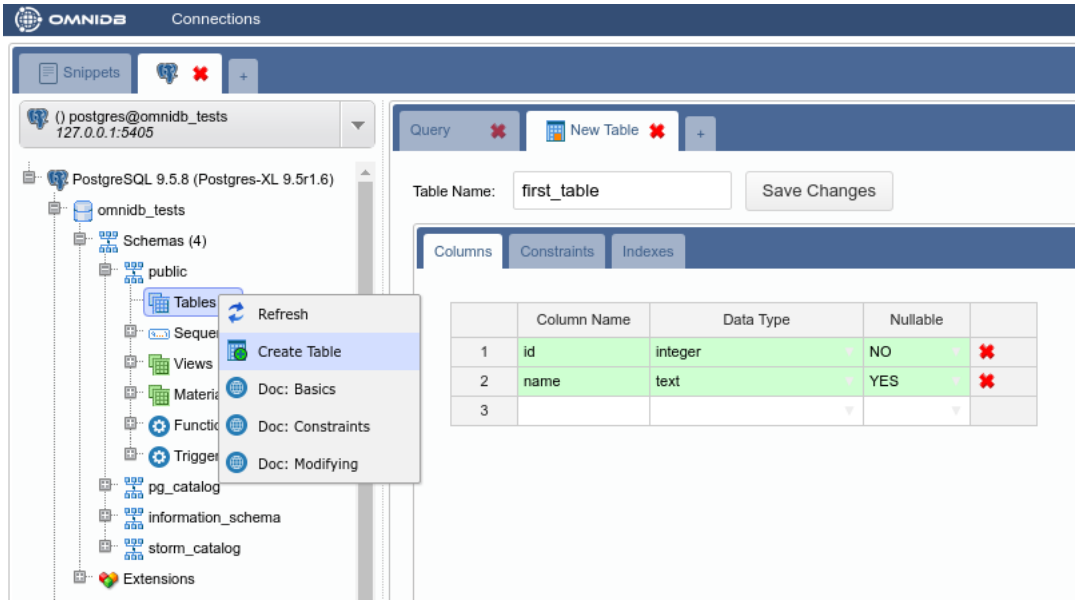


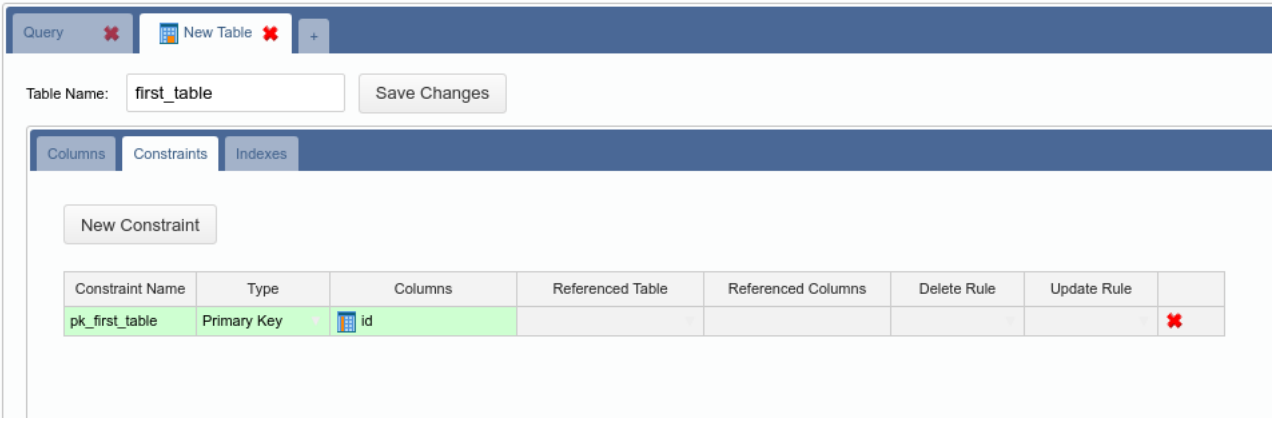
Then select the connection. You will see OmniDB workspace window. Expand the tree root node. Note that OmniDB identifies it is connected to a Postgres-XL cluster and shows a specific node called *Postgres-XL* just inside the tree root node. Expand this node to see all the nodes we have in our cluster:



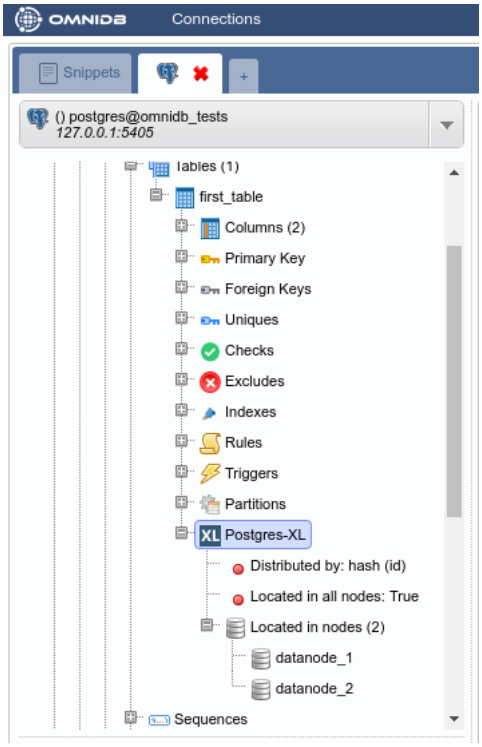
18.0.0.3 Creating a HASH table

From the root node, expand *Schemas*, then *public*, then right click on the *Tables* node. Click on *Create Table*. Name your new table, add some columns to it and do not forget to add a primary key too:



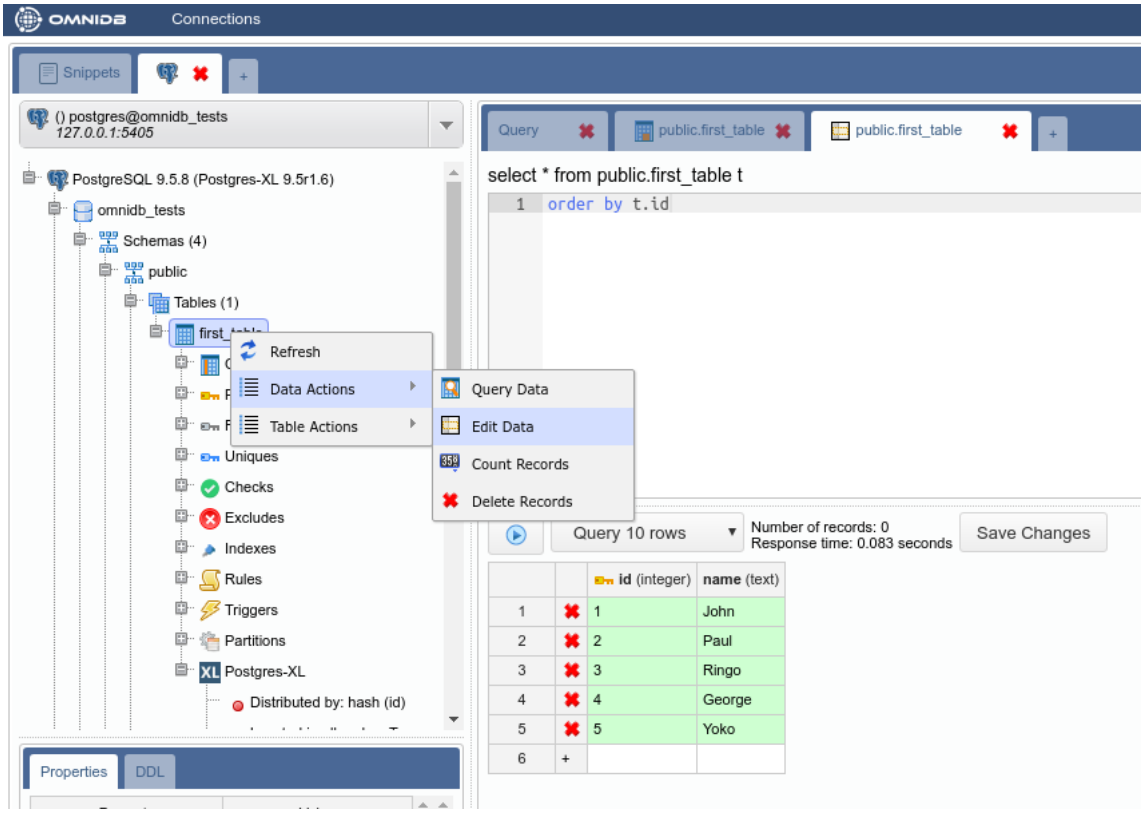


When done, click on the *Save Changes* button. Now right click on the *Tables* node and click on *Refresh*. You will see the new table created. Expand it to see that there is also a *Postgres-XL* node inside of it. Check its properties.

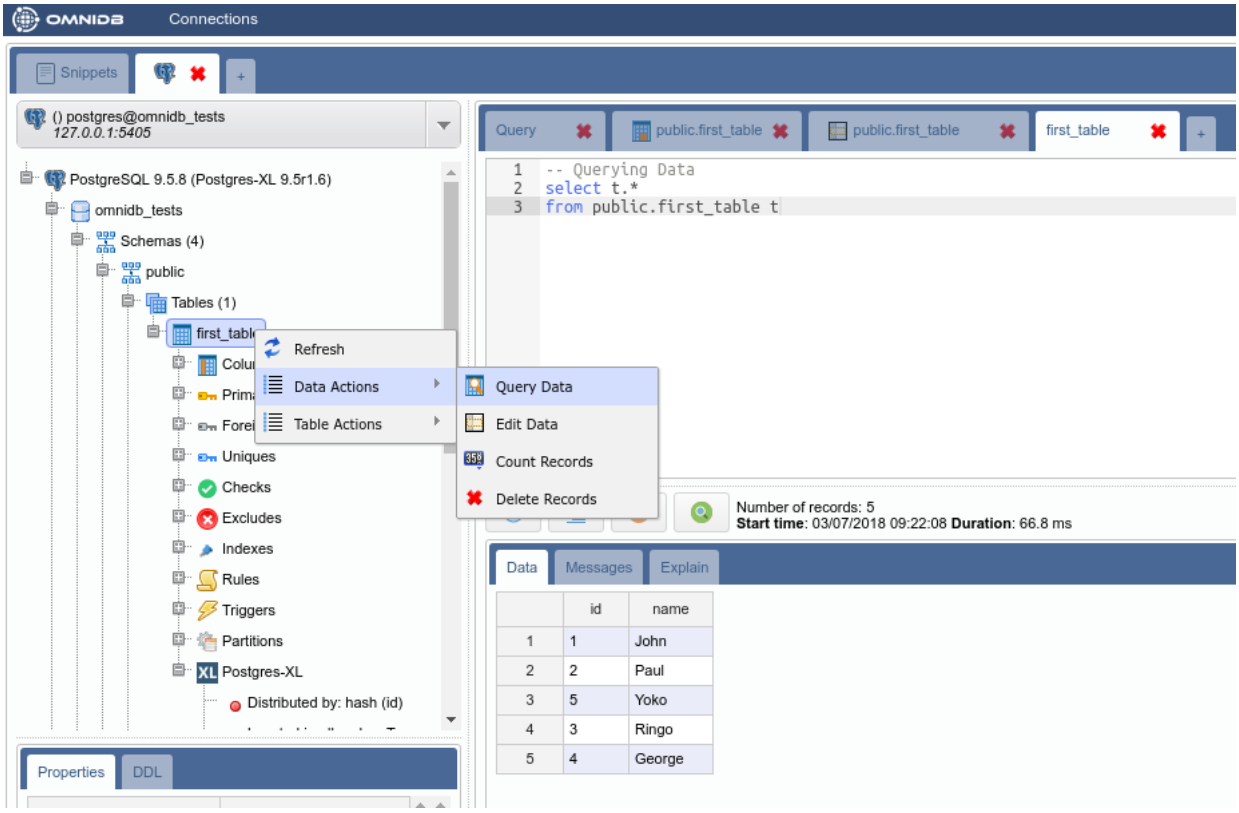


By default, Postgres-XL always try to create a table distributed by HASH. It means that the data will be split into the nodes regularly, through a hash function applied on the specified column. If present, it will use the primary key, or a unique constraint otherwise. If there is no primary key nor unique constraint, Postgres-XL uses the first eligible column. If not possible to distribute by HASH, then Postgres-XL will create the table distributed by ROUNDROBIN, which means that the data will be split in a way that every new row will be added to a different data node.

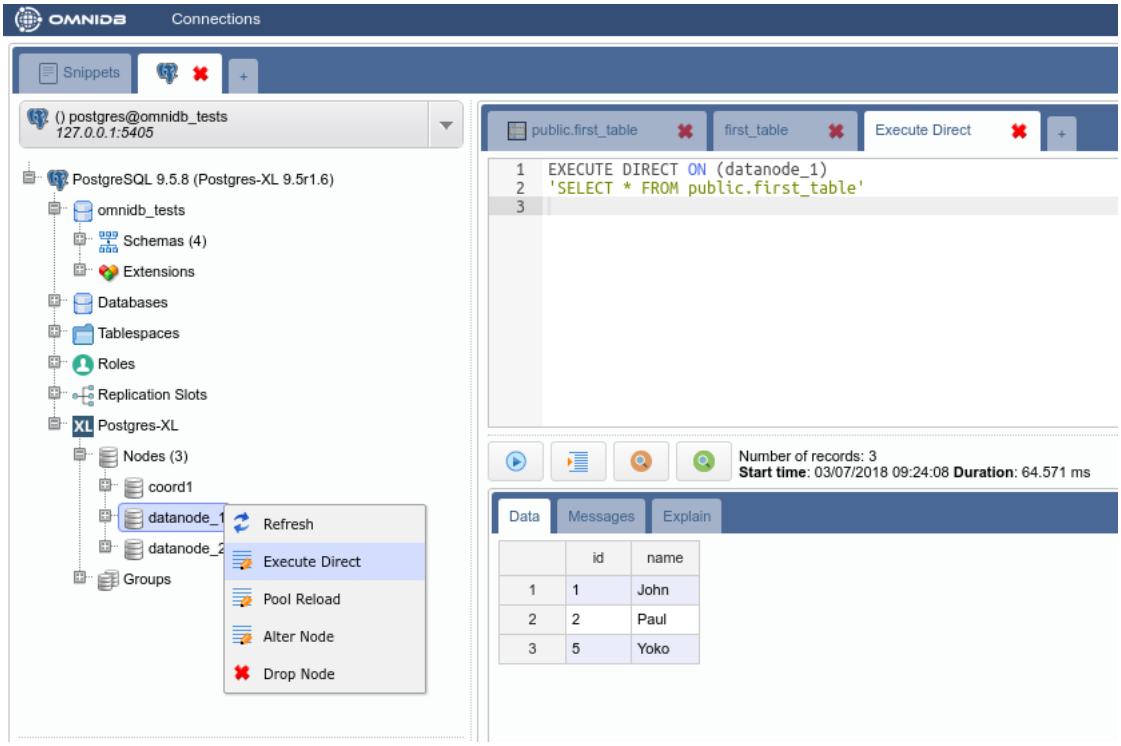
Now let's add some rows in our new table. Right click on the table, then go to *Data Actions* and then click on *Edit Data*. Add some rows and then click on the *Save Changes* button:

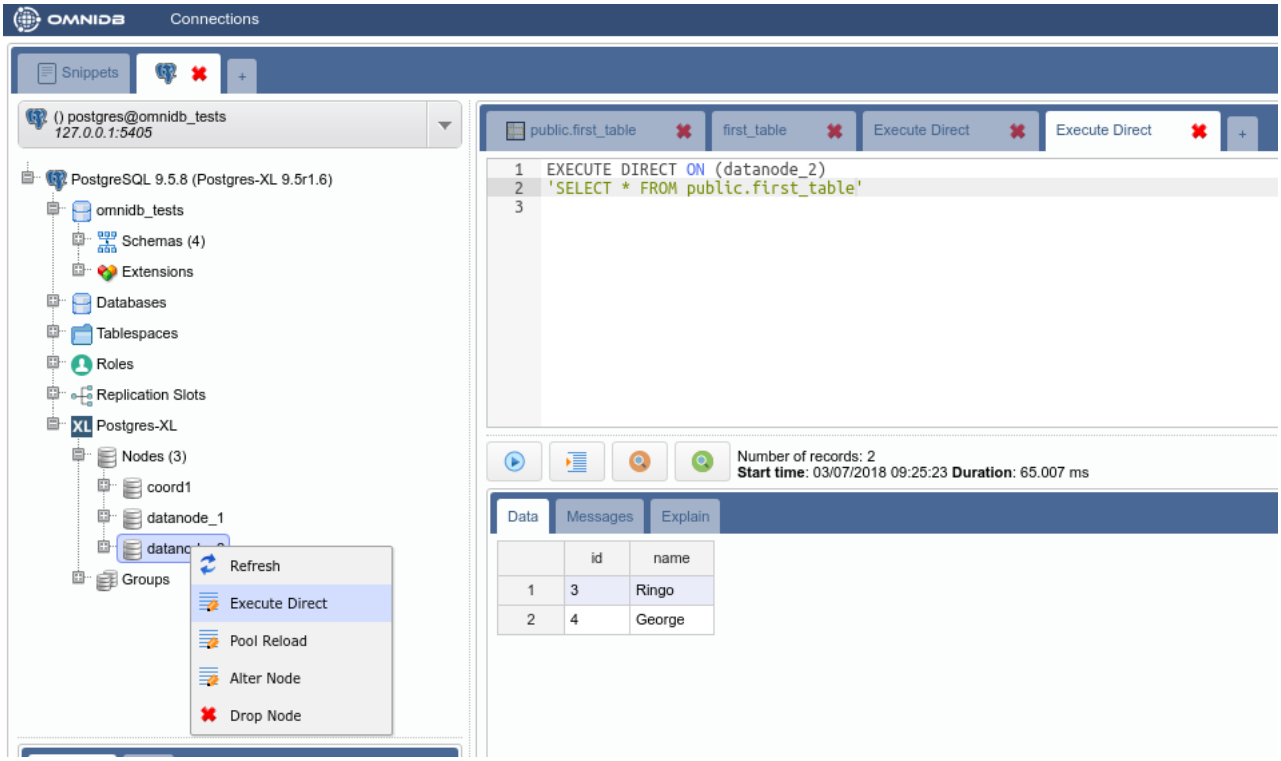


Right click on the table again, *Data Actions*, *Query Data*. You will see that cluster-wide the table has all data inside.



But how the data was distributed in the data nodes? In the *Postgres-XL* main node, right click on each node and click on *Execute Direct*. Adjust the query that will be executed directly into the data node, as you can see below.

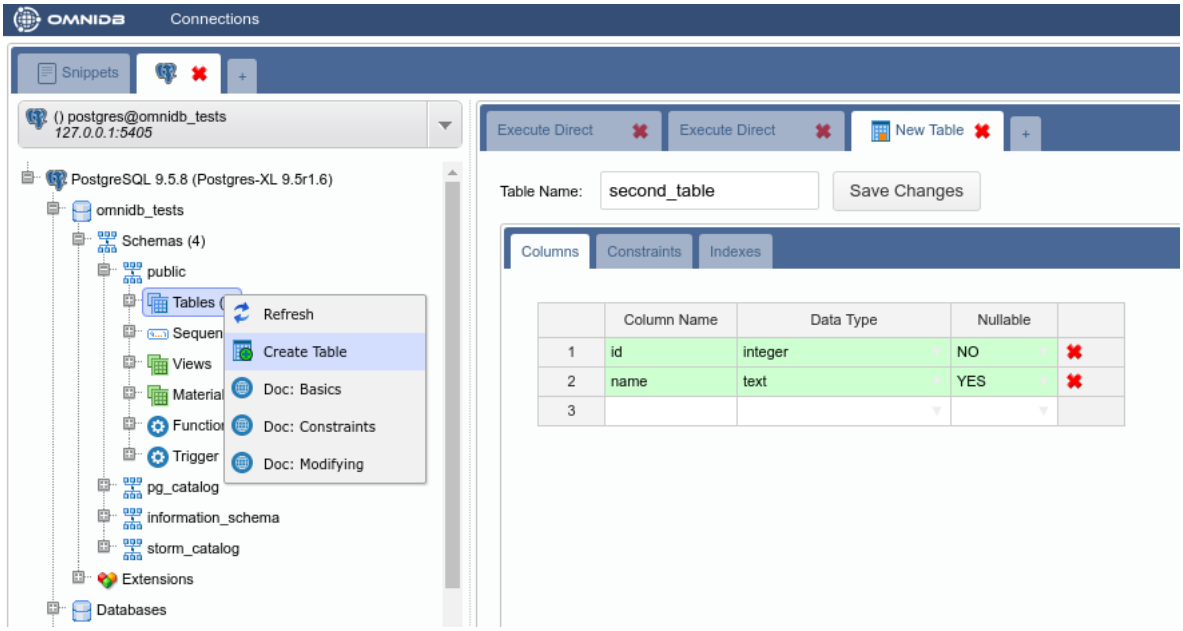


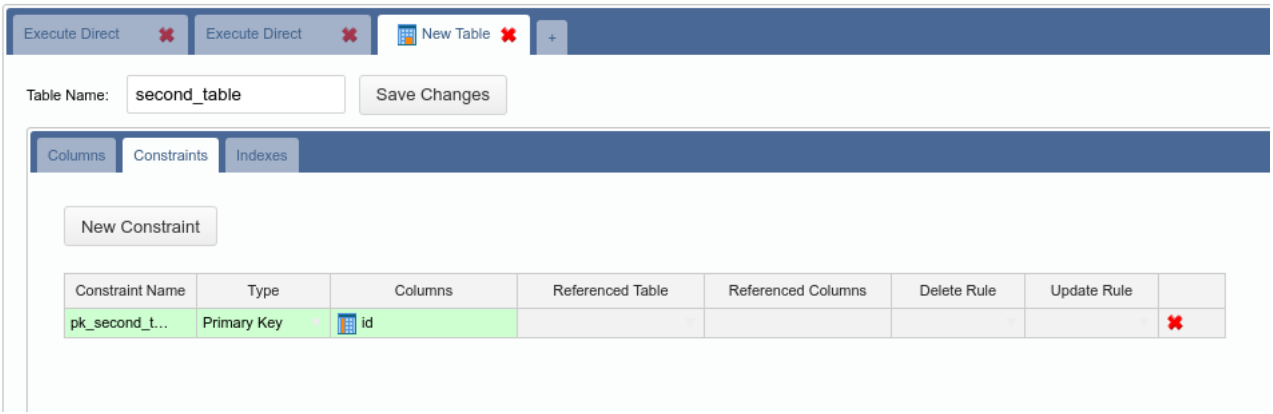


18.0.0.4 Creating a REPLICATION table

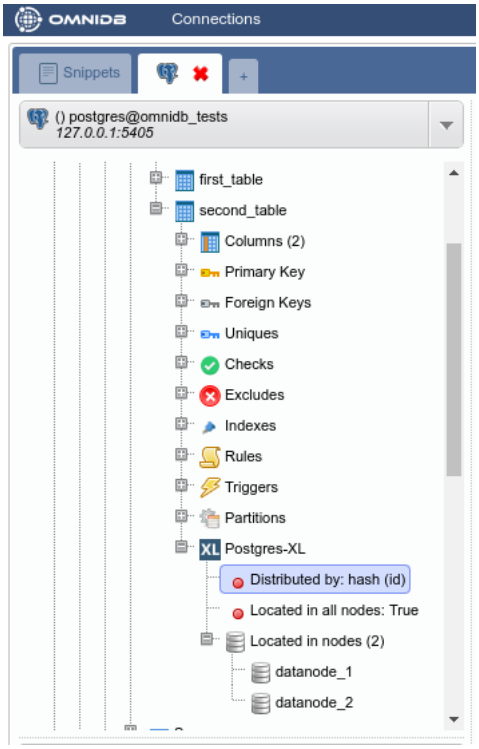
While HASH distribution is great for write-only and write-mainly tables, REPLICATION distribution is great for read-only and read-mainly tables. However, a table distributed by REPLICATION will store all data in all nodes it is located.

In order to create a REPLICATION table, let us create a new table like we did before:

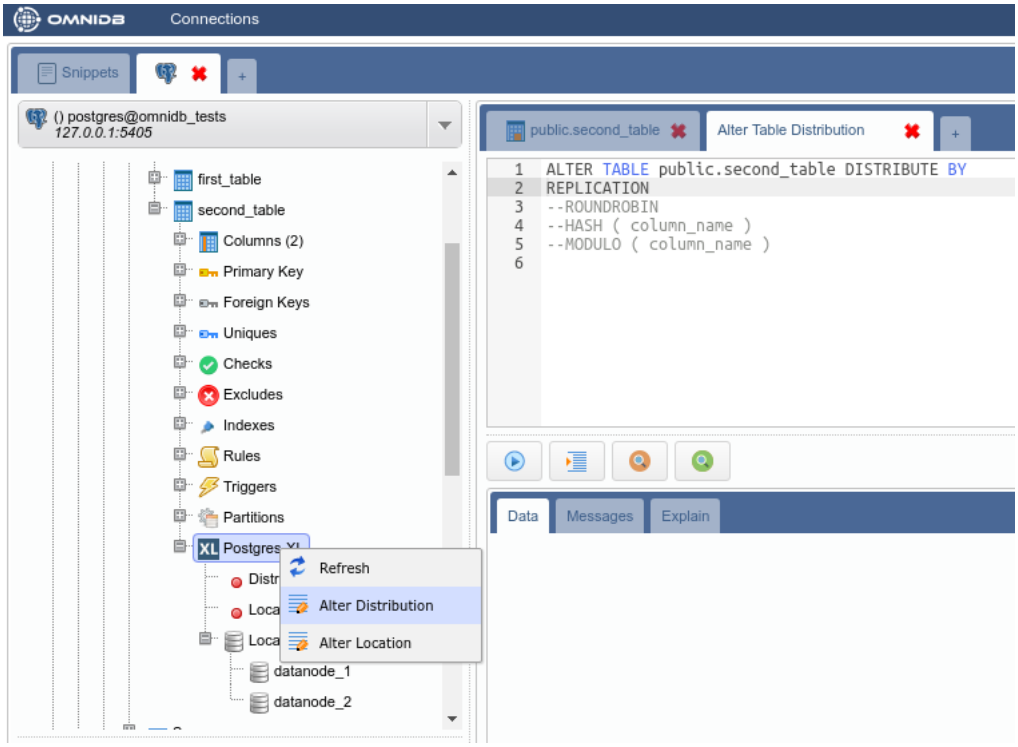




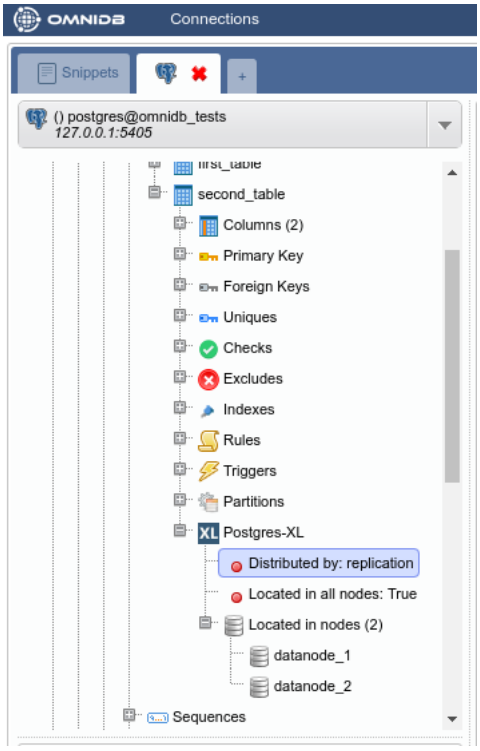
Note how by default it was created as a HASH table:



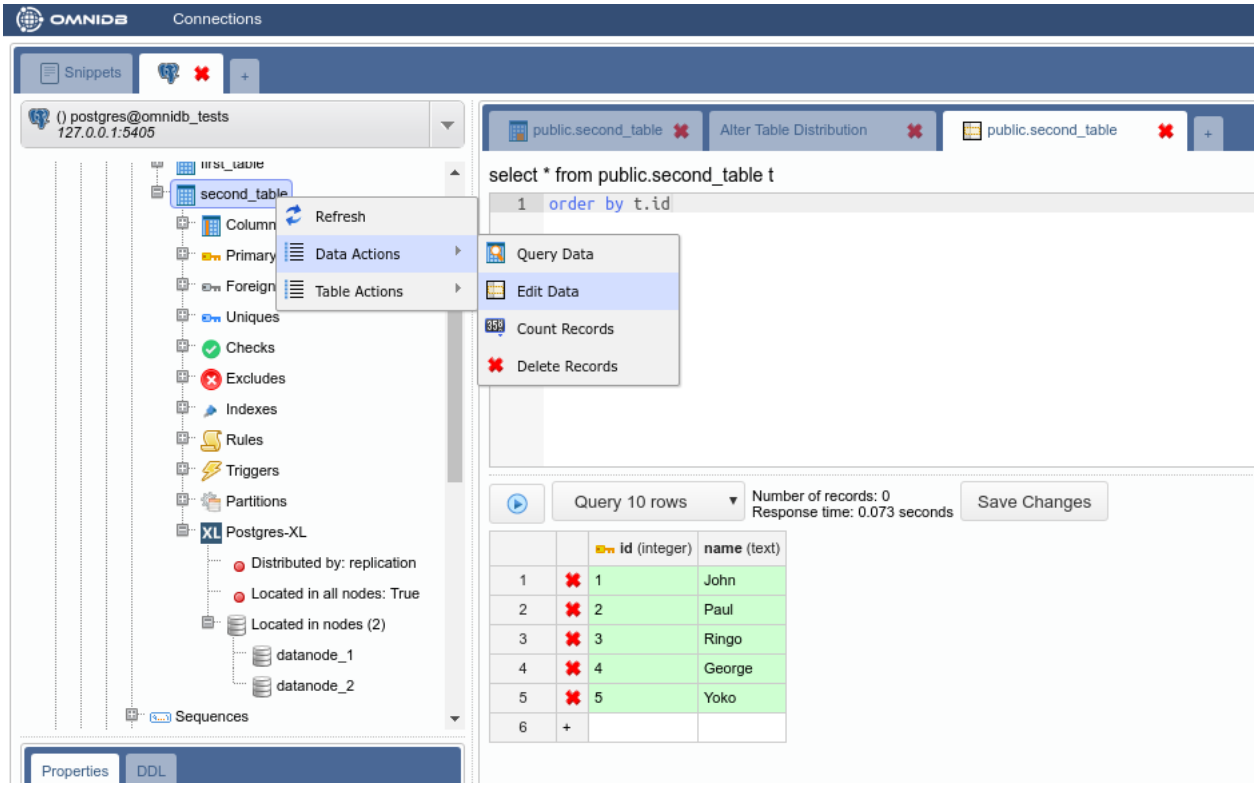
Let us change the distribution type of the table by right-clicking on the *Postgres-XL* node inside the table, and then clicking on *Alter Distribution*. Uncomment the "REPLICATION" line and execute the command:



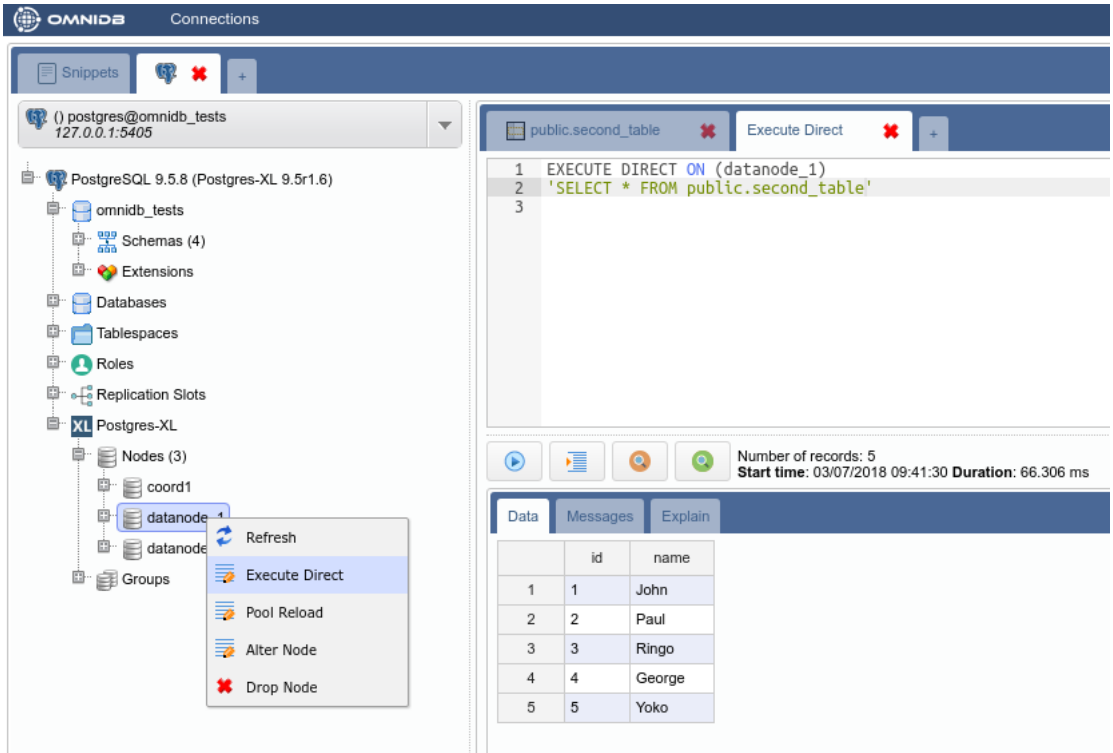
You can check the distribution was successfully changed by right-clicking on the *Postgres-XL* node and clicking on *Refresh*. The properties will now show *Distributed by: replication*.

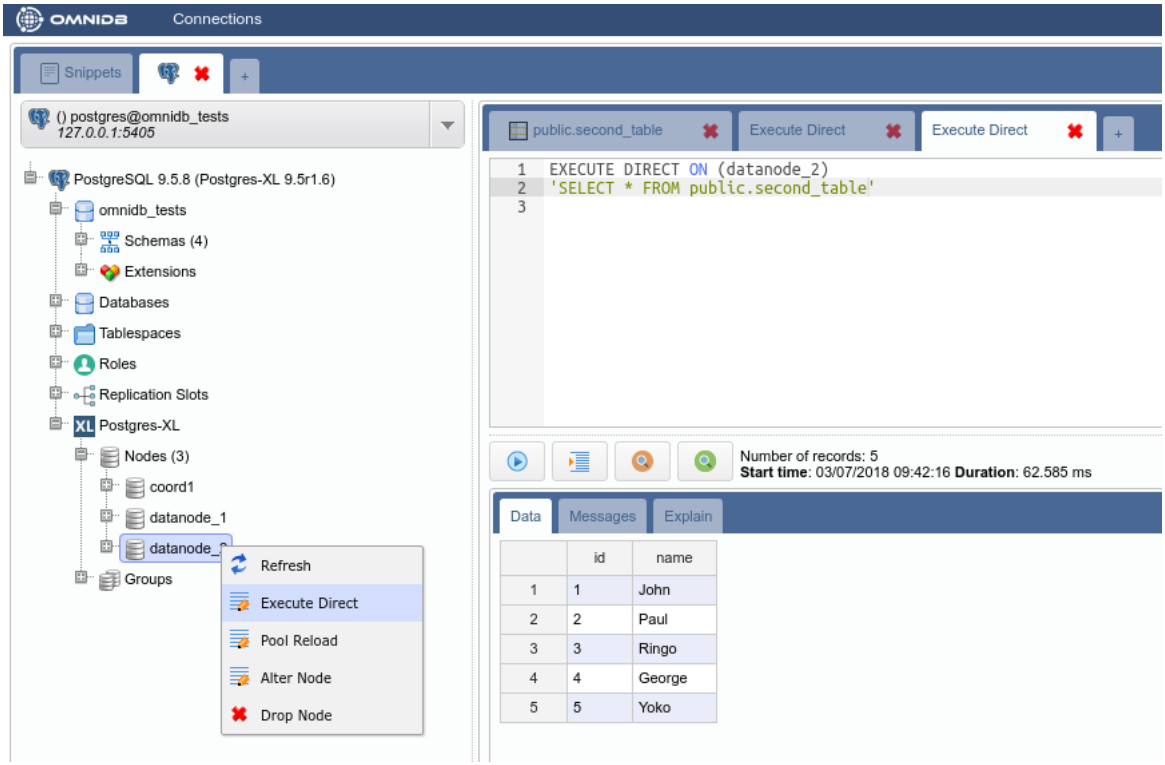


Now add some data to the table:



And then check that all data exist on all data nodes:




OMNIDB
Connections
Snippets
() postgres@omnidb_tests
127.0.0.1:5405
PostgreSQL 9.5.8 (Postgres-XL 9.5r1.6)
omnidb_tests
Schemas (4)
Extensions
Databases
Tablespaces
Roles
Replication Slots
Postgres-XL
Nodes (3)
coord1
datanode_1
datanode_2
Groups
Refresh
Execute Direct
Pool Reload
Alter Node
Drop Node
public.second_table
Execute Direct
Execute Direct
1 EXECUTE DIRECT ON (datanode_2)
2 'SELECT * FROM public.second_table'
3
Number of records: 5
Start time: 03/07/2018 09:42:16 Duration: 62.585 ms
Data
Messages
Explain
id
name
1 1 John
2 2 Paul
3 3 Ringo
4 4 George
5 5 Yoko