Lab 3 Tasks

Task 1: Set up a Database Account

1.1 Connect with the PostgreSQL Server

PostgreSQL system is a Client-Server system, i.e.:

- Many databases, each having multiple tables, reside on a single machine.
- This machine runs a **server** process, which manages all database access.
- Other machines (possibly including the server machine) run client processes, which are sessions connecting the user to a database.

The client processes are run using **pgAdmin4** (https://www.pgadmin.org/): an *open source database front-end* product capable of running on top of the PostgreSQL database system. You can access the database through several tools, notably the *Query Tool* and the *Schema Manager*, as we will find out later in this Lab.

1.2 Login to WVD and Set Up pgAdmin

Log-in using your GUID credentials, i.e., your **GUID email address** as *username* and your **GUID password** as *password*, to the *Remote Desktop Web Client* (WVD) choosing **COSE Desktop**, by visiting the follow link: https://rdweb.wvd.microsoft.com/arm/webclient/index.html
You will see Figure 1 and click on COSE Desktop (*College of Science & Engineering*).

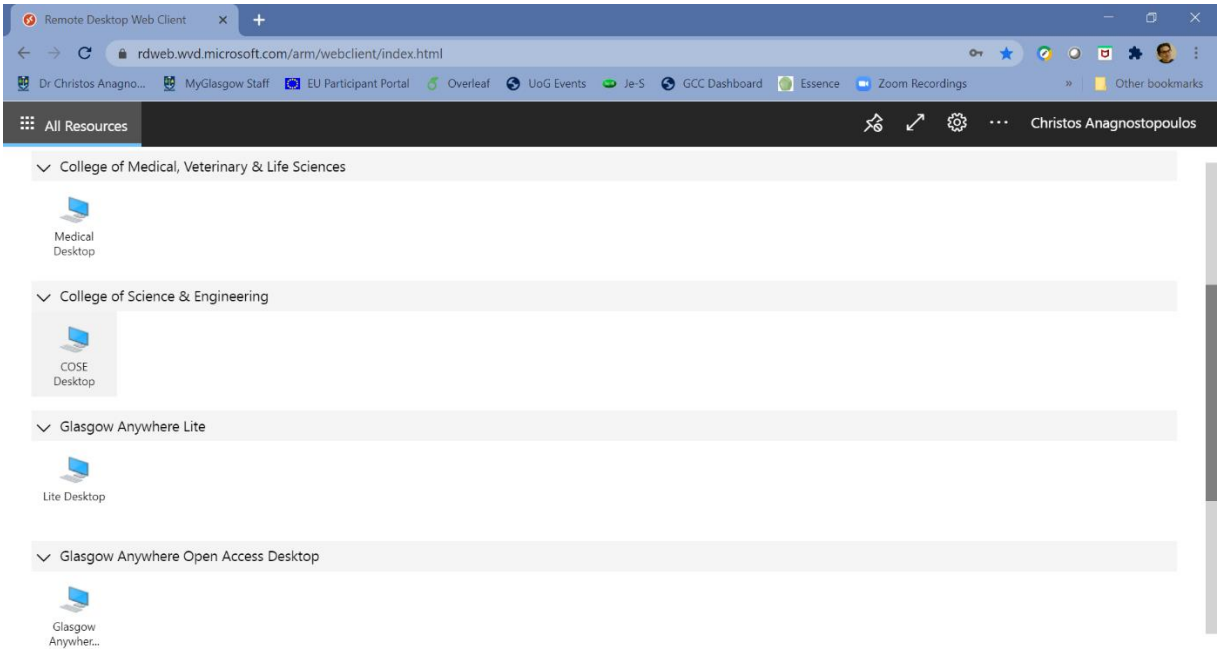


Figure 1

DB(H) Database Systems

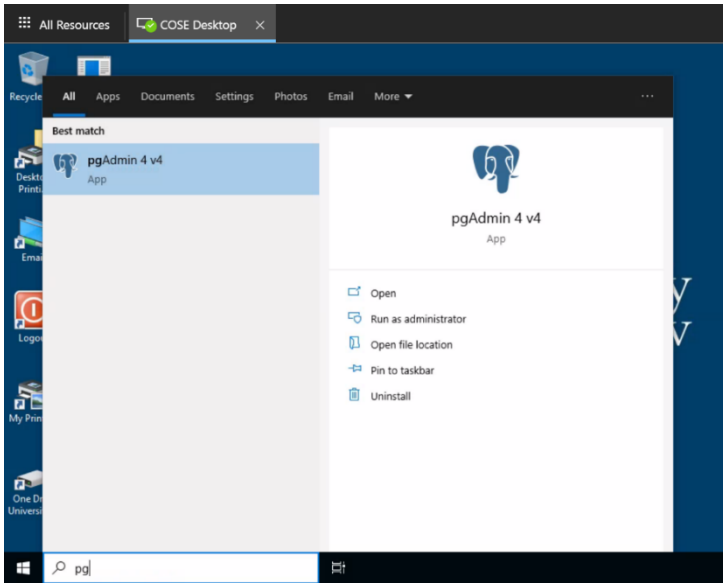


Figure 2

After logging in to your remote desktop, you need to launch **pgAdmin4**, as shown in Figure 2. Type in the following: **pgAdmin4 v4** (the pgAdmin4 version depends on the COSE/Lab PC) and connect to a database server by selecting the toolbar button (top left corner) to *add* a connection to a **server**.

In some implementations of the pgAdmin, you might need to log in to this application with (Figure 3):

Username: postgres
Password: postgres

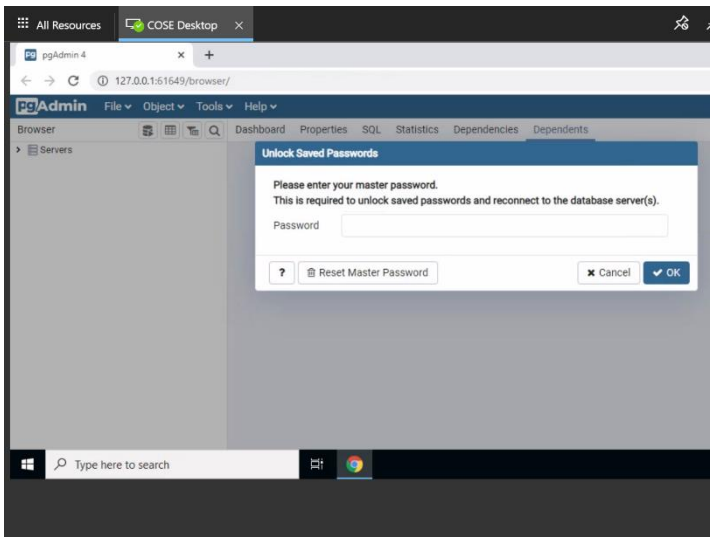


Figure 3

If you are in a Lab PC within the School's network or connected via WVD, then you enter your details to *add* the **socs-db.dcs.gla.ac.uk** PostgreSQL server (see Figure 4). The details for the server connection are (see Figure 5):

Name: socs-db.dcs.gla.ac.uk
Host: socs-db.dcs.gla.ac.uk
Port: 5432
Username: lev3_21_<GUDIusername>
Password: <GUIDusername>

E.g., username: **lev3_21_9991234t** and password is: **9991234t**

all in **lower-case!!**

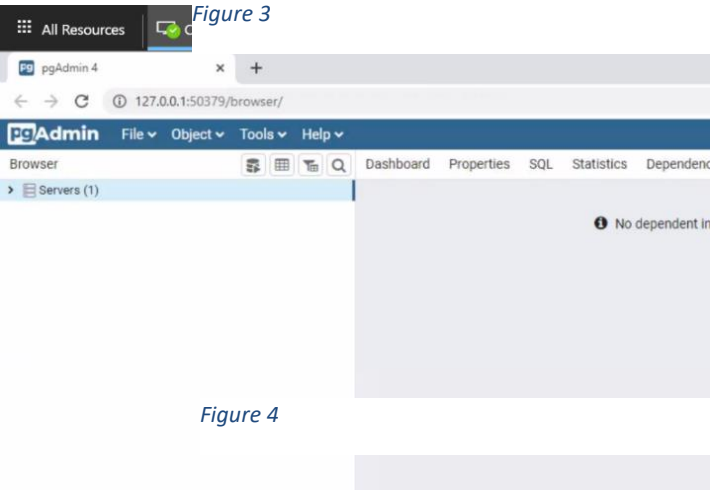


Figure 4

Note: If you have *locally* installed (e.g., in your laptop) the PostgreSQL server, then you add a database server as follows (see Task 1.3 for installations):
Name: localhost
Host: localhost
Port: 5432
Username: postgres
Password: postgres

The local connection to your local PostgreSQL server is advisable when you are not connected in the School's network or to the COSE Desktop.

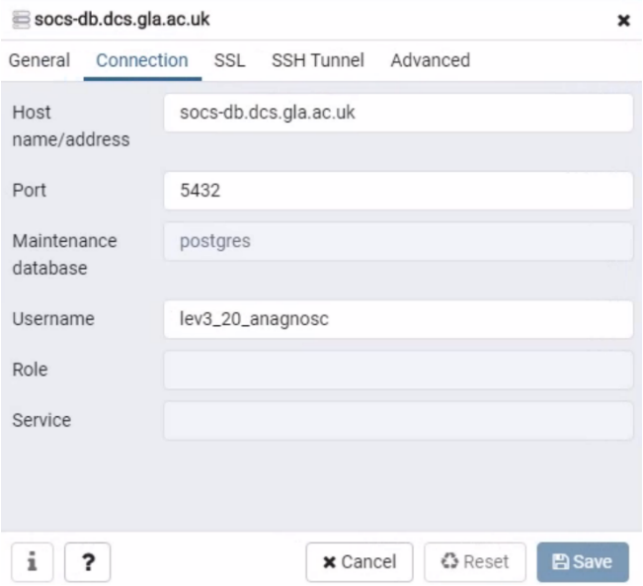


Figure 5

Note: you might change the password **and remember it** by using the Change Password option from the File menu.

You will now see your account added to the local database servers. Once you expand this, you will have access to the list of databases, including one with your login (Figure 6).

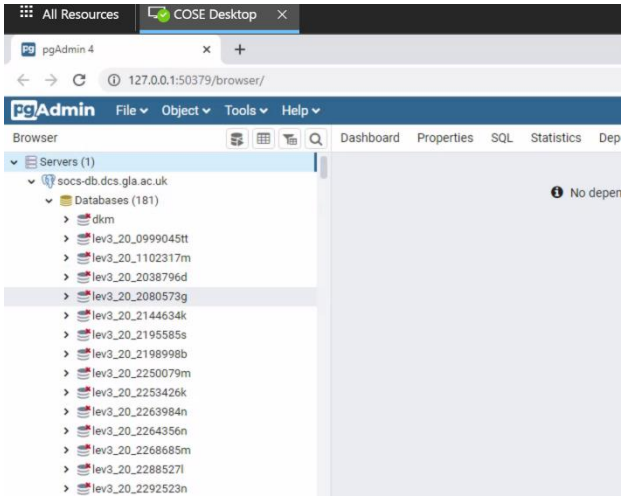


Figure 6

1.3 PostgreSQL & pgAdmin Guide Installation (on your own device only)

Note: This task is provided **if and only if** you desire to use your **own Laptop/PC** to perform locally the exercises. Below, you can find general instructions about installing PostgreSQL on Linux and Windows.

Note: We cannot offer support to students who have difficulties using their own installation.

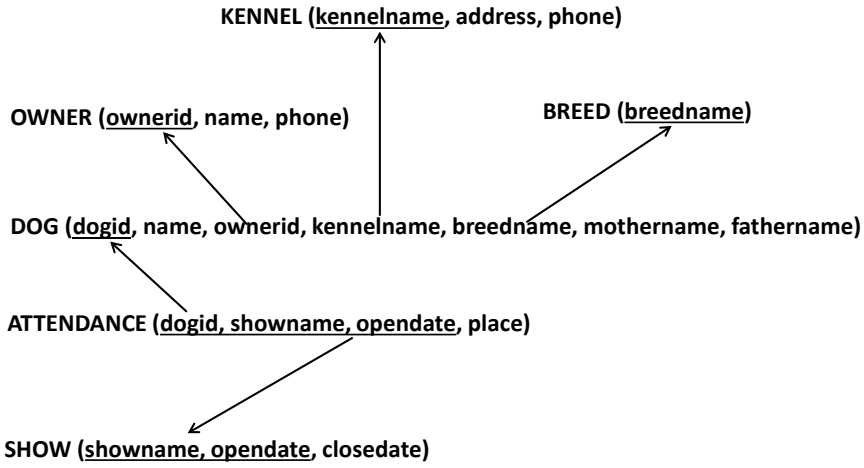
PostgreSQL can be installed in several operating systems. This will show you how to install PostgreSQL on your Ubuntu and Windows. To install PostgreSQL on Ubuntu, first open a command line terminal. Then type the following: **apt-get install postgresql-9.2**

Note: you may need permissions as a root to install. To install PostgreSQL on Windows, there is a graphical installer that includes the PostgreSQL server, pgAdmin and StackBuilder used to download and install additional PostgreSQL applications and drivers. You can download the installation files at: http://www.postgresql.org/download/windows/

To install pgAdmin on Windows, you need to download a graphical installer from https://www.pgadmin.org/download/

Task 2: Define SQL CREATE Statements

Task: Based on the 'Dog' schema, which is shown below, *define* ALL the SQL CREATE TABLE statements, which can also be found in the **DOG_CREATE_SCRIPTS** file in Moodle. The Relational Schema and the attribute specifications for the SQL create statements are provided below. In addition, you *could define* the CONSTRAINTS and PK/FK definitions.



The specifications for the CREATE STATEMENTS are:

- The **Breeds** have only their names stored [VARCHAR(64)]; the breed name is unique.
- The **Owners** have their names [VARCHAR(32)], and contact phone number [VARCHAR(16)], if known; very owner has a unique Integer as identifier.
- The **Kennels** that breed dogs, have their names [VARCHAR(64)], their address [VARCHAR(64)], and their contact phone number [VARCHAR(16)], if known; the kennel name is a unique identifier.
- **Dogs** have their names [VARCHAR(32)], their mother and father names [VARCHAR(64)] if known, and a reference to their owners, breed names and associated kennel. All dog names are unique and every dog has a unique Integer as identifier.
- A **Show** has a show name [VARCHAR(64)], and its opening and closing dates [VARCHAR(12)], if known. The show is identified by a combination of its name and its opening date.
- **Attendance** states whether a particular dog has attended a specific show, identified by show name with the opening date of attendance (VARCHAR(12)), and the rank/place the dog has achieved during the show [Integer], if known.

Task 3: Create a Database in pgAdmin

Step 1: To create your tables, firstly **connect** to the database with the user account details configured during installation. Double click on your server's account icon.

Step 2: Using the SQL editor via **Tools … Query Tool** (Figure 7) from the menu (in pgAdmin 4) you can write your SQL statements, e.g., `CREATE TABLE, SELECT, …`

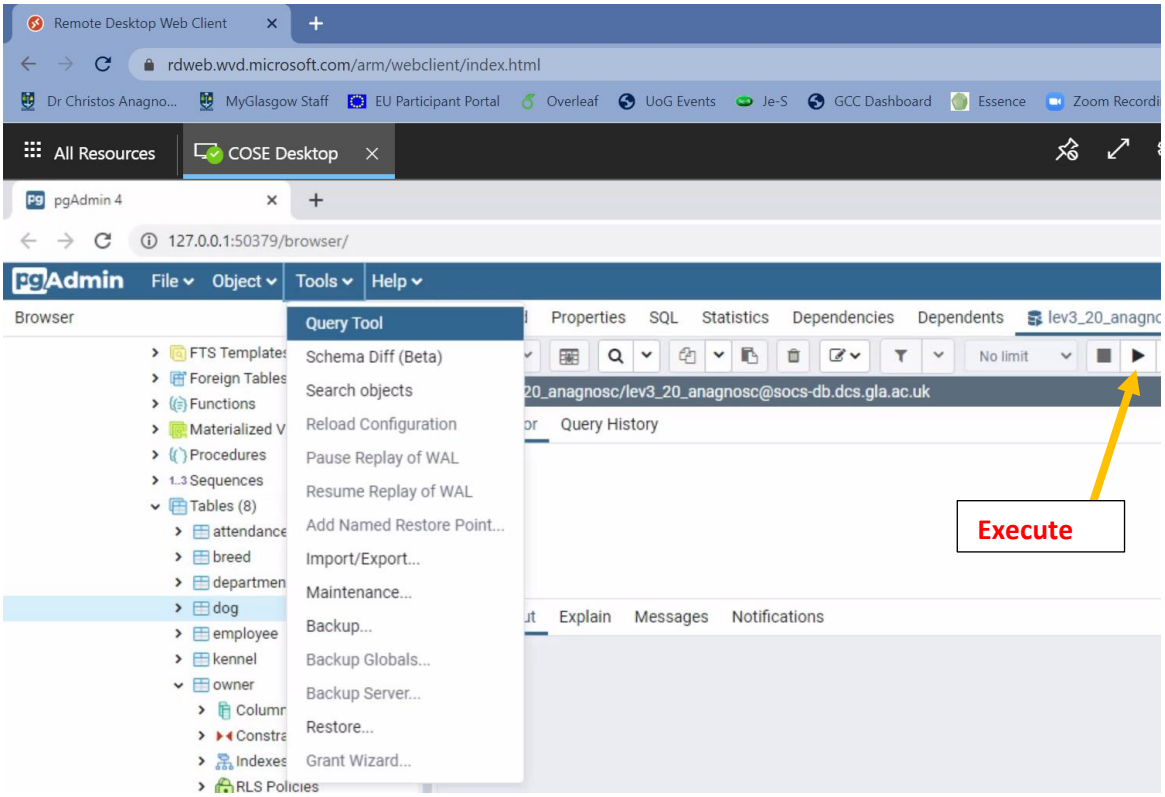


Figure 7

Let's create first the table **OWNER**. In the provided **DOG_CREATE_SCRIPTS** file, **copy** and **paste** in the QueryTool area, as shown below in Figure 8, the CREATE TABLE OWNER statement, and then press the icon 'Execute/Refresh (F5)', which is a small black triangle (see Figure 7).

```
CREATE TABLE owner(
ownerid integer,
name varchar(32),
phone varchar(16),
PRIMARY KEY (ownerid))
```

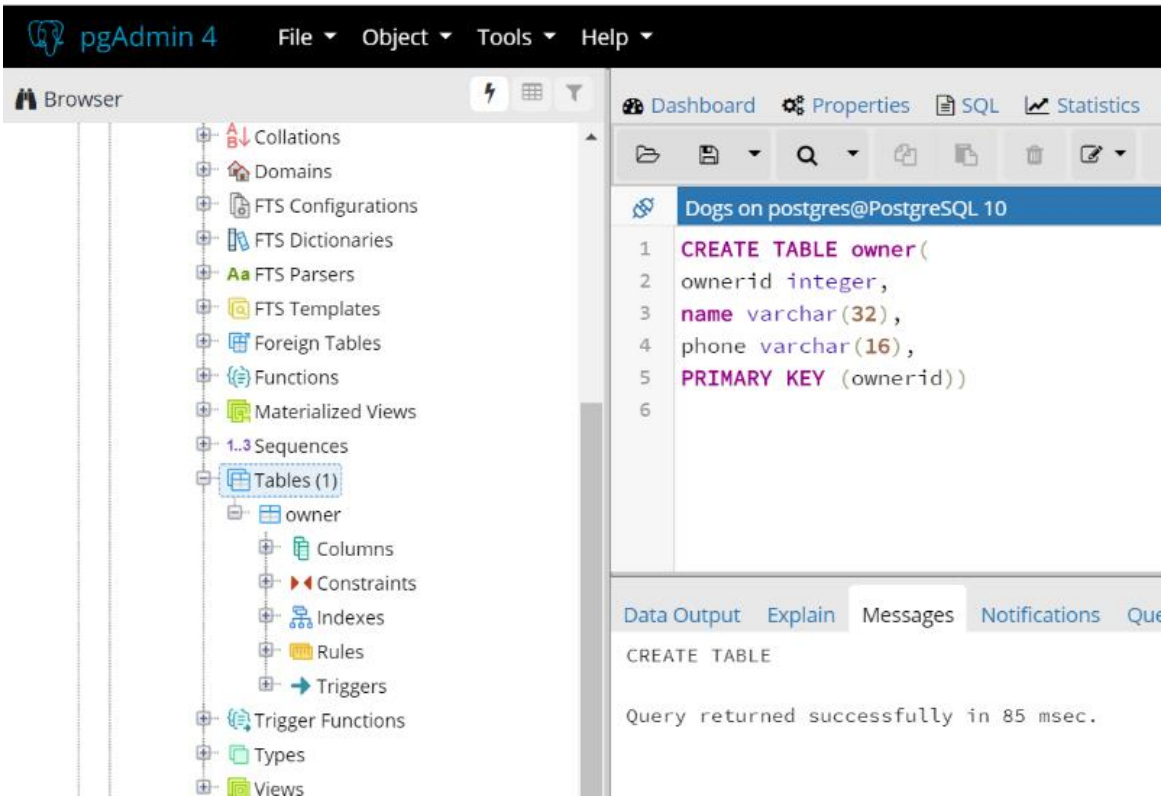


Figure 8

Step 3: The created table(s) can be found under **Schemas -> public -> Tables** (see Figure 8; left). If the tables do not show up, then right click on **Tables** and then **Refresh**.

Create now the tables with *no transitive* dependencies first, i.e., copy and paste the CREATE TABLE statements in the order:

- Table BREED
- Table KENNEL
- Table OWNER (you already have done this, thus, skip this create-statement😊)
- Table DOG
- Table SHOW
- Table ATTENDANCE

Take a break, explore the pgAdmin, look at the definitions of the primary keys and the foreign keys!

Task 4: Populate Tables from a script file

Task: After you have created the six tables from Tasks 2 and 3, you will have to **populate** them. In the file **DOG_INSERT_SCRIPTS** file **i**n Moodle, there are `INSERT INTO` SQL statements that insert tuples to each table. Note that, we will be discussing in a later Lab/Lecture the INSERT modification query. We use it right now, since we need to populate tuples to our tables 😊.

Step 0: Assume that you have e.g., created the table **Owner** using the SQL editor tool. Then, copy the `INSERT INTO OWNER` SQL statements from the script and past to the QueryTool area as shown below in Figure 9. Then, press the 'Execute' icon.

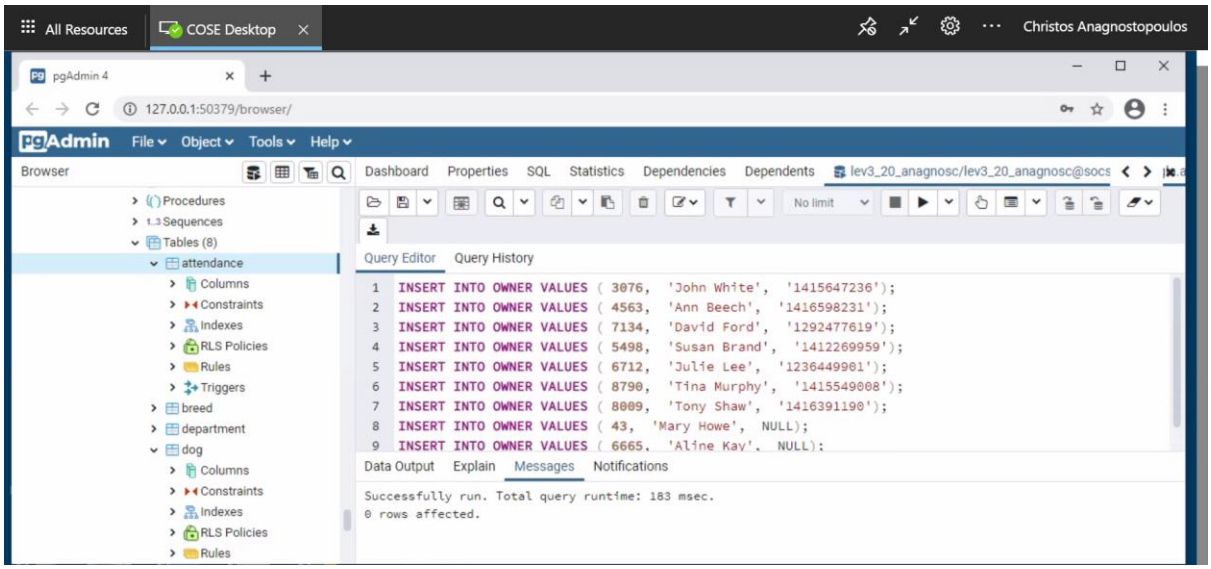


Figure 9

Step 1: Clean the QueryTool area (to avoid re-inserting the same tuples; however, this will not be allowed, since the system will recognize any integrity constraint violations), and type:

```
SELECT * FROM OWNER;
```

to check and view the **tuples** of the Table OWNER, as shown below in Figure 10.

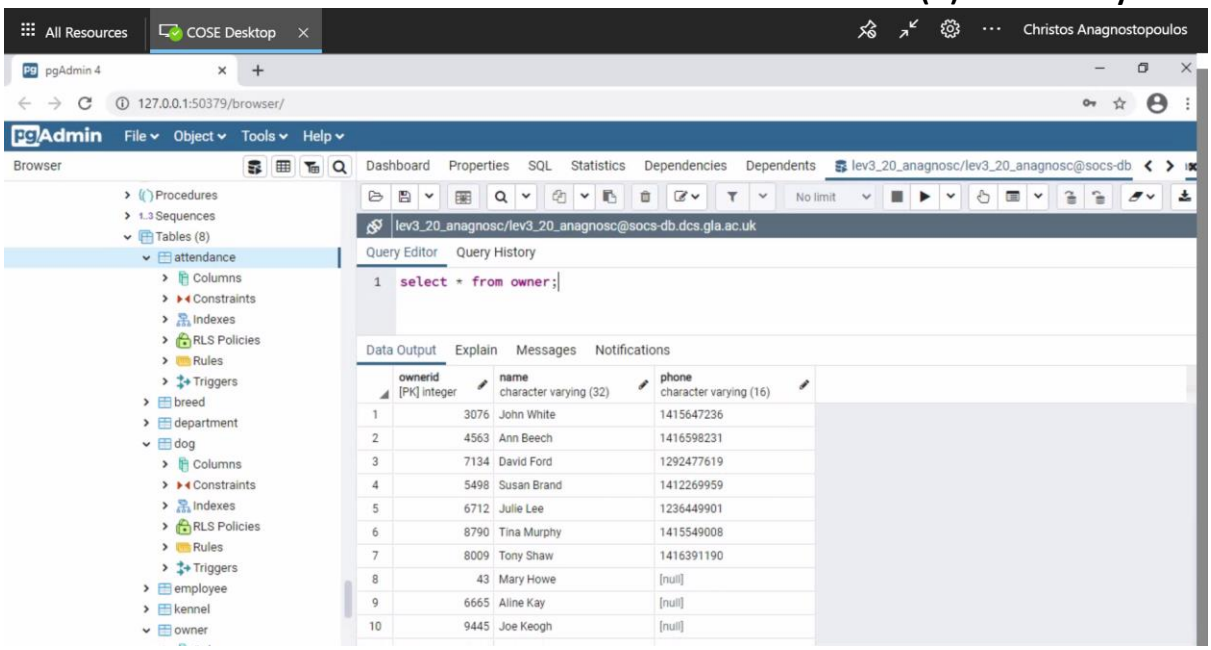


Figure 10

Step 2: Repeat the same for inserting *all* the tuples to *all* the tables in the *same* order as you did with the SQL CREATE statements, i.e., insert tuples to BREED, KENNEL, OWNER (if not yet), DOG, SHOW, and ATTENDANCE.

Every time you insert tuples in a table, *do* check out that the tuples have been successfully inserted. And, here are the SQL SELECT statements for you to check:

```
SELECT * FROM BREED;
SELECT * FROM KENNEL;
SELECT * FROM DOG;
SELECT * FROM SHOW;
SELECT * FROM ATTENDANCE;
```

Important Note: We cannot offer support to students who have difficulties using their own installation for data population.

Task 5: SQL CREATE Statements with Transitive Dependencies by adding CASCADE Triggers [ADVANCED & OPTIONAL]

In this task, we will be discussing step-by-step how we can provide SQL CREATE TABLE statements with relations having *transitive* dependencies. Specifically, let us assume that we would like to create the 'famous' tables:

EMPLOYEE(Name, SSN, SUPER_SSN, DNO)

DEPARTMENT(DNUMBER, DNAME, MGR_SSN)

These two relations are *interrelated* through certain foreign keys:

- DNO (department number) from Employee to Department
- MGR_SSN (manager SSN) from Department to Employee
- SUPER_SSN (supervisor SSN) from Employee (supervisee) to Employee (supervisor).

Due to these interdependencies, we should establish a series of steps to define these tables. This is required since, for instance, during the definition of the SUPER_SSN, the Employee relation should have been created in advance. However, the SUPER_SSN is included in the Employee definition. Hence, we cannot define the SUPER_SSN while we are defining the Employee relation 😊. In this case, we deal with the SQL ALTER TABLE[1] command, where we can add/modify/remove columns and constraints after the definition of a table. This command is used to handle these cases. Let's get starting the following steps/scenario then:

Step 1: Let's define the Employee relation including only these columns that *do not require* the pre-existence of the Employee and the Department relation.

```
CREATE TABLE EMPLOYEE(
     NAME VARCHAR(20) DEFAULT 'CHRIS',
     SSN INT NOT NULL
);
```

Step 2: Now, let's alter the Employee to add the PK constraint (with label: EMP_PK) using the ALTER TABLE command:

```
ALTER TABLE EMPLOYEE
     ADD CONSTRAINT EMP_PK PRIMARY KEY(SSN);
```

Step 3: We can now add the SUPER_SSN attribute as a new column to the Employee relation. Then, we will add a constraint (with label: EMP_FK1) to mention that this attribute is a (recursive) foreign key in the Employee. With these steps, we have now dealt with the recursive foreign key definition. In order to add the DNO attribute to the Employee relation, we need first to create the Department relation.

[1] https://www.postgresql.org/docs/9.1/sql-altertable.html

```
ALTER TABLE EMPLOYEE
     ADD COLUMN SUPER_SSN INT;

ALTER TABLE EMPLOYEE
     ADD CONSTRAINT EMP_FK1
     FOREIGN KEY (SUPER_SSN) REFERENCES EMPLOYEE(SSN);
```

Step 4: Let's create the Department relation. In this definition, we can now add the foreign key MGR_SSN referencing to the already defined Employee relation.

```
CREATE TABLE DEPARTMENT(
DNUMBER INT NOT NULL,
DNAME VARCHAR(20) UNIQUE,
MGR_SSN INT,
CONSTRAINT DEPT_PK PRIMARY KEY (DNUMBER),
CONSTRAINT   DEPT_FK   FOREIGN   KEY   (MGR_SSN)   REFERENCES
EMPLOYEE(SSN));
```

Step 5: Since the Department relation is now created, we go back to the definition of the Employee relation and add the DNO attribute to be a foreign key. Obviously, we need first to define this attribute.

```
ALTER TABLE EMPLOYEE
     ADD COLUMN DNO INT;

ALTER TABLE EMPLOYEE
     ADD CONSTRAINT EMP_FK2
     FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER);
```

Step 6: We are done!....*oops*. We forgot to mention some triggers to the EMP_FK2 constraint. E.g., whenever we change the DNUMBER attributed in the relation Department, then, we would like our Employee tuples to be updated as well to ensure consistency. Hence, we need to update the definition of the EMP_FK2 constraint. In the version of the PostgreSQL, we need to 'drop' the constraint and to add it back with the ON UPDATE CASCADE trigger. That is:

```
ALTER TABLE EMPLOYEE DROP CONSTRAINT EMP_FK2;

ALTER TABLE EMPLOYEE
     ADD CONSTRAINT EMP_FK2
     FOREIGN KEY (DNO) REFERENCES DEPARTMENT(DNUMBER)
     ON UPDATE CASCADE;
```

Step 7: Now, we are all happy! Let's insert some tuples in both relations (using the SQL INSERT, which will be taught in the next lecture(s). However, let's simply experiment with this in order to

demonstrate that after changing the DNUMBER attribute, then the Database Systems automatically updates the associated Employee tuples to ensure consistency based on our EMP_FK2 definition.

Step 8: We insert three employees: Chris, Stella and Philip. Chris does not have a supervisor, while Stella and Philip are supervised by Chris. In this step, the employees are not yet assigned to departments because…obviously, we have not inserted any department yet.

```
INSERT INTO EMPLOYEE VALUES ('CHRIS',1,NULL,NULL);
INSERT INTO EMPLOYEE VALUES ('PHILIP',2,1,NULL);
INSERT INTO EMPLOYEE VALUES ('STELLA',3,1,NULL);
```

Let's see the employees by simply and gently ask:

```
SELECT * FROM EMPLOYEE
```

Step 9: We now add two departments: Research with DNUMBER 1 and Development with DNUMBER 2. Stella (SSN = 3) is the manager of the Research department and Philip (SSN = 2) is the manager of the Development department. That is:

```
INSERT INTO DEPARTMENT VALUES(1,'RESEARCH',3)
INSERT INTO DEPARTMENT VALUES(2,'DEVELOPMENT',2)
```

Step 10: We would like now to assign employees to departments. We need to update their tuples (again the SQL UPDATE will be discussed in the next lecture(s); we use it here to examine the performance of the EMP_FK2 trigger. Specifically, Chris (SSN=1) is working in the department Research (DNO = 1). Philip (SSN = 2) and Stella (SSN = 3) are working in the department Development (DNO = 2).

```
UPDATE EMPLOYEE SET DNO = 1 WHERE SSN = 1;
UPDATE EMPLOYEE SET DNO = 2 WHERE SSN = 2;
UPDATE EMPLOYEE SET DNO = 2 WHERE SSN = 3;
```

Let's see now our database:

```
SELECT * FROM EMPLOYEE
SELECT * FROM DEPARTMENT
```

Step 11: We now examine how the ON UPDATE CASCADE trigger works. In this case, we change the DNUMBER value corresponding to department Research (DNUMBER = 1); the new value is DNUMBER = 100. This is the department where Chris is working. We expect also that Chris' tuple, and specifically, the DNO value with automatically change to 100 in order to avoid inconsistencies. Let's do that:

```
UPDATE DEPARTMENT
SET DNUMBER = 100
WHERE DNUMBER = 1;
```

We now have a look on the Employee relation:

```
SELECT * FROM EMPLOYEE
```

Can you see any updates in Chris' tuple? After updating the Research department tuple, thus update propagates also to the associated employees working in this department (in our case, this is Chris). Thus, now there is no inconsistency.