Assignment 2: Databases

Felipe Orihuela-Espina, Carl Wilding, Pieter Joubert, Daniel Fentham

April 1, 2022

Contents

1 Goals of the assignment

This assignment has been designed to assess your SQL skills on all different stages requested by the module syllabus; 1) database creation and implementation, 2) database population and 3) database querying. Hence, in this assignment, using SQL commands in postgreSQL you will:

1. Create a database and connect to the database
2. Implement the given design of the database
3. Populate the database with the data facilitated in the assignment
4. Perform the requested queries on the database.

To achieve these goals, we have organized the assignment in the form of one exercise per goal. Each exercise is further described below.

2 Preparing your submission

In order to solve the exercises described below, you will have to write several SQL commands. Write all these commands in a single text file with extension .sql. The file should be named:

```
<XXX>_Assignment2.sql
```

where `<XXX>` is the prefix of your student institutional email account at the University of Birmingham.

Submit a single .sql text file with all your commands for all exercises in order. This file will be executed in a 64-bit Debian container with PostgreSQL v13.4 using the `\i` command. For all exercises, assume a user account 'fsad' with password 'fsad2022' permission to create databases already exists. This user also has permissions to read local files which will be important for Exercise 3 (see Sect. 4.3). Your code will be executed and evaluated using this common user account which means that those are the only permissions that "you" will have.

☞ Please note that whether the script run in your machine or not, is irrelevant. The script ought to run in the evaluator's machine which is why we are giving you the exact configuration where you will be tested.

2.1 Style

In the .sql solutions file, clearly separate each exercise with comments e.g.:

```
/* ********************************************************
* Exercise 2. Implement the design of the Smoked Trout database
******************************************************** */
```

Use meaningful names. Use adequate capitalization (of reserved SQL keywords, names in camel case, etc), spacing and comments to make your code readable. Not every attribute has to be NOT NULL. Attributes of type text or varchar do not have to be unnecessarily long. For instance:

Bad style

```
create table x (theattribute int PRIMARY key, z varchar(255) NOT NULL);
Select z from x;
```

Good style

```
CREATE TABLE meaningfulName (
  objectID serial,
  objectName varchar(30),
  PRIMARY KEY (object ID));

-- Retrieve the names of the meaningfulName entities
SELECT objectName
FROM meaningfulName;
```

3 Assessment

The assignment is made of 4 exercises weighted according to Table 1.

Exercise	Weight [%]
Exer. 1 - Create database and connect	10%
Exer. 2 - Implement database	25%
Exer. 3 - Populate the database	25%
Exer. 4 - Query the database	40%

Table 1: Rubric

☞ Beware of cascading errors! In SQL, if one command fails, an error is raised, but the execution of the file is not stop. The failed command can have unexpected consequences on the outcome of subsequent commands. For instance, if the command to populate a table fails, and the table is left empty or with less records than it should, subsequent queries to that table will operate on incorrect information, and hence may produce wrong results.

4 The Scientific Monitoring Key for Taxed Trading Routes (Smoked Trout)

Emperor Knowledgeable VII of the Scientia galaxy is commisionning a new database to control and tax the trade routes on his vast empire to your company SmartStudent Ltd. The database ought to be able to fulfill the following requirements:

- **Trade routes** are composed by a sequential list of ports of call and are identified by a *monitoring key* which is unique. They are operated by a interstellar *shipping company* and assigned a number of space ships to cover it, i.e. *fleet size*. *Taxes* on each route are charged at 12% over revenues of the last fiscal exercise, so each route has also to store the *last exercise revenue* in Experiments, the currency of the Scientifc empire.

- Ports of call are **space stations** located at a certain **longitude** and **latitude** on a certain planet where products are bought and sold.
- **Planets** are located in a *star system* and have a certain *population.* The most common way to refer to a planet is by its *name*, but beware! Different planets may have the same name (usually on different star systems but that is not a hard rule).
- Any planet can have any number of space stations, but they all have at least one.
- Planets are at a certain average **distance** with other planets measured in parsecs. One parsec is equal to about 31 trillion kilometers, which is longer than a light year!. The trade route length is just the sum of the planetary hops among the ports of call in the route visited in due *visit order.* Within planets distances among stations are considered negligible, that is, all stations in a given planet are a distance 0 parsecs from each other.
- There are two types of **products**; **raw materials** and **manufactured goods**. All products have *names*, an *origin* (i.e. planet), occupy a certain *volume per ton* and have a certain *value per ton.* Raw materials are stored in a certain *state of matter* (gas, liquid, solid or plasma), can be either fundamental or composite, and have an associated **extraction date**. Manufactured goods in turn are **made of** a list of other products (whether raw materials or manufactured goods) and have a *manufacturing date* in the Universal Calendar, which oddly very much resembles the Western calendar on Earth.
- Products are extracted or produced in **batches** originating at some planet. There can be repeated batches of the same product, and different batches of the same product may come from different origins. These batches are traded at the space stations. Not all batches may have yet been traded, but every sell is accompanied by a buy. Sells occur at a station of the batch origin i.e. in the same planet that produced it. Buys must occur at a station different of the selling station (whether in the same planet or other).

Your database architect comes up with the conceptual design in Figure 1. You might need to make adaptations to the conceptual design in order to implement an adequate physical design.

Table 2 is a list of the star systems ruled by the Scientific Empire, the inhabited planets and the space stations. In the assignment page, further find the data files with information about the traded products, space stations locations, trading routes, the batches, and the trading operations.

Star System	**Planet**	**Space Stations**
Mathematics	Algebra (p. 1758)	Gauss, and Cantor
	Geometry I (p. 348)	Euclides, Hypathia, Fermat and Descartes
	Geometry II (p. 586)	Riemann and Euler
	Calculus (p. 1396)	Newton, and Leibnitz
	Statistics (p. 2685)	Pearson, Student and Box
	Logic (p. 224)	Dedekind
Computer Science	Algebra (p. 639)	Boole
	Algorithmics (p. 1214)	Al-Khwarizmi, Turing and Lovelace
	Statistics (p. 996)	Kolmogorov
Physics	Electromagnetism (p. 1562)	Maxwell and Boltzmann
	Thermodynamics (p. 907)	Sadi-Carnot
	Relativity (p. 567)	Einstein
Chemistry	Organic Chemistry (p. 331)	Woodward
	Inorganic Chemistry (p. 800)	Lavoisier and Curie
Phylosophy	Logic (p. 76)	Aristotle and Kant

Table 2: Star systems, planets and space stations of the Scientific Empire. The number in brackets represent the population in million of inhabitants.

4.1 Exercise 1: Create database and connect

During your testing it may be convenient to start your .sql command file with something like:

```
\cd '<YOUR\_PATH\_HERE>'
  -- This path may be different in your case...
\connect postgres;
```

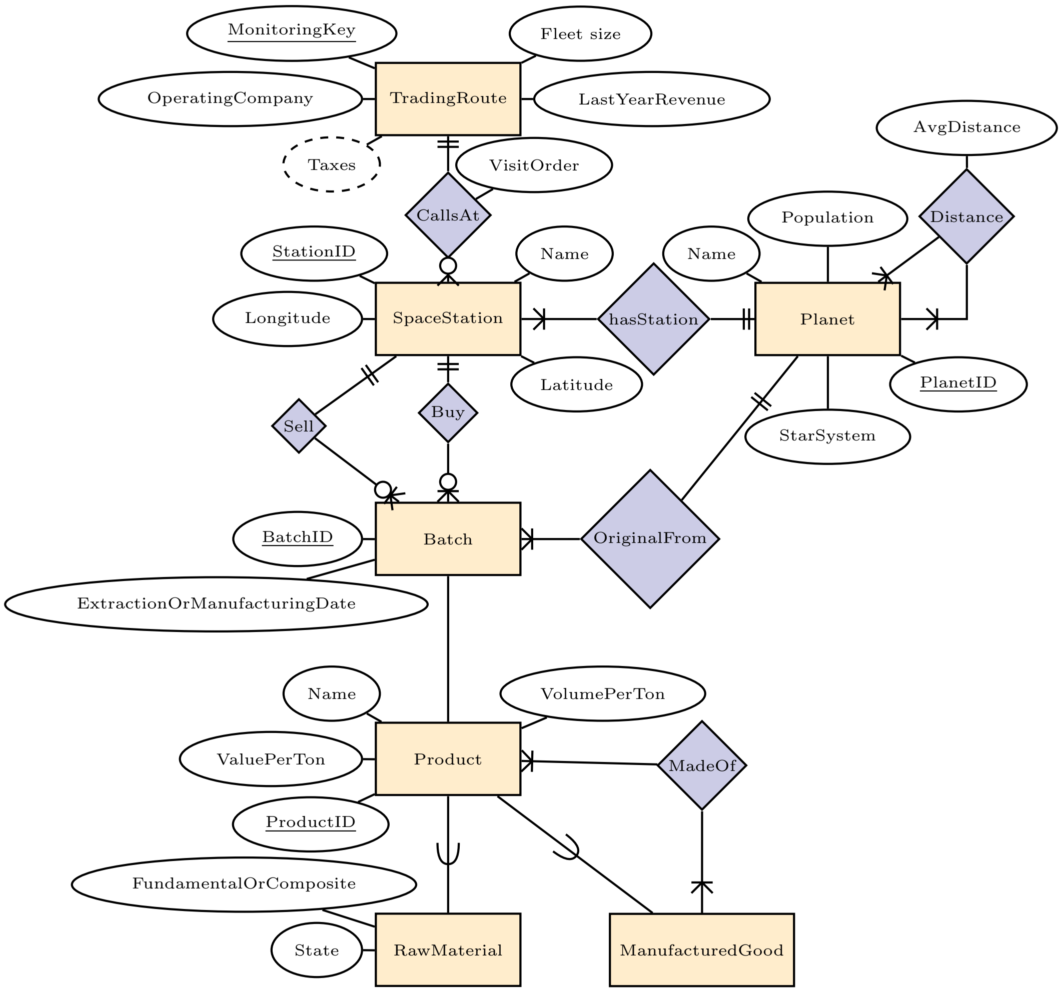

Figure 1: Conceptual ER model of Smoked Trout.

```
DROP DATABASE IF EXISTS "smokedTrout";
```

Thus ensuring you start from a "clean" database each test and you can use relative paths within the command files.

Steps to complete the exercise:

1. Create a database called `SmokedTrout`.
2. Connect to the database

4.2 Exercise 2: Implement the database

Follow the design in Figure 1. Make sure the attributes have the correct names. Before creating the tables, declare the new types that you will need. Having enum types reduces memory waste and chance of making errors.

Then, continue with creating the tables infrastructure. Note that because of foreign keys and inheritances, some tables ought to be created in a certain order. In this exercise we shall fold the relations 1:N relation to the N side. However, the 0:N relations will be implemented in separated tables to avoid the presence of many NULL values (a good design criteria). Note that this is an implementation criterion that we are making beyond the mere conceptual design that we were given. It goes without saying that this would have been the only possible solution, but it is the one that we shall take for this exercise.

You will have to choose the adequate types for your attributes. Have a look at the data provided to get an idea of what to expect for each attribute.

Steps to complete the exercise:

1. Create a new `ENUM` type called `materialState` for storing the raw material state; *Solid*, *Liquid*, *Gas*, *Plasma*.
2. Create a new `ENUM` type called `materialComposition` for storing whether a material is *Fundamental* or *Composite*.
3. Create the table `TradingRoute` with the corresponding attributes.
4. Create the table `Planet` with the corresponding attributes.
5. Create the table `SpaceStation` with the corresponding attributes.
6. Create the *parent* table `Product` with the corresponding attributes.
7. Create the *child* table `RawMaterial` with the corresponding attributes.
8. Create the *child* table `ManufacturedGood`. Note that in principle this table has no additional attributes, yet it is needed to make the proper links in the table `MadeOf` that follows.
9. Create the table `MadeOf` with the corresponding attributes.

☞ As we explained in class, the implementation of the inheritance provided by most DBMS are not (mathematically) strictly correct, which has important consequences. Here is an excellent example; Ideally, attributes linking the table MadeOf to the ManufacturedGood and Product respectively would be foreign keys pointing to the corresponding product. HOWEVER, one of the known caveats of postgreSQL implementation of inheretance is precisely that you can either use foreign keys, or table inheritance, but not both.
`https://www.postgresql.org/docs/current/ddl-inherit.html#DDL-INHERIT-CAVEATS`
So, for this exercise, keep the inheritance and do not declare the foreign key restriction. It goes without saying this open the door to potential incoherences in the database. For this exercise, do not worry, as the data has been curated for you, but in general beware of this limitation!

10. Create the table `Batch` with the corresponding attributes.
11. Create the table `Sells` with the corresponding attributes.
12. Create the table `Buys` with the corresponding attributes.
13. Create the table `CallsAt` with the corresponding attributes.
14. Create the table `Distance` with the corresponding attributes.

4.3 Exercise 3: Populate the database

Now it is time to populate the database with data. Together with this assignment you are given a set of files that contains all the data that you will be using for the assignment. You do not need to create additional data.

Make sure that all paths used in this exercise are relative e.g. ‘`./data`’ rather than absolute. This is because the file tree to your working directory in your machine will be different from that of your evaluator.

☞ When importing, do use the `\copy` command rather than the SQL instruction `COPY`. This is because the `COPY` statement is reserved for admins.

You will notice that the names of the columns in the csv files have different names than the attribute names in the design. This is intentional. Given the different names, in order to import data from one of the *.csv* files into one of the database tables, one possible solution is to create an intermediate `Dummy` table. The dummy table will have the attributes with the names equal to those in the *.csv* file. Then, you can use an `INSERT INTO ... SELECT ... FROM Dummy;` command to import the information. If you opt for this solution, make sure you dispose of the `Dummy` table afterwards using `DROP TABLE Dummy;`. Repeat the trick as many times as needed to populate all tables.

☞ You may be tempted to rename the files or perhaps change the names of some of the columns in the original files to facilitate your importing with the `\copy` command. Don't! First, the exercise is intentionally designed to make you think like you would not have writing permissions over the data files i.e. *as if* you could not change the data files content. And second (and more pragmatical for you here), you will be evaluated with your SQL being executed reading the data files in your evaluator's computer and these will have the given column headers; not yours!

Steps to complete the exercise:

1. Unzip all the data files in a subfolder called `data` from where you have your code file `*.sql`, e.g.

```
<AssignmentFolder>/
    |- <XXX>_Assignment2.sql
    |- data/
        |- Planets.csv
        |- ...
```

2. Populate the table `TradingRoute` with the data in the file *TradeRoutes.csv*.

```
CREATE TABLE Dummy (
  MonitoringKey SERIAL,
  FleetSize int,
  OperatingCompany varchar(40),
  LastYearRevenue real NOT NULL);
-- This table has the same headers that the file
-- Note that there is no need to declare a primary key for the dummy table

\copy Dummy FROM './data/TradeRoutes.csv' WITH (FORMAT CSV, HEADER);

INSERT INTO TradingRoute (MonitoringKey,OperatingCompany,
        FleetSize,LastYearRevenue)
  SELECT MonitoringKey,OperatingCompany,
        FleetSize,LastYearRevenue FROM Dummy;

DROP TABLE Dummy;
```

3. Populate the table `Planet` with the data in the file *Planets.csv*.
4. Populate the table `SpaceStation` with the data in the file *SpaceStations.csv*.

5. Populate the tables `RawMaterial` and `Product` with the data in the file *Products_Raw.csv*. Note that no particular effort is needed to separate the info for the parent table. You can proceed *as if* you were only populating the child table.
6. Populate the tables `ManufacturedGood` and `Product` with the data in the file *Products_Manufactured.csv*.
7. Populate the table `MadeOf` with the data in the file *MadeOf.csv*.
8. Populate the table `Batch` with the data in the file *Batches.csv*.
9. Populate the table `Sells` with the data in the file *Sells.csv*.
10. Populate the table `Buys` with the data in the file *Buys.csv*.
11. Populate the table `CallsAt` with the data in the file *CallsAt.csv*.
12. Populate the table `Distance` with the data in the file *PlanetDistances.csv*.

4.4 Exercise 4: Query the database

Resolve the following queries:

1. Report last year taxes per company
   - Calculate the taxes as derived information from last year revenues and add the taxes across the different trading routes. Then, report each operating company and its total taxes.
2. What's the longest trading route in parsecs?
   - Retrieve the longest route monitoringKey and its total length in parsecs.

☞ Return EXACTLY what you are being requested. Do not return additional information.

Steps to complete the exercise:

- Query 1: Report last year taxes per company
  1. Add an attribute `Taxes` to table `TradingRoute`
  2. Set the derived attribute taxes as 12% of `LastYearRevenue`
  3. Report the operating company and the sum of its taxes group by company.

- Query 2: What's the longest trading route in parsecs?
  1. Create a dummy table `RouteLength` to store the trading route and their lengths.
  2. Create a view `EnrichedCallsAt` that brings together trading route, space stations and planets. You can use an `INNER JOIN` with `CallsAt` and `SpaceStation`.
  3. Add the support to execute an anonymous code block as follows;

```
DO
$$
DECLARE
BEGIN
END;
$$;
```

     All the rest of the steps of this exercise except for the last one will now occur within this `DO` block.
  4. Within the declare section, declare a variable of type *real* to store a route total distance.

```
routeDistance real := 0.0; -- Trading route total distance
```

  5. Within the declare section, declare a variable of type *real* to store a hop partial distance.
  6. Within the declare section, declare a variable of type *record* to iterate over routes.
  7. Within the declare section, declare a variable of type *record* to iterate over hops.
  8. Within the declare section, declare a variable of type *text* to transiently build dynamic queries.
  9. Within the main body section, loop over routes in `TradingRoutes`, e.g.

```
FOR rRoute IN SELECT MonitoringKey FROM TradingRoute
LOOP
END LOOP
```

10. Within the loop over routes, get all visited planets (in order) by this trading route. You can achieve this using a dynamic view `PortsOfCall`. To create a dynamic view, use the variable of type text to create a string that contains the `CREATE VIEW` command, and for which its `WHERE` clause will be dependent on the route monitoring key.

```
query := 'CREATE␣VIEW␣PortsOfCall␣AS␣'
                    || 'SELECT␣Planet,␣VisitOrder␣'
                    || 'FROM␣EnrichedCallsAt␣'
                    || 'WHERE␣MonitoringKey␣=␣' || rRoute.MonitoringKey
                    || '␣ORDER␣BY␣VisitOrder';
```

11. Within the loop over routes, execute the dynamic view using command `EXECUTE`

```
EXECUTE query;
```

12. Within the loop over routes, create a view `Hops` for storing the hops of that route. One way of doing this is by `INNER JOIN`ing the view created in Step 10 with itself `ON` the visit order being consecutive.
13. Within the loop over routes, initialize the route total distance to 0.0.
14. Within the loop over routes, create an inner loop over the hops
15. Within the loop over hops, get the partial distances of the hop. You can achieve this using a dynamic query over table `Distance`, and for which its `WHERE` clause will be dependent on the hop origin and destination planets.
16. Within the loop over hops, execute the dynamic view using command `EXECUTE` and store the outcome `INTO` the hop partial distance.
17. Within the loop over hops, accumulate the hop partial distance to the route total distance.
18. Go back to the routes loop and insert into the dummy table `RouteLength` the pair *(RouteMonitoringKey,Rout*
19. Within the loop over routes, drop the view for `Hops` (and cascade to delete dependent objects).
20. Within the loop over routes, drop the view for `PortsOfCall` (and cascade to delete dependent objects).
21. Finally, just report the longest route in the dummy table `RouteLength`.