case_12.2

May 1, 2020

1 How can we build a company database to handle product sales end-to-end?

1.1 Introduction (5 mts)

Business Context. You are a data analyst for the same large financial services firm as in the previous case. The firm was pleased with your analysis and now the see the value of having databases that can easily be queried using SQL. It would therefore like to move its data, which is currently stored as CSV files, onto a proper database.

Business Problem. The business would like you to create a database to house its existing data and add a few more data tables to track additional information they are interested in.

Analytical Context. The data is split across three tables: "Agents", "Calls", and "Customers", which sit on CSV files. We will be creating a database and then loading the data from these CSV files into that database.

The case is sequenced as follows: you will (1) learn the fundamentals of database management systems; (2) install and set up PostgreSQL, a popular open-source database management system, on your local machine; (3) use SQL to set up a new database with tables in PostgreSQL; (4) query our newly created database to answer business questions; and finally (5) design enhancements to our database to fit expanded business requirements.

1.2 Database management systems (10 mts)

So far, you have been using SQL via `SQLAlchemy` to interact with databases. But databases by themselves are quite useless when there is more than one person involved in writing data to it. Rather, we use **database management systems** in order to manage databases properly. All the examples mentioned in the previous case (SQL Server, Oracle, PostgreSQL) are in fact database management systems (going forward, let's refer to them as DBMS) and not databases (a.k.a. DB).

But why did these DBMS come into existence? Why wasn't a DB enough? To answer that, imagine our previous example where we mentioned the phone book record, where you keep track of all your friends and their phone numbers, but now on a text file on your computer. So you have the following file:

That looks fine and it works as you'd expect. But now imagine you upload a YouTube video of yourself dancing to a new song and, due to your amazing dancing moves, that video goes viral. As a result, you become the most popular person in your neighborhood, and everybody wants to be

your friend. Since you want to keep track of all of your new friends' phone numbers, you decide to add all these new contacts to your "phones.txt" file. But given the number of new friends you are making, you ask your sister to help you with adding your new friends to the list.

Being a tech-savvy guy, you share the folder where the file is located on your home network and create a shortcut to it on your sister's laptop. She opens the file and you both start adding new contacts to the file at the same time. Each of you individually save the file when you are finished and go to sleep. But when you wake up the next morning, you notice that half of all the contacts are gone!

1.2.1 Exercise 1: (5 mts)

What do you think went wrong here?

Answer. Although both save operations worked properly and were actually saved to the database, the first person to save the file later had their version overwritten by the second save request, using only the data inputted by the second person. Preventing this (and other even more complex and dangerous scenarios) is the reason why DBMS were born.

1.3 Setting up PostgreSQL (10 mts)

In this case, we are going to use PostgreSQL because it is easy to install and because it is open-source. As you become more proficient with SQL, you can try the non-commercial version of SQL Server as it offers a lot of advanced features as well as an excellent tool to manage the database (SQL Server Management Studio), and furthermore it is very likely that your organization has a SQL Server database laying around. In later cases, we will also use Amazon RDS in order to work with databases in-cloud.

PostgreSQL was originally developed for UNIX-like platforms, but it was also designed to be portable. This means that PostgreSQL can also run on other platforms such as Mac OS X, Solaris, and Windows. To download PostgreSQL, first go to the download page of PostgreSQL installers: https://www.enterprisedb.com/downloads/postgres-postgresql-downloads, and select your installer version. Once the installer is downloaded, follow these steps:

1. Double click on the installer file. An installation wizard will appear and guide you through multiple steps where you can choose different options that you would like to have in PostgreSQL

```
<td class="second" width="60%"><div align="left">2. Click the "Next" button</div></td>
<td class="second"><img src="images/pg2.png" width="400"></td>

<td class="second" width="60%"><div align="left">3. Specify the installation folder; choose you
<td class="second"><img src="images/pg3.png" width="400"></td>

<td class="second" width="60%"><div align="left">4. Select components to install and click the
<td class="second"><img src="images/pg4.png" width="400"></td>

<td class="second" width="60%"><div align="left">5. Select the database directory to store the
<td class="second"><img src="images/pg5.png" width="400"></td>
```

```
<td class="second" width="60%"><div align="left">6. Enter the password for the database superus
<td class="second"><img src="images/pg6.png" width="400"></td>

<td class="second" width="60%"><div align="left">7. Enter the port for PostgreSQL. Make sure tl
<td class="second"><img src="images/pg7.png" width="400"></td>

<td class="second" width="60%"><div align="left">8. Choose the default locale used by the datal
<td class="second"><img src="images/pg8.png" width="400"></td>

<td class="second" width="60%"><div align="left">9. You are now ready to install PostgreSQL! Cl
<td class="second"><img src="images/pg9.png" width="400"></td>

<td class="second" width="60%"><div align="left">10. Click the "Finish" button to complete the
<td class="second"><img src="images/pg10.png" width="400"></td>
```

1.4 Setting up and querying our tables (15 mts)

Now we can begin working in PostgreSQL. The first thing you will need is an IDE to run your queries. We'll use "pgAdmin 4", which has already been installed for you as part of the main installation. In case you installed a version of PostgreSQL that doesn't contain pgAdmin, you can download it from this link: https://www.pgadmin.org/.

The first thing you'll need to do is to connect to PostgreSQL using the information provided during the installation process. Once you are connected, you will have to create a database. The database name is not important in this context (not always the case when you are working on professional projects with many other people), so choose a name that makes sense for yourself.

1.4.1 Data Definition Language (DDL) statements in SQL (5 mts)

We briefly introduced DDL statements at the end of the previous case; here, we will learn about them and put them to use. Recall that these statements are used to create, modify, and remove database objects themselves as well as the data within them.

The most important statement in DDL space is the `CREATE TABLE` command. To create a table, you need to provide the table's name, its columns, and each column's type. For example, the SQL command below creates a table called `products`, with an `INTEGER` field called `productid` and a `VARCHAR(20)` (string with up to 20 characters) field called `productname`:

```
CREATE TABLE products(productid INT, productname varchar(20))
```

Once you have created a table, you can use certain DML statements used to manipulate data in the tables themselves (rather than merely in the outputs of queries, as you did in the previous case). These commands are:

1. `INSERT`: to insert data into a table
2. `UPDATE`: to update existing data within a table
3. `DELETE`: to delete records from a database table

Below is additional information on each:

1. The INSERT INTO statement is used to add rows to a table. Its syntax is as follows:

```
INSERT INTO table_name (column1, column2, column3,...)
VALUES (value1, value2, value3,...)
```

Alternatively, if you are adding values for all the columns of the table, you do not need to specify the column names.

2. The `UPDATE` statement is used to modify existing records in a table. You indicate which table you are updating, and then give the columns you want modified followed by a condition (a `WHERE` clause) that specifies which record(s) should be updated. If you omit the `WHERE` clause, all records in the table will be updated!

```
UPDATE table_name
SET column1 = value1, column2 = value2,...
WHERE condition;
```

3. The `DELETE` statement is used to delete existing records in a table:

```
DELETE FROM table_name
WHERE condition
```

Similarly to the syntax for `UPDATE`, the `WHERE` clause specifies which record(s) should be deleted. If you omit the `WHERE` clause, all records in the table will be deleted, so be VERY careful with this statement!

1.4.2 Exercise 2: (5 mts)

Set up a new database with the following tables and column details:

1. Table Name: `customers` Columns:
   - customerid INT
   - name VARCHAR(50)
   - occupation VARCHAR(50)
   - email VARCHAR(50)
   - company VARCHAR(50)
   - phonenumber VARCHAR(20)
   - age INT
2. Table Name: `agents` Columns:
   - agentid INT
   - name VARCHAR(50)
3. Table Name: `calls` Columns:
   - callid INT
   - agentid INT
   - customerid INT
   - pickedup SMALLINT
   - duration INT
   - productsold SMALLINT

Answer. One possible solution is given below:

```
CREATE TABLE customers(
    customerid INT primary key,
```

```
    name VARCHAR(50),
    occupation VARCHAR(50),
    email VARCHAR(50),
    company VARCHAR(50),
    phonenumber VARCHAR(20),
    age INT
)

CREATE TABLE agents(
    agentid INT primary key,
    name VARCHAR(50)
)

CREATE TABLE calls(
    callid INT primary key,
    agentid INT,
    customerid INT,
    pickedup SMALLINT,
    duration INT,
    productsold SMALLINT
)
```

1.4.3 Exercise 3: (5 mts)

Load the newly created tables with data. PostgreSQL offers us the COPY command to import a CSV file into a database table. The COPY statement's structure is as follows:

```
COPY tablename(column1, columns2, …., columnN) <br>
FROM [file path] DELIMITER '[delimiter]' CSV HEADER;
```

1. Specify the table with column names after the `COPY` keyword. The order of the columns must be the same as in the CSV file. If the CSV file contains exactly the columns you want in the table, then you don't have to specify the columns explicitly.
2. Specify the CSV file path after the `FROM` keyword. Because CSV file format is used, you need to specify the delimiter after the `DELIMITER` keyword, as well as the `CSV` keyword.

The `HEADER` keyword is used to skip the first row of a CSV file (which normally contains the column names). If you don't do this, PostgreSQL will try to copy all rows into the table, which will probably fail if there is a non-string column (e.g. it will try to copy the string "age" to an integer column for example).

Notice that the file must be read directly by the PostgreSQL server, not by the client application. Therefore, it must be accessible to the PostgreSQL server machine. Also, you can execute `COPY` statement successfully if you have superuser access.

Answer. One possible solution is given below:

```
COPY customer(customerid, name, occupation, email, company, phonenumber, age)
FROM 'CSV file path' DELIMITER ',' CSV HEADER
```

```
COPY agents (agentid, name)
FROM 'CSV file path' DELIMITER ',' CSV HEADER

COPY calls(callid, agentid, customerid, pickedup, duration, productsold)
FROM 'CSV file path' DELIMITER ',' CSV HEADER
```

where `CSV file path` is replaced with wherever you are storing the files locally on your computer.

1.5 Using our data tables to answer business questions (30 mts)

Now that we have set up the exact tables we want, we can reap the rewards of our labor and write DML statements to extract info and answer relevant business questions.

1.5.1 Exercise 4: (15 mts)

Two metrics of sales agent performance that your firm is interested in are: 1) for each agent, how many seconds on average does it take them to sell a product when successful; and 2) for each agent, how many seconds on average do they stay on the phone before giving up when unsuccessful. Write a query which computes this.

Answer. One possible solution is given below:

```
SELECT a.name,
SUM(
   CASE
       WHEN productsold = 0 THEN duration
       ELSE 0
   END)/SUM(
   CASE
       WHEN productsold = 0 THEN 1
       ELSE 0
   END)
AS avgWhenNotSold ,
SUM(
   CASE
       WHEN productsold = 1 THEN duration
       ELSE 0
   END)/SUM(
       CASE WHEN productsold = 1 THEN 1
       ELSE 0
   END)
AS avgWhenSold
FROM calls c
JOIN agents a ON c.agentid = a.agentid
GROUP BY a.name
ORDER BY 1
```

1.5.2 Exercise 5: (15 mts)

In order to incentivize its sales agents, the firm is offering a bonus for the agents who manage to close a sale the fastest. Write a query which gives, for each agent, the duration of that agent's quickest sale and the customer name it was sold to. If there are ties (i.e. for the same agent, two sales with the same duration), pick the one with the highest `customerid` value to be part of your query results.

Answer. One possible solution is given below:

```
SELECT a.name AS AgentName, cu.name AS CustomerName, x.duration
FROM
(
   SELECT ca.agentid, ca.duration, max(customerid) AS cid
   FROM
   (
       SELECT agentid, min(duration) as fastestcall
       FROM calls
       WHERE productsold = 1
       GROUP BY agentid
   ) min
   JOIN calls ca ON ca.agentid = min.agentid AND ca.duration = min.fastestcall
   WHERE productsold = 1
   GROUP BY ca.agentid, ca.duration
) x
JOIN agents a ON x.agentid = a.agentid
JOIN customers cu ON cu.customerid = x.cid
```

1.6 A word about DBMS properties (5 mts)

Every database must exhibit certain properties in order to guarantee that the data inside it is reliable. The **ACID properties** refer to four fundamental transactional properties of DBMS, and stands for "Atomicity, Consistency, Isolation, and Durability". If a tool claims to be a DBMS and does not exhibit all of these properties, then it is not a DBMS.

The properties can be quickly explained as follows:

1. **Atomicity**: Means "all or nothing". Let's take the bank transfer example again; remember that transferring money from Account A to Account B is two separate operations. If the bank system fails right after the money leaves Account A but before it enters Account B, then that's not "all or nothing" (and it's bad). Atomicity guarantees that a unit of work is fully executed or not executed at all.
2. **Consistency**: Means that the DBMS prevents database corruption by guaranteeing that the database is always valid according to the rules on which it was defined (essentially, no "rogue" databases not following instructions).
3. **Isolation**: Ensures that concurrent execution of transactions leaves the database in the same state as would have been obtained if the transactions had been executed sequentially. This is especially important in distributed database systems.

4. **Durability**: This property guarantees that once the transaction is committed to the database, it is durable. Basically, this means that you cannot receive an "OK" message if something bad (like a power failure) happens between when the changes are written from the memory buffer and when they are written to the disk (causing the transaction to fail).

NoSQL databases, on the other hand, support a different set of properties called BASE, which stands for "Basically Available, Soft state, Eventually consistent". We will not get into this here, except to mention that "Eventually consistent" means that "eventually" consumers may not see the latest version of the data, which may or may not be a problem depending on the context. If you would like to learn more, feel free to look up the CAP Theorem for distributed databases systems.

1.7 Database design (40 mts)

The business is happy with what you've done so far, and would like to incorporate more elements of its end-to-end sales process into the database you've created. However, this added complexity is likely to serve up some database design challenges. What are some important principles we should keep in mind as we help the business build this out?

Let's revisit the phones and cities example from earlier, where you wanted to record the city where each one of your contacts lives. This seems straightforward enough - just add a new column to your `phones` table called `city` and record the name of the correct city in each cell.

1.7.1 Exercise 6: (5 mts)

What sorts of problems and/or inefficiencies do you think this method might cause?

Answer. There are several, but some example include:

1. It wastes a lot of space. For example, suppose that you have 1000 friends in this database who all live in "San Francisco". The database would have to store the same exact piece of information, 100 times, with each instance consuming 13 characters of space.
2. If you decide to change "San Francisco" to "San Francisco, CA", the database would have to change it in 1000 different locations.
3. If you accidentally write "san francisco" for one of the cells, then even though you know that it is the same as "San Francisco", the database treats them differently. A similar situation happens with typos.
4. If you write a query with a `WHERE` clause saying `WHERE city = "New York City"`, you may miss rows if either (2) or (3) above are true.

To fix these (and other problems), the concept of **normal forms** was invented, which are a set of rules that should (read: need to) be applied to database tables to get the most out of them. Everyone does implement some of these rules, even without realizing they are doing it, but it's good to be conscious of what they are so that you can consistently practice good habits.

There are a total of 6 normal forms: 1NF, 2NF,..., 6NF, plus some intermediaries and alternatives in between. You don't need to know about all of these now, except 1NF, which we will discuss below. We suggest you spend some time on Google reading about these forms, but you'll only really understand them with practice and experience, so there is no point memorizing them now.

The process of representing a database in terms of relations in normal forms is known as **database normalization**. For example, 1NF says that:

"A relation is in first normal form if and only if the domain of each attribute contains only atomic (indivisible) values, and the value of each attribute contains only a single value from that domain.

It's really unclear if that's even English, so we'll break it down:

1. Individual tables should not contain repeat information (i.e. non-ID info)
2. If there was originally repeated information, create a separate table to group that info
3. Identify each set of repeated information with a primary key

From the above, you can conclude that there was repeated city information in the original design; therefore, a separate `cities` table should be created to store that info. There should be a primary key which corresponds to that repeated city info and which can be used as a foreign key in other tables, like `phones`.

Notice that John and Rita live in New York, while Paul lives in Boston, but the `phones` table only has the ID of the city each of them live in, rather than information about the city itself. Also note that we added an ID to the `phones` table to uniquely identify each friend (which is also consistent with the 1NF definition).

1.7.2 Question:

What are your thoughts on the name of the `phones` table? Is it really a table used to store phone numbers? What if a friend has more than one phone number? How would you proceed? Discuss this with people around you.

1.7.3 Exercise 7: (5 mts)

In order to accommodate the increased complexity of its end-to-end product sales efforts, your firm would like you to set up new tables and/or modify existing tables in your newly-created database. The additional features are as follows:

1. Each product has a name, and customers can buy multiple products
2. Some products are upgrades to others; i.e. you must purchase the baselines version before you can purchase upgrades. For the sake of simplicity, assume each product can be upgraded from at most one other product
3. You need to keep track of customers' purchases of the various products

How would you set up new tables and/or modify existing tables to accommodate for these requirements? Write SQL queries that will accomplish this task. You will need the following additional DDL statements:

The `ALTER TABLE` statement is used to add, delete, or modify columns in an existing table or to add and drop constraints on an existing table.

1. Adding a column:

```
ALTER TABLE "tablename"
ADD "newcolumn" "datatype"
```

2. Dropping a column:

```
ALTER TABLE "tablename"
DROP COLUMN "columnname"
```

Answer. An ideal design would be as follows:

1. A `products` table:
   - `productid` as a unique primary key (reasonable to assume it is an integer)
   - `productname` as the name of the product (string, reasonable to asume VARCHAR(50))
   - `upgradedfromproductid` as the ID of the product from which this is an upgrade
2. Modify the `calls` table:
   - Add `productid` as a foreign key

An appropriate set of SQL queries for this is as follows:

```
CREATE TABLE products(
    productid INT primary key,
    productname VARCHAR(50),
    upgradedfromproductid INT
)

ALTER TABLE calls
ADD productid INT
```

1.7.4 Question: (5 mts)

Discuss this with the people around you. How would you modify your design in Exercise 7 if:

1. a product could be upgraded from multiple previous products
2. certain products can only be sold at certain periods of time (i.e. product availability is seasonal). You can assume this set of periods is not too large
3. You had to keep track of the times when the products are avaiable

1.7.5 Exercise 8: (15 mts)

The business would like you to extend your database design work into a brand-new data warehouse that will be used for real-time reporting. The data warehouse will be populated by a daily Extract-Transform-Load (ETL, which you will learn about in later cases) process that your team will also write. The main source of data comes from the current end-to-end sales process. In addition to the tables in your previous design, here is a list of additional tables you think you will need:

1. `orders`: contains information about the sales date, the expected delivery date, the customer that bought the product, the total value of the sale, the delivery address, the cargo company, and the driver that will perform the delivery
2. `orderItems`: contains information about each item in the order such as: the order item ID, the ID of the order, the product ID, the item quantity, and the product price (note that the same product can be sold for different prices across different orders)
3. `productCategory`: contains the description of the category

How would you design your datawarehouse to maximize read performance? Work on this with a partner. Note that in order to optimize read performance, it is NOT necessarily best to follow normal form protocol.

Answer. One possible solution is given below:

1. Have one `orders` table that contains information about orders and items, where each row is indexed by `orderid` and `orderitemid`. Although this does not conform with normal form protocol, a reporting system that performs a lot of reads will almost certainly want to extract information about both orders and items so this avoids a lot of expensive Cartesian products and JOINs of two tables that will be tied at the hip.
2. Add a detail key `itemTotal` to each row of the `orders` table, which is defined as `itemQuantity` times `productPrice`, as this field will probably be important for reporting, which is part of read performance.
3. Add `productcategoryid` as a foreign key to the `products` table
4. Break out the delivery address field of `orders` into separate components such as `state`, `city`, `zipcode`, etc. Aggregation based on location is a very common sort of reporting function and having to parse addresses every time is a pain unless the groupings are already well-defined.

1.8 Basic properties of data warehouses (5 mts)

We briefly mentioned the term **data warehouse** in the previous exercise, but what is it really? Unlike databases, data is often de-normalized in a data warehouse. In professional settings, sometimes you may be required to read data from a data warehouse instead of a database. Now, this is a very advanced concept so we'll not go too deep into it here, but we'll drop in a few bullet points so that you are familiar with the concept if it comes up in a professional setting:

1. Data warehouses are mainly used for reporting as opposed to day-to-day transactions
   - As such, they are optimized for read rather than write operations
   - If you want to go deeper on this, read about OLTP versus OLAP systems
2. Data warehouses run on different hardware (servers) than the database does
   - Usually, that hardware is a lot more robust than the one the database runs
3. Data warehouses are usually updated via "batch" jobs (ETL)
   - For example, once a day all changes from the database (compared to the previous day) are propagated to the data warehouse
4. Data warehouses don't throw away all the normalization that was done at the database level, they just change them a bit to reduce the number of joins necessary to run the queries
   - In other words, they sacrifice storage efficiency in favour of performance
5. Data warehouses are the source of data for Business Intelligence (BI) systems, particularly those that perform OLAP analysis

1.9 Conclusions

In this case, you ventured outside of SQL in a Python environment and used your newfound knowledge to expand the scope of databaases within a financial products firm. First, you learned about database management systems and downloaded a popular open-source version, PostgreSQL, onto your machine. Within PostgreSQL, you set up an initial database and made upgrades to it

based on more complex business requirements. You also performed queries on your new database to answer relevant business questions.

1.10 Takeaways

Although databases are powerful structures, they are severely limited without database management systems to enable large-scale collaboration among many users. PostgreSQL is a highly popular, open-source database management system that is used recreationally and in professional settings. Within PostgreSQL, you can utilize all of the usual powers of SQL, include DML and DDL statements.