case_12.1

May 1, 2020

1 How can we optimize our sales of financial products?

1.1 Introduction

Business Context. You are a data analyst at a large financial services firm that sells a diverse portfolio of products. In order to make these sales, the firm relies on a call center where sales agents make calls to current as well as prospective customers. The company would like you to dive into their data to devise strategies to increase their revenue or reduce their costs. Specifically, they would like to double down on their most reliable customers, and to cut out sales agents whom are not producing outcomes.

Business Problem. The business would like to answer the following questions: **"What types of customers are most likely to buy our product? And which of my sales agents are the most/least productive?**

Analytical Context. The data is split across 3 tables: "Agents", "Calls", and "Customers", which sit on CSV files. Unlike previous cases though, we will first be reading these CSV files into a SQLite database created within Python. You will learn how this database differs from CSV files and how to interact with it using SQL to extract useful insights.

The case is sequenced as follows: you will (1) learn the fundamentals of databases and SQL; (2) use SQL `SELECT` statements to identify potentially interesting customers; and (4) use SQL aggregation functions to compute summary statistics on your agents and identify the most/least productive ones.

1.2 Why databases?

While we have been dealing with data sitting in CSV files so far, no serious data organization runs their operations off of CSV files on a single person's computer. This practice presents all sorts of hazards, including but not limited to:

1. Destruction of that single device
2. Destruction of the files on that device
3. Inability to connect to that person's device from another device that requires the data
4. Inability to store more than a limited amount of data (since a single device doesn't have that much memory)

Therefore, our data should be stored elsewhere if we want to reliably access it in the future and, more importantly, share it and work on it with others. The **database** is the classic location where modern organizations have chosen to store their data for professional use. A couple of advantages that databases provide are:

- Ability to query only certain recods, instead of fetching and going through the entire csv file
- User based access restrictions - Specify what data each of your users can access from the database. This will strengthen the privacy of the data.

Daabases have been a topic of research since the late 1960s. Many technology vendors picked up on this and developed databases software for companies to consume. Some of these vendors and products are:

1. Microsoft, initially with Microsoft Access and more recently with Microsoft SQL Server
2. Oracle, with their Oracle database
3. The "PostgreSQL Global Development Group", with the open-source PostgreSQL

1.3 Types of databases

At this point, you might believe that databases can be thought of as a collection of data. This is true, but unfortunately it is not that simple. Data cannot simply be thrown in a database the same way you throw your socks in your sock drawer. The data that you wish to store in your database must follow some patterns which are determined by the **database type** you wish to store.

1.3.1 Relational databases

The most common database type is called a **relational database**, and the systems that manage these kinds of databases are called **Relational Database Management Systems (RDBMS)**. Relational databases date back to the early 1970s and can be considered the first type of database ever conceived.

Relational databases deal with "relational data", which is a fancy way to say "tabular" data. This kind of dataset consists of rows and columns (i.e. tables) where each row corresponds to an observation and each column corresponds to an attribute of that observation. So, for example, if we go back to the example where we were keeping track of our friends and their phones, each row on the file (or table) represents one friend and each column represents the information we want to track about that friend (name and phone number). The cell on the intersection of the row and column contains the actual data. Relational data is manipulated using a specific language called **SQL (Structured Query Language)**, which we will learn about soon.

A simple way to conceptualize a table inside a relational database is as a CSV file "copied" to the database. In fact, many databases offer that possibility (assuming your file is correctly formatted, of course).

1.3.2 NoSQL databases

Around 20 years ago, with the advent of the internet and the necessity to store and process unstructured data (i.e. data that does not fit well in the row-by-column paradigm), developers started to discuss another type of database, which eventually ended up being referred to as a **NoSQL**

database. As the name implies, these databases do not rely on SQL and are not relational. They are also built with more "relaxed" rules compared to their predecessors.

1.4 What is this "SQL" thing?

Just like data can't really survive without a database, a database can't be utilized without SQL. SQL is used for a wide variety of tasks, including but not limited to extracting data, creating the internal structure of a database (in the form of tables), and reading and writing data to these tables.

In this case, we will be writing SQL queries using the `SQLAlchemy` package in Python. This allows you to directly interface with relational databases without exiting the Python environment, while using syntax that is identical to what you would write outside of Python. Run the code below to set up this framework:

```
[2]: import pandas as pd
     from sqlalchemy import create_engine, text
     #maximum number of rows to display
     pd.options.display.max_rows = 10

     engine=create_engine('sqlite://')
     df = pd.read_csv('customers.csv').to_sql('customers', engine,␣
      ↪if_exists='replace', index=False)
     df = pd.read_csv('agents.csv').to_sql('agents', engine, if_exists='replace',␣
      ↪index=False)
     df = pd.read_csv('calls.csv').to_sql('calls', engine, if_exists='replace',␣
      ↪index=False)

     def runQuery(sql):
         result = engine.connect().execute((text(sql)))
         return pd.DataFrame(result.fetchall(), columns=result.keys())
```

1.5 Finding potentially interesting customer cohorts

The most important thing you will ever do in SQL is extract a subset of the data from a SQL table based on a set of rules. This is accomplished using the following statement syntax:

1. Start with the keyword `SELECT`
2. Follow with the names of the columns you want to select, separated by commas (alternatively, you can use the * symbol to indicate you wish to select all columns)
3. Follow with the keyword `FROM`
4. Finish with the name of the table you wish to select data from

Additionally, you can use the `WHERE` clause to only return results which satisfy certain conditions (similar to how code within Python if-then blocks only execute if the associated conditions are true). `WHERE` clauses immediately follow the table name you want to select data from.

Since the firm wants to dig deeper into its customers, let's start by pulling some of their data out of our files; namely, information about customers who are not unemployed (and therefore are more likely to buy from us).

1.5.1 Exercise 1:

1.1 Write a query that selects the customer ID and name from the `customer` table, only showing results for customers who are not unemployed. Remember to write your query as a multi-line string (enclosed within a pair of triple quotes `"""`) and pass it to the `runQuery()` function defined in the framework above to check your work!

Answer. One possible solution is given below:

```
SELECT customerid, name
FROM customers
WHERE occupation != 'Unemployed'
```

```
[4]: query1 = """SELECT customerid, name
     FROM customers
     WHERE occupation != 'Unemployed'"""
     runQuery(query1)
```

```
[4]:      customerid              name
     0             1  Michael Gonzalez
     1             2     Amanda Wilson
     2             3     Robert Thomas
     3             4        Eddie Hall
     4             6     Maria Johnson
     ..          …                 …
     755         994      Ruben Steele
     756         995      Ashley Young
     757         996  Mr. Steven Smith
     758         997        Mark Smith
     759         999      Karen Barber

     [760 rows x 2 columns]
```

Of course, for names, it's sensible to try to list them in alphabetical order. SQL allows us to do this rather easily with the `ORDER BY` statement. This is then followed by a comma-separated list of columns on which you want to order your results (columns that come first take priority in the subsequent ordering). Optionally, you can then append the keyword `ASC` or `DESC` (short for ascending and descending, respectively) after each column to determine the ordering type (e.g. alphabetical or reverse-alphabetical for a string column).

We can also use the `AS` statment to change the name of a column returned by your query. However, this change is only temporary and is only valid for that particular query. For example, we can rename the `name` column to `customername` and order it alphabetically. This operation is known as **aliasing**:

```
SELECT customerid, name AS customername
FROM customers
WHERE occupation != 'Unemployed'
ORDER BY customername
```

```
[5]: query2 = """SELECT customerid, name AS customername
     FROM customers
     WHERE occupation != 'Unemployed'
     ORDER BY customername"""
     runQuery(query2)
```

```
[5]:      customerid     customername
     0           900  Aaron Gutierrez
     1           622       Aaron Rose
     2           226        Adam Ward
     3           786    Alan Chambers
     4           985    Alan Mitchell
     ..          …               …
     755         699    Willie Greene
     756         715   Yesenia Wright
     757         952    Yolanda White
     758         421     Zachary Ruiz
     759         392   Zachary Wilson

     [760 rows x 2 columns]
```

This is a great first step; however, while producing the list of customers that are not unemployed, you inevitably spend a lot of time looking at the different professions your customers have and realize how often engineers appear in your database. You know that engineering jobs tend to command higher salaries these days, so you decide to try to extract a list of all the unique types of engineering jobs that are represented in your database. To ensure that you don't get duplicate job titles in your query results, you'll need to write the keyword `DISTINCT` immediately after `SELECT` in your query.

1.5.2 Exercise 2:

Write a query which produces a list, in alphabetical order, of all the distinct occupations in the `customer` table that contain the word "Engineer".

(Hint: The `LIKE` operator can be used when you want to look for similar values. It is included as part of a `WHERE` clause. It needs to be complemented with the `%` symbol, which is a wild card that represents zero, one, or multiple characters. For example, one valid `WHERE` clause utilizing the `LIKE` operator is `WHERE name LIKE 'Matt%'`, which would return any results where the person's name starts with the word "Matt"; e.g. "Matt" or "Matteo" or "Matthew", etc.)

Answer. One possible solution is given below:

```
SELECT DISTINCT occupation
FROM customers
```

```
WHERE occupation LIKE '%Engineer%'
ORDER BY occupation
```

```
[6]: query3 = """SELECT DISTINCT occupation
     FROM customers
     WHERE occupation LIKE '%Engineer%'
     ORDER BY occupation"""
     runQuery(query3)
```

```
[6]:                       occupation
     0            Chemical engineer
     1          Electrical engineer
     2       Engineer, aeronautical
     3       Engineer, agricultural
     4         Engineer, automotive
     ..                         …
     24        Engineer, production
     25              Engineer, site
     26        Engineer, structural
     27   Engineer, technical sales
     28             Engineer, water

     [29 rows x 1 columns]
```

Now, one of your marketing colleagues tells you that people who are 30 or older will have a higher probability of buying your product (presumably because by that point they have more disposable income and savings). You don't want to take your colleague's word for granted, so you decide not to completely ignore people under 30, but instead to add that information on the report regarding the person's age, so that the agent making the subsequent call can decide how they want to use that information. However, due to privacy concerns, you also cannot share the person's exact age.

1.5.3 Exercise 3:

Write a query that retuns the customer ID, their name, and a column `Over30` containing "Yes" if the customer is more than 30 years of age and "No" if not.

(Hint: You will need to use the `CASE-END` clause. The `CASE-END` clause can be used to evaluate conditional statements and returns a value once a condition is met (similar to an if-then-else clause in Python). If no conditions are true, it returns the value in the ELSE clause (or NULL if there is no ELSE statement). For example:

```
CASE
    WHEN name = "Matt" THEN 'Yes'
    WHEN name = "Matteo" THEN 'Maybe'
    ELSE 'No'
END
```

Answer. One possible solution is given below:

```sql
SELECT customerid, name,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers
ORDER BY name DESC
```

[7]:
```python
query4 = """SELECT customerid, name,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers
ORDER BY name DESC"""
runQuery(query4)
```

[7]:
```
     customerid                name Over30
0           392      Zachary Wilson    Yes
1           986  Zachary Stevenson     No
2           421        Zachary Ruiz    Yes
3            18        Zachary Howe     No
4           883    Zachary Anderson     No
..          ...                 ...    ...
995          65        Adam Jimenez     No
996         622          Aaron Rose     No
997         145      Aaron Mcintyre     No
998         461       Aaron Hendrix     No
999         900     Aaron Gutierrez    Yes

[1000 rows x 3 columns]
```

Let's now modify Exercise 3 so that the query only returns customers who work in an engineering profession:

```sql
SELECT customerid, name,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers
WHERE occupation LIKE '%Engineer%'
ORDER BY name DESC
```

[8]:
```
query5 = """SELECT customerid, name,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers
WHERE occupation LIKE '%Engineer%'
ORDER BY name DESC"""
runQuery(query5)
```

[8]:
```
     customerid             name Over30
0           421     Zachary Ruiz    Yes
1           952    Yolanda White     No
2           699    Willie Greene    Yes
3           973  William Jackson    Yes
4           966   William Garcia     No
..          …                …    …
356         918   Alison Vaughan    Yes
357         568        Alice Lee     No
358         432    Alexis Riddle     No
359         985    Alan Mitchell    Yes
360         622       Aaron Rose     No

[361 rows x 3 columns]
```

1.6 Investigating customer conversion rates

In order to validate whether our hypotheses about engineers and age are true (for example, engineers exhibit higher product sales conversion rates, and perhaps engineers over 30 tend to exhibit an even higher conversion rate), we will need to use two tables: `calls` and `customers`. This is because the column `productsold` lies only in the `calls` table, yet information about customer professions and age only lie in the `customers` table.

`SELECT` commands are not restricted to a single table. In fact, theoretically, there is no limit to the number of tables that you can extract data from in a single SQL query. Let's introduce some new concepts that are relevant once we go beyond a single table.

Primary and foreign keys are very important concepts that need to be understood by any database professional. Primary keys:

1. Uniquely identify a record in the table. Their name usually includes the word "id"
    - For example, `customerid` is the primary key of the `customers` table, `agentid` is the primary key of the `agents` table, and `callid` is the primary key of the `calls` table
2. Do not accept null values. And they shouldn't, because they are being used to identify the record
3. Are limited to one per table

On the other hand, foreign keys:

1. Are a field in the table that is the primary key in another table
2. Can accept null values
3. Are not limited in any way per table
    - For example, the `calls` tables has 2 foreign keys: `agentid` and `customerid` pointing to the `agents` and `customers` tables, respectively

1.6.1 Extracting call data for customers working in engineering professions

Let's first extract the relevant data so we can perform this analysis. Here, a `JOIN` clause will come in handy. A `JOIN` clause consists of two parts:

1. The base `JOIN` statement, which is of the form `[Table 1] JOIN [Table 2]`. This performs a Cartesian product on the 2 tables being joined. For example, if we have Table A with 5 rows, and Table 5 with 3 rows, their Cartesian product will return 15 rows (5 x 3)
2. A `JOIN` criteria, which filters the Cartesian product's results, beginning with the `ON` keyword

Here is an example of a `JOIN` criteria in action, which is telling us to only give combinations of rows where the agent ID matches in both tables:

```sql
SELECT callid, a.agentid, name
FROM calls c
JOIN agents a ON c.agentid = a.agentid
ORDER BY name DESC
```

[9]:
```python
query6 = """SELECT callid, a.agentid, name
FROM calls c
JOIN agents a ON c.agentid = a.agentid
ORDER BY name DESC"""
runQuery(query6)
```

[9]:
```
      callid  agentid         name
0         12        3  Todd Morrow
1         28        3  Todd Morrow
2         32        3  Todd Morrow
3         50        3  Todd Morrow
4         60        3  Todd Morrow
…        …       …           …
9934    9985       10      Agent X
9935    9986       10      Agent X
9936    9991       10      Agent X
9937    9992       10      Agent X
9938    9994       10      Agent X

[9939 rows x 3 columns]
```

Note that:

1. `c` and `a` are aliases to the `calls` and `agents` tables to avoid having to type the table name every time. Unlike with column aliasing earlier, we do not need the `AS` keyword here
2. We write `a.agentid` instead of `agentid` in the SELECT statement – this is because the `agentid` column exists in both tables, so we have to tell the database which one to get the result from

1.6.2 Exercise 4:

Write a query which returns all calls made out to customers in the engineering profession, and shows whether they are over or under 30 as well as whether they ended up purchasing the product from that call.

Answer. One possible solution is given below:

```sql
SELECT callid, cu.customerid, name, productsold,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE occupation LIKE '%Engineer%'
ORDER BY name DESC
```

[10]:
```python
query7 = """SELECT callid, cu.customerid, name, productsold,
    CASE
        WHEN age >= 30 THEN 'Yes'
        WHEN age <  30 THEN 'No'
        ELSE 'Missing Data'
    END AS Over30
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE occupation LIKE '%Engineer%'
ORDER BY name DESC"""
runQuery(query7)
```

[10]:
```
      callid  customerid          name  productsold Over30
0       2049         421  Zachary Ruiz            0    Yes
1       2960         421  Zachary Ruiz            0    Yes
2       3365         421  Zachary Ruiz            0    Yes
3       3386         421  Zachary Ruiz            1    Yes
4       4332         421  Zachary Ruiz            0    Yes
…         …          …            …           …     …
3614    6444         622    Aaron Rose            1     No
3615    7994         622    Aaron Rose            0     No
3616    8811         622    Aaron Rose            0     No
3617    9524         622    Aaron Rose            1     No
```

```
3618    9965          622     Aaron Rose             0      No

[3619 rows x 5 columns]
```

1.6.3 Analyzing the call conversion data

Now, we want to determine whether or not customers in our desired cohort exhibit a higher sales conversion rate compared to the overall population of customers. A reasonable way to do this is to count the total number of calls to this cohort which resulted in a sale, and divide that by the total number of calls to this cohort (whether or not they resulted in a sale) to get a percentage, and then compare that with the percentage we compute from the `calls` table overall.

However, to compute these figures, we'll need to learn a bit about **aggregation functions**. An aggregation function allows you to perform a calculation on a set of values to return a single value, essentially computing some sort of summary statistic.

The following are the most commonly used SQL aggregate functions:

1. `AVG()` – calculates the average of a set of values
2. `COUNT()` – counts rows in a specified table or view
3. `MIN()` – gets the minimum value in a set of values
4. `MAX()` – gets the maximum value in a set of values
5. `SUM()` – calculates the sum of values

1.6.4 Exercise 5:

Write two queries: one that computes the total sales and total calls made to customers in the engineering profession, and one that computes the same metrics for the entire customer base. What can you conclude regarding the conversion rate within the engineering customers vs. the overall customer base?

Answer. One possible solution is given below:

```sql
SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE occupation LIKE '%Engineer%'
```

[11]:
```python
query8 = """SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE occupation LIKE '%Engineer%'"""
runQuery(query8)
```

[11]:
```
   totalsales  ncalls
0         760    3619
```

```
SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
```

[12]:
```
query9 = """SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid"""
runQuery(query9)
```

[12]:
```
   totalsales  ncalls
0        2084    9925
```

The conversion rate for both groups is ~20.9%, indicating that engineers are not more likely to purchase our products than the overall population.

1.6.5 Exercise 6:

Write a query that computes the total sales and total calls made to customers over the age of 30. Is there a notable difference between the conversion ratio here and that of the overall customer base?

Answer. One possible solution is given below:

```
SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE age >= 30
```

[13]:
```
query10 = """SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE age >= 30"""
runQuery(query10)
```

[13]:
```
   totalsales  ncalls
0         659    3096
```

The conversion rate is ~21.1% vs. the overall ~20.9%. There may be some difference, but it is quite small so we would need to run statistical significance tests in order to validate this. (You will learn about these in future cases.)

1.6.6 Exercise 7:

How about if you look at the sales conversion rate for engineers over the age of 30?

```
SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
FROM customers cu
JOIN calls ca ON ca.customerid = cu.customerid
WHERE occupation LIKE '%Engineer%' AND age >= 30
```

Answer. One possible solution is given below:

```
[14]: query11 = """SELECT SUM(productsold) AS totalsales, COUNT(*) AS ncalls
      FROM customers cu
      JOIN calls ca ON ca.customerid = cu.customerid
      WHERE occupation LIKE '%Engineer%' AND age >= 30"""
      runQuery(query11)
```

```
[14]:    totalsales  ncalls
      0         376    1816
```

Here, we actually observe the opposite pattern – the conversion rate is only ~20.5%.

From these numbers, we can conclude that a customer's status as an engineering professional has no positive effect on their conversion rate. On the other hand, having an age of at least 30 MAY have some effect; however, we would need to do more in-depth statistical testing to determine this.

1.7 Evaluating our agents' performance

Recall the second part of our business question: we need to figure out which of our agents are the most and least productive. To do this, it makes sense to determine which metrics could be related to productivity. Looking at the features present, the following seem to be reasonable:

1. The number of calls an agent made
2. The lengths of calls an agent made
3. The total number of products an agent sold

1.7.1 Question:

For any given agent, would extracting this info be a good way of quickly analyzing their productivity? Why or why not?

While the above metrics are useful, some of them are too numerous to be easiy analyzed. Specifically, the lengths of calls an agent made is a dataset that is as large as the number of calls the agent made. If the agent made many calls, it will be meaningless to just throw the entire set of call lengths at ourselves. Instead, we ought to compute some summary statistics of this metric; namely, the minimum, maximum, and mean lengths seem reasonable.

1.7.2 Exercise 8:

Write a query that returns, for each agent, the agent's name, number of calls, longest and shortest call lengths, average call length, and total number of products sold. Name the columns `agentname`, `ncalls`, `shortest`, `longest`, `avgduration`, and `totalsales`, and order the table by `agentname` alphabetically. (Make sure to include the `WHERE pickedup = 1` clause to only calculate the average across all the calls that were picked up (otherwise all the minimum durations will be 0)!)

Answer. One possible solution is given below:

```
SELECT name AS agentname, count(*) AS ncalls, MIN(duration) AS shortest, MAX(duration) AS longe
FROM calls c
    Join agents a ON c.agentid = a.agentid
WHERE pickedup = 1
GROUP BY name
ORDER BY name
```

[15]:
```
query12 = """SELECT name AS agentname, count(*) AS ncalls, MIN(duration) AS␣
 ↪shortest, MAX(duration) AS longest, AVG(duration) AS avgduration,␣
 ↪SUM(productsold) AS totalsales
FROM calls c
    Join agents a ON c.agentid = a.agentid
WHERE pickedup = 1
GROUP BY name
ORDER BY name"""
runQuery(query12)
```

[15]:
```
             agentname  ncalls  shortest  longest  avgduration  totalsales
0              Agent X     640        22      334   180.975000         194
1         Angel Briggs     591        12      362   181.081218         157
2   Christopher Moreno     649        47      363   177.979969         189
3           Dana Hardy     554        49      356   177.203971         182
4         Gloria Singh     662        36      349   182.175227         209
..                 …       …        …        …           …           …
6         Lisa Cordova     639        46      344   179.214397         201
7     Michele Williams     685        22      306   177.880292         198
8           Paul Nunez     648        -5      323   181.070988         194
9          Randy Moore     600        16      326   178.595000         177
10         Todd Morrow     631        -3      339   180.711569         204

[11 rows x 6 columns]
```

1.7.3 Question:

Throughout this case, we have defined sales conversion rate as the number of products sold divided by the number of calls made. What are the strengths and weaknesses of this choice of definition? Is there a way you can adjust the definition to correct for some of those weaknesses while retaining all the strengths?

1.8 A word about SQL statement types

In this case, you have used SQL's **Data Manipulation Language (DML)** statements; that is, statements that are used to read or write (manipulate) data from the database. However, SQL also has the ability to create, modify, and remove database objects themselves as well as the data within them. It does this by using **Data Definition Language (DDL)** statements which are commands that define the different structures in a database. You will learn more about these statements in future cases.

There are two other types of SQL statements that are important, but less likely to be used by someone who is merely focused on analyzing data. We'll not not dig into these as it's very unlikely that you'll have to deal with these anytime soon, but you are free to read up about them elsewhere if you are interested. They are:

1. **Data Control Language (DCL)**: These determine who has permission to do what in the database. Everytime you log in to a database, you do that using (your) database user account. By default, a user after being created does not have permission to do anything, so someone (normally a **database administrator (DBA)**) needs to grant permission to that user to perform certain operations on the database.
2. **Transactional Control Language (TCL)**: These commands are used to guarantee that full units of work are either completed as a whole or not at all. An example is a bank transfer: you need to ensure that if money has been withdrawn from account A, then it has also been deposited in account B, which requires wrapping these two commands into a transaction.

1.9 Conclusions (5 mts)

In this case, you learned the basics of SQL and used it to optimize the sales operations of a financial services firm. We narrowed down our set of potentially interesting customer cohorts and were able to compute summary statistics on the sales conversion rates of those cohorts, particularly versus the mean. In particular, we learned that some of our ”no-brainer” hypotheses did not pan out, which illustrates the importance of always investigating the data to validate our thoughts. We also looked at sales agent performance and were able to find the ones that were most/least productive on particular metrics.

1.10 Takeaways (5 mts)

SQL is a powerful tool that can help us navigate and understand data in ways that Python cannot. Sometimes, it can even serve as the first stage of an exploratory data analysis and can sometimes help us answer questions all by itself. Furthermore, SQL is the means through which we can create and persist data in databases for future, large-scale use. No data scientist's toolkit is complete without an understanding of how to interface with and store the raw data that they work with.