Coursebook: SQL Query using pandas

- Part 4 of Data Analytics Specialization
- Course Length: 9 hours
- Last Updated: April 2023 _____
- Author: Samuel Chan
- Developed by Algoritma's product division and instructors team

Background

Top-Down Approach

The coursebook is part 4 of the **Data Analytics Specialization** offered by Algoritma. It takes a more accessible approach compared to Algoritma's core educational products, by getting participants to overcome the "how" barrier first, rather than a detailed breakdown of the "why".

This translates to an overall easier learning curve, one where the reader is prompted to write short snippets of code in frequent intervals, before being offered an explanation on the underlying theoretical frameworks. Instead of mastering the syntactic design of the Python programming language, then moving into data structures, and then the `pandas` library, and then the mathematical details in an imputation algorithm, and its code implementation; we would do the opposite: Implement the imputation, then a succinct explanation of why it works and applicational considerations (what to look out for, what are assumptions it made, when *not* to use it etc).

Training Objectives

This coursebook is intended for participants who have completed the preceding courses offered in the **Data Analytics Developer Specialization**. This is the third course, **SQL and Data Visualization with Pandas**.

The coursebook focuses on:

- Querying from SQL Databases
- SQL Joins
- SQL Conditional Statements
- Flavors and Common Operators
- End to end data analysis

At the end of this course is a Graded Asssignment section, where you are expected to apply all that you've learned on a new dataset, and attempt the given questions.

Working with SQL Databases

There are a great number of python modules that provide functionalities to work with databases of all variants and flavors. For a MySQL database, we may form a connection using `pymysql` or one of many other alternatives:

```
import pymysql
conn = pymysql.connect(
    host=host,
    port=port,
    user=user,
    password=password,
    db=database)
```

We can then use `pd.read_sql_query()`, passing in the connection:

```
sales = pd.read_sql_query("SELECT * FROM sales", conn)
```

Under the hood, `pandas` uses SQLAlchemy so any database supported by that library will work. This isn't something you need to worry about at this stage of your learning journey, but for the sake for practice, let's also see how a connection URI for a SQLite database looks like:

```
import sqlite3
import pandas as pd

conn = sqlite3.connect("data_input/chinook.db")

albums = pd.read_sql_query("SELECT * FROM albums", conn)
albums.head()
```

```
import sqlite3
import pandas as pd

conn = sqlite3.connect("data_input/chinook.db")

albums = pd.read_sql_query("SELECT * FROM employees", conn)
albums.head()
```

In the above command, we asked for all columns of a table to be returned to us through the `SELECT *` command. Well, columns of which table? That would be `tables`. Together they form an SQL query:

`SELECT * FROM albums`

The database we're working with have a few tables populated with sample data. The database has the following schema:

Knowledge Check We'll create a `DataFrame`: this time select all columns from the `artists` table. Recall that when we use `pd.read_sql_query()` command we pass in the SQL query as a string, and add a connection as the second parameter. Save the output as a `DataFrame`.

Your DataFrame should be constructed like this:

`__ = pd.read_sql_query("SELECT __ FROM __ ", conn)`

Question:

1. How many rows are there in your DataFrame?

```
## Your code below


## -- Solution code
```

The `pd.read_sql_query` is most commonly used with that two parameters above, but on its official documentation is a list of other parameters we can use as well.

In the following cell, we use a similar SQL query with an additional `LIMIT` statement to limit the output to the first 5 records (rows). However, notice that we also set `index_col` so the specified column is recognized as the index:

```
pd.read_sql_query("SELECT * FROM artists LIMIT 5",
                  conn,
                  index_col='ArtistId')
```

SQL Joins

`JOIN` statements are used to combine records from two tables. We can have as many `JOIN` operations as we want in a SQL query.

Below is a diagram of the different types of SQL `JOIN` operations:

Credit: Data & Object Factory, LLC

In most business scenarios though, a `LEFT JOIN` is almost always the type of `JOIN` you want - it is very direct (and therefore easy to reason about). Left join return all records in the left table regardless if any of those records have a match in the right table.

The `INNER JOIN` is also very intuitive and easily understood. This query return all of the records in the left table that has a matching record in the right table.

> As a personal side note, I've worked at companies where `RIGHT JOIN` is outright forbidden in favor of `LEFT JOIN`: directness and ease-of-understanding aside, all right joins can be replaced by the opposite left join.
>
> The `OUTER JOIN` (also referred to as `FULL JOIN`) is also quite uncommon in practice. Performance reason aside, an outer join return all of the records from both tables regardless if there is a match or not, resulting in a DataFrame that has potentially a lot of `NULL` values.

Consider the database schema illustraation again and pay attention to two tables and their respective columns:

1. `albums`:
   - `AlbumId`, `Title`, `ArtistId`
2. `artists`: -ArtistId`,`Name‘

We want a `pandas` DataFrame containing the `AlbumId`, `Title` and `Name`. Notice that `Name` is from the `artists` table while the other columns are from the `albums` table. What is a reasonable strategy?

The most straightforward solution is the `LEFT JOIN`, let's see an example:

```
albums = pd.read_sql_query("SELECT AlbumId, Title, a.Name \
                            FROM albums \
                            LEFT JOIN artists as a \
                            ON a.ArtistId = albums.ArtistId", conn)
albums.head()
```

Notice that in the code above, we place a backslash (\) character so we have line continuation and the newline will be ignored. This allows SQL to treat the entire query string as if they were essentially one line.

```
pd.read_sql_query("SELECT * FROM albums", conn).head()
```

Knowledge Check Consider the database schema illustraation again and pay attention to two tables and their respective columns:

1. `albums`: `AlbumId`, `Title`, `ArtistId`
2. `tracks`: `TrackId`, `Name`, `AlbumId`, `GenreId`, ... `UnitPrice`
3. `genres`: `GenreId`, `Name`

Create a `DataFrame` containing all columns from the `tracks` table; Additionally, it should also contain: - The `Title` column from the `albums` table - The `Name` column from the `artists` table - The `Name` column from the `genres` table

> **Hint 1**: In your `SELECT` statement, you can use `SELECT tracks.* FROM TRACKS` to select all columns from the `TRACKS` table
>
> **Hint 2**: When we write `SELECT tracks.Name as tracksName`, we are renaming the output column from `Name` to `tracksName` using a technique called column aliasing. You may optionally consider doing this for columns that share the same name across different tables

Set the `TrackId`column to be the index. The resulting `DataFrame` should has 11 columns.

Give your `DataFrame` a name: name it `tracks`. Perform EDA on `tracks` to answer the following question:

1. Use `tail()` to inspect the last 5 rows of data. Which genre is present in the last 5 rows of our `tracks` DataFrame (Check all that apply)?

   ☐ Latin
   ☐ Classical
   ☐ Soundtrack
   ☐ Pop

2. Apply `pd.crosstab(..., columns='count')`, `.value_counts()`, or any other techniques you've learned to compute the frequency table of Genres in your DataFrame. Which is among the top 3 most represented genres in the `tracks` DataFrame?

   ☐ Latin
   ☐ Classical
   ☐ Soundtrack
   ☐ Pop

3. Use `groupby()` on Artist Name and compute the `mean()` on the `UnitPrice` of each tracks. You will realize that most artists price their tracks at 0.99 (`mean`) but there are several artists where the `mean()` is 1.99. Which of the Artist has a mean of 0.99 `UnitPrice`:

- ☐ The Office
- ☐ Aquaman
- ☐ Pearl Jam
- ☐ Lost

```
## Your code below


## -- Solution code
```

SQL Aggregation

Since you have learned various aggregation tools in `pandas` such as `.crosstab()`, `.pivot_table()`, and `.groupby()`. It is also common practice to perform aggregation function using SQL. Consider the following query:

```
top_cust = pd.read_sql_query("SELECT CustomerId, SUM(Total)  as TotalValue, \
                              COUNT(InvoiceId) as Purchases \
                              FROM INVOICES \
                              GROUP BY CustomerId \
                              ORDER BY TotalValue DESC \
                              LIMIT 5", con=conn, index_col='CustomerId')
top_cust
```

Notice how the query fetched the top 5 customers from all time from the `invoices` table grouped by unique `CustomerId`. We performed two aggregation functions: `SUM()` and `COUNT()`, each aggregating different column from the `invoices` table. At the end, the `ORDER BY` statement is added to order the table based on the `TotalValue` column. Do note that the aggregated columns needs to be a numeric as the available aggregated functions are: `SUM`, `AVG`, `COUNT`, `MIN`, and `MAX`.

Knowledge Check Edit the following code to find out the most popular `genres` from all invoice sales. Use different column to acquire the following information: Summation of `Total` sales, and number of tracks bought from the `Quantity` columns.

hint: The `Total` can be obtained from `UnitPrice` and `Quantity` column in `invoice_items`

```
top_genre = pd.read_sql_query("SELECT genres.GenreId, genres.Name, \
                               __(invoice_items.quantity * invoice_items.unitprice) AS Total, \
                               SUM(invoice_items.__) AS ___ \
                               FROM tracks \
                               LEFT JOIN genres ON _____ \
                               LEFT JOIN invoice_items ON ______ \
                               GROUP BY _____ \
                               "
                               conn,
                               index_col='GenreId'
)
```

Question:

1. What are the top 5 genres that generated the most profit?

```
## Your code below
```

WHERE statements

We've seen how to use do some of the most common SQL operations this far. In particular, we have:

- Learned how to write `SELECT` statements

- Use `index_col` in the `pd.read_sql_query()` method

- SQL Join operations
- Use SQL Aliases

In the following example, we'll look at one more technique in the SQL arsenal: the `WHERE` clause

A `WHERE` clause is followed by a **condition**. If we want to query for all invoices where country of the billing address is Germany, we can add a `WHERE` clause to our sql query string:

```
germany = pd.read_sql_query("SELECT * FROM invoices WHERE BillingCountry = 'Germany'", conn)
germany.head()
```

`WHERE` conditions can be combined with `IS`, `AND`, `OR` and `NOT`. Supposed we want to create a DataFrame containing all invoices where the billing country is **not** Germany, we can do the following:

```
not_germany = pd.read_sql_query("SELECT * FROM invoices WHERE BillingCountry IS NOT 'Germany'", conn)
not_germany.head()
```

Do note that the `IS` operator is the same as its mathematical notation counterpart: `=`. Similar with most programming language, it also supports other mathematical operator such as `>`, `>=`, `<`, and `<=`.

Now let's try another approach by using `IN` operator that enables us to specify multiple values for comparison. For example we'd like to retrieve all invoices from `Canada` and `USA`:

```
america = pd.read_sql_query("SELECT * FROM invoices WHERE BillingCountry IN ('USA', 'Canada')", conn)
america.head()
```

Knowledge Check Edit the following code to include a `WHERE` clause. We want the returned DataFrame to contain only the `Pop` genre and only when the `UnitPrice` of the track is 0.99:

```
popmusic = pd.read_sql_query("SELECT tracks.*, genres.Name as GenreName \
                              FROM tracks \
                              LEFT JOIN genres ON _____ \
                              WHERE genres.Name = ____ AND _____,
                             conn,
                             index_col='TrackId'

)
```

Question:

1. How many rows are there in `popmusic`?

```
## Your code below


## -- Solution code
```

Operating Dates

Continuing from the last `WHERE` statements, we can retrieve all invoices billed to the country `Germany`. However, it is also common to perform a conditional query statement to retrieve a specific data range. Let's take a look at our `germany`data frame for example:

```
germany.dtypes
```

Notice how our `InvoiceDate` is listed as `Object` types. The `pd.read_sql_query` behaves like `pd.read_csv` where, by default, it reads data as numeric and object. This doesn't necessarily means the database is stored using string format (commonly known as `VARCHAR` in SQL databases). Take a look at the following table schema:

```
invoices_table = pd.read_sql_query("SELECT sql FROM sqlite_master \
                                    WHERE name = 'invoices'", conn)
print(invoices_table.loc[0,:].values[0])
```

Note that according to different DBMS you are using, there are different ways to retrieve a table's schema. The query above is used to retrieve table schema from SQLite database. The `DATETIME` type is stored with the following format: `YYYY-MM-DD HH:MI:SS`.

It is often useful to understand the table schema of your database so you can perform the appropriate operation. Within our `invoices`' schema you can see some of useful information such as:

- `InvoiceId` is listed as a primary key
- `InvoiceDate` is stored as `DATETIME`
- `CustomerId` is registered as foreign key to `customers` table

If you are not provided with the database's schema, take some time to study each table schema. Now consider the following case: We are reviewing Germany market of the last year sales and would like to retrieve all invoices from the year 2012.

```
germany_2012 = pd.read_sql_query("SELECT * FROM invoices \
                                  WHERE InvoiceDate >= '2012-01-01' AND InvoiceDate <= '2012-12-31'",
                                 con=conn, parse_dates='InvoiceDate')
germany_2012['InvoiceDate'].describe()
```

Now there are other common approach using the `BETWEEN` operator, try and copy-paste the following code to a code cell:

```
germany_2012 = pd.read_sql_query("SELECT * FROM invoices \
                                    WHERE InvoiceDate BETWEEN \
                                    '___' AND '___'",
                                  con=conn, parse_dates='InvoiceDate')
germany_2012['InvoiceDate'].describe()
```

Try completing the code above and see if the query fetch the same result as the previous one:

```python
# Your code below
```

Using `LIKE` Operator

Using a `WHERE` statement in a query can be beneficial to pull relevant data for our analysis. A common operator for a WHERE statement is `LIKE`. Consider the following SQL Query:

```python
germany = pd.read_sql_query("SELECT * FROM invoices \
                             WHERE BillingCountry = 'Germany' \
                             AND BillingPostalCode LIKE '107%'", conn)
germany.head()
```

A `LIKE` operator is a can be used to match a certain part of the data and can be really useful if we need to perform a partial matching to a specific value rather than using an equal-to operator. The `107%` you see in the query means to extract value in `BillingPostalCode` that starts with the number 107. This is really helpful when you are aiming to extract data specifically on specific region. In Germany, you would know that Wilmersdorf and Tempelhof in Berlin has postal code starting with 107.

Discussion:

If you were to put `%` on the end, you are matching everything that starts with 107, if you put it on the start `%107`, means you are matching everything that ends in 107. What do you think will came up if you use `%` before and after the pattern match?

Dive Deeper: Let's take a look at another case, consider the following data frame:

```python
customerinv = pd.read_sql_query("SELECT firstname, lastname, email, company, \
                                 invoiceid, invoicedate, billingcountry, total \
                                 FROM invoices \
                                 left join customers \
                                 on invoices.customerId = customers.customerId", conn)
customerinv.head()
```

Within the first 6 data, you could see the `Company` column may not be reliable since most of them are filled with None. But if you pay close attention to the `Email` column, you could see some people have an email domain at `apple`, that could be an indicator of their company.

1. How would your conditional `WHERE` statement would be like if you want to count the number of customers that are working at Apple Inc.?

   Try to complete the following codes:

```
applecust = pd.read_sql_query("SELECT firstname, lastname, email, company, \
                               invoiceid, invoicedate, billingcountry, total \
                               FROM invoices \
                               left join customers \
                               on invoices.___ = customers.___ \
                               WHERE email like '___'", conn)

)
```

```
## Your code below


## -- Solution code
```

Based on the data queried, how many of the customers is working at Apple Inc.?

- ☐ 412
- ☐ 49
- ☐ 7
- ☐ 14

SQL Subquery

In some cases, you'd like to fill in some value for the query from the database itself as one of the condition. For example, recall how we retrieved all customers that has the most total invoice in the previous exercise. Say from the information, we'd like to retrieve all the top customers invoice. To do that we will construct a `WHERE` statement using `IN` operator utilizing a subquery to retrieve all top customers:

```
customerinv = pd.read_sql_query("SELECT invoices.*  \
                                 FROM invoices \
                                 WHERE invoices.CustomerId IN ( \
                                 SELECT c.CustomerId FROM Customers as c \
                                 LEFT JOIN invoices as i on i.CustomerId = c.CustomerId \
                                 GROUP BY c.CustomerId \
                                 ORDER BY SUM(Total) DESC LIMIT 10)", conn)
customerinv.head()
```

Notice the subquery is defined as follow:

```
SELECT c.CustomerId FROM Customers as c
LEFT JOIN invoices as i on i.CustomerId = c.CustomerId
GROUP BY c.CustomerId
ORDER BY SUM(Total) DESC LIMIT 10
```

If used on its own, the query will retrieve the top 10 customer's IDs based on the total summation of their purchases. This when then used as the condition using `IN` statement on the previous query. The `IN`, however, can also be used using a hard-coded value like the following:

```
customerinv = pd.read_sql_query("SELECT *  \
                                 FROM invoices \
                                 WHERE InvoiceId IN (46, 175, 198)", conn)
customerinv.head()
```

Knowledge Check Imagine you're being instructed to analyze invoices that consist of a fair amount of tracks within one purchase. Previously, you have known that within one invoice the users would see a total of 1 to 10 tracks per purchase. Using subquery techniques you have learned, create an SQL query that fetched all invoice along with its total tracks quantity of bigger than 10. Complete the following query:

```
SELECT *,
(SELECT ___ FROM invoice_items GROUP BY ___) as Quantity
FROM invoices
WHERE Quantity > 10
```

```
## Your code below


## -- Solution code
```

Under and Over Fetching

Now amidst all the tools selection available for data analysis, you will now ponder which one is better suited for you. To review let's recall what we have learned about Python features:

- Reading flat files (csv / sas files)
- Data cleansing and wrangling
- Exploratory analysis tools
- Visual exploratory tools

Now where does SQL fits in? First you will need to understand client-server architecture.

A bit different with Python when all your data operation is done on your local computer. When you worked with SQL, most likely you have a relational database stored remotely from your machine, usually a centralized database accessible to some clients.

When you perform a query, you execute a command to download the data to your computer. This downloading process require resources, and you need to effectively utilize the tools in order to minimize the overall cost.

Discussion:

You are instructed to perform an analysis for all `Rock` genre sales on the last year (2012). Consider this questions:

- Is it necessary for you to download all tracks table to your computer?
- Will you filter the `Rock` genre tracks using SQL WHERE statements or conditional subsetting using Python?
- Since we need the information of multiple tables, which one is more convenient, querying a joined table or two separate tables from the database?

Try if you can construct your most optimum query below:

```
## Your code below
```

Learn-by-Building

The following learn-by-building exercise will guide you through the process of building out a simple analysis along with some accompanying charts. This module is considerably more difficult than similar exercise blocks in the past, but it sure is a lot more rewarding!

Let's try by first constructing a DataFrame using the `read_sql_query()` method that we've grown familiar to. We want to develop a simple sales visualization report of our top 5 key markets (`Country` column in `customers`) ranked by Sales (`Total` column in `invoices`).

We also want to identify our top 5 customers by name (`FirstName`, `LastName`) in the report.

Last but not least, we want the report to include a day-of-week analysis on sales performance, and for that we will need the `InvoiceDate` column.

> **Hint 1**: `pandas` has built-in methods of extracting the name of day in a week. We've seen this in Part 2 of this specialization (**Working with Datetime chapter**). An example usage is:
>
> `x['InvoiceDOW'] = x['InvoiceDate'].dt.weekday_name`
>
> **Hint 2**: In `read_sql_query`, you can use the `parse_dates='InvoiceDate'` argument to have the specified column parsed as date, saving you from a `to_datetime()` conversion

```
## Your code below


## -- Solution code
```