

SoleSociety Sneaker Boutique Database ETL Migration

SQL/Python Data Engineering



Ricky Camilo
October 2023

Table of Contents

- I. Introduction - What is SoleSociety (Business Use Case)
- II. Getting to Know the Current Data Model
- III. Planning ETL pipeline
- IV. Python Pipeline (psycpg2)
- V. PostgreSQL, PGAdmin4, SQL Queries
- VI. Conclusion

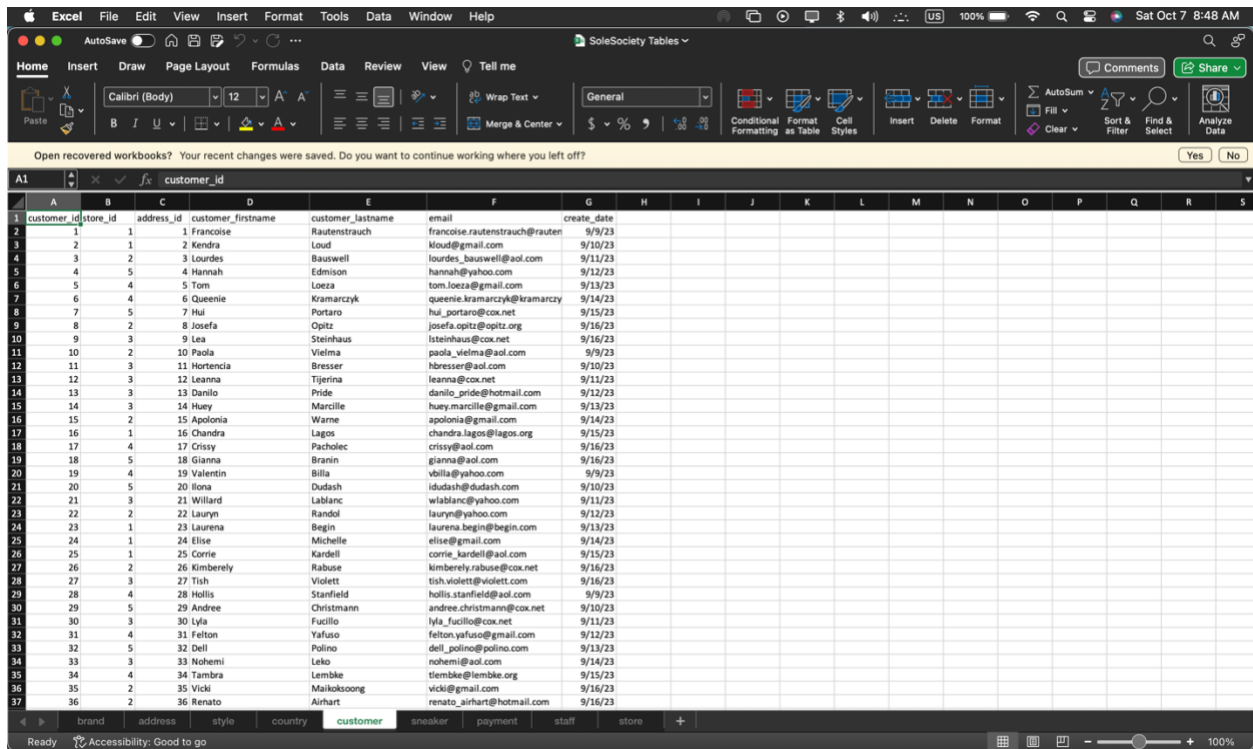
Introduction

SoleSociety is an up-and-coming online sneaker boutique with small warehouse locations spread across a few countries. They have seen enough business growth to warrant moving their business data from a flat spreadsheet with multiple tabs to a database. This project aims to streamline and optimize the data management process for SoleSociety. By transferring their business data from a local spreadsheet to a PostgreSQL database. Upon completion, this ETL project will provide us with enhanced data reliability, improved data access for faster decision-making, scalability to accommodate our growth, and the capability to perform advanced analytics.

We will use python packages **psycopg2** and **pandas** to create a database for SoleSociety as well as extract the data from its native excel format, transform the data as to correct any errors/ clean it up, and then load the data onto the tables that will comprise the SoleSociety database. We will then run through a few SQL queries to answer some business questions that the SoleSociety admin team have.

Getting to Know the Current Data Model

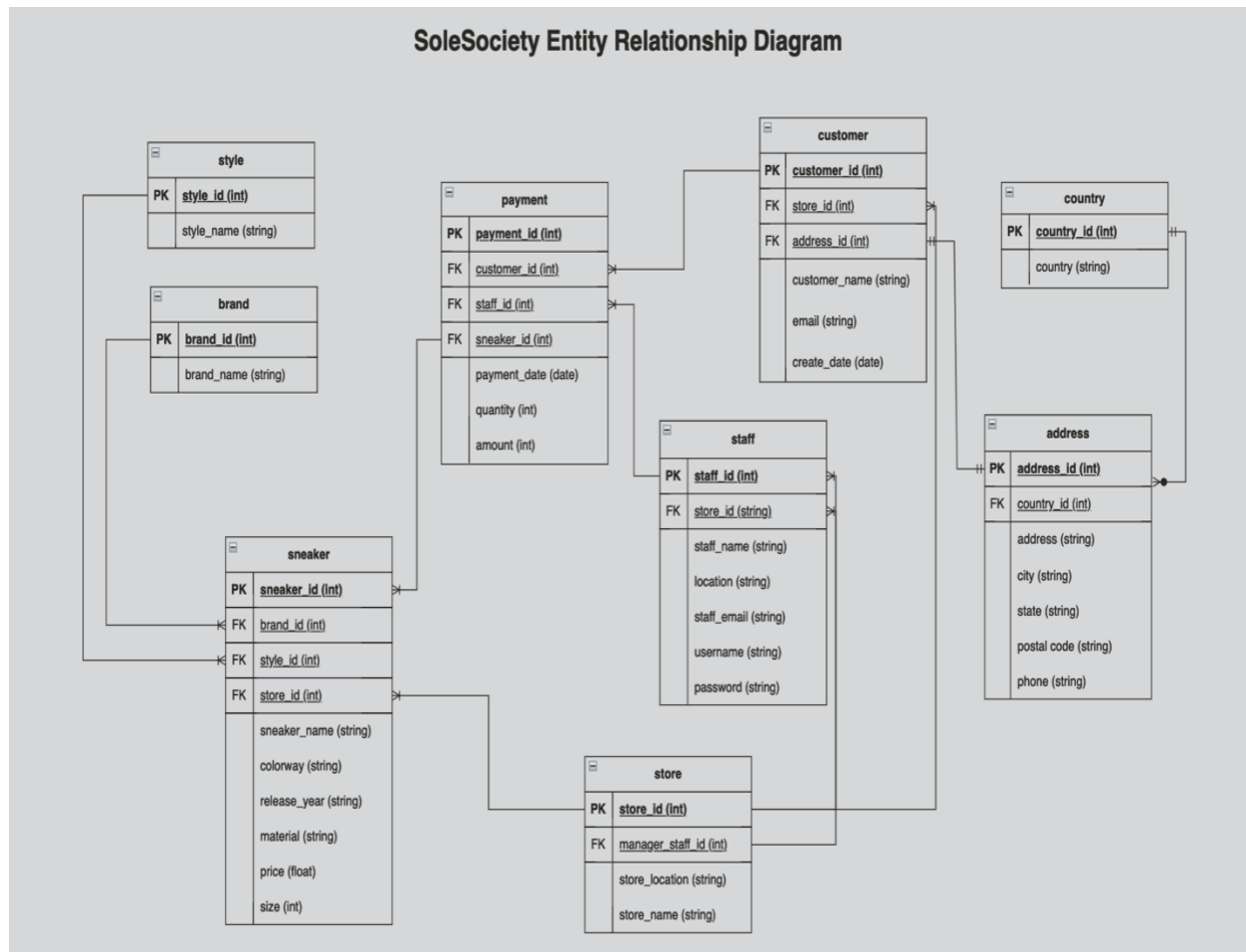
Currently the data is stored in an Excel spreadsheet. Since the business was small an Excel document maintained by a handful of employees was enough to meet demand although we can see how this can become problematic as the business expands. We have the file saved locally on our desktop.



customer_id	store_id	address_id	customer_firstname	customer_lastname	email	create_date
1	1	1	Francoise	Rautenstrauch	francoise.rautenstrauch@rauten	9/9/23
2	1	2	Kendra	Loud	kloud@gmail.com	9/10/23
3	2	3	Lourdes	Bauswell	lourdes_bauswell@aol.com	9/11/23
4	5	4	Hannah	Edmison	hannah@yahoo.com	9/12/23
5	4	5	Tom	Loeza	tom.loeza@gmail.com	9/13/23
6	4	6	Queenie	Kramarczyk	queenie.kramarczyk@kramarczy	9/14/23
7	5	7	Hui	Portaro	hui_portaro@cox.net	9/15/23
8	2	8	Josefa	Opitz	josefa.opitz@opitz.org	9/16/23
9	3	9	Lea	Steinhaus	lsteinhaus@cox.net	9/16/23
10	2	10	Paola	Vielma	paola_vielma@aol.com	9/9/23
11	3	11	Hortencia	Bresser	hbresser@aol.com	9/10/23
12	3	12	Leanna	Tjerina	leanna@cox.net	9/11/23
13	3	13	Omelio	Pride	danilo_pride@hotmail.com	9/12/23
14	3	14	Huey	Marcille	huey.marcille@gmail.com	9/13/23
15	2	15	Apollonia	Warne	apolonia@gmail.com	9/14/23
16	1	16	Chandra	Lagos	chandra.lagos@lagos.org	9/15/23
17	4	17	Crissy	Pacholec	crissy@aol.com	9/16/23
18	5	18	Gianna	Branin	gianna@aol.com	9/16/23
19	4	19	Valentin	Billa	vbilla@yahoo.com	9/9/23
20	5	20	Iiona	Dudash	idudash@dudash.com	9/10/23
21	3	21	Willard	Lablanc	wlablanc@yahoo.com	9/11/23
22	2	22	Lauryn	Randol	lauryn@yahoo.com	9/12/23
23	1	23	Laurena	Begin	laurena.begin@begin.com	9/13/23
24	1	24	Elise	Michelle	elise@gmail.com	9/14/23
25	1	25	Corrie	Kardell	corrie.kardell@aol.com	9/15/23
26	2	26	Kimberely	Rabuse	kimberely.rabuse@cox.net	9/16/23
27	3	27	Tish	Violet	tish.violet@violet.com	9/16/23
28	4	28	Hollis	Stanfield	hollis.stanfield@aol.com	9/9/23
29	5	29	Andree	Christmann	andree.christmann@cox.net	9/10/23
30	3	30	Lyla	Fucillo	lyla_fucillo@cox.net	9/11/23
31	4	31	Felton	Yafuso	felton.yafuso@gmail.com	9/12/23
32	5	32	Delf	Polino	delf_polino@polino.com	9/13/23
33	3	33	Nohemi	Leko	nohemi@aol.com	9/14/23
34	4	34	Tambra	Lembke	tiembke@tiembke.org	9/15/23
35	2	35	Vicki	Maikokoong	vicki@gmail.com	9/16/23
36	2	36	Renato	Airhart	renato_airhart@hotmail.com	9/16/23

Looking at the data it looks like it is already set up so that each tab can be its own table. But the tables need to be relational in some way for it to work in a database. I created this Entity Relationship Diagram that shows our database tables are going to connect. I had to add a primary key column and foreign keys if they were necessary using VLOOKUP formulas to pull in the associated foreign key for each record. The tabs are essentially tables themselves meaning that when we pull in the source data into **psycopg2** we will need to pull in each tab as an individual

data frame to then load into PostgreSQL.



Planning ETL Pipeline

An ETL pipeline is a set of processes that extract, transform, and load data from one or more sources to a destination, typically a data warehouse, database, or other data storage and processing systems.

Extract: This phase involves pulling data from various sources, which could include databases, applications, files, APIs, or other systems. The goal is to gather raw data from these sources.

Transform: In this phase, the extracted data undergoes transformation processes to clean, filter, aggregate, or otherwise modify it. Transformation is crucial for ensuring that the data is in a consistent and usable format for analysis or reporting.

Load: The transformed data is then loaded into the target destination, which is often a data warehouse or database. This step involves organizing the data in a structure that facilitates efficient querying and analysis.

The primary objectives of an ETL pipeline are to centralize and standardize data, making it accessible and useful for business intelligence, reporting, and analytics. ETL pipelines are commonly used in data warehousing, business intelligence, and data integration scenarios, where data from diverse sources needs to be consolidated and processed.

In our case we are building an ETL pipeline that extracts data from a multi-tab spreadsheet which is then transformed and cleaned up a bit using Python's pandas and **psycopg2** packages and, finally loaded into a Postgres database.

Python Pipeline (psycopg2)

You can use your IDE of choice I prefer Visual Studio Code. First begin by pip installing pandas and openpyxl.

```
pip install pandas --upgrade  
pip install openpyxl --upgrade
```

Next install and import pandas and psycopg2

```
install pandas as pd  
install psycopg2
```

```
import pandas as pd
import psycopg2
```

openpyxl is a Python library for reading and writing Excel (xlsx) files. It allows you to work with Excel files in a programmatic way, enabling you to automate tasks such as creating, modifying, and extracting data from Excel spreadsheets.

pandas is an open-source data manipulation and analysis library for Python. It provides data structures for efficiently storing large datasets and tools for working with them. Pandas is particularly well-suited for working with structured data, such as tables or spreadsheets.

psycopg2 is a PostgreSQL adapter for the Python programming language. It is a PostgreSQL database connector that allows Python programs to communicate with PostgreSQL databases, enabling the execution of SQL queries, data retrieval, and database management tasks.

PostgreSQL, often called Postgres, is a robust open-source relational database management system known for its extensibility and SQL standards adherence. It excels in handling complex data types and offers advanced features like transactions. On the other hand, pgAdmin is an open-source graphical administration tool tailored for PostgreSQL. With a user-friendly interface, pgAdmin enables users to connect to databases, execute SQL queries, manage database objects, and perform various administrative tasks seamlessly.

Next, we want to see if we can connect with my already existing Postgresql database to test our connection code out, so we create a function that connects to my local instance of Postgres. Then we will CREATE a new database called solesociety to load our data into.

```
# creating a function that connects to our original database to test our connection to postgres

def create_database():
    # connect to default database
    conn = psycopg2.connect("host=localhost dbname=postgres user=#### password=####")
```

```

conn.set_session(autocommit=True)
cur = conn.cursor()

# create sparkify database with UTF8 encoding
cur.execute("DROP DATABASE IF EXISTS solesociety")
cur.execute("CREATE DATABASE solesociety")

# close connection to default database
conn.close()

# connect to solesociety database
conn = psycopg2.connect("host=localhost dbname=solesociety user=#### password=####")
cur = conn.cursor()

return cur, conn

```

running this will create the solesociety database

```
cur, conn = create_database()
```

Now we want to drop that table we created with the above cur, conn variable to start fresh.

```

# function that drops tables

def drop_tables(cur, conn):
    for query in drop_table_queries:
        cur.execute(query)
        conn.commit()

```

Now we recreate and reconnect to set up our environment to pull the actual data in.

```

# function that creates tables

def create_tables(cur, conn):
    for query in create_table_queries:
        cur.execute(query)
        conn.commit()

```



```

# EXTRACT

# pulling each sheet of the solesociety data spreadsheet as its df so we can easily insert into tables later

style_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="style")

brand_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="brand")

customer_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="customer")

country_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="country")

address_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="address")

store_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="store")

staff_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="staff")

payment_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="payment")

sneaker_df = pd.read_excel("/Users/rickycamilo/Desktop/PostgreSQL Bootcamp and SoleSociety ETL
Project/SoleSociety ETL Migration SQL Project/SoleSociety Tables.xlsx", sheet_name="sneaker")

```

Using pandas we pull in each individual tab into its own dataframe using `pd.read.excel` function.

Now we test to see if our data was pulled in correctly by checking the `head()` of the customer and staff dataframes.

```
# testing that we pulled our data into dfs successfully
```

```
customer_df.head()
```

```
staff_df.head()
```

...	customer_id	store_id	address_id	customer_firstname	customer_lastname	email	create_date
	0	1	1	Francoise	Rautenstrauch	francoise.rautenstrauch@rautenstrauch.com	2023-09-09
	1	2	1	Kendra	Loud	kloud@gmail.com	2023-09-10
	2	3	2	Lourdes	Bauswell	lourdes_bauswell@aol.com	2023-09-11
	3	4	5	Hannah	Edmison	hannah@yahoo.com	2023-09-12
	4	5	4	Tom	Loeza	tom.loeza@gmail.com	2023-09-13

	staff_id	store_id	staff_name	location	staff_email	username	password
0	1	1	Elly Morocco	NYC	elly_morocco@solesociety.com	SSXC0001	rt344!23
1	2	1	Ilene Eroman	NYC	ilene.eroman@solesociety.com	SSXC0002	Bwmm6AuiYQaYZ6u
2	3	1	Vallie Mondella	NYC	vmondella@solesociety.com	SSXC0003	Bda4UmDziWn2C74
3	4	1	Kallie Blackwood	NYC	kallie.blackwood@solesociety.com	SSXC0004	JYIoGEuZ
4	5	1	Johnetta Abdallah	NYC	johnetta_abdallah@solesociety.com	SSXC0005	k,e3Tj0

Data looks like it was pulled in nicely and structured, although going forward I can foresee the `customer_firstname` and `customer_lastname` columns causing an issue of redundancy in the future. We will combine these 2 columns to minimize clutter and since each customer is uniquely identified anyway. Combining these 2 columns into 1 is a good idea. Being that now the customer dataframe will have 3 columns since we will be combining the `customer_firstname` and `customer_lastname` columns into a 3rd column. We will also only include the combined name column in our dataframe and later in our database table.

```
# TRANSFORM
```

```
# concatenating the customer_firstname and customer_lastname columns in the staff table for later insertion into postgres database
```

```
# concatenating the staff_firstname and staff_lastname columns in the staff table for later insertion into postgres database
```

```
customer_df["customer_name"] = customer_df["customer_firstname"] + " " + customer_df["customer_lastname"]
```

```
staff_df["staff_name"] = staff_df["staff_firstname"] + " " + staff_df["staff_lastname"]
```

```
# selecting only the the columns I want to insert into postgres
customer_df = customer_df[["customer_id", "store_id", "address_id", "customer_name", "email", "create_date" ]]
staff_df = staff_df[["staff_id", "store_id", "staff_name", "location", "staff_email", "username", "password" ]]
```

Next up, we will create a table for each dataframe to eventually populate indicating the data type and what field will serve as the primary key.

```
# creating table and executing to postgres

style_table_create = ("""CREATE TABLE IF NOT EXISTS style(
style_id INT PRIMARY KEY,
style_name TEXT)
""")

cur.execute(style_table_create)
conn.commit()
```

```
brand_table_create = ("""CREATE TABLE IF NOT EXISTS brand(
brand_id INT PRIMARY KEY,
brand_name TEXT)
""")

cur.execute(brand_table_create)
conn.commit ()
```

```
sneaker_table_create = ("""CREATE TABLE IF NOT EXISTS sneaker(
sneaker_id BIGINT PRIMARY KEY ,
brand_id INT,
style_id INT,
store_id INT,
sneaker_name TEXT,
colorway TEXT,
```

```
release_year TEXT,  
material TEXT,  
price FLOAT,  
size INT )  
""")  
  
cur.execute(sneaker_table_create)  
conn.commit ()
```

```
payment_table_create = ("""CREATE TABLE IF NOT EXISTS payment(  
payment_id INT PRIMARY KEY,  
customer_id INT,  
staff_id INT,  
payment_date DATE,  
sneaker_id INT,  
quantity INT,  
amount INT)  
""")  
  
cur.execute(payment_table_create)  
conn.commit()
```

```
staff_table_create = ("""CREATE TABLE IF NOT EXISTS staff(  
staff_id INT PRIMARY KEY ,  
store_id INT,  
staff_name TEXT,  
location TEXT,  
staff_email TEXT,  
username TEXT,  
password TEXT )  
""")  
  
cur.execute(staff_table_create)  
conn.commit ()
```

```
customer_table_create = ("""CREATE TABLE IF NOT EXISTS customer(
customer_id INT PRIMARY KEY,
store_id INT,
address_id INT,
customer_name TEXT,
email TEXT,
create_date DATE)
""")

cur.execute(customer_table_create)
conn.commit()
```

```
store_table_create = ("""CREATE TABLE IF NOT EXISTS store(
store_id INT PRIMARY KEY,
manager_staff_id INT,
store_location TEXT,
store_name TEXT )
""")

cur.execute(store_table_create)
conn.commit ()
```

```
address_table_create = ("""CREATE TABLE IF NOT EXISTS address(
address_id INT PRIMARY KEY ,
address TEXT,
country_id INT,
city TEXT,
state TEXT,
postal_code TEXT,
phone TEXT )
""")
```

```
cur.execute(address_table_create)
conn.commit()
```

```
country_table_create = ("""CREATE TABLE IF NOT EXISTS country(
country_id INT PRIMARY KEY,
country TEXT
)""")

cur.execute(country_table_create)
conn.commit()
```

Now we label the column names of the tables comprising the SoleSociety database.

```
# inserting column names into solesociety database
```

```
style_df_table_insert = ("""INSERT INTO style(
style_id,
style_name)
VALUES (%s,%s)
""")
```

```
store_df_table_insert = ("""INSERT INTO store(
store_id,
manager_staff_id,
store_location,
store_name)
VALUES(%s,%s,%s,%s)
""")
```

```
brand_df_table_insert = ("""INSERT INTO brand(
brand_id,
brand_name)
VALUES(%s,%s)
```

```
""")
```

```
sneaker_df_table_insert = ("""INSERT INTO sneaker(
sneaker_id,
brand_id,
style_id,
store_id,
sneaker_name,
colorway,
release_year,
material,
price,
size)
VALUES(%s,%s,%s,%s,%s,%s,%s,%s,%s,%s)
""")
```

```
payment_df_table_insert = ("""INSERT INTO payment(
payment_id,
customer_id,
staff_id,
payment_date,
sneaker_id,
quantity,
amount)
VALUES(%s,%s,%s,%s,%s,%s,%s)
""")
```

```
staff_df_table_insert = ("""INSERT INTO staff(
staff_id,
store_id,
staff_name,
location,
staff_email,
username,
password)
VALUES(%s,%s,%s,%s,%s,%s,%s)
```

```
""")
```

```
customer_df_table_insert = ("""INSERT INTO customer(
customer_id,
store_id,
address_id,
customer_name,
email,
create_date)
VALUES(%s,%s,%s,%s,%s,%s)
""")
```

```
country_df_table_insert = ("""INSERT INTO country(
country_id,
country)
VALUES(%s,%s)
""")
```

```
address_df_table_insert = ("""INSERT INTO address(
address_id,
address,
country_id,
city,
state,
postal_code,
phone)
VALUES(%s,%s,%s,%s,%s,%s,%s)
""")
```

Now it's time to finally load the data!

We create a loop that iterates through every row of the created dataframes and hydrates the tables we just finished creating.

```
# LOAD
```



```
# loop that iterates through every row of the mentioned dataframe. this will hydrate each table we created with the designated dataframe
```

```
for i, row in style_df.iterrows():  
    cur.execute(style_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in brand_df.iterrows():  
    cur.execute(brand_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in sneaker_df.iterrows():  
    cur.execute(sneaker_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in payment_df.iterrows():  
    cur.execute(payment_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in store_df.iterrows():  
    cur.execute(store_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in staff_df.iterrows():  
    cur.execute(staff_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in customer_df.iterrows():  
    cur.execute(customer_df_table_insert, list(row))
```

```
conn.commit()
```

```
for i, row in address_df.iterrows():  
    cur.execute(address_df_table_insert, list(row))  
  
conn.commit()
```

```
for i, row in country_df.iterrows():  
    cur.execute(country_df_table_insert, list(row))  
  
conn.commit()
```

Let's check pgAdmin to see if our database, tables, and data created and hydrated successfully. We can see on the left that we have a database named solesociety and we run a simple SELECT * statement on the brand table.

The screenshot shows the pgAdmin 4 interface. On the left, the 'Object Explorer' pane displays the 'solesociety' database structure, including schemas like 'public' and 'pg_toast', and a list of tables. The 'brand' table is highlighted. The main pane shows the 'Query' editor with the SQL statement 'SELECT * FROM brand;'. Below the query editor, the 'Data Output' pane displays the results of the query as a table with 9 rows. The status bar at the bottom indicates 'Total rows: 9 of 9' and 'Query complete 00:00:00.141'.

brand_id [PK] integer	brand_name text
1	Nike
2	Adidas
3	Puma
4	Reebok
5	Ugg
6	Birkenstock
7	Jordan
8	Bape
9	Timberland

PostgreSQL, pgAdmin 4, SQL Queries

PostgreSQL (Postgres) is an open-source relational database management system known for its ACID compliance, extensibility, and support for various data types. It facilitates scalable and concurrent transactions, offers extensive data type options, and supports foreign data wrappers. On the other hand, pgAdmin is a widely used open-source administration tool for PostgreSQL, providing a graphical interface for tasks such as database connection management, SQL query execution, data manipulation, backup and restore operations, and user role management. Together, PostgreSQL and pgAdmin offer a powerful combination for building, managing, and administering relational databases with features suitable for a broad range of applications. We will run through a handful of SQL statements to answer a handful of questions that the SoleSociety admin team had (what initially prompted the need for a database)

1- ALTER statement to drop needless create_date column in the customers table.

ALTER TABLE customer DROP create_date;

SELECT * FROM customer ;

	customer_id [PK] integer	store_id integer	address_id integer	customer_name text	email text
1	1	1	1	Francoise Rautenstrauch	francoise.rautenstrauch@rautenstrauch.com
2	2	1	2	Kendra Loud	kloud@gmail.com
3	3	2	3	Lourdes Bauswell	lourdes_bauswell@aol.com
4	4	5	4	Hannah Edmison	hannah@yahoo.com
5	5	4	5	Tom Loeza	tom.loeza@gmail.com
6	6	4	6	Queenie Kramarczyk	queenie.kramarczyk@kramarczyk.org
7	7	5	7	Hui Portaro	hui_portaro@cox.net
8	8	2	8	Josefa Opitz	josefa.opitz@opitz.org
9	9	3	9	Lea Steinhaus	lsteinhaus@cox.net
10	10	2	10	Paola Vielma	paola_vielma@aol.com
11	11	3	11	Hortencia Bresser	hbresser@aol.com
Total rows: 300 of 300			Query complete 00:00:00.054		

2- CASE statement to classify sneakers sized 7 and under as children's and 8 and up as adult.

```
SELECT sneaker.sneaker_id, sneaker.sneaker_name, sneaker.size,
CASE
  WHEN sneaker.size > 7 THEN 'Adult'
  WHEN sneaker.size < 8 THEN 'Children'
  ELSE 'N/A'
END AS Size_Classification
FROM sneaker
```

	sneaker_id [PK] bigint	sneaker_name text	size integer	size_classification text
1	1	Nike Air Force One	5	Children
2	2	Nike Air Force One	5	Children
3	3	Nike Air Force One	5	Children
4	4	Nike Air Force One	6	Children
5	5	Nike Air Force One	6	Children
6	6	Nike Air Force One	7	Children
7	7	Nike Air Force One	8	Adult
8	8	Nike Air Force One	8	Adult
9	9	Nike Air Force One	8	Adult
10	10	Nike Air Force One	8	Adult
11	11	Nike Air Force One	9	Adult
Total rows: 336 of 336			Query complete 00:00:00.097	

3- Find the manager of the NYC location.

```
SELECT staff.staff_id,staff.staff_name, store.store_name, store.store_location  
FROM staff  
INNER JOIN store ON staff.staff_id = store.manager_staff_id  
  
WHERE store_location = 'NYC' ;
```

	staff_id integer 🔒	staff_name text 🔒	store_name text 🔒	store_location text 🔒
1	3	Vallie Mondella	Sole Society NYC	NYC

Total rows: 1 of 1

Query complete 00:00:00.055

4- Find the list of customers who have purchased over 2 sneakers worth over \$70 ordered descending.

```
SELECT customer.customer_name, customer.email, customer.address_id,  
payment.payment_id, payment.sneaker_id, payment.payment_date,payment.quantity,  
payment.amount  
FROM customer
```

INNER JOIN payment on customer.customer_id = payment.customer_id

WHERE payment.amount >= '70' AND payment.quantity >= '2'

ORDER BY payment.quantity DESC;

	customer_name text	email text	address_id integer	payment_id integer	sneaker_id integer	payment_date date	quantity integer	amount integer
1	Otis Luong	oluong@gmail.com	274	411	329	2022-09-15	5	
2	An Dosal	adosal@hotmail.com	272	409	327	2022-09-13	5	
3	Corrie Kardell	corrie_kardell@aol.com	25	440	270	2022-10-14	5	
4	Corrie Kardell	corrie_kardell@aol.com	25	439	269	2022-10-13	5	
5	Bong Fears	bfears@fears.org	147	438	268	2022-10-12	5	
6	Carry Ziller	carry@cox.net	146	437	267	2022-10-11	5	
7	Hollis Keomuangtai	hollis.keomuangtai@cox.net	145	436	266	2022-10-10	5	
8	Dominga Mckeon	dominga_mckeon@yahoo.com	144	435	265	2022-10-09	5	
9	Dahlia Benett	dahlia_benett@aol.com	143	434	264	2022-10-08	5	
10	Margo Rand	margo@rand.org	142	433	263	2022-10-07	5	
Total rows: 441 of 441 Query complete 00:00:00.059 Ln 34, Col 1								



5- Multiply the quantity and amount columns from the payments table to find total value and group the results by store that amount was spent in. then filter for store id's greater than.

SELECT customer.store_id, SUM(payment.amount * payment.quantity) AS total_value
FROM payment

JOIN customer on payment.customer_id = customer.customer_id

GROUP BY customer.store_id

HAVING customer.store_id > '2';

	store_id integer 	total_value bigint 
1	3	216832
2	5	139684
3	4	164071

Total rows: 3 of 3

Query complete 00:00:00.086

6- *CREATE a VIEW of clients who have made purchases greater than \$100 after start of 2023*

CREATE VIEW recent_clients AS

**SELECT customer.customer_name, customer.email, customer.address_id,
payment.payment_id, payment.sneaker_id, payment.payment_date, payment.quantity,
payment.amount**

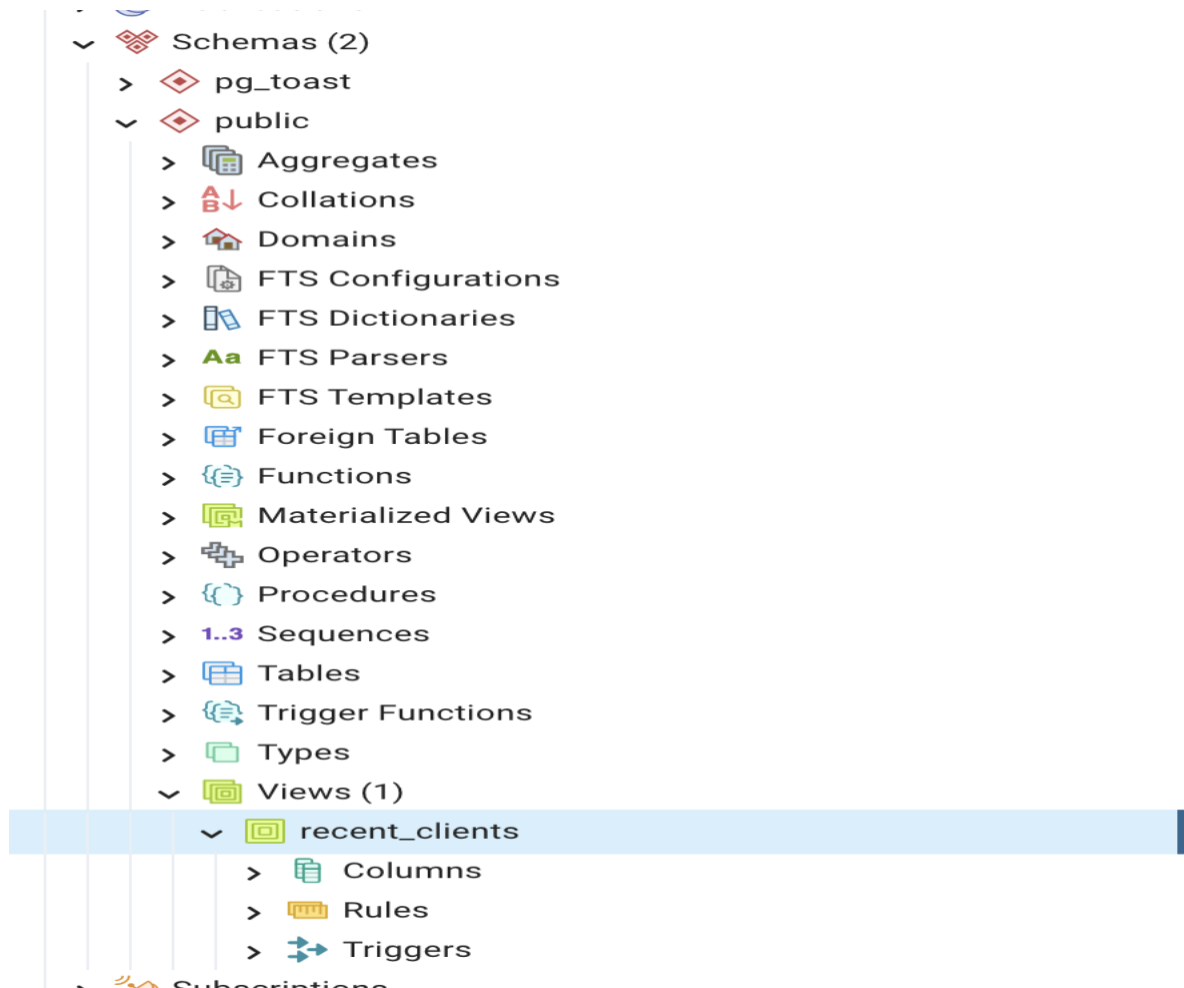
FROM customer

INNER JOIN payment on customer.customer_id = payment.customer_id

WHERE payment.payment_date >= DATE('1/1/2023') AND payment.amount >= '100';

CREATE VIEW

Query returned successfully in 98 msec.



7- Create a sub query that totals the number of purchases made by each client.

SELECT




```
customer.customer_id,  
customer.customer_name,  
(  
    SELECT COUNT(*) AS purchases_made
```



```

FROM payment
WHERE customer.customer_id = payment.customer_id
)
FROM customer;

```

	customer_id [PK] integer 	customer_name text 	purchases_made bigint 
1	1	Francoise Rautenstrauch	5
2	2	Kendra Loud	3
3	3	Lourdes Bauswell	1
4	4	Hannah Edmison	1
5	5	Tom Loeza	1
6	6	Queenie Kramarczyk	1
7	7	Hui Portaro	1
8	8	Josefa Opitz	2
9	9	Lea Steinhaus	2
10	10	Paola Vielma	2
11	11	Hortencia Bresser	2
Total rows: 300 of 300		Query complete 00:00:00.158	




8- Create a sub query that totals the number of sales by staff name

```

SELECT
    staff.staff_id,
    staff.staff_name,
    (
        SELECT SUM(payment.amount * payment.quantity) AS total_value
        FROM payment
        WHERE staff.staff_id = payment.staff_id
    )

```

FROM staff;

	staff_id [PK] integer 	staff_name text 	total_value bigint 
1	1	Elly Morocco	119650
2	2	Ilene Eroman	19445
3	3	Vallie Mondella	19770
4	4	Kallie Blackwood	21475
5	5	Johnetta Abdallah	20910
6	6	Bobbye Rhym	21150
7	7	Micaela Rhymes	21465
8	8	Tamar Hoogland	21600
9	9	Moon Parlato	22710
10	10	Laurel Reitler	22950
11	11	Delisa Crupi	23270
Total rows: 26 of 26		Query complete 00:00:00.067	

Our queries worked and answered the questions the SoleSociety admin team had!

Conclusion

In conclusion, the ETL project successfully leveraged a combination of Excel, PostgreSQL, and Python to streamline the extraction, transformation, and loading of data from a flat Excel file. The initial phase involved extracting raw data from the Excel source, utilizing the `openpyxl` library in Python to interface with Excel files. The subsequent transformation phase was carried out seamlessly with the powerful data manipulation capabilities of Pandas, allowing for efficient cleaning, restructuring, and formatting of the data. Python, serving as the scripting and orchestration language, played a central role in orchestrating the ETL processes, ensuring a smooth flow of data between Excel and PostgreSQL.

The final step of loading the transformed data into a PostgreSQL database was accomplished using the `psycopg2` library. This Python adapter for PostgreSQL facilitated the establishment of connections, execution of SQL queries, and efficient management of data transactions. The result is a well-organized and structured database that is ready for analytical processing, reporting, and decision support. We answered a few business questions using some SQL queries to showcase that our data is not only relational but compliant by ACID standards. By integrating these technologies cohesively, the ETL project not only automated and streamlined the data workflow but also established a foundation for scalable and maintainable data management practices, showcasing the synergy between Excel, PostgreSQL, and Python in modern data integration projects.