

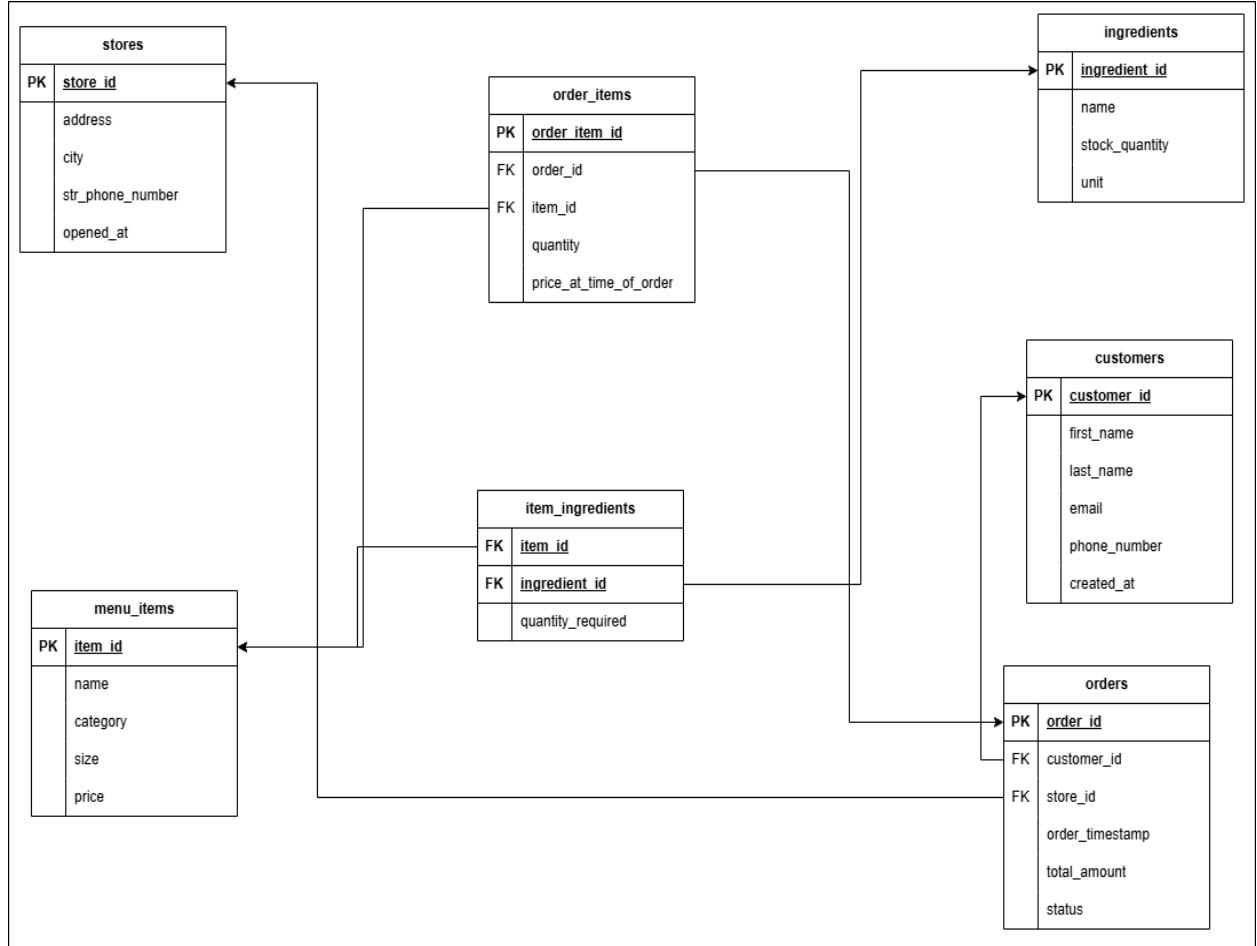
Capstone Project: Designing RushMore Pizzeria's Enterprise Database system

Objective: Design, deploy and populate a production-ready PostgreSQL database in the cloud.

Design & Deploy

Phase 1: Data Modelling

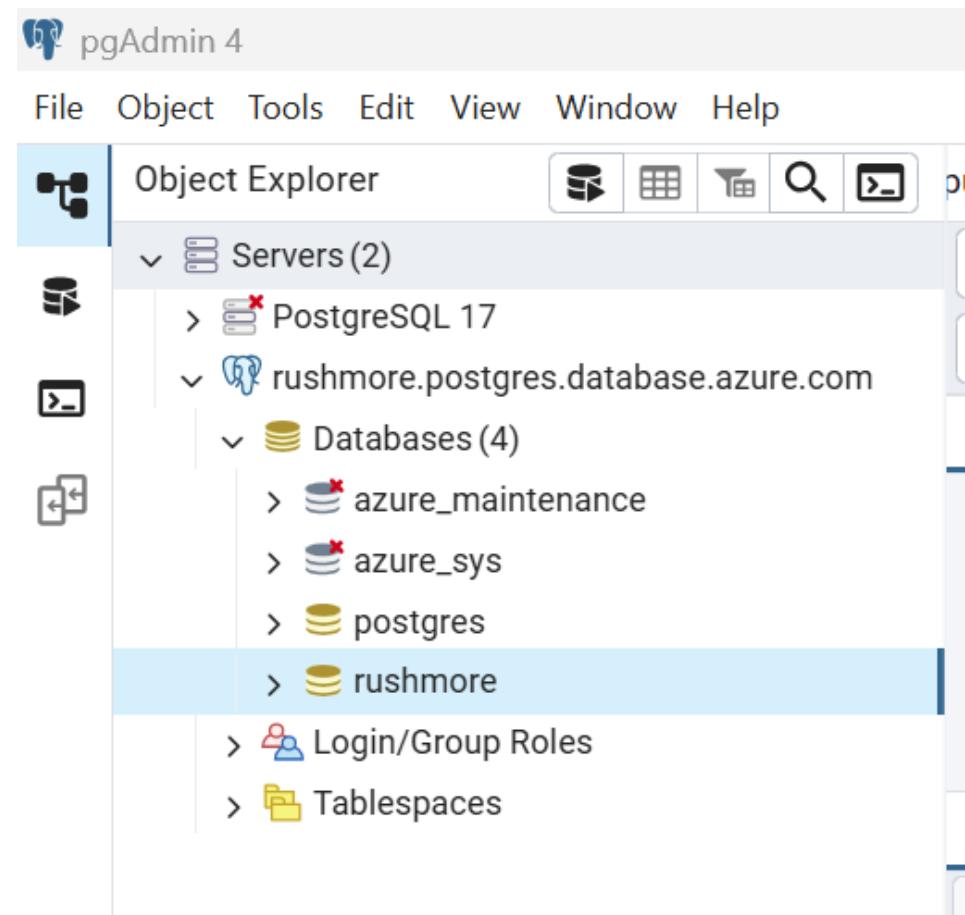
Here we had a design of a 3NF which the core idea is to organize data in our database with minimal redundancy and optimal data integrity by eliminating transitive dependencies (where non key attribute solely dependent directly on a primary key). From our project we had 7 tables to model and create a structured blueprint that would define and organize our data. I used DRAW.IO for my Entity relationship diagram to visually represent data points, their relationships, and how they can be grouped and stored to ensure data quality, consistency, and efficient access for analysis.



Design and Deployment

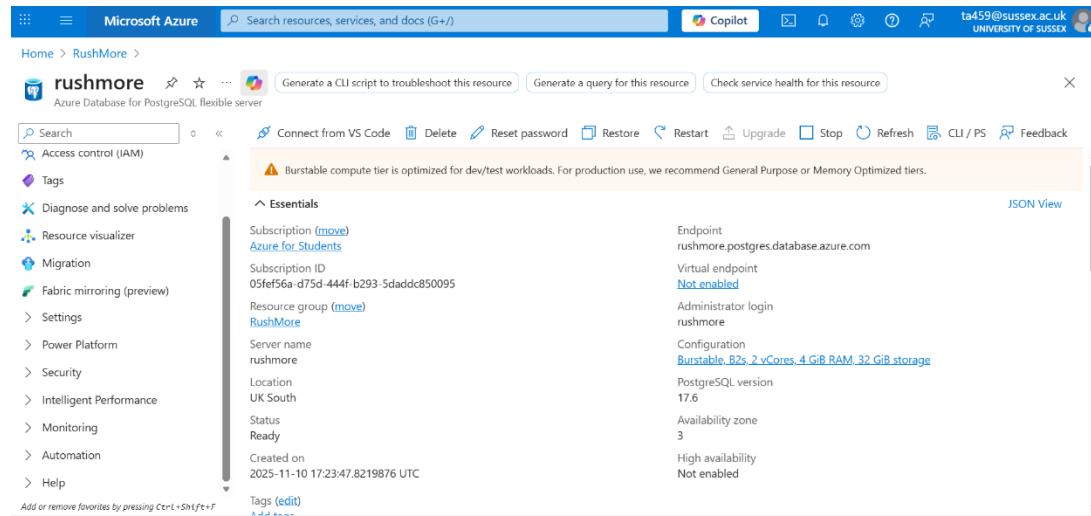
Phase 2: Deploy, Connect Azure Database for PostgreSQL Server to PgAdmin

During this phase I was able to provision PostgreSQL in Azure, which involved creating a resource group, create Azure Database for PostgreSQL and connect it with my PgAdmin on which I already created my **rushmore** database. One thing I observed during this provisioning and connecting is the fact that Azure PostgreSQL give you the option to able to connect to your visual studio code.



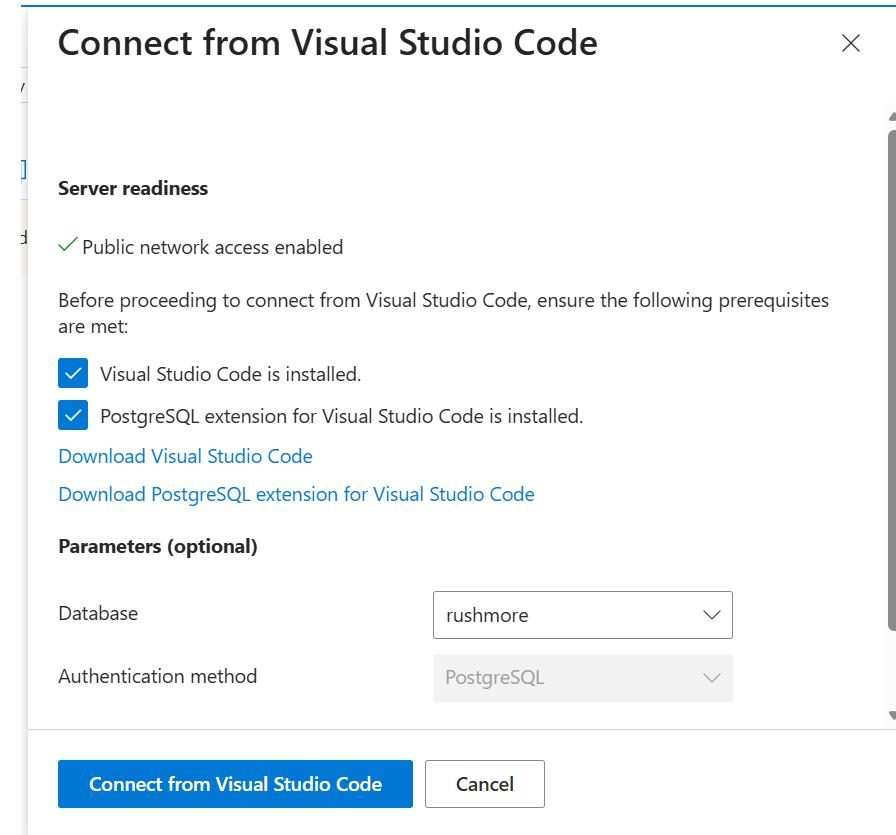
Azure Database for PostgreSQL flexible Server

Azure PostgreSQL under RushMore Resource Group



The screenshot shows the Microsoft Azure portal interface. At the top, there's a navigation bar with 'Microsoft Azure', a search bar, and user information ('ta459@sussex.ac.uk UNIVERSITY OF SUSSEX'). Below the navigation bar, the URL 'Home > RushMore > rushmore' is visible. The main content area displays the 'rushmore' PostgreSQL flexible server resource group. On the left, a sidebar lists various service links like 'Access control (IAM)', 'Tags', 'Diagnose and solve problems', 'Resource visualizer', 'Migration', 'Fabric mirroring (preview)', 'Settings', 'Power Platform', 'Security', 'Intelligent Performance', 'Monitoring', 'Automation', and 'Help'. The main panel shows the 'rushmore' server details. A warning message at the top states: '⚠️ Burstable compute tier is optimized for dev/test workloads. For production use, we recommend General Purpose or Memory Optimized tiers.' Below this, the 'Essentials' section provides detailed information: Subscription (move), Azure for Students, Subscription ID 05fef56a-d75d-444f-b293-5dadcc850095, Resource group (move), RushMore, Server name rushmore, Location UK South, Status Ready, Created on 2025-11-10 17:23:47.8219876 UTC, and Tags (edit). To the right, there's a 'JSON View' button.

Connection made to rushmore database on PgAdmin and Visual Studio Code



The screenshot shows a 'Connect from Visual Studio Code' dialog box. At the top, it says 'Connect from Visual Studio Code' and has a close button ('X'). Below this, the 'Server readiness' section includes a green checkmark next to 'Public network access enabled'. It also contains a note: 'Before proceeding to connect from Visual Studio Code, ensure the following prerequisites are met:'. Two blue checkmarks are present: 'Visual Studio Code is installed.' and 'PostgreSQL extension for Visual Studio Code is installed.'. Below these, there are two download links: 'Download Visual Studio Code' and 'Download PostgreSQL extension for Visual Studio Code'. The 'Parameters (optional)' section contains fields for 'Database' (set to 'rushmore') and 'Authentication method' (set to 'PostgreSQL'). At the bottom, there are 'Connect from Visual Studio Code' and 'Cancel' buttons.

Design: Writing Schema to .sql

Phase 3: Rushmore.sql

Proceeded to Creating my 7 tables: Stores, Customer, Ingredients, Menu_Items, Item_Ingredients, Orders, and Order_Items.

Defined primary keys, foreign keys, Uniqueness, defaults, and delete behaviors

Key design choices:

1. Junction tables

-Item_ingredients links recipes(Menu Item - Ingredient).

-Order-Items links transactions(Order-Menu Item).

2. Delete rules

-Orders.customer_id ON DELETE SET NULL: Keep historical orders even if a customer is removed.

Building the database

-Order_Items.order_id ON DELETE CASCADE: delete line items when an order is deleted(no orphans)

-Item_Ingredient.items_id ON DELETE CASCADE: delete recipe rows when a menu item is removed.

-ON DELETE RESTRICT: to protect historical facts(e.g cant delete a store that has Orders).

3. Data Types & defaults

-Money as NUMERIC(10, 2)

-Timestamps default to CURRENT_TIMESTAMP

-Natural keys avoided; we use surrogate SERIAL PKs for simplicity and performance.

4. Normalization

-No duplicated customer/store data in orders; just FKs.

-Menu catalog and ingredient catalog separated from transactions.

Indexes.sql

What it does

1. Adds secondary indexes to speed up the most common joins and filters used
 - Orders (store_id, order_timestamp) –revenue by store over time, busiest hours.
 - Order_Items (order_id) & (item_id) – fast join from orders to lines to catalog.
 - Item_Ingredients (ingredient_id) – reverse lookup of which items use an ingredient.
 - Menu_Items (category, size) – quick slicers on catalog

2. Fk columns are often required, these indexes make those joins efficient.

Our indexes speed reads(queries, dashboards), joins and time filters become index-friendly, so visuals render fast.

```
indexes.sql > ...
PGSQL Disconnected
1 -- indexes.sql
2 -- Secondary indexes for FK lookups and common analytics filters
3
4 BEGIN;
5
6 -- ITEM_INGREDIENTS lookups by ingredient
7 CREATE INDEX IF NOT EXISTS idx_item_ingredients_ingredient_id
8 | ON Item_Ingredients (ingredient_id);
9
10 -- ORDER_ITEMS lookups by order and by item
11 CREATE INDEX IF NOT EXISTS idx_order_items_order_id
12 | ON Order_Items (order_id);
13 CREATE INDEX IF NOT EXISTS idx_order_items_item_id
14 | ON Order_Items (item_id);
15
16 -- ORDERS filtering by store + time (useful for revenue by store, busy hours)
17 CREATE INDEX IF NOT EXISTS idx_orders_store_timestamp
18 | ON Orders (store_id, order_timestamp);
19
20 -- MENU_ITEMS filtering/grouping by category and size
21 CREATE INDEX IF NOT EXISTS idx_menu_items_category_size
22 | ON Menu_Items (category, size);
23
24 COMMIT;
```

Constraints.sql – “Keep bad data out”

What it does

1. Adds CHECK constraints to enforce business-valid values:
 - Prices and totals ≥ 0 .
 - Quantities > 0 .
 - Order **status** limited to a known set (Pending, In progress, Delivered, Cancelled).
2. Complements schema NOT NULL/ UNIQUE rules so invalid data cannot be inserted even by mistake or buggy code.

Why it matters

Constraints are guardrails at the database layer, they protect data quality regardless of the client app or script. This makes analytics trustworthy.

Note: Our constraints.sql hardens the model in the sense that numbers must be sensible, statuses valid, so we trust every chart.

```
constraints.sql > ...
PGSQL Disconnected
1 BEGIN;
2
3 -- Non-negative money values
4 ALTER TABLE Menu_Items
5 | ADD CONSTRAINT menu_items_price_nonneg CHECK (price >= 0);
6
7 ALTER TABLE Orders
8 | ADD CONSTRAINT orders_total_nonneg CHECK (total_amount >= 0);
9
10 ALTER TABLE Order_Items
11 | ADD CONSTRAINT order_items_price_nonneg CHECK (price_at_time_of_order >= 0);
12
13 -- Positive quantities
14 ALTER TABLE Order_Items
15 | ADD CONSTRAINT order_items_qty_pos CHECK (quantity > 0);
16
17 -- Reasonable recipe quantities
18 ALTER TABLE Item_Ingredients
19 | ADD CONSTRAINT item_ing_qty_pos CHECK (quantity_required > 0);
20
21 -- Status whitelist (optional; keep only the statuses you use)
22 ALTER TABLE Orders
23 | ADD CONSTRAINT orders_status_valid CHECK (status IN ('Pending','In Progress','Delivered',
24
25 -- Basic phone/country constraints (optional; leave blank)
```

Access_role_based.sql

Role based access control(**RBAC**) script was created which includes an admin role, data engineer, data analyst, data scientist, customer service and self service with clear privilege boundaries between them. All user have login + password. My script is safe for production, works in PostgreSQL and follows least-privilege design.

Admin: Controls the full schema, they can create tables, alter structures, add constraints, and manage privilege. It is the highest non super user role.

Data Engineer: maintains pipelines and ETL processes, they have full read/write access to all production tables but cannot change the schema.

Data Analyst: Analyst are read only. They can query all tables and views but cannot change any data.

Data Scientist: Scientist can read production data and have a dedicated sandbox schema where they can build models, run experiments, and create tables safely.

Customer Service Processor: CS staff can update customer profiles and manage orders, but they cannot modify catalog or delete history.

Self Service Role: This role supports a customer-facing application. It can create customers and place orders but cannot modify existing records.

Note: I followed security best practices by keeping real credentials out of source control. The repository includes a **role_template.sql** file with placeholder passwords, an the real **acess_role_based.sql** file is excluded using **.gitignore**, ensuring no secrets are exposed on **Github**.

Validation.sql – “Prove it worked” – Run after populate.py

What it does

1. Row-count dashboard: quick volume sanity checks after Faker runs.
2. Orphan checks: LEFT JOIN tests that should return **0** (e.g., Order_Items must always match an Order and a Menu_Item; Orders must match a Store; etc.).
3. Totals reconciliation: verifies Orders.total_amount == SUM(quantity * price_at_time_of_order) – our critical accounting check.
4. Uniqueness scans: confirms there aren't duplicate emails/phones/ingredient names beyond what the schema allows.

Results

1. All orphan checks returned **0**.
2. Totals reconciliation returned no rows(if it did, those are the orders to fix).
3. Row count meet the target volumes.

table	rows
Stores	4
Customers	1200
Ingredients	45
Menu_Items	25
Item_Ingredients	110
Orders	6000
Order_Items	15230

orphan_order_items_without_order
0

.sql continued

Cascading In Our `rushmore.sql` file

You might want to ask why I don't have a `cascading.sql` file:

My `Rushmore.sql` already includes cascading where appropriate. Looking at my `rushmore.sql` where I defined relationship like this:

```
order_id INTEGER REFERENCES Orders(order_id) ON  
DELETE CASCADE
```

```
item_id INTEGER REFERENCES Menu_Items(item_id) ON  
DELETE CASCADE
```

```
customer_id INTEGER REFERENCES  
Customers(customer_id) ON DELETE SET NULL
```

.

```
store_id INTEGER REFERENCES Stores(store_id) ON  
DELETE RESTRICT
```

So:

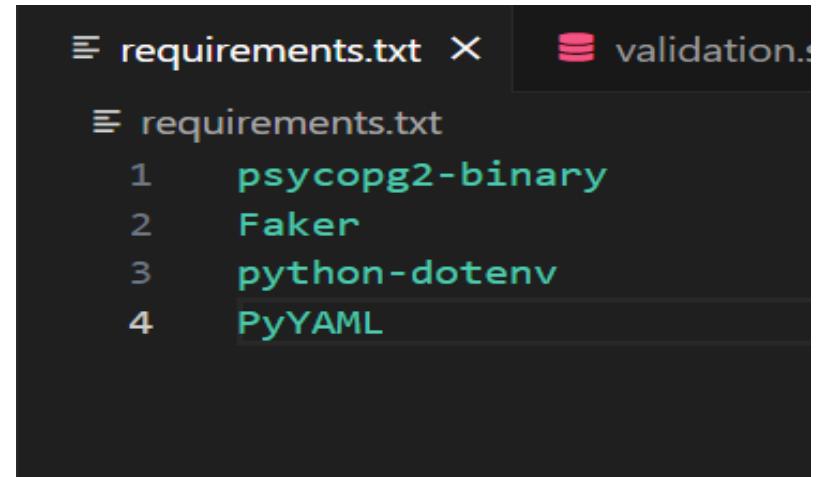
- Some FKs cascade deletes(e.g. if an order is deleted, its `order_items` delete automatically).
- Some FKs keep data but unlink it(set `NULL` if a customer is deleted).
- Some FKs restrict deletes(you cant delete a store if it has orders)

So yes my `Rushschema.sql` handles cascading automatically, since I used `ON DELETE CASCADE` on some tables, I don't really need an extra cascading script.

Phase 4: Synthetic data generation

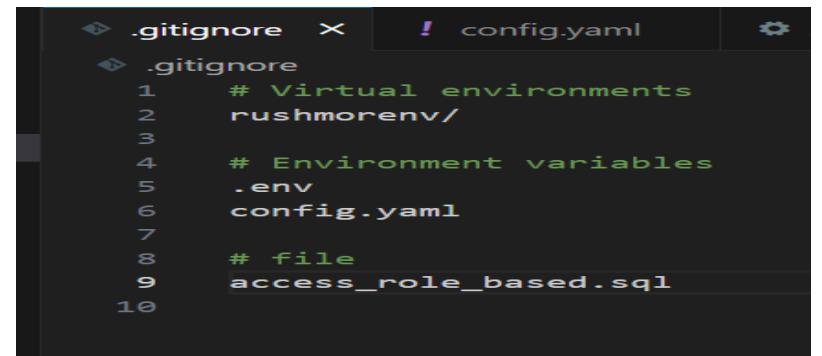
At this point, we write our populate.py that connects to the cloud DB, generates realistic data with Faker, and inserts in dependency-safe order to hit the target volumes. From our Capstone brief which mandates Faker + psycopg2, secrets via env/config, and specific record counts.

Config & Secrets: Here we did not hardcode credentials as I read from .env or config.yaml (host, port, user, password, dbname). Also requirements.txt file was created having my environment dependencies to be installed having psycopg2-binary, Faker, Python-dotenv, MyYaml and I was able to run on my terminal(pip install –r requirements.txt). I created a gitignore file to ignore or not to track my virtual environments with environment variables and file.



The screenshot shows a code editor interface with two tabs: 'requirements.txt' and 'validation.txt'. The 'requirements.txt' tab is active, displaying the following content:

```
psycopg2-binary
Faker
python-dotenv
PyYAML
```



The screenshot shows a code editor interface with two tabs: '.gitignore' and 'config.yaml'. The '.gitignore' tab is active, displaying the following content:

```
# Virtual environments
rushmorenv/
# Environment variables
.env
config.yaml
# file
access_role_based.sql
```

Synthetic data generation continued

Target volumes (minimum)

- Stores: 3-5
- Menu_items: 20-30
- Ingredients: 40-50
- Customers: 1000+
- Orders: 5000+
- Order_items: 15000+ (3 items/order)

These volumes are required for stress testing.

Insertion order (honors FK constraints)

1. Stores
2. Customers
3. Ingredients
4. Menu_Items

5. Item_Ingredients (recipes)
6. Orders
7. Order_items

Data realism rules

- Use Faker: fake.name(), fake.phone_number(), fake.email(), fake.address(), fake.date_time_this_year() for order timestamps.
- Order_items.price_at_time_of_order must capture the item price at order time(not the current catalog price).
- Orders.total_amount = sum of line totals for that order.

Synthetic data generation continued

Script structure

```
# rushmore/populate.py
# 1) load env/yaml
# 2) connect via psycopg2
# 3) helper: executemany batched inserts, commit per batch
# 4) seed STORES (3-5 cities), INGREDIENTS (40-50), MENU_ITEMS (20-30)
#     - For each menu item, assign 2-6 ingredients & quantities into ITEM_INGREDIENTS
# 5) seed CUSTOMERS (>= 1,000) with unique email/phone
# 6) seed ORDERS (>= 5,000) with timestamps this year, random store & customer
# 7) for each order, create ~3 ORDER_ITEMS:
#     - choose item_id at random
#     - quantity 1-4
#     - price_at_time_of_order = current Menu_Items.price (copied at insert)
# 8) compute & update ORDERS.total_amount
# 9) sanity checks (counts, null scans, FK violations, price sums)
# 10) print summary row counts
```

Proof of Population - Customers

The screenshot shows a PostgreSQL database interface with a sidebar navigation and a main query and results area.

Navigation:

- FTS Configurations
- FTS Dictionaries
- FTS Parsers
- FTS Templates
- Foreign Tables
- Functions
- Materialized Views
- Operators
- Procedures
- Sequences
- Tables (7)** (selected)
- customers
- ingredients
- item_ingredients
- menu_items
- order_items
- orders
- stores
- Trigger Functions
- Types
- Views
- Subscriptions
- gin/Group Roles
- blespaces

Query: SELECT * FROM public.customers ORDER BY customer_id ASC LIMIT 100

Data Output:

customer_id	first_name	last_name	email	phone_number	created_at
1	Stephanie	Nielsen	stephanie.nielsen1825@gmail.com	+1-357-901-5430x391	2025-08-06 21:26:34
2	Chad	Hodges	chad.hodges410@gmail.com	(570)624-8963	2024-09-16 23:25:20
3	Wanda	Santos	wanda.santos1507@yahoo.com	371.233.1509x8980	2024-05-07 21:27:04
4	Carlos	Johnson	carlos.johnson4013@yahoo.com	983.547.3829	2025-07-13 12:01:59
5	Thomas	Russell	thomas.russell03658@gmail.com	865.766.7010	2025-07-25 21:47:34
6	Michelle	Wright	michelle.wright228@gmail.com	+1-92-3624-731	2023-12-21 08:42:06
7	Denise	Fernandez	denise.fernandez1680@gmail.com	432-867-7360	2023-11-13 17:50:45
8	Tricia	Baker	tricia.baker8936@yahoo.com	268-272-3430x9805	2025-02-07 19:31:35
9	Thomas	Calhoun	thomas.calhoun1425@hotmail.com	(808)312-1913	2025-10-03 13:49:31
10	Ashley	Riley	ashley.riley9675@gmail.com	+1-816-499-8543x534	2025-04-16 18:52:48
11	Debbie	Bryant	debbie.bryant6913@gmail.com	6794911838	2025-11-05 22:52:57

Total rows: 100 Query complete 00:00:00.195 CRLF Ln 1, Col 1

Synthetic data generation continued

Proof of population- Ingredients

The screenshot shows a database interface with a sidebar containing various database objects like FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is expanded, and 'ingredients' is selected. The main area displays a query window with the following code:

```
1 v SELECT * FROM public.ingredients
2 ORDER BY ingredient_id ASC LIMIT 100
3
```

Below the code is a table titled 'Data Output' showing the results of the query. The table has columns: ingredient_id [PK] integer, name character varying (100), stock_quantity numeric (10,2), and unit character varying (20). The data shows 12 rows of ingredients with their respective details.

ingredient_id [PK] integer	name character varying (100)	stock_quantity numeric (10,2)	unit character varying (20)
1	Mozzarella	44.87	ml
2	Cheddar	36.97	g
3	Parmesan	55.63	liters
4	Goat Cheese	42.79	units
5	Ricotta	73.46	kg
6	Pepperoni	33.33	ml
7	Salami	41.50	ml
8	Ham	101.83	kg
9	Bacon	35.07	units
10	Chicken	25.54	kg
11	Beef	67.11	g
12	Mushrooms	55.63	units

Total rows: 45 Query complete 00:00:00.299

Proof of population- Items_ingredient

The screenshot shows a database interface with a sidebar containing various database objects like FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is expanded, and 'item_ingredients' is selected. The main area displays a query window with the following code:

```
1 v SELECT * FROM public.item_ingredients
2 ORDER BY item_id ASC, ingredient_id ASC LIMIT 100
3
```

Below the code is a table titled 'Data Output' showing the results of the query. The table has columns: item_id [PK] integer, ingredient_id [PK] integer, and quantity_required numeric (10,2). The data shows 12 rows of item-ingredient associations with their required quantities.

item_id [PK] integer	ingredient_id [PK] integer	quantity_required numeric (10,2)
1	1	10
2	1	16
3	1	23
4	1	42
5	2	12
6	2	41
7	3	29
8	3	37
9	3	40
10	3	41
11	3	42
12	3	44

Total rows: 100 Query complete 00:00:00.201

Synthetic data generation continued

Proof of population- Menu_items

The screenshot shows a database interface with a sidebar containing navigation links such as FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is expanded, and 'menu_items' is selected. The main area displays a query history with the following SQL:

```
1 ✓ SELECT * FROM public.menu_items
2 ORDER BY item_id ASC LIMIT 100
3
```

The results are shown in a table titled 'Data Output' with columns: item_id, name, category, size, and price. The table shows 12 rows of synthetic data.

item_id	name	category	size	price
1	Brownie	Dessert	[null]	5.57
2	Four Cheese Special	Pizza	Medium	12.88
3	Coleslaw	Side	[null]	4.36
4	Ice Cream	Dessert	[null]	3.87
5	BBQ Chicken Deluxe	Pizza	Large	16.64
6	Orange Soda	Drink	500ml	1.17
7	Margarita Deluxe	Pizza	Small	17.91
8	Orange Soda	Drink	330ml	2.47
9	Cola	Drink	500ml	1.88
10	Cheesecake	Dessert	[null]	2.23
11	Lemonade	Drink	500ml	1.26
12	Wedges	Side	[null]	6.11

Proof of population- Order_items

The screenshot shows a database interface with a sidebar containing navigation links such as FTS Configurations, FTS Dictionaries, FTS Parsers, FTS Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is expanded, and 'order_items' is selected. The main area displays a query history with the following SQL:

```
1 ✓ SELECT * FROM public.order_items
2 ORDER BY order_item_id ASC LIMIT 100
3
```

The results are shown in a table titled 'Data Output' with columns: order_item_id, order_id, item_id, quantity, and price_at_time_of_order. The table shows 12 rows of synthetic data.

order_item_id	order_id	item_id	quantity	price_at_time_of_order
1	1	1	4	3.87
2	2	1	6	1.17
3	3	2	13	4.43
4	4	2	10	2.23
5	5	2	1	5.57
6	6	3	9	1.88
7	7	3	12	6.11
8	8	3	25	20.49
9	9	4	19	6.12
10	10	4	22	2.77
11	11	4	11	1.26
12	12	4	10	2.23

Total rows: 100 Query complete 00:00:00.202

Synthetic data generation continued

Proof of population- Orders

The screenshot shows a database interface with a sidebar containing navigation links for FTS Configurations, Dictionaries, Parsers, Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is currently selected. Below the sidebar is a 'Query History' section with the following SQL code:

```
1 < SELECT * FROM public.orders
2 ORDER BY order_id ASC LIMIT 100
3
```

Below the query history is a 'Data Output' section displaying the results of the query. The results table has columns: order_id [PK] integer, customer_id integer, store_id integer, order_timestamp timestamp without time zone, total_amount numeric (10,2), and status character varying (50). The table shows 100 rows of synthetic data. The first few rows are:

order_id	customer_id	store_id	order_timestamp	total_amount	status
1	1	963	2025-06-30 00:15:14	8.55	Cancelled
2	2	765	2025-08-24 18:01:40	31.09	Pending
3	3	732	2025-06-30 01:54:33	52.73	Delivered
4	4	494	2025-09-07 00:02:36	36.11	Delivered
5	5	274	2025-04-25 00:28:19	46.39	Delivered
6	6	1119	2025-03-12 14:18:02	129.61	In Progress
7	7	952	2025-01-15 14:16:48	135.05	In Progress
8	8	277	2025-04-14 01:06:12	135.30	Cancelled
9	9	76	2024-11-29 13:54:39	84.32	In Progress
10	10	90	2025-02-09 05:45:07	98.52	Pending
11	11	647	2025-10-05 00:02:51	5.57	Cancelled
12	12	1113	2025-02-12 20:39:36	31.87	Delivered

Total rows: 100 Query complete 00:00:00.204

Proof of population- Stores

The screenshot shows a database interface with a sidebar containing navigation links for FTS Configurations, Dictionaries, Parsers, Templates, Foreign Tables, Functions, Materialized Views, Operators, Procedures, Sequences, and Tables (7). The 'Tables' section is currently selected. Below the sidebar is a 'Query History' section with the following SQL code:

```
1 < SELECT * FROM public.stores
2 ORDER BY store_id ASC LIMIT 100
3
```

Below the query history is a 'Data Output' section displaying the results of the query. The results table has columns: store_id [PK] integer, address character varying (255), city character varying (100), phone_number character varying (30), and opened_at timestamp without time zone. The table shows 4 rows of synthetic data. The first few rows are:

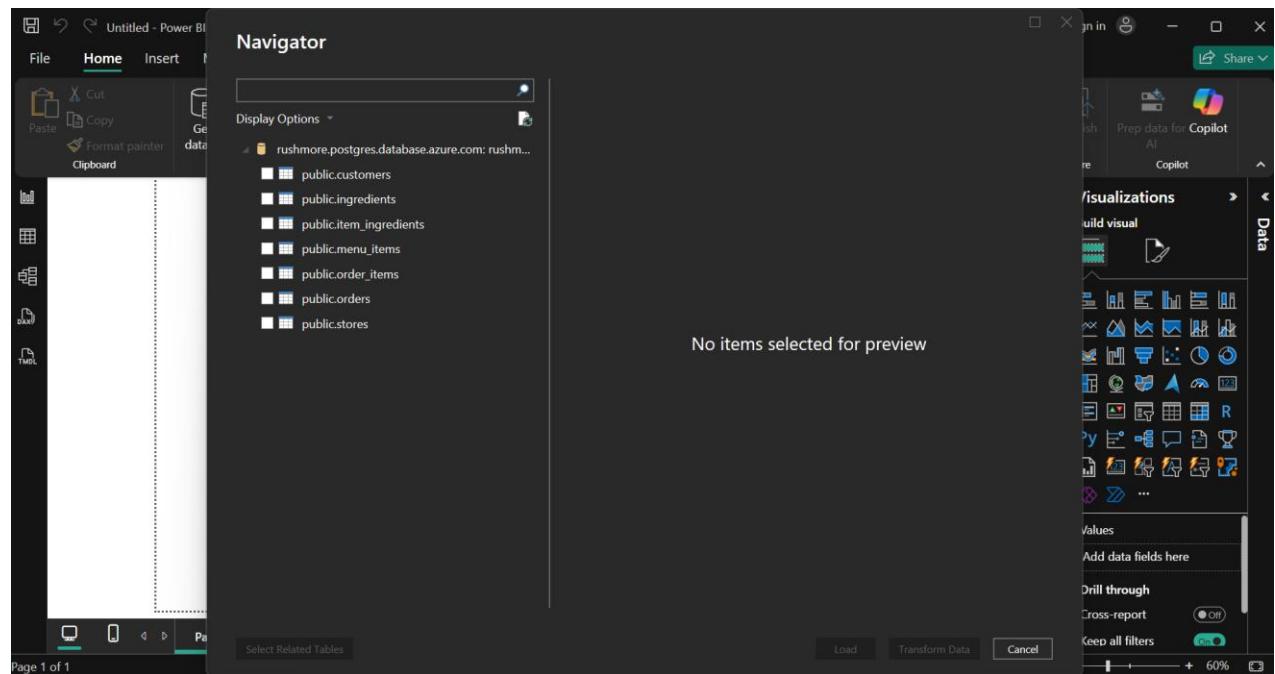
store_id	address	city	phone_number	opened_at
1	495 Janet Cape	New Roberttown	+1-634-313-1647x52553	2025-06-15 01:57:24
2	327 Nelson Route	North Judithbury	001-783-550-3056x413	2025-08-26 00:25:53
3	7242 Julie Plain Suite 969	East Jesettown	(353)828-7101x2269	2024-06-26 15:42:18
4	480 Erin Plain Suite 514	Ericmouth	(727)404-8281x48932	2025-02-26 17:24:52

Total rows: 4 Query complete 00:00:00.204

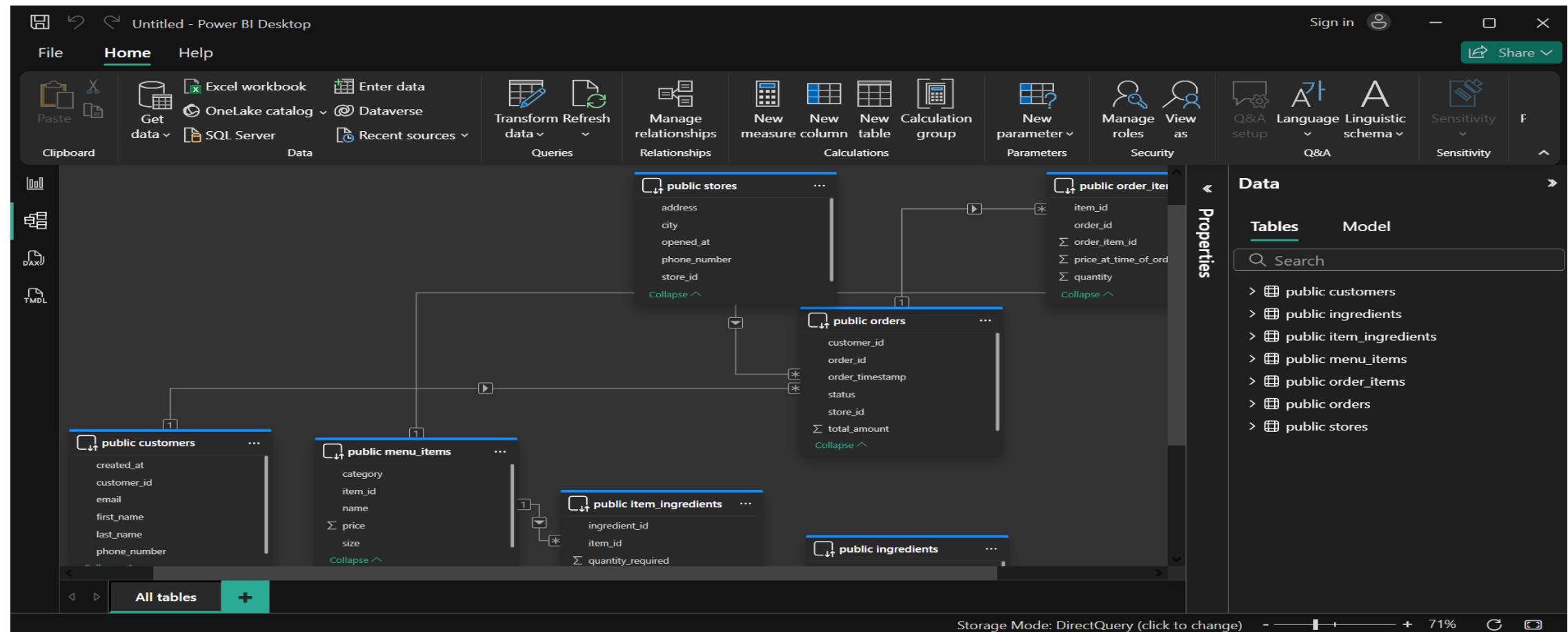
Phase 5: Validation SQL & Analytics(Connect to Power BI & Answer business questions)

After I successfully populate my data generated from Faker, I went ahead to validate my data using the created validation.sql script.

I was able to connect to Power BI as shown below and also created an analquery.sql script which I used to answer the below business questions.



Power BI view of Entity Relationship Diagram



Business Question

Total sales revenue per store

The screenshot shows the pgAdmin 4 interface with two panes. The left pane is the Object Explorer displaying database schemas and tables. The right pane is the Query Editor with the following SQL code:

```
1 < SELECT
2   s.store_id,
3   s.city,
4   SUM(o.total_amount) AS revenue,
5   COUNT(*) AS orders_count
6   FROM orders o
7   JOIN stores s ON s.store_id = o.store_id
8   GROUP BY s.store_id, s.city
9   ORDER BY revenue DESC;
```

The Data Output tab shows the results of the query:

store_id	city	revenue	orders_count
1	New Roberttown	82824.54	1551
2	Ericmouth	81108.10	1507
3	North Judithbury	80737.91	1487
4	East Jessetown	75691.67	1455

Total rows: 4 Query complete 00:00:00.364

Top 10 customers by spend

The screenshot shows the pgAdmin 4 interface with two panes. The left pane is the Object Explorer displaying database schemas and tables. The right pane is the Query Editor with the following SQL code:

```
1 < SELECT
2   c.customer_id,
3   c.first_name,
4   c.last_name,
5   c.email,
6   SUM(o.total_amount) AS total_spent,
7   COUNT(*) AS orders_count
8   FROM orders o
9   JOIN customers c ON c.customer_id = o.customer_id
10  GROUP BY c.customer_id, c.first_name, c.last_name, c.email
11  ORDER BY total_spent DESC
12  LIMIT 10;
```

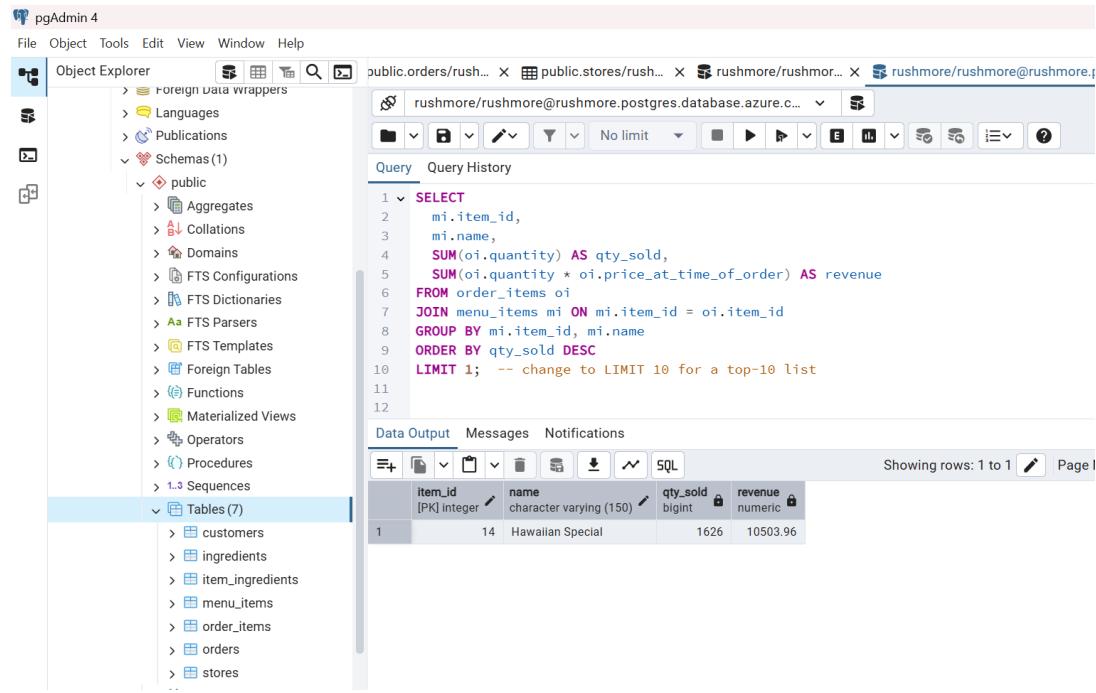
The Data Output tab shows the results of the query:

customer_id	first_name	last_name	email	total_spent	orders_count
976	Sandra	Clarke	sandra.clarke223@yahoo.com	868.75	13
649	Patricia	Davis	patricia.davis604@yahoo.com	772.56	11
500	Megan	Lewis	megan.lewis3849@hotmail.com	745.41	7
191	Rebecca	Calderon	rebecca.calderon1050@yahoo.com	715.10	13
1162	Theresa	Galloway	theresa.galloway9578@hotmail.com	710.73	11
584	Michael	Armstrong	michael.armstrong5764@yahoo.co...	701.77	10
185	Sarah	Johnson	sarah.johnson7178@hotmail.com	701.69	12

Total rows: 10 Query complete 00:00:01.066

Business Question continued

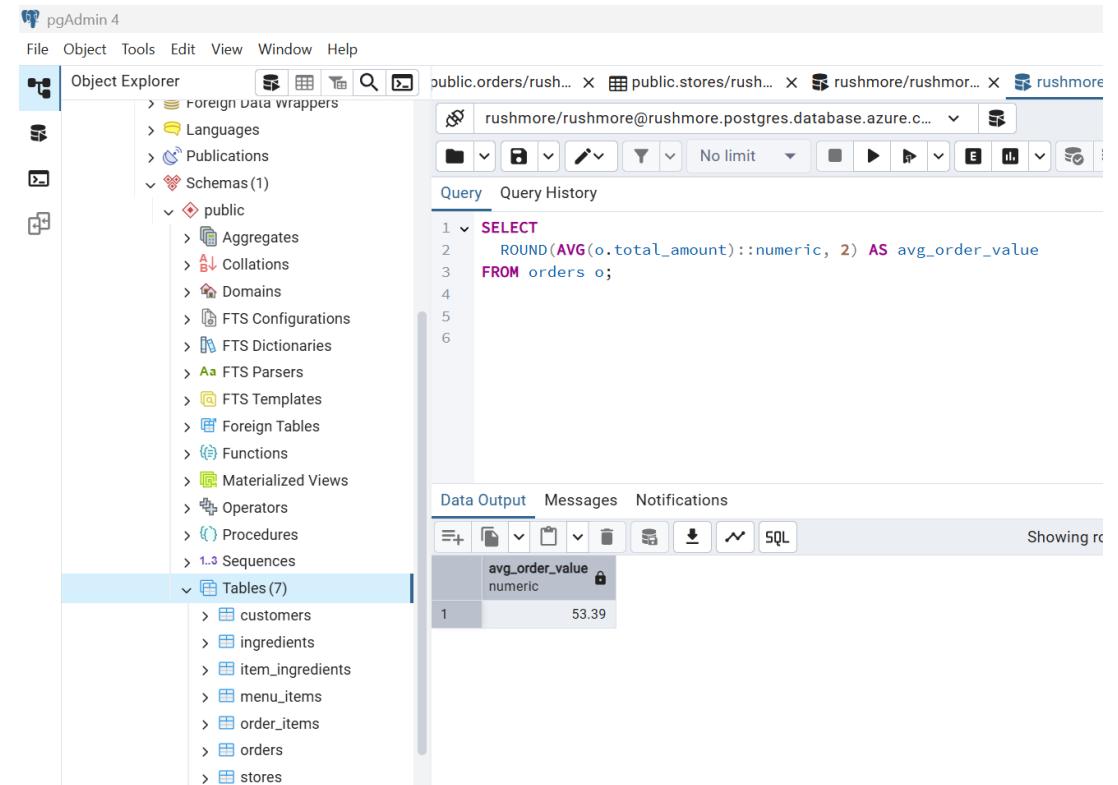
Most popular menu item(by quantity sold)



```
1 ✓ SELECT
2     mi.item_id,
3     mi.name,
4     SUM(oi.quantity) AS qty_sold,
5     SUM(oi.quantity * oi.price_at_time_of_order) AS revenue
6     FROM order_items oi
7     JOIN menu_items mi ON mi.item_id = oi.item_id
8     GROUP BY mi.item_id, mi.name
9     ORDER BY qty_sold DESC
10    LIMIT 1; -- change to LIMIT 10 for a top-10 list
11
12
```

item_id	name	qty_sold	revenue
14	Hawaiian Special	1626	10503.96

Average order value



```
1 ✓ SELECT
2     ROUND(AVG(o.total_amount)::numeric, 2) AS avg_order_value
3     FROM orders o;
4
5
6
```

avg_order_value
53.39

Business question continued

Busiest hours of day

The screenshot shows the pgAdmin 4 interface. The left sidebar is the Object Explorer, displaying various database objects like Schemas, Tables, and Functions. The main area is a query editor window with the following SQL query:

```
1 SELECT
2     EXTRACT(HOUR FROM o.order_timestamp) AS hour_of_day,
3     COUNT(*) AS orders_count
4 FROM orders o
5 GROUP BY hour_of_day
6 ORDER BY orders_count DESC, hour_of_day;
```

The Data Output tab shows the results of the query:

hour_of_day	orders_count
18	276
2	273
9	265
13	265
4	262
5	262
15	261
7	259
21	259
19	258
23	258
6	257

Total rows: 24 Query complete 00:00:00.432

Challenges

Challenges

1. The first challenge resolved after watching video on how to connect Azure Database for PostgreSQL server with PgAdmin.
2. During my Population of data using Faker, I got the `StringDataRightTruncation` error, upon debugging it I noticed faker return phone numbers longer than 20 character(e.g., with spaces, brackets, country codes) but my stores and customers phone number column is `VARCHAR(20)` and so the insert failed. I went ahead to drop my tables and widen my phone number columns to `VARCHAR(30)` as I prefer full formatting.
3. I noticed my `menu_item` size column after data generated has **null** values, I also had to debug where it is coming from as I noticed size was set to N/A(for Side and Desserts) so it becomes **NULL** in postgreSQL. I had to fix that because it is the kind of details that makes projects like this look polished. Since that menu item for side and dessert cannot be either large, medium, small or family as pizza size range, I choose to set it to **Regular**
4. My Power BI connected and I was able to see my table but was unable to fully run my business questions on it as I got error message duplicate values as it is not allowed for columns on the one side of a many to one relationship for columns that are used as the primary key for a table.

Result

- Connected to an Azure PostgreSQL Server using pgAdmin and psql, as I established secure access to my cloud database
- Designed and implemented a fully normalized OLTP schema, my **rushschema.sql** builds a clean, 3rd Normal form(3NF) relational model for a pizza business, drop tables in correct order using **CASCADE** to avoid dependency errors. Optimized performance using **indexes.sql**, protect data quality by enforcing business rules and prevent invalid date from entering the system using **constraints.sql**. I applied role based access so different users have different permissions as seen in my **role_template.sql**.
- Developed python scripts(.py) to generate and load realistic fake data which uses Faker + psycopg2 with thousands of rows(store, customers, menu items, ingredients, recipes, orders and order items).
- Validated all loaded data which confirms no orphaned records, no broken foreign keys, no invalid prices or quantities, order totals match summed line items and Unique constraints are respected.
- Answered key business questions using SQL which I ran analytics queries(revenue by store, top customers, most popular items, busiest hours, etc.).
- Connect Power BI to my Azure PostgreSQL.

This demonstrate a complete modern data engineering workflow:

Design → Build → Populate → Validate → Analyze → Visualize.

Extra – Execution Order

After creating the connection between Azure Database for PostgreSQL server with PgAdmin PostgreSQL and VS Code.

1. Run my schema(**rushschema.sql**)
2. Run **indexes.sql**
3. Run **constraintns.sql**
4. Run RBAC(**access_role_based.sql**), github(**role_template.sql**)
5. Load data with **populate.py**
6. Run **validation.sql** and review outputs.
7. Run **analquery.sql** (see business questions)

PowerPoint 2013

Capstone Project done and presented by
Tochukwu Kingsley Alaneme
(Data Engineering class May 2025)

Find out more at the PowerPoint Getting Started Center →
(Click the arrow when in Slide Show mode)