E-Commerce Site
(An Online Shopping Site)

Name: Naga Venkata Kanakalakshmi
UNT ID: 11725119
DPDB Project 1

Requirement-1:

Introduction:
In the current world, online shopping has become an integral part of everyone's life. The E site plays the vital role in delivering certified products in the shortest possible time. E-commerce sites provide customers with seamless shopping experience and give them a comfortable lifestyle. It helps the customers to browse the wide range of products by applying different filters, which makes browsing easy. The payment gateway helps the customers to make the safe and Security transactions.

Requirements:

- The goal of the project is to create an e-commerce platform with a strong database schema. It supports a variety of functionalities, including customer signup, order placement, address management, managing products, shopping cart, handling transactions, supplier information, the option to review and the rate, exciting discounts and gift vouchers, an excellent delivery system, and high security using error handling mechanism.
- Overall to implement the e-commerce platform used 17 tables. Which is customer, products, product_category, address, cart, orders, transaction_summary, supplier, reviews, gift_vouchers, delivery_partner, customer_address, customer_gift_voucher, customer_delivery_partner, products_cart, order_products, products_suppliers. These tables include relation tables too. Which are the intermediary tables, links between the two entities.

Assumptions:

- A customer can add multiple products to his/her cart as per choice. Whereas in the same way the same product can be added by multiple customers in the cart. It associates the many-to-many to relation between the products and the cart. As the relation is many-to-many an intermediary table formed to link the products and cart table that is products_cart.
- As mentioned in point (1), in the same way a product can be ordered by many customers and during the product's delivery there can be the same product among the multiple orders. By this the products and orders table form the many-to-many relation. As it's relationship is many-to-many, an intermediary relation table is formed to link the orders and products table that is products_orders.

- There will be multiple suppliers who supply the same product and there are few suppliers where they can supply multiple products. For instance, there will multiple suppliers who supply washing machines like Samsung, lg, ifb etc. In the same way a supplier can supply multiple products like fridge, washing machine etc. By this the relation associated with the supplier and the product is many-to-many and there is intermediary table to form the link between them that is the products_suppliers
- Multiple products can be placed in the location, and they have a specific address to it. By this it forms a many-to-one relation between the products and the address. When the group of products are stored in a particular location (warehouse) it helps the delivery partner to pick and deliver the products to the customer.
- Group of products have a specific category, in most of the online shopping sites to make searching easier the products are categorized. For instance, dresses, tops, sarees come under clothing section, electronic gadgets like mobiles, power banks have the specific category. So, the relationship between products and the product_category is many-to-one.
- Each product has multiple reviews, which helps customers to purchase the product more easily. The relationship between the products and reviews is many-to-one which says that a product can have zero reviews, one or more reviews.
- Each customer has at most one cart to add the items to purchase. It associates the one-to-one relation between the cart and the customer.
- A customer can store multiple addresses, where he can order the products to multiple locations as per his/her choice. In the idle case, multiple customers can store the same address as when multiple as working in same company they want to order the products to their office location then the group of customers have the same address. The relationship associated between the customer and the address is many-to-many. There is an intermediary table which links customers and address table, which can also store the default address of the customer.
- Each customer can have multiple gift vouchers to redeem, and a single gift voucher can be given to multiple customers with the same coupon code. Many-to-Many relation forms between the customers and the gift vouchers.
- The site provides the leverage to place any number of orders by the customer. This forms the many-to-one relation between the orders and the customer.
- Delivery Partner can deliver any number of products to many customers, it makes the work faster and delivery of the products on time to the customers. A customer may contact multiple delivery partners for multiple orders. Thereby it forms the many-to-many relation between the customer and delivery partner. As it's a many-to-many relationship if forms the intermediary table cutomer_delivery_partner.
- Each order stores the respective transaction details, this ensures that every order placed within the system is associated with a single recorder which summarizes the transaction details of the specific customer. It helps to easily track the customer transactions. This ensures the one-to-one relation between the orders and the transaction summary.

- Many orders can be placed to a single address, and single order cannot be diverse among the multiple addresses. which ensures a many-to-one relation between the orders and the address.
- A single order is delivered by the specific delivery partner. This ensures the relationship between the orders and the delivery partner is one-to-one.
- On a single order, the customers can provide multiple reviews as per their choice. This forms the many-to-one relationship between the reviews and the order.
- Each supplier has exactly one address to it and each location is dedicated to a supplier. This ensures to form the one-to-one relationship between the suppliers and the address.
- A customer can directly place the order, without adding to cart. Sometimes a customer will add multiple products to cart but he/she won't buy it from cart, he/she can directly place order without using the cart.

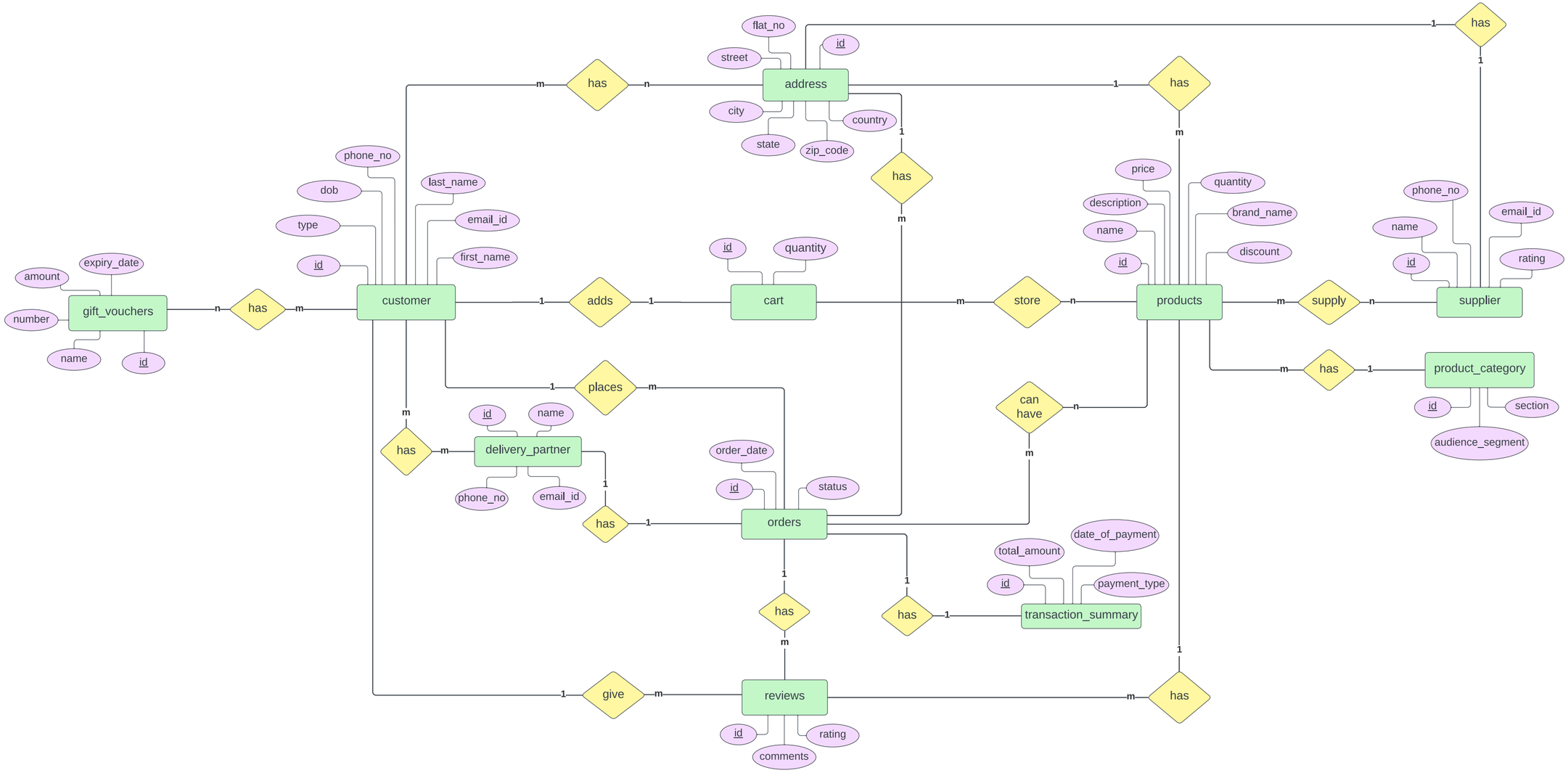
	Analysis				
Table	**Attributes**	**Domain Constraints**	**Check Contraints**	**Relation**	**Grant_Access (Customer/Administrator/SuperUser)**
products	id	Raw(16)	PRIMARY KEY	products has many-to-many relation with cart product has many-to-many relation with orders products has many-to-many relation with suppliers product has many-to-one relation with address product has many-to-one with product_category product has one-to-many with review	Customer - SELECT Adminstrator - SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, EXECUTE Super User - ALL PRIVILEGES
	name	VARCHAR2(50)	dont allow nulls		
	description	VARCHAR2(200)	dont allow nulls		
	price	INTEGER	dont allow nulls		
	quantity	INTEGER	dont allow nulls		
	discount	INTEGER	NA		
	brandname	VARCHAR2(20)	dont allow nulls		
	address_id	Raw(16)	no null values, FOREIGN KEY (address_id) REFERENCES address(id)		
	product_category_id	Raw(16)	dont allow nulls, FOREIGN KEY (product_category_id) REFERENCES product_category(id)		
customer	id	Raw(16)	PRIMARY KEY	customer has one-to-one relation with cart customer has many-to-many relation with address customer has many-to-many relation with giftvoucher customer has one-to-many relation with orders customer has one-to-many relation with reviews customer has many-to-many relation with delivery_partner	Customer - SELECT, INSERT,UPDATE Adminstrator - SELECT, INSERT, UPDATE, DELETE, CREATE, ALTER, DROP, EXECUTE Super User - ALL PRIVILEGES
	first_name	VARCHAR2(20)	dont allow nulls		
	last_name	VARCHAR2(20)	dont allow nulls		
	phone_no	VARCHAR2(20)	dont allow nulls, UNIQUE, CHECK (REGEXP_LIKE (phone, '^\ (\d{3}\) \d{3}-\d {4}$'))		
	email_id	VARCHAR2(255)	no null values, UNIQUE, CHECK (REGEXP_LIKE (email, '^[A-Za-z0-9. _%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'))		
	dob	DATE	dont allow nulls		
	type - regular/premium/gold/vip	VARCHAR2(20)	dont allow nulls		
cart	id	Raw(16)	PRIMARY KEY	cart has one-to-one relation with customer cart has many-to-many relation with products	Customer - SELECT, INSERT, UPDATE, DELETE Administrator - SELECT, UPDATE, DELETE Super User - ALL PRIVILEGES
	customer_id	Raw(16)	dont allow nulls, FOREIGN KEY (customer_id) REFERENCES customer(id)		
	quantity	INTEGER	dont allow nulls		
orders	id	Raw(16)	PRIMARY KEY	orders has many-to-one relation with customer orders has many-to-many relation with products order has one-to-one relation with transaction_summary orders has many-to-one with address order has one-to-many with review order has one-to-one delivery_partner	Customer - SELECT, INSERT, UPDATE, DELETE Administrator - SELECT, INSERT, UPDATE, DELETE, ALTER, DROP Super User - ALL PRIVILEGES
	status	VARCHAR2(20)	dont allow nulls		
	order_date	DATE	dont allow nulls		
	address_id	Raw(16)	dont allow nulls, FOREIGN KEY (address_id) REFERENCES address(id)		
	customer_id	Raw(16)	dont allow nulls, FOREIGN KEY (customer_id) REFERENCES customer(id)		
transaction_summary	id	Raw(16)	PRIMARY KEY	transaction_summary has one-to-one relation with orders	Customer - SELECT Administrator - SELECT Super User - ALL PRIVILEGES
	total_amount_paid	INTEGER	dont allow nulls		
	payment_type - upi/wallets/creditcard/debitcard/net banking/emi/cod	VARCHAR2(20)	dont allow nulls		
	date_of_payment	DATE	dont allow nulls		

	Analysis				
Table	**Attributes**	**Domain Constraints**	**Check Contraints**	**Relation**	**Grant_Access (Customer/Administrator/SuperUser)**
	order_id	Raw(16)	dont allow nulls FOREIGN KEY (order_id) REFERENCES orders(id)		
supplier	id	Raw(16)	PRIMARY KEY	suppliers has many-to-many relation with products supplier has one-to-one relation with address	Customer - NA Administrator - SELECT, INSERT,UPDATE, DELETE Super User - ALL PRIVILEGES
	name	VARCHAR2(50)	dont allow nulls		
	phone_no	VARCHAR2(20)	dont allow nulls, UNIQUE, CHECK (REGEXP_LIKE (phone, '^\ (\d{3}\) \d{3}-\d {4}$'))		
	email	VARCHAR2(255)	dont allow nulls, UNIQUE, CHECK (REGEXP_LIKE (email, '^[A-Za-z0-9. _%+-]+@[A-Za-z0-9.-] +\.[A-Za-z]{2,}$'))		
	rating	INTEGER	NA		
	address_id	Raw(16)	dont allow nulls, FOREIGN KEY (address_id) REFERENCES address(id)		
address	id	Raw(16)	PRIMARY KEY	address has many-to-many relation with customer address has one-to-many relation with product address has one-to-many with orders address has one-to-one relation with supplier	Customer - SELECT, INSERT, UPDATE, DELETE Administrator - SELECT, INSERT, UPDATE, DELETE Super User - ALL PRIVILEGES
	flat_no	NUMBER	dont allow nulls		
	street	VARCHAR2(50)	dont allow nulls		
	city	VARCHAR2(50)	dont allow nulls		
	state	VARCHAR2(50)	dont allow nulls		
	country	VARCHAR2(50)	dont allow nulls		
	zip_code	VARCHAR(10)	dont allow nulls		
product_category	id	Raw(16)	PRIMARY KEY	product_category has one-to-many with product	Customer - SELECT Administrator - SELECT, INSERT, UPDATE, DELETE Super User - ALL PRIVILEGES
	section - clothing/ footwear/ accessories /electronics	VARCHAR2(20)	dont allow nulls		
	audience_segment - men/women/kids/girls/boys	VARCHAR2(20)	dont allow nulls		
reviews	id	Raw(16)	PRIMARY KEY	reviews has many-to-one relation with customer review has many-to-one with product review has many-to-one with order	Customer - SELECT, INSERT, UPDATE, DELETE Administrator - SELECT, DELETE Super User - ALL PRIVILEGES
	rating	NUMBER	NA		
	comments	VARCHAR2(150)	NA		
	product_id	Raw(16)	FOREIGN KEY (product_id) REFERENCES products(id)		
	order_id	Raw(16)	FOREIGN KEY (order_id) REFERENCES orders(id)		
	customer_id	Raw(16)	FOREIGN KEY (customer_id) REFERENCES customer(id)		
gift_vouchers	id	Raw(16)	PRIMARY KEY	gift_voucher has many-to-many relation with customer	Customer - SELECT Administrator - SELECT, INSERT, UPDATE, DELETE Super User - ALL PRIVILEGES
	name	VARCHAR2(50)	NA		
	number	NUMBER	NA		
	amount	INTEGER	NA		
	expiry_date	DATE	NA		
	id	Raw(16)	PRIMARY KEY		
	name	VARCHAR2(50)	dont allow nulls		

	Analysis				
Table	**Attributes**	**Domain Constraints**	**Check Contraints**	**Relation**	**Grant_Access (Customer/Administrator/SuperUser)**
delivery_partner	phone_no	VARCHAR2(20)	dont allow nulls, CHECK (REGEXP_LIKE (phone, '^\ (\d{3}\) \d{3}-\d {4}$'))	delivery_partner has many-to-many relation with customer delivery_partner has one-to-one orders	Customer - NA Administrator - SELECT, INSERT, UPDATE, DELETE Super User - ALL PRIVILEGES
	email	VARCHAR2(255)	dont allow nulls, CHECK (REGEXP_LIKE (email, '^[A-Za-z0-9. _%+-]+@[A-Za-z0-9.-] +\.[A-Za-z]{2,}$'))		
	order_id	Raw(16)	dont allow nulls FOREIGN KEY (order_id) REFERENCES orders(id)		
customer_address (relation table)	customer_id	Raw(16)	PRIMARY KEY dont allow nulls, FOREIGN KEY (customer_id) REFERENCES customer(id)	NA	NA
	address_id	Raw(16)	PRIMARY KEY dont allow nulls, FOREIGN KEY (address_id) REFERENCES address(id)		
	default_address - yes/no	BOOLEAN	dont allow nulls		
customer_gift_voucher (relation table)	customer_id	Raw(16)	PRIMARY KEY dont allow nulls, FOREIGN KEY (customer_id) REFERENCES customer(id)	NA	
	gift_voucher_id	Raw(16)	PRIMARY KEY		
	count	NUMBER	dont allow nulls		
customer_delivery_partner (relation table)	customer_id	Raw(16)	PRIMARY KEY dont allow nulls, FOREIGN KEY (customer_id) REFERENCES customer(id)	NA	NA
	delivery_partner_id	Raw(16)	PRIMARY KEY dont allow nulls, FOREIGN KEY (delivery_partner_id) REFERENCES delivery_partner(id)		
products_cart (relation table)	product_id	Raw(16)	PRIMARY KEY FOREIGN KEY (product_id) REFERENCES products(id)	NA	NA
	cart_id	Raw(16)	PRIMARY KEY		
	count	NUMBER	dont allow nulls		
orders_products (relation table)	orders_id	Raw(16)	PRIMARY KEY dont allow nulls FOREIGN KEY (order_id) REFERENCES orders(id)	NA	NA
	products_id	Raw(16)	PRIMARY KEY FOREIGN KEY (product_id) REFERENCES products(id)		
	quantity	INTEGER	dont allow nulls		
products_suppliers	product_id	Raw(16)	PRIMARY KEY FOREIGN KEY (product_id) REFERENCES products(id)		

	Analysis				
Table	**Attributes**	**Domain Constraints**	**Check Contraints**	**Relation**	**Grant_Access (Customer/Administrator/SuperUser)**
products_suppliers (relation table)	supplier_id	Raw(16)	PRIMARY KEY FOREIGN KEY (supplier_id) REFERENCES supplier(id)	NA	NA
	quantity	INTEGER	dont allow nulls		

E-R Diagram


flat_no
id
street
address
city
country
state
zip_code
has
has
has
has
phone_no
dob
type
id
last_name
email_id
first_name
customer
adds
id
quantity
cart
store
price
quantity
description
brand_name
name
id
discount
products
supply
phone_no
email_id
name
id
rating
supplier
expiry_date
amount
number
gift_vouchers
has
name
id
has
product_category
id
section
audience_segment
places
id
name
has
delivery_partner
phone_no
email_id
can have
order_date
id
status
has
orders
total_amount
date_of_payment
id
payment_type
has
transaction_summary
has
give
reviews
id
rating
comments
has

ERD Transformations:

1. products(**id**, name, description, price, quantity, brand_name, discount, address_id , product_category_id)
2. customer(**id**, first_name, last_name, email_id, phone_no, dob, type)
3. cart(**id**, quantity, customer_id)
4. orders(**id**, order_date, status, address_id , customer_id)
5. transaction_summary(**id**, total_amount, date_of_payment, payment_type, order_id)
6. supplier(**id**, name, phone_no, email_id, rating, address_id)
7. address(**id**, flat_no, street, city, state, country, zip_code)
8. product_category(**id**, section, audience_segment)
9. reviews(**id**, rating, comments, product_id , order_id , customer_id)
10. gift_vouchers(**id**, name, number, amount, expiry_date)
11. delivery_partner(**id**, name, phone, email_id, order_id)
12. customer_address(**customer_id** , **address_id** , default_address)
13. customer_gift_voucher (**customer_id** , **gift_voucher_id** , count)
14. customer_delivery_partner(**customer_id** , **delivery_partner_id**)
15. product_cart(**product_id** , **cart_id** , count)
16. orders_products(**order_id** , **product_id** , quantity)
17. products_suppliers(**product_id** , **supplier_id** , quantity)

Requirement-2:

I've set up Docker to work on DPDB Project 1. Here is a screenshot of the Docker Dashboard.

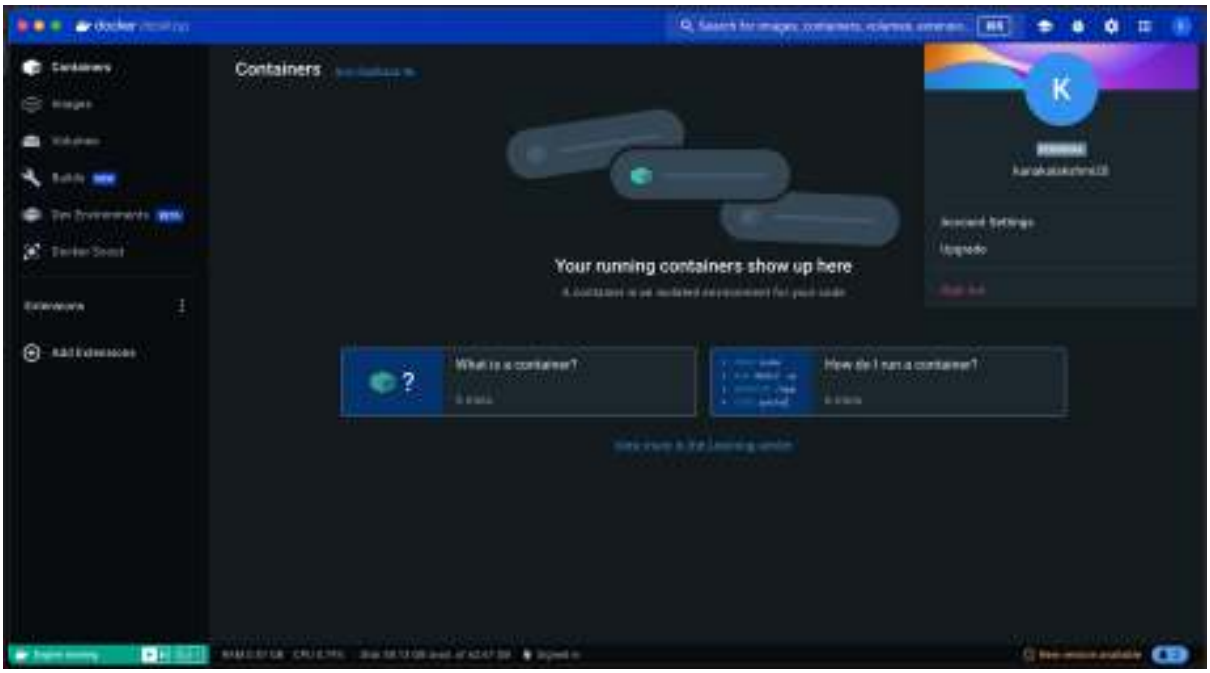


Fig-1: Docker Dashboard

Once Docker is installed, download the postgres:alpine image from the terminal. The command used to pull the postgres image is "**docker pull postgres:alpine**" As alpine Linux is a lightweight, small size, and secure. used Postgres with alpine.

Fig-2: Pull Postgres image.

The postgres instance is created and the command used is
“**docker run -–name kl-dpdb-node1 -e POSTGRES_PASSWORD=password -d -p 5432:5432 postgres:alpine**”

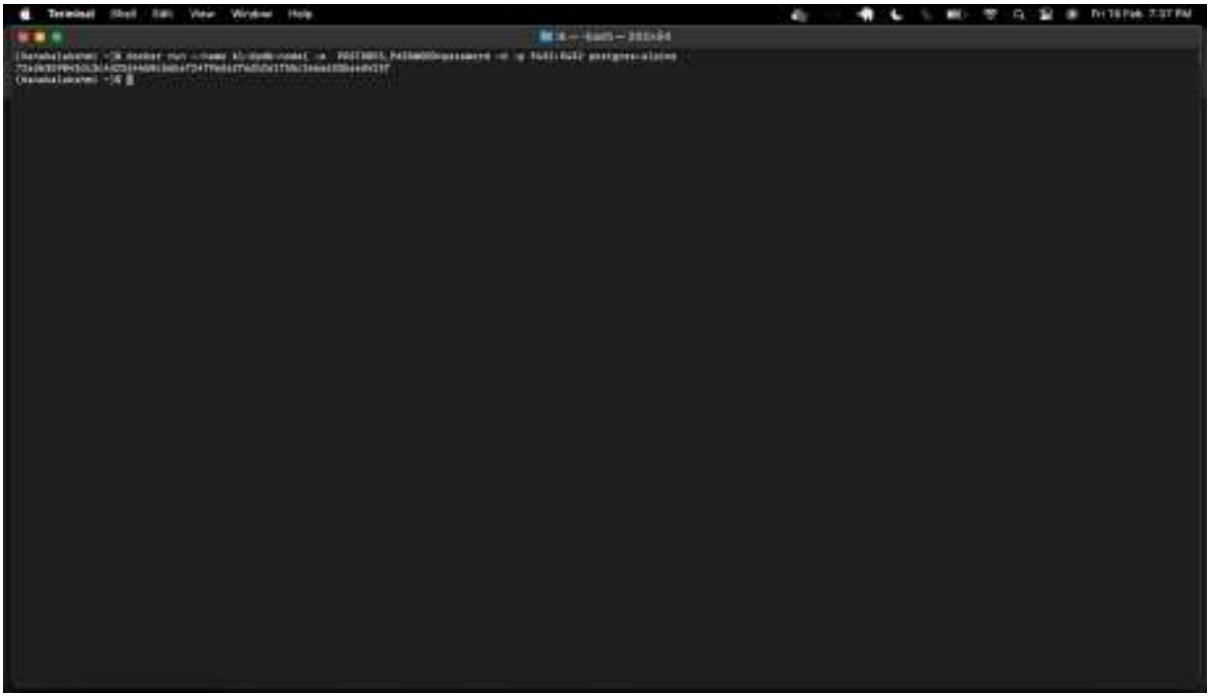


Fig-3: Create postgres instance.

To verify the list of instances running in Docker, the command used is **“docker ps”.** We can see the container details, such as the container ID, status, ports, etc.

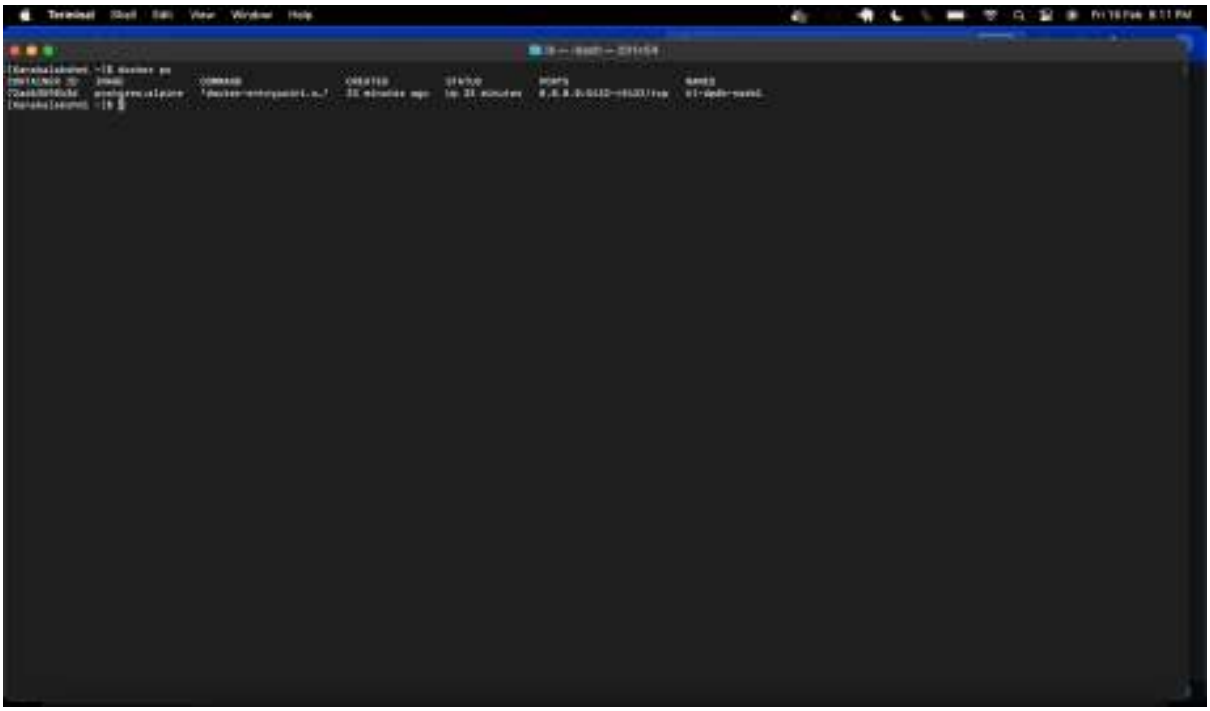


Fig-4: Check the list of Docker instances.

Below is the screenshot, which displays the list of Docker containers in the Docker Dashboard.

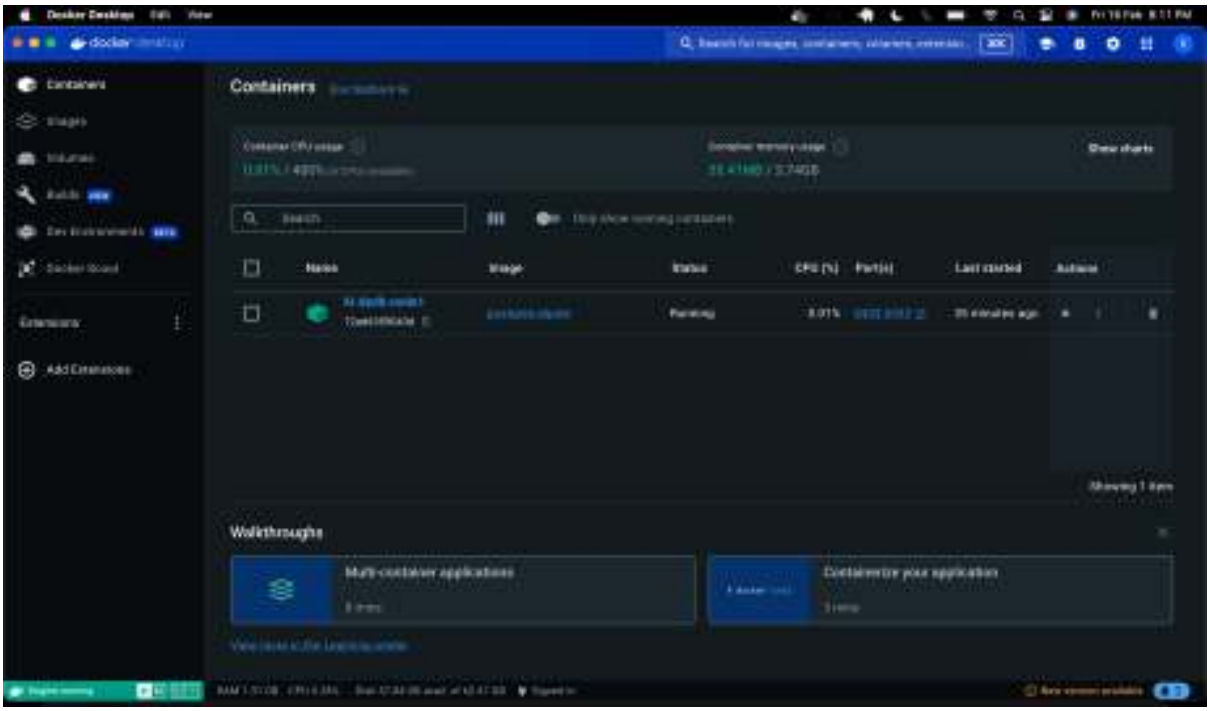


Fig-5: Docker Dashboard with the kl-dpdb-node1 instance.

Once the container is ready, we need to execute the container to start working on the respective machine. The command used to execute the container is
“docker exec -it kl-dpdb-node1 bash”

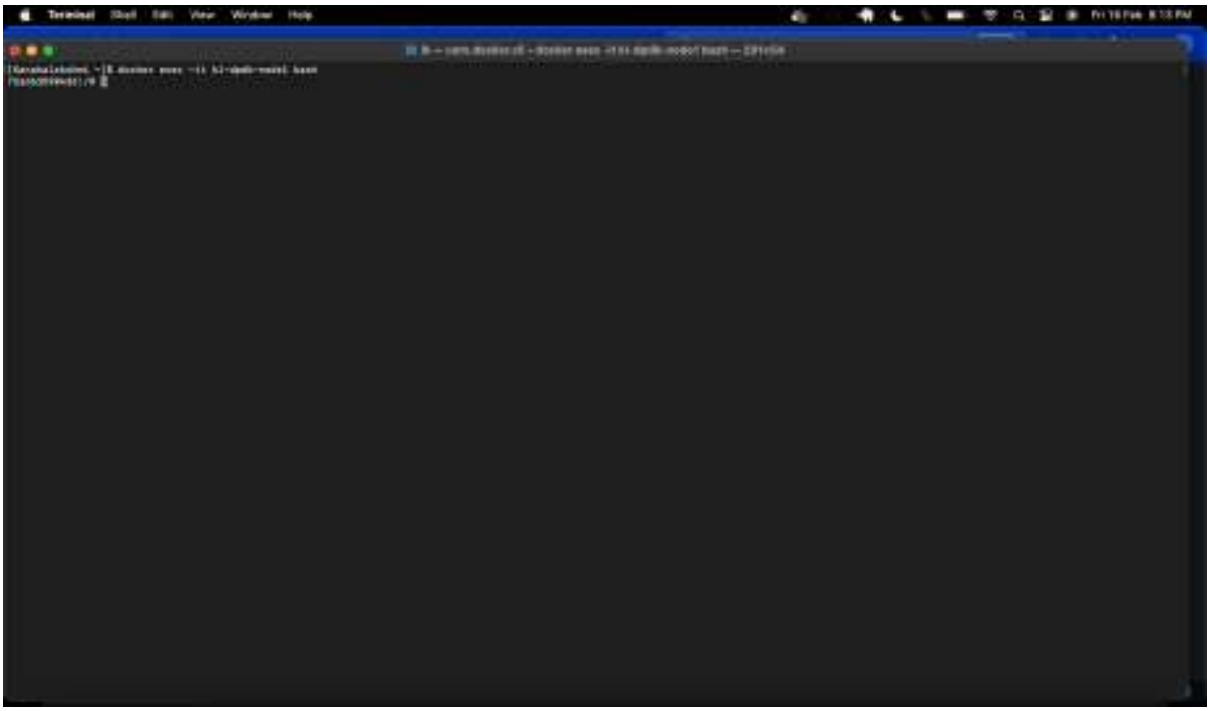


Fig-6: Execute running docker container ’kl-dpdb-node1’.
Now to work with the postgres, we need to connect to postgres from the terminal in docker. To conncect to postgres from docker container the command used is
“psql -U postgres”

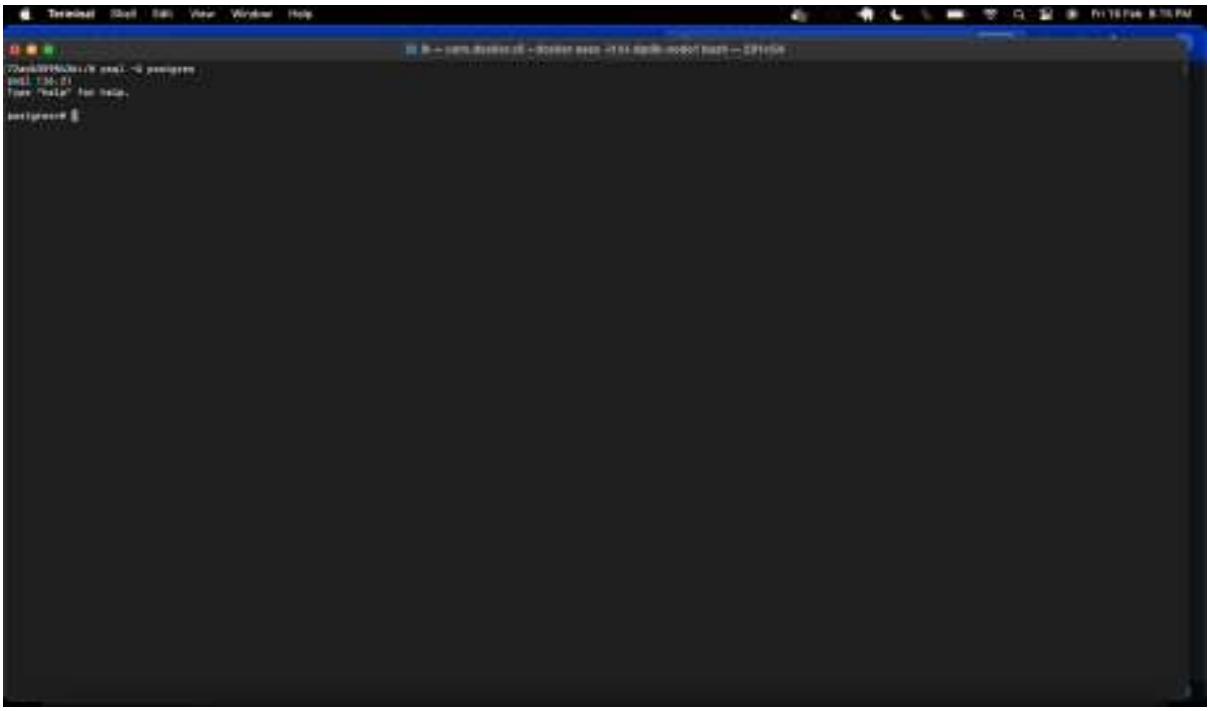


Fig-7: Connect to Postgres.

Create an ecommerce database to start working on project requirements. Command used to create ecommerce database is
“create database kl_dpdb_ecommerce_database;”

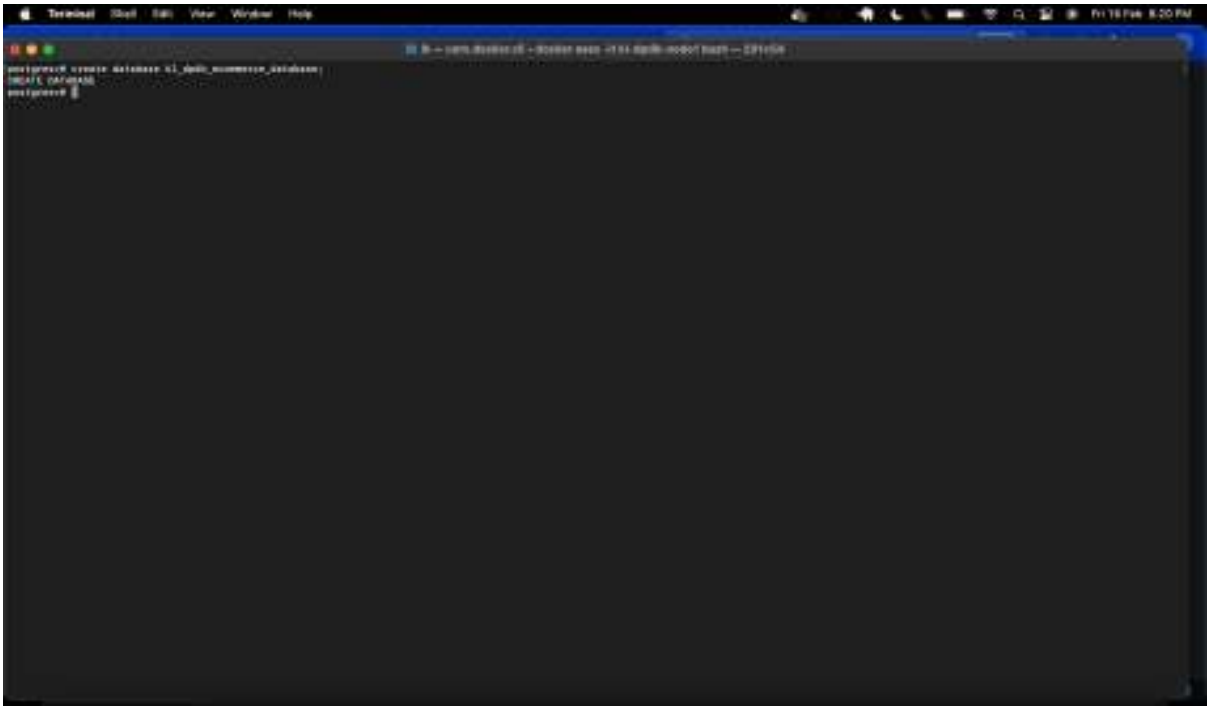


Fig-8: Create ’ kl_dpdb_ecommerce_database’ database in postgres.

Once the database is created, we see the list of databases using **"\l"** command. In the screenshot below, we can see the created database i.e., kl_dpdb_ecommerce_database

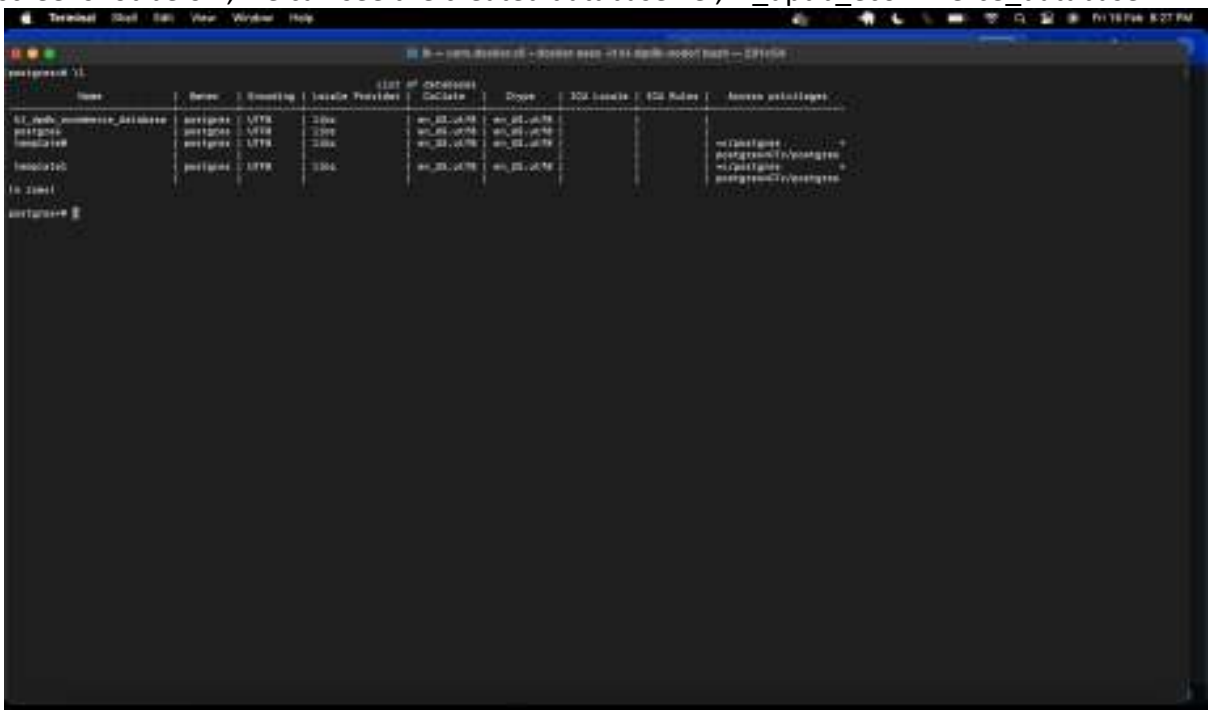


Fig-9: Check the list of databases.

To make use of created database, The command used is
"\c kl_dpdb_ecommerce_database"

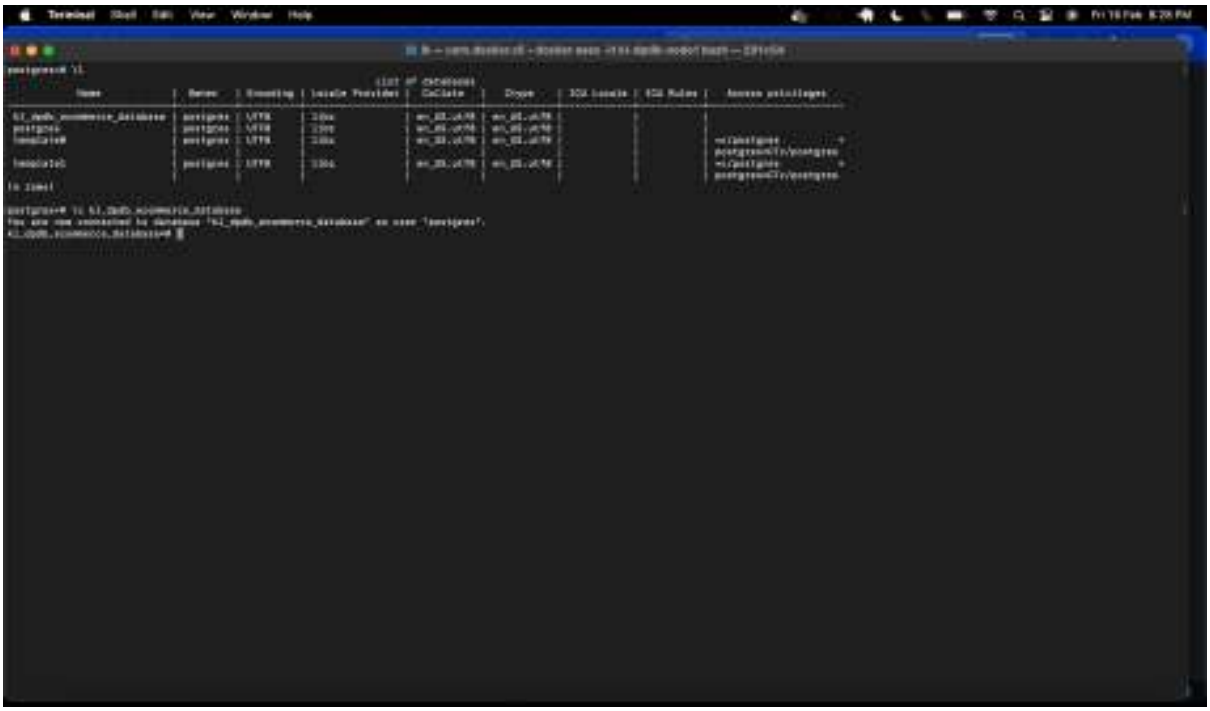


Fig-10: Use kl_dpdb_ecommerce_database.

Creation of Tables in kl_dpdb_ecommerce_database:

To generate the primary key automatically in RAW(16) format, we need to create the extension for (uuid - ossp). The command used to create the extension is:

```
-- uuid-ossp: the extension used to create primary key with UUID.
CREATE EXTENSION IF NOT EXISTS "uuid-ossp";
```

The uuid extension helps to generate the random 16-digit number. Which helps have the less collision.

address table:
query:

```
-- Create the address table
CREATE TABLE address (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
flat_no INTEGER NOT NULL,
street VARCHAR(50) NOT NULL,
city VARCHAR(50) NOT NULL,
state VARCHAR(50) NOT NULL,
country VARCHAR(50) NOT NULL,
zip_code VARCHAR(10) NOT NULL
);
```

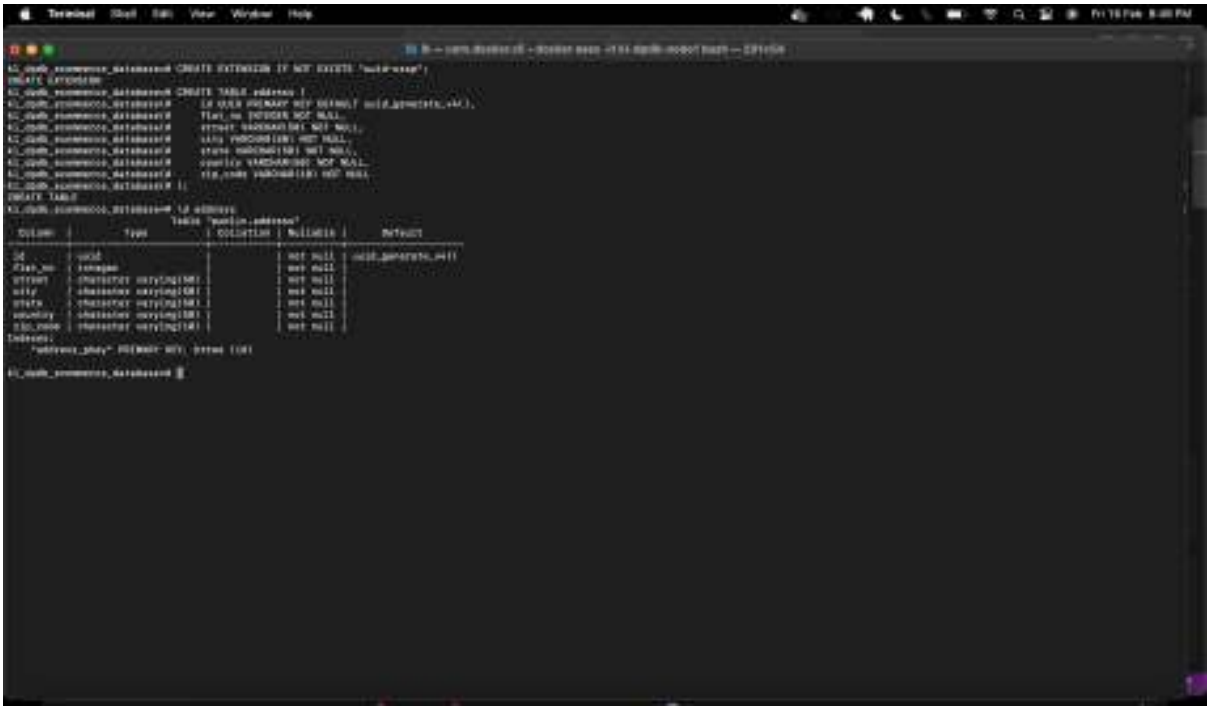


Fig-11: Create address table and display the schema of it.

Explanation: The Address table is used to store the address details of customers, products, and supplier. The address table has a unique identifier with the attribute flat_no, street, city, state, country, and the zip-code. The address table is one of the crucial tables in the e-commerce site which helps the delivery partners where to pick and deliver the products.
A customer can have many addresses of his/her choice. The customer can order the products from any address.

product_category table:
query:

```
-- create product_category table
CREATE TABLE product_category (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
section VARCHAR(20) NOT NULL,
audience_segment VARCHAR(20) NOT NULL
);
```

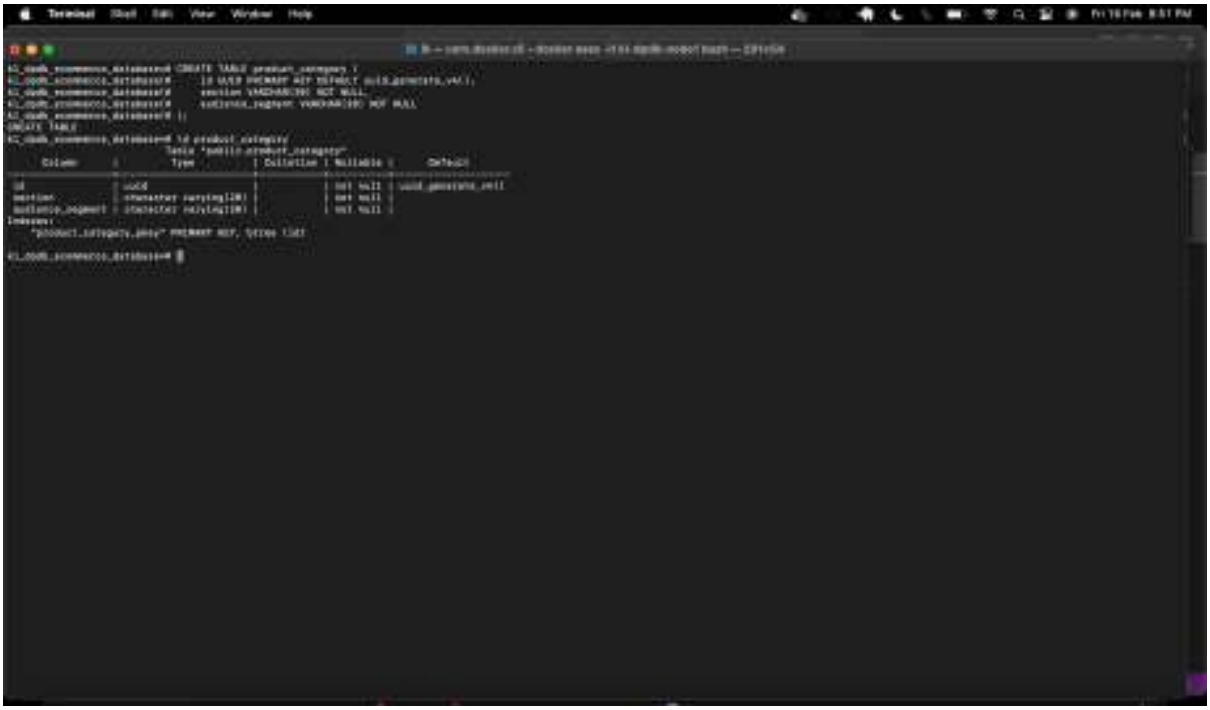


Fig-12: Create the product_category table and display the schema of it.

Explanation: The Products_Category table helps the customer to filter the products based on their choice. It helps the customer to search for products easily. There are different types of categories such as clothing, toys, accessories, etc. The product_category has a unique identifier, with attributes section and audience segment. The section talks about, in which section the products are in like clothing, accessories, Footwear, etc. and the audience_segment talks about to whom that product is for like kids, women, adults, girls, etc.

products table:

query:

```
-- create Products table
CREATE TABLE products (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
name VARCHAR(50) NOT NULL,
description VARCHAR(200) NOT NULL,
price INTEGER NOT NULL,
quantity INTEGER NOT NULL,
discount INTEGER,
brandname VARCHAR(20) NOT NULL,
address_id UUID NOT NULL REFERENCES address(id),
product_category_id UUID NOT NULL REFERENCES product_category(id)
);
```

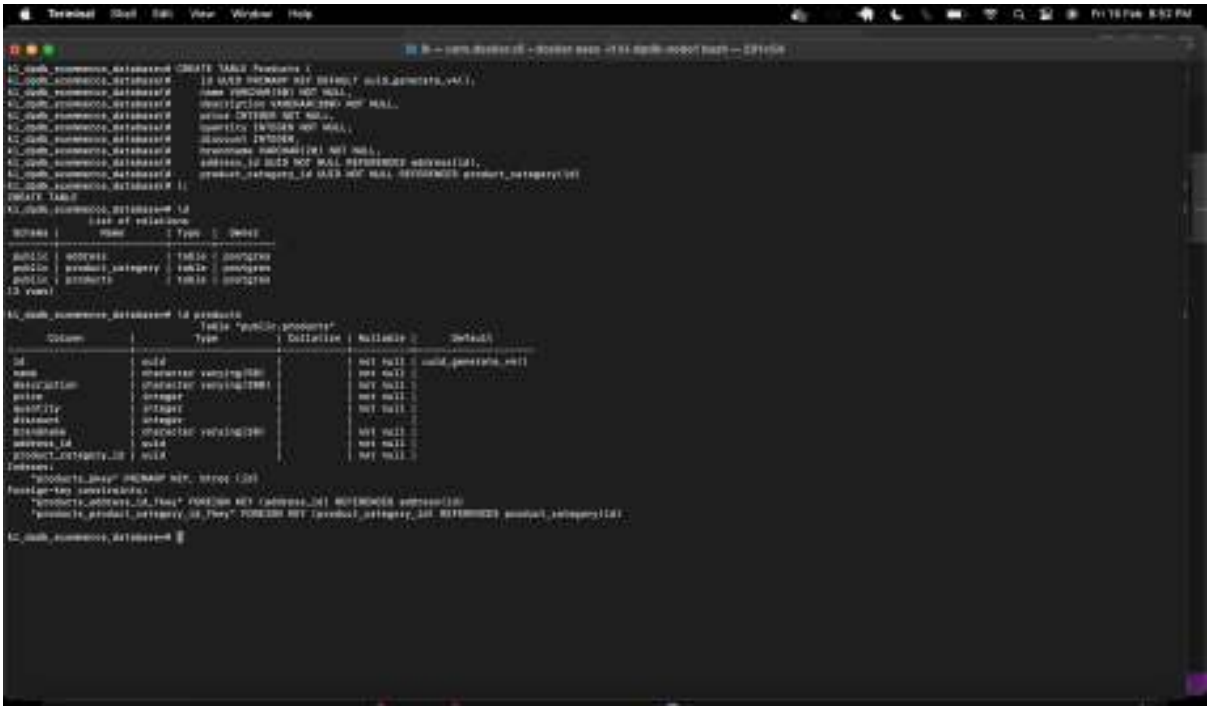


Fig-13: Create the products table and display the schema of it.

Explanation: The Products contain the list of products/items available. The products have a unique identifier with attributes name, description, price, quantity, discount, brandname, address_id and product_category_id. The address_id and the product_category_id are the foreign keys which refers to address and products_category table.

customer table:

query:

```
-- create customer table
CREATE TABLE customer (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
first_name VARCHAR(20) NOT NULL,
last_name VARCHAR(20) NOT NULL,
phone_no VARCHAR(20) NOT NULL,
email_id VARCHAR(255) NOT NULL,
dob DATE NOT NULL,
type VARCHAR(20) NOT NULL,
CONSTRAINT invalid_customer_phone CHECK (phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),
CONSTRAINT invalid_customer_email CHECK (email_id ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-
]+\.[A-Za-z]{2,}$'),
UNIQUE (phone_no),
UNIQUE (email_id)
);
```

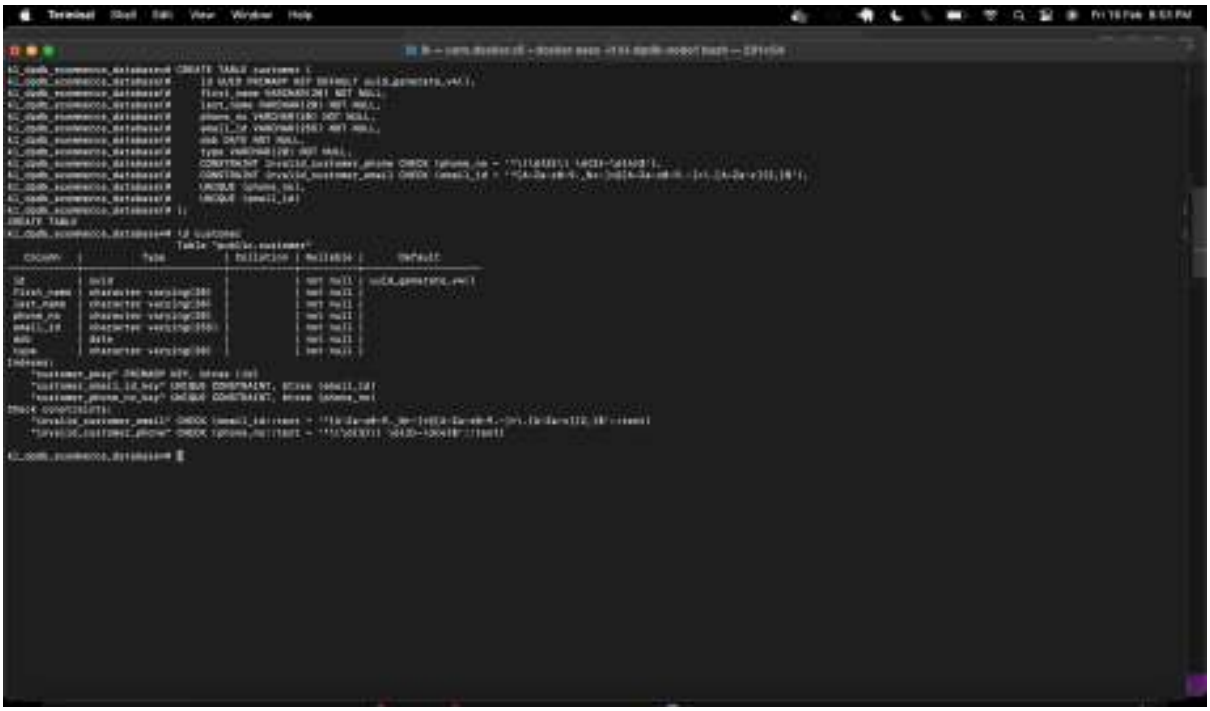


Fig-14: Create the customer table and display the schema of it.

Explanation: The customer table contains the list of customer details. This one is the crucial table. The customer table has a unique identifier with attributes first_name, last_name, phone_no, email_id, dob, and type. The phone_no and email_id have unique check constraints which helps to avoid duplicate customer accounts. The dob of the customer helps to provide the discount/gift voucher to a specific user in the birth month. The type of customer helps to identify whether the customer is new to the e-commerce site and based on the purchases the level of the customer increases like premium, gold, VIP etc.

cart table:

query:

```
-- create cart table
CREATE TABLE cart (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
customer_id UUID NOT NULL REFERENCES customer(id),
quantity INTEGER NOT NULL
);
```

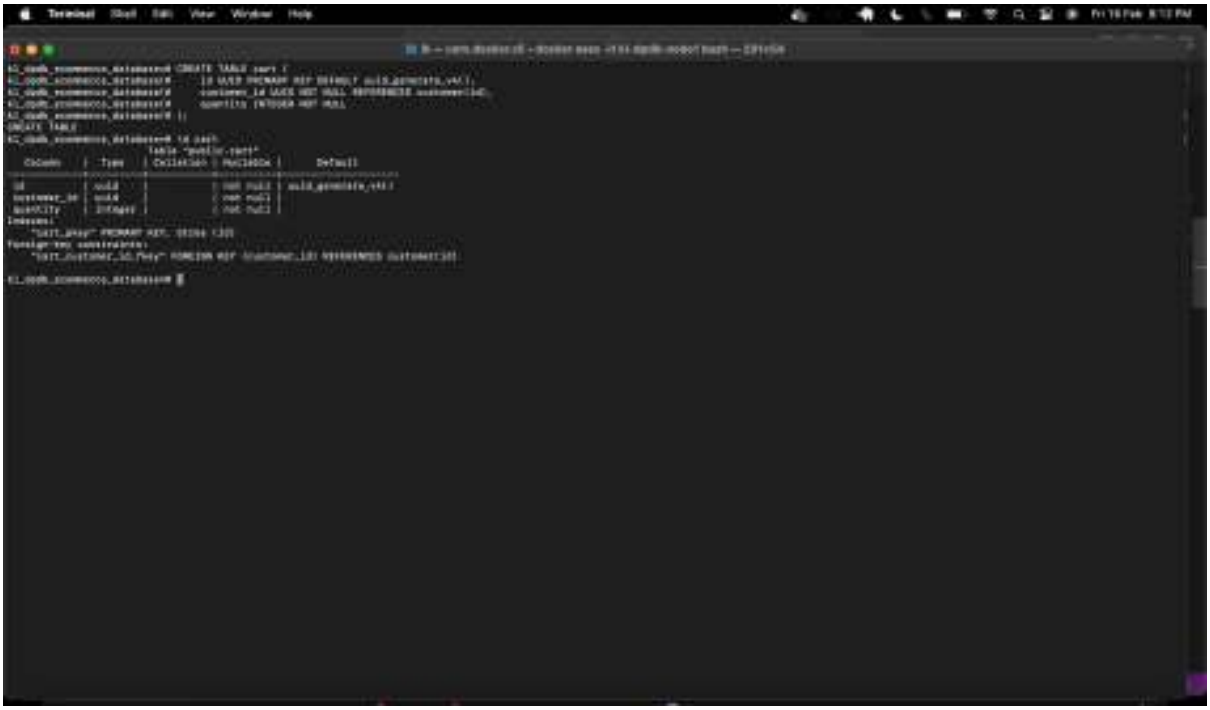


Fig-15: Create the cart table and display the schema of it.

Explanation: The cart table contains the list of products that customers want to buy. Each customer has a cart where they add/store the products. The cart table has a unique identifier with attributes customer_id and quantity. The customer_id is the foreign key reference to the customer table and the quantity talks about how many products the customer added to the cart.

orders table:

query:

```
-- create orders table
CREATE TABLE orders (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
status VARCHAR(20) NOT NULL,
order_date DATE NOT NULL,
address_id UUID NOT NULL REFERENCES address(id),
customer_id UUID NOT NULL REFERENCES customer(id)
);
```

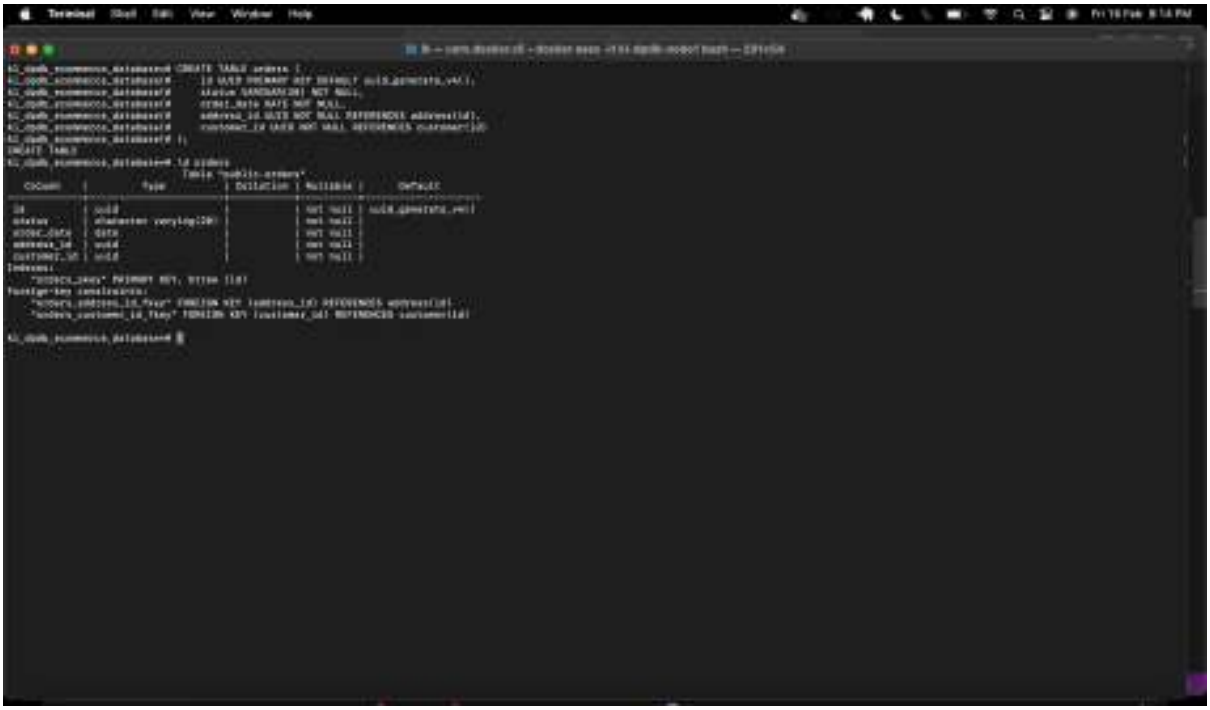


Fig-16: Create the orders table and display the schema of it.

Explanation: when the customer orders any product all the respective details are stored in the orders table. The orders table has a unique identifier with attributes status, order_date, address_id and customer_id. The status talks about the product delivery status such as processing, in transit, delivered etc. order_date talks about the on which date the order has placed by the customer. Here the address_id and customer_id are the foreign keys which refer to address table and customer table. This helps easily to check the orders details if the customer.

transaction_summary table:

query:

```
-- create transaction_summary table
CREATE TABLE transaction_summary (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
total_amount_paid INTEGER NOT NULL,
payment_type VARCHAR(20) NOT NULL,
date_of_payment DATE NOT NULL,
order_id UUID NOT NULL REFERENCES orders(id)
);
```

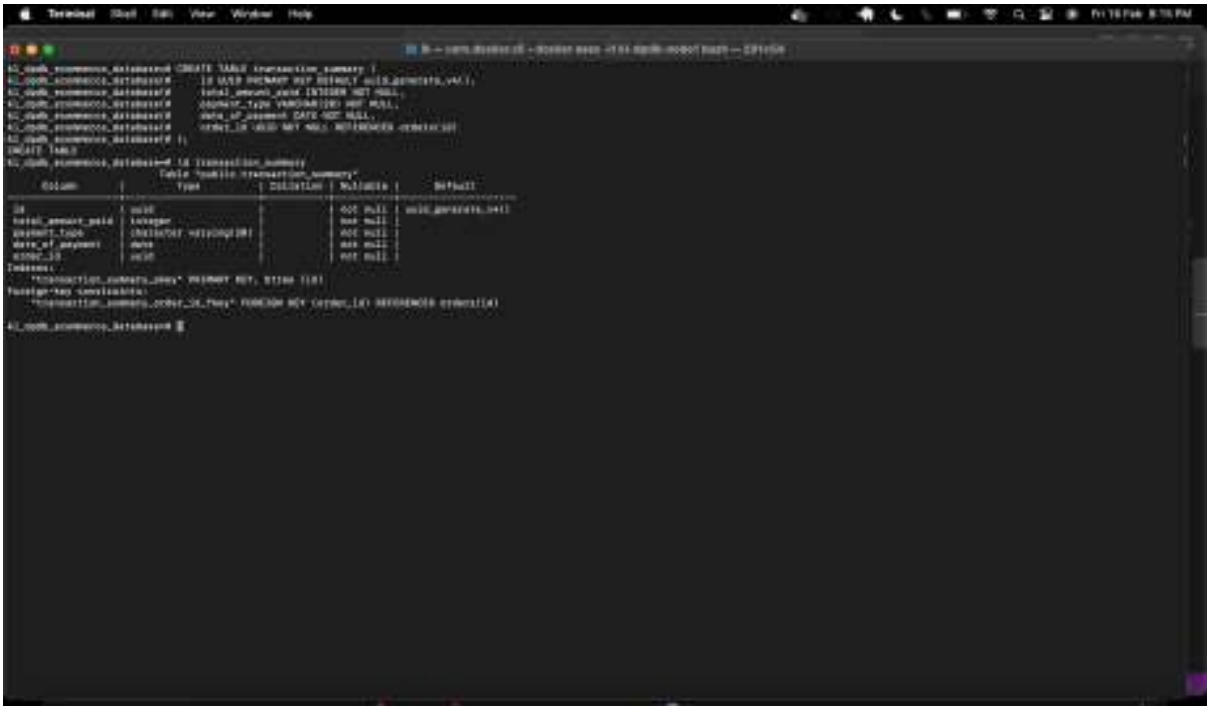


Fig-17: Create the transaction_summary table and display the schema of it.

Explanation: The transaction_summary table talks about the transactions done by the customers. The transaction_summary has a unique identifier with attributes total_amount_paid, payment_type, date_of_payment, order_id. The total_amount, talks about the total amount paid to by the customer on specific order. payment_type talks about the through which type of payment did customer placed the order such as Debit card, credit card, etc. The order_id is the foreign key which references to transaction_summary table.

supplier table:

query:

```
-- create supplier table
CREATE TABLE supplier (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
name VARCHAR(50) NOT NULL,
phone_no VARCHAR(20) NOT NULL UNIQUE CONSTRAINT invalid_supplier_phone_no CHECK
(phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),
email VARCHAR(255) NOT NULL UNIQUE CONSTRAINT invalid_supplier_email CHECK (email ~
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'),
rating INTEGER,
address_id UUID NOT NULL REFERENCES address(id)
);
```

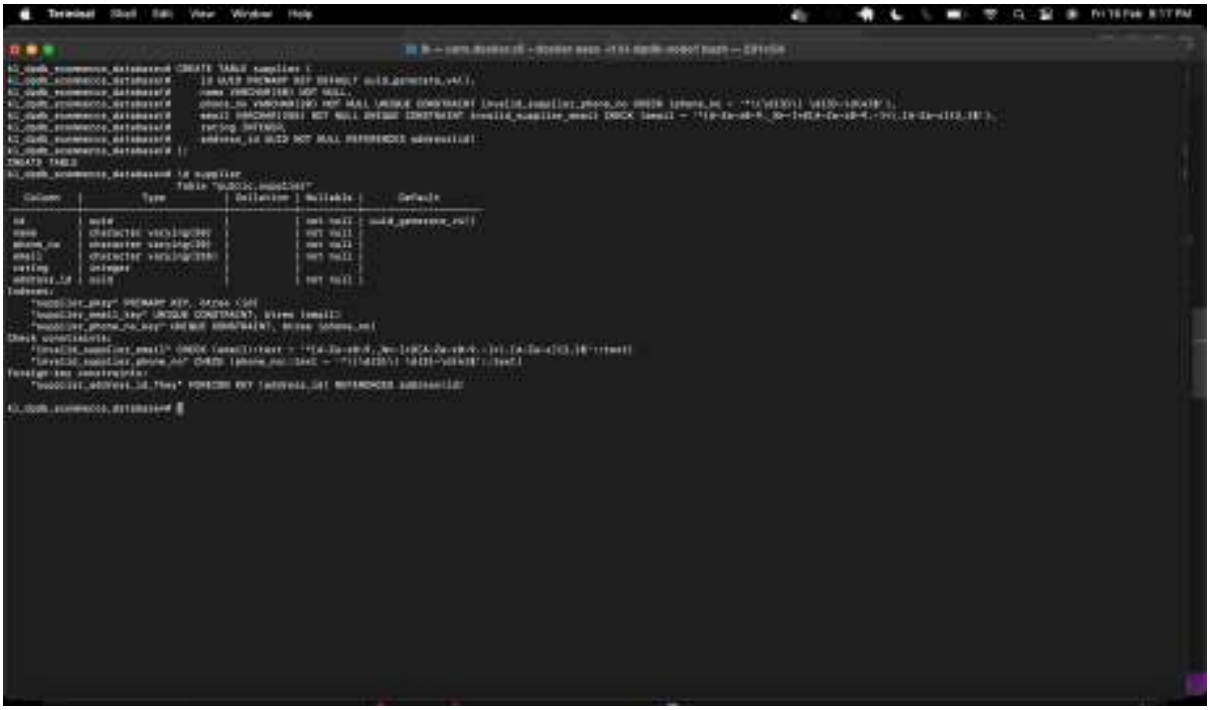


Fig-18: Create the supplier table and display the schema of it.

Explanation: The supplier table stores the details of the supplier. The supplier table has a unique identifier with attributes name, phone_no, email, rating and address_id. The name identifes the name of the supplier, rating helps to purchase the product easily. The phone_no and email has the check constraints to have security and unique to avoid the duplicate accounts. The address_id is the foreign key which references the address table.

reviews table:

query:

```
-- create reviews table
CREATE TABLE reviews (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
rating INTEGER,
comments VARCHAR(150),
product_id UUID REFERENCES products(id),
order_id UUID REFERENCES orders(id),
customer_id UUID REFERENCES customer(id)
);
```

Fig-19: Create the reviews table and display the schema of it.

Explanation: The reviews table plays an important role in knowing more about the product. As the reviews are written by the customers, there will be no chance of false marketing about the products. The reviews table has the unique identifier with attributes rating, comments, product_id, customer_id and order_id. The rating talks about what the rating of the product is on scale of 5 like 1,2,3,4,5 etc. The comments help them to write their opinion about the product how did they felt about it etc. The product_id, customer_id and order_id are the foreign keys which refers to the products, customer and order tables.

gift_vouchers table:

query:

```
-- create gift_vouchers table
CREATE TABLE gift_vouchers (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
name VARCHAR(50),
number INTEGER,
amount INTEGER,
expiry_date DATE
);
```

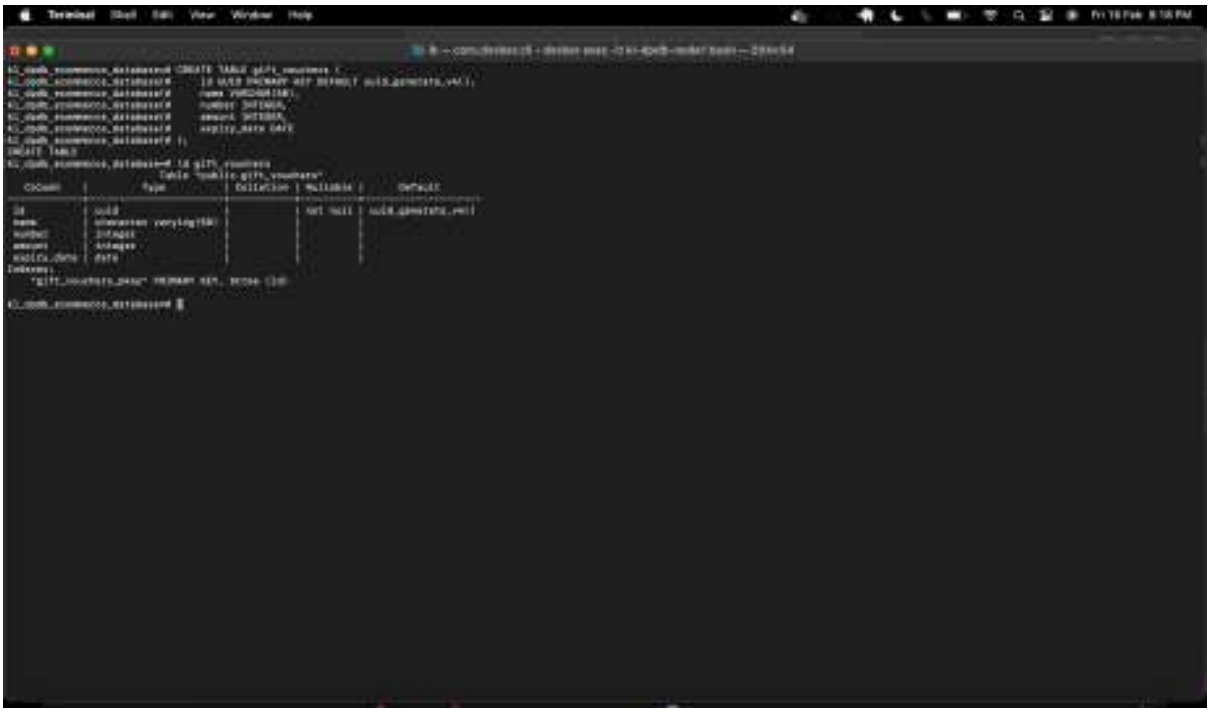


Fig-20: Create the gift_vouchers table and display the schema of it.

Explanation: The gift_vouchers table stores the list of gift voucher details. The gift voucher has a unique identifier and with attributes name, number, amount and expiry_date. The name identifies the name of the gift voucher, number stores the gift voucher number, amount stores the amount allocated to the gift voucher, expiry_date contains date on which day the voucher is going to get expired. The gift voucher is one the marketing techniques used by many of the e-commerce sites.

delivery_partner table:

query:

```
-- create delivery_partner table
CREATE TABLE delivery_partner (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
name VARCHAR(50) NOT NULL,
phone_no VARCHAR (20) NOT NULL CONSTRAINT invalid_delivery_partner_phone_no CHECK
(phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),
email VARCHAR(255) NOT NULL CONSTRAINT invalid_delivery_partner_email CHECK (email ~
'^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'),
order_id UUID NOT NULL REFERENCES orders(id)
);
```

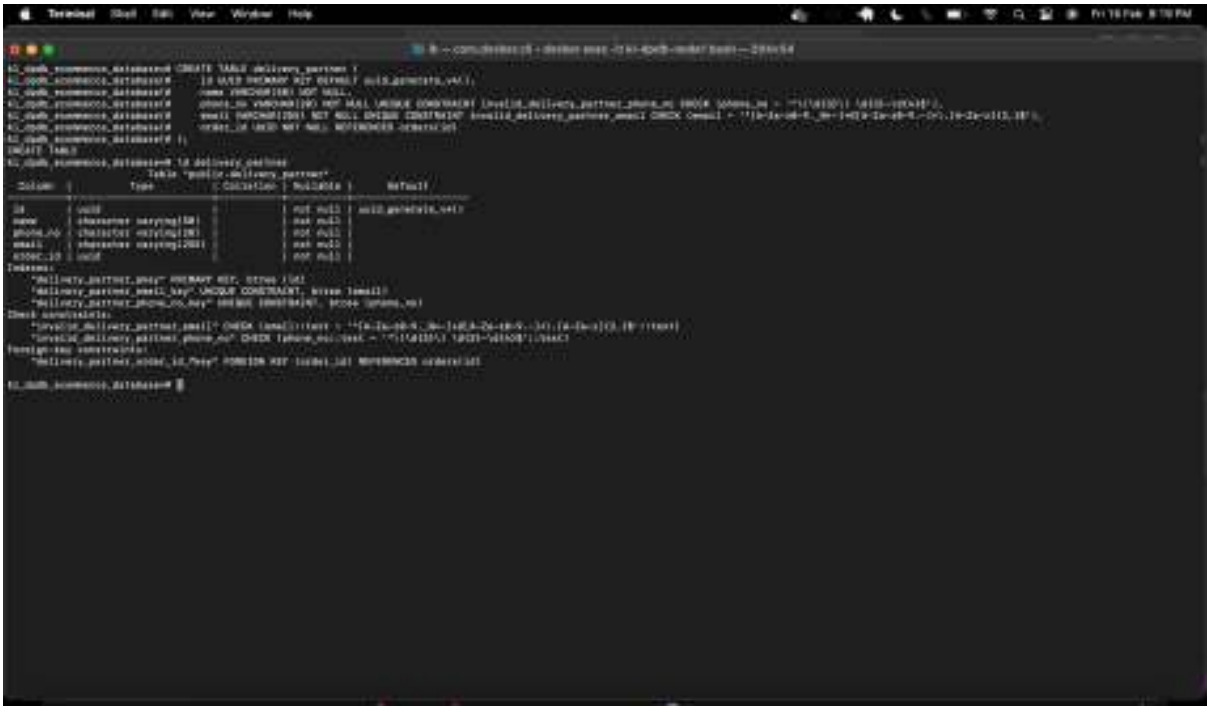


Fig-21: Create the delivery_partner table and display the schema of it.

Explanation: The delivery_partner table stores the delivery_partner details and the order associated to respective delivery partner. The delivery partner has a unique identifier with attributes name, phone_no, email, order_id. The name is the name of the delivery partner, phone_no and email are the details of the delivery partner which helps the customers to contact them. The phone_no and email has check constraints for security reasons. The order_id is the foreign key which references the orders table.

customer_address table:

query:

```
-- create customer_address table
CREATE TABLE customer_address (
customer_id UUID NOT NULL,
address_id UUID NOT NULL,
default_address BOOLEAN NOT NULL,
PRIMARY KEY (customer_id, address_id),
FOREIGN KEY (customer_id) REFERENCES customer(id),
FOREIGN KEY (address_id) REFERENCES address(id)
);
```

Fig-22: Create the customer_address table and display the schema of it.

Explanation: The relationship between the customer and address is many-to-many. This relation table forms the intermidiary link between the customer and address. In this relation table customer_id and address_id is the primary and foreign keys which refer to the customer and address table. There is an attribute, default address – which tells that default address of the customer. This attribute majorly helps when a customer has more than one address.

customer_gift_voucher table:

query:

```
-- create customer_gift_voucher table
CREATE TABLE customer_gift_voucher (
customer_id UUID NOT NULL,
gift_voucher_id UUID NOT NULL,
count INTEGER NOT NULL,
PRIMARY KEY (customer_id, gift_voucher_id),
FOREIGN KEY (customer_id) REFERENCES customer(id),
FOREIGN KEY (gift_voucher_id) REFERENCES gift_vouchers(id)
);
```

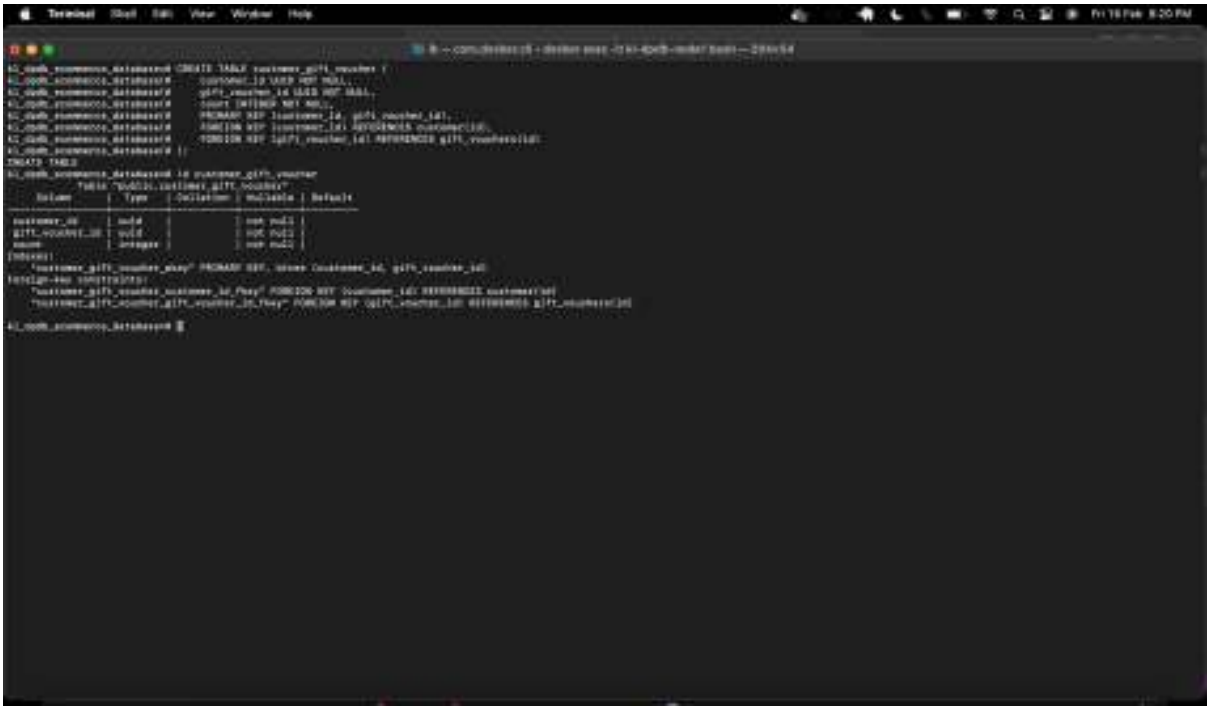


Fig-23: Create the Customer_gift_voucher table and display the schema of it.

Explanation: The relationship between the customer and the gift voucher is many-to-many. This relation table forms the intermediary link between the customer and the gift voucher. The customer_id and gift_voucher_id are the primary and foreign keys, refers to the customer and gift_vouchers table. There is an attribute count which talks about the number of gift vouchers allocated to the customer.

customer_delivery_partner table:

query

```
-- create customer_delivery_partner table
CREATE TABLE customer_delivery_partner (
customer_id UUID NOT NULL,
delivery_partner_id UUID NOT NULL,
PRIMARY KEY (customer_id, delivery_partner_id),
FOREIGN KEY (customer_id) REFERENCES customer(id),
FOREIGN KEY (delivery_partner_id) REFERENCES delivery_partner(id)
);
```

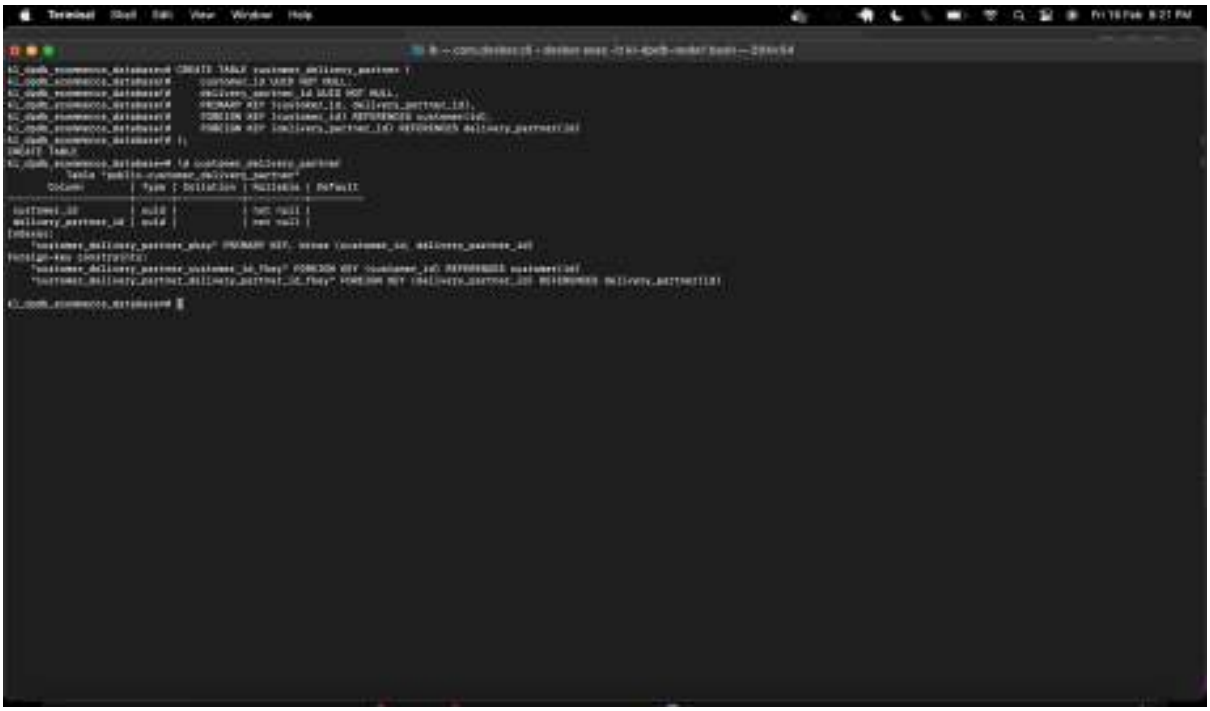


Fig-24: Display and Create the customer_delivery_partner table and display the schema of it.

Explanation: The relationship between the customer and the delivery partner is many-to-many. This relation table forms the intermediary link between customer and the delivery_partner. The table contains customer_id and delivery_partner_id as primary and the foreign keys which refer to customer and delivery_partner tables.

products_cart table:

query:

```
-- create products_cart table
CREATE TABLE products_cart (
product_id UUID NOT NULL,
cart_id UUID NOT NULL,
count INTEGER NOT NULL,
PRIMARY KEY (product_id, cart_id),
FOREIGN KEY (product_id) REFERENCES products(id),
FOREIGN KEY (cart_id) REFERENCES cart(id)
);
```

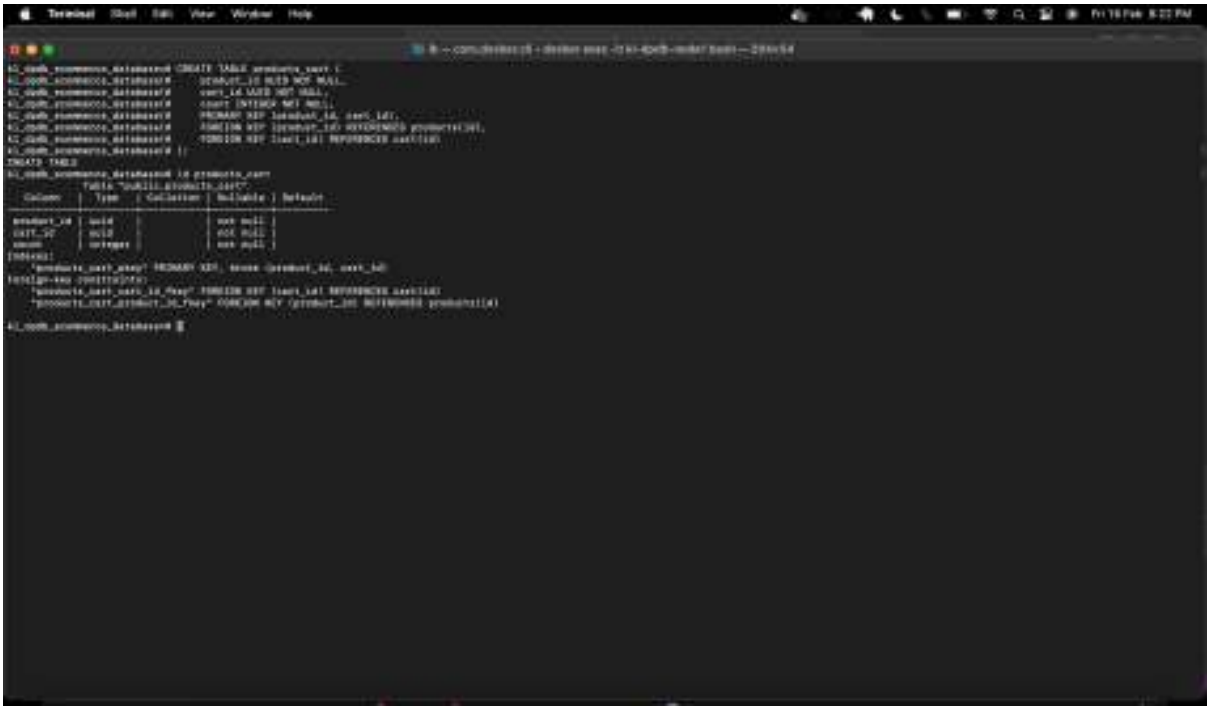


Fig-25: Create the products_cart table and display the schema of it.

Explanation: The relationship between the products and the cart table is many-to-many. The intermediary between these two tables forms the relation table. The table has the primary and foreign keys, product_id and cart_id which refers to the product and the cart table. This contains the attribute as count which helps to identify the count of products in cart. By this we can notify the customers if there is any discount on the respective product.

orders_products:

query:

```
-- create orders_products table
CREATE TABLE orders_products (
order_id UUID NOT NULL,
product_id UUID NOT NULL,
quantity INTEGER NOT NULL,
PRIMARY KEY (order_id, product_id),
FOREIGN KEY (order_id) REFERENCES orders(id),
FOREIGN KEY (product_id) REFERENCES products(id)
);
```

Fig-26: Create and Display the orders_products table and schema of it.

Explanation: The relationship between the orders and the products table is many-to-many. The intermediary link between the orders and the product table forms a relation table. The table has order_id and product_id as primary key and the foreign key, which refers to the order and product table. The table as attribute quantity which stores the quantity of the ordered products by the customer.

products_suppliers table:

query:

```
-- create products_suppliers table
CREATE TABLE products_suppliers (
product_id UUID NOT NULL,
supplier_id UUID NOT NULL,
quantity INTEGER NOT NULL,
PRIMARY KEY (product_id, supplier_id),
FOREIGN KEY (product_id) REFERENCES products(id),
FOREIGN KEY (supplier_id) REFERENCES supplier(id)
);
```

Fig-27: Create and display schema for products_suppliers.

Explanation: The relationship between the products and the suppliers table is many-to-many. The intermediary link between the products and the supplier table forms the relation table. The product_id and the supplier_id are the primary and foreign keys which refer to the product and supplier tables. The table has an attribute quantity which stores the quantity of the products supplied by the supplier. This helps easily to find the products supplied by the supplier.

List of tables created for e-commerce database:

The command used to display the list of tables in a specific database is **"\d".** As part of the e-commerce database created 17 tables (11 entities and the 6 relation tables).

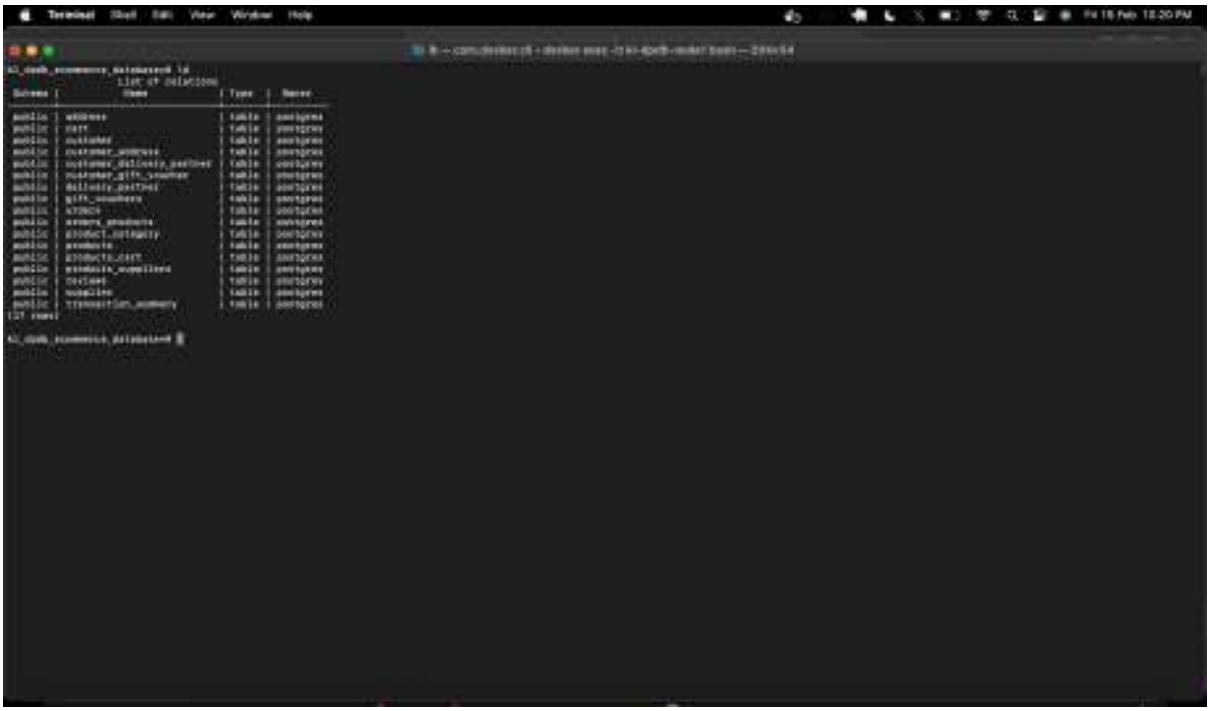


Fig-28: Display the list of tables in kl-dpdb-ecommerce database.

	Normalization							
Table	**Attribute = Short Forms**	**Functional Dependencies**	**Closure**	**Candidate Key**	**Prime**	**Non-Prime**	**Highest Normal Form**	**Justification**
products	id = A(primary key)		A+ = ABCDEFGHI	A	A	B, C, D, E, F, G, H, I	The highest normal form of the products table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	name = B		B+ = B					
	description = C		C+ = C					
	price = D		D+ = D					
	qunatity = E		E+ = E					
	discount = F		F+ = F					
	brandname = G		G+ = G					
	address_id = H		H+ H					
	product_category_id = I	A->B, A ->C, A->D, A->E, A->F, A->G, A->H, A ->I	I+ = I					
customer	id = A(primary key)		A+ = ABCDEFG	A	A	B, C, D, E, F, G	The highest normal form of the customer table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	first_name = B		B+ = B					
	last_name = C		C+ = C					
	phone_no = D		D+ = D					
	email_id = E		E+ = E					
	dob = F		F+ = F					
	type = G	A->B, A ->C, A->D, A->E, A->F, A->G	G+ = G					
cart	id = A		A+ = ABC	A	A	B, C	The highest normal form of the cart table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	customer_id = B		B+ = B					
	quantity = C	A->B, A ->C	C+ = C					
orders	id = A		A+ = ABCDE	A	A	B, C, D, E	The highest normal form of the orders table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	status = B		B+ = B					
	order_date = C		C+ = C					
	address_id = D		D+ = D					
	customer_id = E	A->B, A ->C, A->D, A->E	E+ = E					
transaction_summary	id = A		A+ = ABCDE	A	A	B, C, D, E	The highest normal form of the transaction_summary table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	total_amount_paid = B		B+ = B					
	payment_type = C		C+ = C					
	date_of_payment = D		D+ = D					
	order_id = E	A->B, A ->C, A->D, A->E	E+ = E					
supplier	id = A		A+ = ABCDE	A	A	B, C, D, E, F	The highest normal form of the supplier table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	name = B		B+ = B					
	phone_no = C		C+ = C					
	email = D		D+ = D					
	rating = E		E+ = E					
	address_id = F	A->B, A ->C, A->D, A->E, A-> F	F+ = F					
address	id = A		A+ = ABCDEFG	A	A	B, C, D, E, F, G	The highest normal form of the address table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	flat_no = B		B+ = B					
	street = C		C+ = C					
	city = D		D+ = D					
	state = E		E+ = E					
	country = F		F+ = F					
	zip_code = G	A->B, A ->C, A->D, A->E, A->F, A->G	G+ = G					
	id = A		A+ = ABC				The highest normal	All of the left hand keys of FDs are

	Normalization							
Table	**Attribute = Short Forms**	**Functional Dependencies**	**Closure**	**Candidate Key**	**Prime**	**Non-Prime**	**Highest Normal Form**	**Justification**
product_category	section = B	A->B, A ->C	B+ = B	A	A	B, C	form of the product_category table is "BCNF"	superkeys, hence the table is in BCNF.
	audience_segment = C		C+ = C					
reviews	id = A	A->B, A ->C, A->D, A->E, A->F	A+ = ABCDEFG	A	A	B, C, D, E, F	The highest normal form of the reviews table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	rating = B		B+ = B					
	comments = C		C+ = C					
	product_id = D		D+ = D					
	order_id = E		E+ = E					
	customer_id = F		F+ = F					
gift_vouchers	id = A	A->B, A ->C, A->D, A->E	A+ = ABCDE	A	A	B, C, D, E	The highest normal form of the gift_vouchers table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	name = B		B+ = B					
	number = C		C+ = C					
	amount = D		D+ = D					
	expiry_date = E		E+ = E					
delivery_partner	id = A	A->B, A ->C, A->D, A->E	A+ = ABCDE	A	A	B, C, D, E	The highest normal form of the delivery_partner table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	name = B		B+ = B					
	phone_no = C		C+ = C					
	email = D		D+ = D					
	order_id = E		E+ = E					
customer_address (relation table)	customer_id = A	AB->C	AB+ = ABC	AB	AB	C	The highest normal form of the customer_address table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	address_id = B							
	default_address = C		C+ = C					
customer_gift_voucher (relation table)	customer_id = A	AB->C	AB+ = ABC	AB	AB	C	The highest normal form of the customer_gift_voucher table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	gift_voucher_id = B							
	count = C		C+ = C					
customer_delivery_partner (relation table)	customer_id = A	As the table contains only primary keys there will be no functional dependencies in the table.	NA	AB	AB	NA	The highest normal form of the customer_delivery_partner table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	delivery_partner_id = B							
products_cart (relation table)	product_id = A	AB->C	AB+ = ABC	AB	AB	C	The highest normal form of the products_cart table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	cart_id = B							
	count = C		C+ = C					
orders_products (relation table)	orders_id = A	AB->C	AB+ = ABC	AB	AB	C	The highest normal form of the orders_products table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	products_id = B							
	quantity = C		C+ = C					
products_suppliers (relation table)	product_id = A	AB->C	AB+ = ABC	AB	AB	C	The highest normal form of the products_suppliers table is "BCNF"	All of the left hand keys of FDs are superkeys, hence the table is in BCNF.
	supplier_id = B							
	quantity = C		C+ = C					

Requirement -3:

There are a number of factors to consider when dealing with databases to ensure optimal performance, including data control, data consistency, the need for ACID features, appropriate fragmentation, etc.

When working with databases, the fragmentation of data is an important technique. Data is kept in an organized manner using fragmentation. We won't be aware of the data's scaling capabilities when we create and save the data in the tables. Fragmentation becomes crucial when we have a circumstance where our memory is full. It is possible to execute fragmentation across the records or columns. Fragmenting data across columns is referred to as vertical fragmentation, and fragmenting data across records is referred to as horizontal fragmentation. The fragmentation can be done on both the records and the colums, called as the hybrid fragmentation.

Completing the horizontal fragmentation on the customer table in accordance with the need. After horizontal fragmentation is complete, the fragmentation's accuracy must be assessed. Completeness, reconstruction, and disjointness can be used to determine whether the fragmentation is valid.

To perform the horizontal fragmentation the table should have number of records. So inserted 1200 records.

query:

```
INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type)
VALUES ('5aec2394-4bb8-4dc2-afbf-590f03cd0414', 'tdLmVeuWfq', 'jHgTMhnFTV', '(787)
510-9892', 'tscep@example.com', '1984-11-17', 'regular');
INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type)
VALUES ('47580a39-1b9d-472b-a2b6-6b1bf46d1b6c', 'sfNcUcAYai', 'LCJqrPCVIv', '(463)
662-8591', 'omqwo@example.com', '1964-05-18', 'VIP');
INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type)
VALUES ('c7d39a92-9d22-4ddf-9e0f-3a11860634c2', 'XLLQslUxVc', 'dkNCFcWpOG', '(354)
220-6490', 'dawge@example.com', '1979-12-18', 'platinum');
.
.
.
1200 records.
```

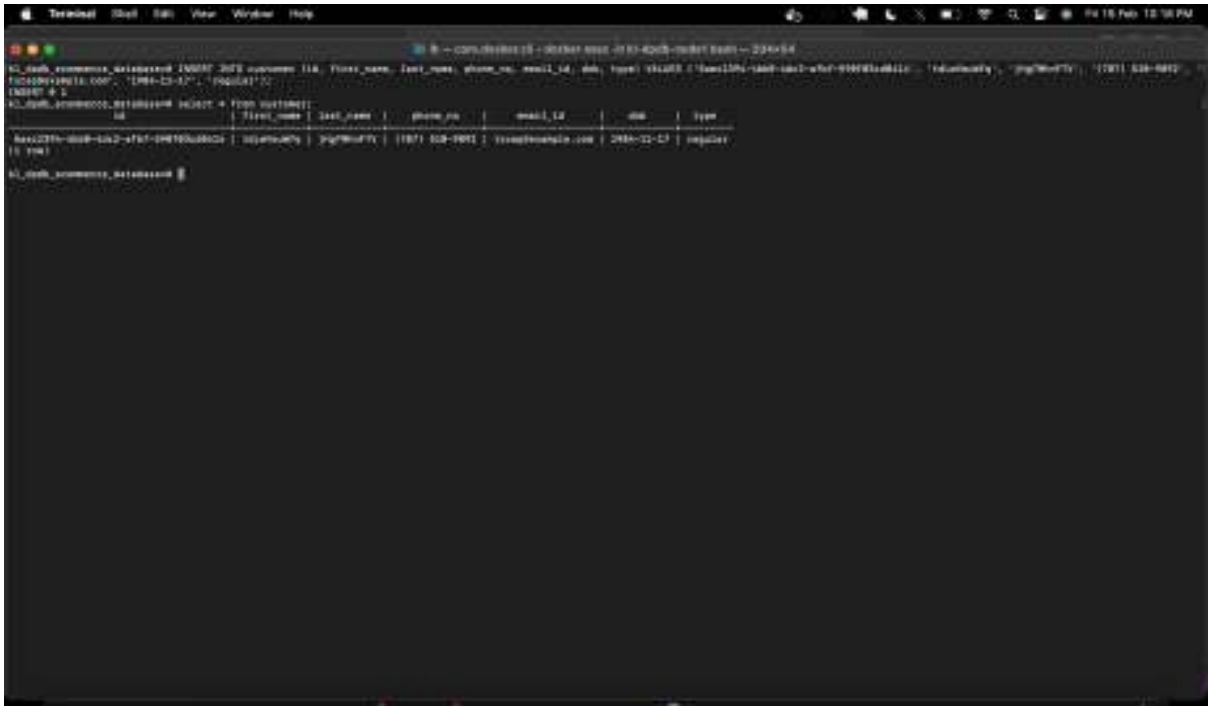


Fig-29: Insert data into the customer table.

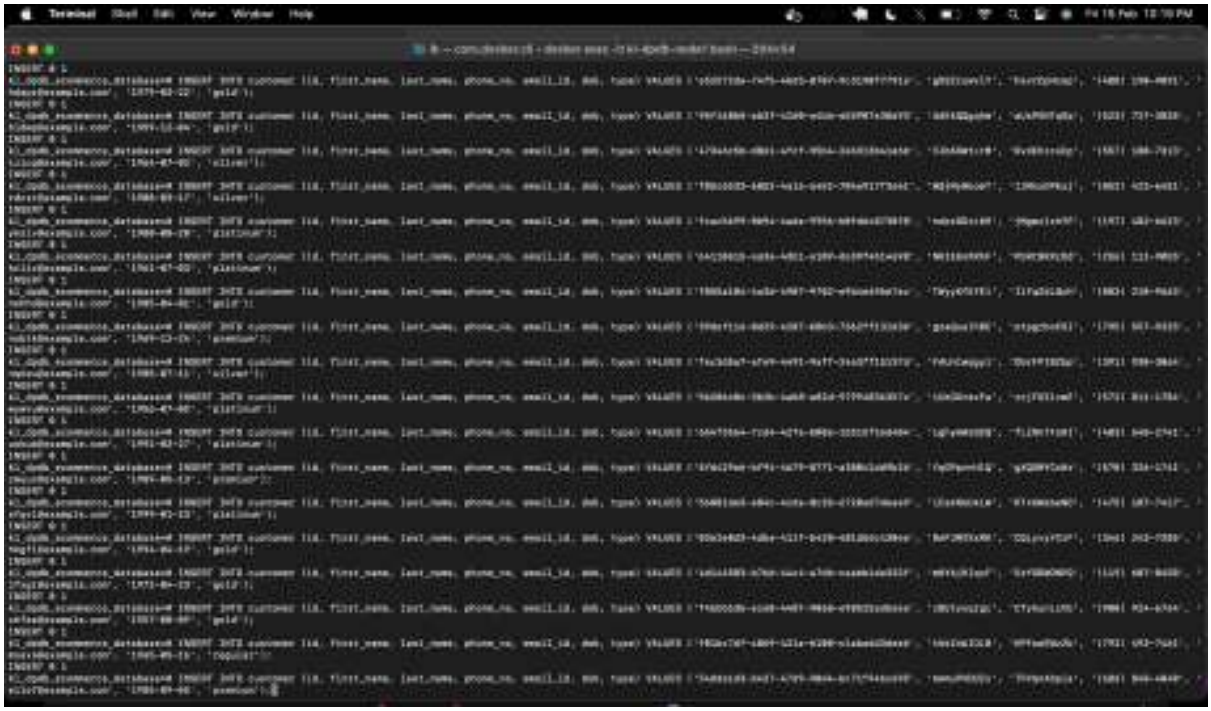


Fig-30: Insert 1200 records into the customer table.

Explanation: Inserted 1200 records to perform the horizontal fragmentation. If we have a greater number of records, we can clearly visualize the performance of implemented horizontal fragmentation.

query:

```
SELECT count(*) from customer;
```

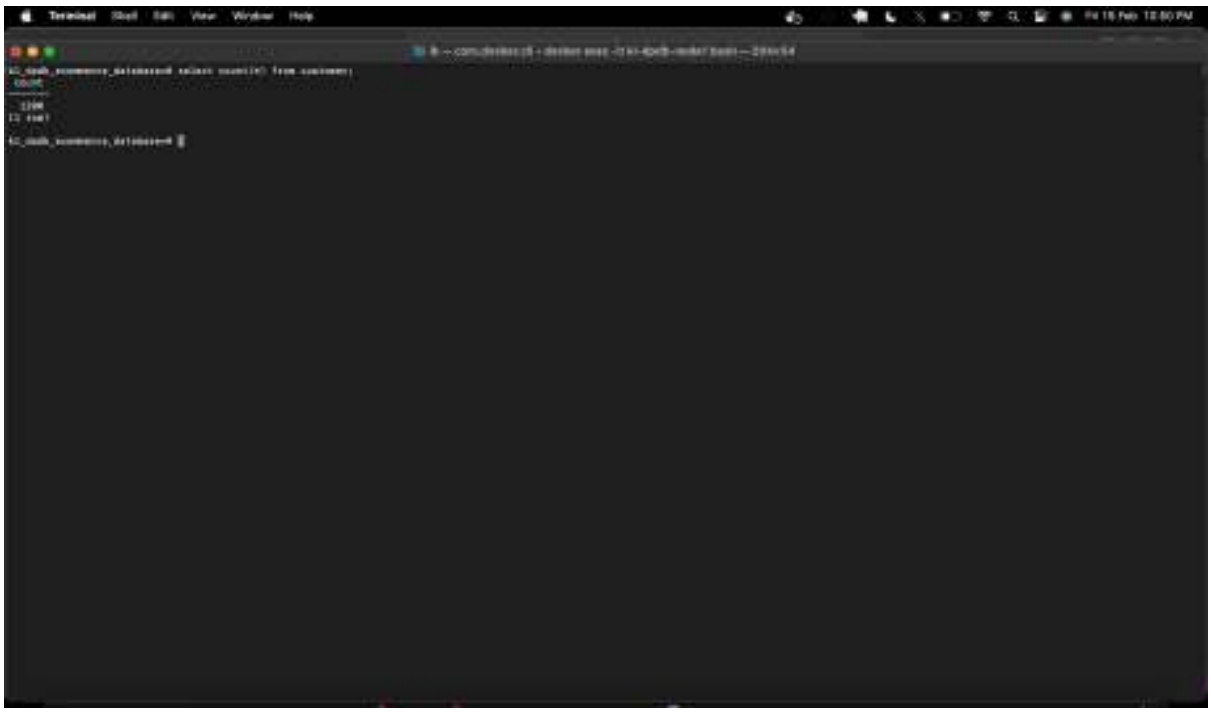


Fig-30: Display the count of records in the customer table.

Explanation: There are total 1200 number of records in the customer table.

Fragmentation:

As working only on the single table used **Primary Horizontal Fragmentation.**

query:

```
EXPLAIN ANALYZE SELECT * FROM customer WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
```

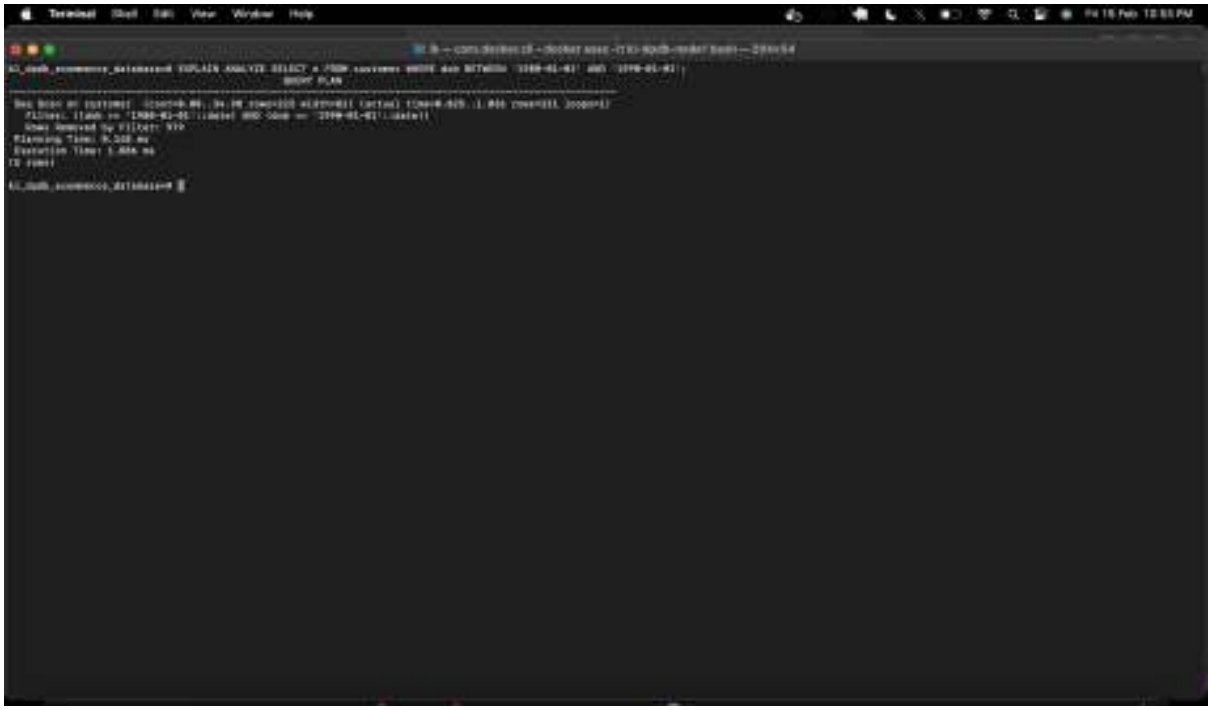


Fig-31: Analyze the performance of customer table.

Explanation: To perform the fragmentation initially we need to analyze the performance of query on the respective table. Observed that, while query is executing it applied the filter based on the given condition i.e., on dob. It took the **planning time of 0.148** milli seconds and **Execution time of 1.086** milli seconds.

query:

```
select version() -- to find the current version of postgres
create extension pgstattuple -- to create pgstattuple extension
```

Fig-32: Verify the postgres version and create pgstattuple extension.

Explanation: when we are working on fragmentation, we need to be aware of tuple information. This can be done using “pgstattule”. This was available from postres-9.4 and later versions. To make use of pgstatuple we need to create the extension for this, so that we can evaluate the statistics of the table.

query:

```
select * from pgstattuple('customer');
```

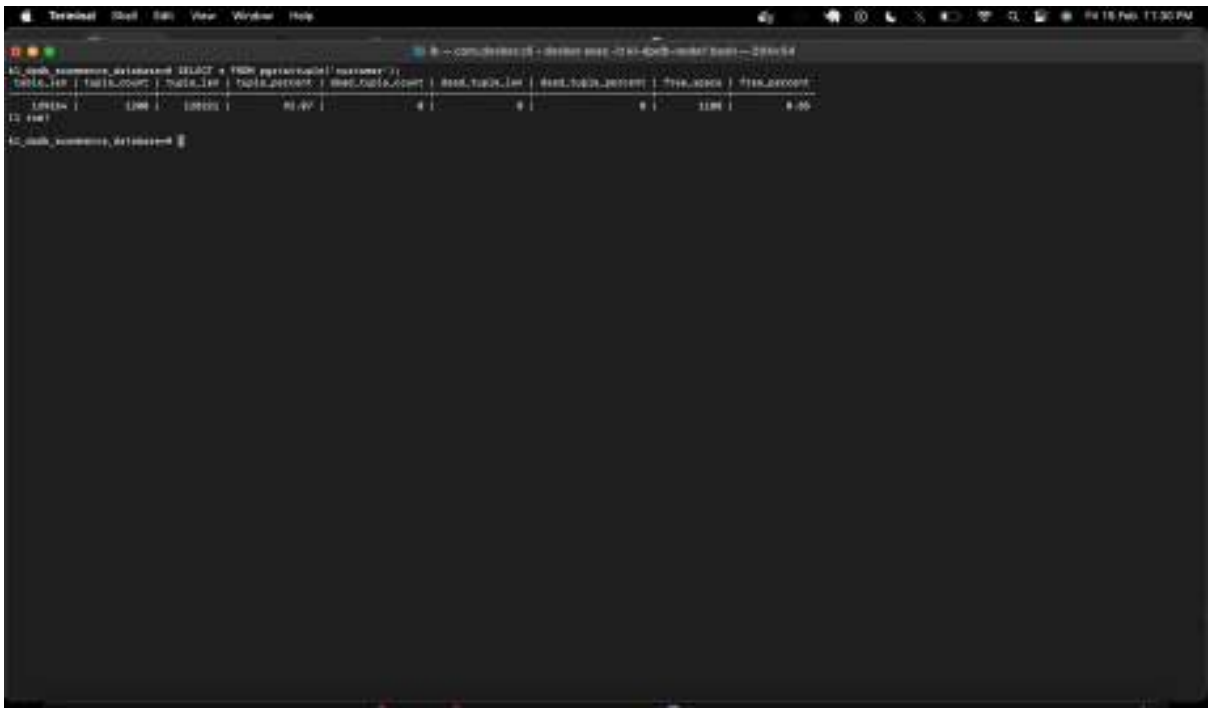


Fig-33: Get the statistical information on the customer table.

Explanation: using pgstattuple extracted the statistics of customer table tuples. The pgstattuple talks about the

- Total length of the table in the disk i.e., table_len is 139264.
- Total records in the table i.e., tuple_count of 1200.
- Total length utilized by all the records in the table i.e., tuple_length of 128221.
- The percentage of space used by the records in the table from total space i.e., tuple_percent of 92.07.
- The dead tuples are the tuple which need to delete. In the customer table, dead_tuple_count - 0, its length is dead_tuple_length – 0, dead_tuple_percent – 0.
- The total amount of free space i.e., free_space is 1188 and its percent i.e., free_percent is 0.85.

As the dead tuples are 0, which define table has no tuples to delete. As the tables has low amount of free space, much of the space is occupied by the tuples. To have the better space utilization fragmentation is performed (Horizontal Fragmentation) over the tuples.

query:

```
select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)
from customer;
```

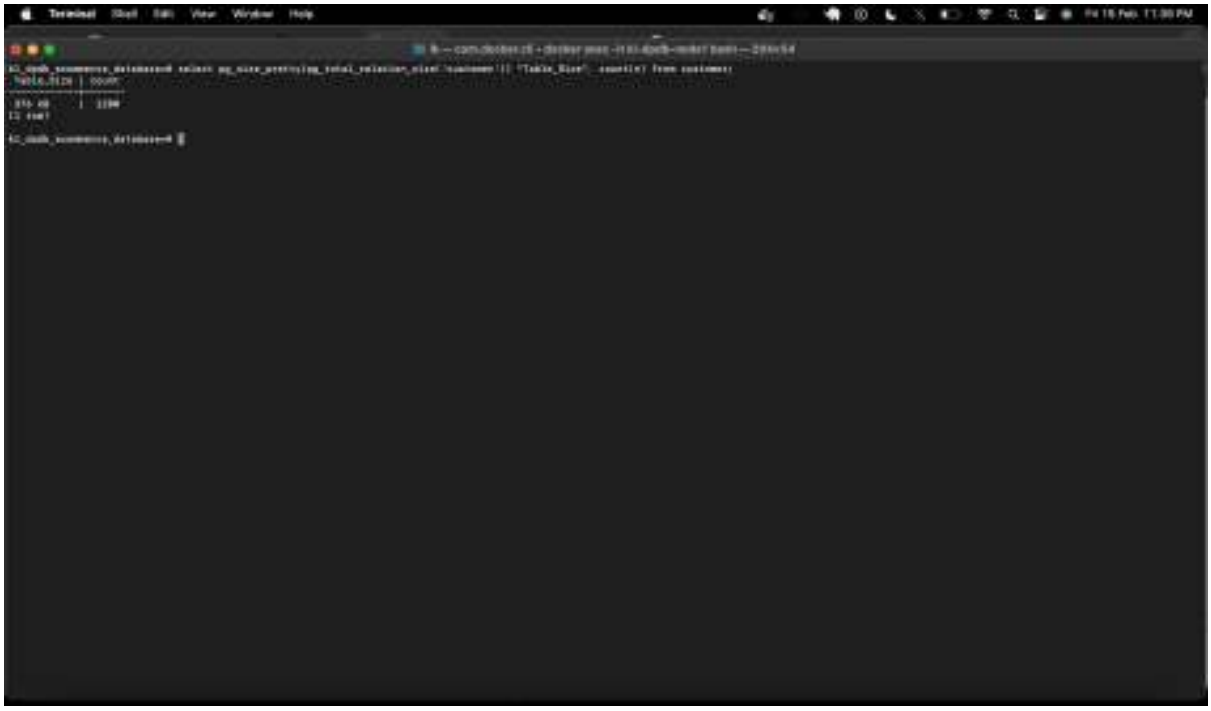


Fig-34: Check the space utilized in customer table using pg_size_pretty.

Explanation: The pg_size_pretty talks about the total space occupied by the customer table. The query also helps to find the records in it as well. So, the space occupied by the table is 376KB.

query:

```
CREATE TABLE customer_1950_1970 AS SELECT * FROM customer WHERE dob BETWEEN '1950-01-01' AND '1970-12-31';
CREATE TABLE customer_1971_1990 AS SELECT * FROM customer WHERE dob BETWEEN '1971-01-01' AND '1990-12-31';
CREATE TABLE customer_1991_2000 AS SELECT * FROM customer WHERE dob BETWEEN '1991-01-01' AND '2000-12-31';
```

Fig-35: Create fragmented tables from customer table.

Explanation: As we discussed above, the table has low amount of free space, performed fragmentation over the tuples. The whole tuples have classified into 3 fragments i.e., **customer_1950_1970, customer_1971_1990, customer_1991_2000 with 498, 469 and 233 records.** The fragmentation is done based on min and max values of the dob.

query:

```
CREATE INDEX idx_dob ON customer (dob);
SELECT * FROM pg_indexes WHERE tablename = 'customer' AND indexname = 'idx_dob'
```

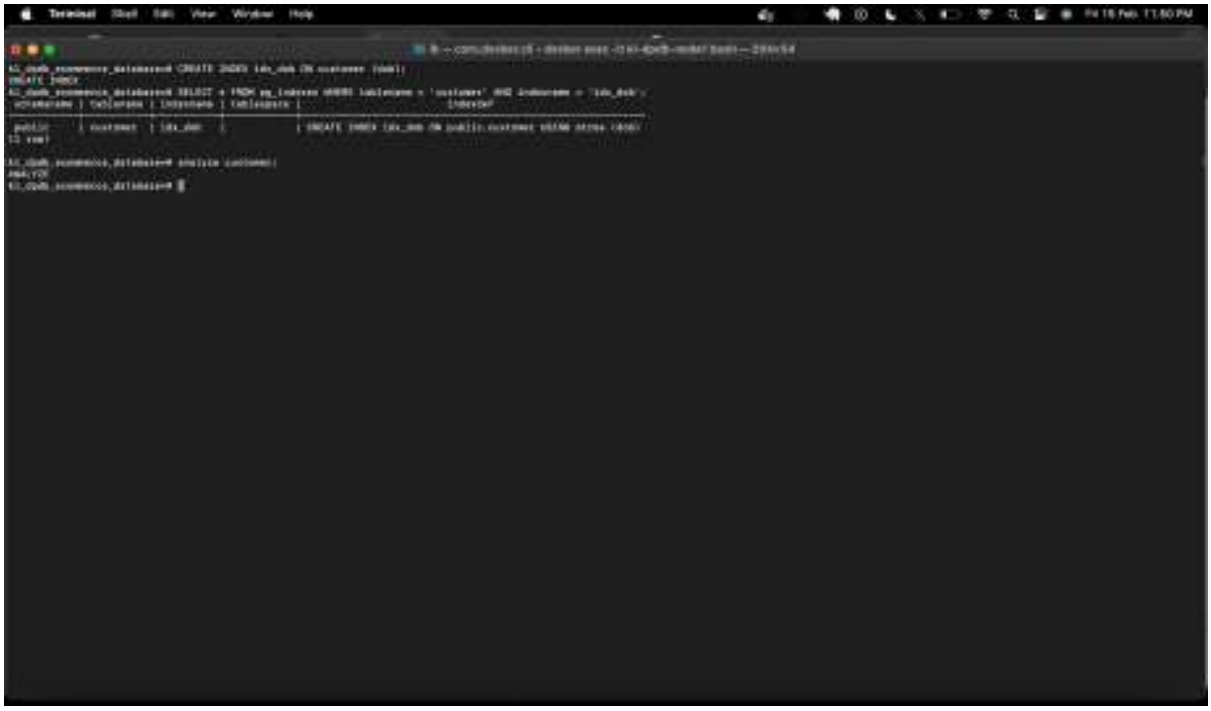


Fig-36: Create the index on the specific column(dob).

Explanation: As the fragmentation is performed using the dob, created the index for the dob column with idx_dob.

query:

```
select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)
from customer;
VACCUM full customer;
select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)
from customer;
```

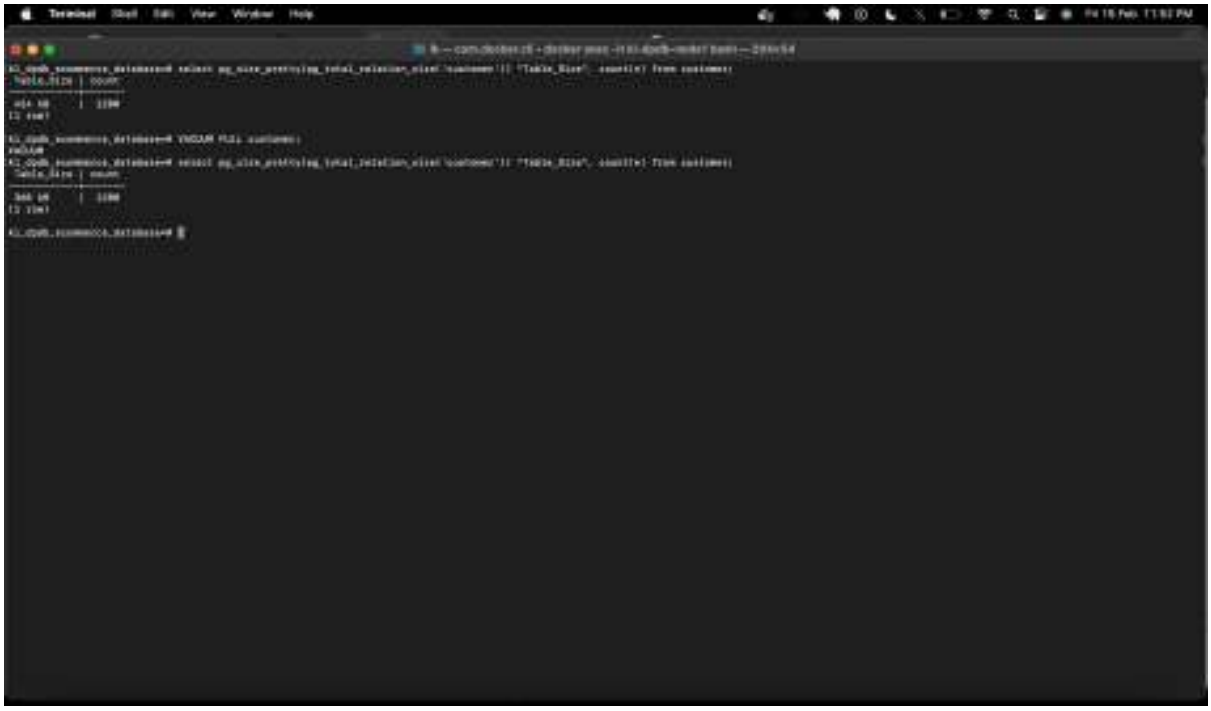


Fig-37: Space after fragmentation.

Explanation: Once the fragmentation is performed and the created the index on the specific column, the size of the table doesn't decrease. It's still increased as we created index on the dob attribute. Ideally VACCUM is used to perform the defragmentation or the optimization. Here also to optimize the size after fragmentation performed VACCUM on the customer table by that we can observe the size of the table is 360KB. The results look better after the fragmentation.

query:

```
EXPLAIN ANALYZE SELECT * FROM customer WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
EXPLAIN ANALYZE SELECT * FROM customer_1950_1970 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
EXPLAIN ANALYZE SELECT * FROM customer_1971_1990 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
EXPLAIN ANALYZE SELECT * FROM customer_1991_2000 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
```

Fig-38: Overall performance

Explanation: The above screenshot talks about the overall performance of the table when we perform querying for multiple times. As we are querying for multiple times performance is getting changed with good efficiency and observed the data is retrieving faster after fragmentation. The space also allocated properly, so that we can insert more data now. The Fragmentation plays the vital role in the space and performance of the data.

Correctness of Fragmentation:

After fragmentation is done, we need to evaluate the correctness of the fragmentation. This can be verified using completeness, reconstruction and disjointness.

Completeness of the fragmentation: The completeness of the data talks about the schema, data distribution and records count of the fragmented tables.

query:

```
Reviewing the fragment tables–
\d customer_1950_1970
\d customer_1971_1990
\d customer_1991_2000
```

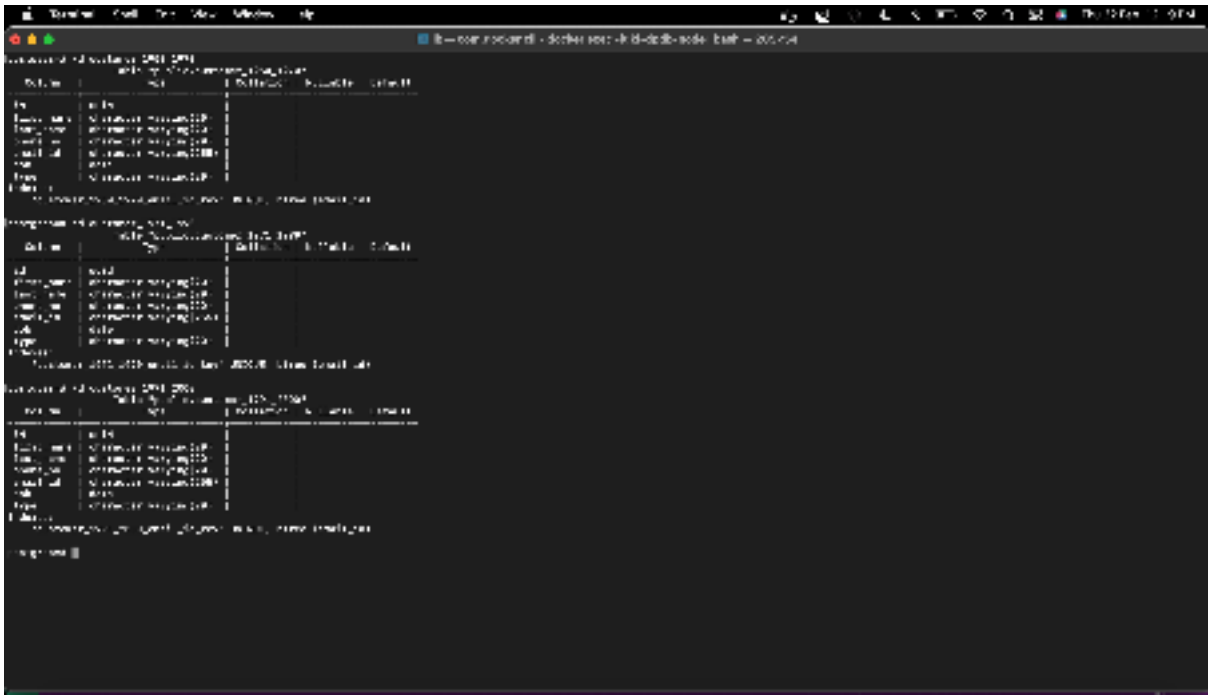


Fig-39: Fragmented tables schema.

Explanation: All the fragmentated tables have the same schema which satisfies the data consistency. In the above screen shot all the fragmented tables have same structure.

query:

```
SELECT MIN(dob), MAX(dob) FROM customer_1950_1970;
SELECT MIN(dob), MAX(dob) FROM customer_1971_1990;
SELECT MIN(dob), MAX(dob) FROM customer_1991_2000;
```

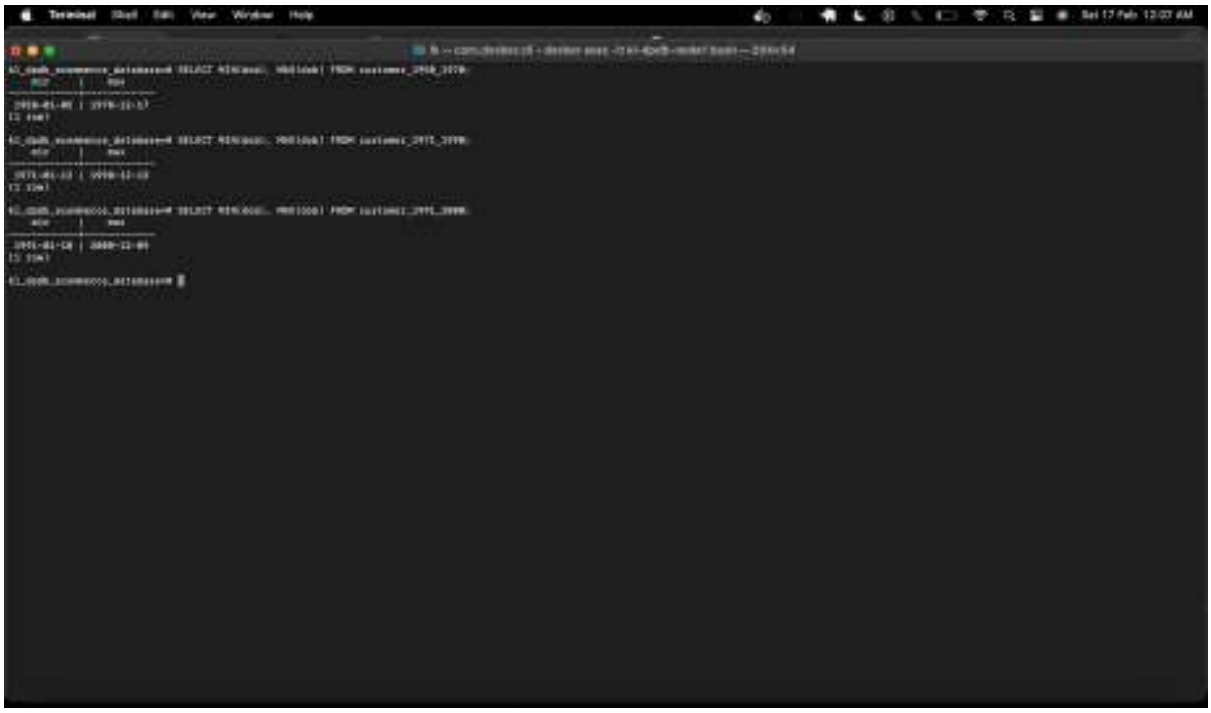


Fig-40: Data Distribution on fragmented tables.

Explanation: The total data in the customer table is distributed in to fragmented tables based on dob. Based on the min and max of value of the fragmented data observed that the data is distributed in the organized way.

query:

```
SELECT count(*) FROM customer;
SELECT count(*) FROM customer_1950_1970;
SELECT count(*) FROM customer_1971_1990;
SELECT count(*) FROM customer_1991_2000;
```

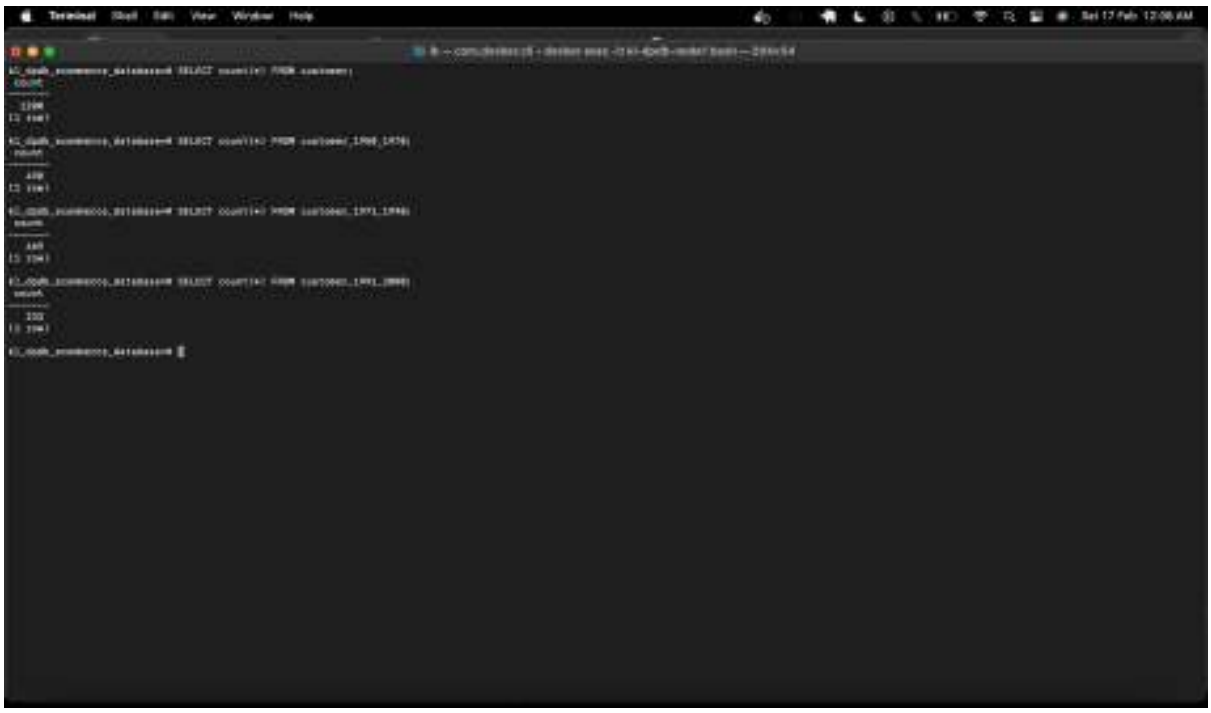


Fig-41: Count of records in fragmented tables.

Explanation: The total count of all the fragmented tables is equal to the count of records in the customer table. The customer table has 1200 records, where as `customer_1950_1970` has 498, `customer_1971_1990` has 469 and `customer_1991_2000` has 233 which satisfies the count of records.

As schema of the table, Data distribution and count of records in the fragmented tables says that the Fragmentation of the customer table achieves the completeness.

Reconstruction: When the original table is fragmented in to multiple tables, the union of all the fragmented table defines the original table.

query:

```
-- Reconstruction:
SELECT SUM(count) AS total_customers_count FROM (
       SELECT count(*) FROM customer_1950_1970
       UNION
       SELECT count(*) FROM customer_1971_1990
       UNION
        SELECT count(*) FROM customer_1991_2000
) AS fragments_count;
```

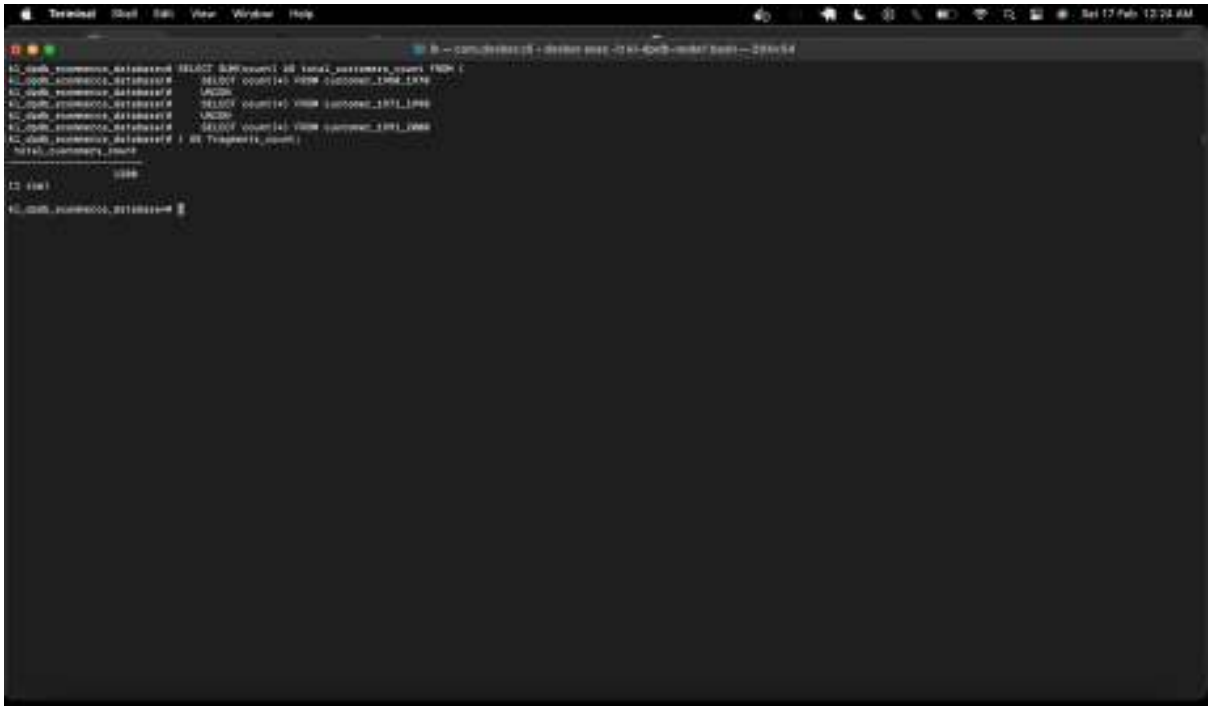


Fig-42: Reconstruction of fragmented tables.

Explanation: The union of all the fragmented tables count is equal to the count of the customer table. By this it achieves the reconstruction.

Disjointness: The disjointness talks about the duplicate of data. There shouldn't be any duplicate data after the fragmentation.

query:

```
-- Disjointness
SELECT id, COUNT(*) AS disjointness_count
FROM (
       SELECT id FROM customer_1950_1970
       UNION ALL
       SELECT id FROM customer_1971_1990
       UNION ALL
       SELECT id FROM customer_1991_2000
) AS all_fragments_customers
GROUP BY id
HAVING COUNT(*) > 1;
```

Fig-43: Duplicate data in the fragmented tables.

Explanation: The duplicate records observed in the fragmented tables is 0. By this it says that the fragmented tables achieve the disjointness as well.

Requirement-4:

To perform the concurrency control strategy, PostgreSQL provides a few techniques such as optimistic, pessimistic and MVCC(multi-version concurrency control). In that the MVCC is the most advanced technique to perform concurrent actions securely in the database.

Implementing the MVCC helps to achieve data consistency, it also helps to perform the reading the data and writing the data into the database smoothly without any conflicts.

Use case 1:
The product should only be assigned to one consumer at a time if two distinct customers are attempting to access it with quantity 1. This indicates that the data and the concurrent actions are consistent.

Available Customers:
query:

```
select * from customer;
available customers-
       52d999b8-de44-489d-8913-7ce999e26c5a
       207397bf-5759-4a51-a44a-3b317971b09a
```

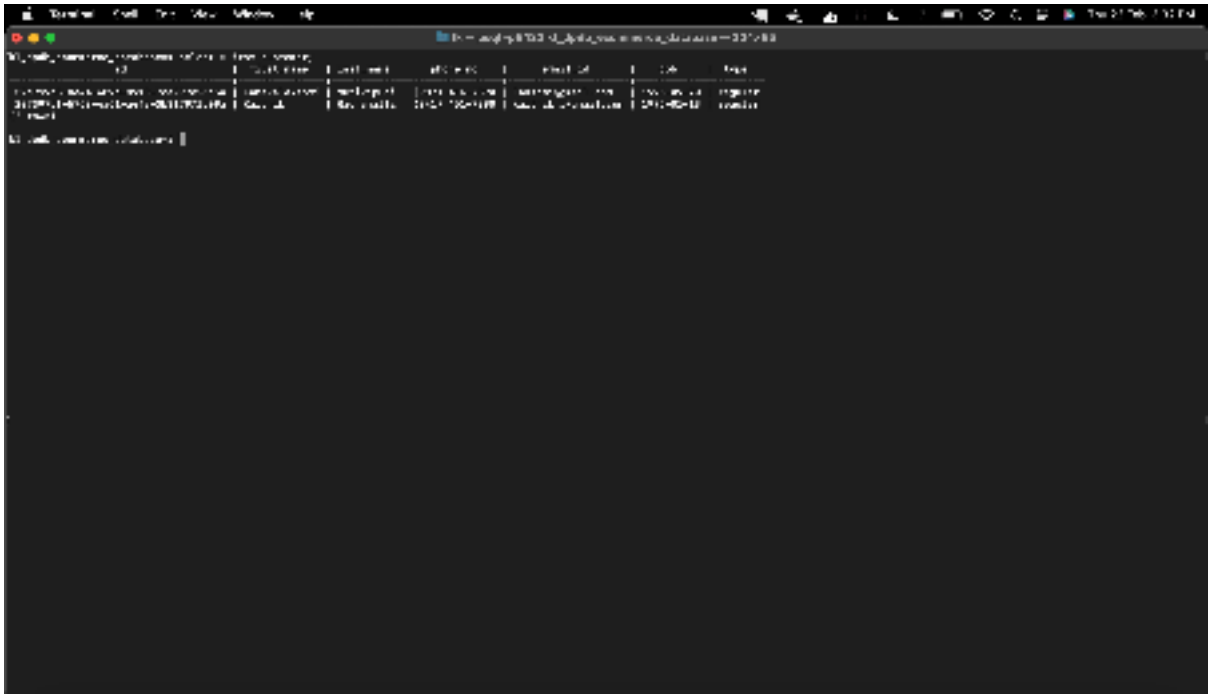


Fig-44: List of customers.

Explanation: Displayed the list of customers in the customer table. There are two users with id `52d999b8-de44-489d-8913-7ce999e26c5a` and `207397bf-5759-4a51-a44a-3b317971b09a`.

Available Products:

query:

```
select * from products;
available products :
       4d205a90-053f-4db1-a22a-37cc18355798
       name - SHOES
        quantity - 1
```

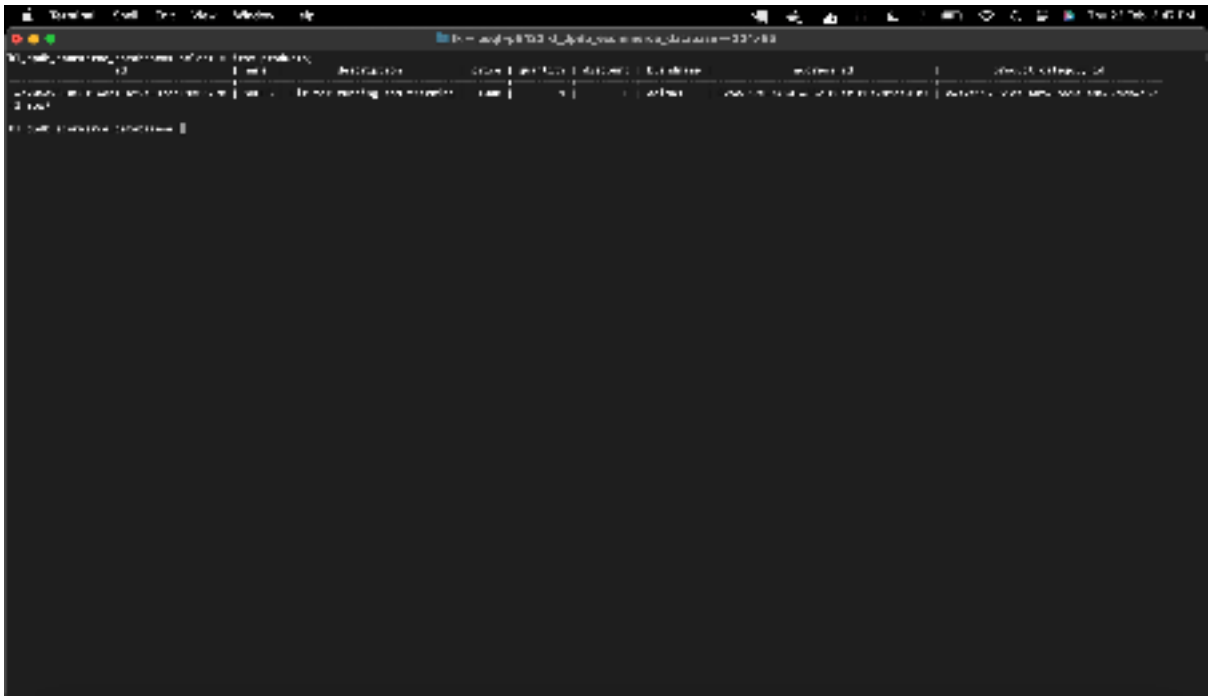


Fig-45: List of products.

Explanation: Displayed the list of available products and its details. Here the product_id `4d205a90-053f-4db1-a22a-37cc18355798` has quantity-1.

PL/pgSQL Function to handle the concurrent actions:

query:

```
CREATE OR REPLACE FUNCTION place_order(
    p_product_id UUID,
    p_customer_id UUID,
    p_quantity INTEGER,
    p_delivery_partner_name VARCHAR(50),
    p_delivery_partner_phone_no VARCHAR(20),
    p_delivery_partner_email VARCHAR(255)
) RETURNS VOID AS $$
DECLARE
    v_order_id UUID;
    v_delivery_partner_id UUID;
    v_product_quantity INTEGER;
BEGIN
    BEGIN
        -- Check if the product is available
        SELECT quantity INTO v_product_quantity FROM products WHERE id =
p_product_id FOR UPDATE;
        -- step-1: Check if the requested quantity is available
        IF v_product_quantity < p_quantity THEN
            RAISE EXCEPTION 'Error in place_order: Product not available. Please
try again later.';
        END IF;
        -- step 2: Insert into orders table
        INSERT INTO orders (status, order_date, address_id, customer_id)
        VALUES ('Processing', CURRENT_DATE, (SELECT address_id FROM
customer_address WHERE customer_id = p_customer_id AND default_address = true),
p_customer_id)
        RETURNING id INTO v_order_id;
        -- Step 3: Update products table
        UPDATE products SET quantity = quantity - p_quantity WHERE id =
p_product_id;
        -- Step 4: Insert into transaction_summary table
        INSERT INTO transaction_summary (total_amount_paid, payment_type,
date_of_payment, order_id)
        VALUES ((SELECT price * p_quantity FROM products WHERE id = p_product_id),
'Credit Card', CURRENT_DATE, v_order_id);
        -- Step 5: Insert into orders_products table
        INSERT INTO orders_products (order_id, product_id, quantity) VALUES
(v_order_id, p_product_id, p_quantity);
        -- Step 6: Insert into delivery_partner table
        INSERT INTO delivery_partner (name, phone_no, email, order_id)
        VALUES (p_delivery_partner_name, p_delivery_partner_phone_no,
p_delivery_partner_email, v_order_id)
        RETURNING id INTO v_delivery_partner_id;
        -- Step 7: Insert into customer_delivery_partner table
        INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
VALUES (p_customer_id, v_delivery_partner_id);
```

```
    EXCEPTION
        WHEN OTHERS THEN
            -- Rollback the transaction in case of an exception
            RAISE EXCEPTION 'Error in place_order: %', SQLERRM;
    END;
END;
$$ LANGUAGE plpgsql;
```

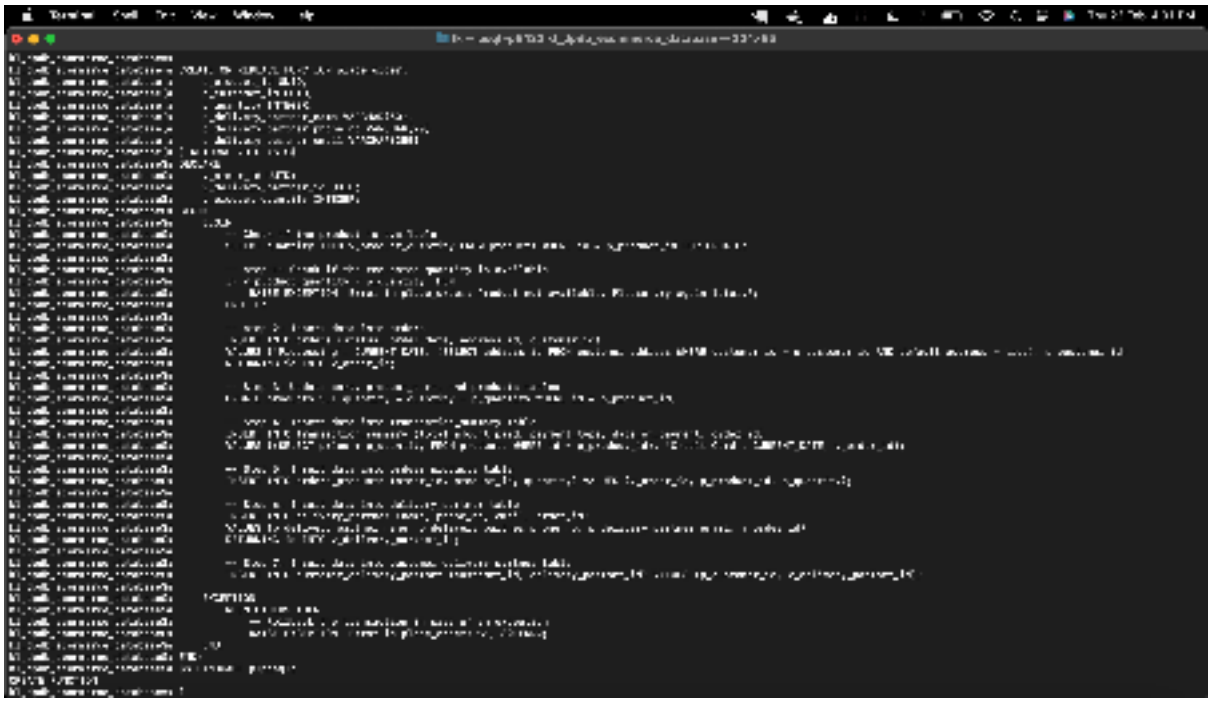


Fig-46: Function created to place order.

Explanation: The place_order function is the PL/pgSQL code. Which is used to place the order by the customer in ecommerce site. This handles the all-database requirements such as data consistency, concurrency control, ROLL Back to have the security etc.

The function reads the input params:
p_product_id UUID: The product id that is ordered by the customer.
p_customer_id UUID: The customer id who is placing the order.
p_quantity INTEGER: The quantity of the product that is ordered.
p_delivery_partner_name VARCHAR(50): The name of the delivery partner.
p_delivery_partner_phone_no VARCHAR(20): The phone_no of the delivery partner.
p_delivery_partner_email VARCHAR(255): The email of the delivery partner.

When an order is placed by the customer, Initially it checks for the product quantity available or not, If the product quantity is zero it returns the message saying that “Product not available. Please try again later.” Else, it places the order and updates all the required transactions with dependency tables. (orders, transaction_summary, count in the products table, assigning the delivery partner)

The function also includes the error handling, which raises the exception and roll backs the data in case of any issue during the transaction.

The above PL/pgSQL internally handles the MVCC, The MVCC creates the snapshot while performing the transactions(one transaction does not affect the other transaction)and provides the locking mechanism to handle the concurrent transactions, dead lock situations, and inconsistent transactions. It majorly follow " The WRITER never blocks the READER" and "THE READER never blocks the WRITER".

Python Script to perform concurrent orders by the customer:
query:

```
import psycopg2
from concurrent.futures import ThreadPoolExecutor

# connection details
host = "localhost"
port = 5432
database = "kl_dpdb_ecommerce_database"
user = "postgres"
password = "spaceman1236"

def place_order(product_id, customer_id, quantity, delivery_partner_name, phone_no,
email):
    try:
        place_order_connection = psycopg2.connect(
            host=host,
            port=port,
            database=database,
            user=user,
            password=password
        )
        place_order_cursor = place_order_connection.cursor()
        place_order_cursor.execute(
            """
            SELECT place_order(
                %s, %s, %s, %s, %s, %s
            );
            """,
            (product_id, customer_id, quantity, delivery_partner_name, phone_no,
email)
        )
        place_order_connection.commit()
        print(f"order Placed for {customer_id}!")

    except psycopg2.Error as e:
        error_message = str(e)
        if "product not available" in error_message.lower():
            print(f"Product not available for {customer_id}! please try again")
        else:
            print(f"Error connecting to PostgreSQL: {e}")
```

```
    finally:
        if place_order_cursor:
            place_order_cursor.close()
        if place_order_connection:
            place_order_connection.close()

if __name__ == "__main__":
    # customer-1
    customer_1_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '52d999b8-de44-
489d-8913-7ce999e26c5a', 1, 'robin', '(817) 777-4089', 'robin@example.com')
    # customer-2
    customer_2_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '207397bf-5759-
4a51-a44a-3b317971b09a', 1, 'josey', '(978) 717-4389', 'josey1@example.com')

    with ThreadPoolExecutor(max_workers=2) as executor:
        # Trigger the function concurrently with different user inputs
        executor.submit(place_order, *customer_1_details)
        executor.submit(place_order, *customer_2_details)
```

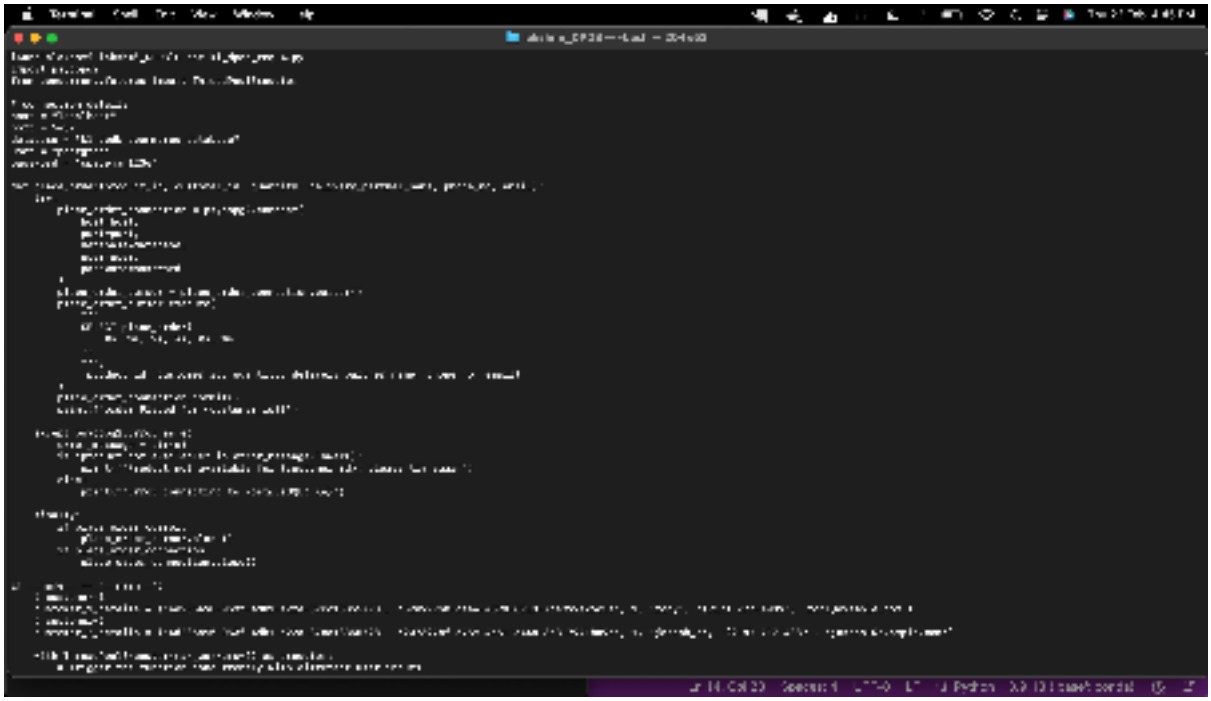


Fig-47: Script to execute concurrent orders.

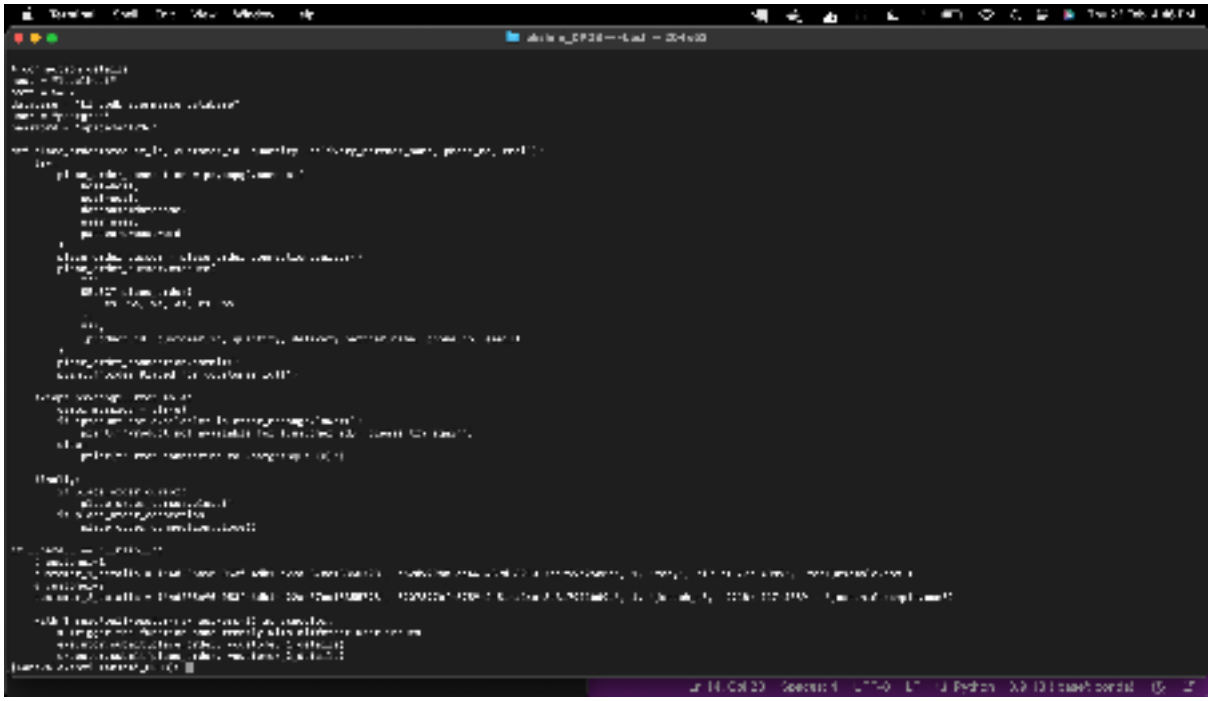


Fig-48: Continuation of the Script to execute concurrent orders.

Explanation: The python script uses the psycopg2 library to interact with the postgresql database. The script has the function ‘place_order’ to place order by the concurrent customers using the **ThreadPoolExecutor** from the **concurrent.futurez**

Initially the connection details are defined with host, port_no, database, user, and the password. The function reads the input parameters product_id, customer_id quantity, delivery_partner details (name, phone_no, email) and establishes the connection using connection details provided. Executes the postgresql query to call the place_order function in the postgtres database. Handles the exceptions, specifically related to the product relevant. Once everything is done it closes the connection.

The Main function handles the multiple customer details and uses the ThreadPoolExecutor to perform the transactions concurrently by the customers while placing the order.

query:

```
python kl_dpdb_req-4.py
```

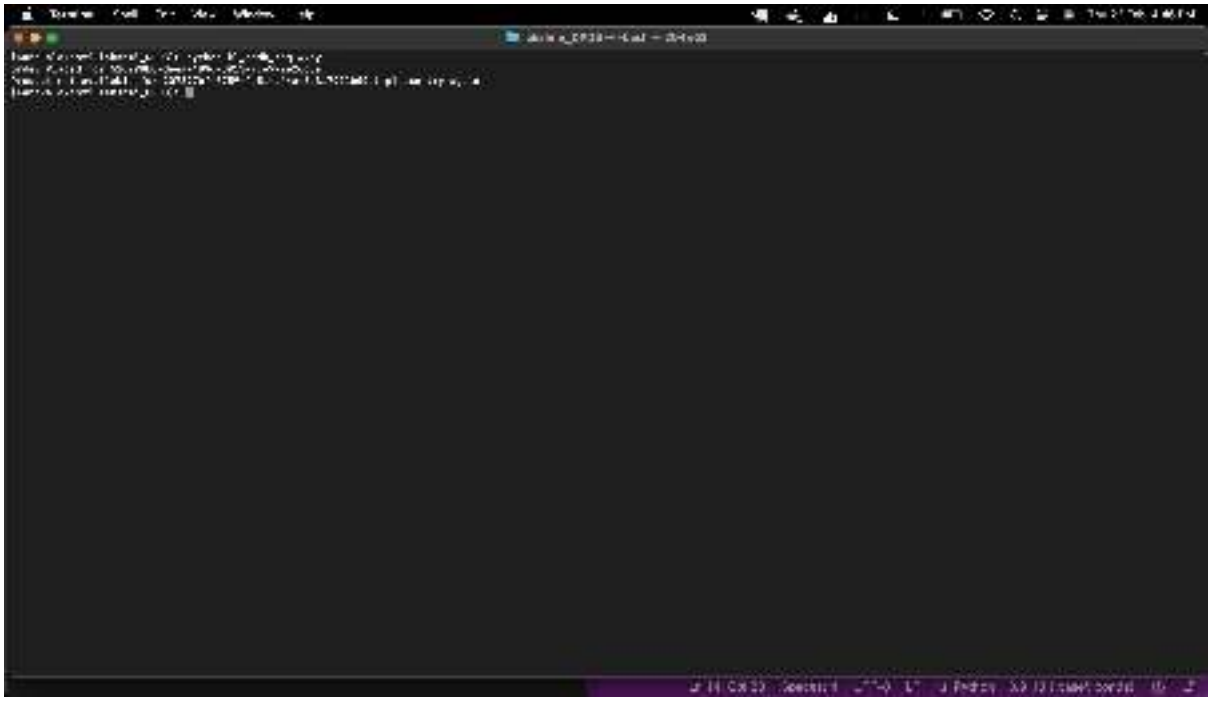


Fig-49: Run the python script.

Explanation: The python script is executed. Observed that the order got placed by only one customer(52d999b8-de44-489d-8913-7ce999e26c5a) as there is only product. For other user the order got rejected and displayed “Product not available for '207397bf-5759-4a51-a44a-3b317971b09a! please try again.” This ensures is data consistent in the database and handles the concurrent orders.

query:

```
select * from products;
```

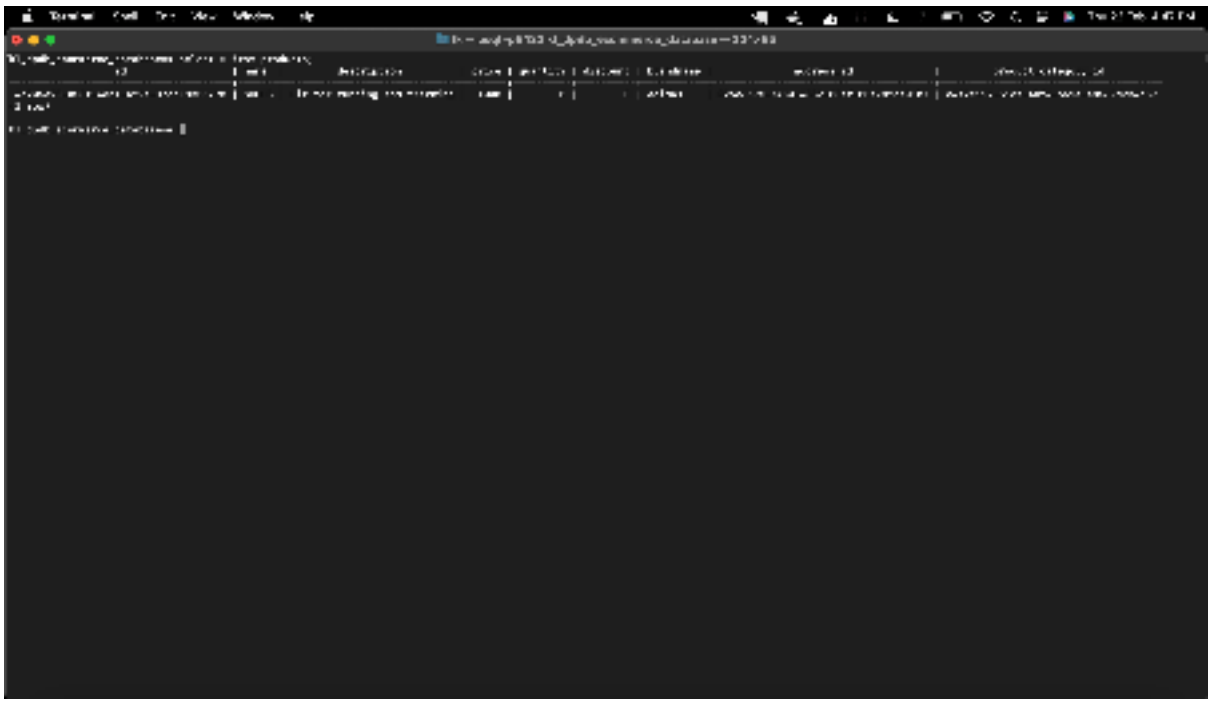


Fig-50: Quantity is 0 in products table.

Explanation: once after order is placed, observed that the quantity is ‘0’ for respective product. This is that data is maintaining the consistency.

query:

```
select * from orders;
```

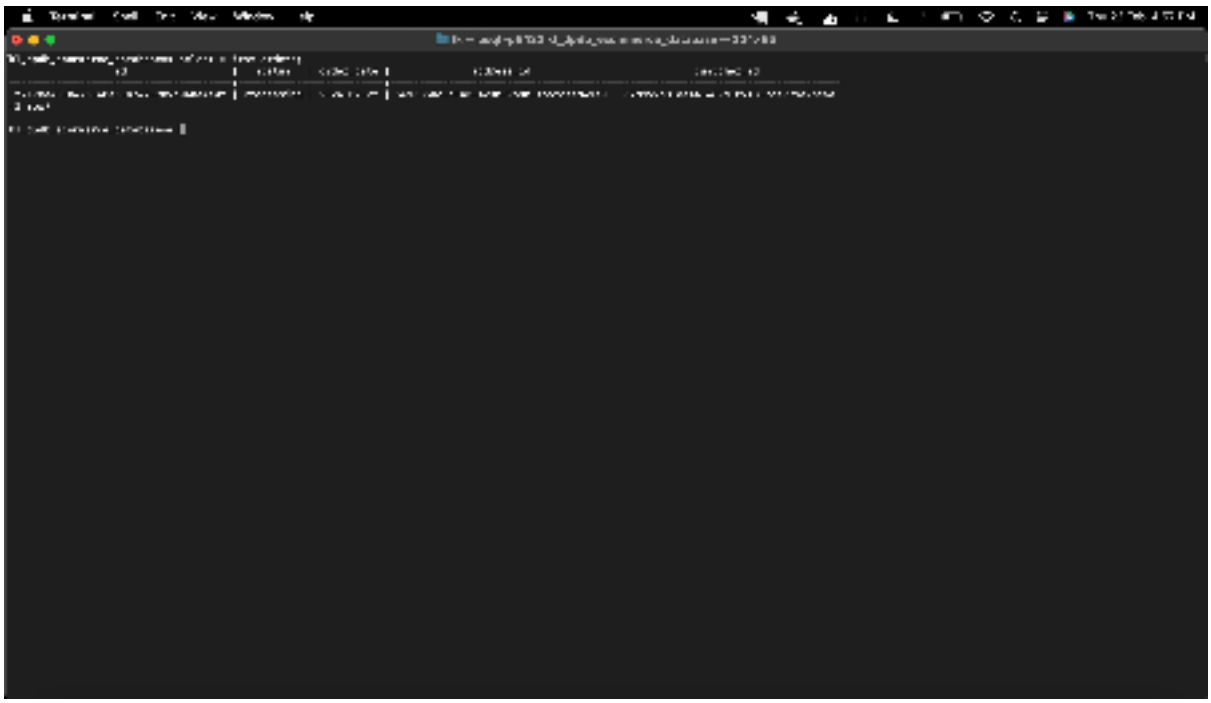


Fig-51: Data in orders table.

Explanation: order placed for customer- 52d999b8-de44-489d-8913-7ce999e26c5a and displayed the respective details.

query:

```
select * from transaction_summary;
select * from orders_products;
select * from delivery_partner;
select * from customer_delivery_partner;
```

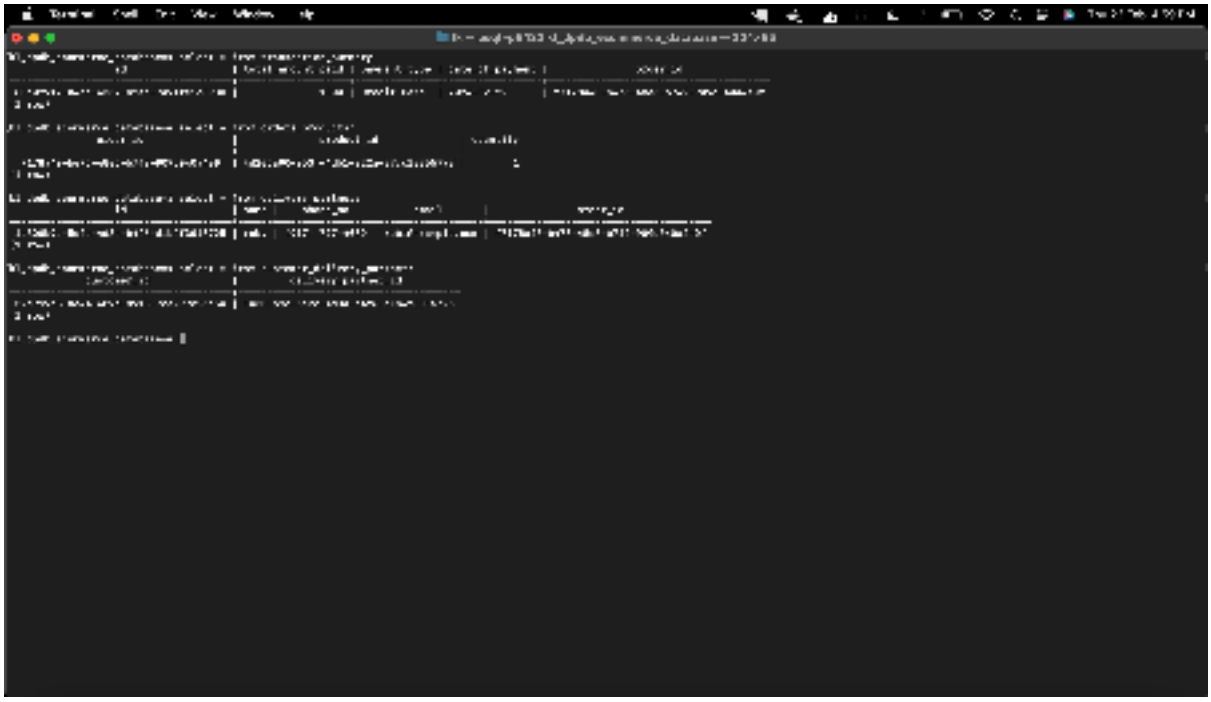


Fig-52: Updated data in all the required tables.

Explanation: updated data in all the required tables (reansaction_summary, orders_products, delivery_partner, customer_delivery_partner). Which says data is consistent.

Use case 2:
when there are two product quantities, and two distinct customers are attempting to place order for the same product. Both clients ought to be able to place the order in this instance.

query:

```
select * from products;
UPDATE products SET quantity = 2 WHERE id='4d205a90-053f-4db1-a22a-37cc18355798';
select * from products;
```

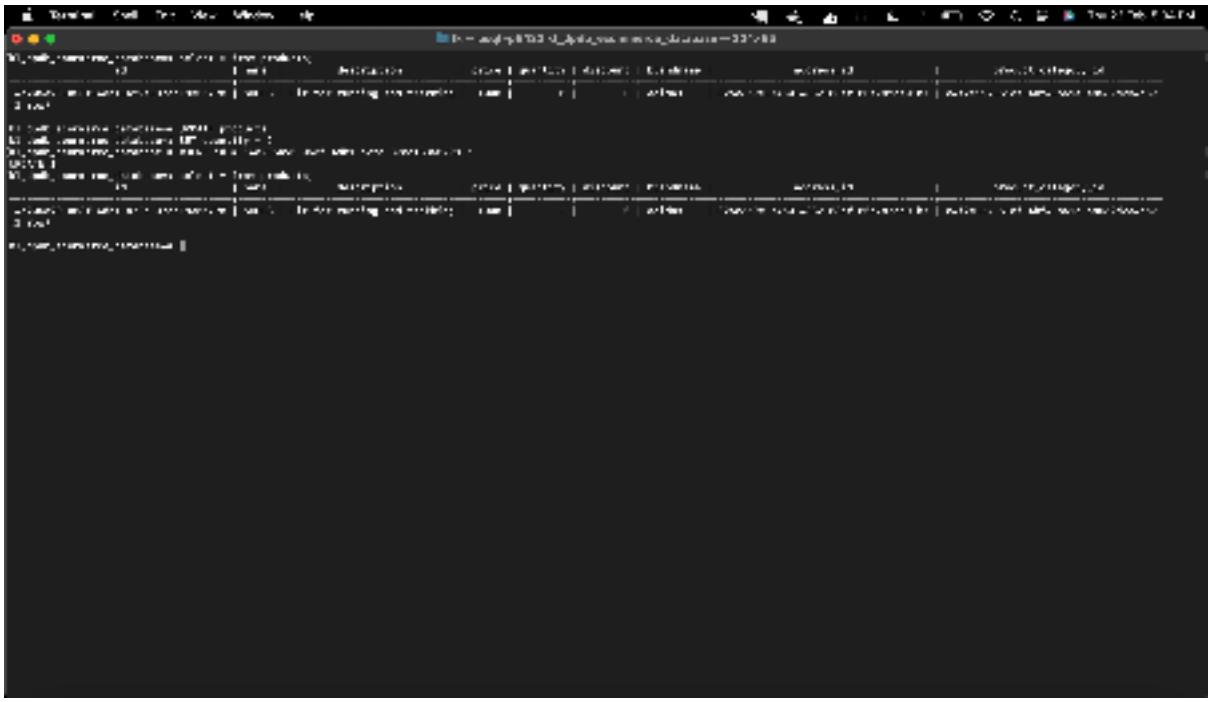


Fig-53: Update the product quantity to 2.

query:

```
if __name__ == "__main__":
    # customer-1
    customer_1_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '52d999b8-de44-
489d-8913-7ce999e26c5a', 1, 'robin', '(817) 777-4089', 'robin@example.com')
    # customer-2
    customer_2_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '207397bf-5759-
4a51-a44a-3b317971b09a', 1, 'josey', '(978) 717-4389', 'josey1@example.com')
```

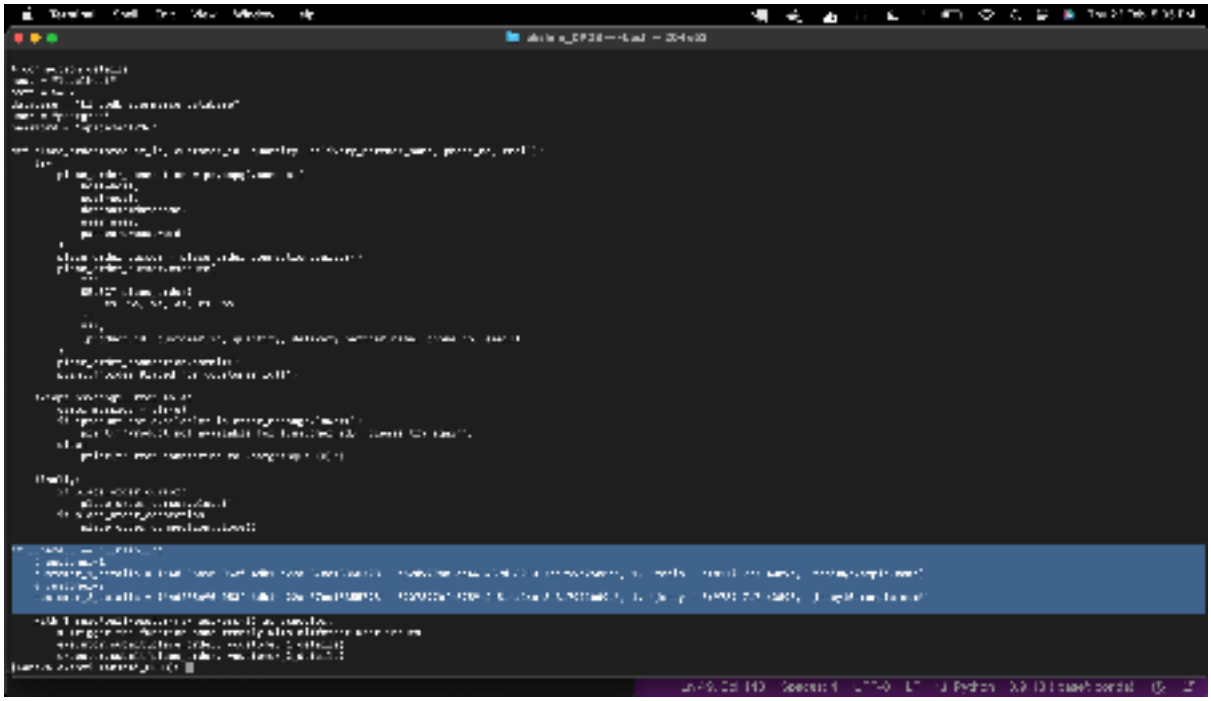


Fig-54: Update the inputs as per the use case-2.

query:

```
python kl_dpdb_req-4.py
```

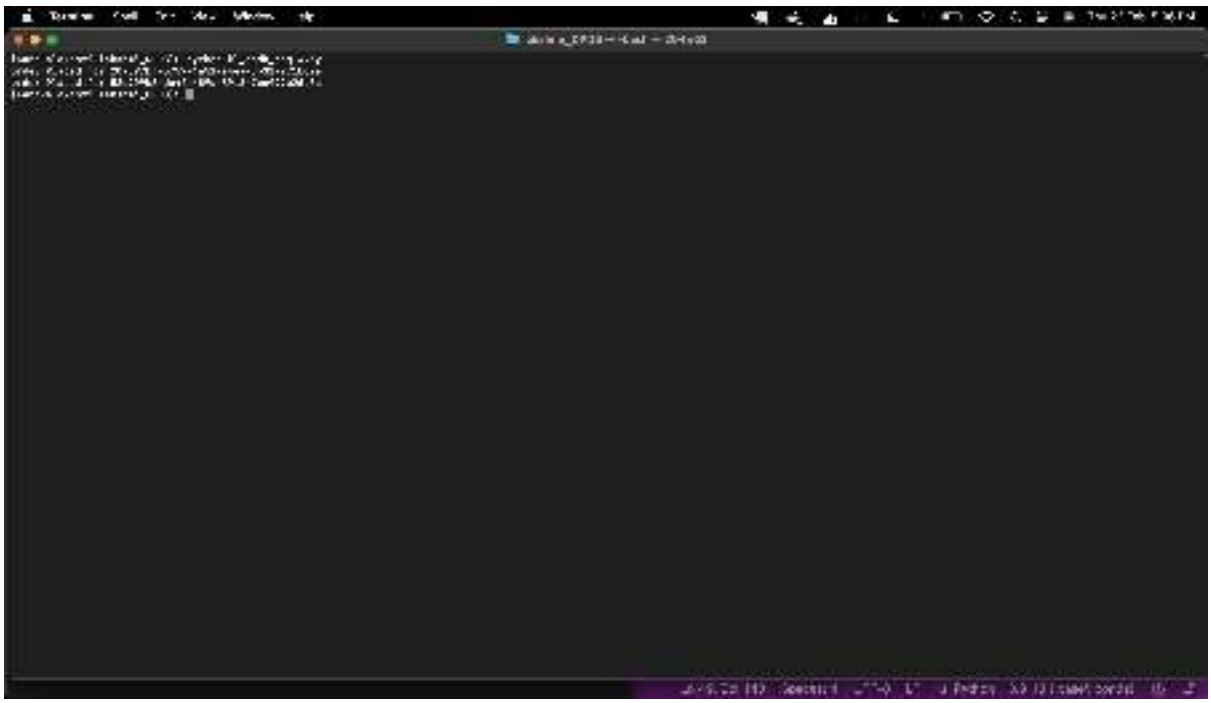


Fig-55: order successfully placed by two customers.

Explanation: Two different users wish to place an order with quantity one, and there are two products available in the products table. So, the order was successfully placed for the two customers.

query:

```
select * from products;
select * from orders;
```

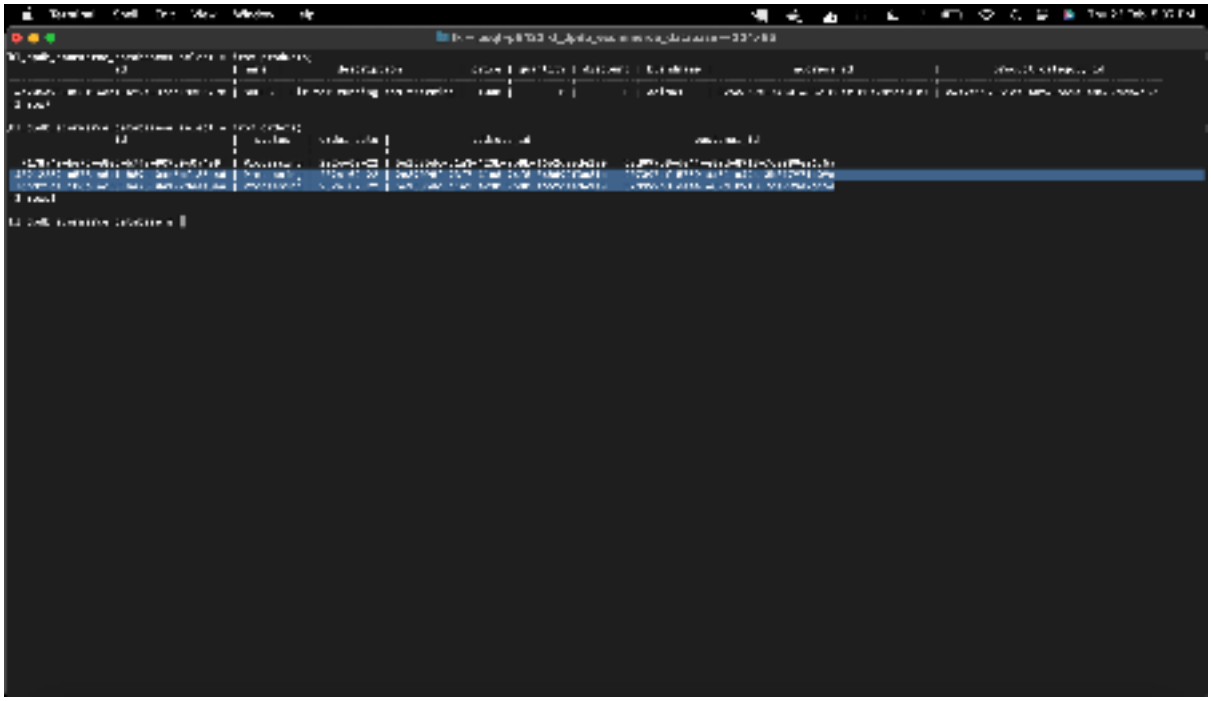


Fig-56: Two orders placed by two different customers.

Explanation: The order details got successfully inserted into the orders table with status as processing, order_date with current date, address associated with the customer.

query:

```
select * from transaction_summary;
```

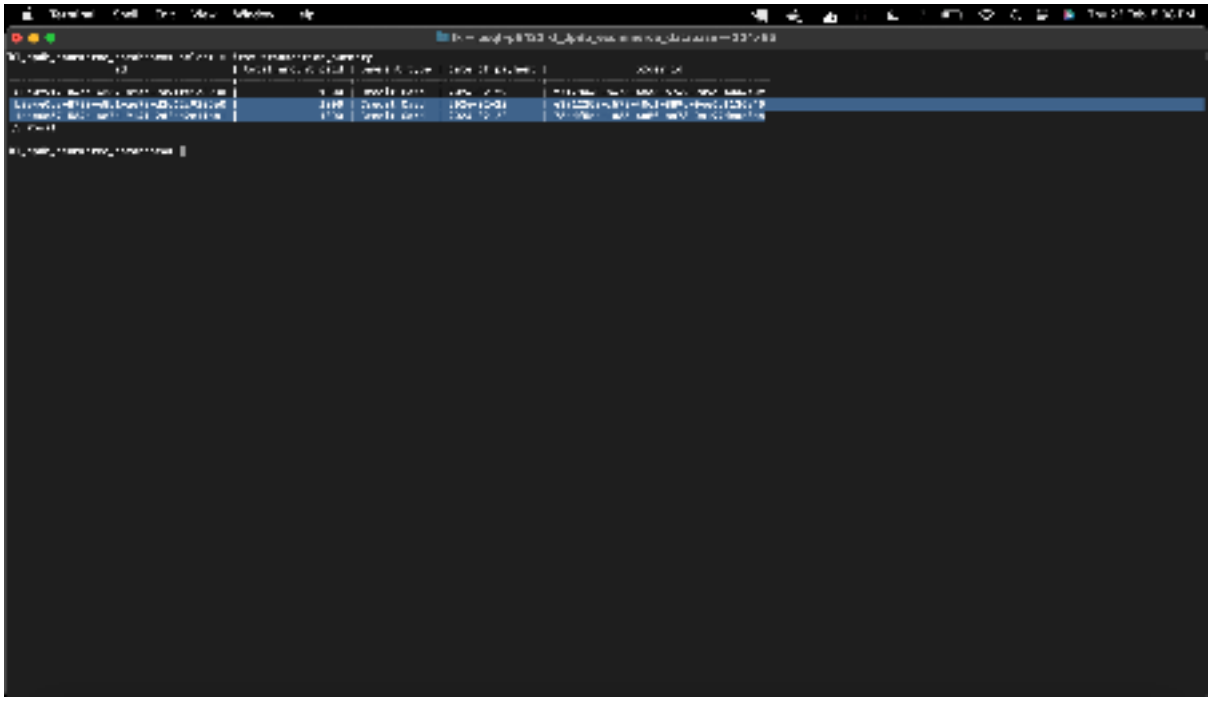


Fig-57: Updated transaction summary for two orders.

query:

```
select * from orders_products;
select * from delivery_partner;
select * from customer_delivery_partner;
```

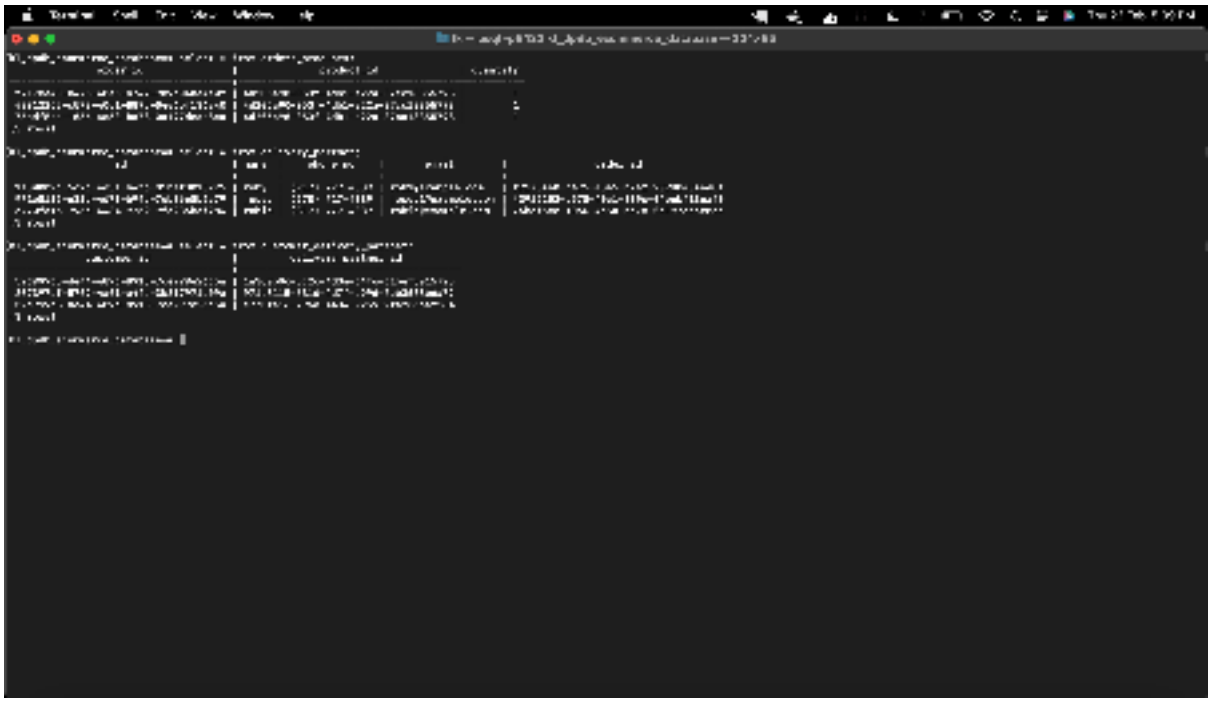


Fig-58: Data updated in all the dependency tables.

use case 3: when two clients attempt to place an order repeatedly. As an example, let's say that a product has quantity "10," and customers 1 and customer 2 each placed three orders.

query:

```
UPDATE products SET quantity = 10 WHERE id='4d205a90-053f-4db1-a22a-37cc18355798';
```

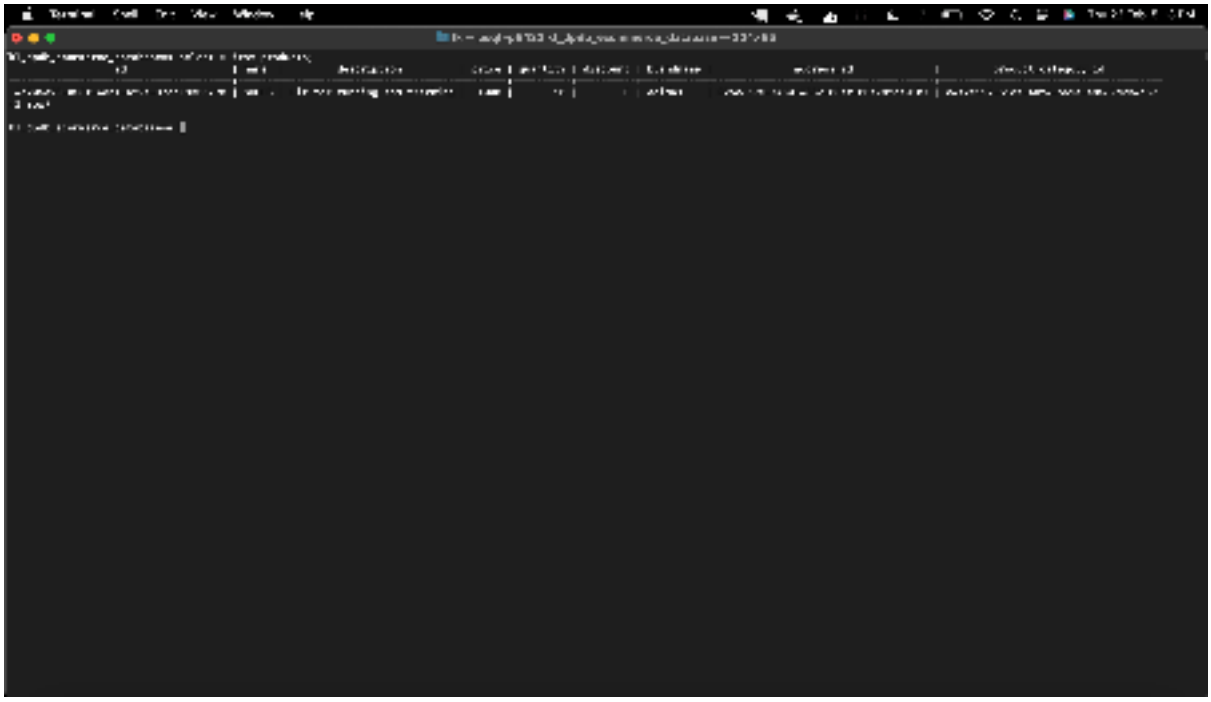


Fig-59: Updated product quantity to 10

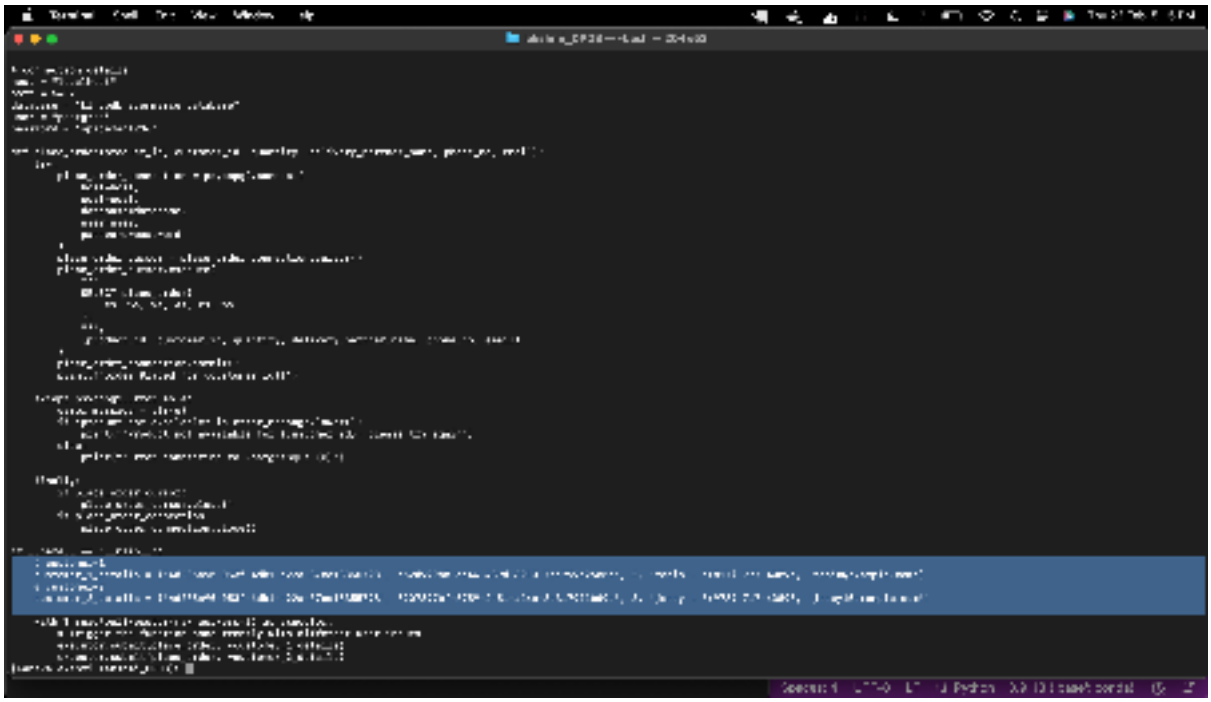


Fig-60: Updated the script as per the use case 3.

query:

```
python kl_dpdb_req-4.py
```

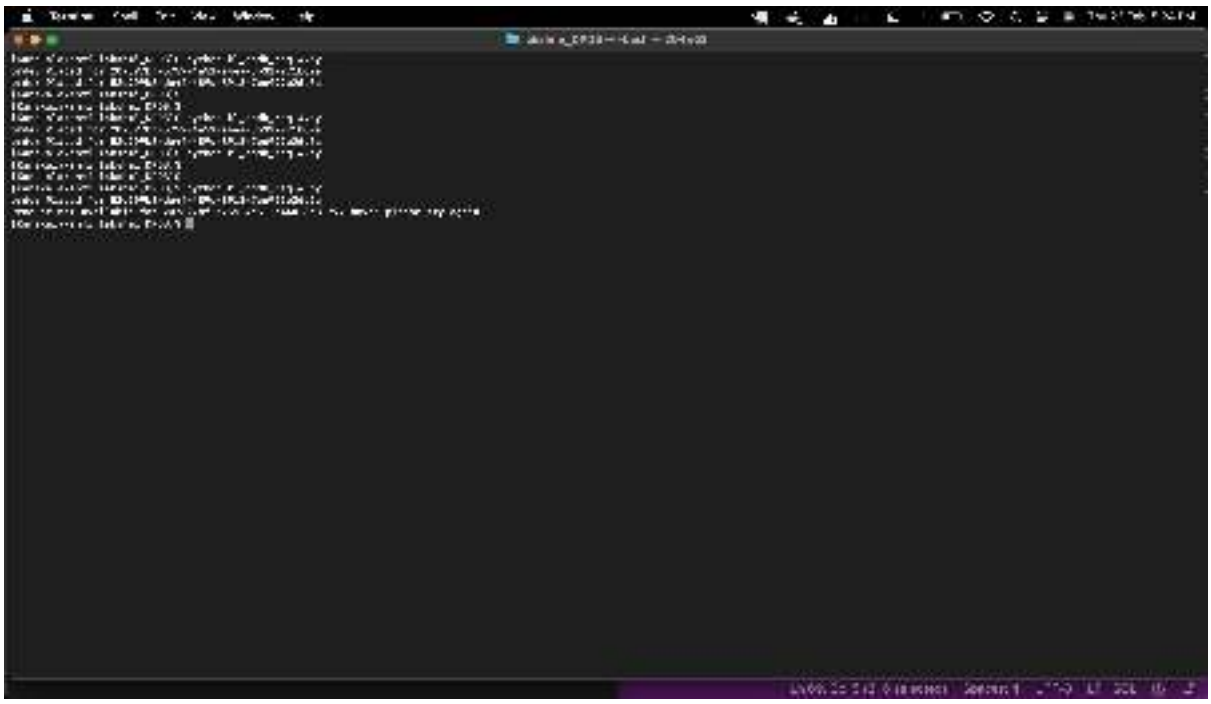


Fig-61: Order placed multiple times with two customers.

query:

```
select * from products;
```

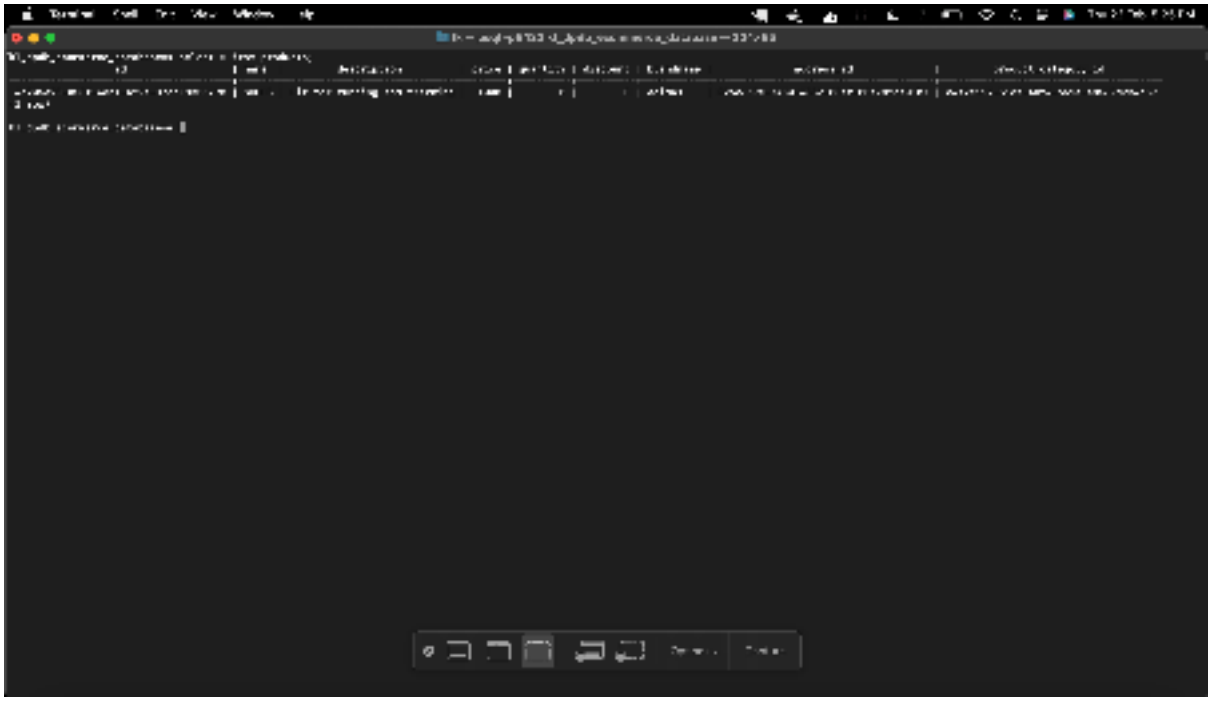


Fig-62: Quantity of the product is 0.

Requirement-5

In the database system they will be multiple users who access to the database. But all the users will not have all privileges to all tables in the database. Users have the privileges based on their responsibilities.

query:

```
-- Create Users:
-- Create superuser
CREATE USER superuser WITH PASSWORD 'password_superuser' SUPERUSER;
-- Create customers user
CREATE USER customer WITH PASSWORD 'password_customer' NOCREATEDB NOCREATEROLE;
-- Create administrators user
CREATE USER administrators WITH PASSWORD 'password_administrators' CREATEDB
CREATEROLE;
-- list of users
SELECT usename FROM pg_user;
```

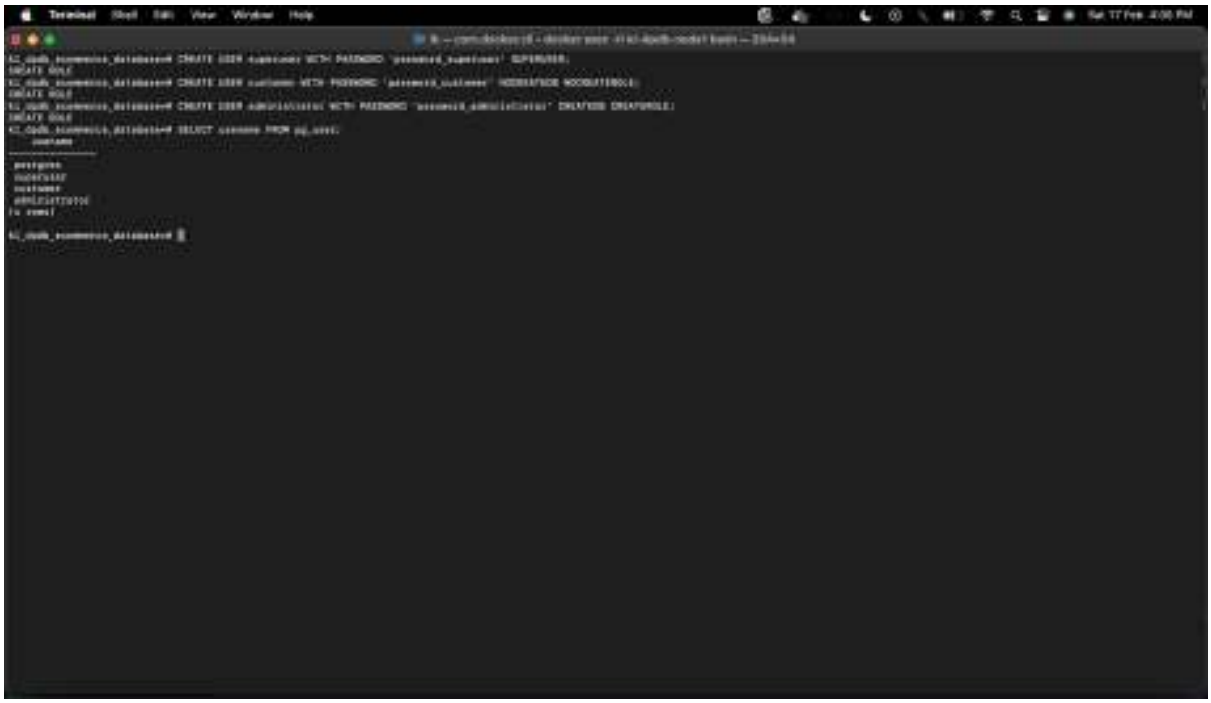


Fig-63: Create users to give access control.

Explanation: Created three different users – superuser, customer and administrator. Once after users are created, displayed the list of users.

Grant access to superuser:

query:

```
-- Grant Access to Users:
-- Super User
GRANT ALL PRIVILEGES ON DATABASE kl_dpdb_ecommerce_database TO SUPERUSER;
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO superuser;
--Cross check the previliges
SELECT table_name, grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_catalog = 'kl_dpdb_ecommerce_database' AND grantee = 'superuser';
```

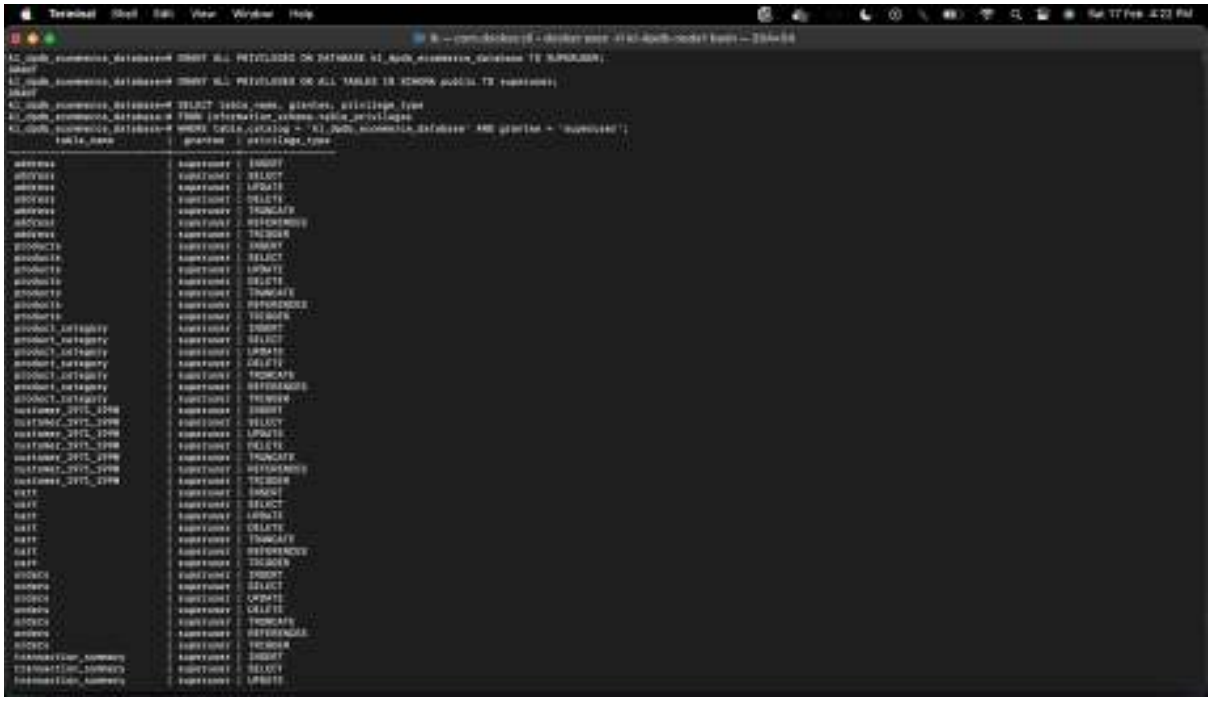


Fig-64: Grant access to superuser.

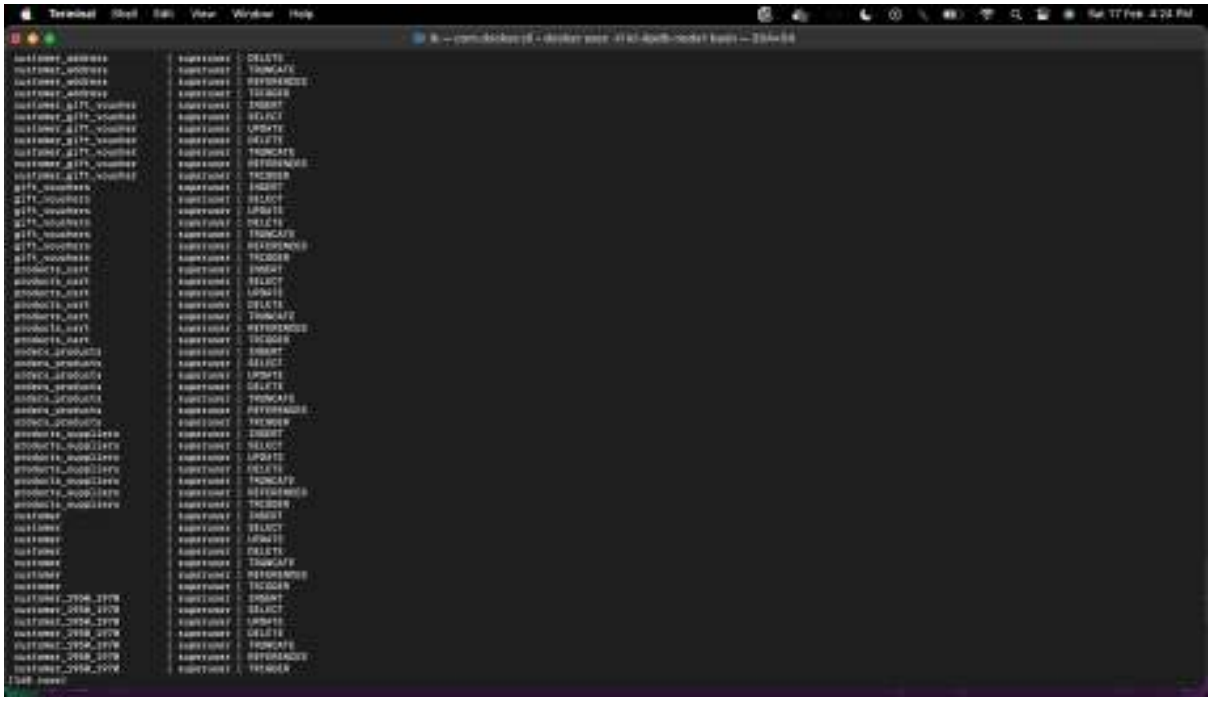


Fig-65: Display the privileges of super user.

Explanation: Ideally super user has access on all tables in the database system with all privileges so, provided all privileges to super user and displayed the complete data.

Grant access to administrator:

query:

```
CREATE OR REPLACE FUNCTION grant_privileges_to_administrator() RETURNS VOID AS $$
BEGIN
-- Granting privileges on the 'Products' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Products TO administrator';
-- Granting privileges on the 'Customer' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Customer TO administrator';
-- Granting privileges on the 'cart' table
EXECUTE 'GRANT SELECT, UPDATE, DELETE ON TABLE cart TO administrator';
-- Granting privileges on the 'orders' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE orders TO administrator';
-- Granting privileges on the 'Transaction summary' table
EXECUTE 'GRANT SELECT ON TABLE "transaction_summary" TO administrator';
-- Granting privileges on the 'supplier' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE supplier TO administrator';
-- Granting privileges on the 'Address' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Address TO administrator';
-- Granting privileges on the 'product_category' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE product_category TO
administrator';
-- Granting privileges on the 'Reviews' table
EXECUTE 'GRANT SELECT, DELETE ON TABLE Reviews TO administrator';
-- Granting privileges on the 'gift_vouchers' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE gift_vouchers TO
administrator';
-- Granting privileges on the 'delivery_partner' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE delivery_partner TO
administrator';
  END;
  $$ LANGUAGE plpgsql;

-- Execute the function to grant privileges
SELECT grant_privileges_to_administrator();

SELECT table_name, grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_catalog = 'kl_dpdb_ecommerce_database' AND grantee = 'administrator';
```

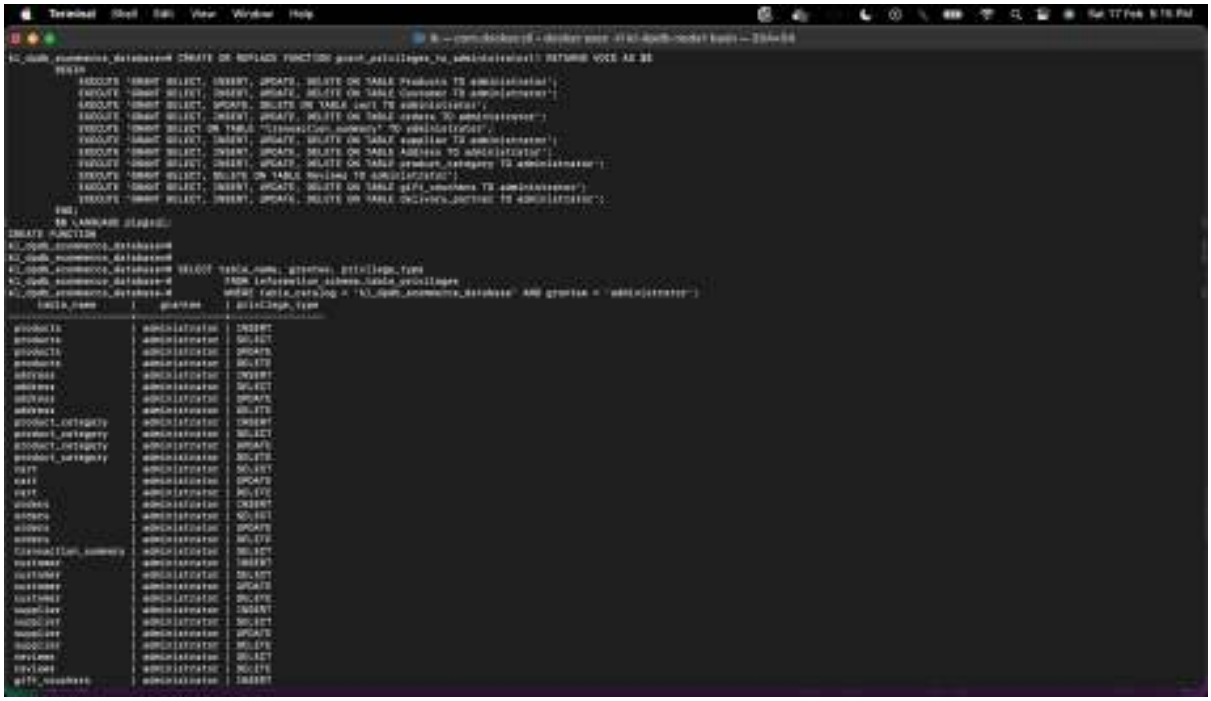


Fig-66: Grant access to administrator.

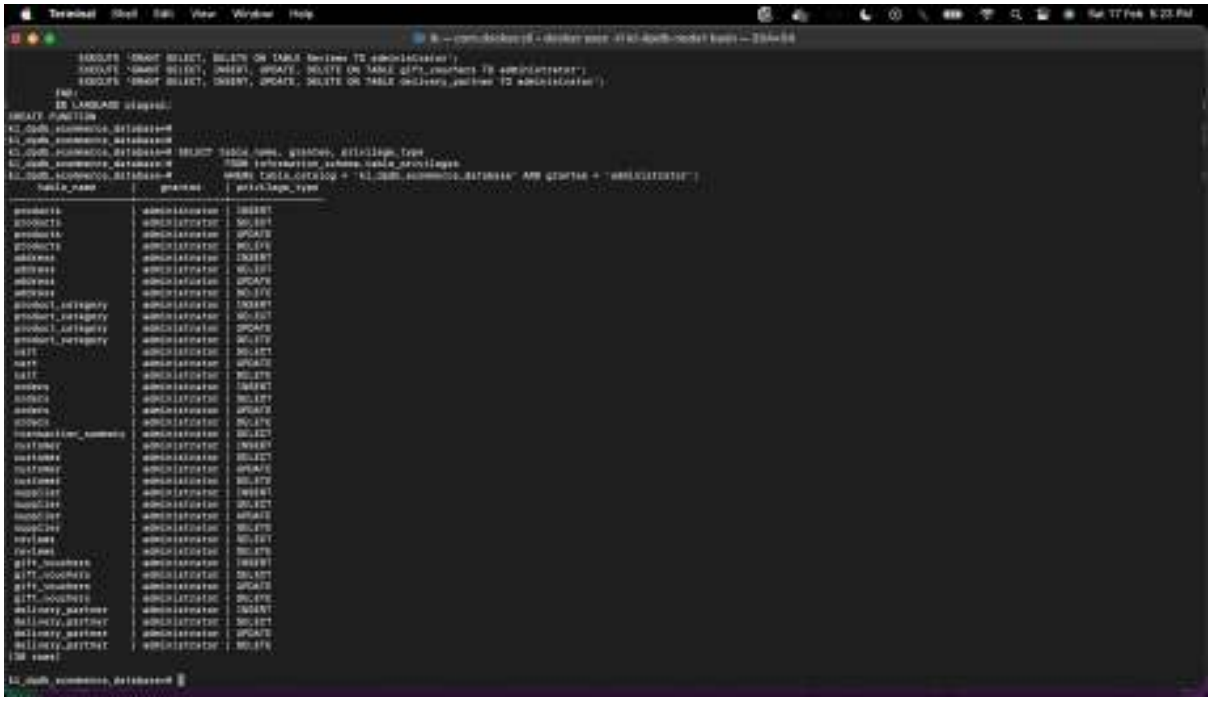


Fig-67: list of complete privileges for administrator.

Explanation: Granted privileges to administrator on the respective tables. Displayed the list of privileges given to the administrator.

Grant access to customer user on customer table:

query:

```
-- Granting SELECT, INSERT, UPDATE privileges on specified columns
GRANT SELECT (first_name, last_name, phone_no, email_id, dob),
INSERT (first_name, last_name, phone_no, email_id, dob),
UPDATE (first_name, last_name, phone_no, email_id, dob)
ON TABLE customer TO customer;
-- Granting SELECT, UPDATE privileges on specified columns
GRANT SELECT (id, type) ON TABLE customer TO customer;
-- cross check the preveligies
SELECT table_name, grantee, privilege_type,column_name
FROM information_schema.column_privileges
WHERE table_catalog = 'kl_dpdb_ecommerce_database'AND table_name = 'customer'AND
grantee = 'customer';
```

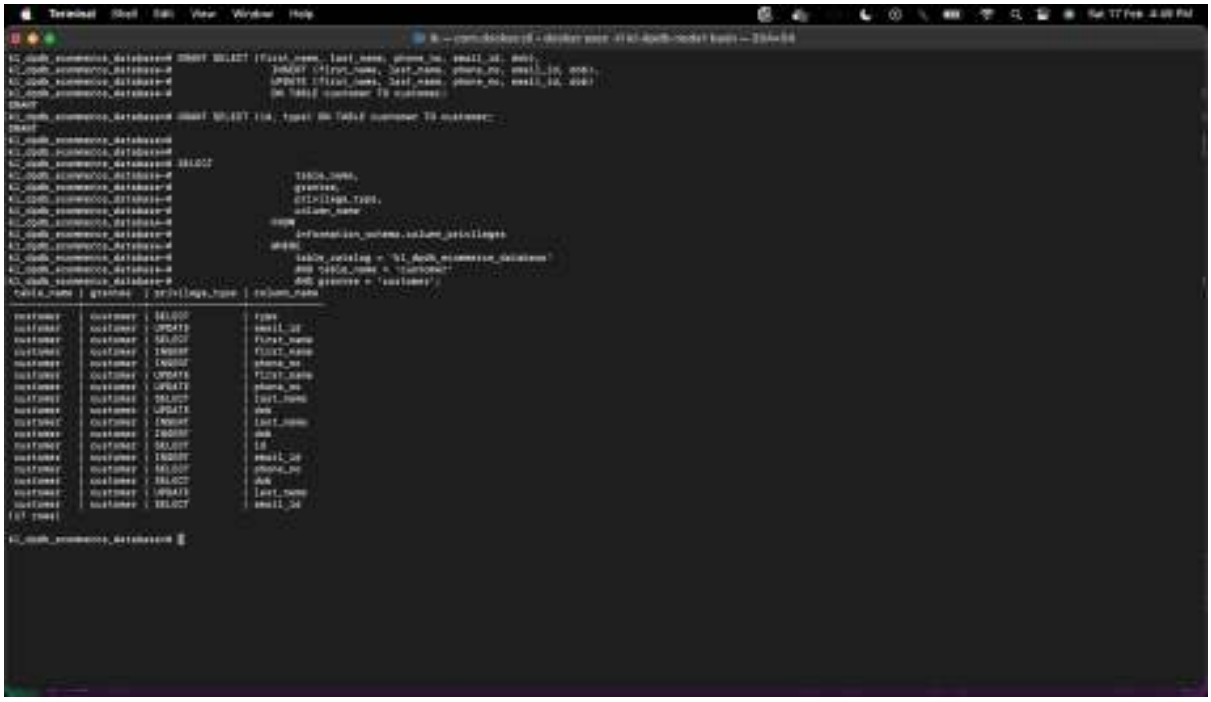


Fig-68: Grant access to customer user on customer table.

Explanation: Provided the privileges to customer on the customer table and displayed it.

Grant access to customer user on remaining tables:

query:

```
CREATE OR REPLACE FUNCTION grant_privileges_to_customer() RETURNS VOID AS $$
BEGIN
-- Granting privileges on the 'Products' table
EXECUTE 'GRANT SELECT ON TABLE Products TO customer';
-- Granting privileges on the 'cart' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE cart TO customer';
-- Granting privileges on the 'orders' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE orders TO customer';
-- Granting privileges on the 'transaction_summary' table
EXECUTE 'GRANT SELECT ON TABLE transaction_summary TO customer';
-- Granting privileges on the 'address' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE address TO customer';
-- Granting privileges on the 'product_category' table
EXECUTE 'GRANT SELECT ON TABLE product_category TO customer';
-- Granting privileges on the 'Reviews' table
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Reviews TO customer';
-- Granting privileges on the 'gift_vouchers' table
EXECUTE 'GRANT SELECT ON TABLE gift_vouchers TO customer';
END;
$$ LANGUAGE plpgsql;

-- Execute the function to grant privileges
SELECT grant_privileges_to_customer();
SELECT table_name, grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_catalog = 'kl_dpdb_ecommerce_database' AND grantee = 'customer';
```

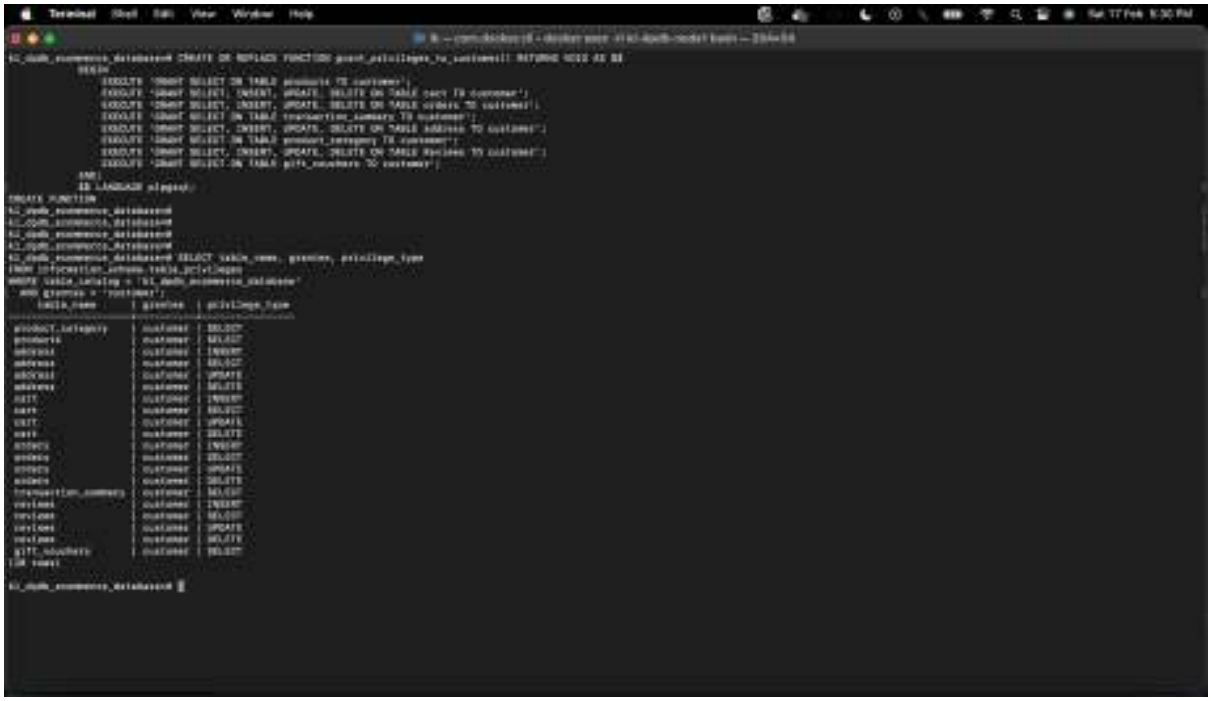


Fig-69: Grant access to customer user on remaining tables and display respective privileges.

query:

```
psql –U customer –d kl_dpdb_ecommerce_database
select * from products;
delete from products where id='b136c1cf-d1e8-483f-9064-1c866f25195f';
```

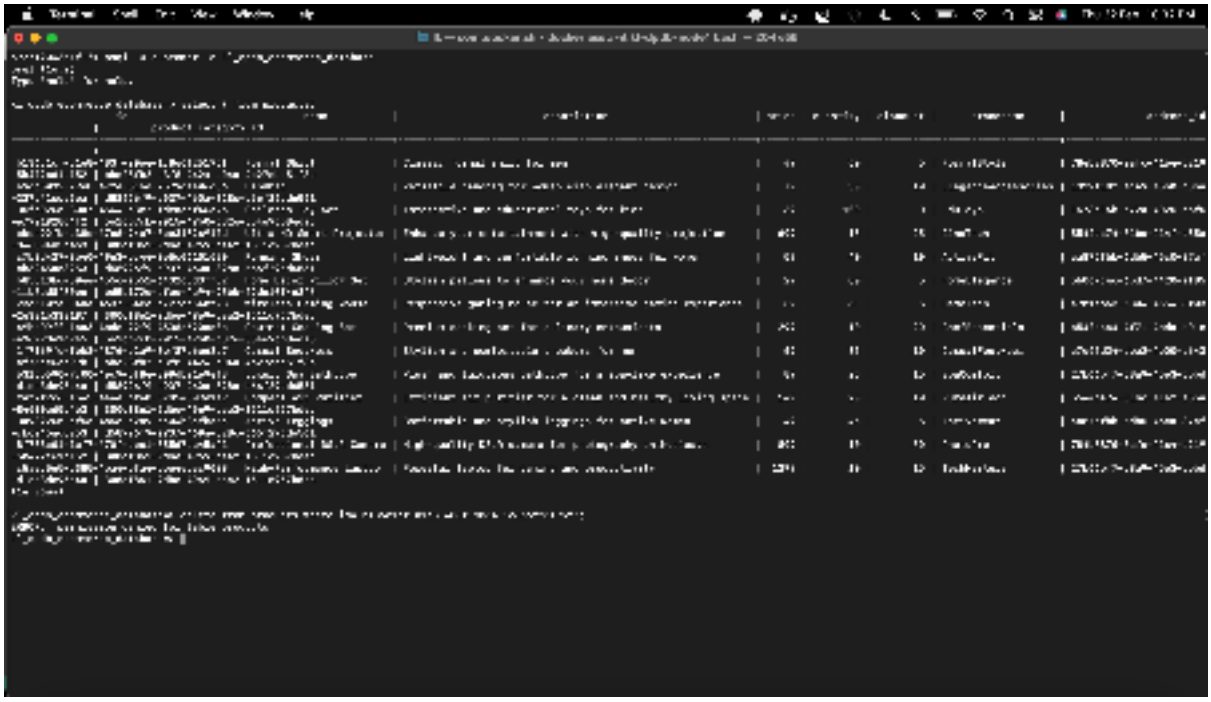


Fig-70: Delete Product from customer user.

Explanation: After providing the grant access to customer, tried to delete the data from products table got an error message " permission denied for table products".

Requirement – 6:

The process of copying the data from one server to another sever in the postgres is known as the postgres data replication. The postgres supports the replication strategy, which achieves fault tolerance, data migration, parallel execution with good performance. It has the single-master Replication, multi-master replication architecture. The replication can implement in uni-directional/bi-directional.

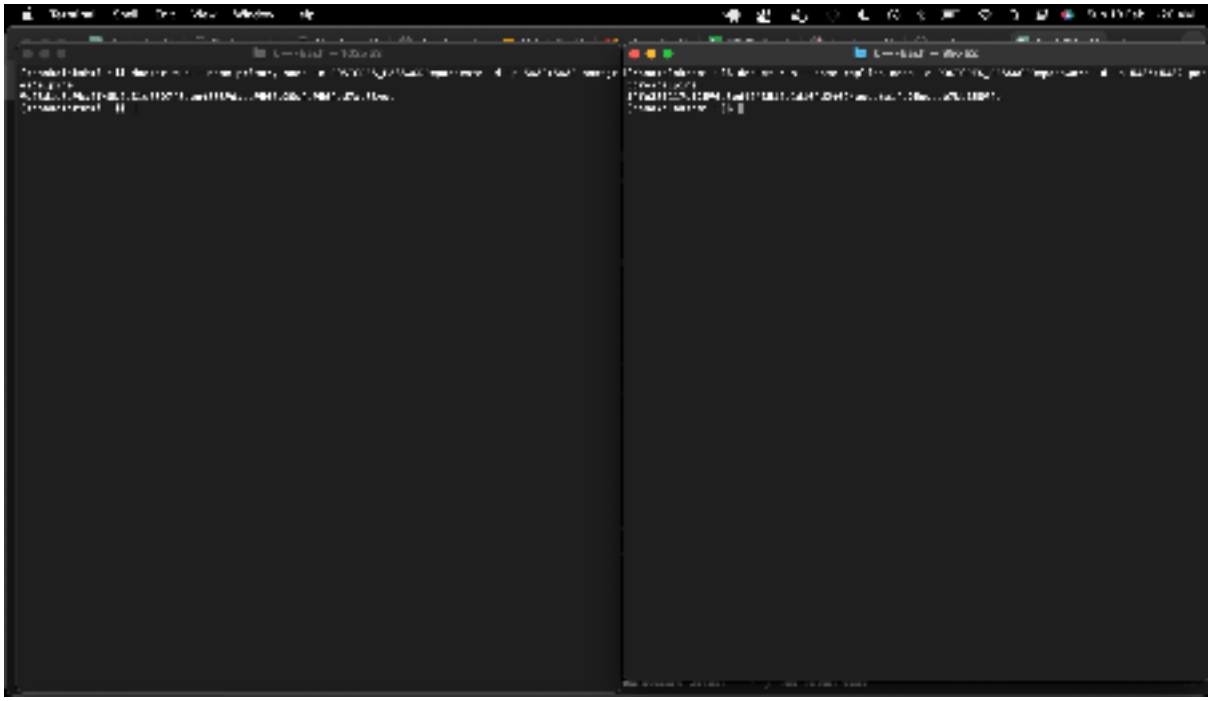


Fig-71: create two postgres instances.

query:

```
which initdb
initdb –D /tmp/db_primary
```

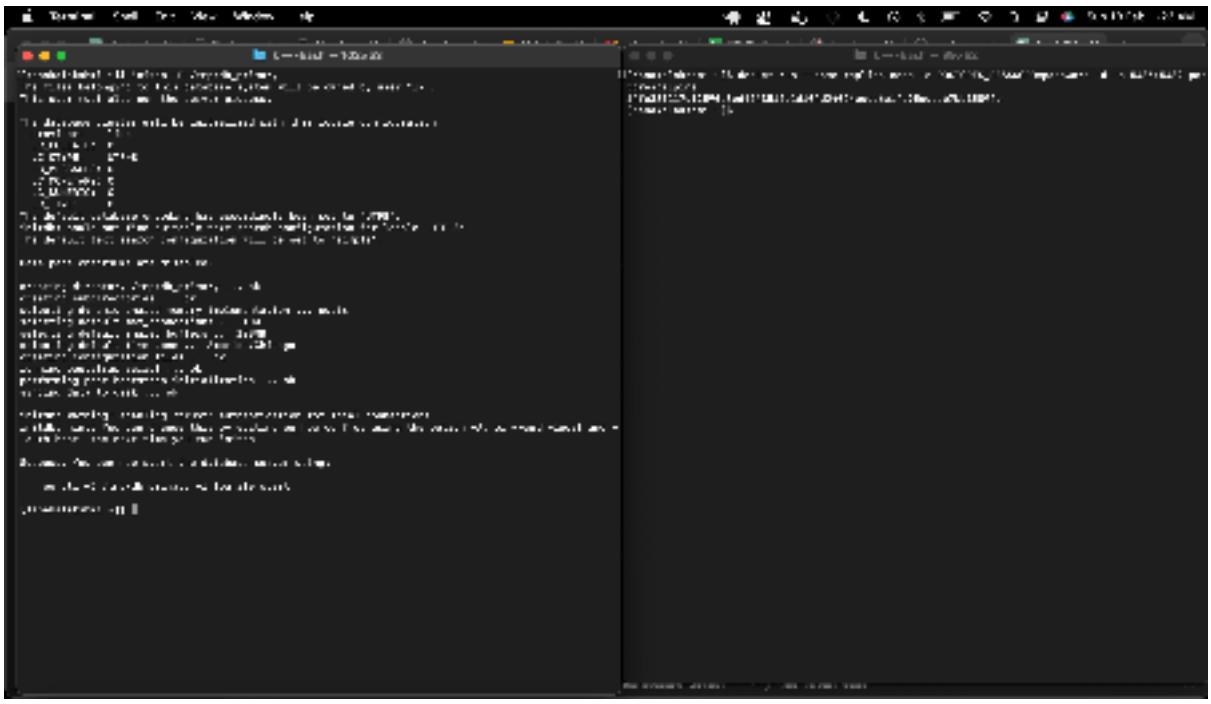


Fig-72: created Directory for the primary DB.

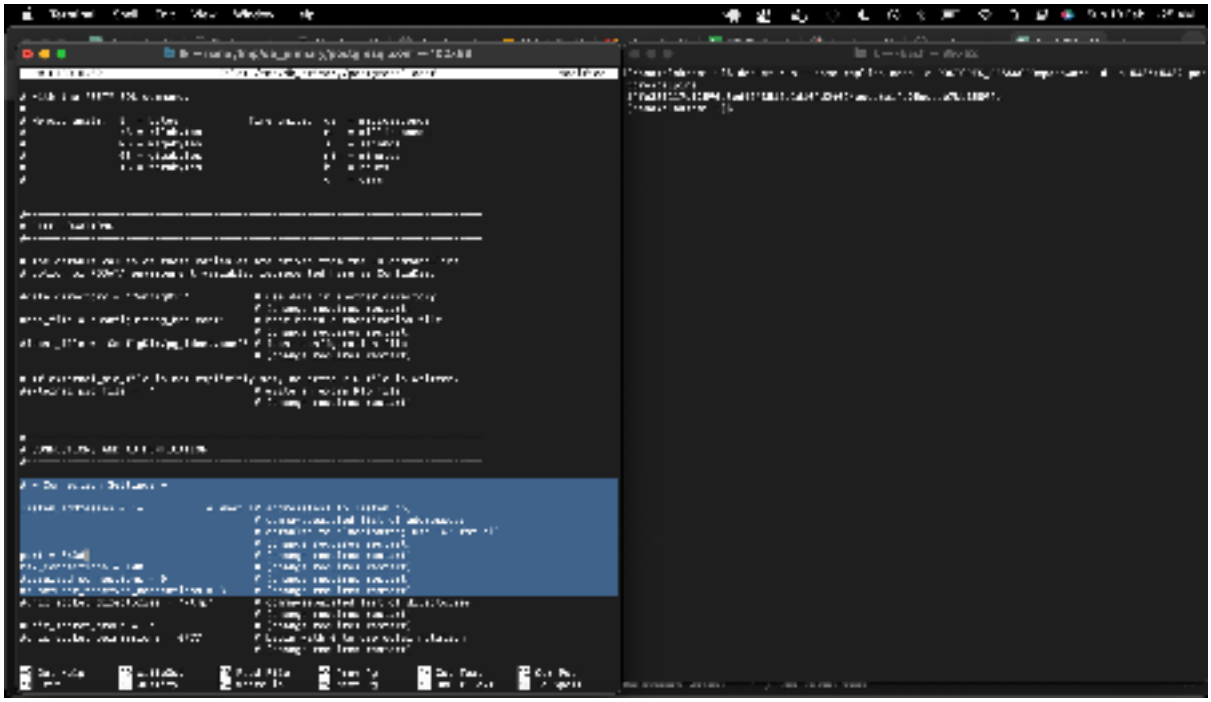


Fig-73: update configs in postgresql.conf

query:

```
nano /tmp/primary_db/postgresql.conf
pg_ctl –D /tmp/primary_db start
```

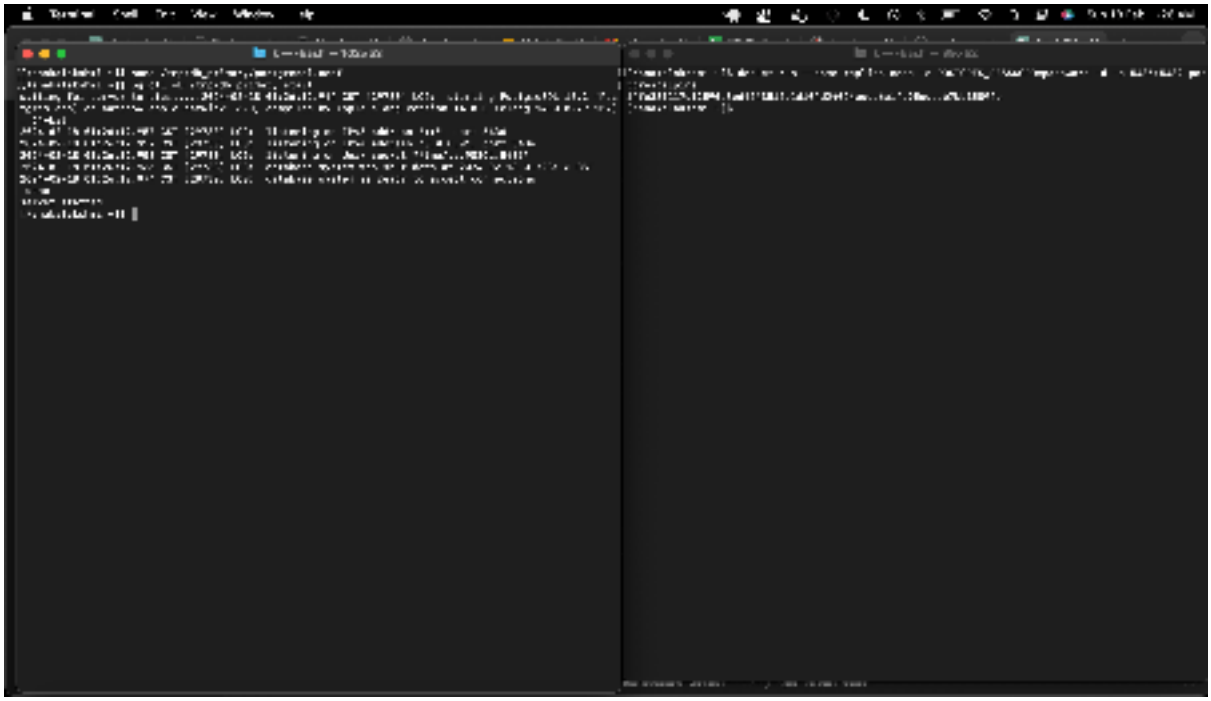


Fig-74: Start the postgres instance.

query:

```
psql --port=5436 postgres
create user replica_user replication;
```

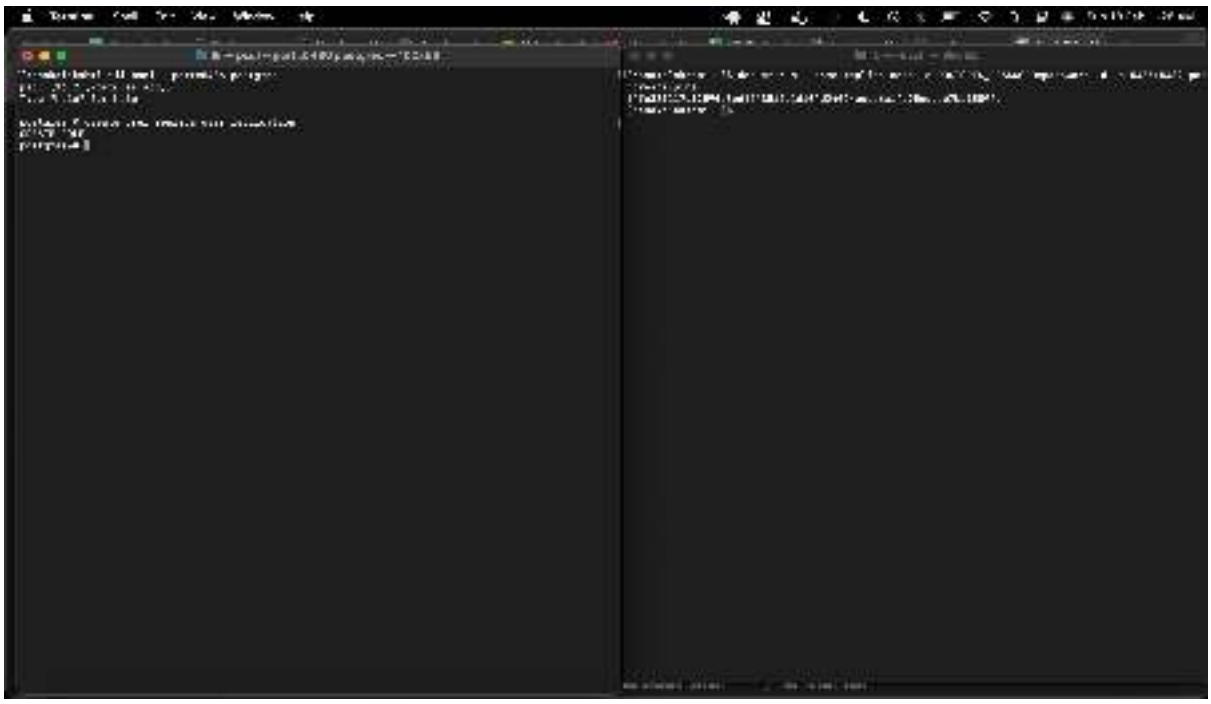


Fig-75: connect to postgres instance using port and create replica user.

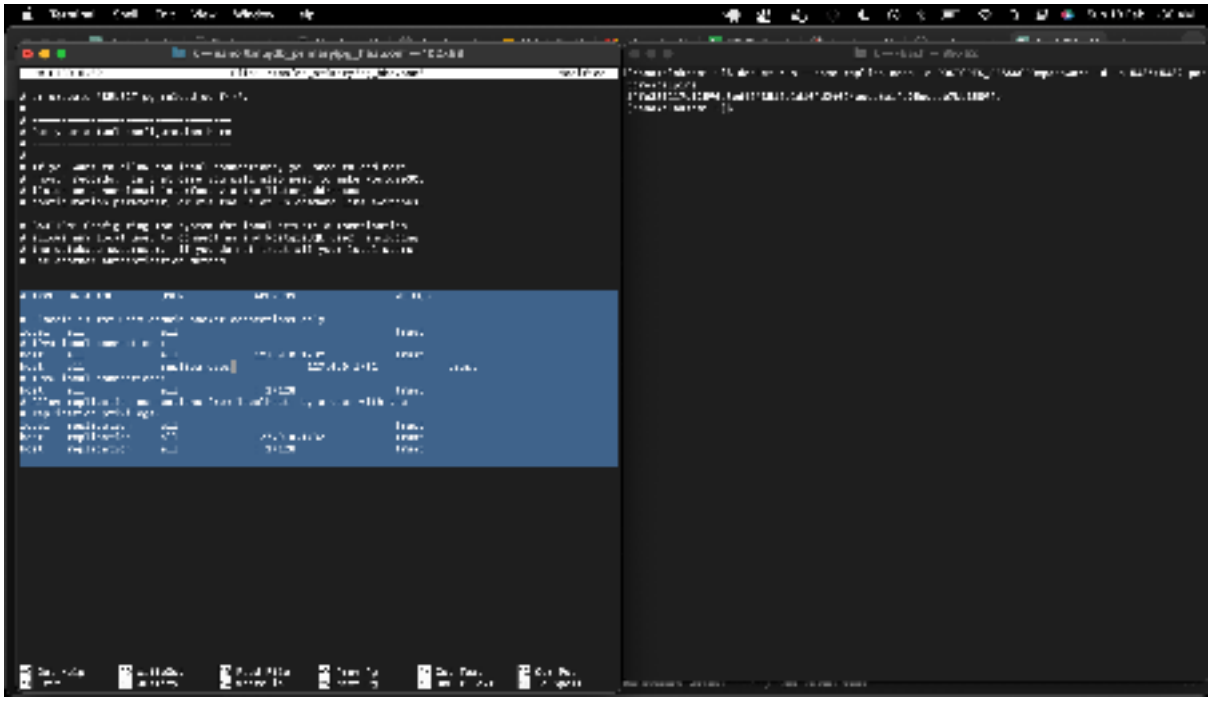


Fig-76: update configs in pg_hba.conf.

query:

```
pg_ctl –D /tmp/primary_db restart
```

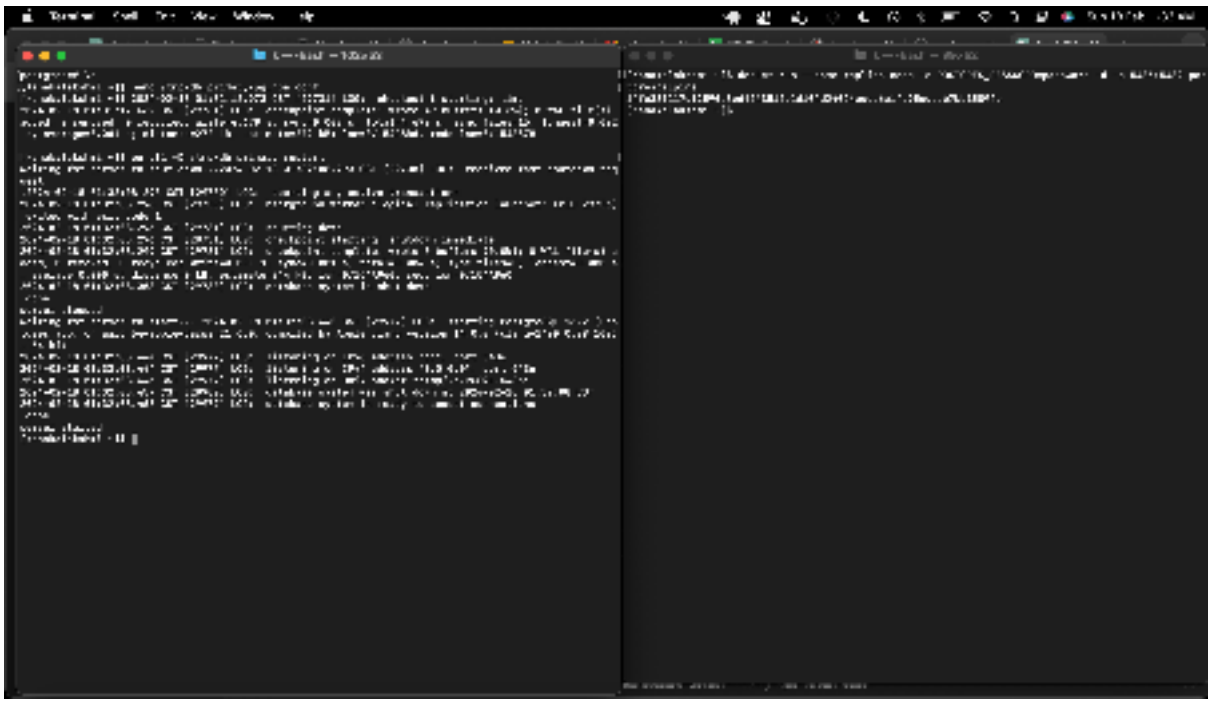


Fig-77: Restart the postgres instance.

Explanation: Once after all the configurations are done, restarted postgres instance to reflect the latest config changes.

query:

```
pg_basebackup -h localhost -U replica_user --checkpoint=fast -D /tmp/replica_db/ -R
--slot=data_node -C --port=5436
```

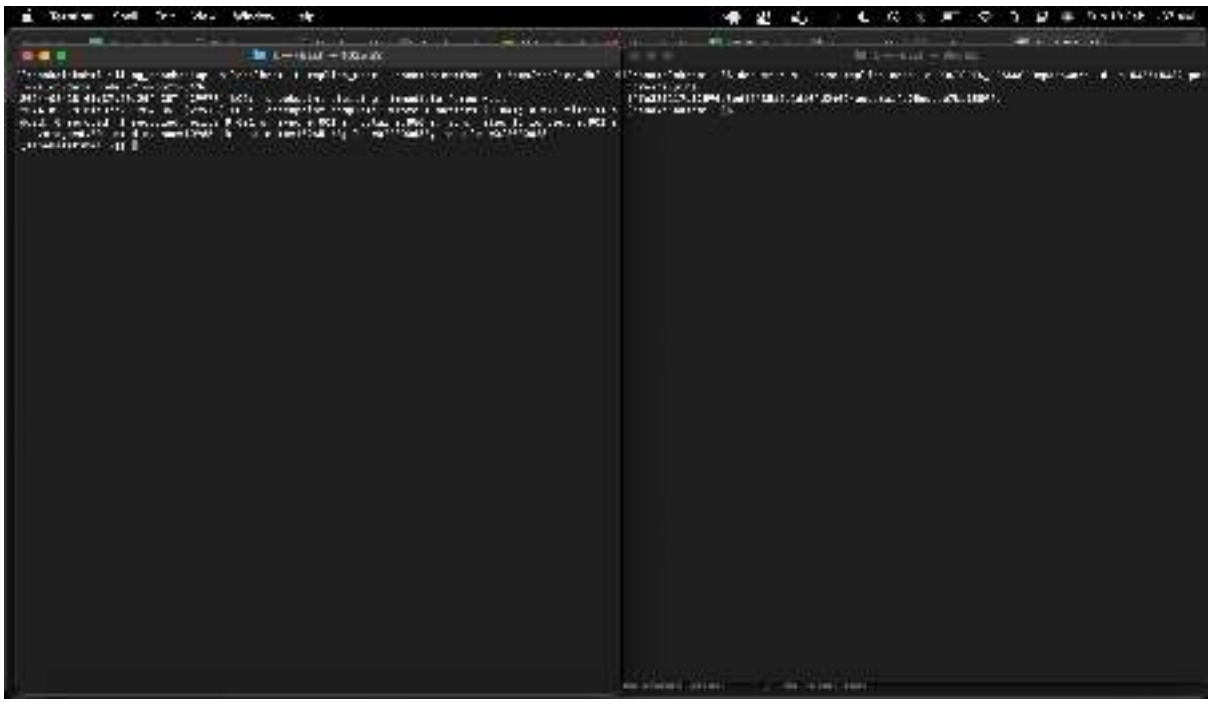


Fig-78: Copy the configurations to be used on the replica node.

Explanation: created the backup file with all the configuration, which makes easier in configuring the replica server.

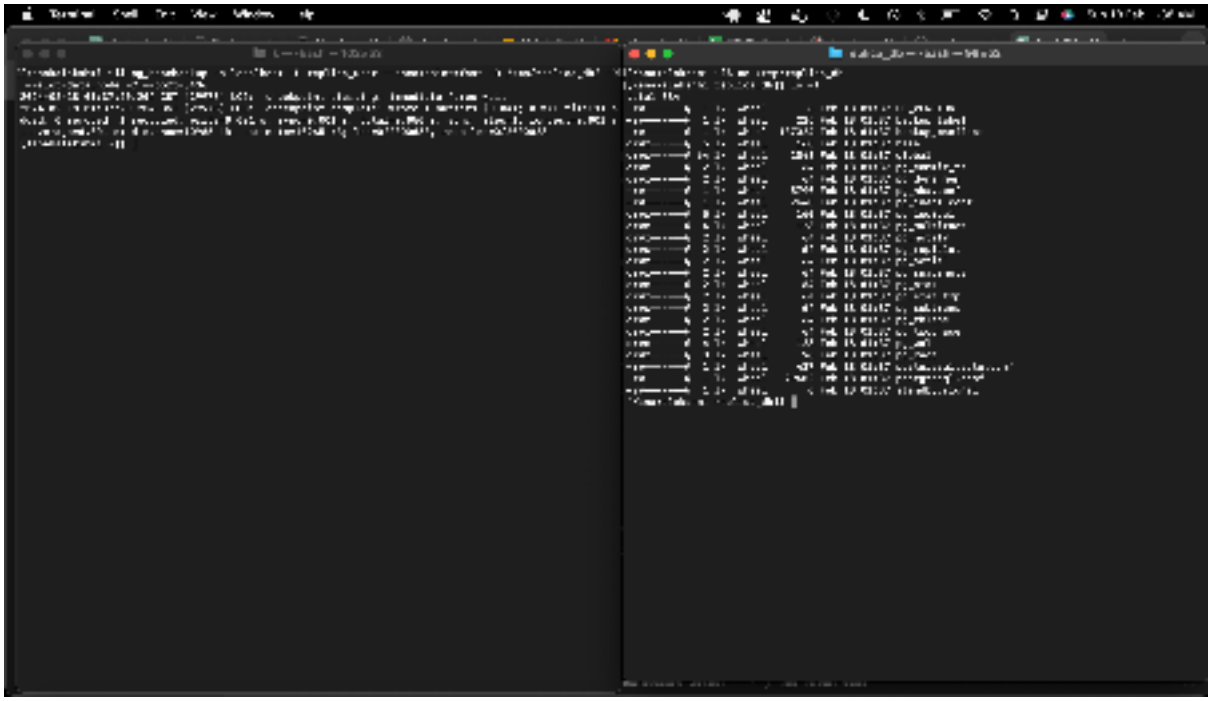


Fig-79: update the configs in replica node.

Explanation: The **standby.signal** is used for streaming replication. It always listens to the primary node to maintain the data synchronized. In case of any node failure, it detects it and communicate us by providing efficient logs.

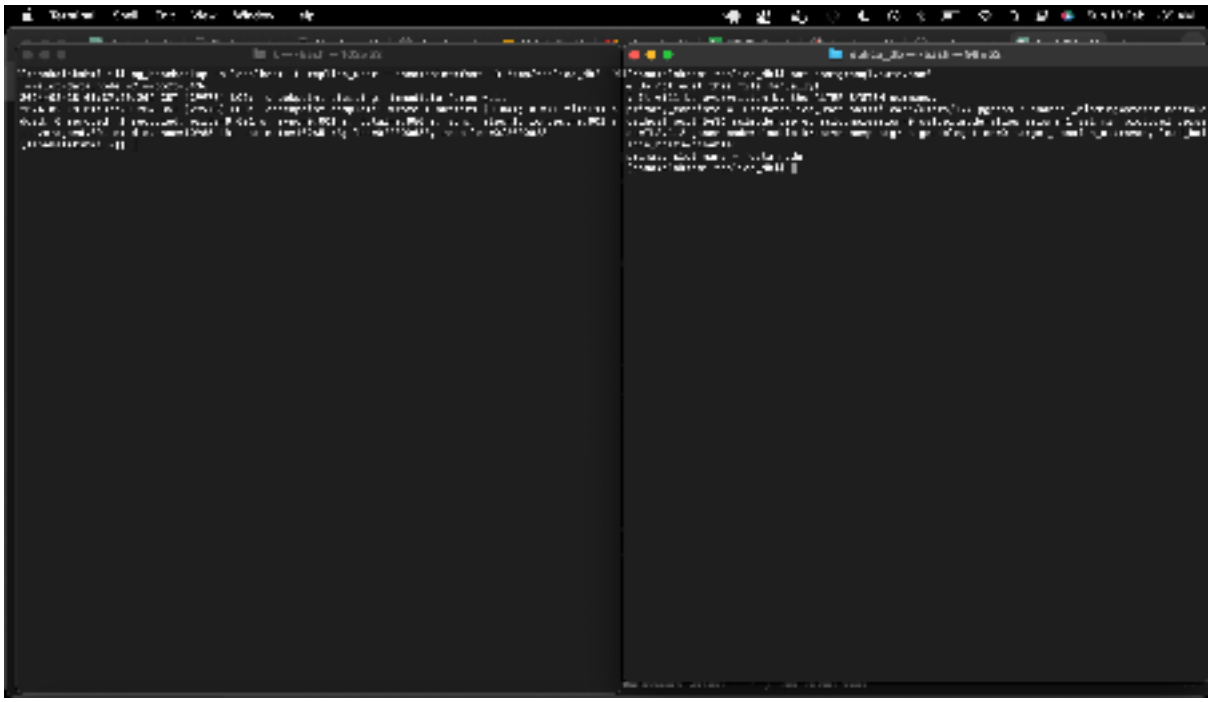


Fig-80: update the configs in replica node.

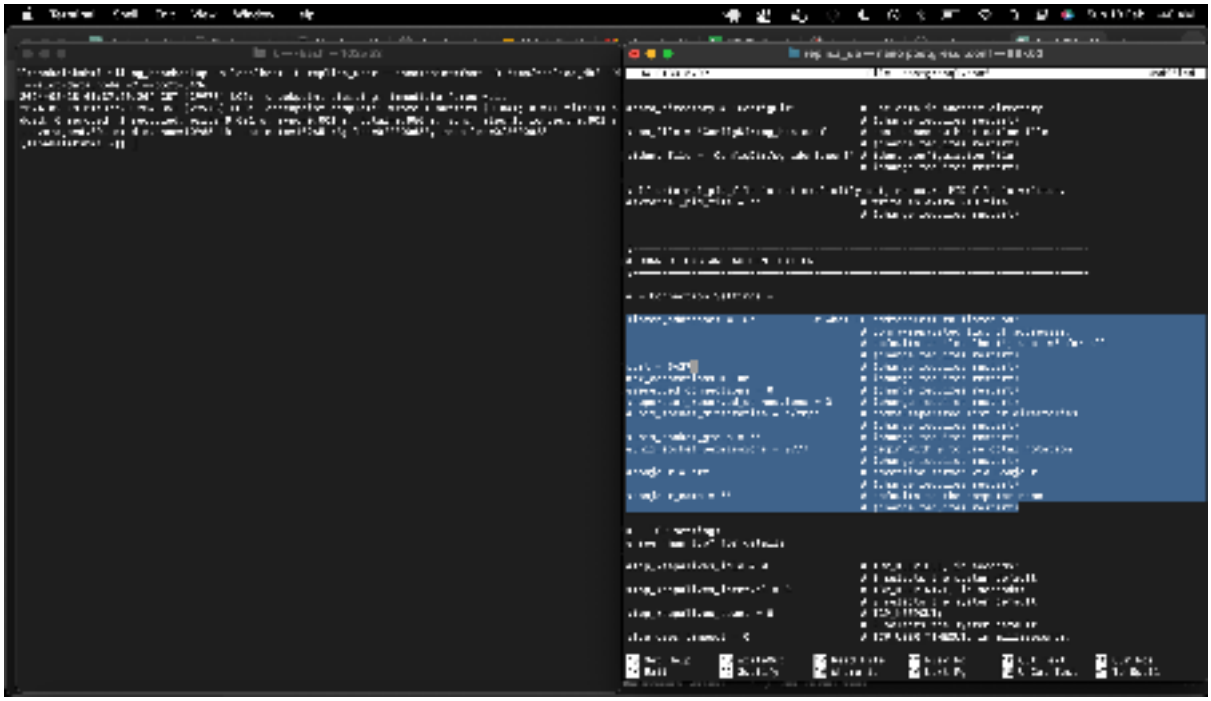


Fig-81: update configs in postgresql.conf file.

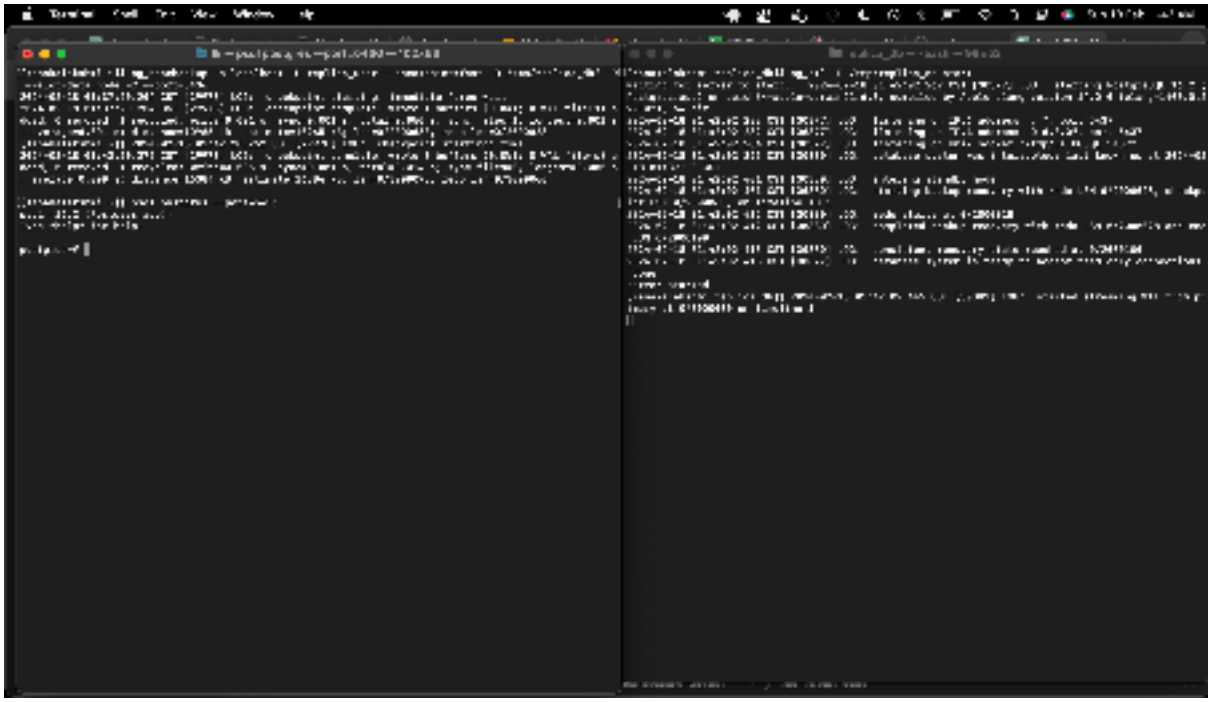


Fig-82: start postgres instance-2.

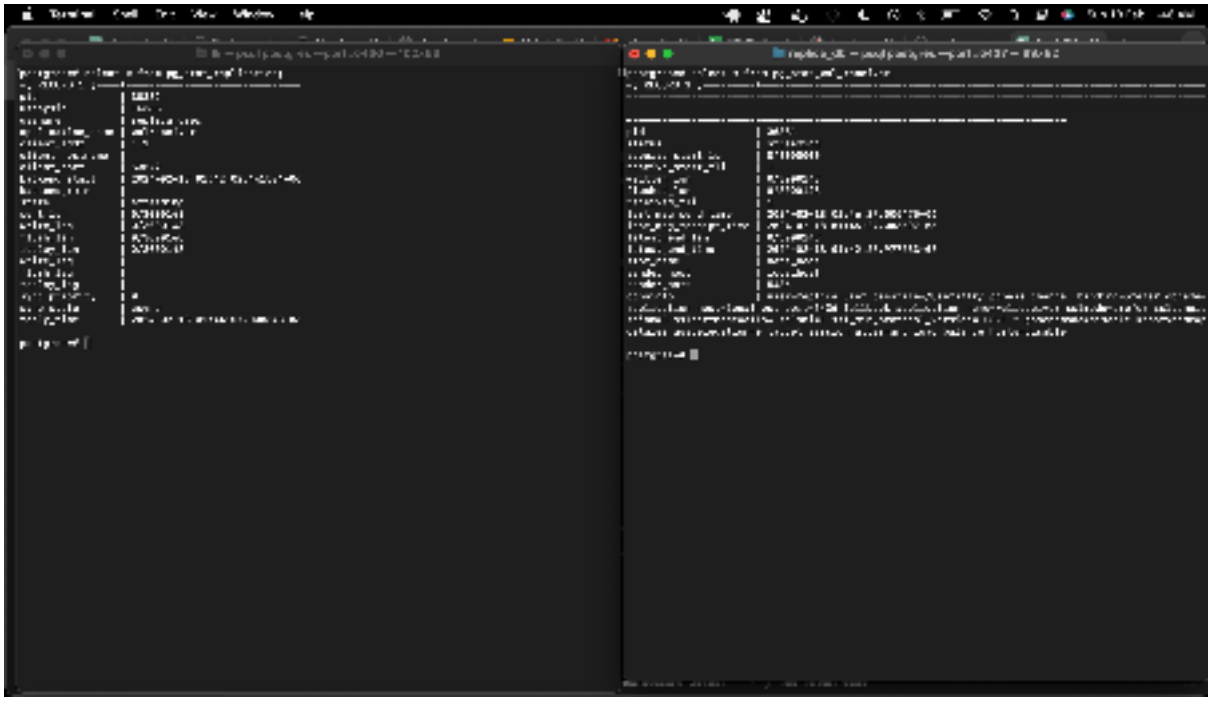


Fig-83: verify the configs in both the instances.

Explanation: After all the configurations done **primary node(port:5436)** and **replica node(5437**), verified the stats of respective instances.

Now, Let’s verify the replication strategy:

query:
instance-1:

```
CREATE TABLE customer (
id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
first_name VARCHAR(20) NOT NULL,
last_name VARCHAR(20) NOT NULL,
phone_no VARCHAR(20) NOT NULL,
email_id VARCHAR(255) NOT NULL,
dob DATE NOT NULL,
type VARCHAR(20) NOT NULL,
CONSTRAINT invalid_customer_phone CHECK (phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),
CONSTRAINT invalid_customer_email CHECK (email_id ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-
9.-]+\.[A-Za-z]{2,}$'),
UNIQUE (phone_no),
UNIQUE (email_id));
```

Instance-2:

```
select * from customer;
```

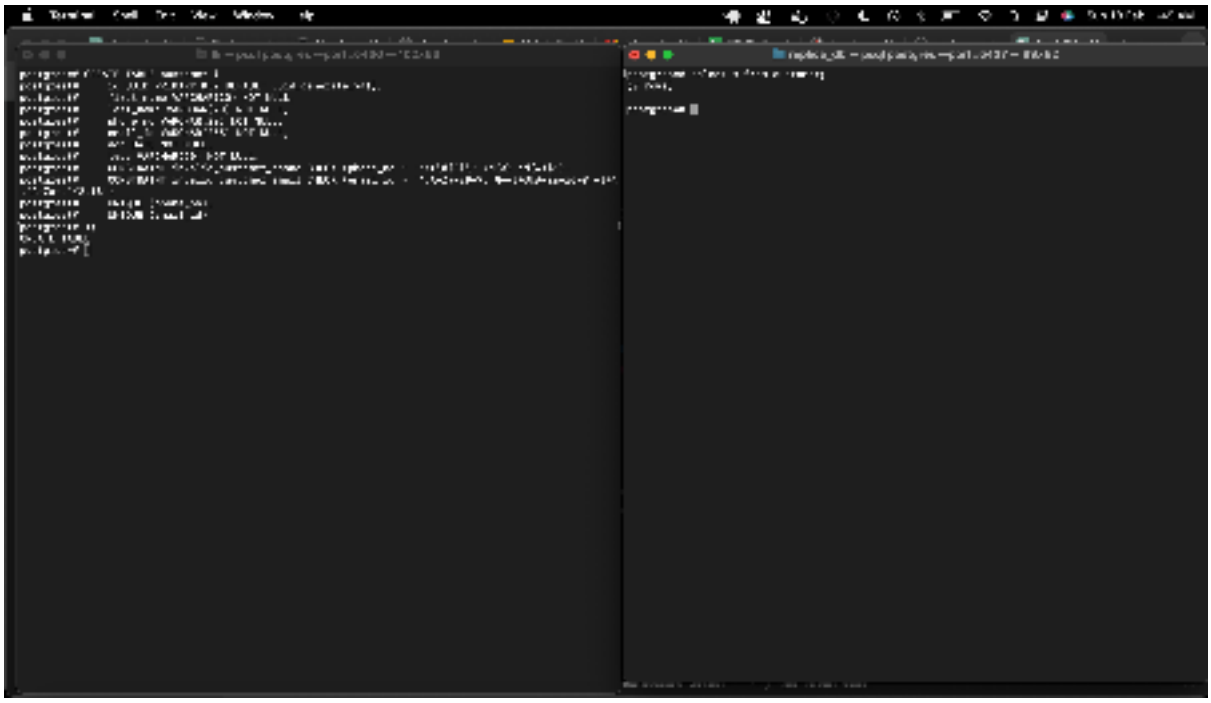


Fig-84: Create the table in instance-1 and verify data in instance-2.

Explanation: created the customer table in primary node(port:5436) and verified the data in the replica node(port:5437).

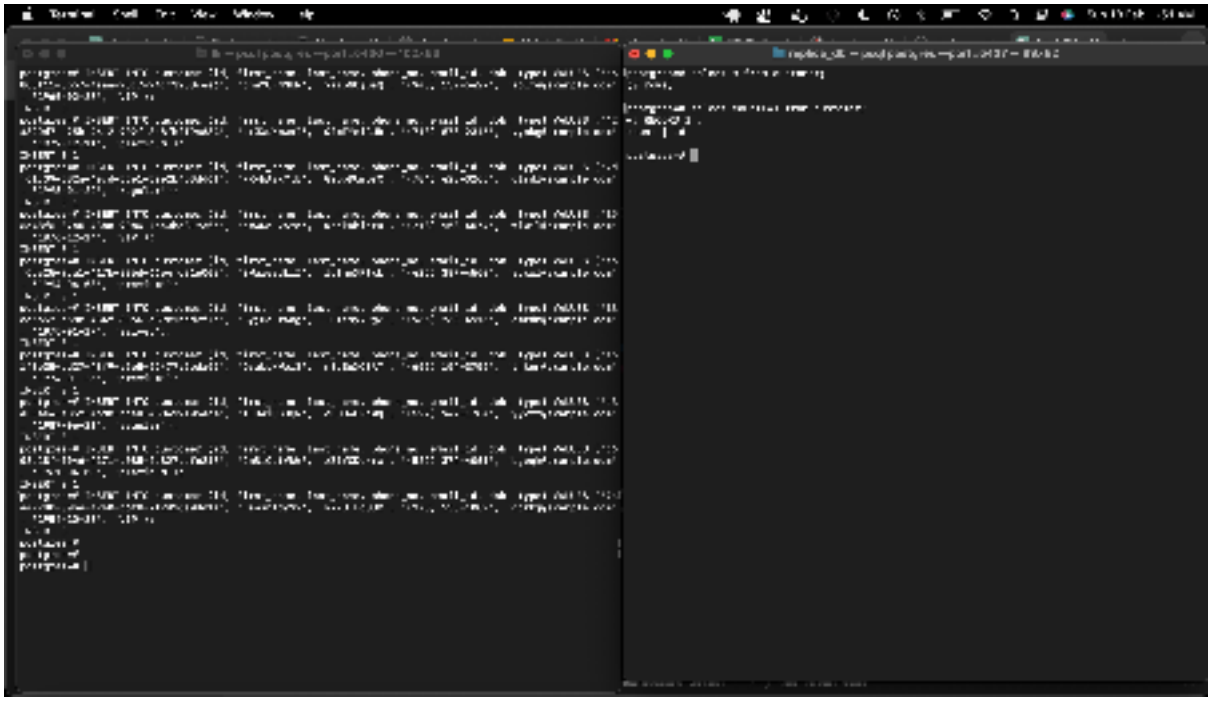


Fig-85: Insert records in instance-1 and verify the count of records from instance-2.

Explanation: inserted data into customer table in primary node(port:5436) and read the count of records in the replica node(port:5437).

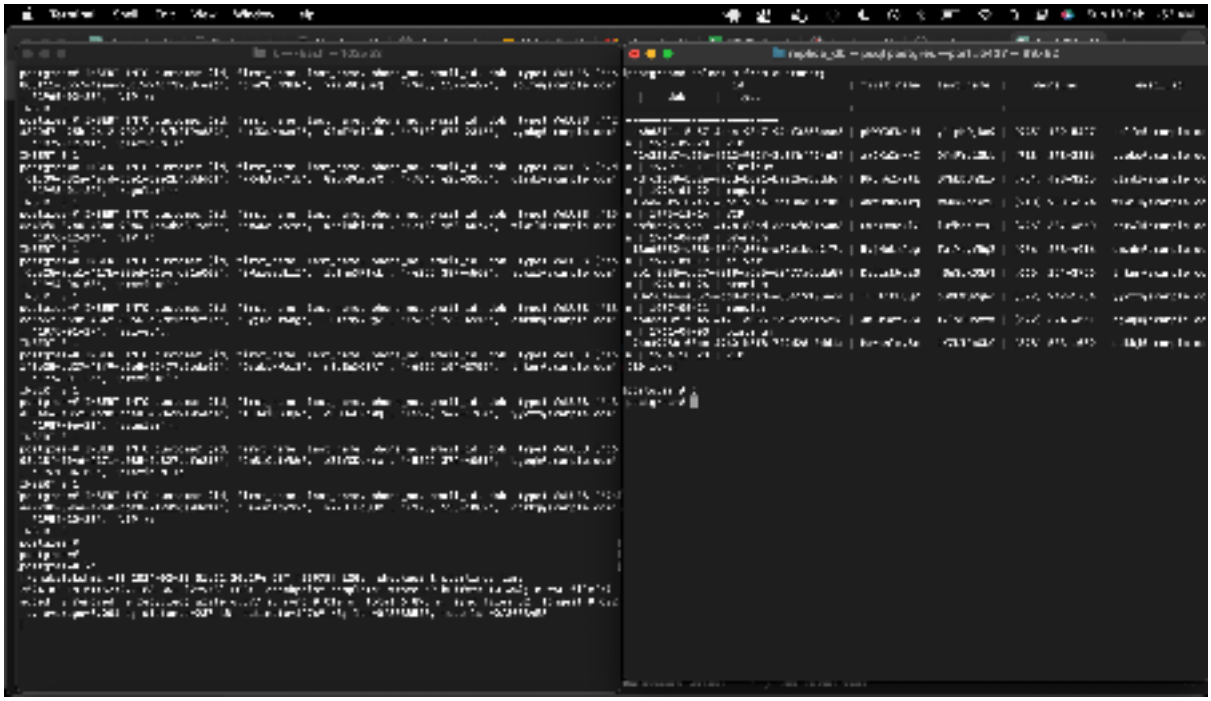


Fig-86: Display the data in instance-2 and exit instance-1.

query:

```
pg_ctl –D /tmp/primary_db stop ––stopping the node1
```

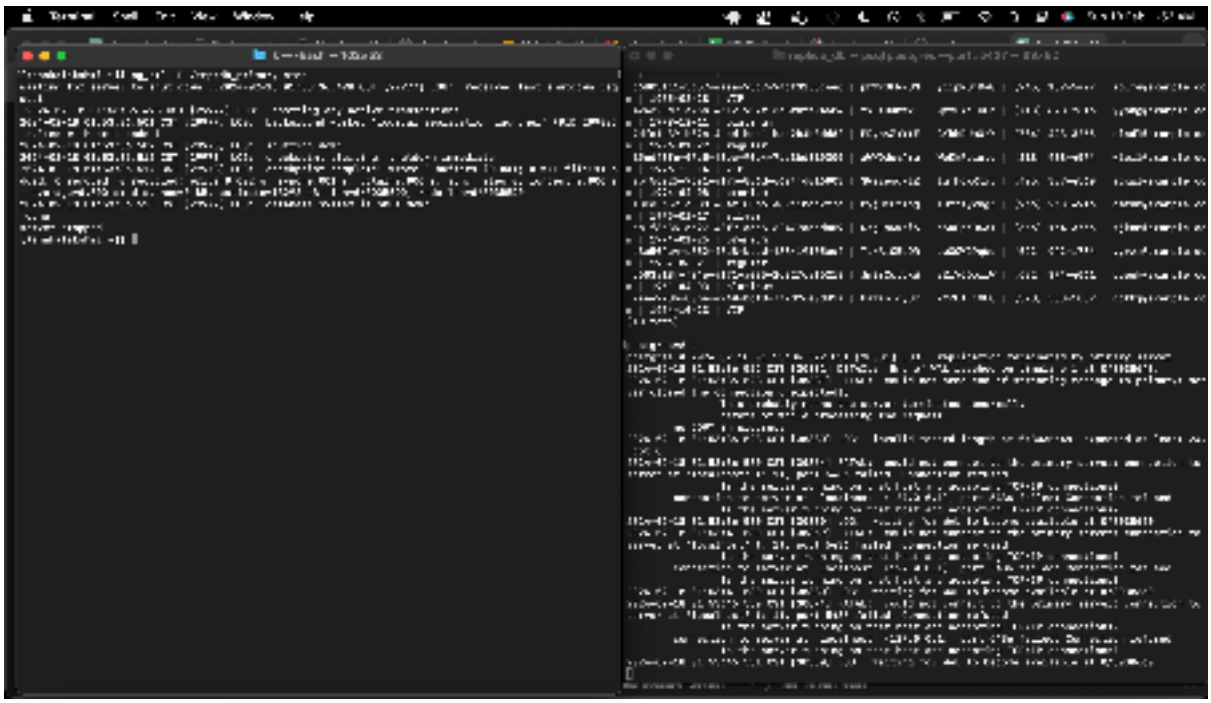


Fig-87: Bring down the instance-1.

Explanation: To verify the node failure, brought the primary node(port:5436) and verified the logs in replica node(port:5437).

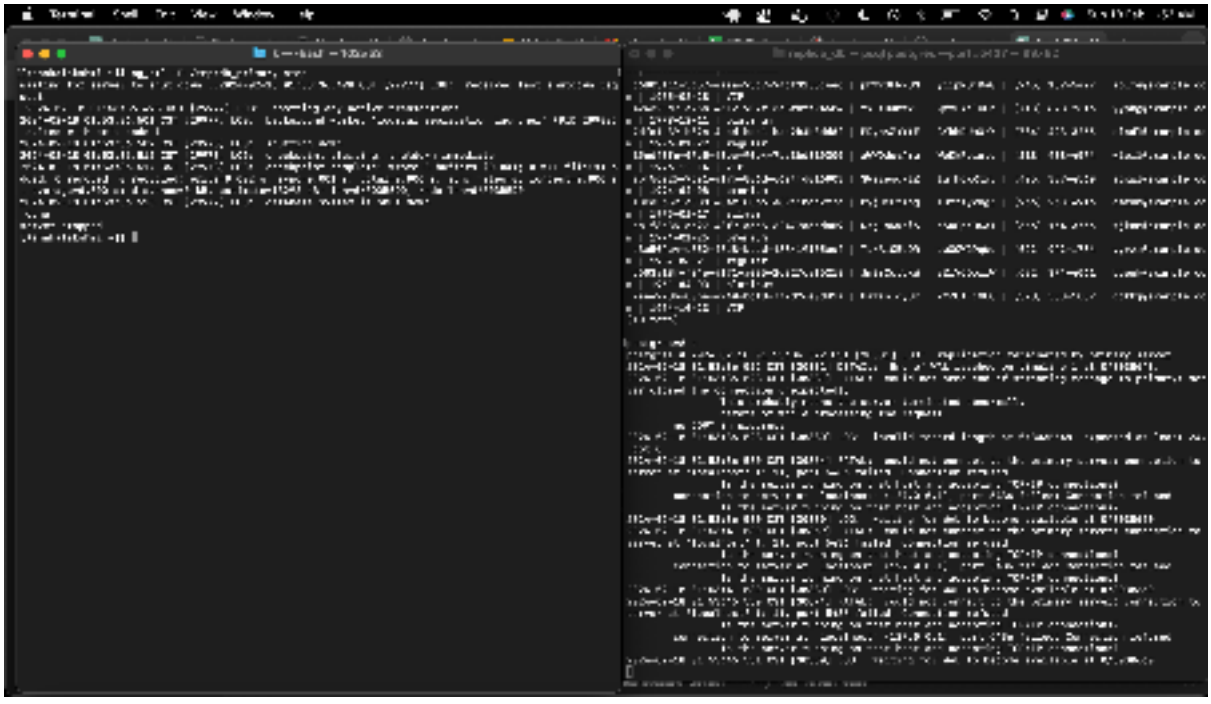


Fig-88: Observe logs in instance-2

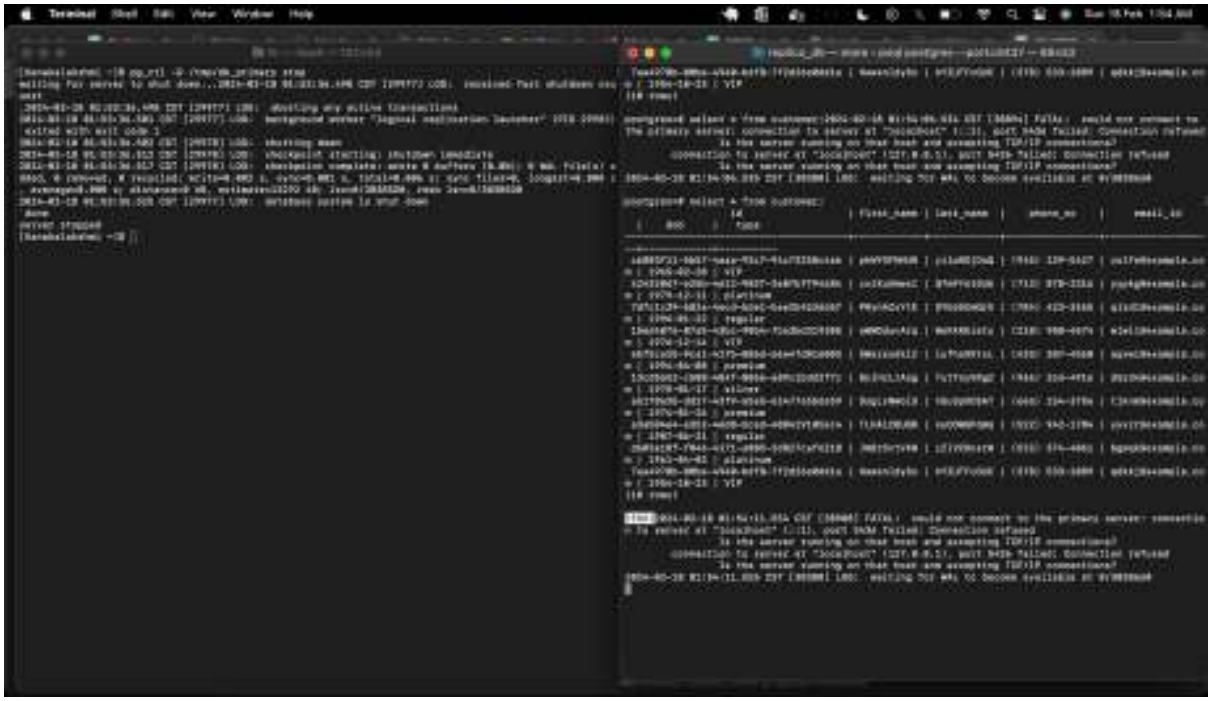


Fig-89: Display the data in instance-2.

Explanation: Even though the primary node is down, can still read the data from replica node. Which achieves the **fault tolerance**.

Requirement – 7:

Parallel Query Execution:
It speeds up the execution of queries. It adjusts the machine usage according to workloads. queries can scan massive datasets, such as join statements.

query:

```
CREATE OR REPLACE FUNCTION create_customer(
    p_first_name VARCHAR(20),
    p_last_name VARCHAR(20),
    p_phone_no VARCHAR(20),
    p_email_id VARCHAR(255),
    p_dob DATE,
    p_flat_no INTEGER,
    p_street VARCHAR(50),
    p_city VARCHAR(50),
    p_state VARCHAR(50),
    p_country VARCHAR(50),
    p_zip_code VARCHAR(10)
) RETURNS VOID AS $$
DECLARE
    v_customer_id UUID;
    v_address_id UUID;
BEGIN
    -- Insert into customer table
    INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type)
    VALUES (uuid_generate_v4(), p_first_name, p_last_name, p_phone_no, p_email_id,
p_dob, 'regular')
    RETURNING id INTO v_customer_id;

    -- Insert into address table
    INSERT INTO address (id, flat_no, street, city, state, country, zip_code)
    VALUES (uuid_generate_v4(), p_flat_no, p_street, p_city, p_state, p_country,
p_zip_code)
    RETURNING id INTO v_address_id;

    -- Insert into customer_address table
    INSERT INTO customer_address (customer_id, address_id, default_address)
    VALUES (v_customer_id, v_address_id, true);
END;
$$ LANGUAGE plpgsql PARALLEL SAFE;
```

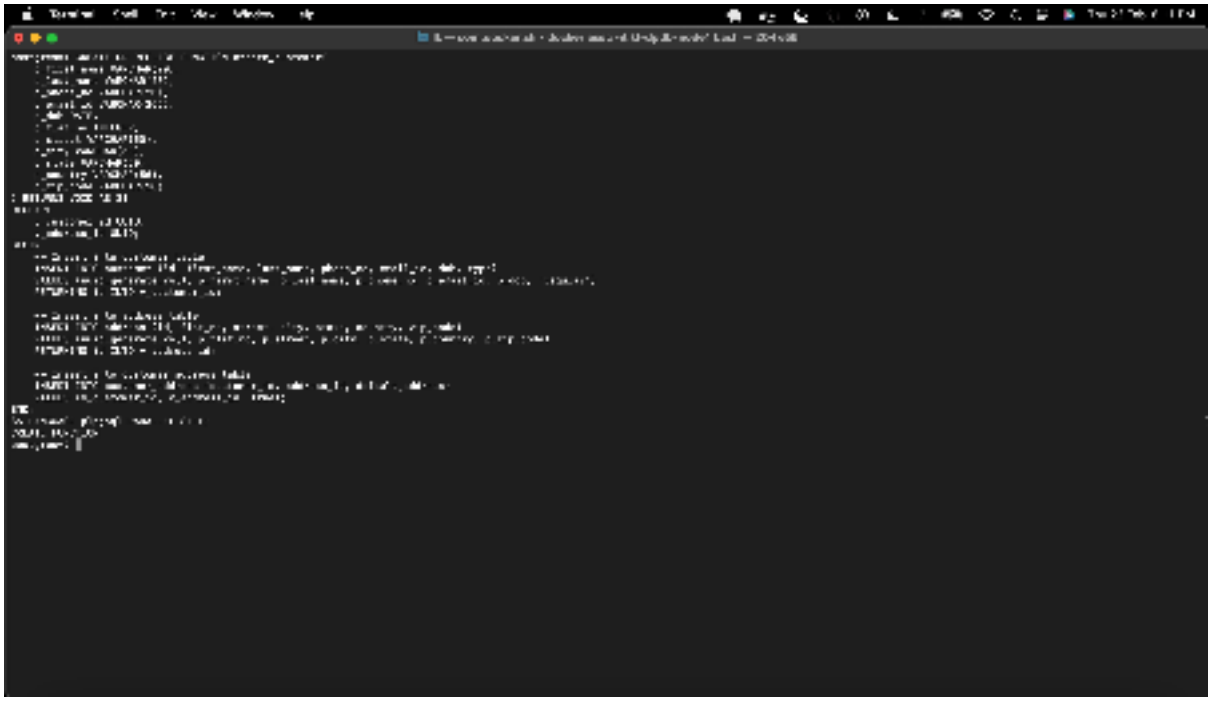


Fig-90: Customer signup function

Explanation: Created create_customer function, which is used as the customer sign-up, which reads the input details of the customer and stores the data in the database. It input details are first_name, last_name, phone_no, emailId, dob, flat_no, street, city, state, country, zip-code. It stores the data in the customer, address, and customer_address table. As it's a new sign-up customer type will be regular, based on the orders of the customer, the type will we upgraded from backend and default address will be true as it's the first address. Implemented the function with **PARALLEL SAFE.**

query:

```
SELECT create_customer(
            'Jack',
            'Teresa',
            '(910) 436-7890',
            'jack.teresa@example.com',
            '1990-01-01',
            401,
            'Main Street',
            'Tampa',
            'Florida',
            'USA',
            '65341'
        );
```

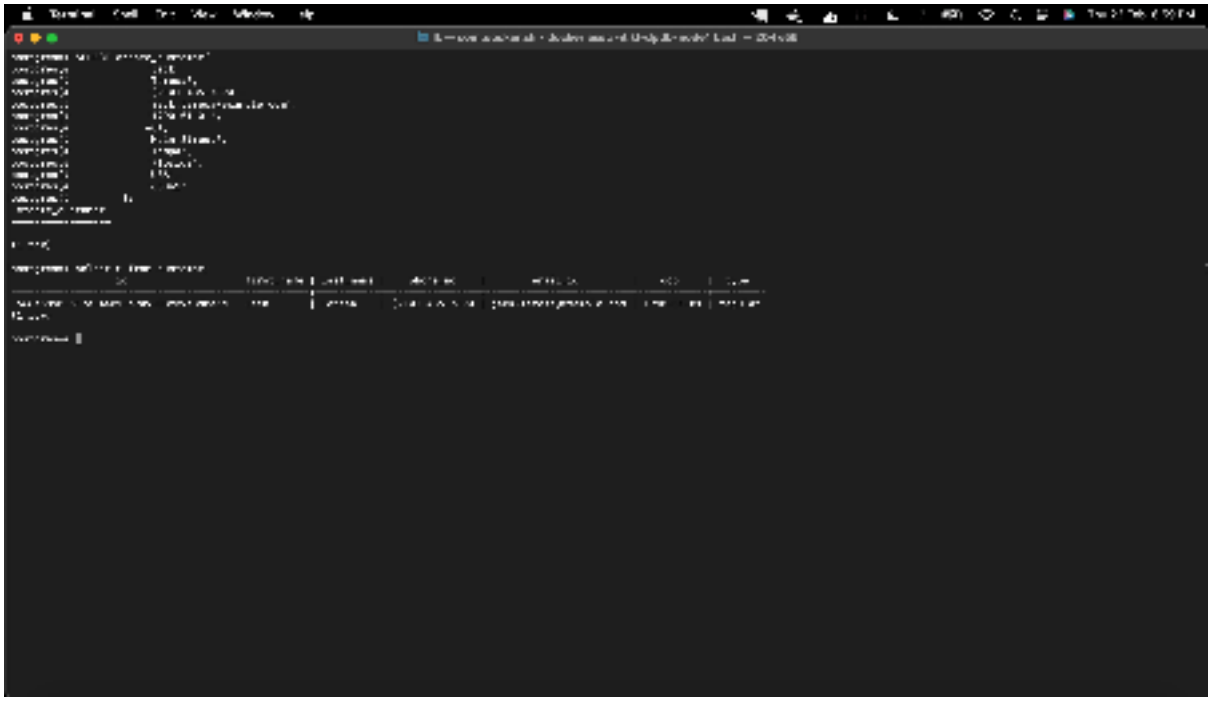


Fig-91: Customer sign-up details

Explanation: This is the sign-up details provided by the customer and triggering the create_customer function. Where will provide all the customer details such as first_name, last_name, phone_no, email_id, dob and type, and all the other details will update to address and customer_address tables.

PL/pgSQL code to place order by customer:

query:

```
CREATE OR REPLACE FUNCTION place_order_by_customer(
p_customer_id UUID,
p_product_id UUID,
p_product_quantity INTEGER,
p_delivery_partner_name VARCHAR(50),
p_delivery_partner_phone_no VARCHAR(20),
p_delivery_partner_email VARCHAR(255)
) RETURNS VOID AS $$
DECLARE
v_order_id UUID;
v_address_id UUID;
v_delivery_partner_id UUID;
BEGIN
-- Step 1: Check if the product and quantity are available
PERFORM 1 FROM products WHERE id = p_product_id AND quantity >= p_product_quantity;
IF NOT FOUND THEN
RAISE EXCEPTION 'Customer %, sorry, the requested product is not available. Please
try again later.', p_customer_id;
END IF;
-- Step 2: Insert data into orders
INSERT INTO orders (status, address_id, customer_id)
VALUES ('Processing', (SELECT address_id FROM customer_address WHERE customer_id =
p_customer_id AND default_address = true), p_customer_id)
RETURNING id INTO v_order_id;
-- Step 3: Insert data into transaction_summary
INSERT INTO transaction_summary (total_amount_paid, payment_type, order_id)
VALUES ((SELECT price * p_product_quantity FROM products WHERE id = p_product_id),
'Credit Card', v_order_id);
-- Step 4: Insert data into delivery_partner
INSERT INTO delivery_partner (name, phone_no, email, order_id)
VALUES (p_delivery_partner_name, p_delivery_partner_phone_no,
p_delivery_partner_email, v_order_id)
RETURNING id INTO v_delivery_partner_id;
-- Step 5: Insert data into customer_delivery_partner
INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
VALUES (p_customer_id, v_delivery_partner_id);
-- Step 6: Insert data into orders_products
INSERT INTO orders_products (order_id, product_id, quantity)
VALUES (v_order_id, p_product_id, p_product_quantity);
-- Step 7: Update product quantity
UPDATE products SET quantity = quantity - p_product_quantity WHERE id =
p_product_id;
EXCEPTION
WHEN OTHERS THEN
-- Rollback the transaction in case of an exception
RAISE EXCEPTION 'Error in place_order_by_customer: %', SQLERRM;
END;
$$ LANGUAGE plpgsql;
```

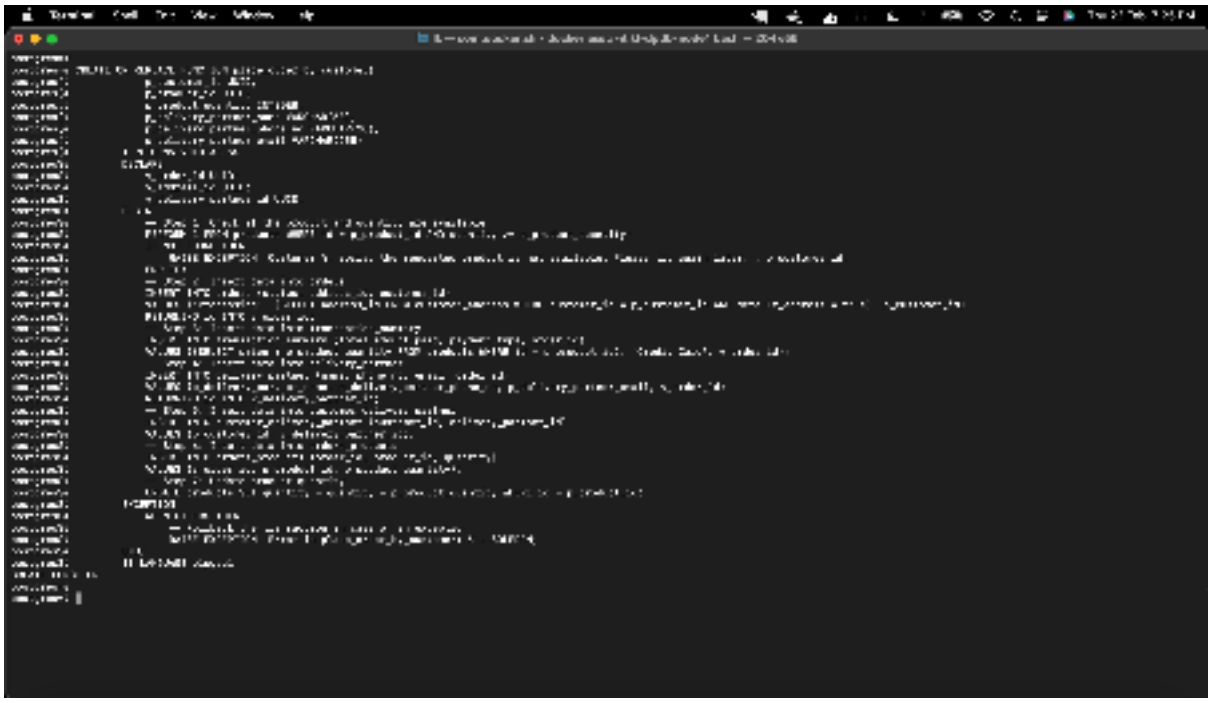


Fig-92: Create function to place order by customer.

Explanation: This is the function, where the customer can place the order successfully, and even delivery partner is assigned to the order. This function updates the data in all the respective tables and achieves the data consistency. Here the input details are customer_id, product_id, product_quantity, and delivery_partner details. Using these details the order of the customer will be placed and delivery partner will get assigned to it. In case of any error occurs during the transactions, ROLLBACK is implemented which helps to maintain the data consistently.

Input data to place the order by customer:

query:

```
DO $$
BEGIN
PERFORM place_order_by_customer(
'668e5890-517a-4ae1-b909-03f39a3d8e6d', -- customer_id
'f5148b50-d8f4-45a7-a5aa-03cf6e047627', -- product_id
2,                                      -- product_quantity
'John Doe',                             -- delivery_partner_name
'(973) 456-7890',                       -- delivery_partner_phone_no
'john.doe@example.com'                  -- delivery_partner_email
);
END $$;
```

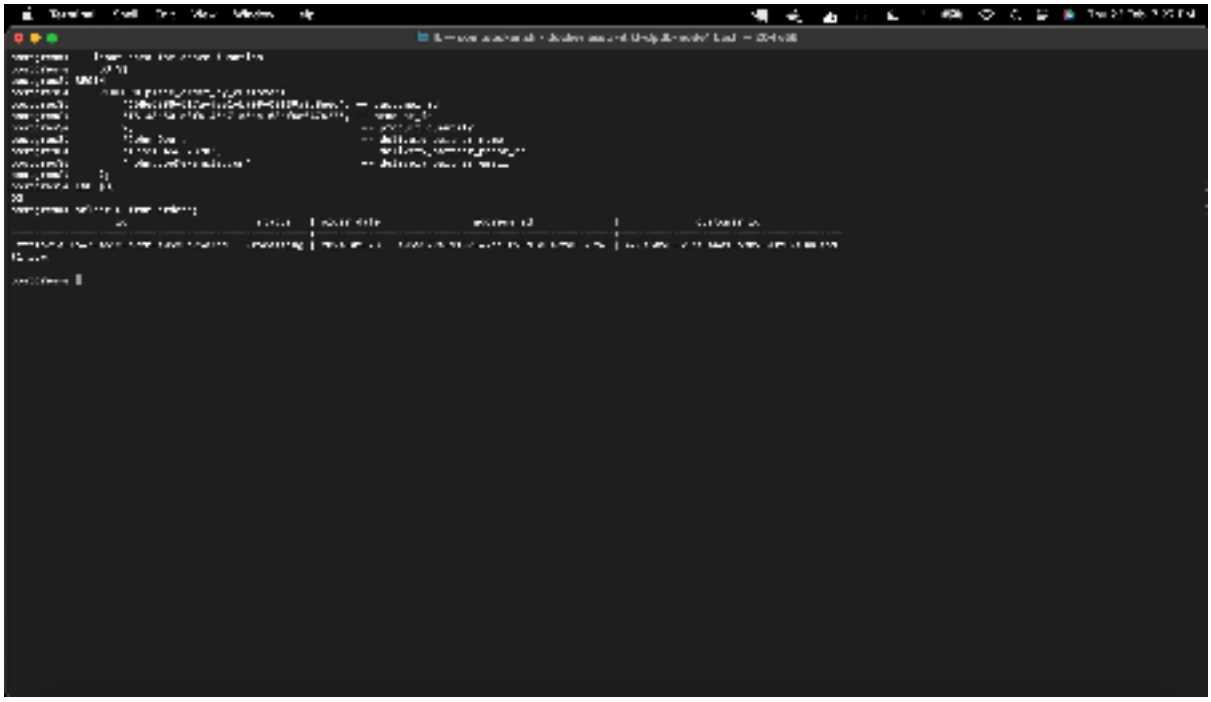


Fig-93: Input data to trigger place_order_by_customer and display the orders.

Explanation: This is the required input data to place the order by the customer. Few details will be read from the customer, and few details will get from our existing database. The delivery partner details are stores in database. Use those delivery partner details to assign an order to delivery partner.

use case: use multiple tables such as customer products, orders, transaction_summary and get the successfully ordered products paid more than their average total.

query:

```
SELECT prod.id AS product_id, prod.name AS product_name, ord.id AS order_id,
ord.status AS order_status,
ts.total_amount_paid, ts.payment_type, ts.date_of_payment
FROM products prod
JOIN orders_products ord_prod ON prod.id = ord_prod.product_id  -- Corrected alias
from 'ord' to 'ord_prod'
JOIN orders ord ON ord_prod.order_id = ord.id  -- Corrected alias from 'o' to 'ord'
JOIN transaction_summary ts ON ord.id = ts.order_id
JOIN product_category pc ON prod.product_category_id = pc.id
WHERE ord.customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'AND pc.section =
'Electronics' AND ts.total_amount_paid > (
                SELECT AVG(total_amount_paid)
                FROM transaction_summary
                WHERE order_id IN (
                    SELECT id
                    FROM orders
                    WHERE customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d')
            );
```

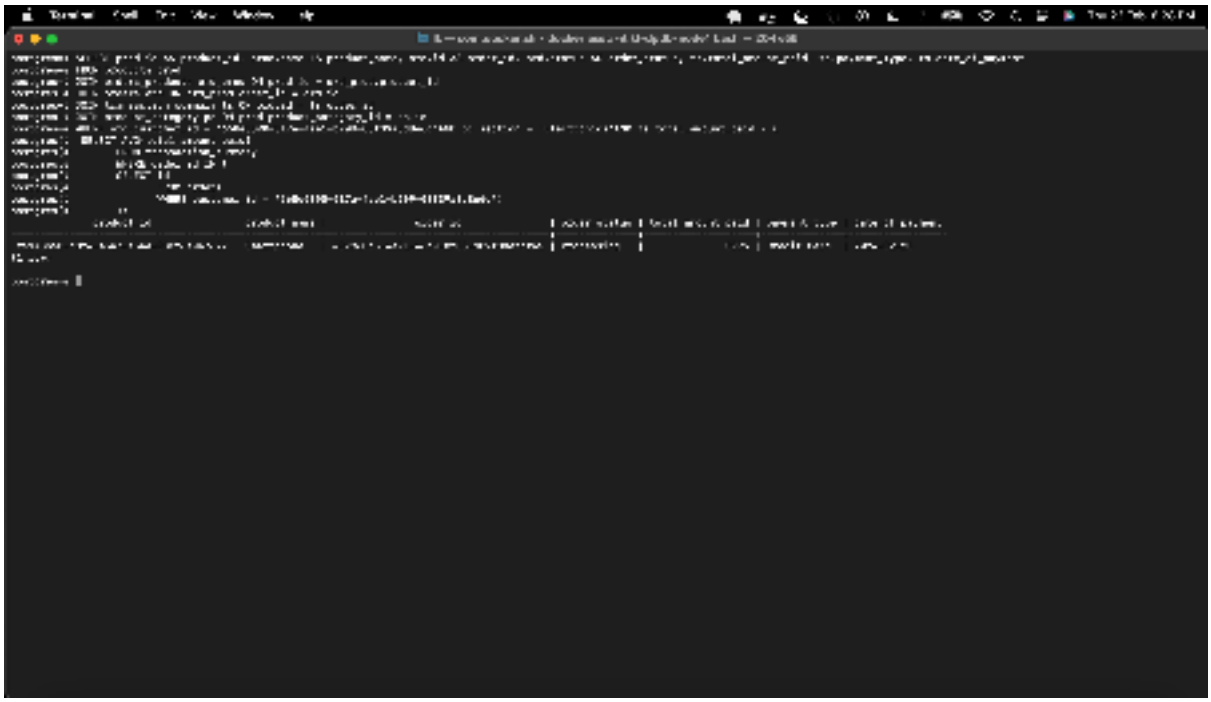


Fig-94: Get the products which has price more than the average of total products price ordered by the customer.

Explanation: customer placed multiple orders, but out of them the product which has the highest than the average price is “smartphone”. Using multiple tables such as products, orders, transaction summary, customers fetched the required details from the database. Without performing the parallel execution mechanism.

query:

```
EXPLAIN
SELECT prod.id AS product_id, prod.name AS product_name, ord.id AS order_id,
ord.status AS order_status, ts.total_amount_paid, ts.payment_type,
ts.date_of_payment
FROM products prod
JOIN orders_products ord_prod ON prod.id = ord_prod.product_id
JOIN orders ord ON ord_prod.order_id = ord.id
JOIN transaction_summary ts ON ord.id = ts.order_id
JOIN product_category pc ON prod.product_category_id = pc.id
WHERE ord.customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'AND pc.section =
'Electronics'AND ts.total_amount_paid > (
       SELECT AVG(total_amount_paid)
        FROM transaction_summary
        WHERE order_id IN (
              SELECT id
               FROM orders
               WHERE customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d')
       );
```

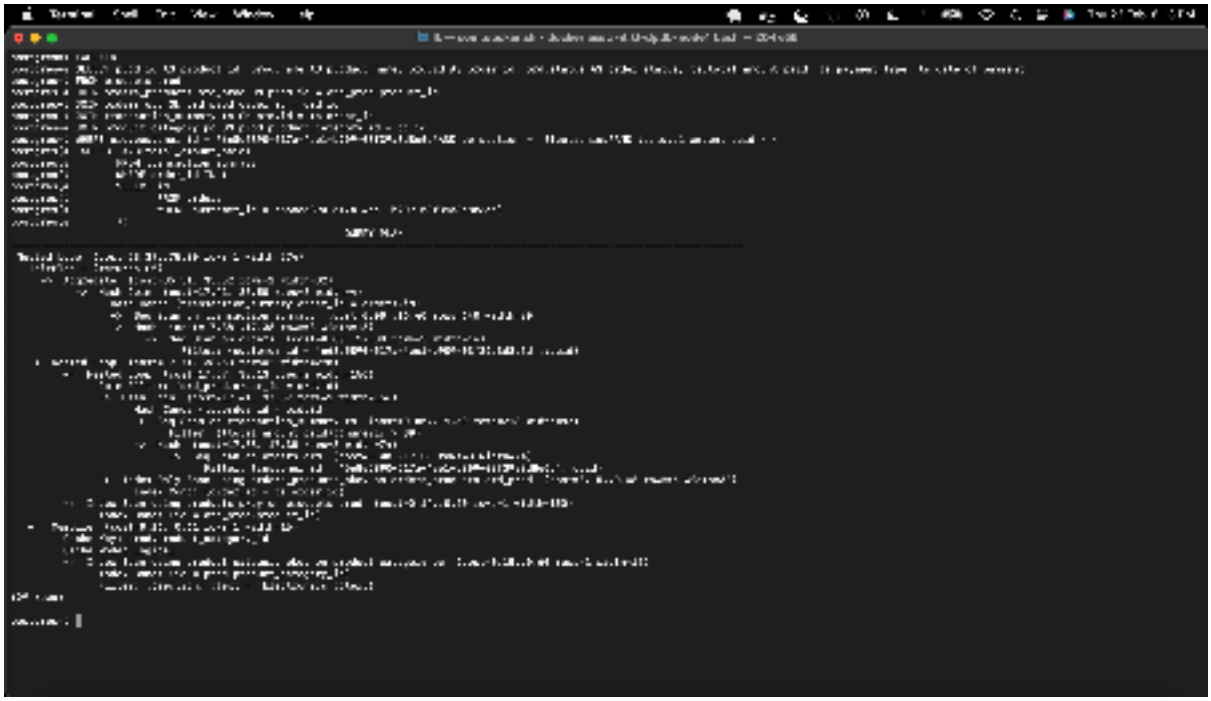


Fig-95: Analyzed the query performance.

Explanation: Without using parallel execution mechanism, analyzed the performance of the query. Observed that fetched the data using multiple nested loops, scanned sequentially to retrieve the required data. Applied filters, aggregation functions to retrieve the data from the database. Depending on the size of the tables, applied filters for better performance.

Validate Parallel execution of query:

query:

```
SET max_parallel_workers = 4;

EXPLAIN
SELECT prod.id AS product_id, prod.name AS product_name, ord.id AS order_id,
ord.status AS order_status,
ts.total_amount_paid, ts.payment_type, ts.date_of_payment
FROM products prod
JOIN orders_products ord_prod ON prod.id = ord_prod.product_id  -- Corrected alias
from 'ord' to 'ord_prod'
JOIN orders ord ON ord_prod.order_id = ord.id  -- Corrected alias from 'o' to 'ord'
JOIN transaction_summary ts ON ord.id = ts.order_id
JOIN product_category pc ON prod.product_category_id = pc.id
WHERE ord.customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'AND pc.section =
'Electronics'AND ts.total_amount_paid > (
                SELECT AVG(total_amount_paid)
                FROM transaction_summary
                WHERE order_id IN (
                    SELECT id
                    FROM orders
                    WHERE customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'
                )
            );
```

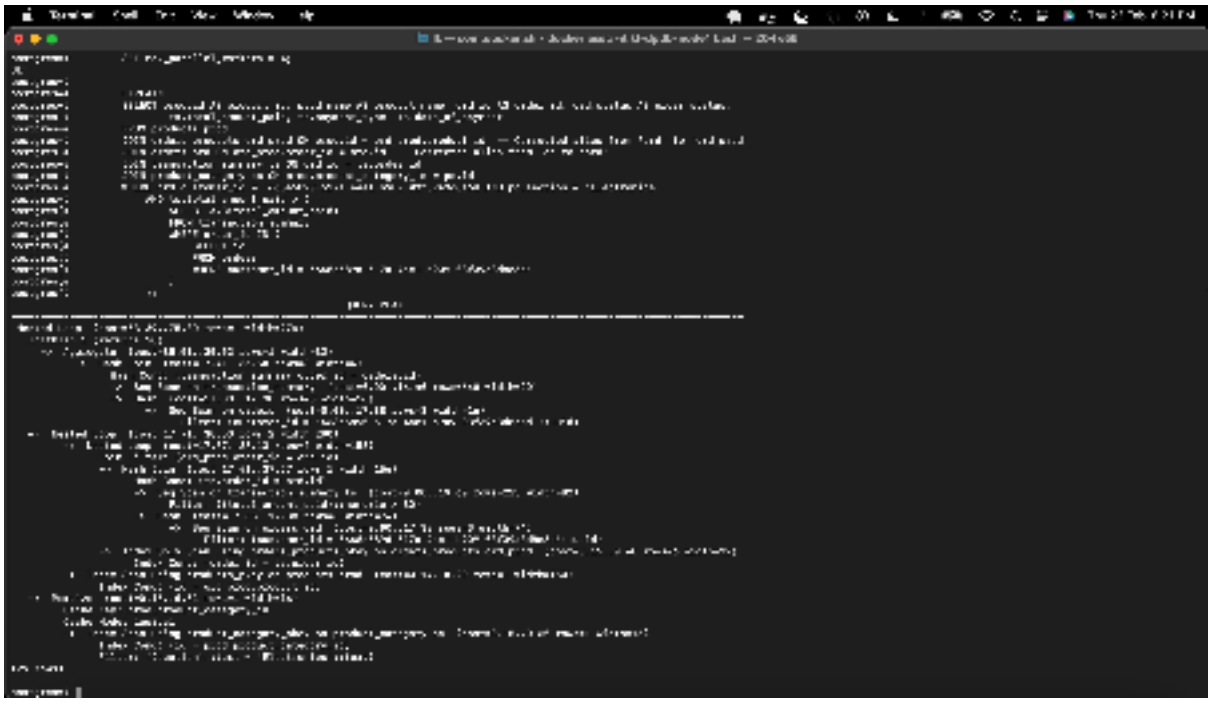


Fig-96: Analyzed and Executed query with 4 parallel worker nodes.

Explanation: Provide the max_parallel_worker nodes a 4. The postgres internally uses used the parallel worker nodes based on the requirement. Depending on the situation, the database uses the parallel worker nodes. The Execution of above query not effected by assigning the parallel worker nodes. Ideally the execution of parallel worker nodes depends

on various factors such as size of the data, query complexity, resources available and configurations. To execute this query the parallel query execution is not applicable.

Requirement-8:

use case 1: A customer can store multiple addresses and choose any one out it as default address. Customer can place order to any address as per choice. There also flexibility for the customer to update the address, incase customer adds the same existing address then database will through an error message “Address already exists. Please make use of it.”

code:

```
DO $$
DECLARE
    -- Input parameters
    input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
    input_flat_no INTEGER := 101;
    input_street VARCHAR(50) := 'Maple lane, Apartment 3C';
    input_city VARCHAR(50) := 'River City';
    input_state VARCHAR(50) := 'Texas';
    input_country VARCHAR(50) := 'USA';
    input_zip_code VARCHAR(10) := '75001';

    -- Variables for data
    get_address_id UUID;
    get_existing_address RECORD;

BEGIN
    BEGIN
        -- Insert data into the address table
        INSERT INTO address (flat_no, street, city, state, country, zip_code)
        VALUES (input_flat_no, input_street, input_city, input_state,
input_country, input_zip_code)
        RETURNING id INTO get_address_id;
        -- Insert data into the customer_address table
        INSERT INTO customer_address (customer_id, address_id, default_address)
        VALUES (input_customer_id, get_address_id, false);
        -- Check if any other address exists for the customer
        FOR get_existing_address IN
            SELECT addr.*
            FROM address addr
            JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
            WHERE cust_addr.customer_id = input_customer_id AND addr.id <>
get_address_id
        LOOP
            -- Validate with the input
            IF get_existing_address.flat_no = input_flat_no AND
               get_existing_address.street = input_street AND
               get_existing_address.city = input_city AND
               get_existing_address.state = input_state AND
               get_existing_address.country = input_country AND
```

```
                get_existing_address.zip_code = input_zip_code THEN
                 RAISE EXCEPTION 'Address already exists. Please make use of it.';
            END IF;
        END LOOP;

    EXCEPTION
        WHEN OTHERS THEN
            -- An error occurred, roll back the transaction
            RAISE NOTICE 'Triggered Error: %', SQLERRM;
            ROLLBACK;
            RETURN;
    END;

    -- Everything went well, commit the transaction
    COMMIT;
    RAISE NOTICE 'Address saved successfully';

END $$;
```

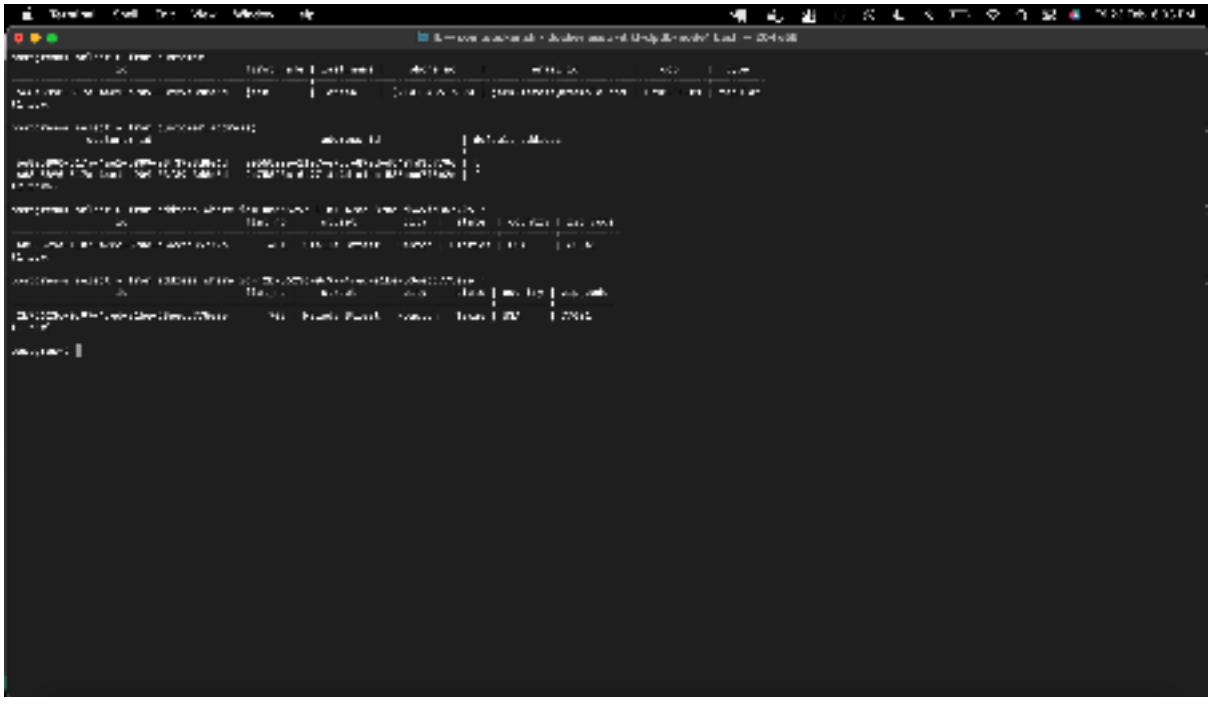


Fig-97: Display all the addresses for a customer.

Explanation: customer_id 668e5890-517a-4ae1-b909-03f39a3d8e6d has total 2 addresses with address id's a055529e-1107-47cc-890d-d549fd357796 and 2b75623c-0c97-4c4d-a1be-536ecc775e2e.
The address for a055529e-1107-47cc-890d-d549fd357796 has flat_no. '401', street 'Stella Street', city 'Denton', state 'Florida', country 'USA' and zip_code '65431'.
The address for 2b75623c-0c97-4c4d-a1be-536ecc775e2e has flat_no. '702', street 'Melody Street', city 'Houston', state 'Texas', country 'USA' and zip_code '77001'.

Now Let’s try to insert a new address to 668e5890-517a-4ae1-b909-03f39a3d8e6d

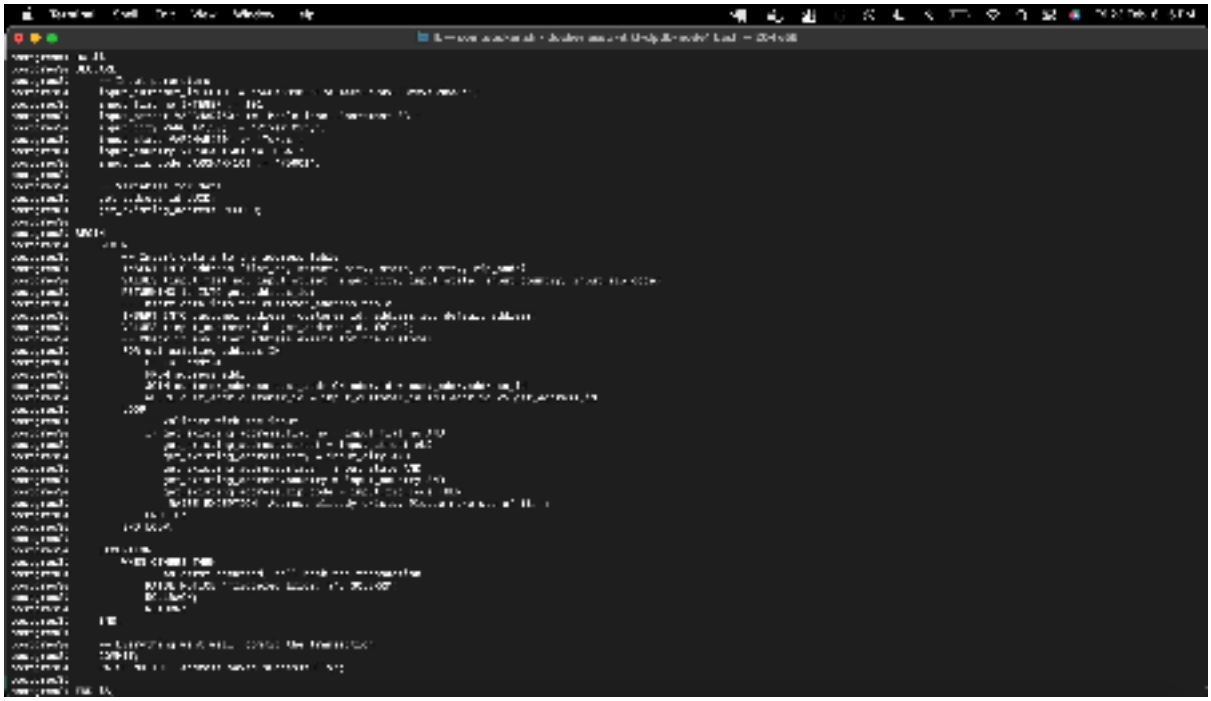


Fig-98: Code to add the Customer Address.

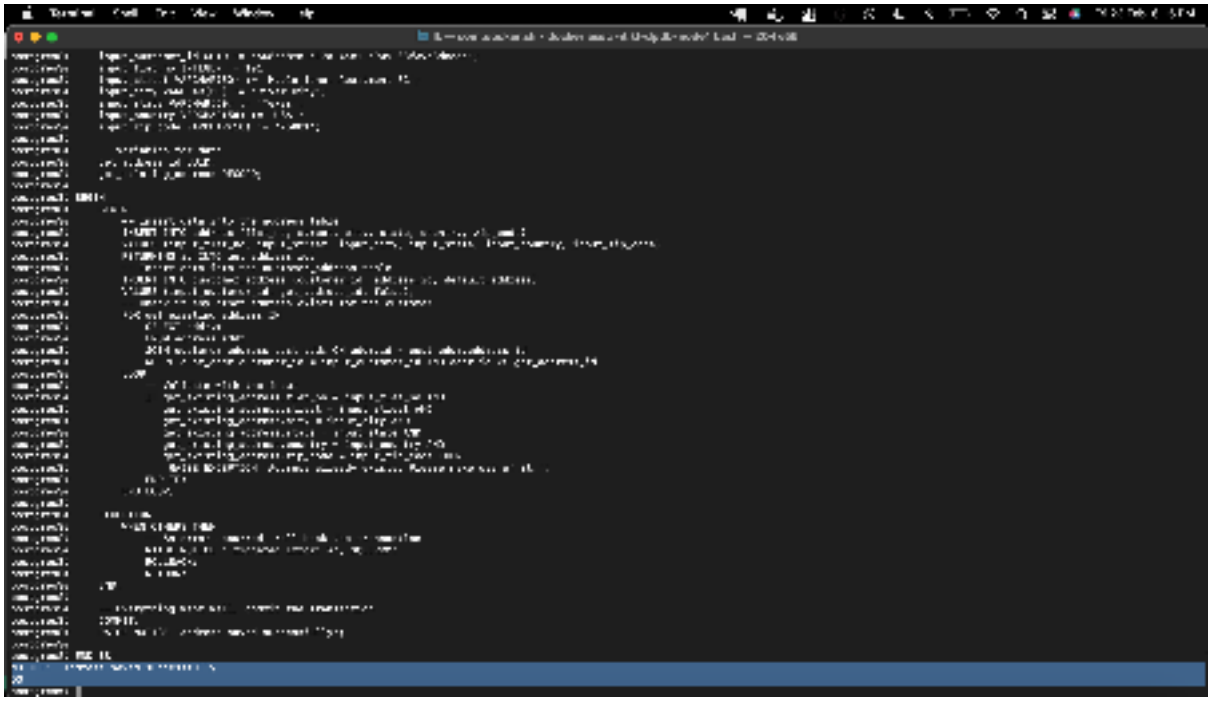


Fig-99: Executed code to add the Customer Address.

Explanation: As per the use case, customer inserted the new address which is not there in existing system associated to the customer. Observed that the address saved successfully into the database system.

Let’s see the saved address in the customer, customer_address and address tables.

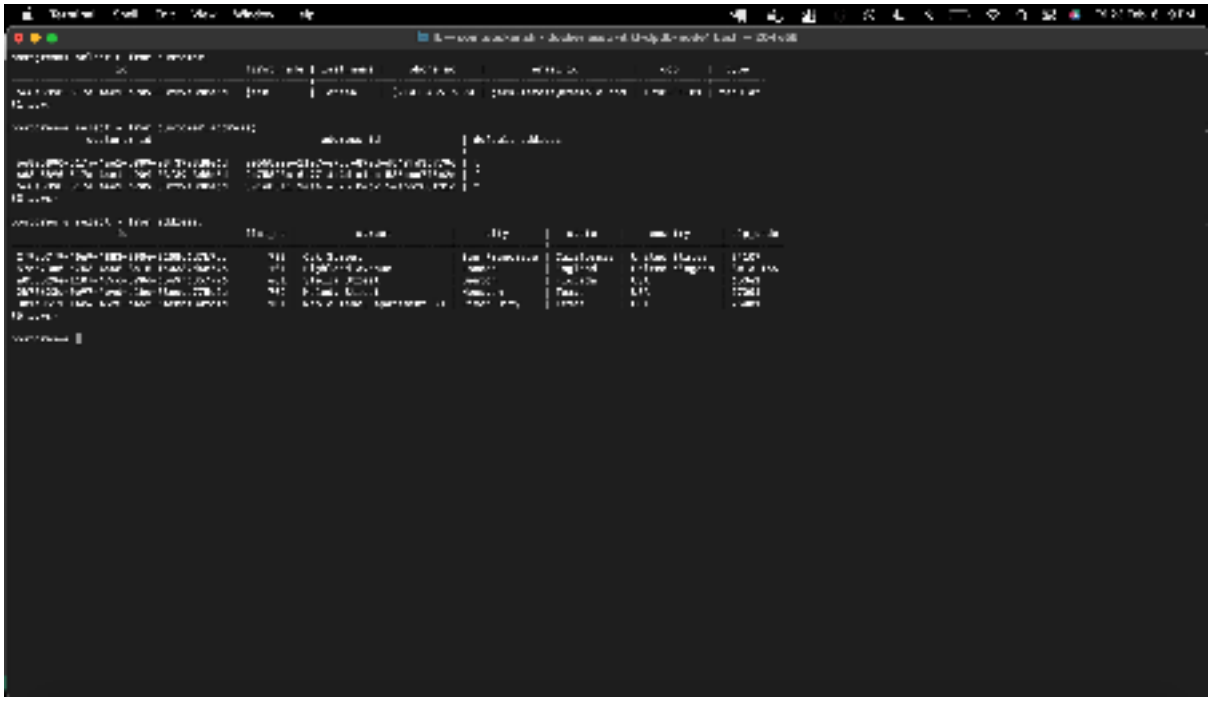


Fig-100: Displayed the latest stored address of the customer.

Explanation: once a new address is saved by the customer, we can observe that address is updated into all the required tables. For customer id '668e5890-517a-4ae1-b909-03f39a3d8e6d, a new address is added with flat_no – 101, street – Maple lane, Apartment 3C, city – River City, state – Texas, country- USA and zip-code 75001.

Now, Let’s add the same address, for the '668e5890-517a-4ae1-b909-03f39a3d8e6d’ and check the **error handling mechanism and rollback.**

code:

```
DO $$
DECLARE
    -- Input parameters
    input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
    input_flat_no INTEGER := 101;
    input_street VARCHAR(50) := 'Maple lane, Apartment 3C';
    input_city VARCHAR(50) := 'River City';
    input_state VARCHAR(50) := 'Texas';
    input_country VARCHAR(50) := 'USA';
    input_zip_code VARCHAR(10) := '75001';

    -- Variables for data
    get_address_id UUID;
    get_existing_address RECORD;

BEGIN
    -- Start the transaction
```

```
    BEGIN
        -- Insert data into the address table
        INSERT INTO address (flat_no, street, city, state, country, zip_code)
        VALUES (input_flat_no, input_street, input_city, input_state,
input_country, input_zip_code)
        RETURNING id INTO get_address_id;
        RAISE NOTICE 'Inserted into address table for address_id: %',
get_address_id;

        -- Insert data into the customer_address table
        INSERT INTO customer_address (customer_id, address_id, default_address)
        VALUES (input_customer_id, get_address_id, false);
        RAISE NOTICE 'Inserted into customer_address table for customer_id: %,
address_id: %', input_customer_id, get_address_id;

        -- Check if any other address exists for the customer
        FOR get_existing_address IN
            SELECT addr.*
            FROM address addr
            JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
            WHERE cust_addr.customer_id = input_customer_id AND addr.id <>
get_address_id
        LOOP
            -- Validate with the input
            IF get_existing_address.flat_no = input_flat_no AND
               get_existing_address.street = input_street AND
               get_existing_address.city = input_city AND
               get_existing_address.state = input_state AND
               get_existing_address.country = input_country AND
               get_existing_address.zip_code = input_zip_code THEN
                RAISE EXCEPTION 'Address already exists. Please make use of it.';
            END IF;
        END LOOP;

    EXCEPTION
        WHEN OTHERS THEN
            -- An error occurred, roll back the transaction
            RAISE NOTICE 'Error occurred: %', SQLERRM;

            -- Log the rollback
            RAISE NOTICE 'Error occurred, rolling back the data';
            ROLLBACK;
            RETURN;
    END;

    -- Everything went well, commit the transaction
    COMMIT;
    RAISE NOTICE 'Address saved successfully';

END $$;
```

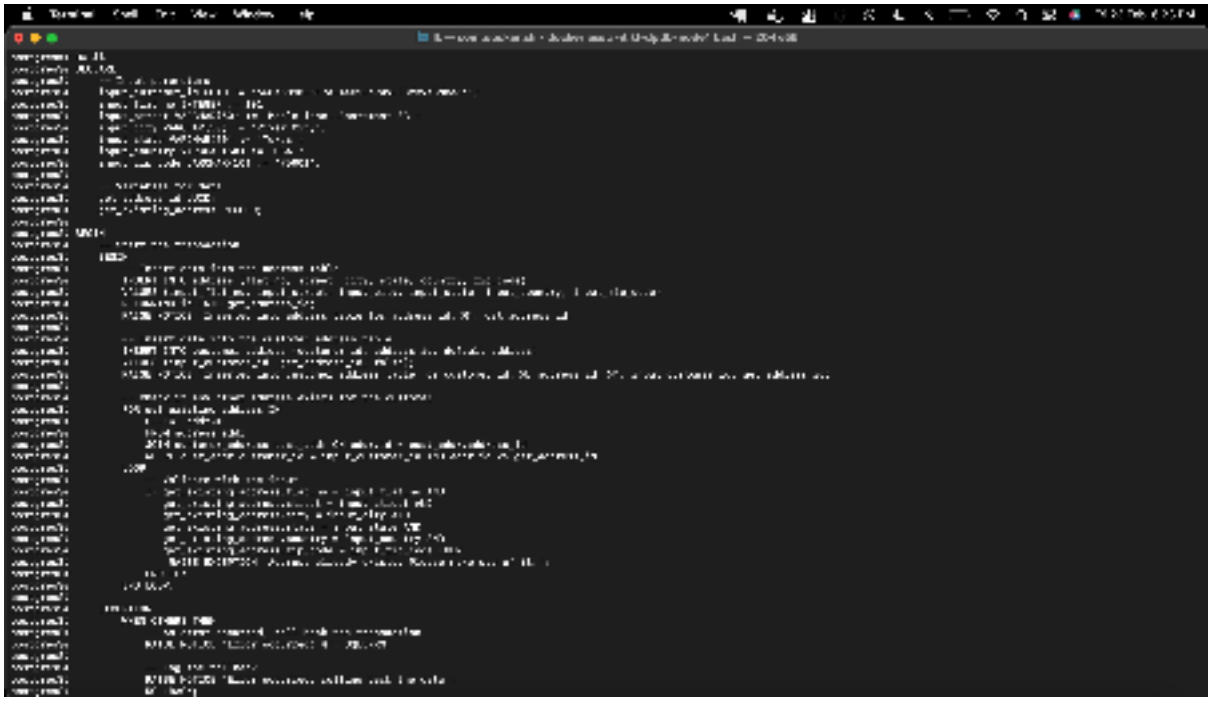


Fig-101: Stored the same address by the ‘'668e5890-517a-4ae1-b909-03f39a3d8e6d’.

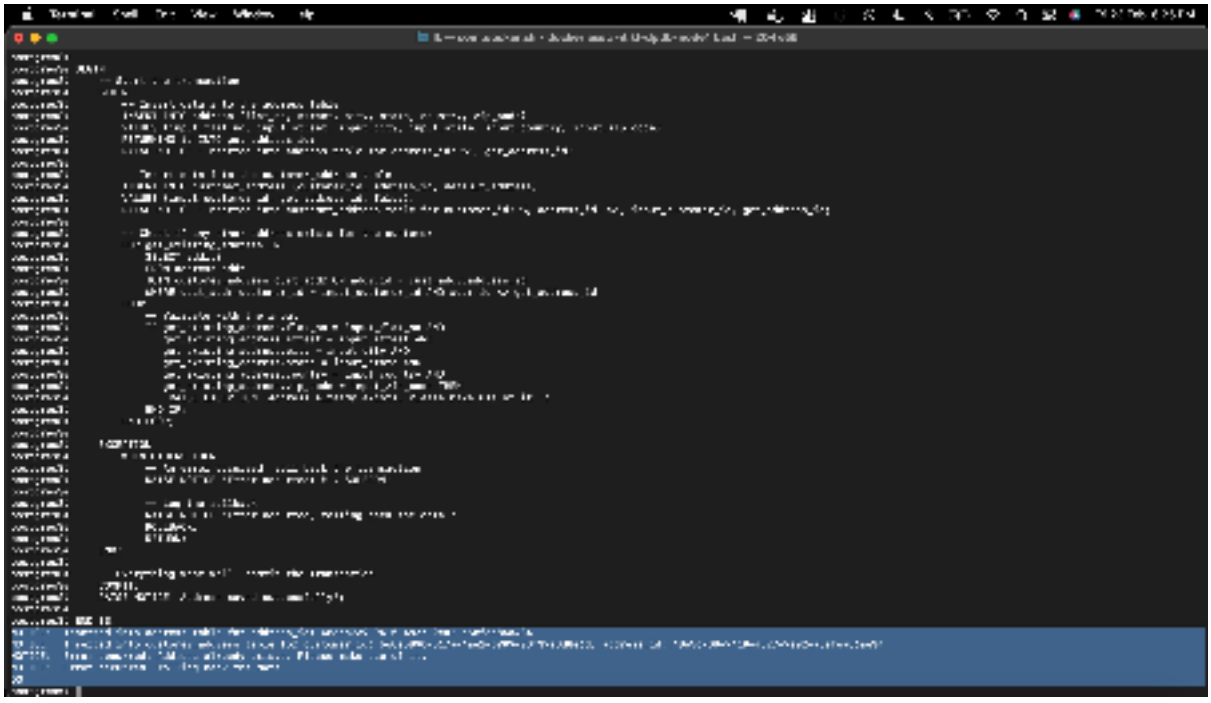


Fig-102: Display the error while saving the same address by the ‘'668e5890-517a-4ae1-b909-03f39a3d8e6d’.

Explanation: When user tried to add the same address again, while executing the function we can see that data inserted into address table, customer_address table and upon validating the inserted data, observed that the customer is trying to add the duplicate address into the database system So, as error occurred due to duplicate address, the data got roll backed from the database.

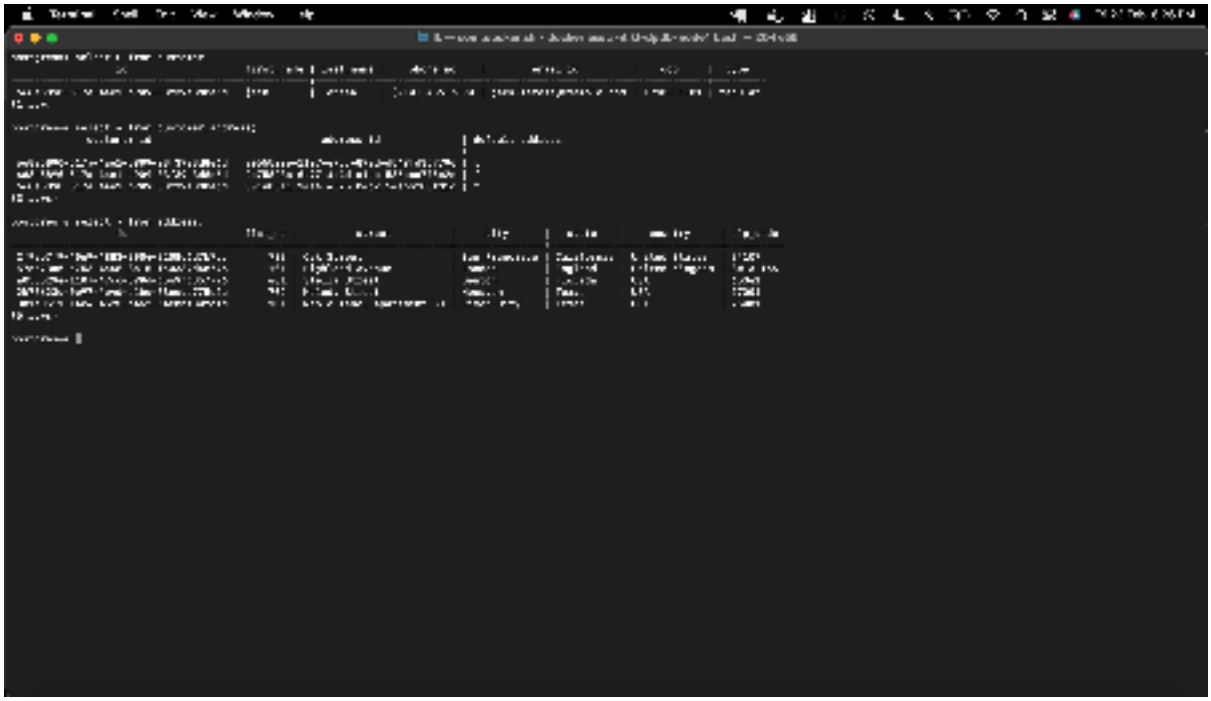


Fig-103: Insertions rolled back, due to error occurred while execution.

Explanation: In the above executed code, as error occurred during the transactions and rollback is performed, the inserted data is also rolled back in respective tables(inserted data before the error occur).

Usecase-2: A customer may use a credit card to make purchases up to Rs. 1,000,000 every month. He/She is unable to place an order using a credit card after their monthly credit card spending exceeds Rs. 1,00,000. He/She ought to employ different ways to pay.

Let's see the customer spendings on credit card:

query:

```
select * from customer;
select * from orders;
select * from transaction_summary;
```

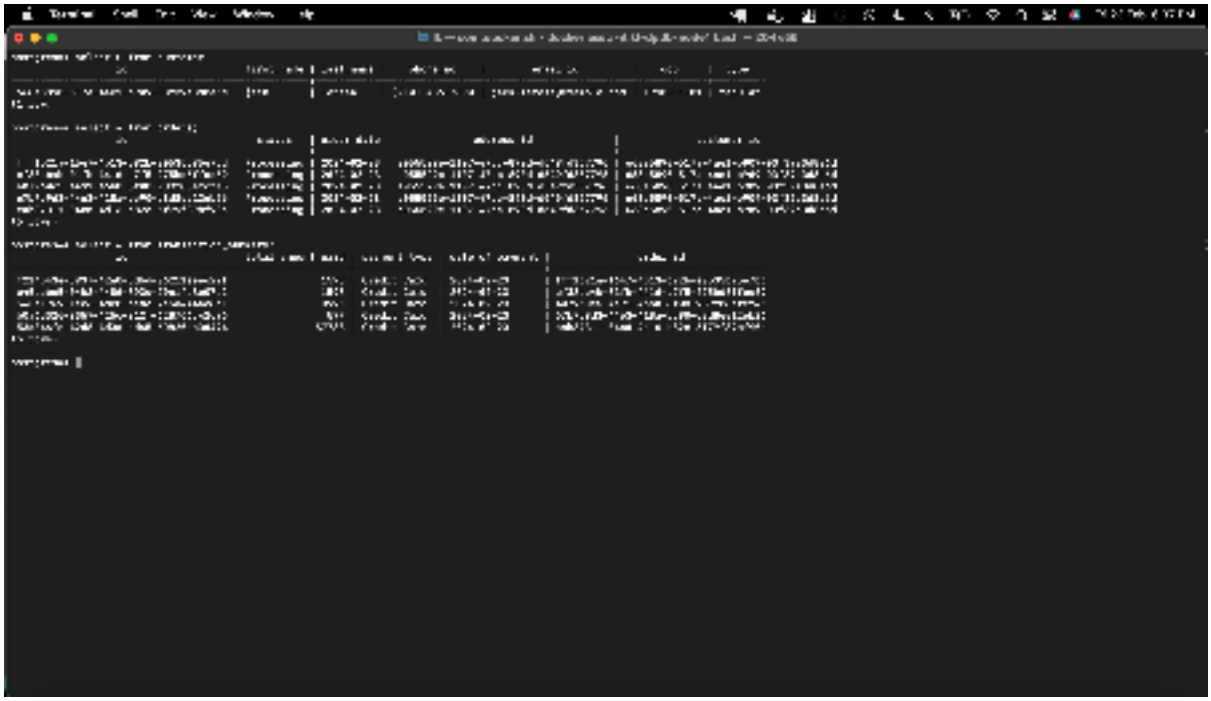


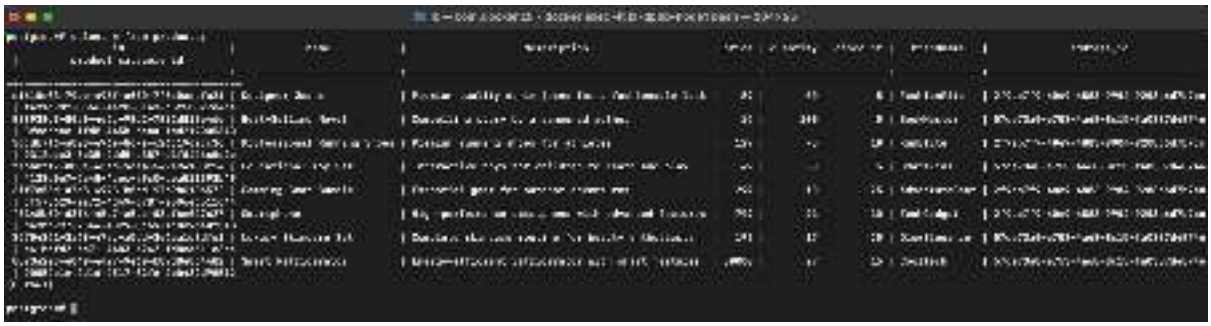
Fig-104: Customer '668e5890-517a-4ae1-b909-03f39a3d8e6d' orders and transaction_summary.

Explanation: From the above screenshot customer has "668e5890-517a-4ae1-b909-03f39a3d8e6d" has 4 orders with the total amount of 65,578 using credit card.

Now Let’s make the one more transaction for same user which exceeds 1,00,000.

query:

```
select * from products;
```



Explanation: In the above screen shot we can observe that that product id “**8893a2e3-e040-4427-9e60-b89d8e0b4482**” has price **50000**

code:

```
DO $$
DECLARE
    -- Input parameters
    input_payment_type VARCHAR(20) := 'Credit Card';
    input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
    input_product_id UUID := '8893a2e3-e040-4427-9e60-b89d8e0b4482';
    input_product_quantity INTEGER := 1;
    input_delivery_partner_name VARCHAR(50) := 'Fedx';
    input_delivery_partner_phone_no VARCHAR(20) := '(980) 426-7190';
    input_delivery_partner_email VARCHAR(255) := 'Fedx@example.com';

    -- Variables for data
    v_total_amount_paid INTEGER;
    v_credit_limit INTEGER := 100000;
    v_remaining_credit INTEGER;
    v_order_amount INTEGER;
    v_order_id UUID;
    v_delivery_partner_id UUID;

BEGIN
    -- Start the transaction
    BEGIN
        -- Step 1: Check if the product and quantity are available
        PERFORM 1 FROM products WHERE id = input_product_id AND quantity >=
input_product_quantity;
        IF NOT FOUND THEN
            RAISE EXCEPTION 'Sorry, the requested product is not available. Please
try again later.';
        END IF;

        -- Step 2: Insert data into orders
        INSERT INTO orders (status, order_date, address_id, customer_id)
```

```
        VALUES ('Processing', CURRENT_DATE,
                (SELECT address_id FROM customer_address WHERE customer_id =
input_customer_id AND default_address = true),
                input_customer_id)
        RETURNING id INTO v_order_id;
        RAISE NOTICE 'Inserted into orders table for order_id: %', v_order_id;

        -- Step 3: Calculate order amount (replace this with your own logic)
        v_order_amount := input_product_quantity * (SELECT price FROM products
WHERE id = input_product_id);

        -- Step 4: Check if the customer has enough credit limit
        SELECT COALESCE(SUM(total_amount_paid), 0) INTO v_total_amount_paid
        FROM transaction_summary
        WHERE order_id IN (SELECT id FROM orders WHERE customer_id =
input_customer_id) AND payment_type = input_payment_type;

        v_remaining_credit := v_credit_limit - v_total_amount_paid -
v_order_amount;

        IF v_remaining_credit >= 0 THEN
            -- Update transaction_summary only if all conditions passed
            INSERT INTO transaction_summary (total_amount_paid, payment_type,
date_of_payment, order_id)
            VALUES ((SELECT price * input_product_quantity FROM products WHERE id =
input_product_id), input_payment_type, CURRENT_DATE, v_order_id);
            RAISE NOTICE 'Inserted into transaction_summary table for customer_id:
%', input_customer_id;
        ELSE
            RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use
another payment method.', input_customer_id;
        END IF;

        -- Step 5: Insert data into delivery_partner
        INSERT INTO delivery_partner (name, phone_no, email, order_id)
        VALUES (input_delivery_partner_name, input_delivery_partner_phone_no,
input_delivery_partner_email, v_order_id)
        RETURNING id INTO v_delivery_partner_id;
        RAISE NOTICE 'Inserted into delivery_partner table for delivery_partner_id:
%', v_delivery_partner_id;

        -- Step 6: Insert data into customer_delivery_partner
        INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
        VALUES (input_customer_id, v_delivery_partner_id);
        RAISE NOTICE 'Inserted into customer_delivery_partner table';

        -- Step 7: Insert data into orders_products
        INSERT INTO orders_products (order_id, product_id, quantity)
        VALUES (v_order_id, input_product_id, input_product_quantity);
        RAISE NOTICE 'Inserted into orders_products table';
```

```
        -- Step 8: Update product quantity
        UPDATE products SET quantity = quantity - input_product_quantity WHERE id =
input_product_id;
        RAISE NOTICE 'Updated product quantity for product_id: %',
input_product_id;

    EXCEPTION
        WHEN OTHERS THEN
            -- An error occurred, roll back the transaction
            RAISE NOTICE 'Error occurred: %', SQLERRM;

            -- Log the rollback
            RAISE NOTICE 'Error occurred, rolling back the data';

            ROLLBACK;
            RETURN;
    END;

    -- Everything went well, commit the transaction
    COMMIT;
    RAISE NOTICE 'Order placed successfully';

END $$;
```

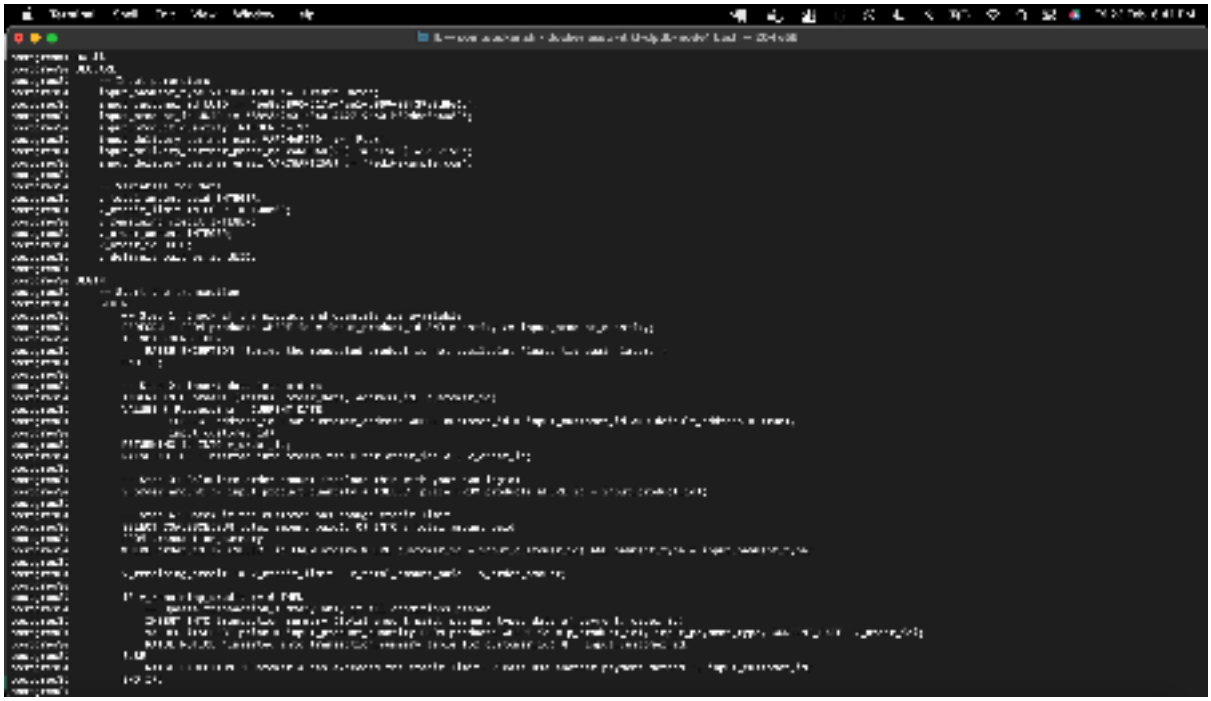


Fig-105: Execute query to place order using credit card.

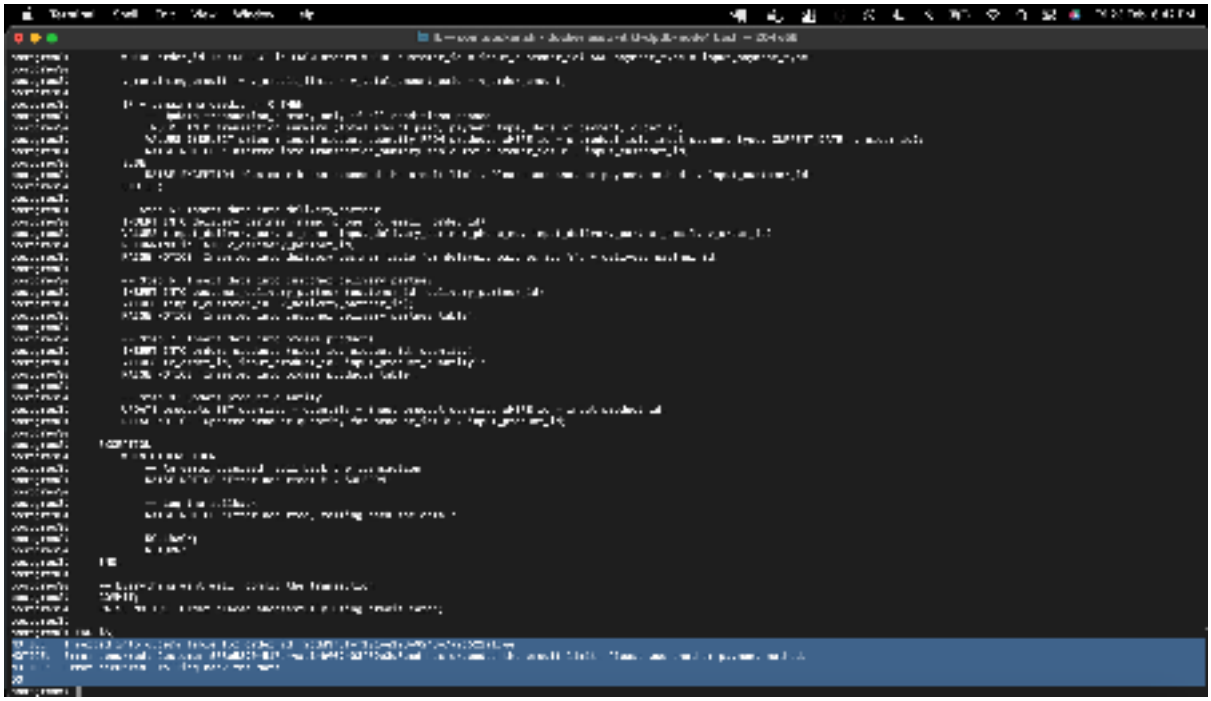


Fig-106: unable to place the order with credit card, limit is exceeded using credit.

Explanation: Order is placed by customer id '668e5890-517a-4ae1-b909-03f39a3d8e6d' for product id ''8893a2e3-e040-4427-9e60-b89d8e0b4482' which has price more than 50,000, but the customer was unable to place the order due to credit card limit check. The inserted data while performing the **transactions also rolled back** due to the error.

Now, Let's do the payment using Debit Card.

code:

```
DO $$
DECLARE
    -- Input parameters
    input_payment_type VARCHAR(20) := 'Debit Card';
    input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
    input_product_id UUID := '8893a2e3-e040-4427-9e60-b89d8e0b4482';
    input_product_quantity INTEGER := 1;
    input_delivery_partner_name VARCHAR(50) := 'Fedx';
    input_delivery_partner_phone_no VARCHAR(20) := '(980) 426-7190';
    input_delivery_partner_email VARCHAR(255) := 'Fedx@example.com';

    -- Variables for data
    v_total_amount_paid INTEGER;
    v_credit_limit INTEGER := 100000;
    v_remaining_credit INTEGER;
    v_order_amount INTEGER;
    v_order_id UUID;
    v_delivery_partner_id UUID;

BEGIN
    -- Start the transaction
```

```
    BEGIN
        -- Step 1: Check if the product and quantity are available
        PERFORM 1 FROM products WHERE id = input_product_id AND quantity >=
input_product_quantity;
        IF NOT FOUND THEN
            RAISE EXCEPTION 'Sorry, the requested product is not available. Please
try again later.';
        END IF;

        -- Step 2: Insert data into orders
        INSERT INTO orders (status, order_date, address_id, customer_id)
        VALUES ('Processing', CURRENT_DATE,
                (SELECT address_id FROM customer_address WHERE customer_id =
input_customer_id AND default_address = true),
                input_customer_id)
        RETURNING id INTO v_order_id;
        RAISE NOTICE 'Inserted into orders table for order_id: %', v_order_id;

        -- Step 3: Calculate order amount (replace this with your own logic)
        v_order_amount := input_product_quantity * (SELECT price FROM products
WHERE id = input_product_id);

        -- Step 4: Check if the customer has enough credit limit
        SELECT COALESCE(SUM(total_amount_paid), 0) INTO v_total_amount_paid
        FROM transaction_summary
        WHERE order_id IN (SELECT id FROM orders WHERE customer_id =
input_customer_id) AND payment_type = input_payment_type;

        v_remaining_credit := v_credit_limit - v_total_amount_paid -
v_order_amount;

        IF v_remaining_credit >= 0 THEN
            -- Update transaction_summary only if all conditions passed
            INSERT INTO transaction_summary (total_amount_paid, payment_type,
date_of_payment, order_id)
            VALUES ((SELECT price * input_product_quantity FROM products WHERE id =
input_product_id), input_payment_type, CURRENT_DATE, v_order_id);
            RAISE NOTICE 'Inserted into transaction_summary table for customer_id:
%', input_customer_id;
        ELSE
            RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use
another payment method.', input_customer_id;
        END IF;

        -- Step 5: Insert data into delivery_partner
        INSERT INTO delivery_partner (name, phone_no, email, order_id)
        VALUES (input_delivery_partner_name, input_delivery_partner_phone_no,
input_delivery_partner_email, v_order_id)
        RETURNING id INTO v_delivery_partner_id;
        RAISE NOTICE 'Inserted into delivery_partner table for delivery_partner_id:
%', v_delivery_partner_id;
```

```
        -- Step 6: Insert data into customer_delivery_partner
        INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
        VALUES (input_customer_id, v_delivery_partner_id);
        RAISE NOTICE 'Inserted into customer_delivery_partner table';

        -- Step 7: Insert data into orders_products
        INSERT INTO orders_products (order_id, product_id, quantity)
        VALUES (v_order_id, input_product_id, input_product_quantity);
        RAISE NOTICE 'Inserted into orders_products table';

        -- Step 8: Update product quantity
        UPDATE products SET quantity = quantity - input_product_quantity WHERE id =
input_product_id;
        RAISE NOTICE 'Updated product quantity for product_id: %',
input_product_id;

    EXCEPTION
        WHEN OTHERS THEN
            -- An error occurred, roll back the transaction
            RAISE NOTICE 'Error occurred: %', SQLERRM;

            -- Log the rollback
            RAISE NOTICE 'Error occurred, rolling back the data';

            ROLLBACK;
            RETURN;
    END;

    -- Everything went well, commit the transaction
    COMMIT;
    RAISE NOTICE 'Order placed successfully';

END $$;
```

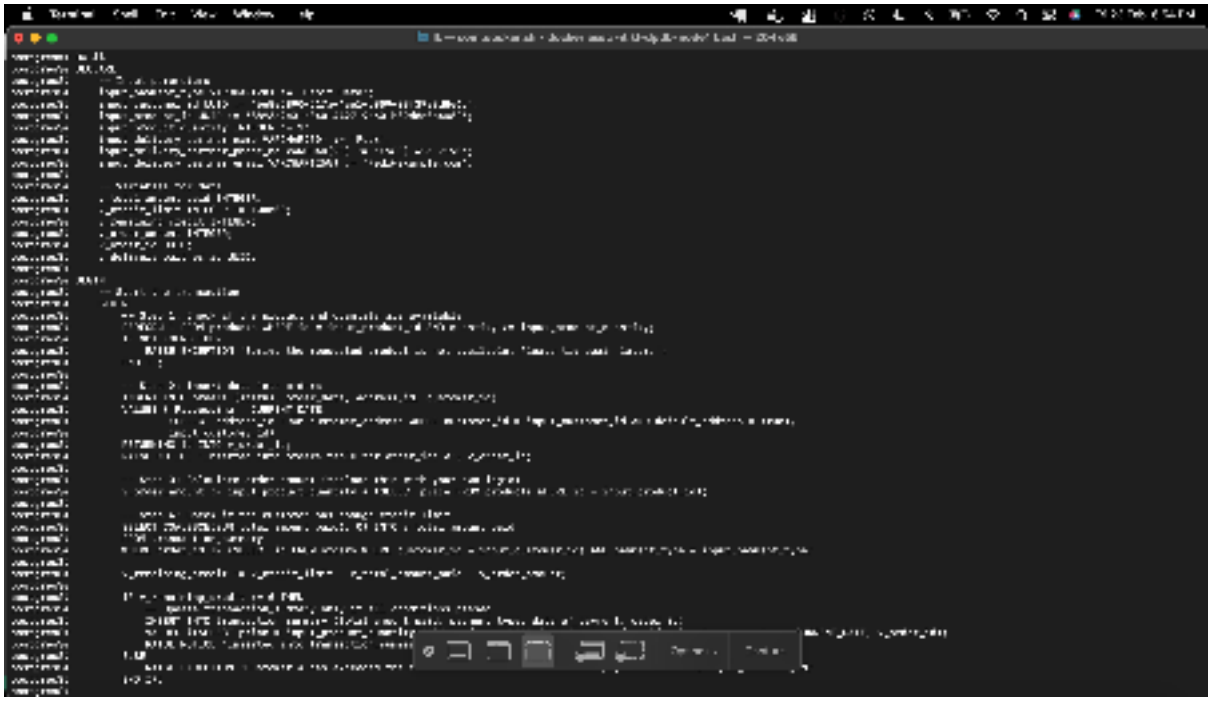


Fig-107: place an order using debit card.

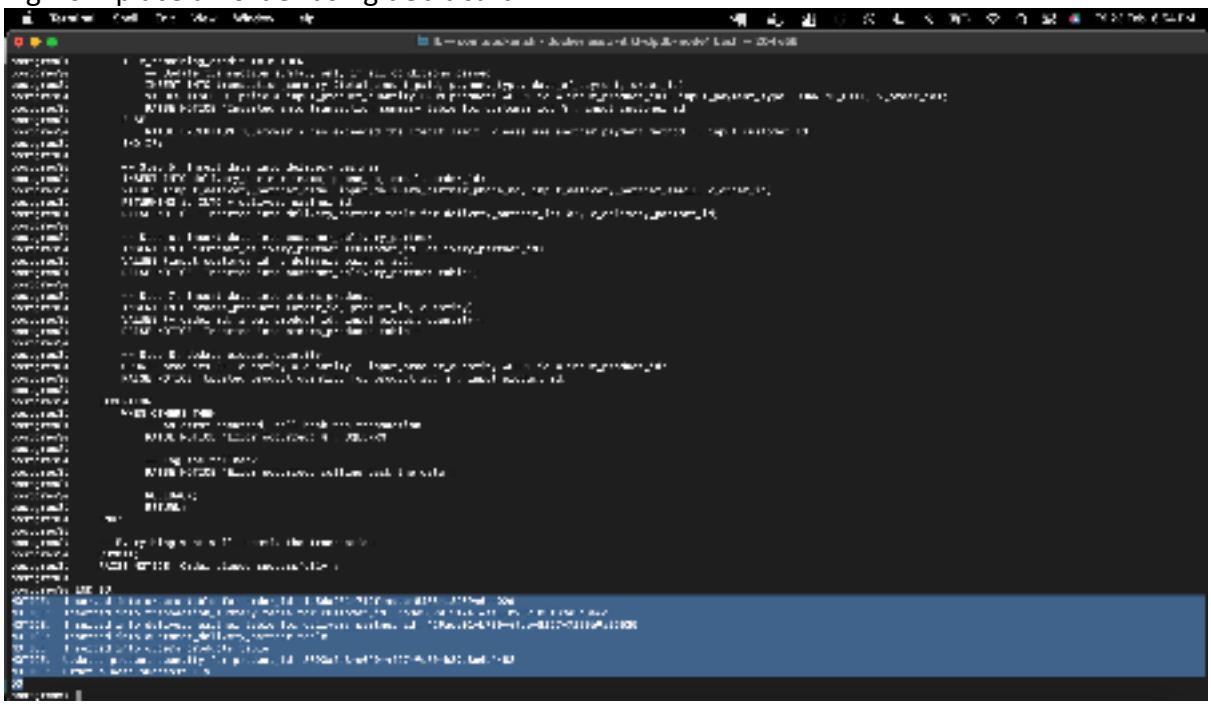


Fig-108: order placed successfully using debit card.

Explanation: Order placed successfully using the debit card. Observed that data is inserted into all the corresponding tables such as orders, transaction_summary, delivery_partner, customer_delivery partner, orders_products tables and update the products quantity in the products table.

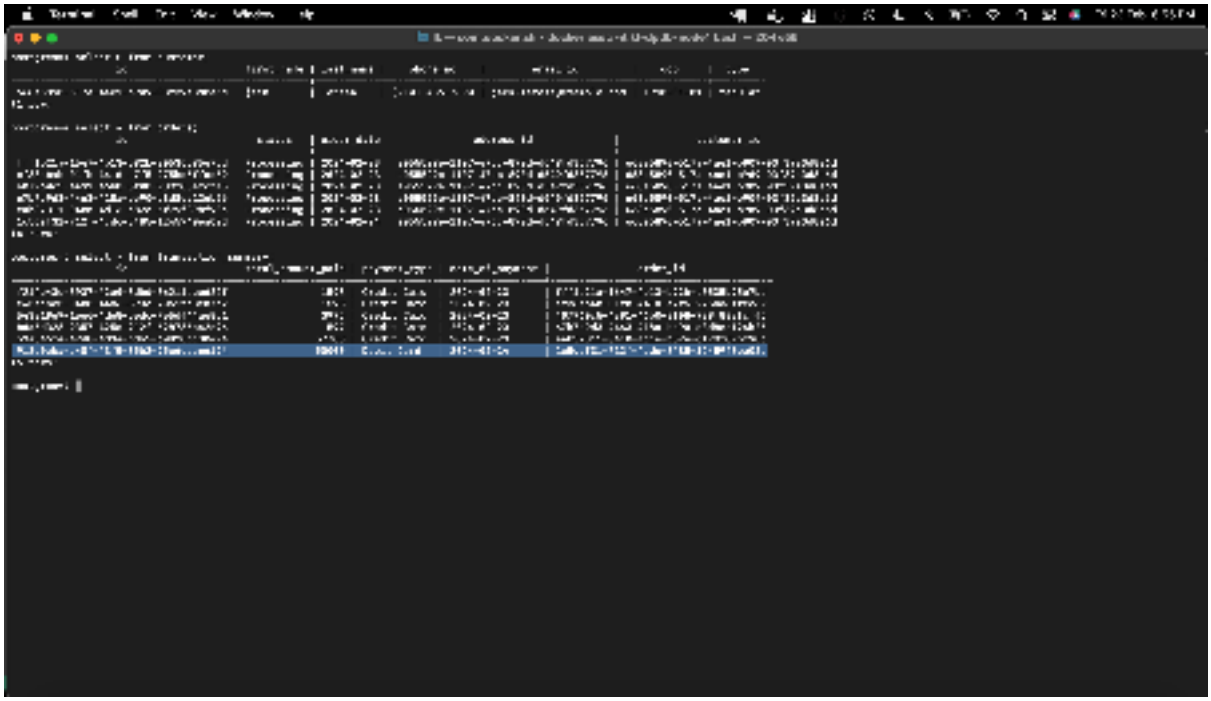


Fig-109: order placed successfully and displayed the data in required tables.

Explanation: order details got updated in all the corresponding tables, observed transaction processed successfully using debit card.

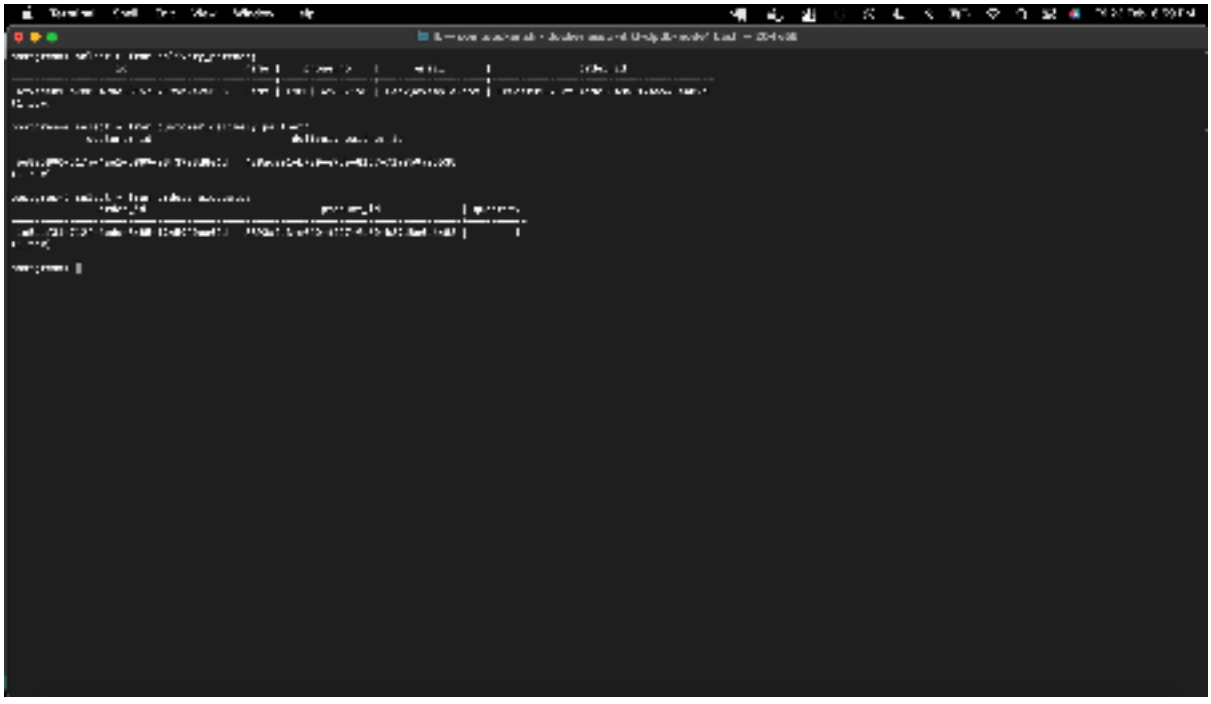


Fig-110: displayed the table in all related tables.

Explanation: Data updated in all the tables which are associated with order placement with the customer.