

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;"`

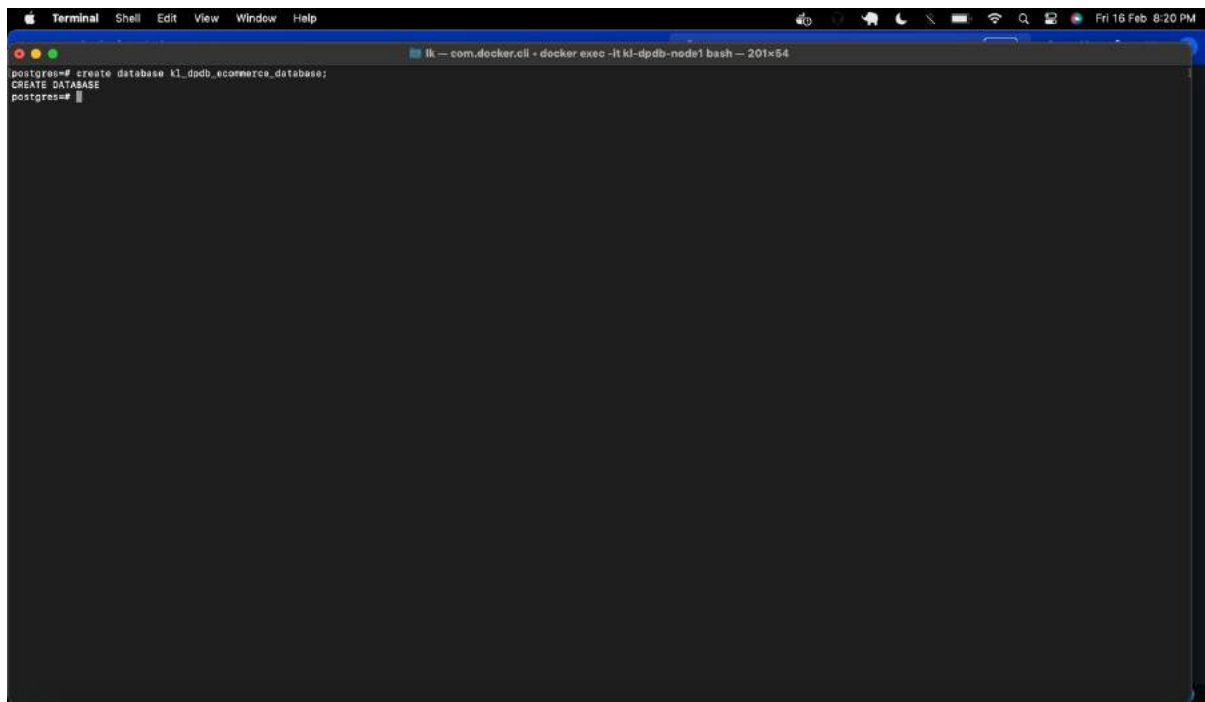
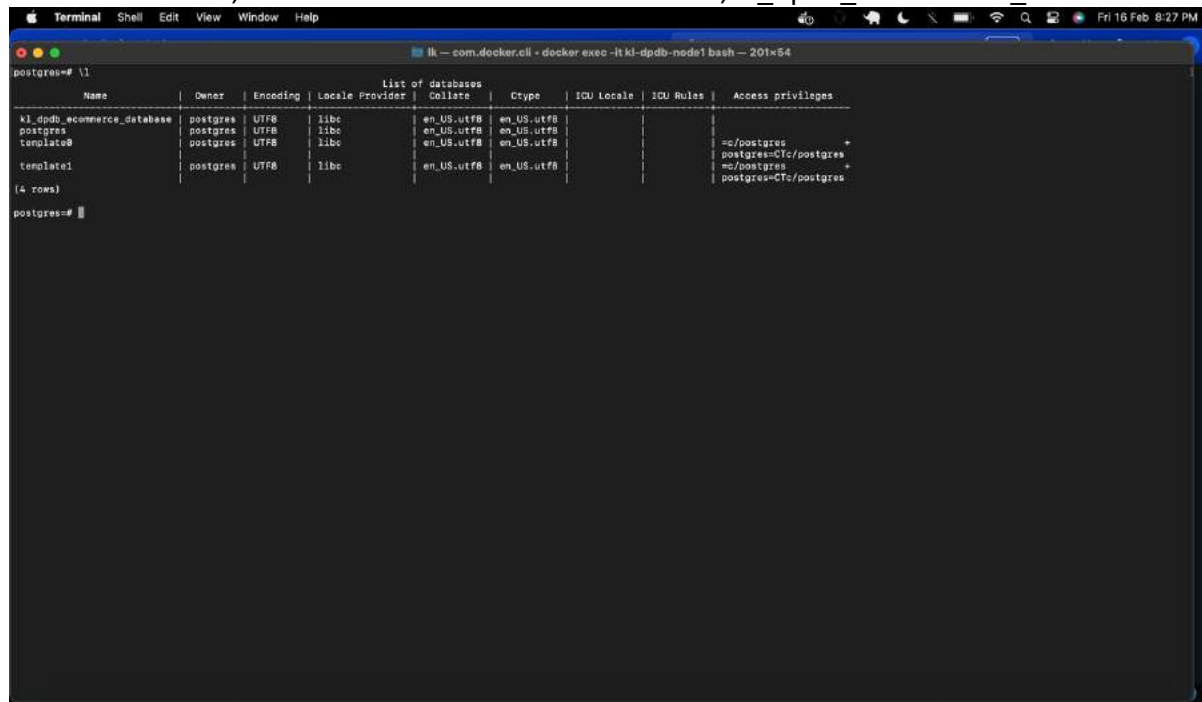


Fig-8: Create 'kl_dpdb_ecommerce_database' database in postgres.

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

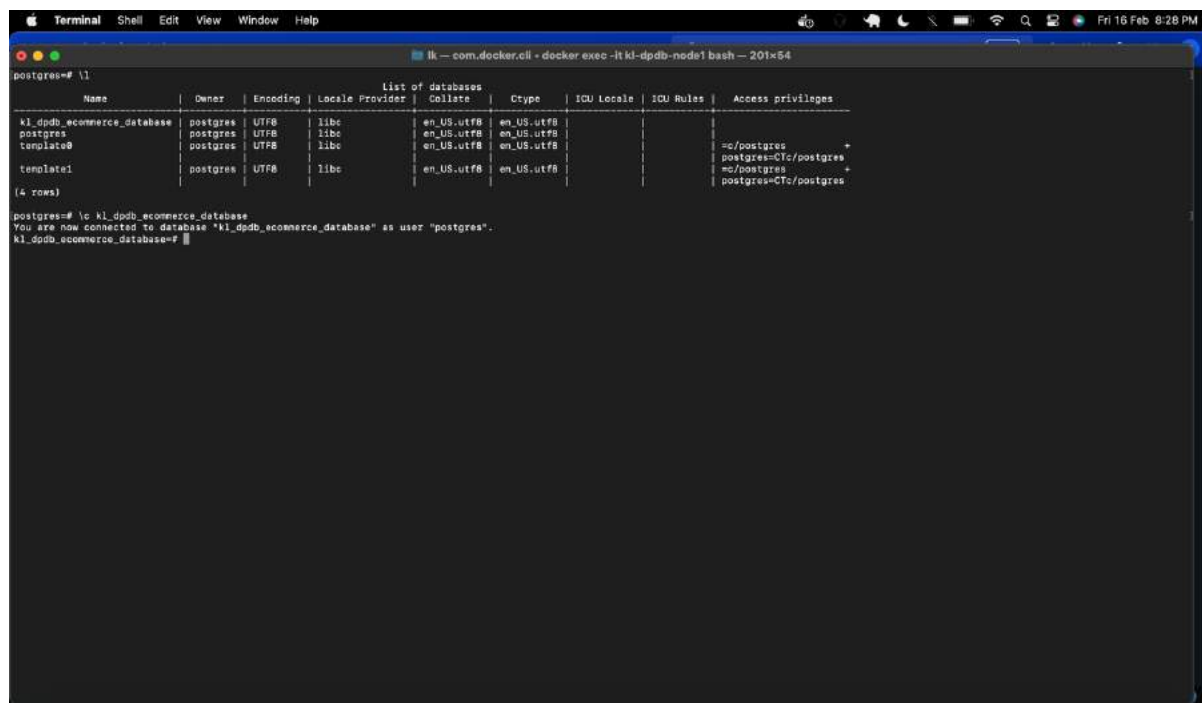


```
postgres=# \l
          list of databases
  Name          | Owner   | Encoding | Locale Provider | Collate | Ctype    | ICU Locale | ICU Rules | Access privileges
-----+-----+-----+-----+-----+-----+-----+-----+-----
kl_dpdb_ecommerce_database | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
postgres          | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
template0         | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
template1         | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
(4 rows)

postgres=#
```

Fig-9: Check the list of databases.

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



```
postgres=# \l
          list of databases
  Name          | Owner   | Encoding | Locale Provider | Collate | Ctype    | ICU Locale | ICU Rules | Access privileges
-----+-----+-----+-----+-----+-----+-----+-----+-----
kl_dpdb_ecommerce_database | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
postgres          | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
template0         | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
template1         | postgres | UTF8     | libc            | en_US.utf8 | en_US.utf8 |             |           |
(4 rows)

postgres=# \c kl_dpdb_ecommerce_database
You are now connected to database "kl_dpdb_ecommerce_database" as user "postgres".
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:

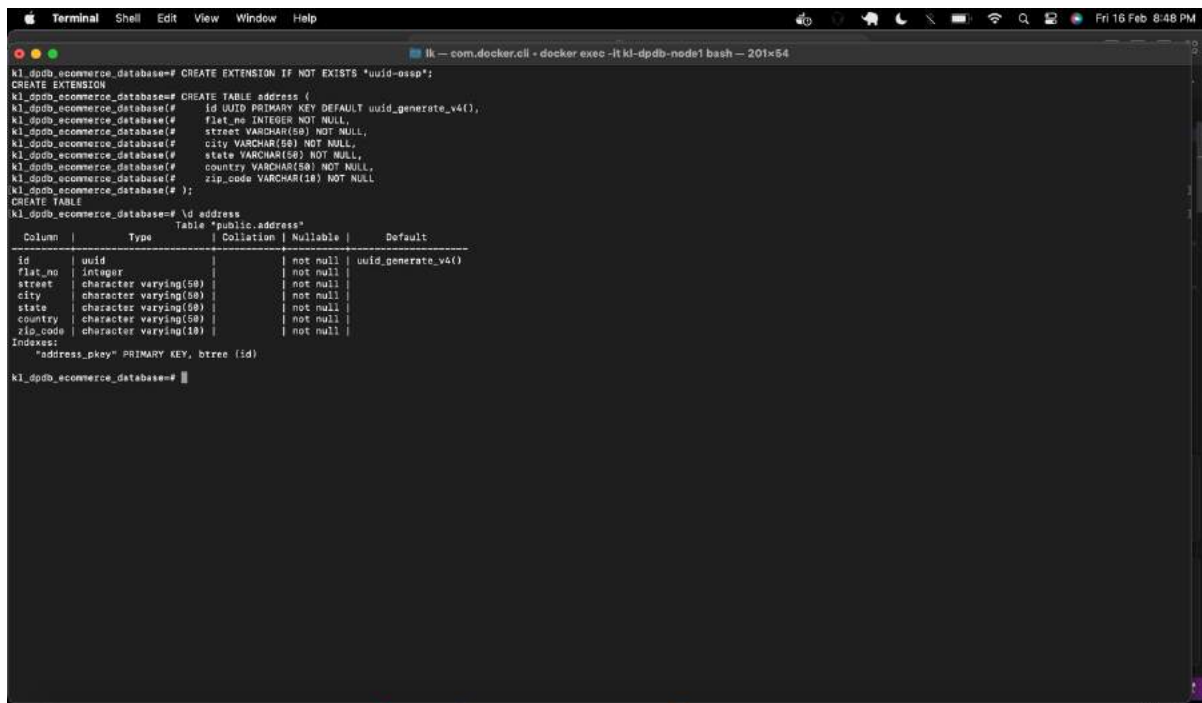
```
-- uuid-oss: the extension used to create primary key with UUID.  
CREATE EXTENSION IF NOT EXISTS "uuid-oss";
```

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  
);
```



```
kl_dpdb_ecommerce_database=# CREATE EXTENSION IF NOT EXISTS "uuid-oss";  
CREATE EXTENSION  
kl_dpdb_ecommerce_database=# CREATE TABLE address (  
kl_dpdb_ecommerce_database=# id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),  
kl_dpdb_ecommerce_database=# flat_no INTEGER NOT NULL,  
kl_dpdb_ecommerce_database=# street VARCHAR(50) NOT NULL,  
kl_dpdb_ecommerce_database=# city VARCHAR(50) NOT NULL,  
kl_dpdb_ecommerce_database=# state VARCHAR(50) NOT NULL,  
kl_dpdb_ecommerce_database=# country VARCHAR(50) NOT NULL,  
kl_dpdb_ecommerce_database=# zip_code VARCHAR(10) NOT NULL  
kl_dpdb_ecommerce_database=# );  
CREATE TABLE  
kl_dpdb_ecommerce_database=# \d address  
Table "public.address"  
+-----+-----+-----+-----+-----+  
Column | Type | Collation | Nullable | Default |  
+-----+-----+-----+-----+-----+  
id      | uuid |           | not null | uuid_generate_v4() |  
flat_no | integer |           | not null |                   |  
street  | character varying(50) |           | not null |                   |  
city    | character varying(50) |           | not null |                   |  
state   | character varying(50) |           | not null |                   |  
country | character varying(50) |           | not null |                   |  
zip_code | character varying(10) |           | not null |                   |  
+-----+-----+-----+-----+-----+  
Indexes:  
"address_pkey" PRIMARY KEY, btree (id)  
kl_dpdb_ecommerce_database=#
```

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

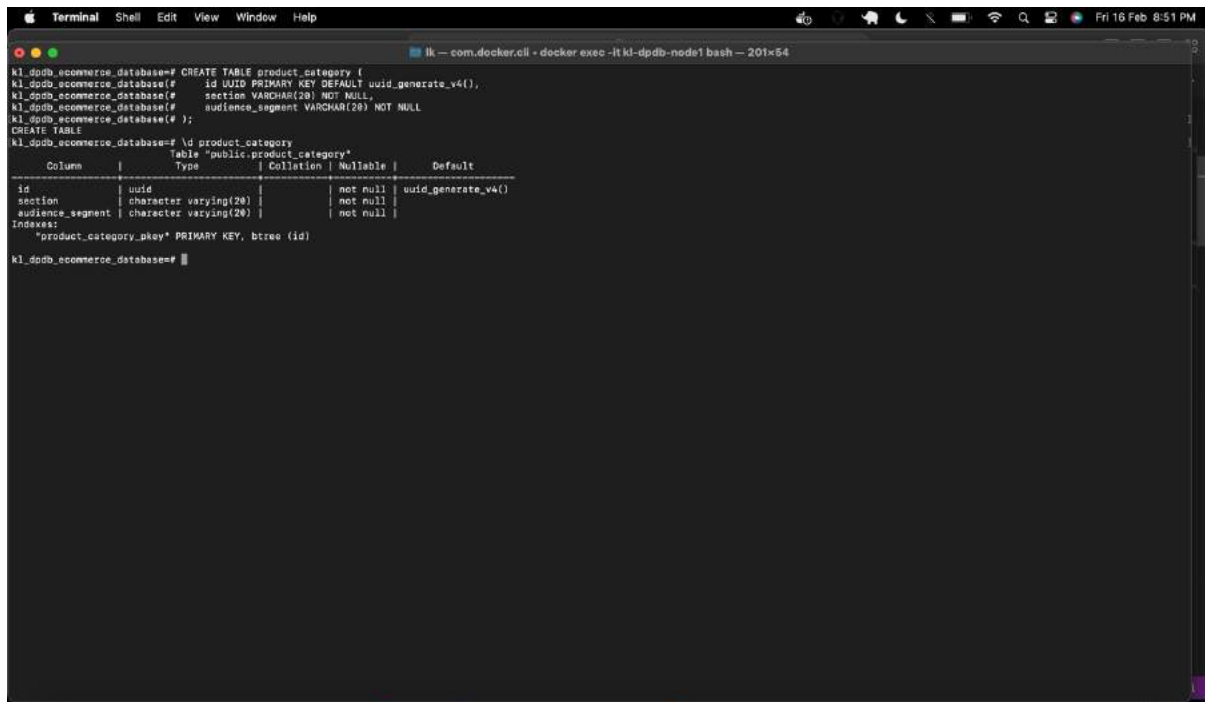
Explanation: The Address table is used to store the address details of customers, products, and supplier. The address table has a unique identifier with the attribute flat_no, street, city, state, country, and the zip-code. The address table is one of the crucial tables in the e-commerce site which helps 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
);
```

A terminal window titled "Terminal" with a dark background. The prompt is "kl_dpdb_ecommerce_database=#". The user enters the SQL command to create the 'product_category' table. The terminal shows the command being executed and then displays the table's schema. The schema table has columns: Column, Type, Collation, Nullable, and Default. The rows are: id (uuid, not null, uuid_generate_v4()), section (character varying(20), not null), and audience_segment (character varying(20), not null). Below the schema table, it shows the index: "product_category_pkey" PRIMARY KEY, btree (id).

```
kl_dpdb_ecommerce_database=# CREATE TABLE product_category (
kl_dpdb_ecommerce_database=#   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
kl_dpdb_ecommerce_database=#   section VARCHAR(20) NOT NULL,
kl_dpdb_ecommerce_database=#   audience_segment VARCHAR(20) NOT NULL
kl_dpdb_ecommerce_database=# );
kl_dpdb_ecommerce_database=#
kl_dpdb_ecommerce_database=# \d product_category
Table "public.product_category"
Column | Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid          |           | not null | uuid_generate_v4()
section | character varying(20) |           | not null |
audience_segment | character varying(20) |           | not null |
Indexes:
    "product_category_pkey" PRIMARY KEY, btree (id)
kl_dpdb_ecommerce_database=#
```

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

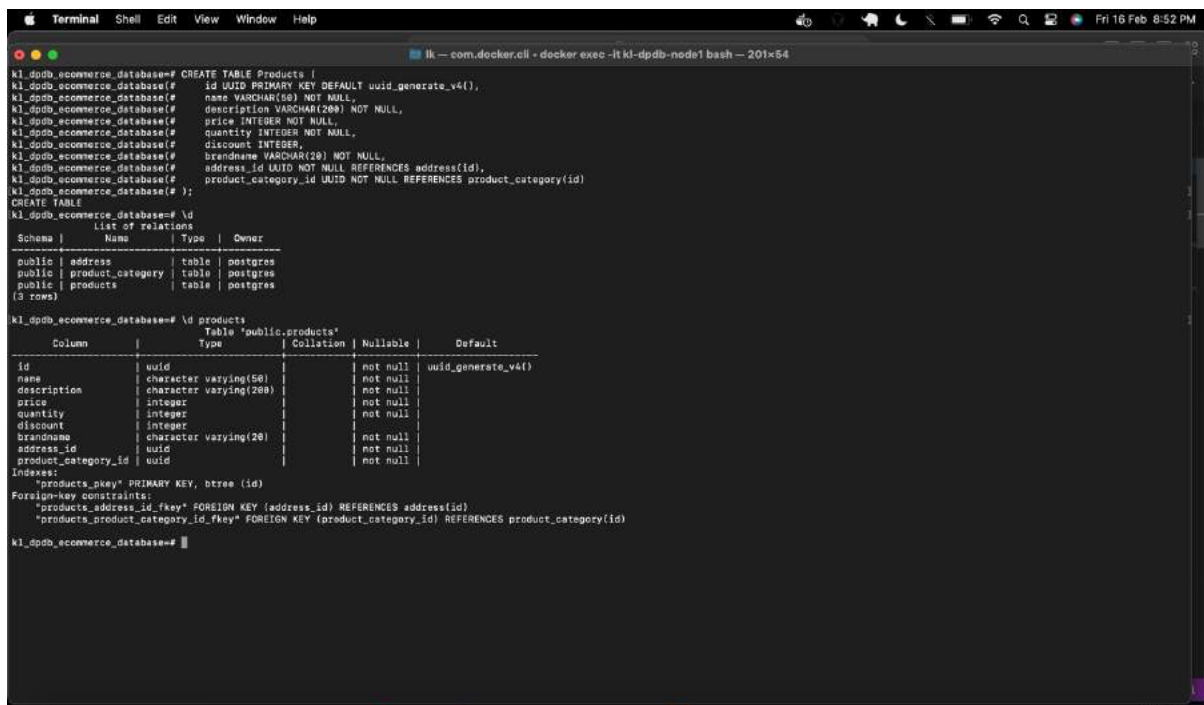
Explanation: The Products_Category table helps the customer to filter the products based on their choice. It helps the customer to search for products easily. There are different types of categories such as clothing, toys, accessories, etc. The product_category has a unique identifier, with attributes 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)  
);
```

A terminal window showing the execution of SQL commands to create a table and view its schema. The commands are run in a PostgreSQL shell. The first command creates the 'products' table with various attributes and foreign key constraints. The second command lists the relations in the database. The third command shows the detailed schema of the 'products' table, including column names, types, collations, nullability, and default values. The output shows the table is successfully created and its schema is displayed.

```
k1_dpdb_ecommerce_database=# 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)  
);  
k1_dpdb_ecommerce_database=#  
k1_dpdb_ecommerce_database=# \d  
List of relations  
Schema | Name | Type | Owner  
-----+-----+-----+-----  
public | address | table | postgres  
public | product_category | table | postgres  
public | products | table | postgres  
(3 rows)  
k1_dpdb_ecommerce_database=# \d products  
Table "public.products"  
Column | Type | Collation | Nullable | Default  
-----+-----+-----+-----+-----  
id | uuid | | not null | uuid_generate_v4()  
name | character varying(50) | | not null |  
description | character varying(200) | | not null |  
price | integer | | not null |  
quantity | integer | | not null |  
discount | integer | | |  
brandname | character varying(20) | | not null |  
address_id | uuid | | not null |  
product_category_id | uuid | | not null |  
Indexes:  
"products_pkey" PRIMARY KEY, btree (id)  
Foreign-key constraints:  
"products_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(id)  
"products_product_category_id_fkey" FOREIGN KEY (product_category_id) REFERENCES product_category(id)  
k1_dpdb_ecommerce_database=#
```

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

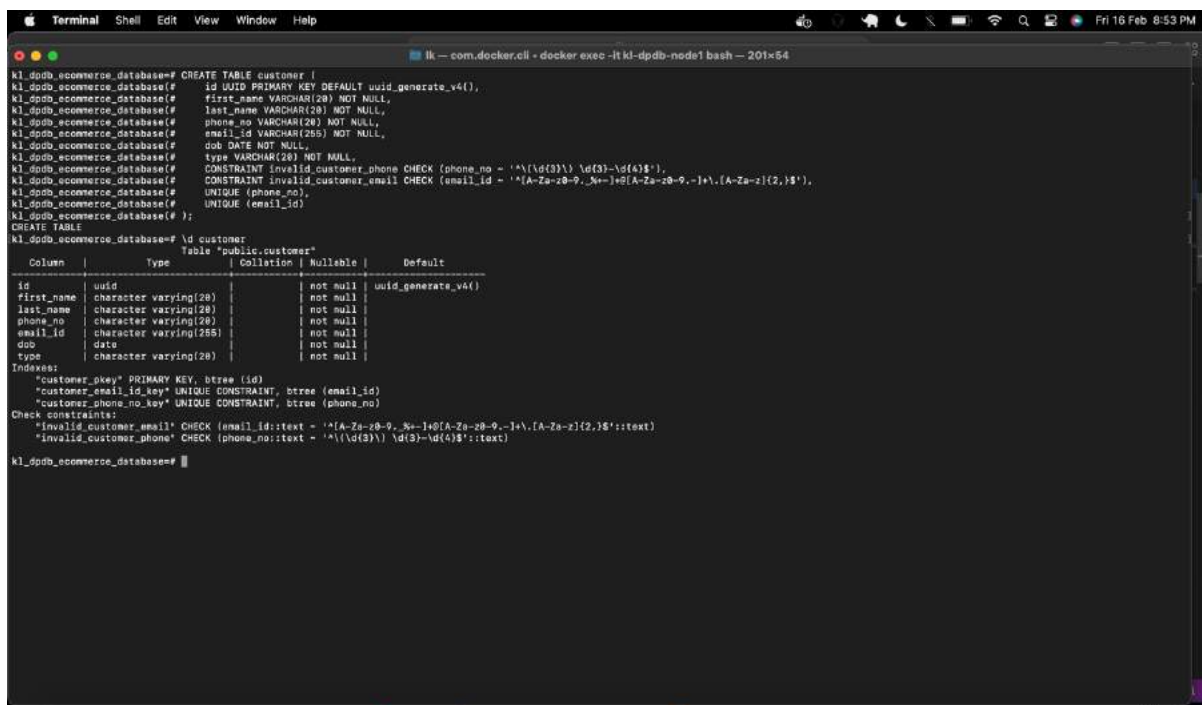
Explanation: The Products contain the list of products/items available. The products have a unique identifier with attributes name, description, price, quantity, discount, brandname, address_id and product_category_id. The address_id and the product_category_id are the foreign keys which refers to address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



The screenshot shows a terminal window with the following content:

```
kl_dadb_ecommerce_database=# 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)  
);  
kl_dadb_ecommerce_database=#  
kl_dadb_ecommerce_database=# \d customer  
Table "public.customer"  
+-----+-----+-----+-----+-----+  
| Column | Type | Collation | Nullable | Default |  
+-----+-----+-----+-----+-----+  
| id | uuid | | not null | uuid_generate_v4() |  
| first_name | character varying(20) | | not null | |  
| last_name | character varying(20) | | not null | |  
| phone_no | character varying(20) | | not null | |  
| email_id | character varying(255) | | not null | |  
| dob | date | | not null | |  
| type | character varying(20) | | not null | |  
+-----+-----+-----+-----+-----+  
Indexes:  
+-----+  
| "customer_pkey" PRIMARY KEY, btree (id) |  
+-----+  
| "customer_email_id_key" UNIQUE CONSTRAINT, btree (email_id) |  
+-----+  
| "customer_phone_no_key" UNIQUE CONSTRAINT, btree (phone_no) |  
+-----+  
Check constraints:  
+-----+  
| "invalid_customer_email" CHECK (email_id::text ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$':text) |  
+-----+  
| "invalid_customer_phone" CHECK (phone_no::text ~ '^\d{3}\d{3}-\d{4}$':text) |  
+-----+  
kl_dadb_ecommerce_database=#
```

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

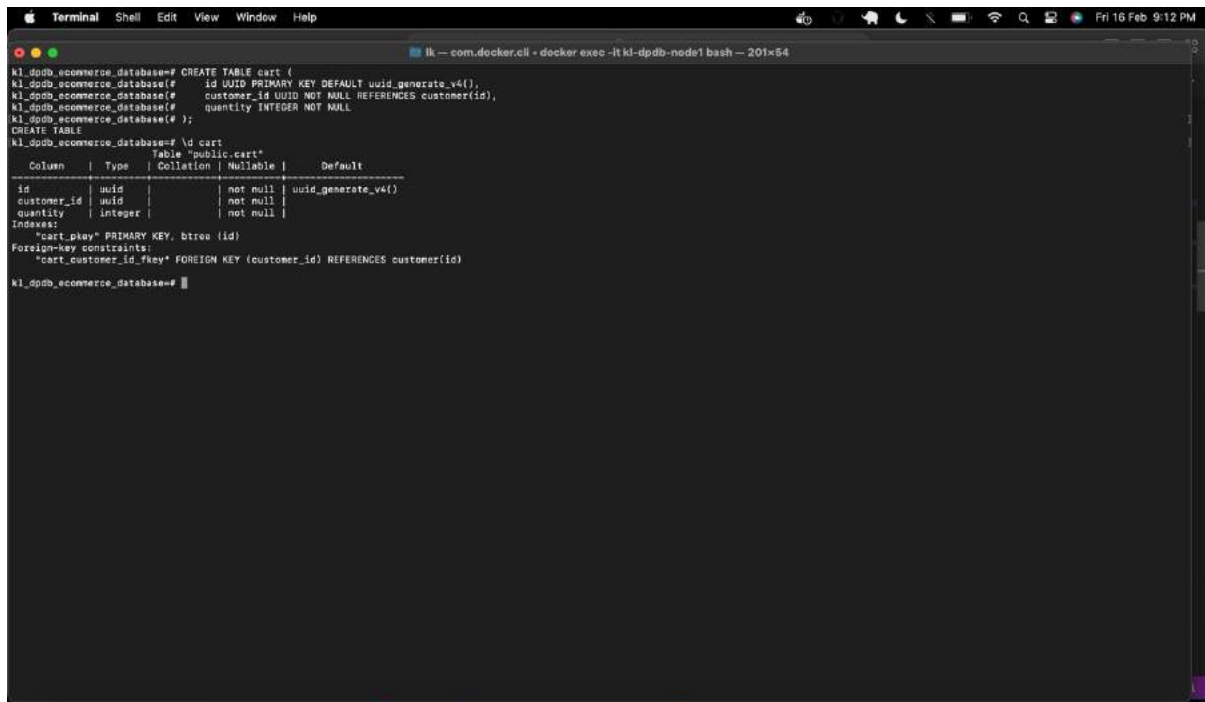
Explanation: The customer table contains the list of customer details. This one is the crucial table. The customer table has a unique identifier with attributes first_name, last_name, phone_no, email_id, dob, and type. The phone_no and email_id have unique check constraints which helps to avoid duplicate 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  
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:12 PM). The terminal shows a Docker command prompt where a table named 'cart' is created in a database. The command is: `kl_dpdb_ecommerce_database=# CREATE TABLE cart (`
`kl_dpdb_ecommerce_database=# id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),`
`kl_dpdb_ecommerce_database=# customer_id UUID NOT NULL REFERENCES customer(id),`
`kl_dpdb_ecommerce_database=# quantity INTEGER NOT NULL`
`kl_dpdb_ecommerce_database=#);`
The prompt returns to `kl_dpdb_ecommerce_database=#`. Then, the command `kl_dpdb_ecommerce_database=# \d cart` is entered, displaying the table's schema. The output shows the table 'public.cart' with columns: 'id' (UUID, not null, default 'uuid_generate_v4()'), 'customer_id' (UUID, not null), and 'quantity' (integer, not null). It also lists indexes: 'cart_pkey' (PRIMARY KEY, btree (id)) and foreign key constraints: 'cart_customer_id_fkey' (FOREIGN KEY (customer_id) REFERENCES customer(id)).

```
kl_dpdb_ecommerce_database=# CREATE TABLE cart (  
kl_dpdb_ecommerce_database=# id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),  
kl_dpdb_ecommerce_database=# customer_id UUID NOT NULL REFERENCES customer(id),  
kl_dpdb_ecommerce_database=# quantity INTEGER NOT NULL  
kl_dpdb_ecommerce_database=# );  
kl_dpdb_ecommerce_database=# \d cart  
Table "public.cart"  
Column | Type | Collation | Nullable | Default  
id | uuid | | not null | uuid_generate_v4()  
customer_id | uuid | | not null |  
quantity | integer | | not null |  
Indexes:  
"cart_pkey" PRIMARY KEY, btree (id)  
Foreign key constraints:  
"cart_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)  
kl_dpdb_ecommerce_database=#
```

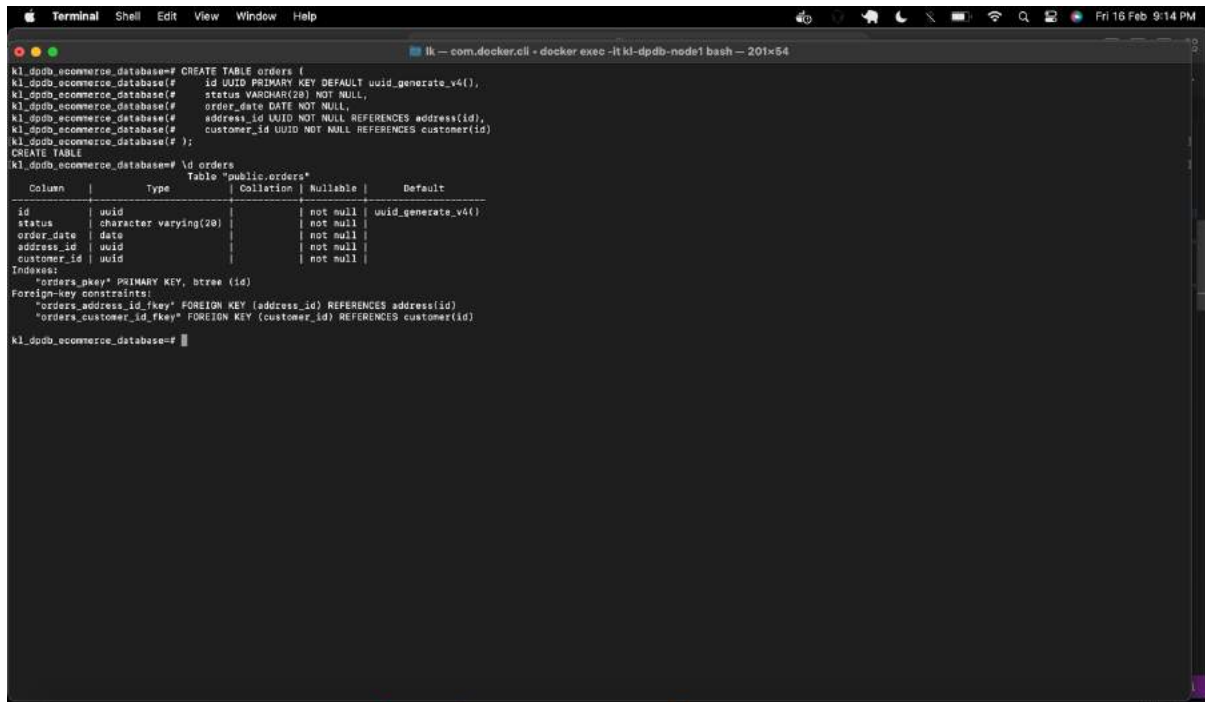
Fig-15: Create the cart table and display the schema of it.

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)
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:14 PM). The terminal shows the execution of SQL commands in a Docker container. The commands create a table named 'orders' with columns: 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)), and customer_id (UUID, NOT NULL, REFERENCES customer(id)). The output shows the table creation and a schema display for the 'orders' table.

```
kl_dpdb_ecommerce_database=# CREATE TABLE orders (
kl_dpdb_ecommerce_database=#   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
kl_dpdb_ecommerce_database=#   status VARCHAR(20) NOT NULL,
kl_dpdb_ecommerce_database=#   order_date DATE NOT NULL,
kl_dpdb_ecommerce_database=#   address_id UUID NOT NULL REFERENCES address(id),
kl_dpdb_ecommerce_database=#   customer_id UUID NOT NULL REFERENCES customer(id)
kl_dpdb_ecommerce_database=# );
CREATE TABLE
kl_dpdb_ecommerce_database=# \d orders
Table "public.orders"
Column | Type | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid |           | not null | uuid_generate_v4()
status  | character varying(20) |           | not null |
order_date | date |           | not null |
address_id | uuid |           | not null |
customer_id | uuid |           | not null |
Indexes:
    "orders_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "orders_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(id)
    "orders_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)
kl_dpdb_ecommerce_database=#
```

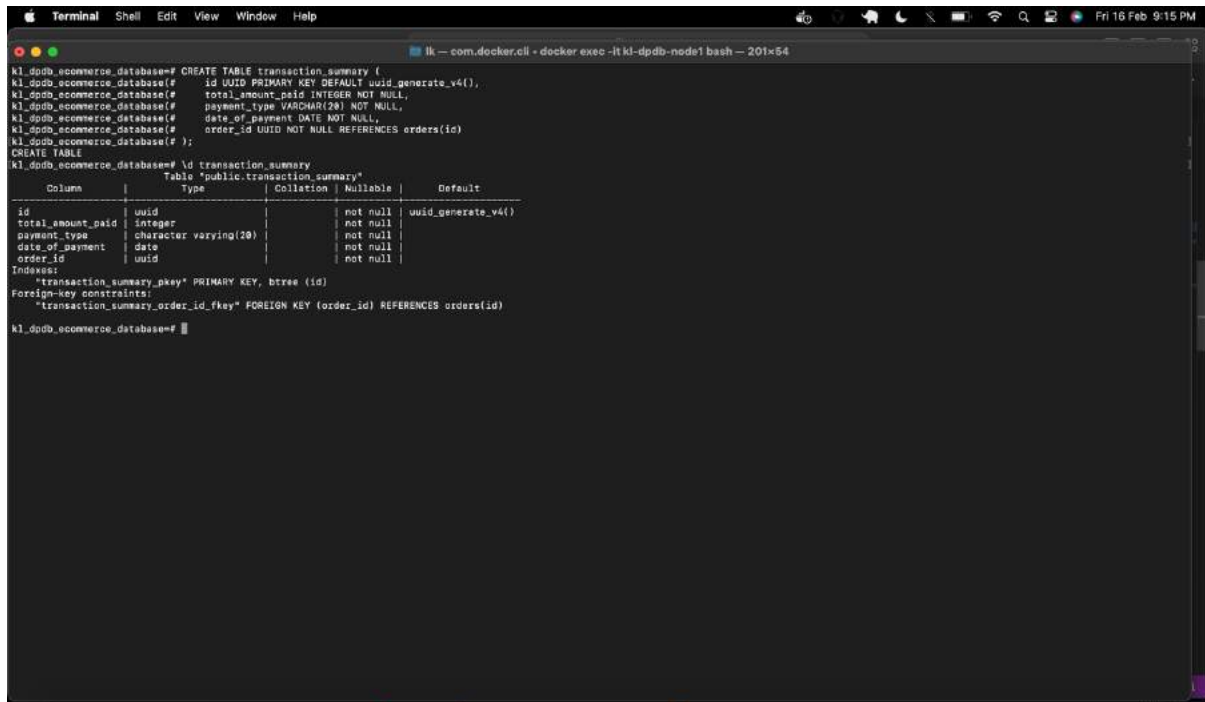
Fig-16: Create the orders table and display the schema of it.

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)
);
```

A terminal window showing the execution of SQL commands to create a table and display its schema. The commands are run in a PostgreSQL environment. The output shows the table creation and a detailed schema view.

```
k1_dpdb_ecommerce_database=# CREATE TABLE transaction_summary (
k1_dpdb_ecommerce_database=#   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
k1_dpdb_ecommerce_database=#   total_amount_paid INTEGER NOT NULL,
k1_dpdb_ecommerce_database=#   payment_type VARCHAR(20) NOT NULL,
k1_dpdb_ecommerce_database=#   date_of_payment DATE NOT NULL,
k1_dpdb_ecommerce_database=#   order_id UUID NOT NULL REFERENCES orders(id)
k1_dpdb_ecommerce_database=# );
CREATE TABLE
k1_dpdb_ecommerce_database=# \d transaction_summary
Table "public.transaction_summary"
  Column      | Type          | Collation | Nullable | Default
-----
id            | uuid         |           | not null | uuid_generate_v4()
total_amount_paid | integer      |           | not null |
payment_type  | character varying(20) |           | not null |
date_of_payment | date         |           | not null |
order_id      | uuid         |           | not null |
Indexes:
    "transaction_summary_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "transaction_summary_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(id)
k1_dpdb_ecommerce_database=#
```

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

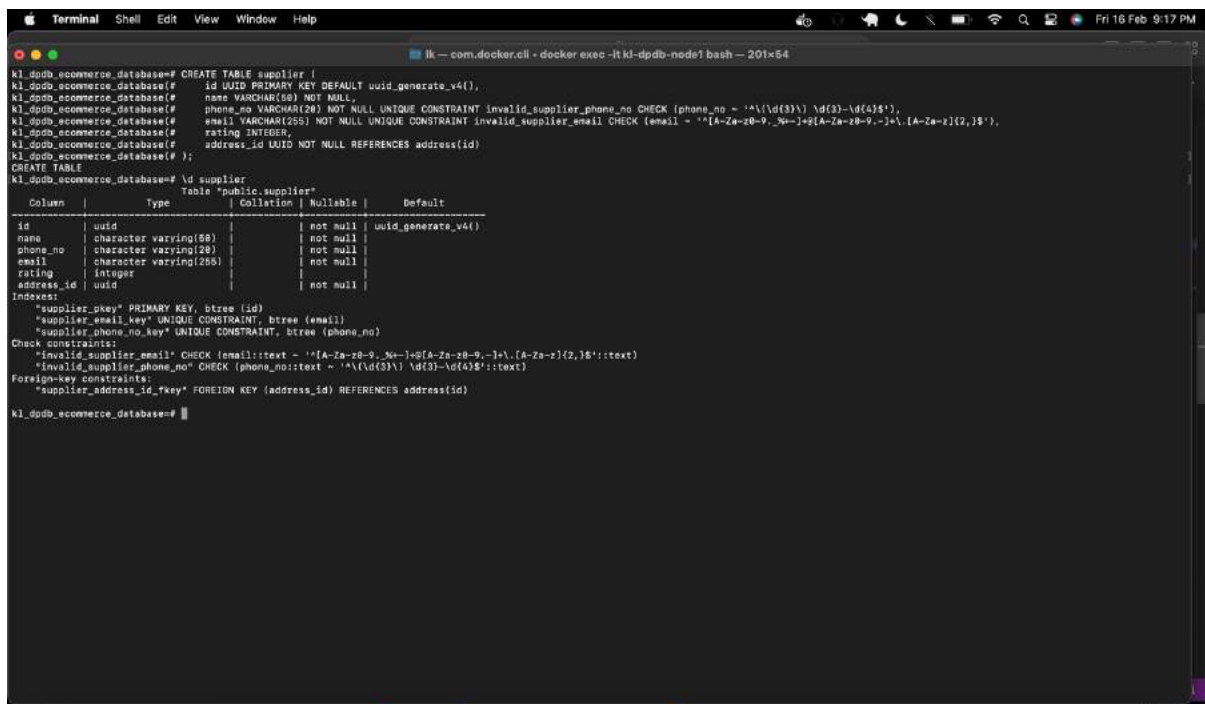
Explanation: The transaction_summary table talks about the transactions done by the customers. The transaction_summary has a unique identifier with attributes total_amount_paid, payment_type, date_of_payment, order_id. The total_amount, talks about the total amount paid to by the customer on specific 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)  
);
```



The terminal screenshot shows the execution of SQL commands to create a table named 'supplier' in a database. The commands are as follows:

```
kl_dadb_ecommerce_database=# CREATE TABLE supplier (  
kl_dadb_ecommerce_database=# id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),  
kl_dadb_ecommerce_database=# name VARCHAR(50) NOT NULL,  
kl_dadb_ecommerce_database=# phone_no VARCHAR(20) NOT NULL UNIQUE CONSTRAINT invalid_supplier_phone_no CHECK (phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),  
kl_dadb_ecommerce_database=# email VARCHAR(255) NOT NULL UNIQUE CONSTRAINT invalid_supplier_email CHECK (email ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'),  
kl_dadb_ecommerce_database=# rating INTEGER,  
kl_dadb_ecommerce_database=# address_id UUID NOT NULL REFERENCES address(id)  
kl_dadb_ecommerce_database=# );  
CREATE TABLE  
kl_dadb_ecommerce_database=# \d supplier  
Table "public.supplier"  
Column | Type | Collation | Nullable | Default  
-----  
id | uuid | | not null | uuid_generate_v4()  
name | character varying(50) | | not null |  
phone_no | character varying(20) | | not null |  
email | character varying(255) | | not null |  
rating | integer | | not null |  
address_id | uuid | | not null |  
Indexes:  
"supplier_pkey" PRIMARY KEY, btree (id)  
"supplier_email_key" UNIQUE CONSTRAINT, btree (email)  
"supplier_phone_no_key" UNIQUE CONSTRAINT, btree (phone_no)  
Check constraints:  
"invalid_supplier_email" CHECK (email::text ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$':text)  
"invalid_supplier_phone_no" CHECK (phone_no::text ~ '^\(\d{3}\) \d{3}-\d{4}$':text)  
Foreign-key constraints:  
"supplier_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(id)  
kl_dadb_ecommerce_database=#
```

The output shows the table structure with columns: id (uuid, primary key), name (character varying(50)), phone_no (character varying(20)), email (character varying(255)), rating (integer), and address_id (uuid, foreign key to address(id)). It also lists the indexes and constraints.

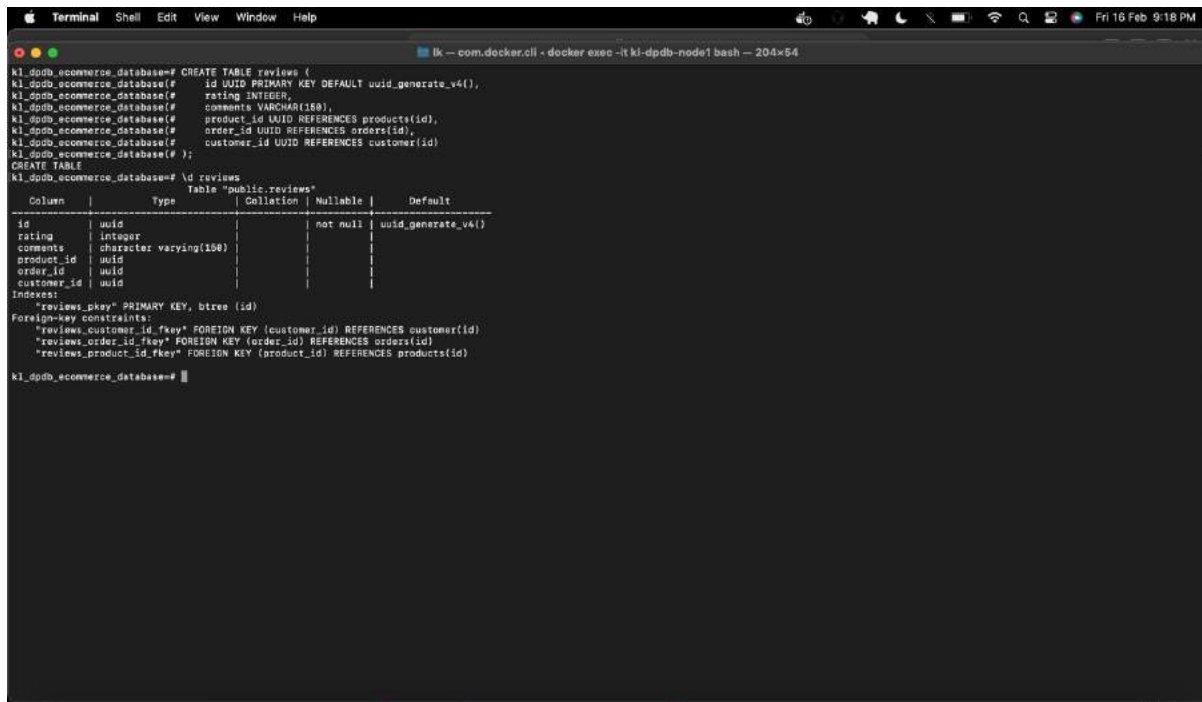
Fig-18: Create the supplier table and display the schema of it.

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 identifies 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)
);
```

A terminal window titled "Terminal" with a dark background. The prompt is "kl_dpgb_ecommerce_database=#". The user enters the SQL command to create the 'reviews' table. The output shows the table creation command repeated, followed by a table schema display for 'public.reviews'. The schema table has columns: Column, Type, Collation, Nullable, and Default. The rows are: id (uuid, not null, uuid_generate_v4()), rating (integer, null), comments (character varying(150), null), product_id (uuid, null), order_id (uuid, null), and customer_id (uuid, null). Below the schema, it lists indexes: 'reviews_pkey' PRIMARY KEY, btree (id) and four foreign key constraints: 'reviews_customer_id_fkey', 'reviews_order_id_fkey', and 'reviews_product_id_fkey'. The prompt returns to "kl_dpgb_ecommerce_database=#".

```
kl_dpgb_ecommerce_database=# CREATE TABLE reviews (
kl_dpgb_ecommerce_database=#     id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
kl_dpgb_ecommerce_database=#     rating INTEGER,
kl_dpgb_ecommerce_database=#     comments VARCHAR(150),
kl_dpgb_ecommerce_database=#     product_id UUID REFERENCES products(id),
kl_dpgb_ecommerce_database=#     order_id UUID REFERENCES orders(id),
kl_dpgb_ecommerce_database=#     customer_id UUID REFERENCES customer(id)
kl_dpgb_ecommerce_database=# );
CREATE TABLE
kl_dpgb_ecommerce_database=# \d reviews
          Table "public.reviews"
   Column   |      Type       | Collation | Nullable | Default
------------+-----+-----+-----+-----
 id         | uuid            |           | not null | uuid_generate_v4()
 rating     | integer         |           |          |
 comments   | character varying(150) |           |          |
 product_id | uuid            |           |          |
 order_id   | uuid            |           |          |
 customer_id | uuid            |           |          |
Indexes:
    "reviews_pkey" PRIMARY KEY, btree (id)
Foreign-key constraints:
    "reviews_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)
    "reviews_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(id)
    "reviews_product_id_fkey" FOREIGN KEY (product_id) REFERENCES products(id)
kl_dpgb_ecommerce_database=#
```

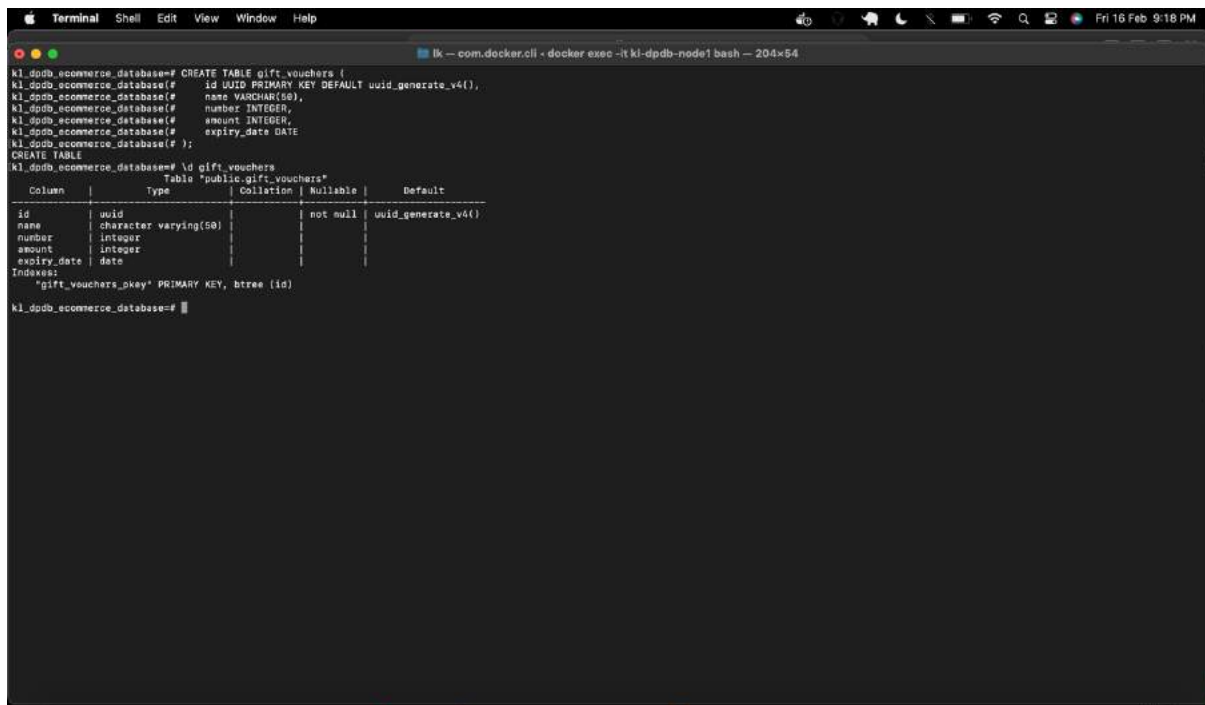
Fig-19: Create the reviews table and display the schema of it.

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
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:16 PM). The terminal shows a Docker command to run a PostgreSQL container. Inside the container, the user is at the prompt "kl_dpdb_ecommerce_database=#". They enter the command "CREATE TABLE gift_vouchers (" followed by the table definition: "id UUID PRIMARY KEY DEFAULT uuid_generate_v4()", "name VARCHAR(50)", "number INTEGER", "amount INTEGER", "expiry_date DATE", and a closing parenthesis and semicolon. The prompt changes to "kl_dpdb_ecommerce_database=#". They then enter "CREATE TABLE", which triggers the database to show the schema for the "public.gift_vouchers" table. The schema is displayed as a table with columns: Column, Type, Collation, Nullable, and Default. The rows are: id (uuid, not null, uuid_generate_v4()), name (character varying(50), not null, null), number (integer, not null, null), amount (integer, not null, null), and expiry_date (date, not null, null). Below the table, it says "Indexes: 'gift_vouchers_pkey' PRIMARY KEY, btree (id)". The prompt returns to "kl_dpdb_ecommerce_database=#".

```
kl_dpdb_ecommerce_database=# CREATE TABLE gift_vouchers (
kl_dpdb_ecommerce_database=#   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
kl_dpdb_ecommerce_database=#   name VARCHAR(50),
kl_dpdb_ecommerce_database=#   number INTEGER,
kl_dpdb_ecommerce_database=#   amount INTEGER,
kl_dpdb_ecommerce_database=#   expiry_date DATE
kl_dpdb_ecommerce_database=# );
CREATE TABLE
kl_dpdb_ecommerce_database=# \d gift_vouchers
Table "public.gift_vouchers"
Column | Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid          |           | not null | uuid_generate_v4()
name    | character varying(50) |           | not null |
number  | integer       |           | not null |
amount  | integer       |           | not null |
expiry_date | date         |           | not null |
Indexes:
    "gift_vouchers_pkey" PRIMARY KEY, btree (id)
kl_dpdb_ecommerce_database=#
```

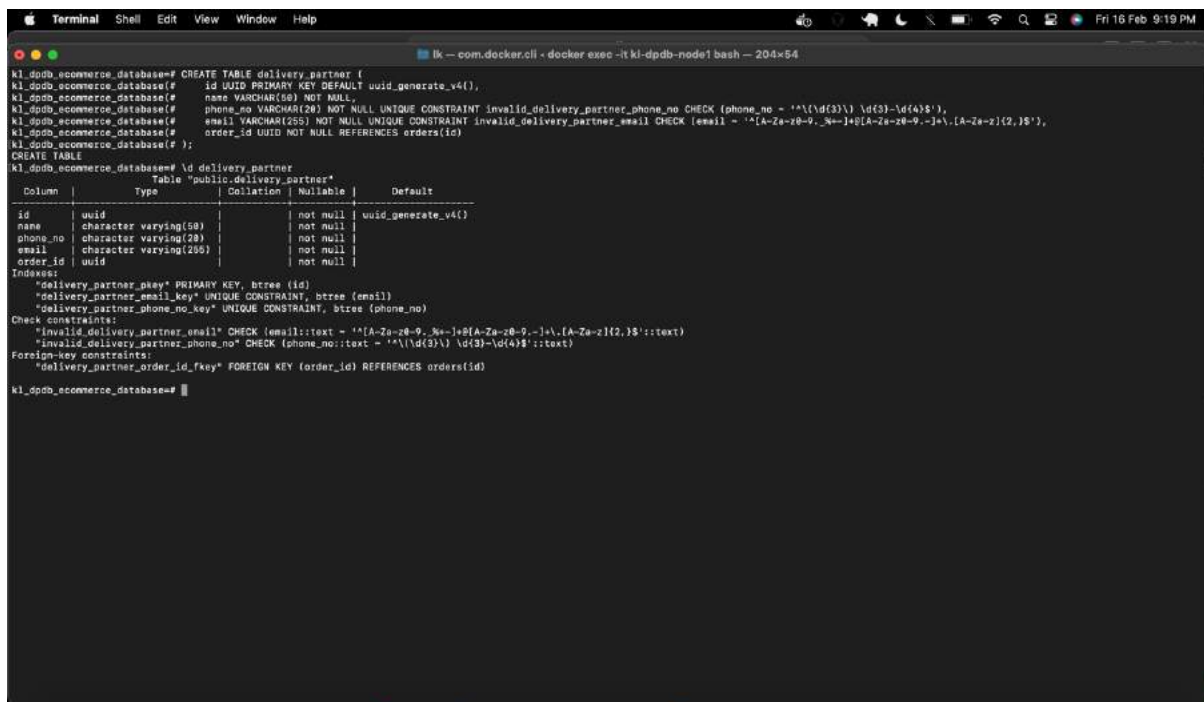
Fig-20: Create the gift_vouchers table and display the schema of it.

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)
);
```

A terminal window titled "Terminal" with a dark background. It shows the execution of SQL commands to create a table named "delivery_partner" in a database. The commands include creating the table with columns: id (UUID, PRIMARY KEY, DEFAULT uuid_generate_v4()), name (VARCHAR(50), NOT NULL), phone_no (VARCHAR(20), NOT NULL, with a CHECK constraint), email (VARCHAR(255), NOT NULL, with a CHECK constraint), and order_id (UUID, NOT NULL, with a FOREIGN KEY constraint referencing orders(id)). After the table is created, the command \d delivery_partner is used to display the table's schema. The output shows the table structure with columns, types, constraints, and indexes.

```
k1_dqdb_ecommerce_database=# CREATE TABLE delivery_partner (
k1_dqdb_ecommerce_database=#   id UUID PRIMARY KEY DEFAULT uuid_generate_v4(),
k1_dqdb_ecommerce_database=#   name VARCHAR(50) NOT NULL,
k1_dqdb_ecommerce_database=#   phone_no VARCHAR(20) NOT NULL UNIQUE CONSTRAINT invalid_delivery_partner_phone_no CHECK (phone_no ~ '^\(\d{3}\) \d{3}-\d{4}$'),
k1_dqdb_ecommerce_database=#   email VARCHAR(255) NOT NULL UNIQUE CONSTRAINT invalid_delivery_partner_email CHECK (email ~ '[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$'),
k1_dqdb_ecommerce_database=#   order_id UUID NOT NULL REFERENCES orders(id)
k1_dqdb_ecommerce_database=# );
CREATE TABLE
k1_dqdb_ecommerce_database=# \d delivery_partner
Table "public.delivery_partner"
Column | Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid          |           | not null | uuid_generate_v4()
name    | character varying(50) |           | not null |
phone_no | character varying(20) |           | not null |
email   | character varying(255) |           | not null |
order_id | uuid          |           | not null |
Indexes:
    "delivery_partner_pkey" PRIMARY KEY, btree (id)
    "delivery_partner_email_key" UNIQUE CONSTRAINT, btree (email)
    "delivery_partner_phone_no_key" UNIQUE CONSTRAINT, btree (phone_no)
Check constraints:
    "invalid_delivery_partner_email" CHECK (email::text ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9.-]+\.[A-Za-z]{2,}$::text)
    "invalid_delivery_partner_phone_no" CHECK (phone_no::text ~ '^\(\d{3}\) \d{3}-\d{4}$::text)
Foreign-key constraints:
    "delivery_partner_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(id)
k1_dqdb_ecommerce_database=#
```

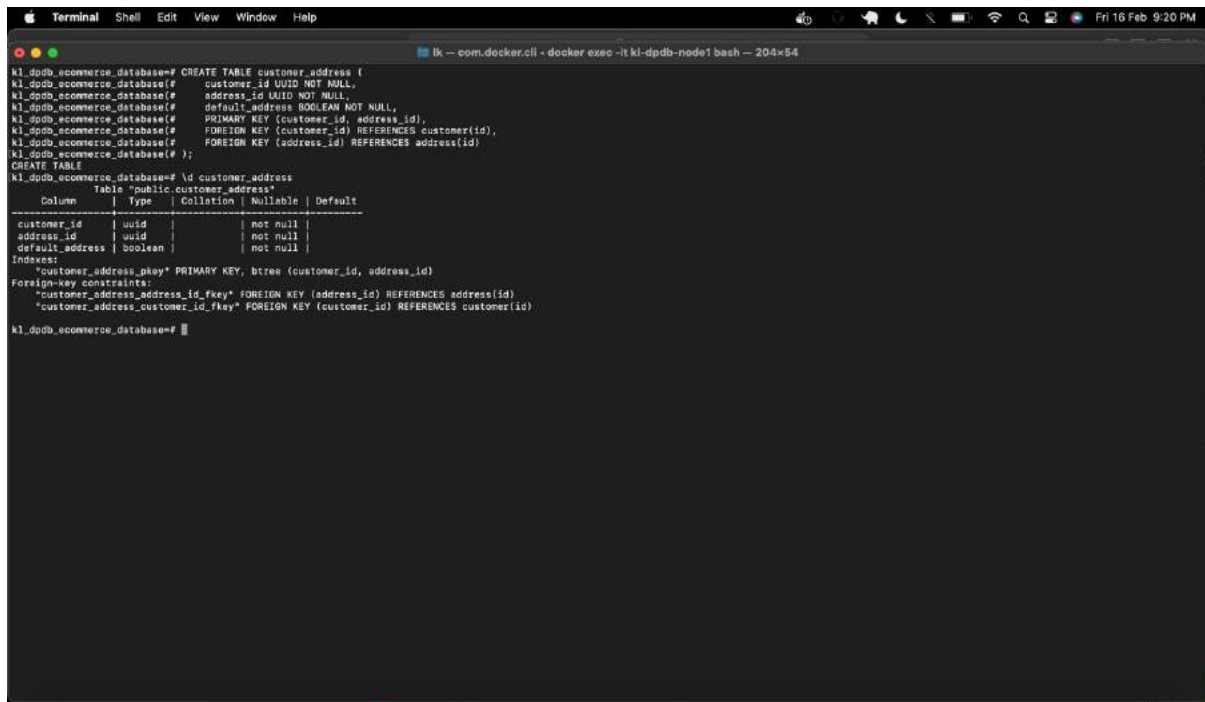
Fig-21: Create the delivery_partner table and display the schema of it.

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)
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:20 PM). The terminal shows the execution of SQL commands to create a table named "customer_address" in a database named "ki_dadb_ecommerce_database". The commands are: "CREATE TABLE customer_address (" followed by column definitions: "customer_id UUID NOT NULL," "address_id UUID NOT NULL," "default_address BOOLEAN NOT NULL," then constraints: "PRIMARY KEY (customer_id, address_id)," "FOREIGN KEY (customer_id) REFERENCES customer(id)," "FOREIGN KEY (address_id) REFERENCES address(id)" and finally a closing parenthesis and semicolon. Below the commands, the terminal displays the schema for the table "public.customer_address". It shows a table with 4 columns: "customer_id" (UUID, not null), "address_id" (UUID, not null), "default_address" (boolean, not null), and an empty "Default" column. Below the table definition, it lists indexes: "customer_address_pkey" PRIMARY KEY, btree (customer_id, address_id). Finally, it lists foreign key constraints: "customer_address_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(id) and "customer_address_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id). The prompt "ki_dadb_ecommerce_database=#" is shown at the bottom.

```
ki_dadb_ecommerce_database=# CREATE TABLE customer_address (
ki_dadb_ecommerce_database=#     customer_id UUID NOT NULL,
ki_dadb_ecommerce_database=#     address_id UUID NOT NULL,
ki_dadb_ecommerce_database=#     default_address BOOLEAN NOT NULL,
ki_dadb_ecommerce_database=#     PRIMARY KEY (customer_id, address_id),
ki_dadb_ecommerce_database=#     FOREIGN KEY (customer_id) REFERENCES customer(id),
ki_dadb_ecommerce_database=#     FOREIGN KEY (address_id) REFERENCES address(id)
ki_dadb_ecommerce_database=# );
CREATE TABLE
ki_dadb_ecommerce_database=# \d customer_address
Table "public.customer_address"
  Column      | Type          | Collation | Nullable | Default
-----
customer_id   | uuid         |           | not null |
address_id    | uuid         |           | not null |
default_address | boolean      |           | not null |
Indexes:
    "customer_address_pkey" PRIMARY KEY, btree (customer_id, address_id)
Foreign key constraints:
    "customer_address_address_id_fkey" FOREIGN KEY (address_id) REFERENCES address(id)
    "customer_address_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)
ki_dadb_ecommerce_database=#
```

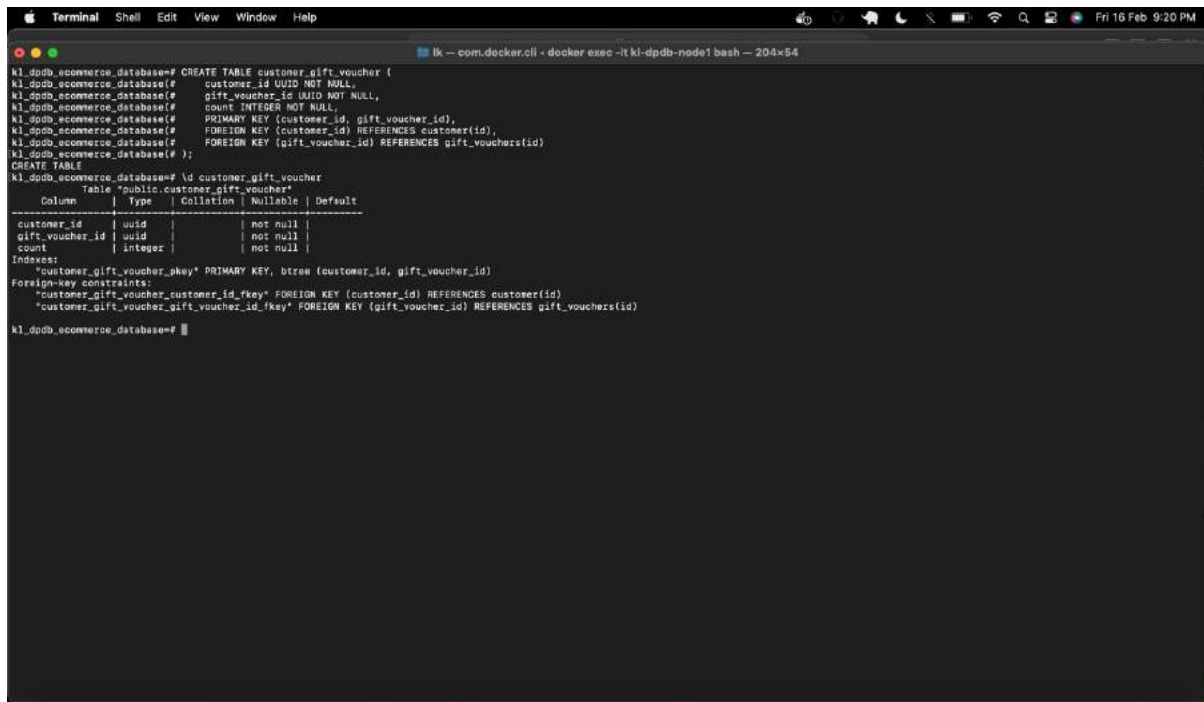
Fig-22: Create the customer_address table and display the schema of it.

Explanation: The relationship between the customer and address is many-to-many. This relation table forms the intermediary 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)
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:20 PM). The terminal shows the execution of a SQL command to create a table. The command is: `kl_dpdb_ecommerce_database=# CREATE TABLE customer_gift_voucher (` followed by the table definition. The output shows the table creation successful and then displays the schema for the table "public.customer_gift_voucher". The schema table has columns: customer_id (uuid, not null), gift_voucher_id (uuid, not null), and count (integer, not null). It also shows the primary key constraint on (customer_id, gift_voucher_id) and two foreign key constraints: one on customer_id referencing customer(id) and one on gift_voucher_id referencing gift_vouchers(id).

```
kl_dpdb_ecommerce_database=# CREATE TABLE customer_gift_voucher (
kl_dpdb_ecommerce_database=# customer_id UUID NOT NULL,
kl_dpdb_ecommerce_database=# gift_voucher_id UUID NOT NULL,
kl_dpdb_ecommerce_database=# count INTEGER NOT NULL,
kl_dpdb_ecommerce_database=# PRIMARY KEY (customer_id, gift_voucher_id),
kl_dpdb_ecommerce_database=# FOREIGN KEY (customer_id) REFERENCES customer(id),
kl_dpdb_ecommerce_database=# FOREIGN KEY (gift_voucher_id) REFERENCES gift_vouchers(id)
kl_dpdb_ecommerce_database=# );
CREATE TABLE
kl_dpdb_ecommerce_database=# \d customer_gift_voucher
Table "public.customer_gift_voucher"
  Column      | Type          | Collation | Nullable | Default
-----
customer_id   | uuid         |           | not null |
gift_voucher_id | uuid        |           | not null |
count         | integer      |           | not null |
Indexes:
    "customer_gift_voucher_pkey" PRIMARY KEY, btree (customer_id, gift_voucher_id)
Foreign-key constraints:
    "customer_gift_voucher_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)
    "customer_gift_voucher_gift_voucher_id_fkey" FOREIGN KEY (gift_voucher_id) REFERENCES gift_vouchers(id)
kl_dpdb_ecommerce_database=#
```

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

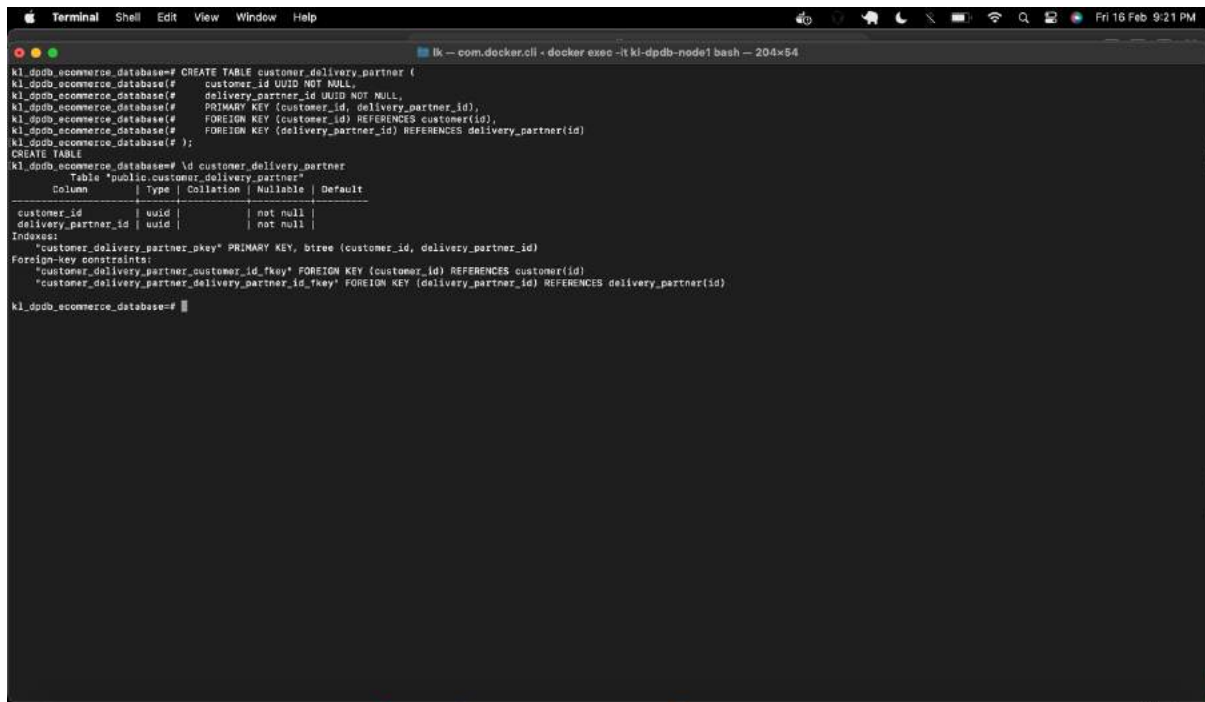
Explanation: The relationship between the customer and the gift voucher is many-to-many. This relation table forms the intermediary link between the customer and the gift voucher. The customer_id and gift_voucher_id are the primary and foreign keys, refers to the customer and gift_vouchers table. There 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)  
);
```



```
Terminal Shell Edit View Window Help  
ik - com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x64  
ki_dpdb_ecommerce_database# CREATE TABLE customer_delivery_partner (  
ki_dpdb_ecommerce_database#     customer_id UUID NOT NULL,  
ki_dpdb_ecommerce_database#     delivery_partner_id UUID NOT NULL,  
ki_dpdb_ecommerce_database#     PRIMARY KEY (customer_id, delivery_partner_id),  
ki_dpdb_ecommerce_database#     FOREIGN KEY (customer_id) REFERENCES customer(id),  
ki_dpdb_ecommerce_database#     FOREIGN KEY (delivery_partner_id) REFERENCES delivery_partner(id)  
ki_dpdb_ecommerce_database# );  
CREATE TABLE  
ki_dpdb_ecommerce_database# \d customer_delivery_partner  
Table "public.customer_delivery_partner"  
Column      | Type      | Collation | Nullable | Default  
-----  
customer_id  | uuid      |           | not null |  
delivery_partner_id | uuid      |           | not null |  
Indexes:  
"customer_delivery_partner_pkey" PRIMARY KEY, btree (customer_id, delivery_partner_id)  
Foreign-key constraints:  
"customer_delivery_partner_customer_id_fkey" FOREIGN KEY (customer_id) REFERENCES customer(id)  
"customer_delivery_partner_delivery_partner_id_fkey" FOREIGN KEY (delivery_partner_id) REFERENCES delivery_partner(id)  
ki_dpdb_ecommerce_database#
```

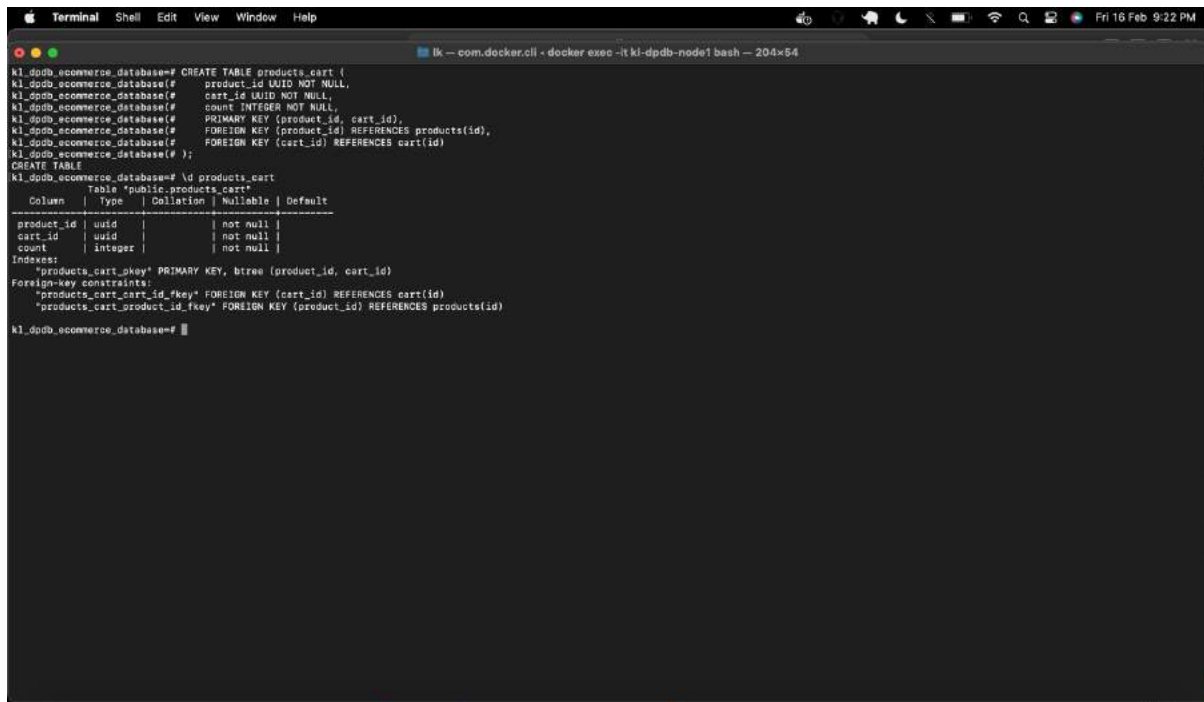
Fig-24: Display and Create the customer_delivery_partner table and display the schema of it.

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)
);
```

A terminal window titled "Terminal" with a dark background. The prompt is "kl_dadb_ecommerce_database=#". The user enters a SQL command to create a table named "products_cart" with columns "product_id" (UUID, NOT NULL), "cart_id" (UUID, NOT NULL), and "count" (INTEGER, NOT NULL). It also includes a primary key constraint on ("product_id", "cart_id") and two foreign key constraints: one for "product_id" referencing "products(id)" and another for "cart_id" referencing "cart(id)". After the command, the user enters "CREATE TABLE", and the terminal displays the schema for the "products_cart" table. The schema shows the columns, their types, collations, nullability, and default values. It also lists the indexes and foreign key constraints for the table.

```
kl_dadb_ecommerce_database=# CREATE TABLE products_cart (
kl_dadb_ecommerce_database=#     product_id UUID NOT NULL,
kl_dadb_ecommerce_database=#     cart_id UUID NOT NULL,
kl_dadb_ecommerce_database=#     count INTEGER NOT NULL,
kl_dadb_ecommerce_database=#     PRIMARY KEY (product_id, cart_id),
kl_dadb_ecommerce_database=#     FOREIGN KEY (product_id) REFERENCES products(id),
kl_dadb_ecommerce_database=#     FOREIGN KEY (cart_id) REFERENCES cart(id)
kl_dadb_ecommerce_database=# );
CREATE TABLE
kl_dadb_ecommerce_database=# \d products_cart
Table "public.products_cart"
  Column   | Type          | Collation | Nullable | Default
-----
product_id | uuid         |           | not null |
cart_id    | uuid         |           | not null |
count      | integer      |           | not null |
Indexes:
    "products_cart_pkey" PRIMARY KEY, btree (product_id, cart_id)
Foreign-key constraints:
    "products_cart_cart_id_fkey" FOREIGN KEY (cart_id) REFERENCES cart(id)
    "products_cart_product_id_fkey" FOREIGN KEY (product_id) REFERENCES products(id)
kl_dadb_ecommerce_database=#
```

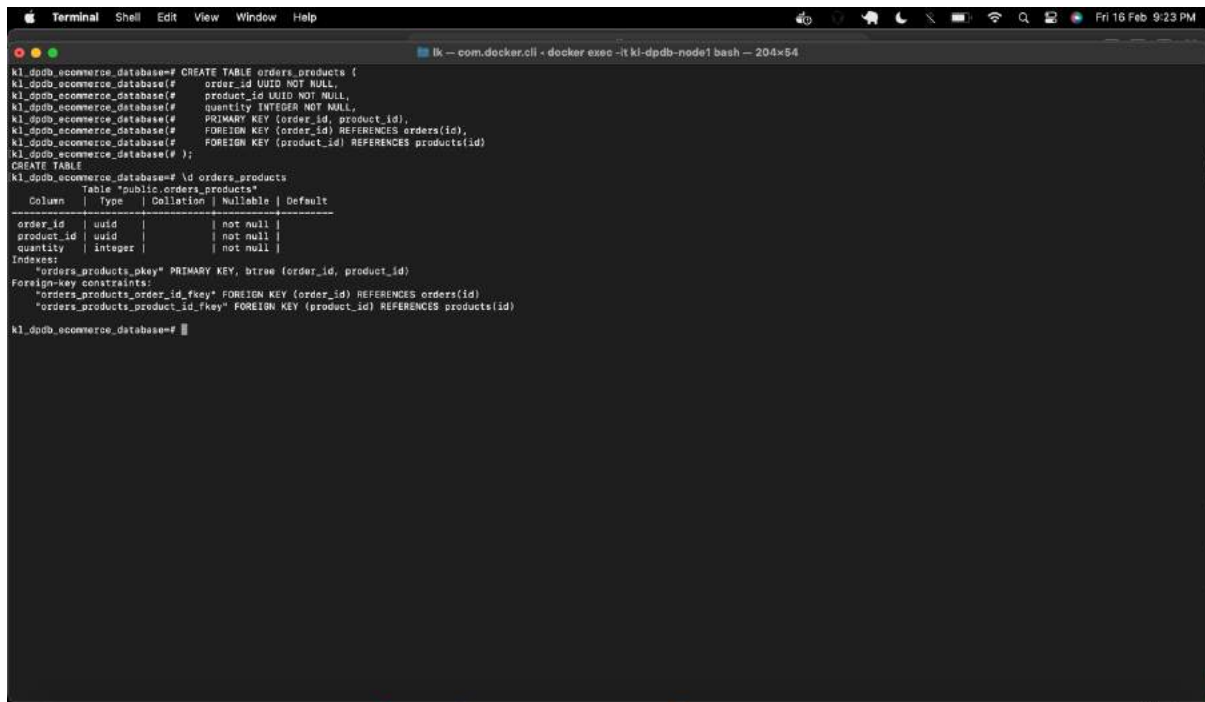
Fig-25: Create the products_cart table and display the schema of it.

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)
);
```

A screenshot of a terminal window with a dark background. The terminal shows the execution of SQL commands to create a table and display its schema. The commands are repeated multiple times. The output shows the table creation success, a table schema with columns order_id, product_id, and quantity, and foreign key constraints. The terminal window has a title bar with standard macOS window controls and a menu bar. The status bar at the bottom shows the date and time as Fri 16 Feb 9:23 PM.

```
kl_dadb_ecommerce_database=# CREATE TABLE orders_products (
kl_dadb_ecommerce_database=#     order_id UUID NOT NULL,
kl_dadb_ecommerce_database=#     product_id UUID NOT NULL,
kl_dadb_ecommerce_database=#     quantity INTEGER NOT NULL,
kl_dadb_ecommerce_database=#     PRIMARY KEY (order_id, product_id),
kl_dadb_ecommerce_database=#     FOREIGN KEY (order_id) REFERENCES orders(id),
kl_dadb_ecommerce_database=#     FOREIGN KEY (product_id) REFERENCES products(id)
kl_dadb_ecommerce_database=# );
CREATE TABLE
kl_dadb_ecommerce_database=# \d orders_products
Table "public.orders_products"
  Column      | Type          | Collation | Nullable | Default
-----
order_id     | uuid         |           | not null |
product_id   | uuid         |           | not null |
quantity     | integer      |           | not null |
Indexes:
    "orders_products_pkey" PRIMARY KEY, btree (order_id, product_id)
Foreign-key constraints:
    "orders_products_order_id_fkey" FOREIGN KEY (order_id) REFERENCES orders(id)
    "orders_products_product_id_fkey" FOREIGN KEY (product_id) REFERENCES products(id)
kl_dadb_ecommerce_database=#
```

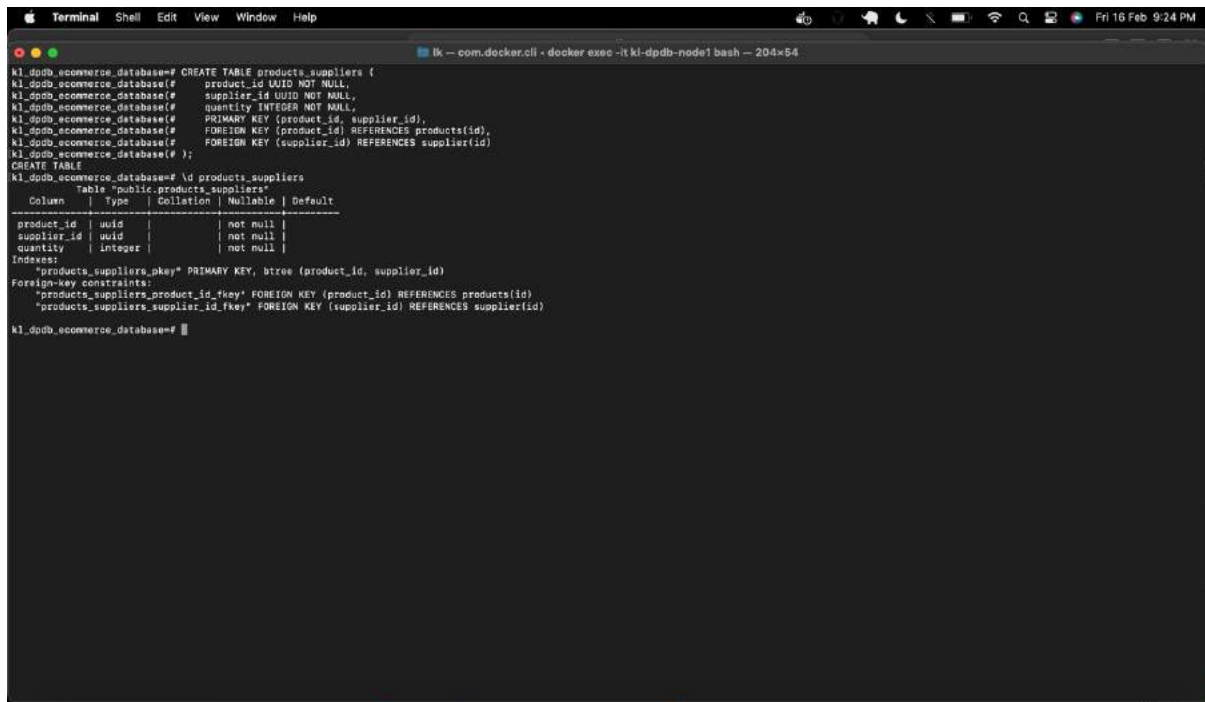
Fig-26: Create and Display the orders_products table and schema of it.

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)
);
```

A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 9:24 PM). The terminal shows a Docker container shell prompt "kl -- com.docker.cli - docker exec -it kl-dpdb-node1 bash -- 204x64". The user enters a SQL command to create a table: "kl_dpdb_ecommerce_database=# 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));". The prompt changes to "kl_dpdb_ecommerce_database=#". The user then enters "CREATE TABLE" followed by a display command. The terminal shows the table schema for "products_suppliers" with columns: product_id (uuid, not null), supplier_id (uuid, not null), and quantity (integer, not null). It also shows the primary key constraint on (product_id, supplier_id) and two foreign key constraints: "products_suppliers_product_id_fkey" referencing products(id) and "products_suppliers_supplier_id_fkey" referencing supplier(id).

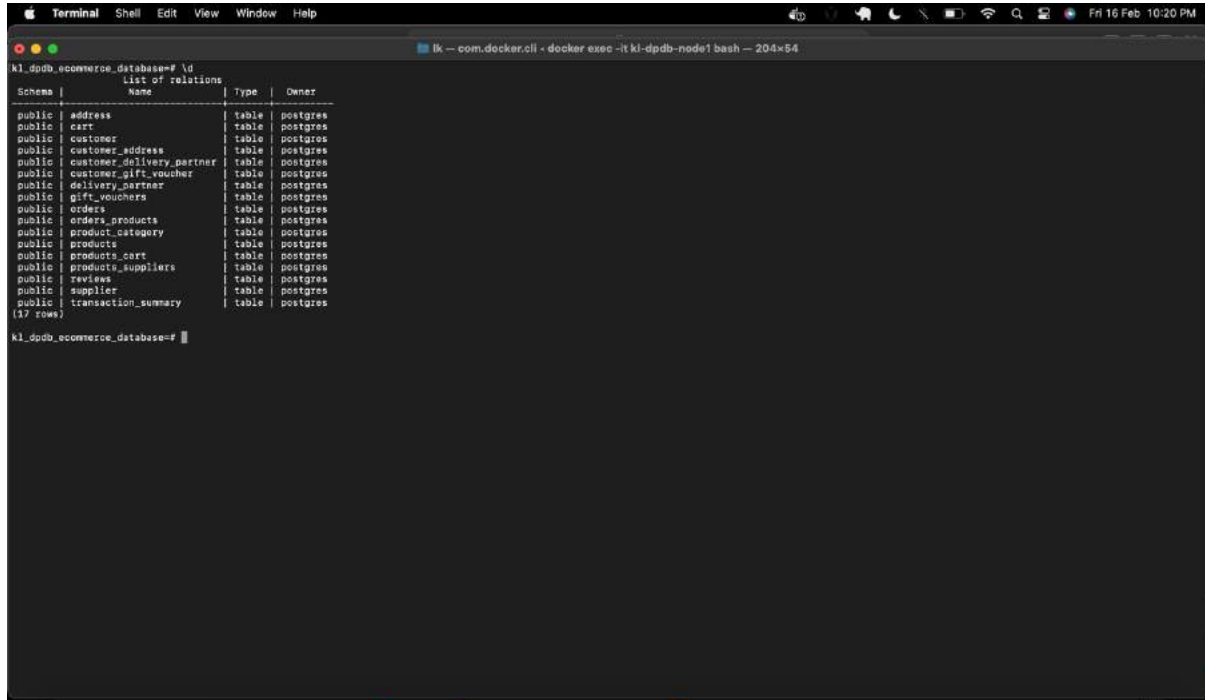
```
kl_dpdb_ecommerce_database=# CREATE TABLE products_suppliers (
kl_dpdb_ecommerce_database=#     product_id UUID NOT NULL,
kl_dpdb_ecommerce_database=#     supplier_id UUID NOT NULL,
kl_dpdb_ecommerce_database=#     quantity INTEGER NOT NULL,
kl_dpdb_ecommerce_database=#     PRIMARY KEY (product_id, supplier_id),
kl_dpdb_ecommerce_database=#     FOREIGN KEY (product_id) REFERENCES products(id),
kl_dpdb_ecommerce_database=#     FOREIGN KEY (supplier_id) REFERENCES supplier(id)
kl_dpdb_ecommerce_database=# );
CREATE TABLE
kl_dpdb_ecommerce_database=# \d products_suppliers
Table "public.products_suppliers"
  Column      | Type      | Collation | Nullable | Default
-----
product_id   | uuid     |           | not null |
supplier_id  | uuid     |           | not null |
quantity     | integer  |           | not null |
Indexes:
    "products_suppliers_pkey" PRIMARY KEY, btree (product_id, supplier_id)
Foreign-key constraints:
    "products_suppliers_product_id_fkey" FOREIGN KEY (product_id) REFERENCES products(id)
    "products_suppliers_supplier_id_fkey" FOREIGN KEY (supplier_id) REFERENCES supplier(id)
kl_dpdb_ecommerce_database=#
```

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).



```
kl_dpdb_ecommerce_database=# \d
List of relations
Schema | Name          | Type  | Owner
-----+-----+-----+-----
public | address       | table | postgres
public | cart          | table | postgres
public | customer      | table | postgres
public | customer_address | table | postgres
public | customer_delivery_partner | table | postgres
public | customer_gift_voucher | table | postgres
public | delivery_partner | table | postgres
public | gift_vouchers | table | postgres
public | orders        | table | postgres
public | orders_products | table | postgres
public | product_category | table | postgres
public | products      | table | postgres
public | products_cart  | table | postgres
public | products_suppliers | table | postgres
public | reviews       | table | postgres
public | supplier       | table | postgres
public | transaction_summary | table | postgres
(17 rows)
```

kl_dpdb_ecommerce_database=#

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->B, A->C, A->D, A->E, A->F, A->G, A->H, A->I	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					
	quantity = E		E+ = E					
	discount = F		F+ = F					
	brandname = G		G+ = G					
	address_id = H		H+ = H					
	product_category_id = I		I+ = I					
customer	id = A(primary key)	A->B, A->C, A->D, A->E, A->F, A->G	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		G+ = G					
cart	id = A	A->B, A->C	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		C+ = C					
orders	id = A	A->B, A->C, A->D, A->E	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		E+ = E					
transaction_summary	id = A	A->B, A->C, A->D, A->E	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		E+ = E					
supplier	id = A	A->B, A->C, A->D, A->E, A->F	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		F+ = F					
address	id = A	A->B, A->C, A->D, A->E, A->F, A->G	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		G+ = G					
	id = A		A+ = ABC				The highest normal form of the	All of the left hand keys of FDs are

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 columns, 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', 'dkNCFcWp0G', '(354)
220-6490', 'dawge@example.com', '1979-12-18', 'platinum');
```

•
•
•

1200 records.

```
Terminal Shell Edit View Window Help
[1] ~ — com.docker.cli - docker exec -it kl-dpdb-node1 bash — 204x54

kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('5aac2394-4bb8-4dc2-afbf-599f83cd8414', 'tdLnVeuWfq', 'JHgTMhnFTV', '(787) 518-9892', 'tscop@example.com', '1984-11-17', 'regular');
INSERT # 1
kl_dpdb_ecommerce_database=# select * from customer;
      id      | first_name | last_name | phone_no | email_id | dob      | type
-----+-----+-----+-----+-----+-----+-----
 5aac2394-4bb8-4dc2-afbf-599f83cd8414 | tdLnVeuWfq | JHgTMhnFTV | (787) 518-9892 | tscop@example.com | 1984-11-17 | regular
(1 row)

kl_dpdb_ecommerce_database=#
```

Fig-29: Insert data into the customer table.

```
Terminal Shell Edit View Window Help
[1] ~ — com.docker.cli - docker exec -it kl-dpdb-node1 bash — 204x54

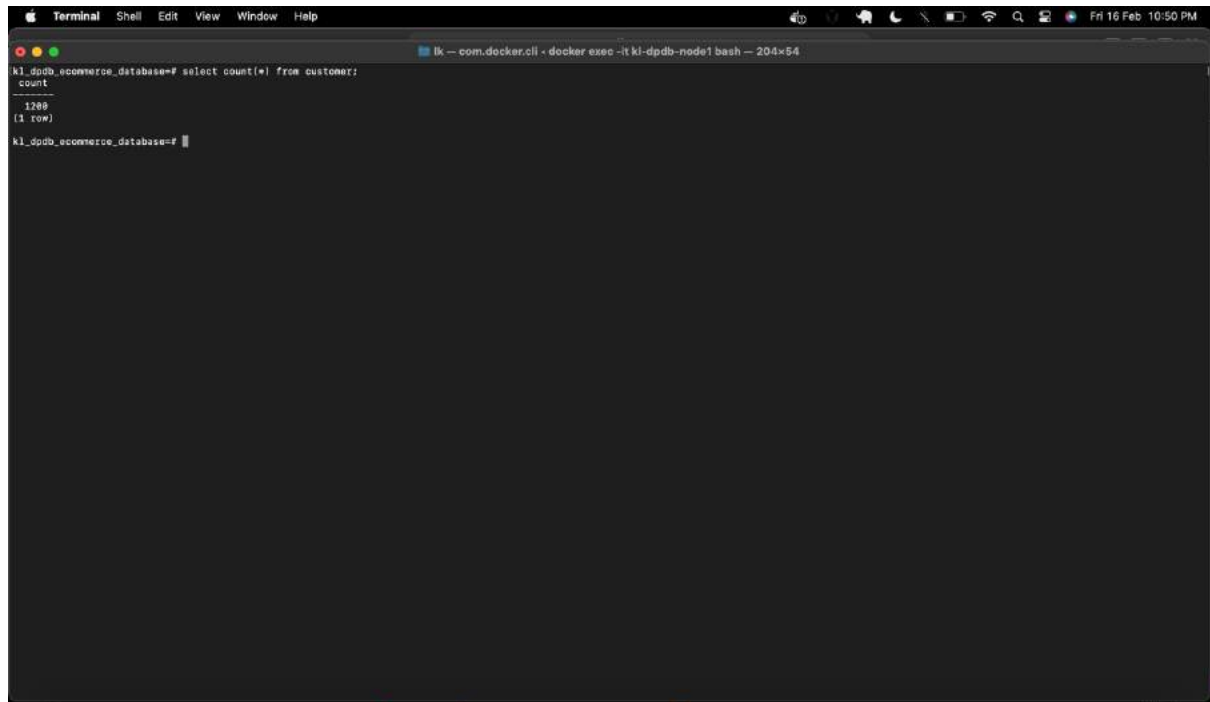
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('e62572da-74f5-4bd3-8787-9c3198f7791e', 'g8S2rywv1Y', 'hevrC0hcep', '((488) 198-9992', 'hdsap@example.com', '1979-03-22', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('95f1a8b5-ab3f-42d8-a4bb-a55987e38a93', 'UdeXQqphn', 'aUaPKNTqBq', '(523) 727-3835', 'blwvq@example.com', '1959-12-04', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('4734dc5b-d861-4fcf-95b4-24b313b41eeb', 'VJkQWtLTM', 'RvdChocEp', '(557) 188-7313', 'rluq@example.com', '1944-07-02', 'silver');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('f85c5533-6852-4a16-a452-784a917f3ae1', 'WQJMyNontT', 'J3MoxDPEUj', '(852) 423-6453', 'rdct@example.com', '1988-03-17', 'silver');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('fcac5699-8b94-4ada-9956-b6fd6c6738f8', 'nebxRDrlKH', 'JMGw1znYP', '(197) 482-6623', 'vkslv@example.com', '1980-08-28', 'platinum');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('6411061b-a33e-43b1-a189-de39f4514698', 'NRIEaoXRHF', 'PbRtBRXdd', '(256) 111-9858', 'krllc@example.com', '1961-07-03', 'platinum');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('f855a184-4a3d-4987-9782-e9b4e59b67ec', 'TWyykFEFE1', 'IlFqZdLQeH', '(883) 218-9663', 'nohhd@example.com', '1985-04-01', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('39def116-6d29-4287-b8b3-7362ff13163b', 'goaQuaInBE', 'etqprboCSI', '(795) 557-9325', 'nxbtk@example.com', '1949-12-24', 'premium');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('fec3d8a7-afe9-4491-9eff-24a3ff15137d', 'FAUnIwggY1', 'DbrFP1BZbp', '(391) 938-3864', 'nohe@example.com', '1985-07-11', 'silver');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('b286dc8c-3d2b-4ab8-a82d-97994836357c', 'iUNGKrecFa', 'zcjFQZ1vwe', '(572) 811-1784', 'www@example.com', '1956-07-05', 'platinum');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('6b4f3864-7cd4-42fa-b88e-15515f3d4d4e', 'cghyeWmQQQ', 'fLZRkTFoRI', '(483) 648-2741', 'uckub@example.com', '1991-03-27', 'platinum');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('5fd429e6-bf94-4679-6771-a38041dd9b26', 'FqQPrnH1Q', 'gQXDMWYcstr', '(578) 326-1741', 'zwyuc@example.com', '1989-05-13', 'premium');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('5681dc3-e84c-4cda-8c3b-6728ad7deae9', 'iCakNUukLW', 'RtxKWabehD', '(476) 187-7417', 'sfvvi@example.com', '1999-01-23', 'platinum');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('83e2e8d3-4dba-411f-b428-481d5dc438ee', 'BePJMZXxRK', 'QCLYvYVXf', '(346) 243-7285', 'higfi@example.com', '1994-06-19', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('4d145833-b76d-44c4-a7db-caabb3da332f', 'mNYAJHjQZF', 'SYORWONPQ', '(119) 687-8458', 'jfgd@example.com', '1972-06-23', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('f4cb563b-ec68-4487-9868-e98525ed66ee', 'zBbtYxergc', 'CFyKunLchb', '(988) 924-6764', 'xHfep@example.com', '1957-08-09', 'gold');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('f01bc7df-c869-421a-b188-c1abadd3dee6', 'tbkZnb1OLB', 'HPFaeEW2zb', '(792) 493-7461', 'mava@example.com', '1965-05-10', 'regular');
INSERT # 1
kl_dpdb_ecommerce_database=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('54dad1d3-b437-47d9-98d4-bc71f946ca95', 'mmuPVDDZx', 'TPVvaAbp1a', '(686) 868-4948', 'ellof@example.com', '1985-09-08', 'premium');
```

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;
```

A screenshot of a macOS Terminal window. The title bar shows 'Terminal' and standard window controls. The terminal content shows a user prompt 'ki_dadb_ecommerce_database#' followed by the SQL query 'select count(*) from customer;'. The output is a table with one row: 'count' with the value '1200'. Below the output, it says '(1 row)'. The prompt 'ki_dadb_ecommerce_database#' is shown again at the bottom. The top status bar of the terminal window indicates the command 'fk -- com.docker.cli - docker exec -it ki-dadb-node1 bash -- 204x54' and the date 'Fri 16 Feb 10:50 PM'.

```
ki_dadb_ecommerce_database# select count(*) from customer;
count
-----
1200
(1 row)
ki_dadb_ecommerce_database#
```

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

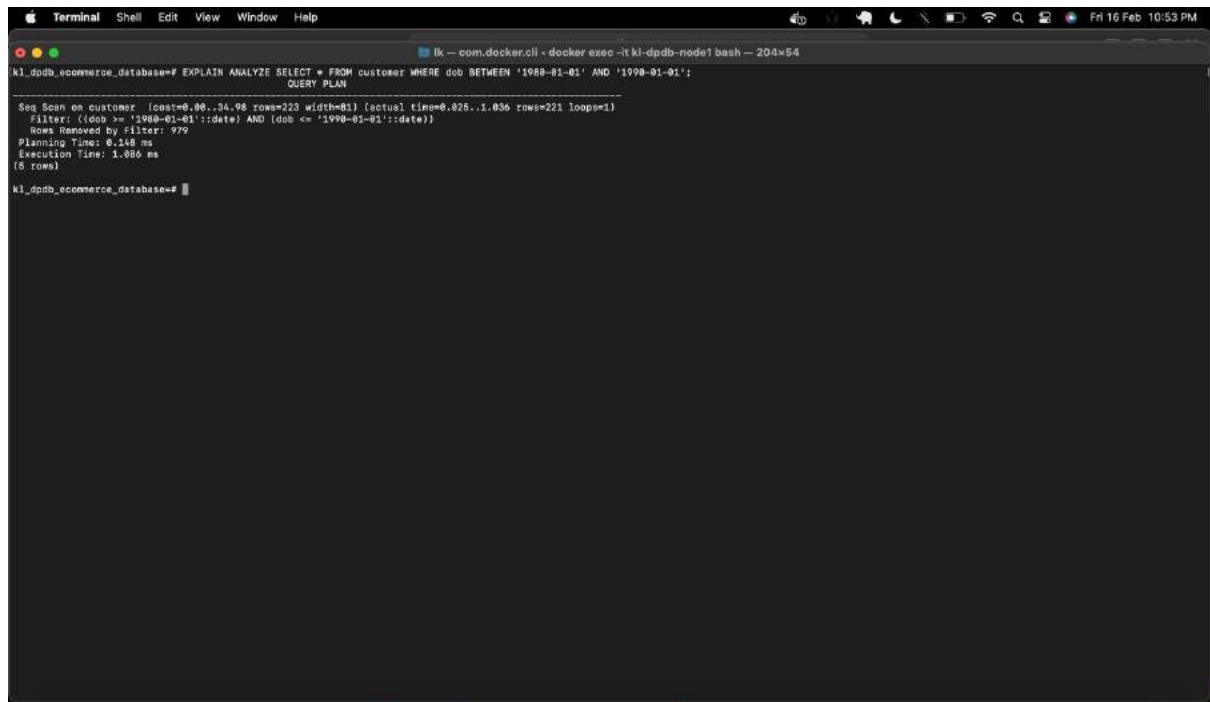
Explanation: There are total 1200 number of records in the customer table.

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';
```



```
kl_dpd_e-commerce_database=# EXPLAIN ANALYZE SELECT * FROM customer WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
                                QUERY PLAN
Seq Scan on customer (cost=0.00..34.98 rows=223 width=81) (actual time=0.025..1.036 rows=221 loops=1)
  Filter: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Rows Removed by Filter: 979
Planning Time: 0.148 ms
Execution Time: 1.086 ms
(0 rows)

kl_dpd_e-commerce_database=#
```

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
```

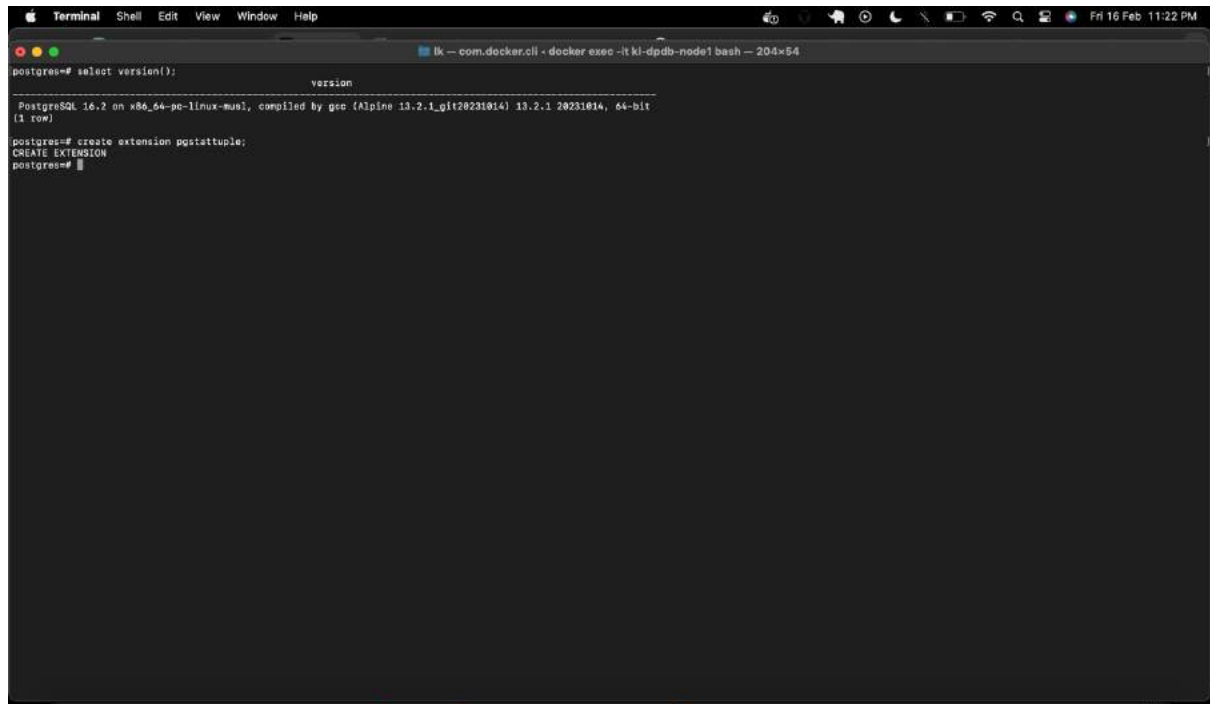
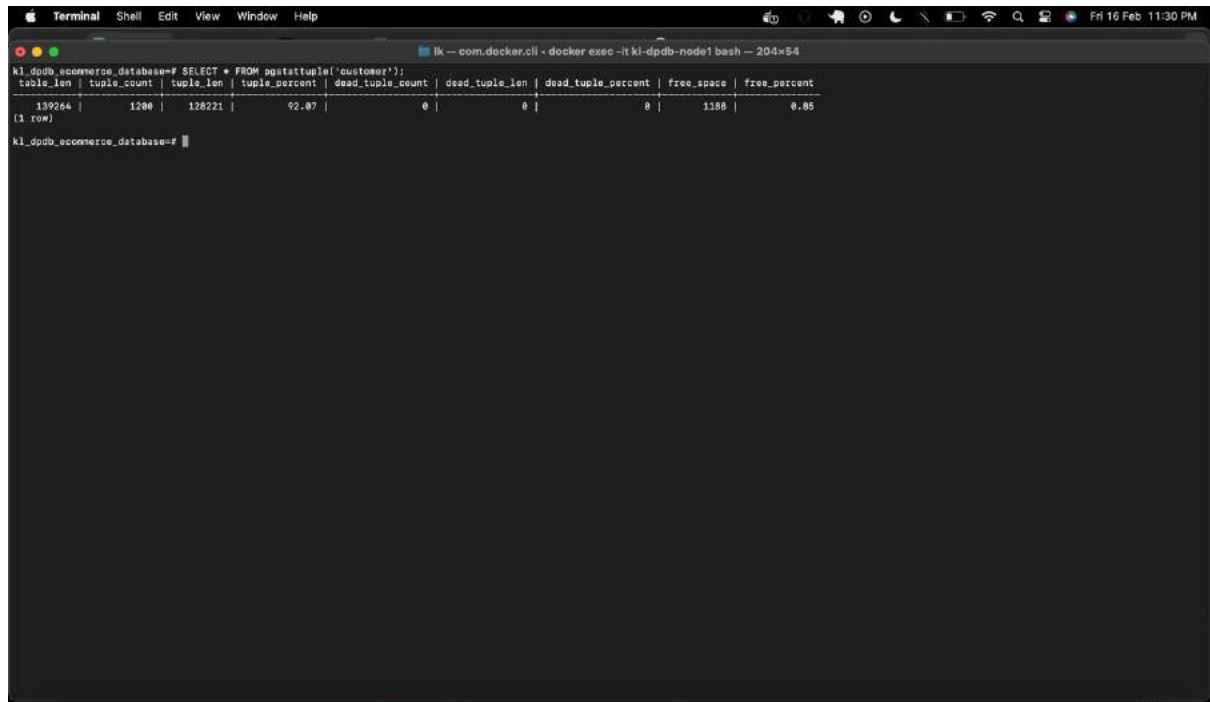
A terminal window titled 'Terminal' with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Fri 16 Feb 11:22 PM). The terminal shows a Docker container shell prompt 'fk -- com.docker.cli - docker exec -it kl-dpdb-node1 bash -- 204x54'. The user runs 'postgres=# select version();'. The output is a table with one row: 'PostgreSQL 16.2 on x86_64-pc-linux-mus, compiled by gcc (Alpine 13.2.1_git20231014) 13.2.1 20231014, 64-bit (1 row)'. The user then runs 'postgres=# create extension pgstattuple;'. The output is 'CREATE EXTENSION'. The prompt returns to 'postgres=#'.

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 “pgstattuple”. This was available from postgres-9.4 and later versions. To make use of pgstattuple we need to create the extension for this, so that we can evaluate the statistics of the table.

query:

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



```
kl_dadb_ecommerce_database=# SELECT * FROM pgstattuple('customer');
table_len | tuple_count | tuple_len | tuple_percent | dead_tuple_count | dead_tuple_len | dead_tuple_percent | free_space | free_percent
-----
139264 | 1200 | 128221 | 92.07 | 0 | 0 | 0 | 1188 | 0.85
(1 row)

kl_dadb_ecommerce_database=#
```

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

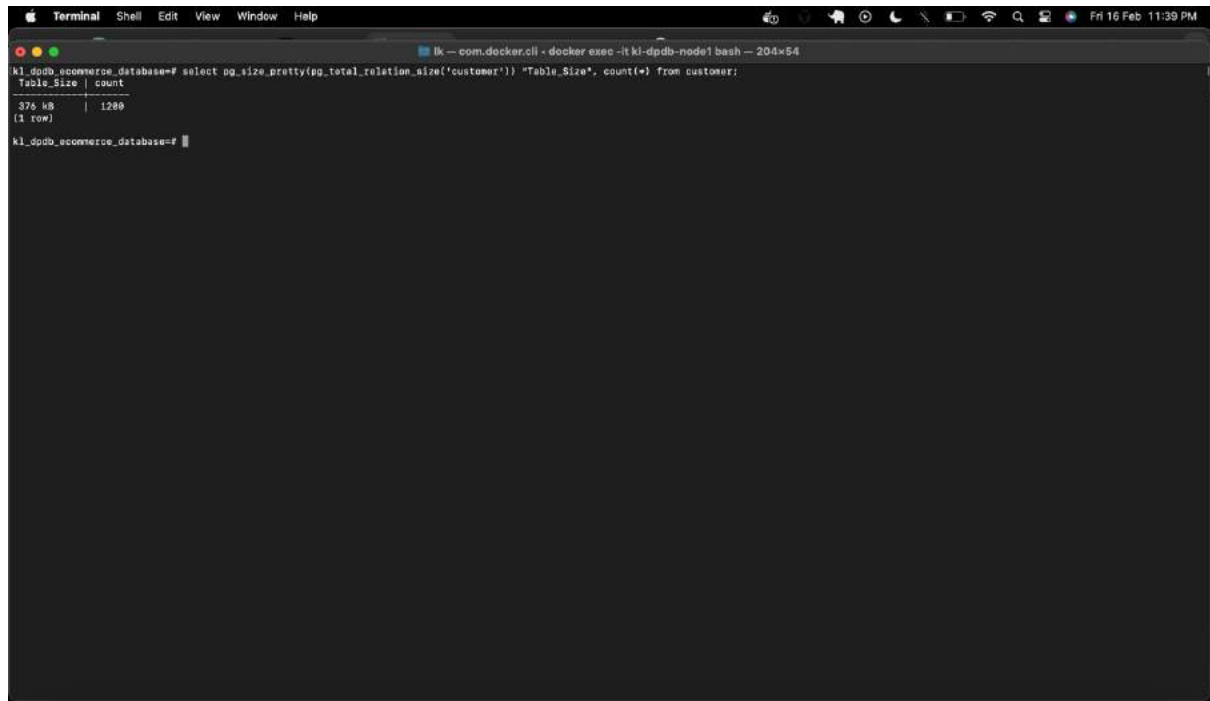
Explanation: using pgstattuple extracted the statistics of customer table tuples. The pgstattuple talks about the

- 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;
```



The image shows a terminal window with a dark background. The title bar at the top reads "Terminal Shell Edit View Window Help". Below the title bar, the terminal shows the command prompt "k1_dadb_ecommerce_database=#" followed by the SQL query: "select pg_size_pretty(pg_total_relation_size('customer')) 'Table_Size', count(*) from customer;". The output of the query is displayed as a table with two columns: "Table_Size" and "count". The first row shows "376 kB" for the table size and "1289" for the count. Below the table, it says "(1 row)". The prompt "k1_dadb_ecommerce_database=#" is shown again at the bottom of the terminal.

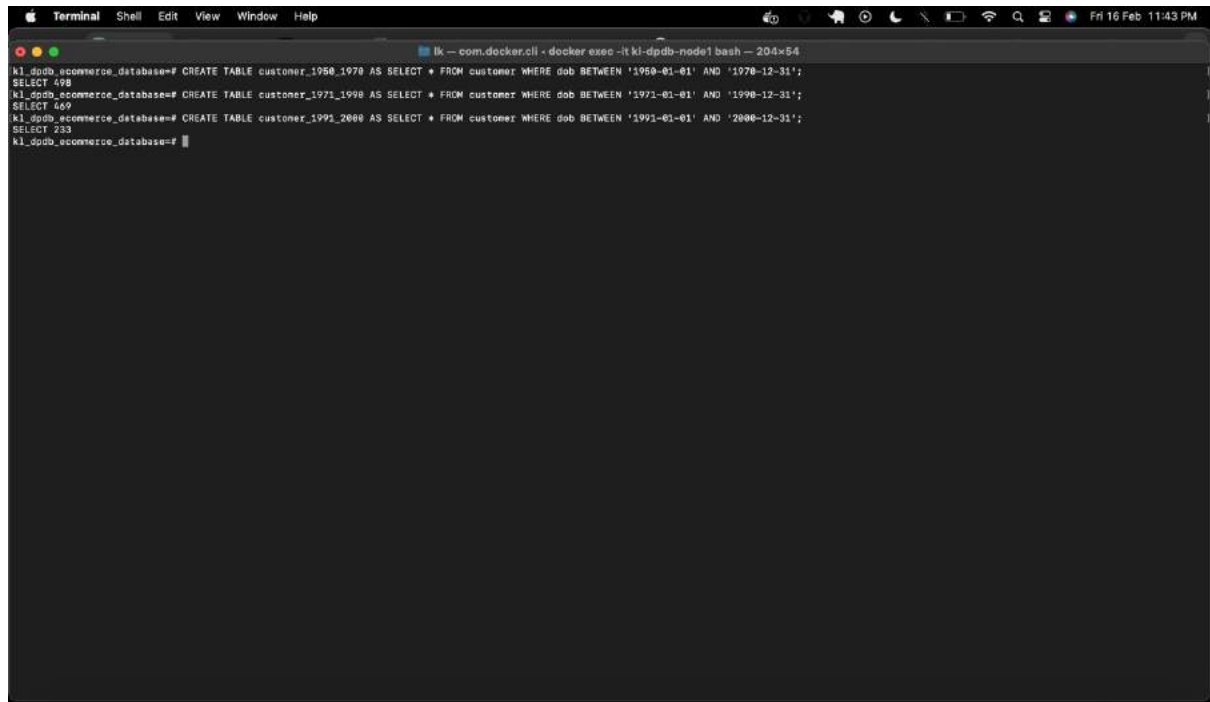
```
k1_dadb_ecommerce_database=# select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)  
from customer;  
Table_Size | count  
-----  
376 kB     | 1289  
(1 row)  
k1_dadb_ecommerce_database=#
```

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';
```

A screenshot of a terminal window with a dark background. The terminal title bar shows 'Terminal' and standard macOS window controls. The prompt is 'k1_dpd@ecommerce_database=#'. The first command is 'CREATE TABLE customer_1950_1970 AS SELECT * FROM customer WHERE dob BETWEEN '1950-01-01' AND '1970-12-31';', followed by 'SELECT * FROM customer_1950_1970;', which returns '498'. The second command is 'CREATE TABLE customer_1971_1990 AS SELECT * FROM customer WHERE dob BETWEEN '1971-01-01' AND '1990-12-31';', followed by 'SELECT * FROM customer_1971_1990;', which returns '469'. The third command is 'CREATE TABLE customer_1991_2000 AS SELECT * FROM customer WHERE dob BETWEEN '1991-01-01' AND '2000-12-31';', followed by 'SELECT * FROM customer_1991_2000;', which returns '233'. The terminal ends with a blank prompt line.

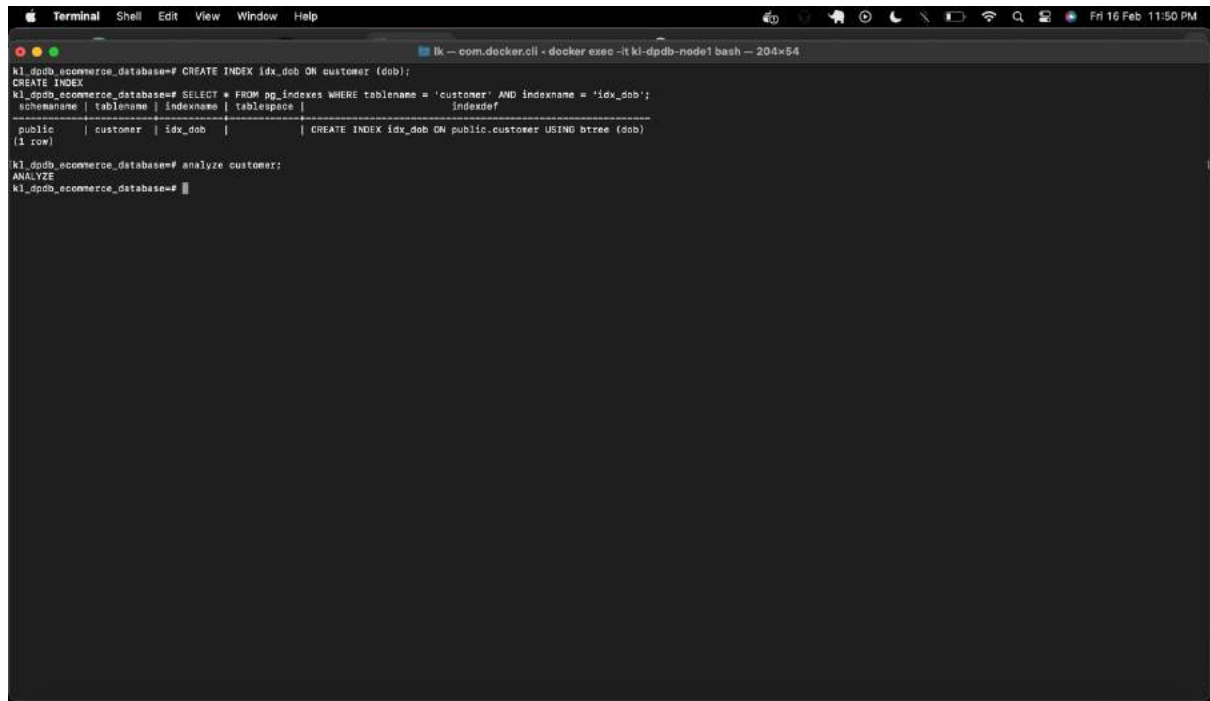
```
k1_dpd@ecommerce_database=# CREATE TABLE customer_1950_1970 AS SELECT * FROM customer WHERE dob BETWEEN '1950-01-01' AND '1970-12-31';
k1_dpd@ecommerce_database=# SELECT * FROM customer_1950_1970;
498
k1_dpd@ecommerce_database=# CREATE TABLE customer_1971_1990 AS SELECT * FROM customer WHERE dob BETWEEN '1971-01-01' AND '1990-12-31';
k1_dpd@ecommerce_database=# SELECT * FROM customer_1971_1990;
469
k1_dpd@ecommerce_database=# CREATE TABLE customer_1991_2000 AS SELECT * FROM customer WHERE dob BETWEEN '1991-01-01' AND '2000-12-31';
k1_dpd@ecommerce_database=# SELECT * FROM customer_1991_2000;
233
k1_dpd@ecommerce_database=#
```

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'
```



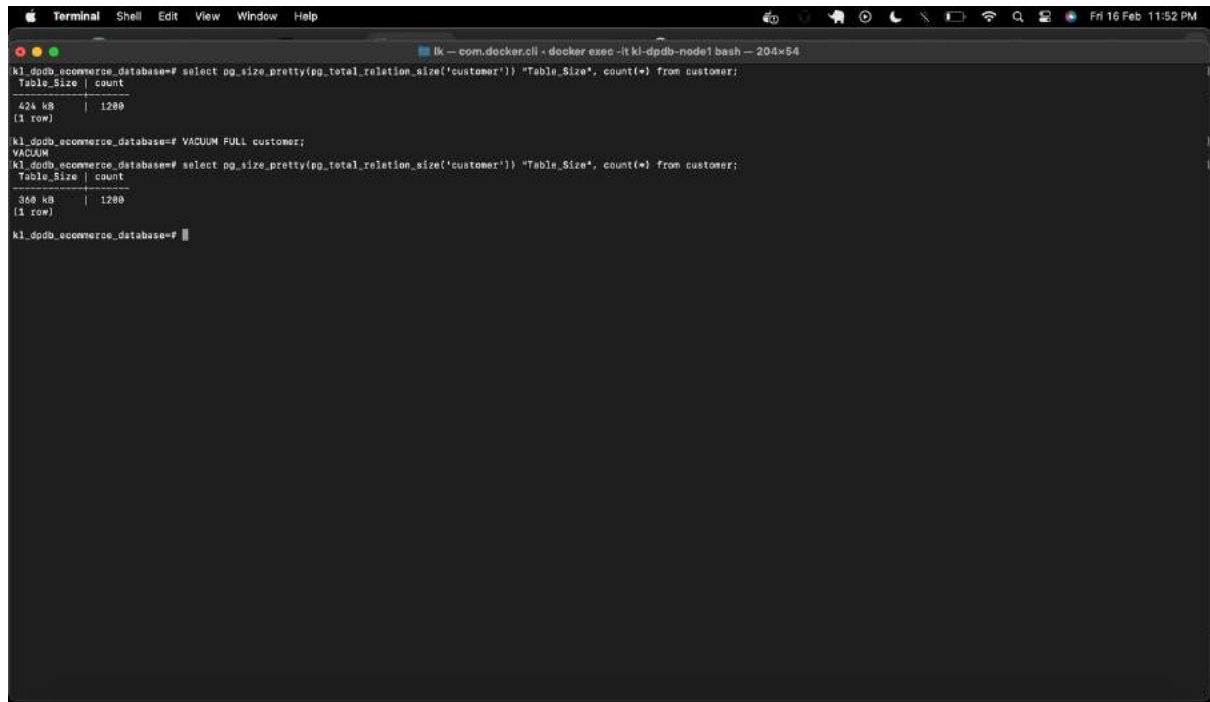
```
kl_dpdb_ecommerce_database=# CREATE INDEX idx_dob ON customer (dob);  
CREATE INDEX  
kl_dpdb_ecommerce_database=# SELECT * FROM pg_indexes WHERE tablename = 'customer' AND indexname = 'idx_dob';  
-----  
 schemaname | tablename | indexname | tablespace | indexdef  
-----  
 public    | customer | idx_dob   |             | CREATE INDEX idx_dob ON public.customer USING btree (dob)  
(1 row)  
  
kl_dpdb_ecommerce_database=# analyze customer;  
ANALYZE  
kl_dpdb_ecommerce_database=#
```

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

A terminal window titled "Terminal" with a dark background. The terminal shows a series of commands and their outputs. The first command is a SQL query to get the size of the 'customer' table, which returns 424 KB. The second command is 'VACUUM FULL customer;', followed by another SQL query to get the size of the 'customer' table, which returns 368 KB. The terminal window has a title bar with standard macOS window controls and a menu bar with "Terminal", "Shell", "Edit", "View", "Window", and "Help". The status bar at the bottom shows the date and time as "Fri 16 Feb 11:52 PM".

```
kl_dpd@ecommerce_database=# select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)
from customer;
Table_Size | count
-----
424 KB | 1280
(1 row)

kl_dpd@ecommerce_database=# VACUUM FULL customer;
VACUUM
kl_dpd@ecommerce_database=# select pg_size_pretty(pg_total_relation_size('customer')) "Table_Size", count(*)
from customer;
Table_Size | count
-----
368 KB | 1280
(1 row)

kl_dpd@ecommerce_database=#
```

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 VACUUM is used to perform the defragmentation or the optimization. Here also to optimize the size after fragmentation performed VACUUM on the customer table by that we can observe the size of the table is 360KB. The results look better after the fragmentation.

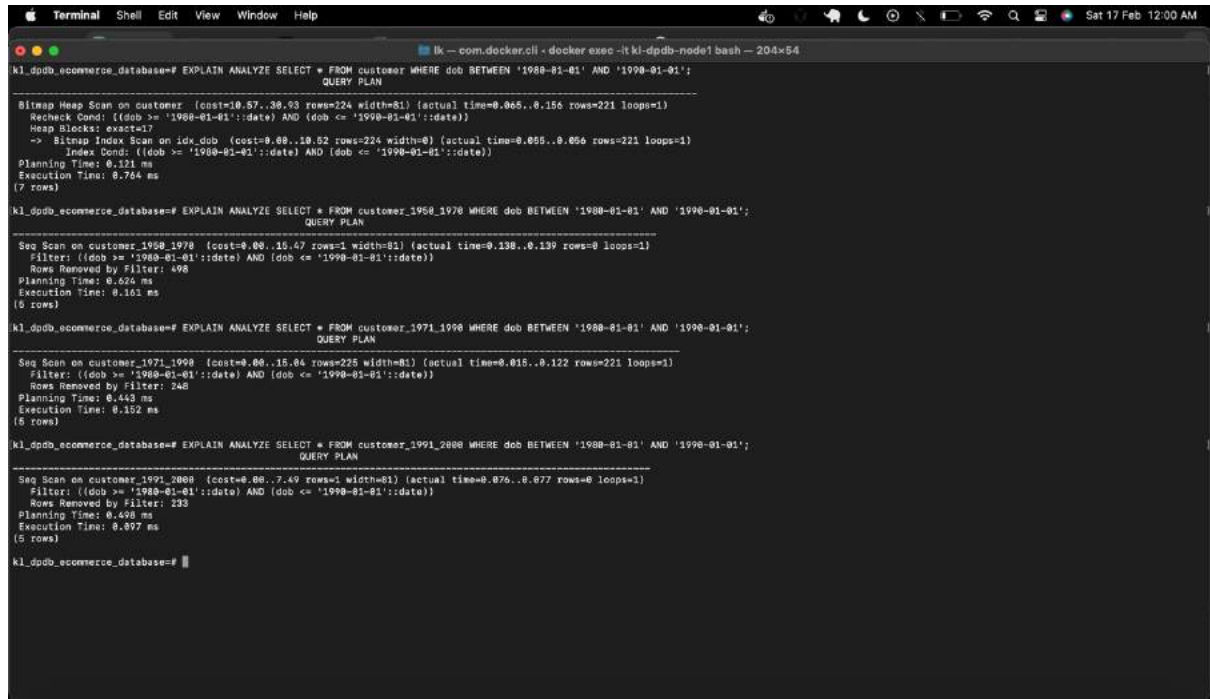
query:

```
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';
```



```
kl_dpd_b_e-commerce_database=# EXPLAIN ANALYZE SELECT * FROM customer WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
QUERY PLAN
-----
Bitmap Heap Scan on customer (cost=10.57..38.93 rows=224 width=81) (actual time=0.065..0.156 rows=221 loops=1)
  Recheck Cond: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Heap Blocks: exact=17
    -> Bitmap Index Scan on idx_dob (cost=0.00..19.52 rows=224 width=0) (actual time=0.055..0.056 rows=221 loops=1)
      Index Cond: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Planning Time: 0.321 ms
  Execution Time: 0.764 ms
(7 rows)

kl_dpd_b_e-commerce_database=# EXPLAIN ANALYZE SELECT * FROM customer_1950_1970 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
QUERY PLAN
-----
Seq Scan on customer_1950_1970 (cost=0.00..15.47 rows=1 width=81) (actual time=0.138..0.139 rows=0 loops=1)
  Filter: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Rows Removed by Filter: 498
  Planning Time: 0.626 ms
  Execution Time: 0.161 ms
(5 rows)

kl_dpd_b_e-commerce_database=# EXPLAIN ANALYZE SELECT * FROM customer_1971_1990 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
QUERY PLAN
-----
Seq Scan on customer_1971_1990 (cost=0.00..15.64 rows=225 width=81) (actual time=0.015..0.122 rows=221 loops=1)
  Filter: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Rows Removed by Filter: 246
  Planning Time: 0.443 ms
  Execution Time: 0.152 ms
(6 rows)

kl_dpd_b_e-commerce_database=# EXPLAIN ANALYZE SELECT * FROM customer_1991_2000 WHERE dob BETWEEN '1980-01-01' AND '1990-01-01';
QUERY PLAN
-----
Seq Scan on customer_1991_2000 (cost=0.00..7.49 rows=1 width=81) (actual time=0.076..0.077 rows=0 loops=1)
  Filter: ((dob >= '1980-01-01'::date) AND (dob <= '1990-01-01'::date))
  Rows Removed by Filter: 233
  Planning Time: 0.698 ms
  Execution Time: 0.077 ms
(5 rows)

kl_dpd_b_e-commerce_database=#
```

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.

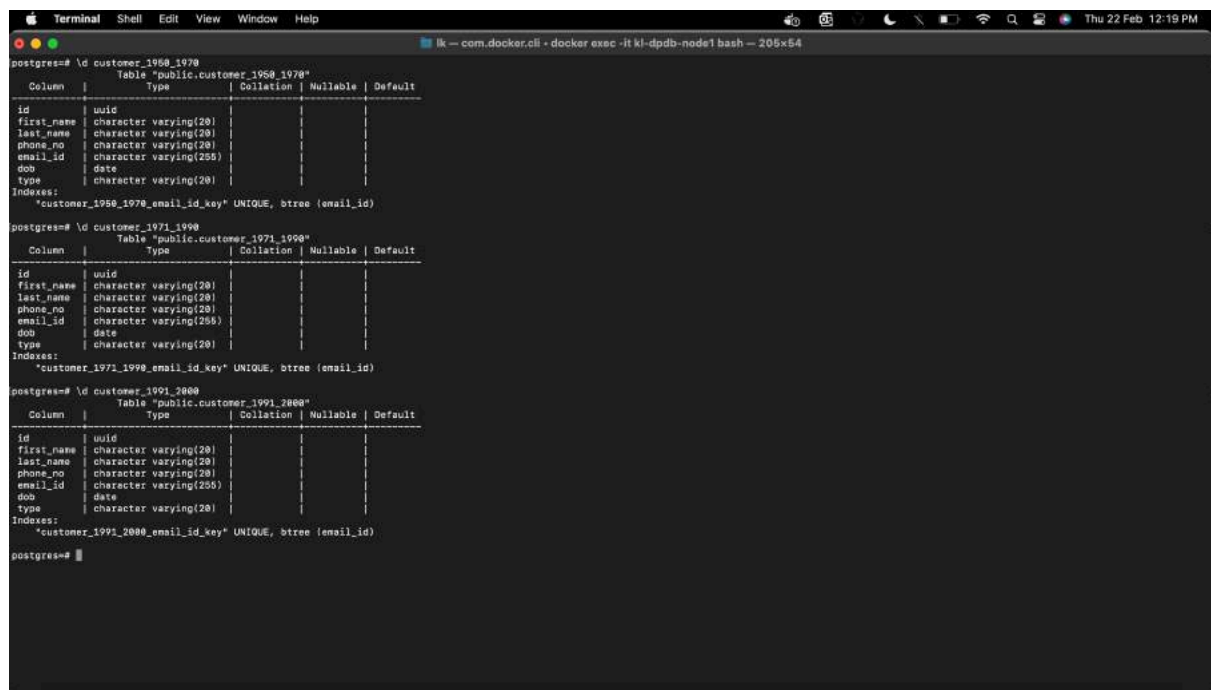
query:

Reviewing the fragment tables-

```
\d customer_1950_1970
```

```
\d customer_1971_1990
```

```
\d customer_1991_2000
```



```
postgres=# \d customer_1950_1970
Table "public.customer_1950_1970"
Column |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid                   |           |          |
first_name | character varying(20) |           |          |
last_name | character varying(20) |           |          |
phone_no | character varying(20) |           |          |
email_id | character varying(255) |           |          |
dob      | date                   |           |          |
type     | character varying(20) |           |          |
Indexes:
  "customer_1950_1970_email_id_key" UNIQUE, btree (email_id)

postgres=# \d customer_1971_1990
Table "public.customer_1971_1990"
Column |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid                   |           |          |
first_name | character varying(20) |           |          |
last_name | character varying(20) |           |          |
phone_no | character varying(20) |           |          |
email_id | character varying(255) |           |          |
dob      | date                   |           |          |
type     | character varying(20) |           |          |
Indexes:
  "customer_1971_1990_email_id_key" UNIQUE, btree (email_id)

postgres=# \d customer_1991_2000
Table "public.customer_1991_2000"
Column |          Type          | Collation | Nullable | Default
-----|-----|-----|-----|-----
id      | uuid                   |           |          |
first_name | character varying(20) |           |          |
last_name | character varying(20) |           |          |
phone_no | character varying(20) |           |          |
email_id | character varying(255) |           |          |
dob      | date                   |           |          |
type     | character varying(20) |           |          |
Indexes:
  "customer_1991_2000_email_id_key" UNIQUE, btree (email_id)

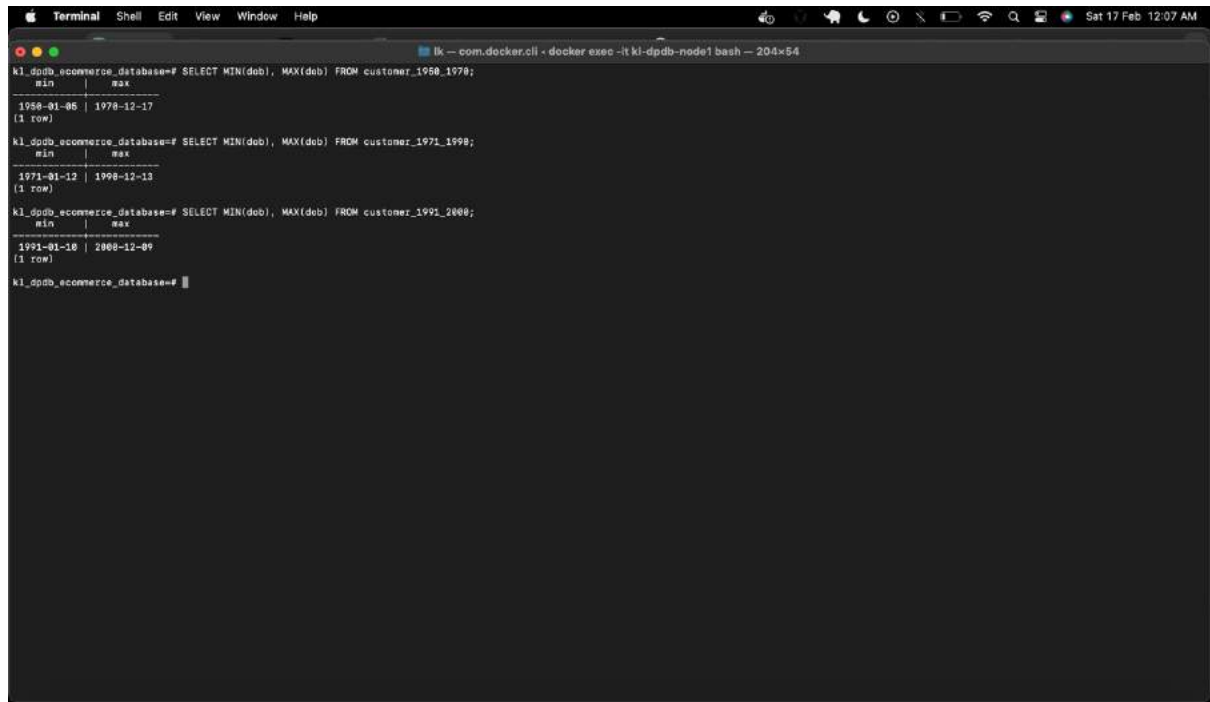
postgres=#
```

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;
```



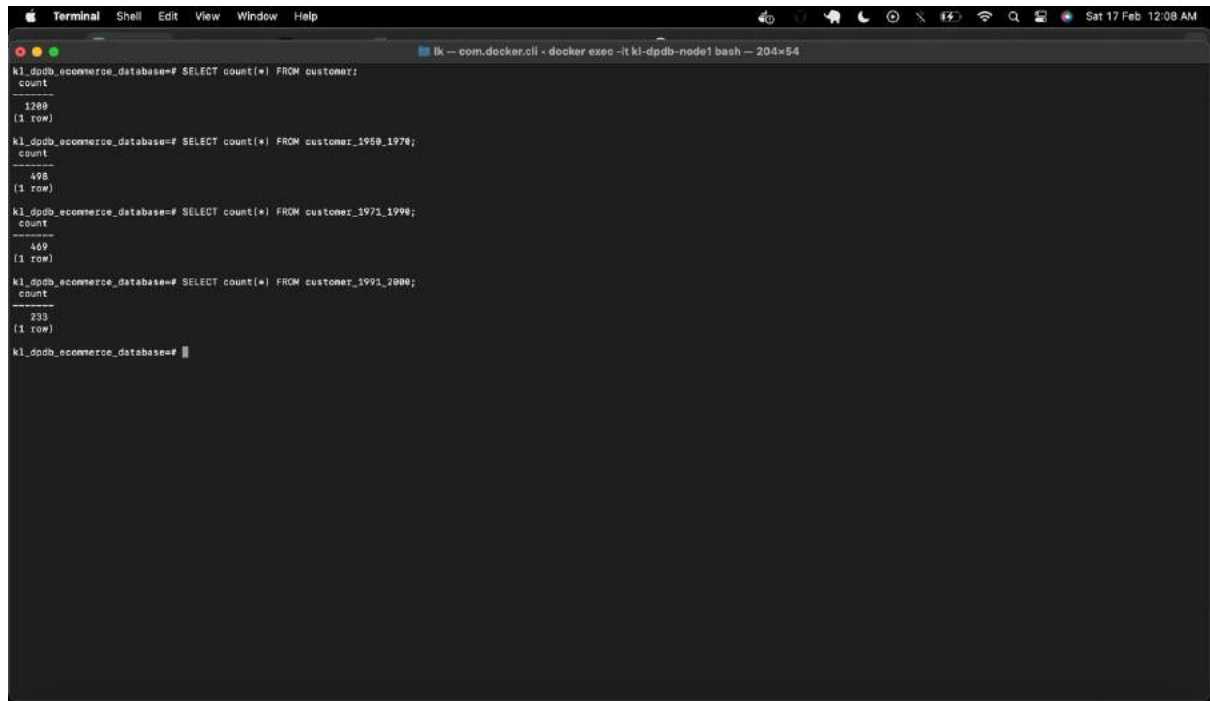
```
kl_dpdb_ecommerce_database=# SELECT MIN(dob), MAX(dob) FROM customer_1950_1970;  
min | max  
-----  
1950-01-05 | 1970-12-17  
(1 row)  
  
kl_dpdb_ecommerce_database=# SELECT MIN(dob), MAX(dob) FROM customer_1971_1990;  
min | max  
-----  
1971-01-12 | 1990-12-13  
(1 row)  
  
kl_dpdb_ecommerce_database=# SELECT MIN(dob), MAX(dob) FROM customer_1991_2000;  
min | max  
-----  
1991-01-10 | 2000-12-09  
(1 row)  
  
kl_dpdb_ecommerce_database=#
```

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;
```

A terminal window titled 'Terminal' with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Sat 17 Feb 12:08 AM). The terminal shows four SQL queries and their results. The first query counts records in the 'customer' table, returning 1200. The second query counts records in 'customer_1950_1970', returning 498. The third query counts records in 'customer_1971_1990', returning 469. The fourth query counts records in 'customer_1991_2000', returning 233. The results are displayed in a table-like format with 'count' as the column header and the count value as the result.

```
k1_dpdb_ecommerce_database=# SELECT count(*) FROM customer;  
count  
-----  
1200  
(1 row)  
  
k1_dpdb_ecommerce_database=# SELECT count(*) FROM customer_1950_1970;  
count  
-----  
498  
(1 row)  
  
k1_dpdb_ecommerce_database=# SELECT count(*) FROM customer_1971_1990;  
count  
-----  
469  
(1 row)  
  
k1_dpdb_ecommerce_database=# SELECT count(*) FROM customer_1991_2000;  
count  
-----  
233  
(1 row)  
  
k1_dpdb_ecommerce_database=#
```

Fig-41: Count of records in fragmented tables.

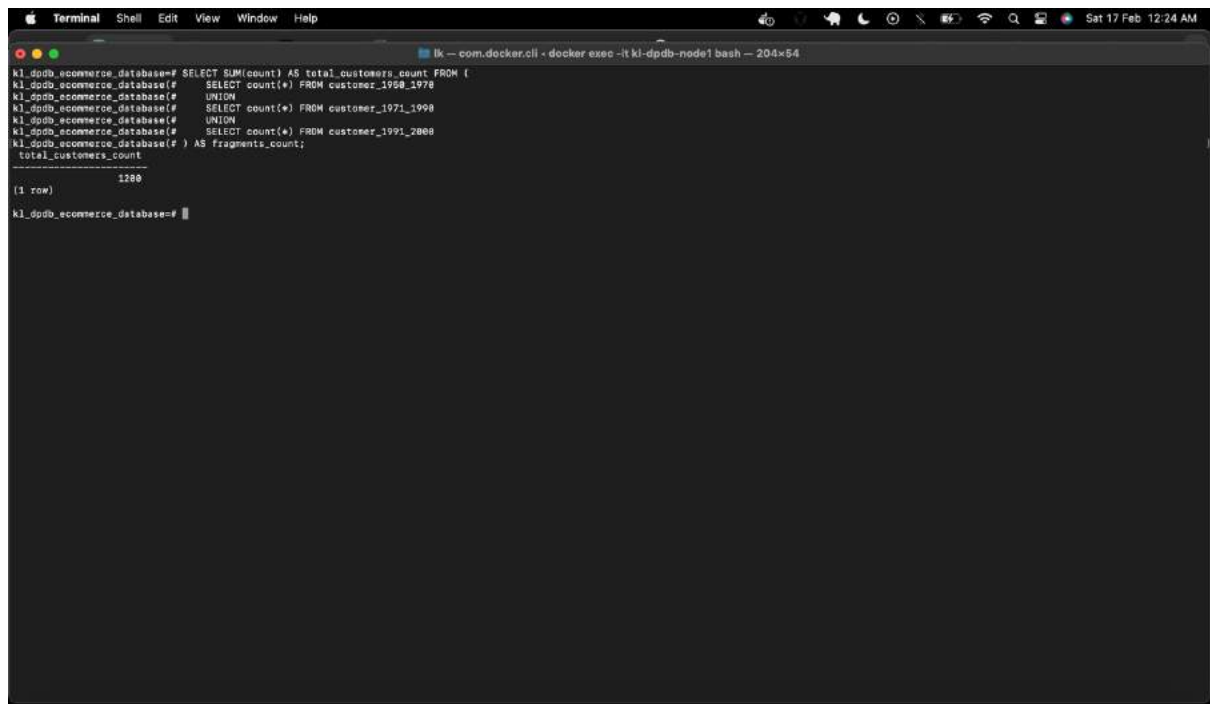
Explanation: The total count of all the fragmented tables is equal to the count of records in the customer table. The customer table has 1200 records, where as customer_1950_1970 has 498, customer_1971_1990 has 469 and customer_1991_2000 has 233 which satisfies the count of records.

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;
```

A screenshot of a macOS Terminal window. The title bar shows 'Terminal' and standard window controls. The terminal prompt is 'k1_dpd@ecommerce_database:~\$'. The user enters a SQL query: '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;'. The output shows a single row with the value '1280'. The prompt returns to 'k1_dpd@ecommerce_database:~\$'.

```
k1_dpd@ecommerce_database:~$ SELECT SUM(count) AS total_customers_count FROM (
k1_dpd@ecommerce_database:~$ SELECT count(*) FROM customer_1950_1970
k1_dpd@ecommerce_database:~$ UNION
k1_dpd@ecommerce_database:~$ SELECT count(*) FROM customer_1971_1990
k1_dpd@ecommerce_database:~$ UNION
k1_dpd@ecommerce_database:~$ SELECT count(*) FROM customer_1991_2000
k1_dpd@ecommerce_database:~$ ) AS fragments_count;
+-----+
| total_customers_count |
+-----+
| 1280                  |
+-----+
(1 row)
k1_dpd@ecommerce_database:~$
```

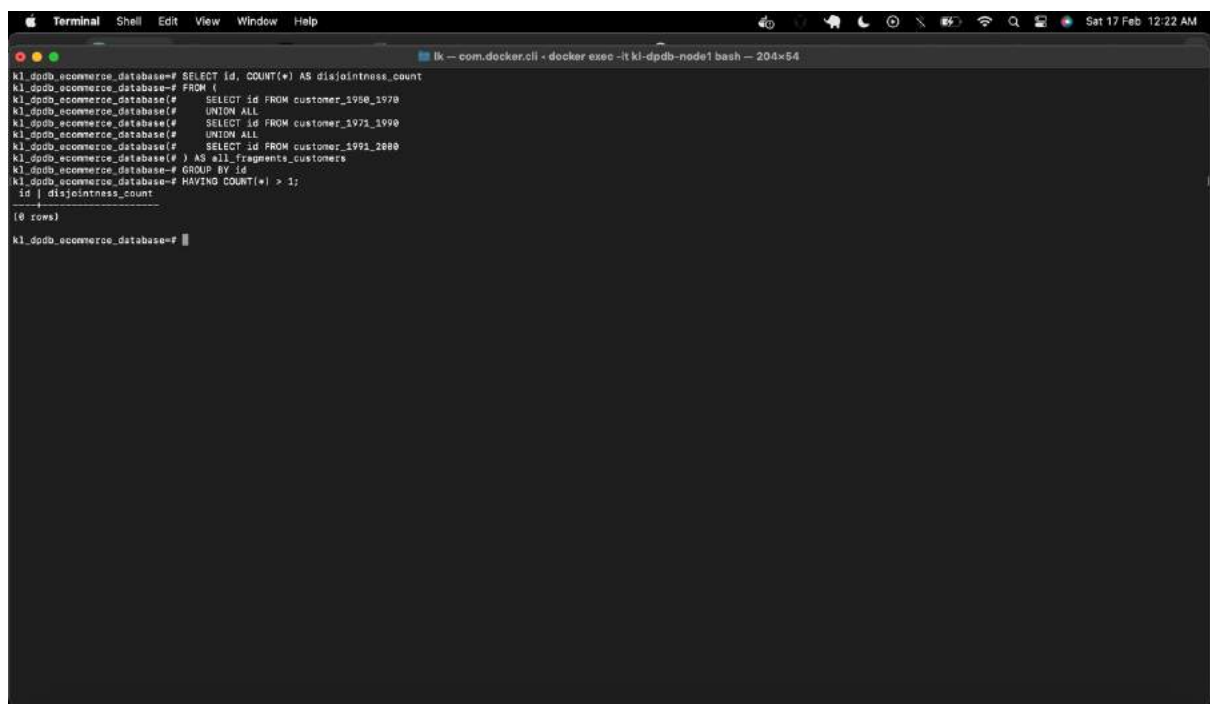
Fig-42: Reconstruction of fragmented tables.

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;
```



```
kl_dpdb_ecommerce_database=# SELECT id, COUNT(*) AS disjointness_count
kl_dpdb_ecommerce_database=# FROM (
kl_dpdb_ecommerce_database=#     SELECT id FROM customer_1950_1970
kl_dpdb_ecommerce_database=#     UNION ALL
kl_dpdb_ecommerce_database=#     SELECT id FROM customer_1971_1990
kl_dpdb_ecommerce_database=#     UNION ALL
kl_dpdb_ecommerce_database=#     SELECT id FROM customer_1991_2000
kl_dpdb_ecommerce_database=# ) AS all_fragments_customers
kl_dpdb_ecommerce_database=# GROUP BY id
kl_dpdb_ecommerce_database=# HAVING COUNT(*) > 1;
 id | disjointness_count
----+-----
(0 rows)

kl_dpdb_ecommerce_database=#
```

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:

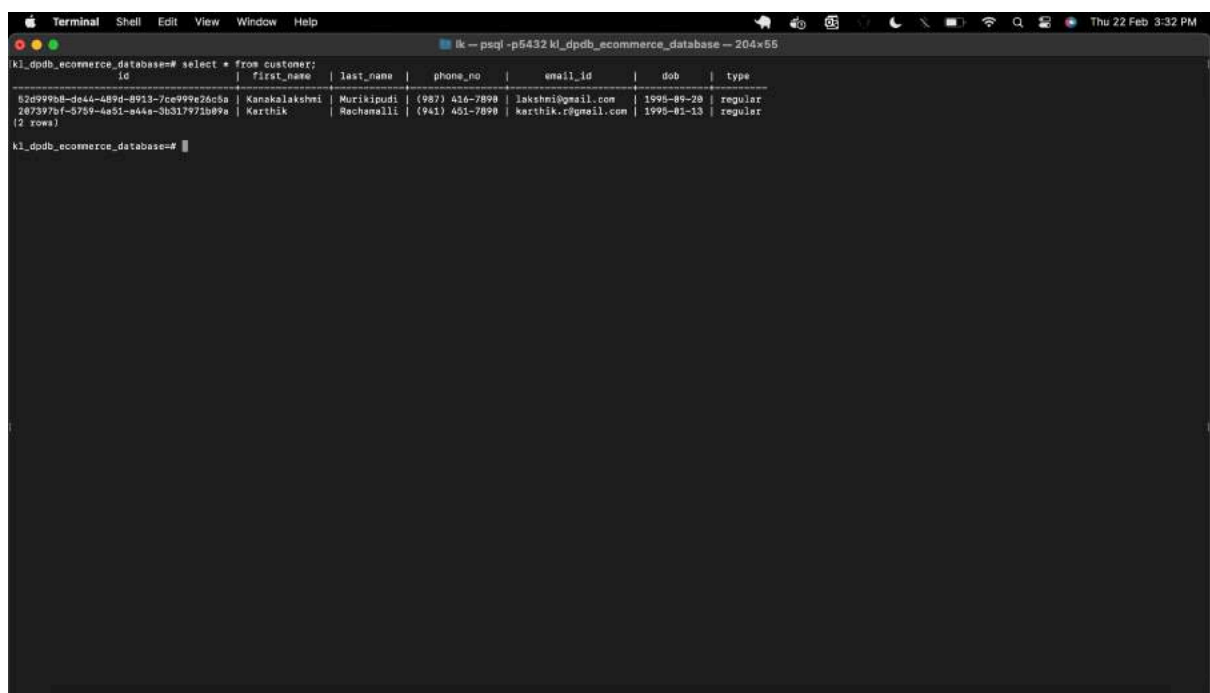
query:

```
select * from customer;
```

available customers-

52d999b8-de44-489d-8913-7ce999e26c5a

207397bf-5759-4a51-a44a-3b317971b09a



The screenshot shows a terminal window with the PostgreSQL command prompt. The query `select * from customer;` has been executed, and the results are displayed in a table format. The table has columns for `id`, `first_name`, `last_name`, `phone_no`, `email_id`, `dob`, and `type`. There are two rows of data.

id	first_name	last_name	phone_no	email_id	dob	type
52d999b8-de44-489d-8913-7ce999e26c5a	Kanakalakshmi	Murikupudi	(987) 414-7898	lakshmi@gmail.com	1995-09-28	regular
207397bf-5759-4a51-a44a-3b317971b09a	Karthik	Rachamalli	(941) 451-7898	karthik.r@gmail.com	1995-01-13	regular

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:

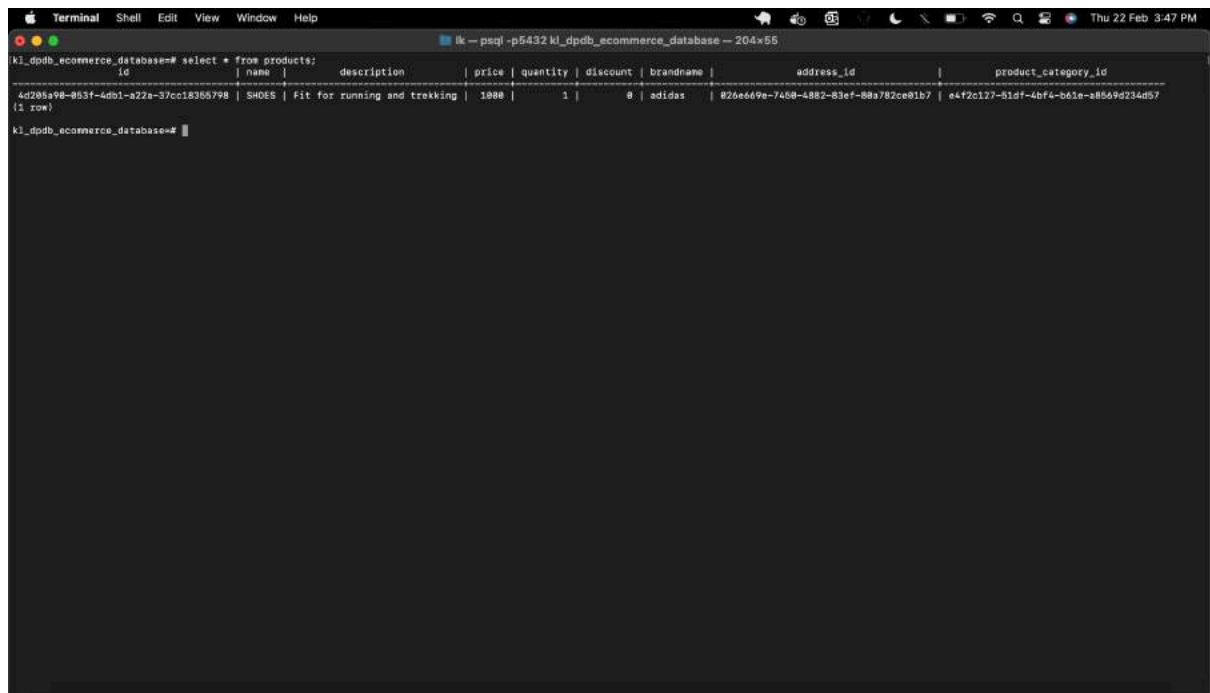
```
select * from products;
```

available products :

4d205a90-053f-4db1-a22a-37cc18355798

name - SHOES

quantity - 1



A terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help) and a status bar (ik - psql - p5432 kl_dpdb_ecommerce_database - 204x55). The terminal shows a SQL query being executed in a PostgreSQL database. The query is "select * from products;". The result is displayed as a table with 8 columns: id, name, description, price, quantity, discount, brandname, address_id, and product_category_id. The first row of data is for the product with id "4d205a90-053f-4db1-a22a-37cc18355798", which is a pair of shoes, priced at 1000, with a quantity of 1, no discount, brand "adidas", and specific address and category IDs. The terminal also shows "(1 row)" and the prompt "kl_dpdb_ecommerce_database=#".

id	name	description	price	quantity	discount	brandname	address_id	product_category_id
4d205a90-053f-4db1-a22a-37cc18355798	SHOES	Fit for running and trekking	1000	1	0	adidas	026e669e-7458-4882-83ef-88a782ce81b7	e4f2c177-81df-4bf6-b61e-a8569d734d57

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;

```

```

k1_dadb_ecommerce_database=# CREATE OR REPLACE FUNCTION place_order(
k1_dadb_ecommerce_database=#     p_product_id UUID,
k1_dadb_ecommerce_database=#     p_customer_id UUID,
k1_dadb_ecommerce_database=#     p_quantity INTEGER,
k1_dadb_ecommerce_database=#     p_delivery_partner_name VARCHAR(50),
k1_dadb_ecommerce_database=#     p_delivery_partner_phone_no VARCHAR(20),
k1_dadb_ecommerce_database=#     p_delivery_partner_email VARCHAR(255)
k1_dadb_ecommerce_database=# ) RETURNS VOID AS $$
k1_dadb_ecommerce_database=# DECLARE
k1_dadb_ecommerce_database=#     v_order_id UUID;
k1_dadb_ecommerce_database=#     v_delivery_partner_id UUID;
k1_dadb_ecommerce_database=#     v_product_quantity INTEGER;
k1_dadb_ecommerce_database=# BEGIN
k1_dadb_ecommerce_database=#     -- Check if the product is available
k1_dadb_ecommerce_database=#     SELECT quantity INTO v_product_quantity FROM products WHERE id = p_product_id FOR UPDATE;
k1_dadb_ecommerce_database=#     -- step-1: Check if the requested quantity is available
k1_dadb_ecommerce_database=#     IF v_product_quantity < p_quantity THEN
k1_dadb_ecommerce_database=#         RAISE EXCEPTION 'Error in place_order: Product not available. Please try again later.';
k1_dadb_ecommerce_database=#     END IF;
k1_dadb_ecommerce_database=#     -- Step 2: Insert data into orders
k1_dadb_ecommerce_database=#     INSERT INTO orders (status, order_date, address_id, customer_id)
k1_dadb_ecommerce_database=#     VALUES ('Processing', CURRENT_DATE, (SELECT address_id FROM customer_address WHERE customer_id = p_customer_id AND default_address = true), p_customer_id)
k1_dadb_ecommerce_database=#     RETURNING id INTO v_order_id;
k1_dadb_ecommerce_database=#     -- Step 3: Update cart, product cart, and products tables
k1_dadb_ecommerce_database=#     UPDATE products SET quantity = quantity - p_quantity WHERE id = p_product_id;
k1_dadb_ecommerce_database=#     -- Step 4: Insert data into transaction summary table
k1_dadb_ecommerce_database=#     INSERT INTO transaction_summary (total_amount_paid, payment_type, date_of_payment, order_id)
k1_dadb_ecommerce_database=#     VALUES ((SELECT price * p_quantity FROM products WHERE id = p_product_id), 'Credit Card', CURRENT_DATE, v_order_id);
k1_dadb_ecommerce_database=#     -- Step 5: Insert data into orders_products table
k1_dadb_ecommerce_database=#     INSERT INTO orders_products (order_id, product_id, quantity) VALUES (v_order_id, p_product_id, p_quantity);
k1_dadb_ecommerce_database=#     -- Step 6: Insert data into delivery_partner table
k1_dadb_ecommerce_database=#     INSERT INTO delivery_partner (name, phone_no, email, order_id)
k1_dadb_ecommerce_database=#     VALUES (p_delivery_partner_name, p_delivery_partner_phone_no, p_delivery_partner_email, v_order_id)
k1_dadb_ecommerce_database=#     RETURNING id INTO v_delivery_partner_id;
k1_dadb_ecommerce_database=#     -- Step 7: Insert data into customer_delivery_partner table
k1_dadb_ecommerce_database=#     INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id) VALUES (p_customer_id, v_delivery_partner_id);
k1_dadb_ecommerce_database=# EXCEPTION
k1_dadb_ecommerce_database=#     WHEN OTHERS THEN
k1_dadb_ecommerce_database=#         -- Rollback the transaction in case of an exception
k1_dadb_ecommerce_database=#         RAISE EXCEPTION 'Error in place_order: %', SQLERRM;
k1_dadb_ecommerce_database=#     END;
k1_dadb_ecommerce_database=# END;
k1_dadb_ecommerce_database=# $$ LANGUAGE plpgsql;
k1_dadb_ecommerce_database=# CREATE FUNCTION
k1_dadb_ecommerce_database=# 1

```

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)

```

```

Terminal Shell Edit View Window Help
lakshmi_OPDB -- -bash-- 204x53
[Kanakalakshmi lakshmi_OPDB]$ cat k1_dpd_req-4.py
import psycopg2
from concurrent.futures import ThreadPoolExecutor

# connection details
host = "localhost"
port = 5432
database = "k1_dpd ecommerce database"
user = "postgres"
password = "spaceman1234"

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, 'roby', '(917) 777-4089', 'roby@example.com')
    # customer-2
    customer_2_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '207397bf-5759-4a51-a44a-3b317971b09a', 1, 'joseph_e', '(918) 717-4389', 'joseph1@example.com')

    with ThreadPoolExecutor(max_workers=2) as executor:
        # Trigger the function concurrently with different user inputs

```

Fig-47: Script to execute concurrent orders.

```
Terminal Shell Edit View Window Help
lakshmi_DPDB -- -bash -- 204x53

# connection details
host = "localhost"
port = 5432
database = "l_dpdb_ecommerce_database"
user = "postgres"
password = "spaceman1234"

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 = ('4d285e90-853f-4db1-a22a-37cc18355798', '82d999b8-de44-489d-8913-7ce999e26cda', 1, 'roby', '(917) 777-4889', 'rob1@example.com')
    # Customer-2
    customer_2_details = ('4d285e90-853f-4db1-a22a-37cc18355798', '287397bf-5759-4a51-e44a-3b317971b89a', 1, 'Joseph_a', '(918) 717-4389', 'joseph19@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)

(Kanakalakshmi lakshmi_DPDB)$
```

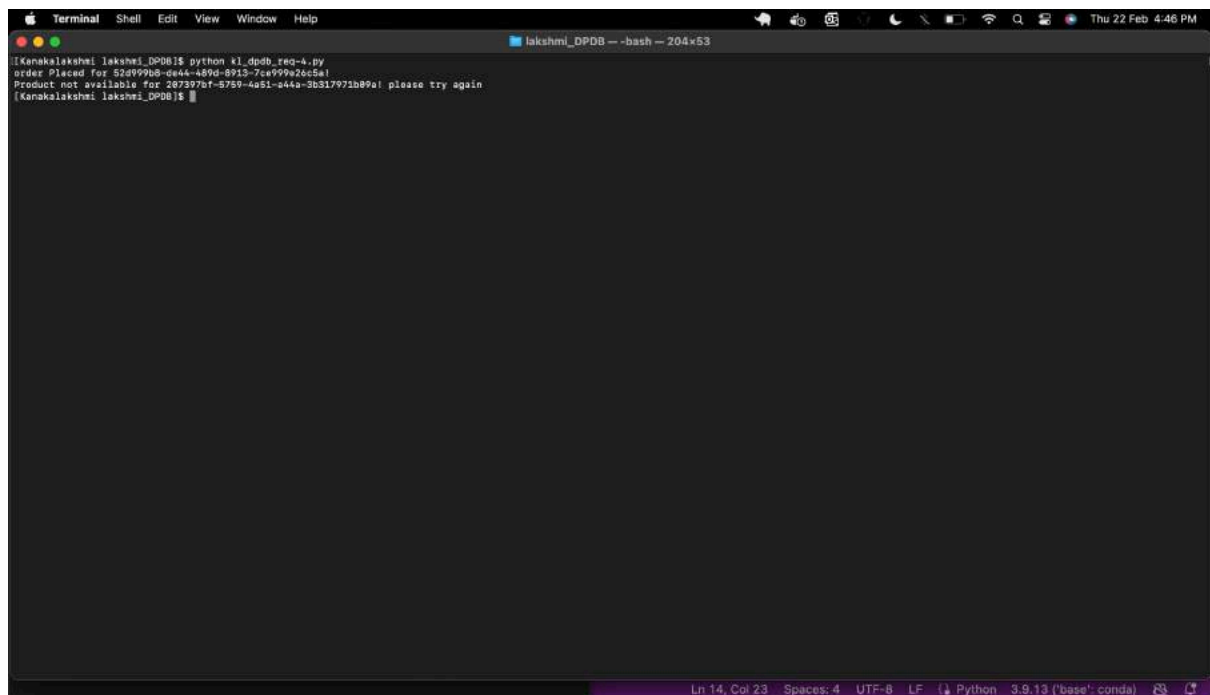
Fig-48: Continuation of the Script to execute concurrent orders.

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.futures**

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 postgres 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

A screenshot of a macOS Terminal window. The title bar shows 'Terminal' and standard menu items. The window title is 'lakshmi_DPDB -- -bash -- 204x53'. The prompt is '[Kanakalakshmi lakshmi_DPDB]\$. The command 'python kl_dpdb_req-4.py' has been executed. The output shows an order being placed for a specific user ID, followed by a rejection for another user ID due to product unavailability. The prompt returns to '\$'.

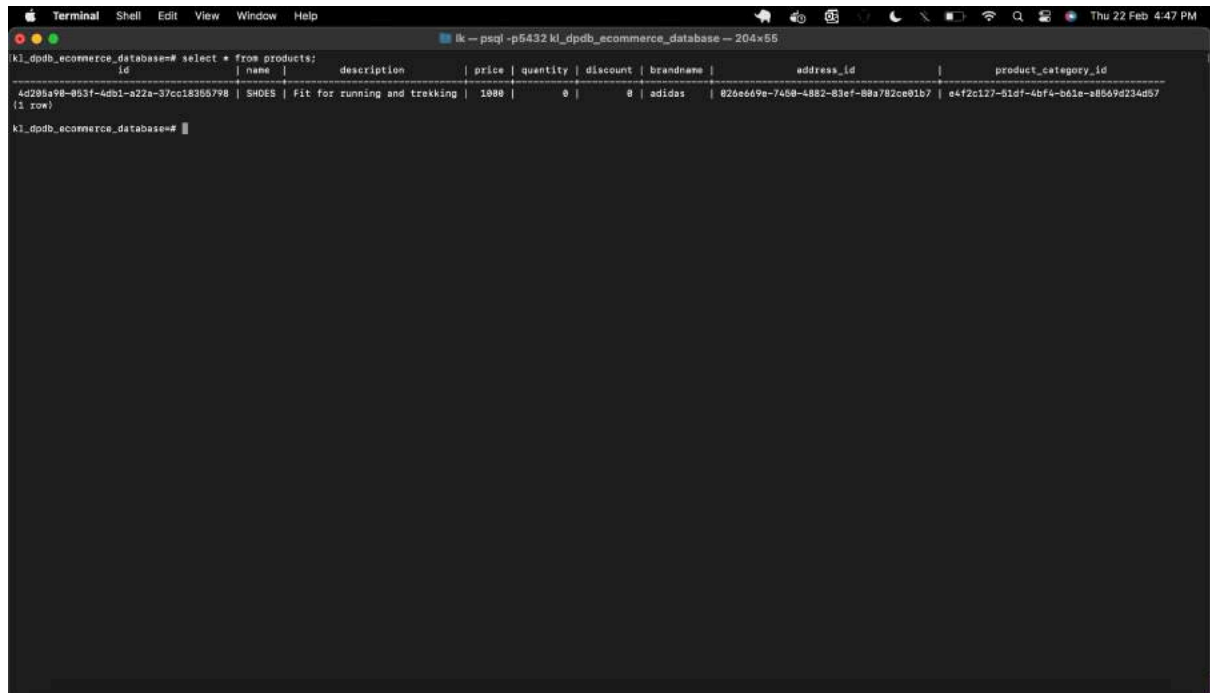
```
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 52d999b8-de44-489d-8913-7ce999e26c5a!
Product not available for 207397bf-5759-4a51-a44a-3b317971b09a! please try again
[Kanakalakshmi lakshmi_DPDB]$
```

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;
```



The image shows a terminal window with a dark background. The title bar at the top reads "Terminal" and "Shell". The terminal prompt is "k1_dadb_ecommerce_database=#". The user has entered the SQL query "select * from products;". The output is a table with 10 columns: id, name, description, price, quantity, discount, brandname, address_id, and product_category_id. The first row of data shows a product with id "4d295a98-853f-4db1-a22a-37cc18365798", name "SHOES", description "Fit for running and trekking", price "1000", quantity "0", discount "8", brandname "adidas", address_id "826e669e-7458-4882-83ef-88a782ce81b7", and product_category_id "e4f2c127-51df-4bf4-b61e-s8569d234d57". The output indicates "(1 row)". The terminal prompt is now "k1_dadb_ecommerce_database=#".

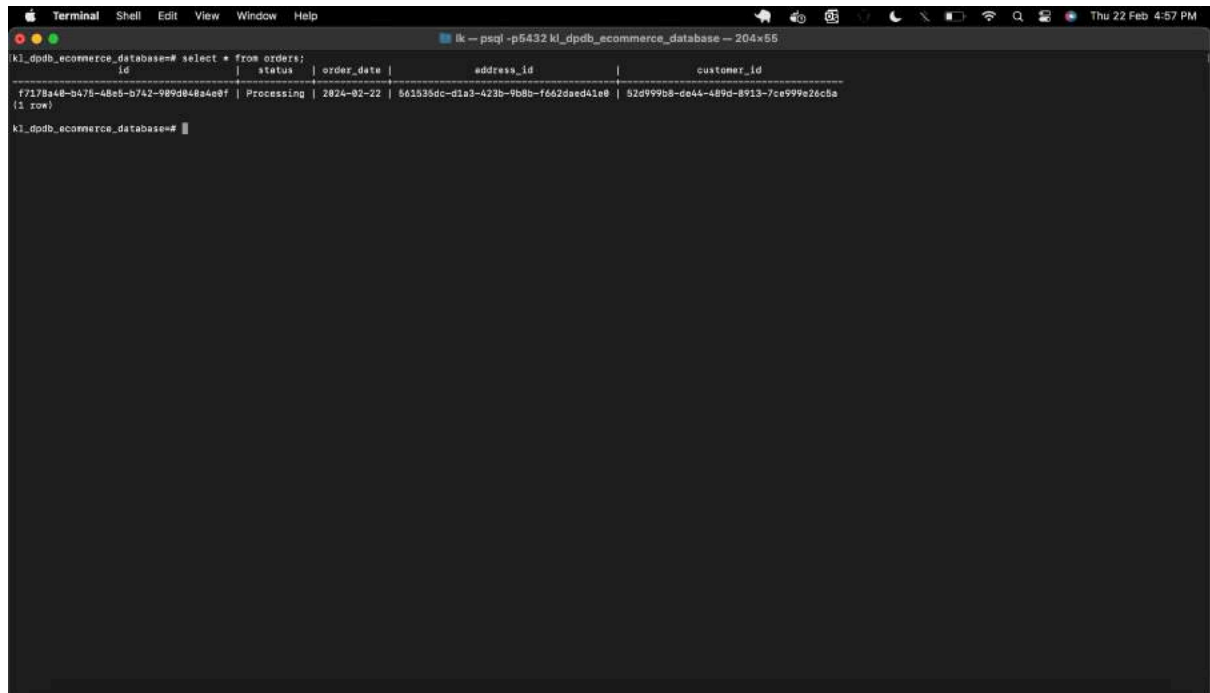
id	name	description	price	quantity	discount	brandname	address_id	product_category_id
4d295a98-853f-4db1-a22a-37cc18365798	SHOES	Fit for running and trekking	1000	0	8	adidas	826e669e-7458-4882-83ef-88a782ce81b7	e4f2c127-51df-4bf4-b61e-s8569d234d57

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;
```



The image shows a terminal window with a PostgreSQL prompt. The user has executed the query 'select * from orders;'. The result is displayed as a table with 5 columns: id, status, order_date, address_id, and customer_id. There is one row of data. Below the table, it says '(1 row)'. The prompt is now 'k1_dadb_ecommerce_database=#'.

id	status	order_date	address_id	customer_id
f7178a48-b476-48e5-b742-989d848a4e0f	Processing	2024-02-22	561535dc-d1a3-423b-9b8b-f662daed41e8	52d999b8-de44-489d-8913-7ce999e26c5a

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;
```

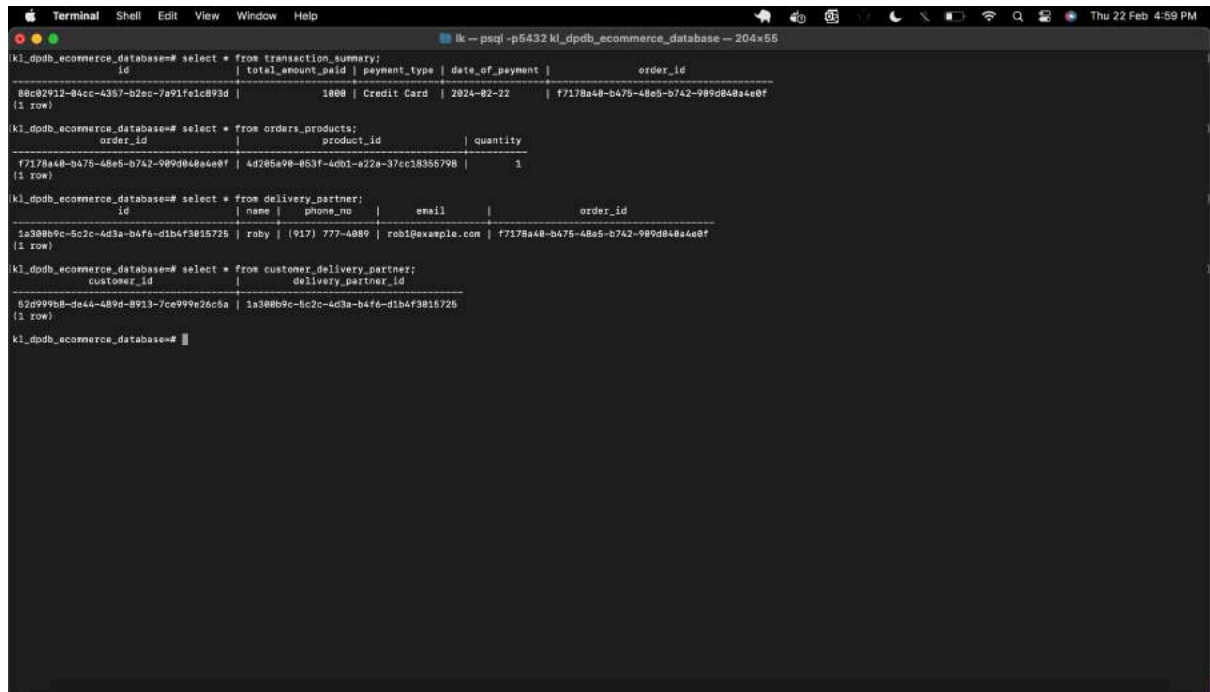
A terminal window titled 'Terminal' with a menu bar (Shell, Edit, View, Window, Help) and a status bar (Thu 22 Feb 4:59 PM). The terminal shows a series of SQL queries executed in a PostgreSQL environment (ik - psql - p5432 kl_dpd_b_e-commerce_database - 204x55). The queries and their results are as follows:
1. Query: `kl_dpd_b_e-commerce_database=# select * from transaction_summary;`
Result: A table with 5 columns: `id`, `total_amount_paid`, `payment_type`, `date_of_payment`, and `order_id`. It contains one row: `88c02912-04cc-43b7-b2ec-7a91fe1c893d`, `1000`, `Credit Card`, `2024-02-22`, and `f7178a48-b475-48e5-b742-989d848a4e0f`.
2. Query: `kl_dpd_b_e-commerce_database=# select * from orders_products;`
Result: A table with 3 columns: `order_id`, `product_id`, and `quantity`. It contains one row: `f7178a48-b475-48e5-b742-989d848a4e0f`, `4d285a98-853f-4db1-a22a-37ec18355798`, and `1`.
3. Query: `kl_dpd_b_e-commerce_database=# select * from delivery_partner;`
Result: A table with 5 columns: `id`, `name`, `phone_no`, `email`, and `order_id`. It contains one row: `1a388b9c-5c2c-4d3a-b4f6-d1b4f3815725`, `robby`, `(917) 777-4089`, `rob1@example.com`, and `f7178a48-b475-48e5-b742-989d848a4e0f`.
4. Query: `kl_dpd_b_e-commerce_database=# select * from customer_delivery_partner;`
Result: A table with 2 columns: `customer_id` and `delivery_partner_id`. It contains one row: `52d999b8-d644-489d-8913-7ce999e26c5a` and `1a388b9c-5c2c-4d3a-b4f6-d1b4f3815725`.
The terminal ends with the prompt `kl_dpd_b_e-commerce_database=#`.

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

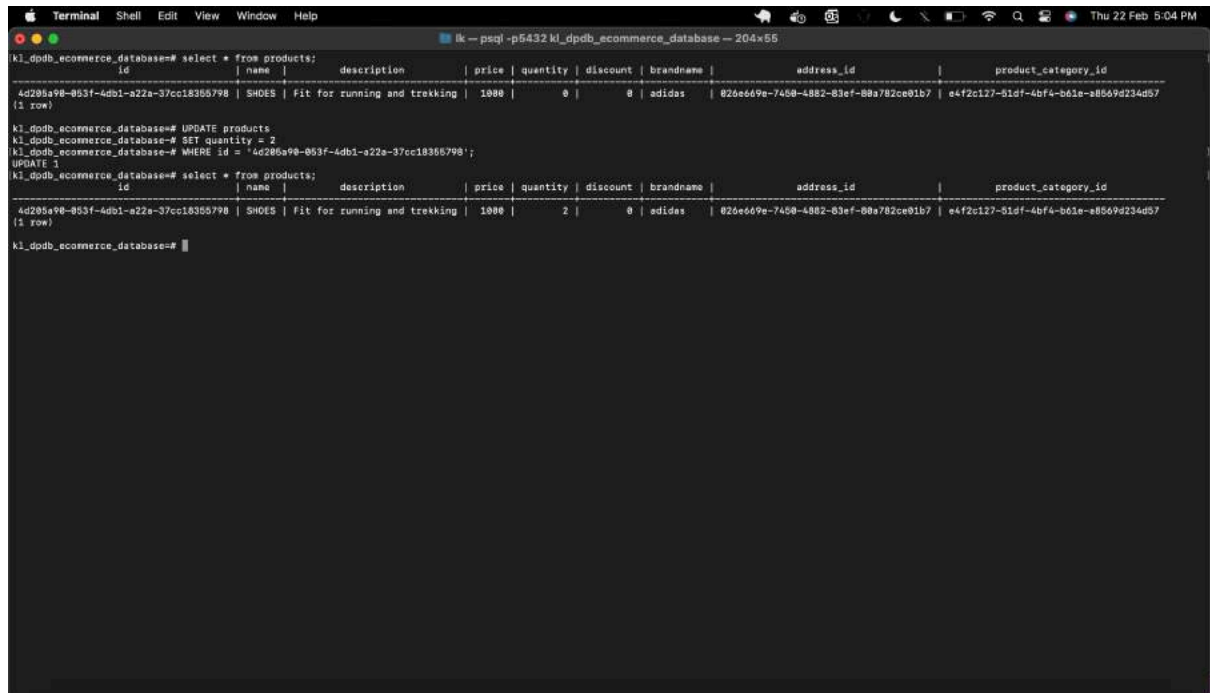
Explanation: updated data in all the required tables (transaction_summary, orders_products, delivery_partner, customer_delivery_partner). Which says data is consistent.

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;
```

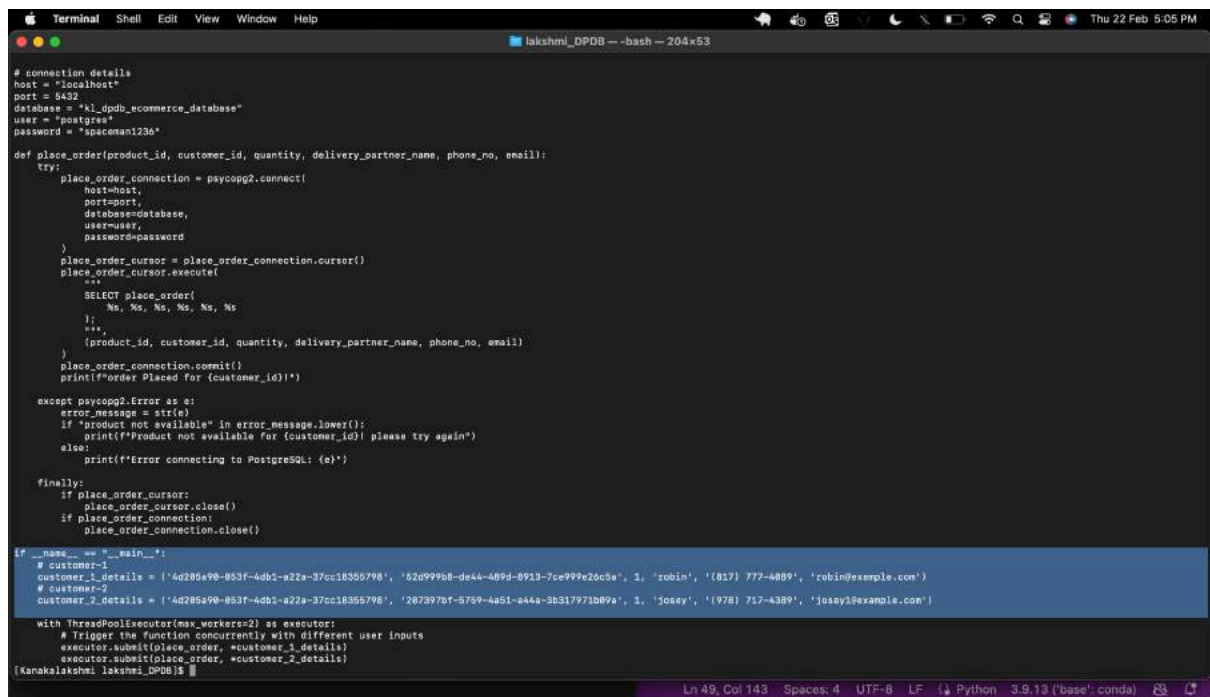


```
Terminal Shell Edit View Window Help  
ik - postgresql k1_dpd_b_e-commerce_database - 204x55  
k1_dpd_b_e-commerce_database=# select * from products;  
id | name | description | price | quantity | discount | brandname | address_id | product_category_id  
-----  
4d205a90-053f-4db1-a22a-37cc18355798 | SHOES | Fit for running and trekking | 1000 | 0 | 0 | adidas | 020e669e-7450-4882-83ef-80a782ce01b7 | e4f2c127-51df-4bf4-b61e-a8569d234d57  
(1 row)  
k1_dpd_b_e-commerce_database=# UPDATE products  
k1_dpd_b_e-commerce_database=# SET quantity = 2  
k1_dpd_b_e-commerce_database=# WHERE id = '4d205a90-053f-4db1-a22a-37cc18355798';  
UPDATE 1  
k1_dpd_b_e-commerce_database=# select * from products;  
id | name | description | price | quantity | discount | brandname | address_id | product_category_id  
-----  
4d205a90-053f-4db1-a22a-37cc18355798 | SHOES | Fit for running and trekking | 1000 | 2 | 0 | adidas | 020e669e-7450-4882-83ef-80a782ce01b7 | e4f2c127-51df-4bf4-b61e-a8569d234d57  
(1 row)  
k1_dpd_b_e-commerce_database=#
```

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')
```



```
Terminal Shell Edit View Window Help
lakshmi_DPDB -- -bash -- 204x53

# connection details
host = "localhost"
port = 5432
database = "l_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')

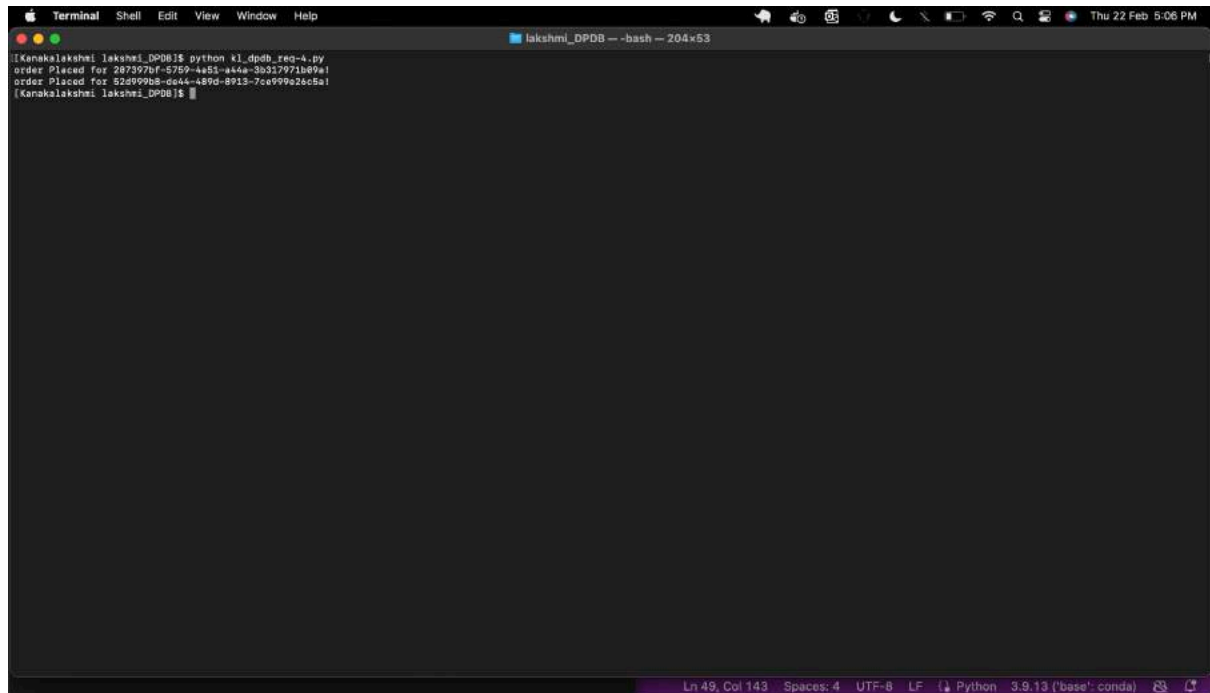
    with ThreadPoolExecutor(max_workers=2) as executor:
        # Trigger the function concurrently with different user inputs
        executor.submit(place_order, *customer_1_details)
        executor.submit(place_order, *customer_2_details)

(Kanakalakshmi lakshmi_DPDB)$
```

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

query:

python kl_dpdb_req-4.py

A screenshot of a macOS Terminal window. The title bar shows 'Terminal' and 'Shell'. The window title is 'lakshmi_DPDB -- -bash -- 204x53'. The prompt is '[Kanakalakshmi lakshmi_DPDB]\$'. The command 'python kl_dpdb_req-4.py' has been executed. The output shows two successful order placements with their respective IDs: 'order Placed for 287397bf-5789-4a85-a44a-3a327971b89a' and 'order Placed for 52d999b8-cc44-a89d-8915-7ce99a25c5a1'. The prompt returns to '[Kanakalakshmi lakshmi_DPDB]\$'. The status bar at the bottom shows 'Ln 49, Col 143', 'Spaces: 4', 'UTF-8', 'LF', 'Python 3.9.13 ('base': conda)', and system icons on the right.

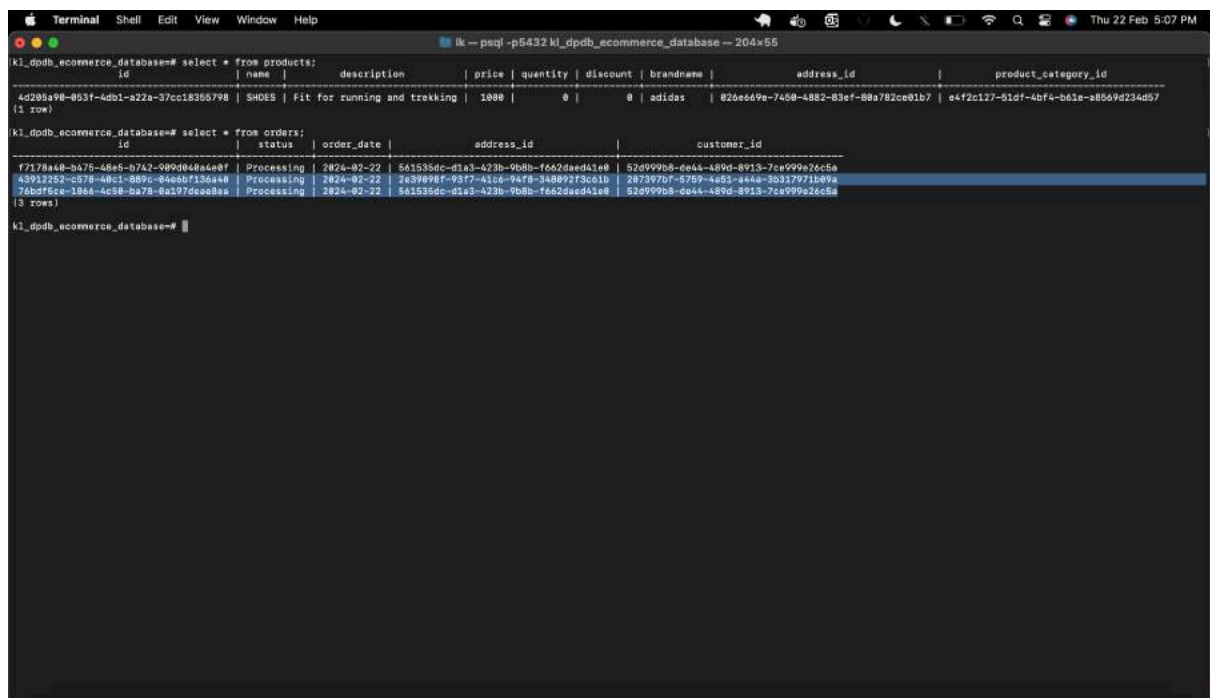
```
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 287397bf-5789-4a85-a44a-3a327971b89a
order Placed for 52d999b8-cc44-a89d-8915-7ce99a25c5a1
[Kanakalakshmi lakshmi_DPDB]$
```

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;
```



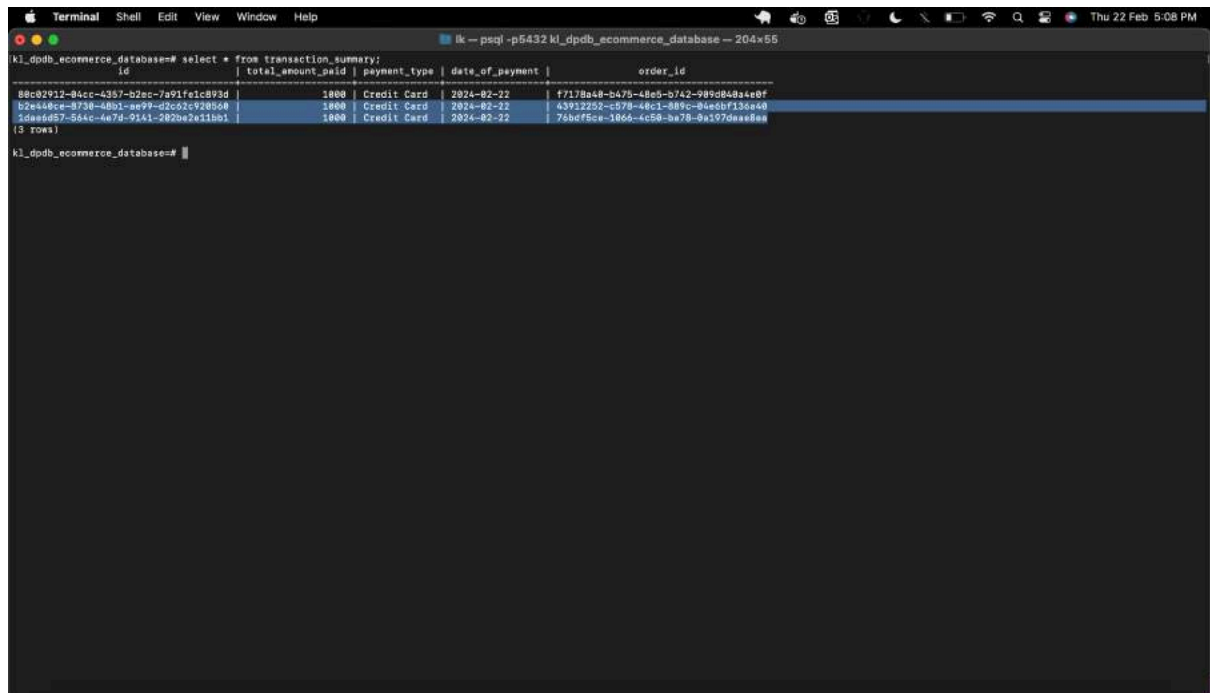
```
Terminal Shell Edit View Window Help  
ik - psql - p5432 kl_dpd_b_ecommerce_database - 204x55  
kl_dpd_b_ecommerce_database=# select * from products;  
id | name | description | price | quantity | discount | brandname | address_id | product_category_id  
(2 rows)  
4d28a98-853f-4db1-a22a-37cc18365798 | SHOES | Fit for running and trekking | 1000 | 0 | 0 | adidas | 82ae649e-7458-4882-83ef-88a782ce81b7 | e4f2c127-51df-4bf4-b61e-a8569d234d57  
kl_dpd_b_ecommerce_database=# select * from orders;  
id | status | order_date | address_id | customer_id  
(3 rows)  
f7178a48-b475-48e5-b742-989d048ae0f | Processing | 2024-02-22 | 561535dc-d1a3-423b-9b8b-f662daed41e8 | 52d999b8-d644-489d-8913-7ce999e26c5a  
43912252-c578-48c1-889c-84e6bf136a48 | Processing | 2024-02-22 | 2e39898f-93f7-41c6-94f8-348892f3c61b | 287397bf-5759-4a51-a44a-30317971b09a  
28b6ff6c-18e4-4c58-ba78-6a197dca88a | Processing | 2024-02-22 | 561535dc-d1a3-423b-9b8b-f662daed41e8 | 52d999b8-d644-489d-8913-7ce999e26c5a  
kl_dpd_b_ecommerce_database=#
```

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;
```



```
Terminal Shell Edit View Window Help
ik -- psql -p5432 ki_dpdb_ecommerce_database -- 204x55

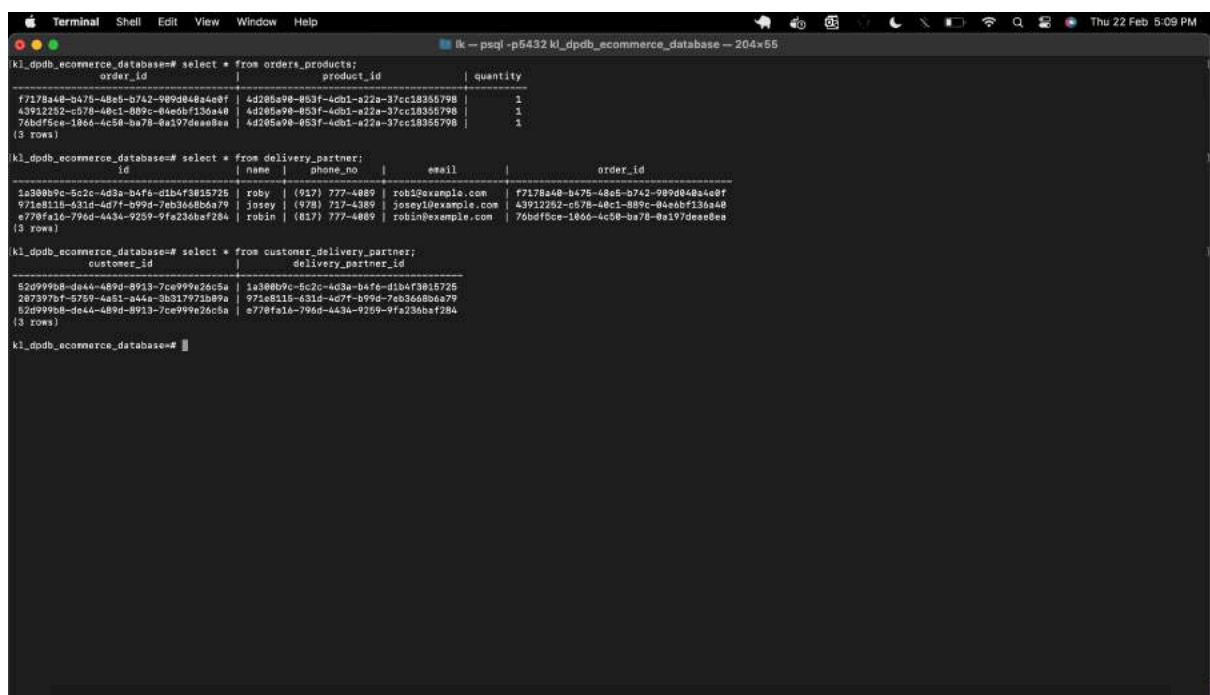
ki_dpdb_ecommerce_database=# select * from transaction_summary;
 id | total_amount_paid | payment_type | date_of_payment | order_id
-----+-----+-----+-----+-----
 88c02912-94cc-4357-b22c-7a91fa1c893d | 1000 | Credit Card | 2024-02-22 | f7178a40-b475-48e5-b742-989d048a4e0f
 b2e448ce-8730-48b1-ae97-d2c2c9205d9 | 1000 | Credit Card | 2024-02-22 | 43912252-c578-48c1-889c-04e0bf136a40
 1deae457-564c-4e7d-9141-202ba2e11b01 | 1000 | Credit Card | 2024-02-22 | 76bdf5ce-1066-4c50-ba78-0a197deae8ea
(3 rows)

ki_dpdb_ecommerce_database=#
```

Fig-57: Updated transaction summary for two orders.

query:

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



```
Terminal Shell Edit View Window Help
ik -- psql -p5432 ki_dpdb_ecommerce_database -- 204x55

ki_dpdb_ecommerce_database=# select * from orders_products;
 order_id | product_id | quantity
-----+-----+-----
 f7178a40-b475-48e5-b742-989d048a4e0f | 4d205a98-853f-4db1-e22a-37cc18355798 | 1
 43912252-c578-48c1-889c-04e0bf136a40 | 4d205a98-853f-4db1-e22a-37cc18355798 | 1
 76bdf5ce-1066-4c50-ba78-0a197deae8ea | 4d205a98-853f-4db1-e22a-37cc18355798 | 1
(3 rows)

ki_dpdb_ecommerce_database=# select * from delivery_partner;
 id | name | phone_no | email | order_id
-----+-----+-----+-----+-----
 1a308b9c-5c2c-4d3a-b4f6-d1b4f3815725 | roby | (917) 777-4889 | roby@example.com | f7178a40-b475-48e5-b742-989d048a4e0f
 971a8115-631d-4d7f-b99d-7eb368b0a79 | jasey | (978) 717-4389 | jasey@example.com | 43912252-c578-48c1-889c-04e0bf136a40
 e778fa1d-79ad-443a-9259-9fa23abaf284 | robin | (817) 777-4889 | robin@example.com | 76bdf5ce-1066-4c50-ba78-0a197deae8ea
(3 rows)

ki_dpdb_ecommerce_database=# select * from customer_delivery_partner;
 customer_id | delivery_partner_id
-----+-----
 52d999b8-dc44-489d-8913-7ce999a26c5a | 1a308b9c-5c2c-4d3a-b4f6-d1b4f3815725
 287397bf-57d9-4a51-e44a-3b317972b07a | 971a8115-631d-4d7f-b99d-7eb368b0a79
 52d999b8-dc44-489d-8913-7ce999a26c5a | e778fa1d-79ad-443a-9259-9fa23abaf284
(3 rows)

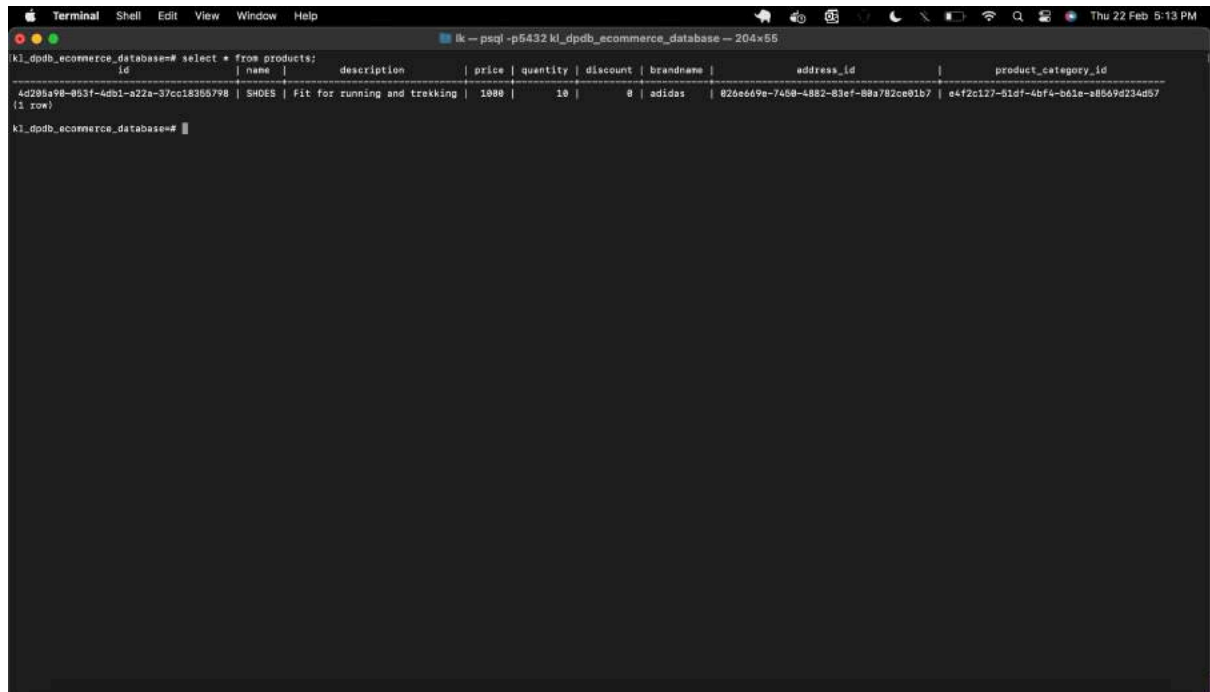
ki_dpdb_ecommerce_database=#
```

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';
```

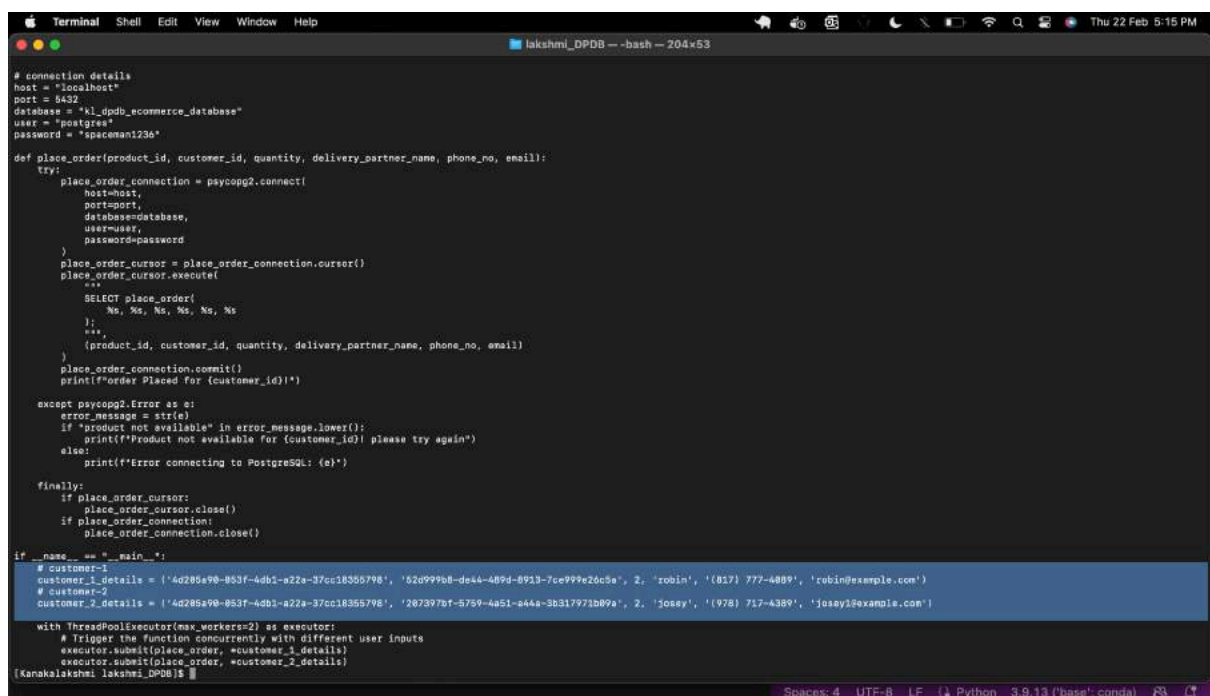


```
Terminal Shell Edit View Window Help
ik - psql -p5432 kl_dpdb_ecommerce_database -- 204x55

kl_dpdb_ecommerce_database=# select * from products;
      id      | name      | description      | price | quantity | discount | brandname | address_id      | product_category_id
-----+-----+-----+-----+-----+-----+-----+-----+-----
 4d205a90-053f-4db1-a22a-37cc18355798 | SHOES    | Fit for running and trekking | 1000  |      10  |         0 | adidas   | 820ee649e-7458-4882-83ef-88a782ce81b7 | ea42c127-51df-4bf4-b61e-a8569d214d57
(1 row)

kl_dpdb_ecommerce_database=#
```

Fig-59: Updated product quantity to 10



```
Terminal Shell Edit View Window Help
lakshmi_OPDB -- -bash -- 204x53

# 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', '5209999b-de44-489d-8913-7ce999e26c8a', 2, 'robin', '(817) 777-4889', 'robin@example.com')
    # Customer-2
    customer_2_details = ('4d205a90-053f-4db1-a22a-37cc18355798', '267397bf-5759-4a51-a44a-3b317971b07a', 2, 'jossy', '(978) 717-4389', 'jossy1@example.com')

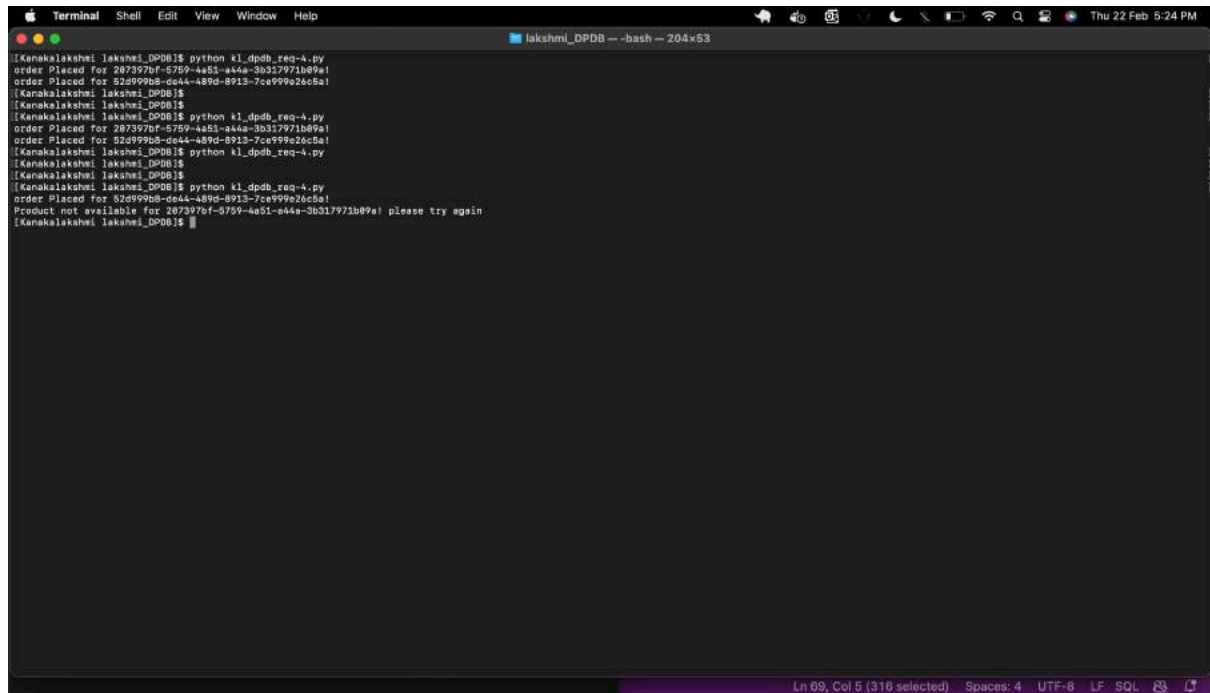
    with ThreadPoolExecutor(max_workers=2) as executor:
        # Trigger the function concurrently with different user inputs
        executor.submit(place_order, *customer_1_details)
        executor.submit(place_order, *customer_2_details)

[Kanakalakshmi lakshmi_OPDB]$
```

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

query:

python kl_dpdb_req-4.py



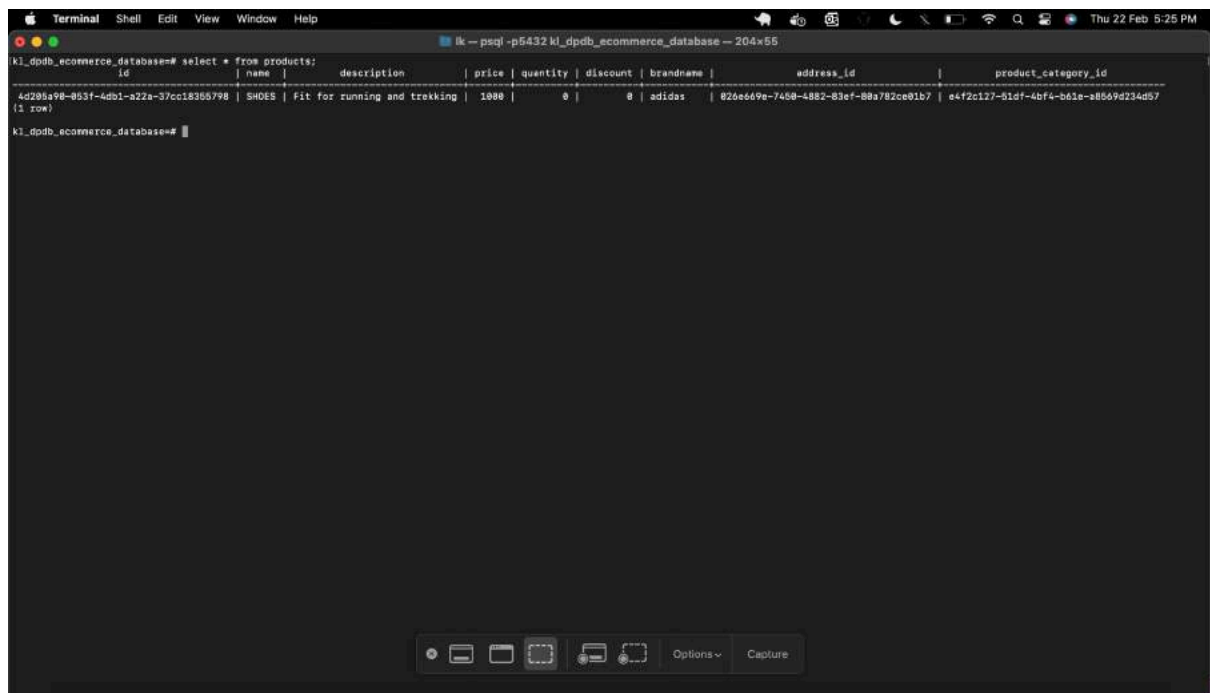
```
Terminal Shell Edit View Window Help
lakshmi_DPDB -- -bash -- 204x53

[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 287397bf-5759-4a51-a44a-3b317971b89e!
[Kanakalakshmi lakshmi_DPDB]$
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 52d999b8-de44-a89d-8913-7ce999a26c5a!
[Kanakalakshmi lakshmi_DPDB]$
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 287397bf-5759-4a51-a44a-3b317971b89e!
[Kanakalakshmi lakshmi_DPDB]$
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 52d999b8-de44-a89d-8913-7ce999a26c5a!
[Kanakalakshmi lakshmi_DPDB]$
[Kanakalakshmi lakshmi_DPDB]$ python kl_dpdb_req-4.py
order Placed for 52d999b8-de44-a89d-8913-7ce999a26c5a!
Product not available for 287397bf-5759-4a51-a44a-3b317971b89e! please try again
[Kanakalakshmi lakshmi_DPDB]$
```

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

query:

select * from products;



```
Terminal Shell Edit View Window Help
lk -- psql -p5432 kl_dpdb_ecommerce_database -- 204x55

kl_dpdb_ecommerce_database=# select * from products;
 id | name | description | price | quantity | discount | brandname | address_id | product_category_id
-----+-----+-----+-----+-----+-----+-----+-----+-----
 4c295a98-053f-4db1-a22a-37cc183b5798 | SHOES | Fit for running and trekking | 1000 | 0 | 0 | adidas | 020e649e-7408-4882-83ef-88a782ce01b7 | a4f2c127-51df-4bf4-b61a-a8569d214d57
(1 row)

kl_dpdb_ecommerce_database=#
```

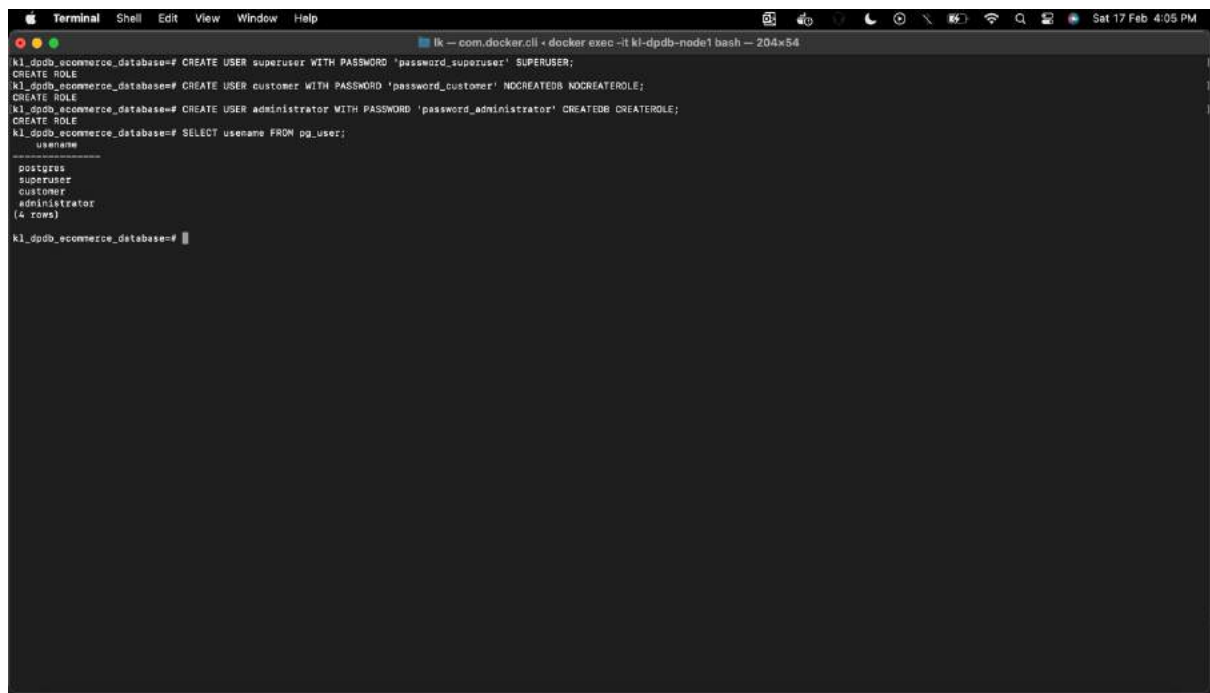
Fig-62: Quantity of the product is 0.

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 username FROM pg_user;
```

A screenshot of a terminal window titled "Terminal" with a menu bar (Shell, Edit, View, Window, Help). The terminal shows a series of SQL commands being executed in a PostgreSQL database. The commands are:
1. `CREATE USER superuser WITH PASSWORD 'password_superuser' SUPERUSER;`
2. `CREATE USER customer WITH PASSWORD 'password_customer' NOCREATEDB NOCREATEROLE;`
3. `CREATE USER administrators WITH PASSWORD 'password_administrators' CREATEDB CREATEROLE;`
4. `SELECT username FROM pg_user;`
The output of the last command shows a table with one column "username" and three rows: "postgres", "superuser", and "customer". The terminal prompt is `ki_dpd_ecommerce_database=#`.

```
ki_dpd_ecommerce_database=# CREATE USER superuser WITH PASSWORD 'password_superuser' SUPERUSER;
CREATE ROLE
ki_dpd_ecommerce_database=# CREATE USER customer WITH PASSWORD 'password_customer' NOCREATEDB NOCREATEROLE;
CREATE ROLE
ki_dpd_ecommerce_database=# CREATE USER administrator WITH PASSWORD 'password_administrator' CREATEDB CREATEROLE;
CREATE ROLE
ki_dpd_ecommerce_database=# SELECT username FROM pg_user;
username
-----
postgres
superuser
customer
administrator
(4 rows)

ki_dpd_ecommerce_database=#
```

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;
```

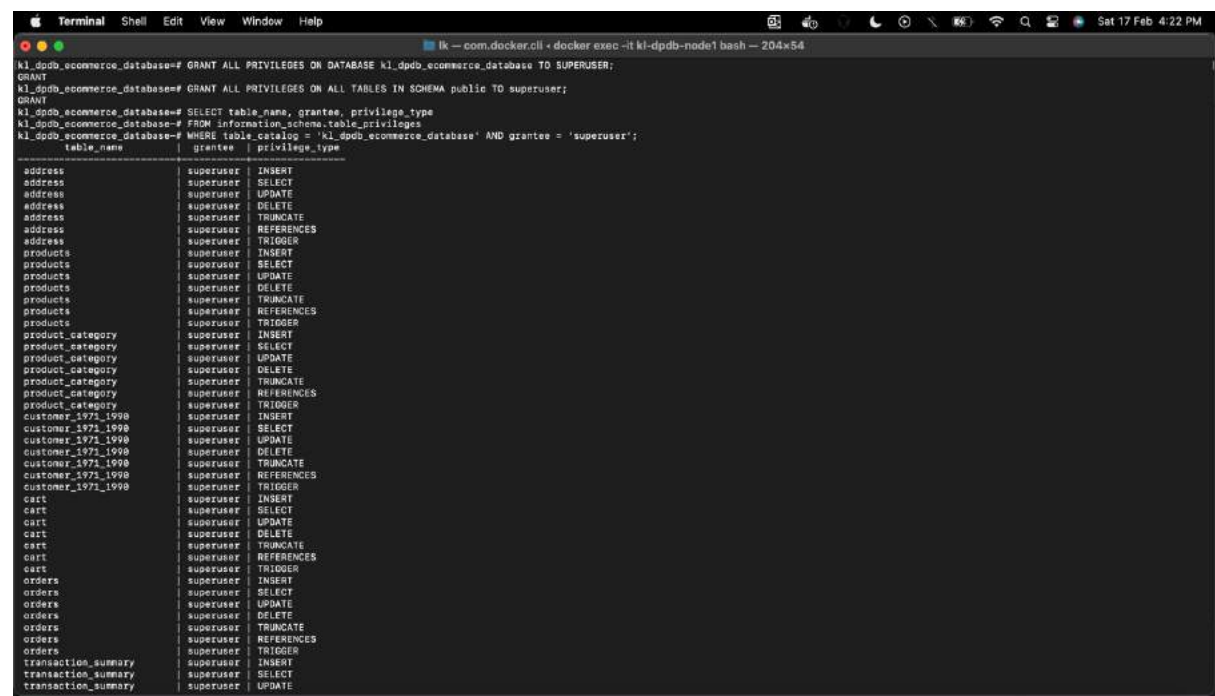
```
GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO superuser;
```

--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';
```



```
kl_dpdb_ecommerce_database=# GRANT ALL PRIVILEGES ON DATABASE kl_dpdb_ecommerce_database TO SUPERUSER;
GRANT
kl_dpdb_ecommerce_database=# GRANT ALL PRIVILEGES ON ALL TABLES IN SCHEMA public TO superuser;
GRANT
kl_dpdb_ecommerce_database=# SELECT table_name, grantee, privilege_type
kl_dpdb_ecommerce_database=# FROM information_schema.table_privileges
kl_dpdb_ecommerce_database=# WHERE table_catalog = 'kl_dpdb_ecommerce_database' AND grantee = 'superuser';
```

table_name	grantee	privilege_type
address	superuser	INSERT
address	superuser	SELECT
address	superuser	UPDATE
address	superuser	DELETE
address	superuser	TRUNCATE
address	superuser	REFERENCES
address	superuser	TRIGGER
products	superuser	INSERT
products	superuser	SELECT
products	superuser	UPDATE
products	superuser	DELETE
products	superuser	TRUNCATE
products	superuser	REFERENCES
products	superuser	TRIGGER
product_category	superuser	INSERT
product_category	superuser	SELECT
product_category	superuser	UPDATE
product_category	superuser	DELETE
product_category	superuser	TRUNCATE
product_category	superuser	REFERENCES
product_category	superuser	TRIGGER
customer_1971_1990	superuser	INSERT
customer_1971_1990	superuser	SELECT
customer_1971_1990	superuser	UPDATE
customer_1971_1990	superuser	DELETE
customer_1971_1990	superuser	TRUNCATE
customer_1971_1990	superuser	REFERENCES
customer_1971_1990	superuser	TRIGGER
cart	superuser	INSERT
cart	superuser	SELECT
cart	superuser	UPDATE
cart	superuser	DELETE
cart	superuser	TRUNCATE
cart	superuser	REFERENCES
cart	superuser	TRIGGER
orders	superuser	INSERT
orders	superuser	SELECT
orders	superuser	UPDATE
orders	superuser	DELETE
orders	superuser	TRUNCATE
orders	superuser	REFERENCES
orders	superuser	TRIGGER
transaction_summary	superuser	INSERT
transaction_summary	superuser	SELECT
transaction_summary	superuser	UPDATE

Fig-64: Grant access to superuser.

```
Terminal Shell Edit View Window Help
fk -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x54

customer_address | superuser | DELETE
customer_address | superuser | TRUNCATE
customer_address | superuser | REFERENCES
customer_address | superuser | TRIGGER
customer_gift_voucher | superuser | INSERT
customer_gift_voucher | superuser | SELECT
customer_gift_voucher | superuser | UPDATE
customer_gift_voucher | superuser | DELETE
customer_gift_voucher | superuser | TRUNCATE
customer_gift_voucher | superuser | REFERENCES
customer_gift_voucher | superuser | TRIGGER
gift_vouchers | superuser | INSERT
gift_vouchers | superuser | SELECT
gift_vouchers | superuser | UPDATE
gift_vouchers | superuser | DELETE
gift_vouchers | superuser | TRUNCATE
gift_vouchers | superuser | REFERENCES
gift_vouchers | superuser | TRIGGER
products_cart | superuser | INSERT
products_cart | superuser | SELECT
products_cart | superuser | UPDATE
products_cart | superuser | DELETE
products_cart | superuser | TRUNCATE
products_cart | superuser | REFERENCES
products_cart | superuser | TRIGGER
orders_products | superuser | INSERT
orders_products | superuser | SELECT
orders_products | superuser | UPDATE
orders_products | superuser | DELETE
orders_products | superuser | TRUNCATE
orders_products | superuser | REFERENCES
orders_products | superuser | TRIGGER
products_suppliers | superuser | INSERT
products_suppliers | superuser | SELECT
products_suppliers | superuser | UPDATE
products_suppliers | superuser | DELETE
products_suppliers | superuser | TRUNCATE
products_suppliers | superuser | REFERENCES
products_suppliers | superuser | TRIGGER
customer | superuser | INSERT
customer | superuser | SELECT
customer | superuser | UPDATE
customer | superuser | DELETE
customer | superuser | TRUNCATE
customer | superuser | REFERENCES
customer | superuser | TRIGGER
customer_1950_1970 | superuser | INSERT
customer_1950_1970 | superuser | SELECT
customer_1950_1970 | superuser | UPDATE
customer_1950_1970 | superuser | DELETE
customer_1950_1970 | superuser | TRUNCATE
customer_1950_1970 | superuser | REFERENCES
customer_1950_1970 | superuser | TRIGGER
(168 rows)
```

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';
```

```
Terminal Shell Edit View Window Help
tk -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x54

ki_dpdb_ecommerce_database=# CREATE OR REPLACE FUNCTION grant_privileges_to_administrator() RETURNS VOID AS $$
BEGIN
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Products TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Customer TO administrator';
    EXECUTE 'GRANT SELECT, UPDATE, DELETE ON TABLE cart TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE orders TO administrator';
    EXECUTE 'GRANT SELECT ON TABLE "transaction_summary" TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE supplier TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Address TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE product_category TO administrator';
    EXECUTE 'GRANT SELECT, DELETE ON TABLE Reviews TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE gift_vouchers TO administrator';
    EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE delivery_partner TO administrator';
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
ki_dpdb_ecommerce_database=#
ki_dpdb_ecommerce_database=# SELECT table_name, grantee, privilege_type
ki_dpdb_ecommerce_database=# FROM information_schemas.table_privileges
ki_dpdb_ecommerce_database=# WHERE table_catalog = 'ki_dpdb_ecommerce_database' AND grantee = 'administrator';
table_name | grantee | privilege_type
-----
products | administrator | INSERT
products | administrator | SELECT
products | administrator | UPDATE
products | administrator | DELETE
address | administrator | INSERT
address | administrator | SELECT
address | administrator | UPDATE
address | administrator | DELETE
product_category | administrator | INSERT
product_category | administrator | SELECT
product_category | administrator | UPDATE
product_category | administrator | DELETE
cart | administrator | SELECT
cart | administrator | UPDATE
cart | administrator | DELETE
orders | administrator | INSERT
orders | administrator | SELECT
orders | administrator | UPDATE
orders | administrator | DELETE
transaction_summary | administrator | SELECT
customer | administrator | INSERT
customer | administrator | SELECT
customer | administrator | UPDATE
customer | administrator | DELETE
supplier | administrator | INSERT
supplier | administrator | SELECT
supplier | administrator | UPDATE
supplier | administrator | DELETE
reviews | administrator | SELECT
reviews | administrator | DELETE
gift_vouchers | administrator | INSERT
```

Fig-66: Grant access to administrator.

```
Terminal Shell Edit View Window Help
tk -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x54

EXECUTE 'GRANT SELECT, DELETE ON TABLE Reviews TO administrator';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE gift_vouchers TO administrator';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE delivery_partner TO administrator';
END;
$$ LANGUAGE plpgsql;

CREATE FUNCTION
ki_dpdb_ecommerce_database=#
ki_dpdb_ecommerce_database=# SELECT table_name, grantee, privilege_type
ki_dpdb_ecommerce_database=# FROM information_schemas.table_privileges
ki_dpdb_ecommerce_database=# WHERE table_catalog = 'ki_dpdb_ecommerce_database' AND grantee = 'administrator';
table_name | grantee | privilege_type
-----
products | administrator | INSERT
products | administrator | SELECT
products | administrator | UPDATE
products | administrator | DELETE
address | administrator | INSERT
address | administrator | SELECT
address | administrator | UPDATE
address | administrator | DELETE
product_category | administrator | INSERT
product_category | administrator | SELECT
product_category | administrator | UPDATE
product_category | administrator | DELETE
cart | administrator | SELECT
cart | administrator | UPDATE
cart | administrator | DELETE
orders | administrator | INSERT
orders | administrator | SELECT
orders | administrator | UPDATE
orders | administrator | DELETE
transaction_summary | administrator | SELECT
customer | administrator | INSERT
customer | administrator | SELECT
customer | administrator | UPDATE
customer | administrator | DELETE
supplier | administrator | INSERT
supplier | administrator | SELECT
supplier | administrator | UPDATE
supplier | administrator | DELETE
reviews | administrator | SELECT
reviews | administrator | DELETE
gift_vouchers | administrator | INSERT
gift_vouchers | administrator | SELECT
gift_vouchers | administrator | UPDATE
gift_vouchers | administrator | DELETE
delivery_partner | administrator | INSERT
delivery_partner | administrator | SELECT
delivery_partner | administrator | UPDATE
delivery_partner | administrator | DELETE
(38 rows)

ki_dpdb_ecommerce_database=#
```

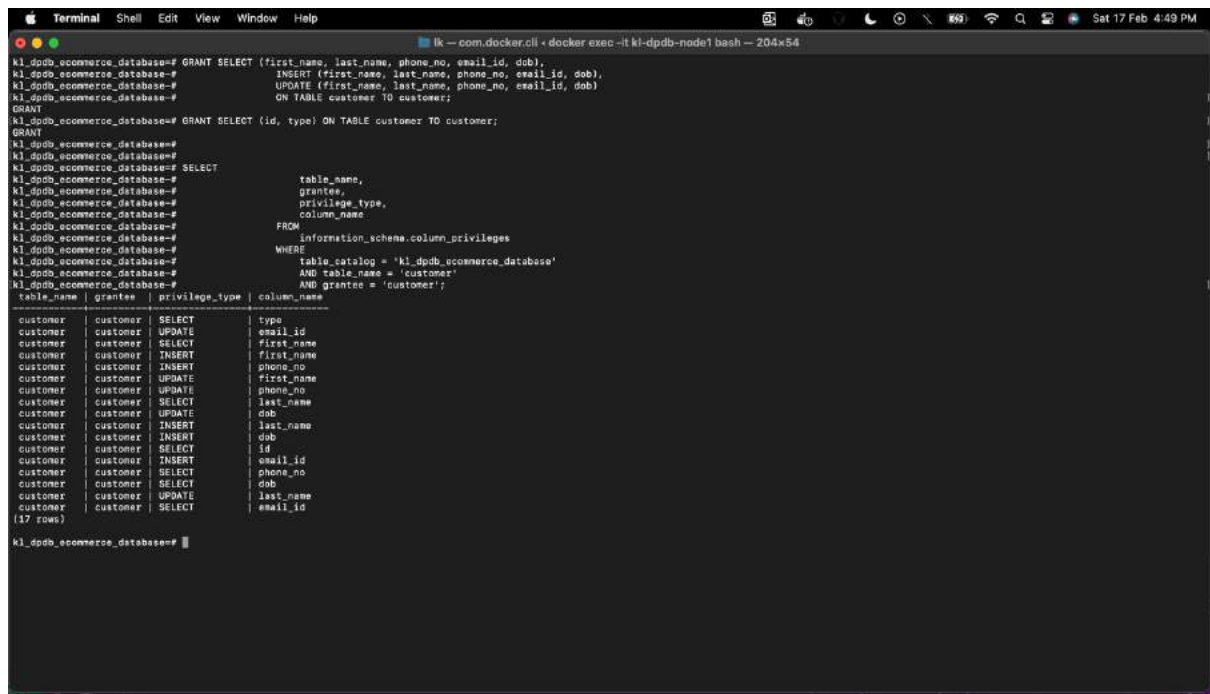
Fig-67: list of complete privileges for administrator.

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';
```



```
kl_dpdb_ecommerce_database# GRANT SELECT (first_name, last_name, phone_no, email_id, dob),
kl_dpdb_ecommerce_database# INSERT (first_name, last_name, phone_no, email_id, dob),
kl_dpdb_ecommerce_database# UPDATE (first_name, last_name, phone_no, email_id, dob)
kl_dpdb_ecommerce_database# ON TABLE customer TO customer;
GRANT
kl_dpdb_ecommerce_database# GRANT SELECT (id, type) ON TABLE customer TO customer;
GRANT
kl_dpdb_ecommerce_database#
kl_dpdb_ecommerce_database#
kl_dpdb_ecommerce_database# SELECT
kl_dpdb_ecommerce_database# table_name,
kl_dpdb_ecommerce_database# grantee,
kl_dpdb_ecommerce_database# privilege_type,
kl_dpdb_ecommerce_database# column_name
kl_dpdb_ecommerce_database# FROM
kl_dpdb_ecommerce_database# information_schema.column_privileges
kl_dpdb_ecommerce_database# WHERE
kl_dpdb_ecommerce_database# table_catalog = 'kl_dpdb_ecommerce_database'
kl_dpdb_ecommerce_database# AND table_name = 'customer'
kl_dpdb_ecommerce_database# AND grantee = 'customer';
table_name | grantee | privilege_type | column_name
-----
customer | customer | SELECT | type
customer | customer | UPDATE | email_id
customer | customer | SELECT | first_name
customer | customer | INSERT | phone_no
customer | customer | UPDATE | first_name
customer | customer | UPDATE | phone_no
customer | customer | UPDATE | last_name
customer | customer | UPDATE | dob
customer | customer | INSERT | last_name
customer | customer | INSERT | dob
customer | customer | SELECT | id
customer | customer | INSERT | email_id
customer | customer | SELECT | phone_no
customer | customer | SELECT | dob
customer | customer | UPDATE | last_name
customer | customer | SELECT | email_id
(17 rows)

kl_dpdb_ecommerce_database#
```

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

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

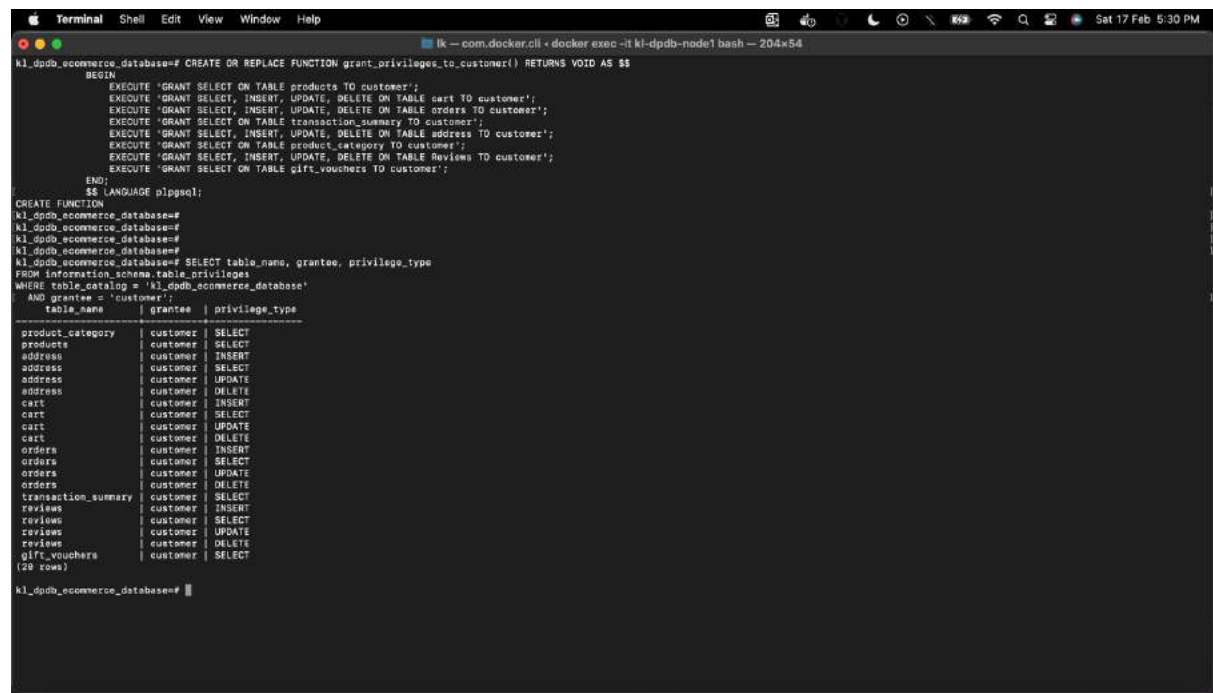
Grant access to customer user on remaining tables:

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';
```



```
kl_dpdb_ecommerce_database=# CREATE OR REPLACE FUNCTION grant_privileges_to_customer() RETURNS VOID AS $$
BEGIN
EXECUTE 'GRANT SELECT ON TABLE products TO customer';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE cart TO customer';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE orders TO customer';
EXECUTE 'GRANT SELECT ON TABLE transaction_summary TO customer';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE address TO customer';
EXECUTE 'GRANT SELECT ON TABLE product_category TO customer';
EXECUTE 'GRANT SELECT, INSERT, UPDATE, DELETE ON TABLE Reviews TO customer';
EXECUTE 'GRANT SELECT ON TABLE gift_vouchers TO customer';
END;
$$ LANGUAGE plpgsql;
kl_dpdb_ecommerce_database=#
kl_dpdb_ecommerce_database=# SELECT grant_privileges_to_customer();
kl_dpdb_ecommerce_database=# SELECT table_name, grantee, privilege_type
FROM information_schema.table_privileges
WHERE table_catalog = 'kl_dpdb_ecommerce_database'
AND grantee = 'customer';
 table_name | grantee | privilege_type
-----
product_category | customer | SELECT
products      | customer | SELECT
address       | customer | INSERT
address       | customer | SELECT
address       | customer | UPDATE
address       | customer | DELETE
cart          | customer | INSERT
cart          | customer | SELECT
cart          | customer | UPDATE
cart          | customer | DELETE
orders        | customer | INSERT
orders        | customer | SELECT
orders        | customer | UPDATE
orders        | customer | DELETE
transaction_summary | customer | SELECT
reviews       | customer | INSERT
reviews       | customer | SELECT
reviews       | customer | UPDATE
reviews       | customer | DELETE
gift_vouchers | customer | SELECT
(28 rows)
```

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';
```

The terminal window shows a PostgreSQL session. The first query, `select * from products;`, returns a table with 10 columns: `id`, `product_category_id`, `name`, `description`, `price`, `quantity`, `discount`, `brandname`, and `address_id`. The table contains 20 rows of product data, including items like 'Formal Shirt', 'Versatile handbag', 'Children Toy Set', 'Ultra HD Smart Projector', 'Running Shoes', 'Home Decor Pillow Set', 'Wireless Gaming Mouse', 'Gourmet Cooking Set', 'Casual Sneakers', 'Luxury Spa Bathrobe', 'Compact Air Purifier', 'Active Leggings', 'Professional DSLR Camera', and 'High-Performance Laptop'.

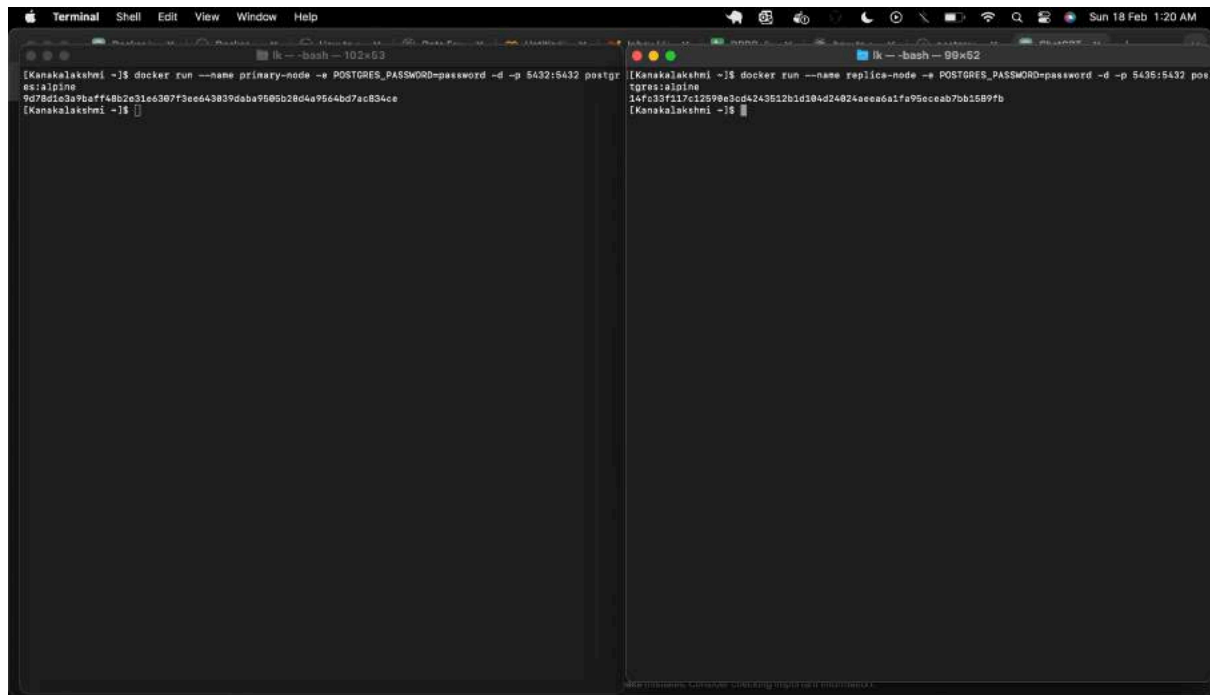
The second query, `delete from products where id='b136c1cf-d1e8-483f-9064-1c866f25195f';`, is executed. The terminal output shows the command being processed, followed by an error message: `ERROR: permission denied for table products`. The session then ends with the prompt `kl_dpdb_ecommerce_database> |`.

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 server 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.



```
[Kanakalakshmi ~]$ docker run --name primary-node --env POSTGRES_PASSWORD=password -d -p 5432:5432 postgres:13.1
9d78d1e3a9baff48b2e31e6307f3ee649839daba9f085b28d4a9564bd7ec834ce
[Kanakalakshmi ~]$

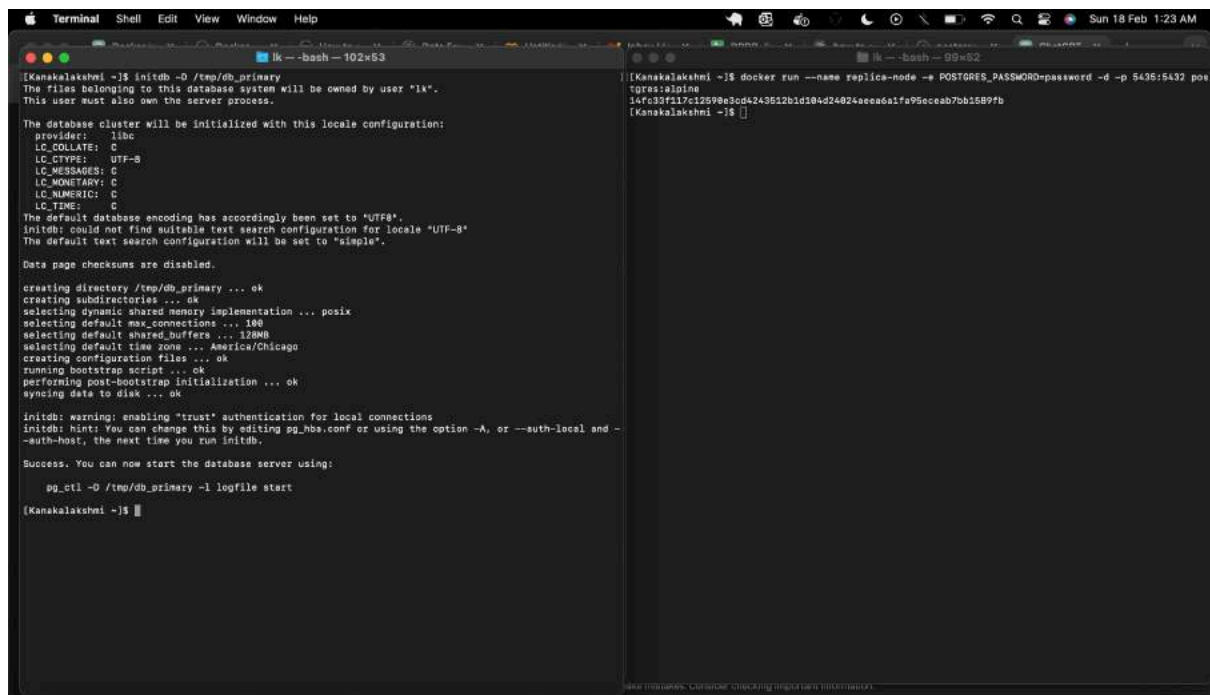
[Kanakalakshmi ~]$ docker run --name replica-node --env POSTGRES_PASSWORD=password -d -p 5435:5432 postgres:13.1
14fc33f117c12598e3cd4243512b1d184d24024eeea6a1fa95ecea7bb1589fb
[Kanakalakshmi ~]$
```

Fig-71: create two postgres instances.

query:

which initdb

initdb -D /tmp/db_primary



```
[KanakaIakshmi ~]$ initdb -D /tmp/db_primary
The files belonging to this database system will be owned by user "ik".
This user must also own the server process.

The database cluster will be initialized with this locale configuration:
provider: libc
LC_COLLATE: C
LC_CTYPE: UTF-8
LC_MESSAGES: C
LC_MONETARY: C
LC_NUMERIC: C
LC_TIME: C
The default database encoding has accordingly been set to 'UTF8'.
initdb: could not find suitable text search configuration for locale 'UTF-8'
The default text search configuration will be set to 'simple'.

Data page checksums are disabled.

creating directory /tmp/db_primary ... ok
creating subdirectories ... ok
selecting dynamic shared memory implementation ... posix
selecting default max_connections ... 100
selecting default shared_buffers ... 128MB
selecting default time zone ... America/Chicago
creating configuration files ... ok
running bootstrap script ... ok
performing post-bootstrap initialization ... ok
syncing data to disk ... ok

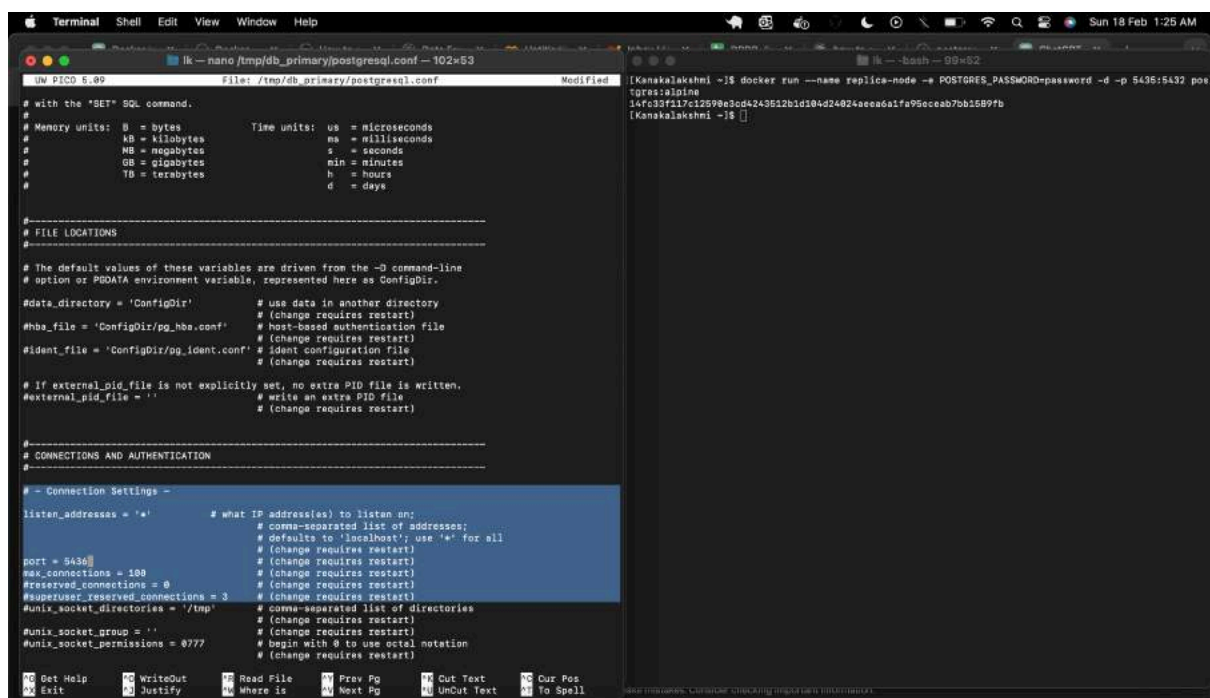
initdb: warning: enabling "trust" authentication for local connections
initdb: hint: You can change this by editing pg_hba.conf or using the option -A, or --auth-local and
--auth-host, the next time you run initdb.

Success. You can now start the database server using:

    pg_ctl -D /tmp/db_primary -l logfile start

[KanakaIakshmi ~]$
```

Fig-72: created Directory for the primary DB.

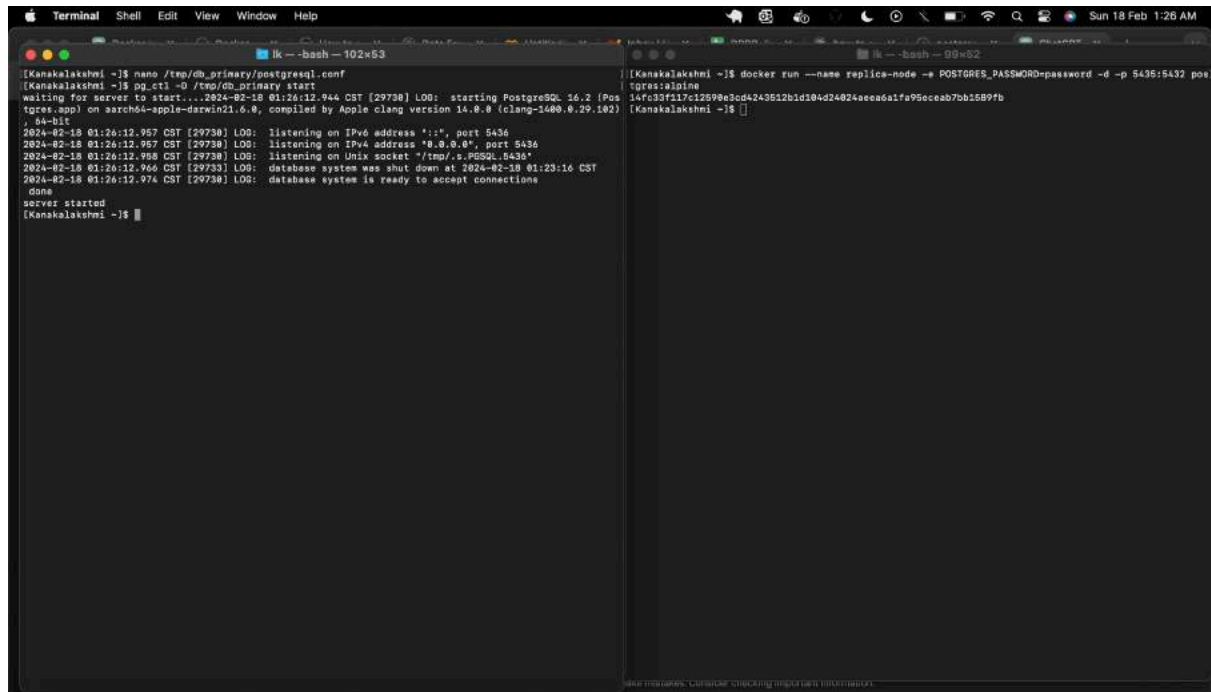


```
File: /tmp/db_primary/postgresql.conf Modified
# with the "SET" SQL command.
#
# Memory units: B = bytes          Time units: us = microseconds
#               kB = kilobytes      ms = milliseconds
#               MB = megabytes       s = seconds
#               GB = gigabytes       min = minutes
#               TB = terabytes       h = hours
#                                   d = days
#
#-----
# FILE LOCATIONS
#
# The default values of these variables are driven from the -D command-line
# option or PGDATA environment variable, represented here as ConfigDir.
#data_directory = 'ConfigDir'      # use data in another directory
#                                   # (change requires restart)
#hba_file = 'ConfigDir/pg_hba.conf' # host-based authentication file
#                                   # (change requires restart)
#ident_file = 'ConfigDir/pg_ident.conf' # ident configuration file
#                                   # (change requires restart)
#
# If external_pid_file is not explicitly set, no extra PID file is written.
#external_pid_file = ''            # write an extra PID file
#                                   # (change requires restart)
#
#-----
# CONNECTIONS AND AUTHENTICATION
#
# - Connection Settings -
#
listen_addresses = '*'             # what IP address(es) to listen on;
#                                   # comma-separated list of addresses;
#                                   # defaults to 'localhost'; use '*' for all
#                                   # (change requires restart)
port = 5432                        # (change requires restart)
max_connections = 100              # (change requires restart)
#superuser_reserved_connections = 3 # (change requires restart)
#unix_socket_directories = '/tmp'  # comma-separated list of directories
#                                   # (change requires restart)
#unix_socket_group = ''            # (change requires restart)
#unix_socket_permissions = 0777   # begin with 0 to use octal notation
#                                   # (change requires restart)
```

Fig-73: update configs in postgresql.conf

query:

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



The image shows two terminal windows side-by-side. The left window is titled 'Terminal' and shows the execution of 'pg_ctl -D /tmp/primary_db start'. It displays log messages from PostgreSQL 16.2, including 'starting PostgreSQL 16.2', 'listening on IPv4 address', 'listening on IPv6 address', 'listening on Unix socket', and 'database system is ready to accept connections'. The right window is titled 'Terminal' and shows the execution of 'docker run --name replica-node -e POSTGRES_PASSWORD=password -d -p 5435:5432 postgres:alpine'. It displays the container ID '14fc337f17c12598e3cd4243512b1d184d24824ee6a1fa95e6eb7bb1589fb'.

```
[Kanakalakshmi ~]$ nano /tmp/db_primary/postgresql.conf  
[Kanakalakshmi ~]$ pg_ctl -D /tmp/db_primary start  
waiting for server to start.... 2024-02-18 01:26:12.944 CST [29738] LOG: starting PostgreSQL 16.2 (PostgreSQL 16.2 on aarch64-apple-darwin21.6.0, compiled by Apple clang version 14.0.0 (clang-1400.0.29.102))  
done  
server started  
[Kanakalakshmi ~]$  
[Kanakalakshmi ~]$ docker run --name replica-node -e POSTGRES_PASSWORD=password -d -p 5435:5432 postgres:alpine  
14fc337f17c12598e3cd4243512b1d184d24824ee6a1fa95e6eb7bb1589fb  
[Kanakalakshmi ~]$
```

Fig-74: Start the postgres instance.

query:

```
psql --port=5436 postgres
```

```
create user replica_user replication;
```

The screenshot shows a terminal window with two panes. The left pane shows a user connecting to a PostgreSQL instance using the command `psql --port=5436 postgres`. The prompt changes to `postgres=#`, and the user enters the command `create user replica_user replication;`. The right pane shows a Docker container running a PostgreSQL instance with the command `docker run --name replica-node -e POSTGRES_PASSWORD=password -d -p 5435:5432 postgres:alpine`. The container ID is `14fc33f17c12598e3cd4243512b1d184d24824ee6a1fa95ceab7bb1589fb`.

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

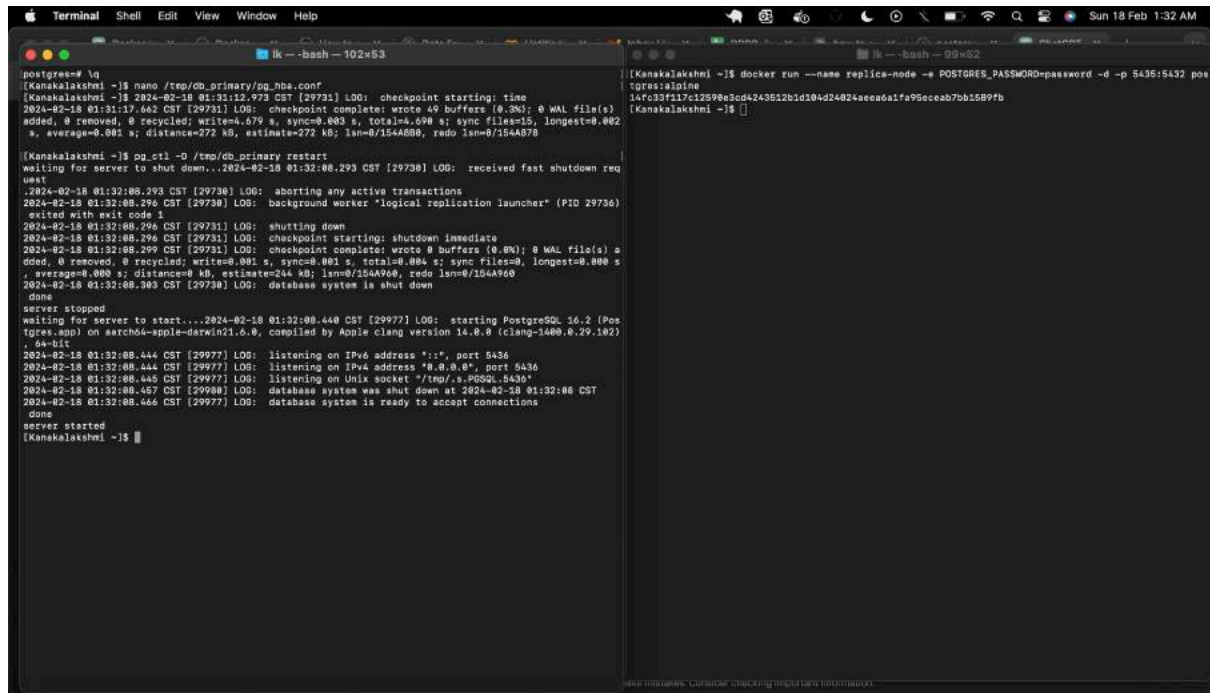
The screenshot shows a terminal window with two panes. The left pane shows a user editing the `pg_hba.conf` file in the `/tmp/db_primary/` directory. The file content is displayed, showing a table of authentication configurations. The right pane shows the same Docker container as in Fig-75.

# TYPE	DATABASE	USER	ADDRESS	METHOD
#	"local"	is for Unix domain socket connections only		
local	all	all		trust
#	IPv4 local connections:			
host	all	all	127.0.0.1/32	trust
host	all	replica_user	127.0.0.1/32	trust
#	IPv6 local connections:			
host	all	all	:::1/128	trust
#	Allow replication connections from localhost, by a user with the			
#	replication privilege.			
local	replication	all		trust
host	replication	all	127.0.0.1/32	trust
host	replication	all	:::1/128	trust

Fig-76: update configs in pg_hba.conf.

query:

pg_ctl -D /tmp/primary_db restart

A terminal window showing the execution of PostgreSQL commands. The user runs 'pg_ctl -D /tmp/primary_db restart'. The output shows the server shutting down, performing a checkpoint, and then starting back up. The terminal text is as follows:

```
postgres=# \q
[Kanakalakshmi ~]$ nano /tmp/db_primary/pg_hba.conf
[Kanakalakshmi ~]$ 2024-02-18 01:31:12.973 CST [29731] LOG:  checkpoint starting: time
2024-02-18 01:31:17.662 CST [29731] LOG:  checkpoint complete: wrote 49 buffers (0.3%); 0 WAL file(s)
added, 0 removed, 0 recycled; write=4.579 s, sync=0.003 s, total=4.690 s; sync files=15, longest=0.002
s, average=0.001 s; distance=272 kB, estimate=272 kB; lsn=0/154A658, redo lsn=0/154A670

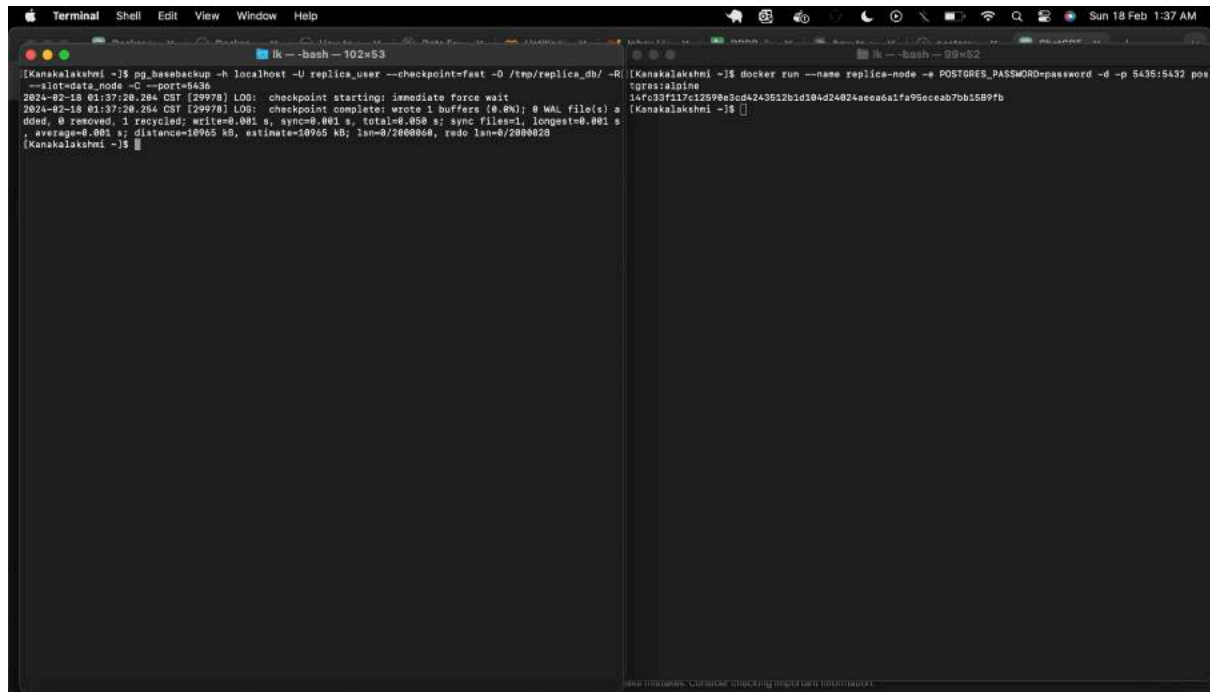
[Kanakalakshmi ~]$ pg_ctl -D /tmp/db_primary restart
waiting for server to shut down...2024-02-18 01:32:06.293 CST [29730] LOG:  received fast shutdown req
uest
2024-02-18 01:32:08.293 CST [29730] LOG:  aborting any active transactions
2024-02-18 01:32:08.296 CST [29730] LOG:  background worker "logical replication launcher" (PID 29736)
exited with exit code 1
2024-02-18 01:32:08.296 CST [29731] LOG:  shutting down
2024-02-18 01:32:08.296 CST [29731] LOG:  checkpoint starting: shutdown immediate
2024-02-18 01:32:08.299 CST [29731] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) a
dded, 0 removed, 0 recycled; write=0.001 s, sync=0.001 s, total=0.004 s; sync files=0, longest=0.000 s
, average=0.000 s; distance=0 kB, estimate=244 kB; lsn=0/154A960, redo lsn=0/154A960
2024-02-18 01:32:08.303 CST [29730] LOG:  database system is shut down
done
server stopped
waiting for server to start...2024-02-18 01:32:08.440 CST [29977] LOG:  starting PostgreSQL 16.2 (Pos
tgres.app) on aarch64-apple-darwin21.0.0, compiled by Apple clang version 14.0.0 (clang-1400.0.29.102)
, 64-bit
2024-02-18 01:32:08.444 CST [29977] LOG:  listening on IPv6 address "::", port 5436
2024-02-18 01:32:08.444 CST [29977] LOG:  listening on IPv4 address "0.0.0.0", port 5436
2024-02-18 01:32:08.445 CST [29977] LOG:  listening on Unix socket "/tmp/.s.PGSQL.5430"
2024-02-18 01:32:08.457 CST [29998] LOG:  database system was shut down at 2024-02-18 01:32:08 CST
2024-02-18 01:32:08.466 CST [29977] LOG:  database system is ready to accept connections
done
server started
[Kanakalakshmi ~]$
```

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
```



```
Terminal Shell Edit View Window Help  
[Kanakalakshmi ~]$ pg_basebackup -h localhost -U replica_user --checkpoint=fast -D /tmp/replica_db/ -R  
--slot=data_node -C --port=5436  
2024-02-18 01:37:20.204 CST [29978] LOG: checkpoint starting: immediate force wait  
2024-02-18 01:37:20.264 CST [29978] LOG: checkpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) a  
dded, 0 removed, 1 recycled; write=0.001 s, sync=0.001 s, total=0.002 s; sync files=1, longest=0.001 s  
, average=0.001 s; distance=10965 kB, estimate=10965 kB; lsn=0/26000000, redo lsn=0/26000020  
[Kanakalakshmi ~]$  
[Kanakalakshmi ~]$ docker run --name replica-node --e POSTGRES_PASSWORD=password -d -p 5435:5432 pos  
tgres:postgres  
14fc33f217c12590e30d4243512b1d184d24024ee6a1fa95e6eb7bb1509fb  
[Kanakalakshmi ~]$
```

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.

```
Terminal Shell Edit View Window Help
[ Kanakalakshmi ~ ] $ pg_basebackup -h localhost -U replica_user --checkpoint=fast -D /tmp/replica_db/ -R
--slot=data_node -C --port=5436
2024-02-18 01:37:20.284 CST [29978] LOG: checkpoint starting: immediate force wait
2024-02-18 01:37:20.284 CST [29978] LOG: checkpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.001 s, sync=0.001 s, total=0.050 s; sync files=1, longest=0.001 s, average=0.001 s; distance=10965 KB, estimate=10965 KB; lsn=0/26000000, redo lsn=0/20000020
[ Kanakalakshmi ~ ] $

[ Kanakalakshmi replica_db ~ ] $ ls -l
total 384
-rw-r--r-- 1 wheel  9 Feb 18 01:37 PG_VERSION
-rw-r--r-- 1 wheel  225 Feb 18 01:37 backup_label
-rw-r--r-- 1 wheel 137324 Feb 18 01:37 backup_manifest
drwxr-xr-x 5 wheel  160 Feb 18 01:37 base
drwxr-xr-x 2 wheel 2848 Feb 18 01:37 global
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_commit_ts
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_dynshmem
-rw-r--r-- 1 wheel 5790 Feb 18 01:37 pg_hba.conf
-rw-r--r-- 1 wheel 2440 Feb 18 01:37 pg_ident.conf
drwxr-xr-x 5 wheel  160 Feb 18 01:37 pg_logical
drwxr-xr-x 4 wheel  128 Feb 18 01:37 pg_multixact
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_notify
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_replicat
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_serial
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_snapshots
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_stat
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_stat_tpm
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_subtrans
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_tblspc
drwxr-xr-x 2 wheel  64 Feb 18 01:37 pg_twophase
drwxr-xr-x 4 wheel  128 Feb 18 01:37 pg_wal
drwxr-xr-x 3 wheel  96 Feb 18 01:37 pg_xact
-rw-r--r-- 1 wheel 437 Feb 18 01:37 postgresql.auto.conf
-rw-r--r-- 1 wheel 29652 Feb 18 01:37 postgresql.conf
-rw-r--r-- 1 wheel  64 Feb 18 01:37 standby.signal
[ Kanakalakshmi replica_db ~ ] $
```

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.

```
Terminal Shell Edit View Window Help
[ Kanakalakshmi ~ ] $ pg_basebackup -h localhost -U replica_user --checkpoint=fast -D /tmp/replica_db/ -R
--slot=data_node -C --port=5436
2024-02-18 01:37:20.284 CST [29978] LOG: checkpoint starting: immediate force wait
2024-02-18 01:37:20.284 CST [29978] LOG: checkpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.001 s, sync=0.001 s, total=0.050 s; sync files=1, longest=0.001 s, average=0.001 s; distance=10965 KB, estimate=10965 KB; lsn=0/26000000, redo lsn=0/20000020
[ Kanakalakshmi ~ ] $

[ Kanakalakshmi replica_db ~ ] $ cat postgresql.auto.conf
# Do not edit this file manually!
# It will be overwritten by the ALTER SYSTEM command.
primary_conninfo = 'user=replica_user password=Users/ik/pgpass' channel_binding=prefer host=lo
calhost port=5436 sslmode=prefer sslcompression=0 sslcertmode=allow sslsin=1 ssl_min_protocol_versi
on=TLSv1.2 gssencmode=disable krbsrvname=postgres gssdelegation=0 target_session_attrs=any load_bal
ance=preferred
primary_slot_name = 'data_node'
[ Kanakalakshmi replica_db ~ ] $
```

Fig-80: update the configs in replica node.

```
Terminal Shell Edit View Window Help
[Kanakalakshmi ~]$ pg_basebackup -h localhost -U replica_user --checkpoint-fast -D /tmp/replica_db/ -R
--slot-data_node -C --port=5436
2024-02-18 01:37:28.284 CST [29978] LOG:  checkpoint starting: immediate force wait
2024-02-18 01:37:28.284 CST [29978] LOG:  checkpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.001 s, sync=0.001 s, total=0.008 s; sync files=1, longest=0.001 s, average=0.001 s; distance=10965 kB, estimate=10965 kB; lsn=0/28000028, redo lsn=0/28000028
[Kanakalakshmi ~]$

replica_db - nano postgresql.conf -- 90x52
LW PIDO 8.00 File: postgresql.conf Modified

#data_directory = 'ConfigDir'           # use data in another directory
#                                     # (change requires restart)
#hba_file = 'ConfigDir/pg_hba.conf'     # host-based authentication file
#                                     # (change requires restart)
#ident_file = 'ConfigDir/pg_ident.conf'  # ident configuration file
#                                     # (change requires restart)

# If external_pid_file is not explicitly set, no extra PID file is written.
#external_pid_file = ''                 # write an extra PID file
#                                     # (change requires restart)

#-----
# CONNECTIONS AND AUTHENTICATION
#-----

# - Connection Settings -

listen_addresses = '*'                  # what IP address(es) to listen on;
#                                     # comma-separated list of addresses;
#                                     # defaults to 'localhost'; use '*' for all
#                                     # (change requires restart)
port = 5437                             # (change requires restart)
max_connections = 100                   # (change requires restart)
#reserved_connections = 0               # (change requires restart)
#superuser_reserved_connections = 3     # (change requires restart)
#unix_socket_directories = '/tmp'       # (change requires restart)
#                                     # comma-separated list of directories
#                                     # (change requires restart)
#unix_socket_group = ''                 # (change requires restart)
#unix_socket_permissions = 0777        # begin with 0 to use octal notation
#                                     # (change requires restart)
#bonjour = off                          # advertise server via Bonjour
#                                     # (change requires restart)
#bonjour_name = ''                     # defaults to the computer name
#                                     # (change requires restart)

# - TCP settings -
# see "man tcp" for details

tcp_keepalives_idle = 0                 # TCP_KEEPTIME, in seconds;
# 0 selects the system default
tcp_keepalives_interval = 0            # TCP_KEEPINTVL, in seconds;
# 0 selects the system default
tcp_keepalives_count = 0               # TCP_KEEPCNT,
# 0 selects the system default
tcp_user_timeout = 0                  # TCP_USER_TIMEOUT, in milliseconds;

Get Help WriteOut Read File Prev Pg Next Pg Cut Text Cur Pos
Exit Justify Where is Next Pg UnCut Text To Spell
```

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

```
Terminal Shell Edit View Window Help
[Kanakalakshmi ~]$ pg_basebackup -h localhost -U replica_user --checkpoint-fast -D /tmp/replica_db/ -R
--slot-data_node -C --port=5436
2024-02-18 01:37:28.284 CST [29978] LOG:  checkpoint starting: immediate force wait
2024-02-18 01:37:28.284 CST [29978] LOG:  checkpoint complete: wrote 1 buffers (0.0%); 0 WAL file(s) added, 0 removed, 1 recycled; write=0.001 s, sync=0.001 s, total=0.008 s; sync files=1, longest=0.001 s, average=0.001 s; distance=10965 kB, estimate=10965 kB; lsn=0/28000028, redo lsn=0/28000028
[Kanakalakshmi ~]$

replica_db - bash -- 90x52
[Kanakalakshmi replica_db]$ pg_ctl -D /tmp/replica_db start
waiting for server to start...2024-02-18 01:42:02.269 CST [30377] LOG:  starting PostgreSQL 16.2 (
Postgres.app) on aarch64-apple-darwin21.6.0, compiled by Apple clang version 14.0.0 (clang-1400.0.2
9.102), 64-bit
2024-02-18 01:42:02.282 CST [30377] LOG:  listening on IPv6 address '::', port 5437
2024-02-18 01:42:02.284 CST [30377] LOG:  listening on IPv4 address '0.0.0.0', port 5437
2024-02-18 01:42:02.295 CST [30380] LOG:  listening on Unix socket '/tmp/.s.PGSQL.5437'
2024-02-18 01:42:02.310 CST [30380] LOG:  database system is ready to accept connections
2024-02-18 01:37:28 CST
2024-02-18 01:42:02.401 CST [30380] LOG:  entering standby mode
2024-02-18 01:42:02.403 CST [30380] LOG:  starting backup recovery with redo LSN 0/28000028, checkpo
int LSN 0/28000000, on timeline ID 1
2024-02-18 01:42:02.412 CST [30380] LOG:  redo starts at 0/28000028
2024-02-18 01:42:02.412 CST [30380] LOG:  completed backup recovery with redo LSN 0/28000028 and end
LSN 0/28000100
2024-02-18 01:42:02.413 CST [30380] LOG:  consistent recovery state reached at 0/28000100
2024-02-18 01:42:02.413 CST [30377] LOG:  database system is ready to accept read-only connections
done
server started
[Kanakalakshmi replica_db]$ 2024-02-18 01:42:02.445 CST [30381] LOG:  started streaming WAL from pr
imary at 0/30000000 on timeline 1
[Kanakalakshmi replica_db]$
```

Fig-82: start postgres instance-2.


```
Terminal Shell Edit View Window Help
ll - psql postgres --port=5436 -- 102x53
postgres=# select * from pg_stat_replication;
-[ RECORD 1 ]-----
 pid          | 38382
 usersysid    | 16388
 username     | replica_user
 application_name | walreceiver
 client_addr   | ::1
 client_hostname |
 client_port   | 53572
 backend_start | 2024-02-18 01:42:02.441506-06
 backend_xmin  |
 state        | streaming
 sent_lsn     | 0/3000148
 write_lsn    | 0/3000148
 flush_lsn    | 0/3000148
 replay_lsn   | 0/3000148
 write_lag    |
 flush_lag    |
 replay_lag    |
 sync_priority | 0
 sync_state    | async
 sync_time    | 2024-02-18 01:44:57.900348-06
 reply_time    |
 postgres=#

replica_db - psql postgres --port=5437 -- 99x52
postgres=# select * from pg_stat_wal_receiver;
-[ RECORD 1 ]-----
 pid          | 30381
 status       | streaming
 receive_start_lsn | 0/30000000
 receive_start_tli | 1
 written_lsn    | 0/3000148
 flushed_lsn    | 0/3000148
 received_tli    | 1
 last_msg_send_time | 2024-02-18 01:46:27.804475-06
 last_msg_receipt_time | 2024-02-18 01:46:27.806607-06
 latest_end_lsn | 0/3000148
 latest_end_time | 2024-02-18 01:42:26.977733-06
 slot_name     | data_node
 sender_host    | localhost
 sender_port    | 5436
 conninfo       | user=replica_user password=Users/ll/.pgpass channel_binding=prefer dbname=
 replication_host=localhost port=5436 fallback_application_name=walreceiver sslmode=prefer sslcompression=
 sslcertmode=allow sslmin=1 ssl_max_protocol_version=TLSv1.2 gssencmode=disable krbservernam=
 postgres gssdelegation= target_session_attrs=any load_balance_hosts=disable
postgres=#
```

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}$'),  
CONSTRAINT invalid_customer_email CHECK (email_id ~ '^[A-Za-z0-9._%+-]+@[A-Za-z0-9-  
9.-]+\.[A-Za-z]{2,}$'),  
UNIQUE (phone_no),  
UNIQUE (email_id));
```

Instance-2:

```
select * from customer;
```

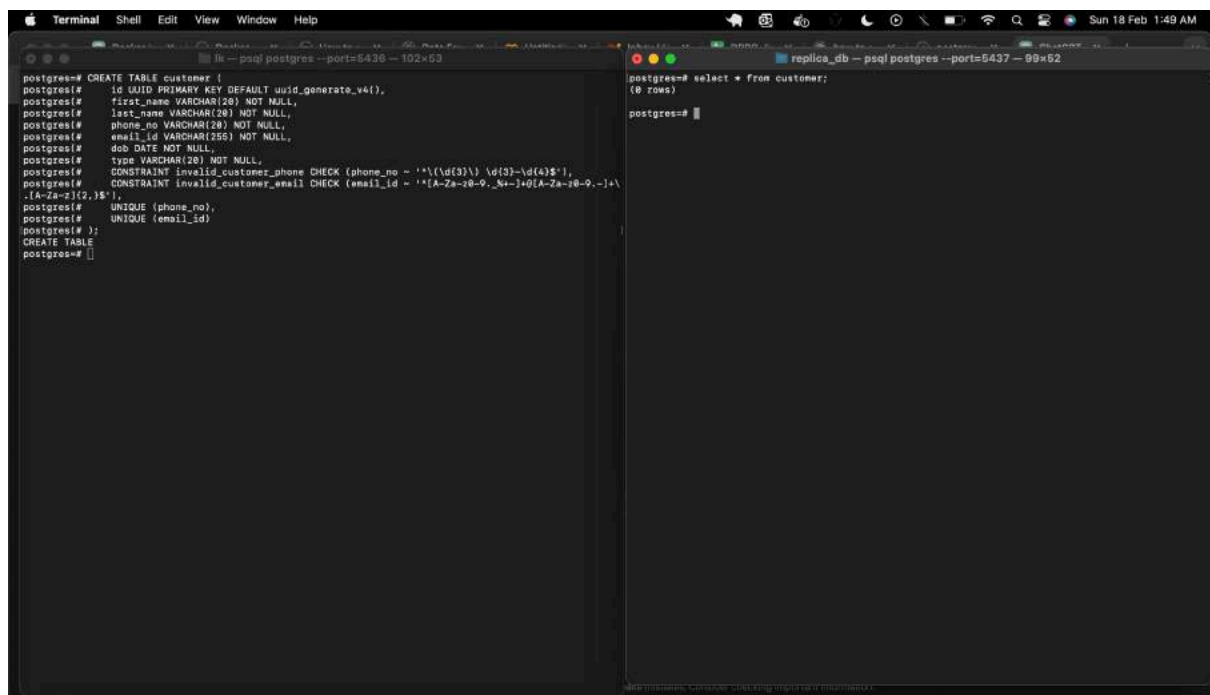


Fig-84: Create the table in instance-1 and verify data in instance-2.

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

```
Terminal Shell Edit View Window Help

psql postgres --port=5436 -- 102x53
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('c6803f11-5b57-4aaa-93c7-94cf3258c4e6', 'pHwYDFNNUN', 'yilpNOJDwQ', '(945) 139-5427', 'zolf6@example.com', '1965-02-28', 'VIP');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('424328d7-e2b0-4612-9e2f-3e8fbff94684', 'zx3XUxwxi', 'QfnFye10Jb', '(713) 878-2316', 'yypkg@example.com', '1979-12-11', 'platinum');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('7dfc1c39-382a-4ecd-bde1-bae2b4b3dd6f', 'PRyhAZxvTE', 'GYbbDUmQrX', '(784) 423-3265', 'qlsd1@example.com', '1996-01-22', 'regular');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('156da8f6-97a5-43bc-98b4-7ba3bd319385', 'aWMDUvArq', 'WeKkRbietu', '(218) 988-4574', 'wli1@example.com', '1976-12-14', 'VIP');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('ebf5ca25-7ca1-417b-886d-66e4fd816803', 'SWezesdk1Z', 'lufhoXRtl', '(425) 387-4568', 'aqxw1@example.com', '1994-04-08', 'premium');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('13cd8652-c588-484f-8866-a89c1bdd2f7c', 'ByjHCLrAog', 'TutYsVHgZ', '(966) 263-4916', 'dmzdn@example.com', '1976-01-17', 'silver');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('ab1f8e35-dd27-43f9-a5e5-62477a5bda59', 'DsgLsWoiS', 'nSuSpXOSAT', '(666) 154-3766', 'tjknm@example.com', '1974-01-26', 'premium');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('a3a584e4-8352-46db-bccd-488419185ac4', 'TLHAICBUQR', 'ouOWGPqMq', '(522) 942-1784', 'yyvzr@example.com', '1987-06-21', 'regular');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('d683a18f-f846-4171-a885-2d827caf6218', 'JmBrCctVKM', 'zZLVDOxrW', '(522) 374-4861', 'bgwqk@example.com', '1965-04-03', 'platinum');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('7a4a978b-880c-494b-94ab-ba5f8-772d26e8dd1a', 'HawnldySx', 'kYZUTFcGUC', '(378) 533-1889', 'qdkk1@example.com', '1984-10-23', 'VIP');
INSERT 0 1
postgres=#
postgres=#
```

```
psql postgres --port=5437 -- 99x52
postgres=# select * from customer;
(8 rows)

postgres=# select count(*) from customer;
-- RECORD 1 |
count | 8

postgres=#
```

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).

```
Terminal Shell Edit View Window Help

psql postgres --port=5436 -- 102x53
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('c6803f11-5b57-4aaa-93c7-94cf3258c4e6', 'pHwYDFNNUN', 'yilpNOJDwQ', '(945) 139-5427', 'zolf6@example.com', '1965-02-28', 'VIP');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('424328d7-e2b0-4612-9e2f-3e8fbff94684', 'zx3XUxwxi', 'QfnFye10Jb', '(713) 878-2316', 'yypkg@example.com', '1979-12-11', 'platinum');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('7dfc1c39-382a-4ecd-bde1-bae2b4b3dd6f', 'PRyhAZxvTE', 'GYbbDUmQrX', '(784) 423-3265', 'qlsd1@example.com', '1996-01-22', 'regular');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('156da8f6-97a5-43bc-98b4-7ba3bd319385', 'aWMDUvArq', 'WeKkRbietu', '(218) 988-4574', 'wli1@example.com', '1976-12-14', 'VIP');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('ebf5ca25-7ca1-417b-886d-66e4fd816803', 'SWezesdk1Z', 'lufhoXRtl', '(425) 387-4568', 'aqxw1@example.com', '1994-04-08', 'premium');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('13cd8652-c588-484f-8866-a89c1bdd2f7c', 'ByjHCLrAog', 'TutYsVHgZ', '(966) 263-4916', 'dmzdn@example.com', '1976-01-17', 'silver');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('ab1f8e35-dd27-43f9-a5e5-62477a5bda59', 'DsgLsWoiS', 'nSuSpXOSAT', '(666) 154-3766', 'tjknm@example.com', '1974-01-26', 'premium');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('a3a584e4-8352-46db-bccd-488419185ac4', 'TLHAICBUQR', 'ouOWGPqMq', '(522) 942-1784', 'yyvzr@example.com', '1987-06-21', 'regular');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('d683a18f-f846-4171-a885-2d827caf6218', 'JmBrCctVKM', 'zZLVDOxrW', '(522) 374-4861', 'bgwqk@example.com', '1965-04-03', 'platinum');
INSERT 0 1
postgres=# INSERT INTO customer (id, first_name, last_name, phone_no, email_id, dob, type) VALUES ('7a4a978b-880c-494b-94ab-ba5f8-772d26e8dd1a', 'HawnldySx', 'kYZUTFcGUC', '(378) 533-1889', 'qdkk1@example.com', '1984-10-23', 'VIP');
INSERT 0 1
postgres=#
postgres=#
postgres=# \q
[Kanakalakshmi -] 2024-02-18 01:52:28.294 CST [29978] LOG: checkpoint starting: time
2024-02-18 01:52:27.188 CST [29978] LOG: checkpoint complete: wrote 69 buffers (0.4%); 0 WAL file(s) added, 0 removed, 0 recycled; write=0.877 s, sync=0.918 s, total=0.895 s; sync files=32, longest=0.002 s, average=0.001 s; distance=237 kB, estimate=16769 kB; lsn=0/3838528, redo lsn=0/38384E8
postgres=#
```

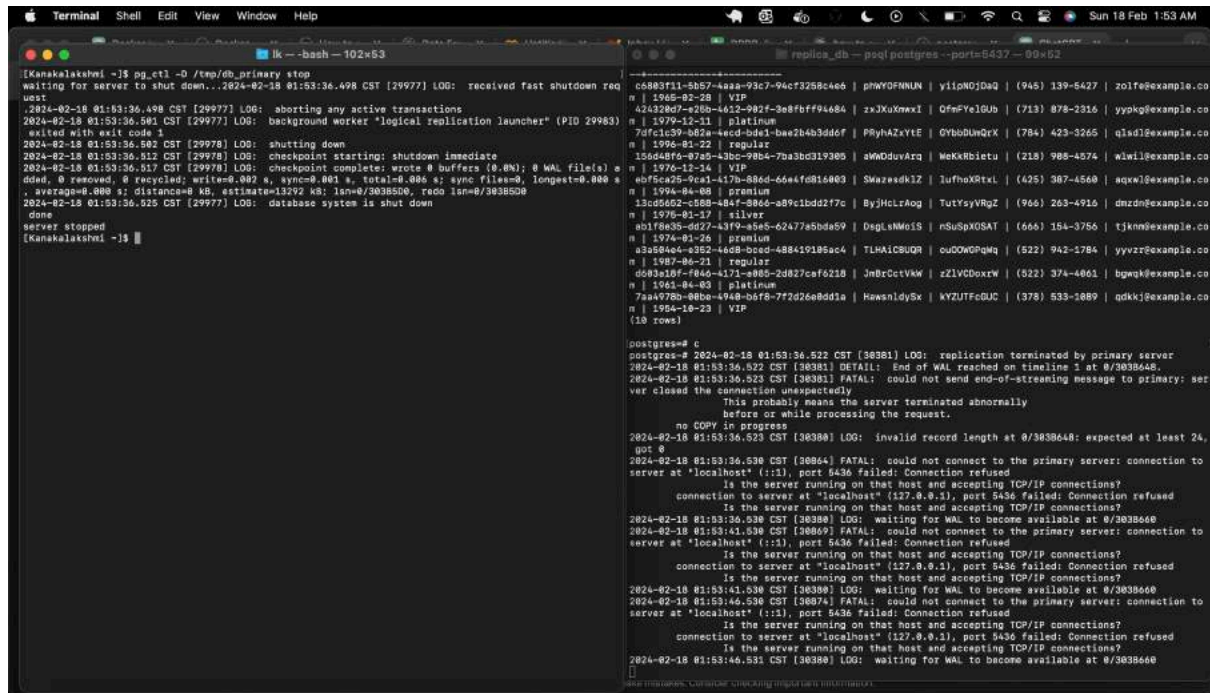
```
psql postgres --port=5437 -- 99x52
postgres=# select * from customer;
 id | dob | type | first_name | last_name | phone_no | email_id
----+----+----+-----+-----+-----+-----
 c6803f11-5b57-4aaa-93c7-94cf3258c4e6 | pHwYDFNNUN | yilpNOJDwQ | (945) 139-5427 | zolf6@example.co
 m | 1965-02-28 | VIP
 424328d7-e2b0-4612-9e2f-3e8fbff94684 | zx3XUxwxi | QfnFye10Jb | (713) 878-2316 | yypkg@example.co
 m | 1979-12-11 | platinum
 7dfc1c39-382a-4ecd-bde1-bae2b4b3dd6f | PRyhAZxvTE | GYbbDUmQrX | (784) 423-3265 | qlsd1@example.co
 m | 1996-01-22 | regular
 156da8f6-97a5-43bc-98b4-7ba3bd319385 | aWMDUvArq | WeKkRbietu | (218) 988-4574 | wli1@example.co
 m | 1976-12-14 | VIP
 ebf5ca25-7ca1-417b-886d-66e4fd816803 | SWezesdk1Z | lufhoXRtl | (425) 387-4568 | aqxw1@example.co
 m | 1994-04-08 | premium
 13cd8652-c588-484f-8866-a89c1bdd2f7c | ByjHCLrAog | TutYsVHgZ | (966) 263-4916 | dmzdn@example.co
 m | 1976-01-17 | silver
 ab1f8e35-dd27-43f9-a5e5-62477a5bda59 | DsgLsWoiS | nSuSpXOSAT | (666) 154-3766 | tjknm@example.co
 m | 1974-01-26 | premium
 a3a584e4-8352-46db-bccd-488419185ac4 | TLHAICBUQR | ouOWGPqMq | (522) 942-1784 | yyvzr@example.co
 m | 1987-06-21 | regular
 d683a18f-f846-4171-a885-2d827caf6218 | JmBrCctVKM | zZLVDOxrW | (522) 374-4861 | bgwqk@example.co
 m | 1965-04-03 | platinum
 7a4a978b-880c-494b-94ab-ba5f8-772d26e8dd1a | HawnldySx | kYZUTFcGUC | (378) 533-1889 | qdkk1@example.co
 m | 1984-10-23 | VIP
(18 rows)

postgres=# c
postgres=#
```

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

query:

pg_ctl -D /tmp/primary_db stop --stopping the node1

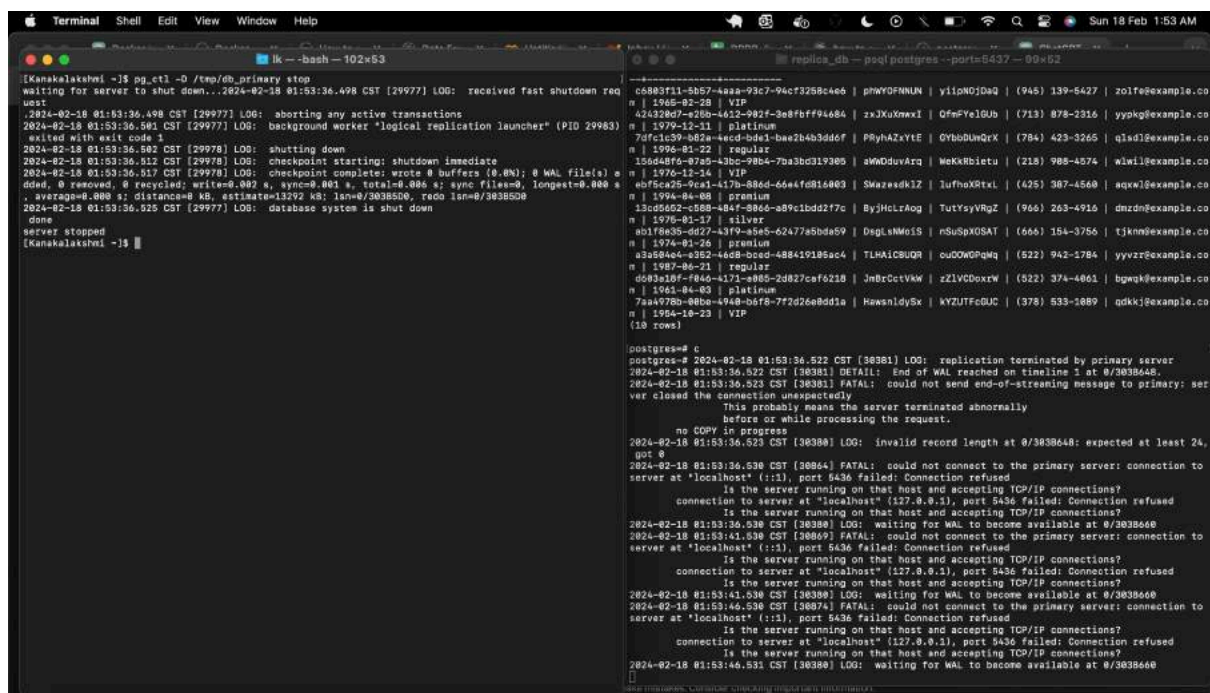


```
[kanakalakshmi ~]$ pg_ctl -D /tmp/primary_db stop
waiting for server to shut down...2024-02-18 01:53:36.498 CST [29977] LOG:  received fast shutdown request
2024-02-18 01:53:36.498 CST [29977] LOG:  aborting any active transactions
2024-02-18 01:53:36.501 CST [29977] LOG:  background worker "logical replication launcher" (PID 29983)
        exited with exit code 1
2024-02-18 01:53:36.502 CST [29978] LOG:  shutting down
2024-02-18 01:53:36.512 CST [29978] LOG:  checkpoint starting: shutdown immediate
2024-02-18 01:53:36.517 CST [29978] LOG:  checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s)
        removed, 0 recycled; write=0.002 s, sync=0.001 s, total=0.006 s; sync files=0, longest=0.000 s
        , average=0.000 s; distance=0 kB, estimate=13292 kB; lsn=0/3038500, redo lsn=0/3038500
2024-02-18 01:53:36.525 CST [29977] LOG:  database system is shut down
done
server stopped
[kanakalakshmi ~]$
```

```
postgres= c
postgres= 2024-02-18 01:53:36.522 CST [30381] LOG:  replication terminated by primary server
2024-02-18 01:53:36.522 CST [30381] DETAIL:  End of WAL reached on timeline 1 at 0/3038648.
2024-02-18 01:53:36.523 CST [30381] FATAL:  could not send end-of-streaming message to primary: ser
        ver closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
no COPY in progress
2024-02-18 01:53:36.523 CST [30380] LOG:  invalid record length at 0/3038648: expected at least 24,
        got 0
2024-02-18 01:53:36.530 CST [30864] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
connection to server at "localhost" (127.0.0.1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.530 CST [30869] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.530 CST [30874] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.531 CST [30880] LOG:  waiting for WAL to become available at 0/3038648
(10 rows)
```

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).



```
[kanakalakshmi ~]$ pgrep -f postgres
postgres= c
postgres= 2024-02-18 01:53:36.522 CST [30381] LOG:  replication terminated by primary server
2024-02-18 01:53:36.522 CST [30381] DETAIL:  End of WAL reached on timeline 1 at 0/3038648.
2024-02-18 01:53:36.523 CST [30381] FATAL:  could not send end-of-streaming message to primary: ser
        ver closed the connection unexpectedly
        This probably means the server terminated abnormally
        before or while processing the request.
no COPY in progress
2024-02-18 01:53:36.523 CST [30380] LOG:  invalid record length at 0/3038648: expected at least 24,
        got 0
2024-02-18 01:53:36.530 CST [30864] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
connection to server at "localhost" (127.0.0.1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.530 CST [30869] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.530 CST [30874] FATAL:  could not connect to the primary server: connection to
        server at "localhost" (::1), port 5436 failed: Connection refused
Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:53:36.531 CST [30880] LOG:  waiting for WAL to become available at 0/3038648
(10 rows)
```

Fig-88: Observe logs in instance-2


```
Terminal Shell Edit View Window Help
[kanakalakshmi ~]$ pg_ctl -D /tmp/db_primary stop
waiting for server to shut down...2024-02-18 01:53:36.498 CST [29977] LOG: received fast shutdown request
2024-02-18 01:53:36.498 CST [29977] LOG: aborting any active transactions
2024-02-18 01:53:36.501 CST [29977] LOG: background worker "logical replication launcher" (PID 29983)
exited with exit code 1
2024-02-18 01:53:36.502 CST [29978] LOG: shutting down
2024-02-18 01:53:36.512 CST [29978] LOG: checkpoint starting: shutdown immediate
2024-02-18 01:53:36.517 CST [29978] LOG: checkpoint complete: wrote 0 buffers (0.0%); 0 WAL file(s) a
dded, 0 removed, 0 recycled; write=0.882 s, sync=0.801 s, total=0.886 s; sync files=0, longest=0.800 s
, average=0.000 s; distance=0 kB, estimate=13292 kB; lsn=0/30385D6, redo lsn=0/30385D6
2024-02-18 01:53:36.525 CST [29977] LOG: database system is shut down
done
server stopped
[kanakalakshmi ~]$

postgres=# select * from customer;
      id      | dob      | type      | first_name | last_name | phone_no | email_id
-----+-----+-----+-----+-----+-----+-----
 c6883f11-5b57-4a9a-93c7-94cf3258c4e6 | 1994-02-28 | VIP      | ylipNOjDeQ |  | 1994-02-28 | zolfe@example.co
m | 2979-12-31 | platinum | 7cfc1c39-082a-4acd-bd61-bae2b4b3d66f | PRynAZxvtE | 6VnbDUmQrX | 784 | 423-3265 | qlsdl@example.co
m | 1996-01-22 | regular  | 156d48f6-07a5-43dc-90b4-7ca3bd319395 | aWMDouVArq | WeKARbietu | 218 | 988-4574 | xwil@example.co
m | 1976-12-14 | VIP      | abf5ca25-9ca1-417b-886d-66e4fd81a003 | SMazeadKIZ | luThoXRtXl | 425 | 387-4568 | aqxw@example.co
m | 1994-04-08 | premium | 13cd5652-c888-484f-8066-a89c1bd2f7fc | ByjHclRag | TuttsyVqZ | 966 | 263-4916 | dmzd@example.co
m | 1976-01-17 | silver   | ab1f8a35-d027-43f9-a5e5-62477a5bda59 | DsplsWoiS | nSutSpKOSAT | 666 | 154-3756 | tjknm@example.co
m | 1974-01-26 | premium | a3a884a4-a3d2-46d8-bced-488419185ac4 | TLHAIcBUQR | ou0OWpQWq | 522 | 942-1784 | yvzr@example.co
m | 1997-06-21 | regular  | d683a18f-f046-4171-a088-2d827c9f6218 | JmBrCotVkw | zZlVCDoxrW | 522 | 374-4861 | bmqw@example.co
m | 1961-04-03 | platinum | 7a5e97db-nb8e-4948-b6f8-7f2d26e8dd1a | HawnldySx | kVZUTFc0UC | 378 | 533-1089 | qdkk@example.co
m | 1994-10-23 | VIP      | 7a5e97db-nb8e-4948-b6f8-7f2d26e8dd1a | HawnldySx | kVZUTFc0UC | 378 | 533-1089 | qdkk@example.co
(18 rows)

postgres=# select * from customer;
      id      | dob      | type      | first_name | last_name | phone_no | email_id
-----+-----+-----+-----+-----+-----+-----
 c6883f11-5b57-4a9a-93c7-94cf3258c4e6 | 1994-02-28 | VIP      | ylipNOjDeQ |  | 1994-02-28 | zolfe@example.co
m | 2979-12-31 | platinum | 7cfc1c39-082a-4acd-bd61-bae2b4b3d66f | PRynAZxvtE | 6VnbDUmQrX | 784 | 423-3265 | qlsdl@example.co
m | 1996-01-22 | regular  | 156d48f6-07a5-43dc-90b4-7ca3bd319395 | aWMDouVArq | WeKARbietu | 218 | 988-4574 | xwil@example.co
m | 1976-12-14 | VIP      | abf5ca25-9ca1-417b-886d-66e4fd81a003 | SMazeadKIZ | luThoXRtXl | 425 | 387-4568 | aqxw@example.co
m | 1994-04-08 | premium | 13cd5652-c888-484f-8066-a89c1bd2f7fc | ByjHclRag | TuttsyVqZ | 966 | 263-4916 | dmzd@example.co
m | 1976-01-17 | silver   | ab1f8a35-d027-43f9-a5e5-62477a5bda59 | DsplsWoiS | nSutSpKOSAT | 666 | 154-3756 | tjknm@example.co
m | 1974-01-26 | premium | a3a884a4-a3d2-46d8-bced-488419185ac4 | TLHAIcBUQR | ou0OWpQWq | 522 | 942-1784 | yvzr@example.co
m | 1997-06-21 | regular  | d683a18f-f046-4171-a088-2d827c9f6218 | JmBrCotVkw | zZlVCDoxrW | 522 | 374-4861 | bmqw@example.co
m | 1961-04-03 | platinum | 7a5e97db-nb8e-4948-b6f8-7f2d26e8dd1a | HawnldySx | kVZUTFc0UC | 378 | 533-1089 | qdkk@example.co
m | 1994-10-23 | VIP      | 7a5e97db-nb8e-4948-b6f8-7f2d26e8dd1a | HawnldySx | kVZUTFc0UC | 378 | 533-1089 | qdkk@example.co
(18 rows)

[END] 2024-02-18 01:54:11.554 CST [30906] FATAL: could not connect to the primary server: connectio
n to server at "localhost" (::1), port 5436 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
        connection to server at "localhost" (127.0.0.1), port 5436 failed: Connection refused
        Is the server running on that host and accepting TCP/IP connections?
2024-02-18 01:54:11.555 CST [30389] LOG: waiting for WAL to become available at 0/3038560
```

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;
```

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash - 204x55

postgres=# 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;

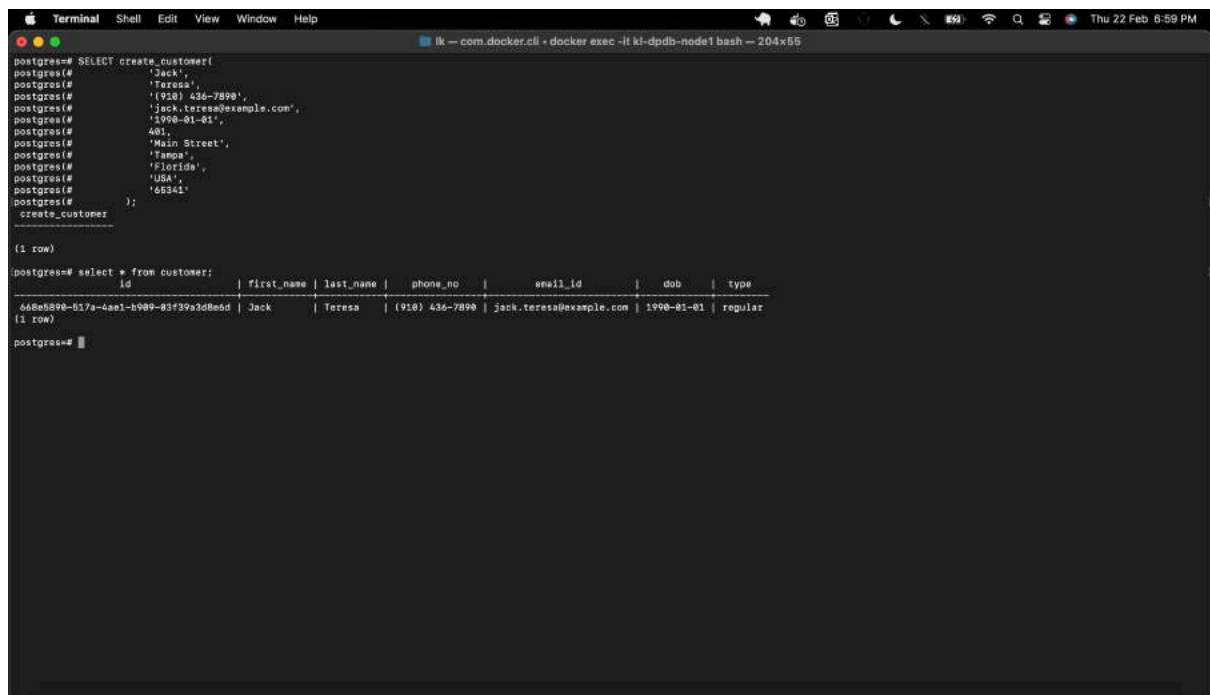
  -- Insert into customer_address table
  INSERT INTO customer_address (customer_id, address_id, default_address)
  VALUES (v_customer_id, v_address_id, true);
END;
$$ LANGUAGE plpgsql PARALLEL SAFE;
CREATE FUNCTION
postgres=#
```

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, email_id, 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 be 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'  
);
```



```
Terminal Shell Edit View Window Help  
ik - com.docker.cli - docker exec -it ki-dpdb-node1 bash - 204x65  
postgres=# SELECT create_customer(  
postgres#     'Jack',  
postgres#     'Teresa',  
postgres#     '(910) 436-7890',  
postgres#     'jack.teresa@example.com',  
postgres#     '1990-01-01',  
postgres#     401,  
postgres#     'Main Street',  
postgres#     'Tampa',  
postgres#     'Florida',  
postgres#     'USA',  
postgres#     '65341'  
postgres# );  
create_customer  
-----  
(1 row)  
  
postgres=# select * from customer;  
          id          | first_name | last_name | phone_no | email_id          | dob          | type  
-----  
668e5898-517a-4ae1-b989-83f39a3d8e6d | Jack      | Teresa    | (910) 436-7890 | jack.teresa@example.com | 1990-01-01 | regular  
(1 row)  
  
postgres=#
```

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;
```

```
Terminal Shell Edit View Window Help
ik - com.docker.cli - docker exec -it kl-dpdb-node1 bash - 204x55

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

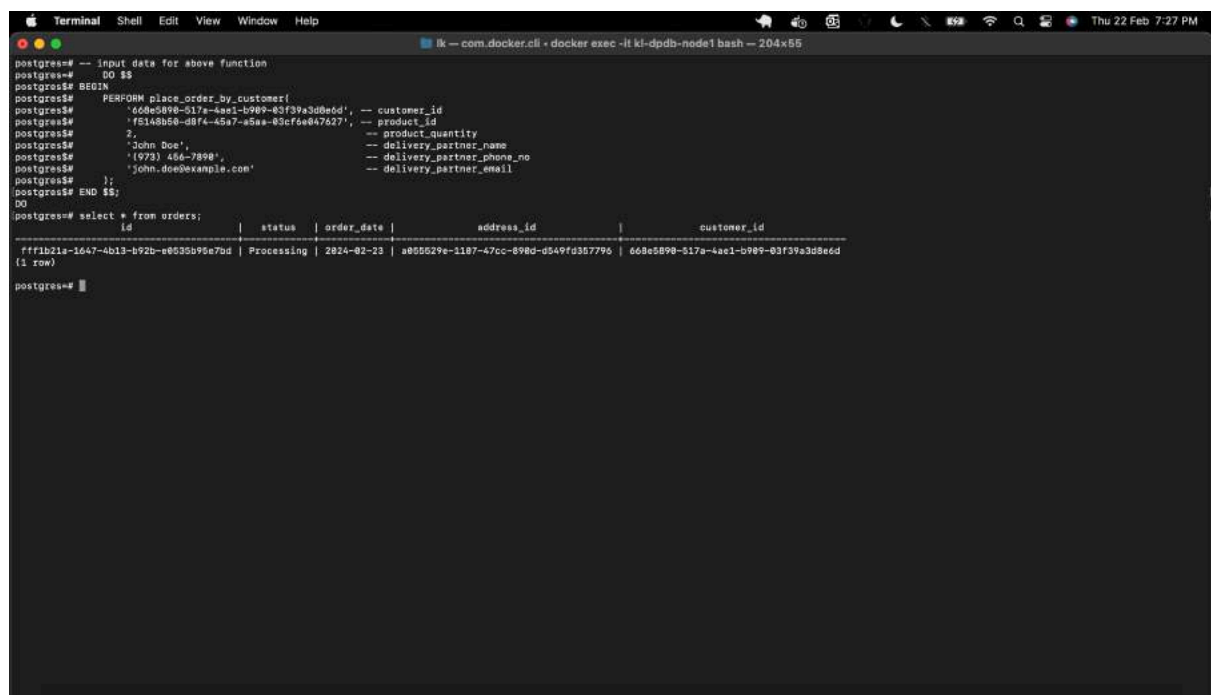
Fig-92: Create function to place order by customer.

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



The screenshot shows a terminal window with a PostgreSQL prompt. The user enters a DO block containing a call to the `place_order_by_customer` function with specific parameters. After execution, the user runs a `select` query to view the `orders` table. The result shows a single row with the order details.

```
postgres=# -- Input data for above function
postgres=# DO $$
postgres=# BEGIN
postgres=# PERFORM place_order_by_customer(
postgres=# '668e5890-517a-4ae1-b909-03f39a3d8e6d', -- customer_id
postgres=# 'f5148b50-d8f4-45a7-a5aa-03cf6e047627', -- product_id
postgres=# 2, -- product_quantity
postgres=# 'John Doe', -- delivery_partner_name
postgres=# '(973) 456-7890', -- delivery_partner_phone_no
postgres=# 'john.doe@example.com' -- delivery_partner_email
postgres=# );
postgres=# END $$;
DO
postgres=# select * from orders;
 id | status | order_date | address_id | customer_id
----+-----+-----+-----+-----
 fffb21a-1647-4b13-b92b-e0035b96e7bd | Processing | 2024-02-23 | a805029e-1187-47cc-898d-d549fd357796 | 668e5890-517a-4ae1-b909-03f39a3d8e6d
(1 row)

postgres=#
```

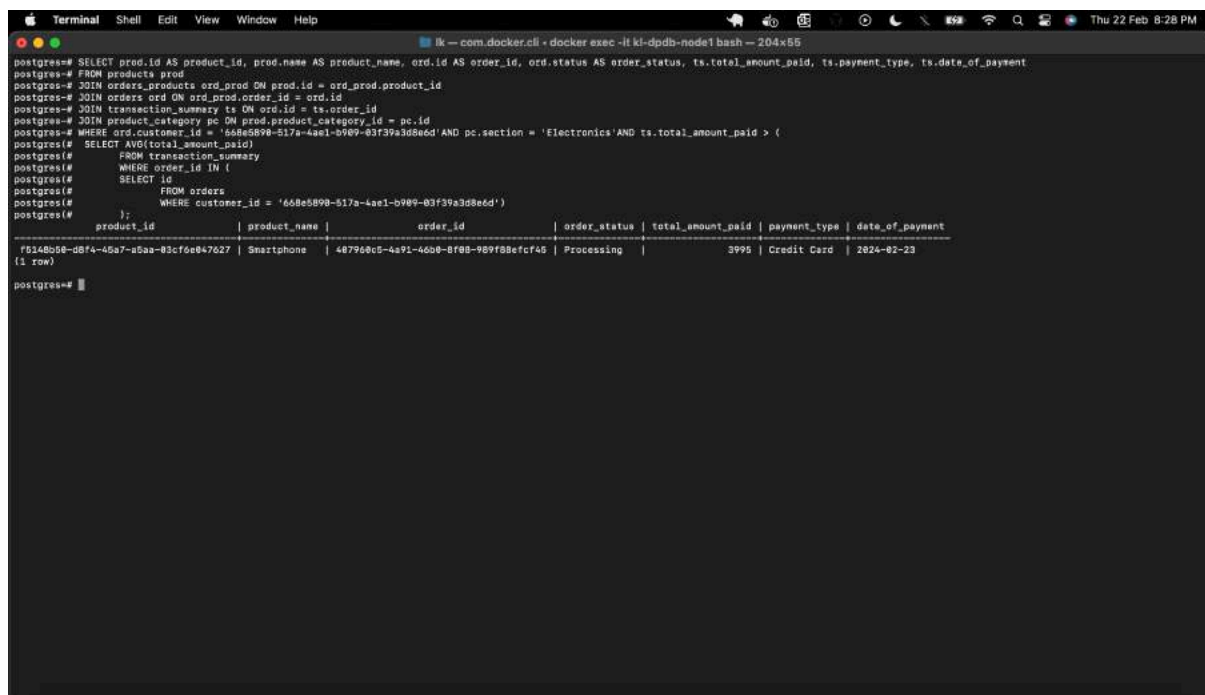
Fig-93: Input data to trigger `place_order_by_customer` and display the orders.

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')
    );
```



```
postgres=# 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
postgres=# FROM products prod
postgres=# JOIN orders_products ord_prod ON prod.id = ord_prod.product_id
postgres=# JOIN orders ord ON ord_prod.order_id = ord.id
postgres=# JOIN transaction_summary ts ON ord.id = ts.order_id
postgres=# JOIN product_category pc ON prod.product_category_id = pc.id
postgres=# WHERE ord.customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d' AND pc.section = 'Electronics' AND ts.total_amount_paid > (
postgres=# SELECT AVG(total_amount_paid)
postgres=# FROM transaction_summary
postgres=# WHERE order_id IN (
postgres=# SELECT id
postgres=# FROM orders
postgres=# WHERE customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d')
postgres=# );
 product_id | product_name | order_id | order_status | total_amount_paid | payment_type | date_of_payment
-----
'668e5890-517a-4ae1-b909-03f39a3d8e6d' | Smartphone | 4e7960c5-4a91-4608-8f08-9b9f88efcf45 | Processing | 3995 | Credit Card | 2024-02-23
(1 row)
```

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')
);
```

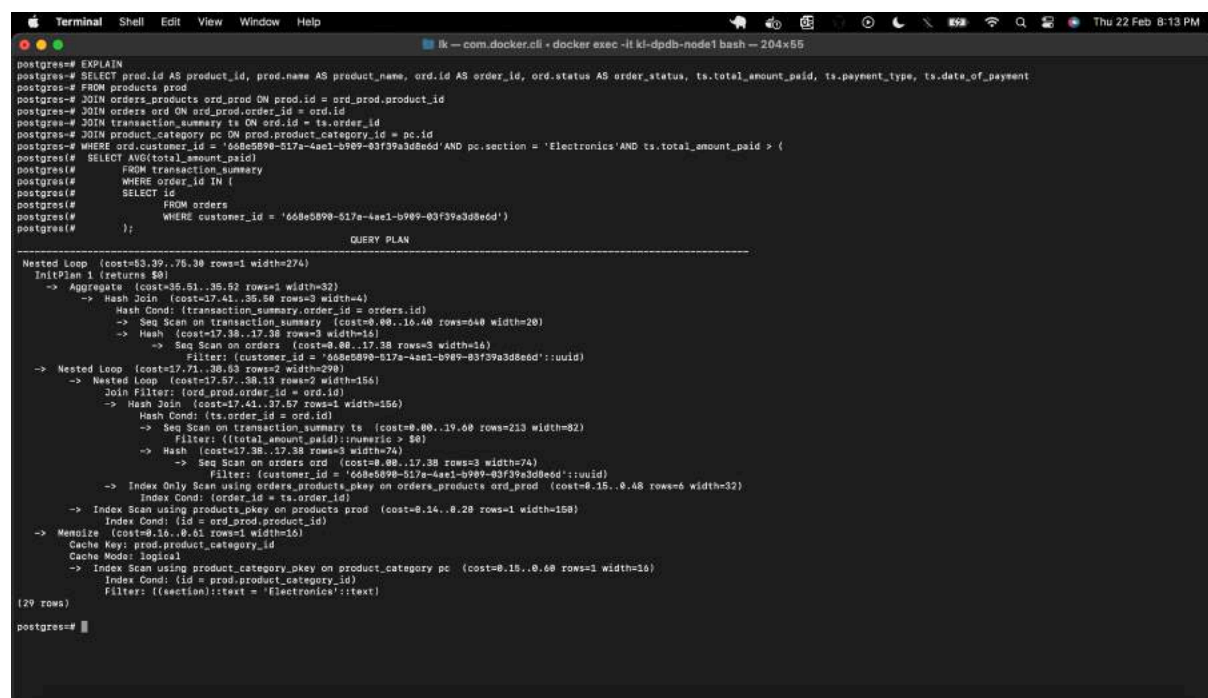
A terminal window titled 'Terminal' with a dark background. It shows the execution of a PostgreSQL query with the EXPLAIN command. The output displays the query plan, including nested loops, hash joins, and index scans. The plan shows a series of joins between tables: products, orders_products, orders, transaction_summary, and product_category. It also includes an aggregate function (AVG) and a filter for the 'Electronics' section. The plan is detailed, showing costs, row counts, and widths for each step. The terminal window has a title bar with standard macOS window controls and a status bar at the bottom showing the date and time (Thu 22 Feb 8:13 PM).

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.

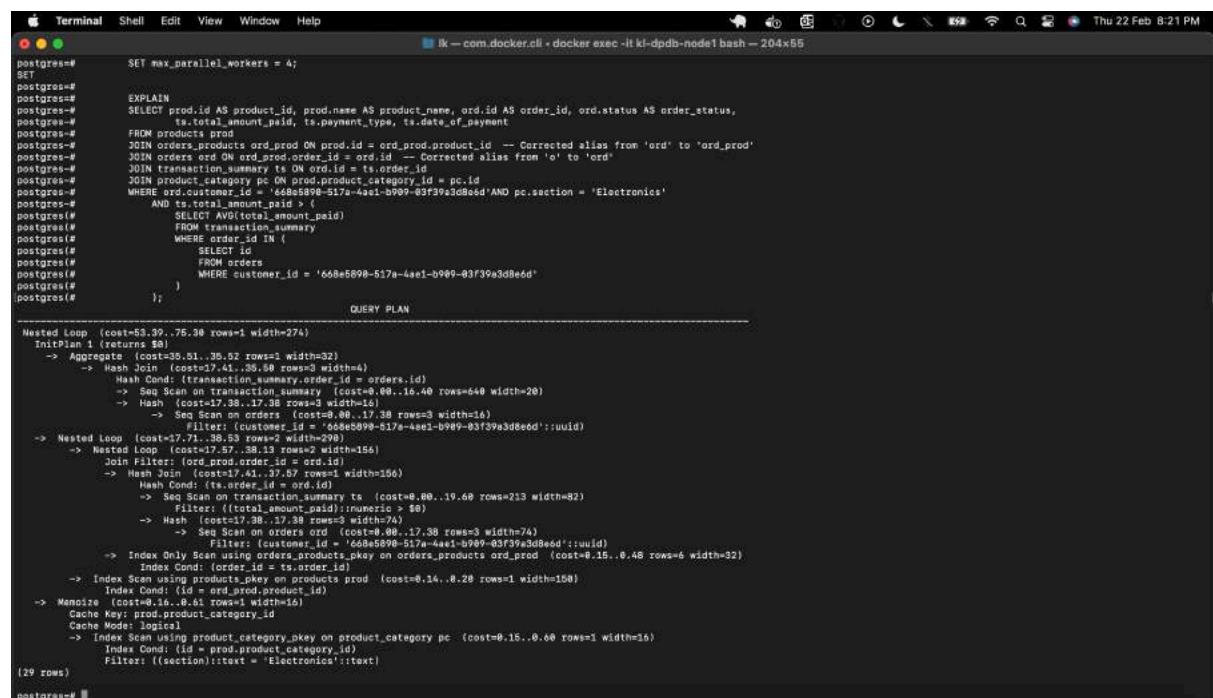
Validate Parallel execution of query:

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'
    )
);
```



```
postgres=# SET max_parallel_workers = 4;
SET
postgres=#
postgres=# 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'
    )
);
QUERY PLAN
Nested Loop (cost=53.99..75.30 rows=1 width=274)
  InitPlan 1 (returns $0)
    -> Aggregate (cost=35.51..35.52 rows=1 width=32)
      -> Hash Join (cost=17.41..35.50 rows=3 width=4)
        Hash Cond: (transaction_summary.order_id = orders.id)
        -> Seq Scan on transaction_summary (cost=0.00..16.40 rows=640 width=20)
        -> Hash (cost=17.38..17.38 rows=3 width=16)
          -> Seq Scan on orders (cost=0.00..17.38 rows=3 width=16)
            Filter: (customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'::uuid)
      -> Nested Loop (cost=17.11..38.53 rows=2 width=290)
        -> Nested Loop (cost=17.57..38.13 rows=2 width=156)
          Join Filter: (ord_prod.order_id = ord.id)
          -> Hash Join (cost=17.41..37.57 rows=1 width=156)
            Hash Cond: (ts.order_id = ord.id)
            -> Seq Scan on transaction_summary ts (cost=0.00..19.60 rows=213 width=82)
              Filter: ((total_amount_paid)::numeric > $0)
            -> Hash (cost=17.38..17.38 rows=3 width=74)
              -> Seq Scan on orders ord (cost=0.00..17.38 rows=3 width=74)
                Filter: (customer_id = '668e5890-517a-4ae1-b909-03f39a3d8e6d'::uuid)
          -> Index Only Scan using orders_products_pkey on orders_products ord_prod (cost=0.15..0.48 rows=6 width=32)
            Index Cond: (order_id = ts.order_id)
        -> Index Scan using products_pkey on products prod (cost=0.14..0.20 rows=1 width=150)
          Index Cond: (id = ord_prod.product_id)
      -> Memoize (cost=0.16..0.61 rows=1 width=16)
        Cache Key: prod.product_category_id
        Cache Mode: logical
        -> Index Scan using product_category_pkey on product_category pc (cost=0.15..0.40 rows=1 width=16)
          Index Cond: (id = prod.product_category_id)
          Filter: ((section)::text = 'Electronics'::text)
(29 rows)
postgres=#
```

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 of it as default address. Customer can place order to any address as per choice. There is also flexibility for the customer to update the address, in case customer adds the same existing address then the database will throw 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 $$;

```

```

postgres=# select * from customer;
      id      | first_name | last_name | phone_no | email_id | dob | type
-----
668e5890-517a-4ae1-b909-03f39a3d8e6d | Jack      | Teresa   | (918) 436-7890 | jack.teresa@example.com | 1990-01-01 | regular
(1 row)

postgres=# select * from customer_address;
      customer_id      | address_id | default_address
-----
668e5890-517a-4ae1-b909-03f39a3d8e6d | a055529e-1107-47cc-890d-d549fd357796 | t
668e5890-517a-4ae1-b909-03f39a3d8e6d | 2b75623c-0c97-4c4d-a1be-536ecc775e2e | f
(2 rows)

postgres=# select * from address where id='a055529e-1107-47cc-890d-d549fd357796';
      id      | flat_no | street | city | state | country | zip_code
-----
a055529e-1107-47cc-890d-d549fd357796 | 401 | Stella Street | Denton | Florida | USA | 65431
(1 row)

postgres=# select * from address where id='2b75623c-0c97-4c4d-a1be-536ecc775e2e';
      id      | flat_no | street | city | state | country | zip_code
-----
2b75623c-0c97-4c4d-a1be-536ecc775e2e | 702 | Melody Street | Houston | Texas | USA | 77001
(1 row)

postgres=#

```

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

```
Terminal Shell Edit View Window Help
ik - com.docker.cli - docker exec -it ki-dpbd-node1 bash -- 204x55

postgres# DO $$
postgres# DECLARE
postgres#   -- Input parameters
postgres#   input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
postgres#   input_flat_no INTEGER := 101;
postgres#   input_street VARCHAR(50) := 'Maple Lane, Apartment 3C';
postgres#   input_city VARCHAR(50) := 'River City';
postgres#   input_state VARCHAR(50) := 'Texas';
postgres#   input_country VARCHAR(50) := 'USA';
postgres#   input_zip_code VARCHAR(10) := '75001';
postgres#
postgres#   -- Variables for data
postgres#   get_address_id UUID;
postgres#   get_existing_address RECORD;
postgres#
postgres# BEGIN
postgres#   -- Insert data into the address table
postgres#   INSERT INTO address (flat_no, street, city, state, country, zip_code)
postgres#   VALUES (input_flat_no, input_street, input_city, input_state, input_country, input_zip_code)
postgres#   RETURNING id INTO get_address_id;
postgres#   -- Insert data into the customer_address table
postgres#   INSERT INTO customer_address (customer_id, address_id, default_address)
postgres#   VALUES (input_customer_id, get_address_id, false);
postgres#   -- Check if any other address exists for the customer
postgres#   FOR get_existing_address IN
postgres#     SELECT addr.*
postgres#     FROM address addr
postgres#     JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
postgres#     WHERE cust_addr.customer_id = input_customer_id AND addr.id <> get_address_id
postgres#   LOOP
postgres#     -- Validate with the input
postgres#     IF get_existing_address.flat_no = input_flat_no AND
postgres#        get_existing_address.street = input_street AND
postgres#        get_existing_address.city = input_city AND
postgres#        get_existing_address.state = input_state AND
postgres#        get_existing_address.country = input_country AND
postgres#        get_existing_address.zip_code = input_zip_code THEN
postgres#       RAISE EXCEPTION 'Address already exists. Please make use of it.';
postgres#     END IF;
postgres#   END LOOP;
postgres#
postgres# EXCEPTION
postgres#   WHEN OTHERS THEN
postgres#     -- An error occurred, roll back the transaction
postgres#     RAISE NOTICE 'Triggered Error: N', SQLERRM;
postgres#     ROLLBACK;
postgres#     RETURN;
postgres#
postgres# -- Everything went well, commit the transaction
postgres# COMMIT;
postgres# RAISE NOTICE 'Address saved successfully';
postgres# END $$;
```

Fig-98: Code to add the Customer Address.

```
Terminal Shell Edit View Window Help
ik - com.docker.cli - docker exec -it ki-dpbd-node1 bash -- 204x55

postgres#   input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
postgres#   input_flat_no INTEGER := 101;
postgres#   input_street VARCHAR(50) := 'Maple Lane, Apartment 3C';
postgres#   input_city VARCHAR(50) := 'River City';
postgres#   input_state VARCHAR(50) := 'Texas';
postgres#   input_country VARCHAR(50) := 'USA';
postgres#   input_zip_code VARCHAR(10) := '75001';
postgres#
postgres#   -- Variables for data
postgres#   get_address_id UUID;
postgres#   get_existing_address RECORD;
postgres#
postgres# BEGIN
postgres#   -- Insert data into the address table
postgres#   INSERT INTO address (flat_no, street, city, state, country, zip_code)
postgres#   VALUES (input_flat_no, input_street, input_city, input_state, input_country, input_zip_code)
postgres#   RETURNING id INTO get_address_id;
postgres#   -- Insert data into the customer_address table
postgres#   INSERT INTO customer_address (customer_id, address_id, default_address)
postgres#   VALUES (input_customer_id, get_address_id, false);
postgres#   -- Check if any other address exists for the customer
postgres#   FOR get_existing_address IN
postgres#     SELECT addr.*
postgres#     FROM address addr
postgres#     JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
postgres#     WHERE cust_addr.customer_id = input_customer_id AND addr.id <> get_address_id
postgres#   LOOP
postgres#     -- Validate with the input
postgres#     IF get_existing_address.flat_no = input_flat_no AND
postgres#        get_existing_address.street = input_street AND
postgres#        get_existing_address.city = input_city AND
postgres#        get_existing_address.state = input_state AND
postgres#        get_existing_address.country = input_country AND
postgres#        get_existing_address.zip_code = input_zip_code THEN
postgres#       RAISE EXCEPTION 'Address already exists. Please make use of it.';
postgres#     END IF;
postgres#   END LOOP;
postgres#
postgres# EXCEPTION
postgres#   WHEN OTHERS THEN
postgres#     -- An error occurred, roll back the transaction
postgres#     RAISE NOTICE 'Triggered Error: N', SQLERRM;
postgres#     ROLLBACK;
postgres#     RETURN;
postgres#
postgres# -- Everything went well, commit the transaction
postgres# COMMIT;
postgres# RAISE NOTICE 'Address saved successfully';
postgres# END $$;
NOTICE: Address saved successfully
DO
postgres#
```

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.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# select * from customer;
      id      | first_name | last_name | phone_no | email_id | dob | type
-----+-----+-----+-----+-----+-----+-----
668e5890-517a-4ae1-b909-03f39a3d8e6d | jack      | Teresa   | (918) 436-7890 | jack.teresa@example.com | 1990-01-01 | regular
(1 row)

postgres=# select * from customer_address;
      customer_id      | address_id | default_address
-----+-----+-----
668e5890-517a-4ae1-b909-03f39a3d8e6d | #855529e-1187-47cc-898c-d549fd357796 | t
668e5890-517a-4ae1-b909-03f39a3d8e6d | 2b75623c-8c97-4cad-a1be-834ecc775e2e | f
668e5890-517a-4ae1-b909-03f39a3d8e6d | 381e8511-14e4-4526-b462-84ebb788fd1d | f
(3 rows)

postgres=# select * from address;
      id      | flat_no | street | city | state | country | zip_code
-----+-----+-----+-----+-----+-----+-----
2f9eb7f9-40e9-4883-9984-9208c6d7b7cc | 789 | Oak Street | San Francisco | California | United States | 94107
57de73a8-e753-4a58-8c15-fa8337648746 | 181 | Highland Avenue | London | England | United Kingdom | SW1A 1AA
a855529e-1187-47cc-898c-d549fd357796 | 401 | Stella Street | Denton | Florida | USA | 65541
2b75623c-8c97-4cad-a1be-834ecc775e2e | 782 | Melody Street | Houston | Texas | USA | 77881
381e8511-14e4-4526-b462-84ebb788fd1d | 101 | Maple lane, Apartment 3C | River City | Texas | USA | 75001
(5 rows)

postgres=#
```

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 $$;

```

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# DO $$
postgres=# DECLARE
postgres=# -- Input parameters
postgres=# input_customer_id UUID := '668e5890-517a-4ae1-b909-03f39a3d8e6d';
postgres=# input_flat_no INTEGER := 101;
postgres=# input_street VARCHAR(50) := 'Maple Lane, Apartment 3C';
postgres=# input_city VARCHAR(50) := 'River City';
postgres=# input_state VARCHAR(50) := 'Texas';
postgres=# input_country VARCHAR(50) := 'USA';
postgres=# input_zip_code VARCHAR(10) := '75001';
postgres=#
postgres=# -- Variables for data
postgres=# get_address_id UUID;
postgres=# get_existing_address RECORD;
postgres=#
postgres=# BEGIN
postgres=# -- Start the transaction
postgres=# BEGIN
postgres=# -- Insert data into the address table
postgres=# INSERT INTO address (flat_no, street, city, state, country, zip_code)
postgres=# VALUES (input_flat_no, input_street, input_city, input_state, input_country, input_zip_code)
postgres=# RETURNING id INTO get_address_id;
postgres=# RAISE NOTICE 'Inserted into address table for address_id: %', get_address_id;
postgres=#
postgres=# -- Insert data into the customer_address table
postgres=# INSERT INTO customer_address (customer_id, address_id, default_address)
postgres=# VALUES (input_customer_id, get_address_id, false);
postgres=# RAISE NOTICE 'Inserted into customer_address table for customer_id: %, address_id: %, input_customer_id, get_address_id';
postgres=#
postgres=# -- Check if any other address exists for the customer
postgres=# FOR get_existing_address IN
postgres=# SELECT addr.*
postgres=# FROM address addr
postgres=# JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
postgres=# WHERE cust_addr.customer_id = input_customer_id AND addr.id <> get_address_id
postgres=# LOOP
postgres=# -- Validate with the input
postgres=# IF get_existing_address.flat_no = input_flat_no AND
postgres=# get_existing_address.street = input_street AND
postgres=# get_existing_address.city = input_city AND
postgres=# get_existing_address.state = input_state AND
postgres=# get_existing_address.country = input_country AND
postgres=# get_existing_address.zip_code = input_zip_code THEN
postgres=# RAISE EXCEPTION 'Address already exists. Please make use of it.';
postgres=# END IF;
postgres=# END LOOP;
postgres=#
postgres=# EXCEPTION
postgres=# WHEN OTHERS THEN
postgres=# -- An error occurred, roll back the transaction
postgres=# RAISE NOTICE 'Error occurred: %', SQLERRM;
postgres=#
postgres=# -- Log the rollback
postgres=# RAISE NOTICE 'Error occurred, rolling back the data';
postgres=# ROLLBACK;
postgres=#
```

Fig-101: Stored the same address by the “668e5890-517a-4ae1-b909-03f39a3d8e6d”.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# BEGIN
postgres=# -- Start the transaction
postgres=# BEGIN
postgres=# -- Insert data into the address table
postgres=# INSERT INTO address (flat_no, street, city, state, country, zip_code)
postgres=# VALUES (input_flat_no, input_street, input_city, input_state, input_country, input_zip_code)
postgres=# RETURNING id INTO get_address_id;
postgres=# RAISE NOTICE 'Inserted into address table for address_id: %', get_address_id;
postgres=#
postgres=# -- Insert data into the customer_address table
postgres=# INSERT INTO customer_address (customer_id, address_id, default_address)
postgres=# VALUES (input_customer_id, get_address_id, false);
postgres=# RAISE NOTICE 'Inserted into customer_address table for customer_id: %, address_id: %, input_customer_id, get_address_id';
postgres=#
postgres=# -- Check if any other address exists for the customer
postgres=# FOR get_existing_address IN
postgres=# SELECT addr.*
postgres=# FROM address addr
postgres=# JOIN customer_address cust_addr ON addr.id = cust_addr.address_id
postgres=# WHERE cust_addr.customer_id = input_customer_id AND addr.id <> get_address_id
postgres=# LOOP
postgres=# -- Validate with the input
postgres=# IF get_existing_address.flat_no = input_flat_no AND
postgres=# get_existing_address.street = input_street AND
postgres=# get_existing_address.city = input_city AND
postgres=# get_existing_address.state = input_state AND
postgres=# get_existing_address.country = input_country AND
postgres=# get_existing_address.zip_code = input_zip_code THEN
postgres=# RAISE EXCEPTION 'Address already exists. Please make use of it.';
postgres=# END IF;
postgres=# END LOOP;
postgres=#
postgres=# EXCEPTION
postgres=# WHEN OTHERS THEN
postgres=# -- An error occurred, roll back the transaction
postgres=# RAISE NOTICE 'Error occurred: %', SQLERRM;
postgres=#
postgres=# -- Log the rollback
postgres=# RAISE NOTICE 'Error occurred, rolling back the data';
postgres=# ROLLBACK;
postgres=# RETURN;
postgres=#
postgres=# -- Everything went well, commit the transaction
postgres=# COMMIT;
postgres=# RAISE NOTICE 'Address saved successfully';
postgres=#
postgres=# END $$;
NOTICE: Inserted into address table for address_id: 4865c739-941b-4827-9e82-e5ef6a86a434
NOTICE: Inserted into customer_address table for customer_id: 668e5890-517a-4ae1-b909-03f39a3d8e6d, address_id: 4865c739-941b-4827-9e82-e5ef6a86a434
NOTICE: Error occurred: Address already exists. Please make use of it.
NOTICE: Error occurred, rolling back the data
postgres=#
```

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.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# select * from customer;
      id      | first_name | last_name | phone_no | email_id | dob | type
-----
648e5898-517a-4ae1-b989-83f39a3d8e0d | jack      | Teresa   | (918) 436-7890 | jack.teresa@example.com | 1990-01-01 | regular
(1 row)

postgres=# select * from customer_address;
      customer_id      | address_id | default_address
-----
648e5898-517a-4ae1-b989-83f39a3d8e0d | a855529a-1187-47cc-8986-d549fd357796 | t
648e5898-517a-4ae1-b989-83f39a3d8e0d | 2d75623c-8c97-4c4d-a1be-53eccc775e2e | f
648e5898-517a-4ae1-b989-83f39a3d8e0d | 381e8511-14e4-4525-b462-54abbf88fd1d | f
(3 rows)

postgres=# select * from address;
      id      | flat_no | street | city | state | country | zip_code
-----
2f9eb7f9-40e9-4883-9984-9208c6d7b7cc | 789 | Oak Street | San Francisco | California | United States | 94107
57de73a8-e753-4a58-8c15-fa8337d46746 | 101 | Highland Avenue | London | England | United Kingdom | SW1A 1AA
a855529a-1187-47cc-8986-d549fd357796 | 401 | Stella Street | Denton | Florida | USA | 65341
2d75623c-8c97-4c4d-a1be-53eccc775e2e | 782 | Melody Street | Houston | Texas | USA | 77001
381e8511-14e4-4525-b462-54abbf88fd1d | 101 | Maple lane, Apartment 3C | River City | Texas | USA | 79001
(5 rows)

postgres=#
```

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;
```

```
Terminal Shell Edit View Window Help
❏ -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# select * from customer;
 id | first_name | last_name | phone_no | email_id | dob | type
-----+-----+-----+-----+-----+-----+-----
 668e5890-517a-4ae1-b909-03f39a3d8e6d | Jack | Teresa | (918) 436-7890 | Jack.teresa@example.com | 1990-01-01 | regular
(1 row)

postgres=# select * from orders;
 id | status | order_date | address_id | customer_id
-----+-----+-----+-----+-----
 fff1b21a-1647-4b13-b92b-e05350967bd | Processing | 2024-02-23 | a055529e-1187-47cc-898d-d549fd35779e | 668e5890-517a-4ae1-b909-03f39a3d8e6d
 e758ab4b-817b-441d-e775-275bd5ffcc69 | Processing | 2024-02-23 | a055529e-1187-47cc-898d-d549fd35779e | 668e5890-517a-4ae1-b909-03f39a3d8e6d
 407968c5-4a91-46b8-8f68-969f88efcf45 | Processing | 2024-02-23 | a055529e-1187-47cc-898d-d549fd35779e | 668e5890-517a-4ae1-b909-03f39a3d8e6d
 67b7c9d3-4463-418a-bc98-d3d8ee126b26 | Processing | 2024-02-23 | a055529e-1187-47cc-898d-d549fd35779e | 668e5890-517a-4ae1-b909-03f39a3d8e6d
 eab37311-84c8-4d16-b02e-6779f80b99e | Processing | 2024-02-23 | a055529e-1187-47cc-898d-d549fd35779e | 668e5890-517a-4ae1-b909-03f39a3d8e6d
(5 rows)

postgres=# select * from transaction_summary;
 id | total_amount_paid | payment_type | date_of_payment | order_id
-----+-----+-----+-----+-----
 f214b43e-3937-42a8-80bd-8e211ee622f | 1598 | Credit Card | 2024-02-23 | fff1b21a-1647-4b13-b92b-e05350967bd
 a4d8c8e5-9408-448b-893c-7851fd8e87b2 | 1598 | Credit Card | 2024-02-23 | e758ab4b-817b-441d-e775-275bd5ffcc69
 843a1969-1a66-4de8-b8dc-7e6044aee8d1 | 3996 | Credit Card | 2024-02-23 | 407968c5-4a91-46b8-8f68-969f88efcf45
 b8e6d326-9357-425c-912f-628766c42c90 | 597 | Credit Card | 2024-02-23 | 67b7c9d3-4463-418a-bc98-d3d8ee126b26
 53a34cfe-4288-4d3a-bd65-60828bc38494 | 67788 | Credit Card | 2024-02-23 | eab37311-84c8-4d16-b02e-6779f80b99e
(5 rows)

postgres=#
```


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;
```



id	product_category_id	name	description	price	quantity	discount	brandname	address_id
e1618e85-79db-496f-a850-7f5a8adeFa21		Designer Jeans	Premium quality denim jeans for a fashionable look	89	50	5	FashionFits	2f9ab7f9-48e9-4883-9984-9288c6d7b7cc
f43140ef-5566-4a28-85a3-c5f55b276d6a		Best-Selling Novel	Compelling story by a renowned author	15	100	0	BookMaster	57de73a0-e753-4a68-8c15-fa8337d48746
899326d9-88a9-e85c-9862-7892d518e4de		Professional Running Shoes	Premium running shoes for athletes	129	75	10	RunElite	2f9ab7f9-48e9-4883-9984-9288c6d7b7cc
3fedcda8-1fd8-44b5-b8a5-1bd219886640		Educational Toy Set	Interactive toys for children to learn and play	49	50	5	SmartKids	57de73a0-e753-4a68-8c15-fa8337d48746
3dcdb7f6-a028-a96a-867a-c26c19acc3c		Camping Gear Bundle	Essential gear for outdoor adventures	299	10	25	AdventureGear	2f9ab7f9-48e9-4883-9984-9288c6d7b7cc
09124ea3-6a03-4d65-a387-d1f84448c2e		Smartphone	High-performance smartphone with advanced features	799	91	10	TechGadget	2f9ab7f9-48e9-4883-9984-9288c6d7b7cc
7adb9e77-802a-43ae-9da4-c49a94183124		Luxury Skincare Set	Complete skincare routine for beauty enthusiasts	199	17	20	GlowElegance	57de73a0-e753-4a68-8c15-fa8337d48746
413340a7-5a48-4ccc-a9c0-ca511392b49		Smart Refrigerator	Energy-efficient refrigerator with smart features	68000	27	15	CoolTech	57de73a0-e753-4a68-8c15-fa8337d48746
3ff7c56d-a7d8-a993-b0bd-97b9b6186572								
cf573629-ea72-4ed9-b78f-e096ac511c44								
651d0a08-d814-a5a7-a5aa-83cf6a874227								
d43ba097-25a4-4329-8c69-1c8e9eae8f348								
35784351-3a58-47ac-a5d0-3a6aa1d8dfc1								
e41bafdf-e071-44a8-a252-f088e41c3aa								
8893a2e3-e040-4427-9e60-b89d8e0b4482				50000				
98854b1e-2d16-4517-82f6-2bb432d99512								

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';

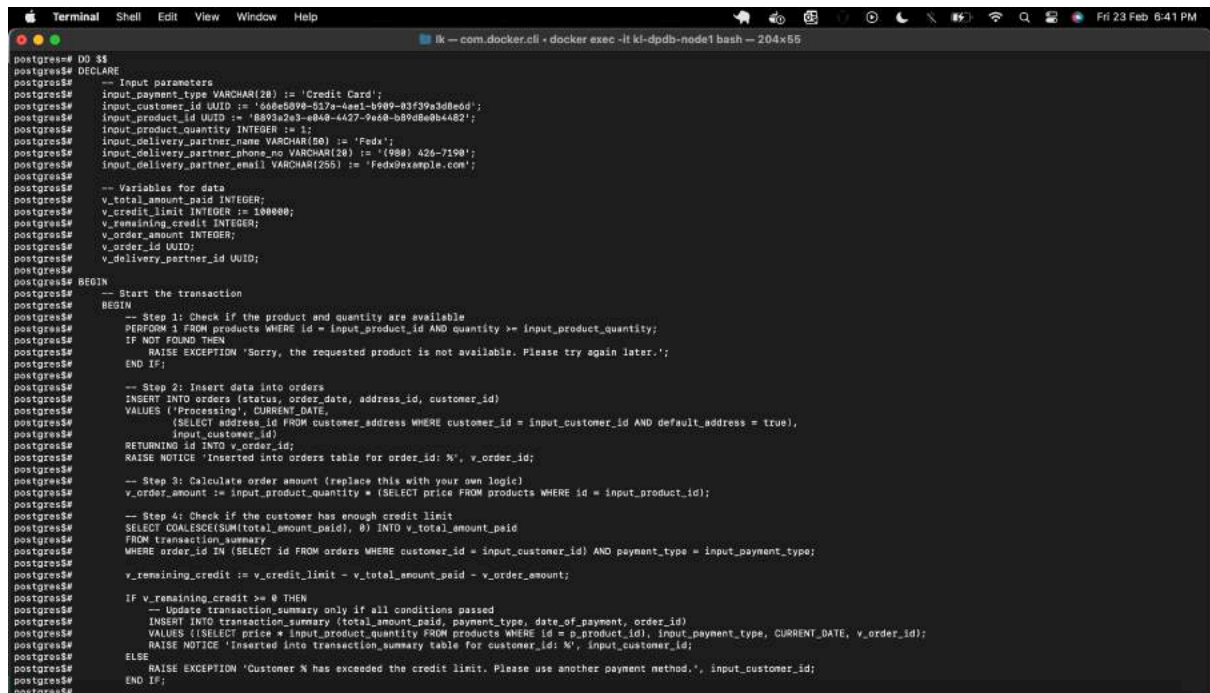
ROLLBACK;
RETURN;

END;

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

END $$;

```



```

Terminal Shell Edit View Window Help
-- com.docker.cli - docker exec -it ki-dpdb-node1 bash - 204x65

postgres=# DO $$
postgres# DECLARE
postgres# -- Input parameters
postgres# input_payment_type VARCHAR(28) := 'Credit Card';
postgres# input_customer_id UUID := '608e5098-017a-44e1-b989-83f9a3d8edd';
postgres# input_product_id UUID := '8892a2a3-e4a8-4427-9e68-b59e8e9b4482';
postgres# input_product_quantity INTEGER := 1;
postgres# input_delivery_partner_name VARCHAR(50) := 'Fedx';
postgres# input_delivery_partner_phone_no VARCHAR(20) := '(988) 456-7198';
postgres# input_delivery_partner_email VARCHAR(255) := 'Fedx@example.com';
postgres# -- Variables for data
postgres# v_total_amount_paid INTEGER;
postgres# v_credit_limit INTEGER := 100000;
postgres# v_remaining_credit INTEGER;
postgres# v_order_amount INTEGER;
postgres# v_order_id UUID;
postgres# v_delivery_partner_id UUID;
postgres# BEGIN
postgres# -- Start the transaction
postgres# BEGIN
postgres# -- Step 1: Check if the product and quantity are available
postgres# PERFORM 1 FROM products WHERE id = input_product_id AND quantity >= input_product_quantity;
postgres# IF NOT FOUND THEN
postgres# RAISE EXCEPTION 'Sorry, the requested product is not available. Please try again later.';
postgres# END IF;
postgres# -- Step 2: Insert data into orders
postgres# INSERT INTO orders (status, order_date, address_id, customer_id)
postgres# VALUES ('Processing', CURRENT_DATE,
postgres# (SELECT address_id FROM customer_address WHERE customer_id = input_customer_id AND default_address = true),
postgres# input_customer_id);
postgres# RETURNING id INTO v_order_id;
postgres# RAISE NOTICE 'Inserted into orders table for order_id: %', v_order_id;
postgres# -- Step 3: Calculate order amount (replace this with your own logic)
postgres# v_order_amount := input_product_quantity * (SELECT price FROM products WHERE id = input_product_id);
postgres# -- Step 4: Check if the customer has enough credit limit
postgres# SELECT COALESCE(SUM(total_amount_paid), 0) INTO v_total_amount_paid
postgres# FROM transaction_summary
postgres# WHERE order_id IN (SELECT id FROM orders WHERE customer_id = input_customer_id AND payment_type = input_payment_type);
postgres# v_remaining_credit := v_credit_limit - v_total_amount_paid - v_order_amount;
postgres# IF v_remaining_credit >= 0 THEN
postgres# -- Update transaction_summary only if all conditions passed
postgres# INSERT INTO transaction_summary (total_amount_paid, payment_type, date_of_payment, order_id)
postgres# VALUES ((SELECT price * input_product_quantity FROM products WHERE id = p_product_id), input_payment_type, CURRENT_DATE, v_order_id);
postgres# RAISE NOTICE 'Inserted into transaction_summary table for customer_id: %', input_customer_id;
postgres# ELSE
postgres# RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use another payment method.', input_customer_id;
postgres# END IF;
postgres# END;

```

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

```

Terminal Shell Edit View Window Help
ik ~ com.docker.cli - docker exec -it ki-dpdb-node1 bash - 204x56

postgres# WHERE order_id IN (SELECT id FROM orders WHERE customer_id = input_customer_id AND payment_type = input_payment_type);
postgres#
postgres# v_remaining_credit := v_credit_limit - v_total_amount_paid - v_order_amount;
postgres#
postgres# IF v_remaining_credit >= 0 THEN
postgres#     -- Update transaction_summary only if all conditions passed
postgres#     INSERT INTO transaction_summary (total_amount_paid, payment_type, date_of_payment, order_id)
postgres#     VALUES (SELECT price * input_product_quantity FROM products WHERE id = p_product_id, input_payment_type, CURRENT_DATE, v_order_id);
postgres#     RAISE NOTICE 'Inserted into transaction_summary table for customer_id: %', input_customer_id;
postgres# ELSE
postgres#     RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use another payment method.', input_customer_id;
postgres# END IF;
postgres#
postgres# -- Step 6: Insert data into delivery_partner
postgres# INSERT INTO delivery_partner (name, phone_no, email, order_id)
postgres# VALUES (input_delivery_partner_name, input_delivery_partner_phone_no, input_delivery_partner_email, v_order_id)
postgres# RETURNING id INTO v_delivery_partner_id;
postgres# RAISE NOTICE 'Inserted into delivery_partner table for delivery_partner_id: %', v_delivery_partner_id;
postgres#
postgres# -- Step 6: Insert data into customer_delivery_partner
postgres# INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
postgres# VALUES (input_customer_id, v_delivery_partner_id);
postgres# RAISE NOTICE 'Inserted into customer_delivery_partner table';
postgres#
postgres# -- Step 7: Insert data into orders_products
postgres# INSERT INTO orders_products (order_id, product_id, quantity)
postgres# VALUES (v_order_id, input_product_id, input_product_quantity);
postgres# RAISE NOTICE 'Inserted into orders_products table';
postgres#
postgres# -- Step 8: Update product quantity
postgres# UPDATE products SET quantity = quantity - input_product_quantity WHERE id = input_product_id;
postgres# RAISE NOTICE 'Updated product quantity for product_id: %', input_product_id;
postgres#
postgres# EXCEPTION
postgres# WHEN OTHERS THEN
postgres#     -- An error occurred, roll back the transaction
postgres#     RAISE NOTICE 'Error occurred: %', SQLERRM;
postgres#
postgres#     -- Log the rollback
postgres#     RAISE NOTICE 'Error occurred, rolling back the data';
postgres#
postgres#     ROLLBACK;
postgres#     RETURN;
postgres#
postgres# END;
postgres#
postgres# -- Everything went well, commit the transaction
postgres# COMMIT;
postgres# RAISE NOTICE 'Order placed successfully using credit card';
postgres#
postgres# END $$;
postgres#
postgres# Inserted into orders table for order_id: 20d3f5f7-fd81-4328-9248-7a26c0a1a4ee
NOTICE: Error occurred: Customer 668e8b90-517a-4ae1-0990-03f39a3d8e6d has exceeded the credit limit. Please use another payment method.
NOTICE: Error occurred, rolling back the data
no
postgres#

```

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:

D0 \$\$

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 $$;

```

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it kl-dpdb-node1 bash -- 204x55

postgres=# DO $$
postgres=# DECLARE
postgres=# -- Input parameters
postgres=# input_payment_type VARCHAR(28) := 'Debit Card';
postgres=# input_customer_id UUID := '668e6b98-517a-4aa1-b989-83f39a3d8edd';
postgres=# input_product_id UUID := '8893a2e3-e848-4427-9e68-b89d8e8b4482';
postgres=# input_product_quantity INTEGER := 1;
postgres=# input_delivery_partner_name VARCHAR(50) := 'Fedx';
postgres=# input_delivery_partner_phone_no VARCHAR(20) := '(989) 426-7198';
postgres=# input_delivery_partner_email VARCHAR(255) := 'Fedx@example.com';
postgres=#
postgres=# -- Variables for data
postgres=# v_total_amount_paid INTEGER;
postgres=# v_credit_limit INTEGER := 100000;
postgres=# v_remaining_credit INTEGER;
postgres=# v_order_amount INTEGER;
postgres=# v_order_id UUID;
postgres=# v_delivery_partner_id UUID;
postgres=#
postgres=# BEGIN
postgres=# -- Start the transaction
postgres=# BEGIN
postgres=# -- Step 1: Check if the product and quantity are available
postgres=# PERFORM 1 FROM products WHERE id = input_product_id AND quantity >= input_product_quantity;
postgres=# IF NOT FOUND THEN
postgres=# RAISE EXCEPTION 'Sorry, the requested product is not available. Please try again later.';
postgres=# END IF;
postgres=#
postgres=# -- Step 2: Insert data into orders
postgres=# INSERT INTO orders (status, order_date, address_id, customer_id)
postgres=# VALUES ('Processing', CURRENT_DATE,
postgres=# (SELECT address_id FROM customer_address WHERE customer_id = input_customer_id AND default_address = true),
postgres=# input_customer_id);
postgres=# RETURNING id INTO v_order_id;
postgres=# RAISE NOTICE 'Inserted into orders table for order_id: %', v_order_id;
postgres=#
postgres=# -- Step 3: Calculate order amount (replace this with your own logic)
postgres=# v_order_amount := input_product_quantity * (SELECT price FROM products WHERE id = input_product_id);
postgres=#
postgres=# -- Step 4: Check if the customer has enough credit limit
postgres=# SELECT COALESCE(SUM(total_amount_paid), 0) INTO v_total_amount_paid
postgres=# FROM transaction_summary
postgres=# WHERE order_id IN (SELECT id FROM orders WHERE customer_id = input_customer_id AND payment_type = input_payment_type);
postgres=# v_remaining_credit := v_credit_limit - v_total_amount_paid - v_order_amount;
postgres=#
postgres=# IF v_remaining_credit >= 0 THEN
postgres=# -- Update transaction_summary only if all conditions passed
postgres=# INSERT INTO transaction_summary (total_amount_paid, payment_type, date_of_payment, order_id)
postgres=# VALUES (SELECT price * input_product_quantity FROM products WHERE id = input_product_id), input_payment_type, CURRENT_DATE, v_order_id);
postgres=# RAISE NOTICE 'Inserted into transaction_summary table for customer_id: %', input_customer_id;
postgres=# ELSE
postgres=# RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use another payment method.', input_customer_id;
postgres=# END IF;
postgres=#
postgres=# -- Step 5: Insert data into delivery_partner
postgres=# INSERT INTO delivery_partner (name, phone_no, email, order_id)
postgres=# VALUES (input_delivery_partner_name, input_delivery_partner_phone_no, input_delivery_partner_email, v_order_id);
postgres=# RETURNING id INTO v_delivery_partner_id;
postgres=# RAISE NOTICE 'Inserted into delivery_partner table for delivery_partner_id: %', v_delivery_partner_id;
postgres=#
postgres=# -- Step 6: Insert data into customer_delivery_partner
postgres=# INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
postgres=# VALUES (input_customer_id, v_delivery_partner_id);
postgres=# RAISE NOTICE 'Inserted into customer_delivery_partner table';
postgres=#
postgres=# -- Step 7: Insert data into orders_products
postgres=# INSERT INTO orders_products (order_id, product_id, quantity)
postgres=# VALUES (v_order_id, input_product_id, input_product_quantity);
postgres=# RAISE NOTICE 'Inserted into orders_products table';
postgres=#
postgres=# -- Step 8: Update product quantity
postgres=# UPDATE products SET quantity = quantity - input_product_quantity WHERE id = input_product_id;
postgres=# RAISE NOTICE 'Updated product quantity for product_id: %', input_product_id;
postgres=#
postgres=# EXCEPTION
postgres=# WHEN OTHERS THEN
postgres=# -- An error occurred, roll back the transaction
postgres=# RAISE NOTICE 'Error occurred: %', SQLERRM;
postgres=# -- Log the rollback
postgres=# RAISE NOTICE 'Error occurred, rolling back the data';
postgres=# ROLLBACK;
postgres=# RETURN;
postgres=# END;
postgres=#
postgres=# -- Everything went well, commit the transaction
postgres=# COMMIT;
postgres=# RAISE NOTICE 'Order placed successfully';
postgres=# END $$;
NOTICE: Inserted into orders table for order_id: 1c5da731-712f-4c2c-8485-12459486a07d
NOTICE: Inserted into transaction_summary table for customer_id: 668e6b98-517a-4aa1-b989-83f39a3d8edd
NOTICE: Inserted into delivery_partner table for delivery_partner_id: 429ade91-b788-49de-8157-7159b9a28536
NOTICE: Inserted into customer_delivery_partner table
NOTICE: Inserted into orders_products table
NOTICE: Updated product quantity for product_id: 8893a2e3-e848-4427-9e68-b89d8e8b4482
NOTICE: Order placed successfully
DO
postgres=#
```

Fig-107: place an order using debit card.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it kl-dpdb-node1 bash -- 204x55

postgres=# IF v_remaining_credit >= 0 THEN
postgres=# -- Update transaction_summary only if all conditions passed
postgres=# INSERT INTO transaction_summary (total_amount_paid, payment_type, date_of_payment, order_id)
postgres=# VALUES (SELECT price * input_product_quantity FROM products WHERE id = input_product_id), input_payment_type, CURRENT_DATE, v_order_id);
postgres=# RAISE NOTICE 'Inserted into transaction_summary table for customer_id: %', input_customer_id;
postgres=# ELSE
postgres=# RAISE EXCEPTION 'Customer % has exceeded the credit limit. Please use another payment method.', input_customer_id;
postgres=# END IF;
postgres=#
postgres=# -- Step 5: Insert data into delivery_partner
postgres=# INSERT INTO delivery_partner (name, phone_no, email, order_id)
postgres=# VALUES (input_delivery_partner_name, input_delivery_partner_phone_no, input_delivery_partner_email, v_order_id);
postgres=# RETURNING id INTO v_delivery_partner_id;
postgres=# RAISE NOTICE 'Inserted into delivery_partner table for delivery_partner_id: %', v_delivery_partner_id;
postgres=#
postgres=# -- Step 6: Insert data into customer_delivery_partner
postgres=# INSERT INTO customer_delivery_partner (customer_id, delivery_partner_id)
postgres=# VALUES (input_customer_id, v_delivery_partner_id);
postgres=# RAISE NOTICE 'Inserted into customer_delivery_partner table';
postgres=#
postgres=# -- Step 7: Insert data into orders_products
postgres=# INSERT INTO orders_products (order_id, product_id, quantity)
postgres=# VALUES (v_order_id, input_product_id, input_product_quantity);
postgres=# RAISE NOTICE 'Inserted into orders_products table';
postgres=#
postgres=# -- Step 8: Update product quantity
postgres=# UPDATE products SET quantity = quantity - input_product_quantity WHERE id = input_product_id;
postgres=# RAISE NOTICE 'Updated product quantity for product_id: %', input_product_id;
postgres=#
postgres=# EXCEPTION
postgres=# WHEN OTHERS THEN
postgres=# -- An error occurred, roll back the transaction
postgres=# RAISE NOTICE 'Error occurred: %', SQLERRM;
postgres=# -- Log the rollback
postgres=# RAISE NOTICE 'Error occurred, rolling back the data';
postgres=# ROLLBACK;
postgres=# RETURN;
postgres=# END;
postgres=#
postgres=# -- Everything went well, commit the transaction
postgres=# COMMIT;
postgres=# RAISE NOTICE 'Order placed successfully';
postgres=# END $$;
NOTICE: Inserted into orders table for order_id: 1c5da731-712f-4c2c-8485-12459486a07d
NOTICE: Inserted into transaction_summary table for customer_id: 668e6b98-517a-4aa1-b989-83f39a3d8edd
NOTICE: Inserted into delivery_partner table for delivery_partner_id: 429ade91-b788-49de-8157-7159b9a28536
NOTICE: Inserted into customer_delivery_partner table
NOTICE: Inserted into orders_products table
NOTICE: Updated product quantity for product_id: 8893a2e3-e848-4427-9e68-b89d8e8b4482
NOTICE: Order placed successfully
DO
postgres=#
```

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.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# select * from customer;
      id      | first_name | last_name | phone_no | email_id | dob | type
-----+-----+-----+-----+-----+-----+-----
 668e5898-517a-4ae1-b989-83f39a3d8edd | Jack      | Teresa   | (918) 436-7890 | Jack.teresa@example.com | 1990-01-01 | regular
(1 row)

postgres=# select * from orders;
      id      | status | order_date | address_id | customer_id
-----+-----+-----+-----+-----
 fff1b21a-1647-4b13-b92b-a8535b95a7bd | Processing | 2024-02-23 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
 ef38eb4b-81fb-441d-a7f5-275bd6ffcc69 | Processing | 2024-02-23 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
 487968c5-4a91-46b8-bf68-98f88efcf45 | Processing | 2024-02-23 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
 67b7c9d3-44d3-418a-bc98-d3d8ee126a2e | Processing | 2024-02-23 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
 eab37311-84c8-4d16-b82e-6f79f89b998c | Processing | 2024-02-23 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
 1c5daf31-712f-4cdc-8485-1245948ea02d | Processing | 2024-02-24 | a855529e-1187-47cc-898d-d549fd35779e | 668e5898-517a-4ae1-b989-83f39a3d8edd
(6 rows)

postgres=# select * from transaction_summary;
      id      | total_amount_paid | payment_type | date_of_payment | order_id
-----+-----+-----+-----+-----
 f214b43e-3937-42a8-8b8d-86211eeea22f | 1598 | Credit Card | 2024-02-23 | fff1b21a-1647-4b13-b92b-a8535b95a7bd
 e48acbe5-84b8-4486-893c-7861fd8c87b2 | 1598 | Credit Card | 2024-02-23 | ef38eb4b-81fb-441d-a7f5-275bd6ffcc69
 5a3a19d9-1ee6-4de8-bedc-7e5d44aa8bd1 | 3995 | Credit Card | 2024-02-23 | 487968c5-4a91-46b8-bf68-98f88efcf45
 88aed32e-9337-425c-912f-42876aa42c9e | 597 | Credit Card | 2024-02-23 | 67b7c9d3-44d3-418a-bc98-d3d8ee126a2e
 53a345fe-42d8-4d3a-bd65-a8828bc3849a | 5788 | Credit Card | 2024-02-23 | eab37311-84c8-4d16-b82e-6f79f89b998c
 912d8abb-e48f-43f8-8db7-95c6eeaa6619a | 5888 | Debit Card | 2024-02-24 | 1c5daf31-712f-4cdc-8485-1245948ea02d
(6 rows)

postgres=#
```

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.

```
Terminal Shell Edit View Window Help
ik -- com.docker.cli - docker exec -it ki-dpdb-node1 bash -- 204x55

postgres=# select * from delivery_partner;
      id      | name | phone_no | email | order_id
-----+-----+-----+-----+-----
 429ade01-b780-49de-8157-7199b9a25535 | FedEx | (988) 424-7198 | FedEx@example.com | 1c5daf31-712f-4cdc-8485-1245948ea02d
(1 row)

postgres=# select * from customer_delivery_partner;
      customer_id | delivery_partner_id
-----+-----
 668e5898-517a-4ae1-b989-83f39a3d8edd | 429ade01-b780-49de-8157-7199b9a25535
(1 row)

postgres=# select * from orders_products;
      order_id | product_id | quantity
-----+-----+-----
 1c5daf31-712f-4cdc-8485-1245948ea02d | 8893a2a3-a840-4427-9a68-b89d8a0a482 | 1
(1 row)

postgres=#
```

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.