



Bharatiya Vidya Bhavan's
SARDAR PATEL INSTITUTE OF TECHNOLOGY
(Autonomous Institute Affiliated to Mumbai University)
Munshi Nagar, Andheri (West), Mumbai-400058
Information Technology Department

Academic Year: 2022-2023 Class: TE Sem.: VI
Course: IT331-ADBMS

Name	Prathamesh Rajan Pawar
UID	2020400040
Course	Advanced Database management system
Lab	2

Aim :

Design a distributed database by applying the concept of horizontal fragmentation

Scenario:

A retail chain database where customer info, customer orders and store inventory is stored. Storing all of this data in a centralized database could lead to slow queries, increased network latency and reduced availability in case of server failure.

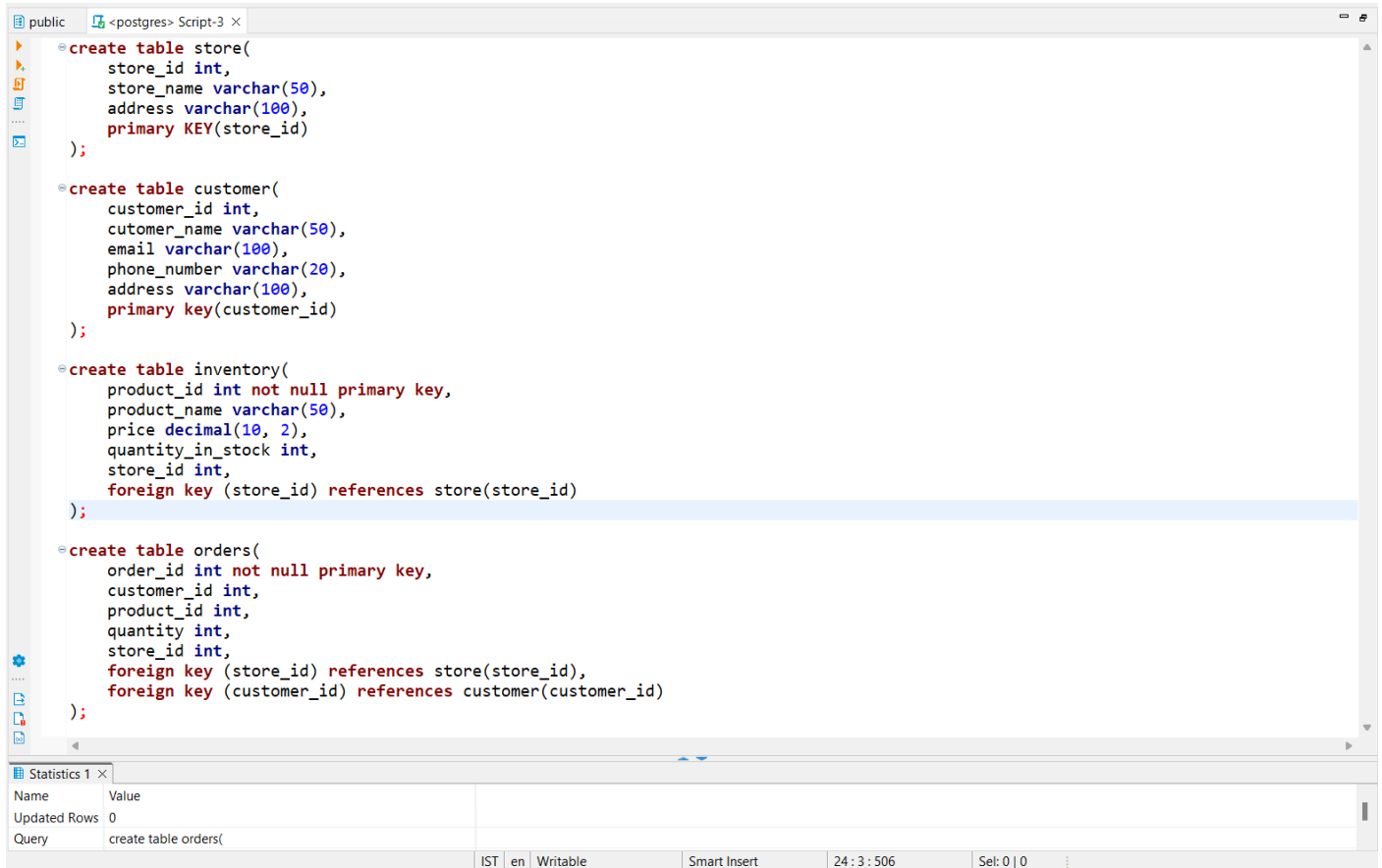
Therefore, we can fragment tables according to our scenario.

We can fragment the inventory and orders table by the store location. This means that each fragment will contain only the orders for a specific store.

This ensures that each store has access only to their inventory and orders data, which improves data security and reduces risk of errors.

Queries:

1. Creation of table



```
create table store(  
  store_id int,  
  store_name varchar(50),  
  address varchar(100),  
  primary KEY(store_id)  
);  
  
create table customer(  
  customer_id int,  
  customer_name varchar(50),  
  email varchar(100),  
  phone_number varchar(20),  
  address varchar(100),  
  primary key(customer_id)  
);  
  
create table inventory(  
  product_id int not null primary key,  
  product_name varchar(50),  
  price decimal(10, 2),  
  quantity_in_stock int,  
  store_id int,  
  foreign key (store_id) references store(store_id)  
);  
  
create table orders(  
  order_id int not null primary key,  
  customer_id int,  
  product_id int,  
  quantity int,  
  store_id int,  
  foreign key (store_id) references store(store_id),  
  foreign key (customer_id) references customer(customer_id)  
);
```

Name	Value
Updated Rows	0
Query	create table orders(

IST en Writable Smart Insert 24 : 3 : 506 Sel: 0 | 0

2. Inserting tuples into tables

The screenshot shows a database client interface with a script editor and a statistics panel. The script editor contains three SQL INSERT statements. The statistics panel shows the execution of the first statement, which inserted 10 rows into the customer table.

```
-- Insertion

INSERT INTO store (store_id, store_name, address) VALUES
(1, 'Store A', '123 Main St.'),
(2, 'Store B', '456 Elm St.');
```

```
INSERT INTO customer (customer_id, customer_name, email, phone_number, address) VALUES
(1, 'John Doe', 'johndoe@example.com', '555-1234', '123 Main St.'),
(2, 'Jane Smith', 'janesmith@example.com', '555-5678', '456 Elm St.'),
(3, 'Bob Johnson', 'bobjohnson@example.com', '555-9101', '123 Main St.'),
(4, 'Sara Lee', 'saralee@example.com', '555-1212', '456 Elm St.'),
(5, 'Mike Brown', 'mikebrown@example.com', '555-3434', '123 Main St.'),
(6, 'Emily Davis', 'emilydavis@example.com', '555-5656', '456 Elm St.'),
(7, 'Tom Wilson', 'tomwilson@example.com', '555-7878', '123 Main St.'),
(8, 'Jenny Garcia', 'jennygarcia@example.com', '555-9090', '456 Elm St.'),
(9, 'David Kim', 'davidkim@example.com', '555-2323', '123 Main St.'),
(10, 'Karen Lee', 'karenlee@example.com', '555-4545', '456 Elm St.');
```

```
INSERT INTO inventory (product_id, product_name, price, quantity_in_stock, store_id) VALUES
(1, 'Product A', 10.99, 100, 1),
(2, 'Product B', 20.99, 50, 2),
(3, 'Product C', 5.99, 200, 1),
(4, 'Product D', 15.99, 75, 2),
(5, 'Product E', 8.99, 150, 1),
(6, 'Product F', 25.99, 25, 2),
(7, 'Product G', 12.99, 125, 1),
(8, 'Product H', 30.99, 10, 2),
(9, 'Product I', 7.99, 175, 1),
(10, 'Product J', 18.99, 60, 2);
```

Name	Value
Updated Rows	10
Query	INSERT INTO customer (customer_id, customer_name, email, phone_number, address) VALUES (1, 'John Doe', 'johndoe@example.com', '555-1234', '123 Main St.'), (2, 'Jane Smith', 'janesmith@example.com', '555-5678', '456 Elm St.'), (3, 'Bob Johnson', 'bobjohnson@example.com', '555-9101', '123 Main St.'), (4, 'Sara Lee', 'saralee@example.com', '555-1212', '456 Elm St.');

IST | en | Writable Smart Insert 36 : 13 : 771 Sel: 0 | 0


The screenshot shows a database client interface with a script editor and a statistics panel. The script editor contains an SQL INSERT statement for the orders table. The statistics panel shows the execution of the first statement, which inserted 10 rows into the orders table.

```
INSERT INTO orders (order_id, customer_id, product_id, quantity, store_id) VALUES
(1, 1, 1, 2, 1),
(2, 2, 3, 1, 1),
(3, 3, 5, 4, 1),
(4, 4, 2, 3, 2),
(5, 5, 4, 1, 2),
(6, 6, 8, 2, 2),
(7, 7, 10, 3, 2),
(8, 8, 7, 1, 1),
(9, 9, 9, 2, 1),
(10, 10, 6, 1, 2);
```












Name	Value
Updated Rows	10
Query	INSERT INTO orders (order_id, customer_id, product_id, quantity, store_id) VALUES (1, 1, 1, 2, 1), (2, 2, 3, 1, 1), (3, 3, 5, 4, 1), (4, 4, 2, 3, 2),

Tables:







Store table:

store  Enter a SQL expression to filter results (use Ctrl+Space)				
Grid		123 store_id ▼	ABC store_name ▼	ABC address ▼
	1	1	Store A	123 Main St.
Text	2	2	Store B	456 Elm St.



























Inventory table:

inventory  Enter a SQL expression to filter results (use Ctrl+Space)						
Grid		123 product_id ▼	ABC product_name ▼	123 price ▼	123 quantity_in_stock ▼	123 store_id ▼
	1	1	Product A	10.99	100	1 
Text	2	2	Product B	20.99	50	2 
	3	3	Product C	5.99	200	1 
	4	4	Product D	15.99	75	2 
	5	5	Product E	8.99	150	1 
	6	6	Product F	25.99	25	2 
	7	7	Product G	12.99	125	1 
	8	8	Product H	30.99	10	2 
	9	9	Product I	7.99	175	1 
	10	10	Product J	18.99	60	2 

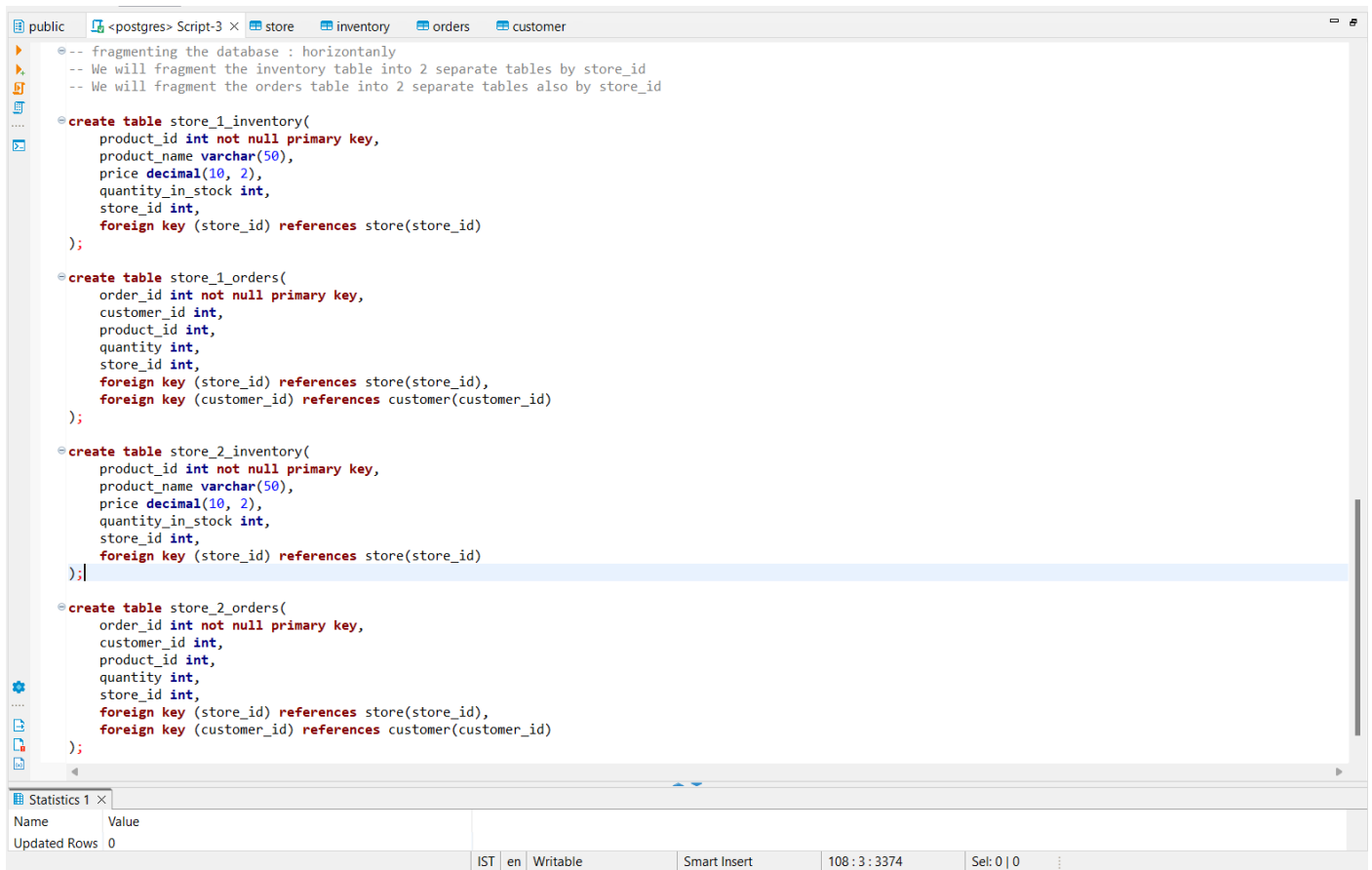
Customer table:

customer  Enter a SQL expression to filter results (use Ctrl+Space)						
	 customer_id	 customer_name	 email	 phone_number	 address	
1	1	John Doe	johndoe@example.com	555-1234	123 Main St.	
2	2	Jane Smith	janesmith@example.com	555-5678	456 Elm St.	
3	3	Bob Johnson	bobjohnson@example.com	555-9101	123 Main St.	
4	4	Sara Lee	saralee@example.com	555-1212	456 Elm St.	
5	5	Mike Brown	mikebrown@example.com	555-3434	123 Main St.	
6	6	Emily Davis	emilydavis@example.com	555-5656	456 Elm St.	
7	7	Tom Wilson	tomwilson@example.com	555-7878	123 Main St.	
8	8	Jenny Garcia	jennygarcia@example.com	555-9090	456 Elm St.	
9	9	David Kim	davidkim@example.com	555-2323	123 Main St.	
10	10	Karen Lee	karenlee@example.com	555-4545	456 Elm St.	

Orders table:

orders  Enter a SQL expression to filter results (use Ctrl+Space)						
	 order_id	 customer_id	 product_id	 quantity	 store_id	
1	1	1 	1	2	1 	
2	2	2 	3	1	1 	
3	3	3 	5	4	1 	
4	4	4 	2	3	2 	
5	5	5 	4	1	2 	
6	6	6 	8	2	2 	
7	7	7 	10	3	2 	
8	8	8 	7	1	1 	
9	9	9 	9	2	1 	
10	10	10 	6	1	2 	

Fragmenting the database by stores:



The screenshot shows a PostgreSQL script editor with a tab titled "Script-3". The script contains SQL code to create four tables, fragmenting the database horizontally by store_id. The tables are store_1_inventory, store_1_orders, store_2_inventory, and store_2_orders. Each table has a primary key on product_id and a foreign key on store_id referencing the store table. The store table is not shown in the script but is referenced in the foreign keys.

```
-- fragmenting the database : horizonatanly
-- We will fragment the inventory table into 2 separate tables by store_id
-- We will fragment the orders table into 2 separate tables also by store_id

create table store_1_inventory(
    product_id int not null primary key,
    product_name varchar(50),
    price decimal(10, 2),
    quantity_in_stock int,
    store_id int,
    foreign key (store_id) references store(store_id)
);

create table store_1_orders(
    order_id int not null primary key,
    customer_id int,
    product_id int,
    quantity int,
    store_id int,
    foreign key (store_id) references store(store_id),
    foreign key (customer_id) references customer(customer_id)
);

create table store_2_inventory(
    product_id int not null primary key,
    product_name varchar(50),
    price decimal(10, 2),
    quantity_in_stock int,
    store_id int,
    foreign key (store_id) references store(store_id)
);

create table store_2_orders(
    order_id int not null primary key,
    customer_id int,
    product_id int,
    quantity int,
    store_id int,
    foreign key (store_id) references store(store_id),
    foreign key (customer_id) references customer(customer_id)
);
```

At the bottom of the window, there is a "Statistics 1" panel. It shows a table with two columns: "Name" and "Value". The "Updated Rows" is 0. The table is in "IST" mode, "en" locale, and "Writable" mode. The "Smart Insert" option is checked. The table has 108 rows, 3 columns, and 3374 bytes. The selection is 0 rows and 0 columns.

Name	Value
Updated Rows	0

IST en Writable Smart Insert 108:3:3374 Sel: 0 | 0

public <postgres> Script-3 x store inventory orders customer

```
-- fragmenting database
insert into store_1_orders(select * from orders where store_id = 1)
insert into store_2_orders(select * from orders where store_id = 2)
insert into store_1_inventory(select * from inventory where store_id = 1)
insert into store_2_inventory(select * from inventory where store_id = 2)

select * from store_1_orders;
select * from store_2_orders;
select * from store_1_inventory;
select * from store_2_inventory;
```

store_1_orders 1 x

select * from store_1_orders

	order_id	customer_id	product_id	quantity	store_id
1	1	1	1	2	1
2	2	2	3	1	1
3	3	3	5	4	1
4	8	8	7	1	1
5	9	9	9	2	1

Value x 2

Refresh Save Cancel Export data 200 5 5 row(s) fetched - 2ms, on 2023-02-13 at 21:01:00

IST en Writable Smart Insert 127 : 1 [29] Sel: 29 | 1

public <postgres> Script-3 x store inventory orders customer

```
-- fragmenting database
insert into store_1_orders(select * from orders where store_id = 1)
insert into store_2_orders(select * from orders where store_id = 2)
insert into store_1_inventory(select * from inventory where store_id = 1)
insert into store_2_inventory(select * from inventory where store_id = 2)

select * from store_1_orders;
select * from store_2_orders;
select * from store_1_inventory;
select * from store_2_inventory;
```

store_2_orders 1 x

select * from store_2_orders

	order_id	customer_id	product_id	quantity	store_id
1	4	4	2	3	2
2	5	5	4	1	2
3	6	6	8	2	2
4	7	7	10	3	2
5	10	10	6	1	2

Value x 4

Refresh Save Cancel Export data 200 5 5 row(s) fetched - 4ms, on 2023-02-13 at 21:02:12

IST en Writable Smart Insert 128 : 1 [29] Sel: 29 | 1

public <postgres> Script-3 x store inventory orders customer

```
-- fragmenting database

insert into store_1_orders(select * from orders where store_id = 1)
insert into store_2_orders(select * from orders where store_id = 2)
insert into store_1_inventory(select * from inventory where store_id = 1)
insert into store_2_inventory(select * from inventory where store_id = 2)

select * from store_1_orders;
select * from store_2_orders;
select * from store_1_inventory;
select * from store_2_inventory;
```

store_1_inventory 1 x

select * from store_1_inventory

	product_id	product_name	price	quantity_in_stock	store_id
1	1	Product A	10.99	100	1
2	3	Product C	5.99	200	1
3	5	Product E	8.99	150	1
4	7	Product G	12.99	125	1
5	9	Product I	7.99	175	1

Refresh Save Cancel Export data 200 5 5 row(s) fetched - 8ms, on 2023-02-13 at 21:02:48

IST en Writable Smart Insert 129: 1 [32] Sel: 32 | 1

public <postgres> Script-3 x store inventory orders customer

```
-- fragmenting database

insert into store_1_orders(select * from orders where store_id = 1)
insert into store_2_orders(select * from orders where store_id = 2)
insert into store_1_inventory(select * from inventory where store_id = 1)
insert into store_2_inventory(select * from inventory where store_id = 2)

select * from store_1_orders;
select * from store_2_orders;
select * from store_1_inventory;
select * from store_2_inventory;
```

store_1_inventory 1 x

select * from store_1_inventory

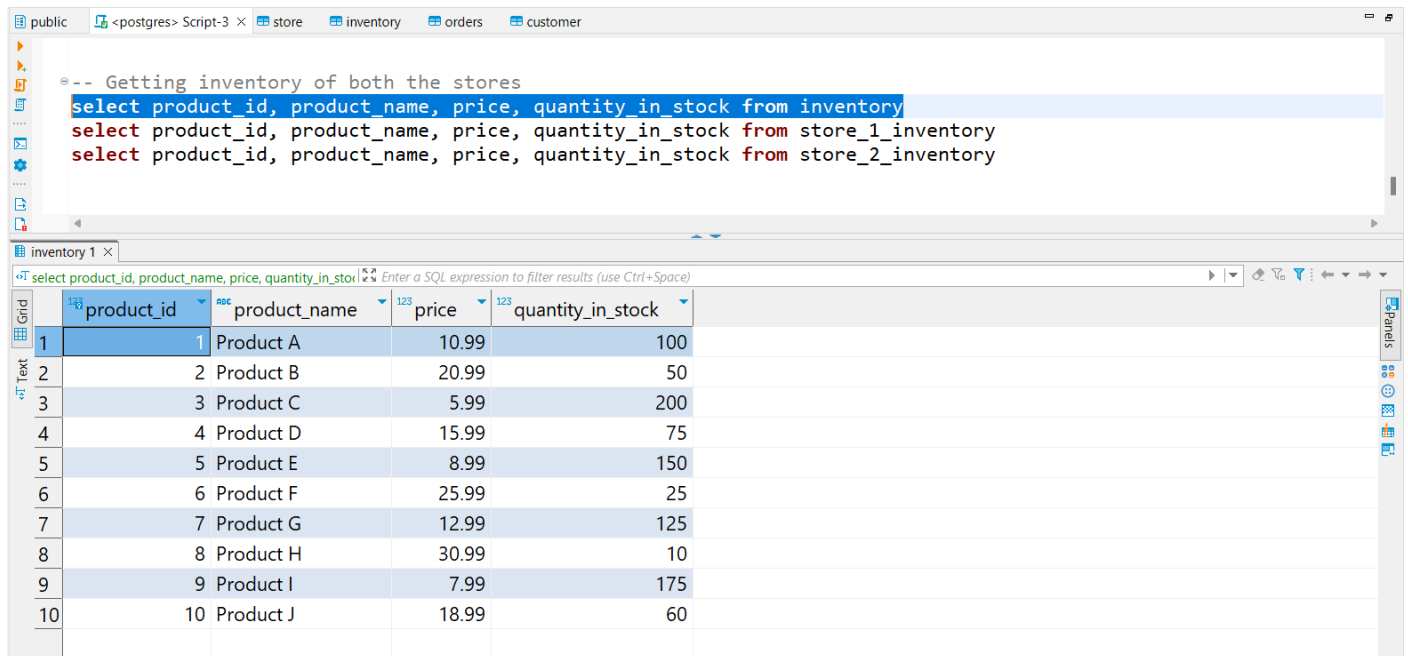
	product_id	product_name	price	quantity_in_stock	store_id
1	1	Product A	10.99	100	1
2	3	Product C	5.99	200	1
3	5	Product E	8.99	150	1
4	7	Product G	12.99	125	1
5	9	Product I	7.99	175	1

Refresh Save Cancel Export data 200 5 5 row(s) fetched - 8ms, on 2023-02-13 at 21:02:48

IST en Writable Smart Insert 130: 1 [32] Sel: 32 | 1

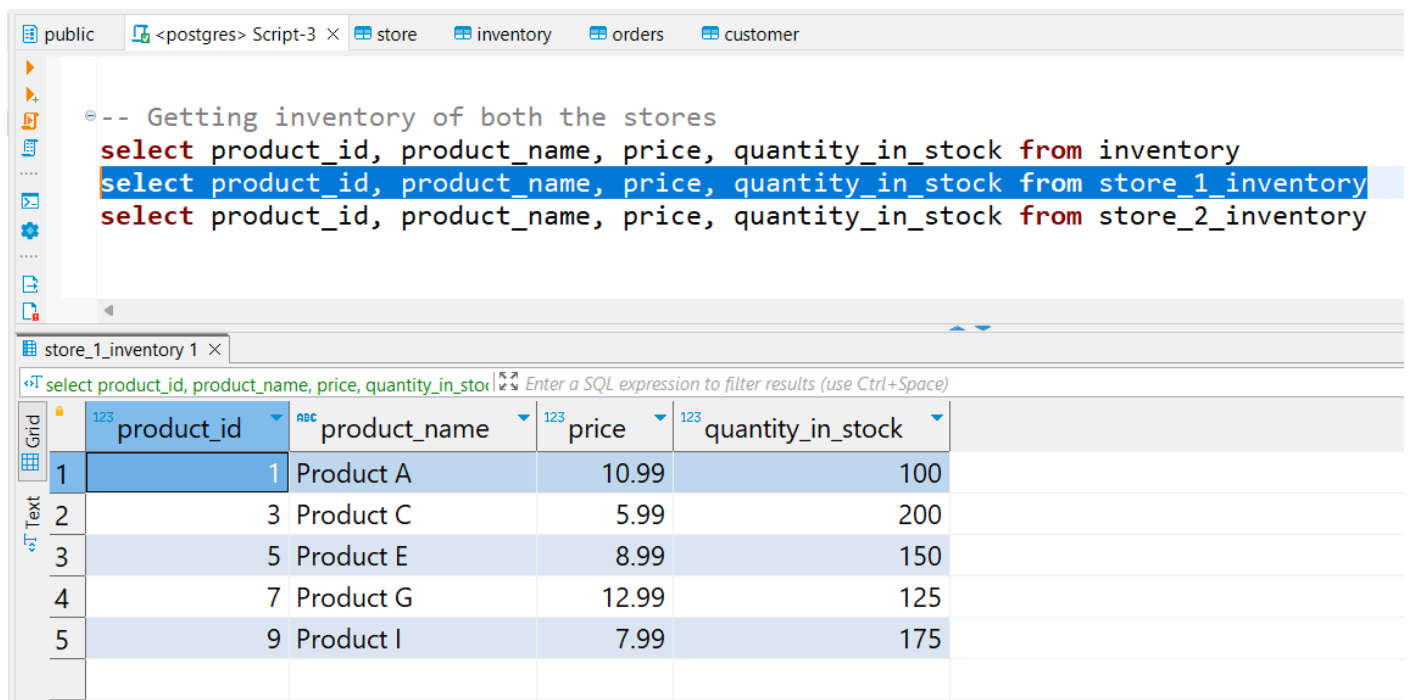
Running Queries in the fragmented tables

1) Here in the below query I am getting the info of inventory of the both stores



```
-- Getting inventory of both the stores
select product_id, product_name, price, quantity_in_stock from inventory
select product_id, product_name, price, quantity_in_stock from store_1_inventory
select product_id, product_name, price, quantity_in_stock from store_2_inventory
```

	product_id	product_name	price	quantity_in_stock
1	1	Product A	10.99	100
2	2	Product B	20.99	50
3	3	Product C	5.99	200
4	4	Product D	15.99	75
5	5	Product E	8.99	150
6	6	Product F	25.99	25
7	7	Product G	12.99	125
8	8	Product H	30.99	10
9	9	Product I	7.99	175
10	10	Product J	18.99	60



```
-- Getting inventory of both the stores
select product_id, product_name, price, quantity_in_stock from inventory
select product_id, product_name, price, quantity_in_stock from store_1_inventory
select product_id, product_name, price, quantity_in_stock from store_2_inventory
```

	product_id	product_name	price	quantity_in_stock
1	1	Product A	10.99	100
2	3	Product C	5.99	200
3	5	Product E	8.99	150
4	7	Product G	12.99	125
5	9	Product I	7.99	175

public <postgres> Script-3 × store inventory orders customer

```
-- Getting inventory of both the stores
select product_id, product_name, price, quantity_in_stock from inventory
select product_id, product_name, price, quantity_in_stock from store_1_inventory
select product_id, product_name, price, quantity_in_stock from store_2_inventory
```

store_2_inventory 1 ×

select product_id, product_name, price, quantity_in_stock | Enter a SQL expression to filter results (use Ctrl+Space)

	product_id	product_name	price	quantity_in_stock
1	2	Product B	20.99	50
2	4	Product D	15.99	75
3	6	Product F	25.99	25
4	8	Product H	30.99	10
5	10	Product J	18.99	60

2) Here in the below query I am getting the orders from both the stores

public <postgres> Script-3 × store inventory orders customer

```
-- Getting orders from both the stores
select order_id, customer_id, product_id, quantity from orders
select order_id, customer_id, product_id, quantity from store_1_orders
select order_id, customer_id, product_id, quantity from store_2_orders
```

orders 1 ×

select order_id, customer_id, product_id, quantity from | Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	product_id	quantity
1	1	1	1	2
2	2	2	3	1
3	3	3	5	4
4	4	4	2	3
5	5	5	4	1
6	6	6	8	2
7	7	7	10	3
8	8	8	7	1
9	9	9	9	2
10	10	10	6	1

public <postgres> Script-3 x store inventory orders customer

```
-- Getting orders from both the stores
select order_id, customer_id, product_id, quantity from orders
select order_id, customer_id, product_id, quantity from store_1_orders
select order_id, customer_id, product_id, quantity from store_2_orders
```

store_1_orders 1 x

select order_id, customer_id, product_id, quantity from Enter a SQL expression to filter results (use Ctrl+Space)

	order_id	customer_id	product_id	quantity
1	1	1	1	2
2	2	2	3	1
3	3	3	5	4
4	8	8	7	1
5	9	9	9	2

public <postgres> Script-3 x store inventory orders customer

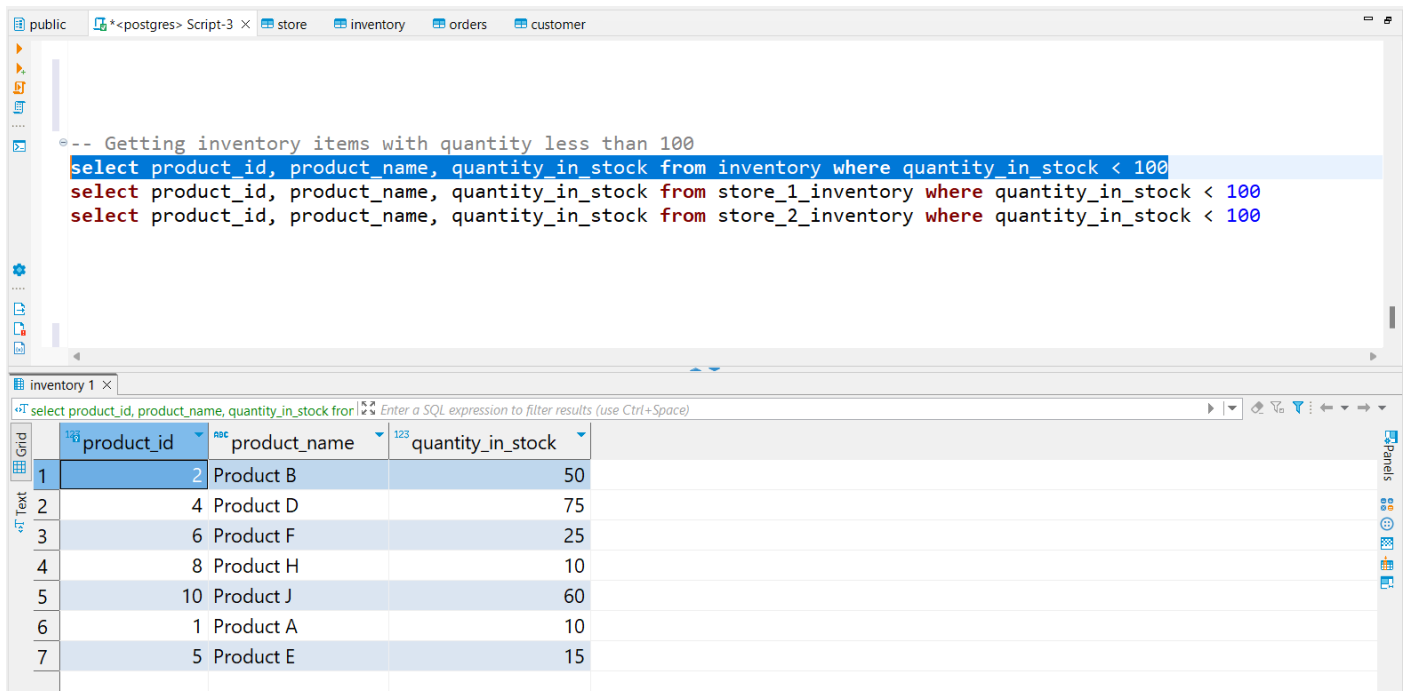
```
-- Getting orders from both the stores
select order_id, customer_id, product_id, quantity from orders
select order_id, customer_id, product_id, quantity from store_1_orders
select order_id, customer_id, product_id, quantity from store_2_orders
```

store_2_orders 1 x

select order_id, customer_id, product_id, quantity from Enter a SQL expression to filter results (use Ctrl+Space)

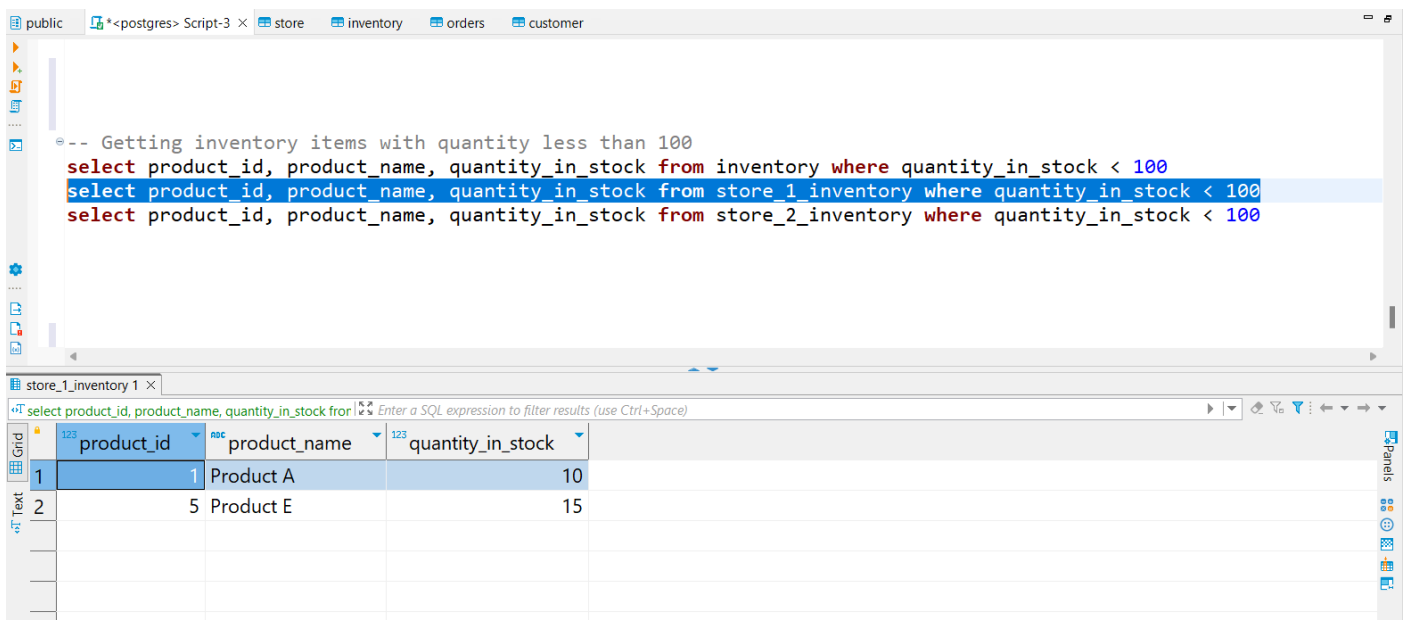
	order_id	customer_id	product_id	quantity
1	4	4	2	3
2	5	5	4	1
3	6	6	8	2
4	7	7	10	3
5	10	10	6	1

3) We are querying inventory items with quantity less than 100



```
-- Getting inventory items with quantity less than 100
select product_id, product_name, quantity_in_stock from inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_1_inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_2_inventory where quantity_in_stock < 100
```

	product_id	product_name	quantity_in_stock
1	2	Product B	50
2	4	Product D	75
3	6	Product F	25
4	8	Product H	10
5	10	Product J	60
6	1	Product A	10
7	5	Product E	15



```
-- Getting inventory items with quantity less than 100
select product_id, product_name, quantity_in_stock from inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_1_inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_2_inventory where quantity_in_stock < 100
```

	product_id	product_name	quantity_in_stock
1	1	Product A	10
2	5	Product E	15

public | <postgres> Script-3 | store | inventory | orders | customer

```
-- Getting inventory items with quantity less than 100
select product_id, product_name, quantity_in_stock from inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_1_inventory where quantity_in_stock < 100
select product_id, product_name, quantity_in_stock from store_2_inventory where quantity_in_stock < 100
```

store_2_inventory 1 x

select product_id, product_name, quantity_in_stock from store_2_inventory

	product_id	product_name	quantity_in_stock
1	2	Product B	50
2	4	Product D	75
3	6	Product F	25
4	8	Product H	10
5	10	Product J	60

4) Now, we are querying the inventory items with price greater than \$10

public | <postgres> Script-3 | store | inventory | orders | customer

```
-- Getting inventory items with price greater than 10
select product_id, product_name, price from inventory where price > 10
select product_id, product_name, price from store_1_inventory where price > 10
select product_id, product_name, price from store_2_inventory where price > 10
```

inventory 1 x

select product_id, product_name, price from inventory

	product_id	product_name	price
1	2	Product B	20.99
2	4	Product D	15.99
3	6	Product F	25.99
4	7	Product G	12.99
5	8	Product H	30.99
6	10	Product J	18.99
7	1	Product A	10.99

Refresh | Save | Cancel | Export data | 200 | 7 | 7 row(s) fetched - 3ms, on 2023-02-13 at 22:46:11 | IST | en | Writable | Smart Insert | 154 : 1 [70] | Sel: 70 | 1

public <postgres> Script-3 × store inventory orders customer

```
-- Getting inventory items with price greater than 10
select product_id, product_name, price from inventory where price > 10
select product_id, product_name, price from store_1_inventory where price > 10
select product_id, product_name, price from store_2_inventory where price > 10
```

store_1_inventory 1 ×

select product_id, product_name, price from store_1_in Enter a SQL expression to filter results (use Ctrl+Space)

	product_id	product_name	price
1	7	Product G	12.99
2	1	Product A	10.99

Refresh Save Cancel Export data 200 2 2 row(s) fetched - 1ms, on 2023-02-13 at 22:46:27

IST en Writable Smart Insert 155 : 1 [78] Sel: 78 | 1

public <postgres> Script-3 × store inventory orders customer

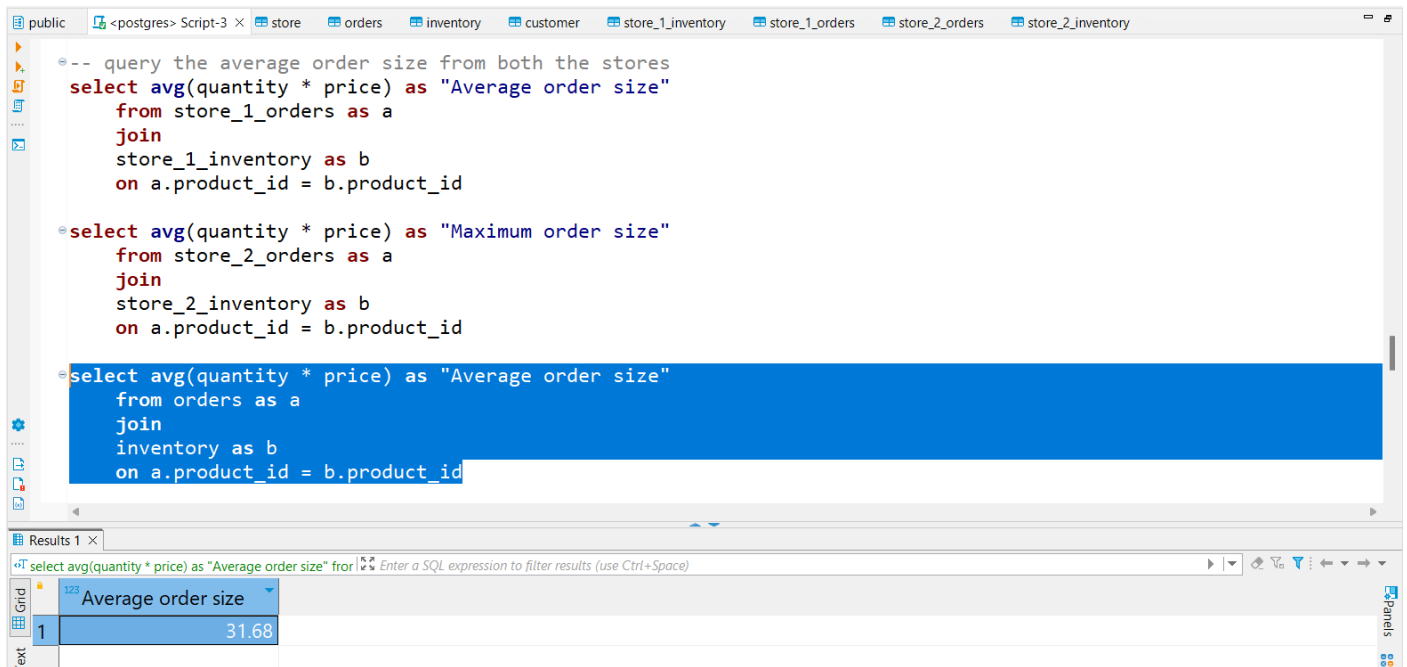
```
-- Getting inventory items with price greater than 10
select product_id, product_name, price from inventory where price > 10
select product_id, product_name, price from store_1_inventory where price > 10
select product_id, product_name, price from store_2_inventory where price > 10
```

store_2_inventory 1 ×

select product_id, product_name, price from store_2_in Enter a SQL expression to filter results (use Ctrl+Space)

	product_id	product_name	price
1	2	Product B	20.99
2	4	Product D	15.99
3	6	Product F	25.99
4	8	Product H	30.99
5	10	Product J	18.99

5) Average order size at each store, it can be used to calculate profits of the store

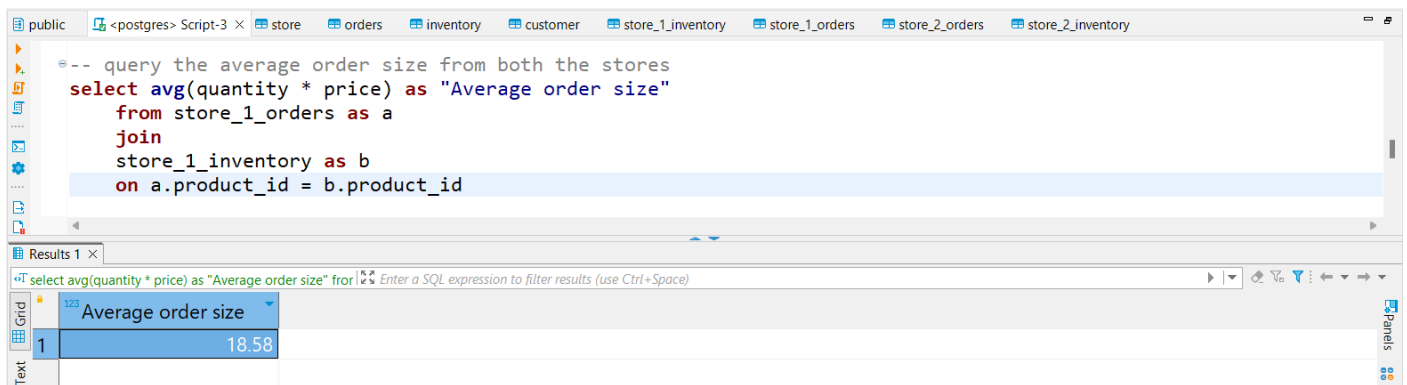


The screenshot shows a PostgreSQL script editor with the following SQL query:

```
-- query the average order size from both the stores
select avg(quantity * price) as "Average order size"
  from store_1_orders as a
    join
      store_1_inventory as b
    on a.product_id = b.product_id
```

The Results panel shows the following table:

	Average order size
1	31.68

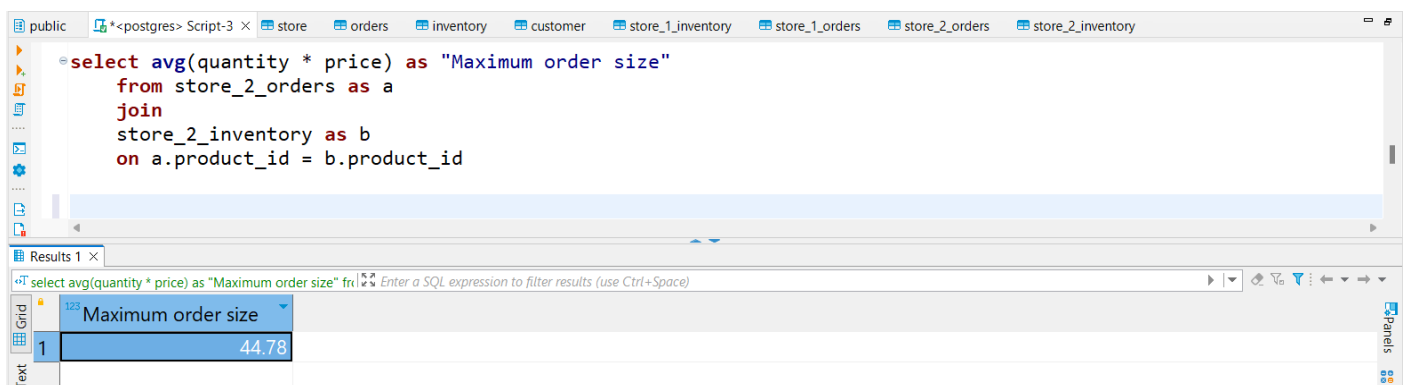


The screenshot shows a PostgreSQL script editor with the following SQL query:

```
-- query the average order size from both the stores
select avg(quantity * price) as "Average order size"
  from store_1_orders as a
    join
      store_1_inventory as b
    on a.product_id = b.product_id
```

The Results panel shows the following table:

	Average order size
1	18.58



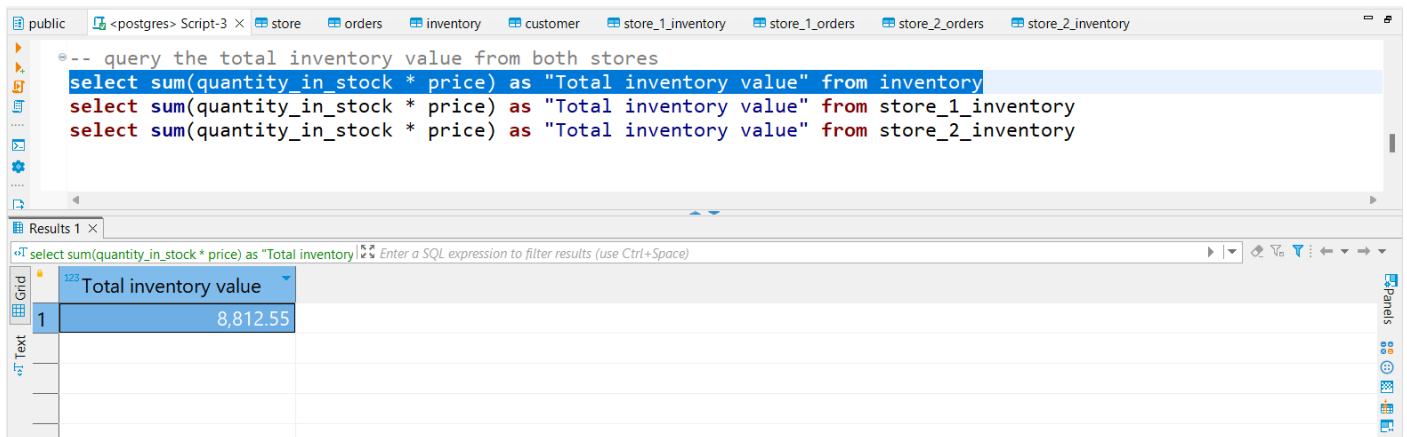
The screenshot shows a PostgreSQL script editor with the following SQL query:

```
select avg(quantity * price) as "Maximum order size"
  from store_2_orders as a
    join
      store_2_inventory as b
    on a.product_id = b.product_id
```

The Results panel shows the following table:

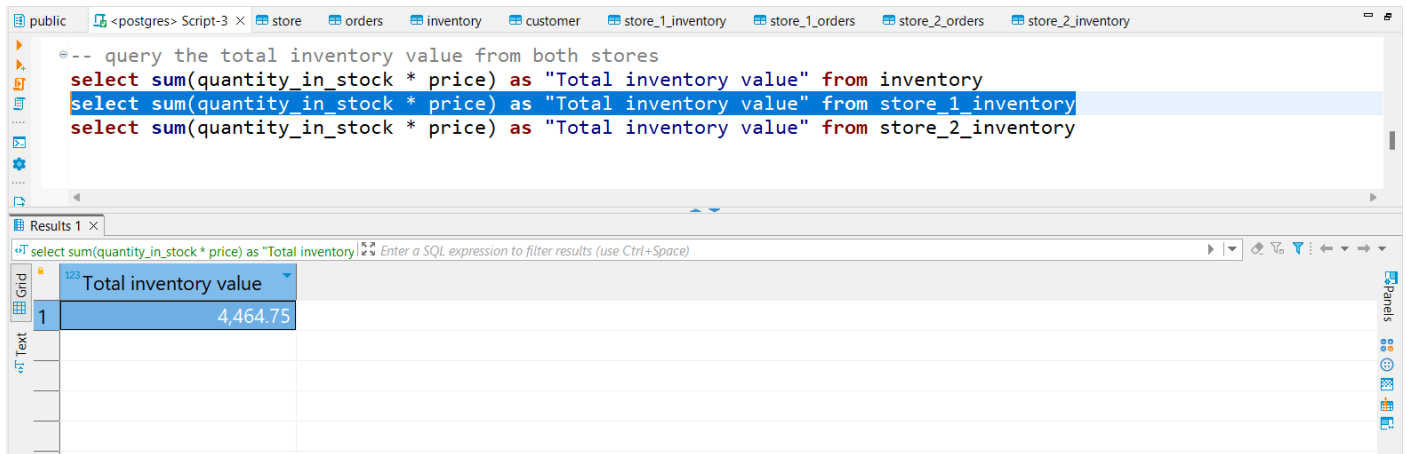
	Maximum order size
1	44.78

6) Query to Find the Total inventory value of the stores



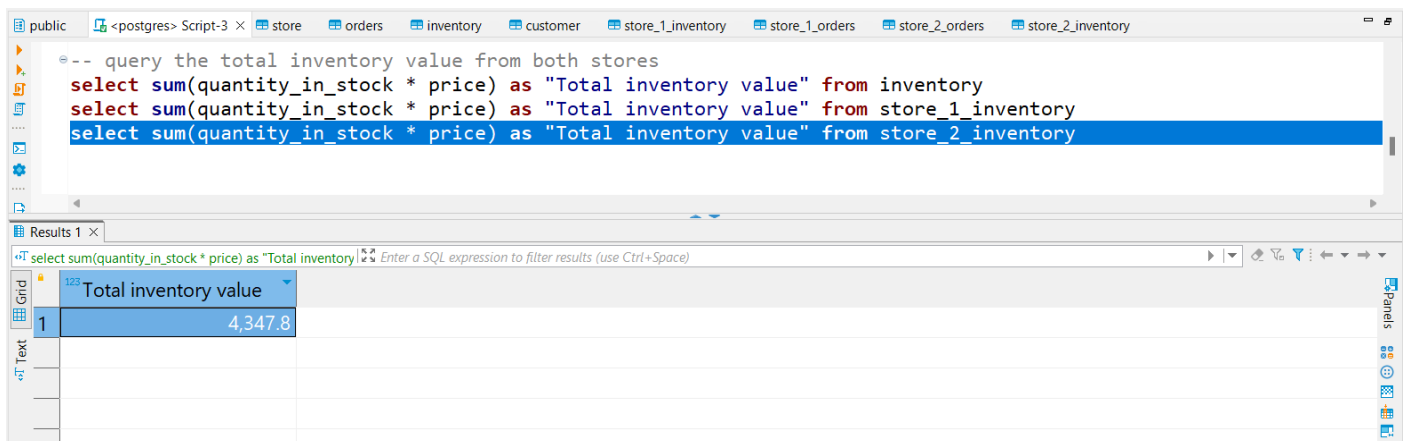
```
-- query the total inventory value from both stores
select sum(quantity_in_stock * price) as "Total inventory value" from inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_1_inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_2_inventory
```

	Total inventory value
1	8,812.55



```
-- query the total inventory value from both stores
select sum(quantity_in_stock * price) as "Total inventory value" from inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_1_inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_2_inventory
```

	Total inventory value
1	4,464.75



```
-- query the total inventory value from both stores
select sum(quantity_in_stock * price) as "Total inventory value" from inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_1_inventory
select sum(quantity_in_stock * price) as "Total inventory value" from store_2_inventory
```

	Total inventory value
1	4,347.8

7) Query to find the customers who have find the products in nearest store

public | <postgres> Script-3 | store | orders | inventory | customer | store_1_inventory | store_1_orders | store_2_orders | store_2_inventory

```
-- query to find customers who have bought the product from nearest store
select c.customer_id, c.cutomer_name, c.email, c.phone_number, o.store_id, c.address
  from customer as c
    join
      orders as o
    on c.customer_id = o.order_id
    join
      store as s
    on s.store_id = o.store_id
 where c.address = s.address
```

customer(+) 1 X

select c.customer_id, c.cutomer_name, c.email, c.phone_number, o.store_id, c.address

	customer_id	cutomer_name	email	phone_number	store_id	address
1	1	John Doe	johndoe@example.com	555-1234	1	123 Main St.
2	3	Bob Johnson	bobjohnson@example.com	555-9101	1	123 Main St.
3	4	Sara Lee	saralee@example.com	555-1212	2	456 Elm St.
4	6	Emily Davis	emilydavis@example.com	555-5656	2	456 Elm St.
5	9	David Kim	davidkim@example.com	555-2323	1	123 Main St.
6	10	Karen Lee	karenlee@example.com	555-4545	2	456 Elm St.

public | <postgres> Script-3 | store | orders | inventory | customer | store_1_inventory | store_1_orders | store_2_orders | store_2_inventory

```
select c.customer_id, c.cutomer_name, c.email, c.phone_number, o.store_id, c.address
  from customer as c
    join
      store_1_orders as o
    on c.customer_id = o.order_id
    join
      store as s
    on s.store_id = o.store_id
 where c.address = s.address
```

customer(+) 1 X

select c.customer_id, c.cutomer_name, c.email, c.phone_number, o.store_id, c.address

	customer_id	cutomer_name	email	phone_number	store_id	address
1	1	John Doe	johndoe@example.com	555-1234	1	123 Main St.
2	3	Bob Johnson	bobjohnson@example.com	555-9101	1	123 Main St.
3	9	David Kim	davidkim@example.com	555-2323	1	123 Main St.

8) Query to find the maximum order size from both the stores

The screenshot shows a PostgreSQL client window with multiple tabs. The active tab is titled "<postgres> Script-3". It contains three SQL queries. The first query is highlighted in blue and is a comment: `-- query to find maximum order size from both the stores`. The second query is `select max(quantity * price) as "Maximum order size" from orders as a join inventory as b on a.product_id = b.product_id`. The third query is `select max(quantity * price) as "Maximum order size" from store_1_orders as a join store_1_inventory as b on a.product_id = b.product_id`. The fourth query is `select max(quantity * price) as "Maximum order size" from store_2_orders as a join store_2_inventory as b on a.product_id = b.product_id`. Below the queries, the "Results 1" tab is active, showing a single row of results. The row is labeled "Row #1" and contains the value "62.97" for the column "Maximum order size". The bottom status bar shows "Refresh", "Save", "Cancel", "Export data", "200", "1", "Row 1/1", "IST en Writable", "Smart Insert", "215 : 1 [128]", and "Sel: 128 | 5".

```
-- query to find maximum order size from both the stores
select max(quantity * price) as "Maximum order size"
  from orders as a
      join
      inventory as b
      on a.product_id = b.product_id

select max(quantity * price) as "Maximum order size"
  from store_1_orders as a
      join
      store_1_inventory as b
      on a.product_id = b.product_id

select max(quantity * price) as "Maximum order size"
  from store_2_orders as a
      join
      store_2_inventory as b
      on a.product_id = b.product_id
```

Row #1
Maximum order size 62.97

Refresh Save Cancel Export data 200 1 Row 1/1 IST en Writable Smart Insert 215 : 1 [128] Sel: 128 | 5

public | <postgres> Script-3 | store | orders | inventory | customer | store_1_inventory | store_1_orders | store_2_orders | store_2_inventory

```
-- query to find maximum order size from both the stores
select max(quantity * price) as "Maximum order size"
  from orders as a
    join
      inventory as b
    on a.product_id = b.product_id

select max(quantity * price) as "Maximum order size"
  from store_1_orders as a
    join
      store_1_inventory as b
    on a.product_id = b.product_id

select max(quantity * price) as "Maximum order size"
  from store_2_orders as a
    join
      store_2_inventory as b
    on a.product_id = b.product_id
```

Results 1 x

select max(quantity * price) as "Maximum order size" fr Enter a SQL expression to filter results (use Ctrl+Space)

	Row #1
Maximum order size	35.96

Refresh Save Cancel Export data 200 1 Row 1/1 IST en Writable Smart Insert 221 : 1 [144] Sel: 144 | 5

```
select max(quantity * price) as "Maximum order size"
  from store_2_orders as a
    join
      store_2_inventory as b
    on a.product_id = b.product_id
```

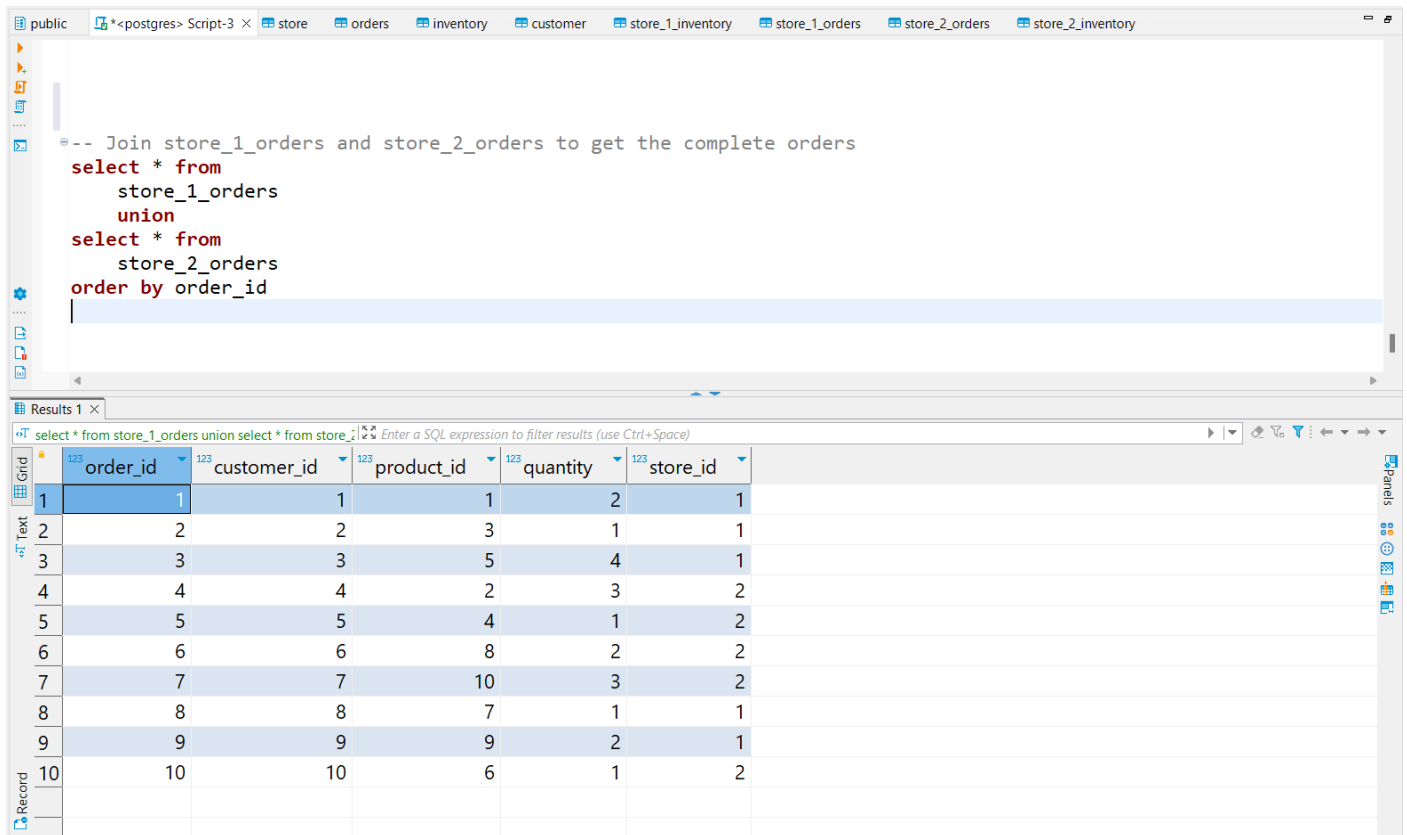
Results 1 x

select max(quantity * price) as "Maximum order size" fr Enter a SQL expression to filter results (use Ctrl+Space)

	Row #1
Maximum order size	62.97

Refresh Save Cancel Export data 200 1 Row 1/1 IST en Writable Smart Insert 227 : 1 [144] Sel: 144 | 5

9) Joining store_1_orders and store_2_orders to get the complete Orders table



The screenshot shows a PostgreSQL client window with a script editor and a results pane. The script editor contains a SQL query that unions the results of two SELECT statements from store_1_orders and store_2_orders, ordered by order_id. The results pane displays the output of this query as a table with 10 rows and 7 columns: order_id, customer_id, product_id, quantity, and store_id. The first 10 rows of the results correspond to the data in the table below.

```
-- Join store_1_orders and store_2_orders to get the complete orders
select * from
  store_1_orders
union
select * from
  store_2_orders
order by order_id
```

	order_id	customer_id	product_id	quantity	store_id
1	1	1	1	2	1
2	2	2	3	1	1
3	3	3	5	4	1
4	4	4	2	3	2
5	5	5	4	1	2
6	6	6	8	2	2
7	7	7	10	3	2
8	8	8	7	1	1
9	9	9	9	2	1
10	10	10	6	1	2

10) Joining store_1_inventory and store_2_inventory to get the complete inventory table

The screenshot shows a PostgreSQL client window with a script editor and a results pane. The script editor contains a SQL query that joins two inventory tables. The results pane displays the output of the query as a table with 10 rows and 6 columns.

```
-- Join store_1_inventory and store_2_inventory to get the complete inventory
select * from
  store_1_inventory
union
select * from
  store_2_inventory
order by product_id
```

Results 1 x

select * from store_1_inventory union select * from store_2_inventory Enter a SQL expression to filter results (use Ctrl+Space)

	product_id	product_name	price	quantity_in_stock	store_id
1	1	Product A	10.99	10	1
2	2	Product B	20.99	50	2
3	3	Product C	5.99	200	1
4	4	Product D	15.99	75	2
5	5	Product E	8.99	15	1
6	6	Product F	25.99	25	2
7	7	Product G	12.99	125	1
8	8	Product H	30.99	10	2
9	9	Product I	7.99	175	1
10	10	Product J	18.99	60	2

Correctness rules of fragmentation

Fragmentation is the major concept in distributed databases. We fragment a table horizontally, vertically, or both and distribute the data to different sites (servers at different geographical locations).

While we perform the fragmentation process, as a result we expect the following as outcomes;

- We should not lose data because of fragmentation
- We should not get redundant data because of fragmentation Hence, to ensure these properties we need to verify whether we performed the fragmentation correctly or not. For this verification we use the correctness rules.

The rules are as follows;

• Completeness -

To ensure that there is no loss of data due to fragmentation. Completeness property ensures this by checking whether all the records which were part of a table (before fragmentation) are found in at least one of the fragments after fragmentation. [We can see in the above queries that the Total number of unique ids in the inventory and orders table match with the number of unique ids in the sum of two fragmented tables.](#)

• Reconstruction -

This rule ensures the ability to reconstruct the original table from the fragments that are created. This rule is to check whether the functional dependencies are preserved or not.

If a table R is partitioned into fragments R1, R2, ..., Rn,
then Reconstruction insists the following; $R = R1 \cup R2 \cup \dots \cup Rn$

For the above scenario we can see that the Union of the two tables gives us back the original table thus making sure that Reconstruction is possible.

• Disjointness -

This rule ensures that no record will become a part of two or more different fragments during the fragmentation process.

If a table R is partitioned into fragments R1, R2, ..., Rn,
then Disjointness insists the following; $R1 \cap R2 \cap \dots \cap Rn = \text{Null set}$

For the above scenario we see that the relations "<" and ">=" do not share any common values thus inherently satisfying the Disjoint condition.

Lossless decomposition

Lossless join decomposition is a way to decompose a relation R into two or more relations R_1, R_2 such that their natural join returns the original relation R . This helps to eliminate data redundancy in a database while retaining the original data. The decomposed tables can be joined together to reconstruct the original relation R . Only the normal forms 1NF, 2NF, 3NF, and BCNF are suitable for lossless join decomposition. The common attribute used for decomposition must be a candidate key or a super key in either R_1, R_2 , or both.

Conclusion:

By performing the above experiment:

- I learnt how to apply horizontal fragmentation on a table and fragmentation increases the efficiency by distributing the database
- I learnt how to check whether the fragmentation was correct by checking the correctness rules and about lossless decomposition