Database Management System (DBMS)

Project Name:
Order Management Enterprise

Author:
Tirth Bhimani

TABLE OF CONTENTS

PROJECT DESCRIPTION

An order management system (OMS) is a digital way to manage the lifecycle of an order. It tracks all the information and processes, including order entry, inventory management, fulfilment and after-sales service. An OMS offers visibility to both the business and the buyer. Organizations can have near real-time insight into inventories and customers can check when an order will arrive. Order management starts when a customer places an order and ends once they receive their package or service. It allows a business to coordinate the entire fulfilment process.

Why is Order Management Important?

Order management touches virtually every system and process in the supply chain. Most companies no longer contain order management within their organization. They involve multiple partners, such as parts and components suppliers, assembly and packaging services or distribution centers — making it easy to lose visibility and control of an order. This results in costly manual processes to complete and deliver the order without errors. An OMS can help control costs and generate revenue by automating manual processes and reducing errors.

Why do we need database?

With a high volume of orders frequently coming in from many channels, there would be a high error margin if traditional order processing methods are used. Task automation and the centralization of warehouse and inventory data not only takes care of such errors but also reduces chances of data security and loss. Therefore, it can be concluded that usage of the manual maintenance there are a number of difficulties and drawbacks exist in the system.

Cons of Existing System (Pen-Paper system or File System):

- Large volumes of Orders, Products, Customers etc. can not be stored in one place only.
- Multiple Users should not be able to modify data at the same time.
- Manual maintenance is time consuming and can cause many errors.
- While access to text files or spreadsheets can be secured, once someone has access to a file, they have access to all data within that file.

Whenever using a hand-written register or book for entering a particular data, it can have many difficult consequences which cannot be solved practically or if solved then it could be much expensive or time consuming. Thinking from supply chain perspective, if a local shopkeeper at Delhi wants to access a particular data from database of the other branch situated at Mumbai, then if the system uses hand-written registers, then it would be impractical to send the registers from Delhi to Mumbai via any mode of transportation.

The ordering industry mainly works on trust. Whenever a person buys his or her product to any location via the internet then he/she mainly worries whether the purchased product have been confirmed or the size/colour of product to which they are buying by the help of the merchant website whether it would be successful and enjoyable or would turn into ashes. So, with the help of our database, we would like to assure the customers by adding some attributes which ensure whether the products they are buying would be safely delivered to them. We would also think of the host company which is providing the products to the customer that has available storage for the same product and they have completed the payments or not.

Beneficiaries of Database

Customers expect to be informed every step of the way across the buying journey with regular email updates, tracking information, and delivery status alerts. Without order management solutions in place to compile customer information and filter data into one place, there are more opportunities for error.

From buy online/pickup in-store (BOPIS) to same-day shipping, customers expect as much choice in delivery as they do in channels of engagement. Shoppers expect to be able to have an order fulfilled in a way, time, and place that is most convenient to them. Each touchpoint in the journey presents an opportunity to provide great customer experience and boost retention and revenue.

These days it's about complexity, not about size. If you have multiple stores, warehouses, or sales channels and websites in various countries, and you want to try and bring them all together, order management makes sense. With order management, you're able to pull all of those different platforms for different countries together – in one place.

What requirements must be checked before making a database?

For safety requirements, the system shall save the data if the server goes down or any other harmful things happen to the server for the sake of safety. For security purposes, the application will be password protected or we can have an OTP system, i.e., when the user enters the application, the One Time Password (OTP) will be sent to the respective user. The user should be strictly advised that he/she should not share the login details and the same should be applied with the manager and developer. The planning and the logical structure of the database should not be shared with anyone except the trustful. Similarly, the applications made using the database must be accessible using trusted platforms only and not by any third-party apps. The complexity of the system should be kept in such an

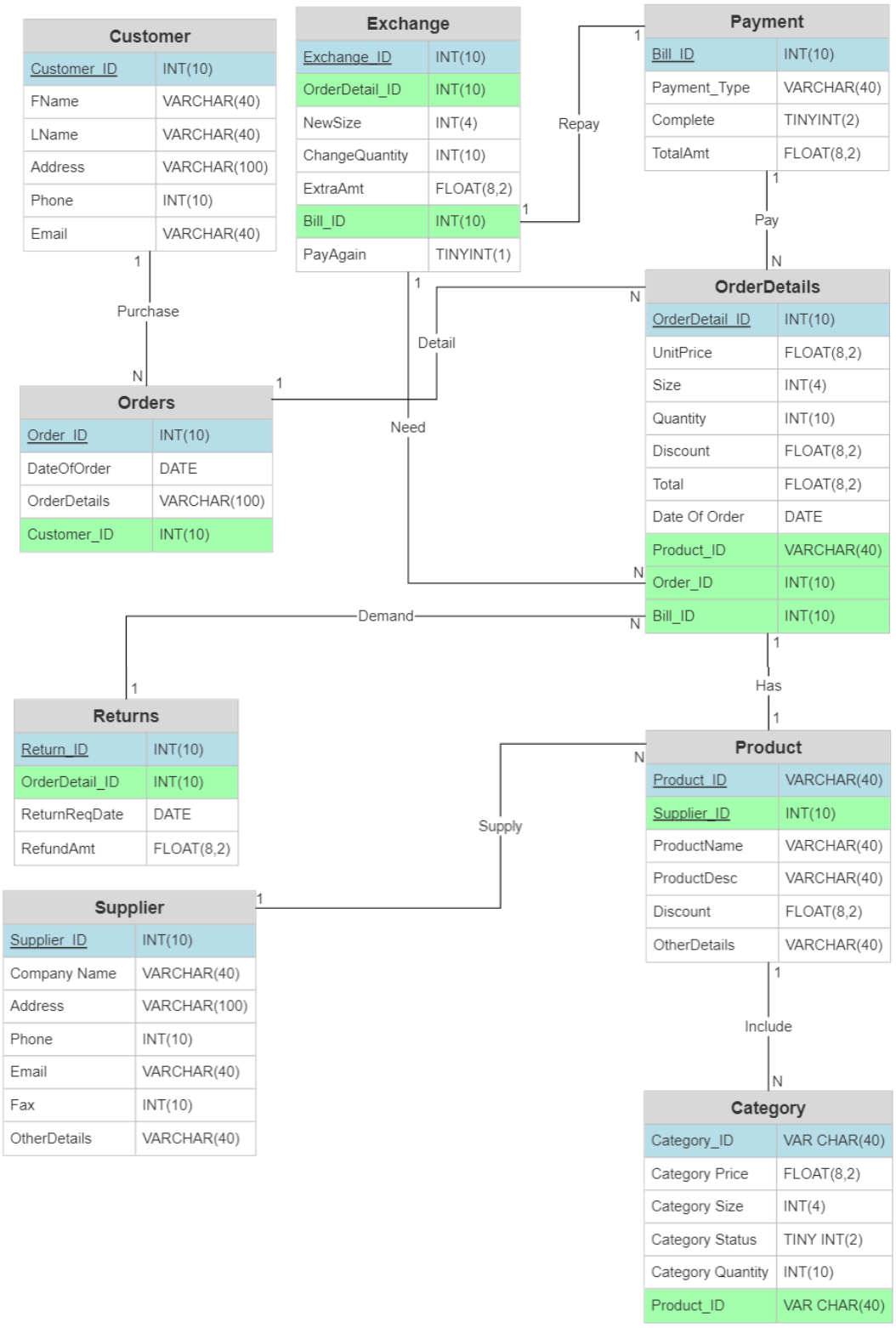
order that whenever we need to manipulate any entity or table data it should be done easily. If we want to insert new data/attributes to a table then it should not create data redundancy or inconsistency in the database. So, the system needs to be made such that it can be maintained whenever needed.

The application will be easily accessible on any OS-based system. For different classes of users, Data requirements will be different. Customers who want to purchase various products will be required to submit one of the id proofs and need to complete their user profile. For seller: Licence of their company will be required. Using Data integrity, users know what data is being collected and who has the right to access it and how they are able to interact with it and so all that information will be inaccessible to unauthorized users and privacy of data is at its fullest. Database should be error-free by storing accurate and consistent data in a database.

ER Model

ER Diagram

Link of ER Diagram - [Link]



Conventions followed in ER diagram:

- Table name is highlighted in the centre of first row in bold.
- Name of attributes are written in the table following the table name.
- Data type of attributes are shown in same row.
- Primary key for the table is underlined and highlighted with blue colour
- Foreign keys for the table are highlighted with green colour
- Relationship between 2 tables is showed along the edge. Type of Relationships:
    - One-to-many Relation (1-N): Customer-Payment, Orders-Orderdetails, Payment-Orderdetails, Exchange-Orderdetails, Supplier-Product, Product-Category, Returns-Orderdetails
    - One-to-One Relation (1-1): Payment-Exchange
    - Many-to-Many Relation (N-M): None
- Name of the relationship is shown in the middle of the edge

Relational Schema

- Customer (**Customer_ID**, FName, LName, Address, Phone, Email)
- Orders (**Order_ID**, DateOfOrder, *Customer_ID,* OrderDetails)
- OrderDetails (**OrderDetail_ID**, UnitPrice, Color, Quantity, Discount, Total, Date, *Category_ID, Order_ID, Bill_ID*)
- Payment (**Bill_ID**, PaymentType, Complete , TotalAmt)
- Exchange (**Exchange_ID**, *OrderDetail_ID,* ChangeQuantity, CategoryColor, NewAmt, *Bill_ID,* PayAgain, Category_ID)
- Product (**Product_ID**, ProductName, ProductDesc, Discount, *Supplier_ID*, OtherDetails)
- Category (**Category_ID**, CategoryPrice, CategoryColor, CategoryStatus, CategoryQuantity, *Product_ID*)
- Supplier (**Supplier_ID**, CompanyName, Address, Phone, Email, Fax, OtherDetails)
- Returns (**Return_ID**, *OrderDetail_ID*, ReturnReqDate)

NORMALIZATION AND SCHEMA REFINEMENT

Functional Dependencies

Entity Name	**Dependencies**
Customer	Customer_ID -> FName Customer_ID -> LName Customer_ID -> Address Customer_ID -> Phone Customer_ID -> Email Phone -> Customer_ID Phone -> FName Phone -> LName Phone -> Address Phone -> Email Email -> Customer_ID Email -> FName Email -> LName Email -> Address Email -> Phone
Order	Order_ID -> DateOfOrder Order_ID -> OrderDetails Order_ID -> Customer_ID
OrderDetails	OrderDetail_ID -> UnitPrice OrderDetail_ID -> Color OrderDetail_ID -> Quantity OrderDetail_ID -> Discount OrderDetail_ID -> Total OrderDetail_ID -> Date OrderDetail_ID -> Category_ID OrderDetail_ID -> Order_ID OrderDetail_ID -> Bill_ID Category_ID -> Discount Order_ID -> Date Order_ID -> Bill_ID
Payment	Bill_ID -> Payment_Type Bill_ID -> Complete Bill_ID -> TotalAmt

Entity Name	Dependencies
Exchange	Exchange_ID -> OrderDetail_ID Exchange_ID -> NewColor Exchange_ID -> ChangeQuantity Exchange_ID -> NewAmt Exchange_ID -> Bill_ID Exchange_ID -> PayAgain NewAmt -> PayAgain
Product	Product_ID -> ProductName Product_ID -> ProductDesc Product_ID -> Discount Product_ID -> Supplier_ID Product_ID -> OtherDetails ProductName -> Supplier_ID ProductDesc -> ProductName ProductDesc -> Discount ProductDesc -> Supplier_ID
Category	Category_ID -> CategoryPrice Category_ID -> CategoryColor Category_ID -> CategoryStatus Category_ID -> CategoryQuantity Category_ID -> Product_ID CategoryQuantity -> CategoryStatus
Supplier	Supplier_ID -> Name Supplier_ID -> Address Supplier_ID -> Phone Supplier_ID -> Email Supplier_ID -> Fax Supplier_ID -> OtherDetails Email -> Name Email -> Address Email -> Phone Email -> Fax Email -> OtherDetails Email -> Supplier_ID Phone -> Name Phone -> Address Phone -> Email

Entity Name	Dependencies
	Phone -> Fax Phone -> OtherDetails Phone -> Supplier_ID Fax -> Name Fax -> Address Fax -> Phone Fax -> Email Fax -> OtherDetails Fax -> Supplier_ID
Return	Return_ID -> OrderDetail_ID Return_ID -> ReturnReqDate Return_ID -> RefundAmt

Normalization

1. First Normal Form(1NF):

In order table, idea of Orderdetails attribute was to contain list of all orderdetails in a string format which leads to making the variable multivalued. For that reason, orderdetails attribute is dropped off. Remaining Tables satisfiy the First Normal Form as they do not contain any multivalued attributes within the table.

2. Second Normal Form (2NF):

Second Normal Form (2NF) is based on the concept of functional dependency. It applies to a relation in which composite keys are present i.e., relations with a primary key composed of two or more attributes. In above mentioned schema each and every relation has single attribute primary key and hence every relation is in 2NF. Every Relation has no partial dependency i.e., no non - prime attributes are dependent on any proper subset of any candidate key of the table.

3. Redundancies:

Data redundancy occurs when the same piece of data exists in multiple places, whereas data inconsistency is when the same data exists in different formats in multiple tables. By analysing functional dependencies, some redundancies can be found. For Example,

- In Category table, it is clear that Category Quantity can itself verify CategoryStatus. If CategoryQuantity has value greater than zero than is clear that item is available for shopping. That means CategoryStatus can be dropped from Category table to reduce redundancy.
- In OrderDetails table, It can be observed that TotalAmt can be derived from (UnitPrice – Discount) * Quantity. So, TotalAmt can be dropped from the table.

- In Exchange table, CategoryColor is dependent variable on Category_ID. As by Category_ID, CategoryColor can simply be removed.
- In OrderDetails table, Color attribute is dependent on Category_ID. Color can be dropped.
- In Exchange table, PayAgain is dependent on NewAmt which can be removed.

4. 3rd Normal Form (3NF):

A given relation is said to be in its third normal form when it's in 2NF but has no transitive partial dependency. Meaning, when no transitive dependency exists for the attributes that are non-prime, then the relation can be said to be in 3NF. For given functional dependencies. It can be said that after applying 2nd Normal form it will automatically get converted into 3rd normal form.

Final Relational Schema

- Customer (**Customer_ID**, FName, LName, Address, Phone, Email)
- Orders (**Order_ID**, DateOfOrder, *Customer_ID,* ~~OrderDetails~~)
- OrderDetails (**OrderDetail_ID**, UnitPrice, ~~Color~~, Quantity, Discount, ~~Total~~, Date, *Category_ID, Order_ID, Bill_ID*)
- Payment (**Bill_ID**, PaymentType, Complete , TotalAmt)
- Exchange (**Exchange_ID**, *OrderDetail_ID,* ChangeQuantity, ~~CategoryColor~~, NewAmt, *Bill_ID,* PayAgain, Category_ID)
- Product (**Product_ID**, ProductName, ProductDesc, Discount, *Supplier_ID*, OtherDetails)
- Category (**Category_ID**, CategoryPrice, CategoryColor, ~~CategoryStatus~~, CategoryQuantity, *Product_ID*)
- Supplier (**Supplier_ID**, CompanyName, Address, Phone, Email, Fax, OtherDetails)
- Returns (**Return_ID**, *OrderDetail_ID*, ReturnReqDate)

MySQL Queries

Table Creation

```
create table if not exists Customer(
        Customer_ID int not null auto_increment,
   FName varchar(40) not null,
   LName varchar(40) not null,
   Address varchar(100) not null,
   Email varchar(40) not null,
   Phone varchar(20) not null,
   primary key (Customer_ID),
   unique (Email),
   unique (Phone)
);

create table if not exists Supplier(
        Supplier_ID int not null auto_increment,
   CompanyName varchar(40) not null,
   Address varchar(100) not null,
   Phone varchar(20) not null,
   Email varchar(40) not null,
   Fax varchar(8),
   OtherDetails varchar(100),
   primary key (Supplier_ID),
   unique (Phone),
   unique (Email),
   unique (Fax)
);
```

```
create table if not exists Product(
        Product_ID int not null auto_increment,
   ProductName varchar(100) not null,
   ProductDesc varchar(40) not null,
   Discount float not null,
   OtherDetails varchar(100),
   Supplier_ID int not null,
   primary key (Product_ID),
   foreign key (Supplier_ID) references Supplier(Supplier_ID) match simple
   on update cascade
   on delete cascade
);
```

```
create table if not exists Category(
        Category_ID int not null auto_increment,
   CategoryPrice float not null,
   CategoryColor varchar(10) not null,
   CategoryQuantity int not null,
   Product_ID int not null,
   primary key (Category_ID),
   foreign key (Product_ID) references Product(Product_ID) match simple
   on update cascade
   on delete cascade
);
```

```
create table if not exists Orders(
        Order_ID int not null auto_increment,
   DateOfOrder date not null,
   Customer_ID int not null,
   primary key (Order_ID),
   foreign key (Customer_ID) references Customer(Customer_ID) match simple
   on update cascade
   on delete cascade
);
```

```
create table if not exists Payment(
        Bill_ID int not null auto_increment,
   PaymentType varchar(40) not null,
   Complete tinyint default(0),
   TotalAmt float not null,
   primary key (Bill_ID)
);
```

```
create table if not exists OrderDetails(
        OrderDetail_ID int not null auto_increment,
    UnitPrice float not null,
    Quantity int not null,
    Discount float not null,
    DateofOrder date default(CURRENT_DATE),
    Category_ID int not null,
    Order_ID int not null,
    Bill_ID int not null,
    primary key (OrderDetail_ID),
    foreign key (Category_ID) references Category(Category_ID) match simple
    on update cascade
    on delete cascade,
    foreign key (Order_ID) references Orders(Order_ID) match simple
    on update cascade
    on delete cascade,
    foreign key (Bill_ID) references Payment(Bill_ID) match simple
    on update cascade
    on delete cascade
);

create table if not exists Returns(
        Return_ID int not null auto_increment,
    OrderDetail_ID int not null,
    ReturnReqDate date not null,
    primary key (Return_ID),
    foreign key (OrderDetail_ID) references OrderDetails(OrderDetail_ID) match simple
    on update cascade
    on delete cascade
);
```

```
create table if not exists Exchange(
        Exchange_ID int not null auto_increment,
   OrderDetail_ID int not null,
   Category_ID int not null,
   ChangeQuantity int not null,
   NewAmt float not null,
   Bill_ID int not null,
   primary key (Exchange_ID),
   foreign key (OrderDetail_ID) references OrderDetails(OrderDetail_ID) match simple
   on update cascade
   on delete cascade,
   foreign key (Bill_ID) references Payment(Bill_ID) match simple
   on update cascade
   on delete cascade,
   foreign key (Category_ID) references Category(Category_ID) match simple
   on update cascade
   on delete cascade
);
```

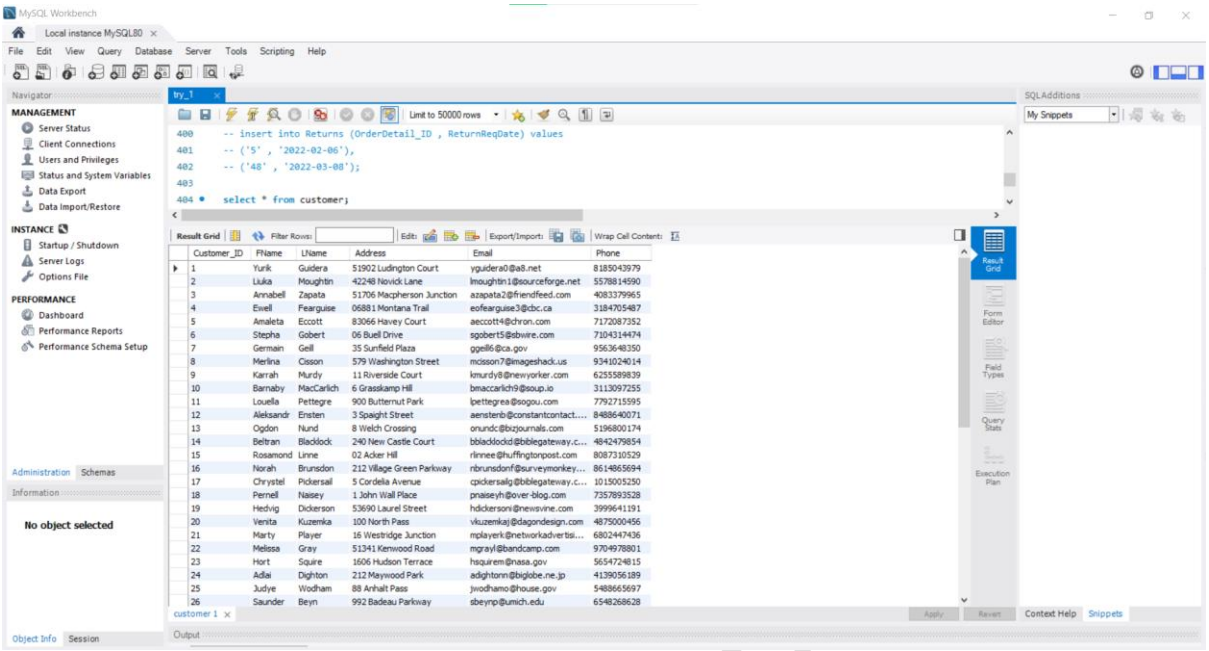
Data Insertion

Link of Website Used for Random Data Generation - Link

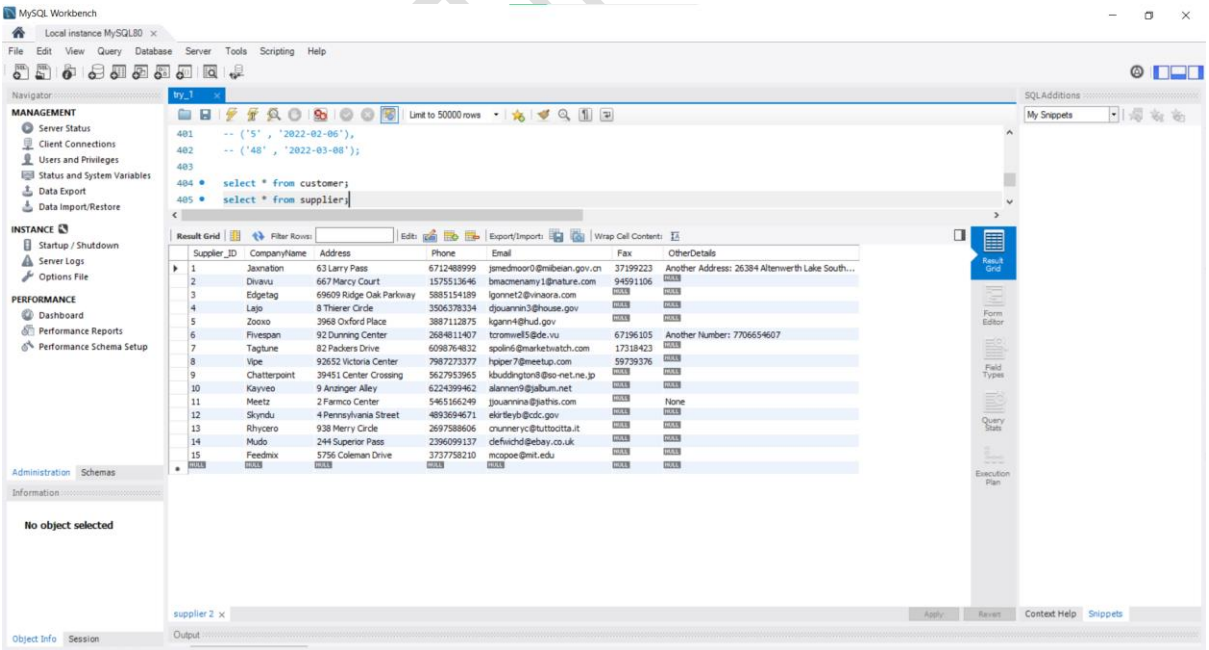
Inserted data is generated randomly. Logical part is added with Python script in data generation to derive some meaningful data from random data.

Show Table Entries

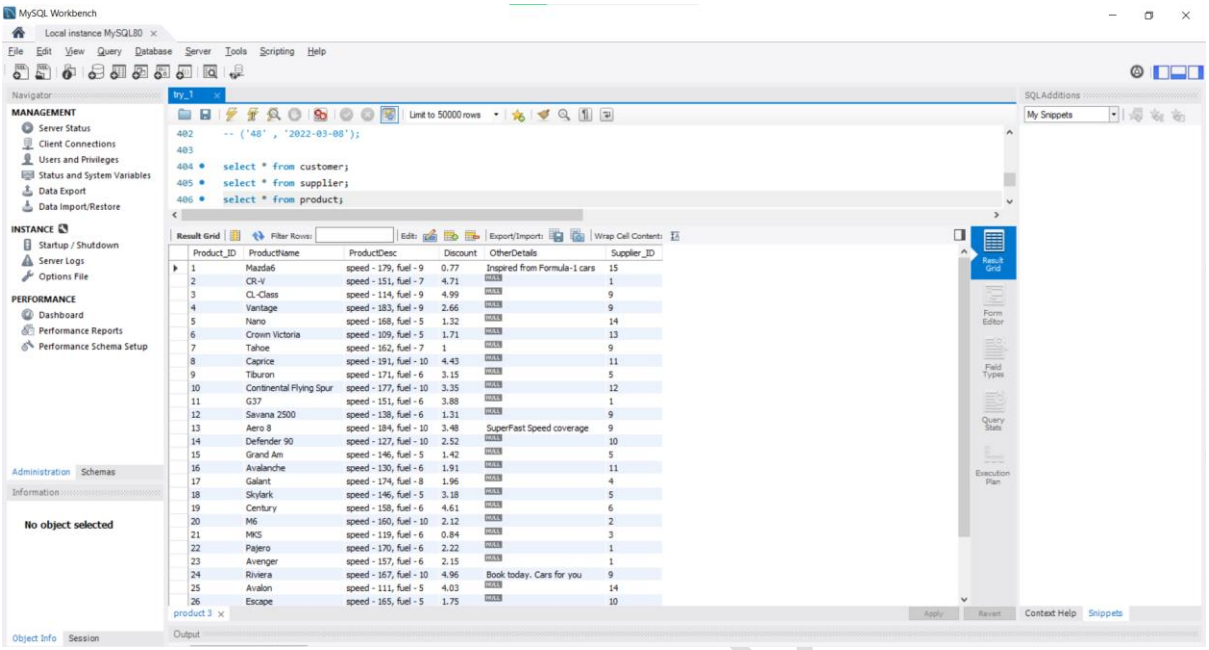
1. Show the all details of Customer
   select * from customer;



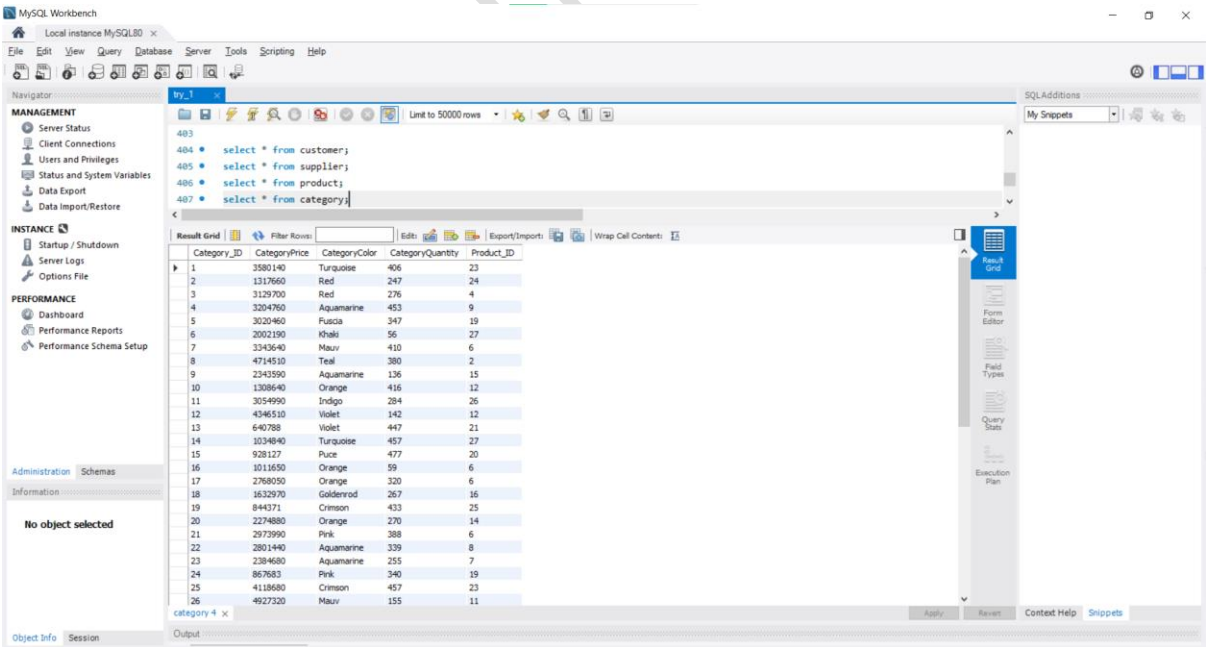
2. Show the all details of Supplier
   select * from supplier;



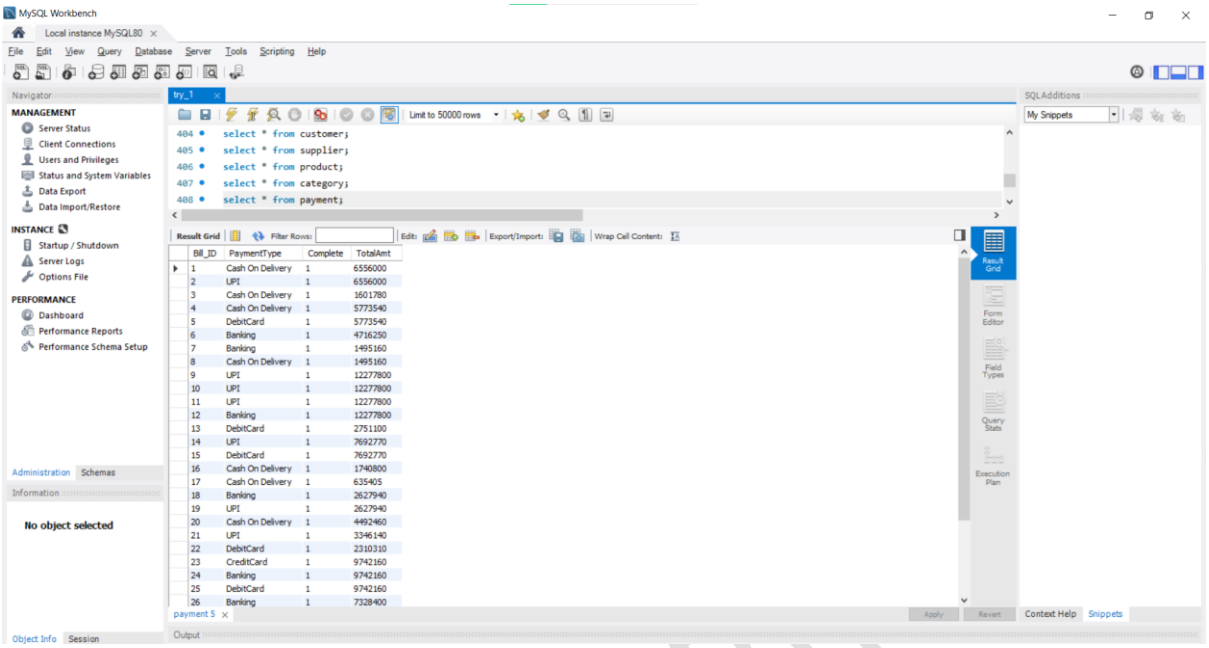
3. Show the all details of Product
   select * from product;



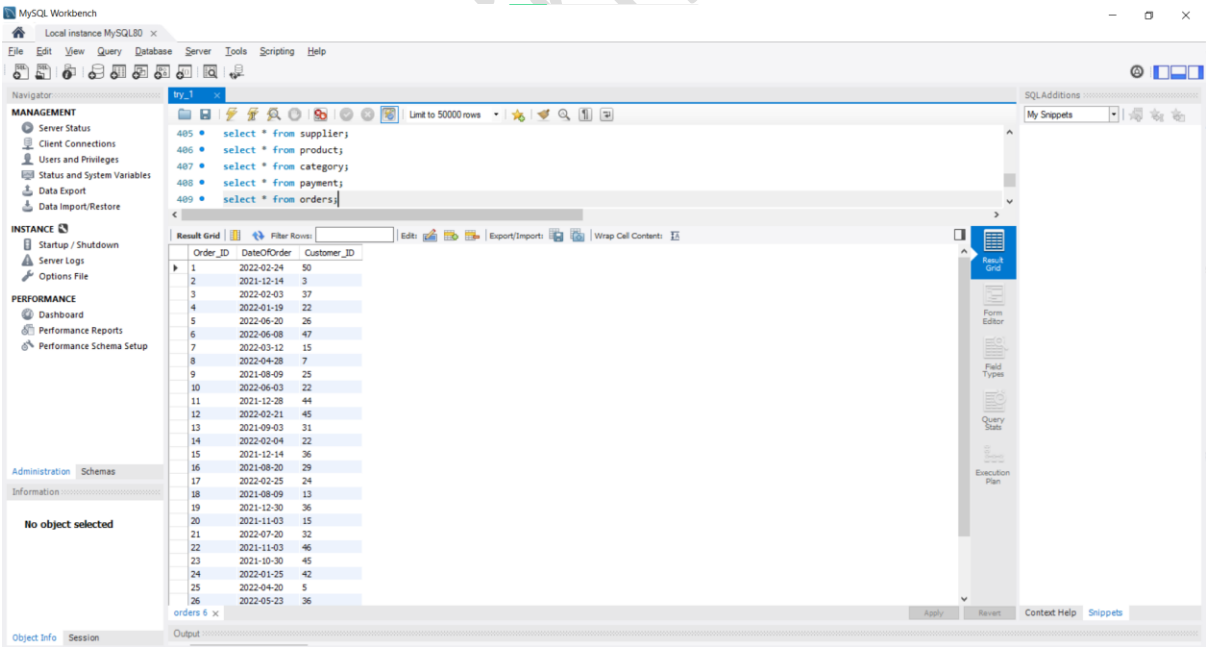
4. Show the all details of Category
   select * from categoy;



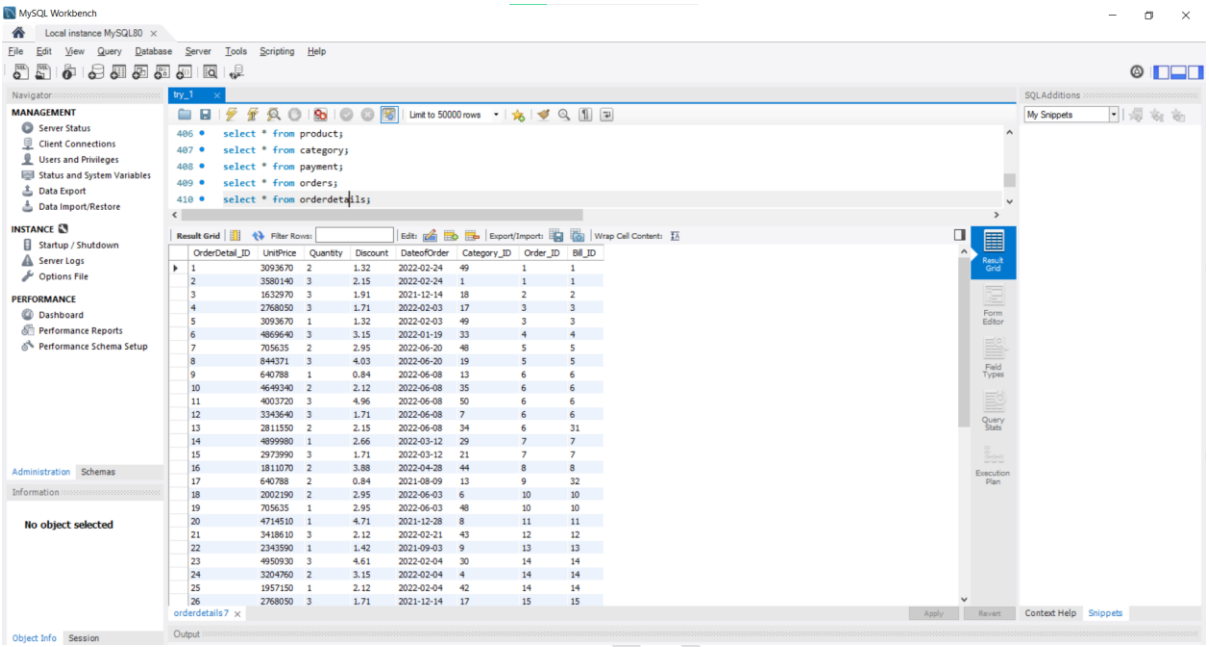
5. Show the all details of Payment
   select * from payment;



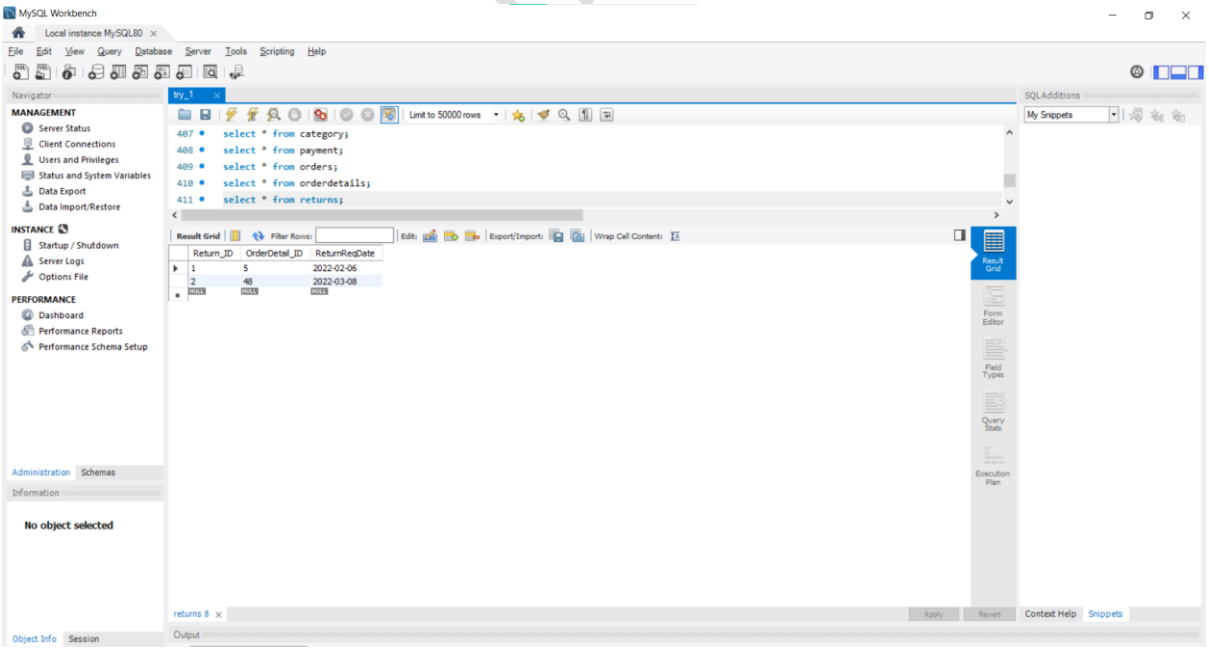
6. Show the all details of Order
   select * from orders;



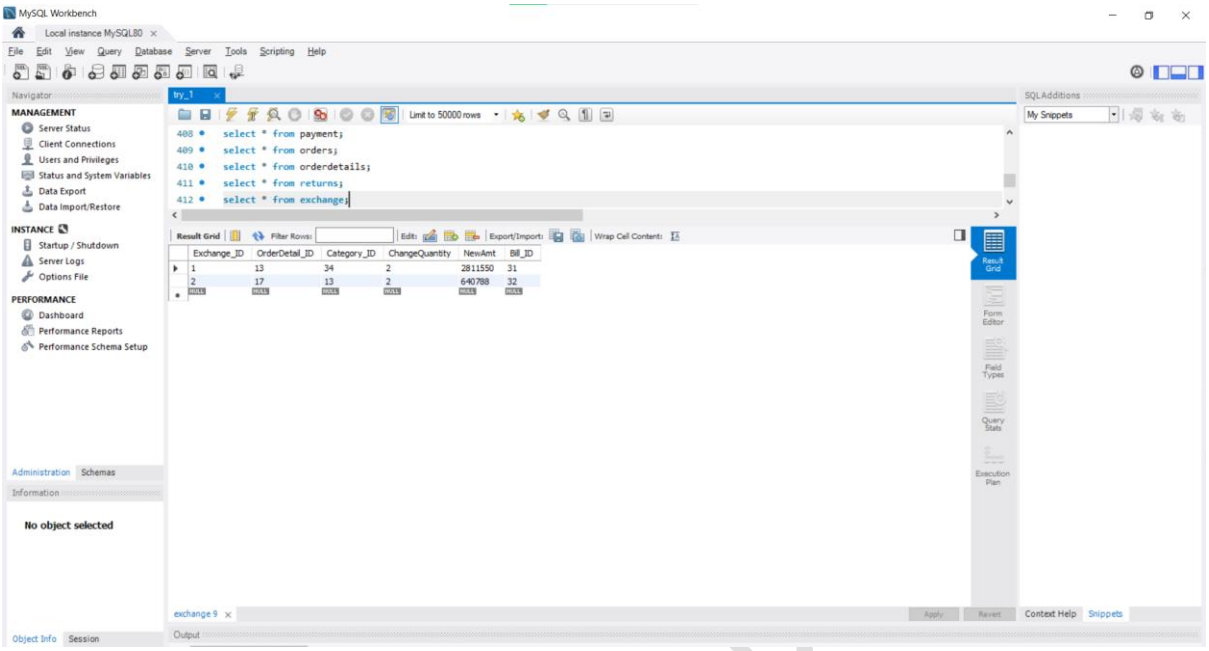
7. Show the all details of OrderDetails
   select * from orderdetails;



8. Show the all details of Return
   select * from returns;



9. Show the all details of Exchange
   select * from exchange;



Length of Tables

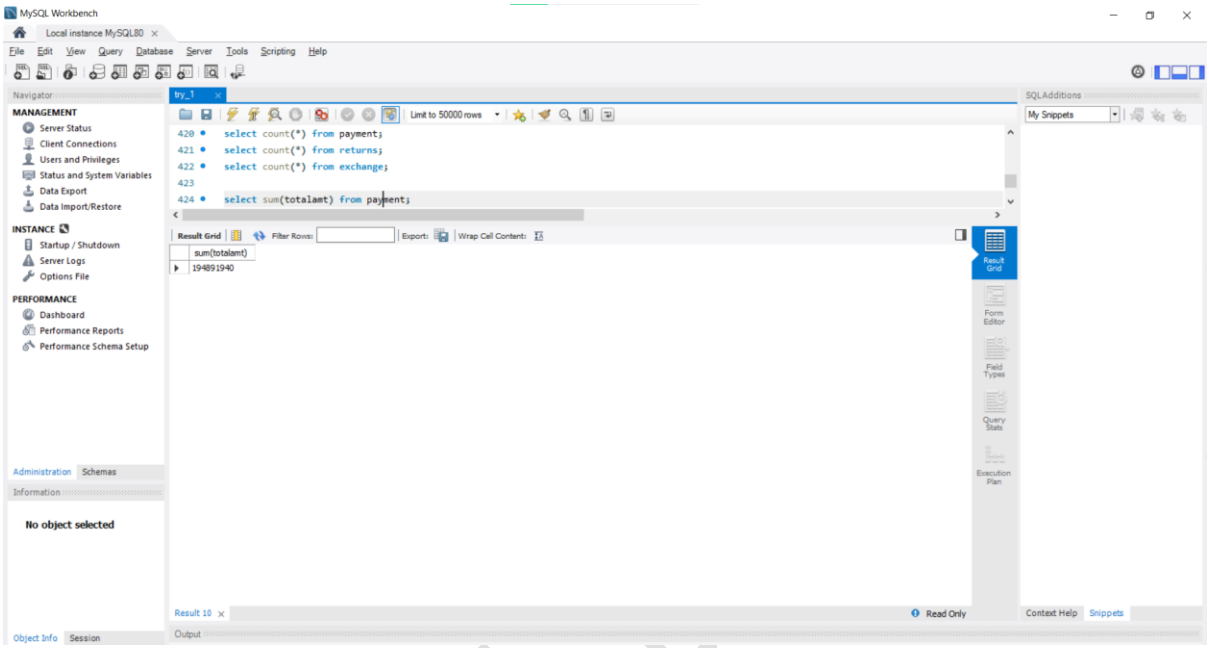
Query – select count(*) from Table_Name

Table Name	Length query response
Customer	count(*) 50
Supplier	count(*) 15
Order	count(*) 30
OrderDetails	count(*) 50
Product	count(*) 30
Category	count(*) 50
Payment	count(*) 32
Return	count(*) 2
Exchange	count(*) 2

SQL Queries

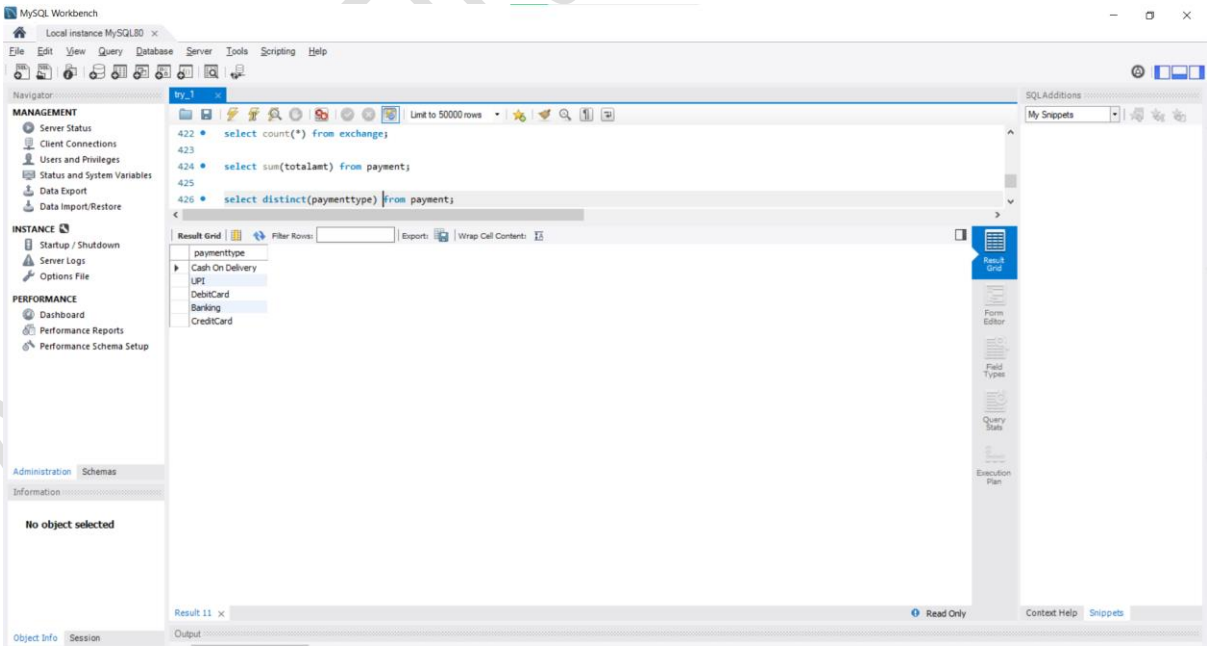
Select … from queries

1. Show the sum of all payments made till now.
   select sum(totalamt) from payment;



Tuple count = 1

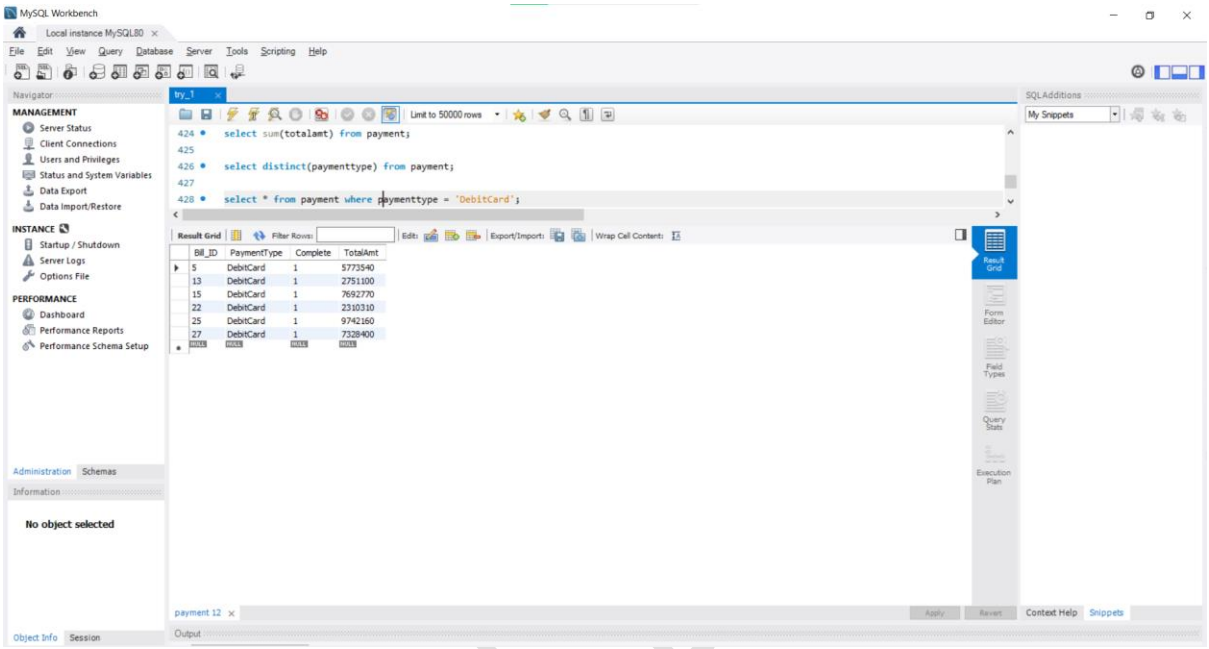
2. Show different type of payments made.
   select distinct(paymenttype) from payment;



Tuple count = 5

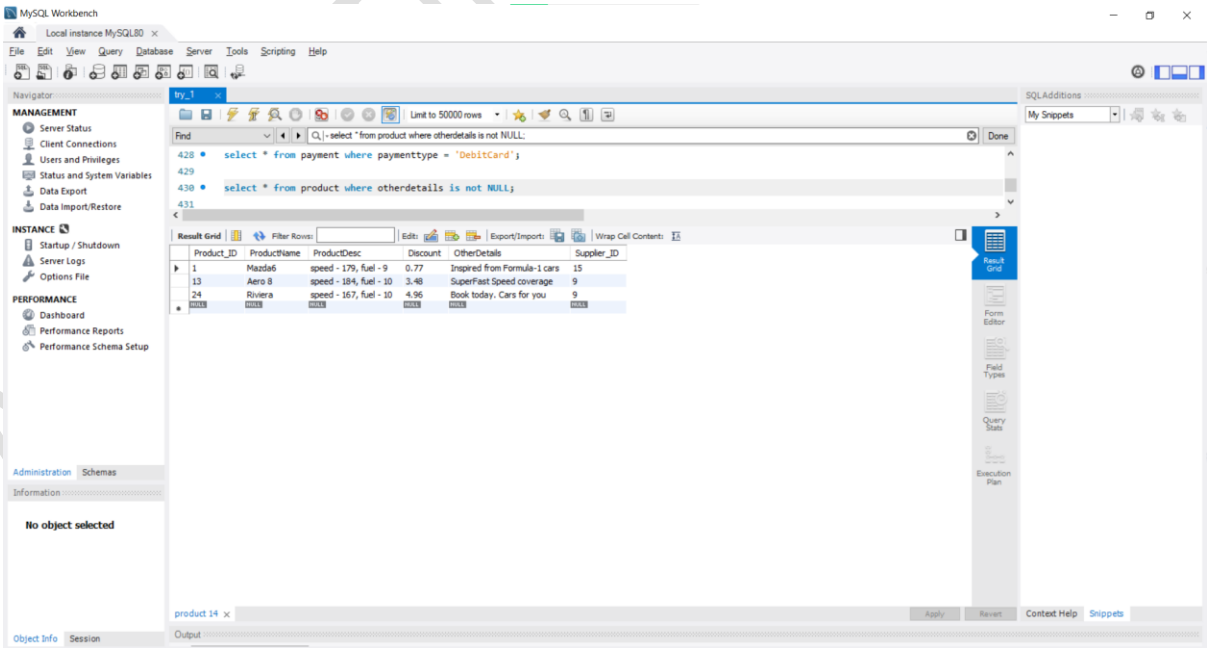
Select … from … where queries

3. Show all the payments made with debit card.
   select * from payment where paymenttype = ‘DebitCard’



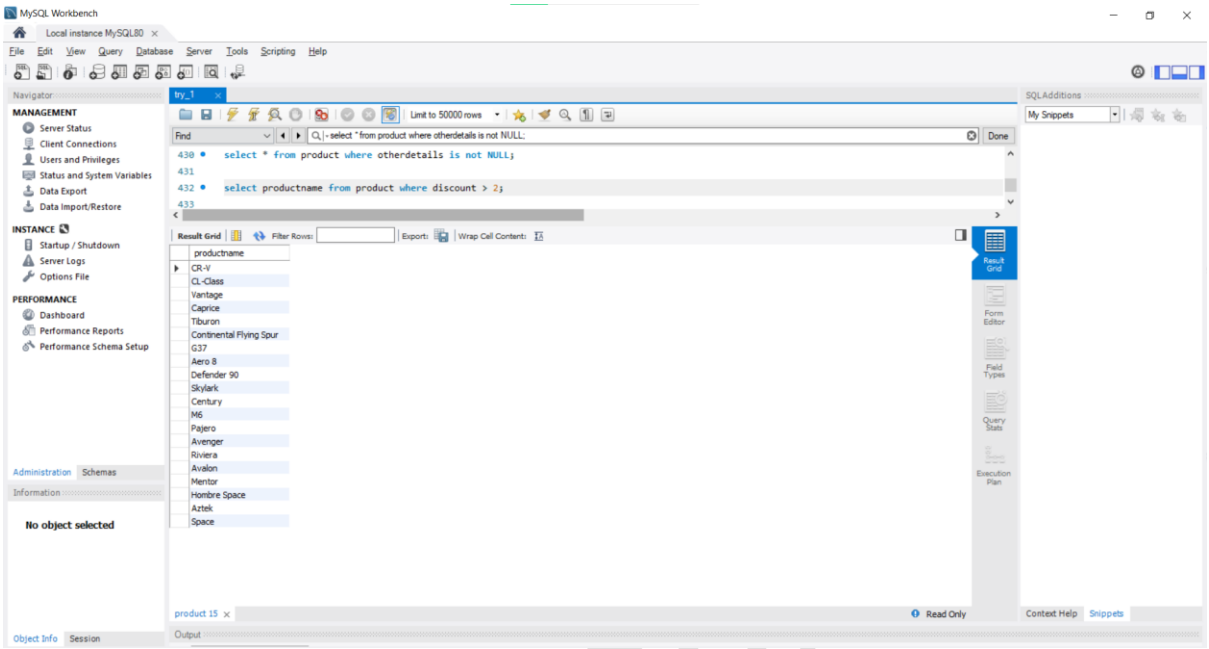
Tuple count = 6

4. Show the details of product in which otherdetails field is not empty.
   select * from product where otherdetails is not NULL;



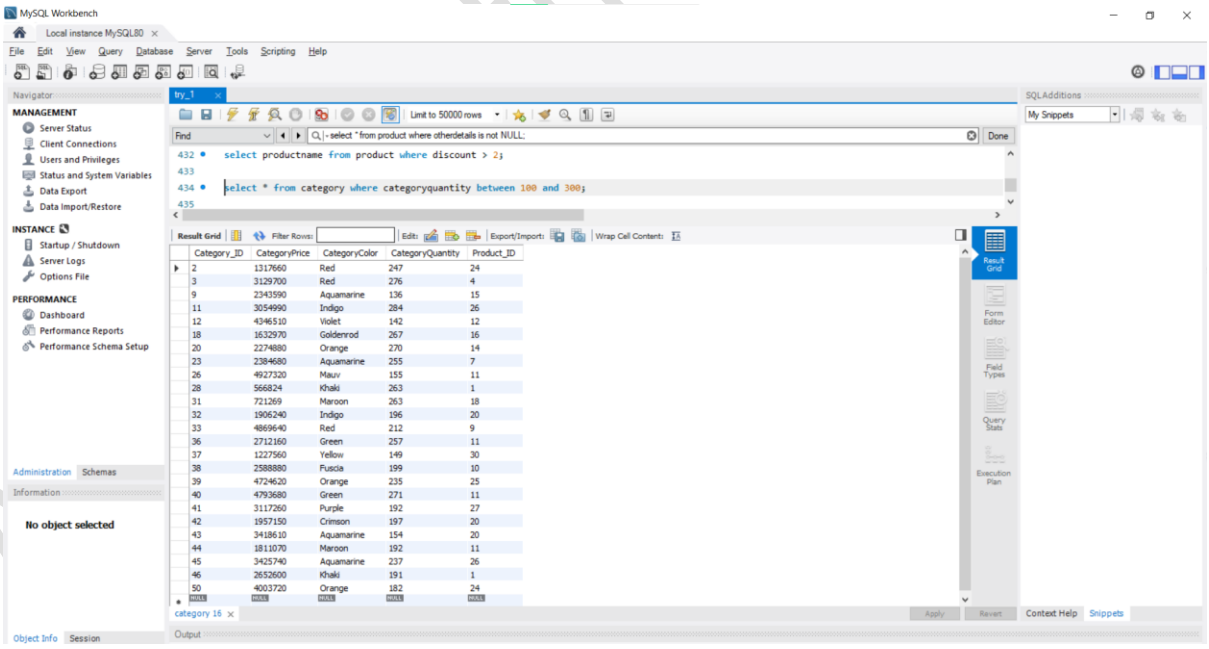
Tuple Count = 3

5. Show all product names with discount greater that 2%
   select productname from product where discount > 2;



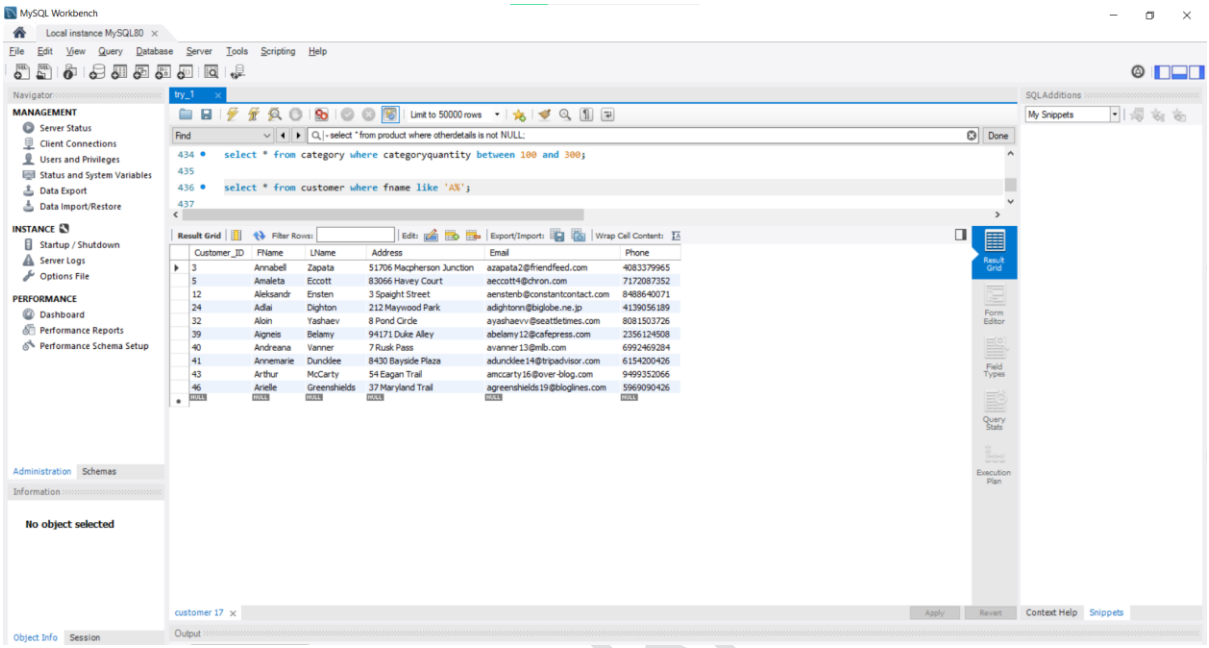
Tuple count = 20

6. Show category of product available with between 100 to 300.
   select * from category where categoryquantity between 100 and 300;



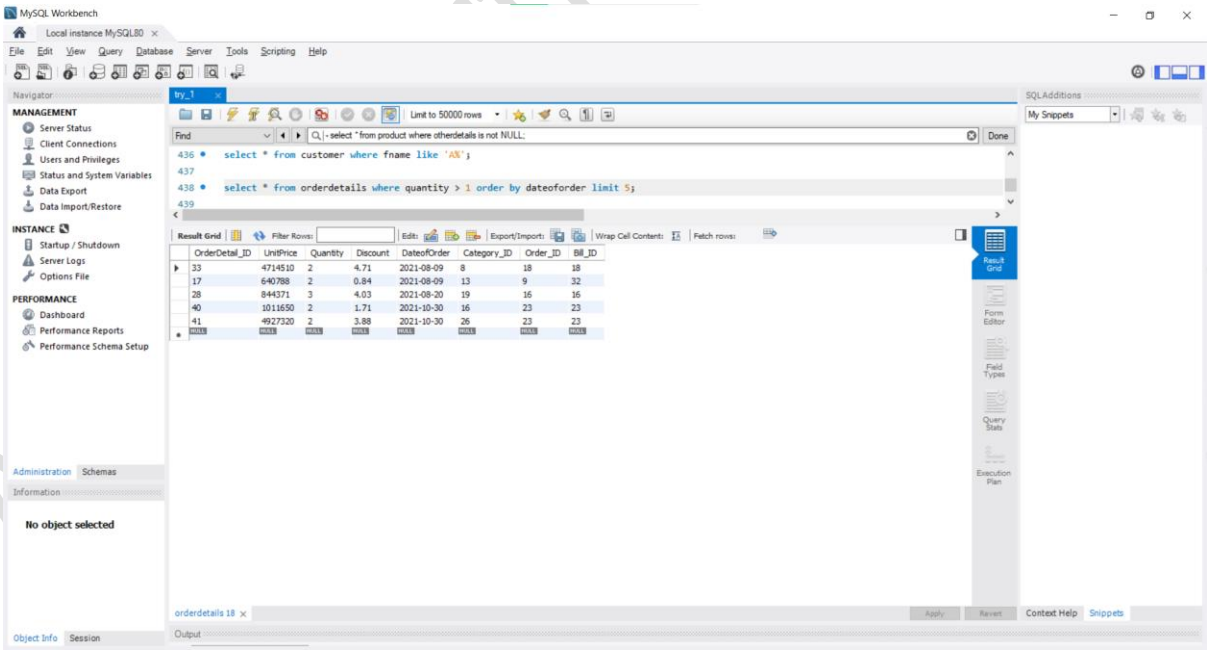
Tuple count = 25

7. Show details of customer with first name starting from A.
   select * from customer where fname like ‘A%’;



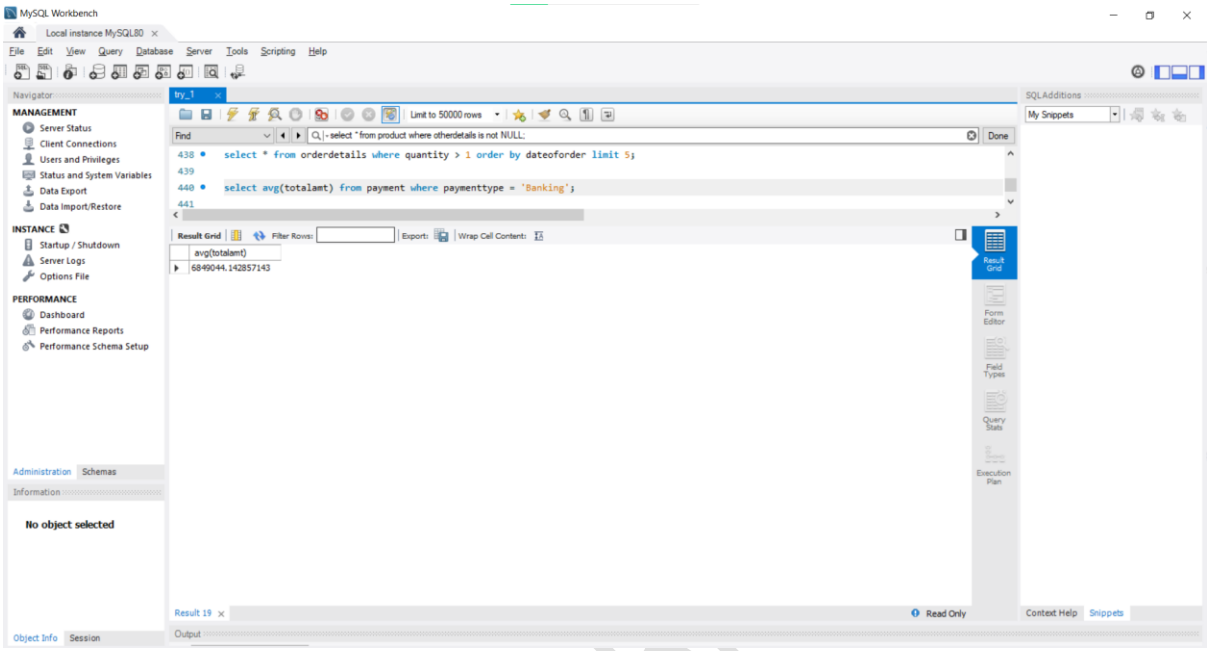
Tuple count = 10

8. Show details of first 5 orders in OrderDetails with quantity greater than 1.
   select * from orderdetails where quantity > 1 order by dateoforder limit 5;



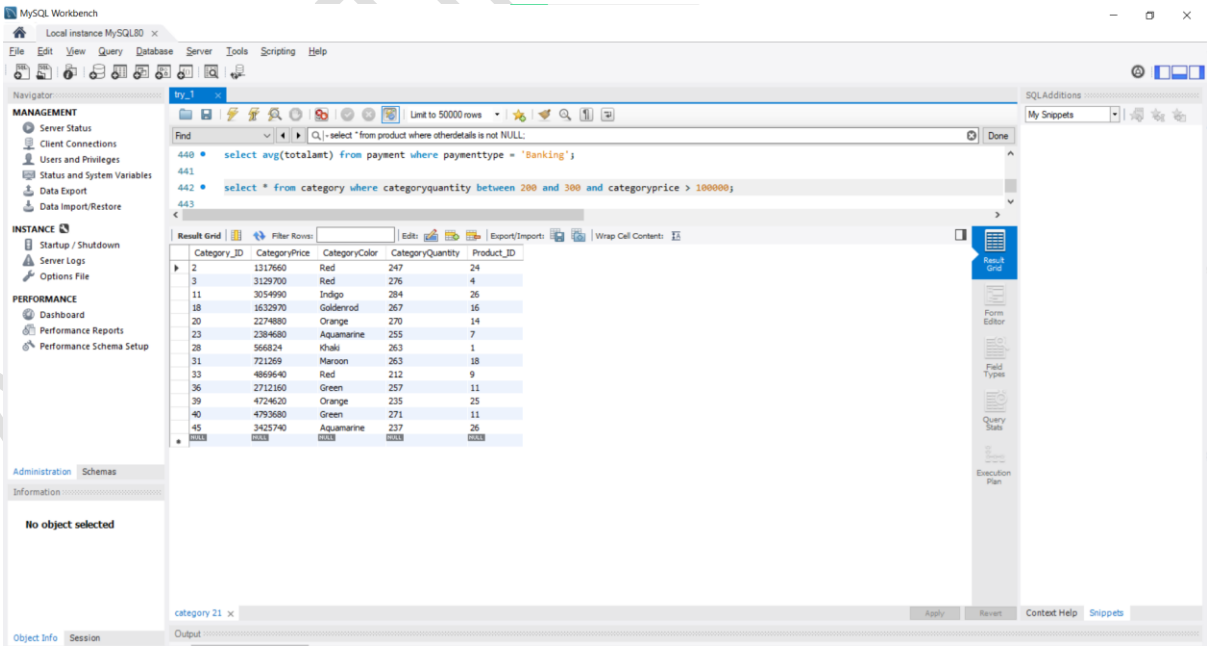
Tuple count = 5

9. Calculate average amount paid via Banking mode.
   select avg(totalamt) from payment where paymenttype = 'Banking';



Tuple count = 1

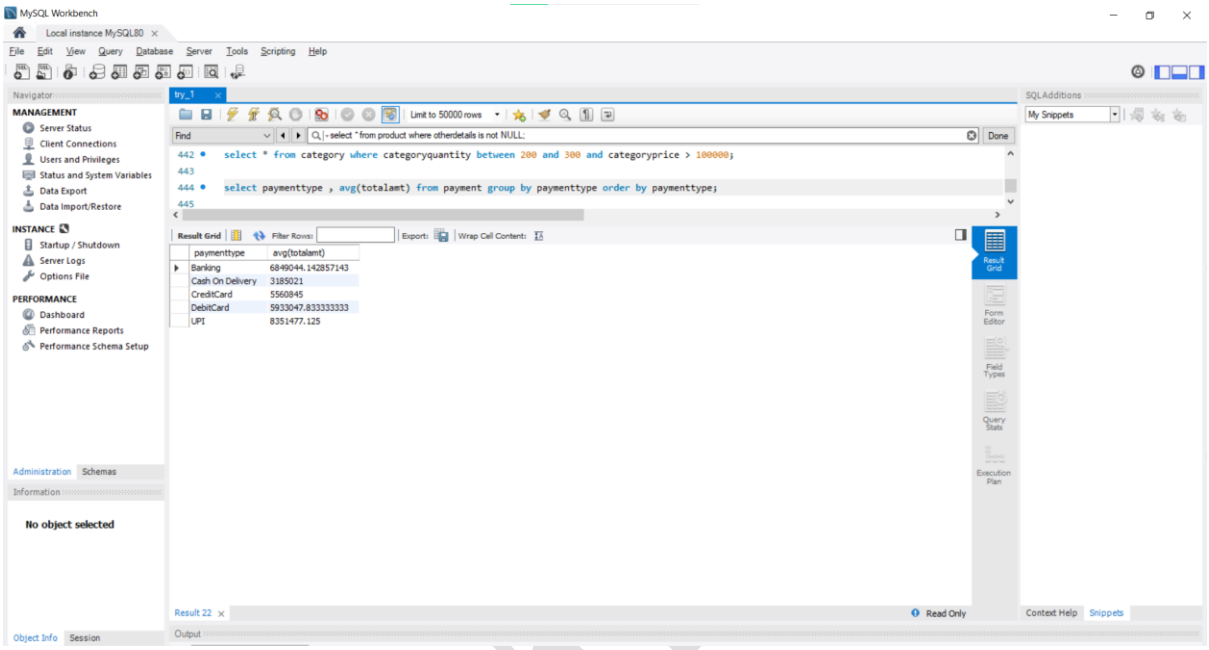
10. Show details of category of product with quantity between 200 to 300 and price more than 200000.
    select * from category where categoryquantity
    between 200 and 300 and categoryprice > 100000;



Tuple count = 13

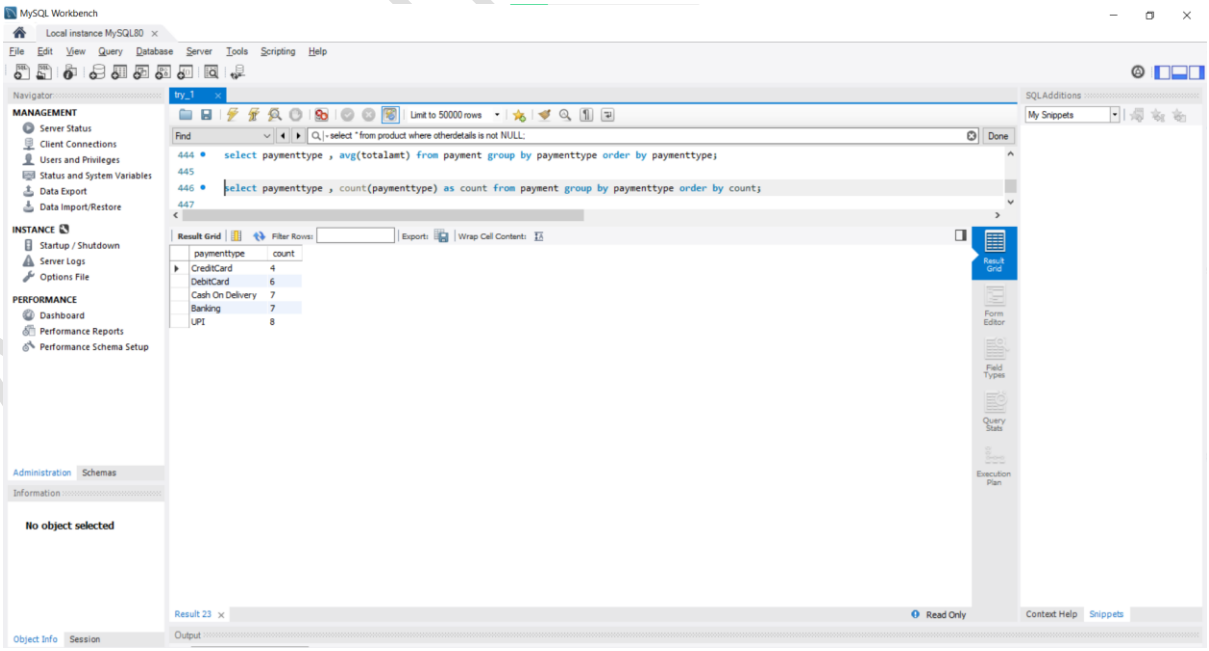
Select … from … orderby/groupby queries

11. Show the different ways of payment with average amount of payment in that type.
    select paymenttype , avg(totalamt) from payment
    group by paymenttype
    order by paymenttype;



Tuple count = 5

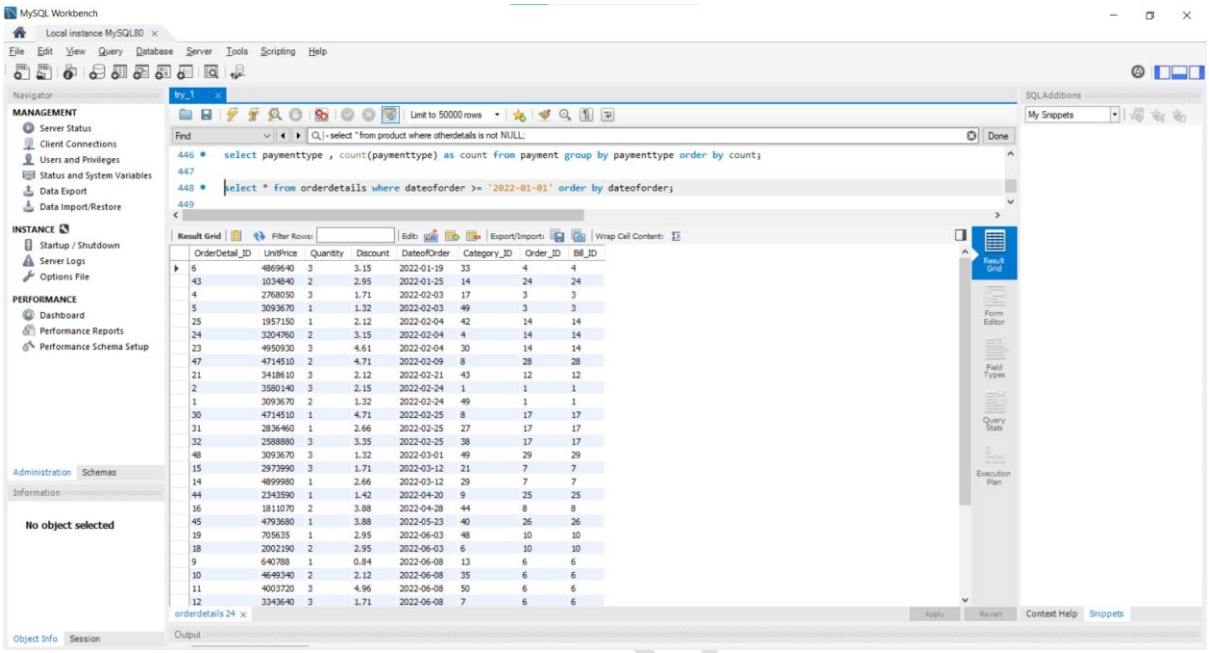
12. Show different type of payment and count of it sorted from lower to higher count.
    select paymenttype , count(paymenttype) as count from payment
    group by paymenttype order by count;



Tuple count = 5

13. Show orderdetails of orders made in 2022 year in sorted manner.

```
select * from orderdetails
where dateoforder >= '2022-01-01'
order by dateoforder;
```

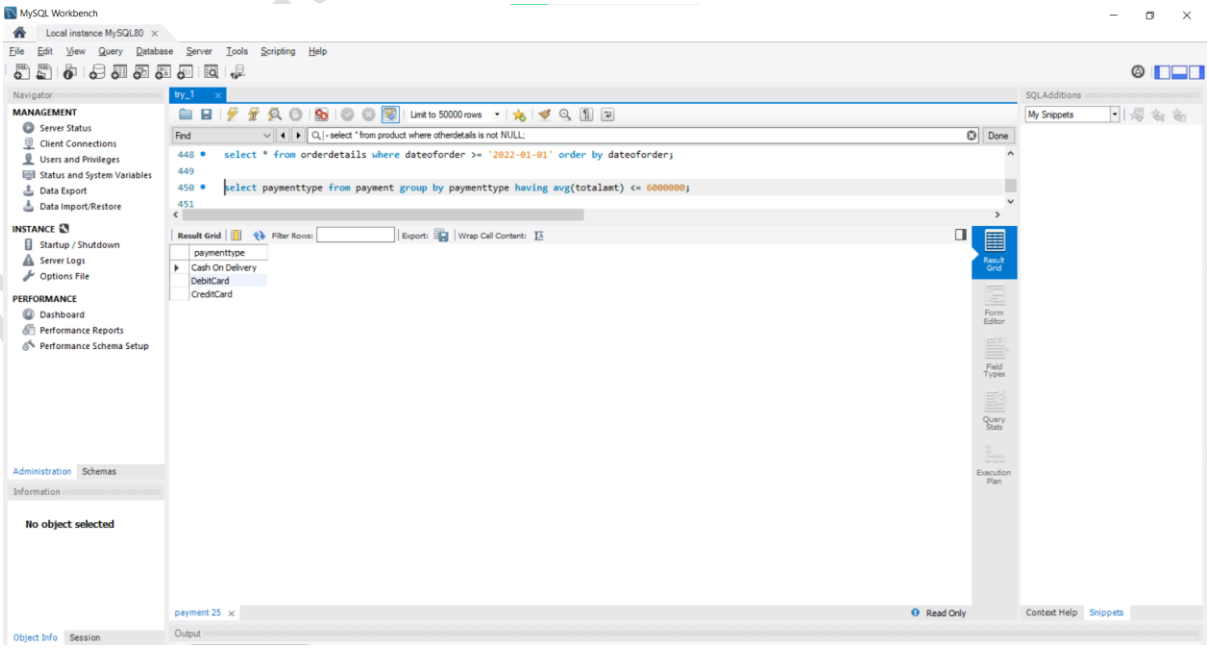


Tuple count = 34

Select … from … where … like queries

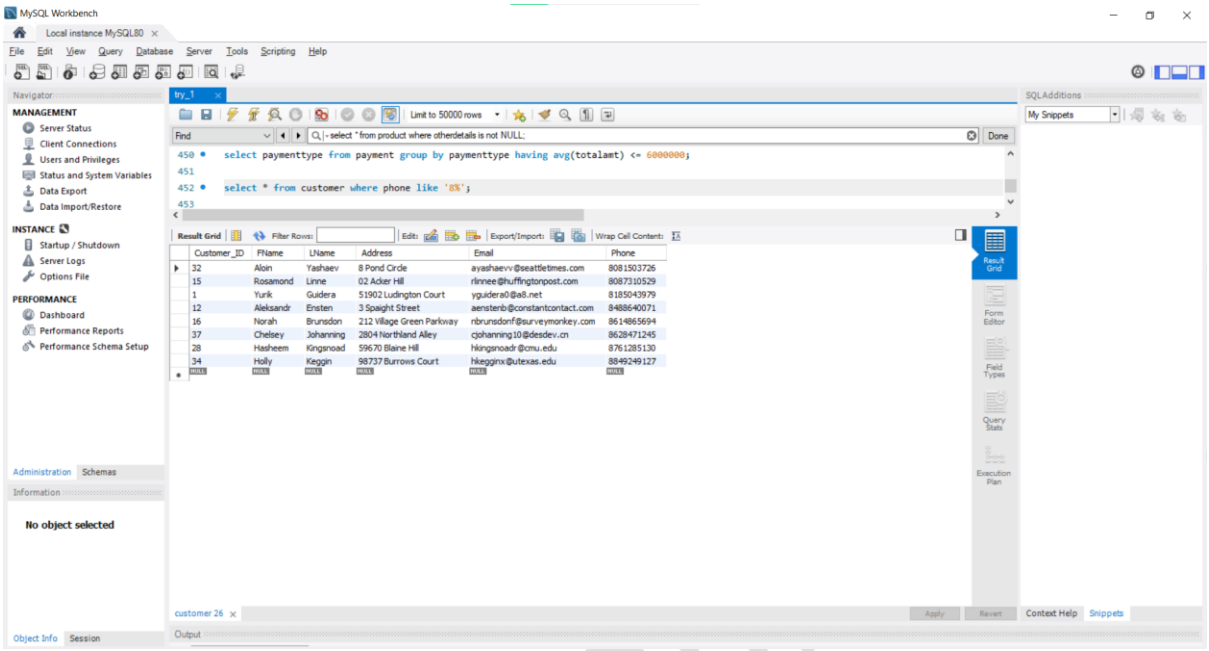
14. Show paymentypes with average amount paid less than 6000000.

```
select paymenttype from payment
group by paymenttype
having avg(totalamt) <= 6000000;
```



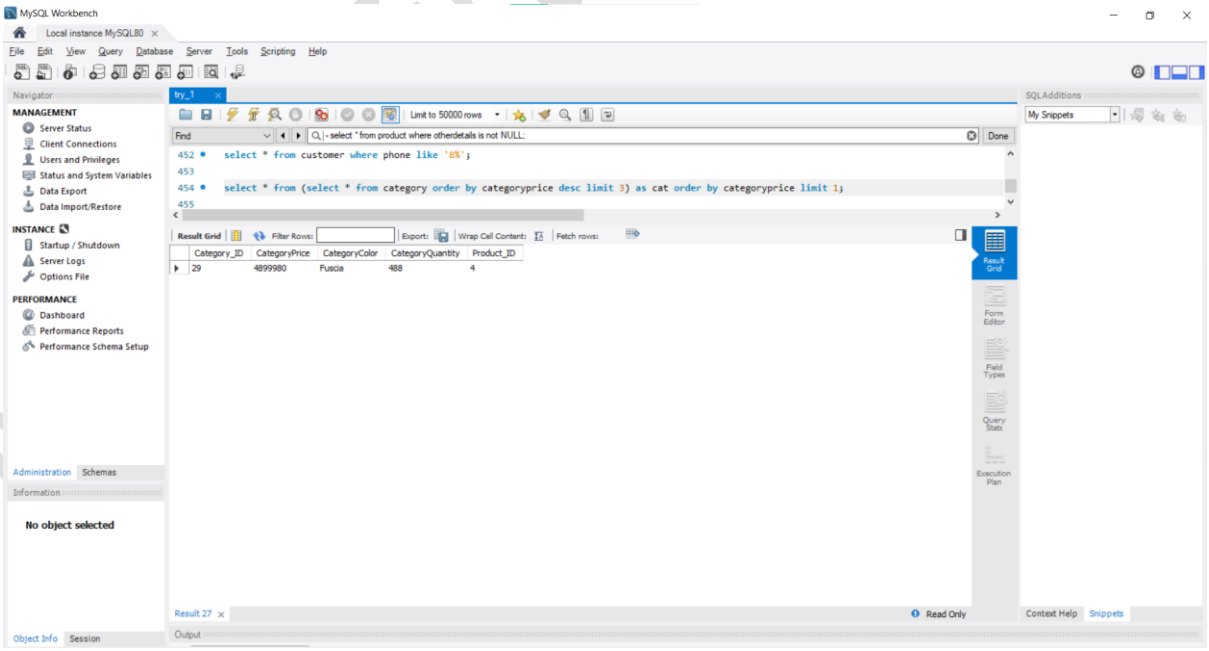
Tuple Count = 3

15. Show customer details of thos customers whose number start with 8.
    select * from customer where phone like '8%';



Tuple count = 8

16. Show third most costly product category.
    select * from
    (select * from category order by categoryprice desc limit 3) as cat
    order by categoryprice limit 1;

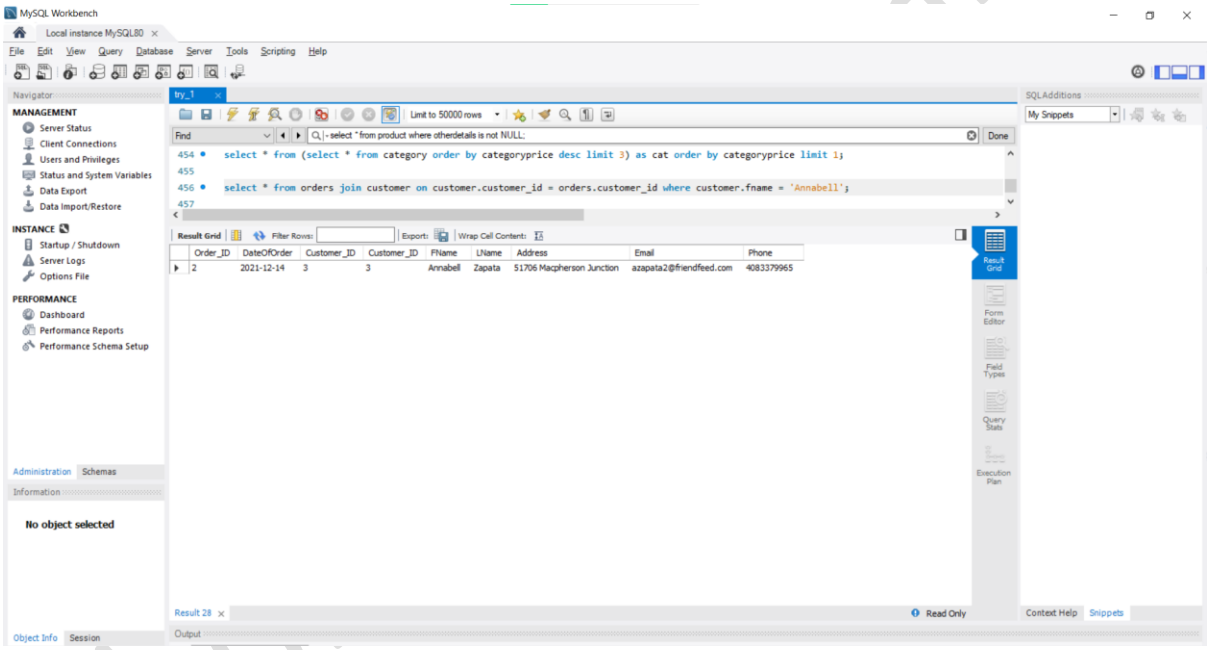


Tuple count = 1

Select … from … join … where queries

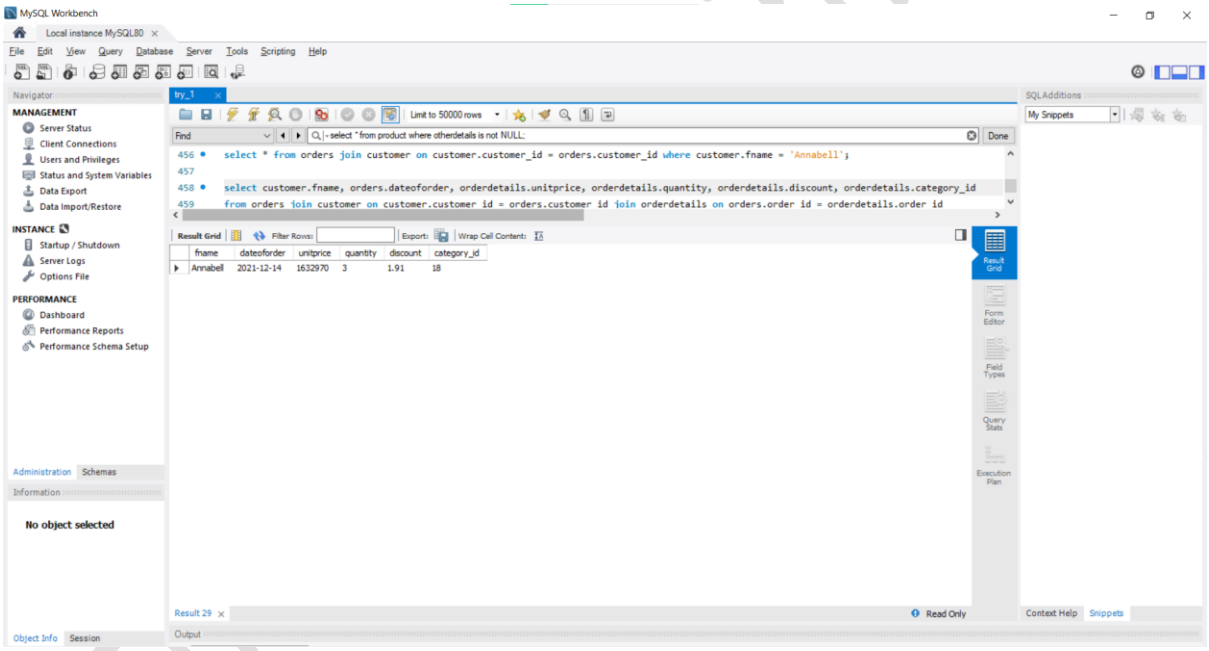
17. Show all the orders made by customer Annabell as first name.

```
select * from orders
join customer on customer.customer_id = orders.customer_id
where fname = 'Annabell';
```



Tuple count = 1

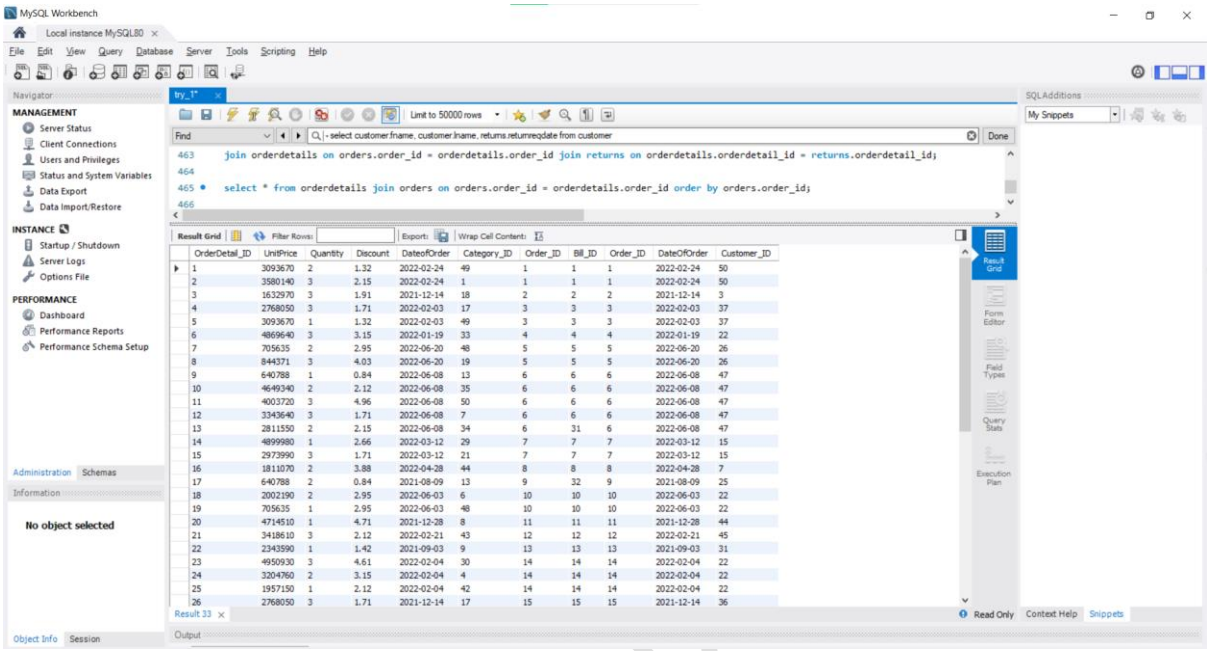
18. Show order history of Annabell with order details like name, dateoforder, category_id, price, discount and quantity.
    select customer.fname, orders.dateoforder, orderdetails.unitprice, orderdetails.quantity, orderdetails.discount, orderdetails.category_id
    from orders join customer on customer.customer_id = orders.customer_id
    join orderdetails on orders.order_id = orderdetails.order_id
    where customer.fname = 'Annabell';



Tuple count = 1

19. Show orders and their orderdetails in sorted order of order_id.

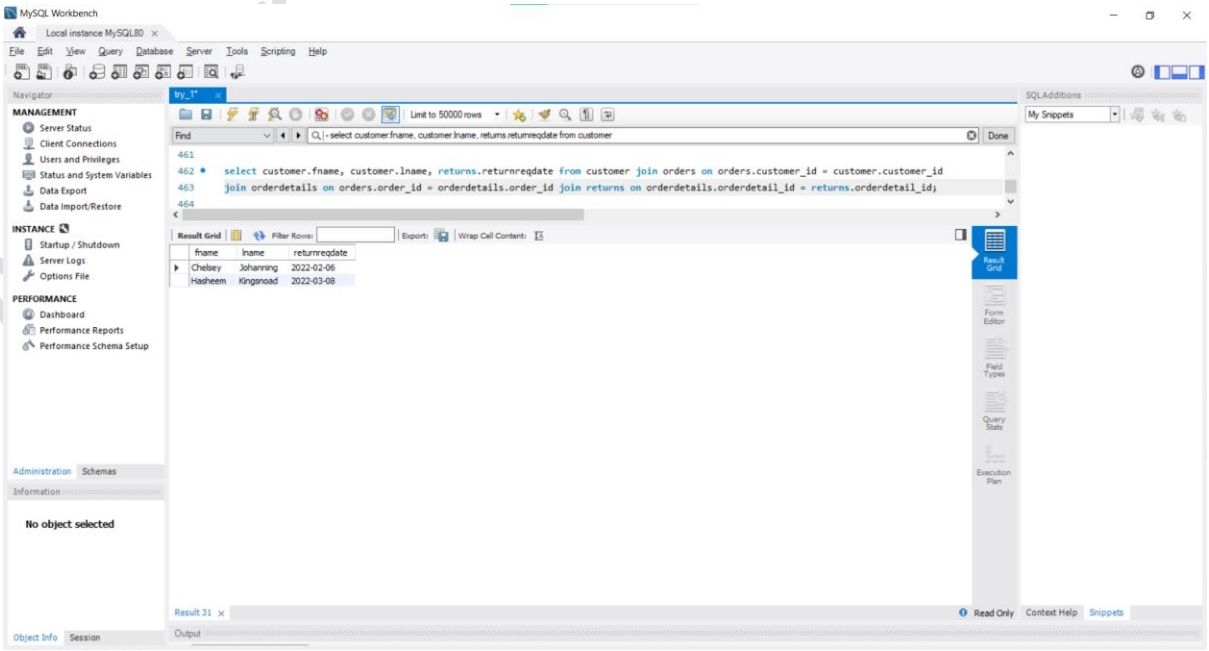
```
select * from orderdetails
join orders on orders.order_id = orderdetails.order_id
order by orders.order_id;
```



Tuple Count = 50

20. Show full name and date of return request for the customers who have returned their orders.

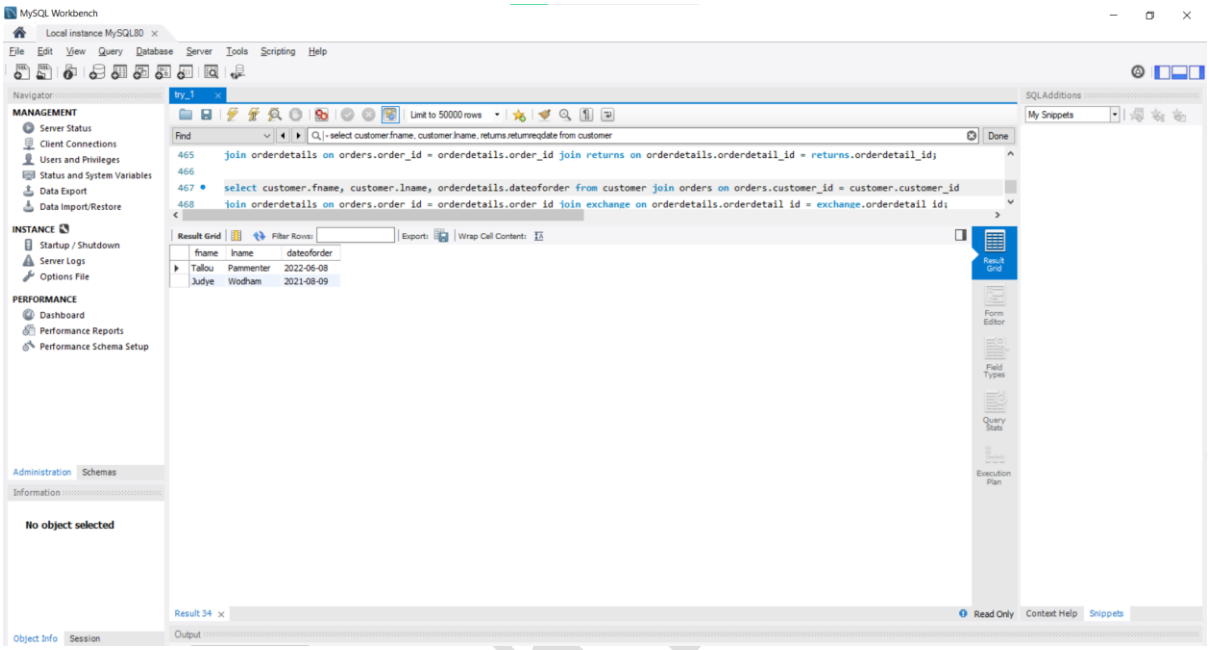
```
select customer.fname, customer.lname, returns.returnreqdate from customer
join orders on orders.customer_id = customer.customer_id
join orderdetails on orders.order_id = orderdetails.order_id
join returns on orderdetails.orderdetail_id = returns.orderdetail_id;
```



Tuple count = 2

21. Show name and date of exchange for the customers who have exchanged order.

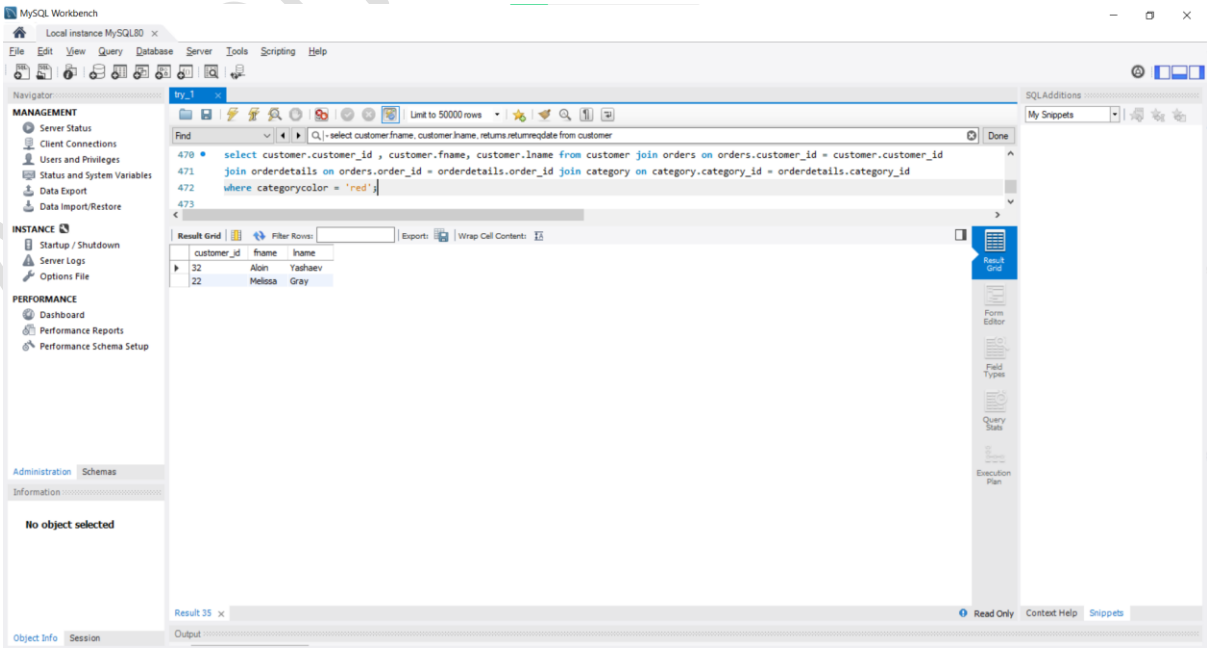
```
select customer.fname, customer.lname, orderdetails.dateoforder from customer
join orders on orders.customer_id = customer.customer_id
join orderdetails on orders.order_id = orderdetails.order_id
join exchange on orderdetails.orderdetail_id = exchange.orderdetail_id;
```



Tuple count = 2

22. Show the name and id of customers who have bought any product of red color.

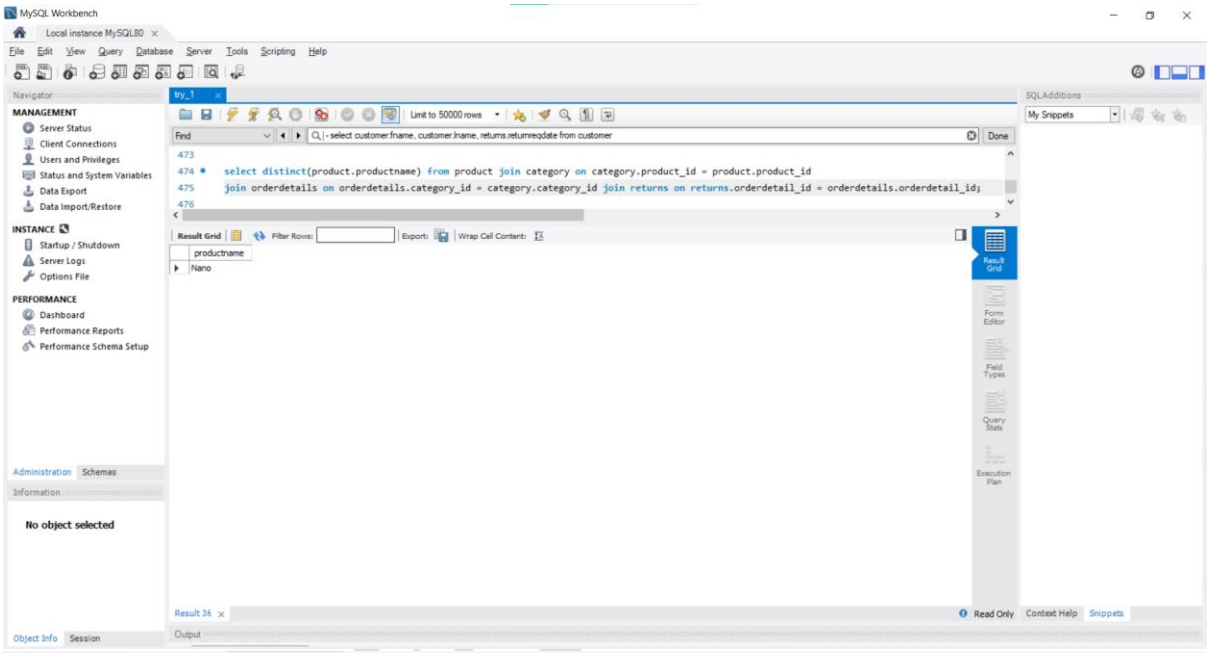
```
select customer.customer_id , customer.fname, customer.lname from customer
join orders on orders.customer_id = customer.customer_id
join orderdetails on orders.order_id = orderdetails.order_id
join category on category.category_id = orderdetails.category_id
where categorycolor = 'red';
```



Tuple count = 2

23. Show product name which has been returned by customer.

```
select distinct(product.productname) from product
join category on category.product_id = product.product_id
join orderdetails on orderdetails.category_id = category.category_id
join returns on returns.orderdetail_id = orderdetails.orderdetail_id;
```

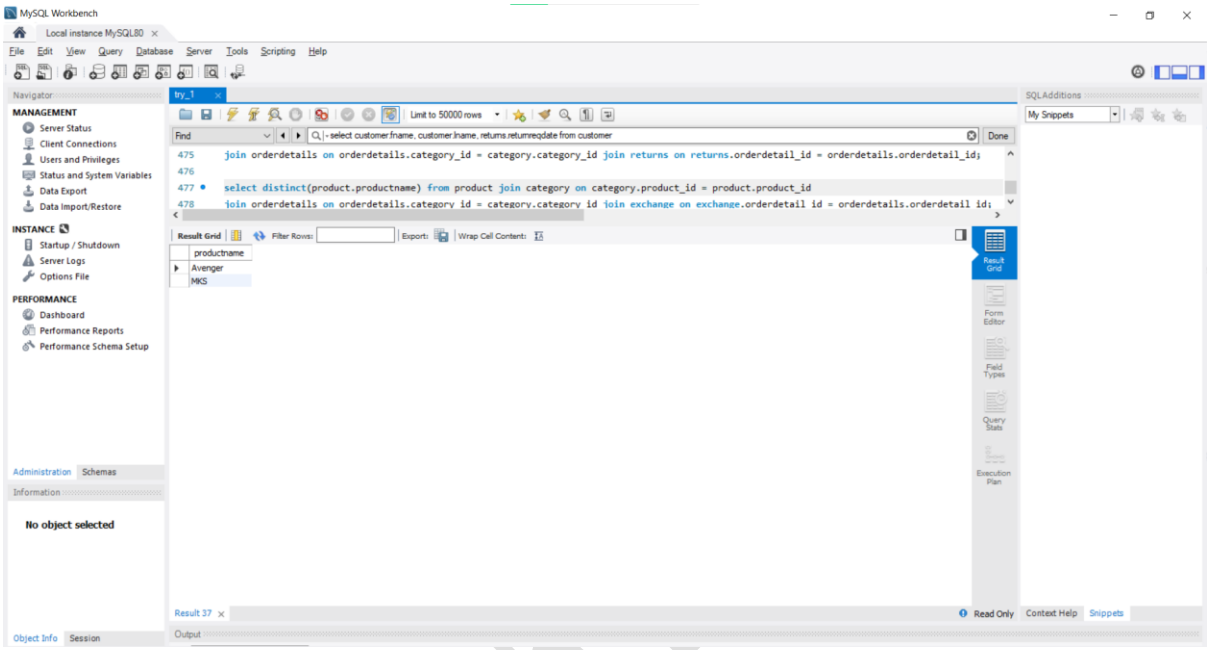


productname
Nano

Tuple Count = 1

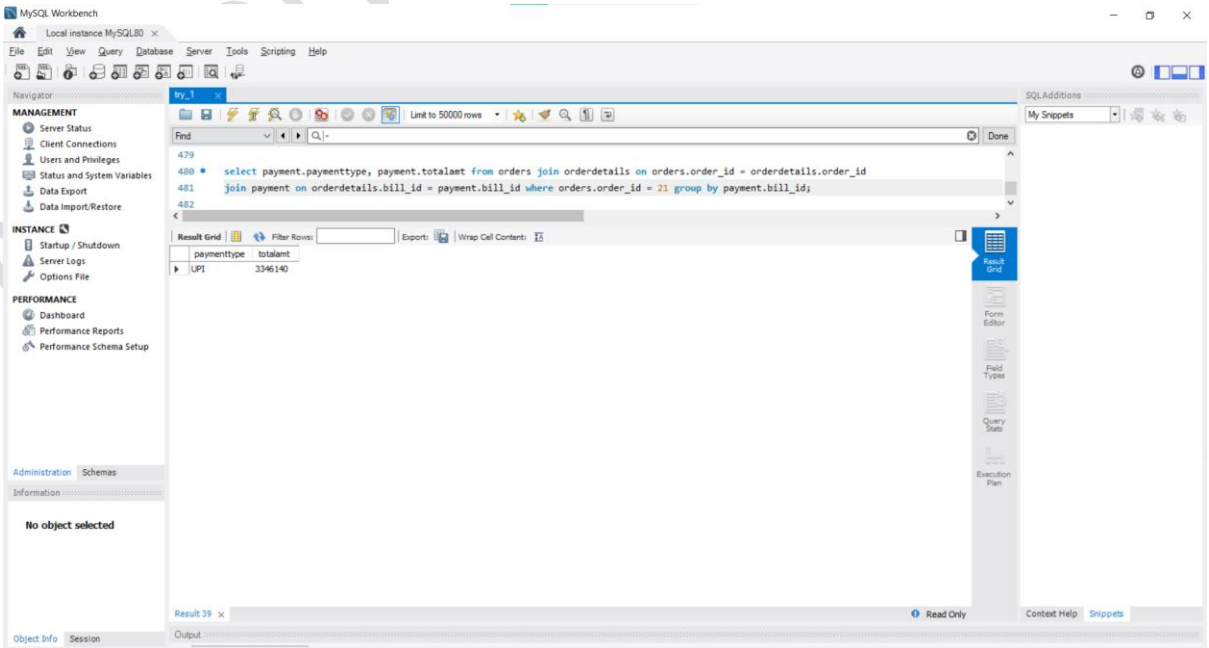
This result is quite an amazing observation. Though data was created in random orders (Though some logic part was added) it can be seen that both the returned orders are on product name ‘Nano’.

24. Show product name which has been exchanged by customer.
    select distinct(product.productname) from product
    join category on category.product_id = product.product_id
    join orderdetails on orderdetails.category_id = category.category_id
    join exchange on exchange.orderdetail_id = orderdetails.orderdetail_id;



Tuple Count = 2

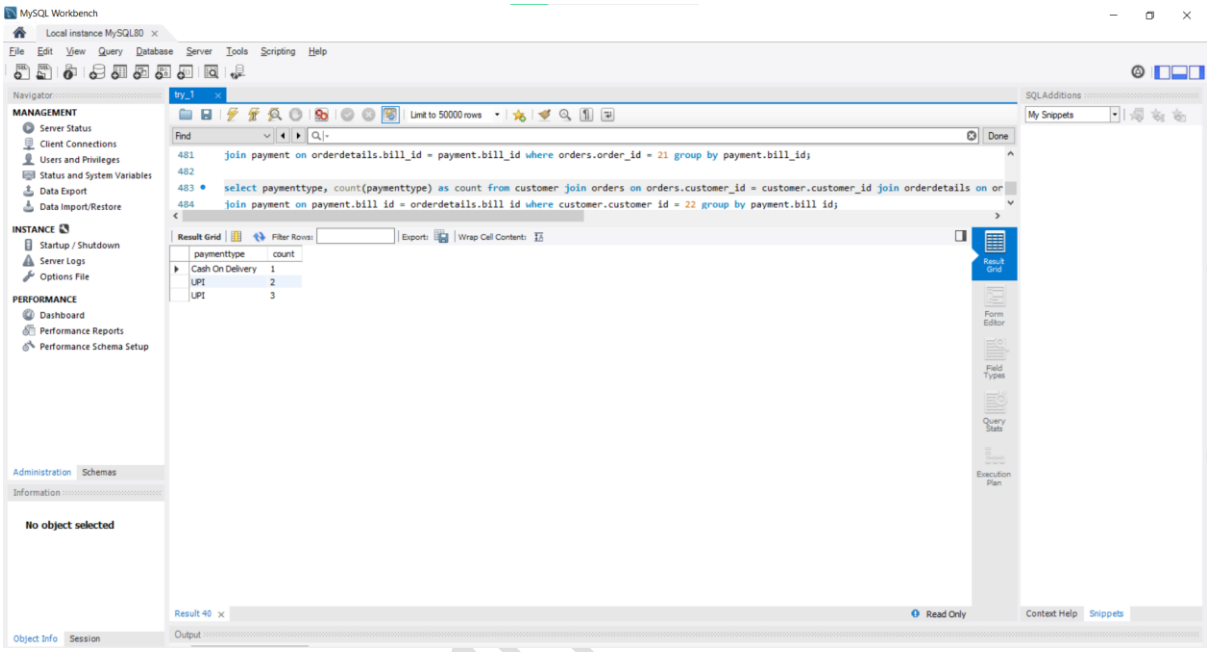
25. Show the paymenttype and total amt paid for order number 21.
    select payment.paymenttype, payment.totalamt from orders
    join orderdetails on orders.order_id = orderdetails.order_id
    join payment on orderdetails.bill_id = payment.bill_id
    where orders.order_id = 21 group by payment.bill_id;



Tuple Count = 1

26. Show different type of payment with their count made by customer with id 22.

```
select paymenttype, count(paymenttype) as count from customer
join orders on orders.customer_id = customer.customer_id
join orderdetails on orders.order_id = orderdetails.order_id
join payment on payment.bill_id = orderdetails.bill_id
where customer.customer_id = 22
group by payment.bill_id;
```

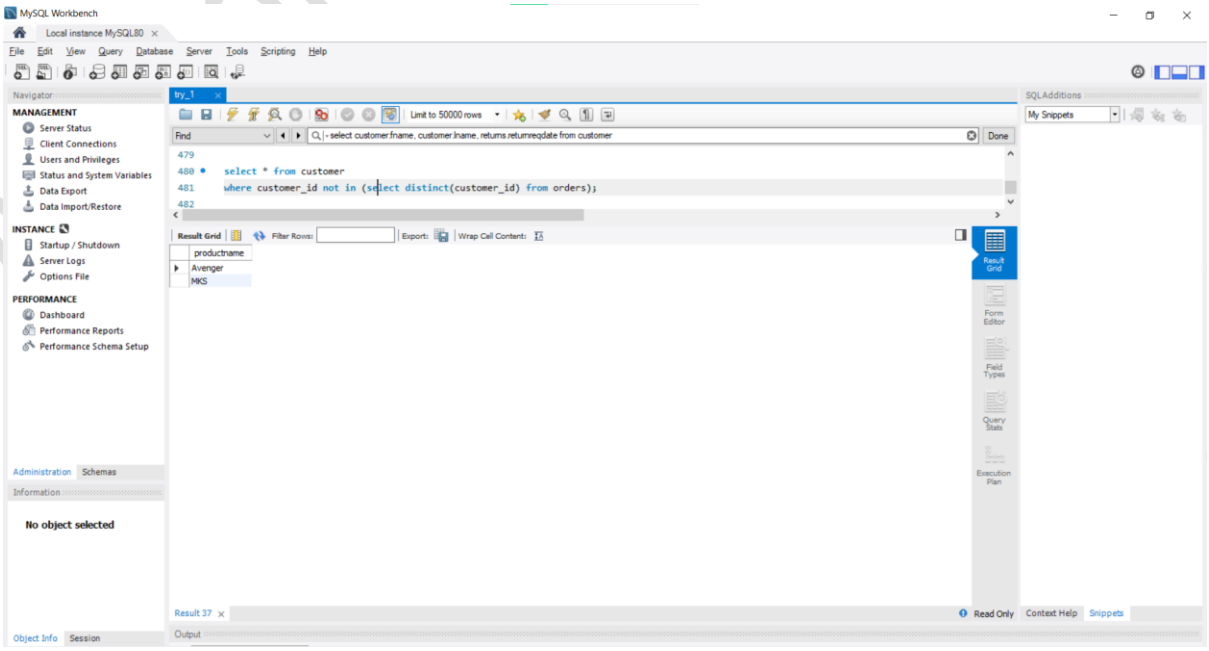


Tuple Count = 2

Select … from … where … not in queries

27. Show details of customers who have not ordered anything.

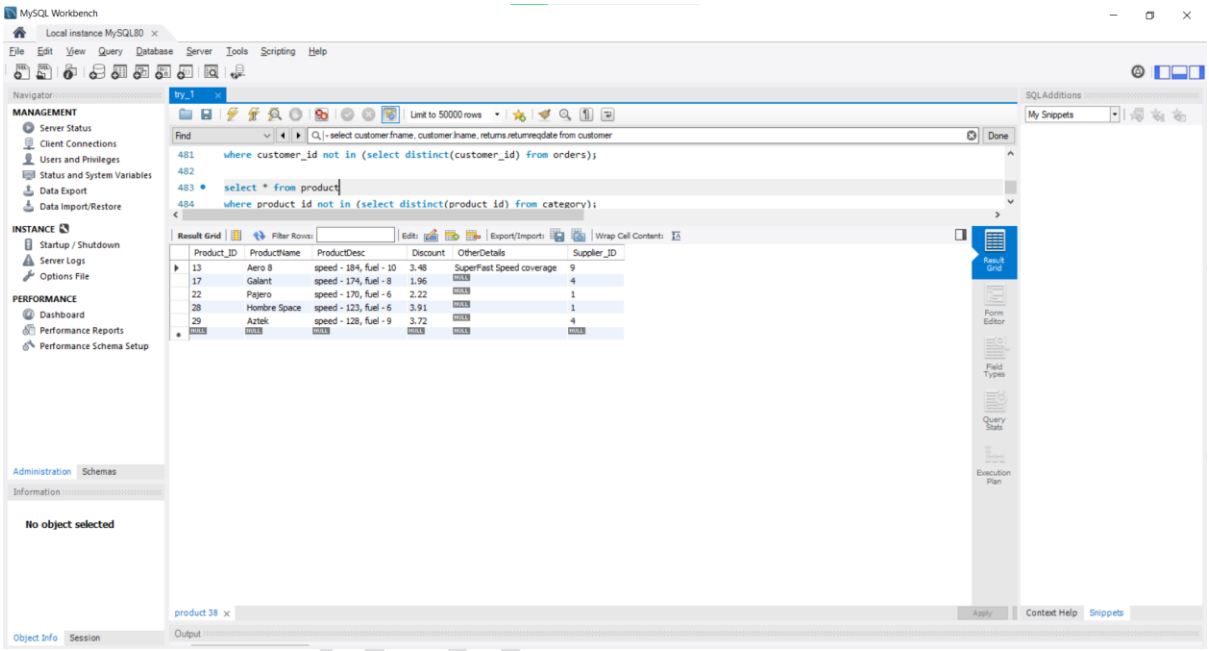
```
select * from customer
where customer_id not in (select distinct(customer_id) from orders);
```



Tuple Count = 26

28. Show details of product which has no categories under it.

```
select * from product
where product_id not in (select distinct(product_id) from category);
```

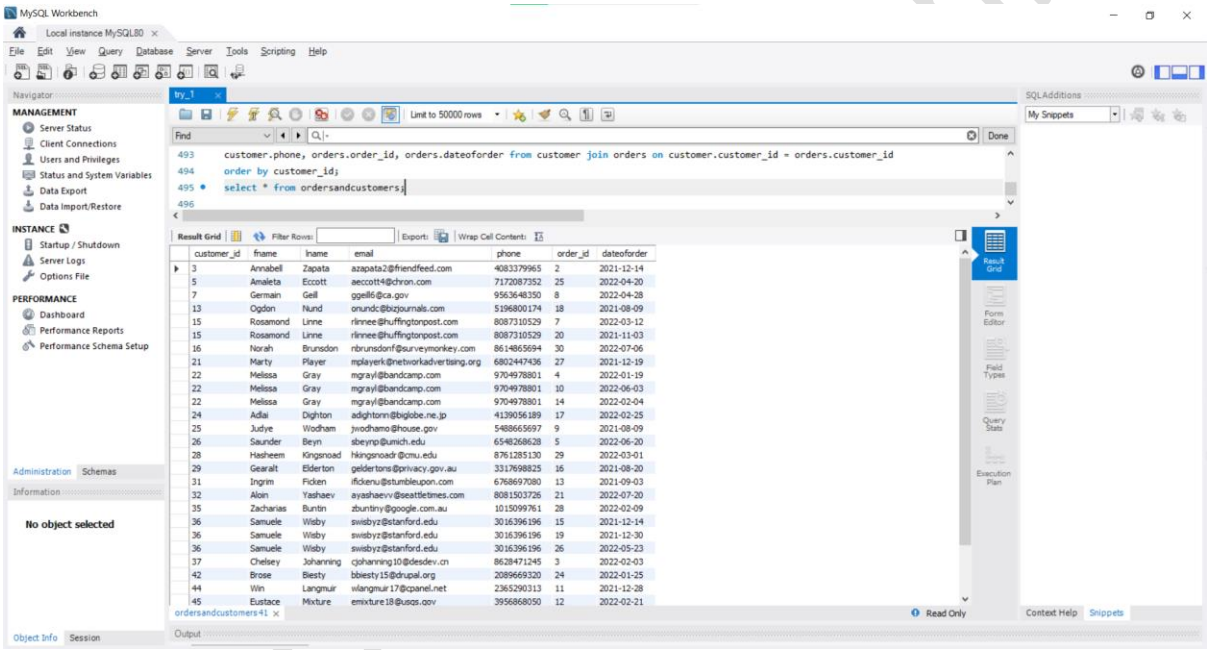


Tuple Count = 5

View Queries

29. Create and show view for customers and their orders made sorted with customer_id.

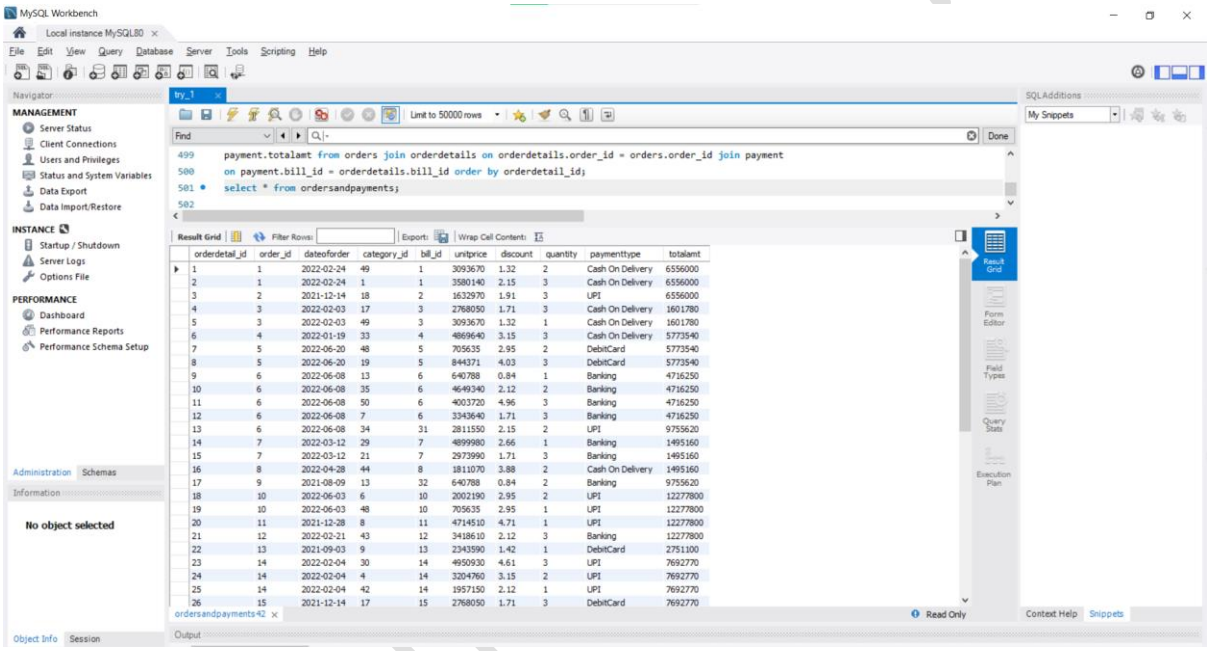
```
create or replace view OrdersAndCustomers as
select customer.customer_id, customer.fname, customer.lname, customer.email,
customer.phone, orders.order_id, orders.dateoforder from customer
join orders on customer.customer_id = orders.customer_id
order by customer_id;
select * from ordersandcustomers;
```



Tuple Count = 50

30. Create and show view orders, their orderdetails and payment method with required features.

```
create or replace view OrdersAndPayments as select orderdetails.orderdetail_id,
orders.order_id, orderdetails.dateoforder, orderdetails.category_id, orderdetails.bill_id,
orderdetails.unitprice, orderdetails.discount, orderdetails.quantity, payment.paymenttype,
payment.totalamt from orders
join orderdetails on orderdetails.order_id = orders.order_id
join payment on payment.bill_id = orderdetails.bill_id
order by orderdetail_id;
select * from ordersandpayments;
```



Tuple Count = 50