

Table of Contents

Submission Details.....	3
Introduction.....	4
Objective.....	4
Key Deliverables.....	5
TASKS	6
Question 1:.....	6
Introduction.....	6
The Entity Relationship Diagram	6
Detailed Explanation of The Relationship Between the Entities in The ER Diagram	7
Database Creation	8
Creation Of Tables	8
Conclusion.....	17
Question 2:.....	18
Question 3:.....	25
Comments Before Every Task and Screenshot of ER Diagram from MySQL. ...	25
Question 4	27
Deleting Useless Columns in Productlines Table	27
Question 5:.....	28
Verifying All Inserting Using the Select Statement	28
Question 6:.....	34
Finding The Highest and The Lowest Amount.	34
Screenshot Of Highest Amount Value and SQL Code Used:.....	34
Screenshot Of Lowest Amount Value and SQL Code Used:.....	35
Question 7	36
Finding The Unique Count of Customername from Customers:	36
Question 8	37
Creating View from Customers and Payments	37
Question 9:.....	38
Creating A Stored Procedure	38
Question 10	39
Creating A Function to Get Customer's CreditLimit.....	39

Question 11	40
Creating Trigger to Store Transaction Record	40
Question 12:.....	41
Creating A Trigger to Display Customer Number	41
Question 13	42
Creating Users, Roles and Logins.....	42
Question 14	44
Scheduled Job Which Backups and Schedule It.	44
Create a Backup Script:	44
Schedule the Backup Script:	45
Question 15	47
Steps Involve in Opening Activity Monitor in My SQL	47
Question 16	48
Migrate The Following SQL Server Workload to Azure.	48
Conclusion.....	48

Submission Details

Project Name: Microsoft SQL Server Certification Project II

Tools Used: MYSQL Workbench, Draw io

Internship Program: Data Science and Machine Learning (DSML)

Submitted By: Abegyah Matthew

Instructor: Miss Anu

Date: February 10, 2025

Introduction

In today's data-driven world, efficient management of information is critical for businesses to operate effectively and make informed decisions. HerculesMotoCorp, a prominent retailer and garage handler in the United States, deals with a wide range of automobiles, including custom-made vehicles. To streamline their operations, ensure compliance with federal laws, and provide stakeholders with easy access to data, HerculesMotoCorp requires a robust and well-structured database system.

This project focuses on designing and implementing a comprehensive database named **MotorsCertification** for **HerculesMotoCorp**. The database will store critical information about customers, employees, orders, products, payments, and more. The goal is to create a system that not only simplifies data management for **HerculesMotoCorp** employees but also provides shareholders and other stakeholders with clear and accessible insights into the company's operations.

Objective

The primary objectives of this project are:

Design a Relational Database:

- Create an Entity-Relationship (ER) model that represents the structure of the database, including tables, attributes, and relationships.
- Define primary keys, foreign keys, and indexes to ensure data integrity and optimize query performance.

Implement the Database:

- Create the database and all necessary tables using SQL.
- Populate the tables with sample data provided in the dataset.

Perform Data Operations:

- Insert, update, and delete records as required.
- Write SQL queries to retrieve and analyse data.

Enhance Functionality:

- Create views, stored procedures, and functions to simplify complex queries and improve usability.
- Implement triggers to automate tasks such as logging transactions or enforcing business rules.

Ensure Security and Accessibility:

- Create users, roles, and logins with appropriate permissions to ensure data security.
- Grant access to different roles (Admin, HR, Employee) based on their responsibilities.

Optimize and Maintain the Database:

- Schedule regular backups to prevent data loss.
- Monitor database performance using tools like SQL Server Activity Monitor.

Migrate to the Cloud:

- Explore the possibility of migrating the database to Azure for scalability and flexibility.

Document the Process:

- Provide detailed explanations, comments, and screenshots for each task.
- Include the final ER diagram, SQL scripts, and backup file in the project submission.

Key Deliverables

- **ER Diagram:** A visual representation of the database structure, including entities, attributes, and relationships.
- **SQL Scripts:** SQL files containing all the queries used to create tables, insert data, and perform operations.
- **Screenshots:** Screenshots of the ER diagram, query results, and other relevant outputs.
- **Backup File:** A back-up file of the final database for submission.
- **Documentation:** A detailed report explaining the project, including the introduction, objectives, and step-by-step implementation.

TASKS

Question 1:

Name the database as **MotorsCertification**. Design an ER model based on the following parameters: **Note:** Build an ER diagram with proper entities, relationship etc.

Solution:

Introduction

For this question, I will design an Entity-Relationship (ER) model for the database, which will help visualize how different entities like **orderdetails**, **customers**, **orders**, **products**, **offices**, **payments**, **employees**, and **productlines** are connected. Once the ER diagram is complete, I will move on to creating the database **MotorsCertification** and the actual tables in the database. These tables will include all the necessary attributes, along with primary keys, foreign keys, and indexes to ensure the database is well-structured and efficient.

The Entity Relationship Diagram

The below image is an Entity relationship Diagram (ER Diagram), a visual representation of the structure of all the entities and their relationship within the database.

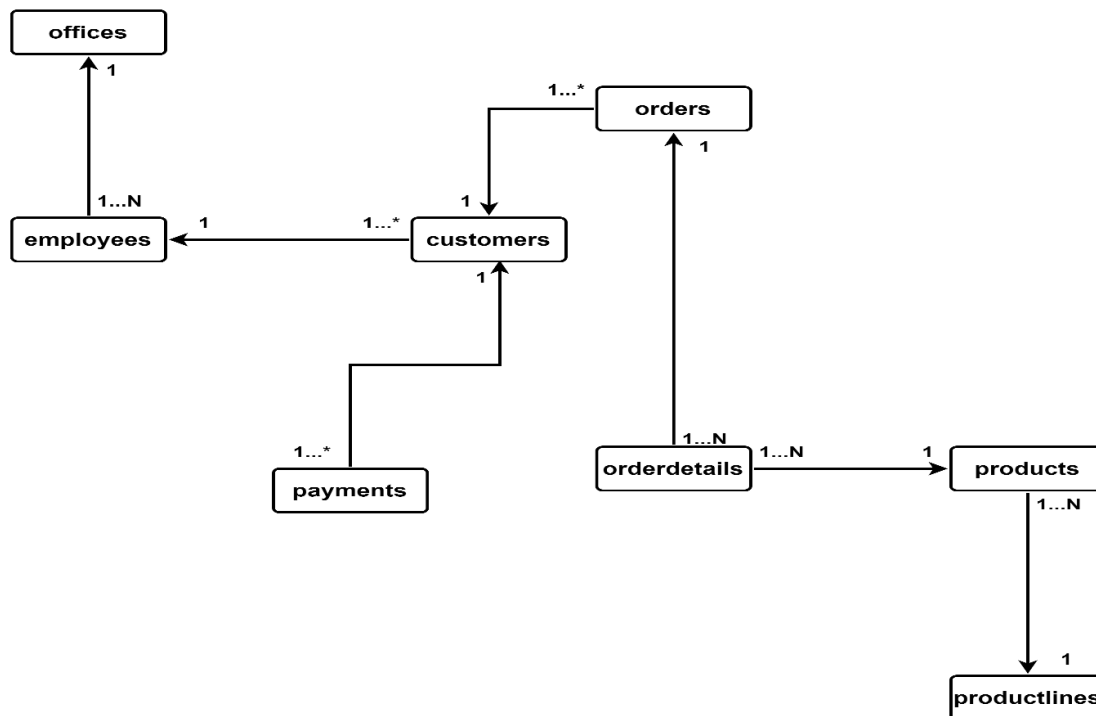


Fig. 0.1

Detailed Explanation of The Relationship Between the Entities in The ER Diagram

offices → **employees**

- Each office can have multiple employees, but each employee belongs to only one office.
- **Relationship:** One-to-Many.

employees → **customers**

- Each employee can be a sales representative for multiple customers, but each customer is assigned to only one sales representative.
- **Relationship:** One-to-Many.

customers → **orders**

- Each customer can place multiple orders, but each order is associated with only one customer.
- **Relationship:** One-to-Many.

orders → **orderDetails**

- Each order can have multiple order details (products), but each order detail belongs to only one order.

- **Relationship:** One-to-Many.

products → orderDetails

- Each product can appear in multiple order details, but each order detail refers to only one product.
- **Relationship:** One-to-Many.

productlines → products

- Each product line can have multiple products, but each product belongs to only one product line.
- **Relationship:** One-to-Many.

customers → payments

- Each customer can make multiple payments, but each payment is associated with only one customer.
- **Relationship:** One-to-Many.

Database Creation

To begin creating all the needed tables to hold the data for each entity, there is the need to create database with the name **MotorsCertification**, which will serve as the foundation for storing all data related to HerculesMotoCorp's business operations. In doing so, the **CREATE DATABASE IF NOT EXISTS** syntax was used to check if there is an existing database with the name **MotorsCertification** and create it if it does not exist. The **USE** statement was used to select the **MotorsCertification** database to be used. This screenshot below shows the created database and the query used in MySQL environment.

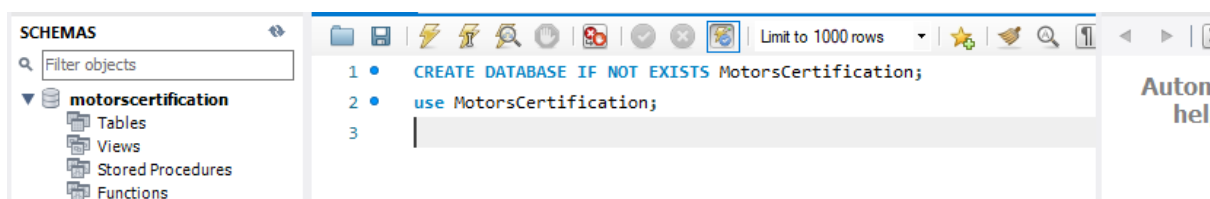


Fig. 0.2

Creation Of Tables

Below are table creation requirements and the SQL syntaxes that was used to create all the stated tables in the ER Diagram based on the company's requirements:

Orderdetails Table:

Requirements:

Design a table/database object named **orderdetails** with the following attributes/columns:

orderNumber int(), Primary Key

productCode varchar()

quantityOrdered int()

priceEach float

orderLineNumber smallint()

Foreign Key: orders (orderNumber → orderNumber) and products (productCode → productCode)

Index 1: PRIMARY, Type: BTREE, Unique Yes, Visible No, Columns orderNumber

Index 2: productCode, Type: BTREE, Unique: No, Visible: No, Columns productCode

Solution:

The below screenshot contains SQL code that creates an **orderdetails** table to store information about individual items within an order. The table includes columns for the orderNumber (a unique identifier for each order), productCode (the identifier for the product), quantityOrdered (the number of units ordered), priceEach (the price per unit of the product), and orderLineNumber (the line number for the item in the order). The orderNumber and productCode columns are linked to the orders and products tables, respectively, through foreign key constraints. Additionally, an index is created on the productCode column to improve query performance when searching by product.

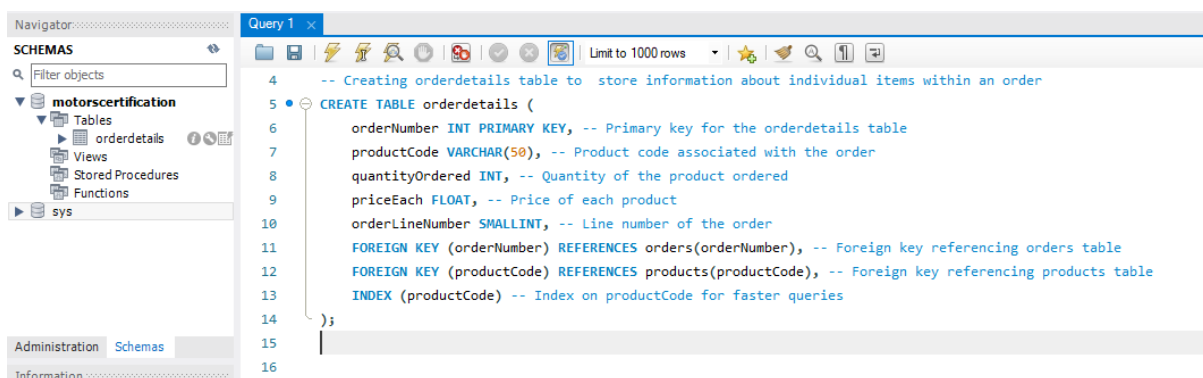


Fig. 0.3

Customers Table

Requirements

Design a table/database object named **customers** with the following attributes/columns:

customerNumber int(11) PK

customerName varchar(50)

contactLastName varchar(50)

contactFirstName varchar(50)

phone varchar(50)

addressLine1 varchar(50)

addressLine2 varchar(50)

city varchar(50)

state varchar(50)

postalCode varchar(15)

country varchar(50)

salesRepEmployeeNumber int(11)

creditLimit float

Foreign Key: employees (salesRepEmployeeNumber → employeeNumber).

Index: PRIMARY, Type: BTREE, Unique: Yes, Visible: No, Columns:

customerNumber

Solution:

The below screenshot shows SQL codes that create customers' table to store customer-related information. It includes columns such as **customerNumber** (a unique identifier and primary key), **customerName**, contact details (**contactLastName**, **contactFirstName**, and **phone**), address details (**addressLine1**, **addressLine2**, **city**, **state**, **postalCode**, and **country**), and financial details (**creditLimit**). Additionally, the **salesRepEmployeeNumber** column serves as a foreign key referencing the employees table, linking each customer to a specific sales representative. This structure ensures data integrity and facilitates efficient customer management within the database.

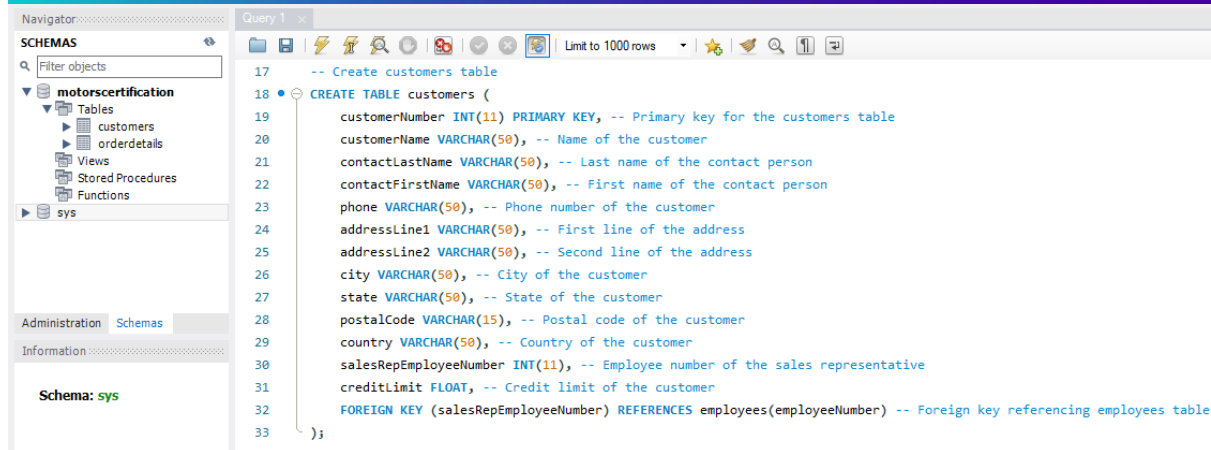


Fig. 0.4

Employees Table

Requirements

Design a table/database object named employees with the following attributes/columns:

employeeNumber int() PK

lastName varchar()

firstName varchar()

extension varchar()

email varchar()

officeCode varchar()

reportsTo int()

jobTitle varchar()

Foreign Key: employees (reportsTo → employeeNumber) and offices (officeCode → officeCode)

Index 1: PRIMARY, Type: BTREE, Unique: Yes, Visible No, Columns: employeeNumber

Index 2: reportsTo, Type: BTREE, Unique: No, Visible: No, Columns reportsTo

Index 3: officeCode, Type: BTREE, Unique: No, Visible: No, Columns officeCode

Solution:

This SQL codes in the screenshot below create **employees table** to store employee details. It includes columns for **employeeNumber** (a unique identifier and primary key), **lastName**, **firstName**, **extension**, **email**, and **jobTitle**. The officeCode column links each employee to their office, while reportsTo is a self-referencing foreign key that establishes a hierarchical relationship by indicating the manager's employeeNumber. The table also includes indexes

on reportsTo and officeCode to optimize query performance, ensuring efficient retrieval of hierarchical and office-related data.

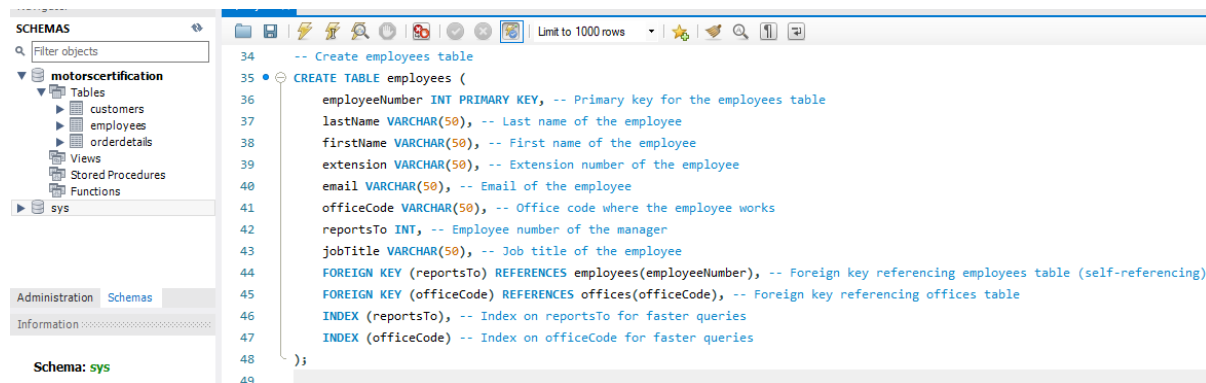


Fig. 0.5

Orders Table

Requirements:

attributes/columns:

orderNumber int() PK

orderDate date

requiredDate date

shippedDate date

status varchar()

comments text

customerNumber int()

Foreign Key: customers (customerNumber → customerNumber).

Index 1: PRIMARY, Type: BTREE, Unique: Yes, Visible: No, Columns orderNumber

Index 2: customerNumber, Type: BTREE, Unique: No, Visible: No, Columns customerNumber

Solution:

This SQL code creates an orders table to store order-related information. It includes orderNumber as the primary key, along with details such as orderDate (when the order was placed), requiredDate (when it is needed), and shippedDate (when it was shipped). The status column tracks the order's progress, while the comments field stores any additional notes. The customerNumber column establishes a relationship with the customers table through a foreign

key, ensuring that each order is linked to a valid customer. An index on customerNumber is also included to optimize query performance when searching for orders by customer.

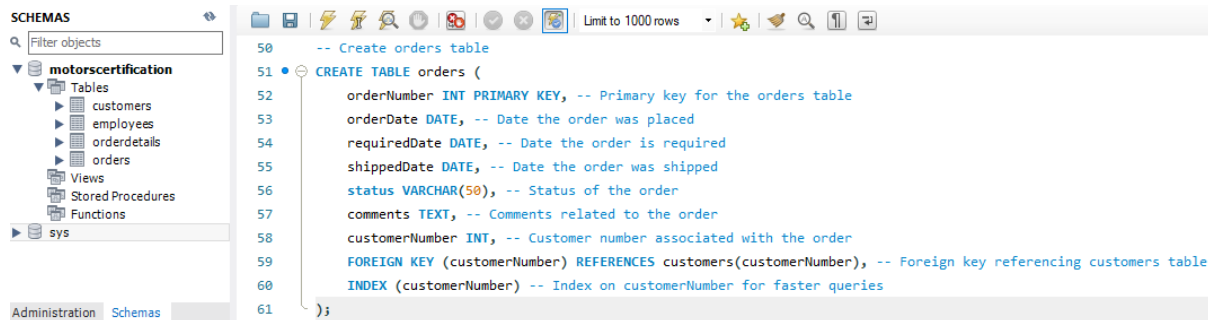


Fig. 0.6

Offices Table

Requirements:

Design a table/database object named **offices** with the following attributes/columns:

officeCode varchar() PK

city varchar()

phone varchar()

addressLine1 varchar()

addressLine2 varchar()

state varchar()

country varchar()

postalCode varchar()

territory varchar()

Index 1: PRIMARY, Type: BTREE, Unique: Yes, Visible No, Columns: officeCode

Solution:

This SQL code creates an offices table to store details about company office locations. The **officeCode** serves as the primary key, uniquely identifying each office. Other columns include **city**, **state**, **country**, and **territory** to specify the office's location, while **addressLine1** and **addressLine2** store detailed address information. The **phone** column records the office's contact number, and **postalCode** ensures proper geographical identification. This table helps manage office locations efficiently and supports relationships with employees assigned to specific offices.

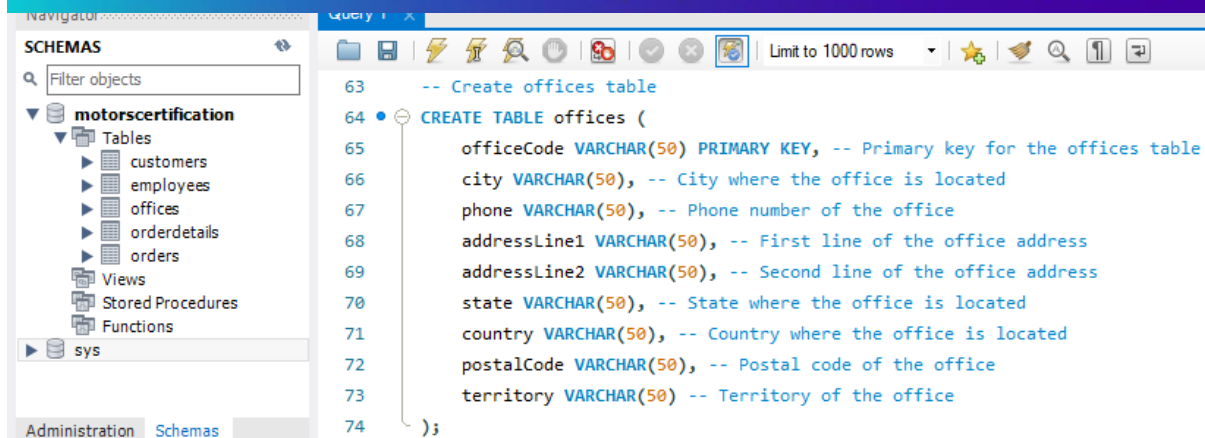


Fig. 0.7

Payments Table

Requirements:

Design a table/database object named payments with the following attributes/columns:

customerNumber int() PK

checkNumber varchar(50)

paymentDate date

amount float

Foreign Key: customers (customerNumber → customerNumber)

Index: PRIMARY, Type: BTREE, Unique: Yes, Visible: No, Columns customerNumber and checkNumber

Solution:

This SQL code creates a **payments table** to store payment transaction details. The table includes **customerNumber**, which links each payment to a customer, and **checkNumber**, which uniquely identifies the payment. The **paymentDate** column records when the payment was made, while the **amount** column stores the payment value. A composite primary key consisting of customerNumber and checkNumber ensures that each payment entry is uniquely identified. Additionally, the customerNumber column is a foreign key referencing the customers table, maintaining data integrity by ensuring payments are linked to valid customers.

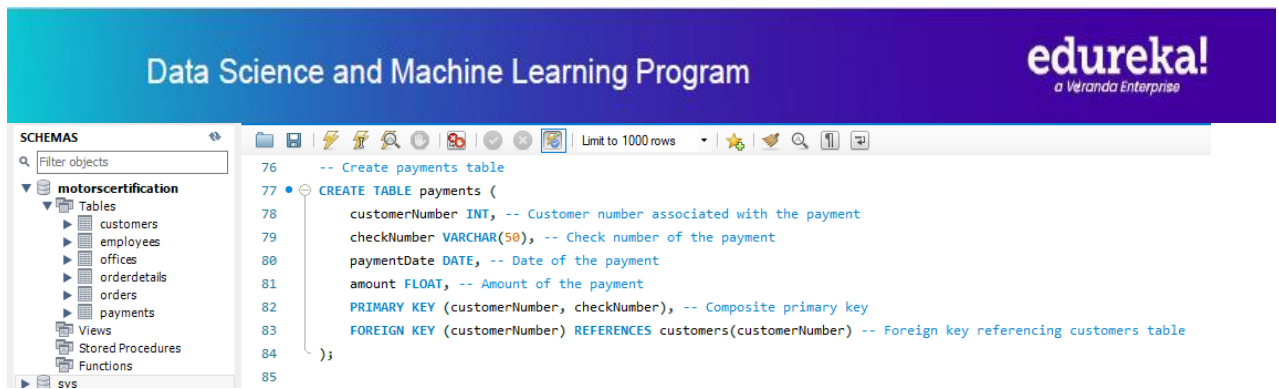


Fig. 0.8

Productlines Table

Requirements

Design a table/database object named productlines with the following attributes/columns:

productLine varchar(50) PK

textDescription varchar(4000)

htmlDescription NULL

image NULL

Solution

This SQL code creates a **productlines table** to categorize products into different product lines. The **productLine** column serves as the primary key, uniquely identifying each product line. The **textDescription** column provides a brief textual description, while the **htmlDescription** allows for more detailed, formatted descriptions in HTML format. The **image column**, stored as a BLOB (Binary Large Object), can hold an image representation of the product line. This table helps organize and describe product categories effectively within the database.

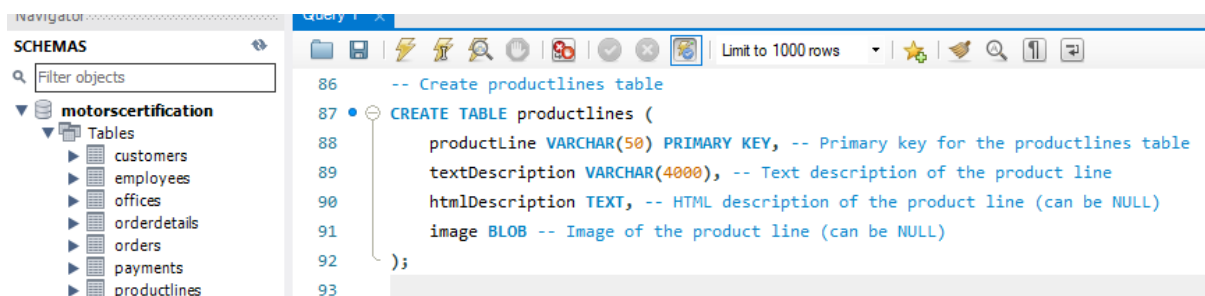


Fig. 0.9

Products Table

Requirements

Design a table/database object named products with the following attributes/columns:

productCode varchar() PK

productName varchar()

productLine varchar()

productScale varchar()

productVendor varchar()

productDescription text()

quantityInStock smallint(6)

buyPrice float

MSRP float

Foreign Key: productlines (productLine → productLine).

Index 1: PRIMARY, Type: BTREE, Unique: Yes, Visible: No, Columns productCode

Index 2: productLine, Type: BTREE, Unique: No, Visible: No, Columns productLine

Solution

This SQL code creates a **products** table to store details about individual products. The **productCode** serves as the primary key, uniquely identifying each product. Other columns include **productName**, **productScale** (indicating size or model scale), **productVendor** (supplier name), and **productDescription**. The **quantityInStock** column tracks available inventory, while **buyPrice** and **MSRP** (Manufacturer's Suggested Retail Price) store product pricing details. The **productLine** column establishes a foreign key relationship with the productlines table, ensuring each product belongs to a valid product category. An index on productLine optimizes queries related to product classification.

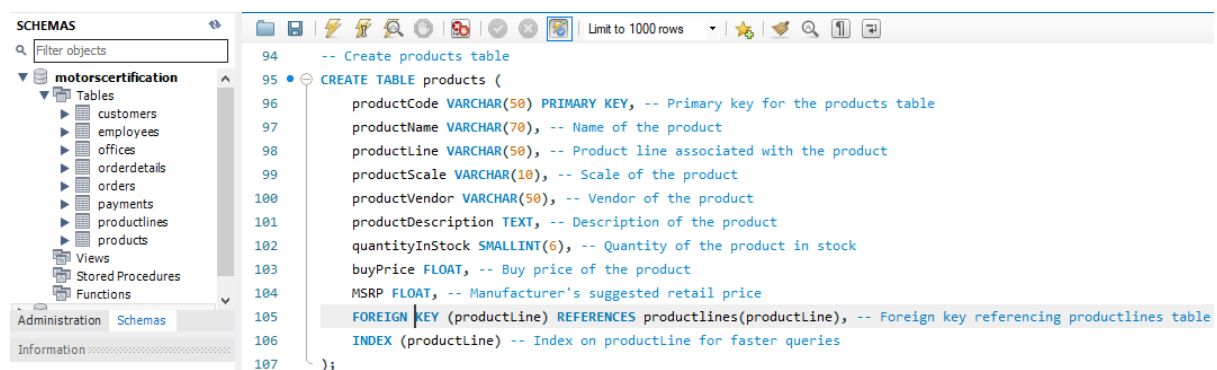


Fig. 1.0

Conclusion

I have completed this question which I designed **ER Diagram** and a well-structured **relational database** by defining key tables, relationships, and constraints. Each table was carefully designed based on the project's requirements with primary and foreign keys to ensure data integrity and optimize query performance. Indexing was also implemented on relevant columns to enhance database efficiency. This structured approach provides a solid foundation for managing customers, employees, orders, payments, and product details, ensuring smooth data handling and retrieval. In the next question, I will be populating the tables with the given data.

Question 2:

After designing the table insert records in the following orderdetails, employees, payments, products, customers, offices and orders table.

Note: Refer to the CSV files provided on the LMS.

Solution

Introduction:

This step involves populating the tables with relevant data. This process requires inserting records into the orderdetails, employees, payments, products, customers, offices, orders and productlines tables in the correct sequence to maintain referential integrity. Data from the provided CSV files will be used to ensure accuracy and consistency. Proper insertion of records is crucial for testing database functionality and ensuring it supports real-world business operations effectively. Screenshot

Orderdetails Table

Screenshot Of Code:

The below screenshot of the SQL code shows the INSERT INTO statement used to populate the orderdetails table with sample data. Each record consists of five values: orderNumber, productCode, quantityOrdered, priceEach, and orderLineNumber. These values represent individual product orders, linking them to existing orders and products in the database. By executing this script, multiple rows are inserted simultaneously, ensuring efficient data entry while maintaining database integrity. This step is essential for testing order-related queries and analysing the purchasing patterns of customers.

```

Filter objects
motorscertification
└─ Tables
   ├── customers
   ├── employees
   ├── offices
   ├── orderdetails
   ├── orders
   ├── payments
   ├── productlines
   └── products
Views
Stored Procedures
Functions
Administration Schemas
No object selected

181 • insert into orderdetails values
182     (10100,'S18_1749',30,'136.00',3),
183     (10101,'S18_2248',50,'55.09',2),
184     (10102,'S18_4409',22,'75.46',4),
185     (10103,'S24_3969',49,'35.29',1),
186     (10104,'S18_2325',25,'108.06',4),
187     (10105,'S18_2795',26,'167.06',1),
188     (10106,'S24_1937',45,'32.53',3),
189     (10107,'S24_2022',46,'44.35',2),
190     (10108,'S18_1342',39,'95.55',2),
191     (10109,'S18_1367',41,'43.13',1),
192     (10110,'S10_1949',26,'214.30',11),
193     (10111,'S10_4962',42,'119.67',4),
194     (10112,'S12_1666',27,'121.64',8),
195     (10113,'S18_1097',35,'94.50',10),
196     (10114,'S18_2432',22,'58.34',2),
197     (10115,'S18_2949',27,'92.19',12),
198     (10116,'S18_2957',35,'61.84',14),
199     (10117,'S18_3136',25,'86.92',13),
200     (10118,'S18_3320',46,'86.31',16),
201     (10119,'S18_4600',36,'98.07',5);
202 • select * from orderdetails;
  
```

Fig. 2.1

Customers Table

Screenshot of code:

The screenshot of the SQL code demonstrates how the customers table is populated using the INSERT INTO statement. Each row consists of multiple attributes, including customerNumber, customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber, and creditLimit. These records represent different customers, each associated with a unique identifier and relevant contact details. By executing this script, the database now contains essential customer information, which is crucial for processing orders, payments, and sales analysis.

```

117 INSERT INTO customers VALUES
118 (103,'Atelier graphique','Schmitt','Carine ','40.32.2555','54, rue Royale',NULL,'Nantes',NULL,'44000','France',1370,'21000.00'),
119 (112,'Signal Gift Stores','King','Jean','7025551838','8409 Strong St.',NULL,'Las Vegas','NV','83030','USA',1166,'71800.00'),
120 (114,'Australian Collectors, Co.','Ferguson','Peter','03 9520 4555','636 St Kilda Road','Level 3','Melbourne','Victoria','3004','Australia',1611,'118200.00'),
121 (119,'La Rochelle Gifts','Labrun','Janine ','40.67.8555','67, rue des Cinquante Otages',NULL,'Nantes',NULL,'44000','France',1370,'118200.00'),
122 (121,'Baane Mini Imports','Bergulfsen','Jonas ','07-98 9555','Erling Skakkas gate 78',NULL,'Stavern',NULL,'4110','Norway',1504,'81700.00'),
123 (124,'Mini Gifts Distributors Ltd.','Nelson','Susan','4155551450','5677 Strong St.',NULL,'San Rafael','CA','97562','USA',1165,'210500.00'),
124 (125,'Havel & Zbyszek Co.','Piestrzeniewicz','Zbyszek ','(26) 642-7555','ul. Filtrowa 68',NULL,'Warszawa',NULL,'01-012','Poland',NULL,'0.00'),
125 (128,'Blauer See Auto, Co.','Keitel','Roland','49 69 66 90 2555','Lyonerstr. 34',NULL,'Frankfurt',NULL,'60528','Germany',1504,'59700.00'),
126 (129,'Mini Wheels Co.','Murphy','Julie','6505555787','5557 North Pendale Street',NULL,'San Francisco','CA','94217','USA',1165,'64600.00'),
127 (131,'Land of Toys Inc.','Lee','Kwai','2125557818','897 Long Airport Avenue',NULL,'NYC','NY','10022','USA',1323,'114900.00'),
128 (141,'Euro+ Shopping Channel','Freyre','Diego ','(91) 555 94 44','C/ Moralzarzal, 86',NULL,'Madrid',NULL,'28034','Spain',1370,'227600.00'),
129 (144,'Volvo Model Replicas, Co','Berglund','Christina ','0921-12 3555','Berguvsvägen 8',NULL,'Luleå',NULL,'S-958 22','Sweden',1504,'53100.00'),
130 (145,'Danish Wholesale Imports','Petersen','Jytte ','31 12 3555','Vinbæltet 34',NULL,'København',NULL,'1734','Denmark',1401,'83400.00'),
131 (146,'Saveley & Henriot, Co.','Saveley','Mary ','78.32.5555','2, rue du Commerce',NULL,'Lyon',NULL,'69004','France',1337,'123900.00'),
132 (148,'Dragon Souvenirs, Ltd.','Natividad','Eric','465 221 7555','Bronz Sok.','Bronz Apt. 3/6 Tesvikiye','Singapore',NULL,'079903','Singapore',1621,'97900.00'),
133 (151,'Muscle Machine Inc','Young','Jeff','2125557413','4092 Furth Circle','Suite 400','NYC','NY','10022','USA',1286,'138500.00'),
134 (157,'Diecast Classics Inc.','Leong','Kelvin','2155551555','7586 Pompton St.',NULL,'Allentown','PA','70267','USA',1216,'100600.00'),
135 (161,'Technics Stores Inc.','Hashimoto','Juri','6505556809','9408 Furth Circle',NULL,'Burlingame','CA','94217','USA',1165,'84600.00'),
136 (166,'Handji Gifts & Co','Victorino','Wendy','465 224 1555','106 Linden Road Sandown','2nd Floor','Singapore',NULL,'069045','Singapore',1612,'97900.00'),
137 (167,'Herkuu Gifts','Oeztan','Veysel','447 2267 3215','Brehmen St. 121','PR 334 Sentrum','Bergen',NULL,'N 5804','Norway ','1504','96800.00');
138 • select * from customers;

```

Fig. 2.2

Employees Table

Screenshot Of Code:

The screenshot of the SQL code illustrates how the **employees** table is populated using the INSERT INTO statement. Each row represents an employee and contains attributes such as employeeNumber, lastName, firstName, extension, email, officeCode, reportsTo, and jobTitle. These records establish the hierarchical structure within the organization, with employees reporting to specific managers. Executing this script ensures that all employee data is correctly stored, forming the basis for processing sales, customer relationships, and office assignments.



```

142 insert into employees values
143 (1002,'Murphy','Diane','x5800','dmurphy@classicmodelcars.com','1',NULL,'President'),
144 (1056,'Patterson','Mary','x4611','mpatterso@classicmodelcars.com','1',1002,'VP Sales'),
145 (1076,'Firrelli','Jeff','x9273','jfirrelli@classicmodelcars.com','1',1002,'VP Marketing'),
146 (1088,'Patterson','William','x4871','wpatterson@classicmodelcars.com','6',1056,'Sales Manager (APAC)'),
147 (1102,'Bondur','Gerard','x5408','gbondur@classicmodelcars.com','4',1056,'Sale Manager (EMEA)'),
148 (1143,'Bow','Anthony','x5428','abow@classicmodelcars.com','1',1056,'Sales Manager (NA)'),
149 (1165,'Jennings','Leslie','x3291','ljennings@classicmodelcars.com','1',1143,'Sales Rep'),
150 (1166,'Thompson','Leslie','x4065','lthompson@classicmodelcars.com','1',1143,'Sales Rep'),
151 (1188,'Firrelli','Julie','x2173','jfirrelli@classicmodelcars.com','2',1143,'Sales Rep'),
152 (1216,'Patterson','Steve','x4334','spatterson@classicmodelcars.com','2',1143,'Sales Rep'),
153 (1286,'Tseng','Foon Yue','x2248','ftseng@classicmodelcars.com','3',1143,'Sales Rep'),
154 (1323,'Vanaufl','George','x4102','gvanauf@classicmodelcars.com','3',1143,'Sales Rep'),
155 (1337,'Bondur','Loui','x6493','lbondur@classicmodelcars.com','4',1102,'Sales Rep'),
156 (1370,'Hernandez','Gerard','x2028','ghernande@classicmodelcars.com','4',1102,'Sales Rep'),
157 (1401,'Castillo','Pamela','x2759','pcastillo@classicmodelcars.com','4',1102,'Sales Rep'),
158 (1501,'Bott','Larry','x2311','lbott@classicmodelcars.com','7',1102,'Sales Rep'),
159 (1504,'Jones','Barry','x102','bjones@classicmodelcars.com','7',1102,'Sales Rep'),
160 (1611,'Fixter','Andy','x101','afixter@classicmodelcars.com','6',1088,'Sales Rep'),
161 (1612,'Marsh','Peter','x102','pmarsh@classicmodelcars.com','6',1088,'Sales Rep'),
162 (1619,'King','Tom','x103','tking@classicmodelcars.com','6',1088,'Sales Rep');
163 • select * from employees;

```

Fig. 2.3

Orders Table

Screenshot Of Code:

The screenshot of the SQL code demonstrates the insertion of records into the **orders table** using the INSERT INTO statement. Each row represents a customer order and includes attributes such as orderNumber, orderDate, requiredDate, shippedDate, status, comments, and customerNumber. The data records various orders with different shipment statuses, comments, and customer IDs, ensuring a structured record of sales transactions. Running this script successfully ensures that the system maintains a complete history of customer orders for future reference and processing.

```

200 • insert into orders values
201 (10100,'2003-01-06','2003-01-13','2003-01-10','Shipped',NULL,363),
202 (10101,'2003-01-09','2003-01-18','2003-01-11','Shipped','Check on availability.',128),
203 (10102,'2003-01-10','2003-01-18','2003-01-14','Shipped',NULL,181),
204 (10103,'2003-01-29','2003-02-07','2003-02-02','Shipped',NULL,121),
205 (10104,'2003-01-31','2003-02-09','2003-02-01','Shipped',NULL,141),
206 (10105,'2003-02-11','2003-02-21','2003-02-12','Shipped',NULL,145),
207 (10106,'2003-02-17','2003-02-24','2003-02-21','Shipped',NULL,278),
208 (10107,'2003-02-24','2003-03-03','2003-02-26','Shipped','Difficult to negotiate with customer. We need more marketing materials',131),
209 (10108,'2003-03-03','2003-03-12','2003-03-08','Shipped',NULL,385),
210 (10109,'2003-03-10','2003-03-19','2003-03-11','Shipped','Customer requested that FedEx Ground is used for this shipping',486),
211 (10110,'2003-03-18','2003-03-24','2003-03-20','Shipped',NULL,187),
212 (10111,'2003-03-25','2003-03-31','2003-03-30','Shipped',NULL,129),
213 (10112,'2003-03-24','2003-04-03','2003-03-29','Shipped','Customer requested that ad materials (such as posters, pamphlets) be included in the shipment',
214 (10113,'2003-03-26','2003-04-02','2003-03-27','Shipped',NULL,124),
215 (10114,'2003-04-01','2003-04-07','2003-04-02','Shipped',NULL,172),
216 (10115,'2003-04-04','2003-04-12','2003-04-07','Shipped',NULL,424),
217 (10116,'2003-04-11','2003-04-19','2003-04-13','Shipped',NULL,381),
218 (10117,'2003-04-16','2003-04-24','2003-04-17','Shipped',NULL,148),
219 (10118,'2003-04-21','2003-04-29','2003-04-26','Shipped','Customer has worked with some of our vendors in the past and is aware of their MSRP',216),
220 (10119,'2003-04-28','2003-05-05','2003-05-02','Shipped',NULL,382);
221 • select * from orders;

```

Fig. 2.4

Offices Table

Screenshot Of Code:

The screenshot of the SQL code demonstrates the insertion of records into the **offices table** using the INSERT INTO statement. Each row represents a company office, including attributes such as officeCode, city, phone, addressLine1, addressLine2, state, country, postalCode, and territory. The data records offices located in different global regions (e.g., North America, EMEA, APAC), ensuring structured tracking of company locations. Successfully running this

script ensures that the system maintains a clear record of all company offices for internal management and operations.

```
165 • insert into offices values
166 ('1','San Francisco','+1 650 219 4782','100 Market Street','Suite 300','CA','USA','94080','NA'),
167 ('2','Boston','+1 215 837 0825','1550 Court Place','Suite 102','MA','USA','02107','NA'),
168 ('3','NYC','+1 212 555 3000','523 East 53rd Street','apt. 5A','NY','USA','10022','NA'),
169 ('4','Paris','+33 14 723 4404','43 Rue Jouffroy Dabbans',NULL,NULL,'France','75017','EMEA'),
170 ('5','Tokyo','+81 33 224 5000','4-1 Kioicho',NULL,'Chiyoda-Ku','Japan','102-8578','Japan'),
171 ('6','Sydney','+61 2 9264 2451','5-11 Wentworth Avenue','Floor #2',NULL,'Australia','NSW 2010','APAC'),
172 ('7','London','+44 20 7877 2041','25 Old Broad Street','Level 7',NULL,'UK','EC2N 1HN','EMEA');
173 • select * from offices;
```

Fig. 2.5

Payments Table

Screenshot Of Codes:

The screenshot captures the SQL script used to insert records into the payments table. The INSERT INTO statement adds multiple payment transactions, each identified by a customerNumber, checkNumber, paymentDate, and amount. These entries represent payments made by customers over different time periods, ensuring financial transactions are properly recorded. Successfully executing this script ensures accurate financial tracking within the database, allowing for efficient management of customer payments and financial reporting.

```

221 • insert into payments values
222     (101, 'HQ336336', '2004-10-19', '6066.78'),
223     (102, 'JM555205', '2003-06-05', '14571.44'),
224     (103, 'OM314933', '2004-12-18', '1676.14'),
225     (104, 'B0864823', '2004-12-17', '14191.12'),
226     (105, 'HQ55022', '2003-06-06', '32641.98'),
227     (106, 'ND748579', '2004-08-20', '33347.88'),
228     (107, 'GG31455', '2003-05-20', '45864.03'),
229     (108, 'MA765515', '2004-12-15', '82261.22'),
230     (109, 'NP603840', '2003-05-31', '7565.08'),
231     (110, 'NR27552', '2004-03-10', '44894.74'),
232     (111, 'DB933704', '2004-11-14', '19501.82'),
233     (112, 'LN373447', '2004-08-08', '47924.19'),
234     (113, 'NG94694', '2005-02-22', '49523.67'),
235     (114, 'DB889831', '2003-02-16', '50218.95'),
236     (115, 'FD317790', '2003-10-28', '1491.38'),
237     (116, 'KI831359', '2004-11-04', '17876.32'),
238     (117, 'MA302151', '2004-11-28', '34638.14'),
239     (118, 'AE215433', '2005-03-05', '101244.59'),
240     (119, 'BG255406', '2004-08-28', '85410.87'),
241     (120, 'CQ287967', '2003-04-11', '11044.30');
242 • select * from payments;

```

Fig. 2.6

Productlines Table

Screenshot Of Code:

The screenshot captures the SQL script used to insert records into the productlines table. This table categorizes different types of model vehicles, including Classic Cars, Motorcycles, Planes, Ships, Trains, Trucks and Buses, and Vintage Cars. Each category includes a detailed description highlighting unique features, materials, and scales of the model replicas. These product lines serve as the foundation for organizing inventory and ensuring efficient product management in the database.


```
insert into productlines values
```

```
('Classic Cars','Attention car enthusiasts: Make your wildest car ownership dreams come true. Whether you are looking for classic muscle cars, dream sports cars or movie-inspired mi
('Motorcycles','Our motorcycles are state of the art replicas of classic as well as contemporary motorcycle legends such as Harley Davidson, Ducati and Vespa. Models contain stunnin
('Planes','Unique, diecast airplane and helicopter replicas suitable for collections, as well as home, office or classroom decorations. Models contain stunning details such as offic
('Ships','The perfect holiday or anniversary gift for executives, clients, friends, and family. These handcrafted model ships are unique, stunning works of art that will be treasure
('Trains','Model trains are a rewarding hobby for enthusiasts of all ages. Whether you are looking for collectible wooden trains, electric streetcars or locomotives, you will find a
('Trucks and Buses','The Truck and Bus models are realistic replicas of buses and specialized trucks produced from the early 1920s to present. The models range in size from 1:12 to
('Vintage Cars','Our Vintage Car models realistically portray automobiles produced from the early 1900s through the 1940s. Materials used include Bakelite, diecast, plastic and wood
select * from productlines;
```

Activate Windows

Go to Settings to activate Windows.

Fig. 2.7

Products Table

Screenshot Of Codes:

The screenshot shows the SQL script for inserting product details into the products table. The products represent models of various types of vehicles such as Motorcycles, Classic Cars, Vintage Cars, Trucks and Buses, and others. Each product contains a detailed description, product name, manufacturer, and other specifics, including scale, price, and additional features such as opening doors, moving parts, and other collectible aspects.

```
255 • insert into products values
256 ('S10_1678','1969 Harley Davidson Ultimate Chopper','Motorcycles','1:10','Min Lin Diecast','This replica features working kickstand, front suspension, gear-shift lever, footbrake le
257 ('S10_1949','1952 Alpine Renault 1300','Classic Cars','1:10','Classic Metal Creations','Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; o
258 ('S10_2016','1996 Moto Guzzi 1100i','Motorcycles','1:10','Highway 66 Mini Classics','Official Moto Guzzi logos and insignias, saddle bags located on side of motorcycle, detailed eng
259 ('S10_4698','2003 Harley-Davidson Eagle Drag Bike','Motorcycles','1:10','Red Start Diecast','Model features, official Harley Davidson logos and insignias, detachable rear wheelie ba
260 ('S10_4757','1972 Alfa Romeo GTA','Classic Cars','1:10','Motor City Art Classics','Features include: Turnable front wheels; steering function; detailed interior; detailed engine; op
261 ('S10_4962','1962 Lancia Delta 16V','Classic Cars','1:10','Second Gear Diecast','Features include: Turnable front wheels; steering function; detailed interior; detailed engine; ope
262 ('S12_1099','1968 Ford Mustang','Classic Cars','1:12','Autoart Studio Design','Hood, doors and trunk all open to reveal highly detailed interior features. Steering wheel actually tu
263 ('S12_1108','2001 Ferrari Enzo','Classic Cars','1:12','Second Gear Diecast','Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trun
264 ('S12_1666','1958 Setra Bus','Trucks and Buses','1:12','Welly Diecast Productions','Model features 30 windows, skylights & glare resistant glass, working steering system, original l
265 ('S12_2823','2002 Suzuki XRE0','Motorcycles','1:12','Unimax Art Galleries','Official logos and insignias, saddle bags located on side of motorcycle, detailed engine, working steerin
266 ('S12_3148','1969 Corvair Monza','Classic Cars','1:18','Welly Diecast Productions','1:18 scale die-cast about 10\" long doors open, hood opens, trunk opens and wheels roll',6906,'89
267 ('S12_3380','1968 Dodge Charger','Classic Cars','1:12','Welly Diecast Productions','1:12 scale model of a 1968 Dodge Charger. Hood, doors and trunk all open to reveal highly detaile
268 ('S12_3891','1969 Ford Falcon','Classic Cars','1:12','Second Gear Diecast','Turnable front wheels; steering function; detailed interior; detailed engine; opening hood; opening trunk
269 ('S12_3990','1970 Plymouth Hemi Cuda','Classic Cars','1:12','Studio M Art Models','Very detailed 1970 Plymouth Cuda model in 1:12 scale. The Cuda is generally accepted as one of the
270 ('S12_4473','1957 Chevy Pickup','Trucks and Buses','1:12','Exoto Designs','1:12 scale die-cast about 20\" long Hood opens, Rubber wheels',6125,'55.70','118.50'),
271 ('S12_4675','1969 Dodge Charger','Classic Cars','1:12','Welly Diecast Productions','Detailed model of the 1969 Dodge Charger. This model includes finely detailed interior and exteri
272 ('S18_1097','1940 Ford Pickup Truck','Trucks and Buses','1:18','Studio M Art Models','This model features soft rubber tires, working steering, rubber mud guards, authentic Ford logo
273 ('S18_1129','1993 Mazda RX-7','Classic Cars','1:18','Highway 66 Mini Classics','This model features, opening hood, opening doors, detailed engine, rear spoiler, opening trunk, worki
274 ('S18_1342','1937 Lincoln Berline','Vintage Cars','1:18','Motor City Art Classics','Features opening engine cover, doors, trunk, and fuel filler cap. Color black',8693,'60.62','102.
275 ('S18_1367','1936 Mercedes-Benz 500K Special Roadster','Vintage Cars','1:18','Studio M Art Models','This 1:18 scale replica is constructed of heavy die-cast metal and has all the fe
276 • select * from products;
```

Fig. 2.8

Question 3:

Provide comments before every task that is performed describing the operation that is being performed and attach a screenshot of ER diagram from SSMS.

Solution:

Comments Before Every Task and Screenshot of ER Diagram from MySQL.

Explanation:

When working with databases, it is essential to document each operation to enhance readability, maintainability, and collaboration. This ensures that developers and database administrators can understand the intent of the queries, track modifications, and troubleshoot any issues efficiently.

I've already provided detailed comments before each task in the SQL File to describe its purpose when performing the structured database operations. Additionally, I have captured and attach a screenshot of the Entity-Relationship (ER) Diagram from MySQL Workbench to visualize the database structure, including relationships between tables, constraints, and dependencies.

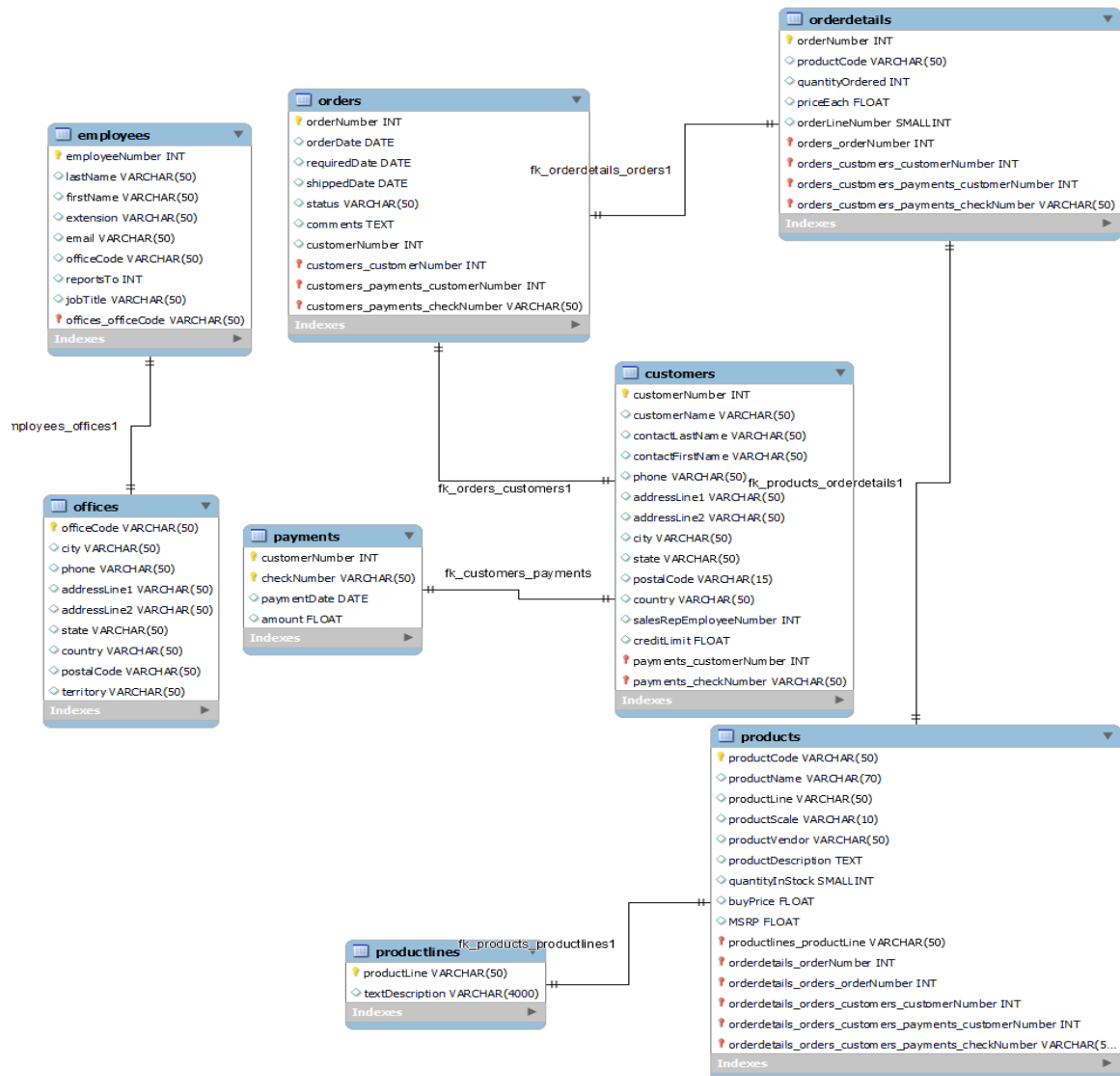


Fig. 3.0

Question 4

Delete the columns in productlines which are useless that do not infer anything

Solution:

Deleting Useless Columns in Productlines Table

Explanation:

When designing and managing a database, it is crucial to remove unnecessary columns to maintain efficiency, optimize storage, and enhance data integrity. Some columns may not provide meaningful information or contribute to business logic, making them redundant and ideal candidates for deletion.

In this task, I will identify and remove useless columns from the productlines table that do not infer any significant value. Specifically, the htmlDescription and image columns are considered unnecessary.

To achieve this, I will use:

- ALTER TABLE Statement: Allows modification of an existing table structure.
- DROP COLUMN Statement: Permanently removes specified columns from a table.

By executing these SQL statements, we ensure that the database remains clean, optimized, and free from redundant data.

```
447 -- The useless columns in the product line table is are the htmlDescription and image
448 -- to delete thses columns, the ALTER TABLE and the DROP COLUMN Statement was used
449 -- ALTER TABLE Statement ensures that the already exixting table, productlines, can be modified
450 -- DROP COLUMN statement ensures that the a specific column can be modified / removed
451 -- Drop htmlDescription column
452 • ALTER TABLE productlines DROP COLUMN htmlDescription; -- Removes the htmlDescription column
453 -- Drop image column
454 • ALTER TABLE productlines DROP COLUMN image; -- Removes the image column
---
```

Fig. 4.0

Question 5:

Use a select statement to verify all insertions as well as updates.

Solution:

Verifying All Inserting Using the Select Statement

Explanation:

To verify that the inserted and all updated columns for each table are stored, I will use the select statement to retrieve all the data from each table to confirm the storage. To do so, I will use:

- **Select *:** responsible for retrieve all the required data. And
- **From (followed by the table name):** which will specify the table I want to retrieve the data from. As shown below:

Orderdetails Table:

Verification of Inserted Data into Orderdetails:

The below screenshot displays the result of executing a **SELECT * FROM orderdetails;** statement, confirming that the data has been successfully inserted into the table. The output table includes the columns orderNumber, productCode, quantityOrdered, priceEach, and orderLineNumber, showing the exact values inserted. This verification step is crucial for ensuring that all records have been correctly stored, enabling further data manipulation, reporting, and analysis. Any discrepancies between the expected and actual results can be resolved by rechecking the insert queries or database constraints.

Result Grid					
Filter Rows:					
	orderNumber	productCode	quantityOrdered	priceEach	orderLineNumber
	10101	S18_2248	50	55.09	2
	10102	S18_4409	22	75.46	4
	10103	S24_3969	49	35.29	1
	10104	S18_2325	25	108.06	4
	10105	S18_2795	26	167.06	1
	10106	S24_1937	45	32.53	3
	10107	S24_2022	46	44.35	2
	10108	S18_1342	39	95.55	2
	10109	S18_1367	41	43.13	1
	10110	S10_1949	26	214.3	11
	10111	S10_4962	42	119.67	4
	10112	S12_1666	27	121.64	8
	10113	S18_1097	35	94.5	10
	10114	S18_2432	22	58.34	2
	10115	S18_2949	27	92.19	12
	10116	S18_2957	35	61.84	14
	10117	S18_3136	25	86.92	13
	10118	S18_3320	46	86.31	16
	10119	S18_4600	36	98.07	5
	NULL	NULL	NULL	NULL	NULL

Fig. 5.0

Customers Table

Verification Of Inserted Data into Customers:

The below screenshot presents the result of executing a `SELECT * FROM customers;` statement, confirming that the data has been successfully inserted into the table. The displayed table includes all inserted customer records, verifying that each field is correctly stored. This step is essential for ensuring data integrity and accuracy, allowing for further queries and transactions involving customer-related operations. Any discrepancies or missing records can be identified and corrected through additional insertions or updates.

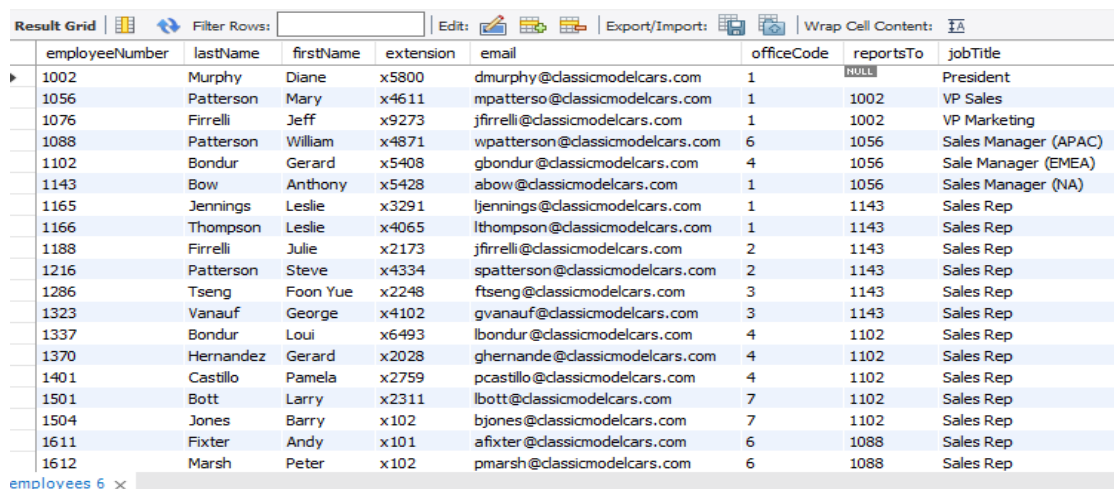
	customerNumber	customerName	contactLastName	contactFirstName	phone	addressLine1	addressLine2	city	state	postalCode	country
▶	103	Atelier graphique	Schmitt	Carine	40.32.2555	54, rue Royale	NULL	Nantes	NULL	44000	France
	112	Signal Gift Stores	King	Jean	7025551838	8489 Strong St.	NULL	Las Vegas	NV	83030	USA
	114	Australian Collectors, Co.	Ferguson	Peter	03 9520 4555	636 St Kilda Road	Level 3	Melbourne	Victoria	3004	Australia
	119	La Rochelle Gifts	Labruno	Janine	40.67.8555	67, rue des Cinquante Otages	NULL	Nantes	NULL	44000	France
	121	Baane Mini Imports	Bergulfsen	Jonas	07-98 9555	Erling Skakkes gate 78	NULL	Stavern	NULL	4110	Norway
	124	Mini Gifts Distributors Ltd.	Nelson	Susan	4155551450	5677 Strong St.	NULL	San Rafael	CA	97562	USA
	125	Havel & Zbyszek Co	Piestrzeniewicz	Zbyszek	(26) 642-7555	ul. Filtrów 68	NULL	Warszawa	NULL	01-012	Poland
	128	Blauer See Auto, Co.	Keitel	Roland	+49 69 66 90 2555	Lyonerstr. 34	NULL	Frankfurt	NULL	60528	Germany
	129	Mini Wheels Co.	Murphy	Julie	6505555787	5557 North Pendale Street	NULL	San Francisco	CA	94217	USA
	131	Land of Toys Inc.	Lee	Kwai	2125557818	897 Long Airport Avenue	NULL	NYC	NY	10022	USA
	141	Euro+ Shopping Channel	Freyre	Diego	(91) 555 94 44	C/ Moralzarzal, 86	NULL	Madrid	NULL	28034	Spain
	144	Volvo Model Replicas, Co	Berglund	Christina	0921-12 3555	Berguvsvägen 8	NULL	Luleå	NULL	S-958 22	Sweden
	145	Danish Wholesale Imports	Petersen	Jytte	31 12 3555	Vinbæltet 34	NULL	Kobenhavn	NULL	1734	Denmark
	146	Saveley & Henriot, Co.	Saveley	Mary	78.32.5555	2, rue du Commerce	NULL	Lyon	NULL	69004	France
	148	Dragon Souvenirs, Ltd.	Natividad	Eric	+65 221 7555	Bronz Sok.	Bronz Apt. 3...	Singapore	NULL	079903	Singapore
	151	Muscle Machine Inc	Young	Jeff	2125557413	4092 Furth Circle	Suite 400	NYC	NY	10022	USA
	157	Diecast Classics Inc.	Leong	Kelvin	2155551555	7586 Pompton St.	NULL	Allentown	PA	70267	USA
	161	Technics Stores Inc.	Hashimoto	Juri	6505556809	9408 Furth Circle	NULL	Burlingame	CA	94217	USA

Fig. 5.1

Employees Table

Verification Of Inserted Data into Employees:

The second screenshot displays the result of executing the `SELECT * FROM employees;` statement, verifying that the employee records have been successfully inserted into the table. The displayed table contains all inserted employee data, confirming the accuracy of names, job titles, office assignments, and reporting structures. This step is crucial for ensuring the integrity of employee information, which plays a significant role in organizational operations such as sales management and customer service. Any inconsistencies in the data can be identified and rectified through updates or additional insertions.



employeeNumber	lastName	firstName	extension	email	officeCode	reportsTo	jobTitle
1002	Murphy	Diane	x5800	dmurphy@classicmodelcars.com	1	NULL	President
1056	Patterson	Mary	x4611	mpatterson@classicmodelcars.com	1	1002	VP Sales
1076	Firrelli	Jeff	x9273	jfirrelli@classicmodelcars.com	1	1002	VP Marketing
1088	Patterson	William	x4871	wpatterson@classicmodelcars.com	6	1056	Sales Manager (APAC)
1102	Bondur	Gerard	x5408	gbondur@classicmodelcars.com	4	1056	Sale Manager (EMEA)
1143	Bow	Anthony	x5428	abow@classicmodelcars.com	1	1056	Sales Manager (NA)
1165	Jennings	Leslie	x3291	ljennings@classicmodelcars.com	1	1143	Sales Rep
1166	Thompson	Leslie	x4065	lthompson@classicmodelcars.com	1	1143	Sales Rep
1188	Firrelli	Julie	x2173	jfirrelli@classicmodelcars.com	2	1143	Sales Rep
1216	Patterson	Steve	x4334	spatterson@classicmodelcars.com	2	1143	Sales Rep
1286	Tseng	Foon Yue	x2248	ftseng@classicmodelcars.com	3	1143	Sales Rep
1323	Vanauf	George	x4102	gvanauf@classicmodelcars.com	3	1143	Sales Rep
1337	Bondur	Loui	x6493	lbondur@classicmodelcars.com	4	1102	Sales Rep
1370	Hernandez	Gerard	x2028	ghernande@classicmodelcars.com	4	1102	Sales Rep
1401	Castillo	Pamela	x2759	pcastillo@classicmodelcars.com	4	1102	Sales Rep
1501	Bott	Larry	x2311	lbott@classicmodelcars.com	7	1102	Sales Rep
1504	Jones	Barry	x102	bjones@classicmodelcars.com	7	1102	Sales Rep
1611	Fixter	Andy	x101	afixter@classicmodelcars.com	6	1088	Sales Rep
1612	Marsh	Peter	x102	pmarsh@classicmodelcars.com	6	1088	Sales Rep

Fig. 5.2

Orders Table

Verification of Inserted Data into orders:

The second screenshot displays the result of executing the `SELECT * FROM orders;` statement, confirming that all orders have been successfully inserted into the table. The output includes order details such as order dates, shipping dates, and any special comments, verifying that the data aligns with expectations. This step is crucial in ensuring that the orders have been correctly recorded and are accessible for further processing, such as order tracking and inventory management. Any discrepancies in the data can be identified and corrected as necessary.

	orderNumber	orderDate	requiredDate	shippedDate	status	comments	customerNumber
▶	10100	2003-01-06	2003-01-13	2003-01-10	Shipped	NULL	363
	10101	2003-01-09	2003-01-18	2003-01-11	Shipped	Check on availability.	128
	10102	2003-01-10	2003-01-18	2003-01-14	Shipped	NULL	181
	10103	2003-01-29	2003-02-07	2003-02-02	Shipped	NULL	121
	10104	2003-01-31	2003-02-09	2003-02-01	Shipped	NULL	141
	10105	2003-02-11	2003-02-21	2003-02-12	Shipped	NULL	145
	10106	2003-02-17	2003-02-24	2003-02-21	Shipped	NULL	278
	10107	2003-02-24	2003-03-03	2003-02-26	Shipped	Difficult to negotiate with customer. We need m...	131
	10108	2003-03-03	2003-03-12	2003-03-08	Shipped	NULL	385
	10109	2003-03-10	2003-03-19	2003-03-11	Shipped	Customer requested that FedEx Ground is used...	486
	10110	2003-03-18	2003-03-24	2003-03-20	Shipped	NULL	187
	10111	2003-03-25	2003-03-31	2003-03-30	Shipped	NULL	129
	10112	2003-03-24	2003-04-03	2003-03-29	Shipped	Customer requested that ad materials (such as ...	144
	10113	2003-03-26	2003-04-02	2003-03-27	Shipped	NULL	124
	10114	2003-04-01	2003-04-07	2003-04-02	Shipped	NULL	172
	10115	2003-04-04	2003-04-12	2003-04-07	Shipped	NULL	424
	10116	2003-04-11	2003-04-19	2003-04-13	Shipped	NULL	381
	10117	2003-04-16	2003-04-24	2003-04-17	Shipped	NULL	148
	10118	2003-04-21	2003-04-29	2003-04-26	Shipped	Customer has worked with some of our vendors...	216

orders 8 x

Fig. 5.3

Offices Table

Verification Of Inserted Data into Offices:

The screenshot below displays the result of executing the `SELECT * FROM offices;` statement, confirming that all office records have been successfully inserted. The output includes details such as office locations, contact numbers, and territories, verifying that the data aligns with expectations. This verification step ensures that the company's office records are accurately stored and accessible for further operations such as employee assignments and regional sales tracking. Any discrepancies in the data can be identified and corrected as necessary.

	officeCode	city	phone	addressLine1	addressLine2	state	country	postalCode	territory
▶	1	San Francisco	+1 650 219 4782	100 Market Street	Suite 300	CA	USA	94080	NA
	2	Boston	+1 215 837 0825	1550 Court Place	Suite 102	MA	USA	02107	NA
	3	NYC	+1 212 555 3000	523 East 53rd Street	apt. 5A	NY	USA	10022	NA
	4	Paris	+33 14 723 4404	43 Rue Jouffroy Dabbans	NULL	NULL	France	75017	EMEA
	5	Tokyo	+81 33 224 5000	4-1 Kioicho	NULL	NULL	Chiyoda-Ku Japan	102-8578	Japan
	6	Sydney	+61 2 9264 2451	5-11 Wentworth Avenue	Floor #2	NULL	Australia	NSW 2010	APAC
	7	London	+44 20 7877 2041	25 Old Broad Street	Level 7	NULL	UK	EC2N 1HN	EMEA
*	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL	NULL

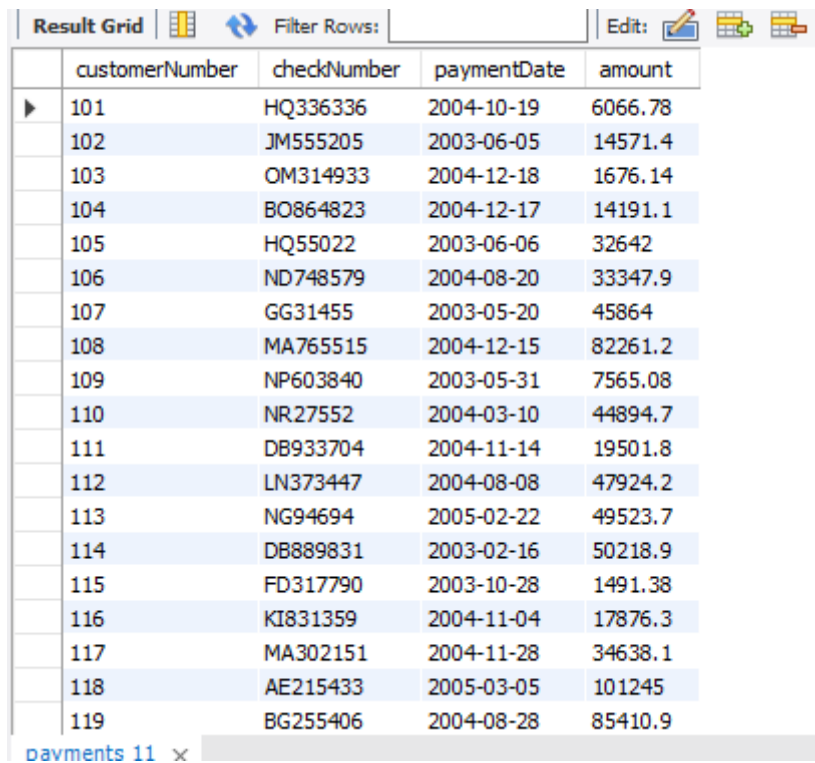
Fig. 5.4

Payments Table

Verification Of Inserted Data into Payments:

The second screenshot shows the output of executing the `SELECT * FROM payments;` statement, confirming that all payment transactions have been successfully inserted into the

database. The retrieved data includes customerNumber, checkNumber, paymentDate, and amount, ensuring each payment is properly logged. This verification process guarantees the integrity of financial records, enabling smooth payment reconciliation, audits, and customer account management. Any missing or incorrect data can be reviewed and corrected as necessary.



	customerNumber	checkNumber	paymentDate	amount
▶	101	HQ336336	2004-10-19	6066.78
	102	JM555205	2003-06-05	14571.4
	103	OM314933	2004-12-18	1676.14
	104	BO864823	2004-12-17	14191.1
	105	HQ55022	2003-06-06	32642
	106	ND748579	2004-08-20	33347.9
	107	GG31455	2003-05-20	45864
	108	MA765515	2004-12-15	82261.2
	109	NP603840	2003-05-31	7565.08
	110	NR27552	2004-03-10	44894.7
	111	DB933704	2004-11-14	19501.8
	112	LN373447	2004-08-08	47924.2
	113	NG94694	2005-02-22	49523.7
	114	DB889831	2003-02-16	50218.9
	115	FD317790	2003-10-28	1491.38
	116	KI831359	2004-11-04	17876.3
	117	MA302151	2004-11-28	34638.1
	118	AE215433	2005-03-05	101245
	119	BG255406	2004-08-28	85410.9

payments 11 x

Fig. 5.5

Productlines Table

Verification Of Inserted Data into Productlines:

The second screenshot displays the output of executing `SELECT * FROM productlines;` verifying that all categories have been successfully added to the database. The retrieved data confirms that each product line includes a proper description, ensuring clear classification for customers and inventory management. This verification step ensures data accuracy and completeness, allowing for better organization and future queries related to product categorization.

Result Grid		Filter Rows:	Edit:	Export
	productLine	textDescription		
▶	Classic Cars	Attention car enthusiasts: Make your wildest ca...		
	Motorcycles	Our motorcycles are state of the art replicas of ...		
	Planes	Unique, diecast airplane and helicopter replicas ...		
	Ships	The perfect holiday or anniversary gift for exec...		
	Trains	Model trains are a rewarding hobby for enthusi...		
	Trucks and Buses	The Truck and Bus models are realistic replicas o...		
	Vintage Cars	Our Vintage Car models realistically portray aut...		
*	NULL	NULL		

Fig. 5.6

Products Table

Verification Of Inserted Data into Products:

The screenshot below displays the output from `SELECT * FROM products`. This confirms that all 18 products have been successfully inserted into the database. The output shows that each product is correctly categorized, contains the necessary details, and can now be queried for inventory and customer management purposes. The successful insertion ensures that data is well-organized for future reference or analysis.

productCode	productName	productLine	productScale	productVendor	productDescription	quantityInStock	buyPrice	MSRP
S10_1678	1969 Harley Davidson Ultimate Chopper	Motorcycles	1:10	Min Lin Diecast	This replica features working kickstand, front su...	7933	48.81	95.7
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10	Classic Metal Creations	Turnable front wheels; steering function; detail...	7305	98.58	214.3
S10_2016	1996 Moto Guzzi 1100i	Motorcycles	1:10	Highway 66 Mini Classics	Official Moto Guzzi logos and insignias, saddle b...	6625	68.99	118.94
S10_4698	2003 Harley-Davidson Eagle Drag Bike	Motorcycles	1:10	Red Start Diecast	Model features, official Harley Davidson logos a...	5582	91.02	193.66
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10	Motor City Art Classics	Features include: Turnable front wheels; steeri...	3252	85.68	136
S10_4962	1962 Lancia Delta 16V	Classic Cars	1:10	Second Gear Diecast	Features include: Turnable front wheels; steeri...	6791	103.42	147.74
S12_1099	1968 Ford Mustang	Classic Cars	1:12	Autoart Studio Design	Hood, doors and trunk all open to reveal highly ...	68	95.34	194.57
S12_1108	2001 Ferrari Enzo	Classic Cars	1:12	Second Gear Diecast	Turnable front wheels; steering function; detail...	3619	95.59	207.8
S12_1666	1958 Setra Bus	Trucks and Buses	1:12	Welly Diecast Productions	Model features 30 windows, skylights & glare re...	1579	77.9	136.67
S12_2823	2002 Suzuki XREO	Motorcycles	1:12	Unimax Art Galleries	Official logos and insignias, saddle bags located ...	9997	66.27	150.62
S12_3148	1969 Corvair Monza	Classic Cars	1:18	Welly Diecast Productions	1:18 scale die-cast about 10" long doors open, ...	6906	89.14	151.08
S12_3380	1968 Dodge Charger	Classic Cars	1:12	Welly Diecast Productions	1:12 scale model of a 1968 Dodge Charger. Ho...	9123	75.16	117.44
S12_3891	1969 Ford Falcon	Classic Cars	1:12	Second Gear Diecast	Turnable front wheels; steering function; detail...	1049	83.05	173.02
S12_3990	1970 Plymouth Hemi Cuda	Classic Cars	1:12	Studio M Art Models	Very detailed 1970 Plymouth Cuda model in 1:1...	5663	31.92	79.8
S12_4473	1957 Chevy Pickup	Trucks and Buses	1:12	Exoto Designs	1:12 scale die-cast about 20" long Hood opens, ...	6125	55.7	118.5
S12_4675	1969 Dodge Charger	Classic Cars	1:12	Welly Diecast Productions	Detailed model of the 1969 Dodge Charger. Thi...	7323	58.73	115.16
S18_1097	1940 Ford Pickup Truck	Trucks and Buses	1:18	Studio M Art Models	This model features soft rubber tires, working s...	2613	58.33	116.67
S18_1129	1993 Mazda RX-7	Classic Cars	1:18	Highway 66 Mini Classics	This model features, opening hood, opening do...	3975	83.51	141.54
S18_1342	1937 Lincoln Berline	Vintage Cars	1:18	Motor City Art Classics	Features opening engine cover, doors, trunk, a...	8693	60.62	102.74
S18_1367	1936 Mercedes-Benz 500K Special Roa...	Vintage Cars	1:18	Studio M Art Models	This 1:18 scale replica is constructed of heavy d...	8635	24.26	53.91

Fig. 5.7

Question 6:

Find out the highest and the lowest amount.

Solution:

Finding The Highest and The Lowest Amount.

The below screenshot shows SQL query that retrieves the maximum payment amount from the payments table.

Explanation of the code used:

- **SELECT:** This statement is used to specify the columns or expressions that you want to retrieve from the database.
- **MAX(amount):** The MAX() function is an aggregate function in SQL that returns the largest value from a specified column. Here, it is applied to the amount column in the payments table.
- **amount:** This column contains the payment amounts in the payments table.
- **AS HighestAmount:** The AS keyword is used to rename the output column. In this case, the result of the MAX(amount) function is labelled as HighestAmount.
- **MIN(amount):** The MIN() function is an aggregate function that returns the smallest value from a specified column. Here, it is applied to the amount column in the payments table.
- **AS LowestAmount:** Renames the result of the MIN(amount) function as LowestAmount, making the output more readable and understandable.
- **FROM payments:** This clause specifies the table from which the data will be retrieved. In this case, it is the payments table.

Screenshot Of Highest Amount Value and SQL Code Used:

The query calculates and returns the highest payment amount recorded in the payments table.

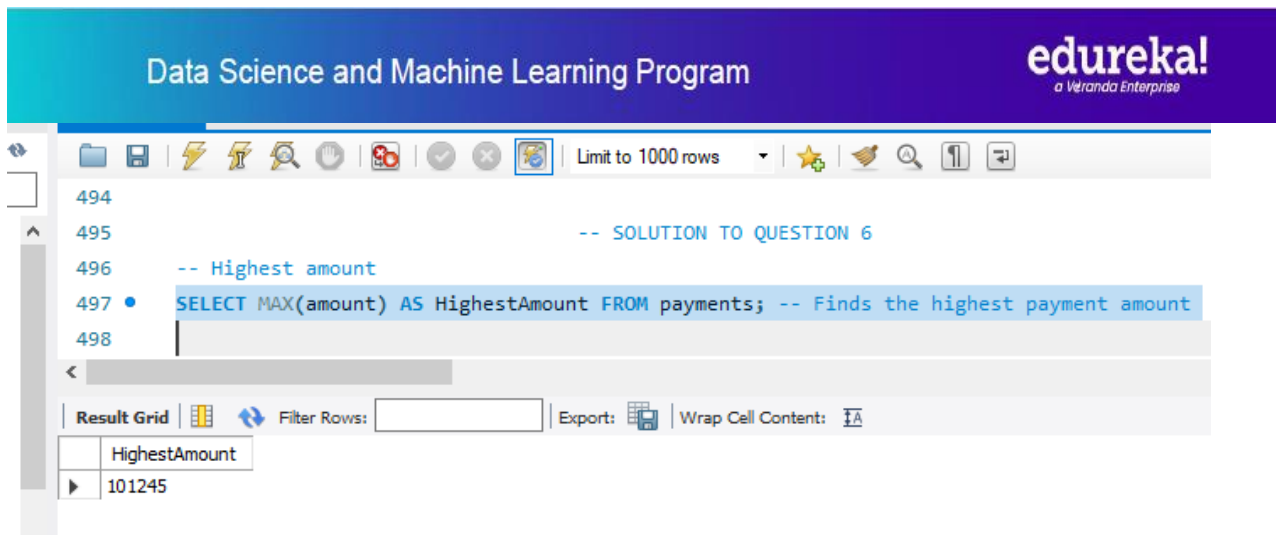


Fig. 6.0

Screenshot Of Lowest Amount Value and SQL Code Used:

The query calculates and returns the lowest payment amount recorded in the payments table.

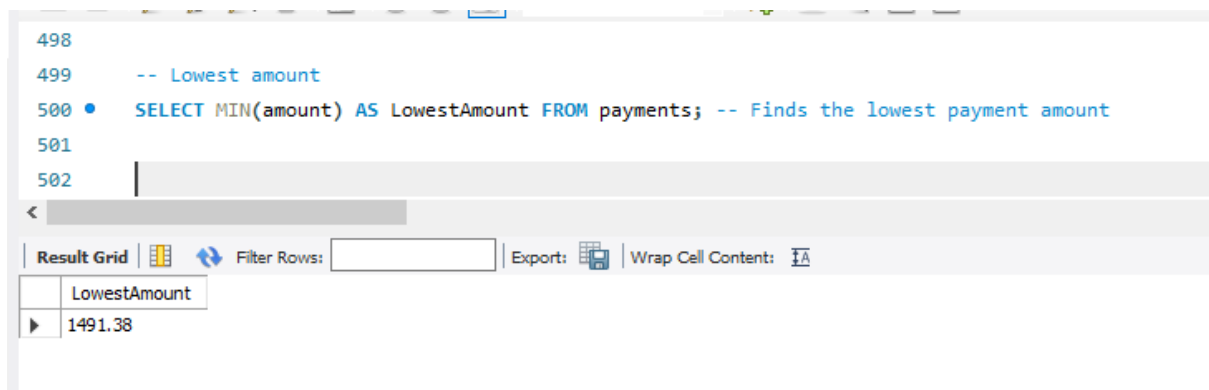


Fig. 6.0

Question 7


Give the unique count of customerName from customers.

Solutions:

Finding The Unique Count of Customername from Customers:

The below screenshot shows SQL query used to count the number of unique customer names in the customers table. The SELECT statement retrieves data from the database, while the COUNT(DISTINCT customerName) function counts the distinct customer names, ensuring that duplicates are not included in the count. The DISTINCT keyword ensures that only unique values in the customerName column are considered. The result of this count is then labelled as UniqueCustomerCount using the AS keyword, which renames the output column. The query is performed on the customers table, as specified by the FROM customers clause. In summary, this query calculates and returns the total number of unique customer names in the table, providing a precise count without duplicates.

```
511 -- Unique count of customerName
512 -- The SELECT statement retrieves data from the database
513 -- COUNT(DISTINCT customerName) function counts the distinct customer names, ensuring that duplicates are not included in the count.
514 -- The DISTINCT keyword ensures that only unique values in the customerName column are considered.
515 • SELECT COUNT(DISTINCT customerName) AS UniqueCustomerCount FROM customers; -- Counts unique customer names
516
517
518 /*
```



The screenshot shows a SQL query execution interface. The query is: `SELECT COUNT(DISTINCT customerName) AS UniqueCustomerCount FROM customers;`. The result is displayed in a table with one column, `UniqueCustomerCount`, and one row with the value `20`. The interface includes a 'Result Grid' tab, a 'Filter Rows' input field, and an 'Export' button.

UniqueCustomerCount
20

Fig. 7.0

Question 8

Create a view from customers and payments named cust_payment and select customerName, amount, contactLastName, contactFirstName who have paid.


Truncate and drop the view after operation.

Solution:

Creating View from Customers and Payments

This SQL script creates a view named cust_payment, retrieves data from it, and then drops it after use. The CREATE VIEW statement defines a virtual table that includes selected columns from the customers and payments tables. The SELECT statement within the view extracts customerName, amount, contactLastName, and contactFirstName, ensuring that only customers who have made payments are included. The JOIN operation links both tables using customerNumber, a common field in both tables. After the view is created, the SELECT * FROM cust_payment; statement retrieves and displays the data stored in the view. Finally, the DROP VIEW cust_payment; statement removes the view from the database, ensuring no residual data remains.

```
527 -- Create view
528 -- The CREATE VIEW statement defines a virtual table that includes selected columns from the customers and payments tables.
529 -- The SELECT statement within the view extracts customerName, amount, contactLastName, and contactFirstName, ensuring that only customers who have made payments are included.
530 -- The JOIN operation links both tables using customerNumber, a common field in both tables
531 • CREATE VIEW cust_payment AS
532     SELECT c.customerName, p.amount, c.contactLastName, c.contactFirstName -- retrieve columns needed to create the view
533     FROM customers c
534     JOIN payments p ON c.customerNumber = p.customerNumber; -- Combines customer and payment data
535
536 -- Select from view
537 • SELECT * FROM cust_payment; -- Displays data from the view
538
539 -- Drop view
540 • DROP VIEW cust_payment; -- Deletes the view
```



customerName	amount	contactLastName	contactFirstName
Atelier graphique	1676.14	Schmitt	Carine
Signal Gift Stores	47924.2	King	Jean
Australian Collectors, Co.	50218.9	Ferguson	Peter
La Rochelle Gifts	85410.9	Labruno	Janine

Fig. 8.0

Question 9:

Create a stored procedure on products which displays productLine for Classic Cars.

Solution:

Creating A Stored Procedure

The screenshot below displays SQL script used to create and execute a stored procedure named GetClassicCars, which retrieves all products belonging to the Classic Cars category from the products table. The DELIMITER // statement changes the default delimiter to //, ensuring that the procedure body is correctly interpreted without conflicts from semicolons inside the procedure. The CREATE PROCEDURE GetClassicCars() statement defines the procedure, and within the BEGIN ... END block, a SELECT * FROM products WHERE productLine = 'Classic Cars'; query is used to fetch all relevant records. After defining the procedure, the DELIMITER; resets the delimiter back to its default. Finally, CALL GetClassicCars(); executes the stored procedure, displaying all products that fall under the Classic Cars category.

```

551 -- The DELIMITER // statement changes the default delimiter to //, ensuring that the procedure body is correctly interpreted without conf
552 -- The CREATE PROCEDURE GetClassicCars() statement defines the procedure, and within the BEGIN ... END block
553 -- SELECT * FROM products WHERE productLine = 'Classic Cars'; query is used to fetch all relevant records.
554 -- After defining the procedure, the DELIMITER; resets the delimiter back to its default.
555 -- CALL GetClassicCars(); executes the stored procedure, displaying all products that fall under the Classic Cars category.
556 DELIMITER //
557 • CREATE PROCEDURE GetClassicCars()
558 BEGIN
559     SELECT * FROM products WHERE productLine = 'Classic Cars'; -- Selects products from Classic Cars
560 END //
561 DELIMITER ;
562 • -- Call stored procedure
563 CALL GetClassicCars(); -- Executes the stored procedure

```

productCode	productName	productLine	productScale	productVendor	productDescription	quantityInStock	buyPrice	MSRP
S10_1949	1952 Alpine Renault 1300	Classic Cars	1:10	Classic Metal Creations	Turnable front wheels; steering function; detail...	7305	98.58	214.3
S10_4757	1972 Alfa Romeo GTA	Classic Cars	1:10	Motor City Art Classics	Features include: Turnable front wheels; steeri...	3252	85.68	136
S10_4962	1962 Lancia Delta 16V	Classic Cars	1:10	Second Gear Diecast	Features include: Turnable front wheels; steeri...	6791	103.42	147.74
S12_1099	1968 Ford Mustang	Classic Cars	1:12	Autoart Studio Design	Hood, doors and trunk all open to reveal highly ...	68	95.34	194.57
S12_1108	2001 Ferrari Enzo	Classic Cars	1:12	Second Gear Diecast	Turnable front wheels; steering function; detail...	3619	95.59	207.8
S12_3148	1969 Corvair Monza	Classic Cars	1:18	Welly Diecast Productions	1:18 scale die-cast about 10" long doors open, ...	6906	89.14	151.08
S12_3380	1968 Dodge Charger	Classic Cars	1:12	Welly Diecast Productions	1:12 scale model of a 1968 Dodge Charger. Ho...	9123	75.16	117.44
S12_3891	1969 Ford Falcon	Classic Cars	1:12	Second Gear Diecast	Turnable front wheels; steering function; detail...	1049	83.05	173.02
S12_3990	1970 Plymouth Hemi Cuda	Classic Cars	1:12	Shurin M. Art Models	Very detailed 1970 Plymouth Cuda model in 1:1	5563	31.92	79.8

Fig. 9.0

Question 10

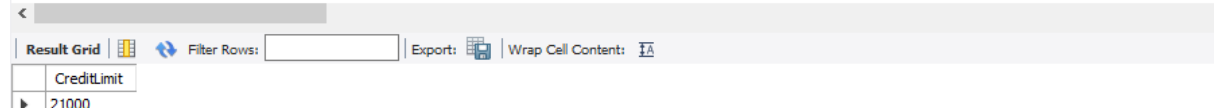
Create a function to get the creditLimit of customers less than 96800.

Solution:

Creating A Function to Get Customer's CreditLimit

This SQL script in the screenshot below defines and utilizes a function named GetCreditLimit to retrieve the credit limit of a specific customer from the customers table. The function takes a parameter customer_id of type INT, representing the unique customer number. It returns a FLOAT value, which corresponds to the customer's credit limit. The function is marked as DETERMINISTIC, meaning it will always return the same output for a given input. Inside the function body, the DECLARE credit_limit FLOAT; statement initializes a variable to store the retrieved credit limit. The SELECT creditLimit INTO credit_limit FROM customers WHERE customerNumber = customer_id; query fetches the credit limit for the specified customer. Finally, the function returns the stored value using RETURN credit_limit;. To test and use the function, the SELECT GetCreditLimit(103) AS CreditLimit; statement is executed, retrieving the credit limit of the customer with customerNumber = 103. However, to filter customers with a credit limit of less than 96,800, an additional condition (WHERE creditLimit < 96800) should be included when calling the function within a query.

```
575 -- SELECT creditLimit INTO credit_limit FROM customers WHERE customerNumber = customer_id; query fetches the credit li
576 -- the function returns the stored value using RETURN credit_limit
577 -- the SELECT GetCreditLimit(103) AS CreditLimit; statement is executed, retrieving the credit limit of the customer v
578 DELIMITER //
579 • CREATE FUNCTION GetCreditLimit(customer_id INT) RETURNS FLOAT
580 DETERMINISTIC
581 BEGIN
582     DECLARE credit_limit FLOAT;
583     SELECT creditLimit INTO credit_limit FROM customers WHERE customerNumber = customer_id; -- Retrieves credit limit
584     RETURN credit_limit; -- Returns the credit limit
585 END //
586 DELIMITER ;
587
588 • -- Use function
589 SELECT GetCreditLimit(103) AS CreditLimit; -- Calls the function for customer 103
```



The screenshot shows a SQL IDE interface. At the bottom, there is a 'Result Grid' tab. Below it, a table with one row is displayed. The table has a column header 'CreditLimit' and a single data row with the value '21000'.

CreditLimit
21000

Fig. 10.0

Question 11

Create Trigger to store transaction record for employee table which displays employeeNumber, lastName, FirstName and office code upon insertion

Solutions:

Creating Trigger to Store Transaction Record

The below screenshot shows SQL script that creates a trigger named AfterEmployeeInsert that automatically records transaction details when a new employee is inserted into the employees table. The trigger is defined using the CREATE TRIGGER statement and is set to execute AFTER INSERT, meaning it activates only after a new row is added to the employees table. The FOR EACH ROW clause ensures that the trigger executes for every new employee entry. Inside the BEGIN...END block, an INSERT INTO employee_transactions statement is used to store the newly inserted employee's details (employeeNumber, lastName, firstName, and officeCode) in the employee_transactions table. The NEW keyword is used to reference the newly inserted row's values. The delimiter // is used to handle the multi-statement block and is reset after the trigger definition. This trigger ensures that every employee insertion is automatically logged for tracking purposes.

```
399 -- Create trigger
300 -- creates a trigger named AfterEmployeeInsert that automatically records transaction details when a new employee is inserted into the employees table
301 -- The trigger is defined using the CREATE TRIGGER statement and is set to execute new employees row is added
302 -- FOR EACH ROW clause ensures that the trigger executes for every new employee entry
303 -- an INSERT INTO employee_transactions statement is used to store the newly inserted employee's details (employeeNumber, lastName, firstName, and officeCode)
304 -- NEW keyword is used to reference the newly inserted row's values
305 -- DELIMITER // is used to handle the multi-statement block and is reset after the trigger definition
306 DELIMITER //
307 • CREATE TRIGGER AfterEmployeeInsert
308 AFTER INSERT ON employees
309 FOR EACH ROW
310 BEGIN
311     INSERT INTO employee_transactions (employeeNumber, lastName, firstName, officeCode)
312     VALUES (NEW.employeeNumber, NEW.lastName, NEW.firstName, NEW.officeCode); -- Inserts transaction record
313 END //
314 DELIMITER ;
```

Fig. 11.0

Question 12:

Create a Trigger to display customer number if the amount is greater than 10,000

Solutions:

Creating A Trigger to Display Customer Number

The screenshot below contains an SQL script that defines a trigger named **AfterPaymentInsert** that automatically logs high-value payments into a separate table called **high_value_payments**. The trigger is executed AFTER INSERT on the payments table, meaning it activates only after a new payment record is added. The FOR EACH ROW clause ensures the trigger runs for every new row inserted.

Inside the **BEGIN...END** block, an **IF condition** checks if the amount of the newly inserted payment exceeds 10,000. If the condition is met, an **INSERT INTO high_value_payments** statement is executed, storing the **customerNumber** and **amount** of that transaction in the **high_value_payments** table. The **NEW keyword** is used to reference the values of the newly inserted row.

The **delimiter //** is used to manage the trigger definition since MySQL treats semicolons (;) as the default statement terminator. After defining the trigger, the delimiter is reset to its default (;).

This trigger ensures that all payments above 10,000 are automatically recorded in a separate table for tracking or reporting purposes.

```
618                                     QUESTION 12
619  Create a Trigger to display customer number if the amount is greater than 10,000
620  */
621                                     -- SOLUTION TO QUESTION 12
622  -- Create a trigger to display customer number if the amount is greater than 10,000
623  DELIMITER // -- manages the trigger definition
624  CREATE TRIGGER AfterPaymentInsert -- Trigger name
625  AFTER INSERT ON payments          -- Ensures trigger executes AFTER INSERT on the payments table
626  FOR EACH ROW                     -- ensures the trigger runs for every new row inserted
627  BEGIN
628      IF NEW.amount > 10000 THEN -- checks if the amount of the newly inserted payment exceeds 10,000
629          INSERT INTO high_value_payments (customerNumber, amount) -- identifies the column to be used for the insertion
630          VALUES (NEW.customerNumber, NEW.amount); -- Inserts record if amount > 10000
631      END IF;
632  END //
633  DELIMITER ; -- reset the delimiter to default (;).
```

Fig. 12.0

Question 13

Create Users, Roles and Logins according to 3 Roles: Admin, HR, and Employee. Admin can view full database and has full access, HR can view and access only employee and offices table. Employee can view all tables only.

Note: work from Admin role for any changes to be made for database

Solutions:

Creating Users, Roles and Logins

The screenshot shows the SQL scripts used to establish role-based access control for the MotorsCertification database by creating roles, granting privileges, and assigning these roles to specific users. First, it defines three roles: **Admin**, **HR**, and **Employee**. The admin role is given full access to all database operations, allowing complete control over every table. The HR role is granted permissions to manage employee and office records by allowing SELECT, INSERT, UPDATE, and DELETE actions on the employees and offices tables. The Employee role is restricted to read-only access across all tables, ensuring that employees can view information but cannot modify it. After defining the roles, the script proceeds to create three users: admin_user, hr_user, and employee_user, each with a unique password. These users are then assigned their respective roles, ensuring that admin_user has full control, hr_user can manage employee-related records, and employee_user has read-only access. This structured access control system enhances database security by limiting user privileges based on their roles, ensuring that only authorized personnel can perform specific operations.

```
641 -- SOLUTION TO QUESTION 13
642 -- CREATE ROLES FOR USER TYPES
643 CREATE ROLE Admin; -- Create Admin role (Has full access to the database)
644 CREATE ROLE HR; -- Create HR role (Can access only employees and offices tables)
645 CREATE ROLE Employee; -- Create Employee role (Can view all tables but cannot modify them)
646 -- ASSIGN PERMISSIONS TO EACH ROLE
647 GRANT ALL PRIVILEGES ON MotorsCertification.* TO Admin; -- Grant full access to Admin (Can read, write, update, delete everything)
648 GRANT SELECT, INSERT, UPDATE, DELETE ON MotorsCertification.employees TO HR;-- Grant HR role permission to manage employees tables
649 GRANT SELECT, INSERT, UPDATE, DELETE ON MotorsCertification.offices TO HR;-- Grant HR role permission to manage offices tables
650 GRANT SELECT ON MotorsCertification.* TO Employee; -- Grant Employee role read-only access to all tables
651 -- CREATE USERS AND SET ROLES
652 CREATE USER 'admin_user'@'%' IDENTIFIED WITH mysql_native_password BY 'admin_password'; -- Create Admin user with password (Modify as needed)
653 CREATE USER 'hr_user'@'%' IDENTIFIED WITH mysql_native_password BY 'hr_password'; -- Create HR user with password (This is the HR login credentials)
654 CREATE USER 'employee_user'@'%' IDENTIFIED WITH mysql_native_password BY 'employee_password';-- Create Employee user with password
655 GRANT Admin TO 'admin_user'@'%;-- Assign Admin role to 'admin_user' (Gives full access)
656 GRANT HR TO 'hr_user'@'%;-- Assign HR role to 'hr_user' (Grants limited access)
657 -- Assign Employee role to 'employee_user' (Grants read-only access)
658 GRANT Employee TO 'employee_user'@'%;
659 -- SET DEFAULT ROLE FOR USERS (Ensures users activate their assigned roles automatically)
660 SET DEFAULT ROLE Admin FOR 'admin_user'@'%;
661 SET DEFAULT ROLE HR FOR 'hr_user'@'%;
662 SET DEFAULT ROLE Employee FOR 'employee_user'@'%;
```

Fig. 13.0

Question 14

Schedule a Job which backups and schedule it according to developer preference.

Solutions:

Scheduled Job Which Backups and Schedule It.

Since MySQL does not necessary support scheduling backup jobs, **MySQL Workbench** to create the backup script and then schedule it using Windows operating system's task scheduler.

Below are the steps I followed to create the backup script and scheduling using the operating system's task scheduler:

Create a Backup Script:

- I opened MySQL Workbench.
- I selected **Server** from the ribbons and navigated to **Data Export**.
- I select the database (MotorsCertification) and including the tables to back up.
- I choose the export option (export to a self-contained file).
- I then save the export file as a MotorsCertification.sql.

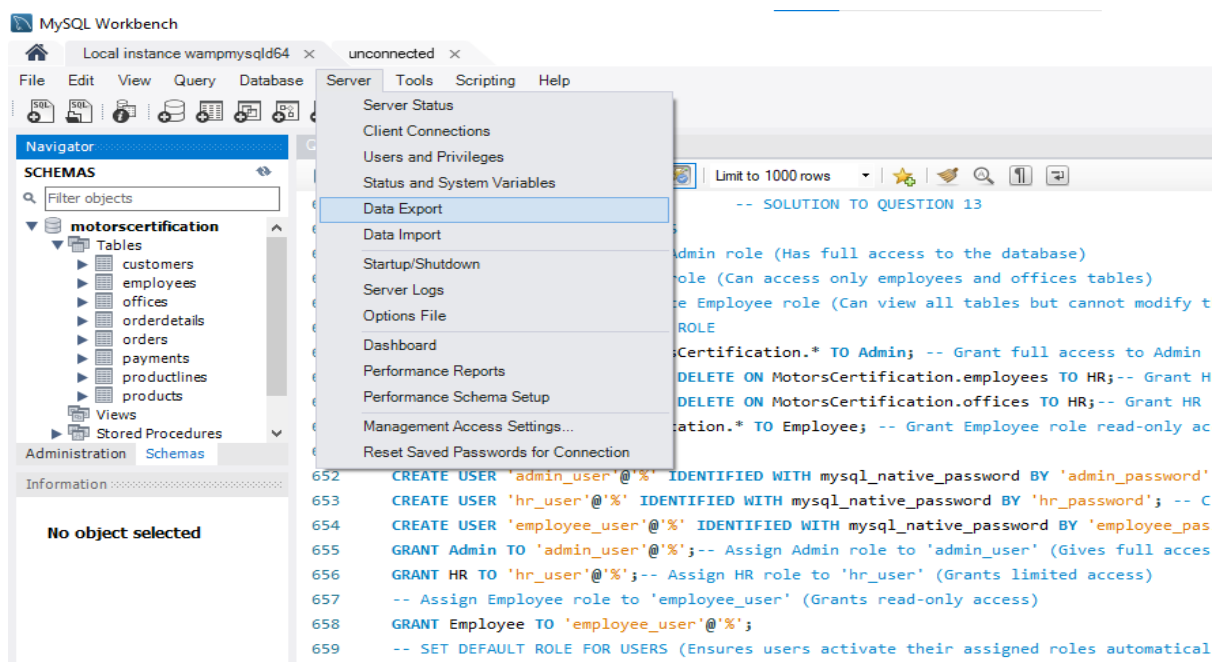


Fig. 14.0

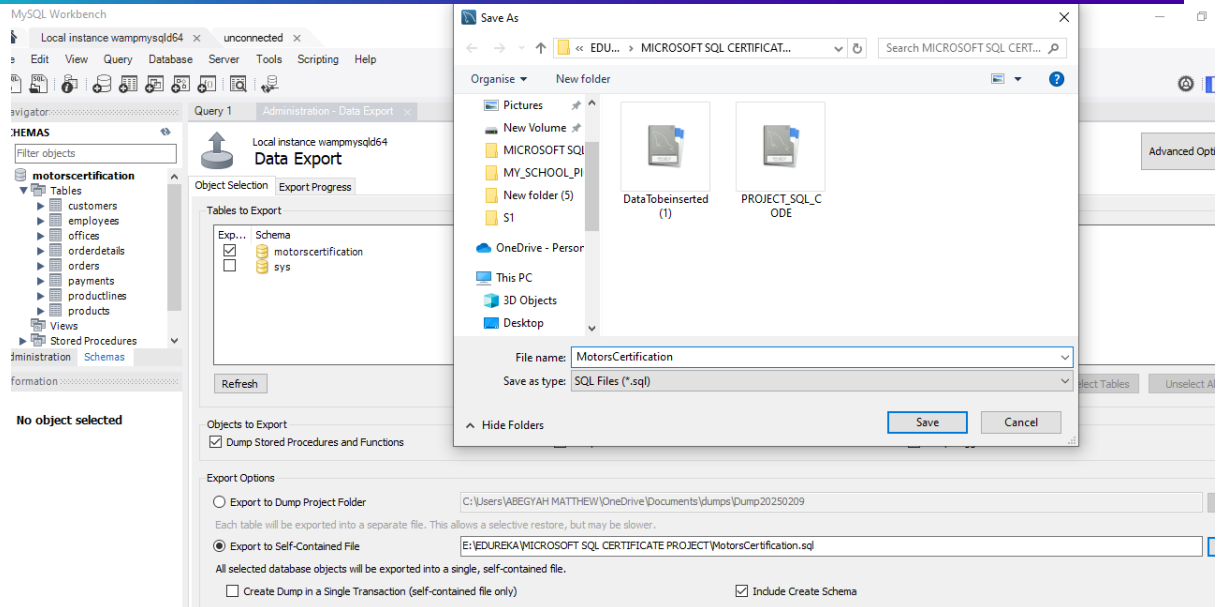


Fig. 14.1

Schedule the Backup Script:

- I opened **Task Scheduler**.
- A new task was created and set to run the **mysqldump** command at a specific time (mysqldump -u [root]-p[] MotorsCertification > E:\EDUREKA\MICROSOFT SQL CERTIFICATE PROJECT\MotorsCertification.sql).
- The command was kept in notepad and saved with a **.bat** extension.
- In the Actions pane on the right, I clicked on **Create Basic Task**.
- In the Name field, I entered **MySQL Database Backup** as a name for the task and clicked **Next** to continue.
- Under the trigger, I selected the starting and ending date and clicked **Next**.
- I then selected **Start a Program** and click Next.
- In the Program/script field, I browsed to the .bat file I created earlier (E:\EDUREKA\MICROSOFT SQL CERTIFICATE PROJECT\MotorsCertification.sql) to complete backup schedule.

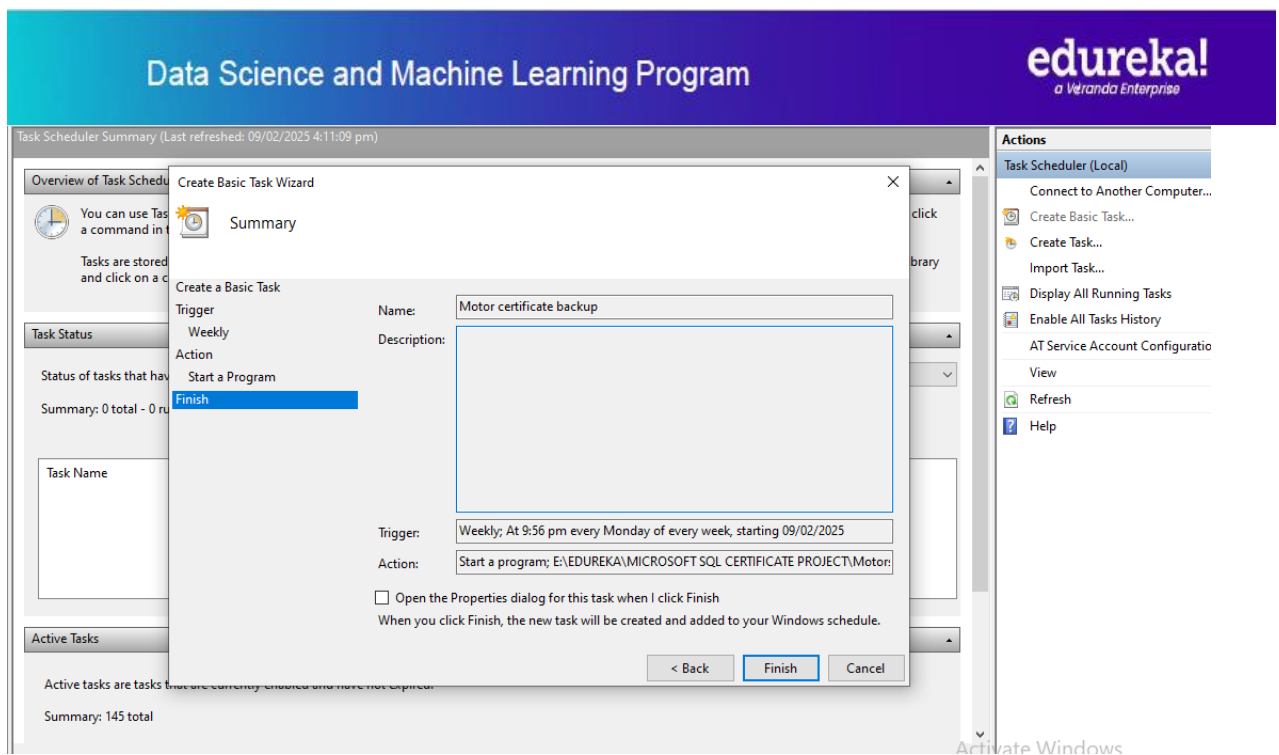


Fig. 14.2

Question 15

Open Activity Monitor and list down some minor observations including Processes, Resource Waits, and Active Expensive Queries.

Solutions:

Steps Involve in Opening Activity Monitor in My SQL

- I opened MySQL Workbench and connected to the MySQL server.
- I navigated to the Performance tab in the sidebar.
- I accessed the Dashboard: Provides an overview of server performance, including CPU usage, memory usage, and active connections.
- Checked Query Statistics: which shows the most expensive queries and their execution times.
- Client Connections: Displays active connections to the database.

The screenshot below shows some minor activities such as **network traffic**, **client connection**, and others from the dashboard:

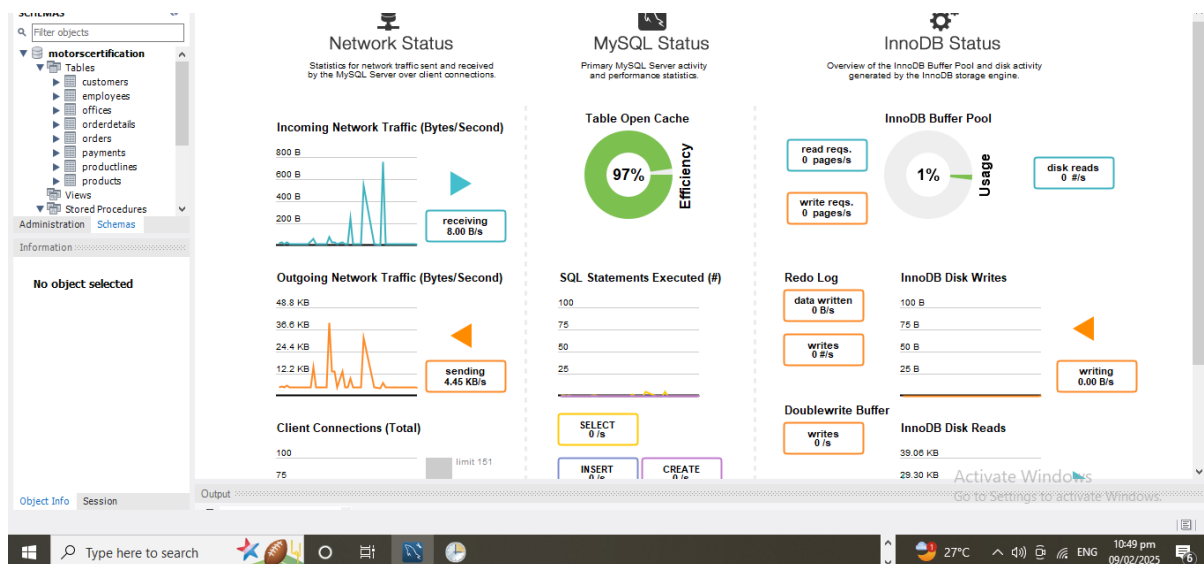


Fig. 15.0

Question 16

Migrate The Following SQL Server Workload to Azure.

Solutions:

Steps involve in migrating the MySQL server workload to azure.

- I logged in to the Azure Portal.
- Navigated to Create a Resource > Databases > Azure Database for MySQL.
- I configure the server Create the database.
- I then used the Azure MySQL Workbench to import the MotorsCertification.sql, the file I exported MySQL workbench, into Azure Database for MySQL.

Conclusion

The **MotorsCertification** database project successfully addressed the requirements of HerculesMotoCorp by designing a comprehensive ER model, creating and populating the necessary tables, and implementing advanced database functionalities such as views, stored procedures, triggers, and user roles. The project also included tasks like data validation, backup scheduling, and performance monitoring, ensuring the database is robust, secure, and efficient. Additionally, the migration of the database workload to Azure was explored, providing a scalable and cloud-based solution for future growth. The project deliverables, including the SQL queries, ER diagram, and backup file, demonstrate a well-structured and functional database system tailored to meet the needs of HerculesMotoCorp and its stakeholders.