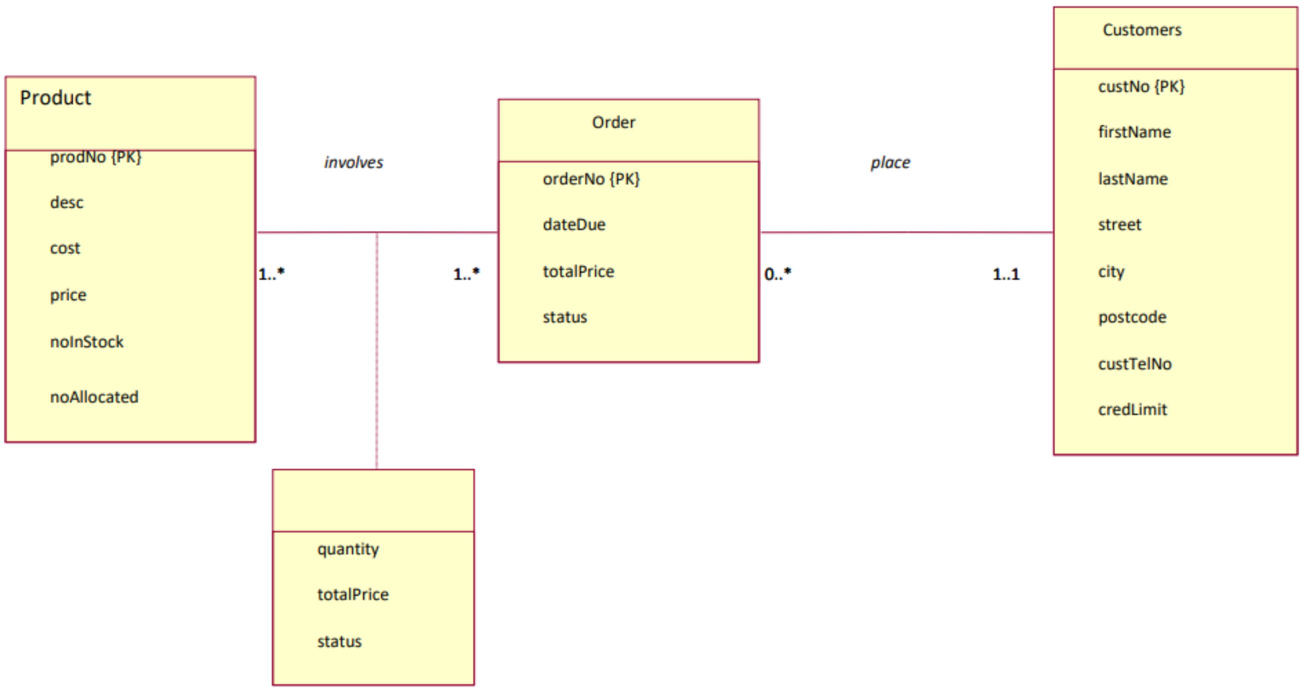
Practice 1: Derive Relations



SCHEMA (PRATICE 1)	
Product(prodNo, description, cost, price, noInStock, noAllocated) **Primary Key:** prodNo	Order(orderNo, dateDue, totalPrice, status, prodNo, custNo) **Primary Key:** orderNo **Foreign Key:** prodNo referencing to Product **Foreign Key:** custNo referencing Customers
Customers(custNo, firstName, lastName, street, city, postcode, custTelNo, credLimit) **Primary Key:** custNo	OderItem(orderNo, prodNo**,** quantity) **Foreign Key**: orderNo referencing Order **Foreign Key:** prodNo referencing Product

The relationships can be summarized as follows:

1. Product (1..) to Order (1..): Each product can be included in multiple orders, and each order can include multiple products.

2. Order (0..*) to Customers (1..1): Each order belongs to a single customer, and each customer can have zero or more orders.

3. Product (1) to OrderItem (1..*) Each Product can be in cluded in one or more OrderItem.

4. Order (1) to OrderItem (1..*) Each Order can have one or more OrderItem associated with it.

The line that descends without a table name represents additional attributes related to the relationship between Product and Order, where generating a **separate table** for the quantity of each element in a customer order allows for a more **organised** and **normalised** database design, improving data **integrity**, **flexibility**, and **relationship clarity**. Because an **order** may contain **many products** with **varying quantities**, a separate database or data structure may be required to handle that scenario. As a result, a table called **OderItem** is **formed**, as illustrated above. The OrderItem table can be used to get specific products and quantities for each order, whereas the totalPrice and status can be accessed straight from the Order table.

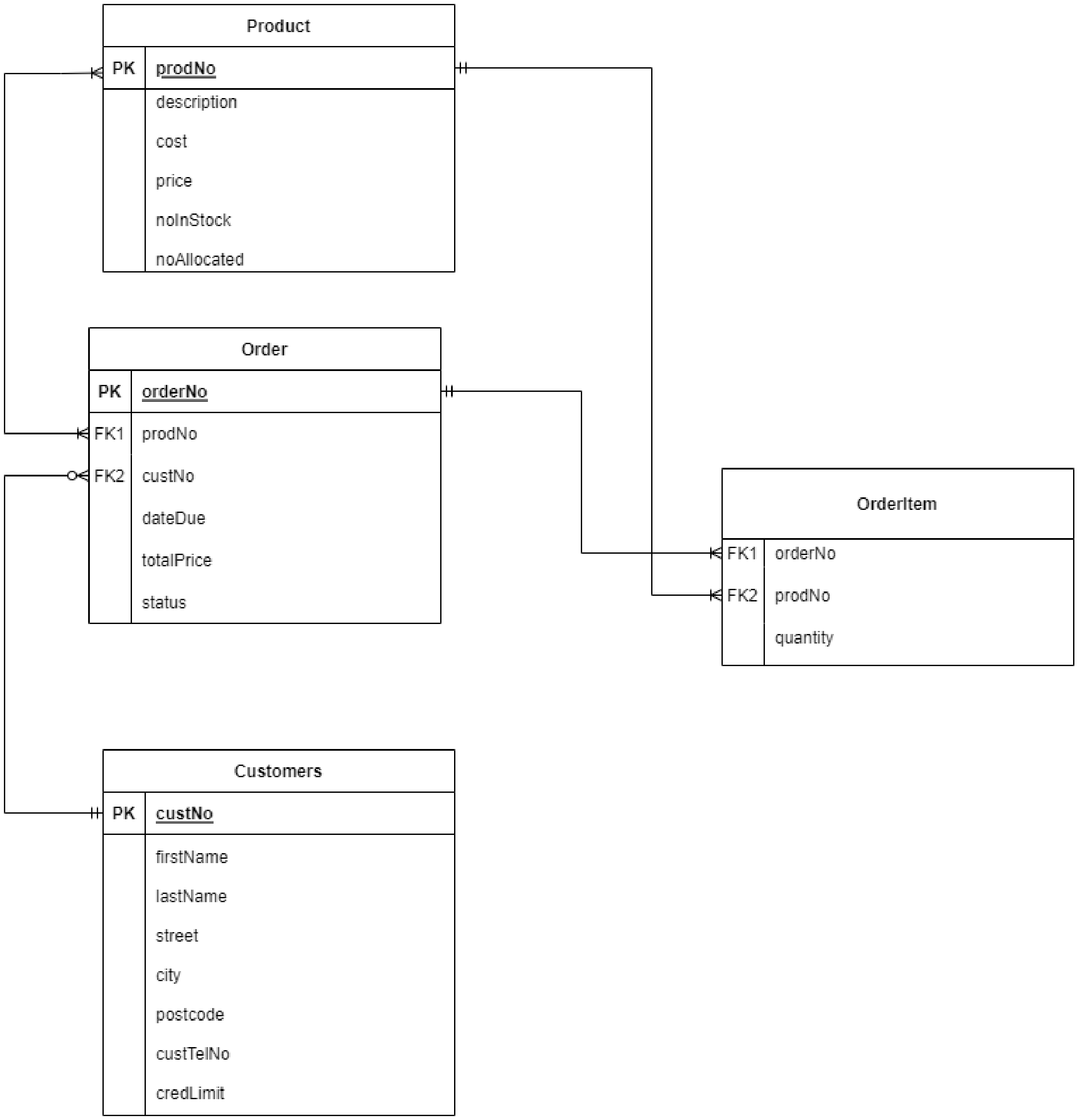
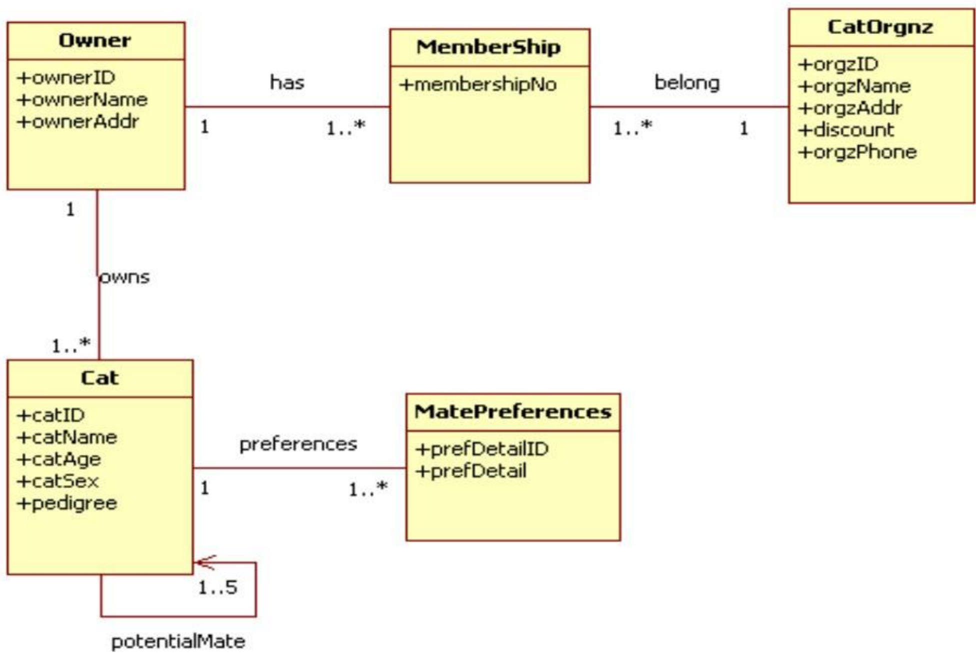


Figure 1: ERD PRACTICE 12:

Practice 2: Derive Relations



SCHEMA (PRATICE 2)	
Owner(ownerID, ownerName, ownerAddr) **Primary Key:** ownerID	Membership(membershipNo, ownerID, orgzID) **Primary Key:** membershipNo **Foreign Key:** ownerID referencing to Owner **Foreign Key:** orgzID referencing to CatOrgnz
CatOrgnz(orgzID, orgzName, orgzAddr, discount, orgzPhone) **Primary Key:** orgzID	Cat(catID, catName, catAge, catSex, pedigree, ownerID, potentialMate (values 1 to 5)) **Primary Key:** catID **Foreign Key:** ownerID referencing to Owner **Foreign Key:** potentialMate referencing to Cat.catID
MatePreferences(prefDetailID, prefDetail, catID) **Primary Key:** prefDetailID **Foreign Key:** catID referencing to Cat	

The relationships can be summarized as follows:

1. Owner (1) to Membership (1..*): An owner can have multiple memberships, but each membership belongs to only one owner.

2. Membership (1..*) to CatOrgnz (1): A membership belongs to one cat organization.

3. Owner (1) to Cat (1..*): An owner can have multiple cats, but each cat is owned by only one owner.

4. Cat (1) to MatePreferences (1..*): Each cat can have multiple mate preferences.

5. Cat to itself (1..5): Each cat can have up to five potential mates, forming a self-referencing relationship.

- The self-referencing relationship between Cat and itself can be implemented by adding a foreign key column in the Cat table that references the catID of another cat.

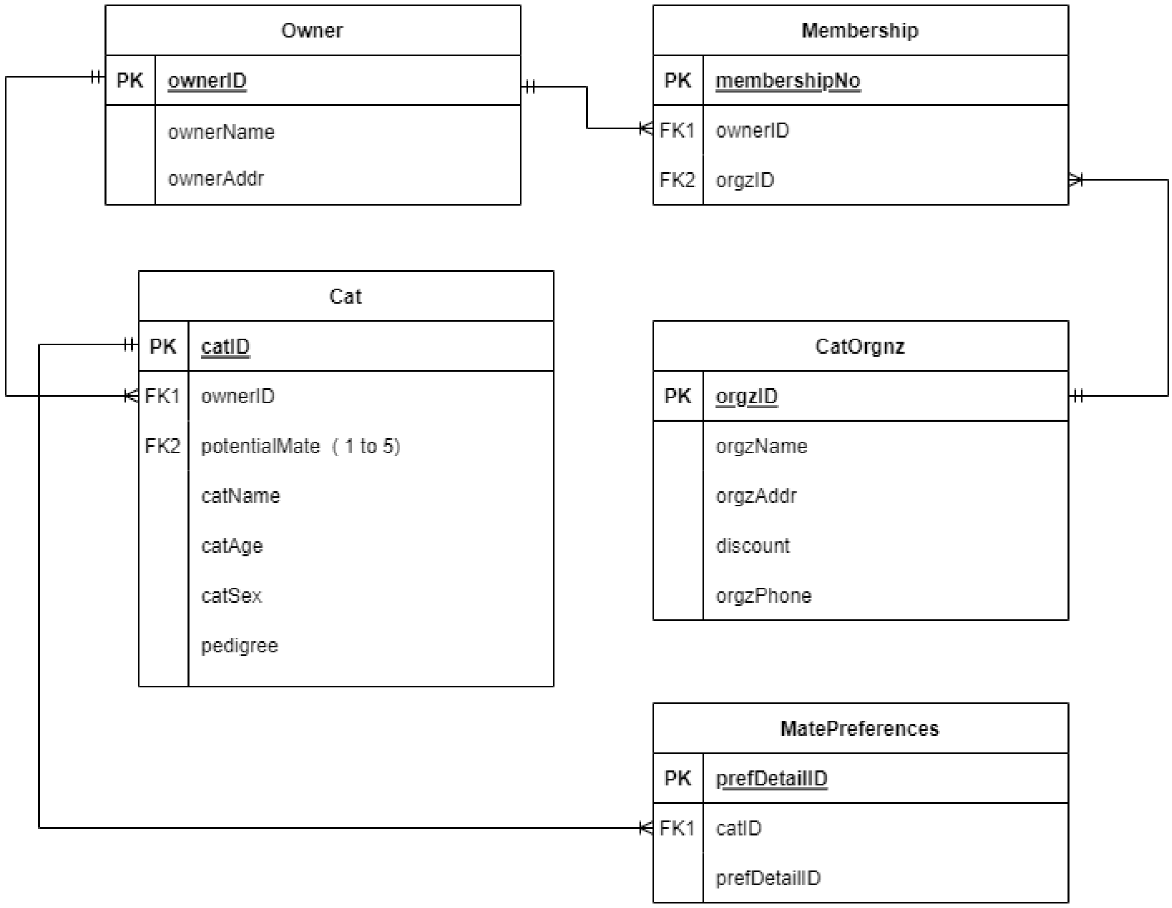
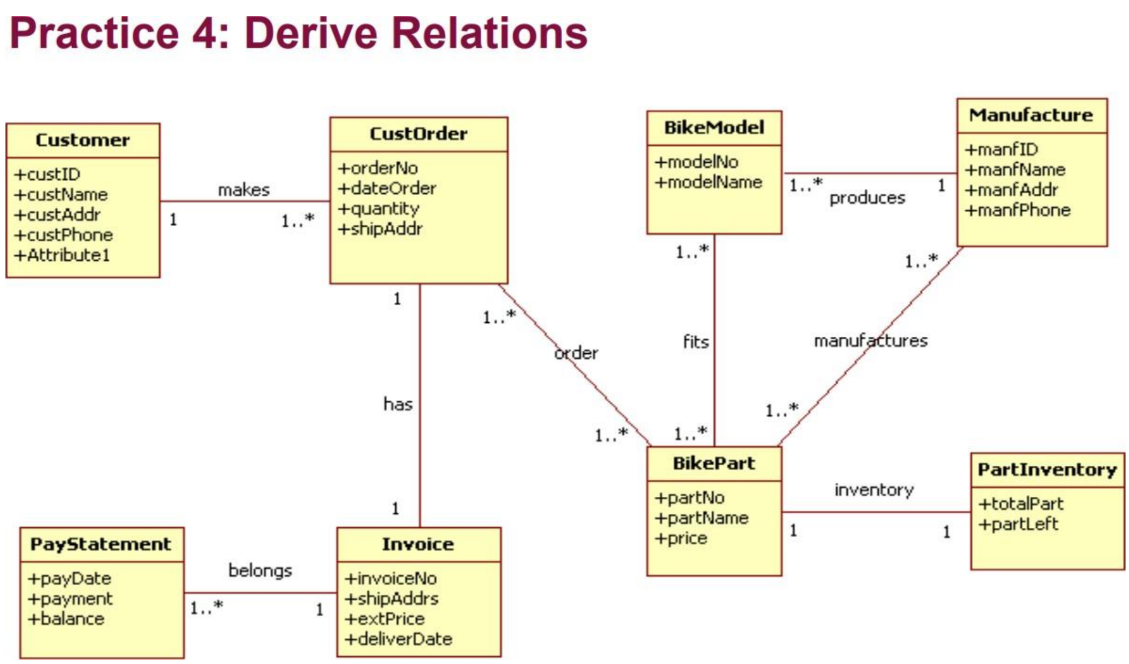


Figure 2: ERD PRACTICE 2

PRACTICE 4:



SCHEMA (PRATICE 4)	
Customer(custID, custName, custAddr, custPhone, Attribute1) **Primary Key:** custID	CustOrder(orderNo, dateOrder. shipAddr, custID) **Primary Key:** orderNo **Foreign Key:** custID referencing to Customer
BikeModel(modelNo, modelName) **Primary Key:** modelNo	PayStatement(payDate, payment, balance) **Primary Key:** payDate
Manufacture(manfID, manfName, manfAddr, manfPhone, modelNo) **Primary Key:** manfID **Foreign Key:** modelNo referencing to BikeModel	BikePart(partNo, partName, price, modelNo, manfID) **Primary Key:** partNo **Foreign Key:** modelNo referencing to BikeModel **Foreign Key:** manfID referencing to Manufacture
PartInventory(totalPart, partLeft, partNo) **Foreign Key:** partNo referencing to BikePart	Invoice(invoiceNo, shipAddrs, extPrice, deliverDate, orderNo, payDate) **Primary Key:** invoiceNo **Foreign Key:** orderNo referencing to CustOrder **Foreign Key:** payDate referencing to PayStatement
OrderParts(orderNo, partNo, quantity) **Foreign Key:** orderNo referencing to CustOrder **Foreign Key:** partNo referencing to BikePart	

The relationships described in the schema are as follows:

1. Customer (1) to CustOrder (1..*): Each customer can have multiple orders. One customer can place one or more orders.

2. CustOrder (1) to Invoice (1): Each order is associated with one invoice. One order corresponds to one invoice.

3. CustOrder (1..*) to BikePart (1..*): Each order can have multiple bike parts. One order can include one or more bike parts, and each bike part can appear in multiple orders.

4. BikePart (1..*) to BikeModel (1..*): One bike part is associated with multiple bike models, and each bike model can have multiple bike parts.

5. BikeModel (1..*) to Manufacture (1): Each bike model is manufactured by one manufacturer. One bike model is associated with one manufacturer, and each manufacturer can produce multiple bike models.

6. Manufacture (1..*) to BikePart (1..*): Each manufacturer can produce multiple bike parts. One manufacturer is associated with one or more bike parts, and each bike part can be produced by one manufacturer.

7. BikePart (1) to PartInventory (1): Each bike part is associated with one inventory. One bike part corresponds to one inventory.

8. PayStatement (1..*) to Invoice (1): Each invoice is linked to one payment statement. One invoice has one associated payment statement, and each payment statement can be linked to multiple invoices.

9. CustOrder (1) to OrderParts (1..*) Each CustOrder can have one or more OrderParts associated with it.

10. BikePart (1) to OrderParts (1..*) Each BikePart can be included in one or more OrderParts.

Because an **order** can contain **many components** with **varied quantities**, a **new table**, such as **OrderParts**, would be useful to express the **many-to-many** link between **CustOrder** and **BikePart**. The foreign keys referencing the CustOrder and BikePart tables, as well as the amount property, would then be included in the OrderParts database. This provides **greater flexibility** and **scalability** when dealing w

ith orders containing several pieces.

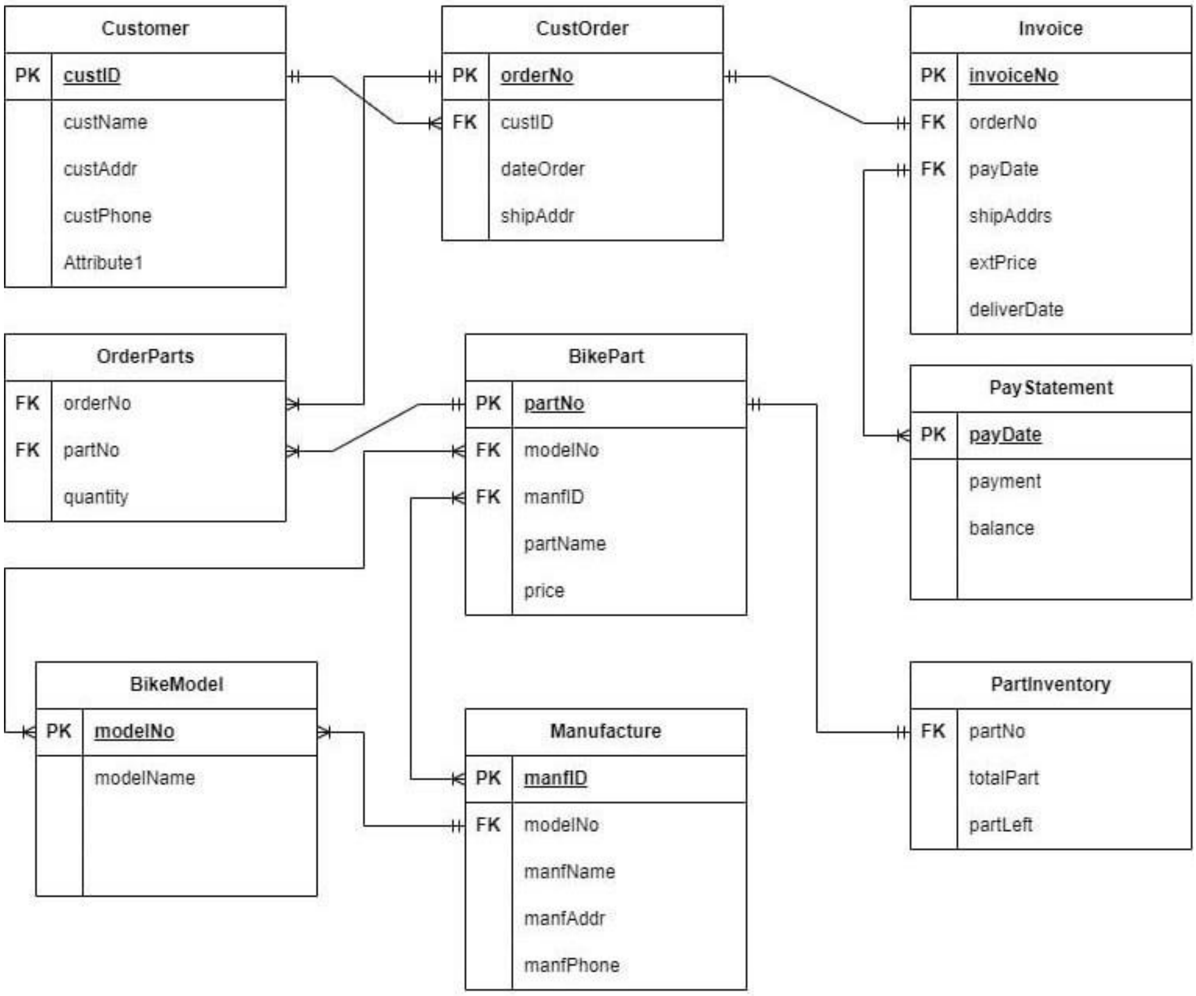


Figure 3: ERD PRACTICE 4

QUESTION 2: SQL Queries

a)

Create a database and construct 3 tables as following SQL DDL. Create a suitable primary key and foreign key to link the tables

Student table

Name name	Data type
Id	Integer
name	Text

Term_gpa table

Column name	Data type
Id	integer
term	integer
gpa	float

Degree table

Column name	Data type
Id	integer
term	integer
degree	char

and insert the following data in each of table

Student table

id	Name
1	William
2	Kate
3	Hisham
4	Ahmad
5	Hussien

Term_GPA table

id	Term	GPA
1	2011	3.32
1	2012	3.51
2	2011	2.22
2	2013	1.7
3	2011	3.70
4	2011	3.10
4	2012	3.21
4	2013	3.30
5	2013	2.99

Degree table

id	Term	Degree
1	2012	Econ
3	2011	Math
3	2011	Comp
4	2012	Eng

```sql
CREATE DATABASE student_info;

USE student_info;

-- Create the Student table
CREATE TABLE Student (
  id INT PRIMARY KEY,
  name VARCHAR(50)
);

-- Create the Term_GPA table
CREATE TABLE Term_GPA (
  id INT,
  term INT,
  gpa FLOAT,
  FOREIGN KEY (id) REFERENCES Student(id)
);

-- Create the Degree table
CREATE TABLE Degree (
  id INT PRIMARY KEY,
  term INT,
  degree CHAR(10),
  FOREIGN KEY (id) REFERENCES Student(id)
);
```

Figure 4: Creating Database

```sql
-- Insert data into the Student table
INSERT INTO Student (id, name)
VALUES (1, 'William'),
       (2, 'Kate'),
       (3, 'Hisham'),
       (4, 'Ahmad'),
       (5, 'Hussien');

-- Insert data into the Term_GPA table
INSERT INTO Term_GPA (id, term, gpa)
VALUES (1, 2011, 3.32),
       (1, 2012, 3.51),
       (2, 2011, 2.22),
       (2, 2013, 1.7),
       (3, 2011, 3.7),
       (4, 2011, 3.1),
       (4, 2012, 3.21),
       (4, 2013, 3.3),
       (5, 2013, 2.99);


-- Insert data into the Degree table
INSERT INTO Degree (id, term, degree)
VALUES (1, 2012, 'Econ'),
       (2, 2011, 'Math'),
       (3, 2012, 'Comp'),
       (4, 2011, 'Eng');
```

Figure 5: Input values into table

b) Run appropriate query to obtain student named Ahmad for every term.

```sql
#b)
SELECT Student.id, Student.name, Term_GPA.term, Term_GPA.gpa
FROM Student
JOIN Term_GPA ON Student.id = Term_GPA.id
WHERE Student.name = 'Ahmad';
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	term	gpa
4	Ahmad	2011	3.1
4	Ahmad	2012	3.21
4	Ahmad	2013	3.3

Figure 6: Ahmad GPA

c) i) Create view called performance_2011 that keep the information on student performance in 2011.


```sql
#c)
CREATE VIEW performance_2011 AS
SELECT Student.id, Student.name, Term_GPA.gpa
FROM Student
JOIN Term_GPA ON Student.id = Term_GPA.id
WHERE Term_GPA.term = 2011;
```

Figure 7: Creating view

id	name	gpa
1	William	3.32
2	Kate	2.22
3	Hisham	3.7
4	Ahmad	3.1

Figure 8: Showing the view

ii) Run a query to obtain student with highest GPA.

```sql
SELECT Student.id, Student.name, Term_GPA.gpa
FROM Student
JOIN Term_GPA ON Student.id = Term_GPA.id
WHERE Term_GPA.gpa = (SELECT MAX(gpa) FROM Term_GPA);
```

id	name	gpa
3	Hisham	3.7

Figure 9: Highest GPA

d) Add column called **status** in the query to the view created in (b) that will state the following status given if the GPA id as follows:

GPA	Status
>3.5	Excellent
3.0 <GPA<3.5	Very Good
2.7 <GPA<3.0	Good
2.0<GPA<2.7	Satisfactory
< 2	Fail

```
#d)
CREATE OR REPLACE VIEW performance_2011 AS
SELECT Student.id, Student.name, Term_GPA.term, Term_GPA.gpa,
    CASE
        WHEN Term_GPA.gpa > 3.5 THEN 'Excellent'
        WHEN Term_GPA.gpa > 3.0 THEN 'Very Good'
        WHEN Term_GPA.gpa > 2.7 THEN 'Good'
        WHEN Term_GPA.gpa > 2.0 THEN 'Satisfactory'
        ELSE 'Fail'
    END AS status
FROM Student
JOIN Term_GPA ON Student.id = Term_GPA.id
WHERE Term_GPA.term = 2011;
```

Figure 10: Add column to view

id	name	term	gpa	status
1	William	2011	3.32	Very Good
2	Kate	2011	2.22	Satisfactory
3	Hisham	2011	3.7	Excellent
4	Ahmad	2011	3.1	Very Good

Figure 11: Showing the view

e) Used appropriate join command to join table degree, student and term_gpa.

```sql
#e)
SELECT Student.id, Student.name, Term_GPA.gpa, Degree.degree
FROM Student
LEFT JOIN Term_GPA ON Student.id = Term_GPA.id
LEFT JOIN Degree ON Student.id = Degree.id AND Term_GPA.term = Degree.term;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	gpa	degree
1	William	3.32	NULL
1	William	3.51	Econ
2	Kate	2.22	Math
2	Kate	1.7	NULL
3	Hisham	3.7	NULL
4	Ahmad	3.1	Eng
4	Ahmad	3.21	NULL
4	Ahmad	3.3	NULL
5	Hussien	2.99	NULL

Figure 12: Join tables

f) Calculate the average GPA for students who the id in degree table is null.

```sql
#f)
SELECT S.id, S.name, ROUND(AVG(T.gpa), 2) AS average_gpa
FROM Student S
JOIN Term_GPA T ON S.id = T.id
WHERE S.id NOT IN (SELECT id FROM Degree WHERE id IS NOT NULL)
GROUP BY S.id, S.name;
```

Result Grid | Filter Rows: | Export: | Wrap Cell Content:

id	name	average_gpa
5	Hussien	2.99

Figure 13: Average GPA for NULL id