**REPORT DBI ASSIGNMENT - JEWELRY MANAGEMENT**

**\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_**

# ***INTRODUCTION:***

In this assignment, we are going to develop and maintain a jewelry shop, learn how it works and how it operates

**MEMBER: Lê Viết Mạnh, Đoàn Trọng Lực, Lê Đức Sự, Văn Minh Kiệt**

**( All sql files for references are in this link )**

**\*\*GitHub:** [**https://github.com/v1etmanh/final\_sql.git**](https://github.com/v1etmanh/final_sql.git)

In managing the jewelry shop, my group will:

# **Part 1: Planning and Design**

## **1)Identify business requirements and gather necessary data.**

**Customers**: Stores customer information and tracks purchase history.

**Jewelry**: Contains details of jewelry products available in the shop.

**Category**: Classifies jewelry into specific categories for organization.  
**Orders**: Records customer purchases and order details.

**OrderDetails**: Tracks the jewelry items, quantities, and prices in each order.  
**Suppliers**: Maintains information about jewelry suppliers.  
**Supplier\_Address**: Stores detailed address information for suppliers.

**Employees**: Stores employee details and tracks their order management.

**Warranty**: Records warranty details for jewelry items.  
**Service**: Tracks jewelry repair or maintenance services provided to customers.  
**JewelryPriceHistory**: Maintains a history of price changes for jewelry items.

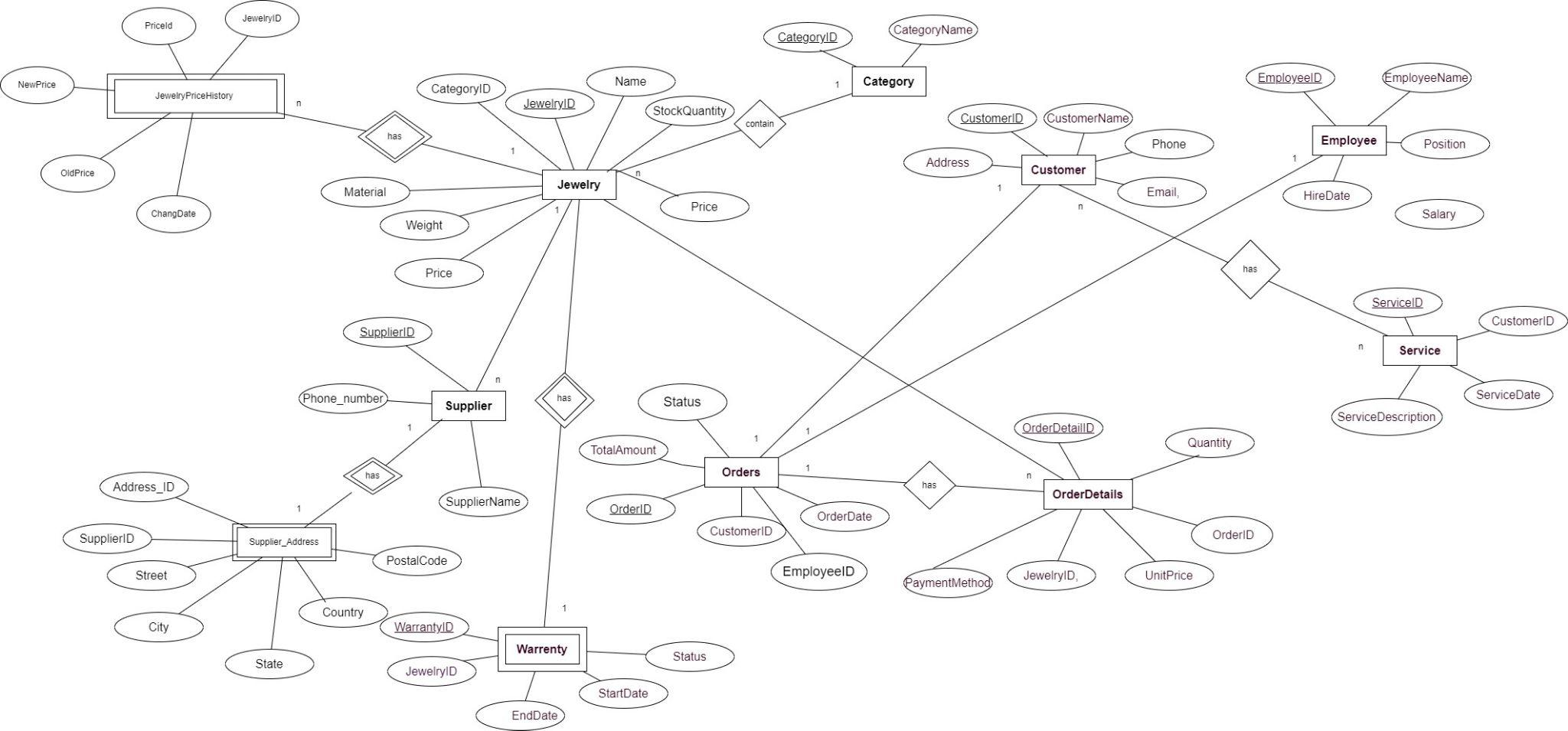
## **2)Design the ERD (Entity-Relationship Diagram)**:

Identify key entities like **Customers**, **Orders**,**Jewelry**, **Employees**, **Suppliers**, ,.....

Define relationships (e.g., **Customers** place **Orders**, **OderDetails** contains **JewelryID,........**).

Establish attributes (e.g., product name, price, order date).

( next page )



Relationship of this ERD:

**Customers - Orders**: One-to-Many, each customer can place multiple orders.p.

**Orders - OrderDetails**: One-to-Many, each order can have multiple order details.

**Jewelry - OrderDetails**: One-to-Many, one jewelry item can be in multiple order details..

**Jewelry - Category**: Many-to-One, jewelry items belong to one category.

**Jewelry - JewelryPriceHistory**: One-to-Many, each jewelry item can have multiple price changes..

**Suppliers - Jewelry**: One-to-Many, one supplier can supply multiple jewelry items.

**Suppliers - Supplier\_Address**: One-to-One, each supplier has one address

**Employees - Orders**: One-to-Many, one employee can manage multiple orders..

**Jewelry - Warranty**: One-to-One, each jewelry item can have one warranty.

**Customers - Service**: One-to-Many, one customer can request multiple services.

**Service - Employees**: Many-to-One, multiple services can be handled by one employee.

There are some weak entities in this ERD such as : Supplier\_Address,Warrenty,JewelryPriceHistory

\*\* this is only the draft of our database management, our first step to develop it.

# Part 2: Database Creation

## 1)**Create the Database in SQL Server**

Implement the ERD by creating tables using **CREATE TABLE** statements

( just a few tables showed here)

CREATE TABLE Supplier (

SupplierID INT PRIMARY KEY identity(1,1),

SupplierName NVARCHAR(50) NOT NULL,

Email nvarchar(100) not null unique,--đảm bảo nó là duy nhất

Phone\_Number nvarchar(15) not null unique,-- đảm bảo là duy nhất

Address nvarchar(255) not null

);

CREATE TABLE Supplier\_Address (

AddressID INT IDENTITY(1,1) PRIMARY KEY,

SupplierID INT NOT NULL,

Street VARCHAR(255) NOT NULL,

City VARCHAR(100) NOT NULL,

State VARCHAR(100) NULL,

Country VARCHAR(100) NOT NULL,

PostalCode VARCHAR(20) NULL,

## 2)Establish **Primary Keys**, **Foreign Keys**, and constraints

Primary Keys (PK)

**SupplierID, AddressID, CategoryID, JewelryID, CustomerID, WarrantyID, EmployeeID, OrderID, ServiceID, PriceID**:  
Each table has a primary key to uniquely identify records.  
Primary keys are often set using IDENTITY(1,1) for auto-increment.

\*\*Composite PK for **OrderDetails** using OrderID and JewelryID.

Foreign Keys (FK)

**SupplierID in Jewelry and Supplier\_Address**: Links jewelry items and supplier addresses to the corresponding supplier.  
**CategoryID in Jewelry**: Connects jewelry items to their categories.  
**CustomerID in Orders, Warranty, and Service**: Tracks which customer made the order, warranty, or service request.  
**JewelryID in OrderDetails and Warranty**: Tracks jewelry items in orders and warranty records.  
**EmployeeID in Orders and Service**: Tracks which employee managed the order or provided service.  
and more….

Constraints

**CHECK Constraints**:  
 **Price >= 0**: Ensures no negative jewelry prices.  
 **Quantity >= 0**: Prevents negative order quantities.  
 **TotalAmount >= 0**: Ensures total order amount is non-negative.  
 **EndDate >= StartDate**: Guarantees valid warranty periods.  
 **Warranty.Status** (Only allows ACTIVE or EXPIRED)  
 **Orders.Status** (Only allows PENDING, COMPLETED, or CANCELED**)**

**JewelryPriceHistory.OldPrice, NewPrice** (Ensures both are positive)

**Service.ServicePrice >= 0**

**Warranty.EndDate >= Warranty.StartDate**

**UNIQUE Constraints**:  
 Applied to **Supplier Email, Phone Number**, and **Customer Phone Number, TaxCode,Supplier\_Address.SupplierID**  to ensure no duplicates.

**DEFAULT Constraints**:  
 **GETDATE()**: Automatically sets the current date for order dates and price change dates.

# Part 3: Data Management

## 1)Perform CRUD operations using **INSERT**, **SELECT**, **UPDATE**, and **DELETE**.

use jewelry\_management

SET IDENTITY\_INSERT Supplier ON;

INSERT INTO Supplier (SupplierID, SupplierName, Email, Phone\_Number)

VALUES

(1, 'Golden Gems Supplier', 'contact@goldengems.com', '0987654321'),

(2, 'Precious Stones Inc.', 'sales@preciousstones.com', '0123456789'),

(3, 'Diamond World', 'info@diamondworld.com', '0345678912'),

(4, 'Luxury Jewels Co.', 'sales@luxuryjewels.com', '0456789123'),

(5, 'Royal Gems Enterprise', 'contact@royalgems.com', '0567891234'),

INSERT INTO Supplier\_Address (SupplierID, Street, City, State, Country, PostalCode)

VALUES

(1, '123 Jewelry Street', 'Ho Chi Minh City', 'Long Ba Phu Tho', 'Vietnam', '700000'),

(2, '456 Gem Avenue', 'Hanoi', 'Ba Dinh', 'Vietnam', '100000'),

(3, '789 Diamond Road', 'Da Nang', 'Hai Chau', 'Vietnam', '550000'),

(4, '321 Luxury Lane', 'Can Tho', 'Ninh Kieu', 'Vietnam', '900000'),

(5, '654 Royal Street', 'Hue', 'Thua Thien', 'Vietnam', '530000'),

Notes: \*\*All the insert query included in the insert.sql files in github link at the beginning

SET IDENTITY\_INSERT is used to manually insert data into tables with identity columns (SupplierID, CategoryID, etc.).

## 2)Creating procedure,trigger,function to operate our database

**Trigger**

#### 1) Trigger1: trg\_Jewelry\_Price\_Update

CREATE TRIGGER trg\_Jewelry\_Price\_Update

ON Jewelry

AFTER UPDATE

AS

BEGIN

IF UPDATE(Price)

BEGIN

INSERT INTO JewelryPriceHistory (JewelryID, OldPrice, NewPrice, ChangeDate)

SELECT d.JewelryID, d.Price, i.Price, GETDATE()

FROM deleted d

JOIN inserted i ON d.JewelryID = i.JewelryID

WHERE d.Price <> i.Price;

END

END;

**Purpose:** Tracks price changes in the Jewelry table

**Functionality:** After the Price column is updated, it inserts a record into the **JewelryPriceHistory** table ( records both old and new price)

#### 2)Trigger 2: trg\_Check\_Stock\_Before\_Order

CREATE TRIGGER trg\_Check\_Stock\_Before\_Order

ON OrderDetails

AFTER INSERT

AS

BEGIN

IF EXISTS (

SELECT 1

FROM inserted i

JOIN Jewelry j ON i.JewelryID = j.JewelryID

WHERE i.Quantity > j.StockQuantity

)

BEGIN

RAISERROR ('Số lượng đặt hàng vượt quá số lượng trong kho!', 16, 1);

ROLLBACK TRANSACTION;

END

END;

**Purpose:** Prevents quantity exceeds the stock quantity.

**Functionality:**After an order is placed in **OrderDetails,** it checks if the ordered quantity is greater than the available stock

#### 3)Trigger 3:trg\_Update\_Stock\_After\_Order

CREATE TRIGGER trg\_Update\_Stock\_After\_Order

ON OrderDetails

AFTER INSERT

AS

BEGIN

UPDATE j

SET j.StockQuantity = j.StockQuantity - i.Quantity

FROM Jewelry j

JOIN inserted i ON j.JewelryID = i.JewelryID;

END;

**Purpose:** Automatically updates stock after an order

**Functionality:** After inserting into **OrderDetails,** it subtracts the ordered quantity from the stock quantity in the **Jewelry** table.

#### 4)Trigger 4: trg\_Auto\_Update\_Warranty\_Status

CREATE TRIGGER trg\_Auto\_Update\_Warranty\_Status

ON Warranty

AFTER INSERT, UPDATE

AS

BEGIN

UPDATE Warranty

SET Status = 'EXPIRED'

WHERE EndDate < GETDATE() AND Status <> 'EXPIRED';

END;

**Purpose:** Updates the warranty status automatically after it expires.

**Functionality:** After inserting or updating the **Warranty** table, it checks for expired warranties (using EndDate < GETDATE()) to update status..

#### 5) Trigger 5: trg\_Validate\_Warranty\_Date

CREATE TRIGGER trg\_Validate\_Warranty\_Dates ON Warranty

Instead of INSERT, UPDATE

AS

BEGIN

IF EXISTS (

SELECT 1 FROM inserted WHERE StartDate > EndDate

)

BEGIN

RAISERROR ('Ngày bắt đầu bảo hành không thể lớn hơn ngày kết thúc!', 16, 1);

ROLLBACK TRANSACTION;

END

**Purpose:** Ensures that warranty dates are valid.

**Functionality:** During INSERT or UPDATE, it checks if the StartDate is greater than the EndDate

#### 6) Trigger 6: trg\_Restore\_Stock\_After\_Cancel

CREATE TRIGGER trg\_Restore\_Stock\_After\_Cancel

ON Orders

AFTER UPDATE

AS

BEGIN

IF UPDATE(Status) AND EXISTS (

SELECT 1 FROM inserted WHERE Status = 'CANCELED'

)

BEGIN

UPDATE j

SET j.StockQuantity = j.StockQuantity + od.Quantity

FROM Jewelry j

JOIN OrderDetails od ON j.JewelryID = od.JewelryID

JOIN inserted i ON od.OrderID = i.OrderID

WHERE i.Status = 'CANCELED';

END

END;

**Purpose:** Restores stock quantity when an order is canceled.

**Functionality:** After an **UPDATE** on the **Orders** table, if the status changes to 'CANCELED', the jewelry stock is restored.

#### 7) Trigger 7: trg\_AfterOrder\_Insert

CREATE TRIGGER trg\_AfterOrder\_Insert

ON OrderDetails

AFTER INSERT

AS

BEGIN

SET NOCOUNT ON;

INSERT INTO Warranty (JewelryID, CustomerID, StartDate, EndDate, Status)

SELECT

i.JewelryID,

o.CustomerID,

GETDATE() AS StartDate,

-- Xác định thời gian bảo hành dựa trên giá trị của sản phẩm

CASE

WHEN j.Price < 10000000 THEN DATEADD(MONTH, 6, GETDATE())

WHEN j.Price BETWEEN 10000000 AND 25000000 THEN DATEADD(MONTH, 12, GETDATE())

ELSE DATEADD(MONTH, 24, GETDATE())

END AS EndDate,

'ACTIVE'

FROM inserted i

JOIN Orders o ON i.OrderID = o.OrderID -- Lấy thông tin khách hàng từ đơn hàng

JOIN Jewelry j ON i.JewelryID = j.JewelryID; -- Lấy giá sản phẩm

END;

Purpose: Automatically creates a warranty when a new order is inserted

Functionality: After inserting into **OrderDetails,** a warranty is generated in the **Warranty** table.

The warranty duration depends on the product's price:

* < 10,000,000 VND → 6 months
* 10,000,000 - 25,000,000 VND → 12 months
* > 25,000,000 VND → 24 months

**PROCEDURE**

#### 1) Procedure 1: AddOrder

CREATE PROCEDURE AddOrder

@CustomerID INT,

@PaymentMethod NVARCHAR(50),

@Employid int,

@OrderDetails OrderDetailsType READONLY

AS

BEGIN

SET NOCOUNT ON;

DECLARE @NewOrderID INT;

-- Bước 1: Tạo đơn hàng mới

INSERT INTO Orders (CustomerID, OrderDate, PaymentMethod,EmployeeID, TotalAmount, Status)

VALUES (@CustomerID, GETDATE(), @PaymentMethod,@Employid, 0, 'PENDING');

-- Lấy OrderID vừa tạo

SET @NewOrderID = SCOPE\_IDENTITY();

-- Bước 2: Chèn tất cả sản phẩm vào OrderDetails một lần

INSERT INTO OrderDetails (OrderID, JewelryID, Quantity, UnitPrice)

SELECT @NewOrderID, JewelryID, Quantity, UnitPrice

FROM @OrderDetails;

-- Bước 3: Cập nhật tổng tiền đơn hàng

UPDATE Orders

SET TotalAmount = (SELECT SUM(Quantity \* UnitPrice) FROM OrderDetails WHERE OrderID = @NewOrderID)

WHERE OrderID = @NewOrderID;

-- Trả về OrderID mới

SELECT @NewOrderID AS NewOrderID;

END;

**Purpose:** simplifies the process of creating a new order.( Re-use )

Functionality: it will create , add , calculate and returns OrderID

VIEW

#### 1)View 1: CustomerTransactionHistory

CREATE VIEW CustomerTransactionHistory

AS

SELECT C.CustomerID, C.FullName, O.OrderID, O.OrderDate, O.TotalAmount, O.Status, O.PaymentMethod

FROM Customer C

JOIN Orders O ON C.CustomerID = O.CustomerID

**Purpose**: Show Customer Transaction History:

#### 

#### 2)View 2: JewelryPriceChangeHistory

CREATE VIEW JewelryPriceChangeHistory AS

SELECT J.JewelryID, J.Name AS JewelryName, PH.OldPrice, PH.NewPrice, PH.ChangeDate

FROM Jewelry J

JOIN JewelryPriceHistory PH ON J.JewelryID = PH.JewelryID;

**Purpose**: Show all the changes of jewelry prices all the time

#### 3)View 3: OrderWithDetails

CREATE VIEW OrderWithDetails AS

SELECT O.OrderID, O.OrderDate, O.TotalAmount, O.Status, OD.JewelryID, J.Name AS JewelryName, OD.Quantity, OD.UnitPrice

FROM Orders O

JOIN OrderDetails OD ON O.OrderID = OD.OrderID

JOIN Jewelry J ON OD.JewelryID = J.JewelryID;

**Purpose:** Show the details of every orders

#### 4) View 4: WarrantyStatus

CREATE VIEW WarrantyStatus AS

SELECT W.WarrantyID, C.FullName AS CustomerName, J.Name AS JewelryName, W.StartDate, W.EndDate, W.Status

FROM Warranty W

JOIN Customer C ON W.CustomerID = C.CustomerID

JOIN Jewelry J ON W.JewelryID = J.JewelryID;

**Purpose**: Extract all the warranty status of following jewelry

5)View 5: InventoryWithSupplier

CREATE VIEW InventoryWithSupplier AS

SELECT J.JewelryID, J.Name AS JewelryName, J.StockQuantity, J.Price, S.SupplierName, S.Email, S.Phone\_Number

FROM Jewelry J

JOIN Supplier S ON J.SupplierID = S.SupplierID;

**Purpose**: Show some specific information about the supplier and jewelry that supplier provided

## 3) Testing procedure, trigger,view…..

DECLARE @OrderDetails OrderDetailsType;

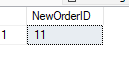
DELETE FROM @OrderDetails;

INSERT INTO @OrderDetails (JewelryID, Quantity, UnitPrice) VALUES

(1, 2, 5000000), (2, 1, 7000000), (3, 3, 4000000);

EXEC AddOrder 1, 'Credit Card',1, @OrderDetails;

--



The OrderId auto-increment after running ( in this example, we just executed 11 orders)