**Phase 2 Documentation: Car Warehouse Database**

**Introduction**

The car warehouse database is a relational database system designed to manage the operations of a model car and vehicle retail business in Pakistan. Its primary purpose is to track offices, employees, customers, products (model cars, motorcycles, and other vehicles), orders, order details, and warehouses. The database supports key business processes such as inventory management, order processing, customer management, and employee oversight, ensuring efficient operations and data integrity through a normalized schema and automated processes like triggers and stored procedures.

**Schema Overview**

The database consists of seven tables, implemented in SQL Server as per the SQL script (CarWarehouseDB.sql). Each table serves a specific purpose and includes primary and foreign key constraints to maintain referential integrity. Below is a summary of the tables and their key constraints:

1. **offices**
   * **Purpose**: Stores information about the company’s office locations across Pakistan.
   * **Columns**: officeCode (PK), city, phone, addressLine1, addressLine2, state, country, postalCode, territory.
   * **Constraints**: Primary key (officeCode).
2. **employees**
   * **Purpose**: Manages employee details, including their roles and reporting structure.
   * **Columns**: employeeNumber (PK), lastName, firstName, extension, email, officeCode (FK), reportsTo (FK), jobTitle.
   * **Constraints**: Primary key (employeeNumber), foreign keys (officeCode references offices.officeCode, reportsTo references employees.employeeNumber), indexes (idx\_reportsTo, idx\_officeCode).
3. **customers**
   * **Purpose**: Stores customer information for order processing and sales tracking.
   * **Columns**: customerNumber (PK), customerName, contactLastName, contactFirstName, phone, addressLine1, addressLine2, city, state, postalCode, country, salesRepEmployeeNumber (FK), creditLimit.
   * **Constraints**: Primary key (customerNumber), foreign key (salesRepEmployeeNumber references employees.employeeNumber), index (idx\_salesRepEmployeeNumber).
4. **products**
   * **Purpose**: Maintains details of model vehicles and sports equipment available for sale.
   * **Columns**: productCode (PK), productName, productLine, productScale, productVendor, productDescription, quantityInStock, warehouseCode (FK), buyPrice, MSRP.
   * **Constraints**: Primary key (productCode).
5. **orders**
   * **Purpose**: Tracks customer orders, including dates and status.
   * **Columns**: orderNumber (PK), orderDate, requiredDate, shippedDate, status, comments, customerNumber (FK).
   * **Constraints**: Primary key (orderNumber), foreign key (customerNumber references customers.customerNumber), index (idx\_customerNumber).
6. **orderdetails**
   * **Purpose**: Stores details of products ordered in each order, including quantities and prices.
   * **Columns**: orderNumber (PK, FK), productCode (PK, FK), quantityOrdered, priceEach, orderLineNumber.
   * **Constraints**: Composite primary key (orderNumber, productCode), foreign keys (orderNumber references orders.orderNumber, productCode references products.productCode), index (idx\_productCode).
7. **warehouses**
   * **Purpose**: Manages warehouse details and their capacity utilization.
   * **Columns**: warehouseCode (PK), warehouseName, warehousePctCap.
   * **Constraints**: Primary key (warehouseCode).

The schema is normalized to at least 3NF, ensuring no data redundancy and maintaining referential integrity through foreign key constraints.

**CRUD Queries**

The following SQL queries demonstrate Create, Read, Update, and Delete operations on key tables (customers, orders, orderdetails, products). These are included in Queries.sql.

**Create Queries**

1. **Insert a New Customer**

INSERT INTO customers (customerNumber, customerName, contactLastName, contactFirstName, phone, addressLine1, city, country, postalCode, salesRepEmployeeNumber, creditLimit)

VALUES (216, 'Lahore Auto Hub', 'Khan', 'Asad', '+92-42-7890123', 'Gulberg II, Main Boulevard', 'Lahore', 'Pakistan', '54600', 1059, 40000.00);

* + **Purpose**: Adds a new customer to the database, ensuring all required fields are populated and the salesRepEmployeeNumber references a valid employee.

1. **Insert a New Order**

INSERT INTO orders (orderNumber, orderDate, requiredDate, status, customerNumber)

VALUES (316, '2025-06-10', '2025-06-17', 'In Process', 216);

* + **Purpose**: Creates a new order for a customer, linking it to customerNumber = 216.

1. **Insert Order Details**

INSERT INTO orderdetails (orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber)

VALUES (316, 'PK\_MC\_1001', 3, 70.99, 1);

* + **Purpose**: Adds details of products ordered, specifying quantity and price, linked to orderNumber = 316 and a valid productCode.

**Read Queries**

1. **Retrieve Products with Warehouse Details**

SELECT p.productCode, p.productName, p.quantityInStock, w.warehouseName, w.warehousePctCap

FROM products p

JOIN warehouses w ON p.warehouseCode = w.warehouseCode

WHERE p.quantityInStock > 0;

* + **Purpose**: Fetches available products and their warehouse details, useful for inventory management.

1. **Retrieve Order Details for a Customer**

SELECT o.orderNumber, o.orderDate, o.status, od.productCode, od.quantityOrdered, od.priceEach

FROM orders o

JOIN orderdetails od ON o.orderNumber = od.orderNumber

WHERE o.customerNumber = 201;

* + **Purpose**: Displays all orders and their details for a specific customer, aiding in order tracking.

1. **Retrieve Employees and Office Locations**

SELECT e.employeeNumber, e.firstName, e.lastName, e.jobTitle, o.city, o.phone

FROM employees e

JOIN offices o ON e.officeCode = o.officeCode;

* + **Purpose**: Lists employees with their office locations, useful for organizational management.

**Update Queries**

1. **Update Product Stock**

UPDATE products

SET quantityInStock = quantityInStock - 2

WHERE productCode = 'PK\_MC\_1001';

* + **Purpose**: Reduces stock for a product after a sale, ensuring accurate inventory tracking.

1. **Update Customer Credit Limit**

UPDATE customers

SET creditLimit = creditLimit + 5000.00

WHERE customerNumber = 201;

* + **Purpose**: Increases a customer’s credit limit, reflecting updated financial agreements.

1. **Update Order Status**

UPDATE orders

SET status = 'Shipped', shippedDate = '2025-06-10'

WHERE orderNumber = 302;

* + **Purpose**: Marks an order as shipped and sets the shipped date, updating order status.

**Delete Queries**

1. **Delete an Order and Its Details**

DELETE FROM orderdetails WHERE orderNumber = 316;

DELETE FROM orders WHERE orderNumber = 316;

* + **Purpose**: Removes an order and its details, deleting from orderdetails first to maintain referential integrity.

1. **Delete a Customer (if No Orders Exist)**

DELETE FROM customers WHERE customerNumber = 216 AND customerNumber NOT IN (SELECT customerNumber FROM orders);

* + **Purpose**: Deletes a customer only if they have no associated orders, preventing constraint violations.

**Stored Procedures**

Two stored procedures are implemented to automate common tasks, included in Queries.sql.

1. **GetCustomerOrderSummary**

CREATE PROCEDURE GetCustomerOrderSummary

@CustomerNumber INT

AS

BEGIN

SELECT

c.customerNumber,

c.customerName,

COUNT(o.orderNumber) AS TotalOrders,

SUM(od.quantityOrdered \* od.priceEach) AS TotalSpent

FROM customers c

LEFT JOIN orders o ON c.customerNumber = o.customerNumber

LEFT JOIN orderdetails od ON o.orderNumber = od.orderNumber

WHERE c.customerNumber = @CustomerNumber

GROUP BY c.customerNumber, c.customerName;

END;

* + **Description**: Retrieves the total number of orders and total spending for a given customer.
  + **Input**: @CustomerNumber (INT) – The customer’s ID.
  + **Output**: A result set with customerNumber, customerName, TotalOrders, and TotalSpent.
  + **Use Case**: Useful for sales analysis and customer loyalty tracking.
  + **Example Execution**: EXEC GetCustomerOrderSummary @CustomerNumber = 201;

1. **UpdateProductStock**

CREATE PROCEDURE UpdateProductStock

@ProductCode VARCHAR(15),

@QuantityOrdered INT,

@ErrorMessage NVARCHAR(100) OUTPUT

AS

BEGIN

DECLARE @CurrentStock SMALLINT;

SELECT @CurrentStock = quantityInStock

FROM products

WHERE productCode = @ProductCode;

IF @CurrentStock >= @QuantityOrdered

BEGIN

UPDATE products

SET quantityInStock = quantityInStock - @QuantityOrdered

WHERE productCode = @ProductCode;

SET @ErrorMessage = 'Stock updated successfully';

END

ELSE

BEGIN

SET @ErrorMessage = 'Insufficient stock for product ' + @ProductCode;

END

END;

* + **Description**: Updates product stock after an order, checking for sufficient inventory.
  + **Input**: @ProductCode (VARCHAR(15)) – The product’s code; @QuantityOrdered (INT) – Quantity to deduct.
  + **Output**: @ErrorMessage (NVARCHAR(100)) – Success or error message.
  + **Use Case**: Ensures inventory is updated only when sufficient stock is available, preventing overselling.
  + **Example Execution**: DECLARE @Error NVARCHAR(100); EXEC UpdateProductStock @ProductCode = 'PK\_MC\_1001', @QuantityOrdered = 5, @ErrorMessage = @Error OUTPUT; SELECT @Error AS Result;

**Triggers**

Two triggers are implemented to automate database tasks, included in Queries.sql.

1. **UpdateWarehouseCapacity**

CREATE TRIGGER UpdateWarehouseCapacity

ON products

AFTER UPDATE

AS

BEGIN

UPDATE warehouses

SET warehousePctCap =

CASE

WHEN EXISTS (SELECT 1 FROM inserted i WHERE i.warehouseCode = warehouses.warehouseCode)

THEN CAST((SELECT SUM(quantityInStock) \* 100.0 / 1000 AS INT) + '%' FROM products p WHERE p.warehouseCode = warehouses.warehouseCode)

ELSE warehousePctCap

END

FROM inserted

WHERE warehouses.warehouseCode = inserted.warehouseCode;

END;

* + **Description**: Updates the warehousePctCap in the warehouses table based on the total quantityInStock of products in that warehouse after any update to the products table.
  + **Trigger Event**: AFTER UPDATE on the products table.
  + **Purpose**: Ensures accurate tracking of warehouse capacity utilization, assuming a maximum capacity of 1000 units for simplicity.
  + **Example Test**: UPDATE products SET quantityInStock = quantityInStock + 50 WHERE productCode = 'PK\_MC\_1001'; updates the capacity for the associated warehouse.

1. **UpdateCustomerCreditLimit**

CREATE TRIGGER UpdateCustomerCreditLimit

ON orderdetails

AFTER INSERT

AS

BEGIN

UPDATE c

SET c.creditLimit = c.creditLimit + (

SELECT SUM(i.quantityOrdered \* i.priceEach) \* 0.1

FROM inserted i

JOIN orders o ON i.orderNumber = o.orderNumber

WHERE o.customerNumber = c.customerNumber

)

FROM customers c

WHERE c.customerNumber IN (

SELECT DISTINCT o.customerNumber

FROM inserted i

JOIN orders o ON i.orderNumber = o.orderNumber

);

END;

* + **Description**: Increases a customer’s creditLimit by 10% of the total spent on newly inserted order details, calculated as quantityOrdered \* priceEach.
  + **Trigger Event**: AFTER INSERT on the orderdetails table.
  + **Purpose**: Supports customer loyalty by automatically adjusting credit limits based on purchase amounts, encouraging repeat business.
  + **Example Test**: INSERT INTO orderdetails (orderNumber, productCode, quantityOrdered, priceEach, orderLineNumber) VALUES (311, 'PK\_MC\_1002', 2, 85.50, 2); increases the creditLimit for customerNumber = 210 by 17.10 (10% of 171.00). Verified with: SELECT customerNumber, customerName, creditLimit FROM customers WHERE customerNumber = 210;.

**Sample Data**

The database is populated with comprehensive sample data, as provided in CarWarehouseDB.sql. The data includes:

* **offices**: 15 offices across Pakistan (e.g., Islamabad, Karachi, Lahore).
* **employees**: 15 employees with roles like Regional Manager, Sales Manager, and Sales Representative.
* **customers**: 15 customers with details like name, contact info, and credit limits.
* **products**: 16 products, including model cars, motorcycles, trucks etc.
* **orders**: 15 orders with varying statuses (e.g., Shipped, In Process, Pending).
* **orderdetails**: Multiple entries linking orders to products with quantities and prices.
* **warehouses**: 7 warehouses with names and capacity percentages.

These records ensure the database supports realistic querying and testing of CRUD operations, stored procedures, and triggers.