Database Practice Questions

* #### 1. Normalization
* ##### 1.1 Question 1: Identify and Fix Normal Form Violations
* a) Violates 1NF (multiple values in single field: Courses, Instructor, InstructorEmail), 2NF (non-prime attribute dependent on part of composite key), and 3NF (transitive dependency on InstructorEmail via Instructor).
* b) Normalize to 3NF:
* sql
* Student(StudentID, Name)
* Course(CourseID, CourseName)
* Instructor(InstructorID, InstructorName, InstructorEmail)
* StudentCourse(StudentID, CourseID)
* CourseInstructor(CourseID, InstructorID)
* c) Improves data integrity by eliminating redundant storage of instructor data and course lists per student, allowing accurate updates.
* ##### 1.2 Question 2: Normalize a Sales Table
* a) Functional Dependencies:
  + OrderID → CustomerName, CustomerPhone
  + Product → Supplier
  + (OrderID, Product) → Quantity
* b)Normalize to 3NF:
* sql
* Customer(CustomerID, CustomerName, CustomerPhone)
* Product(ProductID, ProductName, Supplier)
* SalesOrder(OrderID, CustomerID)
* SalesOrderDetails(OrderID, ProductID, Quantity)
* c)Normalization avoids anomalies:
  + Insertion:New product can be added without order.
  + Deletion:Deleting an order won't lose customer/product data.
  + Update: Changing supplier info in one place.
* ##### 1.3 Question 3: Partial and Transitive Dependencies
* a)Partial: EmpID → EmpName
* Transitive: ProjectID → ManagerName
* b)Normalize to 3NF:
* sql
* Employee(EmpID, EmpName)
* Project(ProjectID, ProjectName, ManagerName)
* EmployeeProject(EmpID, ProjectID)
* c)Sample Data:
* sql
* Employee: E1, Alice | E2, Carol
* Project: P1, CRM, Bob | P2, ERP, Dave
* EmployeeProject: E1, P1 | E1, P2 | E2, P2
* ##### 1.4 Question 4: Real-World Scenario
* a) Violates 1NF (multiple borrowers per book), 2NF, and 3NF.

b)Normalize:

* sql
* Book(BookID, Title)
* Borrower(BorrowerID, BorrowerName, BorrowerAddress)
* Loan(BookID, BorrowerID, BorrowDate)
* c)Allows tracking loans by joining Loan with Book/Borrower tables.
* #### 2. Joins
* ##### 2.1 Question 5: Types of Joins
* a) INNER JOIN:
* sql
* SELECT s.Name, c.ClassName, t.TeacherName
* FROM Students s
* JOIN Classes c ON s.ClassID = c.ClassID
* JOIN Teachers t ON c.TeacherID = t.TeacherID;
* b) LEFT OUTER JOIN:
* sql
* SELECT s.Name, c.ClassName
* FROM Students s
* LEFT JOIN Classes c ON s.ClassID = c.ClassID;
* c)FULL OUTER JOIN (syntax varies):
* sql
* SELECT s.Name, c.ClassName
* FROM Students s
* FULL OUTER JOIN Classes c ON s.ClassID = c.ClassID;
* ##### 2.2 Question 6: Multi-Table Joins
* a)
* sql
* SELECT cu.CustomerName, o.OrderDate, p.ProductName, (od.Quantity \* p.UnitPrice) AS TotalCost
* FROM Orders o
* JOIN Customers cu ON o.CustomerID = cu.CustomerID
* JOIN OrderDetails od ON o.OrderID = od.OrderID
* JOIN Products p ON od.ProductID = p.ProductID;
* b) Add condition:
* sql
* WHERE cu.City = 'New York';
* c)Customers without orders:
* sql
* SELECT c.CustomerName
* FROM Customers c
* LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
* WHERE o.OrderID IS NULL;
* ##### 2.3 Question 7: Self Join
* a)
* sql
* SELECT e.Name AS Employee, m.Name AS Manager
* FROM Employees e
* LEFT JOIN Employees m ON e.ManagerID = m.EmployeeID;
* b) Included in above (LEFT JOIN).
* c)Necessary since manager is in same table as employee.
* ##### 2.4 Question 8: Complex Joins
* sql
* SELECT pr.ProjectName, d.DepartmentName, e.Name, a.HoursWorked
* FROM Projects pr
* JOIN Departments d ON pr.DepartmentID = d.DepartmentID
* LEFT JOIN Assignments a ON pr.ProjectID = a.ProjectID
* LEFT JOIN Employees e ON a.EmployeeID = e.EmployeeID;
* ##### 2.5 Question 9: Cross Join for Scheduling
* a)
* sql
* SELECT r.RoomName, t.TimePeriod
* FROM Rooms r
* CROSS JOIN Timeslots t;
* b)
* sql
* WHERE NOT (r.RoomID = 'R1' AND t.TimePeriod = 'Morning');
* c) Useful for creating combinations (e.g., schedules). Risk: Cartesian explosion with large tables.
* ##### 2.6 Question 10: Non-Equi Joins
* a)
* sql
* SELECT od.OrderID, od.ProductID, od.Quantity, pr.DiscountPrice
* FROM OrderDetails od
* JOIN PriceRanges pr ON od.Quantity BETWEEN pr.MinQuantity AND pr.MaxQuantity;
* b) Include unmatched:
* sql
* RIGHT JOIN WHERE pr.DiscountPrice IS NULL;
* c) Equi: equality match, Non-equi: range-based matching.
* ##### 2.7 Question 11: Joins with Aggregation
* a)
* sql
* SELECT c.CustomerName, SUM(od.Quantity) AS TotalProducts
* FROM Customers c
* JOIN Orders o ON c.CustomerID = o.CustomerID
* JOIN OrderDetails od ON o.OrderID = od.OrderID
* GROUP BY c.CustomerName;
* b)
* sql
* HAVING SUM(od.Quantity) > 10;
* c)
* sql
* SELECT c.CustomerName
* FROM Customers c
* LEFT JOIN Orders o ON c.CustomerID = o.CustomerID
* WHERE o.OrderID IS NULL;
* #### 3. Aggregate Functions
* ##### 3.1 Question 12: Basic Aggregates
* a)
* sql
* SELECT Category, SUM(Stock) AS TotalStock
* FROM Inventory
* GROUP BY Category;
* b)
* sql
* SELECT Category, SUM(Stock \* UnitPrice) AS InventoryValue
* FROM Inventory
* GROUP BY Category;
* c)
* sql
* SELECT Category, ProductName
* FROM Inventory i
* WHERE Stock = (
* SELECT MAX(Stock)
* FROM Inventory i2
* WHERE i2.Category = i.Category
* );
* ##### 3.2 Question 13: GROUP BY with Multiple Columns
* a)
* sql
* SELECT ProductID, StoreID, SUM(Quantity) AS TotalSold
* FROM Sales
* GROUP BY ProductID, StoreID;
* b)
* sql
* HAVING SUM(Quantity) > 50;
* c)
* sql
* SELECT YEAR(SaleDate) AS Year, StoreID, COUNT(\*) AS SalesCount
* FROM Sales
* GROUP BY YEAR(SaleDate), StoreID;
* ##### 3.3 Question 14: HAVING Clause Scenarios
* a)
* sql
* SELECT CustomerID, COUNT(OrderID) AS OrderCount
* FROM Orders
* GROUP BY CustomerID
* HAVING COUNT(OrderID) > 5;
* B)
* sql
* SELECT MONTH(OrderDate) AS Month, SUM(TotalAmount) AS MonthlyTotal
* FROM Orders
* GROUP BY MONTH(OrderDate)
* HAVING SUM(TotalAmount) > 5000;
* c)
* sql
* SELECT CustomerID, AVG(TotalAmount) AS AvgAmount
* FROM Orders
* GROUP BY CustomerID
* HAVING AVG(TotalAmount) > 200;
* ##### 3.4 Question 15: Advanced Aggregation with Joins
* a)
* sql
* SELECT d.DepartmentName, SUM(a.HoursWorked) AS TotalHours
* FROM Departments d
* JOIN Projects p ON d.DepartmentID = p.DepartmentID
* JOIN Assignments a ON p.ProjectID = a.ProjectID
* GROUP BY d.DepartmentName;
* b)
* sql
* HAVING SUM(a.HoursWorked) > 100;
* c)
* sql
* SELECT d.DepartmentName
* FROM Departments d
* JOIN Employees e ON d.DepartmentID = e.DepartmentID
* GROUP BY d.DepartmentName
* ORDER BY AVG(e.Salary) DESC
* LIMIT 1;
* ##### 3.5 Question 16: Nested Aggregates
* a)
* sql
* SELECT CustomerID, MAX(Amount) AS MaxTransaction
* FROM Transactions
* GROUP BY CustomerID;
* b)
* ql
* HAVING MAX(Amount) > (SELECT AVG(Amount) FROM Transactions);
* c)
* sql
* SELECT CustomerID, SUM(Amount) AS TotalAmount
* FROM Transactions
* GROUP BY CustomerID
* ORDER BY TotalAmount DESC
* LIMIT 1;
* ##### 3.6 Question 17: Aggregation with Date Functions
* a)
* sql
* SELECT RoomID, QUARTER(BookingDate) AS Qtr, SUM(Revenue) AS TotalRevenue
* FROM Bookings
* GROUP BY RoomID, QUARTER(BookingDate);
* b)
* sql
* HAVING SUM(Revenue) > 1000;
* c)
* sql
* SELECT RoomID, AVG(Revenue) AS AvgDailyRevenue
* FROM Bookings
* GROUP BY RoomID
* ORDER BY AvgDailyRevenue DESC
* LIMIT 1;
* ##### 3.7 Question 18: Combining Aggregates and Joins
* a)
* sql
* SELECT c.CategoryName, SUM(s.Quantity) AS TotalSold
* FROM Categories c
* JOIN Products p ON c.CategoryID = p.CategoryID
* JOIN Sales s ON p.ProductID = s.ProductID
* GROUP BY c.CategoryName;
* b)
* sql
* HAVING SUM(s.Quantity) < 100;
* c)
* sql
* SELECT c.CategoryName
* FROM Categories c
* JOIN Products p ON c.CategoryID = p.CategoryID
* JOIN Sales s ON p.ProductID = s.ProductID
* GROUP BY c.CategoryName
* ORDER BY COUNT(DISTINCT p.ProductID) DESC
* LIMIT 1;