SQL Practice Schemas & Full SQL Topic List

Practice Schemas

```
-- ============================================================
-- TABLE: categories
-- ============================================================
CREATE TABLE categories (
   CategoryID INT PRIMARY KEY AUTO_INCREMENT,
   CategoryName VARCHAR(100),
   DescriptionText VARCHAR(255)
);

-- ============================================================
-- TABLE: suppliers
-- ============================================================
CREATE TABLE suppliers (
   SupplierID INT PRIMARY KEY AUTO_INCREMENT,
   SupplierName VARCHAR(150) NOT NULL,
   ContactName VARCHAR(100),
   Address VARCHAR(255),
   City VARCHAR(100),
   PostalCode VARCHAR(20),
   Country VARCHAR(100),
   Phone VARCHAR(50)
);

-- ============================================================
-- TABLE: products
-- ============================================================
CREATE TABLE products (
   ProductID INT PRIMARY KEY AUTO_INCREMENT,
   ProductName VARCHAR(100) NOT NULL,
   SupplierID INT NOT NULL,
   CategoryID INT NOT NULL,
   QuantityPerUnit VARCHAR(50),
   UnitPrice DECIMAL(10,2),
   UnitsInStock INT,
   UnitsOnOrder INT,
   ReorderLevel INT,
```

```
  Discontinued TINYINT DEFAULT 0,
  FOREIGN KEY (CategoryID) REFERENCES categories(CategoryID),
  FOREIGN KEY (SupplierID) REFERENCES suppliers(SupplierID)
);

-- ============================================================
-- TABLE: customers
-- ============================================================
CREATE TABLE customers (
  CustomerID INT PRIMARY KEY AUTO_INCREMENT,
  CustomerName VARCHAR(150),
  ContactName VARCHAR(150),
  Address VARCHAR(255),
  City VARCHAR(100),
  PostalCode VARCHAR(20),
  Country VARCHAR(100)
);

-- ============================================================
-- TABLE: employees
-- ============================================================
CREATE TABLE employees (
  EmployeeID INT PRIMARY KEY AUTO_INCREMENT,
  LastName VARCHAR(100) NOT NULL,
  FirstName VARCHAR(100) NOT NULL,
  BirthDate DATE
);

-- ============================================================
-- TABLE: shippers
-- ============================================================
CREATE TABLE shippers (
  ShipperID INT PRIMARY KEY AUTO_INCREMENT,
  ShipperName VARCHAR(100) NOT NULL,
  Phone VARCHAR(50)
);

-- ============================================================
-- TABLE: orders
-- ============================================================
CREATE TABLE orders (
  OrderID INT PRIMARY KEY AUTO_INCREMENT,
  CustomerID INT NOT NULL,
```

```
  EmployeeID INT NOT NULL,
  OrderDate DATE NOT NULL,
  ShipperID INT NOT NULL,
  FOREIGN KEY (CustomerID) REFERENCES customers(CustomerID),
  FOREIGN KEY (EmployeeID) REFERENCES employees(EmployeeID),
  FOREIGN KEY (ShipperID) REFERENCES shippers(ShipperID)
);

-- ============================================================
-- TABLE: order_details
-- ============================================================
CREATE TABLE order_details (
  OrderDetailID INT PRIMARY KEY AUTO_INCREMENT,
  OrderID INT NOT NULL,
  ProductID INT NOT NULL,
  Quantity INT NOT NULL,
  FOREIGN KEY (OrderID) REFERENCES orders(OrderID),
  FOREIGN KEY (ProductID) REFERENCES products(ProductID)
);
```

SQL Basics
Filtering & Logical Operations

• Comparison operators (=, !=, >, <, >=, <=)

📘 **Definition:**
Comparison operators are used to **compare values** in SQL.

🔧 **Use:**

- Filter data based on a condition.
- Common operators: =, != (or <>), >, <, >=, <=.

• Logical operators (AND, OR, NOT)

📘 **Definition:**
Logical operators are used to **combine multiple conditions** in SQL.

🔧 **Use:**

- **AND:** Both conditions must be true.
- **OR:** At least one condition must be true.
- **NOT:** Reverses the condition.

• BETWEEN

📘 **Definition:**
BETWEEN is used to **filter data within a range** (inclusive).

🔧 **Use:**

- Makes it easy to check if a value lies between two numbers or dates.

• IN

📘 **Definition:**
IN is used to **match a value against a set of values**.

🔧 **Use:**

- Check if a column value exists in a **list of options**.

• LIKE (pattern matching)

📘 **Definition:**
LIKE is used to **search for a specific pattern** in a column.

🔧 **Use:**

- Useful for **partial matches** with % (any number of characters) or _ (single character).

• IS NULL, IS NOT NULL

📘 **Definition:**

- IS NULL checks if a column **has no value**.
- IS NOT NULL checks if a column **has a value**.

🔧 **Use:**

- Filter rows where data is **missing or present**.

Aggregations

• COUNT()

📘 **Definition:**
COUNT() is used to **count the number of rows** in a table or query result.

🔧 **Use:**

- Count total records or specific records that meet a condition

• SUM()

📘 **Definition:**
SUM() adds up **all numeric values** in a column.

🔧 **Use:**

- Calculate total sales, total prices, total quantity, etc.

• AVG()

📘 **Definition:**
AVG() calculates the **average value** of a numeric column.

🔧 **Use:**

- Find average price, salary, score, or quantity.

• MIN(), MAX()

📘 **Definition:**

- MIN() finds the **smallest value** in a column.
- MAX() finds the **largest value** in a column.

🔧 **Use:**

- Identify highest/lowest price, salary, quantity, etc.

• GROUP BY

📘 **Definition:**
GROUP BY groups rows **having the same values** in specified columns so you can perform **aggregations on each group**.

🔧 **Use:**

- Useful for summarizing data by categories, departments, cities, etc.

• HAVING

📘 **Definition:**
HAVING filters **groups created by GROUP BY**, similar to how WHERE filters rows.

🔧 **Use:**

- Use HAVING to filter aggregated results.

Joins

• INNER JOIN

📘 **Definition:**
Returns **only the matching rows** from both tables.

🔧 **Use:**

- To get related data that exists in **both tables**.

• LEFT JOIN

📘 **Definition:**
Returns **all rows from the left table**, and matching rows from the right table. If no match, **NULL** is shown.

🔧 **Use:**

- Useful when you want **all records from one table**, even if the other table has no matching data.

• RIGHT JOIN

📘 **Definition:**
Returns **all rows from the right table**, and matching rows from the left table. If no match, **NULL** is shown.

🔧 **Use:**

- Useful to get **all data from the second table**, even if the first table has no match.

• FULL OUTER JOIN

📘 **Definition:**
Returns **all rows from both tables**, matching when possible. If no match, **NULL** is shown for missing values.

🔧 **Use:**

- Useful to **combine everything** from two tables.

• CROSS JOIN

📘 **Definition:**
Returns **all possible combinations** of rows from two tables (Cartesian product).

🔧 **Use:**

- Rarely used, mostly for **combinatorial results**.

• SELF JOIN

📘 **Definition:**
A table **joins with itself**. Useful for hierarchical or related data in the same table.

🔧 **Use:**

- Find **manager-employee relationships** or comparisons within a table.

• Multiple table JOINs

📘 **Definition:**
You can **join more than two tables** in a single query.

🔧 **Use:**

- Useful for **complex queries** across multiple tables.

• JOIN with conditions

📘 **Definition:**
JOINs can use **additional conditions** besides matching keys.

🔧 **Use:**

- Filter joined data for more **precise results**.

Subqueries

• Single-row subquery

📘 **Definition:**
A **single-row subquery** returns **only one row** (and usually one column).

🔧 **Use:**

- Compare a column with a **single value** returned by a subquery.

• Multi-row subquery

📘 **Definition:**
A **multi-row subquery** returns **more than one row**.

🔧 **Use:**

- Compare a column with **multiple values** using IN, ANY, or ALL.

• Nested subqueries

3️⃣ Nested Subqueries

📘 **Definition:**
A **nested subquery** is a subquery **inside another subquery**.

🔧 **Use:**

- Useful for **multi-step calculations** or filters.

• Correlated subqueries

📘 **Definition:**
A **correlated subquery** depends on the **outer query** and is evaluated **for each row**.

🔧 **Use:**

- Useful when **subquery needs outer row values.**

• Subquery vs JOIN

📘 Definition:

- **Subquery:** Query inside another query.
- **JOIN:** Combines two or more tables directly in the main query.

🔧 Use:

- Subquery → Good for **filtering, nested logic**.
- JOIN → Good for **fetching related data** efficiently.

Window Functions

• OVER()

📘 **Definition:**
OVER() tells SQL that a function is a **window function**. It defines the **set of rows** the function works on.

🔧 **Use:**

- Used with functions like ROW_NUMBER(), SUM(), AVG() to calculate **over a specific window of rows.**

• PARTITION BY

📘 **Definition:**
PARTITION BY **divides rows into groups**, like GROUP BY, but keeps individual rows.

🔧 **Use:**

- Apply window functions **within each group**.

• ORDER BY inside window

📘 **Definition:**
ORDER BY inside OVER() defines the **order of rows within the partition**.

🔧 **Use:**

- Necessary for ranking, running totals, or lag/lead calculations

• ROW_NUMBER()

📘 **Definition:**
Gives a **unique sequential number** to each row in a partition.

🔧 **Use:**

- Identify first/last row, remove duplicates, create row IDs.

• **RANK()**

📘 **Definition:**
Gives **rank to rows**, with **gaps** if there are ties.

🔧 **Use:**

- Rank employees, products, or sales, handling ties.

• **DENSE_RANK()**

📘 **Definition:**
Like RANK(), but **no gaps** in ranking numbers.

🔧 **Use:**

- Useful for **continuous ranking** even if ties exist.

• **LAG()**

📘 **Definition:**
Returns the **previous row's value** in a partition.

🔧 **Use:**

- Compare current row with previous row.

• **LEAD()**

📘 **Definition:**
Returns the **next row's value** in a partition.

🔧 **Use:**

- Compare current row with next row.

• **FIRST_VALUE()**

📘 **Definition:**
Returns the **first value** in the window/partition.

🔧 **Use:**

- Identify the top performer or earliest record in each group.

• LAST_VALUE()

📘 **Definition:**
Returns the **last value** in the window/partition.

🔧 **Use:**

- Identify the lowest value or last record in each group.

• NTILE()

📘 **Definition:**
Divides rows into **equal number of buckets (tiles)** and assigns a tile number.

🔧 **Use:**

- Useful for **quartiles, deciles, percentiles**.

• Moving averages, Running total, Rolling sums

📘 **Definition:**

- **Running Total:** Cumulative sum up to current row.
- **Moving Average:** Average over **last N rows**.
- **Rolling Sum:** Sum over a **sliding window**.

🔧 **Use:**

- Common in **analytics, finance, and dashboards**.

String Functions

- **CONCAT**

📘 **Definition:**
CONCAT() joins **two or more strings together**.

🔧 **Use:**

- Combine first name and last name, or any text.

- **CONCAT_WS**

📘 **Definition:**
CONCAT_WS() joins strings **with a separator**. WS = With Separator.

🔧 **Use:**

- Combine multiple strings with a **specific character** like comma, dash, or space.

- **LENGTH**

📘 **Definition:**
LENGTH() returns the **number of characters** in a string.

🔧 **Use:**

- Find string length for validation or reporting.

- **SUBSTRING**

📘 **Definition:**
SUBSTRING() extracts a **part of a string**.

🔧 **Use:**

- Get a portion of text from a string.

- **LEFT / RIGHT**

📘 **Definition:**

- LEFT(string, n) → first **n characters** from left.
- RIGHT(string, n) → last **n characters** from right.

🔧 **Use:**

- Extract start or end part of a string.

• INSTR

📘 **Definition:**
INSTR(string, substring) returns the **position of substring** in a string.

🔧 **Use:**

- Find where a word or character appears.

• REPLACE

📘 **Definition:**
REPLACE(string, old, new) replaces **old text with new text**.

🔧 **Use:**

- Correct typos, update text in queries.

• LOWER / UPPER

📘 **Definition:**

- LOWER() → converts text to **lowercase**
- UPPER() → converts text to **uppercase**

🔧 **Use:**

- Standardize text for comparison or display.

• TRIM, LTRIM, RTRIM

📘 **Definition:**

- TRIM() → removes **spaces from both sides**
- LTRIM() → removes **spaces from left**
- RTRIM() → removes **spaces from right**

🔧 **Use:**

- Clean data before comparison or display.

Date & Time Functions

- **NOW()**

📘 **Definition:**
NOW() returns the **current date and time** of the system.

🔧 **Use:**

- To record timestamps, log activities, or fetch current time.

- **CURDATE()**

📘 **Definition:**
CURDATE() returns the **current date only** (without time).

🔧 **Use:**

- Useful when you only need the **date** part.

- YEAR(), MONTH(), DAY()

📘 **Definition:**

- YEAR(date) → Extracts the **year**
- MONTH(date) → Extracts the **month**
- DAY(date) → Extracts the **day**

🔧 **Use:**

- Break a date into components for reporting or filtering.

- DATE_ADD(), DATE_SUB()

📘 **Definition:**

- DATE_ADD(date, INTERVAL n unit) → Adds days, months, or years to a date.
- DATE_SUB(date, INTERVAL n unit) → Subtracts days, months, or years from a date.

🔧 **Use:**

- Calculate future or past dates.

• DATEDIFF()

📘 **Definition:**
DATEDIFF(date1, date2) returns the **difference between two dates in days**.

🔧 **Use:**

- Find age, subscription duration, or days between events.

• TIMESTAMPDIFF()

📘 **Definition:**
TIMESTAMPDIFF(unit, datetime1, datetime2) returns the difference between two dates in a **specified unit** (DAY, MONTH, YEAR, HOUR, MINUTE, SECOND).

🔧 **Use:**

- More flexible than DATEDIFF for **years, months, hours, etc.**

• DATE_FORMAT()

📘 **Definition:**
DATE_FORMAT(date, format) formats a date into a **custom string**.

🔧 **Use:**

- Display date in **friendly format** or for reports.

Data Cleaning Functions

• COALESCE()

📘 **Definition:**
COALESCE(value1, value2, ...) returns the **first non-NULL value** in the list.

🔧 **Use:**

- Replace **NULL values** with default values.

• NULLIF()

📘 **Definition:**
NULLIF(value1, value2) returns **NULL** if the two values are equal; otherwise, returns the first value.

🔧 **Use:**

- Prevent division by zero or mark certain values as NULL.

• IFNULL()

📘 **Definition:**
IFNULL(value, replacement) returns the **replacement** if the value is NULL.

🔧 **Use:**

- Similar to COALESCE() but only checks **one column**.

• CAST(), CONVERT()

📘 **Definition:**

- CAST(expression AS datatype) → Change data type of a column.
- CONVERT(expression, datatype) → Another way to convert data type.

🔧 **Use:**

- Convert string to number, number to string, or date formatting.

• CASE WHEN

📘 **Definition:**
CASE WHEN is used to **apply conditional logic** in SQL.

🔧 **Use:**

- Create new columns or categories based on conditions.

• Removing duplicates → DISTINCT

📘 **Definition:**
DISTINCT removes **duplicate rows** in the result set.

🔧 **Use:**

- Get **unique values** for analysis or reporting.

• Data transformation

📘 **Definition:**
Data transformation is **changing data format, type, or structure** for analysis.

🔧 **Use:**

- Standardize data, combine columns, replace values, or calculate new columns.

Data Modeling & Constraints

• Primary Key

📘 **Definition:**
A **Primary Key** uniquely identifies each row in a table.

🔧 **Use:**

- Ensures **no duplicate records** and **fast lookup**.
- Only **one PK per table**.

• Foreign Key

📘 **Definition:**
A **Foreign Key** links one table to another by referencing the **Primary Key** of the other table.

🔧 **Use:**

- Maintain **relationships** between tables.
- Enforces **referential integrity**.

• Unique

📘 **Definition:**
Ensures that a column (or set of columns) has **only unique values**.

🔧 **Use:**

- Prevent duplicate values in a column.

• Check

📘 **Definition:**
CHECK enforces that a column satisfies a **specific condition**.

🔧 **Use:**

- Validate data before inserting or updating.

• Default

📘 **Definition:**
DEFAULT sets a **default value** for a column if no value is provided.

🔧 **Use:**

- Automatically fill a column when inserting new rows.

• Referential integrity

📘 **Definition:**
Ensures **relationships between tables are consistent**.

🔧 **Use:**

- Prevent orphaned records (e.g., Orders without a valid Customer).

• Normalization (1NF, 2NF, 3NF)

Definition:
Process to **organize data** in tables to reduce redundancy and improve integrity.

🔧 **Use:**

- Avoid duplicate data, make updates easier, and ensure consistency.

🔍 **Examples:**

- **1NF (First Normal Form):** Each column has **atomic values** (no lists).
- **2NF (Second Normal Form):** 1NF + **no partial dependency** (columns depend on full primary key).
- **3NF (Third Normal Form):** 2NF + **no transitive dependency** (columns depend only on primary key).

• ER Diagrams

📘 **Definition:**
Entity-Relationship Diagram (ERD) **visualizes tables and their relationships**.

🔧 **Use:**

- Plan database structure before creating tables.
- Shows **entities, attributes, primary keys, foreign keys, and relationships**.

🔍 **Example:**

- **Entities:** Customers, Orders, Products
- **Relationships:** Customers → Orders → Products
- **ER Diagram:** Draw rectangles for tables, connect with lines showing relationships, label PK and FK.

Views

• CREATE VIEW

📘 **Definition:**
A **View** is a **virtual table** based on a SQL query. It does **not store data physically** (except materialized views).

🔧 **Use:**

- Simplify complex queries.
- Hide sensitive columns.
- Reuse queries easily.

• UPDATE View

📘 **Definition:**
Some views allow **updating, inserting, or deleting data** if they are **simple and based on a single table**.

🔧 **Use:**

- Modify underlying table data through the view.

• Drop View

📘 **Definition:**
DROP VIEW deletes a view from the database.

🔧 **Use:**

- Remove views that are no longer needed.

• Materialized view (concept)

📘 **Definition:**
A **Materialized View** is a view that **stores data physically**.

🔧 **Use:**

- Improve performance for **complex queries**.
- Can be refreshed periodically to reflect changes in underlying tables.

Indexes

• CREATE INDEX

📘 **Definition:**
An **Index** is a database structure that **speeds up data retrieval**. Think of it like an index in a book.

🔧 **Use:**

- Improve performance of SELECT queries.
- Especially useful on **columns used in WHERE, JOIN, ORDER BY**.

• Composite Index

📘 **Definition:**
A **Composite Index** is an index on **multiple columns together**.

🔧 **Use:**

- Improves performance when queries filter by **more than one column**.

• Clustered/Nonclustered indexes

📘 **Definition:**

- **Clustered Index:**
  - Sorts and stores the **actual table data** in index order.
  - Only **one clustered index per table**.
- **Nonclustered Index:**
  - Creates a **separate structure** pointing to table data.
  - You can have **many nonclustered indexes** per table.

🔧 **Use:**

- Clustered → fast retrieval + sorted data.
- Nonclustered → multiple lookup paths.

Temporary & Derived Tables

• TEMP tables

📘 **Definition:**
A **Temporary Table** is a table that **exists only during the session** or until the database connection is closed.

🔧 **Use:**

- Store **intermediate results** for complex queries.
- Helps in **breaking large queries into smaller steps**.

• Common Table Expressions (CTE)

📘 **Definition:**
A **CTE** is a **named temporary result set** that you can reference **within a single query** using WITH.

🔧 **Use:**

- Simplify **complex queries**, especially with aggregations or joins.
- Improves **readability** of SQL code.

• Recursive CTE

📘 **Definition:**
A **Recursive CTE** refers to itself to handle **hierarchical or iterative data** (like organization charts, bill of materials, or paths).

🔧 **Use:**

- Generate sequences or traverse hierarchical relationships.

Stored Procedures

• CREATE PROCEDURE

📘 **Definition:**
A **Stored Procedure** is a **pre-written SQL program** stored in the database. You can **call it anytime** without rewriting queries.

🔧 **Use:**

- Reuse SQL logic
- Simplify complex operations
- Improve performance by reducing client-server communication

• Parameters

📘 **Definition:**
Procedures can accept **input, output, or input-output parameters**.

🔧 **Use:**

- Pass values to procedures to **filter or calculate dynamically**.

• IF...ELSE logic

📘 **Definition:**
You can use **IF...ELSE** inside stored procedures for **conditional execution**.

🔧 **Use:**

- Execute different SQL based on conditions.

• Loops (concept)

📘 **Definition:**
Loops allow executing SQL statements **repeatedly** inside a procedure. Common types: WHILE or LOOP.

🔧 **Use:**

- Process **multiple rows or iterations** programmatically.