SQL Training Material

1.0 Introduction to Database

1.1 Introduction to DBMS

- **Description:** A Database Management System (DBMS) is software designed to store, retrieve, define, and manage data in a database. It acts as an interface between the user and the database, allowing users to create, read, update, and delete data.

- **Quiz:** What is the primary function of a DBMS?
  a) To perform complex mathematical calculations
  b) To design graphical user interfaces
  c) To manage and retrieve data efficiently
  d) To write operating systems
- **Tip/Trick:** Understanding the role of a DBMS is crucial before diving into SQL, as SQL is the language used to interact with a DBMS.
- **Fact:** Early database systems often used punch cards or magnetic tapes for data storage!

1.2 Characteristics of DBMS

- **Description:** Key characteristics of a DBMS include
  - data independence (separation of data from applications),
  - data security (access control),
  - data integrity (accuracy and consistency),
  - concurrency control (multiple users accessing data simultaneously), and
  - robust data recovery.

- **Quiz:** Which of the following is NOT a typical characteristic of a modern DBMS?
  a) Data Security
  b) Data Redundancy (aims to reduce it)
  c) Concurrency Control

d) Data Integrity
Answer: b) Data Redundancy

- **Tip/Trick:** Data integrity is maintained through various constraints defined within the DBMS.
- **Fact:** The concept of data independence allows applications to evolve without being tightly coupled to the physical storage structure of the data.

1.3 DBMS Models

- **Description:** Different ways databases can be structured. Common models include:
  - **Hierarchical:** Tree-like structure.
  - **Network:** More complex, allowing many-to-many relationships.
  - **Relational:** Data organized in tables (rows and columns), the most widely used model today.
  - **Object-Oriented:** Data stored as objects, similar to object-oriented programming.
  - **NoSQL:** A diverse category of non-relational databases for specific use cases (e.g., document, key-value, graph).

- **Quiz:** Which DBMS model organizes data into tables with rows and columns, making it the most common for SQL?
  a) Hierarchical
  b) Network
  c) Object-Oriented
  d) Relational
- **Tip/Trick:** Focus on the Relational model as SQL is primarily designed for it.
- **Fact:** The term "NoSQL" stands for "not only SQL," indicating they can coexist with relational databases for specific needs.

1.4 Relational DBMS

- **Description:** A Relational Database Management System (RDBMS) is a DBMS based on the **relational** model. Data is stored in relations (tables), which consist of rows (records/tuples) and columns (attributes/fields). Relationships between tables are established using **primary** and **foreign keys**.
- **Sample SQL Query/Code:**

```sql
-- Example of a simple RDBMS structure
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
```

```
    ProductName VARCHAR(255),
    Price DECIMAL(10, 2)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE,
    CustomerID INT,
    ProductID INT,
    Quantity INT,
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

- **Quiz:** In an RDBMS, what is a "row" also commonly referred to as?
  a) Attribute
  b) Relation
  c) Record
  d) Field
- **Tip/Trick:** Understanding primary and foreign keys is fundamental to relational database design and joining tables.
- **Fact:** Dr. E.F. Codd, an IBM researcher, is credited with developing the relational model in the late 1960s.

1.5 Data Integrity

- **Description:** Data integrity refers to the accuracy, consistency, and reliability of data over its entire lifecycle. It ensures that the data is valid and adheres to predefined rules. Types include
  - entity integrity (primary key uniqueness),
  - referential integrity (foreign key validity), and
  - domain integrity (data types, ranges).

PHYSICAL INTEGRITY refers to the rules and procedures which ensure the accuracy of data as it is stored and retrieved. Threats to physical integrity include external factors such as power outages, natural disasters and hackers and internal factors such as storage erosion, human error or design flaws. Typically, the affected dataset is unusable.

LOGICAL INTEGRITY seeks to ensure that the data accurately makes sense in a specific context (whether it's "logical"). Logical integrity also has the challenge of human errors

and design flaws. Thankfully, a dataset can be overwritten with new data and reused if it has a logical error. There are four topics of logical integrity as follows:

- **Domain integrity** refers to the range of values such as integer, text, or date which are acceptable to be stored in a particular column in a database. This set of values (the “domain”) has constraints which limit the format, amount, and types of data entered. All entries must be available in the domain of the data type. As shown in the example below, the entry for the number of Jean’s orders is not an integer so it is out of domain. This would cause the database management system to produce an error.

- Entity integrity uses primary keys to uniquely identify records saved in a table in a relational database. This prevents them from being duplicated. It also means they can’t be NULL because then you couldn’t uniquely identify the row if the other fields in the rows are the same. For example, you might have two customers with the same name and age, but without the unique identifier of the customer ID primary key, you could have errors or confusion when pulling the data.

Customer ID	Customer name	Age	Orders
44922945	Oliver Twist	34	21
30091920	James Gatz	42	9
75568215	Jean Finch	18	w#2@jk_1

Value is “out of domain” because it is not an integer.

Primary Key	Customer ID	Customer name	Age
	44922945	Oliver Twist	34
	30091920	James Gatz	42
Value cannot be NULL		James Gatz	42

- **Referential integrity** refers to the collection of rules and procedures used to maintain data consistency between two tables. These rules are embedded into the database structure regarding how foreign keys can be used to ensure the data entry is accurate, there is no duplicate data, and, as in the example below, data which doesn't apply is not entered. You can see below how referential integrity is maintained by not allowing an order ID which does not exist in the order table.

First Table (Customers)

Customer ID	Customer name	Age	Order ID
44922945	Oliver Twist	34	498721009-87
30091920	James Gatz	42	448902161-53
75568215	Jean Finch	18	324163384-92

This value is not permitted because this value is not defined as a primary key in the Order ID table.

Second Table (Orders)

Order ID	Product ID	Order date
498721009-87	KF-62	03162022
448902161-53	KF-65	04112022

- **User-defined** integrity acts as a way to catch errors which domain, referential and entity integrity do not. Here, you define your own specific business rules and constraints which trigger automatically when predefined events occur. For instance, you could define the constraint that customers must reside in a certain country to be entered into the database. Or, as in the example below, you might require that customers provide both first and last names.

Customer ID	Customer name	Age	Order ID
44922945	Oliver Twist	34	498721009-87
30091920	James Gatz	42	448902161-53
75568215	Jean	18	324163384-92

Value does not include a last name.

- **Ensure Data Integrity**
    - Educating business leaders on the risks
    - Establishing a robust data governance framework
    - Investing in the right tools and expertise.

- **Ensure Physical integrity**
    - You can ensure physical integrity in your database system by taking steps such as:
    - Having an uninterruptible power supply
    - Setting up redundant hardware
    - Controlling the physical environment against heat, dust or electromagnetic pulses
    - Using a clustered file system
    - Using error-correcting memory and algorithms
    - Using simple algorithms such as Damm or Luhn to detect human transcription errors
    - Using hash functions to detect computer-induced transcription errors

- **Sample SQL Query/Code:**

```
-- Enforcing entity integrity (Primary Key)
```

```sql
CREATE TABLE Users (
    UserID INT PRIMARY KEY, -- Ensures unique and non-NULL UserID
    UserName VARCHAR(100)
);

-- Enforcing referential integrity (Foreign Key)
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    UserID INT,
    OrderDate DATE,
    FOREIGN KEY (UserID) REFERENCES Users(UserID) -- Ensures
UserID exists in Users table
);

-- Enforcing domain integrity (CHECK constraint for age)
ALTER TABLE Employees
ADD CONSTRAINT CHK_Age CHECK (age >= 18);
```

- **Quiz:** Which type of integrity ensures that a foreign key value refers to an existing primary key value in another table?
  a) Entity Integrity
  b) Domain Integrity
  c) Referential Integrity
  d) User-defined Integrity
- **Tip/Trick:** Always define appropriate constraints (PRIMARY KEY, FOREIGN KEY, UNIQUE, NOT NULL, CHECK) when creating tables to maintain data integrity.
- **Fact:** Data integrity is so important that data warehousing projects often spend 80% of their time on data cleaning and ensuring integrity.

2.0 Structured Query Language (SQL)

2.1 Interacting SQL using SQL*Plus (General concept applies to any CLI tool)

- **Description:** SQL*Plus is a command-line interface tool specifically for Oracle databases, allowing users to execute SQL statements and scripts. Many other database systems have similar command-line tools (e.g., psql for PostgreSQL, MySQL Shell for MySQL) that provide an interactive environment for executing SQL.
- **Sample SQL Query/Code:** (Example of a command in such a tool)

```
-- In SQL*Plus or similar CLI tool
SQL> SELECT SYSDATE FROM DUAL; -- Oracle specific
-- Or for any database:
SQL> SELECT 'Hello, SQL World!';
```

- **Quiz:** What is the primary function of a command-line interface tool like SQL*Plus or psql?
  a) To design visual database diagrams
  b) To create web applications
  c) To execute SQL statements directly against a database
  d) To manage network configurations
- **Tip/Trick:** Learning keyboard shortcuts in your chosen CLI tool can significantly speed up your workflow.
- **Fact:** Before graphical user interfaces (GUIs), command-line tools were the primary way developers interacted with databases.

2.2 What is SQL?

- **Description** SQL (Structured Query Language) is the standard language for managing and manipulating relational databases. It is used to perform tasks such as creating database structures, querying data, inserting, updating, and deleting records, and managing user permissions.
- **Sample SQL Query/Code:**

```
-- A fundamental SQL query
SELECT name FROM Employees WHERE department = 'Sales';
```

- **Quiz:** What does SQL stand for?
  a) Simple Query Logic
  b) Standard Query Language
  c) Structured Query Language
  d) Sequential Query List
- **Tip/Trick:** SQL is a declarative language, meaning you describe what you want, not how to get it. The database engine figures out the best way to execute your request.

- **Fact:** Although SQL is standardized, almost every database system has its own proprietary extensions to the standard SQL.

2.3 Rules for SQL Statements

- **Description:**
  - SQL statements typically end with a semicolon (`;`). This allows multiple statements to be executed in a single batch.
  - SQL keywords are generally case-insensitive (e.g., `SELECT` is the same as `select`). However, data values and sometimes table/column names might be case-sensitive depending on the database configuration.
  - Whitespace (spaces, tabs, newlines) is generally ignored, allowing for formatting to improve readability.
  - Comments are used to explain code and are ignored by the database.
- **Sample SQL Query/Code:**

```sql
-- This is a single-line comment
SELECT
    employee_id, -- Selects the employee ID
    first_name   -- Selects the first name
FROM
    Employees; -- From the Employees table
/* This is a
   multi-line comment */
```

- **Quiz:** Which character is commonly used to terminate an SQL statement?
  a) Colon (:)
  b) Comma (,)
  c) Period (.)
  d) Semicolon (;)
- **Tip/Trick:** Consistent formatting (indentation, capitalization of keywords) makes your SQL code much easier to read and debug, especially for complex queries.
- **Fact:** Some older SQL environments might require each statement on its own line without a semicolon if executed individually.

2.4 Standard SQL Statement Groups

- **Description:** SQL commands are broadly categorized into several types based on their function:
    - DDL (Data Definition Language): Defines database structure (e.g., `CREATE`, `ALTER`, `DROP`).
    - DML (Data Manipulation Language): Manipulates data within objects (e.g., `INSERT`, `UPDATE`, `DELETE`).
    - DQL (Data Query Language): Retrieves data (e.g., `SELECT`).
    - DCL (Data Control Language): Manages permissions and access (e.g., `GRANT`, `REVOKE`).
    - TCL (Transaction Control Language): Manages transactions (e.g., `COMMIT`, `ROLLBACK`, `SAVEPOINT`).
- **Sample SQL Query/Code:**

```sql
-- DDL: Create a table
CREATE TABLE Products (ProductID INT PRIMARY KEY);
-- DML: Insert data
INSERT INTO Products (ProductID) VALUES (1);
-- DQL: Query data
SELECT * FROM Products;
-- DCL: Grant permission
GRANT SELECT ON Products TO public;
-- TCL: Commit transaction
COMMIT;
```

- **Quiz:** Which group of SQL commands is used to retrieve data from a database?
  a) DDL
  b) DML
  c) DCL
  d) DQL
- **Tip/Trick:** Remembering these categories helps you understand the purpose of each SQL command.

2.5 Basic Data Types

- **Description:** Data types specify the kind of data a column can hold. Choosing the correct data type is crucial for efficient storage and data integrity.
    - Numeric: `INT`, `SMALLINT`, `BIGINT`, `DECIMAL`, `NUMERIC`, `FLOAT`, `REAL`

        - String/Character: `VARCHAR`, `CHAR`, `TEXT`
        - Date/Time: `DATE`, `TIME`, `DATETIME`, `TIMESTAMP`
        - Boolean: `BOOLEAN` (or `BIT` in some systems)
        - Binary: `BLOB`, `VARBINARY`
    - **Sample SQL Query/Code:**

```sql
CREATE TABLE UserProfiles (
    UserID INT PRIMARY KEY,
    UserName VARCHAR(100) NOT NULL,
    Email VARCHAR(255) UNIQUE,
    Age SMALLINT,
    RegistrationDate DATE,
    IsActive BOOLEAN,
    Balance DECIMAL(10, 2)
);
```

- **Quiz:** Which SQL data type would you use to store a person's birth date?
  a) VARCHAR
  b) INT
  c) DATE
  d) BOOLEAN
- **Tip/Trick:** Always use the most specific data type for your data. For example, use SMALLINT for age (0-150) instead of INT if you know the range, to save storage.
- **Fact:** Different database systems have slightly different names or variations for common data types (e.g., TEXT in MySQL vs. VARCHAR(MAX) in SQL Server).

2.6 Rules for Naming a Table

- **Description:**
    - Start with a letter: Table names usually begin with an alphabet character.
    - Avoid spaces: Use underscores (_) instead of spaces (e.g., `customer_orders` not `customer orders`).
    - Be descriptive: Choose names that clearly indicate the table's purpose (e.g., `Employees`, `Products`, `Orders`).
    - Unique within schema: Table names must be unique within their schema.
    - Avoid reserved keywords: Do not use SQL keywords (e.g., `SELECT`, `FROM`, `WHERE`) as table names.

    - Maximum length: Most databases have a maximum length for identifiers (e.g., 30 characters for Oracle, 64 for MySQL).
- **Sample SQL Query/Code:**

```sql
-- Good table name
CREATE TABLE customer_feedback (
    feedback_id INT PRIMARY KEY,
    comment TEXT
);

-- Bad table name (reserved keyword)
-- CREATE TABLE ORDER (order_id INT);

-- Bad table name (spaces)
-- CREATE TABLE My Table (id INT);
```

- **Quiz:** Which character is generally recommended to replace spaces in table names for better readability and compatibility?
  a) Hyphen (-)
  b) Dot (.)
  c) Underscore (_)
  d) Space ( )
- **Tip/Trick:** Use singular names for tables (e.g., Employee instead of Employees) as each row represents a single entity. However, plural names are also commonly used and accepted (e.g., Employees). Choose one convention and stick to it.
- **Fact:** Some database systems allow identifiers with spaces or special characters if they are enclosed in specific delimiters (e.g., double quotes in PostgreSQL, square brackets in SQL Server), but it's generally best practice to avoid them.

2.7 Specifying Integrity Constraints

- **Description:** Integrity constraints are rules applied to columns or tables to enforce data validity and consistency.
    - `NOT NULL`: Ensures a column cannot contain NULL values.
    - `UNIQUE`: Ensures all values in a column (or set of columns) are distinct.
    - `PRIMARY KEY`: A column (or set of columns) that uniquely identifies each row; it's `NOT NULL`and `UNIQUE`.

    - `FOREIGN KEY`: A column (or set of columns) that links to the primary key of another table, ensuring referential integrity.
    - `CHECK`: Defines a condition that must be true for all values in a column.
    - `DEFAULT`: Provides a default value for a column if no value is explicitly provided.
- **Sample SQL Query/Code:**

```sql
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(255) NOT NULL UNIQUE,
    Price DECIMAL(10, 2) DEFAULT 0.00,
    StockQuantity INT CHECK (StockQuantity >= 0)
);

CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    ProductID INT,
    OrderDate DATE DEFAULT CURRENT_DATE,
    Quantity INT NOT NULL,
    FOREIGN KEY (ProductID) REFERENCES Products(ProductID)
);
```

- **Quiz:** Which constraint ensures that every value in a specific column is different from every other value in that column?
  a) NOT NULL
  b) PRIMARY KEY
  c) FOREIGN KEY
  d) UNIQUE
- **Tip/Trick:** Define constraints at the table creation stage for best practice, but you can also ALTER TABLE to add or modify them later.
- **Fact:** Constraints are processed by the database engine itself, making them more efficient and reliable than application-level validation.

2.8 DDL Statements: Create, Alter, Drop

- **Description:** DDL (Data Definition Language) commands are used to define, modify, and delete database objects like tables, indexes, and views.
    - `CREATE`: Used to create new database objects (e.g., `CREATE TABLE`, `CREATE INDEX`).
    - `ALTER`: Used to modify the structure of an existing database object (e.g., add/drop columns, change data types, add constraints).
    - `DROP`: Used to delete existing database objects (e.g., `DROP TABLE`, `DROP INDEX`).
    - `TRUNCATE`: Removes all rows from a table, but the table structure remains. It's faster than `DELETE` without a `WHERE` clause because it doesn't log individual row deletions.
- **Sample SQL Query/Code:**

```sql
-- CREATE TABLE
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50)
);

-- ALTER TABLE: Add a new column
ALTER TABLE Customers ADD Email VARCHAR(100);

-- ALTER TABLE: Modify an existing column (syntax varies by DB)
-- For PostgreSQL/Oracle: ALTER TABLE Customers ALTER COLUMN Email
TYPE VARCHAR(150);
-- For MySQL: ALTER TABLE Customers MODIFY COLUMN Email
VARCHAR(150);
-- For SQL Server: ALTER TABLE Customers ALTER COLUMN Email
VARCHAR(150);

-- ALTER TABLE: Drop a column
ALTER TABLE Customers DROP COLUMN LastName;

-- DROP TABLE
DROP TABLE Customers;
```

```sql
-- TRUNCATE TABLE
TRUNCATE TABLE Customers; -- Empties the table, but keeps
structure
```

- **Quiz:** Which DDL command is used to change the structure of an existing table?
  a) CREATE TABLE
  b) DROP TABLE
  c) ALTER TABLE
  d) MODIFY TABLE
- **Tip/Trick:** Be very careful with DROP and TRUNCATE commands in production environments, as they permanently delete data or objects and are often irreversible without backups.
- **Fact:** TRUNCATE is a DDL command because it implicitly commits the current transaction and resets identity columns, unlike DELETE which is DML.

2.9 Regular vs Temporary Tables

- **Description:**
  - Regular (Permanent) Tables: These tables persist in the database until explicitly dropped. Their data is stored permanently.
  - Temporary Tables: These tables exist only for the duration of a session or a transaction. They are automatically dropped when the session ends or the transaction commits/rolls back. They are useful for storing intermediate results in complex queries or procedures.
- **Sample SQL Query/Code:**

```sql
-- Creating a regular table
CREATE TABLE PermanentData (ID INT);

-- Creating a temporary table (syntax varies by DB)
-- SQL Server, Oracle:
CREATE TEMPORARY TABLE SessionData (ID INT); -- or GLOBAL/LOCAL
TEMPORARY in some
-- MySQL:
CREATE TEMPORARY TABLE SessionData (ID INT);
```

```
-- PostgreSQL:
CREATE TEMPORARY TABLE SessionData (ID INT) ON COMMIT DROP; -- or
ON COMMIT PRESERVE ROWS
```

- **Quiz:** What happens to a temporary table when the user's database session ends?
  a) It becomes a permanent table.
  b) It remains in the database until manually dropped.
  c) It is automatically dropped.
  d) It is converted into a view.
- **Tip/Trick:** Temporary tables are excellent for breaking down complex multi-step data processing into manageable chunks without affecting the main database schema.
- **Fact:** Some databases manage temporary tables in memory or in a dedicated temporary file space, making them very fast for short-lived data.

3.0 Data Manipulation Language (DML)

3.1 Inserting Rows Into a Table

- **Description:** The INSERT statement is used to add new rows (records) into an existing table. You can insert values for all columns, or specify a subset of columns.
- **Sample SQL Query/Code:**

```
-- Create a sample table
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(255),
    Price DECIMAL(10, 2)
);

-- Insert a row with values for all columns
INSERT INTO Products (ProductID, ProductName, Price)
VALUES (1, 'Laptop', 1200.00);

-- Insert a row, specifying only some columns (others will be NULL
or default)
```

```sql
INSERT INTO Products (ProductID, ProductName)
VALUES (2, 'Mouse');

-- Insert multiple rows in a single statement (supported by most modern DBs)
INSERT INTO Products (ProductID, ProductName, Price)
VALUES
    (3, 'Keyboard', 75.00),
    (4, 'Monitor', 300.00);
```

- **Quiz:** Which SQL keyword is used to add new records to a table?
  a) ADD RECORD
  b) CREATE ROW
  c) INSERT INTO
  d) PUT DATA
- **Tip/Trick:** Always list the columns explicitly in your INSERT statement (INSERT INTO TableName (col1, col2) ...). This makes your code more robust to future table schema changes.
- **Fact:** If you're inserting many rows from another table, INSERT INTO ... SELECT ... is generally much faster than multiple individual INSERT statements.

3.2 Deleting Rows from a Table

- **Description:** The DELETE statement is used to remove one or more rows from a table. The WHERE clause is crucial to specify which rows to delete. If WHERE is omitted, all rows will be deleted.
- **Sample SQL Query/Code:**

```sql
-- Delete a specific product by ID
DELETE FROM Products
WHERE ProductID = 1;

-- Delete all products with a price less than 100
DELETE FROM Products
WHERE Price < 100;
```

```
-- !!! DANGER ZONE: Delete all rows from the table (no WHERE
clause) !!!
-- DELETE FROM Products;
```

- **Quiz:** What is the consequence of executing a DELETE FROM Customers; statement without a WHERE clause?
  a) Only the Customers table structure is removed.
  b) Only the first row of the Customers table is removed.
  c) An error is generated.
  d) All rows in the Customers table are removed.
- **Tip/Trick:** Always use a SELECT statement with the same WHERE clause before executing a DELETE or UPDATE to verify you're targeting the correct rows.
- **Fact:** DELETE operations are logged, allowing them to be rolled back. TRUNCATE TABLE(DDL) is not logged per row and cannot be rolled back in the same way.

3.3 Updating Rows in a Table

- **Description:** The UPDATE statement is used to modify existing data within a table. The SETclause specifies the column(s) to modify and their new values, and the WHERE clause determines which rows will be affected.
- **Sample SQL Query/Code:**

```
-- Update the price of a specific product
UPDATE Products
SET Price = 1250.00
WHERE ProductID = 1;

-- Update the name for all products with a certain price
UPDATE Products
SET ProductName = 'Discounted Item'
WHERE Price < 100;

-- Update multiple columns for a specific product
UPDATE Products
SET
    ProductName = 'Gaming Laptop',
```

```
    Price = 1500.00
WHERE ProductID = 1;

-- !!! DANGER ZONE: Update all rows in the table (no WHERE clause) !!!
-- UPDATE Products SET Price = Price * 1.10; -- 10% price increase for ALL products
```

- **Quiz:** Which two clauses are essential for correctly using an UPDATE statement to modify specific rows?
  a) FROM and SELECT
  b) INSERT and VALUES
  c) GROUP BY and HAVING
  d) SET and WHERE
- **Tip/Trick:** Similar to DELETE, always run a SELECT with your WHERE clause first to ensure you're modifying the intended rows.

3.4 Database Objects

3.4.1 Index

- **Description:** An index is a special lookup table that the database search engine can use to speed up data retrieval. Think of it like an index in a book. Without an index, the database would have to scan every row in the table to find the matching data, which can be very slow for large tables.
- **Sample SQL Query/Code:**

```
-- Create an index on the 'ProductName' column to speed up searches by name
CREATE INDEX idx_product_name ON Products (ProductName);

-- Create a unique index to enforce uniqueness (like a UNIQUE constraint)
CREATE UNIQUE INDEX idx_email_unique ON Users (Email);
```

```sql
-- Drop an index
DROP INDEX idx_product_name ON Products;
```

- **Quiz:** What is the primary purpose of creating an index on a database table?
  a) To store redundant data for backup.
  b) To enforce data types on columns.
  c) To speed up data retrieval operations.
  d) To prevent users from modifying data.
- **Tip/Trick:** Index columns that are frequently used in WHERE clauses, JOIN conditions, or ORDER BY clauses. However, too many indexes can slow down INSERT, UPDATE, and DELETEoperations.
- **Fact:** The B-tree (balanced tree) is the most common data structure used for database indexes.

3.4.2 Synonym

- **Description:** A synonym is an alternative name for a table, view, sequence, or other database object. It provides an alias, which can be useful for simplifying long object names, providing abstraction from the underlying object's location, or granting users access to objects in another schema without specifying the schema name.
- **Sample SQL Query/Code:**

```sql
-- Create a synonym for a table (syntax varies slightly by DB,
e.g., Oracle, SQL Server)
-- Oracle:
CREATE SYNONYM my_products FOR scott.products;

-- SQL Server:
CREATE SYNONYM my_products FOR
database_name.schema_name.products;

-- Now you can query using the synonym
SELECT * FROM my_products;

-- Drop a synonym
DROP SYNONYM my_products;
```

- **Quiz:** What is the main benefit of using a synonym in SQL?
  a) It creates a copy of the table.
  b) It automatically backs up data.
  c) It provides an alternative name for a database object, simplifying queries.
  d) It speeds up data insertion.
- **Tip/Trick:** Synonyms are especially useful in environments with multiple schemas or complex object naming conventions, making it easier for users to query data without knowing the full object path.
- **Fact:** Synonyms don't store any data themselves; they are just pointers to the actual objects.

3.4.3 Sequence

- **Description:** A sequence is a database object that automatically generates unique numbers. It's commonly used to generate primary key values for tables, ensuring that each new record gets a unique identifier.
- **Sample SQL Query/Code:**

```sql
-- Create a sequence (syntax varies by DB)
-- Oracle:
CREATE SEQUENCE product_id_seq
START WITH 1
INCREMENT BY 1
NOCACHE;

-- PostgreSQL:
CREATE SEQUENCE product_id_seq START 1 INCREMENT 1;

-- Use the sequence to insert a new product (Oracle example)
INSERT INTO Products (ProductID, ProductName, Price)
VALUES (product_id_seq.NEXTVAL, 'New Gadget', 50.00);

-- Get current value (Oracle)
SELECT product_id_seq.CURRVAL FROM DUAL;

-- Drop a sequence
```

```
DROP SEQUENCE product_id_seq;
```

- **Quiz:** What is the primary purpose of a database sequence?
  a) To order query results.
  b) To encrypt data.
  c) To generate unique sequential numbers.
  d) To store historical data.
- **Tip/Trick:** While AUTO_INCREMENT (MySQL), IDENTITY (SQL Server), or SERIAL (PostgreSQL) columns are often simpler for auto-generating primary keys, sequences offer more control over the numbering (e.g., starting value, increment, caching).
- **Fact:** Sequences are thread-safe, meaning multiple concurrent sessions can request the next value without conflicting or generating duplicates.

3.4.4 Views

- **Description:** A view is a virtual table based on the result-set of an SQL query. A view contains rows and columns, just like a real table. However, it does not store data itself; it is derived from one or more underlying tables. Views simplify complex queries, restrict data access, and present data in different ways.
- **Sample SQL Query/Code:**

```
-- Create a view showing employee names and their department names
CREATE VIEW EmployeeDepartmentView AS
SELECT
    E.Name AS EmployeeName,
    D.DepartmentName
FROM
    Employees E
JOIN
    Departments D ON E.dept_id = D.DepartmentID;

-- Query the view just like a table
SELECT EmployeeName FROM EmployeeDepartmentView WHERE
DepartmentName = 'HR';

-- Drop a view
DROP VIEW EmployeeDepartmentView;
```

- **Quiz:** What is a key characteristic of a SQL view?
  a) It stores data physically like a table.
  b) It is a virtual table based on a SELECT query.
  c) It can only be created from a single table.
  d) It is automatically updated when the underlying data changes.
- **Tip/Trick:** Use views to simplify queries for end-users, hide complex joins or calculations, and provide a security layer by showing only specific columns or rows.
- **Fact:** Some advanced database systems support "materialized views," which physically store the result of the query and refresh periodically, offering performance benefits for complex aggregations.

4.0 Data Query Language (DQL)

4.1 SELECT Statement

- **Description:** The SELECT statement is the most fundamental SQL command. It is used to retrieve data from one or more tables in a database. It specifies which columns to retrieve, from which tables, and under what conditions.
- **Sample SQL Query/Code:**

```sql
-- Select all columns from the Employees table
SELECT * FROM Employees;

-- Select specific columns (Name, Salary) from the Employees table
SELECT Name, Salary FROM Employees;

-- Select with an alias for a column
SELECT Name AS EmployeeName, Salary FROM Employees;
```

- **Quiz:** Which SQL statement is used to retrieve data from a database?
  a) GET
  b) EXTRACT
  c) SELECT
  d) FETCH
- **Tip/Trick:** Using SELECT * is convenient for quick checks, but in production code, always list the specific columns you need. This improves performance and makes your queries more robust to schema changes.
- **Fact:** The SELECT statement is often considered the heart of SQL, as data retrieval is arguably the most common database operation.

4.2 Distinct Clause

- **Description:** The DISTINCT keyword is used in the SELECT statement to return only unique values for the specified column(s), eliminating duplicate rows from the result set.
- **Sample SQL Query/Code:**

```sql
-- Get all unique department names from the Employees table
SELECT DISTINCT department FROM Employees;

-- Get unique combinations of department and age
SELECT DISTINCT department, age FROM Employees;
```

- **Quiz:** What is the purpose of the DISTINCT keyword in a SELECT statement?
  a) To sort the results in ascending order.
  b) To count the number of rows.
  c) To return only unique values, eliminating duplicates.
  d) To filter rows based on a condition.
- **Tip/Trick:** DISTINCT applies to all selected columns. If you select multiple columns, DISTINCTreturns unique combinations of those columns.
- **Fact:** DISTINCT can sometimes be computationally expensive on very large datasets, as the database needs to sort and compare all values to identify duplicates.

4.3 Comparison, Arithmetic & Logical Operators

4.3.1 SQL Operators

- **Description:** Operators are used in SQL statements to perform comparisons, arithmetic calculations, or logical operations.
  - Comparison Operators: `=`, `!=` (or `<>`), `>`, `<`, `>=`, `<=`, `BETWEEN`, `LIKE`, `IN`, `IS NULL`
  - Arithmetic Operators: `+`, `-`, `*`, `/`, `%` (modulo)
  - Logical Operators: `AND`, `OR`, `NOT`
- **Sample SQL Query/Code:**

```sql
-- Comparison: Employees older than 30
SELECT Name, Age FROM Employees WHERE Age > 30;
```

```sql
-- Arithmetic: Calculate increased salary
SELECT Name, Salary, Salary * 1.05 AS NewSalary FROM Employees;

-- Logical: Employees in HR AND salary > 50000
SELECT Name, department, Salary FROM Employees
WHERE department = 'HR' AND Salary > 50000;

-- Combining logical (NOT) and comparison (IN)
SELECT Name, department FROM Employees
WHERE department NOT IN ('HR', 'IT');
```

- **Quiz:** Which logical operator returns true if at least one of its conditions is true?
  a) AND
  b) NOT
  c) OR
  d) BETWEEN
- **Tip/Trick:** Use parentheses () to clarify the order of operations in complex WHERE clauses involving multiple AND and OR conditions.
- **Fact:** SQL has operator precedence, just like in mathematics (e.g., multiplication and division are performed before addition and subtraction). Logical NOT has highest precedence, then AND, then OR.

4.3.2 The ORDER BY Clause

- **Description:** The ORDER BY clause is used to sort the result set of a SELECT statement. You can sort by one or more columns, in ascending (ASC, default) or descending (DESC) order.
- **Sample SQL Query/Code:**

```sql
-- Order employees by name alphabetically
SELECT Name, Department FROM Employees ORDER BY Name ASC;

-- Order employees by salary, highest first
SELECT Name, Salary FROM Employees ORDER BY Salary DESC;

-- Order by department (ASC) then by salary (DESC) within each
department
```

```sql
SELECT Name, Department, Salary FROM Employees
ORDER BY Department ASC, Salary DESC;
```

- **Quiz:** To sort the results of a query from the highest value to the lowest value in a specific column, which keyword would you use with ORDER BY?
  a) ASC
  b) SORT_DESC
  c) DESC
  d) TOP
- **Tip/Trick:** You can refer to columns by their alias or their position in the SELECT list (e.g., ORDER BY 2 DESC to sort by the second selected column) though using names is generally clearer.
- **Fact:** Without an ORDER BY clause, the order of rows returned by a SELECT statement is not guaranteed and can vary from one execution to another, even on the same data.

4.3.3 Tips and Tricks (for basic DQL/Operators)

- **Description:** Practical advice for writing effective and readable DQL queries.
- **Sample SQL Query/Code:**

```sql
-- Tip: Use aliases for table names for shorter, cleaner joins
SELECT E.Name, D.DepartmentName
FROM Employees AS E -- 'AS' is optional but good for clarity
JOIN Departments AS D ON E.dept_id = D.DepartmentID;

-- Trick: Use `LIKE` with wildcards (`%`, `_`) for pattern
matching
SELECT ProductName FROM Products WHERE ProductName LIKE 'L%'; --
Starts with 'L'
SELECT ProductName FROM Products WHERE ProductName LIKE '%top';
-- Ends with 'top'
SELECT ProductName FROM Products WHERE ProductName LIKE '_ouse';
-- 5 letters, second to last is 's'

-- Tip: Use `IN` for multiple OR conditions
```

```sql
SELECT Name FROM Employees WHERE Department IN ('HR', 'Sales',
'Marketing');
-- Equivalent to: WHERE Department = 'HR' OR Department = 'Sales'
OR Department = 'Marketing'
```

- **Quiz:** What wildcard character typically represents zero or more characters in the LIKEoperator?
  a) _ (underscore)
  b) ? (question mark)
  c) # (hash)
  d) % (percent sign)
- **Tip/Trick:** NOT LIKE and NOT IN are also very useful for excluding patterns or lists of values.
- **Fact:** Regular expressions (REGEXP or RLIKE) offer even more powerful pattern matching capabilities than LIKE in some databases (e.g., MySQL, PostgreSQL).

4.4 Aggregate Functions, Group By and Having Clause

4.4.1 The GROUP BY Clause

- **Description:** The GROUP BY clause is used with aggregate functions (like COUNT, SUM, AVG, MIN, MAX) to group rows that have the same values in specified columns into summary rows. It performs the aggregation on each group.
- **Sample SQL Query/Code:**

```sql
-- Count the number of employees in each department
SELECT department, COUNT(*) AS NumberOfEmployees
FROM Employees
GROUP BY department;

-- Calculate the average salary for each department
SELECT department, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY department;

-- Count employees per department and manager
```

```sql
SELECT department, manager_id, COUNT(*)
FROM Employees
GROUP BY department, manager_id;
```

- **Quiz:** When using an aggregate function like SUM() on a specific column, what does the GROUP BY clause allow you to do?
  a) Sum all values in the column globally.
  b) Calculate the sum for each distinct group of rows.
  c) Sort the results by the sum.
  d) Filter out rows before summation.
- **Tip/Trick:** Any column in the SELECT list that is not part of an aggregate function must be included in the GROUP BY clause.
- **Fact:** GROUP BY is one of the most powerful features in SQL for data summarization and reporting.

4.4.2 HAVING Clause

- **Description:** The HAVING clause is used to filter groups based on a specified condition. It works like a WHERE clause, but it applies to groups created by GROUP BY, not individual rows. WHERE filters rows before grouping, while HAVING filters groups after grouping and aggregation.
- **Sample SQL Query/Code:**

```sql
-- Find departments with more than 5 employees
SELECT department, COUNT(*) AS NumberOfEmployees
FROM Employees
GROUP BY department
HAVING COUNT(*) > 5;

-- Find departments where the average salary is above 60000
SELECT department, AVG(Salary) AS AverageSalary
FROM Employees
GROUP BY department
HAVING AVG(Salary) > 60000;
```

- **Quiz:** What is the primary difference between a WHERE clause and a HAVING clause?
  a) WHERE filters rows, HAVING sorts results.

b) WHERE is used with SELECT, HAVING is used with INSERT.
c) WHERE filters individual rows before grouping, HAVING filters groups after aggregation.
d) WHERE uses aggregate functions, HAVING does not.

- **Tip/Trick:** If you can filter rows before grouping using WHERE, it's usually more efficient to do so, as it reduces the data volume for the aggregation process. Use HAVING only when the filtering condition relies on an aggregate function.

4.4.3 ROLLUP Operation

- **Description:** The ROLLUP operation, used with GROUP BY, generates subtotals for combinations of columns and a grand total. It creates aggregation for hierarchies of columns specified in the GROUP BY clause.
- **Sample SQL Query/Code:**

```
-- Calculate sum of salaries by department, then by department and manager, and a grand total.
SELECT department, manager_id, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY ROLLUP(department, manager_id);
-- Output might look like:
-- Department | Manager_ID | TotalSalary
-- Sales      | 101        | 50000
-- Sales      | 102        | 70000
-- Sales      | NULL       | 120000 (Subtotal for Sales)
-- HR         | 201        | 45000
-- HR         | NULL       | 45000  (Subtotal for HR)
-- NULL       | NULL       | 165000 (Grand Total)
```

- **Quiz:** What additional rows does ROLLUP typically add to a GROUP BY result?
  a) Duplicate rows
  b) Only rows with NULL values
  c) Subtotal and grand total rows
  d) Rows from another table
- **Tip/Trick:** ROLLUP is powerful for generating summary reports with hierarchical totals. The order of columns in ROLLUP matters as it defines the hierarchy.
- **Fact:** ROLLUP is a type of "grouping set" operation, along with CUBE.

4.4.4 CUBE Operation

- **Description:** The CUBE operation, also used with GROUP BY, generates subtotals for all possible combinations of the grouping columns. Unlike ROLLUP which creates a hierarchy, CUBE creates a full "data cube" of aggregations.
- **Sample SQL Query/Code:**

```
-- Calculate sum of salaries for all possible combinations of department and manager
SELECT department, manager_id, SUM(Salary) AS TotalSalary
FROM Employees
GROUP BY CUBE(department, manager_id);
-- Output will include:
-- (department, manager_id) totals
-- (department, NULL) totals (subtotal for each department)
-- (NULL, manager_id) totals (subtotal for each manager)
-- (NULL, NULL) grand total
```

- **Quiz:** How does CUBE differ from ROLLUP in terms of subtotals generated?
  a) CUBE only generates a grand total.
  b) CUBE generates only subtotals for individual columns.
  c) CUBE generates subtotals for all possible combinations of grouping columns, not just hierarchical ones.
  d) CUBE does not generate any subtotals.
- **Tip/Trick:** CUBE is more computationally intensive than ROLLUP but provides a comprehensive set of summary aggregations, ideal for analytical reporting where all cross-tabulations are needed.
- **Fact:** The CUBE operation gets its name because if you have N grouping columns, it generates 2^N grouping combinations, which can be visualized as a multi-dimensional cube.

4.4.5 Tips and Tricks (for Aggregates/Grouping)

- **Description:** Practical advice for working with aggregate functions and grouping.
- **Sample SQL Query/Code:**

```
-- Tip: Use COUNT(DISTINCT column) to count unique values in a group
SELECT department, COUNT(DISTINCT manager_id) AS UniqueManagers
FROM Employees
```

```sql
GROUP BY department;

-- Trick: Be careful with NULLs in aggregate functions
-- COUNT(*) counts all rows. COUNT(column_name) counts non-NULL values in that column.
SELECT COUNT(*), COUNT(manager_id) FROM Employees;

-- Tip: Filtering with WHERE before GROUP BY for efficiency
SELECT department, AVG(Salary)
FROM Employees
WHERE Salary > 40000 -- Filter out low earners before grouping
GROUP BY department
HAVING COUNT(*) > 2; -- Then filter groups
```

- **Quiz:** If a column contains NULL values, how does COUNT(column_name) behave differently from COUNT(*)?
  a) COUNT(column_name) counts NULL values, COUNT(*) does not.
  b) COUNT(column_name) only counts unique values.
  c) COUNT(column_name) only counts non-NULL values, COUNT(*) counts all rows.
  d) They behave identically.
- **Tip/Trick:** Always ensure consistency in your GROUP BY clause. If a column is in SELECT but not aggregated, it must be in GROUP BY.
- **Fact:** MIN() and MAX() aggregate functions can be applied to text and date data types, not just numbers, returning the alphabetically first/last or chronologically earliest/latest values.

4.5 SQL (Single Row) Functions

4.5.1 Character Functions

- **Description:** Functions that operate on string (character) data, performing tasks like converting case, extracting substrings, or padding.
- **Sample SQL Query/Code:**

```sql
    -- UPPER: Convert to uppercase
SELECT UPPER(Name) AS UppercaseName FROM Employees;
```

```sql
-- LOWER: Convert to lowercase
SELECT LOWER(Department) AS LowercaseDept FROM Employees;

-- LENGTH (or LEN): Get length of string
SELECT Name, LENGTH(Name) AS NameLength FROM Employees;

-- SUBSTR (or SUBSTRING): Extract a part of a string
SELECT SUBSTR(Email, 1, INSTR(Email, '@') - 1) AS Username FROM
Employees; -- Oracle/PostgreSQL INSTR
-- SQL Server: SELECT SUBSTRING(Email, 1, CHARINDEX('@', Email) -
1) AS Username FROM Employees;

-- CONCAT: Combine strings (syntax varies)
-- SQL Standard: SELECT CONCAT(FirstName, ' ', LastName) FROM
Employees;
-- Or using || (PostgreSQL, Oracle): SELECT FirstName || ' ' ||
LastName FROM Employees;
```

- **Quiz:** Which function would you use to convert a string to all uppercase letters?
  a) TO_UPPER()
  b) STR_UPPER()
  c) UPPER()
  d) CAPITALIZE()
- **Tip/Trick:** Be aware that character function names and exact syntax can vary significantly between different database systems (e.g., LENGTH vs LEN, SUBSTR vs SUBSTRING, INSTR vs CHARINDEX).
- **Fact:** Many character functions have roots in older programming languages like FORTRAN and COBOL.

4.5.2 Number Functions

- **Description:** Functions that perform mathematical operations on numeric data.
- **Sample SQL Query/Code:**

```sql
-- ROUND: Round a number to a specified number of decimal places
SELECT ROUND(123.456, 2) AS RoundedNumber; -- Output: 123.46
```

```sql
-- TRUNC (or TRUNCATE): Truncate a number to a specified number of decimal places
SELECT TRUNC(123.456, 2) AS TruncatedNumber; -- Output: 123.45 (Oracle/PostgreSQL)
-- MySQL: SELECT TRUNCATE(123.456, 2);

-- ABS: Absolute value
SELECT ABS(-10) AS AbsoluteValue; -- Output: 10

-- CEIL (or CEILING): Round up to the nearest integer
SELECT CEIL(12.1) AS CeilValue; -- Output: 13

-- FLOOR: Round down to the nearest integer
SELECT FLOOR(12.9) AS FloorValue; -- Output: 12

-- MOD (or %): Modulo (remainder of a division)
SELECT MOD(10, 3) AS Remainder; -- Output: 1 (Oracle/PostgreSQL)
-- SQL Server, MySQL: SELECT 10 % 3;
```

- **Quiz:** Which function would you use to round a number down to the nearest whole integer?
  a) ROUND()
  b) CEIL()
  c) FLOOR()
  d) TRUNC()
- **Tip/Trick:** Understand the subtle differences between ROUND and TRUNC (or TRUNCATE) when dealing with decimal numbers. ROUND rounds to the nearest, TRUNC simply cuts off.
- **Fact:** Many programming languages share similar math functions with SQL, making it easier to transition between them.

4.5.3 Data Conversion Function

- **Description:** Functions used to convert data from one data type to another. This is crucial for ensuring compatibility between different data types in expressions or for displaying data in a specific format.

- **Sample SQL Query/Code:**

```sql
-- TO_CHAR (Oracle/PostgreSQL): Convert number or date to string
SELECT TO_CHAR(12345, 'FM99,999') AS FormattedNumber; -- Output:
"12,345"
SELECT TO_CHAR(CURRENT_DATE, 'YYYY-MM-DD') AS FormattedDate; --
Output: "2023-10-26" (example date)

-- TO_NUMBER (Oracle/PostgreSQL): Convert string to number
SELECT TO_NUMBER('123.45') + 10 AS ConvertedNumber; -- Output:
133.45

-- TO_DATE (Oracle/PostgreSQL): Convert string to date
SELECT TO_DATE('2023-10-26', 'YYYY-MM-DD') AS ConvertedDate;

-- CAST (SQL Standard - widely supported):
SELECT CAST('123' AS INT) + 10 AS ConvertedInt; -- Output: 133
SELECT CAST(456.78 AS DECIMAL(5,1)) AS ConvertedDecimal; --
Output: 456.8

-- CONVERT (SQL Server):
SELECT CONVERT(INT, '123') + 10;
SELECT CONVERT(VARCHAR(10), GETDATE(), 120); -- YYYY-MM-DD format
```

- **Quiz:** Which standard SQL function is used to convert data from one type to another (e.g., string to integer)?
  a) CHANGE_TYPE()
  b) TRANSFORM()
  c) CAST()
  d) ALTER_TYPE()
- **Tip/Trick:** Always explicitly convert data types when necessary rather than relying on implicit conversions, as implicit conversions can lead to unexpected results or performance issues.
- **Fact:** The specific format codes for converting dates/times to strings vary significantly across database systems (e.g., Oracle/PostgreSQL TO_CHAR vs. SQL Server CONVERT styles).

4.5.4 Formats for Date Functions

- **Description:** When converting between strings and dates/times, or formatting date/time output, specific format models are used to define the pattern of the date and time components. These formats are database-specific but share common concepts.
- **Sample SQL Query/Code:**

```
-- Common Date/Time Format Elements (examples, specific to
Oracle/PostgreSQL TO_CHAR/TO_DATE):
-- YYYY: 4-digit year (e.g., 2023)
-- MM: 2-digit month (01-12)
-- DD: 2-digit day (01-31)
-- HH24: 2-digit hour (00-23)
-- MI: 2-digit minute (00-59)
-- SS: 2-digit second (00-59)
-- MON: Abbreviated month name (e.g., OCT)
-- MONTH: Full month name (e.g., OCTOBER)
-- DY: Abbreviated day of week (e.g., THU)
-- DAY: Full day of week (e.g., THURSDAY)

-- Example using CURRENT_DATE (PostgreSQL)
SELECT TO_CHAR(CURRENT_DATE, 'Day, Month DD, YYYY') AS
FormattedDate;
-- Output: "Thursday, October 26, 2023" (example)

-- Example using TO_DATE (Oracle)
SELECT TO_DATE('26-OCT-2023 14:30:00', 'DD-MON-YYYY HH24:MI:SS')
AS DateTimeValue;
```

- **Quiz:** In a date format string (e.g., for TO_CHAR), what does YYYY typically represent?
  a) Day of the year
  b) Four-digit year
  c) Two-digit year
  d) Year and month
- **Tip/Trick:** When parsing strings to dates, ensure your format model exactly matches the input string's format, including separators (e.g., MM/DD/YYYY for 01/26/2023).

- **Fact:** Storing dates as actual DATE or TIMESTAMP data types is always better than storing them as VARCHAR, as it allows for proper date arithmetic and indexing.

4.5.5 Date Functions

- **Description:** Functions specifically designed to perform calculations and extract components from date and time values.
- **Sample SQL Query/Code:**

```sql
-- CURRENT_DATE / GETDATE() / NOW(): Get current date/time
SELECT CURRENT_DATE; -- PostgreSQL, Oracle
-- SQL Server: SELECT GETDATE();
-- MySQL: SELECT NOW();

-- EXTRACT (PostgreSQL, Oracle): Extract part of a date/time
SELECT EXTRACT(YEAR FROM hire_date) AS HireYear FROM Employees;
SELECT EXTRACT(MONTH FROM hire_date) AS HireMonth FROM
Employees;
SELECT EXTRACT(DAY FROM hire_date) AS HireDay FROM Employees;

-- DATEDIFF (SQL Server, MySQL): Calculate difference between
dates
-- SQL Server: SELECT DATEDIFF(year, hire_date, GETDATE()) AS
YearsSinceHire FROM Employees;
-- MySQL: SELECT DATEDIFF(NOW(), hire_date) AS DaysSinceHire FROM
Employees;

-- DATE_ADD / DATE_SUB (MySQL): Add/subtract intervals
-- MySQL: SELECT DATE_ADD(CURRENT_DATE, INTERVAL 7 DAY); -- Add 7
days
-- PostgreSQL: SELECT CURRENT_DATE + INTERVAL '7 days';
-- Oracle: SELECT SYSDATE + 7;
```

- **Quiz:** Which SQL function is commonly used to find the difference between two dates (e.g., in days, months, or years)?
  a) DATE_ADD()

b) EXTRACT()
c) CURRENT_DATE()
d) DATEDIFF() (or equivalent like date arithmetic)

- **Tip/Trick:** Be careful with date arithmetic across different database systems as they have highly varied syntax. Always test thoroughly.
- **Fact:** Calculating age from a birthdate can be tricky due to leap years and date formatting. It's often best to use dedicated date functions for this.

4.5.6 Miscellaneous Functions

- **Description:** A category for other useful single-row functions that don't fit neatly into character, number, or date categories, such as NVL, COALESCE, CASE.
- **Sample SQL Query/Code:**

```sql
-- NVL (Oracle) / ISNULL (SQL Server) / COALESCE (SQL Standard):
-- Replace NULL values with a specified value
SELECT NVL(commission, 0) AS ActualCommission FROM Sales; -- Oracle
SELECT ISNULL(commission, 0) AS ActualCommission FROM Sales; -- SQL Server
SELECT COALESCE(commission, 0) AS ActualCommission FROM Sales; -- Standard SQL, widely supported

-- CASE Statement: Conditional logic
SELECT
    Name,
    Salary,
    CASE
        WHEN Salary >= 100000 THEN 'High Earner'
        WHEN Salary >= 50000 THEN 'Mid-Range'
        ELSE 'Junior'
    END AS SalaryBracket
FROM Employees;

-- DECODE (Oracle specific): Similar to a simple CASE statement
-- SELECT Name, DECODE(Department, 'IT', 'Tech', 'HR', 'People', 'Other') AS DepartmentGroup FROM Employees;
```

- **Quiz:** Which SQL function allows you to implement conditional logic, returning different values based on different conditions, similar to an if-else statement?
  a) COALESCE()
  b) NVL()
  c) CASE
  d) DECODE()
- **Tip/Trick:** COALESCE is generally preferred over NVL or ISNULL if your database supports it, as it's standard SQL and can take multiple arguments, returning the first non-NULL value.
- **Fact:** The CASE statement is incredibly versatile and can be used in SELECT, WHERE, ORDER BY, and GROUP BY clauses.

4.5.7 Tips and Tricks (for Single Row Functions)

- **Description:** General tips for effectively using SQL's built-in functions.
- **Sample SQL Query/Code:**

```sql
-- Tip: Chain functions for complex transformations
SELECT UPPER(SUBSTR(ProductName, 1, 3)) AS AbbrName
FROM Products; -- Gets first 3 chars, then converts to uppercase

-- Trick: Use `NULLIF` to avoid division by zero
SELECT TotalSales / NULLIF(NumberOfOrders, 0) AS AvgSalePerOrder
FROM SalesData; -- If NumberOfOrders is 0, it becomes NULL,
avoiding error

-- Tip: Understand the distinction between single-row functions
and aggregate functions.
-- Single-row functions operate on each row independently.
-- Aggregate functions operate on a set of rows and return a
single summary value.
```

- **Quiz:** What is the result of SELECT NULLIF(10, 10);?
  a) 10
  b) 0
  c) NULL
  d) An error

- **Tip/Trick:** Experiment with different functions and combinations. The best way to learn them is by trying them out on sample data.
- **Fact:** Many single-row functions have their roots in mathematical or string manipulation functions found in programming languages like C or Pascal.

4.6 Transactions

4.6.1 Transaction

- **Description:** A transaction is a single logical unit of work performed in a database. It can consist of one or more SQL statements, but it is treated as an indivisible sequence of operations. Transactions follow the ACID properties (Atomicity, Consistency, Isolation, Durability) to ensure data reliability.
- **Sample SQL Query/Code:**

```sql
-- Start a transaction (syntax varies, AUTOCOMMIT is often on by
default)
-- SQL Server, PostgreSQL, Oracle:
BEGIN TRANSACTION; -- or START TRANSACTION (MySQL)

-- Perform multiple DML operations within the transaction
UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;
UPDATE Accounts SET Balance = Balance + 100 WHERE AccountID = 2;

-- If both updates succeed, commit the transaction
COMMIT;

-- If an error occurs, rollback (undo) the transaction
-- ROLLBACK;
```

- **Quiz:** Which property of a database transaction ensures that all operations within it are completed successfully, or none are?
  a) Consistency
  b) Isolation
  c) Durability
  d) Atomicity

- **Tip/Trick:** Always explicitly COMMIT or ROLLBACK your transactions when working in a transaction-aware environment to avoid leaving pending changes or holding locks unnecessarily.
- **Fact:** The concept of database transactions and ACID properties was largely formalized in the 1980s and is fundamental to reliable database systems.

4.6.2 Commit Command

- **Description:** The COMMIT command is used to permanently save all changes made during the current transaction to the database. Once committed, the changes become permanent and visible to other users.
- **Sample SQL Query/Code:**

```
BEGIN TRANSACTION; -- Start a new transaction
INSERT INTO LogTable (Message) VALUES ('Starting batch update');
UPDATE Orders SET Status = 'Processed' WHERE OrderDate <
CURRENT_DATE - INTERVAL '30 days';
DELETE FROM OldData WHERE CreatedDate < CURRENT_DATE - INTERVAL
'1 year';
COMMIT; -- Permanently save all changes
```

- **Quiz:** After a BEGIN TRANSACTION, what command makes the changes permanent in the database?
  a) SAVEPOINT
  b) ROLLBACK
  c) END TRANSACTION
  d) COMMIT
- **Tip/Trick:** If your database has AUTOCOMMIT enabled (common in MySQL workbench, for example), each individual DML statement is automatically committed unless explicitly wrapped in a transaction block.

4.6.3 Rollback and Savepoints

- **Description:**
  - ROLLBACK: The `ROLLBACK` command is used to undo all changes made during the current transaction, effectively restoring the database to its state before the transaction began.

    - SAVEPOINT: A `SAVEPOINT` is a point within a transaction to which you can later roll back. This allows for partial rollbacks within a larger transaction, rather than having to undo the entire transaction.
- Sample SQL Query/Code:

```
BEGIN TRANSACTION;

INSERT INTO TempTable (ID, Value) VALUES (1, 'Initial');
SAVEPOINT sp1; -- Set a savepoint

INSERT INTO TempTable (ID, Value) VALUES (2, 'Second');
SAVEPOINT sp2; -- Set another savepoint

INSERT INTO TempTable (ID, Value) VALUES (3, 'Third - Problem
here');
-- Something goes wrong with the 'Third' insert

ROLLBACK TO sp2; -- Rollback only to sp2, 'Third' insert is
undone, 'Initial' and 'Second' remain
-- Now, you might fix the problem and continue
INSERT INTO TempTable (ID, Value) VALUES (4, 'Fourth - Fixed');

COMMIT; -- Commit the 'Initial', 'Second', and 'Fourth' inserts
```

- **Quiz:** If you execute ROLLBACK TO SAVEPOINT_NAME;, what happens to the changes made after that specific savepoint?
  a) They are committed.
  b) They are permanently saved to the database.
  c) They are undone, but changes before the savepoint are retained.
  d) The entire transaction is rolled back.
- **Tip/Trick:** Savepoints are useful in complex stored procedures or applications where you might want to retry parts of a transaction if certain conditions aren't met, without losing all work.

4.7 Joins and Subqueries

4.7.1 Inner/Equi Join

- **Description:** An INNER JOIN (or JOIN) returns only the rows that have matching values in both tables based on the join condition. It's the most common type of join. "Equi Join" specifically refers to an inner join using an equality operator (=).
- **Sample SQL Query/Code:**

```sql
-- Get employee names and their corresponding department names
SELECT E.Name, D.DepartmentName
FROM Employees E -- Alias for Employees table
INNER JOIN Departments D ON E.dept_id = D.DepartmentID; -- Join condition

-- Implicit Inner Join (older syntax, not recommended for clarity)
-- SELECT E.Name, D.DepartmentName
-- FROM Employees E, Departments D
-- WHERE E.dept_id = D.DepartmentID;
```

- **Quiz:** What type of join only returns rows when there is a match in both tables based on the join condition?
  a) LEFT JOIN
  b) FULL OUTER JOIN
  c) INNER JOIN
  d) CROSS JOIN
- **Tip/Trick:** Always use explicit JOIN syntax (INNER JOIN ... ON ...) instead of implicit joins in the WHERE clause for better readability and to avoid accidental Cartesian products.
- **Fact:** An INNER JOIN is sometimes compared to an intersection in set theory.

4.7.2 Outer Join (LEFT, RIGHT, FULL)

- **Description:** Outer joins return all rows from one table, and the matching rows from the other. If there's no match, NULL values are returned for the columns from the non-matching side.
  - LEFT (OUTER) JOIN: Returns all rows from the left table, and matching rows from the right.
  - RIGHT (OUTER) JOIN: Returns all rows from the right table, and matching rows from the left.

    - FULL (OUTER) JOIN: Returns all rows when there is a match in one of the tables. If there is no match, the rows from the unmatched side will have `NULL` values.
- **Sample SQL Query/Code:**

```sql
-- LEFT JOIN: Get all employees, and their department name if they have one.
-- Employees without a department will still appear, with NULL for DepartmentName.
SELECT E.Name, D.DepartmentName
FROM Employees E
LEFT JOIN Departments D ON E.dept_id = D.DepartmentID;

-- RIGHT JOIN: Get all departments, and the names of employees in them.
-- Departments with no employees will still appear, with NULL for Name.
SELECT E.Name, D.DepartmentName
FROM Employees E
RIGHT JOIN Departments D ON E.dept_id = D.DepartmentID;

-- FULL OUTER JOIN: Get all employees AND all departments.
-- Employees without departments and departments without employees will both appear.
SELECT E.Name, D.DepartmentName
FROM Employees E
FULL OUTER JOIN Departments D ON E.dept_id = D.DepartmentID;
```

- **Quiz:** Which type of join returns all records from the left table, and the matching records from the right table (with NULL where there is no match)?
  a) INNER JOIN
  b) LEFT JOIN
  c) RIGHT JOIN
  d) FULL OUTER JOIN
- **Tip/Trick:** Think about which table you want to preserve all rows from. If it's the first table in your FROM clause, use LEFT JOIN. If it's the second, use RIGHT JOIN.

- **Fact:** FULL OUTER JOIN can sometimes be emulated using a LEFT JOIN combined with a UNION ALL and a RIGHT JOIN (with a condition to exclude duplicates), especially in databases that don't directly support FULL OUTER JOIN.

4.7.3 Self-Join

- **Description:** A SELF-JOIN is a join in which a table is joined with itself. It is used to combine and compare rows within the same table. This is typically done by using aliases to differentiate between the two instances of the table. Common use cases include finding employees who report to a specific manager (who is also an employee), or finding duplicate records.
- Sample SQL Query/Code:

```sql
-- Find employees and their managers (assuming manager_id points to another employee's ID)
SELECT
    E.Name AS EmployeeName,
    M.Name AS ManagerName
FROM
    Employees E
JOIN
    Employees M ON E.manager_id = M.ID;

-- Find employees who earn more than their manager
SELECT
    E.Name AS EmployeeName,
    E.Salary AS EmployeeSalary,
    M.Name AS ManagerName,
    M.Salary AS ManagerSalary
FROM
    Employees E
JOIN
    Employees M ON E.manager_id = M.ID
WHERE
    E.Salary > M.Salary;
```

- **Quiz:** When performing a self-join, what is absolutely necessary to distinguish between the two instances of the same table?
  a) A WHERE clause
  b) A GROUP BY clause
  c) Table aliases
  d) A UNION operator
- **Tip/Trick:** Always use clear aliases for each instance of the table in a self-join (e.g., E for employee, M for manager) to avoid ambiguity.
- **Fact:** Self-joins are particularly useful for querying hierarchical data structures where parent-child relationships are stored within the same table.

4.7.4 Subquery

- **Description:** A subquery (or inner query, nested query) is a query embedded inside another SQL query. The inner query executes first, and its result is used by the outer query. Subqueries can appear in SELECT, FROM, WHERE, and HAVING clauses.
- Sample SQL Query/Code:

```sql
-- Subquery in WHERE clause (to find employees in departments with average salary > 60000)
SELECT Name, Salary
FROM Employees
WHERE dept_id IN (
    SELECT dept_id
    FROM Employees
    GROUP BY dept_id
    HAVING AVG(Salary) > 60000
);

-- Subquery in FROM clause (derived table)
SELECT D.DepartmentName, HighEarners.NumHighEarners
FROM Departments D
JOIN (
    SELECT dept_id, COUNT(*) AS NumHighEarners
    FROM Employees
    WHERE Salary > 80000
    GROUP BY dept_id
```

```
) AS HighEarners ON D.DepartmentID = HighEarners.dept_id;
```

- **Quiz:** Where can a subquery NOT typically be placed in a standard SQL statement?
  a) SELECT clause
  b) FROM clause
  c) WHERE clause
  d) CREATE DATABASE statement
- **Tip/Trick:** While powerful, deeply nested subqueries can sometimes be hard to read and optimize. Consider using CTEs (WITH clause) for better readability and potentially better performance.
- **Fact:** Subqueries are often used when a query needs to compute an intermediate result that depends on the data of the outer query.

4.7.5 SUBQUERIES Using Comparison Operators

- **Description:** Subqueries can be used with comparison operators (=, !=, <, >, <=, >=) when the subquery returns a single value (a scalar value).
- Sample SQL Query/Code:

```
-- Find employees who earn more than the average salary of all
employees
SELECT Name, Salary
FROM Employees
WHERE Salary > (SELECT AVG(Salary) FROM Employees);

-- Find the employee with the highest salary
SELECT Name, Salary
FROM Employees
WHERE Salary = (SELECT MAX(Salary) FROM Employees);
```

- **Quiz:** What must be true about the result of a subquery used with a standard comparison operator (=, >, <)?
  a) It must return multiple rows.
  b) It must return multiple columns.
  c) It must return a single value (scalar).
  d) It must be sorted in ascending order.
- **Tip/Trick:** If a subquery returns more than one row and you use a single-value comparison operator, it will result in an error. For multiple rows, use IN, ANY, or ALL.

4.7.6 Co-related Subquery (Correlated Subquery)

- **Description:** A correlated subquery is a subquery that depends on the outer query for its values. It executes once for each row processed by the outer query. This means the inner query cannot be executed independently; it references a column from the outer query.
- Sample SQL Query/Code:

```
-- Find employees whose salary is higher than the average salary
in their OWN department
SELECT E1.Name, E1.Salary, E1.department
FROM Employees E1
WHERE E1.Salary > (
    SELECT AVG(E2.Salary)
    FROM Employees E2
    WHERE E2.department = E1.department -- Correlated part:
E2.department depends on E1.department
);

-- Find the employees who are the highest paid in their department
SELECT E1.Name, E1.Salary, E1.department
FROM Employees E1
WHERE E1.Salary = (
    SELECT MAX(E2.Salary)
    FROM Employees E2
    WHERE E2.department = E1.department
);
```

- **Quiz:** What is a defining characteristic of a correlated subquery?
  a) It executes only once for the entire outer query.
  b) It references a column from the outer query and executes for each row of the outer query.
  c) It must return multiple rows.
  d) It cannot be used in a WHERE clause.
- **Tip/Trick:** Correlated subqueries can sometimes be less performant than equivalent queries using joins or window functions, especially on large datasets, because they are re-evaluated for each row. Consider alternatives if performance is critical.

- **Fact:** Correlated subqueries are sometimes called "dependent subqueries."

4.7.7 Exists / Not Exists Operator

- **Description:** The EXISTS operator is used with a subquery to test for the existence of rows. It returns TRUE if the subquery returns any rows, and FALSE otherwise. NOT EXISTS returns TRUE if the subquery returns no rows. These are efficient because the subquery can stop as soon as it finds the first matching row.
- Sample SQL Query/Code:

```sql
-- Find departments that have at least one employee
SELECT DepartmentName
FROM Departments D
WHERE EXISTS (SELECT 1 FROM Employees E WHERE E.dept_id =
D.DepartmentID);

-- Find departments that have NO employees
SELECT DepartmentName
FROM Departments D
WHERE NOT EXISTS (SELECT 1 FROM Employees E WHERE E.dept_id =
D.DepartmentID);

-- Find employees who have ordered a product
SELECT Name FROM Customers C
WHERE EXISTS (SELECT 1 FROM Orders O WHERE O.CustomerID =
C.CustomerID);
```

- **Quiz:** What does the EXISTS operator check for?
  a) Whether a value is equal to another value.
  b) The number of rows returned by a subquery.
  c) The presence of any rows returned by a subquery.
  d) If a column contains NULL values.
- **Tip/Trick:** EXISTS is often more performant than IN for subqueries returning a large number of rows, especially when dealing with non-existence (NOT EXISTS).
- **Fact:** EXISTS only cares if any row is returned, not the content of the rows. So, SELECT 1(or SELECT NULL, SELECT *) is a common and efficient practice within an EXISTS subquery.

4.7.8 Connect By and Start with clauses (Oracle specific - Hierarchical Queries)

- **Description:** CONNECT BY and START WITH clauses are specific to Oracle SQL and are used for hierarchical queries. They allow you to traverse tree-structured data (like organizational charts, bill of materials) where parent-child relationships are stored in the same table.
    - `START WITH`: Defines the root(s) of the hierarchy.
    - `CONNECT BY`: Defines the relationship between parent and child rows (e.g., `PRIOR employee_id = manager_id`).
    - `LEVEL`: Pseudo-column indicating the depth in the hierarchy.
- Sample SQL Query/Code:

```sql
-- Example: Assuming Employees table has ID and Manager_ID
-- List employees and their hierarchy under a specific manager
SELECT
    ID,
    Name,
    manager_id,
    LEVEL AS HierarchyLevel,
    LPAD(' ', 2*(LEVEL-1)) || Name AS FormattedName
FROM Employees
START WITH manager_id IS NULL -- Starts with top-level employees (no manager)
CONNECT BY PRIOR ID = manager_id; -- Connects child (current row) to its parent (prior row's ID)

-- You can also start at a specific employee
-- START WITH ID = 100 -- Start with employee ID 100 and find all their subordinates
```

- **Quiz:** Which database system primarily uses CONNECT BY and START WITH for hierarchical queries?
  a) MySQL
  b) SQL Server

c) PostgreSQL
d) Oracle

- **Tip/Trick:** For non-Oracle databases, hierarchical queries are typically achieved using Recursive CTEs (WITH RECURSIVE).
- **Fact:** This syntax existed in Oracle long before standard SQL introduced recursive CTEs, and it's still widely used in Oracle environments.

4.7.9 Windows Function

Allow you to perform calculations across a set of table rows that are related to the current row, without collapsing those rows into a single output. They provide powerful analytics capabilities such as ranking, running totals, moving averages, and access to preceding/following rows.

`ROW_NUMBER()` function is a **window function** that assigns a **unique, sequential number** to rows in a result set **based on a specified order**. Unlike `RANK()` or `DENSE_RANK()`, it **does not allow ties** — each row always gets a unique number.

`RANK()` window function assigns a **rank** to each row within a partition of the result set. The ranks are assigned in **order of a specified column**, and **ties (duplicate values)** receive the **same rank**, but **gaps** are left in the ranking sequence.

`DENSE_RANK()` is a window function in PostgreSQL that assigns ranks to rows within a result setpartition similar to `RANK()`, but with a key difference: it ranks rows **with ties** without gaps in the ranking sequence.

- It assigns a unique rank to each distinct value in the ordered partition.
- Rows with equal values receive the same rank.
- Unlike `RANK()`, it does not skip rank positions after ties.
- Useful for ranking items where consecutive rank numbering is desired without gaps.

Query: SELECT employee, department, sale_amount,

```
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY
    sale_amount DESC) AS row_num,
  RANK() OVER (PARTITION BY department ORDER BY sale_amount DESC) AS rank,
  DENSE_RANK() OVER (PARTITION BY department ORDER BY sale_amount DESC)
AS dense_rank
FROM sales1;
```

The LAG() function is used to get value from the row that **precedes** the current row.

The LEAD() function is used to get value from a row that **succeeds** the current row.

`LAG()` window function is used to **access data from a previous row** in the same result set — without using self-joins. It's helpful for comparisons, trends, and change detection between rows.

- LAG(price) OVER (ORDER BY date) fetches the price from the previous date row.
- price - LAG(price) OVER (ORDER BY date) calculates the daily price change.
- The first row's one_day_before and daily_change will be NULL since no previous row exists.

	date date	price double precision	one_day_before double precision	daily_change double precision
1	2023-06-01	28.8	[null]	[null]
2	2023-06-02	30	28.8	1.1999999999999993
3	2023-06-03	28.8	30	-1.1999999999999993
4	2023-06-04	27.6	28.8	-1.1999999999999993
5	2023-06-05	28.5	27.6	0.8999999999999986
6	2023-06-06	29.1	28.5	0.6000000000000014
7	2023-06-07	30	29.1	0.8999999999999986

LEAD() window function gives you access to **subsequent row values** (i.e., the *next row's* value) **without using a self-join**. It's commonly used for comparing rows, calculating differences, and identifying trends.

FIRST_VALUE() window function returns the **first value in a sorted window frame**. It's useful for comparing each row to the **first row** in its group or partition.

LAST_VALUE() window function returns the **last value** in a sorted window **frame** — based on the `ORDER BY` clause. It works opposite to `FIRST_VALUE()` and is useful for getting the most recent (or final) value **within a group** or **over time**.

Function	Description
`ROW_NUMBER()`	Assigns a unique sequential number starting at 1 within each partition.
`RANK()`	Assigns rank with gaps for ties (e.g., 1, 2, 2, 4).
`DENSE_RANK()`	Assigns rank without gaps for ties (e.g., 1, 2, 2, 3).
`LAG()`	Accesses data from previous row(s) within the partition.
`LEAD()`	Accesses data from following row(s) within the partition.
`SUM()`, `AVG()`, `MIN()`, `MAX()`	Compute aggregate values over the window (not grouped like GROUP BY).
`FIRST_VALUE()`	Returns the first value in the partition.

`LAST_VALUE()`	Returns the last value in the partition.
`NTILE(n)`	Divides rows into `n` approximately equal groups and assigns group numbers.

4.7.10 Common Table Expression (CTE)

A CTE (Common Table Expression) is a temporary result set that you can reference within a SELECT, INSERT, UPDATE, or DELETE statement. It makes complex SQL queries cleaner, modular, and easier to read — especially when using recursion, window functions, or multi-step logic.

Advantages:

- Query Organization and Readability
- Hierarchical Data Traversal
- Multi-Level Aggregations
- Combining Data from Multiple Tables

–Step 1

```
SELECT EmployeeID, FirstName, LastName, Salary
FROM Employees
WHERE Salary > 40000;
```

–Step 2

```
WITH EmployeeHighIncome AS (
   SELECT EmployeeID, FirstName, LastName, Salary
   FROM Employees
   WHERE Salary > e0000
)
```

–Step 3

```
-- Define a Common Table Expression (CTE)
WITH EmployeeHighIncome AS (
   SELECT EmployeeID, FirstName, LastName, Salary
   FROM Employees
   WHERE Salary > 50000
)
-- Use the CTE to select high-earning employees
SELECT EmployeeID, FirstName, LastName
FROM EmployeeHighIncome;
```

4.7.10 Tips and Tricks (for Joins and Subqueries)

- **Description:** General advice for optimizing and understanding joins and subqueries.
- Sample SQL Query/Code:

```sql
-- Tip: Always use aliases for tables in joins, especially with
many tables
SELECT
    O.OrderID,
    C.CustomerName,
    P.ProductName
FROM
    Orders O
JOIN
    Customers C ON O.CustomerID = C.CustomerID
JOIN
    Products P ON O.ProductID = P.ProductID;

-- Trick: When debugging complex joins, start with a small number
of tables
-- and add them one by one, checking the results at each step.

-- Tip: Understand JOIN order. The database optimizer tries to
find the best order,
-- but sometimes explicit hints or restructuring can help.

-- Trick: LEFT JOIN with IS NULL for "NOT IN" type scenarios
(often more performant)
-- Find customers who have NOT placed any orders:
SELECT C.CustomerName
FROM Customers C
LEFT JOIN Orders O ON C.CustomerID = O.CustomerID
WHERE O.OrderID IS NULL; -- Where there's no match in the Orders
table
```

- **Quiz:** What is a common and often more performant alternative to NOT IN when checking for the absence of related records?
  a) HAVING COUNT(*) = 0
  b) RIGHT JOIN
  c) FULL OUTER JOIN
  d) LEFT JOIN with WHERE ... IS NULL
- **Tip/Trick:** Use EXPLAIN (or EXPLAIN PLAN, SHOW PLAN) commands in your database to see how the query optimizer plans to execute your query. This helps identify performance bottlenecks.