SQL Training Material

Presentation Agenda

- Introduction to Database Management Systems (DBMS)
- Characteristics & Models of DBMS
- Data Integrity Concepts
- Structured Query Language (SQL) Fundamentals
- SQL Data Manipulation Language (DML)
- Database Objects: Indexes, Synonyms, Sequences, Views
- Data Query Language (DQL): SELECT, DISTINCT, Operators
- Aggregate Functions & Grouping
- SQL Single-Row Functions

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.
- **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.
- **Code Example**: Simple table structure for Products and Orders, including primary and foreign keys.
  - `CREATE TABLE Products (ProductID INT PRIMARY KEY, ProductName VARCHAR(255), Price DECIMAL(10, 2));`
  - `CREATE TABLE Orders (OrderID INT PRIMARY KEY, ProductID 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.

1.5 Data Integrity

- **Description**: Data integrity refers to the accuracy, consistency, and reliability of data over its entire lifecycle. It ensures 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).
- **Code Example**: Enforcing different types of integrity.
  - Entity Integrity (Primary Key): `CREATE TABLE Users (UserID INT PRIMARY KEY, UserName VARCHAR(100));`
  - Referential Integrity (Foreign Key): `FOREIGN KEY (UserID) REFERENCES Users(UserID);`
  - Domain Integrity (CHECK constraint): `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,

2.1 Interacting SQL with CLI Tools

- **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.
- **Code Example**: Basic CLI commands.
  - Oracle specific: `SQL> SELECT SYSDATE FROM DUAL;`
  - General: `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.
- **Code Example**: 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).
  - Whitespace (spaces, tabs, newlines) is generally ignored, allowing for formatting to improve readability.
  - Comments (e.g., `--` for single-line, `/* */` for multi-line) are used to explain code and are ignored by the database.
- **Code Example**: `SELECT employee_id, first_name FROM Employees;`
- **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.

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`).
- **Code Example**: Commands for DDL, DML, DQL, DCL, TCL (e.g., `CREATE TABLE`, `INSERT INTO`, `SELECT *`, `GRANT SELECT`, `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.
- **Common Types**:
  - 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`
- **Code Example**: `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

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).
- **Code Example**: Good vs. Bad table names.
  - Good: `CREATE TABLE customer_feedback (...);`
  - Bad (reserved): `CREATE TABLE ORDER (...);`
  - Bad (spaces): `CREATE TABLE My Table (...);`
- **Quiz**: Which character is generally recommended to replace spaces in table names for better readability and compatibility?
  - a) Hyphen (-)

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.
- **Code Example**: `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));`
- **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

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.
- **Code Example**: `CREATE TABLE Customers (...); ALTER TABLE Customers ADD Email VARCHAR(100); DROP TABLE Customers; TRUNCATE TABLE Customers;`
- **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

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. Useful for storing intermediate results.
- **Code Example**: `CREATE TABLE PermanentData (ID INT); CREATE TEMPORARY TABLE SessionData (ID INT);` (syntax varies by DB)
- **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.

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.
- **Code Example**: Inserting data into a `Products` table.
  - Insert all columns: `INSERT INTO Products (ProductID, ProductName, Price) VALUES (1, 'Laptop', 1200.00);`
  - Insert specific columns: `INSERT INTO Products (ProductID, ProductName) VALUES (2, 'Mouse');`
  - Insert multiple rows (modern DBs): `INSERT INTO Products (...) 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

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 (DANGER!).
- **Code Example**: Deleting product data.
  - Specific product by ID: `DELETE FROM Products WHERE ProductID = 1;`
  - All products with price < 100: `DELETE FROM Products WHERE Price < 100;`
  - **DANGER ZONE**: Delete all rows: `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.

3.3 Updating Rows in a Table

- **Description**: The `UPDATE` statement is used to modify existing data within a table. The `SET` clause specifies the column(s) to modify and their new values, and the `WHERE` clause determines which rows will be affected.
- **Code Example**: Updating product data.
  - Update specific product price: `UPDATE Products SET Price = 1250.00 WHERE ProductID = 1;`
  - Update all products with certain price: `UPDATE Products SET ProductName = 'Discounted Item' WHERE Price < 100;`
  - Update multiple columns: `UPDATE Products SET ProductName = 'Gaming Laptop', Price = 1500.00 WHERE ProductID = 1;`
  - **DANGER ZONE**: Update all rows: `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

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, which can be very slow for large tables.
- **Code Example**: Creating and dropping indexes.
  - Create an index: `CREATE INDEX idx_product_name ON Products (ProductName);`
  - Create a unique index: `CREATE UNIQUE INDEX idx_email_unique ON Users (Email);`
  - 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 `DELETE` operations.

3.4.2 Synonym

- **Description**: A synonym is an alternative name for a table, view, sequence, or other database object. It provides an alias, useful for simplifying long object names, providing abstraction, or granting access to objects in another schema.
- **Code Example**: Creating and dropping synonyms.
  - Create synonym (Oracle): `CREATE SYNONYM my_products FOR scott.products;`
  - Query using synonym: `SELECT * FROM my_products;`
  - Drop 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 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 each new record gets a unique identifier.
- **Code Example**: Creating, using, and dropping sequences.
  - Create sequence: `CREATE SEQUENCE product_id_seq START WITH 1 INCREMENT BY 1 NOCACHE;`
  - Use sequence to insert: `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 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) are simpler, sequences offer more control (e.g., starting value, increment, caching).

3.4.4 Views

- **Description**: A view is a virtual table based on the result-set of an SQL query. It 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.
- **Code Example**: Creating and using views.
  - Create a view: `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: `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

4.1 SELECT Statement

- **Description**: The `SELECT` statement is the most fundamental SQL command. It retrieves data from one or more tables, specifying columns, tables, and conditions.
- **Code Example**: Basic `SELECT` queries.
  - Select all columns: `SELECT * FROM Employees;`
  - Select specific columns: `SELECT Name, Salary FROM Employees;`
  - Select with an alias: `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.
- **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.
- **Code Example**: Using `DISTINCT`.
  - Unique department names: `SELECT DISTINCT department FROM Employees;`
  - Unique combinations: `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, `DISTINCT` returns 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.

4.3.1 SQL Operators

- **Description**: Operators are used in SQL statements to perform comparisons, arithmetic calculations, or logical operations.
- **Types of Operators**:
  - Comparison Operators: `=`, `!=` (or `<>`), `>`, `<`, `>=`, `<=`, `BETWEEN`, `LIKE`, `IN`, `IS NULL`
  - Arithmetic Operators: `+`, `-`, `*`, `/`, `%` (modulo)
  - Logical Operators: `AND`, `OR`, `NOT`
- **Code Example**: Using various operators.
  - Comparison: `SELECT Name, Age FROM Employees WHERE Age > 30;`
  - Arithmetic: `SELECT Name, Salary, Salary * 1.05 AS NewSalary FROM Employees;`
  - Logical: `SELECT Name, department, Salary FROM Employees WHERE department = 'HR' AND Salary > 50000;`
- **Quiz**: Which logical operator returns true if at least one of its conditions is true?
  - a) AND
  - b) NOT
  - c) 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.
- **Code Example**: Sorting employee data.
  - Order by name alphabetically: `SELECT Name, Department FROM Employees ORDER BY Name ASC;`
  - Order by salary, highest first: `SELECT Name, Salary FROM Employees ORDER BY Salary DESC;`
  - Order by department (ASC) then by salary (DESC): `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`) though using names is generally clearer.

4.3.3 Tips and Tricks

- **Tip**: Use aliases for table names for shorter, cleaner joins: `SELECT E.Name, D.DepartmentName FROM Employees AS E JOIN Departments AS D ON E.dept_id = D.DepartmentID;`
- **Trick**: Use `LIKE` with wildcards (`%` for zero or more, `_` for single character) for pattern matching.
  - Starts with 'L': `SELECT ProductName FROM Products WHERE ProductName LIKE 'L%';`
  - Ends with 'top': `SELECT ProductName FROM Products WHERE ProductName LIKE '%top';`
  - 5 letters, 2nd to last 's': `SELECT ProductName FROM Products WHERE ProductName LIKE '_ouse';`
- **Tip**: Use `IN` for multiple `OR` conditions: `SELECT Name FROM Employees WHERE Department IN ('HR', 'Sales', 'Marketing');`
- **Quiz**: What wildcard character typically represents zero or more characters in the `LIKE` operator?
  - 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

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.
- **Code Example**: Counting and averaging grouped data.
  - Count employees in each department: `SELECT department, COUNT(*) AS NumberOfEmployees FROM Employees GROUP BY department;`
  - Calculate average salary per department: `SELECT department, AVG(Salary) AS AverageSalary FROM Employees GROUP BY department;`
  - Count employees per department and manager: `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.

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.
- **Code Example**: Filtering groups with `HAVING`.
  - Find departments with more than 5 employees: `SELECT department, COUNT(*) AS NumberOfEmployees FROM Employees GROUP BY department HAVING COUNT(*) > 5;`
  - Find departments where 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) HAVING uses aggregate functions, WHERE does not.
- **Tip/Trick**: If you can filter rows before grouping using `WHERE`, it's usually more efficient. 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.
- **Code Example**: 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 Example**:
  - | 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?

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.
- **Code Example**: 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 Example**: Will include (department, manager_id) totals, (department, NULL) totals (subtotal for each department), (NULL, manager_id) totals (subtotal for each manager), and (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

4.4.5 Tips & Tricks

- **Tip**: Use `COUNT(DISTINCT column)` to count unique values in a group: `SELECT department, COUNT(DISTINCT manager_id) AS UniqueManagers FROM Employees GROUP BY department;`
- **Trick**: Be careful with `NULL`s in aggregate functions: `COUNT(*)` counts all rows; `COUNT(column_name)` counts non-NULL values.
- **Tip**: Filter with `WHERE` before `GROUP BY` for efficiency: `SELECT department, AVG(Salary) FROM Employees WHERE Salary > 40000 GROUP BY department HAVING COUNT(*) > 2;`
- **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

4.5.1 Character Functions

- **Description**: Functions that operate on string (character) data, performing tasks like converting case, extracting substrings, or padding.
- **Common Functions**:
  - `UPPER()`: Convert to uppercase.
  - `LOWER()`: Convert to lowercase.
  - `LENGTH()` (or `LEN()`): Get length of string.
  - `SUBSTR()` (or `SUBSTRING()`): Extract a part of a string.
  - `CONCAT()`: Combine strings (syntax varies, e.g., `||`).
- **Code Example**: `SELECT UPPER(Name), SUBSTR(Email, 1, INSTR(Email, '@')-1), CONCAT(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()

4.5.2 Number Functions

- **Description**: Functions that perform mathematical operations on numeric data.
- **Common Functions**:
  - `ROUND()`: Round a number to a specified number of decimal places.
  - `TRUNC()` (or `TRUNCATE()`): Truncate a number to a specified number of decimal places.
  - `ABS()`: Absolute value.
  - `CEIL()` (or `CEILING()`): Round up to the nearest integer.
  - `FLOOR()`: Round down to the nearest integer.
  - `MOD()` (or `%`): Modulo (remainder of a division).
- **Code Example**: `SELECT ROUND(123.456, 2), ABS(-10), CEIL(12.1), FLOOR(12.9), MOD(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` (rounds to nearest) and `TRUNC`

4.5.3 Data Conversion Functions

- **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.
- **Common Functions**:
  - `TO_CHAR()` (Oracle/PostgreSQL): Convert number or date to string.
  - `TO_NUMBER()` (Oracle/PostgreSQL): Convert string to number.
  - `TO_DATE()` (Oracle/PostgreSQL): Convert string to date.
  - `CAST()` (SQL Standard): Convert between types.
  - `CONVERT()` (SQL Server): Convert between types.
- **Code Example**: `SELECT TO_CHAR(12345, 'FM99,999'), CAST('123' AS INT) + 10, CONVERT(VARCHAR(10), GETDATE(), 120);`
- **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()

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.
- **Common Date/Time Format Elements (Examples)**:
  - `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)
- **Code Example**: `SELECT TO_CHAR(CURRENT_DATE, 'Day, Month DD, YYYY') AS FormattedDate; SELECT TO_DATE('26-OCT-2023 14:30:00', 'DD-MON-YYYY HH24:MI:SS');`
- **Quiz**: In a date format string (e.g., for `TO_CHAR`), what does `YYYY` typically represent?

4.5.5 Date Functions

- **Description**: Functions specifically designed to perform calculations and extract components from date and time values.
- **Common Functions**:
  - `CURRENT_DATE` / `GETDATE()` / `NOW()`: Get current date/time.
  - `EXTRACT()`: Extract part of a date/time (e.g., YEAR, MONTH, DAY).
  - `DATEDIFF()`: Calculate difference between dates (e.g., in years, days).
  - `DATE_ADD()` / `DATE_SUB()`: Add/subtract intervals.
- **Code Example**: `SELECT EXTRACT(YEAR FROM hire_date), DATEDIFF(year, hire_date, GETDATE()), DATE_ADD(CURRENT_DATE, INTERVAL 7 DAY);`
- **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)

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`.
- **Common Functions**:
  - `NVL()` (Oracle) / `ISNULL()` (SQL Server) / `COALESCE()` (SQL Standard): Replace `NULL` values with a specified value.
  - `CASE` Statement: Conditional logic (similar to an if-else statement).
  - `DECODE()` (Oracle specific): Similar to a simple `CASE` statement.
- **Code Example**: `SELECT COALESCE(commission, 0) AS ActualCommission FROM Sales; SELECT Name, Salary, CASE WHEN Salary >= 100000 THEN 'High Earner' WHEN Salary >= 50000 THEN 'Mid-Range' ELSE 'Junior' END AS SalaryBracket 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()

4.5.7 Tips & Tricks

- **Tip**: Chain functions for complex transformations: `SELECT UPPER(SUBSTR(ProductName, 1, 3)) AS AbbrName FROM Products;`
- **Trick**: Use `NULLIF` to avoid division by zero: `SELECT TotalSales / NULLIF(NumberOfOrders, 0) AS AvgSalePerOrder FROM SalesData;`
- **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

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.
- **Code Example**: Starting, operating within, committing, and rolling back transactions.
  - Start transaction: `BEGIN TRANSACTION;` (or `START TRANSACTION` for MySQL)
  - Perform DML: `UPDATE Accounts SET Balance = Balance - 100 WHERE AccountID = 1;`
  - Commit: `COMMIT;`
  - Rollback: `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

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.
- **Code Example**: Using `COMMIT` after a batch update.
  - `BEGIN TRANSACTION; INSERT INTO LogTable (...); UPDATE Orders (...); DELETE FROM OldData (...); COMMIT;`
- **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**: Undoes 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.
- **Code Example**: Using `SAVEPOINT` and `ROLLBACK TO`.
  - `BEGIN TRANSACTION; INSERT INTO TempTable (1, 'Initial'); SAVEPOINT sp1; INSERT INTO TempTable (2, 'Second'); SAVEPOINT sp2; INSERT INTO TempTable (3, 'Third - Problem here'); ROLLBACK TO sp2; INSERT INTO TempTable (4, 'Fourth - Fixed'); COMMIT;`
- **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

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 (`=`).
- **Code Example**: `INNER JOIN` syntax.
  - Explicit: `SELECT E.Name, D.DepartmentName FROM Employees E INNER JOIN Departments D ON E.dept_id = D.DepartmentID;`
  - Implicit (older syntax, not recommended): `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.

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.
- **Code Example**: Using `LEFT`, `RIGHT`, and `FULL OUTER` joins.
  - LEFT JOIN: `SELECT E.Name, D.DepartmentName FROM Employees E LEFT JOIN Departments D ON E.dept_id = D.DepartmentID;`
  - RIGHT JOIN: `SELECT E.Name, D.DepartmentName FROM Employees E RIGHT JOIN Departments D ON E.dept_id = D.DepartmentID;`
  - FULL OUTER JOIN: `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

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, or finding duplicate records.
- **Code Example**: Finding employees and their managers, and employees earning more than their manager.
  - Employees and their managers: `SELECT E.Name AS EmployeeName, M.Name AS ManagerName FROM Employees E JOIN Employees M ON E.manager_id = M.ID;`
  - Employees earning 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

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.
- **Code Example**: Subqueries in `WHERE` and `FROM` clauses.
  - Subquery in WHERE clause (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.

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).
- **Code Example**: Finding employees earning more than the average salary.
  - More than average: `SELECT Name, Salary FROM Employees WHERE Salary > (SELECT AVG(Salary) FROM Employees);`
  - With 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 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.
- **Code Example**: Finding employees whose salary is higher than their OWN department's average.
  - `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);`
  - 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.

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.
- **Code Example**: Finding departments with/without employees, and customers who ordered a product.
  - Departments with at least one employee: `SELECT DepartmentName FROM Departments D WHERE EXISTS (SELECT 1 FROM Employees E WHERE E.dept_id = D.DepartmentID);`
  - Departments with NO employees: `SELECT DepartmentName FROM Departments D WHERE NOT EXISTS (SELECT 1 FROM Employees E WHERE E.dept_id = D.DepartmentID);`
  - Employees who 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.

4.7.8 Connect By and Start with clauses

- **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.
- **Code Example**: Listing 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 CONNECT BY PRIOR ID = manager_id;`
- **Quiz**: Which database system primarily uses `CONNECT BY` and `START WITH` for hierarchical queries?
  - a) MySQL
  - b) SQL Server
  - c) PostgreSQL
  - d) Oracle

4.7.9 Tips & Tricks

- **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) to 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;`
- **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