Introduction to DataBase

Reading Material

PW SKILLS

Topics Covered

1. Introduction to Database & Types
   - Definition of Database
   - Types of Databases (Relational, Object-Oriented, NoSQL, etc.)
   - Database Management System (DBMS)
   - Database Schema and Instances

2. DBMS vs RDBMS
   - Definitions and Basic Concepts
   - Key Differences
   - Advantages and Disadvantages
   - Popular DBMS and RDBMS Systems

3. Introduction to SQL
   - What is SQL?
   - History and Evolution of SQL
   - SQL Standards
   - SQL in Different Database Systems

4. SQL Command Categories
   - Data Definition Language (DDL)
   - Data Manipulation Language (DML)
   - Data Control Language (DCL)
   - Transaction Control Language (TCL)
   - Data Query Language (DQL)

5. CRUD Operations
   - CREATE: Inserting Data
   - READ: Querying Data
   - UPDATE: Modifying Existing Data
   - DELETE: Removing Data

6. DataTypes & Operators and its Types
   - Numeric Data Types
   - Character and String Data Types
   - Date and Time Data Types
   - Boolean Data Type
   - Arithmetic Operators
   - Comparison Operators
   - Logical Operators

7. Constraints in SQL
   - Primary Key
   - Foreign Key
   - Unique
   - Not Null
   - Check
   - Default

8.Set

9. SQL JOINs
- INNER JOIN
- LEFT (OUTER) JOIN
- RIGHT (OUTER) JOIN
- FULL (OUTER) JOIN
- CROSS JOIN
- SELF JOIN

10.Subquery
11. Clause
12.Functions
13.Window Functions
14.Normalization
15.ACID Properties

Introduction to Databases & its Types

What is Data? :
Data refers to raw facts, figures, or information about things under examination. It can be:
- Quantitative (numbers, measurements)
- Qualitative (descriptions, opinions)
- Structured (organized in a predefined format)
- Unstructured (no predefined format)

Examples of data include:
- Personal information (name, age, address)
- Business data (sales figures, inventory levels)
- Scientific measurements (temperature readings, chemical compositions)
- Digital media (images, videos, audio files)

What is a Database ?

A database is a structured collection of data organized for efficient storage, retrieval, and manipulation. Key characteristics include:

- **Organized structure:** Data is arranged in tables, documents, or other formats
- **Relationships:** Connections between different data elements
- **Accessibility:** Multiple users can access and manipulate data simultaneously
- **Data integrity:** Mechanisms to ensure accuracy and consistency of data
- **Scalability:** Ability to grow as data volume increases
- **Security:** Controls to protect data from unauthorized access

Purposes of databases:

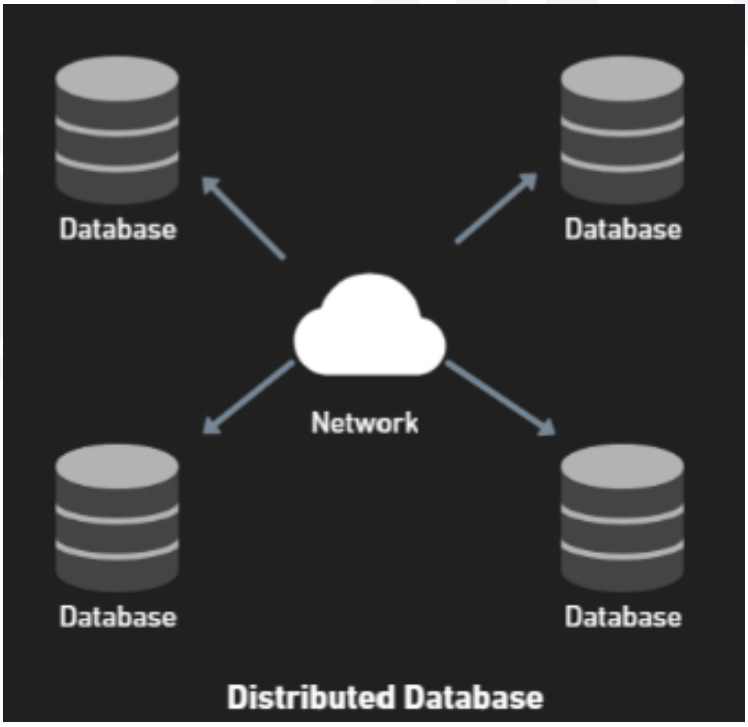
- Centralized data management
- Efficient data retrieval and analysis
- Data consistency and accuracy
- Support for complex queries and reporting
- Backup and recovery capabilities

Types of Databases:

- Distributed
- Relational
- Object Oriented
- Hierarchical
- Network
- NoSQL

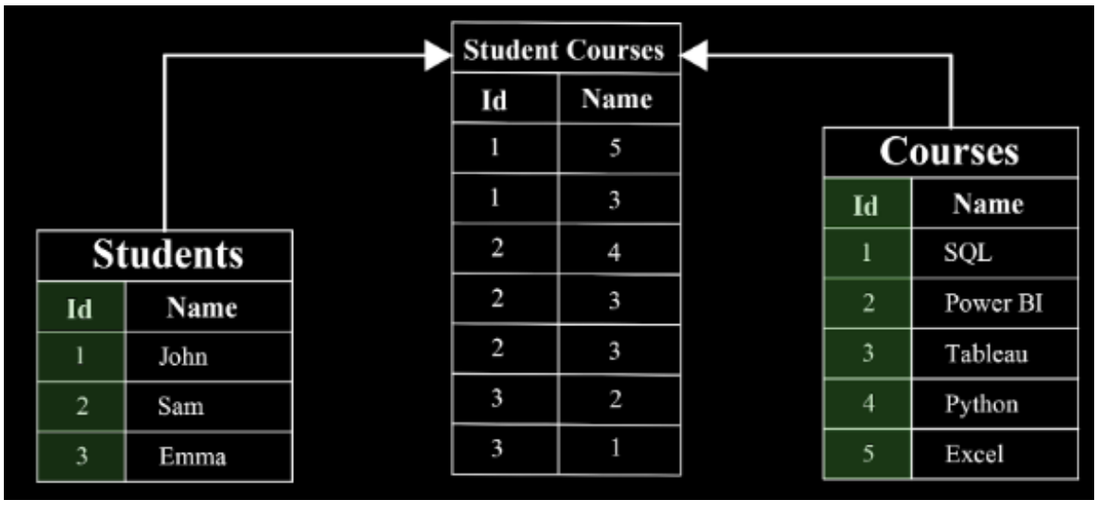
a) Distributed Databases:

- Data is stored across multiple physical locations
- Connected through a computer network
- Appears to users as a single logical database
- **Advantages:** improved performance, reliability, and availability
- **Challenges:** complex synchronization and consistency maintenance



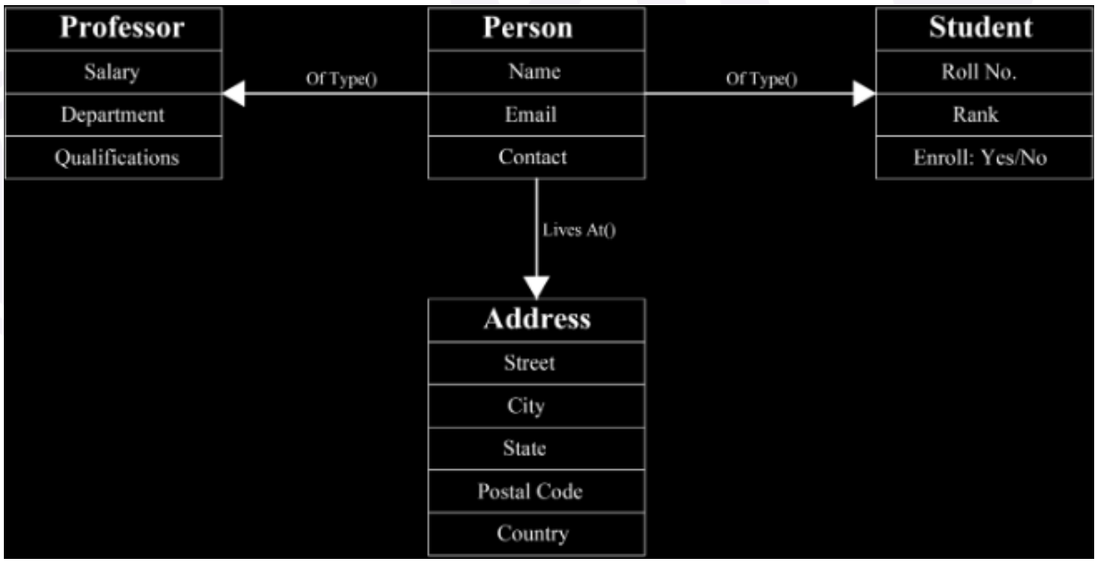
b) Relational Databases:

- Based on the relational model proposed by E.F. Codd
- Data organized in tables (relations) with rows and columns
- Uses SQL (Structured Query Language) for querying and management
- Emphasizes data integrity and ACID properties (Atomicity, Consistency, Isolation, Durability)
- **Examples:** MySQL, PostgreSQL, Oracle, Microsoft SQL Server
- **Best for:** structured data with complex relationships



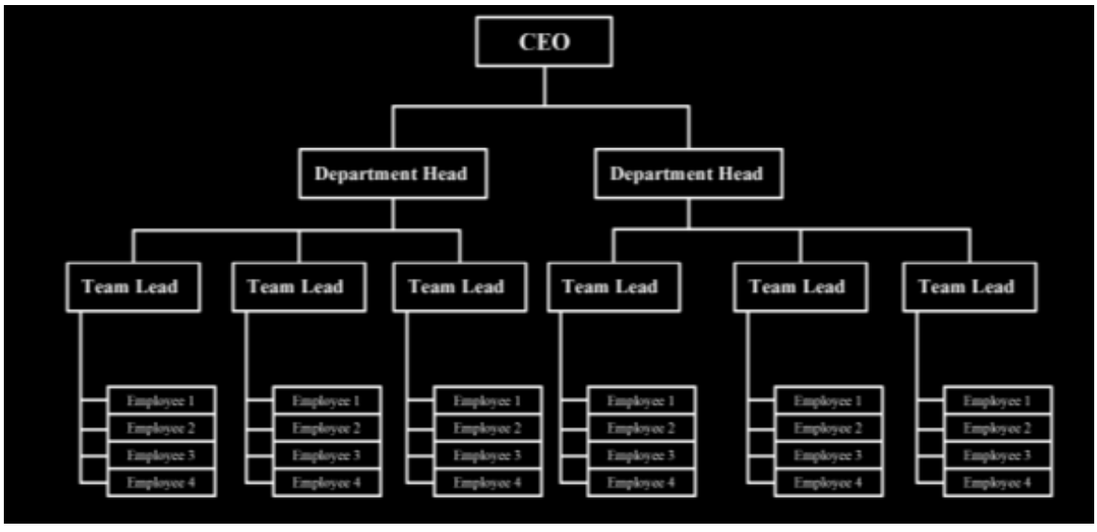
c) Object-Oriented Databases:

- Stores data as objects, similar to object-oriented programming
- Supports complex data types and relationships
- Allows direct representation of real-world entities
- Advantages: natural modeling of complex systems, support for multimedia
- **Examples:** ObjectDB, Versant
- **Best for:** complex systems with diverse data types



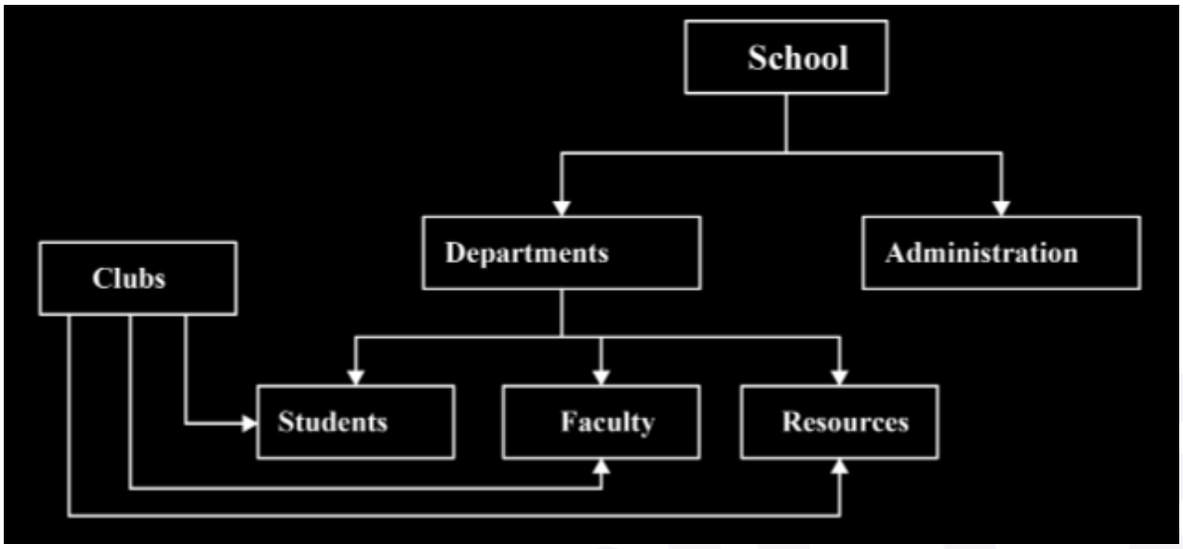
d) Hierarchical Databases:

- Organizes data in a tree-like structure
- Parent-child relationships between data elements
- Efficient for one-to-many relationships
- Limited flexibility for complex relationships
- **Example**: IBM's Information Management System (IMS)
- **Best for**: systems with clear hierarchical structures (e.g., file systems)



e) Network Databases:

- Extends hierarchical model to allow many-to-many relationships
- Uses sets to represent relationships between data
- More flexible than hierarchical, but more complex
- **Example:** Integrated Data Store (IDS)
- **Best for:** complex relationships that don't fit well in relational model



f) NoSQL Databases:

- "Not Only SQL" - designed for distributed data stores
- Emphasizes scalability and flexibility over ACID guarantees

Various types:

1. **Document stores (e.g., MongoDB):** store data in JSON-like documents
2. **Key-value stores (e.g., Redis):** simple key-value pairs
3. **Wide-column stores (e.g., Cassandra):** tables with rows and dynamic columns
4. **Graph databases (e.g., Neo4j):** optimized for interconnected data

- **Advantages:** high scalability, flexible schema, better performance for certain use cases
- **Best for:** big data, real-time web applications, content management systems

Mongo DB falls in this category

DBMS Vs RDBMS

DBMS – Database Management System

A group of programmes known as a database management system (DBMS) give users access to databases and the ability to alter, report, and depict data. Controlling access to the database is aided by it as well. Since database management systems are not a new idea, they were initially used in the 1960s.

The Integrated Data Store (IDS) by Charles Bachman is regarded as the first DBMS ever created. As database technology advanced throughout time, so did demand for and expectations of their features.

A software programme called a database management system (DBMS) is made to manage and arrange data in a systematic way. Users may manage the security and access settings for a database as well as create, change, and query databases.

Features of DBMS

1. **Data Modelling:** A DBMS offers tools for building and altering data models, which specify the organization and connections of the data in a database.

2. **Data storage and retrieval:** A DBMS can offer a variety of techniques for finding and querying the data. It is responsible for storing and retrieving data from the database.

3. **Concurrency control:** A DBMS offers tools for managing simultaneous access to the database, ensuring that different users may access the information without interfering with one another.

4. **Data security and integrity:** A database management system (DBMS) offers tools for implementing security and integrity restrictions, such as limitations on the data's values and access controls that limit who may access the data.

5. **Backup and recovery:** A DBMS provides mechanisms for backing up and recovering the data in the event of a system failure.

6. Relational Database Management Systems (RDBMS) and Non-Relational Database Management Systems (NoSQL or Nosql) are the two categories into which DBMS may be divided.

7. **RDBMS:** Data is arranged into tables, each of which has a specific number of rows and columns. Through main and foreign keys, the data are connected to one another.

8. **NoSQL:** Data is arranged as columns, documents, graphs, or key-value pairs. These are made to handle high-performance, large-scale situations.

A database is a group of connected data that facilitates effective data retrieval, insertion, and deletion. The data is organized in the database using tables, views, schemas, reports, etc. For instance, a university database organizes information on students, professors, administrative staff, etc., enabling effective data retrieval, insertion, and deletion.

Advantages of DBMS

1. A range of methods for storing and retrieving data are provided by DBMS.
2. The demands of several applications using the same data are efficiently handled by DBMS.
3. Uniform data administration practices.
4. Application programmers are never exposed to the specifics of data storage and representation.
5. A DBMS makes effective use of a number of strong functions to store and retrieve data.
6. Provides security and data integrity.
7. Integrity restrictions are implied by the DBMS to get a high level of security against unauthorized access to the data.
8. Only one user can access the same data at once due to the way a DBMS schedules concurrent access to the data.
9. Shortened time for application development

Disadvantages

1. Because DBMS hardware and software are relatively expensive, your organization's budget grows.
2. Since most database management systems are sophisticated, users must get training before using them.
3. In certain businesses, all data is combined into a single database that can be harmed by an electrical failure or a corrupted database on the storage medium.
4. Multiple people using the same programme simultaneously might occasionally result in data loss.
5. DBMS is not capable of complex computations.

RDBMS

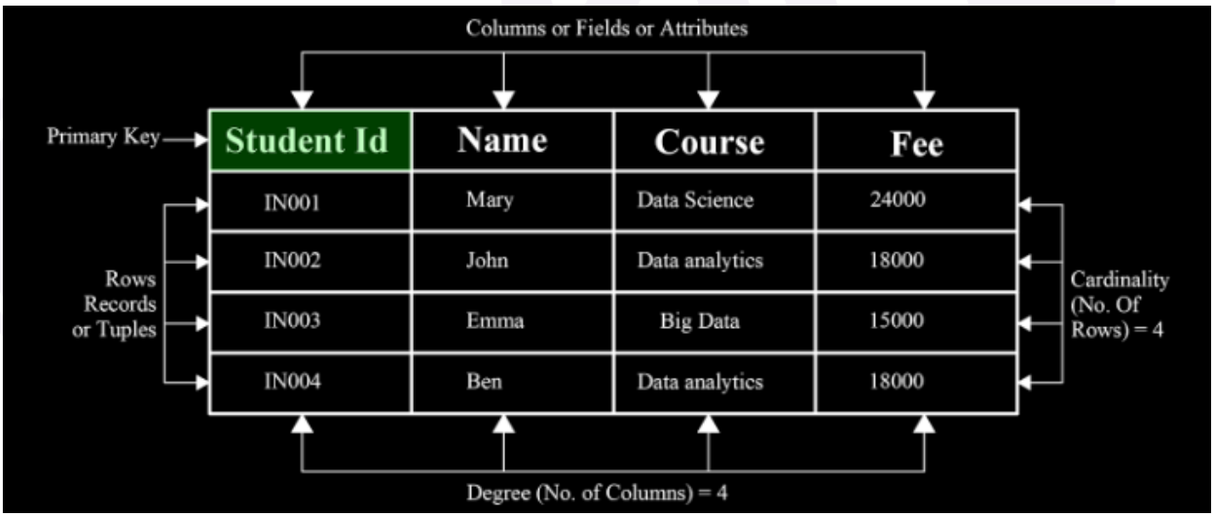
In a relational database, data is kept in one or more tables (or "relations") of columns and rows, making it simple to see and comprehend how various data structures connect to one another. Data is organized in relational databases according to predetermined relationships. Relationships are logical connections that have been made between several tables as a result of their interaction.

Relational Database Model

The relational database paradigm, created by E.F. Codd at IBM in the 1970s, enables any table to be associated with another table using a common attribute. Codd suggested switching to a data model, where data is stored, retrieved, and connected in tables without reorganizing the tables that hold them, as opposed to utilizing hierarchical structures to organize data.

A data type is specified by attributes (columns), and the value for that particular data type is contained in each record (or row). Each row may be used to establish a relationship between multiple tables using a foreign key, which is a reference to a primary key of an existing table. All tables in a relational database have an attribute known as the primary key, which is a unique identifier of a row.

The relational model represents the database as a collection of relations. In relational model terminology, a row is called a tuple, a column header is called an attribute and the table is called a relation. The data type describing the type of values that can appear in each column is called a domain.



Structure of relational database

In the given table the SID, SNAME, CLASS, AGE are attributes.

What is Table/Relation?

A relational database stores data in the form of relations for everything. Tables are used in the RDBMS database to hold data. A table is a collection of connected data elements that stores data in rows and columns. Each table represents a specific real-world item, such as a person, location, or event, about which data is being gathered. The logical view of the database is the organized collection of data included in a relational table.

Properties of a Relation

- Each relation has a distinct name that may be used to locate it in the database.
- There are no duplicate tuples in the relationship.
- A relation's tuples are not in any particular sequence.
- Each cell of a connection holds precisely one value since all characteristics in a relation are atomic.

What is a Row or Record or Tuple?

A record or tuple is another name for a database row. It includes all of the precise details for each table entry. In the table, it is a horizontal element. For instance, the table above has 5 records.

Properties of a Row

- In all of their entries, no two tuples are identical to one another.
- The format and amount of items are the same for all tuples in the relation.
- The tuple's order is not important. They are recognised by their ideas, not by where they are located.

What is Column/Attribute?
In a table, a column is a vertical object that holds all the data pertaining to a single field. For instance, the column "name" in the table above provides all the details on a student's name.

Properties of a Column

- A relation's attributes must all have names.
- The attributes are allowed to have null values.
- If no additional value is supplied for an attribute, default values can be specified and added automatically.
- The primary key refers to the characteristics that specifically identify each tuple in a relation.

What are Data Items/Cells?

The individual data item is the smallest type of data in the table. It is kept at the point where tuples and attributes overlap.

Properties of a Cells

- Items in data are atomic.
- An attribute's data elements ought to come from the same domain.

What is Degree?

The degree of the table refers to the total number of characteristics in a relation.

What is Cardinality?

The cardinality of a table is the total number of tuples present in a relation at any one time. An empty table is a relation whose cardinality is 0.

What is a NULL Value?

The table's NULL value indicates that the field was left empty when the record was created. It differs from a value that is blank or a field with white space.

Here's an example of what our Books table might look like:

ISBN (Primary Key)	Title	Author	Price	Quantity
978-0061120084	To Kill a Mockingbird	Harper Lee	12.99	50
978-0141439518	Pride and Prejudice	Jane Austen	9.99	30
978-0316769174	The Catcher in the Rye	J.D. Salinger	11.99	25
978-0743273565	The Great Gatsby	F. Scott Fitz.	14.99	40

Introduction to SQL

SQL stands for Structured Query Language. It's the standard language for interacting with relational databases. Think of SQL as the way you communicate with your database - it's how you ask for information, add new data, or make changes.

SQL vs. MySQL:

- SQL is the language itself. It's like English - a set of rules and vocabulary for expressing database operations.
- MySQL is a specific database management system that uses SQL. It's like a particular dialect of English, with some unique features and ways of doing things.

Other systems like PostgreSQL, Oracle, and Microsoft SQL Server also use SQL, but each may have its own slight variations or additional features.

SQL Commands

SQL commands fall into several categories:

a) Data Definition Language (DDL):
These commands are used to define and manage the structure of your database.

- **CREATE:** Used to create new database objects (like tables)
- **ALTER:** Used to modify existing database objects
- **DROP:** Used to delete database objects
- **TRUNCATE:** Used to remove all data from a table
- **RENAME:** Used to change the name of database objects

Example:

```sql
CREATE TABLE Books (
    ISBN VARCHAR(13) PRIMARY KEY,
    Title VARCHAR(100),
    Author VARCHAR(50),
    Price DECIMAL(5,2),
    Quantity INT
);
```

This command creates our Books table.

b) Data Manipulation Language (DML):
These commands are used to manipulate the data within the database.

- **INSERT:** Used to add new data
- **UPDATE:** Used to modify existing data
- **DELETE:** Used to remove data

Example:
```sql
```

```sql
INSERT INTO Books (ISBN, Title, Author, Price, Quantity)
VALUES ('978-0061120084', 'To Kill a Mockingbird', 'Harper Lee', 12.99, 50);
```

This command adds a new book to our Books table.

c) Data Query Language (DQL):
This category really only includes one command, but it's the most commonly used:

- SELECT: Used to retrieve data from the database

Example:

```sql
SELECT Title, Author FROM Books WHERE Price < 15.00;
```

This command retrieves the titles and authors of all books priced under $15.

d) Transaction Control Language (TCL):
These commands are used to manage transactions (groups of database operations).

- **COMMIT:** Saves transaction changes permanently
- **ROLLBACK:** Undoes transaction changes
- **SAVEPOINT:** Creates points within a transaction to roll back to

e) Data Control Language (DCL):
These commands are used to control access to the database.

- **GRANT:** Gives user access privileges
- **REVOKE:** Removes user access privileges

CRUD Operations

CRUD stands for Create, Read, Update, and Delete. These are the four basic operations you can perform on data in a database.

````
1. Create (INSERT):
```sql
INSERT INTO Books (ISBN, Title, Author, Price, Quantity)
VALUES ('978-0743273565', 'The Great Gatsby', 'F. Scott Fitzgerald', 14.99, 40);
```

2. Read (SELECT):
```sql
SELECT * FROM Books WHERE Author = 'Jane Austen';
```

3. Update (UPDATE):
```sql
UPDATE Books SET Price = 13.99 WHERE ISBN = '978-0061120084';
```

4. Delete (DELETE):
```sql
DELETE FROM Books WHERE Quantity = 0;
```
````

These operations form the foundation of most database interactions. As you become more comfortable with databases, you'll find yourself using these commands frequently.

SQL Data Types and Operators

1. SQL Data Types

SQL provides various data types to store different kinds of information. Think of these as containers specifically designed for certain types of data.

a) Numeric Data Types:

- **INT:** For whole numbers. Example: 42 (number of books in stock)
- **DECIMAL or NUMERIC:** For precise decimal numbers. Example: 12.99 (book price)
- **FLOAT:** For approximate decimal numbers. Example: 3.14159 (when precise decimals aren't crucial)

b) Date and Time Data Types:

- **DATE:** Stores date. Example: '2023-08-15' (publication date)
- **TIME:** Stores time. Example: '14:30:00' (store closing time)
- **DATETIME:** Stores both date and time. Example: '2023-08-15 14:30:00' (order timestamp)

c) String Data Types:

- **CHAR:** Fixed-length string. Example: 'Y' or 'N' (for yes/no flags)
- **VARCHAR:** Variable-length string. Example: 'To Kill a Mockingbird' (book title)
- **TEXT:** For longer strings. Example: Book descriptions or reviews

d) Special Data Types:

- **BOOLEAN:** True or false values. Example: Whether a book is in stock
- **ENUM:** A string object with a value chosen from a predefined list. Example: Book categories ('Fiction', 'Non-Fiction', 'Biography')

Example of using these in our bookstore database:

```sql
CREATE TABLE Books (
    ISBN VARCHAR(13) PRIMARY KEY,
    Title VARCHAR(100),
    Author VARCHAR(50),
    Price DECIMAL(5,2),
    PublicationDate DATE,
    InStock BOOLEAN,
    Category ENUM('Fiction', 'Non-Fiction', 'Biography', 'Science', 'History')
);
```

2. SQL Operators

Operators in SQL help you perform operations on data. They're like the basic math symbols you learned in school, but for databases.

a) Arithmetic Operators:

- **Addition (+):** SELECT Price + 5 AS IncreasedPrice FROM Books;
- **Subtraction (-):** SELECT Price - 2 AS DiscountedPrice FROM Books;
- **Multiplication (*):** SELECT Price * 0.9 AS SalePrice FROM Books;

b) Comparison Operators:

- **Equal (=):** SELECT * FROM Books WHERE Author = 'Jane Austen';
- **Not Equal (<> or !=):** SELECT * FROM Books WHERE Category <> 'Fiction';
- **Greater Than (>):** SELECT * FROM Books WHERE Price > 15;
- **Less Than (<):** SELECT * FROM Books WHERE Quantity < 10;
- **Greater Than or Equal To (>=):** SELECT * FROM Books WHERE PublicationDate >= '2020-01-01';
- **Less Than or Equal To (<=):** SELECT * FROM Books WHERE Price <= 20;

c) Logical Operators:

- **AND:** SELECT * FROM Books WHERE Price > 10 AND Category = 'Fiction';
- **OR:** SELECT * FROM Books WHERE Author = 'Jane Austen' OR Author = 'Charles Dickens';
- **NOT:** SELECT * FROM Books WHERE NOT Category = 'Fiction';

d) Operators for Negating Conditions:

- **NOT IN:** SELECT * FROM Books WHERE Category NOT IN ('Fiction', 'Biography');
- **NOT LIKE:** SELECT * FROM Books WHERE Title NOT LIKE 'The%';
- **NOT BETWEEN:** SELECT * FROM Books WHERE Price NOT BETWEEN 10 AND 20;

Constraints in SQL

Constraints are rules we set on our database columns to maintain data integrity. They're like the rules in a game - they keep everything fair and consistent.

a) Primary Key:
Uniquely identifies each record in a table.

Example:

```
CREATE TABLE Customers (
    CustomerID INT PRIMARY KEY,
    Name VARCHAR(50),
    Email VARCHAR(100)
);
```

b) Foreign Key:
Creates a link between two tables.

Example:

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

c) Not Null:
Ensures a column cannot have NULL values.

```sql

```sql
CREATE TABLE Books (
    ISBN VARCHAR(13) PRIMARY KEY,
    Title VARCHAR(100) NOT NULL,
    Author VARCHAR(50) NOT NULL,
    Price DECIMAL(5,2)
);
```

d) Unique:
Ensures all values in a column are different.

```sql
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Email VARCHAR(100) UNIQUE,
    Name VARCHAR(50)
);
```

e) Check:
Specifies a condition for values in a column.

Example:

```sql
CREATE TABLE Books (
    ISBN VARCHAR(13) PRIMARY KEY,
    Title VARCHAR(100),
    Price DECIMAL(5,2) CHECK (Price > 0)
);
```

f) Default:
Provides a default value for a column.

Example:

```sql
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    OrderDate DATE DEFAULT CURRENT_DATE,
    Status VARCHAR(20) DEFAULT 'Pending'
);
```

Advanced constraints include concepts like composite keys (primary keys made of multiple columns) and calculated columns (columns whose values are derived from other columns).

SQL Set Operators

Set Operations in SQL allow you to combine and manipulate the results of two or more `SELECT` queries using operators like `UNION`, `UNION ALL`, `INTERSECT`, and `EXCEPT` (or `MINUS` in certain databases).

Key Set Operators

1. UNION: Combines the result sets of two or more queries, removing duplicate rows.

Example:

- SELECT Comment FROM PositiveFeedback
- UNION
- SELECT Comment FROM NegativeFeedback;

2. UNION ALL: Combines result sets and includes all rows, including duplicates.

Example:

- SELECT Comment FROM PositiveFeedback
- UNION ALL
- SELECT Comment FROM NegativeFeedback;

3. EXCEPT (or MINUS): Returns rows from the first query that are not in the second query. Only distinct rows are returned.

Example:

- SELECT ProductName FROM Sales
- EXCEPT
- SELECT ProductName FROM Returns;

4. INTERSECT: Returns only the common rows between two queries.

Example:

- SELECT ProductName FROM Sales
- INTERSECT
- SELECT ProductName FROM Returns;

SQL Joins:

Joins are used to combine rows from two or more tables based on a related column between them. Think of joins as ways to connect the dots between different pieces of information in your database.

a) Inner Join:
Returns only the matching rows from both tables.

Example:

```
SELECT Books.Title, Orders.OrderDate
FROM Books
INNER JOIN Orders ON Books.ISBN = Orders.BookISBN;
```

This shows all books that have been ordered, with their order dates.

b) Left Join:
Returns all rows from the left table and matching rows from the right table.

```
SELECT Customers.Name, Orders.OrderID
FROM Customers
LEFT JOIN Orders ON Customers.CustomerID = Orders.CustomerID;
```

This shows all customers and their orders, including customers who haven't made any orders.

c) Right Join:
Returns all rows from the right table and matching rows from the left table.

Example:

```
SELECT Books.Title, Reviews.Rating
FROM Books
RIGHT JOIN Reviews ON Books.ISBN = Reviews.BookISBN;
```

This shows all reviews and the books they're for, including reviews for books not in our database.

d) Full Outer Join:
Returns all rows when there's a match in either the left or right table.

```
SELECT Employees.Name, Departments.DepartmentName
FROM Employees
FULL OUTER JOIN Departments ON Employees.DepartmentID = Departments.DepartmentID;
```

This shows all employees and all departments, matching them where possible.

e) Self Join:
Joins a table to itself.

Example:

```
SELECT A.Title AS Book, B.Title AS RelatedBook
FROM Books A, Books B
WHERE A.Author = B.Author AND A.ISBN <> B.ISBN;
```

This finds pairs of books by the same author.

These join operations allow you to create complex queries that pull together information from multiple tables, giving you a comprehensive view of your data.

Sub-Query

Introduction:
SQL queries placed within another query are referred to as subqueries (nested queries or inner queries). They can be used in different sections of a SQL statement to carry out particular operations, like data retrieval, aggregation, and filtering. With the help of SQL's advanced subqueries, you can break down difficult jobs into smaller, more manageable chunks. Subqueries come in a variety of forms, each with a unique function.

Types of Sub-queries:

1. Scalar Subquery:
An outer query can employ a scalar subquery to deliver a single value for comparison or arithmetic operations. Typically, scalar subqueries are inserted into query sections when a single deal is expected.

Example:

```
-- Create the products table
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    price DECIMAL(10, 2)
);

-- Insert sample data into products
INSERT INTO products (product_id, product_name, price)
VALUES
    (1, 'Laptop', 800.00),
    (2, 'Phone', 500.00),
    (3, 'Tablet', 300.00),
    (4, 'Monitor', 250.00),
    (5, 'Keyboard', 50.00);
```

Query:

```
SELECT (SELECT COUNT(*) FROM products) AS total_products;
```

Output:

```
+---------------+
| total_products |
+---------------+
| 5             |
+---------------+
```

The scalar subquery calculates the total number of products in the products table, and the outer query displays this value using the alias total_products.

2. Single-row subquery :
A single-row subquery returns a single row of data compared to a single row from the outer query.

Example:

```
-- Create the products table
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(50),
    price DECIMAL(10, 2)
);

-- Insert sample data into products
INSERT INTO products (product_id, product_name, price)
VALUES
    (1, 'Laptop', 800.00),
    (2, 'Phone', 500.00),
    (3, 'Tablet', 300.00),
    (4, 'Monitor', 800.00),
    (5, 'Keyboard', 50.00);
```

Query:

```
SELECT product_name
FROM products
WHERE price = (SELECT MAX(price) FROM products);
```

Output:

```
+--------------+
| product_name |
+--------------+
| Laptop       |
| Monitor      |
+--------------+
```

Since, the subquery (SELECT MAX(price) FROM products) provides a single number (800.00), which is compared with the cost of each row in the outer query, this query is considered a single-row subquery. The aim is to find rows where the price is equal to the maximum price.

3. Multiple Layer subquery:
An outer query's numerous rows are compared with the multiple rows of data returned by a multi-row subquery.

```
Example:
-- Create the customers table
```

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);

-- Insert sample data into customers
INSERT INTO customers (customer_id, customer_name)
VALUES
    (1, 'Alice'),
    (2, 'Bob'),
    (3, 'Charlie'),
    (4, 'David');

-- Create the orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT
);

-- Insert sample data into orders
INSERT INTO orders (order_id, customer_id)
VALUES
    (101, 1),
    (102, 2),
    (103, 1),
    (104, 3),
    (105, 2);
```

Query:

```sql
SELECT customer_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders);
```

Output:

```
+-------------+
| customer_name |
+-------------+
| Alice       |
```

```
| Bob         |
| Charlie     |
+-------------+
```

4. Correlated subquery:
One type of subquery that references values from the outer query is called a correlated subquery. It can filter or retrieve related data and is executed for each row in the external question.

Example:

"Retrieve the last name, salary, and department ID of employees whose salary is greater than the average salary within their respective departments."

```sql
-- Create the employees table
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    last_name VARCHAR(50),
    salary DECIMAL(10, 2),
    department_id INT
);

-- Insert sample data into employees
INSERT INTO employees (employee_id, last_name, salary, department_id)
VALUES
    (1, 'Smith', 60000.00, 1),
    (2, 'Johnson', 70000.00, 1),
    (3, 'Williams', 55000.00, 2),
    (4, 'Jones', 80000.00, 2),
    (5, 'Brown', 65000.00, 3),
    (6, 'Davis', 72000.00, 3),
    (7, 'Miller', 61000.00, 1),
    (8, 'Wilson', 59000.00, 2);
```

Query:

```sql
-- Execute the given query
SELECT e1.last_name, e1.salary, e1.department_id
FROM employees e1
WHERE e1.salary > (
    SELECT AVG(e2.salary)
    FROM employees e2
    WHERE e2.department_id = e1.department_id
    GROUP BY e2.department_id
);
```

Output:

```
+-----------+--------+---------------+
| last_name | salary | department_id |
+-----------+--------+---------------+
| Johnson   | 70000  | 1             |
| Jones     | 80000  | 2             |
| Davis     | 72000  | 3             |         |+-----------+--------+-------------+
```

5.Nested Subquery:

A nested subquery is a subquery that appears inside another query's context. It is sometimes referred to as an inner or nested subquery. It's a robust approach that lets you use a query's outcome as a value or condition in another question.

Example:

The query you provided retrieves employees from departments where the average salary of the employees in that department is higher than the overall average salary of all employees in the company.

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(30)
);
-- Insert sample data into departments
INSERT INTO departments (department_id, department_name)
VALUES
    (10, 'Administration'),
    (20, 'Marketing'),
    (30, 'Purchasing'),
```

```
    (40, 'Human Resources'),
    (50, 'Shipping'),
    (60, 'IT');
```

```
--Table employee
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    first_name VARCHAR(50),
    last_name VARCHAR(50),
    salary DECIMAL(10, 2),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

```
-- Insert sample data into employees
INSERT INTO employees (employee_id, first_name, last_name, salary, department_id)
VALUES
    (101, 'John', 'Doe', 50000, 50),
    (102, 'Jane', 'Smith', 60000, 50),
    (103, 'Alice', 'Johnson', 70000, 60),
    (104, 'Bob', 'Williams', 55000, 60),
    (105, 'Charlie', 'Brown', 75000, 40),
    (106, 'David', 'Davis', 65000, 40),
    (107, 'Eva', 'Miller', 80000, 30),
    (108, 'Frank', 'Wilson', 72000, 30),
    (109, 'Grace', 'Taylor', 58000, 20),
    (110, 'Henry', 'Martin', 62000, 20);
```

Query:

```
    SELECT e.first_name, e.last_name, e.salary, e.department_id
FROM employees e
WHERE e.department_id IN (
    SELECT d.department_id
    FROM departments d
    WHERE (SELECT AVG(e2.salary)
           FROM employees e2
           WHERE e2.department_id = d.department_id
          ) > (SELECT AVG(salary) FROM employees)
);
```

Output:

```
# first_name, vlast_name,  salary,    department_id
Charlie,      Brown,      75000.00,        40
David,        Davis,      65000.00,        40
Eva,          Miller,     80000.00,        30
Frank,        Wilson,     72000.00,        30
```

Subquery using From and Where

Subqueries in SQL allow you to nest one query inside another. They are often used to fetch data required for the main query and can be applied within SELECT, FROM, and WHERE clauses.

Subquery in the FROM Clause

Also known as an inline view or derived table, a subquery in the FROM clause allows you to treat its result as a temporary table. This is useful for performing complex calculations or filtering before joining the derived table with other tables.

Example: Calculate the average number of items per order using the orders and order_items tables.

```sql
-- Create the order_items table
CREATE TABLE order_items (
    order_id INT,
    item_id INT
);



-- Insert sample data into order_items
INSERT INTO order_items (order_id, item_id)
VALUES
    (1, 101),
    (1, 102),
    (2, 103),
    (2, 104),
    (2, 105),
    (3, 106),
    (3, 107),
    (3, 108);

-- Create the orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY
);



-- Insert sample data into orders
INSERT INTO orders (order_id)
VALUES
    (1),
    (2),
    (3);
```

Solution:

```
  SELECT AVG(sub.avg_items)
FROM (
   SELECT order_id, COUNT(*) AS avg_items
   FROM order_items
   GROUP BY order_id
) AS sub;
```

How does it work?

- **SELECT AVG (sub.avg_items):** This is the main outer query. It calculates the average of the avg_items column from the subquery.
- **(SELECT order_id, COUNT(*) AS avg_items FROM order_items GROUP BY order_id) AS sub:** This is a subquery (also called a derived table) that calculates the number of items per order. Let's break down the subquery:
- **SELECT order_id, COUNT(*) AS avg_items:** In the subquery, it selects the order_id and uses the COUNT(*) function to count the number of items in each order. The result of this count is given the alias avg_items.
- **FROM order_items:** This specifies that the subquery is operating on the order_items table.
- **GROUP BY order_id:** This groups the results by order_id, which means the count of items is calculated for each unique order_id.
- **AS sub:** The subquery itself is given an alias sub, which is then used in the outer query.
- The outer query then calculates the average of the avg_items column from the subquery, which effectively gives you the average number of items per order across all orders.

In summary, this query first calculates the count of items per order using a subquery and then calculates the average of those counts using the outer query. This helps you find the average number of items in each order.

Advantage:

- Divide complicated queries into digestible chunks to make them easier to understand.
- Permits you to use the derived table more than once in your question.
- It can enhance the readability and structure of the query.

Disadvantage:

- Because the subquery must materialize as a temporary table, it may result in poorer performance if it produces a significant result set.
- If overused or nested too deeply, it can make the query more challenging to interpret.

2. The WHERE Clause subquery:

- The WHERE clause frequently contains a subquery that filters the main query's results according to the subquery's findings.
- This is helpful if you wish to apply extra conditions based on the outcome of the subquery or compare information from multiple tables.

Example:

Assume you have a products table and an orders table and want to find the names of products that have been ordered at least twice.

```sql
-- Create the products table
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(255)
);

-- Insert data into the products table
INSERT INTO products (product_id, product_name)
VALUES
    (1, 'Chocolate'),
    (2, 'Icecream'),
    (3, 'Milkshake');

-- Create the orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    product_id INT,
    order_date DATE
);

-- Insert data into the orders table
INSERT INTO orders (order_id, product_id, order_date)
VALUES
    (1, 1, '2023-08-01'),
    (2, 1, '2023-08-02'),
    (3, 2, '2023-08-01'),
    (4, 3, '2023-08-02'),
    (5, 2, '2023-08-03');
```

Solution:

```sql
  SELECT product_name
FROM products
WHERE product_id IN (
    SELECT product_id
    FROM orders
    GROUP BY product_id
    HAVING COUNT(*) >= 2
);
```

The subquery in this example calculates the count of orders for each product using the GROUP BY clause and filters only those products that have been ordered at least twice using the HAVING clause. The main query then retrieves the names of those products from the products table.

Advantage:

- Enable complex filtering and conditional reasoning based on the outcome of another query.
- Sometimes, it can be more efficient than Joining tables in questions.

Disadvantage:

- It is inefficient if the subquery yields extensive results because it must be run for every entry in the main query.
- While specific database systems may optimize certain subqueries, performance may occasionally be an issue.

Filter Query Results using Query From Different Table using subqueries

So, before getting into the concept, let us see the example and understand the idea.

Scenario:
You have two tables, orders and customers, and you want to retrieve the order details for customers who have placed orders on the same day as a specific customer.

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(50)
);

-- Insert sample data into customers
INSERT INTO customers (customer_id, customer_name)
VALUES
    (101, 'Alice'),
    (102, 'Bob'),
    (103, 'Charlie');

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE
);

-- Insert sample data into orders
INSERT INTO orders (order_id, customer_id, order_date)
VALUES
    (1, 101, '2023-08-01'),
    (2, 102, '2023-08-02'),
    (3, 101, '2023-08-03'),
    (4, 103, '2023-08-04');
```

Query:

```
SELECT o.order_id, o.customer_id, o.order_date
FROM orders o
WHERE o.order_date IN (SELECT order_date FROM orders
 WHERE customer_id = 101);
```

- This query gets the details of orders from the orders table with the same order date as any order placed by customer_id 101.
- To do this, the subquery (SELECT order_date FROM orders WHERE customer_id = 101) selects the order dates of orders placed by customer_id 101.
- The outer query then selects the order_id, customer_id, and order_date from the orders table, where the order_date matches any order dates obtained from the subquery.

Output:

```
+----------+-------------+------------+
| order_id | customer_id | order_date |
+----------+-------------+------------+
| 1        | 101         | 2023-08-01 |
| 3        | 101         | 2023-08-03 |
| 4        | 103         | 2023-08-03 |
+----------+-------------+------------+
```

Certainly! I'll expand this into a more detailed chapter format with examples. Here's a comprehensive overview of Windows Functions in SQL:

Windows Functions in SQL

Windows Functions in SQL are powerful tools that allow you to perform calculations across a set of rows that are related to the current row. Unlike regular aggregate functions, window functions return a result for every row in your query. They're called "window" functions because they operate on a window of data, which is a set of rows defined by the OVER clause.

These functions are particularly useful for computing running totals, moving averages, rankings, and for comparing values between rows. They can significantly simplify complex queries and improve query performance.

Basic Syntax of Window Functions

The general syntax for a window function is:

```
<window_function> (<expression>) OVER (
    [PARTITION BY <partition_clause>]
    [ORDER BY <order_clause>]
    [ROWS or RANGE <window_frame_clause>]
)
```

- `<window_function>` is the function you're using (e.g., SUM, RANK, LAG).
- `<expression>` is often a column name, but can be more complex.
- OVER clause defines the window of rows the function operates on.
- PARTITION BY divides the result set into partitions.
- ORDER BY specifies the logical order of rows within each partition.
- ROWS or RANGE clause defines the window frame within the partition.

Types of Window Functions

1. Aggregate Window Functions

These include SUM(), AVG(), COUNT(), MIN(), and MAX(). They perform a calculation across a set of rows and return a single value.

Example: Calculate a running total of sales

```
CREATE TABLE sales (
    sale_date DATE,
    amount DECIMAL(10,2)
);

INSERT INTO sales VALUES
('2023-01-01', 100),
('2023-01-02', 150),
('2023-01-03', 200),
('2023-01-04', 120);

SELECT sale_date, amount,
    SUM(amount) OVER (ORDER BY sale_date) AS running_total FROM sales;

--------------------------------------------------------------------------------
Output:
```
```
sale_date  | amount | running_total
-----------+--------+---------------
2023-01-01 | 100.00 | 100.00
2023-01-02 | 150.00 | 250.00
2023-01-03 | 200.00 | 450.00
2023-01-04 | 120.00 | 570.00
-----------------------------------
```

2. Ranking Window Functions

These include RANK(), DENSE_RANK(), ROW_NUMBER(), and NTILE(). They assign a rank or row number to each row within a partition.

Example: Rank employees by salary within each department

```
CREATE TABLE employees (
    emp_id INT,
    dept_id INT,
    salary DECIMAL(10,2)
);

INSERT INTO employees VALUES
(1, 1, 50000),
(2, 1, 60000),
(3, 2, 55000),
(4, 2, 55000),
(5, 1, 55000);

SELECT emp_id,dept_id, salary,
    RANK() OVER (PARTITION BY dept_id ORDER BY salary DESC) AS salary_rank FROM employees;

--------------------------------------------------------------------------------

Output:

emp_id | dept_id | salary   | salary_rank
-------+---------+----------+------------
2      | 1       | 60000.00| 1
5      | 1       | 55000.00| 2
1      | 1       | 50000.00| 3
3      | 2       | 55000.00| 1
4      | 2       | 55000.00| 1
---------------------------------------
```

3. Value Window Functions

These include LAG(), LEAD(), FIRST_VALUE(), and LAST_VALUE(). They access data from another row in the result set without having to join the table to itself.

Example: Compare each day's sales with the previous day

```
SELECT sale_date, amount,
    LAG(amount) OVER (ORDER BY sale_date) AS prev_day_amount,
    amount - LAG(amount) OVER (ORDER BY sale_date) AS day_over_day_change
FROM sales;

Output:
-------------------------------------------------------------------------------------
sale_date  | amount | prev_day_amount | day_over_day_change
-----------+--------+-----------------+---------------------
2023-01-01 | 100.00 | NULL            | NULL
2023-01-02 | 150.00 | 100.00          | 50.00
2023-01-03 | 200.00 | 150.00          | 50.00
2023-01-04 | 120.00 | 200.00          | -80.00
-------------------------------------------------------------
```

Advanced Window Function Concepts

1. Window Frames

Window frames allow you to define precisely which rows to include in the window function calculation. The syntax is:

```sql
ROWS BETWEEN <start> AND <end>
```

Where `<start>` and `<end>` can be:

- UNBOUNDED PRECEDING: Start from the first row of the partition
- n PRECEDING: n rows before the current row
- CURRENT ROW: The current row
- n FOLLOWING: n rows after the current row
- UNBOUNDED FOLLOWING: End at the last row of the partition

Example: Calculate a 3-day moving average of sales

```
SELECT
    sale_date,
    amount,
    AVG(amount) OVER (
        ORDER BY sale_date
        ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING
    ) AS moving_avg
FROM sales;


Output:
```
```
sale_date  | amount | moving_avg
-----------+--------+------------
2023-01-01 | 100.00 | 125.00
2023-01-02 | 150.00 | 150.00
2023-01-03 | 200.00 | 156.67
2023-01-04 | 120.00 | 160.00
```

2. NTILE Function

NTILE divides the rows into a specified number of ranked groups.

Example: Divide employees into 3 salary tiers

```
SELECT
    emp_id,
    salary,
    NTILE(3) OVER (ORDER BY salary DESC) AS salary_tier
FROM employees;

Output:

emp_id | salary   | salary_tier
-------+---------+------------
2      | 60000.00| 1
3      | 55000.00| 1
4      | 55000.00| 2
5      | 55000.00| 2
1      | 50000.00| 3
```

Conclusion

Window functions are a powerful feature in SQL that can simplify complex queries and provide valuable insights into your data. They allow you to perform calculations across sets of rows that are related to the current row, without collapsing the result into a single row as traditional aggregate functions do. By mastering window functions, you can write more efficient and expressive SQL queries, enabling advanced data analysis directly in your database.

Functions In SQL

Functions in SQL are a collection of predefined procedures that carry out specific tasks and produce a single value. Functions can be used to format numbers, conduct calculations, and manipulate data, among other things. They offer an easy way to incorporate computation and logic into SQL statements.

Types of Functions:

- System Defined Functions
- User Defined Functions

Scalar Functions:

Functions that operate on a single value and return a single value are called scalar functions. They are frequently used to modify data or carry out calculations within SQL statements.

- **UPPER():** Converts a string to uppercase.
- **LOWER():** Converts a string to lowercase.
- **LEN() or LENGTH():** Returns the length of a string.
- **ROUND():** Rounds a numeric value to a specified number of decimal places.
- **CONCAT():** Concatenates two or more strings

Aggregate Function:
Functions that perform calculations on a group of items and return a single result are called aggregate functions. Data summaries are created with them, and the GROUP BY clause is frequently used.

Examples:

SUM(): Calculates the sum of values in a column.
AVG(): Calculates the average of values in a column.
COUNT(): Counts the number of rows in a column.
MIN(): Returns the minimum value in a column.
MAX(): Returns the maximum value in a column.

Date and Time Function:
Date and time functions operate on date and time values, enabling you to conduct calculations, extract portions of dates, and format date/time information.

Examples:

- **GETDATE():** Returns the current date and time.
- **DATEADD():** Adds a specified interval to a date.
- **DATEDIFF():** Calculates the difference between two dates.
- **YEAR(), MONTH(), and DAY():** Extract parts of a date.

String Functions:
String functions operate on string values, allowing you to manipulate and format text data.

Examples:

- **LEFT(), RIGHT():** Extracts a specified number of characters from the left or right side of a string.
- **SUBSTRING():** Extracts a substring from a string.
- **CHARINDEX():** Returns the position of a substring within a string.
- **REPLACE():** Replaces occurrences of a substring with another substring.

Conversion Functions:
Conversion functions are used to convert data from one data type to another.

Examples:

- **CAST(), and Convert():** Convert one data type to another.
- **TO_CHAR(), TO_NUMBER():** Converts values to string or numeric formats (Oracle)

Window Functions:
Window functions perform calculations across a set of table rows related to the current row, similar to aggregate functions, but without reducing the result to a single value. They are used with the OVER clause.

Examples:

- **ROW_NUMBER():** Assigns a unique number to each row in a result set.
- **RANK(), DENSE_RANK():** Calculates the rank of rows based on a specified column's values.
- **LEAD(), LAG():** Accesses the value of a row at a specified physical offset from that row.

User-defined Functions:
You can write user-defined functions (UDFs) to perform particular computations or operations in SQL. UDFs let you design your functions with unique logic suited to your needs, in contrast to built-in functions offered by the database system. UDFs can be used in SQL statements much like built-in functions because they are created in SQL or another supported programming language.

Types:
- Scaler UDF
- Table-Valued Function

Scaler UDF:
Scalar UDFs return a single value based on the input parameters. They are commonly used to perform calculations, transformations, or data manipulations on a single value.

Create Table:

```
CREATE TABLE Products (
    ProductID INT PRIMARY KEY,
    ProductName VARCHAR(50),
    OriginalPrice DECIMAL(10, 2)
);

INSERT INTO Products (ProductID, ProductName, OriginalPrice) VALUES
    (1, 'Widget A', 100.00),
    (2, 'Widget B', 150.00),
    (3, 'Widget C', 200.00);
```

Create Calculated Discoubt price Function:

```
CREATE FUNCTION CalculateDiscountedPrice(
    @OriginalPrice DECIMAL(10, 2),
    @DiscountPercentage DECIMAL(5, 2)
)
RETURNS DECIMAL(10, 2)
BEGIN
    DECLARE @DiscountedPrice DECIMAL(10, 2);
    SET @DiscountedPrice = @OriginalPrice * (1 - @DiscountPercentage / 100);
    RETURN @DiscountedPrice;
END;
```

Using the calculated discounted function:

```
SELECT
    ProductID,
    ProductName,
    OriginalPrice,
    CalculateDiscountedPrice(OriginalPrice, 10) AS DiscountedPrice10,
    CalculateDiscountedPrice(OriginalPrice, 20) AS DiscountedPrice20
FROM Products;
```

Table-Valued Functions:
Table-valued UDFs return a table as a result. They are used to encapsulate complex queries and return a set of rows that you can use in subsequent SQL statements.

Example:

Let's say you want to create a function that returns all employees in a specific department.

Creating Table:

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    FirstName VARCHAR(50),
    LastName VARCHAR(50),
    DepartmentID INT
);

INSERT INTO Employees (EmployeeID, FirstName, LastName, DepartmentID) VALUES
    (5, 'Amit', 'Patel', 1),
    (6, 'Priya', 'Sharma', 2),
    (7, 'Rahul', 'Gupta', 1),
    (8, 'Sneha', 'Verma', 3);
```

Create the GetEmployeesInDepartment Function:

```
CREATE FUNCTION GetEmployeesInDepartment(
    @DepartmentID INT
)
RETURNS TABLE
AS
RETURN (
    SELECT EmployeeID, FirstName, LastName
    FROM Employees
    WHERE DepartmentID = @DepartmentID
);
```

Using the GetEmployeesInDepartment Function:

Now let's use the GetEmployeesInDepartment function to retrieve employees from a specific department.

```
SELECT * FROM GetEmployeesInDepartment(1);
```

To retrieve employees from the department with DepartmentID equal to 1, we are using the function GetEmployeesInDepartment in this query.

One input argument, @DepartmentID, is required for the GetEmployeesInDepartment function. For the chosen department, a table containing employee information (EmployeeID, FirstName, and LastName) is returned.

Views in SQL

- A view is actually a query, and the query's output becomes the view's content.
- The VIEW is a foundation table that can be updated, inserted into, deleted from, and joined with other tables and views.
- A view, like a table, can be accessed with a SQL SELECT command.
- A view can also be created by combining data from multiple tables.
- A view behaves like a virtual table. Since you can code a view name anywhere
- you'd code a table name, a view is sometimes called a viewed table.
- Views can be used to restrict the data that a user is allowed to access. In some
- databases, users may be allowed to access data only through views.

Creating a view

```
Syntax-

CREATE VIEW view_name AS
SELECT column1, column2.....
FROM table_name
WHERE [condition];


Check the following example-


CREATE VIEW PWskillsview
AS SELECT * FROM PWskills_student;


We have created a view "PWskillsview" as the table PWskills_students.
--------------------------------------------------------------------------------
Output-

12:15:18 CREATE VIEW PWskillsview AS SELECT * FROM
PWskills_student 0 row(s) affected 0.078 sec
```

Select view-

select * from PWskillsview ;
You will get the same output here as you select * from PWskills_student.

SQL Create View with specific columns and WHERE-

```
Here we have created a student view for the PWskills_student table.


CREATE VIEW studentview
AS SELECT Student_ID, First_name, Email_ID
FROM PWskills_student
WHERE Course_name = "Machine Learning";


Check that your view is created -
select * from studentview;

Student_ID   First_name         Email_ID
103          Muskan         musku@PWskills.ai
104          Aryan          arya@PWskills.ai
105          Naina          naina@PWskills.ai
```

Creating a view from multiple tables-

```
Here we are using two tables: PWskills_student and PWskills_courses.

CREATE VIEW multiple_table_view
AS SELECT a.First_name ,c.Course_ID
FROM PWskills_student a , PWskills_courses c
WHERE a.Student_ID = c.Student_ID;



Result-

First_name     Course_ID
Aashu           1000
Aashu           1001
Shiv            1002
Shiv            1003
Aashu           1004
shiv            1005
```

Drop view -
If you want to drop the view, use the following syntax-

DROP VIEW view_name;

```
Example-
DROP VIEW multiple_table_view;

Result-
13:55:37 drop view multiple_table_view 0 row(s) affected 0.047
sec
```

Popular DBMS software:

- **MySQL:** Open-source, great for web applications
- **PostgreSQL:** Open-source, known for reliability and advanced features
- **Oracle:** Commercial, used by many large corporations
- **Microsoft SQL Server:** Commercial, integrates well with other Microsoft products
- **MongoDB:** Popular NoSQL database for handling unstructured data

Choosing the right DBMS depends on factors like:

- Scale of your data
- Type of data you're working with
- Budget
- Required features (e.g., transaction support, scalability)
- Existing technology stack

For our bookstore example, MySQL or PostgreSQL might be good choices if we're building a web application, while MongoDB could be suitable if we're dealing with a lot of unstructured data like book reviews.

Database Design Principles

Good database design is crucial for efficiency, data integrity, and ease of use. Two key concepts in database design are normalization and Entity-Relationship Diagrams.

a) Normalization:

Normalization is a technique for organizing data in a database. It's like tidying up your room - everything has its place, and there's no unnecessary clutter.

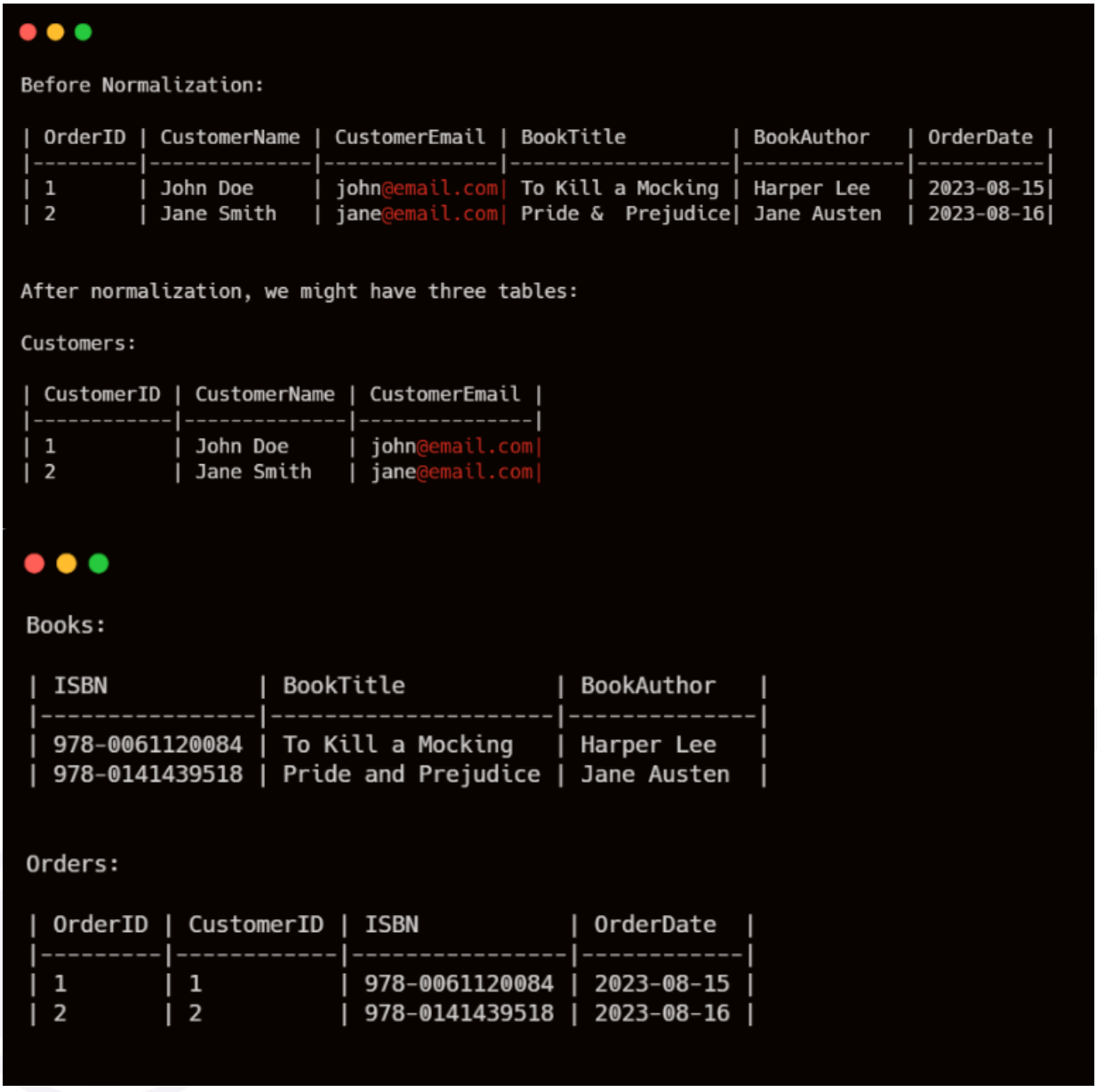
The main goals of normalization are:

- Minimize data redundancy (repeated data)
- Ensure data dependencies make sense
- Simplify data management

There are several levels of normalization, called normal forms. The most common are:

- First Normal Form (1NF): Each table cell should contain a single value, and each record needs to be unique.
- Second Normal Form (2NF): It's in 1NF and all non-key attributes are fully dependent on the primary key.
- Third Normal Form (3NF): It's in 2NF and all the attributes are only dependent on the primary key.

Example:
Consider an unnormalized table for our bookstore:



Before Normalization:

OrderID	CustomerName	CustomerEmail	BookTitle	BookAuthor	OrderDate
1	John Doe	john@email.com	To Kill a Mocking	Harper Lee	2023-08-15
2	Jane Smith	jane@email.com	Pride & Prejudice	Jane Austen	2023-08-16

After normalization, we might have three tables:

Customers:

CustomerID	CustomerName	CustomerEmail
1	John Doe	john@email.com
2	Jane Smith	jane@email.com

Books:

ISBN	BookTitle	BookAuthor
978-0061120084	To Kill a Mocking	Harper Lee
978-0141439518	Pride and Prejudice	Jane Austen

Orders:

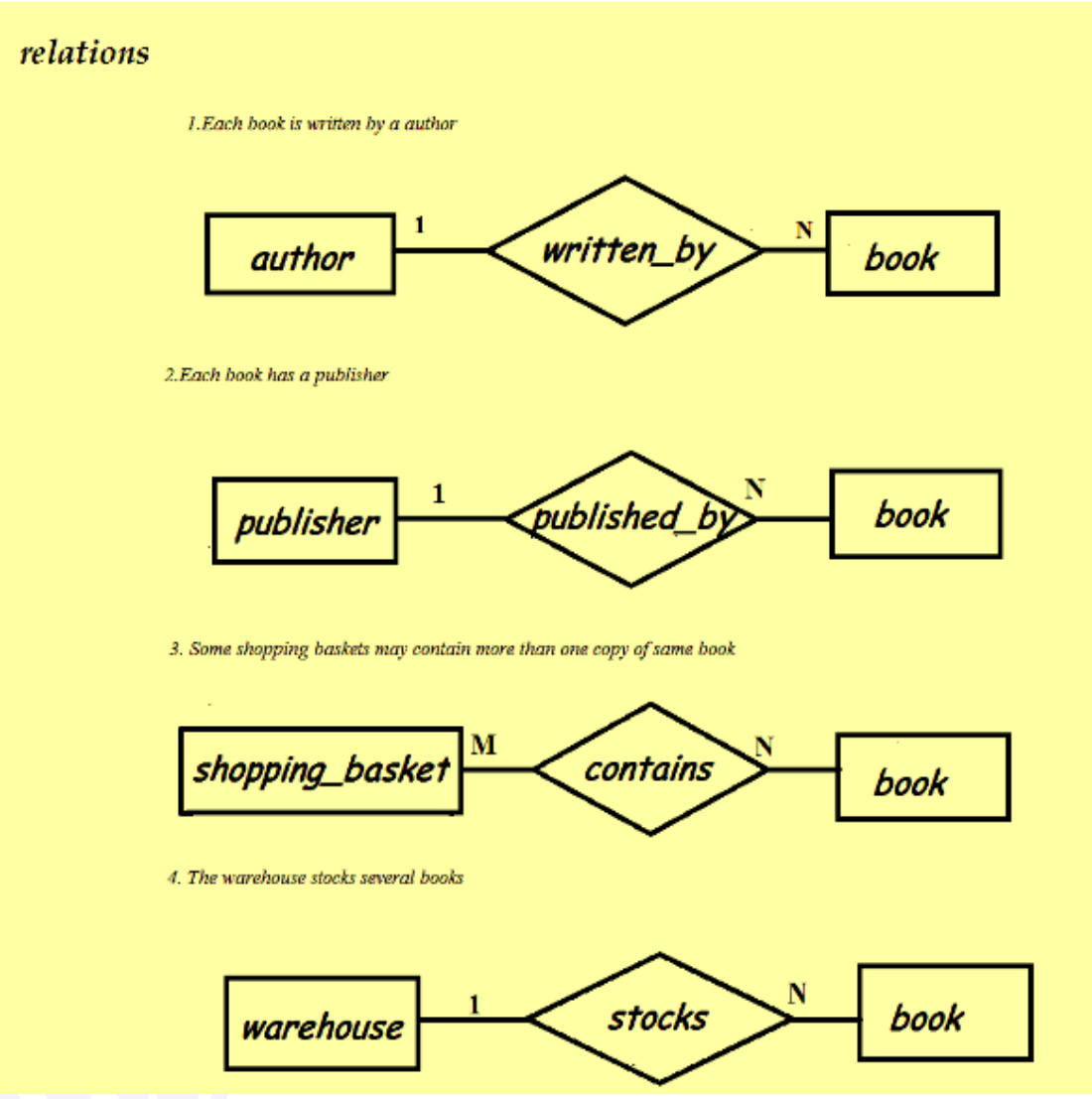
OrderID	CustomerID	ISBN	OrderDate
1	1	978-0061120084	2023-08-15
2	2	978-0141439518	2023-08-16

b) Entity-Relationship Diagrams (ERD):
An ERD is a visual representation of the relationships between entities in a database. It's like a map of your database.

Key components of an ERD:

- **Entities:** Represented by rectangles (e.g., Customers, Books, Orders)
- **Attributes:** Represented by ovals (e.g., CustomerName, ISBN, OrderDate)
- **Relationships:** Represented by diamond shapes connecting entities

For example, a simple ERD for our bookstore might look like:



This shows that Customers place Orders, and Orders include Books.

ACID Properties:

ACID properties ensure that database transactions are reliable and maintain data integrity:

Atomicity: A transaction is atomic, meaning it either completes in its entirety or not at all. If any part fails, the entire transaction returns to its original state.

Consistency: The database remains consistent before and after the transaction. All constraints, rules, and relationships defined in the database are enforced during the transaction.

Isolation: Each transaction is isolated from other transactions until it is completed. This ensures that the intermediate state of one transaction is invisible to other concurrent transactions.

Durability: Once a transaction is committed, its changes are permanent and persist even in system failure. The changes are stored permanently in non-volatile memory (e.g., disk).