Aktu | Database Management System | BCS-501

Unit-2

Syllabus

Relational data Model and Language: Relational Data Model Concepts, Integrity Constraints, Entity Integrity, Referential Integrity, Keys Constraints, Domain Constraints, Relational Algebra, Relational Calculus, Tuple and Domain Calculus. Introduction on SQL: Characteristics of SQL, Advantage of SQL. SQl Data Type and Literals. Types of SQL Commands. SQL Operators and Their Procedure. Tables, Views and Indexes. Queries and Sub Queries. Aggregate Functions. Insert, Update and Delete Operations, Joins, Unions, Intersection, Minus, Cursors, Triggers, Procedures in SQL/PL SQL

Topics Covered

- Relational Data Model Concepts
- Integrity Constraints:
- Relational Algebra
- Relational Calculus
- Tuple and Domain Calculus
- Introduction on SQL:
- Characteristics of SQL
- Advantage of SQL
- SQl Data Type and Literals.
- Types of SQL Commands. SQL Operators
- Tables, Views & Indexes
- Queries & SubQueries
- Aggregate Functions
- Insert, Update & Delete
- SQL Joins - 3PYQ
- Set Operations
- Cursors
- Triggers - 2PYQ
- Procedures in SQL/PL

Multi Atoms

Relational Data Model Overview

- Developed by E.F. Codd in 1970, the relational model organizes data into tables (relations) with rows and columns.
- Each table represents real-world entities and relationships, making it easier to manage and query than hierarchical or network databases.
- Examples of relational databases include MySQL, Oracle, DB2, and SQL Server.

Key Terminologies

Relation (Table): A table with rows and columns.
Example: A Student table with columns like **Stu_No, S_Name, and Gender**.

Tuple (Row): A single record in a table.
Example: A row in the Student table, like **(10112, 'Rama', 'F')**

Attribute (Column): A single property of a relation.
Example: **Stu_No, S_Name, Gender**.

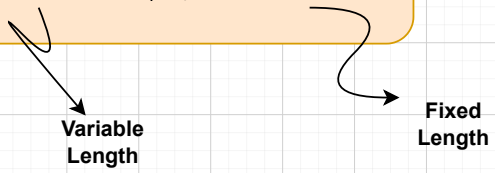
Domain: The set of valid values for an attribute.
Example: The domain for Gender is **{M, F}**.

Cardinality: The number of rows in a table.
Example: If Student has 5 rows, its cardinality is **5**.

Degree: The number of columns in a table.
Example: If Student has 5 attributes, its degree is **5**.

Multi Atoms

Relational Schema: Defines the table's structure, including names and data types for each column.
Example: Student schema could be **Student(Stu_No: INT, S_Name: VARCHAR, Gender: CHAR)**.



Relational Key: Unique identifiers for rows (e.g., **primary key**).
Example: Stu_No uniquely identifies each student.

Properties of Relations

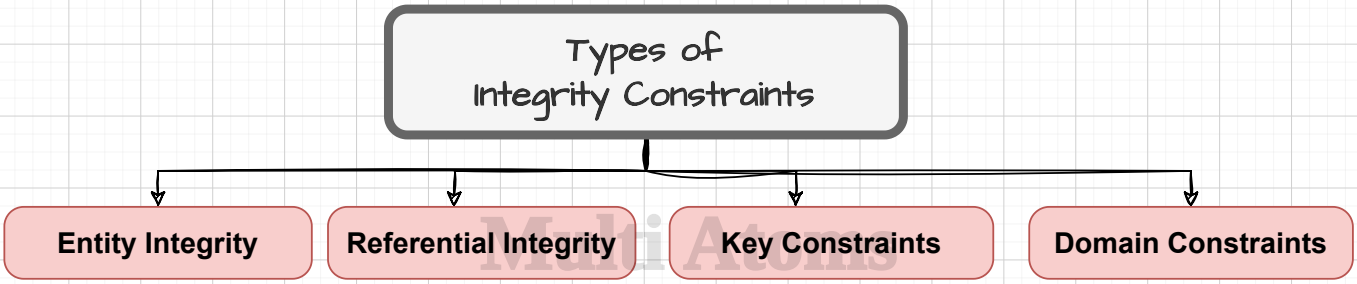
- Each table has a unique name, and each attribute also has a unique name.
- Tables do not have duplicate rows; each row is distinct.
- Each cell contains a single (atomic) value.
- Order of rows and columns is not significant. -> The order of rows and columns doesn't change the meaning of the data.

Basic Operations

1. **Insert**: Adds new rows to a table.
2. **Delete**: Removes rows from a table.
3. **Update**: Changes values in rows.
4. **Retrieve**: Fetches data based on queries.

Integrity Constraints

- Integrity constraints are rules that ensure data remains accurate, consistent, and reliable in a database. They prevent errors and keep the data meaningful.



I. Entity Integrity

It constraints enforce that every table must have a primary key that uniquely identifies each row, and that primary key cannot be NULL.

Product_ID	Product_Name	Price	Stock_Quantity
P1001	Laptop	70000	10
P1002	Phone	30000	50
P1003	Tablet	25000	30
NULL	Smartwatch	15000	20

- **Entity Integrity Constraint**: Product_ID (primary key) must not be NULL.
- **Explanation**: The row with NULL in Product_ID violates the entity integrity constraint, as the primary key must uniquely identify each row.
- **Violation Example**: Every product should have a non-null unique Product_ID.

2. Referential Integrity

It ensures that foreign keys in a table correctly reference primary keys in another related table.

StudentID	Name	Age
101	Alice	20
102	Bob	22
103	Carol	19

EnrollmentID	StudentID	CourseID
1	101	CS101
2	102	CS102
3	104	CS103

- **Referential Integrity Rule**: Each StudentID in Enrollment must exist in the Student table.
- **Explanation**: In the Enrollment table, StudentID acts as a foreign key referencing the primary key StudentID in the Student table.
- **Violation Example**: In the row with EnrollmentID 3, the StudentID 104 does not exist in the Student table, which violates referential integrity since it references a non-existent student.

Multi Atoms

3. Key Constraints

Key constraints ensure that the value in a column or set of columns must uniquely identify each row. A table can have multiple keys, but one will be designated as the primary key.

StudentID	Name	Age
101	Alice	20
102	Bob	22
103	Carol	19

- **Key Constraint**: Customer_ID must be unique.
- **Explanation**: The row with Customer_ID C001 violates the key constraint because it appears more than once in the table.
- **Violation Example**: The database should not allow two customers with the same Customer_ID.

4. Domain Constraints

Domain constraints define the set of valid values for a given column. They ensure that only permissible data types and values are allowed in the column.

Employee_ID	Name	Department	Salary
E001	Alice	HR	50000
E002	Bob	IT	60000
E003	Charlie	Marketing	45000
E004	David	HR	55000

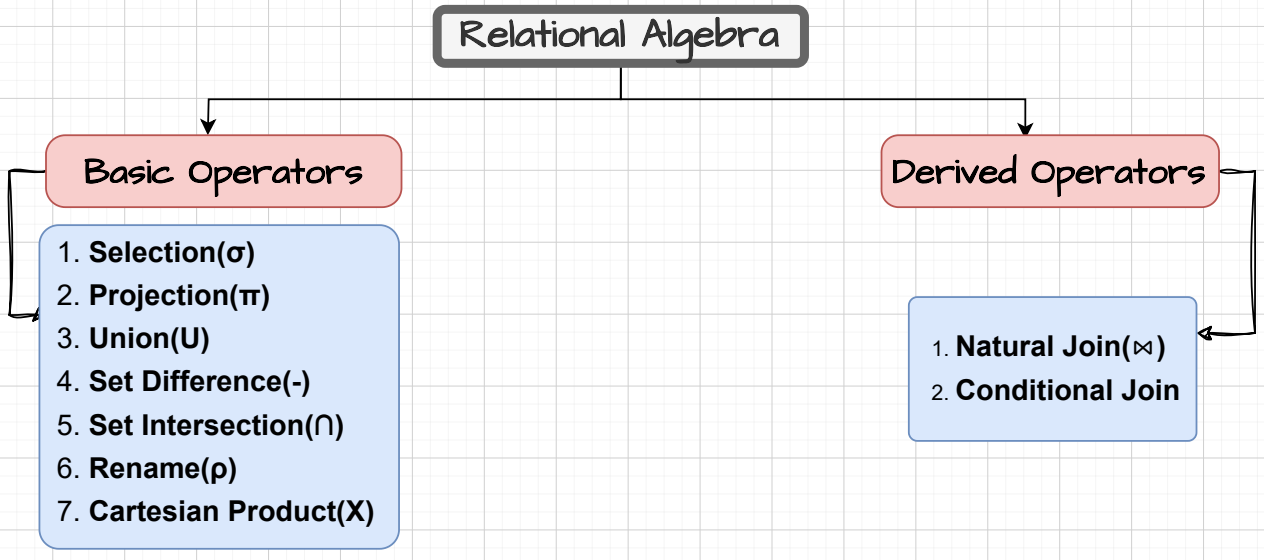
- **Domain Constraint:** Salary must be a positive integer.
- **Explanation:** The Salary column can only contain numerical values that are positive (no negative values or invalid data types).
- **Violation Example:** If the salary for an employee is entered as -5000, it would violate the domain constraint.

Relational Algebra

Multi Atoms

Relational Algebra is a **procedural query language** used to work with data in **relational databases**. It uses a **set of operators to retrieve, filter, and combine data, and it provides a foundation for SQL** and other database query languages.

Each operator takes relations (tables) as input and **outputs a new relation**, making it easy to build complex queries by combining operations.



Basic Operators in Relational Algebra

1. Selection (σ)
The Selection operator filters rows based on a specific condition.

Example: Suppose we have a Students table and want to find students who scored above 80.

Student_ID	Name	Age	Score
1	Alice	22	85
2	Bob	20	75
3	Charlie	23	90
4	David	21	65

Query: σ(Score > 80) Students

Result:

Student_ID	Name	Age	Score
1	Alice	22	85
3	Charlie	23	90

2. Projection (π)
The Projection operator retrieves specific columns from a table, discarding others.

Example: To view only the Name and Score columns from a Students table:

TutorialAtoms

Student_ID	Name	Age	Score
1	Alice	22	85
2	Bob	20	75
3	Charlie	23	90
4	David	21	65

Query: π(Name, Score) Students

Result:

Name	Score
Alice	85
Bob	75
Charlie	90
David	65

3. Union (U)
The Union operator combines rows from two tables into a single result, removing any duplicate rows. Both tables must have the same structure (same columns).

Example: Combine two tables Class_A and Class_B to get a single list of students:

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A U Class_B

Result:

Student_ID	Name
1	Alice
2	Bob
3	Charlie

4. Set Difference (-)
The Set Difference operator returns rows that are in one table but not in the other. Both tables must have the same structure.

Example: Find students in Class_A who are not in Class_B

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A - Class_B

Multi Atoms

Result:

Student_ID	Name
1	Alice

5. Set Intersection (∩)
The Intersection operator returns rows that are common to both tables. Both tables must have the same structure.

Example: Find students who are enrolled in both Class_A and Class_B

Student_ID	Name
1	Alice
2	Bob

Student_ID	Name
3	Charlie
2	Bob

Query: Class_A ∩ Class_B

Result:

Student_ID	Name
2	Bob

6. Rename (ρ)
The Rename operator allows you to give a temporary name to a table or a column, helpful for complex queries.

Example: Rename the Students table to Student_Info:

Query: ρ(Student_Info) Students

Result: The table is temporarily named Student_Info, allowing easier reference in complex queries.

7. Cartesian Product (X)
The Cartesian Product (or Cross Product) operator combines every row of one table with every row of another, resulting in all possible row combinations.

Example: Combine a Students table and a Courses table to show all possible student-course pairings:

Query: Students X Courses

Muki Atoms

Student_ID	Name
1	Alice
2	Bob

Course_ID	Course_Name
101	Math
102	Physics

Result:

Student_ID	Name	Course_ID	Course_Name
1	Alice	101	Math
1	Alice	102	Physics
2	Bob	101	Math
2	Bob	102	Physics

Additional Operators

1. Natural Join (⋈)
The Natural Join operator combines rows from two tables based on a common attribute with matching values, producing a single table with columns from both tables.

Example: Join Employees and Departments tables based on the Dept_ID column:

Emp_ID	Name	Dept_ID
1	Alice	101
2	Bob	102
3	Charlie	101

Query: Employees ⋈ Departments

Result:

Emp_ID	Name	Dept_ID	Dept_Name
1	Alice	101	HR
3	Charlie	101	HR
2	Bob	102	Sales

2. Conditional Join

The Conditional Join works similarly to the natural join but allows conditions other than equality.

Example: Combine Students and Scores tables where Students.Score is greater than Scores.Min_Score.

Student_ID	Name	Score
1	Alice	85
2	Bob	75
3	Charlie	90
4	David	60

Operation: Perform a conditional join on Students and Scores where Students.Score > Scores.Min_Score.

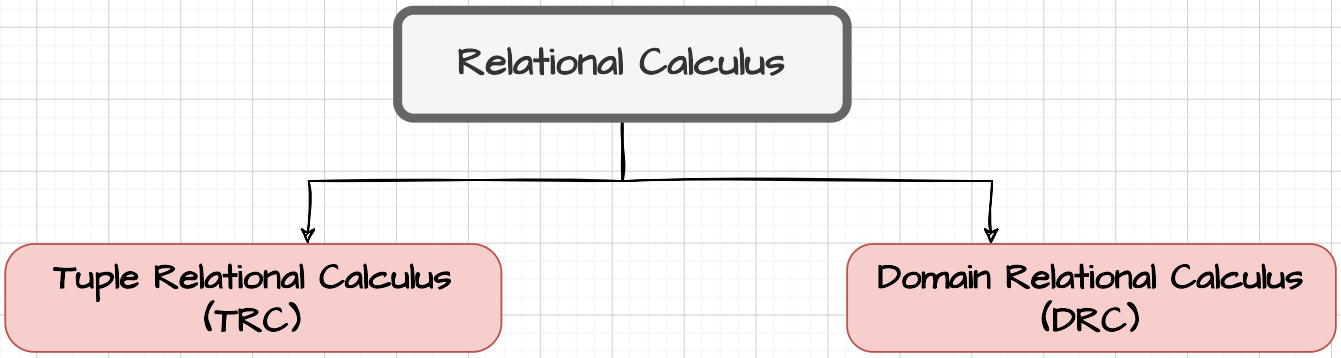
Query: Students ⋈ (Score > Min_Score) Scores

Result:

Student_ID	Name	Score	Min_Score	Level
1	Alice	85	70	Average
1	Alice	85	80	Good
2	Bob	75	70	Average
3	Charlie	90	70	Average
3	Charlie	90	80	Good
3	Charlie	90	90	Excellent

Relational Calculus

Relational Calculus is a **non-procedural query language** used in databases, meaning it focuses on **what data to retrieve rather than the steps to retrieve it**. Unlike Relational Algebra, which specifies a sequence of operations, Relational Calculus describes the desired result.



Multi Atoms

1. Tuple Relational Calculus (TRC)

In TRC, we use a tuple variable (such as t) that iterates over each row of the table, checking if the row meets the specified condition.

Syntax: { t | Condition(t) }
Here, t represents a tuple, and Condition(t) defines the condition that t must satisfy.

Example: Customer Database:

Customer_id	Name	Zip code
1	Rohit	12345
2	Rahul	13245
3	Rohit	56789
4	Amit	12345

1. Get all data for customers with Zip code 12345:

Query: { t | t ∈ Customer ∧ t.Zipcode = 12345 }

Result:

Customer_id	Name	Zip code
1	Rohit	12345
4	Amit	12345

2. Domain Relational Calculus (DRC)

In DRC, we use domain variables that represent specific columns. DRC queries specify conditions that the values in these columns must satisfy.

Syntax: { <d1, d2, ..., dn> | Condition(d1, d2, ..., dn) }
Here, d1, d2, ..., dn are domain variables, representing individual columns.

Example: Customer Database:

Customer_id	Name	Zip code
1	Rohit	12345
2	Rahul	13245
3	Rohit	56789
4	Amit	12345

1. Get all data for customers with Zip code 12345:

Query: { <d1, d2, d3> | <d1, d2, d3> ∈ Customer ∧ d3 = 12345 }

Result:

Customer_id	Name	Zip code
1	Rohit	12345
4	Amit	12345

Introduction to SQL

SQL (Structured Query Language) is a standard language used to manage and interact with relational databases. It allows users to **create, read, update, and delete data**, making it essential for data-driven applications across various industries.

- **Unified Database Management:** SQL provides a standardized way to handle relational

databases.

- **Core Functions:** It includes tools for data retrieval, manipulation, control, and definition, making it powerful for complex data queries and analysis

Characteristics of SQL

- **Declarative Language:** SQL focuses on what data is needed, not how to get it (Procedural), leaving optimization to the database engine.
- **Set-Based Operations:** SQL processes multiple rows at once, making it efficient for bulk operations.
- **Data Independence:** Users interact with data without needing to know its physical storage.
- **Standardized Syntax:** SQL is standardized across platforms (e.g., MySQL, PostgreSQL, Oracle), promoting compatibility.
- **Relational Model:** SQL uses tables (rows and columns), supporting flexible data relationships.

Multi Atoms

Advantages of SQL

- **Ease of Use:** SQL has readable, user-friendly commands (e.g., SELECT, INSERT).
- **Portability:** SQL works across platforms and database systems with minimal changes.
- **Scalability:** Suitable for small databases and large, complex datasets.
- **Data Security:** SQL supports access controls to protect data.
- **Integration:** SQL is widely supported in programming and applications, enhancing its utility in tech stacks.

SQL Data Types

Data types in SQL define the kind of data each column can hold. Here are some of the most commonly used data types:

1. Character Types:

VARCHAR(size): Stores variable-length text data. Only occupies the space needed, up to the defined maximum size.

- Example: VARCHAR(50) can store up to 50 characters. It's commonly used for names, addresses, and other variable-length text.

CHAR(size): Stores fixed-length text data, padding shorter values with spaces.

- Example: CHAR(10) will store exactly 10 characters, even if some are empty spaces.

2. Numeric Types:

INT: Stores integer (whole number) values.

- Example: Used for age, count, or other whole number fields without decimals.

DECIMAL(p, s) or NUMERIC(p, s): Stores fixed-point decimal numbers. p is the precision (total digits), and s is the scale (digits after the decimal).

Multi Atoms

- Example: DECIMAL(5, 2) allows up to 5 digits, with 2 after the decimal (e.g., 123.45).

FLOAT/REAL/DOUBLE: Stores floating-point numbers, useful for scientific and approximate calculations.

3. Date and Time Types:

- **DATE**: Stores date values as `YYYY-MM-DD`.
  - *Example*: `2023-01-15` could represent a birth date.
- **TIME**: Stores time values in `HH:MM:SS` format.
  - *Example*: `12:45:30` could represent the time of an event.
- **TIMESTAMP**: Stores both date and time, usually with time zone information.
  - *Example*: `2023-01-15 12:45:30` could log the exact time of a transaction

4. Boolean Types:

- **BOOLEAN:** Stores true or false values, typically represented as TRUE, FALSE, or NULL (unknown).

Example: Useful for fields like is_active or has_paid to indicate yes/no statuses.

5. Binary Types:

- **BLOB (Binary Large Object):** Stores large binary data, often used for files, images, or multimedia content.
- **BINARY(size) and VARBINARY(size):** Similar to CHAR and VARCHAR, but for binary data.

Multi Atoms

Literals in SQL

Literals are **fixed values explicitly written in SQL statements**. They include strings, numbers, and dates, and are commonly used in SELECT, INSERT, WHERE, and other SQL statements.

- **String Literals**:
  - Enclosed in single quotes (`'...'`).
  - *Example*: `SELECT * FROM Students WHERE name = 'Alice';`
- **Numeric Literals**:
  - Can be written without quotes, as they represent direct numeric values.
  - *Example*: `SELECT * FROM Products WHERE price > 100;`
- **Date Literals**:
  - Typically enclosed in single quotes and formatted as `YYYY-MM-DD`.
  - *Example*: `SELECT * FROM Orders WHERE order_date = '2023-01-15';`

- **Boolean Literals**:
  - Often used directly as `TRUE` or `FALSE`.
  - *Example*: `SELECT * FROM Users WHERE is_active = TRUE;`

Example Queries Using Data Types and Literals

Creating a Table with Different Data Types:

```
CREATE TABLE Employees (
    employee_id INT,
    name VARCHAR(50),
    birth_date DATE,
    salary DECIMAL(10, 2),
    is_manager BOOLEAN
);
```

Inserting Literal Values into a Table:

Multi Atoms

```
INSERT INTO Employees (employee_id, name, birth_date, salary, is_manager)
VALUES (1, 'John Doe', '1990-05-15', 75000.00, TRUE);
```

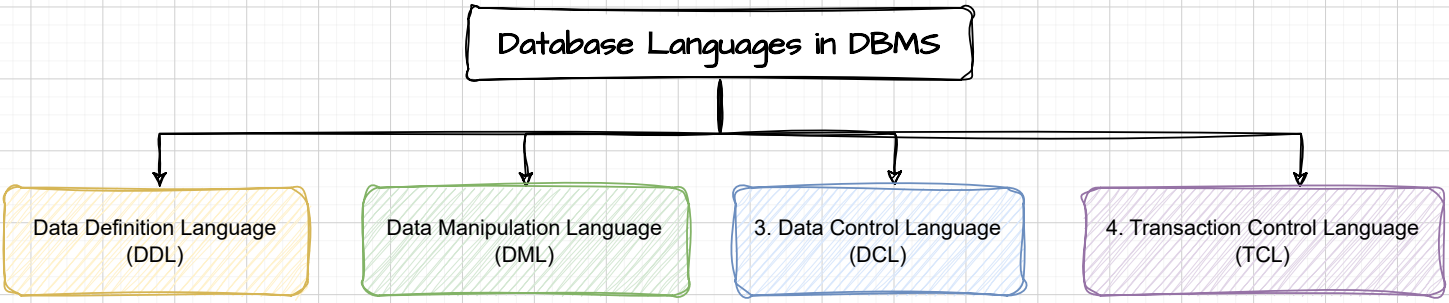
Query Using Literals in **WHERE** Clause:

```
SELECT * FROM Employees
WHERE salary > 50000 AND birth_date < '1995-01-01';
```

In DBMS Playlist Check Lec-1.3

Types of SQL Commands

SQL commands are grouped into categories based on their functionality. These categories define how we interact with data, manage the database structure, control access, and handle transactions. Here's an overview of each command type and its typical commands:



Category	Full Form	Purpose	Key Commands
DDL	Data Definition Language	Defines or modifies database structures and schema.	`CREATE`, `ALTER`, `DROP`, `TRUNCATE`, `RENAME`, `COMMENT`
DML	Data Manipulation Language	Manages and manipulates data within tables.	`SELECT`, `INSERT`, `UPDATE`, `DELETE`, `MERGE`
DCL	Data Control Language	Controls access to data and permissions.	`GRANT`, `REVOKE`
TCL	Transaction Control Language	Manages transactions in the database, ensuring data integrity.	`COMMIT`, `ROLLBACK`, `SAVEPOINT`, `SET TRANSACTION`

Multi Atoms

1. Data Definition Language (DDL)

DDL commands define and modify the structure of a database, including tables, schemas, and indexes. These commands directly affect the database schema and are often used to set up the initial database structure.

Common DDL Commands:

- **CREATE**: Creates new database objects (e.g., tables, indexes).

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

- **ALTER**: Modifies existing database objects (e.g., adding a new column to a table)

```
ALTER TABLE Employees ADD Salary DECIMAL(10, 2);
```

- **DROP**: Deletes existing objects from the database (e.g., tables, indexes).

```
DROP TABLE Employees;
```

- **TRUNCATE**: Removes all records from a table without deleting the table itself.

```
TRUNCATE TABLE Employees;
```

- **RENAME**: The RENAME command only changes the name of the object; it does not affect the data or structure of the table.

```
RENAME Employees TO Staff;
```

Purpose: DDL commands help manage the structure of the database by creating, altering, and deleting database objects.

2. Data Manipulation Language

DML commands manipulate data within existing tables. They are the core commands used for querying and modifying records in a database.

Multi Atoms

Common DML Commands:

- **SELECT**: Retrieves data from the database.

```
SELECT * FROM Employees WHERE Department = 'IT';
```

- **INSERT**: Adds new records to a table.

```
INSERT INTO Employees (EmployeeID, Name, Department) VALUES (1, 'John Doe', 'HR');
```

- **UPDATE**: Modifies existing data in the table.

```
UPDATE Employees SET Department = 'Finance' WHERE EmployeeID = 1;
```

- **DELETE**: Removes data from a table.

```
DELETE FROM Employees WHERE EmployeeID = 1;
```

- **DML commands focus on manipulating the data inside tables, as opposed to DDL, which deals with database structure.**
- **These commands are typically part of day-to-day database operations.**
- **SELECT is the most commonly used DML command for retrieving data.**

3. Data Control Language (DCL)

DCL is used to control access to the database. It includes commands that manage user permissions and access controls.

Common DCL Commands:

- **GRANT**: Gives user access privileges to database objects.

```
GRANT SELECT, INSERT ON Employees TO 'user1';
```

Multi Atoms

- **REVOKE**: Removes user access privileges.

```
REVOKE INSERT ON Employees FROM 'user1';
```

4. Transaction Control Language (TCL)

TCL commands manage transactions within the database, ensuring that groups of operations are either completely executed or not executed at all.

Common TCL Commands:

- **COMMIT**: Saves all changes made during the transaction.

```
COMMIT;
```

- **ROLLBACK**: Reverts the database to its previous state before the transaction began.

```
ROLLBACK;
```

- **SAVEPOINT**: Sets a point within a transaction to which you can roll back.

```
SAVEPOINT savepoint1;
```

SQL Operators and Their Precedence

SQL uses various operators to perform **operations on data within queries**. These operators fall into categories, each serving specific purposes, such as **calculations, comparisons, or logic-based decisions**. Understanding the order in which SQL evaluates these o**perators is essential for building accurate and efficient queries**. Here's a breakdown:

1. Arithmetic Operators

Arithmetic operators perform mathematical calculations in SQL queries. They are often used to calculate values directly within SELECT statements.

Multi Atoms

- **+ (Addition):** Adds two numbers.
  - *Example*: `SELECT salary + bonus AS Total_Pay FROM Employees;`
- **- (Subtraction):** Subtracts one number from another.
  - *Example*: `SELECT salary - deduction AS Net_Salary FROM Employees;`
- *** (Multiplication)**: Multiplies two numbers.
  - *Example*: `SELECT price * quantity AS Total_Cost FROM Orders;`
- **/ (Division):** Divides one number by another.
  - *Example*: `SELECT total_cost / items AS Cost_Per_Item FROM Purchases;`

2. Comparison Operators

Comparison operators are used to compare two values and return a boolean result (TRUE or FALSE). They are primarily used in conditions within WHERE clauses to filter data.

- **= (Equal to):** Checks if two values are equal.
  - *Example*: `SELECT * FROM Students WHERE grade = 'A';`
- **<> or != (Not equal to):** Checks if two values are not equal.
  - *Example*: `SELECT * FROM Employees WHERE department <> 'HR';`
- **> (Greater than):** Checks if the left value is greater than the right value.
  - *Example*: `SELECT * FROM Products WHERE price > 100;`

- **< (Less than):** Checks if the left value is less than the right value.
  - *Example*: `SELECT * FROM Orders WHERE quantity < 50;`
- **>= (Greater than or equal to)**: Checks if the left value is greater than or equal to the right value.
  - *Example*: `SELECT * FROM Sales WHERE amount >= 500;`
- **<= (Less than or equal to):** Checks if the left value is less than or equal to the right value.
  - *Example*: `SELECT * FROM Inventory WHERE stock <= 20;`

3. Logical Operators

Multi Atoms

Logical operators are used to combine multiple conditions in SQL queries, allowing for complex filtering. They are often used in conjunction with comparison operators.

- **AND**: Returns `TRUE` if both conditions are true.
  - *Example*: `SELECT * FROM Students WHERE grade = 'A' AND age > 18;`
- **OR**: Returns `TRUE` if at least one condition is true.
  - *Example*: `SELECT * FROM Employees WHERE department = 'Sales' OR department = 'Marketing';`
- **NOT**: Reverses the result of a condition.
  - *Example*: `SELECT * FROM Products WHERE NOT price < 50;`

4. Precedence of Operators

Operator precedence determines **the order in which SQL evaluates operators in an expression.** Understanding precedence is crucial to ensuring that SQL expressions are interpreted as intended. SQL follows a **standard precedence order**, with **higher precedence operators evaluated first**:

1. **Arithmetic Operators (*, /, +, -)**
2. **Comparison Operators (=, <>, !=, >, <, >=, <=)**
3. **Logical NOT**
4. **Logical AND**
5. **Logical OR**

Note: Use parentheses () to control or override the default precedence when needed.

Examples of Operator Precedence in SQL

Without Parentheses:

```
SELECT * FROM Employees WHERE department = 'Sales' OR department = 'HR' AND salary > 60000;
```

Interpretation: Since AND has higher precedence than OR, this query filters employees in the "HR" department with a salary greater than 60,000 or anyone in "Sales."

Multi Atoms

With Parentheses:

```
SELECT * FROM Employees WHERE (department = 'Sales' OR department = 'HR') AND salary > 60000;
```

Interpretation: The use of parentheses ensures that the query returns employees in either "Sales" or "HR" who also have a salary above 60,000.

Tables, Views, and Indexes in SQL

Tables, Views, and Indexes are fundamental database structures used to store, manage, and optimize data access in SQL.

Creating Tables: A table is a collection of rows and columns used to organize data in a relational database. To create a table, use the CREATE TABLE command, specifying the columns and data types.

```
CREATE TABLE table_name (
    column1 datatype constraints,
    column2 datatype constraints,
    ...
);
```

Constraints:

PRIMARY KEY: Uniquely identifies each row in the table. Example

```
CREATE TABLE Employees (
    EmployeeID INT PRIMARY KEY,
    Name VARCHAR(50)
);
```

FOREIGN KEY: Links two tables to enforce referential integrity. Example:

```
CREATE TABLE Orders (
    OrderID INT PRIMARY KEY,
    EmployeeID INT,
    FOREIGN KEY (EmployeeID)
REFERENCES Employees(EmployeeID)
);
```

Multi Atoms

Views

A view is a virtual table based on the result of a query. It allows you to present data in a simplified or secure way without modifying the underlying tables.

```
CREATE VIEW view_name AS
SELECT columns
FROM table
WHERE condition;
```

```
CREATE VIEW EmployeeNames
AS
SELECT EmployeeID, Name
FROM Employees;
```

Advantages of Views:

- Security: Restrict access to specific data by only showing selected columns.
- Simplicity: Simplify complex queries by creating reusable views.

Indexes

An index is a database structure that improves the speed of data retrieval on columns frequently used in search conditions, such as in WHERE clauses.

Syntax:
CREATE INDEX index_name ON table_name (column_name);

Ex:
CREATE INDEX idx_employee_name ON Employees (Name);

Benefits of Indexes:

- Speed up search operations.
- Improve the performance of complex queries, especially in large tables.

Queries

Basic SELECT Queries

```
SELECT column1, column2, ...
FROM table_name
WHERE condition
ORDER BY column [ASC|DESC];
```

Multi Atoms

- **SELECT:** Specifies the columns to retrieve.
- **WHERE:** Filters rows based on conditions.
- **ORDER BY:** Sorts the result in ascending (ASC) or descending (DESC) order.

Example:

```
SELECT employee_id, name, department
FROM Employees
WHERE department = 'Sales';
```

```
SELECT employee_id, name, salary
FROM Employees
WHERE salary BETWEEN 30000 AND 50000
ORDER BY salary DESC;
```

Subqueries

A subquery is a query embedded within another query, typically in the WHERE, HAVING, or FROM clause.

Key Points

- Purpose: Subqueries retrieve data to be used by the main query.
- Placement: Used in SELECT, INSERT, UPDATE, DELETE statements.
- Operators: Work with =, <, >, IN, and LIKE.

```
SELECT column_name
FROM table_name
WHERE column_name operator
   (SELECT column_name FROM table_name WHERE ...);
```

Examples:

1.Retrieve Data
Get employees working in the same department as 'John':

```
SELECT NAME, DEPARTMENT
FROM EMPLOYEE
WHERE DEPARTMENT IN (SELECT DEPARTMENT FROM
EMPLOYEE WHERE NAME = 'John');
```

2.Insert Data
Insert customers from New_Customers table into Customers table:

Multi Atoms

```
INSERT INTO Customers (Name, City)
SELECT Name, City
FROM New_Customers
WHERE City = 'New York';
```

3.Delete Data
Delete products from Products table that are not in stock:

```
DELETE FROM Products
WHERE Product_ID NOT IN (SELECT Product_ID FROM Stock WHERE Quantity > 0);
```

4.Update Data
Increase salaries of employees whose department has an average salary greater than $5000:

```
UPDATE EMPLOYEE
SET SALARY = SALARY * 1.1
WHERE DEPARTMENT IN (SELECT DEPARTMENT FROM EMPLOYEE GROUP BY
DEPARTMENT HAVING AVG(SALARY) > 5000);
```

Aggregate Functions

AKTU-2023-24

Aggregate functions in SQL allow us to perform calculations on multiple rows of data to return a single value or grouped results

1. **COUNT()**: Counts the number of rows.

```
SELECT COUNT(*) AS TotalEmployees
FROM Employees;
```

2. **SUM()**: Calculates the sum of numeric values.

```
SELECT SUM(Revenue) AS TotalRevenue
FROM Sales;
```

3. **AVG()**: Finds the average value of a numeric column

```
SELECT AVG(Salary) AS AverageSalary
FROM Employees;
```

4. **MAX()**: Returns the maximum value in a column.

```
SELECT MAX(Salary) AS HighestSalary
FROM Employees;
```

Multi Atoms

5. **MIN()**: Returns the minimum value in a column.

```
SELECT MIN(Price) AS LowestPrice
FROM Products;
```

GROUP BY Clause

The GROUP BY clause groups rows with the same values in specified columns and performs aggregate functions on each group.

Example: Group sales data by region and calculate the total revenue per region.

```
SELECT Region, SUM(Revenue) AS
TotalRevenue
FROM Sales
GROUP BY Region;
```

input:

Region	Revenue
North	1000
South	1500
East	1200
North	2000
West	1800
South	1300
East	1000
North	1700

output:

Region	TotalRevenue
North	4700
South	2800
East	2200
West	1800

HAVING Clause

The HAVING clause filters groups based on aggregate function results (similar to WHERE but for grouped data).

Example: Find regions with total revenue greater than $10,000.

Multi Atoms

```
SELECT Region, SUM(Revenue) AS TotalRevenue
FROM Sales
GROUP BY Region
HAVING SUM(Revenue) > 10000;
```

input:

Region	Revenue
North	5000
South	4000
East	3500
West	3000
North	6000
South	7000
East	4500
West	2500

output:

Region	TotalRevenue
North	11000
South	11000

Insert, Update, and Delete Operations

These commands are fundamental for managing data in SQL tables. Here's a concise explanation of their usage, syntax, and examples.

INSERT Operation

Purpose: Adds new rows to a table.

Syntax:

```
INSERT INTO TableName (Column1, Column2, ...)
VALUES (Value1, Value2, ...);
```

Example:

```
INSERT INTO Book (ISBN, Title, Author, Publisher)
VALUES ('9781234567897', 'SQL Basics', 'John Doe',
'TechBooks');
```

UPDATE Operation

Purpose: Modifies existing rows in a table.

Syntax:

```
UPDATE TableName
SET Column1 = Value1, Column2 = Value2, ...
WHERE Condition;
```

Example:

```
UPDATE Book
SET Publisher = 'NewPublisher'
WHERE ISBN = '9781234567897';
```

DELETE Operation

Purpose: Removes rows from a table.

Syntax:

```
DELETE FROM TableName
WHERE Condition;
```

Example:

Multi Atoms

```
DELETE FROM Book
WHERE Publisher = 'OldPublisher';
```

SQL Joins

Aktu-2021-22, 2022-23, 2023-24

SQL JOINs are used to combine rows from two or more tables based on a related column. Below are different types of JOINs with simplified examples.

1. INNER JOIN

Combines rows from two tables where there is a match in the common column.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
INNER JOIN table2
ON table1.common_column =
table2.common_column;
```

Example:

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
INNER JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics

2. LEFT JOIN

Multi Atoms

Returns all rows from the left table and the matching rows from the right table. If there is no match, NULL is returned.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
LEFT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
LEFT JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
Bob	NULL

3. RIGHT JOIN

Returns all rows from the right table and the matching rows from the left table. If there is no match, NULL is returned.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
RIGHT JOIN table2
ON table1.common_column = table2.common_column;
```

Example:

Multi Atoms

Student Table:

Roll_No	Name	Branch
1	John	CSE
2	Alice	ECE
3	Bob	MECH

Course Table:

Roll_No	Course_Name
1	Math
2	Physics
4	Chemistry

```
SELECT Student.Name, Course.Course_Name
FROM Student
RIGHT JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
NULL	Chemistry

4. FULL JOIN

Combines the results of both LEFT JOIN and RIGHT JOIN. Returns all rows from both tables, with NULL where there is no match.

```
SELECT Student.Name, Course.Course_Name
FROM Student
FULL JOIN Course
ON Student.Roll_No = Course.Roll_No;
```

Output:

Name	Course_Name
John	Math
Alice	Physics
Bob	NULL
NULL	Chemistry

5. CROSS JOIN

Returns the Cartesian product of two tables, combining each row from the first table with every row from the second table.

Syntax:

```
SELECT table1.column1, table2.column2
FROM table1
CROSS JOIN table2
```

Multi Atoms

Example:

Student Table:

Roll_No	Name
1	John
2	Alice

Course Table:

Course_ID	Course_Name
101	Math
102	Physics

```
SELECT Student.Name, Course.Course_Name
FROM Student
CROSS JOIN Course;
```

Output:

Name	Course_Name
John	Math
John	Physics
Alice	Math
Alice	Physics

6. Natural Join

A Natural Join joins tables based on columns with the same names and data types in both tables.
It removes duplicate columns and keeps only one copy of the common column in the result.

Example:

Employee Table:

Emp_ID	Name	Dept_ID
1	Alice	101
2	Bob	102
3	Carol	103

Department Table:

Dept_ID	Dept_Name
101	IT
102	HR
104	Finance

```
SELECT *
FROM Employee
NATURAL JOIN Department;
```

Output:

Multi Atoms

Emp_ID	Name	Dept_ID	Dept_Name
1	Alice	101	IT
2	Bob	102	HR

Set Operations in SQL

Set operations allow combining the results of two or more queries. They work on the principle of sets in mathematics and require the queries to have the same number of columns with compatible data types.

1. UNION

- Combines the result sets of two queries.
- Removes duplicate rows by default.
- Use UNION ALL to include duplicates.

Syntax:

```
SELECT column1, column2
FROM table1
UNION
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
UNION
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
1	Alice
2	Bob
3	Carol

Multi Atoms

2. INTERSECT

- Retrieves common rows from the result sets of two queries.

Syntax:

```
SELECT column1, column2
FROM table1
INTERSECT
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
INTERSECT
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
2	Bob

3. MINUS

- Retrieves rows from the first query that are not present in the second query.

Syntax:

```
SELECT column1, column2
FROM table1
MINUS
SELECT column1, column2
FROM table2;
```

Example:

Emp-1 Table:

Emp_ID	Name
1	Alice
2	Bob

Emp-2 Table:

Emp_ID	Name
2	Bob
3	Carol

```
SELECT Emp_ID, Name
FROM Employee_Table1
MINUS
SELECT Emp_ID, Name
FROM Employee_Table2;
```

Output:

Emp_ID	Name
1	Alice

Multi Atoms

What is a Cursor?

A cursor is a temporary memory structure or workspace allocated by the database server to handle row-by-row data processing. It is used during Data Manipulation Language (DML) operations such as INSERT, UPDATE, DELETE, and SELECT. Cursors facilitate operations on a result set one row at a time, which is useful when row-specific logic is required.

Key Features:

- **Temporary Storage:** Acts as a buffer to hold query results.
- **Row-by-Row Processing:** Unlike set-based operations, cursors enable operations on individual rows.

Types of Cursors:

1. Implicit Cursors:

- Automatically created by the database when you execute simple queries like INSERT, UPDATE, or DELETE.
- You don't need to define or control it.

2. Explicit Cursors:

- Created by you (the user) when you need more control, like going through rows one by one and performing some actions.
- You manually declare, open, fetch data, and close it.

How Does a Cursor Work?

1. **Declare**: You define the cursor and the query that gives the list of rows to process.
2. **Open:** The cursor is prepared, and the query runs.
3. **Fetch:** The cursor moves to the first row and processes it. Then it moves to the next row, and so on.
4. **Close**: Once all rows are processed, the cursor is closed to free resources.

Multi Atoms

What is Triggers?

Aktu-2022-23, 2021-22

A trigger is a special kind of stored procedure that is executed automatically when a specific event (such as an insert, update, or delete) occurs on a table or view. Triggers are used to automate tasks, enforce business rules, maintain data integrity, audit changes, and replicate data.

Types of Triggers

1.Data Manipulation Language (DML) Triggers: DML triggers are executed when a DML operation like INSERT, UPDATE, or DELETE is performed on a table or view.

-> AFTER Triggers:

These triggers are executed after the DML statement completes but before the changes are committed to the database.

-> INSTEAD OF Triggers:

These triggers replace the triggering DML action (INSERT, UPDATE, DELETE) with a different action defined in the trigger body.

2.Data Definition Language (DDL) Triggers: DDL triggers are fired when a DDL operation like CREATE, ALTER, DROP, GRANT, REVOKE, or UPDATE STATISTICS is executed.

-> DATABASE-scoped DDL triggers

fire when DDL statements that affect the database schema (like creating tables or altering procedures) are executed.

-> SERVER-scoped DDL triggers

fire when server-level changes are made (e.g., creating or dropping databases, managing logins).

3. LOGON Triggers:

- LOGON triggers are automatically executed after a successful user login but before the user session is established.
- If authentication fails, the LOGON trigger is not executed.

CLR Triggers:

Multi Atoms

- CLR (Common Language Runtime) triggers are written using .NET languages such as C# or VB.NET.
- These triggers are useful for scenarios that require heavy computation or interaction with objects outside SQL, such as web services or external applications.

Example: Automatically Update Stock Levels After a Sale

Scenario:

We have two tables:

1. Products: Stores product details and stock levels.
2. Sales: Stores sales transactions.

When a sale is recorded in the Sales table, we want the stock quantity in the Products table to update automatically.

ProductID	ProductName	Stock
1	Laptop	50
2	Phone	100

SaleID	ProductID	Quantity
1	1	2

```
CREATE TRIGGER UpdateStockAfterSale
AFTER INSERT ON Sales
FOR EACH ROW
BEGIN
    UPDATE Products
    SET Stock = Stock - NEW.Quantity
    WHERE ProductID = NEW.ProductID;
END;
```

How It Works:

1. A new sale is added to the Sales table (e.g., 2 laptops sold).
2. The trigger automatically reduces the stock in the Products table.

Products Table Before:

ProductID	ProductName	Stock
1	Laptop	50
2	Phone	100

Insert a sale:

```
INSERT INTO Sales (SaleID, ProductID, Quantity)
VALUES (2, 1, 2);
```

Products Table After:

ProductID	ProductName	Stock
1	Laptop	48
2	Phone	100

Procedures in PL/SQL

PL/SQL (Procedural Language/Structured Query Language) is a block-structured programming language that allows combining SQL commands with procedural constructs like loops and conditions. A stored procedure in PL/SQL is a precompiled block of SQL statements stored in the database catalog for reuse.

A procedure serves as a reusable program that performs a specific task, which can be invoked by:

- Triggers
- Other procedures

When a procedure is executed, all its statements are sent to the Oracle engine at once, improving processing speed and reducing network traffic.

Advantages of Procedures in PL/SQL

- Improved Performance
- Reduced Network Traffic
- Code Reusability

Multi Atoms

Disadvantages of Procedures in PL/SQL

- Increased Memory Usage
- Complexity for Developers
- Lack of Debugging in MySQL

Example of a Stored Procedure in PL/SQL

Let's create a stored procedure to calculate the total salary of employees in a specific department.

EMP_ID	NAME	DEPT_ID	SALARY
1	John	101	5000
2	Alice	102	6000
3	Bob	101	5500
4	Charlie	103	7000

Stored Procedure

```
CREATE PROCEDURE CalculateTotalSalary (
    dept_no IN NUMBER,
    total_salary OUT NUMBER
)
AS
BEGIN
    -- Calculate the total salary of employees in the given department
    SELECT SUM(SALARY)
    INTO total_salary
    FROM Employee
    WHERE DEPT_ID = dept_no;
END;
/
```

Calling the Procedure

```
DECLARE
    dept_id NUMBER := 101; -- Input: Department ID
    total_sal NUMBER;      -- Output: Total Salary
BEGIN
    -- Call the procedure
    CalculateTotalSalary(dept_id, total_sal);
    -- Display the result
    DBMS_OUTPUT.PUT_LINE('Total Salary for Department ' || dept_id || ': ' || total_sal);
END;
/
```

Multi Atoms

Output:

```
Total Salary for Department 101: 10500
```

Subscribe Multi Atoms & Multi Atoms Plus
Join Telegram Channel
Check Description for Notes